NATURAL LANGUAGE PROCESSING

for hackers

GEORGE-BOGDAN IVANOV

LEARN TO BUILD APPS THAT CAN UNDERSTAND PEOPLE

NLPFORHACKERS.IO

Natural Language Processing For Hackers

Learn to build awesome apps that can understand people

George-Bogdan Ivanov

This book is for sale at http://leanpub.com/nlpforhackers

This version was published on 2018-07-09

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface

Today, written text is still the most popular method of communication. We read stuff on the Internet, we write each other emails, send texts and we read books. We're very comfortable with this type of communication and that's one of the main reasons for which chatbots are so popular right now. Being able to chat over the Internet is not enough though. We need to quickly interact with companies and access our various online accounts. This book's final chapter is indeed about building a chatbot, but I consider the book being much more. It is a great introduction to *Natural Language Processing* from a practical perspective. It touches all adjacent subjects such as: Machine Learning, Sentiment Analysis, data gathering, cleaning and various corpora.

Who this book is for

- You're a Python Engineer, comfortable with the language, and interested in this whole Machine Learning/Natural Language Processing thing.
- You're a researcher in Machine Learning/Natural Language Processing and are interested in seing a more practical facet of the field.
- You're already a NLP Engineer and would like to level up.

Revision History

2018-04-11: **First Edition.** Initial Release

Feedback and Updates

This book is updated regularly based on reader feedback. The plan is to create an updated version every three or four months, depending on the quantity of feedback and my availability. Anyone who purchased the book will have access to all further updates of the book.

Credits

- **Ioana Grigoropol** - Technical Reviewer
- **Raluca Mitarca** - Cover Designer
- **Madalina Munteanu** - Editor
- **Otilia Muntean** - Editor

Introduction

In 1956, John McCarthy coined **Artificial Intelligence** as the science of creating computer systems that simulate human intelligence process. For many years, the Artificial Intelligence field was present mostly in the academic environment. As technology evolved, companies started to pay more attention to this field but they were hiring only PhDs, without taking into consideration that the mindset of a scientist isn't completely aligned with the development process. This lead to **many AI systems using quirky libraries** that weren't best friends with the production environment. Therefore, the AI systems were living in a different universe than the rest of the application's code.

As of recently, the AI software landscape has changed and many developer-friendly tools, suitable for real-world applications, have emerged. Nowadays, **most of the applications we use on a daily basis have a tiny piece of AI integrated.** You can spot the AI features in applications that use face recognition, speech recognition, predictive analytics, recommendation of similar products or summarization of hard to digest information.

Since the Artificial Intelligence aims to imitate intelligent human behavior, it becomes obvious that this field isn't only broad and complex, but it also faces a number of diverse problems: reasoning, planning, learning, natural language processing, social awareness, perception, motion and manipulation, creativity. Broadly speaking, **we can say that an AI system is like a baby: it has all the resources it needs, but it has to learn how to use them in order to do intelligent things.** So, how will it learn to use them? Well, like a baby: experiences, examples, and communication.

Let's say you want to build an app to play Tic-Tac-Toe, but you don't know how to play this game. You can still build the app by giving it examples of previous games and letting it learn from those examples. This approach is called Machine Learning (ML) and it represents, in simpler terms, a method on how to make a machine learn throughout observations from which it **gains experience and improves its performance**.

Machine Learning is an absolutely huge and evolving field, under heavy research. At this moment, its most popular subfield is Deep Learning which studies how to build models using large neural network architectures with several layers (hence the attribute *"deep"*). Due to the depth of its applicability, Deep Learning is a major vector used to achieve undreamt results and to smash barriers we thought were unbreakable before.

This book doesn't require advanced Machine Learning knowledge, but some basic information on this subject might come in handy. Either way, we will have a brief introduction to Machine Learning, enough for you to be able to understand the whole book, but not to get bored.

Now let's say you also want to chat with the app while playing Tic-Tac-Toe. You will have to make the machine learn how to understand natural (human) language. Well, the solution for this would be Natural Language Processing (NLP), that also goes by the name of **Natural Language Understanding** or **Computational Linguistics**. NLP serves as a bridge between Human Language and Computer-Generated Language and can be integrated with ML in order to achieve amazing results.

What is Natural Language Processing?

The objective of NLP is to transform unstructured text data into knowledge (structured data) and use that in order to properly interact with humans. These days, the most popular example is the chatbot. A chatbot is a conversational agent that interacts directly with a human via a chat-like interface. The agent should feel in all aspects human-like because, well, humans already know how to interact with humans so the whole experience should be natural. The agent's algorithm transforms the user's requests into actions, using internal resources or external APIs to achieve desired goals, and communicates the results in a natural language back to the human counterpart.

I like to think about grammar as the science of turning plain language into mathematical objects. In fact, in the early days of NLP, almost all systems were Rule-Based systems (based on grammar rules) or modeled as state machines. These systems proved not to be as flexible as one would desire. If the sentence is missing a single word making it to not fit the human-crafted pattern, should we just ignore it? In other words, the boundaries created by this type of system were intransigent (black or white). **We needed more flexibility when dealing with complex phenomena such as the human language. Here's where Machine Learning came to help.**

NLP spans multiple disciplines. Here are some of them:

- **Machine Learning**: Nowadays, almost any NLP process is built upon a Machine Learning Systems.
- **Language & Grammar**: Understanding how a language works and general grammar rules gets you a long way.
- **General AI**: Knowledge about Rule-Based Systems, Knowledge-Based Systems.
- **Statistics**: Building language models.
- **Algebra**: Matrix operations.

Challenges in Natural Language Processing

It can be quite hard to point exactly why Natural Language Processing is so challenging. Maybe because natural language is extremely dynamic and we have been using it for a long time now. In fact, natural language is suffering constant modifications and we have evolved alongside it in deep symbiosis. Although some of them may sound stupid (as in simple), here are some of the most popular NLP challenges:

- **Splitting text into sentences** - Although this challenge is almost completely solved, if you come to think of it, it's not at all trivial. In the non-formal text, not all sentences are ended by a punctuation mark. Moreover, not all sentences even start with a capital letter. People use deep domain knowledge to actually split sentences. We might not even actually split sentences but rather extract pieces of information and just understand where one idea ends and another starts.
- **Ambiguity** - A lot of our communication is ambiguous or figurative and we often use sarcasm or irony. Let's look at this sentence: "You look sharp!". What do you think a computer should make of this sentence? Should the machine think that the person has a geometry that's particularly sharp? We,

humans, know that the meaning of the sentence is a figure of speech (it means that the person is well-dressed). Another popular example of a sentence with multiple meanings is: "I saw a man on a hill with a telescope". When reading it you probably instantly decided that it means: "There's a man on a hill, and I'm watching him with my telescope.". But there are another four possible meanings and I will leave you the pleasure of discovering them. (Answers: http://www.byrdseed.com/ambiguous-sentences/[1])
- **Emotions** - Emotions are hard to define and even harder to identify. We can think of them as complex states of feeling which are often intertwined with personality and mood. One of the popular challenges in NLP is called Sentiment Analysis and it implies detecting the polarity of a text (positive vs negative). It has applications in analyzing huge numbers of product reviews, quantify online presence and so on. Here's a head-scratcher:

  "The best I can say about this product is that it was definitely interesting."

In this case, the word "interesting" plays a different and more complex role than the classical one.

What makes this book different?

I have an MSc degree in AI and after working several years as a web developer, I became intrigued about building real-world AI systems. As I studied this field, I had to deal with irreproducible, incomplete and hard to grasp research papers. Sure, there are a lot of ML and NLP books out there, but here's what I hated about most books: **they only present one part of the problem.** This made me start writing a blog and after researching a bunch of sources, I've succeeded to build some complete, concise and compilable (as in runnable) code samples.

NLP for Hackers has emerged from the great response I had to my blog and it contains several recipes on how to solve common problems regarding Natural Language Processing.

This is not your typical research-oriented book that exposes the theoretical approach and uses clean datasets that you can only find in introductory courses and never in the real world. This is a hands-on, practical course on getting started with Natural Language Processing and learning its key concepts while coding. **No guesswork required.**

Here's where I think most books fall short:

[1] http://www.byrdseed.com/ambiguous-sentences/

- They only focus on how to train a machine learning model.
- **They don't train useful models**, like a part of speech tagger or a named entity extractor (We're going to do both here). The process of training this kind of models remains a mystery for most people.
- **They use perfect datasets** (from the academia) which are useless in the real world. In the real world, data is not perfect.
- They almost never explain the data gathering process.
- The systems built usually **remain stuck in a Python script or a Jupyter notebook**. They rarely get wrapped in an API that can be queried and plugged into a real system.

Throughout the book, you'll get to touch some of the most important and practical areas of Natural Language Processing. Everything you do will have a working result. Here are some things you will get to tackle:

- Build your own Text Analysis engine
- Understand how data-gathering works in the real world
- Build a Twitter listener that performs Sentiment Analysis on a certain subject
- Clean and standardise messy datasets
- Understand how to fine tune Natural Language models
- Learn how chatbots work
- Understand how the classic NLP tools are actually built, enabling you to create your own: Part of Speech Tagger, Shallow Parser, Named Entity Extractor and Dependency Parser

The book contains complete code snippets and step-by-step examples. No need to fill in the blanks or wonder what the author meant. **Everything is written in concise, easy-to-read Python3 code.**

Part 1: Introduction to NLTK

In this part we're going to dive head first into Natural Language Processing. We're going to get familiar with some basic NLP concepts. Here are a few things we're going to cover:

- Splitting text into sentences and splitting sentences into words.
- Learning how to use NLTK's default models for tagging and entity extraction.
- Building a vocabulary.
- Understanding why stemmers and lemmatizers are useful and what's the difference between them.

NLTK Fundamentals

NLTK (Natural Language ToolKit) is probably the most well known Python Natural Language Processing library. You can find the official website here: http://www.nltk.org/[2]. There's a lot of discussion around it whether it really is a production-ready library. I believe that the pre-trained models that come with it are not the best out there, but NLTK is definitely a great library for:

- **Learning NLP** - the API is simple & well-designed, full of good didactic examples
- **Prototyping** - before building performant systems, you should build a prototype so you can prove how cool your new app is.

Here's a detailed list with pros & cons for NLTK:

PROS	CONS
Perfect for getting started with Natural Language Processing	Doesn't come with super powerful pretrained models (there are other frameworks out there that provide far-better models)
Comes bundled with a lot of useful corpora, great for experimenting	No DeepLearning capabilities
Simple/Pythonic API, beautiful abstractions	Slow and unscalable
Some connectors to various libraries (like `StanfordPOSTagger`, `SennaTagger`)	

Here's how I usually go about starting a project:

1. Build a prototype using the stuff that's in NLTK
2. Augment it with custom models (maybe built with `scikit-learn`) for better accuracy and performance using the base classes in NLTK.

[2] http://www.nltk.org/

3. If more is needed, move to a high-performance library or to 3rd-party specialized services.

Installing NLTK

I remind you that this book requires basic Python programming experience and package management. That being said, the best way to get started is to create a separate Python3 virtual environment:

Create Python3 Environment

```
$ virtualenv -p python3 envname
```

and install NLTK using the following command:

Install nltk

```
$ pip install nltk
```

After installing, you need to download all the NLTK data. That includes the various corpora, grammars and pre-trained models. Open a Python console and run:

Download nltk data

```
import nltk
nltk.download('all')
```

The best way to read this book is to write the code as you progress. You can either create a simple Python3 file and write the code from every section, or if you have experience, you can create Jupyter Notebooks for a more interactive experience.

Splitting Text

Until now, you probably haven't put a lot of thought into how difficult can the task of splitting text into sentences and/or words be. Let's identify the most common issues using a sample text from NLTK and see how the NLTK sentence splitter performs in various situations.

Getting sample text from nltk.corpus

```
from nltk.corpus import reuters

print(reuters.raw('test/21131')[:1000], '...')
```

Let's pick a text from a news article that includes a few examples of the difficulties one may encounter:

> *AMPLE SUPPLIES LIMIT U.S. STRIKE'S OIL PRICE IMPACT Ample supplies of OPEC crude weighing on world markets helped limit and then reverse oil price gains that followed the U.S. Strike on an Iranian oil platform in the Gulf earlier on Monday, analysts said. December loading rose to 19.65 dlrs, up 45 cents before falling to around 19.05/15 later, unchanged from last Friday. "Fundamentals are awful," said Philip Lambert, analyst with stockbrokers Kleinwort Grieveson, adding that total OPEC production in the first week of October could be above 18.5 mln bpd, little changed from September levels. Peter Nicol, analyst at Chase Manhattan Bank, said OPEC production could be about 18.5-19.0 mln in October. Reuter and International Energy Agency (IEA) estimates put OPEC September production at 18.5 mln bpd. The U.S. Attack was in retaliation of last Friday's hit of a Kuwaiti oil products tanker flying the U.S. Flag, the Sea Isle City. …*

I chose this particular text from a news article because it emphasizes how humans split into sentences by using a lot of prior knowledge about the world. Taking it even deeper, this prior knowledge can't be seen as a massive library of information because it is filtered by the author's subjectivism. This being said, here are some challenges you may face when splitting text into sentences:

- **not all punctuation marks indicate the end of a sentence** (from the example above: *"U.S."*, *"19.65"*, *"19.05/15"*, etc.)
- not all sentences end with a punctuation mark (e.g. text from social platforms)
- not all sentences start with a capitalized letter (e.g. some sentences start with a quotation mark or with numbers)
- not all capitalized letters mark the start of a sentence (from the example above: *"The U.S. Attack"*, *"U.S. Flag"*)

Now that we've identified some difficulties, let's see how the NLTK sentence splitter works and how well it performs by analyzing the results we get.

Introducing nltk.sent_tokenize

```
import nltk
from nltk.corpus import reuters

sentences = nltk.sent_tokenize(reuters.raw('test/21131')[:1000])
print("#sentences={0}\n\n".format(len(sentences)))
for sent in sentences:
    print(sent, '\n')
```

Results from `nltk.sent_tokenize`:

nltk.sent_tokenize results

```
#sentences=8


AMPLE SUPPLIES LIMIT U.S. STRIKE'S OIL PRICE IMPACT
  Ample supplies of OPEC crude weighing on
  world markets helped limit and then reverse oil price gains
  that followed the U.S. Strike on an Iranian oil platform in the
  Gulf earlier on Monday, analysts said.

December loading rose to 19.65 dlrs, up 45 cents before
  falling to around 19.05/15 later, unchanged from last Friday.

"Fundamentals are awful," said Philip Lambert, analyst with
  stockbrokers Kleinwort Grieveson, adding that total OPEC
  production in the first week of October could be above 18.5 mln
  bpd, little changed from September levels.

Peter Nicol, analyst at Chase Manhattan Bank, said OPEC
  production could be about 18.5-19.0 mln in October.

Reuter and
  International Energy Agency (IEA) estimates put OPEC September
  production at 18.5 mln bpd.

The U.S.

Attack was in retaliation of last Friday's hit of
  a Kuwaiti oil products tanker flying the U.S.

Flag, the Sea
  Isle City.
```

As you can see, the results are not perfect: the last three sentences are actually a single sentence. The NLTK sentence splitter was fooled by the full stop and the capitalized letter.

The NLTK splitter is a rule-based system that keeps lists of abbreviations, words that usually go together and words that appear at the start of a sentence. Let me show you an example where the NLTK splitter doesn't fail:

Introducing nltk.sent_tokenize, take 2

```
import nltk

print(nltk.sent_tokenize("The U.S. Army is a good example."))

# ['The U.S. Army is a good example.'] - only one sentence, no false splits
```

Let's talk about splitting sentences into words. The process is almost the same, but the solution for word tokenization is easier and more straightforward.

Introducing nltk.word_tokenize

```
import nltk

print(nltk.word_tokenize('The U.S. Army is a good example.'))
# ['The', 'U.S.', 'Army', 'is', 'a', 'good', 'example', '.']
```

The `word_tokenize` function uses the `nltk.tokenize.treebank.TreebankWordTokenizer` class. The `TreebankWordTokenizer` is a rule-based system that splits sentences into words according to the rules in the Penn Treebank corpus that makes heavy use of regular expressions.

Here are the rules followed by the word tokenizer, extracted from the NLTK documentation:

- split standard contractions, e.g. *"don't"* → *"do n't"* and *"they'll"* → *"they 'll"*
- treat most punctuation characters as separate tokens
- split off commas and single quotes, when followed by whitespace
- separate periods that appear at the end of line

Building a vocabulary

You will often want to compute some word statistics. NLTK has some helpful classes to quickly compute the metrics you're after, like `nltk.FreqDist`. Here's an example of how you can use it:

Build a vocabulary

```
import nltk


fdist = nltk.FreqDist(nltk.corpus.reuters.words())

# top 10 most frequent words
print(fdist.most_common(n=10))
# [('.', 94687), (',', 72360), ('the', 58251), ('of', 35979), ('to', 34035), ('in', 26478), ('said', 25224), ('and', 2504\
3), ('a', 23492), ('mln', 18037)]

# get the count of the word `stock`
print(fdist['stock'])  # 2346

# get the count of the word `stork`
print(fdist['stork'])  # 0  :(

# get the frequency of the word `the`
print(fdist.freq('the'))  # 0.033849129031826936

# get the words that only appear once (these words are called hapaxes)
print(fdist.hapaxes())
# Hapaxes usually are `mispeled` or weirdly `cApiTALIZED` words.

# Total number of distinct words
print(len(fdist.keys()))  # 41600

# Total number of samples
print(fdist.N())  # 1720901
```

Fun with Bigrams and Trigrams

You'll hear about bigrams and trigrams a lot in NLP. There are nothing but pairs or triplets of adjacent words. If we would generalize the term to bigger lengths, we get ngrams.

Ngrams are used to build approximate language models, but they are also used in text classification tasks or as features for various other natural language statistical models. Bigrams and trigrams are especially popular because usually going further in size, you don't get any significant performance boost but rather a more complex model. Here are some shortcuts to work with:

Extracting bigrams & trigrams

```
from nltk import bigrams, trigrams, word_tokenize

text = "John works at Intel."
tokens = word_tokenize(text)

print(list(bigrams(tokens)))  # the `bigrams` function returns a generator, so we must unwind it
# [('John', 'works'), ('works', 'at'), ('at', 'Intel'), ('Intel', '.')]

print(list(trigrams(tokens)))  # the `trigrams` function returns a generator, so we must unwind it
# [('John', 'works', 'at'), ('works', 'at', 'Intel'), ('at', 'Intel', '.')]
```

A particular subset of a texts bigrams/trigrams are the collocations. To better understand what the collocations are, you can think of them like an expression of multiple words which commonly co-occur. Collocations can tell us a lot about the text they are extracted from and can be used as important features in different tasks.

Here's how to compute them using some handy NLTK functions:

Extracting collocations

```
import nltk
from nltk.collocations import BigramAssocMeasures, BigramCollocationFinder
from nltk.collocations import TrigramAssocMeasures, TrigramCollocationFinder

bigram_measures = BigramAssocMeasures()
trigram_measures = TrigramAssocMeasures()

# Compute length-2 collocations
finder = BigramCollocationFinder.from_words(nltk.corpus.reuters.words())

# only bigrams that appear 5+ times
finder.apply_freq_filter(5)

# return the 50 bigrams with the highest PMI (Pointwise Mutual Information)
print(finder.nbest(bigram_measures.pmi, 50))
# among the collocations we can find stuff like: (u'Corpus', u'Christi') ...

# Compute length-3 collocations
finder = nltk.collocations.TrigramCollocationFinder.from_words(nltk.corpus.reuters.words())

# only trigrams that appear 5+ times
finder.apply_freq_filter(5)

# return the 50 trigrams with the highest PMI
print(finder.nbest(trigram_measures.pmi, 50))
# among the collocations we can find stuff like: (u'Special', u'Drawing', u'Rights') ...
```

Pointwise Mutual Information

PMI is a measure that indicates the association between two variables: *how likely is that these two values appear together?* $pmi(x, y) = log(\frac{p(x, y)}{p(x)p(y)})$

If you'd take a look at the results, you'll recognize that most of the collocations are in fact real world entities like: people, companies, events, documents, laws, etc.

Part Of Speech Tagging

Part of Speech Tagging (or POS Tagging, for short) is probably the most popular challenge in the history of NLP. POS Tagging basically implies assigning a grammatical label to every word in a sequence (usually a sentence). When I say *grammatical label*, I mean: **Noun**, **Verb**, **Preposition**, **Pronoun**, etc.

In NLP, a collection of such labels is called a tag set. The most widespread one is: Penn Treebank Tag Set[3]. Below is the alphabetical list of part-of-speech tags used in the Penn Treebank Project:

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense

[3] (https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html)

Tag	Description
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

Your instinct now might be to run to your high school grammar book but don't worry, you don't really need to know what all those POS tags mean. In fact, not even all corpora implement this exact tag set, but rather a subset of it. For example, I've never encountered the `LS` (list item marker) or the `PDT` (predeterminer) anywhere.

Part-Of-Speech tagging also serves as a base of deeper NLP analyses. There are just a few cases when you'll work directly with the tagged sentence. A scenario that comes to mind is keyword extraction, when usually you only want to extract the adjectives and nouns. In this case, you use the tags to filter out those words that can't be a keyword.

If you're not familiar with the task, nowadays, POS tagging is done with machine learning models. But a while ago, POS taggers we're rule-based. Using regular expressions and various heuristics, the POS tagger would determine an appropriate tag. Here's an example of such subset of rules:

```
IF word in (I, you, he, her, it, we, them) THEN tag='PRONOUN'

IF previous_word.tag == 'PRONOUN' and word.last_letter == 's' THEN tag='VERB'
```

It may seem like a rather massive oversimplification, but this is the general idea. The challenge was to come up with rules that best described the phenomena that is the human language. But, as you go deeper and deeper, the rules become more and more complex, getting harder to keep track with which rules did what, if there were rules that were contradicting one another etc. Moreover, humans don't usually excel at noticing correlations between a great number of variables. In this case, the variables could have been:

- Previous words
- Following words
- Prefixes, Suffixes
- Previous POS tags
- Word capitalization

Mathematical algorithms are better at estimating which are the optimum rules for correctly tagging words. Thus, the field turned to machine learning and the approach became something like this:

1. Get some humans to annotate some texts with POS tags (we'll call this the gold standard)
2. Get other humans to build some mathematical models to predict tags using a large part of the gold standard corpus (this is called training the model)
3. Using the remaining part of the gold standard corpus, assess how well the model is performing on data the model hasn't seen yet (this is called testing the model)

This may seem complicated, and it usually is presented as such, but throughout this book, we'll be demistifying all of these algorithms. To start doing POS tagging we don't need much because NLTK comes with some pre-trained POS tagger models. It's super easy to get started and here's how:

Introducing nltk.pos_tag

```
import nltk


sentence = "Things I wish I knew before I started blogging."
tokens = nltk.word_tokenize(sentence)

print("Tokens: ", tokens)
# Tokens:  ['Things', 'I', 'wish', 'I', 'knew', 'before', 'I', 'started', 'blogging', '.']

tagged_tokens = nltk.pos_tag(tokens)

print("Tagged Tokens: ", tagged_tokens)
# Tagged Tokens:  [('Things', 'NNS'), ('I', 'PRP'), ('wish', 'VBP'), ('I', 'PRP'), ('knew', 'VBD'), ...
```

It is as straight-forward and easy as this, but keep in mind that the NLTK trained model is not the best. It's a bit slow and not the most precise. It is well suited though for doing toy projects or prototyping.

Named Entity Recognition

Named Entity Recognition (NER for short) is almost as well-known and studied as POS tagging. NER implies extracting named entities and their classes from a given text. The usual named entities we're dealing with stand for: *People*, *Organizations*, *Locations*, *Events*, etc. Sometimes, things like currencies, numbers, percents, dates and time expressions can be considered named entities even though they technically aren't. The entities are used in information extraction tasks and usually, these entities can be attributed to a real-life object or concept. To make things clearer, let me give you some examples:

- if we extract a name of a person from a text, we can associate it with a Facebook profile, email address or even a unique identification number
- if we extract a date/time, we can associate the string with an actual slot in a calendar.
- if we extract a location, we can associate it with some exact coordinates in Google Maps.

Using the nltk.ne_chunk function

```
import nltk

# Adapted from Wikipedia:
# https://en.wikipedia.org/wiki/Surely_You%27re_Joking,_Mr._Feynman!

sentence = """The closing chapter, is adapted from the address that
Feynman gave during the 1974 commencement exercises
at the California Institute Of Technology. """

# tokenize and pos tag
tokens = nltk.word_tokenize(sentence)
tagged_tokens = nltk.pos_tag(tokens)
ner_annotated_tree = nltk.ne_chunk(tagged_tokens)

print(ner_annotated_tree)

# (S
#   The/DT
#   closing/NN
#   chapter/NN
#   ,/,
#   is/VBZ
#   adapted/VBN
#   from/IN
#   the/DT
#   address/NN
#   that/IN
#   (PERSON Feynman/NNP)
#   gave/VBD
#   during/IN
#   the/DT
#   1974/CD
#   commencement/NN
#   exercises/NNS
#   at/IN
#   the/DT
#   (ORGANIZATION California/NNP Institute/NNP Of/IN Technology/NNP)
#   ./.)
```

Notice that the result has a single or multiple tokens bundled up in *entities*:

1. *Feynman* = `PERSON`

2. *California Institute Of Technology* = `ORGANIZATION`

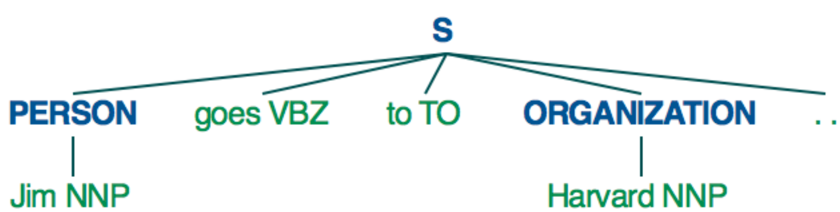
Also notice that we needed to POS tag the sentence first before feeding it to the `ne_chunk` function. This is because the function uses the POS tags as features that contribute decisively to predicting whether something is an entity or not. The most accessible examples are the tags `NNP` and `NNPS`. What do you think is the reason these tags help the NE extractor find entities?

Let's pay a closer look at the `ne_chunk` function. Chunking means taking the tokens of a sentence and grouping them together, in chunks. In this case, we grouped the tokens belonging to the same named entity into a single chunk. The data structure that facilitates this is `nltk.Tree`. The tokens of a chunk are all children of the same node. All the other non-entity nodes and the chunk nodes are children of the same root node: `S`. Here's a quick way to visualize these trees:

Drawing a nltk.Tree

```
import nltk

sentence = """Jim goes to Harvard."""
tokens = nltk.word_tokenize(sentence)
tagged_tokens = nltk.pos_tag(tokens)
ner_annotated_tree = nltk.ne_chunk(tagged_tokens)
ner_annotated_tree.draw()
```

This is how the tree will look:



Named Entity Tree

The same things that we've said about the NLTK tagger are true for the NLTK Named Entity Chunker: it's not perfect.

In the next section we will explore another valuable resource that's bundled up in NLTK: Wordnet.

Getting started with Wordnet

Wordnet[4] is a lexical database created at Princeton University. Due to its size and features, Wordnet is one of the most useful tools you can have in your NLP arsenal.

Wordnet Structure

You can think of Wordnet as a graph of concepts. The edges between the concepts describe the relationship between them. A concept node is, in fact, a set of synonyms representing the same concept, and is called a **Synset**. The synonyms within the synset are represented by their **Lemma**. A Lemma is the base form (the dictionary form) of the word.

Using this type of structure, we are able to build mechanisms that can *reason*. Let's take an example: I can say either *"Bob drives the car"* or *"Bob dives the automobile"*, both being true because the nouns *"car"* and *"automobile"* are synonyms. Also, *"Bob drives the vehicle"* can be true since all cars are vehicles.

NLTK provides one of the best interfaces for Wordnet. It is extremely easy to use and before we dive into it, make this exercise: try to write down all the meanings of the word *"car"*.

Getting started with Wordnet

```
from nltk.corpus import wordnet as wn


# Let's investigate what are the various synsets for `car`
# Remember that each synset represents a separate sense of the word `car`
for car in wn.synsets('car'):
    print([l.name() for l in car.lemmas()])
    print(car.definition())
    print()


# ['car', 'auto', 'automobile', 'machine', 'motorcar']
# a motor vehicle with four wheels; usually propelled by an internal combustion engine
#
# ['car', 'railcar', 'railway_car', 'railroad_car']
# a wheeled vehicle adapted to the rails of railroad
#
# ['car', 'gondola']
# the compartment that is suspended from an airship and that carries personnel and the cargo ...
```

[4] https://wordnet.princeton.edu/

```
#
# ['car', 'elevator_car']
# where passengers ride up and down
#
# ['cable_car', 'car']
# a conveyance for passengers or freight on a cable railway
```

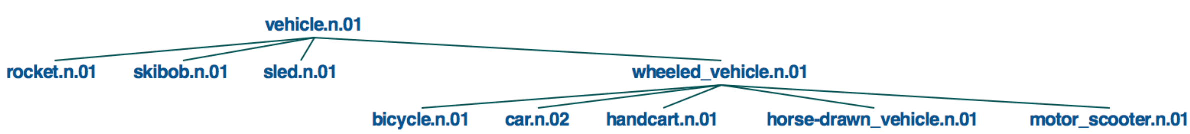
The most important relationships in Wordnet other than the synonymy relationship are the **hyponymy/hypernymy**. Hyponyms are more specific concepts while hypernyms are more general concepts.

Let's build a concept tree. Due to aesthetics, we won't use all the concepts because it would be hard to visualize. I have conveniently chosen only 4 hyponyms of the concept *Vehicle* and 5 more for its *Wheeled-Vehicle* hyponym:

Wordnet concept tree

```
import nltk
from nltk.corpus import wordnet as wn


# Let's get the first sense of vehicle
vehicle = wn.synsets('vehicle')[0]

# Let's build a concept tree
t = nltk.Tree(vehicle.name(), children=[
    nltk.Tree(vehicle.hyponyms()[3].name(), children=[]),
    nltk.Tree(vehicle.hyponyms()[4].name(), children=[]),
    nltk.Tree(vehicle.hyponyms()[5].name(), children=[]),
    nltk.Tree(vehicle.hyponyms()[7].name(), children=[
        nltk.Tree(vehicle.hyponyms()[7].hyponyms()[1].name(), children=[]),
        nltk.Tree(vehicle.hyponyms()[7].hyponyms()[3].name(), children=[]),
        nltk.Tree(vehicle.hyponyms()[7].hyponyms()[4].name(), children=[]),
        nltk.Tree(vehicle.hyponyms()[7].hyponyms()[5].name(), children=[]),
        nltk.Tree(vehicle.hyponyms()[7].hyponyms()[6].name(), children=[]),
    ]),
])

t.draw()
```

The result would be the following concept tree:



Concept Tree

Maybe it's not obvious from the previous queries, but synsets have an associated part of speech. You can see it in the visualization on node labels: `vehicle.n.01` where `n` stands for *noun*. We can perform generic queries without specifying the part of speech like this:

Querying Wordnet for synsets

```
from nltk.corpus import wordnet as wn

print(wn.synsets('fight'))

# [
#     Synset('battle.n.01'),
#     Synset('fight.n.02'),
#     Synset('competitiveness.n.01'),
#     Synset('fight.n.04'),
#     Synset('fight.n.05'),
#     Synset('contend.v.06'),
#     Synset('fight.v.02'),
#     Synset('fight.v.03'),
#     Synset('crusade.v.01')
# ]
```

We can also perform particular queries, specifying the part of speech like this:

Querying Wordnet for synsets and filtering by part of speech

```
from nltk.corpus import wordnet as wn

print(wn.synsets('fight', wn.NOUN))

# [
#     Synset('battle.n.01'),
#     Synset('fight.n.02'),
#     Synset('competitiveness.n.01'),
#     Synset('fight.n.04'),
#     Synset('fight.n.05')
# ]
```

Moreover, we can perform a query for a very specific synset, like this one:

Fetching specific synset

```
from nltk.corpus import wordnet as wn

# Synset id format = {lemma}.{part_of_speech}.{sense_number}
walk = wn.synset('walk.v.01')
```

Lemma Operations

Until now, we've been looking into Wordnet synsets relationships, but lemmas have some interesting properties as well. To begin with, the lemmas in a synset are sorted by their frequency, like this:

Lemma operations

```
from nltk.corpus import wordnet as wn


talk = wn.synset('talk.v.01')
print({lemma.name(): lemma.count() for lemma in talk.lemmas()})
# {'talk': 108, 'speak': 53}

# Get antonyms for the adjective `beautiful`
beautiful = wn.synset('beautiful.a.01')
print(beautiful.lemmas()[0].antonyms())
# Lemma('ugly.a.01.ugly')

# Get the derivationally related forms of a lemma
able = wn.synset('able.a.01')
print(able.lemmas()[0].derivationally_related_forms())
# [Lemma('ability.n.02.ability'), Lemma('ability.n.01.ability')]
```

As we discovered so far, Wordnet is great because it gives us a way of getting synonyms, antonyms, different senses of a word, related words, how common a word is and so on. Let's go even further and discover some more useful features of Wordnet.

Lemmatizing and Stemming

One of the most useful operations in Natural Language Processing is finding the base form of a word. This is such a common operation because it's so useful in so many situations. Let's take the case of a search engine which should provide similar results for different queries (e.g. *"write book"* / *"writing books"*). In order for the search engine to provide similar results, it has to find the base form of the words. There are two distinct approaches to this task:

a. **Stemming**, which is an algorithmic method of a shaving off prefixes and suffixes from a given word form
b. **Lemmatizing**, which is a corpus-based method for finding the base form of a word

There are fundamental differences between these two methods, each having advantages and disadvantages. Here's a short comparison:

- Both stemmers and lemmatizers bring inflected words to the same form, or at least they try to
- Stemmers can't guarantee a valid word as a result (in fact, the result usually isn't a valid word)
- Lemmatizers always return a valid word (the dictionary form)
- Stemmers are faster
- Lemmatizers depend on a corpus (basically a dictionary) to perform the task

How stemmers work

When stemming, what you actually do is to apply various rules to a specific word form until you reduce the word to its basic form and no other rule can be applied. This method is fast and also, the best choice for most real-world applications. The downside of stemming is the incertitude of the result because it usually isn't a valid word, but on the bright side, you can have two related words which can resolute to the same stem. Let's see an example:

Snowball Stemmer

```
from nltk.stem import SnowballStemmer


stemmer = SnowballStemmer('english')

# all resolve to `friend`
print(stemmer.stem('friend'))
print(stemmer.stem('friends'))
print(stemmer.stem('friendly'))

# Not working well with irregulars
print(stemmer.stem('drink'))  # `drink`
print(stemmer.stem('drunk'))  # `drunk`

# Not a proper word (`slowli`)
print(stemmer.stem('slowly'))

# Works with non-existing words (`xyze`)
print(stemmer.stem('xyzing'))
```

How lemmatizers work

You can think of lemmatizers as a huge Python dictionary where the word forms are the keys and the base form of the words are the values. Because two words with different parts of speech can have the same form, we need to specify the part of speech.

Wordnet Lemmatizer

```
from nltk.stem import WordNetLemmatizer


lemmatizer = WordNetLemmatizer()

# Different parts of speech with the same form
print(lemmatizer.lemmatize('haunting', 'n'))  # `haunting`
print(lemmatizer.lemmatize('haunting', 'v'))  # `haunt`


# Resolve to `friend`
print(lemmatizer.lemmatize('friend', 'n'))
print(lemmatizer.lemmatize('friends', 'n'))

# Resolving to `friendly` because it's a different part of speech
print(lemmatizer.lemmatize('friendly', 'a'))

# Working well with irregulars
print(lemmatizer.lemmatize('drink', 'v'))  # `drink`
print(lemmatizer.lemmatize('drunk', 'v'))  # `drink`

# Always a proper word (`slowly`)
```

```
print(lemmatizer.lemmatize('slowly', 'r'))

# Not working with non-existing words (`xyzing`)
print(lemmatizer.lemmatize('xyzing', 'v'))
```

Part 2: Create a Text Analysis service

Creating a Text Analysis service may seem a complex task at this point, but at the end of this chapter, I am confident you will understand the process and appreciate its simplicity. Text Analysis is a broad expression that encompasses several Natural Language Processing tasks. One of the most popular tasks is probably Text Classification, which represents the process of assigning a label to a text. The labels can be for various facets of the text, but the most common classification criteria are:

1. **Domain Classification:** it entails assigning a domain label (politics, social, education etc.) and it's usually applied to news articles or similar media.
2. **Sentiment Analysis:** it entails assigning an emotional label (negative, neutral or positive) and it's used for reviews, comments or social media messages. We will discuss this classification criterion in one of the following chapters of this book.

Introduction to Machine Learning

In this chapter, we won't be doing the classic introduction to Machine Learning. You can find a lot of good classical introductions that usually start with a drawing which shows how a black box is a function that maps inputs to outputs, but I am going for a different, more intuitive approach of understanding ML.

Machine Learning is a form of AI which, in general terms, implies performing some reasoning given some previous experience.

At the other end of the spectrum we have classic AI reasoning paradigms, and here some of the most popular:

- **Rule Based Systems:** given a set of rules, figure out which ones apply. The process would look like this: apply the rules, get the results and by interpreting them, take one decision or another.
- **Case Based Reasoning:** given a list of previous cases and the correct label, get the most similar ones and figure out a way to use the labels of these cases to come up with a label for a new given case.
- **Knowledge Based Reasoning:** use a graph-like taxonomy of the world to perform reasoning (e.g. if all mammals give birth to living babies & dolphins are fish & dolphins are mammals, then there are some fish that give birth to alive babies)

These are great approaches, but here's the main flow for all of them: they are as smart as the one programming them. They also have a limited view of the world and and a small amount of expressivity. Moreover, they lack flexibility and need human input all along the way (e.g. someone has to notice if a rule doesn't perform well, someone has to figure out what's wrong with it and fix it and, furthermore, someone can make it either more general or more specific, or split it into two or more rules).

Machine learning is basically a way of finding out what are the best rules for some given data and also a way of updating the system when we get new information.

Let's take an example to better understand this concept. Suppose we want to predict the weather and, from a classic approach, we can build the following rules:

- if in the previous day it rained and today the sky is cloudy then today it will rain

- if today the humidity is above a threshold T and the sky is cloudy then today it will rain
- otherwise it won't rain today

Though I have used some common sense when creating these rules, they have no real-world foundation. I mainly made them up. If I were to implement this system, it might have certain predictive powers, thus presenting some intelligent behaviour.

Now let's say that for one particular day the conditions for the first rule were accurate (previous day was rainy & today the sky is cloudy), but also in that particular day the temperature was very low so it didn't rain, thus, the prediction was wrong. In a classical AI approach, we would manually correct the rule: one needs to get into the system, identify which rule predicted the rain and figure out what was the parameter that wasn't taken into consideration (in this case, the temperature). In addition, one has to figure out what would be a good threshold for temperature (like humidity, in the second rule). As you can imagine, this can become pretty unmanageable.

Taking the same example (predicting the weather), but from the machine learning approach, the process would be totally different. First, we would sit back and relax for a year and just collect weather data. Secondly, we would build a mathematical model that minimizes prediction error because we now have data. Then we would put the system online and keep monitoring the weather, feeding the system with new data to better itself.

So, what's the problem now? Well, machine learning systems have this one big issue: they need data. It may not seem like an issue at first, but depending on what system you want to build, collecting data may be hard. It may take a long time to gather it and it can be really expensive. Think for example about the data needed to model the effects of total solar eclipses on deep ocean fish migration patterns.

We, humans, perform very similar data collection and labelling processes all the time without realizing it. As we age, the processes become extremely convoluted. We take into account so many parameters that we can't even keep track of. It's interesting to observe this on small kids because they haven't experienced the world for so long, so they haven't yet perceived all of its facets. I know one cute baby that used to be friendly with everyone. For that baby, everyone had a GOOD label attached. One time, the baby got a bit sick so he had to be taken to the hospital. A nurse that was wearing a white medical gown gave him a shot. This was his first BAD label. Some days later, a neighbour wearing a white t-shirt visited the baby. Instantly, the baby began crying. From the baby's experience, people wearing white are labelled as BAD.

Getting back to our weather example, such an application for predicting weather will need to work in a similar way by learning from experience, in its case:

data. When we are talking about real-world systems, this gets even more difficult: the system might not experience all possible temperature or humidity values, hardware may malfunction and collect incorrect or incomplete data, there can be errors in measurements, etc. Generally speaking, real-world systems usually can't experience the full spectrum of valid inputs and here's why:

- a considerable amount of system inputs are real values[5] and, by definition, there is an infinity of possible real values.
- majority of systems have multiple parameters, and even if we split the input values into small intervals, the combinatorial explosion will make the data impossible to gather and to work with
- the systems aren't able to continuously extract samples, usually performing measurements at certain time intervals
- the systems may encounter a malfunction and stop collecting data for certain periods of time
- the systems may have a finite measurement precision
- sensors used by the system may introduce errors

Moving one step deeper: what happens when we need to predict the weather given some meteorological conditions that do not match any of previously seen data? Hard to say. Get the closest labelled data point and use the same label? Maybe. Get the most common label of the 10 most similar cases? Possibly. Do something different altogether? That's definitely a possibility.

The process we're trying to simulate here is called generalization, and the gist of it is: we don't need to know everything in the world to have a pretty good idea about something. We can just fill in the blanks. We can make informed guesses about data we have never seen before, that is somewhat alike to other data we have seen. **This is Machine Learning.**

A Practical Machine Learning Example

A while ago, I have written on my blog a post about a practical Machine Learning example. I will use the same example in this book, because it fits very well. We will get started with Machine Learning by building a predictor that is able to tell if a name is a girl name or a boy name. I think that this example is one of the best starting points, and here's why:

- it's works on text and is thus, representative for this book's subject

[5] https://en.wikipedia.org/wiki/Real_number

- it forces you to do actual feature extraction, rather than getting them from a table
- it models a problem that is familiar rather than dealing with numbers or abstract concepts

But before diving into the ML example, I want to continue the analogy with human reasoning from the previous example (predicting the weather). Building the rules for the weather prediction system, we took into consideration things like:

- Is the temperature above threshold `T`? (Y/N)
- Is the humidity below threshold `H`? (Y/N)
- Did it rain yesterday? (Y/N)

It is obvious that we're not taking into consideration all the information we have. We don't care that 100 days ago it rained, 101 days ago it didn't, and 102 days ago it poured cats and dogs. Some information might be relevant, but we don't use it. We try to simplify the world we live in, aiming only to extract some of its features.

Now let's dive into the machine learning problem, starting by examining the data a bit:

Names Corpus

```
import collections
from nltk.corpus import names


# Check the girl names
girl_names = names.words('female.txt')
print(girl_names[:10], '...')
print('#GirlNames=', len(girl_names))  # 5001

# Check the boy names
boy_names = names.words('male.txt')
print(boy_names[:10], '...')
print('#BoyNames=', len(boy_names))  # 2943

# We know that there are a lot of girl names that end in `a`
# Let's see how many
girl_names_ending_in_a = [name for name in girl_names if name.endswith('a')]
print('#GirlNamesEndingInA=', len(girl_names_ending_in_a))  # 1773

# Approx. a third of girl names end in `a`. That's a good insight
# Let's see what are the most common letters girl names end with
girl_ending_letters = collections.Counter([name[-1] for name in girl_names])
print("MostCommonEndingLettersForGirls=", girl_ending_letters)

# Our intuition was right. The most common letter is `a`
# Here are the first 3: 'a': 1773, 'e': 1432, 'y': 461

# I'm not sure what's the most common last letter for English boy names
```

```
# Let's see the stats
boy_ending_letters = collections.Counter([name[-1] for name in boy_names])
print("MostCommonEndingLettersForBoys=", boy_ending_letters)
# Here are the first 3: 'n': 478, 'e': 468, 'y': 332,
```

Here's a problem: the 2nd and 3rd most common last letters are the same for both genders. Let's see how to deal with that.

First, we will build a predictive system and analyze its results. This type of system is called a classifier. As you can guess, the role of a classifier is to classify data. The idea is pretty basic: it takes some input and it assigns that input to a class. At first, we are going to use the NLTK classifier because it is a little bit easier to use. After we clarify the concepts, we will move to a more complex classifier from Scikit-Learn.

Build the first name classifier

```
import nltk
import random
from nltk.corpus import names


def extract_features(name):
    """
    Get the features used for name classification
    """
    return {
        'last_letter': name[-1]
    }


# Get the names
boy_names = names.words('male.txt')
girl_names = names.words('female.txt')

# Build the dataset
boy_names_dataset = [(extract_features(name), 'boy') for name in boy_names]
girl_names_dataset = [(extract_features(name), 'girl') for name in girl_names]

# Put all the names together
data = boy_names_dataset + girl_names_dataset

# Mix everything together
random.shuffle(data)

# Let's take a look at the first few entries
print(data[:10])

# Split the dataset into training data and test data
cutoff = int(0.75 * len(data))
train_data, test_data = data[:cutoff], data[cutoff + 1:]

# Let's train probably the most popular classifier in the world
name_classifier = nltk.DecisionTreeClassifier.train(train_data)

# Take if for a spin
print(name_classifier.classify(extract_features('Bono')))  # boy
print(name_classifier.classify(extract_features('Latiffa')))  # girl
```

```
print(name_classifier.classify(extract_features('Gaga')))  # girl
print(name_classifier.classify(extract_features('Joey')))  # girl

# Sorry Joey :(
# Test how well it performs on the test data:
# Accuracy = correctly_labelled_samples / all_samples
print(nltk.classify.accuracy(name_classifier, test_data))  # 0.7420654911838791

# Look at the tree
print(name_classifier.pretty_format())
```

At this point, in our example, we only have one input (the last letter) and two classes (boy/girl). Turns out that our experiment with the four names (Bono, Latiffa, Gaga and Joey) was pretty good since one in four names got misclassified.

Using the `nltk.DecisionTreeClassifier` class we're effectively building a tree, where nodes hold conditions that try to optimally classify the training data. Here's how ours looks like:

tree name classifier

```
last_letter= ? ....................................... girl
last_letter=a? ....................................... girl
last_letter=b? ....................................... boy
last_letter=c? ....................................... boy
last_letter=d? ....................................... boy
last_letter=e? ....................................... girl
last_letter=f? ....................................... boy
last_letter=g? ....................................... boy
last_letter=h? ....................................... girl
last_letter=i? ....................................... girl
last_letter=j? ....................................... boy
last_letter=k? ....................................... boy
last_letter=l? ....................................... boy
last_letter=m? ....................................... boy
last_letter=n? ....................................... boy
last_letter=o? ....................................... boy
last_letter=p? ....................................... boy
last_letter=r? ....................................... boy
last_letter=s? ....................................... boy
last_letter=t? ....................................... boy
last_letter=u? ....................................... boy
last_letter=v? ....................................... boy
last_letter=w? ....................................... boy
last_letter=x? ....................................... boy
last_letter=y? ....................................... girl
last_letter=z? ....................................... boy
```

It may not seem like much of a tree at this point (it only has one level), but don't worry, by the time we'll be done with it, it will.

Let's add another feature: the number of vowels in the name. In order to do this, we need to change the `extract_features` function and retrain.

New feature function

```python
def extract_features(name):
    """
    Get the features used for name classification
    """
    return {
        'last_letter': name[-1],
        'vowel_count': len([c for c in name if c in 'AEIOUaeiou'])
    }
```

After retraining we got a `0.77` accuracy (77%). That means we have a 3% boost. Let's see the new tree which now has two levels:

two level tree name classifier

```
last_letter= ? ....................................... girl
last_letter=a? ....................................... girl
last_letter=b? ....................................... boy
  vowel_count=0? .................................... girl
  vowel_count=1? .................................... boy
  vowel_count=2? .................................... boy
last_letter=c? ....................................... boy
last_letter=d? ....................................... boy
last_letter=e? ....................................... girl
  vowel_count=1? .................................... boy
  vowel_count=2? .................................... girl
  vowel_count=3? .................................... girl
  vowel_count=4? .................................... girl
  vowel_count=5? .................................... girl
  vowel_count=6? .................................... girl
etc ...
```

At this point, we can notice that more features are somehow better, allowing us to capture different aspects of a given sample. Let's add a few more and see how far it takes us:

Final feature extraction function

```python
import string


def extract_features(name):
    """
    Get the features used for name classification
    """
    features = {
        # Last letter
        'last_letter': name[-1],
        # First letter
        'first_letter': name[0],
        # How many vowels
        'vowel_count': len([c for c in name if c in 'AEIOUaeiou'])
    }
    # Build letter and letter count features
    for c in string.ascii_lowercase:
```

```
        features['contains_' + c] = c in name
        features['count_' + c] = name.lower().count(c)
    return features
```

I got a `0.79` (79%) accuracy. Please be aware that the numbers may be slightly different in your case and that's due to the randomization of the training/test set.

Ok, now that we got comfortable with these concepts, let's repeat this process using Scikit-Learn models and see if we can do better.

Getting Started with Scikit-Learn

While NLTK models are definitely great for getting started with ML, Scikit-Learn implementations are way better from many points of view. Before we repeat the process from the previous section, let me emphasize some Scikit-Learn characteristics:

- provides a greater diversity of models available, whereas NLTK offers only implementations for: *MaxentClassifier*, *DecisionTreeClassifier* and *NaiveBayesClassifier*
- provides consistent API for data preprocessing, vectorization and training models.
- has extensively documented API and various tutorials around
- is faster than most alternatives due to the fact that it is built upon `numpy`
- provides some parallelization features

> **About NumPy**
>
> NumPy is a scientific computing package with datastructures optimized for fast numeric operations

Installing Scikit-Learn and building a dataset

Let's install `scikit-learn`:

```
$ pip install scikit-learn
```

Now let's build the dataset using a useful `sklearn` utility: `train_test_split`.

Build dataset with scikit-learn

```
from nltk.corpus import names
from sklearn.model_selection import train_test_split


def extract_features(name):
    """
    Get the features used for name classification
    """
    return {
        'last_letter': name[-1]
    }

# Get the names
boy_names = names.words('male.txt')
girl_names = names.words('female.txt')

# Build the dataset
boy_names_dataset = [(extract_features(name), 'boy') for name in boy_names]
girl_names_dataset = [(extract_features(name), 'girl') for name in girl_names]

# Put all the names together
data = boy_names_dataset + girl_names_dataset

# Split the data in features and classes
X, y = list(zip(*data))

# split and randomize
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25, shuffle=True)
print(X_train)
print(y_train)
```

Training a Scikit-Learn Model

We are going to rewrite the training process using the equivalent `scikit-learn` models. Here are the main building blocks of the scikit-learn API:

There are two main types of `scikit-learn` components:

1. *Transformers*
2. *Predictors*

Here are the most used methods:

- `fit` - used for training a `scikit-learn` component.
- `transform` - Transformer method for converting the data format
- `predict` - Predictor method for classifying a data point (in our case)
- `score` - Evaluate a Predictor's accuracy

Train a Decision Tree Classifier

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction import DictVectorizer

dict_vectorizer = DictVectorizer()
name_classifier = DecisionTreeClassifier()

# Scikit-Learn models work with arrays not dicts
# We need to train the vectorizer so that
# it knows what's the format of the dicts
dict_vectorizer.fit(X_train)

# Vectorize the training data
X_train_vectorized = dict_vectorizer.transform(X_train)

# Train the classifier on vectorized data
name_classifier.fit(X_train_vectorized, y_train)

# Test the model
X_test_vectorized = dict_vectorizer.transform(X_test)
# Compute Accuracy (0.75)
```

I encourage you to retrain the classifier using the extended feature extraction function.

At this point, some concepts might seem ambiguous. For instance, what gets passed between the `DictVectorizer` and `DecisionTreeClassifier`? Well, Scikit-Learn models only work with numeric matrices. The role of vectorizers is to convert data from various formats to numeric matrices. If we take a look at output type of a vectorizer we will get this: `scipy.sparse.csr.csr_matrix`. This is a sparse matrix, which is a memory-efficient way of storing a matrix. Unfortunately, it's a bit hard to read such a matrix. However, we need to be able to work with it and to understand this data format and here's how we can do that:

Understanding vectorized data

```
NAMES = ['Lara', 'Carla', 'Ioana', 'George', 'Steve', 'Stephan']
transformed = dict_vectorizer.transform([extract_features(name) for name in NAMES])

# We get a scipy sparse matrix, which is a bit hard to read
print(transformed)
print(type(transformed)) # <class 'scipy.sparse.csr.csr_matrix'>

# We can transform the data back
print(dict_vectorizer.inverse_transform(transformed))

# We can check the feature names
print(dict_vectorizer.feature_names_)
# ['contains_a', 'contains_b', 'contains_c', 'contains_d', 'contains_e', 'contains_f', ...

# Just as an excercise, let's build a feature dictionary in reverse:
import numpy as np

# build a vector with 0 value for each feature
```

```
vectorized = np.zeros(len(dict_vectorizer.feature_names_))

# Let's set the features by hand for an imagined name: "Wwbwi"
# The index in `feature_names_` represents the index of the feature
# in a row of the sparse matrix we previously discussed

# Our name has the last letter `i`
vectorized[dict_vectorizer.feature_names_.index('last_letter=i')] = 1.0

# Contains 3 `w`s
vectorized[dict_vectorizer.feature_names_.index('count_w')] = 3.0
vectorized[dict_vectorizer.feature_names_.index('count_b')] = 1.0
vectorized[dict_vectorizer.feature_names_.index('count_i')] = 1.0

# It contains the letter `b`, `i`, `w`
vectorized[dict_vectorizer.feature_names_.index('contains_b')] = 1.0
vectorized[dict_vectorizer.feature_names_.index('contains_w')] = 1.0
vectorized[dict_vectorizer.feature_names_.index('contains_i')] = 1.0

print(vectorized)
# Let's see what we've built:
# [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
#   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
#   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  3.  0.  0.  0.  0.  0.
#   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
#   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.
#   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

# Let's now apply the inverse transformation, to get the feature dict
print(dict_vectorizer.inverse_transform([vectorized]))
# [{'contains_a': 1.0, 'contains_b': 1.0, 'contains_w': 1.0,
#   'count_a': 1.0, 'count_b': 1.0, 'count_w': 3.0, 'last_letter=a': 1.0}]
```

Making Predictions

As we covered in the Scikit-Learn API summary, predictions can be made using the predict method. This is an important step in our process, but keep in mind that you can call this method only if you previously called the `fit` method on a trained model.

Making predictions

```
# Let's make some predictions
NAMES = ['Lara', 'Carla', 'Ioana', 'George', 'Steve', 'Stephan']
transformed = dict_vectorizer.transform([extract_features(name) for name in NAMES])
print(name_classifier.predict(transformed))
# We get: ['girl' 'girl' 'girl' 'girl' 'boy' 'boy']
# Oops, sorry George!
```

Finding the data

Our overall goal in this chapter is to build a service that knows how to classify textual data, the kind we encounter every day. We want our service to tell us which general category a blog post or a news article belongs to. Classifying data like this can be useful in many ways: building the readers' profile and serve relevant ads, recommend products or personalize content served.

Existing corpora

There is a limited number of existing corpora that we could use to achieve our goal.

Check out available corpora

```
# Let's try out Reuters corpus
from nltk.corpus import reuters


# Let's see what are the Reuters categories
print(reuters.categories())
# ['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa', ...


# Let's check out the 20 newsgroups dataset
from sklearn.datasets import fetch_20newsgroups
news20 = fetch_20newsgroups(subset='train')
print(list(news20.target_names))
# ['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', ...
```

As you can notice, the categories we got aren't that helpful for our task. They're narrow in scope and don't cover the entire news spectrum as we would like. This corpora is mostly used in benchmarking tasks rather than in real-world applications. We can do better by gathering our own data, but bare in mind that usually building a corpus is a tedious and expensive endeavour. If you start from scratch, you need to manually go through a lot of news articles and blog posts and pick the most relevant category for each of them. Most the tutorials on text classification use bogus data, e.g. for doing sentiment analysis, or use an existing corpora that will produce in a useless model with no practical application. In the following section, we will take another path and explore some other ideas for gathering data.

Ideas for Gathering Data

The goal here is to build a general tiny corpus. This task is complex and but is only adjacent to our main goal of building a news classifier, so we will use some shortcuts. For practical reasons, we could look at these web resources:

- Project Gutenberg Categories[6] – this can prove useful depending on the domain you are trying to cover. You will need to write a script to download the books, transform them into plain text and use them to train your classifier.
- Reddit – This is a good source of already categorized data. Pick a list of subreddits, assign them to a category, crawl the subreddits and extract the links. Consider each of all the articles extracted from the subreddit as belonging to the same category Here's a list of all subreddits[7]
- Use the Bing Search API[8] to get relevant articles for your categories

The general idea is to find places on the Internet where content is placed in predefined buckets or categories. These buckets can then be assigned to a category from your own taxonomy. Obviously, the process is quite error-prone. After gathering the data, I suggest looking at some random samples and assessing the percentage of it that's correctly categorized. This will give you a rough idea of the quality of your corpus.

Another trick I like to use after collecting the data is building a script that goes through the labelled data and asks if the sample is correctly classified. Don't stress too much on the interface, its only purpose is to do the labelling. A command line interface that accepts `y/n` input, or even a Tinder-like system will do the trick. If the numbers allow you, you can then go manually through the samples that aren't correctly classified and fix them. It may seem like a lot of work, but keep in mind that if it's done right, it can save you a lot of time, especially given that the alternative is to search for articles yourself and manually assign the appropriate label.

Getting the Data

Getting back to our task at hand, we'll use a different web resource for building our corpus: the web bookmarking service Pocket[9]. This service offers an explore feature[10] that requires us to input clear, unambiguous queries in order to get well-

[6] https://www.gutenberg.org/wiki/Category:Bookshelf
[7] https://www.reddit.com/r/ListOfSubreddits/wiki/listofsubreddits
[8] https://azure.microsoft.com/en-us/services/cognitive-services/bing-web-search-api/
[9] https://getpocket.com
[10] https://getpocket.com/explore/

classified articles. Here's why this is a really good idea:

- data is socially curated and highly qualitative: the articles suggested by Pocket are bookmarked by a big number of users
- data is current: suggestions are frequently added to the service
- data is easy to gather: the explore feature can be easily crawled and, at this moment, it doesn't seem to block crawlers

If you want to skip the corpus creation step, I already prepared it in advance and you can download from here: Text Classification Data[11]

If you want to get your hands dirty and do it anyway, here's how we go about it. First, let's figure out which should be the categories and then proceed to collecting the data.

Category Structure and Keywords

```
# If you have specific needs for your corpus, remember to adjust these categories and keywords accordingly.
CATEGORIES = {
    'business': [
        "Business", "Marketing", "Management"
    ],
    'family': [
        "Family", "Children", "Parenting"
    ],
    'politics': [
        "Politics", "Presidential Elections",
        "Politicians", "Government", "Congress"
    ],
    'sport': [
        "Baseball", "Basketball", "Running", "Sport",
        "Skiing", "Gymnastics", "Tenis", "Football", "Soccer"
    ],
    'health': [
        "Health", "Weightloss", "Wellness", "Well being",
        "Vitamins", "Healthy Food", "Healthy Diet"
    ],
    'economics': [
        "Economics", "Finance", "Accounting"
    ],
    'celebrities': [
        "Celebrities", "Showbiz"
    ],
    'medical': [
        "Medicine", "Doctors", "Health System",
        "Surgery", "Genetics", "Hospital"
    ],
    'science & technology': [
        "Galaxy", "Physics",
        "Technology", "Science"
    ],
    'information technology': [
        "Artificial Intelligence", "Search Engine",
```

[11] https://www.dropbox.com/s/715w0myrjdmujox/TextClassificationData.zip?dl=0

```
        "Software", "Hardware", "Big Data",
        "Analytics", "Programming"
    ],
    'education': [
        "Education", "Students", "University"
    ],
    'media': [
        "Newspaper", "Reporters", "Social Media"
    ],
    'cooking': [
        "Cooking", "Gastronomy", "Cooking Recipes",
        "Paleo Cooking", "Vegan Recipes"
    ],
    'religion': [
        "Religion", "Church", "Spirituality"
    ],
    'legal': [
        "Legal", "Lawyer", "Constitution"
    ],
    'history': [
        "Archeology", "History", "Middle Ages"
    ],
    'nature & ecology': [
        "Nature", "Ecology",
        "Endangered Species", "Permaculture"
    ],
    'travel': [
        "Travel", "Tourism", "Globetrotter"
    ],
    'meteorology': [
        "Tornado", "Meteorology", "Weather Prediction"
    ],
    'automobiles': [
        "Automobiles", "Motorcycles", "Formula 1", "Driving"
    ],
    'art & traditions': [
        "Art", "Artwork", "Traditions",
        "Artisan", "Pottery", "Painting", "Artist"
    ],
    'beauty & fashion': [
        "Beauty", "Fashion", "Cosmetics", "Makeup"
    ],
    'relationships': [
        "Relationships", "Relationship Advice",
        "Marriage", "Wedding"
    ],
    'astrology': [
        "Astrology", "Zodiac", "Zodiac Signs", "Horoscope"
    ],
    'diy': [
        'Gardening', 'Construction', 'Decorating',
        'Do it Yourself', 'Furniture'
    ]
}
```

Moving forward, here's what we'll be doing next:

- querying the service and scraping the article URLs from the page using beauti-

fulsoup4[12].
- iterating through the links and fetching the content of the articles using newspaper3k[13] a library that helps us extract only the main content of a webpage.
- save everything, including the category, in a dataframe and dumping it in a CSV file.

Use Pocket Explore to Build a Corpus

```
import uuid
import atexit
import urllib
import random
import requests
import pandas as pd
from time import sleep, time
from bs4 import BeautifulSoup
from newspaper import Article, ArticleException


POCKET_BASE_URL = 'https://getpocket.com/explore/%s'

df = pd.DataFrame(columns=['title', 'excerpt', 'url', 'file_name', "keyword", "category"])

@atexit.register
def save_dataframe():
    """ Before exiting, make sure we save the dataframe to a CSV file """
    dataframe_name = "dataframe_{0}.csv".format(time())
    df.to_csv(dataframe_name, index=False)

# Shuffle the categories to make sure we are not exhaustively crawling only the first categories
categories = list(CATEGORIES.items())
random.shuffle(categories)

for category_name, keywords in categories:
    print("Exploring Category=\"{0}\"".format(category_name))
    for kw in keywords:
        # Get trending content from Pocket's explore endpoint
        result = requests.get(POCKET_BASE_URL % urllib.parse.quote_plus(kw))

        # Extract the media items
        soup = BeautifulSoup(result.content, "html5lib")
        media_items = soup.find_all(attrs={'class': 'media_item'})
        for item_html in media_items:
            title_html = item_html.find_all(attrs={'class': 'title'})[0]
            title = title_html.text


            url = title_html.a['data-saveurl']
            print("Indexing article: \"{0}\" from \"{1}\"".format(title, url))

            excerpt = item_html.find_all(attrs={'class': 'excerpt'})[0].text

            try:
                article = Article(url)
                article.download()
```

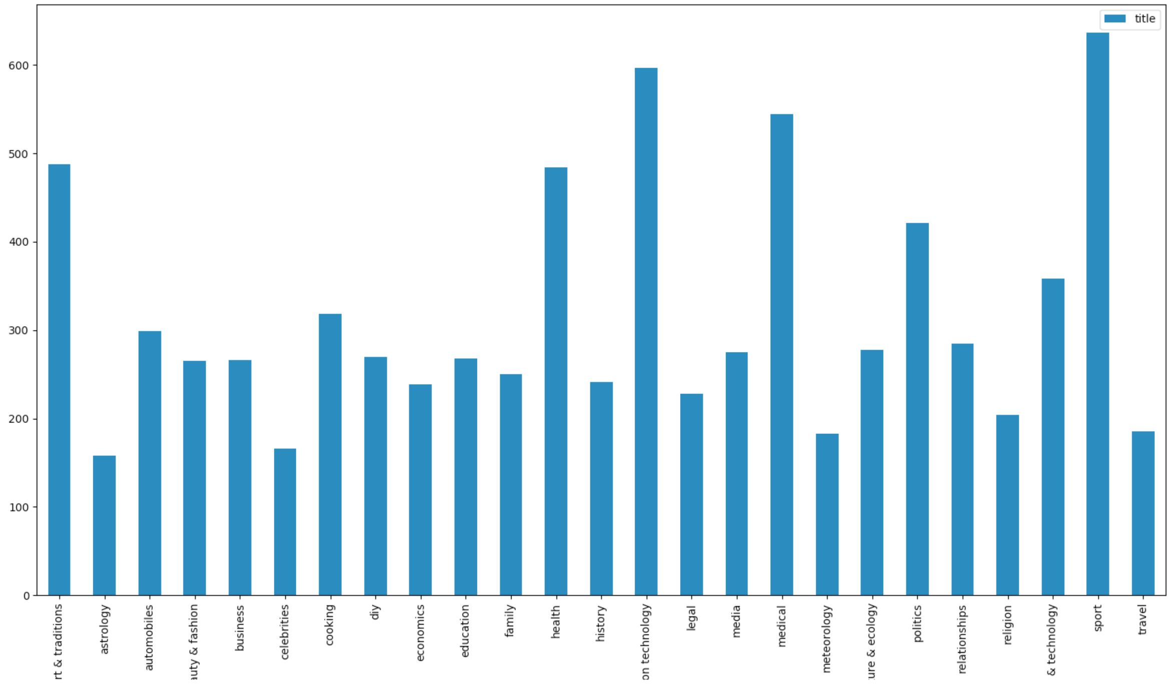
[12] https://pypi.python.org/pypi/beautifulsoup4
[13] https://pypi.python.org/pypi/newspaper3k/0.2.2

```
        article.parse()
        content = article.text
    except ArticleException as e:
        print("Encoutered exception when parsing \"{0}\": \"{1}\"".format(url, str(e)))
        continue

    if not content:
        print("Couldn't extract content from \"{0}\"".format(url))
        continue

    # Save the text file
    file_name = "{0}.txt".format(str(uuid.uuid4()))
    with open('./data/files/{0}'.format(file_name), 'w+') as text_file:
        text_file.write(content)

    # Append the row in our dataframe
    df.loc[len(df)] = [title, excerpt, url, file_name, kw, category_name]
    # Need to sleep in order to not get blocked
    sleep(random.randint(5, 15))
```

This is an extensive process and it will take a while for everything to download. If you can't leave it running overnight, you can run the script several times and then concatenate the results. Remember, if you want to get the data directly, or you want to work on the exact same data as me, you can download the data from here: Text Classification Data[14]

Here's how the distribution of the samples included in the above link looks like:

[14] https://www.dropbox.com/s/715w0myrjdmujox/TextClassificationData.zip?dl=0



Sample size for each category

Learning to Classify Text

With the data we gathered in the previous section, we can now build our text analysis service. At the core, the service will use a classifier similar to the one we created earlier, in chapter 2.2. The main difference would be that the input for this classifier won't be a single word, but an entire block of text.

This means we can use words as features and more precisely, we can use word counts as features. We will then train a classifier to decipher the correlations between the word counts for us. The main thing to note here is that categories contain certain words with a certain frequency.

Text Feature Extractor

Remember that in the previous chapter we transformed a word, a name to be more specific, into a feature vector. Now, will perform a similar operation on a block of text. Let's take a random sentence and transform it into the feature space:

> *How much wood does a woodchuck chuck if a woodchuck could chuck wood*

This is how this sentence would look transformed into the feature space:

Tongue Twister to Feature Space

```
{
    'wood': 2,
    'a': 2,
    'woodchuck': 2,
    'chuck': 2,
    'wow': 1,
    'much': 1,
    'does': 1,
    'if': 1,
    'could': 1
}
```

Here's how simple it is to transform it:

Convert Text to Dictionary

```
import collections

text = """
How much wood does a woodchuck chuck
if a woodchuck could chuck wood
"""

print(collections.Counter(text.lower().split()))
```

One thing that might throw you off at this point is that this method doesn't take into consideration the order of the words inside a text. This is one of the known drawbacks of this method. Using word counts is an approximation and this type of approximation is called **Bag of Words**.

There are methods that deal with actual sequences, such as *Hidden Markov Models* or *Recurrent Neutral Networks*, but using these type of methods are not the subject of this book.

Bag of Words models are really popular and widely used because they are simple and can perform pretty well. We can get even better approximations of the sequence using *bigram* or *trigram* models.

As we discussed in the first chapters, a *bigram* is a pair of adjacent words inside a sentence and a trigram is a triplet of such words. Here's how to compute them for a given text using `nltk` utils:

Compute Bigram and Trigram Features

```
import nltk
import collections
from pprint import pprint

text = """
How much wood does a woodchuck chuck
if a woodchuck could chuck wood
"""

bigram_features = collections.Counter(
    list(nltk.bigrams(text.lower().split())))

trigram_features = collections.Counter(
    list(nltk.trigrams(text.lower().split())))


pprint(bigram_features)
pprint(trigram_features)
```

Here are the results:

Bigram and Trigram Features

```
{
    ('a', 'woodchuck'): 2,
    ('how', 'much'): 1,
    ('much', 'wood'): 1,
    ('wood', 'does'): 1,
    ('does', 'a'): 1,
    ('woodchuck', 'chuck'): 1,
    ('chuck', 'if'): 1,
    ('if', 'a'): 1,
    ('woodchuck', 'could'): 1,
    ('could', 'chuck'): 1,
    ('chuck', 'wood'): 1
}

{
    ('how', 'much', 'wood'): 1,
    ('much', 'wood', 'does'): 1,
    ('wood', 'does', 'a'): 1,
    ('does', 'a', 'woodchuck'): 1,
    ('a', 'woodchuck', 'chuck'): 1,
    ('woodchuck', 'chuck', 'if'): 1,
    ('chuck', 'if', 'a'): 1,
    ('if', 'a', 'woodchuck'): 1,
    ('a', 'woodchuck', 'could'): 1,
    ('woodchuck', 'could', 'chuck'): 1,
    ('could', 'chuck', 'wood'): 1
}
```

Some things to have in mind before we move forward: using bigrams and trigrams makes the feature space way larger because of the combinatorial explosion, meaning:

- The vocabulary size is $|V| = N$
- The size of feature space of the Bag Of Words model is N
- The size of feature space of the Bigram model is at most N^2
- The size of feature space of the Trigram model is at most N^3

Scikit-Learn Feature Extraction

Now that we've covered how a text gets transformed to features, we can move on to how this is actually done in practice. Scikit-Learn has special vectorizers for dealing with text that come in handy.

Scikit-Learn CountVectorizer

```
from sklearn.feature_extraction.text import CountVectorizer


text = """
How much wood does a woodchuck chuck
if a woodchuck could chuck wood
"""

vectorizer = CountVectorizer(lowercase=True)

# "train" the vectorizer, aka compute the vocabulary
vectorizer.fit([text])

# transform text to features
print(vectorizer.transform([text]))

#  (0, 0)        2
#  (0, 1)        1
#  (0, 2)        1
#  (0, 3)        1
#  (0, 4)        1
#  (0, 5)        1
#  (0, 6)        2
#  (0, 7)        2
```

Analyzing the results, you will notice that the right column represents the counts of the words which have exactly the same values as the ones we computed earlier with the Counter function, while the left column represents the indices `(sample_index, word_index_in_vocabulary)`.

Scikit-Learn works mainly with matrices and almost all components require a matrix as input. The vectorizer transforms a list of texts (notice how both `fit` and `transform` get a list of texts as input) into a matrix of size `(sample_count, vocabulary_size)`. The purpose of the `fit` method is to compute the vocabulary (thus the vocabulary size) by computing how many different words we have. Let's find out what happens when we use words outside the vocabulary:

Scikit-Learn CountVectorizer

```
result = vectorizer.transform(["Unseen words", "BLT sandwich"])
print(type(result), result, result.shape)
```

As you can notice, we get an empty matrix with all values set to 0. That means none of the known features (words) have been detected and that's why our print statement doesn't output anything.

Text Classification with Naive Bayes

In my opinion, understanding how vectorizers work on text is probably the hardest part of text classification. By now we covered all the essentials and that is well understood, so we're now ready to read the data and train our classifier.

Scikit-Learn MultinomialNB

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split

# Let's print a snippet of the documentation for Naive Bayes classifier
print(MultinomialNB.__doc__[:415])

# Remember this is where we saved all the data we crawled previously
data = pd.read_csv('./text_analysis_data.csv')

# Where we keep the actual texts
text_samples, labels = []
for idx, row in data.iterrows():
    with open('./clean_data/{0}'.format(row['file_name']), 'r') as text_file:
        text = text_file.read()
        text_samples.append(text)
        labels.append(row['category'])

# Split the data for training and for testing and shuffle it
# keep 20% for testing, and use the rest for training
# shuffling is important because the classes are not random in our dataset
labels = data['category'].as_matrix()
X_train, X_test, y_train, y_test = train_test_split(
    text_samples, labels, test_size=0.2, shuffle=True)

vectorizer = CountVectorizer(lowercase=True)

# Compute the vocabulary using only the training data
vectorizer.fit(X_train)

# Transform the text list to a matrix form
X_train_vectorized = vectorizer.transform(X_train)

classifier = MultinomialNB()

# Train the classifier
classifier.fit(X_train_vectorized, y_train)

# Vectorize the test data
X_test_vectorized = vectorizer.transform(X_test)

# Check our classifier performance
score = classifier.score(X_test_vectorized, y_test)
print("Accuracy=", score)
```

For this task, I have opted for the Naive Bayes classifier as it's one of the most used and well-known types of classifiers for text. Moreover, even its documentation states this:

```
Naive Bayes classifier for multinomial models

The multinomial Naive Bayes classifier is suitable for classification with
discrete features (e.g., word counts for text classification). The
multinomial distribution normally requires integer feature counts. However,
in practice, fractional counts such as tf-idf may also work.

Read more in the :ref:`User Guide <multinomial_naive_bayes>`.
```

Chances are, if you've ever went through a text classification tutorial, you have used a *Naive Bayes* classifier.

After training, we got an accuracy of around `0.65` (or `65%`). The accuracy may vary a bit because we shuffle the dataset. This means we don't train the classifier on the same data every time we run the script. Getting a `65%` accuracy might not seem good enough, and frankly, it probably isn't if you expect people to pay for your text analysis service. However, it's not a bad number and here's why:

- The dataset size was pretty small. It's hard to say what's an appropriate dataset size for this task but my recommendation would be at least a couple of tens of thousands.
- The dataset is not gold quality, meaning it wasn't humanly annotated. If you analyze the dataset, you'll see misclassified samples (even if you only look at titles). This misclassified samples can damage the overall performance of the classifier.
- A `65%` accuracy on a 2-class problem is different than a `65%` accuracy on a 25-class problem (which is our case). If we were to select a random class from the 25 classes for every single sample, we'd only achieve a `4%` accuracy. But if we select randomly from the 2-class problem, we would achieve `50%`. Therefore, the fact that we were able to achieve 65% on our problem doesn't sound that bad. It's a good practice to compare its performance to a random classifier's performance.

The next step now is to actually classify a text. Let's see how to do that:

Taking the classifier for a spin

```
# Pick a text from the test samples
import random
random_choice = random.randint(0, len(X_test))
text, label = X_test[random_choice], y_test[random_choice]

# Check the text we picked and the expected label
print(text, label)

# Let's use the classifier to predict a label
text_vectorized = vectorizer.transform([text])

# As other scikit-learn methods, predict works on matrices
print(classifier.predict(text_vectorized))
```

Persisting models

The process of creating a text analysis service can be divided into the following steps:

1. Training the model given some labelled data. Usually, the trained model has to satisfy some acceptance criteria. That is why this step is often semi-automated as it requires a certain amount of human intervention, such as adjusting parameter or the training set.
2. Storing and loading the model. We need a way to save the trained model to disk because we don't want to retrain the model every time the server is restarted. Our purpose is to load the trained model from disk, so we can use it later when calling the API.
3. Using the model. The analysis step is usually done through the API and is 100% automated. No need for human intervention. We just use the trained model to classify given texts.

We have already covered steps 1 and 3, now let's see how we can do step 2:

Persisting the model

```
import time
from sklearn.externals import joblib

timestamp = int(time.time())
# Save the vectorizer
joblib.dump(vectorizer, './text_analysis_vectorizer_%s.joblib' % timestamp)

# Save the classifier
joblib.dump(classifier, './text_analysis_classifier_%s.joblib' % timestamp)
```

`joblib` is part of the Python scientific computing ecosystem. Among others, this module provides methods for persisting `numpy` objects (like the ones inside our classifiers and vectorizers) in an efficient manner. An alternative would be using the classic `pickle` package.

Let's load the model and use it:

Loading the model

```
# Load the vectorizer
vectorizer = joblib.load('./text_analysis_vectorizer_%s.joblib' % timestamp)

# Load the classifier
classifier = joblib.load('./text_analysis_classifier_%s.joblib' % timestamp)

# Test the loaded component by classifying a text
print(classifier.predict(vectorizer.transform([
    "Marketing for Business Professionals"])))
```

Building the API

My approach for building the API is really simple and straightforward: we will use the `Flask` web framework in order to create a minimalistic API endpoint for classifying texts.

Building a Flask API

Flask is the usual choice for building a hassle-free API. Let's dive straight in by installing `Flask`.

```
$ pip install Flask
```

Now let's create a file named `api.py`. This will hold our endpoint definition.

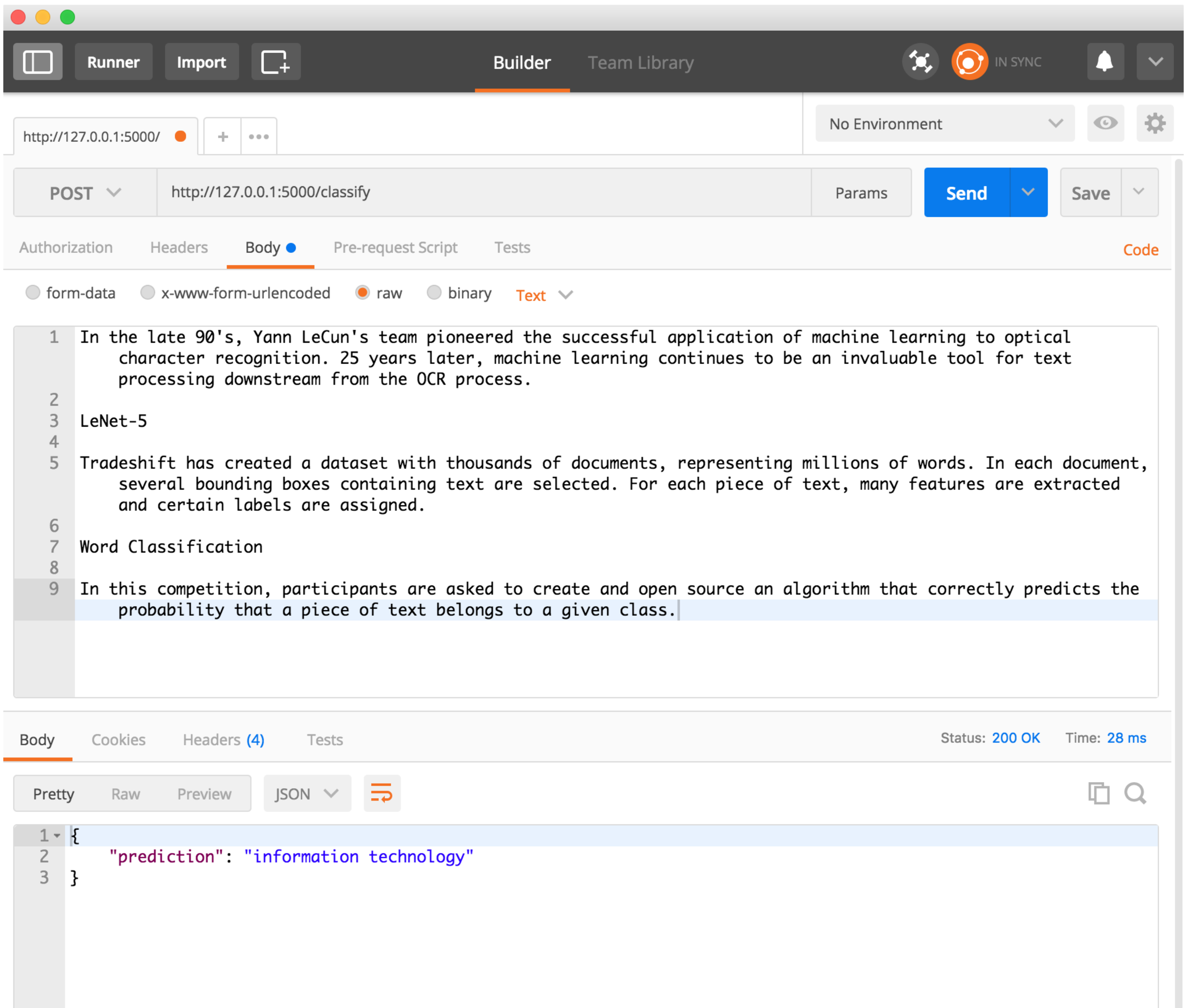
API code

```python
from flask import Flask
from flask import request, jsonify
from sklearn.externals import joblib


app = Flask(__name__)


VECTORIZER_MODEL_PATH = './text_analysis_vectorizer_1504879283.joblib'
CLASSIFIER_MODEL_PATH = './text_analysis_classifier_1504879283.joblib'

# Load the vectorizer and the classifier
vectorizer = joblib.load(VECTORIZER_MODEL_PATH)
classifier = joblib.load(CLASSIFIER_MODEL_PATH)


@app.route("/classify", methods=['POST'])
def classify():
    prediction = classifier.predict(
        vectorizer.transform([request.data]))[0]
    return jsonify(prediction=prediction)
```

Here's how to run the server locally:

```
$ export FLASK_APP=api.py
$ python -m flask run
```

Now you can use your favourite REST client to POST raw text to the `/classify` endpoint. I usually use Postman[15] for this task. Here is how it looks:



Using the API with Postman

If this wasn't a book for the hacker in you, we would have stopped here. But we are going to take things a step further and we'll actually deploy this as an online service using Heroku.

After finishing this chapter, you'll have a text analysis endpoint online and believe it or not, people actually pay money for this kind of service.

[15] https://www.getpostman.com/

First things first: create or login into your Heroku account, and download the Heroku CLI[16], then create a `requirements.txt` file:

requirements.txt file

```
click==6.7
Flask==0.12.2
gunicorn==19.7.1
itsdangerous==0.24
Jinja2==2.9.6
MarkupSafe==1.0
numpy==1.13.1
scikit-learn==0.19.0
scipy==0.19.1
Werkzeug==0.12.2
```

The next step is to install the requirements in a virtual environement:

```
$ pip install -r requirements.txt
```

Heroku needs a `Procfile` to know which one is the file that contains the application initialization:

Heroku Proc file

```
web: gunicorn api:app
```

Before deploying to Heroku, you might also want to create a `.gitignore` file in order to commit only relevant files:

.gitignore file

```
*.pyc
__pycache__
```

Deploy to Heroku

Now the project is ready to be deployed to Heroku:

Login to Heroku:

```
$ heroku login
```

Init the git repository:

[16] https://devcenter.heroku.com/articles/heroku-cli

```
$ git init
```

Init the Heroku application:

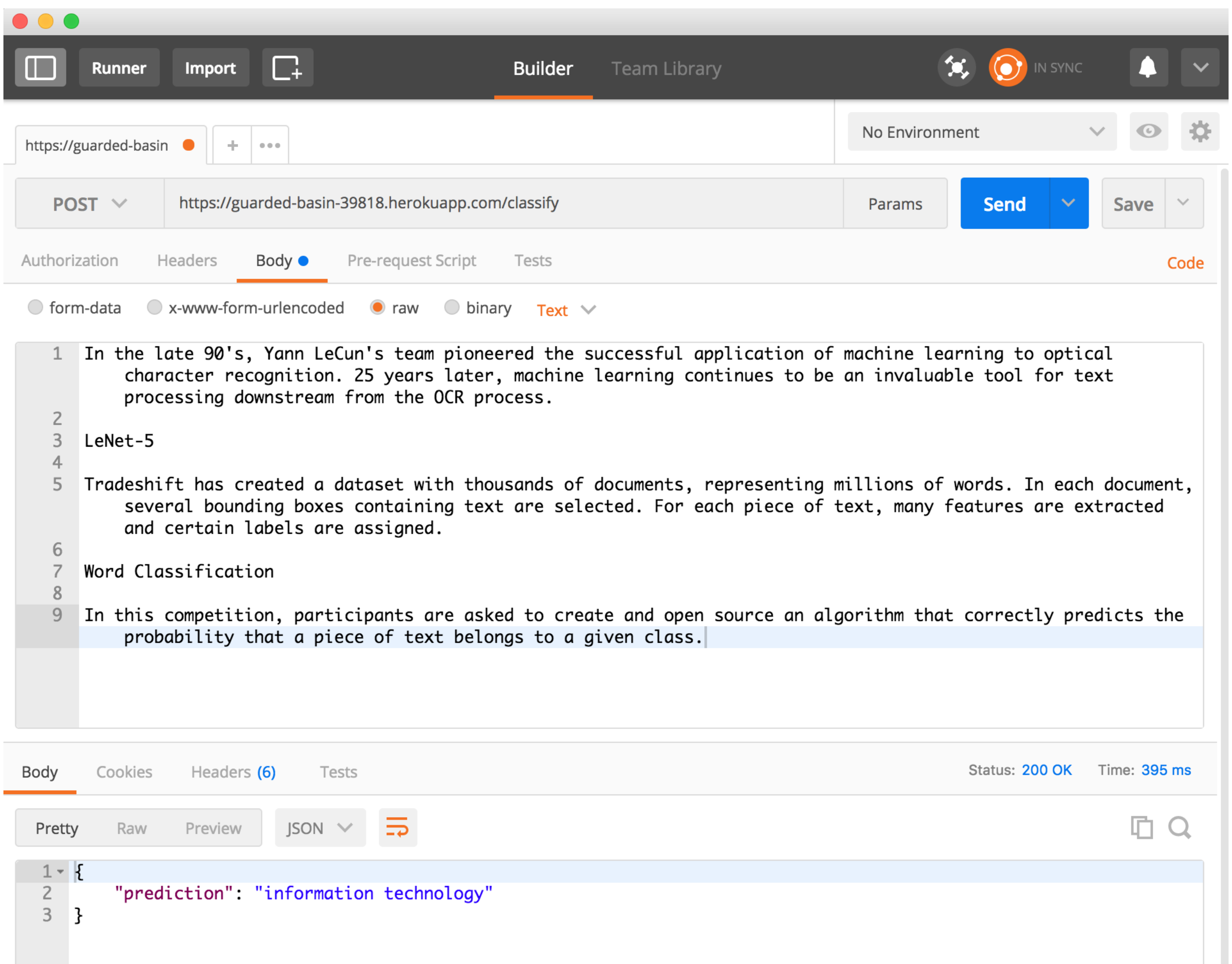
```
$ heroku create
```

Finally, deploy by pushing to the heroku remote master branch:

```
$ git push heroku master
```

If you followed all the steps correctly, you should get a Heroku application URL. I got this one: `https://guarded-basin-39818.herokuapp.com`.

We're ready now to test the online API:



Using the Heroku API with Postman

Part 3: Create a Social Media Monitoring Service

In this third part, we're going to talk about social media monitoring and, in particular, about *Sentiment Analysis* which is one of the most popular and widely used sub-branches of *Natural Language Processing and Understanding*.

Sentiment Analysis means identifying the underlying affective state of a text. In practice, we usually talk about polarity: *is the text transmitting good or bad vibes?* The main objective of sentiment analysis is very similar to the one of text classification. In fact, sentiment analysis is exactly text classification with predefined categories such as:

- *Positive / Negative*
- *Positive / Neutral / Negative*
- *Very-Positive / Positive / Neutral / Negative / Very-Negative*
- *Objective / Subjective*

Sentiment Analysis is used in various applications and the most common are:

1. **Customer Feedback Analysis** (identify the polarity of user reviews)
2. **Brand Monitoring** (quantify a brand's online reputation)
3. **Analyze Political Discourse** (identify political bias)
4. **Subjectivity/Objectivity Analysis** (automatically label news articles for being not objective)

Due to its practical application, there are many more resources available for Sentiment Analysis than for general text classification, and we are going to explore a good deal of them in this chapter.

Basics of Sentiment Analysis

As I mentioned before, Sentiment Analysis represents the task of establishing if a certain text is positive or negative. While this task may sound pretty straightforward, once you get into the details of it, you might encounter some difficulties.

Be Aware of Negations

Consider the following two sentences:

1. *I like hiking very much.*
2. *I don't like hiking very much.*

To a human being, these two sentences are diametrically opposed in sentiment, even if the only difference between them is the word *"don't"*. A single negation might seem negligible when working with full news articles but it might have an important impact on the overall sentiment assessment. How would we handle this situation in a classic ML approach? Things can get even more complicated:

- *"I don't dislike hiking."* - Double negation - Would the sentiment be positive or negative?
- *"I don't like punching in the face."* Basically, I dislike a bad thing. Does that make the sentiment to be positive?

Machine Learning doesn't get Humour

Let's say you're disappointed about a product you bought and when asked in a survey about it, you leave the following comment:

> *Yes, sure … I'm going to buy ten more. Can't wait! Going to the store right now! NOT!*

Well, you might have felt the irony, but computer algorithms and machine learning models have difficulties in understanding humour and figurative speech.
Here's a simpler example:

The experience I had at the Swan Resort Hotel was … interesting

You can sense that *"interesting"* was used in this case with a negative connotation (it suggests that the person was unpleasantly surprised) rather than in its classic positive sense. The suspension points are an indicator of this subtlety. So, would be fair to say that a ML model will interpret the comment in the same way if it had this rule implemented (suspension points = slightly negative sense / unpleasantly surprised)? Would this rule be accurate? Are the suspension points generally used in this way? Well, no and even more, people tend to express humour and figurate speech in many different ways.

Multiple and Mixed Sentiments

What happens if a text expresses multiple feelings? What happens if the sentiments expressed are for different objects (Multiple Sentiments) and what happens if they are for the same object (Mixed Sentiments)? What's the overall feeling of the text?

These are important questions to have in mind, especially when you consider the practical side of Sentiment Analysis. Most of the reviews, comments, articles, etc., have more than one sentence and more than one sentiment expressed towards an object or multiple objects.

Consider the following examples:

"The phone's design is the best I've seen so far, but the battery can definitely use some improvements"

"I know the burger is unhealthy as hell, but damn it feels good eating it!"

Which one expresses multiple sentiments and which one expresses mixed sentiments?

Non-Verbal Communication

Another area where computers still fall short is non-verbal communication. The text we're reading doesn't contain any non-verbal cues. We, using experience, usually attribute a certain tonality to a text. Did it ever happen to fight with somebody in a chat application or email because you weren't correctly interpreting eachother's tone?

Here is a list of what dimensions of communication can be added to a system that is analyzing sentiment:

1. **The Content** (what is being transmitted, basically the text)
2. **The Tone** (the voice inflexions, laughter, etc.)
3. **The Body Language** (nodding, smiling, facial cues, gestures)
4. **Cultural Context** (different cultures have different gestures)

Can you imagine how complex a system can became if it were to take into account all these factors?

Twitter Sentiment Data

Twitter is one of the most open platforms and that makes it a perfect fit for analyzing text from different perspectives. Twitter has been used for predicting election results, reputation management, brand monitoring, customer support, various bots, etc.

Twitter Corpora

Twitter has been noticed in the academic environment for a while now, and various corpora from tweets have been created. Here are some of them:

- `nltk.corpus.twitter_samples` - Sentiment annotated tweets -
- Twitter Airline Reviews[17]
- First GOP Debate Twitter Sentiment[18]
- Sanders Analytics Twitter Sentiment Corpus[19] - 5513 hand-classified tweets
- OSU Twitter NLP Tools[20] - Contains POS, Chunk and NER annotated tweets
- Tweebank[21] - Twitter CoNLL-like annotated data

Other Sentiment Analysis Corpora

Besides the ones stated so far, there is a multitude of other corpora related to sentiment analysis and I am going to list some of the most well-known:

- Sentiment Annotated 50.000 IMDB movie reviews[22]
- Amazon Fine Food Reviews[23]
- Multi-Domain Sentiment Dataset[24]
- UMICH SI650 - Sentiment Classification on Kaggle[25]

---

[17] https://www.kaggle.com/crowdflower/twitter-airline-sentiment
[18] https://www.kaggle.com/crowdflower/first-gop-debate-twitter-sentiment
[19] http://www.sananalytics.com/lab/twitter-sentiment/
[20] https://github.com/aritter/twitter_nlp/tree/master/data/annotated
[21] https://github.com/ikekonglp/TweeboParser/tree/master/Tweebank/Raw_Data
[22] https://www.kaggle.com/c/word2vec-nlp-tutorial/data
[23] https://www.kaggle.com/snap/amazon-fine-food-reviews
[24] http://www.cs.jhu.edu/~mdredze/datasets/sentiment/index2.html
[25] https://inclass.kaggle.com/c/si650winter11/data

- SentiWordnet: Sentiment Polarities for Wordnet[26]
- Miscellaneous Opinion annotated datasets[27]

Building a Tweets Dataset

Let's start by gathering the Twitter data and putting it all together. We are going to use three out of the six resources I have mentioned earlier in the Twitter Corpora chapter: NLTK Twitter Samples, Twitter Airline Reviews and First GOP Debate Twitter Sentiment.

Indexing NLTK Twitter Samples

```
import pandas as pd
import matplotlib.pyplot as plt


# Initialize a dataframe for storing tweets
df = pd.DataFrame(columns=['tweet', 'source', 'sentiment'])

####################################
#
#  NLTK Twitter Samples
#
####################################
from nltk.corpus import twitter_samples

# Add the positive tweets
for tweet in twitter_samples.strings('positive_tweets.json'):
    df.loc[len(df)] = [tweet, 'nltk.corpus.twitter_samples', 'positive']

for tweet in twitter_samples.strings('negative_tweets.json'):
    df.loc[len(df)] = [tweet, 'nltk.corpus.twitter_samples', 'negative']
```

For the next step, you need to download the Twitter Airline Reviews corpus from Kaggle: https://www.kaggle.com/crowdflower/twitter-airline-sentiment[28]

[26] http://sentiwordnet.isti.cnr.it/
[27] https://code.google.com/archive/p/dataset/downloads
[28] https://www.kaggle.com/crowdflower/twitter-airline-sentiment

Indexing Twitter Airline Reviews

```
####################################
#
#  Twitter Airline Reviews
#
####################################
airline_tweets = pd.read_csv('./Tweets.csv')

# Select only the columns of interest
airline_df = airline_tweets[['text', 'airline_sentiment']]

# Rename the columns to fit the header
airline_df = airline_df.rename(columns={'text': 'tweet', 'airline_sentiment': 'sentiment'})

# Add a constant column as the source
airline_df['source'] = 'https://www.kaggle.com/crowdflower/twitter-airline-sentiment'
```

Now let's download the First GOP Debate Twitter Sentiment corpus from Kaggle and process it as well: https://www.kaggle.com/crowdflower/first-gop-debate-twitter-sentiment[29]

Indexing First GOP Debate Twitter Sentiment

```
####################################
#
#  First GOP Debate Twitter Sentiment
#
####################################
debate_tweets = pd.read_csv('./Sentiment.csv')

# Select only the columns of interest
debate_df = debate_tweets[['text', 'sentiment']]

# Rename the columns to fit the header
debate_df = debate_df.rename(columns={'text': 'tweet'})

# Standardize the sentiment column
debate_df['sentiment'] = debate_df['sentiment'].apply(lambda s: s.lower())

# Add a constant column as the source
debate_df['source'] = 'https://www.kaggle.com/crowdflower/first-gop-debate-twitter-sentiment'
```

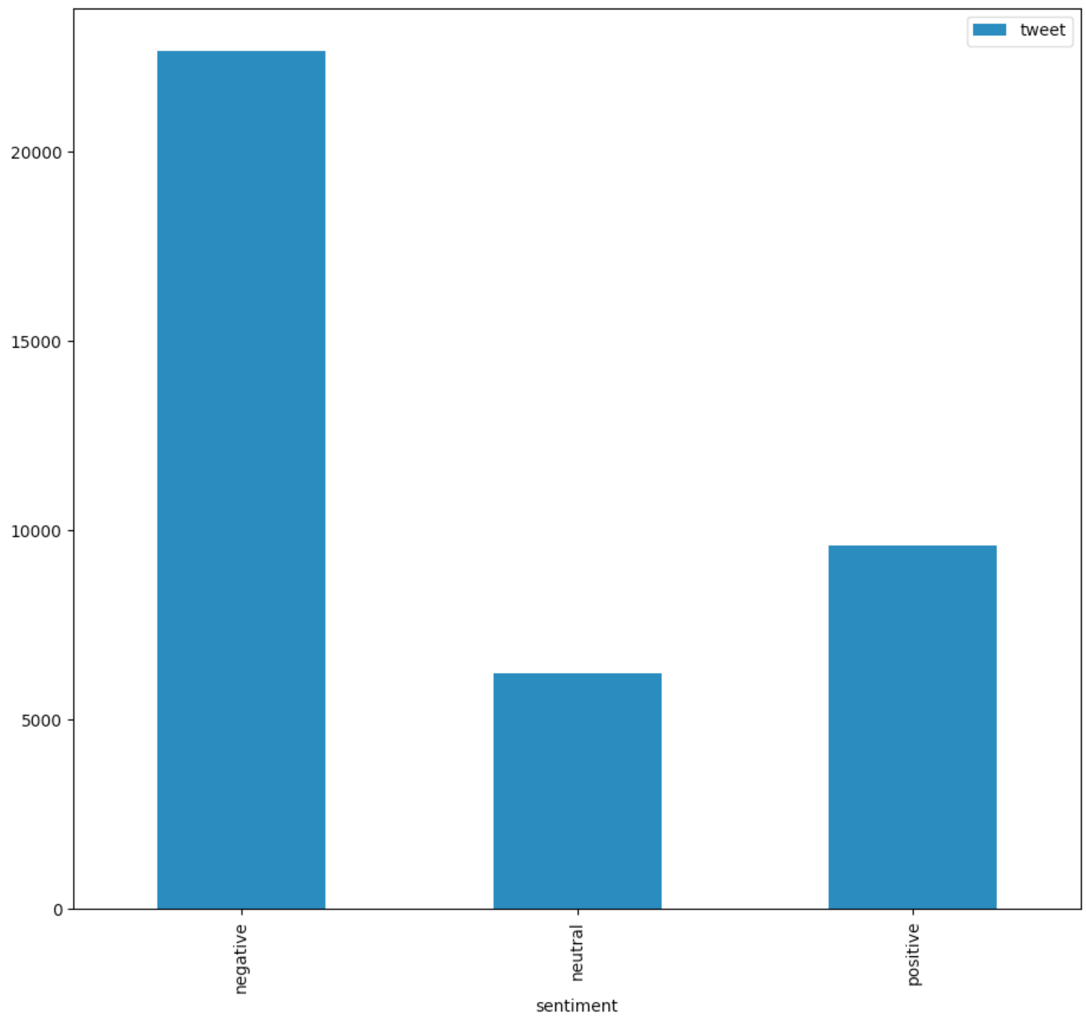
As a final step, let's put everything together and see how many tweets we've got for each polarity:

[29] https://www.kaggle.com/crowdflower/first-gop-debate-twitter-sentiment

Concatenate the data frames and bar plot

```
# Put everything together recomputing the index
df = pd.concat([df, airline_df, debate_df], ignore_index=True)
print(df)

# Let's see how many positive/neutral/negative samples we've got
df[['tweet', 'sentiment']].groupby(['sentiment']).count().plot(kind='bar')

# Make sure the plot doesn't immediately disappear
plt.show(block=True)

print("Total tweets: ", len(df))

# Save all data to file
df.to_csv('twitter_sentiment_analysis.csv')
```

Here's how the plot should look:



Tweet count by sentiment

Observations

- We gathered a good amount of tweets: 38.510.
- Data gathered comes from different areas: a great deal of them are either political or customer interactions tweets.
- Missing neutral class: the NLTK annotated tweets don't have a neutral class which means there might be some tweets, at least in theory, that should be classified as neutral.
- Dataset is rather unbalanced, skewed towards the negative class.

Sentiment Analysis - A First Attempt

We now have the data all together, so let's train a model to predict tweet sentiments by taking the same approach as we did in the Text Analysis chapter.

Training a Sentiment Classifier - First Attempt

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split

data = pd.read_csv('./twitter_sentiment_analysis.csv')

tweets = data['tweet'].values.astype(str)
sentiments = data['sentiment'].values.astype(str)

# Split the data for training and for testing and shuffle it
X_train, X_test, y_train, y_test = train_test_split(tweets, sentiments,
                                                    test_size=0.2, shuffle=True)

vectorizer = CountVectorizer(lowercase=True)

# Compute the vocabulary only on the training data
vectorizer.fit(X_train)

# Transform the text list to a matrix form
X_train_vectorized = vectorizer.transform(X_train)

classifier = MultinomialNB()

# Train the classifier
classifier.fit(X_train_vectorized, y_train)

# Vectorize the test data
X_test_vectorized = vectorizer.transform(X_test)

# Check our classifier performance
score = classifier.score(X_test_vectorized, y_test)
print("Accuracy=", score)
```

I got an accuracy of around `0.71` (71%). It's not bad, but there are some small adjustments we can make to obtain better results. We'll see how that goes in the next chapters.

Better Tokenization

How does the Vectorizer transform text into words? According to the documentation, the vectorizer has two parameters of interest:

- `tokenizer` - a function that performs tokenization on a string.
- `token_pattern` - a regex for extracting tokens from a string. (this is a fallback in case no `tokenizer` function was provided)

Let's take the default value for the `token_pattern` which is `r"(?u)\b\w\w+\b"`. and see how it performs on this tweet:

> *@BeaMiller u didn't follow me :((*

Bad Tweet Tokenization

```
import re
import nltk


tweet = "@BeaMiller u didn't follow me :(("

print(re.findall(r"(?u)\b\w\w+\b", tweet))
# ['BeaMiller', 'didn', 'follow', 'me']

print(nltk.word_tokenize(tweet))
# ['@', 'BeaMiller', 'u', 'did', "n't", 'follow', 'me', ':', '(', '(']
```

We've tried both the regex token pattern and the `nltk.word_tokenize`, but we did not get satisfactory results: none of them caught the emoticon (which is a huge sentiment indicator) or the Twitter handle (which has no sentiment value).

Feel free to build a better performing regex, as it would be a good exercise at this point.

Moving forward, let's use another tokenizer that comes bundled with NLTK and see how it performs: `nltk.tokenize.TweetTokenizer`

NLTK Tweet Tokenization

```
from nltk.tokenize.casual import TweetTokenizer


tweet = "@BeaMiller u didn't follow me :(("

tokenizer = TweetTokenizer()
print(tokenizer.tokenize(tweet))
# ['@BeaMiller', 'u', "didn't", 'follow', 'me', ':(', '(']

tokenizer = TweetTokenizer(strip_handles=True)
print(tokenizer.tokenize(tweet))
# ['u', "didn't", 'follow', 'me', ':(', '(']
```

As you can see, it's not perfect, but it's slightly better. Let's try our new fancy tokenizer for the sentiment analysis task and see where it takes us:

Better Tweet Tokenization

```
# ...
from nltk.tokenize.casual import TweetTokenizer


tweet_tokenizer = TweetTokenizer(strip_handles=True)

# ...

vectorizer = CountVectorizer(lowercase=True, tokenizer=tweet_tokenizer.tokenize)

# ...
```

I got around `0.78` accuracy (78%)! That a **huge** boost in accuracy with a very small adjustment. Pretty cool `:)`!

Fine Tuning

In this chapter we will go over several strategies for boosting the accuracy of a classifier.

Try a different classifier

One of the classifiers that is always worth trying out is the `LogisticRegression` one from `nltk`. It is very versatile and especially good with text. The main advantage of this classifier is that it doesn't need any parameter adjustments, just like the *Naive Bayes* we've been experimenting with. Only change you need to make to the previous script to try this out is:

Try LogisticRegression Classifier

```
# ...

from sklearn.linear_model import LogisticRegression

# ...

classifier = LogisticRegression()

# ...
```

Just by doing this change, we got a boost up to `0.80` in accuracy. Nice!

Use Ngrams Instead of Words

The *Scikit-Learn* vectorizer API allows us to use ngrams rather than just words. Remember what we've covered in the previous chapters about ngrams? It's exactly the same procedure. Instead of using only single-word features, we use consecutive, multi-word features as well. Changes to the previous script to make this happen are minimal:

Using Ngram Features

```
# ...

vectorizer = CountVectorizer(lowercase=True,
                             tokenizer=tweet_tokenizer.tokenize,
                             ngram_range=(1, 3))

# ...
```

Happy day, now we're up to `0.82` in accuracy.

Using a Pipeline

Using a pipeline has a bunch of benefits:

- the ansamble of components behaves as a single classifier
- the code is clean and encapsulated
- it is easy to iterate on improving the model (more about that in the following section)

Using a Pipeline

```
import pandas as pd
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from nltk.tokenize.casual import TweetTokenizer

tweet_tokenizer = TweetTokenizer(strip_handles=True)

data = pd.read_csv('./twitter_sentiment_analysis.csv')

tweets = data['tweet'].values.astype(str)
sentiments = data['sentiment'].values.astype(str)

# Split the data for training and for testing and shuffle it
X_train, X_test, y_train, y_test = train_test_split(tweets, sentiments,
                                                    test_size=0.2, shuffle=True)

# Put everything in a Pipeline
pipeline = Pipeline([
    ('vectorizer', CountVectorizer(
        lowercase=True,
        tokenizer=tweet_tokenizer.tokenize,
        ngram_range=(1, 3))),
    ('classifier', LogisticRegression())
])

# Compute the vocabulary and train the classifier
```

```
pipeline.fit(X_train, y_train)

# Check our classifier performance
score = pipeline.score(X_test, y_test)
print("Accuracy=", score)
```

Cross Validation

This strategy might seem a bit harder to grasp, but bare with me and we'll get to the bottom of it.

I stated earlier in the book that we need to keep the test data separate from the train data, in order to not influence classifier's output. Did you wonder why that is? Well, if we test the system with the same data we trained on, obviously we would get awesome results, but biased. In order to get valid results, we need to test the system with data it hasn't seen yet.

If we continuously tweak the parameters to improve the results on the test set, we indirectly overfit the system on the test dataset. That would be an undesired result because it makes the system worse at generalizing. That means that if we will test on unseed data, outside of the test set, it will underperform. One way to fix this problem is to keep some more data aside and never test on this data while we tune the parameters. This type of data is called the *Validation Set*. After we're satisfied with the results on the test set, then and only then we use the *Validation Set* to check how our system is doing on unseen data.

This approach has a huge drawback: Data is usually scarce, and we will be putting even more data aside that's not going to be used for training.

An approach for getting around this drawback would be doing *Cross Validation*. This implies splitting the dataset into `N` folds. The system will be trained `N` times on all the data, each time excluding a different fold, out of the `N` total ones.
At the end, the scores of all trains are averaged. This way we don't waste much data. Here's an example of Cross Validation with `N` = 5 folds we do that:

Cross Validation Score

```
import pandas as pd
from sklearn.utils import shuffle
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import cross_val_score
from nltk.tokenize.casual import TweetTokenizer


tweet_tokenizer = TweetTokenizer(strip_handles=True)

data = pd.read_csv('./twitter_sentiment_analysis.csv')

tweets = data['tweet'].values.astype(str)
sentiments = data['sentiment'].values.astype(str)

# Put everything in a Pipeline
pipeline = Pipeline([
    ('vectorizer', CountVectorizer(
        lowercase=True,
        tokenizer=tweet_tokenizer.tokenize,
        ngram_range=(1, 3))),
    ('classifier', LogisticRegression())
])

tweets, sentiments = shuffle(tweets, sentiments)
print("MeanAccuracy=", cross_val_score(pipeline, tweets, sentiments, cv=5).mean())
```

Using the *Cross Validation* strategy, we're still around `0.82` in accuracy.

Grid Search

As we've seen so far, there are quite a few parameters we can tune to improve accuracy, and we have not explored that many yet. Moreover, there's no way to know for sure what will be the effects of tuning a parameter in a certain way. There's no exact algorithm for tuning a model. Mastering this implies curiosity and lots of practice.

However, here's a simple way of optimizing parameter combinations called *Grid Search*. This technique implies using *Cross Validation* for every possible parameter combination. That's a lot of work, so it will take a while.

Tuning Parameters with GridSearch

```
import pandas as pd
from sklearn.utils import shuffle
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.feature_extraction.text import CountVectorizer
from nltk.tokenize.casual import TweetTokenizer


tweet_tokenizer = TweetTokenizer(strip_handles=True)

data = pd.read_csv('./twitter_sentiment_analysis.csv')

tweets = data['tweet'].values.astype(str)
sentiments = data['sentiment'].values.astype(str)

# Shuffle the data
tweets, sentiments = shuffle(tweets, sentiments)

# Put everything in a Pipeline
pipeline = Pipeline([
    ('vectorizer', CountVectorizer(
        lowercase=True,
        tokenizer=tweet_tokenizer.tokenize,
        ngram_range=(1, 3))),
    ('classifier', LogisticRegression())
])

classifier = GridSearchCV(pipeline, {
    # try out different ngram ranges
    'vectorizer__ngram_range': ((1, 2), (2, 3), (1, 3)),
    # check if setting all non zero counts to 1 makes a difference
    'vectorizer__binary': (True, False),
}, n_jobs=-1, verbose=True, error_score=0.0, cv=5)

# Compute the vocabulary and train the classifier
classifier.fit(tweets, sentiments)

print("Best Accuracy: ", classifier.best_score_)
print("Best Parameters: ", classifier.best_params_)

# Best Accuracy:  0.81920859947
```

The results of Grid searching tell us that the best decision is to go with word, bigram and trigram features, but we should use binary values instead of counts.

Picking the Best Tokenizer

One of the first points we've covered in this book was stemmers. As you remember, stemmers are good for standardizing words to a root form. Ideally, if two words are derived forms of the same root, they should have the same stem. This trick can significantly reduce the number of features, helping us build less complex and more robust models.

Another thing to try improving is negation handling. As you will remember, Sentiment Analysis was performing poorly when negations were used. NLTK has a utility function, `mark_negation`, that tries to solve this problem by marking all words after a negation as negated until the end of the sentence or until another negation is encountered.

We can now use `GridSearchCV` again, to decide what's the best tokenization method to choose, out of these four combinations:

1. `TweetTokenizer` (the one we've been using throughout this part)
2. `TweetTokenizer` + `PorterStemmer`
3. `TweetTokenizer` + `mark_negation`
4. `TweetTokenizer` + `PorterStemmer` + `mark_negation`

Picking the Best Tokenizer

```
from nltk.stem import PorterStemmer
from nltk.sentiment.util import mark_negation

stemmer = PorterStemmer()


def stemming_tokenizer(text):
    return [stemmer.stem(t) for t in tweet_tokenizer.tokenize(text)]


def tokenizer_negation_aware(text):
    return mark_negation(tweet_tokenizer.tokenize(text))


def stemming_tokenizer_negation_aware(text):
    return mark_negation(stemming_tokenizer(text))

tweet = "@rebeccalowrie No, it's not just you.  Twitter have decided to take that feature away :-("

print(tweet_tokenizer.tokenize(tweet))
# ['No', ',', "it's", 'not', 'just', 'you', '.', 'Twitter',
# 'have', 'decided', 'to', 'take', 'that', 'feature', 'away', ':-(']

print(stemming_tokenizer(tweet))
# ['No', ',', "it'", 'not', 'just', 'you', '.', 'twitter',
# 'have', 'decid', 'to', 'take', 'that', 'featur', 'away', ':-(']

print(tokenizer_negation_aware(tweet))
# ['No', ',', "it's", 'not', 'just_NEG', 'you_NEG', '.', 'Twitter',
# 'have', 'decided', 'to', 'take', 'that', 'feature', 'away', ':-(']

print(stemming_tokenizer_negation_aware(tweet))
# ['No', ',', "it'", 'not', 'just_NEG', 'you_NEG', '.', 'twitter',
# 'have', 'decid', 'to', 'take', 'that', 'featur', 'away', ':-(']
```

```python
classifier = GridSearchCV(pipeline, {
    'vectorizer__tokenizer': (
        tweet_tokenizer.tokenize,
        stemming_tokenizer,
        tokenizer_negation_aware,
        stemming_tokenizer_negation_aware,
    )
}, n_jobs=-1, verbose=True, error_score=0.0, cv=5)

# Compute the vocabulary and train the classifier
classifier.fit(tweets, sentiments)

print("Best Accuracy: ", classifier.best_score_)
print("Best Parameters: ", classifier.best_params_)

# Best Accuracy:  0.820688580776
# Best Parameters:  {'vectorizer__tokenizer': <function stemming_tokenizer at 0x10c519488>}
```

We got a small boost using a stemmer, up to `0.82` accuracy, while all the other options performed poorly. The model we got now is good enough. We should save it to disk and reuse it in the next steps:

Picking the Best Tokenizer


```
# Best Accuracy:  0.820688580776
# Best Parameters:  {'vectorizer__tokenizer': <function stemming_tokenizer at 0x10c519488>}
```

Building the Twitter Listener

Having built our model, we now need to make it an active listener for tweets. In order to do that, we need to connect it to a Twitter *"pipe"* through which we will receive tweets, process them and compute statistics. Twitter offers a Streaming API[30] to get tweets sent to us in real time. There's a nice and popular Python library for interacting with Twitter called Tweepy[31]. This library is exactly what we need because it offers support for streaming. Let's install Tweepy:

```
$ pip install tweepy
```

If you don't already have a Twitter app, go ahead and create one here: Twitter Apps[32]. For the purpose of this project, don't set a `callback_url`, we won't need it since we're not building a web application. Instead of getting a callback, we'll be directed to a page with a code we need to copy/paste in the console. After creating your app, go to the *Keys and Access Tokens* tab and save your authentication credentials. We'll need them in the next part.

Let's build a Tweepy Stream Listener that classifies tweets and saves the count for each category. From time to time, the listener will report the insights it has collected:

Building a Tweepy Listener

```python
import tweepy
import webbrowser
from collections import Counter
from sklearn.externals import joblib


class SentimentAnalysisStreamListener(tweepy.StreamListener):
    def __init__(self, model_path):
        self.counts = Counter()
        self.sentiment_classifier = joblib.load(model_path)
        super(SentimentAnalysisStreamListener, self).__init__()

    # this method will be called when a tweet we are interested in is published
    def on_status(self, status):
        sentiment = self.sentiment_classifier.predict([status.text])[0]
        self.counts[sentiment] += 1
        tweet_count = sum(self.counts.values())

        if not tweet_count % 100:
            print({k: v/tweet_count for k, v in self.counts.items()})
            print("Tweets Collected: %d" % tweet_count)
```

30 https://dev.twitter.com/streaming/overview
31 http://tweepy.readthedocs.io/en/v3.5.0/index.html
32 https://apps.twitter.com

Here's how to authenticate your app and grant access:

Launching the Listener

```
if __name__ == "__main__":
    auth = tweepy.OAuthHandler('YOUR_TOKEN', 'YOUR_SECRET')

    redirect_url = auth.get_authorization_url()
    webbrowser.open_new(redirect_url)

    verifier = input('Verifier:')
    auth.get_access_token(verifier)

    listener = SentimentAnalysisStreamListener(
        model_path='./twitter_sentiment.joblib')
    # this registers our Listener as a handler for tweets
    stream = tweepy.Stream(auth=auth, listener=listener)

    stream.filter(track=['trump']) # I've selected "trump" here as a filter for tweets
```

At the time of writing, there are a lot of tweets on the *Trump* subject. It's a prolific example to use as a simple Twitter filter. Here's how the stats look like:

```
{'neutral': 0.21528571428571427, 'negative': 0.7385714285714285, 'positive': 0.046142857142857145}
Tweets Collected: 7000
```

Classification Metrics

Up until now we've been judging our models by a measure called *accuracy*. The accuracy represents the number of correctly classified samples divided by the total number of samples:

$$Accuracy = \frac{CorrectlyClassifiedSamples}{TotalSamples}$$

Here's a quick way of computing that in code using `numpy` or `scikit-learn`:

Computing Accuracy

```
import numpy as np


PREDICTED_CLASSES = np.array(['C1', 'C2', 'C2', 'C1', 'C3', 'C3', 'C2'])
CORRECT_CLASSES   = np.array(['C1', 'C1', 'C2', 'C1', 'C3', 'C2', 'C2'])

print((PREDICTED_CLASSES == CORRECT_CLASSES).mean())
# 0.714285714286


from sklearn.metrics import accuracy_score
print(accuracy_score(CORRECT_CLASSES, PREDICTED_CLASSES))
# 0.714285714286
```

Binary Classification

Let's get back to our tweet sentiment classification task. For simplification we're going to reduce the problem to only two classes: positive and negative. Let's train a classifier and compute its accuracy:

Binary Tweet Classification

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


data = pd.read_csv('./twitter_sentiment_analysis.csv')
# filter out the `neutrals`
data = data[(data['sentiment'].isin(['positive', 'negative']))]

tweets = data['tweet'].values.astype(str)
sentiments = data['sentiment'].values.astype(str)

# Split the data for training and for testing and shuffle it
X_train, X_test, y_train, y_test = train_test_split(tweets, sentiments,
                                                    test_size=0.2, shuffle=True)

vectorizer = CountVectorizer(lowercase=True)
vectorizer.fit(X_train)

X_train_vectorized = vectorizer.transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)

classifier = MultinomialNB()
classifier.fit(X_train_vectorized, y_train)

# Compute our classifier's accuracy
print(accuracy_score(y_test, classifier.predict(X_test_vectorized)))
```

Binary classification is very common, especially in *detection* type problems, such as:

- Predict the presence of a disease
- Spam detection
- Fraud detection

In the case of the disease example, the classes can represent:

- **Positive** – we predict the disease is present
- **Negative** – we predict the disease is not present

Let's elaborate on the following example: say that we have a disease called *DiseaseX* which affects only 0.1% of the population: 1 in every 1000 individuals suffers from *DiseaseX*. We lock ourselves in a laboratory and after a few days of intense Machine Learning work, we build a very accurate classifier for predicting the presence of the DiseaseX. Our predictive system has an accuracy of 99.9%! The secret behind such a high-performant system is: *Always predict the absence of the disease*.

Even though our system is super performant in terms of accuracy, in fact, it doesn't do anything. The system doesn't predict any disease, so it won't actually help anybody. Thus, we need a different performance measure. Let's introduce the following notations:

- **TP** – True Positives (Samples the classifier has correctly classified as positives)
- **TN** – True Negatives (Samples the classifier has correctly classified as negatives)
- **FP** – False Positives (Samples the classifier has incorrectly classified as positives)
- **FN** – False Negatives (Samples the classifier has incorrectly classified as negatives)

This means that:

- *False Positives are Actually Negatives*
- *False Negatives are Actually Positives*

Now, because accuracy has failed us, we want a measure that's better suited for such phenomena. The measure we are looking for doesn't have to be inversely proportional to the whole dataset size, as accuracy is. Ask yourself this question: what is the probability of a positive sample to be detected by our system? Since our system is the always-predict-the-absence-of-the-disease type, **the probability of a positive sample to be detected by our system is ZERO.** This method is called **Recall** and has the following formula:

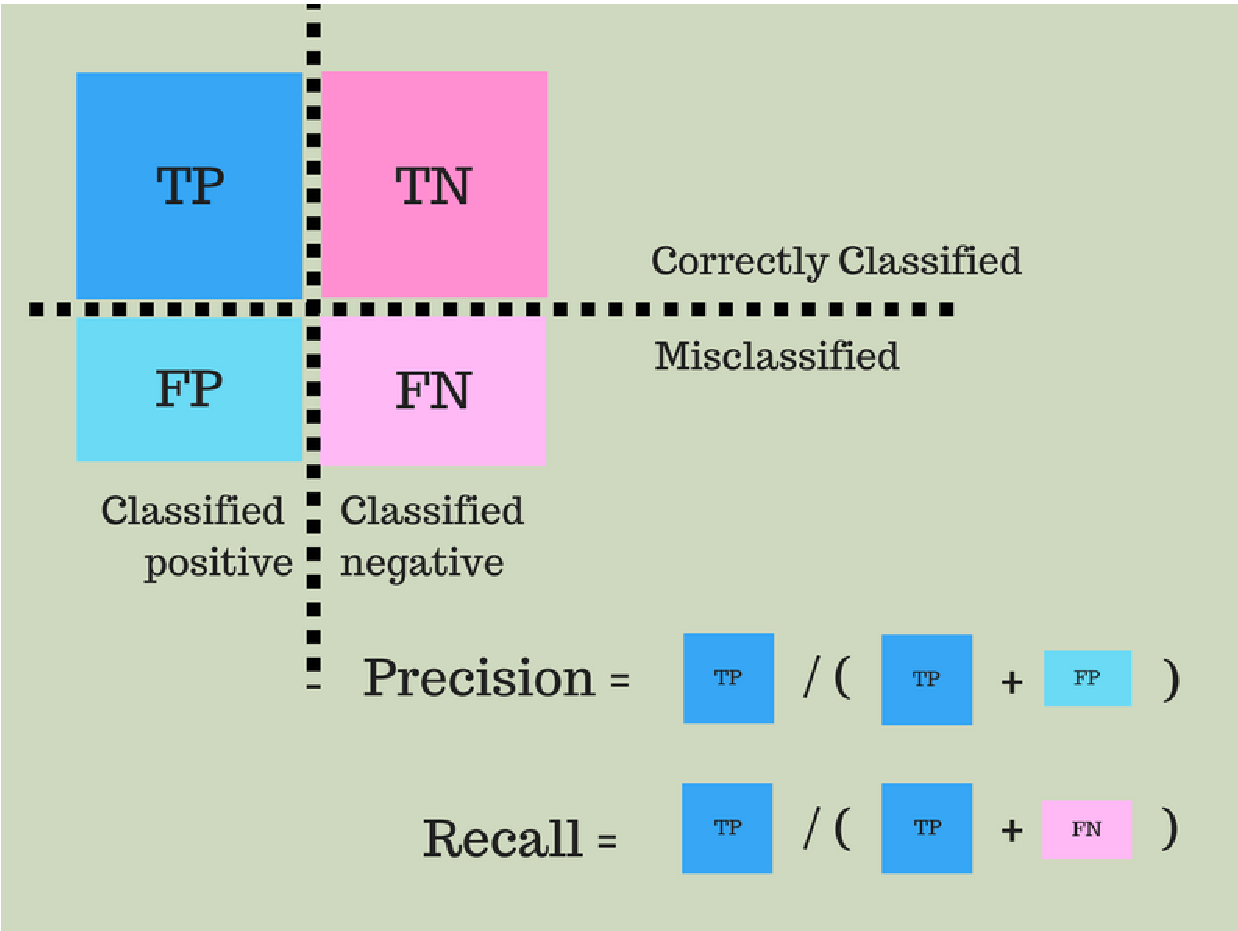
$$Recall = \frac{TP}{TP + FN}$$

Our system has great accuracy, but extremely low recall. If we would make a 180 degrees change and create a system that always predicts the presence of the disease, we would have `0.1%` accuracy, but `100%` recall!

Now ask yourself another question: what's the probability of actually having the disease, if we predicted that the sample has the disease? This measure is called precision and has the following formula:

$$Precision = \frac{TP}{TP + FP}$$

In this case, the precision of the system is 0.1%. Obviously, this scenario is not ideal either.



Simple Way of Remembering Precision and Recall

In order to avoid optimizing one or the other, we can use the *F1-Score* which is the harmonic mean of the *Recall* and *Precision*:

$$F1Score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

Let's go back to our Twitter Sentiment Analysis and see how well we did by looking at these new measures. Usually, Sentiment Analysis is applied in customer support, because we are more interested in solving complaints than reponding to praises. For this reason, the positive class should be represented by the negative samples:

Precision / Recall / F1-Score

```
from sklearn.metrics import recall_score, precision_score, f1_score


print(recall_score(
    y_test, classifier.predict(X_test_vectorized), pos_label='negative'))
# 0.945172564439

print(precision_score(
    y_test, classifier.predict(X_test_vectorized), pos_label='negative'))
# 0.828768435166

print(f1_score(
    y_test, classifier.predict(X_test_vectorized), pos_label='negative'))
# 0.883151341974
```

For performing exploratory analysis, *Scikit-Learn* provides a quick shortcut to view all the metrics for a model:

Classification Report

```
from sklearn.metrics import classification_report

print(classification_report(y_test, classifier.predict(X_test_vectorized)))

#              precision    recall  f1-score   support
#
#    negative       0.82      0.94      0.88      4499
#    positive       0.80      0.54      0.64      1955
#
# avg / total       0.82      0.82      0.81      6454
```

Multi-Class Metrics

Swiching focus to a multi-class problem, let's see how the above metrics are computed. How is this done? Every class takes turns at being a positive class, while all the other classes take the role of the negative. When we are done, we aggregate the numbers we obtained. Here's the code we'll be working with:

Multiclass Sentiment Analysis

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score


data = pd.read_csv('./twitter_sentiment_analysis.csv')
tweets = data['tweet'].values.astype(str)
sentiments = data['sentiment'].values.astype(str)

# Split the data for training and for testing and shuffle it
X_train, X_test, y_train, y_test = train_test_split(tweets, sentiments,
                                                    test_size=0.2, shuffle=True)

vectorizer = CountVectorizer(lowercase=True)
vectorizer.fit(X_train)

X_train_vectorized = vectorizer.transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)

classifier = MultinomialNB()
classifier.fit(X_train_vectorized, y_train)

# Compute our classifier's accuracy
print(accuracy_score(y_test, classifier.predict(X_test_vectorized)))
# 0.698818642087
```

The most common ways of aggregation are:

- Averaging (`macro`)
- Weighted Averaging (`weighted`)

Here's how to do that in code:

Multiclass Precision Recall F1-Score

```
from sklearn.metrics import recall_score, precision_score, f1_score


print(recall_score(
    y_test, classifier.predict(X_test_vectorized), average='weighted'))
# 0.698818642087

print(precision_score(
    y_test, classifier.predict(X_test_vectorized), average='weighted'))
# 0.687808952468

print(f1_score(
    y_test, classifier.predict(X_test_vectorized), average='weighted'))
# 0.670033645784


from sklearn.metrics import classification_report

print(classification_report(
    y_test, classifier.predict(X_test_vectorized)))

#              precision    recall  f1-score   support
#
#    negative       0.71      0.92      0.80      4530
#     neutral       0.51      0.25      0.33      1238
#    positive       0.76      0.48      0.59      1935
#
# avg / total       0.69      0.70      0.67      7703
```

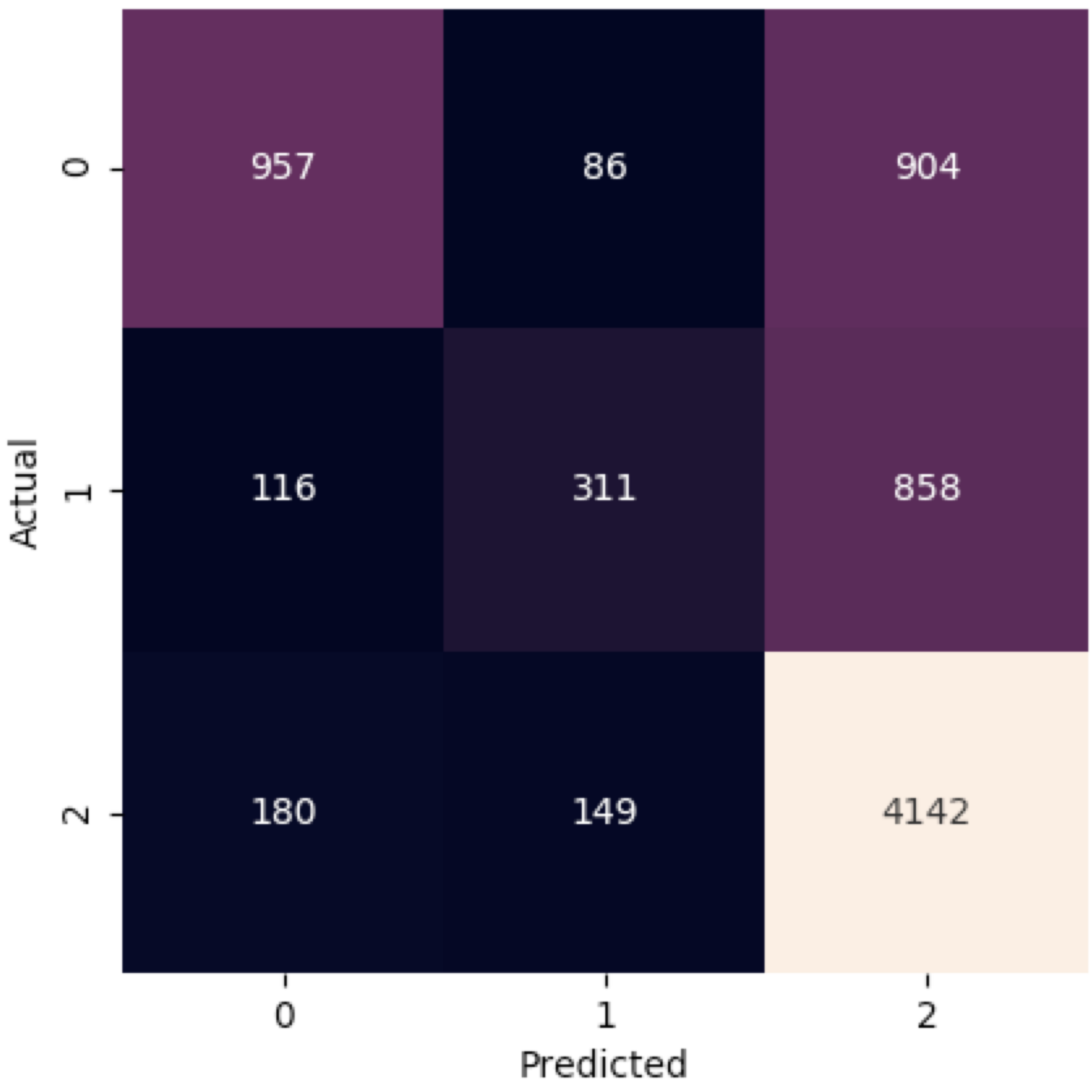
The Confusion Matrix

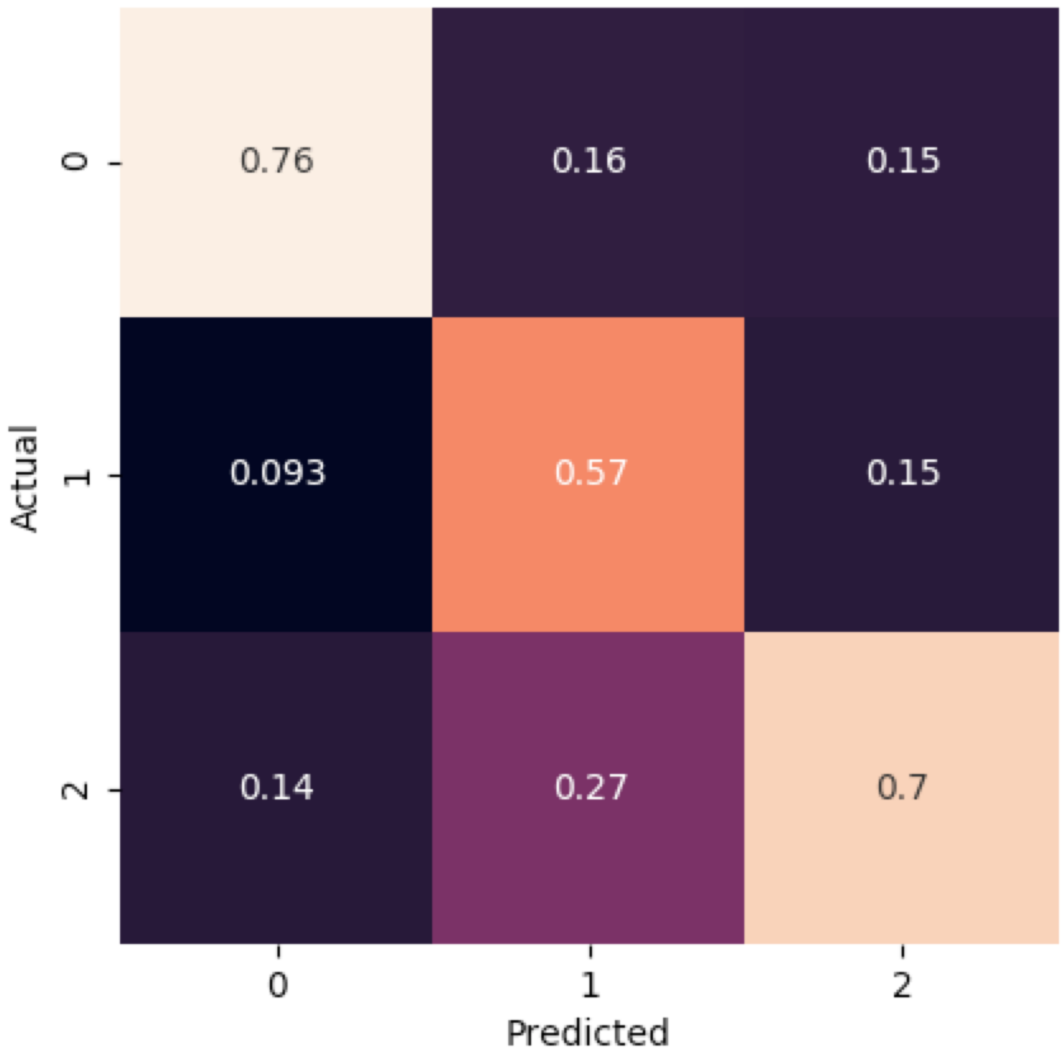
When *investigating* poor classification results, the confusion matrix is a great tool to use. Let's see how to build one for our Twitter multi-class example:

Multiclass Precision Recall F1-Score

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

cm = confusion_matrix(y_test, classifier.predict(X_test_vectorized),
                      labels=['positive', 'neutral', 'negative'])
plt.show(block=True)

sns.heatmap(cm, square=True, annot=True, cbar=False, fmt='d')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show(block=True)

# normalize the confusion matrix
cm = cm.astype(float) / cm.sum(axis=0)
```

```
print(cm)
sns.heatmap(cm, square=True, annot=True, cbar=False)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show(block=True)
```





In the results above, `cm[i][j]` represents the number of items (or the proportion of items) that belong to class `i` but are classified as `j`.

In conclusion: * Ideally, we should see large values only on the first diagonal of the confusion matrix(class `i` gets classified as `i`) * If there are other large values in the matrix, we should investigate because it might mean that our system is oftenly confusing those two classes.

Part 4: Build Your Own NLP Toolkit

While I was learning about NLP, I struggled to understands how the part of speech tagger I was using was built. That's because even if there are a lot of academic papers on this subject, there are no simple step-by-step tutorials that can get you from finding a corpus to building a model.

For this reason, in this chapter I will show you how to build your own toolkit, with available corpora and using libraries we're now familiar with from previous chapters.

What we are going to build in this part of the book:

1. Part Of Speech Tagger
2. Chunker (or Shallow Parser)
3. Named Entity Recognizer
4. Dependency Parser

Build Your Own Part-Of-Speech Tagger

If you feel like you miss some information on Part Of Speech tagging, I recommend you to go and revise that chapter in the book. It's important to be really confident on this topic in order to go through this part successfully. Don't worry, I'll wait here :)

Part-Of-Speech Corpora

To achieve our goal of building a Part of Speech Tagger, we are going to use the Universal Dependecies Dataset[33] You can find complete information on this ambitious project on their website[34] and I recommend you to read a bit about this project before moving forward.

Let's start by downloading the two data files: *"en-ud-train.conllu"*, *"en-ud-dev.conllu"*, and place them somewhere handy.

Have a look at these data files. You will notice how the sentences are separated by a double newline and how each row contains annotations separated by tabs.

Ok, now that we have the data, our first task is to write a function that goes through the data:

Reading Universal Dependencies POS Data

```
import os
from nltk import conlltags2tree


def read_ud_pos_data(filename):
    """
    Iterate through the Universal Dependencies Corpus Part-Of-Speech data
    Yield sentences one by one, don't load all the data in memory
    """
    current_sentence = []

    with open(filename, 'r') as f:
        for line in f:
            line = line.strip()

            # ignore comments
            if line.startswith('#'):
```

[33] https://github.com/UniversalDependencies/UD_English
[34] http://universaldependencies.org/

```
            continue

        # empty line indicates end of sentence
        if not line:
            yield current_sentence

            current_sentence = []
            continue

        annotations = line.split('\t')

        # Get only the word and the part of speech
        current_sentence.append((annotations[1], annotations[4]))
```

Building Toy Models

Instead of aiming for building the best model first, we are going to start by building a simple one. The plan for performing this task are:

1. Train a trigram tagger: When looking for a word `w3`, if we have already encountered a trigram of form: `w1`, `w2`, `w3`, with computed tags `t1`, `t2`, `t3`, we will output tag `t3` for word `w3`. Otherwise, we fallback on the bigram tagger.
2. Train a bigram tagger: When looking for a word `w2`, if we have already encountered a bigram of form: `w1`, `w2`, with computed tags `t1`, `t2`, we will output tag `t2` for word `w2`. Otherwise, we fallback on the unigram tagger.
3. Train a unigram tagger: When looking for a word `w1`, if we have already encountered that word and computed its tags `t1`, we will output tag `t1` for word `w1`. Otherwise, we fallback on a default choice.
4. Implement a default choice: Since all the above methods failed, we can output the most common tag in the dataset.

Starting from the default choice, we'll compute the most common tag and build the taggers:

Training our NLTK Ngram Tagger

```
import nltk
import time
from collections import Counter
from utils import read_ud_pos_data

# Compute the most common tag
tag_counter = Counter()
train_data = read_ud_pos_data('../../../data/en-ud-train.conllu')
for sentence in train_data:
    tag_counter.update([t for _, t in sentence])

# Peek at what are the most common 5 tags
print(tag_counter.most_common(5))
# [('NN', 26915), ('IN', 20724), ('DT', 16817), ('NNP', 12449), ('PRP', 12193)]

most_common_tag = tag_counter.most_common()[0][0]
print("Most Common Tag is: ", most_common_tag) # NN

# Load the data for training
train_data = read_ud_pos_data('../../../data/en-ud-train.conllu')

# Load the data for testing
test_data = read_ud_pos_data('../../../data/en-ud-dev.conllu')

start_time = time.time()
print("Starting training ...")

t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(train_data, backoff=t0)
t2 = nltk.BigramTagger(train_data, backoff=t1)
t3 = nltk.TrigramTagger(train_data, backoff=t2)

end_time = time.time()
print("Training complete. Time={0:.2f}s".format(end_time - start_time))

# Compute test set accuracy
print(t3.evaluate(list(test_data)))  # 0.8391919834579291

# Here's how to use our new tagger
print(t3.tag("This is a test".split()))
```

As you can notice, we've obtained a `0.83` (83%) accuracy. We'll consider this value as the baseline for the statistical models we're going to build next.

About Feature Extraction

Going back to the name gender classifier we built in earlier chapters, you will remember that the first step was to engineer the features to be extracting from the names. Similarly, for this task, we will start by extracting features from the word itself, the previous word and the one following it.

One of the most powerful feature for POS tagging is the shape of the word. Describing a word's shape can include:

- Numbers (1, 1.25, 100000)
- Punctuation (. , ;)
- *Capitalized*
- *UPPERCASE*
- *lowercase*
- *CamelCase*
- *MiXEdCaSE*
- *__WILDCARD__*
- *EndingWithADot.*
- *ABB.RE.VI.ATION.*
- *Contains-Hyphen*

First, let's implement this feature extraction using a regex:

Detecting Word Shape

```
import re


def shape(word):
    if re.match('[0-9]+(\.[0-9]*)?|[0-9]*\.[0-9]+$', word):
        return 'number'
    elif re.match('\W+$', word):
        return 'punct'
    elif re.match('[A-Z][a-z]+$', word):
        return 'capitalized'
    elif re.match('[A-Z]+$', word):
        return 'uppercase'
    elif re.match('[a-z]+$', word):
        return 'lowercase'
    elif re.match('[A-Z][a-z]+[A-Z][a-z]+[A-Za-z]*$', word):
        return 'camelcase'
    elif re.match('[A-Za-z]+$', word):
        return 'mixedcase'
    elif re.match('__.+__$', word):
        return 'wildcard'
    elif re.match('[A-Za-z0-9]+\.$', word):
        return 'ending-dot'
    elif re.match('[A-Za-z0-9]+\.[A-Za-z0-9\.]+\.$', word):
        return 'abbreviation'
    elif re.match('[A-Za-z0-9]+\-[A-Za-z0-9\-]+.*$', word):
        return 'contains-hyphen'

    return 'other'
```

Using the NLTK Base Classes

Since NLTK has some pretty good abstractions available, we're going to follow mostly its structure for defining our base classes. NLTK contains a wrapper class for Scikit-Learn classifiers in `nltk.classify.SklearnClassifier`, but we're not going to use it. That's because we want to write it ourselves, mostly for demonstrative purposes, but also because later on, we'll need some extra functionalities that the NLTK implementation doesn't expose. Let's write our own `SklearnClassifier` wrapper, called `ScikitClassifier`:

Scikit Classifier Wrapper

```
import nltk
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression


class ScikitClassifier(nltk.ClassifierI):
    """
    Wrapper over a scikit-learn classifier
    """
    def __init__(self, classifier=None, vectorizer=None, model=None):
        if model is None:
            if vectorizer is None:
                vectorizer = DictVectorizer(sparse=False)

            if classifier is None:
                classifier = LogisticRegression()

            self.model = Pipeline([
                ('vectorizer', vectorizer),
                ('classifier', classifier)
            ])
        else:
            self.model = model

    @property
    def vectorizer(self):
        return self.model[0][1]

    @property
    def classifier(self):
        return self.model[1][1]

    def train(self, featuresets, labels):
        self.model.fit(featuresets, labels)

    def partial_train(self, featuresets, labels, all_labels):
        self.model.partial_fit(featuresets, labels, all_labels)

    def test(self, featuresets, labels):
        self.model.score(featuresets, labels)

    def labels(self):
        return list(self.model.steps[1][1].classes_)
```

```
    def classify(self, featureset):
        return self.model.predict([featureset])[0]

    def classify_many(self, featuresets):
        return self.model.predict(featuresets)
```

Writing the Feature Extractor

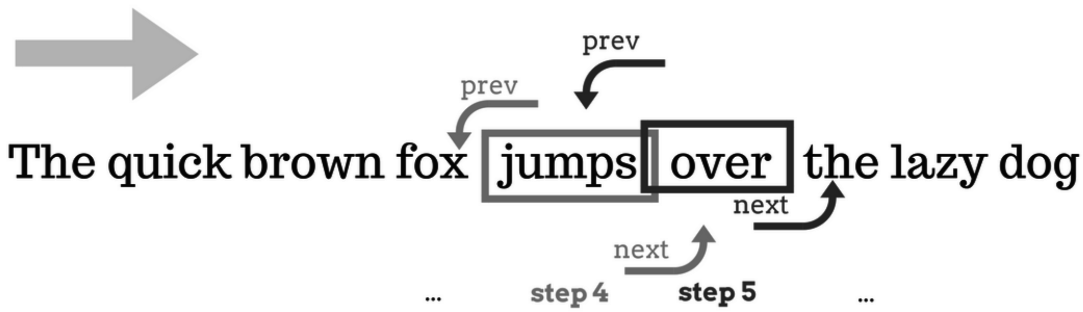
Now, we need to write a feature extraction for a given word in a sentence. Here's how to do it:

Feature Extraction Function

```
from nltk.stem.snowball import SnowballStemmer
from features import shape


stemmer = SnowballStemmer('english')


def pos_features(sentence, index, history):
    """
    sentence = list of words: [word1, word2, ...]
    index = the index of the word we want to extract features for
    history = the list of predicted tags of the previous tokens
    """
    # Pad the sequence with placeholders
    # We will be looking at two words back and forward, so need to make sure we do not go out of bounds
    sentence = ['__START2__', '__START1__'] + list(sentence) + ['__END1__', '__END2__']

    # We will be looking two words back in history, so need to make sure we do not go out of bounds
    history = ['__START2__', '__START1__'] + list(history)

    # shift the index with 2, to accommodate the padding
    index += 2

    return {
        # Intrinsic features
        'word': sentence[index],
        'stem': stemmer.stem(sentence[index]),
        'shape': shape(sentence[index]),

        # Suffixes
        'suffix-1': sentence[index][-1],
        'suffix-2': sentence[index][-2:],
        'suffix-3': sentence[index][-3:],

        # Context
        'prev-word': sentence[index - 1],
        'prev-stem': stemmer.stem(sentence[index - 1]),
        'prev-prev-word': sentence[index - 2],
        'prev-prev-stem': stemmer.stem(sentence[index - 2]),
        'next-word': sentence[index + 1],
```

```
        'next-stem': stemmer.stem(sentence[index + 1]),
        'next-next-word': sentence[index + 2],
        'next-next-stem': stemmer.stem(sentence[index + 2]),

        # Historical features
        'prev-pos': history[-1],
        'prev-prev-pos': history[-2],

        # Composite
        'prev-word+word': sentence[index - 1].lower() + '+' + sentence[index],
    }
```

The function *slides* over the sentence extracting features for every word, taking into account the the tags already assigned (the `history` parameter).



Sliding feature extraction function

Following the NLTK class patterns, we arrive at a class called `nltk.tag.ClassifierBasedTagger`. This is a wrapper over a classifier that knows how to POS tag words. The classifier will be passed to the `ClassifierBasedTagger` via a training function. The class handles the application of the feature detection function over the sentences to transform them into a dataset and knows how to pass everything correctly to a `ClassifierI` instance (in our case this is a `ScikitClassifier` instance).

Let's write our training function:

Tagger Training Function

```
from sklearn.svm import LinearSVC
from classify import ScikitClassifier


def train_scikit_classifier(dataset):
    """
    dataset = list of tuples: [({feature1: value1, ...}, label), ...]
    """
    # split the dataset into featuresets and the predicted labels
    featuresets, labels = zip(*dataset)
```

```
    classifier = ScikitClassifier(classifier=LinearSVC())
    classifier.train(featuresets, labels)
    return classifier
```

I've chosen `LinearSVC` (Linear Support Vector Classifier) as the classifier implementation for this example because it performs very well on this sort of tasks.

Training the Tagger

We are now ready to put everything together. Keep in mind that we are going to use only the first 2000 sentences from the dataset, because otherwise will take an unbearable amount of time for training. Don't worry though, we'll come back to this later.

Training the tagger

```
import time
from nltk.tag import ClassifierBasedTagger
from utils import read_ud_pos_data
from tag import pos_features


if __name__ == "__main__":
    print("Loading data ...")
    train_data = list(read_ud_pos_data('../../../data/en-ud-train.conllu'))
    test_data = list(read_ud_pos_data('../../../data/en-ud-dev.conllu'))
    print("train_data", train_data)
    print("Data loaded .")

    start_time = time.time()
    print("Starting training ...")
    tagger = ClassifierBasedTagger(
        feature_detector=pos_features,
        train=train_data[:2000],
        classifier_builder=train_scikit_classifier,
    )
    end_time = time.time()
    print("Training complete. Time={0:.2f}s".format(end_time - start_time))


    print("Computing test set accuracy ...")
    print(tagger.evaluate(test_data))  # 0.8949021790997296
```

Congratulations, you have just trained your first statistical Part of Speech tagger. An important thing to note here is that even though we used only 2000 sentences for training, we managed to do significantly better (`0.89` accuracy) than using the Ngram tagger (`0.83` accuracy) trained on the full dataset of 12543 sentences.

Out-Of-Core Learning

We got some pretty good results, but we can do even better. The issue with the previous approach is that the data is too large to fit entirely in RAM. In order to tackle this issue, we can divide the data into batches and train the classifier iteratively on each batch.

To implement this approach, we will need to:

- Use a classifier type that can be trained online: This technique is called Out-Of-Core Learning and implies presenting chunks of data to the classifier, several times, rather than presenting the whole dataset at once. There are only a few Scikit-Learn classifiers that have this functionality, mainly the ones implementing the `partial_fit` method. We will use `sklearn.linear_model.Perceptron`.
- Use a vectorizer able to accommodate new features as they are *"discovered"*: This method is called the *Hashing Trick*. Since we won't keep the entire dataset in memory, it will be impossible to know all the features from the start. Instead of computing the feature space beforehand, we create a fixed space that accommodates all our features and assigns a feature to a slot of the feature space using a hashing function. For this purpose, we will use the `sklearn.feature_extraction.FeatureHasher` from Scikit-Learn. The Scikit-Learn vectorizers we've studied so far cannot be altered after calling 'fit' method.

To accommodate all this changes, first we need to extend the functionality of the `ClassifierBasedTagger` class. We'll extract the routine of passing the feature extraction function over the sentence in a separate method. Rather than building the dataset in the `_train` function, we'll need to build a dataset from batches inside the `classifier_builder` function. To achieve that, we need to pass the dataset creation function to the builder. This might sound a bit intricate, but the code is fairly simple:

Extending ClassifierBasedTagger

```python
from nltk import ClassifierBasedTagger
from nltk.metrics import accuracy


class ClassifierBasedTaggerBatchTrained(ClassifierBasedTagger):
    def _todataset(self, tagged_sentences):
        classifier_corpus = []
        for sentence in tagged_sentences:
            history = []
            untagged_sentence, tags = zip(*sentence)
            for index in range(len(sentence)):
                featureset = self.feature_detector(untagged_sentence,
                                                   index, history)
                classifier_corpus.append((featureset, tags[index]))
                history.append(tags[index])
```

```
        return classifier_corpus

    def _train(self, tagged_corpus, classifier_builder, verbose):
        """
        Build a new classifier, based on the given training data
        *tagged_corpus*.
        """

        if verbose:
            print('Constructing training corpus for classifier.')

        self._classifier = classifier_builder(tagged_corpus, lambda sents: self._todataset(sents))

    def evaluate(self, gold):
        dataset = self._todataset(gold)
        featuresets, tags = zip(*dataset)
        predicted_tags = self.classifier().classify_many(featuresets)
        return accuracy(tags, predicted_tags)
```

We wanted to take advantage of the Scikit-Learn vectorized operations, so we had to rewrite the evaluation method as well. The builder function now needs to transform the sentences to a dataset on its own, so we will have to rewrite it:

Rewriting the Classifier builder

```
import time
import itertools
from sklearn.feature_extraction import FeatureHasher
from sklearn.linear_model import Perceptron
from classify import ScikitClassifier
from classify import ClassifierBasedTaggerBatchTrained
from tag import pos_features
from utils import read_ud_pos_data


def incremental_train_scikit_classifier(
        sentences,
        feature_detector,
        batch_size,
        max_iterations):

    initial_corpus_iterator, sentences = itertools.tee(sentences)

    # compute all labels
    ALL_LABELS = set([])

    for sentence in initial_corpus_iterator:
        for w, t in sentence:
            ALL_LABELS.add(t)

    ALL_LABELS = list(ALL_LABELS)

    # This vectorizer doesn't need to be fitted
    vectorizer = FeatureHasher(n_features=1000000)

    classifier = Perceptron(tol=0.00001, max_iter=25, n_jobs=-1)

    for _ in range(max_iterations):
```

```
        current_corpus_iterator, sentences = itertools.tee(sentences)
        batch_count = 0

        while True:
            batch = list(itertools.islice(current_corpus_iterator, batch_size))
            if not batch:
                break
            batch_count += 1
            print("Training on batch={0}".format(batch_count))

            dataset = feature_detector(batch)

            # split the dataset into featuresets and the predicted labels
            featuresets, labels = zip(*dataset)

            classifier.partial_fit(vectorizer.transform(featuresets), labels, ALL_LABELS)

    scikit_classifier = ScikitClassifier(classifier=classifier, vectorizer=vectorizer)

    return scikit_classifier
```

Notice how we never call the `fit` function on the `FeatureHasher`. We just assign a fixed size to it from the start.

We also added some other params to our incremental training function:

- `batch_size` is the number of sentences in each batch
- `max_iterations` is the total number of passes over the dataset

At this point, we need to pass these values to the function via a clojure, since the `_train` function doesn't pass them. Without further ado, here's how to train your dragon. Ah, ups, your online learning system:

Train Online Learning POS Tagger

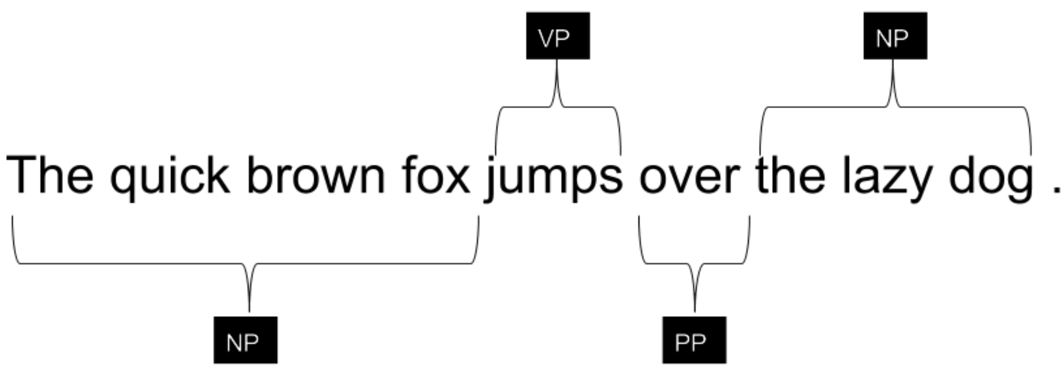
```
if __name__ == "__main__":
    test_data = read_ud_pos_data('../../../data/en-ud-dev.conllu')

    start_time = time.time()
    print("Starting training ...")

    tagger = ClassifierBasedTaggerBatchTrained(
        feature_detector=pos_features,
        train=read_ud_pos_data('../../../data/en-ud-train.conllu'),
        classifier_builder=lambda iterator, detector: incremental_train_scikit_classifier(
            iterator, detector, batch_size=500, max_iterations=30),
    )
    end_time = time.time()
    print("Training complete. Time={0:.2f}s".format(end_time - start_time))

    print("Computing test set accuracy ...")
    print(tagger.evaluate(test_data))  # 0.9255606807698425

    print(tagger.tag("This is a test".split()))
```

I know it takes a while, but don't worry, it is supposed to. And yes, we got another significant boost: `0.93` accuracy!

You have successfully managed to create a complex NLP artefact: a customized Part of Speech Tagger. Congratulations!

Build a Chunker

Chunking, or *Shallow Parsing*, adds more structure on top of a Part-Of-Speech annotated text. The result of chunking is a grouping of consecutive words that serve a single role.



Sentence split into chunks

There's also *normal* parsing, called *deep parsing*, which has as a result an entire syntax tree. As you can probably guess, Deep Parsing is a much more complex task, more prone to errors and much slower than simple chunking. Obviously, it really depends on the situation and deciding which tool would best solve the task should be made on a case-by-case basis. Let's see how each type of parsing looks like:

Deep parse result

```
(S
    (NP The/DT quick/JJ brown/JJ fox/NN)
    (VP jumps/VBZ
        (PP over/IN
            (NP the/DT lazy/JJ dog/NN)))
(. .))
```

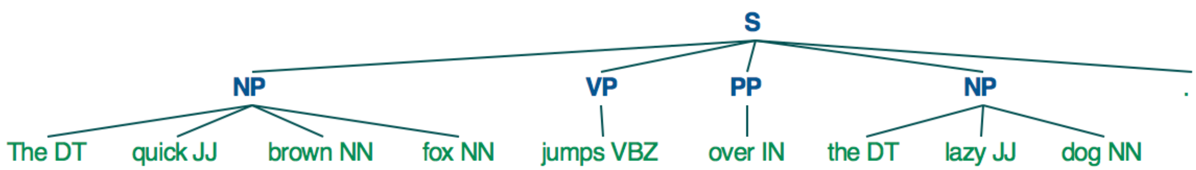
This type of parsing was created using *Stanford Parser*. You can check out their Online Demo[35].

[35] http://nlp.stanford.edu:8080/parser/index.jsp

Shallow parse result

```
(S
    (NP The/DT quick/JJ brown/NN fox/NN)
    (VP jumps/VBZ)
    (PP over/IN)
    (NP the/DT lazy/JJ dog/NN)
./.)
```

Using `nltk.Tree` we can visualize how the chunking of this sentence looks like:



Chunk tree

Here are the chunk labels and their meaning:

Chunk Type	Description
NP	Noun Phrase
VP	Verb Phrase
PP	Prepositional Phrase

IOB Tagging

Shallow parsing is a pretty easy task and is similar to Part-of-Speech tagging. In fact, it can be reduced to a tagging problem. To understand how it works, we need to learn a new tagging technique, called IOB Tagging (or BIO Tagging).

IOB Tagging is useful especially for annotating multi-word (or consecutive words) sequences. As you probably noticed in the previous example, some tokens are grouped under a single label representing such a grouping. Here's how the IOB tags look:

- **B-<CHUNK_TYPE>** : for the word from the **B**eginning of the chunk
- **I-<CHUNK_TYPE>** : for words **I**nside the chunk
- **O** : **O**utside any chunk

To reiterate, the **B-<CHUNK_TYPE>** marks the beginning of a chunk **CHUNK_TYPE** and the **I-<CHUNK_TYPE>** marks that we're inside of a chunk **CHUNK_TYPE**. The

O marks that we're outside of any type of chunk. Here's how our sample sentence looks like when presented in this form:

IOB tagged sentence

```
[
    ('The', 'DT', 'B-NP'),
    ('quick', 'JJ', 'I-NP'),
    ('brown', 'NN', 'I-NP'),
    ('fox', 'NN', 'I-NP'),
    ('jumps', 'VBZ', 'B-VP'),
    ('over', 'IN', 'B-PP'),
    ('the', 'DT', 'B-NP'),
    ('lazy', 'JJ', 'I-NP'),
    ('dog', 'NN', 'I-NP'),
    ('.', '.', 'O')
]
```

Since NLTK comes with a chunking corpus, `CoNLL Dataset`, we don't have any excuse not to explore it:

Exploring the CoNLL Dataset

```
from nltk.corpus import conll2000

print(len(conll2000.chunked_sents()))  # 10948
print(len(conll2000.chunked_words()))  # 166433

chunked_sentence = conll2000.chunked_sents()[0]
print(chunked_sentence)

# (S
#   (NP Confidence/NN)
#   (PP in/IN)
#   (NP the/DT pound/NN)
#   (VP is/VBZ widely/RB expected/VBN to/TO take/VB)
#   (NP another/DT sharp/JJ dive/NN)
#   if/IN
#   (NP trade/NN figures/NNS)
#   (PP for/IN)
#   (NP September/NNP)
#   ,/,
#   due/JJ
#   (PP for/IN)
#   (NP release/NN)
#   (NP tomorrow/NN)
#   ,/,
#   (VP fail/VB to/TO show/VB)
#   (NP a/DT substantial/JJ improvement/NN)
#   (PP from/IN)
#   (NP July/NNP and/CC August/NNP)
#   (NP 's/POS near-record/JJ deficits/NNS)
#   ./.)
```

As you can notice, in NLTK the default way of representing Shallow Parses is with `nltk.Tree`. Here's how we can transform the chunked sentence (represented as a `nltk.Tree`) to IOB tagged triplets:

Converting from Tree to CoNLL triplets

```
from nltk.chunk import tree2conlltags
iob_tagged = tree2conlltags(chunked_sentence)
print(iob_tagged)

# [
#   ('Confidence', 'NN', 'B-NP'),
#   ('in', 'IN', 'B-PP'),
#   ('the', 'DT', 'B-NP'),
#   ('pound', 'NN', 'I-NP'),
#   ('is', 'VBZ', 'B-VP'),
#   ('widely', 'RB', 'I-VP'),
#   ('expected', 'VBN', 'I-VP'),
#   ('to', 'TO', 'I-VP'),
#   ('take', 'VB', 'I-VP'),
#   ('another', 'DT', 'B-NP'),
#   ('sharp', 'JJ', 'I-NP'),
#   ('dive', 'NN', 'I-NP'),
#   ('if', 'IN', 'O'),
#   ('trade', 'NN', 'B-NP'),
#   ('figures', 'NNS', 'I-NP'),
#   ('for', 'IN', 'B-PP'),
#   ('September', 'NNP', 'B-NP'),
#   (',', ',', 'O'),
#   ('due', 'JJ', 'O'),
#   ('for', 'IN', 'B-PP'),
#   ('release', 'NN', 'B-NP'),
#   ('tomorrow', 'NN', 'B-NP'),
#   (',', ',', 'O'),
#   ('fail', 'VB', 'B-VP'),
#   ('to', 'TO', 'I-VP'),
#   ('show', 'VB', 'I-VP'),
#   ('a', 'DT', 'B-NP'),
#   ('substantial', 'JJ', 'I-NP'),
#   ('improvement', 'NN', 'I-NP'),
#   ('from', 'IN', 'B-PP'),
#   ('July', 'NNP', 'B-NP'),
#   ('and', 'CC', 'I-NP'),
#   ('August', 'NNP', 'I-NP'),
#   ("'s", 'POS', 'B-NP'),
#   ('near-record', 'JJ', 'I-NP'),
#   ('deficits', 'NNS', 'I-NP'),
#   ('.', '.', 'O')
# ]
```

Now let's convert back from IOB tagged triplets to `nltk.Tree`:

Converting from CoNLL triplets to Tree

```
from nltk.chunk import conlltags2tree
chunk_tree = conlltags2tree(iob_tagged)
print(chunk_tree)

# (S
#   (NP Confidence/NN)
#   (PP in/IN)
#   (NP the/DT pound/NN)
#   (VP is/VBZ widely/RB expected/VBN to/TO take/VB)
#   (NP another/DT sharp/JJ dive/NN)
#   if/IN
#   (NP trade/NN figures/NNS)
#   (PP for/IN)
#   (NP September/NNP)
#   ,/,
#   due/JJ
#   (PP for/IN)
#   (NP release/NN)
#   (NP tomorrow/NN)
#   ,/,
#   (VP fail/VB to/TO show/VB)
#   (NP a/DT substantial/JJ improvement/NN)
#   (PP from/IN)
#   (NP July/NNP and/CC August/NNP)
#   (NP 's/POS near-record/JJ deficits/NNS)
#   ./.)
```

As I said, and hopefully, as I demonstrate it above, the task of chunking can be reduced to a simple tagging task. Rather than POS tagging words, we're IOB tagging POS-tagged words.

Implementing the Chunk Parser

The base class for implementing chunkers in NLTK is called `nltk.ChunkParserI`. Let's extend that class and wrap inside it the `ClassifierBasedTaggerBatchTrained` we built in the previous chapter.

ClassifierBasedChunkParser Class

```
from nltk import ChunkParserI, tree2conlltags, conlltags2tree
from classify import ClassifierBasedTaggerBatchTrained
from nltk.metrics import accuracy


def triplets2tagged_pairs(iob_sent):
    """
        Transform the triplets to tagged pairs:
        [(word1, pos1, iob1), (word2, pos2, iob2), ...] ->
        [((word1, pos1), iob1), ((word2, pos2), iob2),...]
    """
    return [((word, pos), chunk) for word, pos, chunk in iob_sent]
```

```python
def tagged_pairs2triplets(iob_sent):
    """
        Transform the triplets to tagged pairs:
        [((word1, pos1), iob1), ((word2, pos2), iob2),...] ->
        [(word1, pos1, iob1), (word2, pos2, iob2), ...]
    """
    return [(word, pos, chunk) for (word, pos), chunk in iob_sent]


class ClassifierBasedChunkParser(ChunkParserI):
    def __init__(self, chunked_sents, feature_detector, classifier_builder, **kwargs):
        # Transform the trees in IOB annotated sentences [(word, pos, chunk), ...]
        chunked_sents = [tree2conlltags(sent) for sent in chunked_sents]

        chunked_sents = [triplets2tagged_pairs(sent) for sent in chunked_sents]

        self.feature_detector = feature_detector

        self.tagger = ClassifierBasedTaggerBatchTrained(
            train=(sent for sent in chunked_sents),
            feature_detector=self.feature_detector,
            classifier_builder=classifier_builder
        )

    def parse(self, tagged_sent):
        chunks = self.tagger.tag(tagged_sent)

        iob_triplets = tagged_pairs2triplets(chunks)

        # Transform the list of triplets to nltk.Tree format
        return conlltags2tree(iob_triplets)

    def evaluate(self, gold):
        # Convert nltk.Tree chunked sentences to (word, pos, iob) triplets
        chunked_sents = [tree2conlltags(sent) for sent in gold]

        # Convert (word, pos, iob) triplets to tagged tuples ((word, pos), iob)
        chunked_sents = [triplets2tagged_pairs(sent) for sent in chunked_sents]

        print(chunked_sents)

        dataset = self.tagger._todataset(chunked_sents)
        featuresets, tags = zip(*dataset)
        predicted_tags = self.tagger.classifier().classify_many(featuresets)
        return accuracy(tags, predicted_tags)
```

Chunker Feature Detection

Before finishing, we also need to implement the feature detection function. Let's see how:

Chunker feature detection function

```python
from nltk.stem.snowball import SnowballStemmer

stemmer = SnowballStemmer('english')


def chunk_features(tokens, index, history):
    """
    `tokens`  = a POS-tagged sentence [(w1, t1), ...]
    `index`   = the index of the token we want to extract features for
    `history` = the previous predicted IOB tags
    """

    # Pad the sequence with placeholders
    tokens = ([('__START2__', '__START2__'), ('__START1__', '__START1__')] +
              list(tokens) +
              [('__END1__', '__END1__'), ('__END2__', '__END2__')])
    history = ['__START2__', '__START1__'] + list(history)

    # shift the index with 2, to accommodate the padding
    index += 2

    word, pos = tokens[index]
    prevword, prevpos = tokens[index - 1]
    prevprevword, prevprevpos = tokens[index - 2]
    nextword, nextpos = tokens[index + 1]
    nextnextword, nextnextpos = tokens[index + 2]

    return {
        'word': word,
        'lemma': stemmer.stem(word),
        'pos': pos,

        'next-word': nextword,
        'next-pos': nextpos,

        'next-next-word': nextnextword,
        'nextnextpos': nextnextpos,

        'prev-word': prevword,
        'prev-pos': prevpos,

        'prev-prev-word': prevprevword,
        'prev-prev-pos': prevprevpos,

        # Historical features
        'prev-chunk': history[-1],
        'prev-prev-chunk': history[-2],
    }
```

Now all that's left is putting everything together in the exact same way we did with the tagger previously:

Train chunker

```
if __name__ == "__main__":
    # Prepare the training and the test set
    conll_sents = list(conll2000.chunked_sents())
    random.shuffle(conll_sents)
    train_sents = conll_sents[:int(len(conll_sents) * 0.9)]
    test_sents = conll_sents[int(len(conll_sents) * 0.9 + 1):]

    print("Training Classifier")
    classifier_chunker = ClassifierBasedChunkParser(
        train_sents,
        chunk_features,
        lambda iterator, detector: incremental_train_scikit_classifier(iterator, detector, 1000, 4),
    )
    print("Classifier Trained")
    print(classifier_chunker.evaluate(test_sents))

    print(classifier_chunker.parse(
        pos_tag(word_tokenize("The quick brown fox jumps over the lazy dog."))))

    # (S
    #   (NP The/DT quick/JJ brown/NN fox/NN)
    #   (VP jumps/VBZ)
    #   (PP over/IN)
    #   (NP the/DT lazy/JJ dog/NN)
    #   ./.)
```

We've got a pretty good value for the accuracy of our new chunker: `0.93`.

Conclusions

- Chunking, also called Shallow Parsing, represents a simplified version of Deep Parsing.
- Chunking can be reduced to a tagging problem.
- Chunking can be viewed as IOB tagging of POS-tagged words.

Build a Named Entity Extractor

Named Entity Recognition, or NER,is one of the most useful NLP tools. The purpose is to extract real-world entities from a given text. Integrating NER in an applications helps with identifying entities referred in text. Some of the most common entity types are: persons, organizations, geopolitical entities, locations and events. Under the general Named Entity umbrella, we can also include things like: time expression, money, percentages, numbers, etc.

It may have crossed your mind at this point that NERs are actually chunkers. Well, you're right! And the awesome part is that we already know how to build those. Here's how to extract named entities using `nltk`:

Named Entity Recognition Chunks

```
from nltk import pos_tag, ne_chunk


print(ne_chunk(pos_tag(
    "Twitter Inc. is based in San Francisco , California , United States , "
    "and has more than 25 offices around the world .".split())))


# (S
#   (PERSON Twitter/NNP)
#   (ORGANIZATION Inc./NNP)
#   is/VBZ
#   based/VBN
#   in/IN
#   (GPE San/NNP Francisco/NNP)
#   ,/,
#   (GPE California/NNP)
#   ,/,
#   (GPE United/NNP States/NNPS)
#   ,/,
#   and/CC
#   has/VBZ
#   more/JJR
#   than/IN
#   25/CD
#   offices/NNS
#   around/IN
#   the/DT
#   world/NN
#   ./.)
```

This example above proves two things:

1. Entities are just chunks (multi-word expressions).
2. The NLTK `ne_chunk` is definitely not perfect.

NER Corpora

In order to train our own Named Entity Recognizer, we need a suitable corpus. There is some data available in NLTK, like `ConLL`, `IEER`, that we could use for this purpose. Let's give it a go:

Explore NLTK data

```
from nltk.corpus import conll2002

# Language-independent named entity recognition
print(conll2002.chunked_sents()[0])

from nltk.corpus import ieer

# XML documents without POS tags
print(ieer.raw('APW_19980424'))
```

As you can notice, the NLTK data is not that satisfying: `ConLL` is language independent, while `IEER` is lacking POS tags. Let's give it another try with Groningen Meaning Bank[36], but before we start, I recommend you to download the data[37] first and then put it somewhere accessible.

The Groningen Meaning Bank Corpus

The Groningen Meaning Bank is not a gold standard corpus, meaning it's not manually annotated and it's not 100% correct. GMB is significantly larger than corpuses we've worked with so far.

Let's start by opening a `.tags` file from the corpus and observe its format. As you can notice, the tags are not in standard format:

- Instead of using the standard IOB convention, they use the NE tag: `O, PER, PER, O` instead of `O, B-PER, I-PER, O`
- Some tags are composed: `{TAG}-{SUBTAG}`. For example, `per-giv` stands for `person - given name.`

We need to standardize these tags: use only IOB format, consider only main tags, and not subtags.

Here's a list of the main tags and their meaning:

[36] http://gmb.let.rug.nl/
[37] http://gmb.let.rug.nl/data.php

- geo = Geographical Entity
- org = Organization
- per = Person
- gpe = Geopolitical Entity
- tim = Temporal indicator
- art = Artifact
- eve = Event
- nat = Natural Phenomenon

One important thing to take into consideration is that some tags are very under-represented. Here are the NE tags counts from this corpus:

```
Counter({u'O': 1146068, u'geo': 58388, u'org': 48094, u'per': 44254, u'tim': 34789, u'gpe': 20680, u'art': 867, u'eve': 7\
09, u'nat': 300})
```

We'll filter out the `art`, `eve` and `nat` tags due to this reason.

Here's how you can read the corpus:

Reading the GMB corpus

```
def ner2conlliob(annotated_sentence):
    """
    Transform the pseudo NER annotated sentence to proper IOB annotation

    Example:
    [(word1, pos1, O), (word2, pos2, PERSON), (word3, pos3, PERSON),
     (word4, pos4, O), (word5, pos5, O), (word6, pos6, LOCATION)]

    transforms to:

    [(word1, pos1, O), (word2, pos2, B-PERSON), (word3, pos3, I-PERSON),
     (word4, pos4, O), (word5, pos5, O), (word6, pos6, B-LOCATION)]
    """
    iob_tokens = []
    for idx, annotated_token in enumerate(annotated_sentence):
        tag, word, ner = annotated_token

        if ner != 'O':
            # If it's the first NE is also the first word
            if idx == 0:
                ner = "B-{0}".format(ner)

            # If the previous NE token is the same as the current one
            elif annotated_sentence[idx - 1][2] == ner:
                ner = "I-{0}".format(ner)

            # The NE sequence just started
            else:
                ner = "B-{0}".format(ner)

        iob_tokens.append((tag, word, ner))
    return iob_tokens
```

```python
def read_gmb_ner(corpus_root, start_index=None, end_index=None):
    current_file = -1
    for root, _, files in os.walk(corpus_root):
        for filename in files:
            # Skip other files
            if not filename.endswith(".tags"):
                continue

            current_file += 1
            # Skip files until we get to the start_index
            if start_index is not None and current_file < start_index:
                continue

            # Stop reading after end_index
            if end_index is not None and current_file > end_index:
                return

            with open(os.path.join(root, filename), 'rb') as file_handle:
                # Read the entire file
                file_content = file_handle.read().decode('utf-8').strip()

                # Split into sentences
                annotated_sentences = file_content.split('\n\n')

                for annotated_sentence in annotated_sentences:
                    # Split into annotated tokens
                    rows = [row for row in annotated_sentence.split('\n') if row]

                    ner_triplets = []
                    for row in rows:
                        annotations = row.split('\t')
                        word, tag, ner = annotations[0], annotations[1], annotations[3]

                        # Get only the main tag
                        if ner != 'O':
                            ner = ner.split('-')[0]

                        # Make these tags NLTK compatible
                        if tag in ('LQU', 'RQU'):
                            tag = "``"

                        # Ignore the art,eve,nat tags because they are underrepresented
                        if tag in ('art', 'eve', 'nat'):
                            tag = 'O'

                        ner_triplets.append((word, tag, ner))

                    iob_triplets = ner2conlliob(ner_triplets)

                    # Yield a nltk.Tree
                    yield conlltags2tree(iob_triplets)
    print("Total files=", current_file)
```

You may notice that I've added some extra parameters that we don't necessarily use, such as `start_index` and `end_index`. The role of these parameters is to keep files aside for testing. At this point, there are exactly 10.000 files in the corpus (from 0 to 9999).

Feature Detection

Here is the feature detection function we are going to be using:

NE feature detection

```
from nltk.stem.snowball import SnowballStemmer
from features import shape


stemmer = SnowballStemmer('english')


def ner_features(tokens, index, history):
    """
    `tokens`  = a POS-tagged sentence [(w1, t1), ...]
    `index`   = the index of the token we want to extract features for
    `history` = the previous predicted IOB tags
    """

    # Pad the sequence with placeholders
    tokens = [('__START2__', '__START2__'), ('__START1__', '__START1__')] + list(tokens) + [('__END1__', '__END1__'), ('_\
_END2__', '__END2__')]
    history = ['__START2__', '__START1__'] + list(history)

    # shift the index with 2, to accommodate the padding
    index += 2

    word, pos = tokens[index]
    prevword, prevpos = tokens[index - 1]
    prevprevword, prevprevpos = tokens[index - 2]
    nextword, nextpos = tokens[index + 1]
    nextnextword, nextnextpos = tokens[index + 2]
    previob = history[-1]
    prevpreviob = history[-2]

    return {
        'word': word,
        'lemma': stemmer.stem(word),
        'pos': pos,
        'shape': shape(word),

        'next-word': nextword,
        'next-pos': nextpos,
        'next-lemma': stemmer.stem(nextword),
        'next-shape': shape(nextword),

        'next-next-word': nextnextword,
        'next-next-pos': nextnextpos,
        'next-next-lemma': stemmer.stem(nextnextword),
        'next-next-shape': shape(nextnextword),

        'prev-word': prevword,
        'prev-pos': prevpos,
        'prev-lemma': stemmer.stem(prevword),
        'prev-iob': previob,
        'prev-shape': shape(prevword),
```

```
        'prev-prev-word': prevprevword,
        'prev-prev-pos': prevprevpos,
        'prev-prev-lemma': stemmer.stem(prevprevword),
        'prev-prev-iob': prevpreviob,
        'prev-prev-shape': shape(prevprevword),
    }
```

NER Training

The training routine we're going to use is exactly the one we've used for the chunker, and here's how it looks:

Training NER

```
if __name__ == "__main__":
    # Prepare the training and the test set
    conll_sents = list(conll2000.chunked_sents())
    random.shuffle(conll_sents)
    train_sents = conll_sents[:int(len(conll_sents) * 0.9)]
    test_sents = conll_sents[int(len(conll_sents) * 0.9 + 1):]

    train_sents = read_gmb_ner('../../../data/gmb-2.2.0/data', start_index=0, end_index=8999)
    test_sents = read_gmb_ner('../../../data/gmb-2.2.0/data', start_index=9000, end_index=9999)

    print("Training Classifier")
    classifier_chunker = ClassifierBasedChunkParser(
        train_sents,
        ner_features,
        lambda iterator, detector: incremental_train_scikit_classifier(iterator, detector, 500, 2),
    )
    print("Classifier Trained")
    print(classifier_chunker.evaluate(test_sents))

    print(classifier_chunker.parse(
        pos_tag(word_tokenize(
            "Twitter Inc. is based in San Francisco , California , United States , "
            "and has more than 25 offices around the world ."))))
```

I obtained a `0.975` accuracy. That's pretty awesome. Here's the result of applying the NER on our sample sentence:

```
(S
  (org Twitter/NNP Inc./NNP)
  is/VBZ
  based/VBN
  in/IN
  (geo San/NNP Francisco/NNP)
  ,/,
  (geo California/NNP)
  ,/,
  (geo United/NNP States/NNPS)
  ,/,
  and/CC
  has/VBZ
  more/JJR
  than/IN
  25/CD
  offices/NNS
  around/IN
  the/DT
  world/NN
  ./.)
```

An important thing to consider is that Named Entity Recognition is strongly skewed towards the `O` class. This means that if we always output `O` we would get a decent accuracy anyway. Remember what we talked about in the Precision/Recall chapter?

Conclusions

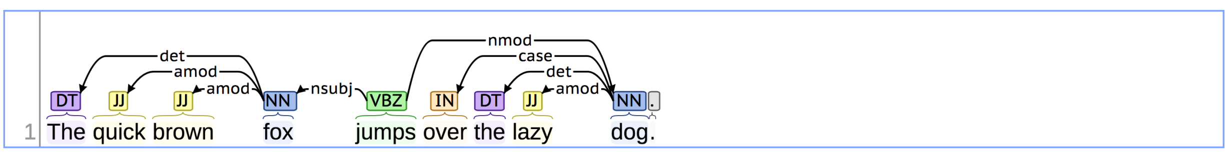
- Named Entity Recognition is in fact pure chunking.
- We used data pre-processing techniques in order to get better results with the GMB corpus: we removed under represented classes and unified sub-tags.

Build a Dependency Parser

In previous chapters, we talked about shallow parsing and mentioned deep parsing as a more complex and heavy version of it. Deep parsing algorithms are called constituency parsers and that's because they try to split the sentence in constituents, then the constituents in its constituents and so on.

There is one more type of parsing we haven't talked about yet, called **Dependency Parsing**, that is neither shallow, nor deep, and this what we will be talking about in this chapter.

Dependency parsing algorithms try to identify the dependency relations between the words of a sentence. Instead of presenting you with a formal definition, let's me show you an of such parsing from the Stanford Core NLP Demo[38]:



Stanford Dependency Parser

Here are the things you should notice:

- All nodes have a single parent, called head. This effectively means, that all nodes have exactly one inbound edge.
- A special ROOT node, added to the sentence for uniformity, has no head, and thus no inbound edge.
- A node can have 0 to any number of dependants, represented through an edge to that dependent
- Edges have labels, due to which this is called a **typed** dependency parse. Here's the Stanford Parser guide to the various types of dependencies[39].
- The edges do not cross. This is true for most languages. It's called *projective dependency parse*. The opposite is called *non-projective dependency parse* and is specific to languages like: German, Dutch and Czech.

As we already mentioned, a node can have only one **head** but it can have any number of **dependents**. That's because the head is the main word of a *phrase*(noun

[38] http://nlp.stanford.edu:8080/corenlp/
[39] http://ufal.mff.cuni.cz/~hladka/2015/docs/Stanford-dependencies-manual.pdf

phrase, verb phrase, etc.) from the sentence. The dependents provide further details about the head. In the example provided above, the root node is *“jumps”*. Here’s how to think about heads and dependents:

1. “jumps”
2. **nsubj**: Who jumps? → “fox jumps”
3. **det**: Which fox jumps? → “The fox jumps”
4. **amod**: What kind of fox jumps? → “The quick brown fox jumps”

You’ll need to read what every dependency relation stands for, but here’s a quick reference of the three we’ve mentioned so far:

- **nsubj**: nominal subject → the dependent is the subject of the sentence
- **det**: determiner → ties a noun to its determiner
- **amod**: adjectival modifier → modifies a noun phase

Until recently, dependency parsers were a very sought-after resource. The corpora was scarce and there weren’t many (if at all) viable ready built dependency parsers to use in a project. Things have changed since then and today we have quite a few options available:

- spaCy Dependency Parser[40]
- Stanford Parser[41]
- Google SyntaxNet[42]
- nlp4j[43]

To get a better grip of how dependency parsing works, I strongly recommend playing with the Stanford Core NLP Demo[44]. For our purposes, we’re are going to use Universal Dependencies and you can find the manual for that here[45].

Now, we’ll try get a deeper understanding of the dependency parsing problem. By this time, you might be wondering how can we reduce dependency parsing to tagging. Sorry, I’ve led you to believe that everything leads to chunking, chunking to tagging, etc. Unfortunately, this is not the case.

[40] https://spacy.io/docs/usage/dependency-parse
[41] https://nlp.stanford.edu/software/lex-parser.shtml
[42] https://github.com/tensorflow/models/tree/master/research/syntaxnet
[43] https://emorynlp.github.io/nlp4j/
[44] http://nlp.stanford.edu:8080/corenlp/
[45] http://universaldependencies.org/u/dep/index.html

Understanding the Problem

One way of looking at the dependency parsing problem might be by trying to reduce it to a classification problem. This is a valid approach, even if at a first glance it doesn't look like a traditional classification problem. The solution of the problem will tell us what's the head of every word in the sentence. Since each nodes has exactly one inbound edge from its *head* node, we need to identify which node is the inbound edge coming from for each node in the sentence. Therefore, we might consider this problem more of a selection problem rather than a classification one.

We could compute the probability of each node being the head of every other node and pick the optimal combination. However, the result is not guaranteed to have a valid structure that respects all the rules listed above. We need to force our algorithm somehow to only give us valid results: without crossing edges, etc.

Leaving the edge labels aside, and only focusing on identifying the source and destination of the edges, let me present to you how Transition-based Dependency Parsing works:

- We start with a stack and a buffer
- The stack is initialized with the first word in the sentence
- The buffer contains all the other words in the sentence
- At each step we have 3 possible operations:
  - **SHIFT** = take the first node from the buffer and add it on top of the stack
  - **LEFT-ARC** = create a left arc from the current item in the buffer to the top node in the stack. Remove the dependent from the stack.
  - **RIGHT-ARC** = create a right arc between the top 2 nodes in the stack (from the one underneath to the one on top). Pop the stack (remove the dependant).

Let's take a really simple example: *I buy green apples*. Now let's apply the operations to get the dependency parse, step-by-step:

Step 0

```
stack  = ["I"]
buffer = ["buy", "green", "apples"]
arcs   = []
```

Step 1 - LEFT-ARC

```
stack  = []
buffer = ["buy", "green", "apples"]
arcs   = [("buy", "I")]
```

Step 2 - SHIFT

```
stack  = ["buy"]
buffer = ["green", "apples"]
arcs   = [("buy", "I")]
```

Step 3 - SHIFT

```
stack  = ["buy", "green"]
buffer = ["apples"]
arcs   = [("buy", "I")]
```

Step 4 - LEFT-ARC

```
stack  = ["buy"]
buffer = ["apples"]
arcs   = [("buy", "I"), ("apples", "green")]
```

Step 5 - SHIFT

```
stack  = ["buy", "apples"]
buffer = []
arcs   = [("buy", "I"), ("apples", "green")]
```

Step 6 - RIGHT-ARC

```
stack  = ["buy"]
buffer = []
arcs   = [("buy", "I"), ("apples", "green"), ("buy", "apples")]
```

Step 7 - LEFT-ARC (draw an arc from the ROOT node to the remaining node)

```
stack  = []
buffer = []
arcs   = [("buy", "I"), ("apples", "green"), ("buy", "apples"), (ROOT, "buy")]
```

The solution to the dependency parse is: `[LEFT-ARC, SHIFT, SHIFT, LEFT-ARC, SHIFT, RIGHT-ARC, LEFT-ARC]`.

Stanford Dependency Parser - Sample Sentence

It may seem pretty straightforward at this point, but there's a catch: for a given dependency tree, more than one solution is possible and there's really no way of knowing which one is the best one.

Now that we have a method for building the tree, it is easy to see that this problem can be turned into a classification one. Since we have a dependency tree dataset, Universal Dependencies one, we now need to convert it to a transitions dataset using the steps described above. The classifier we will train should be able to tell us which transition (`SHIFT`, `LEFT-ARC` or `RIGHT-ARC`) we should apply at each step, given a current `stack` and `buffer`.

Because for a given tree there can be more than one solution, we can't build the transitions dataset, at least not yet. At this point, we really need a way to decide if one transition sequence is better than another. After all, we aim to use the one with the highest score to build this dataset.

Greedy Transition-Based Parsing

Greedy Transition-Based Parsing means that instead of looking for a global optimal solution, we only look at the highest scoring transition sequence at each step. If you studied algorithms, you know already that Greedy Algorithms consider that the global optimal solution is built out of local optimums. Ideally, we should be able to score transitions at every step and there are several ways of doing this. For our purposes, if there are more than one transitions possible, we'll be picking one at random.

Here are a few components of the algorithm we'll need to build:

- `DependencyParse` class: A wrapper over all the data we need to encapsulate: words, tags, dependencies.
- `ParserState` class: Since we're implementing a transition based parsing system, the parser holds an internal state, a wrapper for the buffer, stack, and a `DependencyParse` instance. This state changes when we apply transitions. Once a transition is applied to the state, it will also reflect onto the parse.

- `DependencyParser` class: The parser class that wraps the needed statistical models. It is able to train the models and to use them to parse a tagged sentence.

Bare in mind that at first we're just going to draw the edges. After we fully understand this mechanism, we'll rewrite the code, and add the labelling functionality as well. As promised, this will be an end-to-end solution.

Dependency Dataset

As always, let's start by reading the dataset:

Reading dependency data

```
def iter_ud_dep_data(filename):
    current_sentence = []

    with open(filename, 'r') as f:
        for line in f:
            line = line.strip()

            # ignore comments
            if line.startswith('#'):
                continue

            # empty line indicates end of sentence
            if not line:
                yield current_sentence

                current_sentence = []
                continue

            annotations = line.split('\t')

            # Get only the word, pos, head and dep rel
            try:
                int(annotations[0])
                current_sentence.append(
                    (annotations[1], annotations[4], annotations[6], annotations[7])
                )
            except ValueError:
                pass
```

This is a simple procedure that reads the Universal Dependency file and returns a list of quadruplets for each sentence: `(word, tag, head, label)`. `head` is the index of the node's parent. In fact, the simplest representation of a dependency parse is a parent list: `[parent(word) for word in sentence]`. Here's the `DependencyParse` class we'll be using:

DependencyParse class

```python
class DependencyParse(object):
    def __init__(self, tagged_sentence):
        # pad the sentence with a dummy __START__ and a __ROOT__ node
        self._tagged = [('__START__', '__START__')] + tagged_sentence + [('__ROOT__', '__ROOT__')]

        # parent list and labels
        self._heads = [None] * len(self)
        self._labels = [None] * len(self)

        # keep a list of dependents and the labels
        self._deps = [[] for _ in range(len(self))]

        # keep a list of left dependants and the labels
        self._left_deps = [[] for _ in range(len(self))]

        # keep a list of the right dependants and the labels
        self._right_deps = [[] for _ in range(len(self))]


    def add_edge(self, head, child, label=None):
        """ Add labelled edge between 2 nodes in the dependency graph """
        self._heads[child] = head
        self._labels[child] = label

        # keep track of all dependents of a node
        self._deps[head].append(child)

        # keep track of the left/rights dependents
        if child < head:
            self._left_deps[head].append(child)
        else:
            self._right_deps[head].append(child)

    def words(self):
        return [tt[0] for tt in self._tagged]

    def tags(self):
        return [tt[1] for tt in self._tagged]

    def tagged_words(self):
        return self._tagged

    def heads(self):
        return self._heads

    def lefts(self):
        return self._left_deps

    def rights(self):
        return self._right_deps

    def deps(self):
        return self._deps

    def labels(self):
        return self._labels

    def __getitem__(self, item):
        return {
```

```
            'word': self._tagged[item][0],
            'pos': self._tagged[item][1],
            'head': self._heads[item],
            'label': self._labels[item],
            'children': list(zip(self._deps[item], self._deps_labels[item])),
            'left_children': self._left_deps[item],
            'right_children': self._right_deps[item],
        }

    def __len__(self):
        return len(self._tagged)

    def __str__(self):
        return str(self._heads)

    def __repr__(self):
        return str(self._heads)
```

We store extra information that might seem redundant now(e.g: dependents list), but it will prove useful for future work and for doing feature extraction.

You probably noticed that in the `iter_ud_dep_data` we don't actually yield parses, but list of quadruplets. We need parses because that's what our parser will work with. Let's use the `iter_ud_dep_data` function to read the raw data and build the parses. This is done in the `iter_ud_dependencies` function:

iter_ud_dependencies function

```
def iter_ud_dependencies(filename):
    for annotated_sentence in iter_ud_dep_data(filename):
        words = []
        tags = []

        heads = [None]
        labels = [None]
        for i, (word, pos, head, label) in enumerate(annotated_sentence):
            # Skip some fictive nodes that don't belong in the graph
            if head == '_':
                continue

            words.append(word)
            tags.append(pos)

            if head == '0':
                # Point to the dummy node
                heads.append(len(annotated_sentence) + 1)
            else:
                heads.append(int(head))

            labels.append(label)

        dep_parse = DependencyParse(list(zip(words, tags)))
        for child, head in enumerate(heads):
            if head is not None:
                dep_parse.add_edge(head, child, labels[child])

        yield dep_parse
```

We now know what a `DependencyParse` object looks like and we know how to read it from the data files. Let's start talking about the `ParserState` we mentioned before.

Our system works with 3 types of transitions: `SHIFT`, `LEFT_ARC` and `RIGHT_ARC`. The `ParserState` class this is a very important component. Here are its responsibilities: * It knows how to apply a transition to a parse * It is able to tell us what are the next valid transitions from a given state * Given the current parse and the gold parse (the annotated parse from the dataset) it is able to tell us what are the next correct transitions towards the gold parse.

Let's talk about the last 2 points.

The valid transitions issue is pretty straight-forward. It makes sure we have enough nodes in the stack/buffer to perform the operation. For example, we can't apply `SHIFT` to a state that has an empty buffer.

The next transitions towards the gold parse part is a bit more subtle. It makes sure that out of the valid transitions, we only choose the ones that take us towards the gold parse. To make sure we're on the right road towards the *"gold path"*, we need to make sure that by applying a transition we don't lose any dependencies. There are of course cases when if we make a certain transition, we can't go back and add a certain dependency. For example, if a node is moved via a SHIFT transition, that node can't be a parent of any node present in the current stack. Knowing the gold parse, we can avoid such situations.

ParseState class

```python
class Transitions:
        SHIFT = 0
        LEFT_ARC = 1
        RIGHT_ARC = 2

        ALL = (SHIFT, LEFT_ARC, RIGHT_ARC)


class ParserState(object):

    def __init__(self, parse):
        self.parse = parse

        # Put the first word of the parse in the stack
        self.stack = [1]

        # The buffer index is pointing to the second word
        self.buffer_index = 2

    def apply(self, transition, label=None):
        if transition == Transitions.SHIFT:
            self.stack.append(self.buffer_index)
            self.buffer_index += 1

            # No new edge was created
            return None, None
```

```
        # Add new edge
        if transition == Transitions.RIGHT_ARC:
            # New edge from `stack[-2]` to `stack[-1]`
            head, child = self.stack[-2], self.stack.pop()

        # transition == Transitions.LEFT_ARC
        else:
            # New edge from `buffer_index` to `stack[-1]`
            head, child = self.buffer_index, self.stack.pop()

        self.parse.add_edge(head, child, label)
        return head, child

    def next_valid(self):
        """ The list of transitions that are allowed in the current state """
        valid = []

        # We can do a SHIFT if the buffer is not empty
        if (self.buffer_index + 1) < len(self.parse):
            valid.append(Transitions.SHIFT)

        # We can do a RIGHT_ARC if there are at least 2 words in the stack
        if len(self.stack) >= 2:
            valid.append(Transitions.RIGHT_ARC)

        # We can do a LEFT_ARC if there is at least one node in the stack
        if len(self.stack) >= 1:
            valid.append(Transitions.LEFT_ARC)

        return valid

    def next_gold(self, gold_parse):
        """ What transitions would you choose in this state if you knew the final result """
        gold_heads = gold_parse.heads()
        buffer_nodes = range(self.buffer_index + 1, len(self.parse) - 1)

        # Let's start from the set of valid transitions
        valid = set(self.next_valid())

        def has_dependency(word1, word2):
            """  check if there's any dependency between the 2 words """
            return gold_heads[word1] == word2 or gold_heads[word2] == word1

        if not self.stack or (
                    Transitions.SHIFT in valid and
                    gold_heads[self.buffer_index] == self.stack[-1]):
            return [Transitions.SHIFT]

        # If there's an edge between the node on top of the
        # stack and the current index in buffer, we have no choice but to
        # draw that edge, otherwise we'll loose it
        if gold_heads[self.stack[-1]] == self.buffer_index:
            return [Transitions.LEFT_ARC]

        # If stack[-2] is the parent of stack[-1] then drawing an edge from
        # stack[-1] to stack[-2] is incorrect (wrong direction)
        if len(self.stack) >= 2 and gold_heads[self.stack[-1]] == self.stack[-2]:
            valid.discard(Transitions.LEFT_ARC)

        # If there's any dependency between the current item in the buffer and
        # a node in the stack, moving the item in the stack would loose the dependency
```

```
        if any([has_dependency(self.buffer_index, w) for w in self.stack]):
            valid.discard(Transitions.SHIFT)

        # If there's any dependency between stack[-1] and a node in the buffer,
        # popping the stack would loose that dependency
        if any([has_dependency(self.stack[-1], w) for w in buffer_nodes]):
            valid.discard(Transitions.LEFT_ARC)
            valid.discard(Transitions.RIGHT_ARC)

        return list(valid)
```

Let's now build the our dependency parser. We'll be taking the same approach we did previously of training the model in batches and using the `FeatureHasher` vectorizer for incrementally building the feature space. There's one difference: we'll going to need a model that can be trained in batches as well as being capable of giving us the prediction probabilities (should implement the `predict_proba` method). `Scikit-Learn` doesn't offer many choices for this type of model. One of them, that matches our requirements is `SGDClassifier` (Stochastic Gradient Descent Classifier) that uses the `modified_huber` loss function. We're not going to go into the details of how this classifier or its loss function works, but I encourage you to look it up in the `sklearn` documentation. In case you are wondering why we need the probabilities for the predicted classes, here's why: we will use the classifier to get the best next transition. Since the classifier can predict any class, but only some can be valid, we need to know what's the most probable one out of the valid options.

Feature extraction is also a very important component of our system. In this case, the feature extraction method will be applied on a `ParserState` instance. We'll be exploring its implementation later on.

Writing the Dependency Parser Class

Like the chunker and NER we've built earlier, the parser also deals with tagged sentences.

Parser class

```
class Parser(ParserI):

    @staticmethod
    def build_transition_dataset(parses, feature_extractor):
        """ Transform a list of parses to a dataset """
        transitions_X, transitions_y = [], []
        for gold_parse in parses:
            # Init an empty parse
            dep_parse = DependencyParse(gold_parse.tagged_words()[1:-1])

            # Start from an empty state
            state = ParserState(dep_parse)
```

```
        while state.stack or (state.buffer_index + 1) < len(dep_parse):
            features = feature_extractor(state)
            gold_moves = state.next_gold(gold_parse)

            if not gold_moves:
                # Something is wrong here ...
                break

            # Pick one of the possible transitions
            t = random.choice(gold_moves)

            # Append the features and transition to the dataset
            transitions_X.append(features)
            transitions_y.append(t)

            # Apply the transition to the state
            state.apply(t)

    return transitions_X, transitions_y

def __init__(self, feature_detector):
    self.feature_extractor = feature_detector
    self._vectorizer = FeatureHasher()
    self._model = SGDClassifier(loss='modified_huber')

def evaluate(self, parses):
    correct, total = 0, 0

    for parse in parses:
        predicted_parse = self.parse(parse.tagged_words()[1:-1])
        heads, predicted_heads = np.array(parse.heads()[1:]), np.array(predicted_parse.heads()[1:])
        total += len(heads)
        correct += np.sum(heads == predicted_heads)
    return correct / total

def parse(self, sent, *args, **kwargs):
    """ Parse a tagged sentence """
    state = ParserState(DependencyParse(sent))

    while state.stack or (state.buffer_index + 1) < len(state.parse):
        # Extract the features of the current state
        features = self.feature_extractor(state)
        vectorized_features = self._vectorizer.transform([features])

        # Get probabilities for the next transitions
        predictions = self._model.predict_proba(vectorized_features)[0]
        scores = dict(zip(list(self._model.classes_), list(predictions)))

        # Check what moves are actually valid
        valid_moves = state.next_valid()

        # Get the most probable valid mode
        guess = max(valid_moves, key=lambda move: scores[move])

        # apply the transition to the state
        state.apply(guess)

    return state.parse

def train(self, corpus_iterator, n_iter=5, batch_size=100):
```

```
        """ Train a model on a given corpus """
        for _ in range(n_iter):
            # Fork the iterator
            corpus_iterator, parses = itertools.tee(corpus_iterator)
            batch_count = 0

            while True:
                batch_count += 1
                print("Training on batch={0}".format(batch_count))
                batch = list(itertools.islice(parses, batch_size))

                # No more batches
                if not batch:
                    break

                # Train the model on a batch
                self.train_batch(batch)

    def train_batch(self, gold_parses):
        """ Train the model on a single batch """
        t_X, t_Y = self.build_transition_dataset(
            gold_parses, self.feature_extractor)

        self._model.partial_fit(self._vectorizer.transform(t_X), t_Y,
                                classes=Transitions.ALL)
```

In my opinion, after building the `ParserState` class and after understanding how the transition system works, the parser implementation is pretty simple. The missing piece is the feature extraction function. The code is pretty ugly and boring. Nevertheless, we must talk about it. The features are obviously more convoluted than what we're used to from the previous models we've built. As we previously mentioned, the feature extraction function is applied on the current state of the parser. It is a mix of words and tags, from the stack or from the buffer, from the dependents (left or right), by themselves or combined into bigrams and trigrams.

Here's a useful function for padding a list to the right:

pad_right function

```
def pad_right(l, n, padding=None):
    if n <= len(l):
        return l
    return l + ([padding] * (n - len(l)))
```

Let's write the feature detection part:

dependency_features function

```
def dependency_features(state):
    """ Extract features from a parser state """
    length = len(state.parse)
    words = state.parse.words()
    tags = state.parse.tags()

    def stack_features(data, width=3):
        context = [data[idx] for idx in reversed(state.stack[-width:])]
        return tuple(pad_right(context, width, '__NONE__'))

    def buffer_features(data, width=3):
        context = [data[idx] for idx in
                   range(state.buffer_index, min(state.buffer_index + width, length))]
        return tuple(pad_right(context, width, '__NONE__'))

    def parse_features(dependencies, data, width=2):
        context = [data[idx] for idx in reversed(dependencies[-width:])]
        return tuple(pad_right(context, width, '__NONE__'))

    f = {}

    # Top node in stack
    top_s = state.stack[-1] if len(state.stack) else -1

    # stack - words/tags
    f['w-s0'], f['w-s1'], f['w-s2'] = stack_features(words)
    f['t-s0'], f['t-s1'], f['t-s2'] = stack_features(tags)

    # buffer - words/tags
    f['w-b0'], f['w-b1'], f['w-b2'] = buffer_features(words)
    f['t-b0'], f['t-b1'], f['t-b2'] = buffer_features(tags)

    # ============================================

    # left children - buffer - words
    f['w-lp-b0'], f['w-lp-b1'] = parse_features(
        state.parse.lefts()[state.buffer_index], words)

    # left children - buffer - tags
    f['t-lp-b0'], f['t-lp-b1'] = parse_features(
        state.parse.lefts()[state.buffer_index], tags)

    # left children - buffer - count
    f['#lp-b'] = len(state.parse.lefts()[state.buffer_index])

    # ============================================

    # right children - buffer - words
    f['w-rp-b0'], f['w-rp-b1'] = parse_features(
        state.parse.rights()[state.buffer_index], words)

    # right children - buffer - tags
    f['t-rp-b0'], f['t-rp-b1'] = parse_features(
        state.parse.rights()[state.buffer_index], tags)

    # right children - buffer - count
    f['#rp-b'] = len(state.parse.rights()[state.buffer_index])

    # ============================================
```

```
# left children - stack - words
f['w-lp-s0'], f['w-lp-s1'] = parse_features(
    state.parse.lefts()[top_s], words)

# left children - stack - tags
f['t-lp-s0'], f['t-lp-s1'] = parse_features(
    state.parse.lefts()[top_s], tags)

# left children - stack - count
f['#lp-s'] = len(state.parse.lefts()[top_s])

# ==============================================

# right children - stack - words
f['w-rp-s0'], f['w-rp-s1'] = parse_features(
    state.parse.rights()[top_s], words)

# right children - stack - tags
f['t-rp-s0'], f['t-rp-s1'] = parse_features(
    state.parse.rights()[top_s], tags)

# right children - stack - count
f['#rp-s'] = len(state.parse.rights()[top_s])

# ==============================================

# distance between the top node in the stack and the current node in the buffer
f['dist-b-s'] = state.buffer_index - top_s

# ==============================================

# Word/Tag pairs
f['w-s0$t-s0'] = f['w-s0'] + '$' + f['t-s0']
f['w-s1$t-s1'] = f['w-s1'] + '$' + f['t-s1']
f['w-s2$t-s2'] = f['w-s2'] + '$' + f['t-s2']

f['w-b0$t-b0'] = f['w-b0'] + '$' + f['t-b0']
f['w-b1$t-b1'] = f['w-b1'] + '$' + f['t-b1']
f['w-b2$t-b2'] = f['w-b2'] + '$' + f['t-b2']

# ==============================================

# Bigrams
f['w-s0$w-b0'] = f['w-s0'] + '$' + f['w-b0']
f['t-s0$t-b0'] = f['t-s0'] + '$' + f['t-b0']

f['w-s1$w-b1'] = f['w-s1'] + '$' + f['w-b1']
f['t-s1$t-b1'] = f['t-s1'] + '$' + f['t-b1']

f['w-s2$w-b2'] = f['w-s2'] + '$' + f['w-b2']
f['t-s2$t-b2'] = f['t-s2'] + '$' + f['t-b2']

f['w-s0$w-s1'] = f['w-s0'] + '$' + f['w-s1']
f['w-s1$w-s2'] = f['w-s1'] + '$' + f['w-s2']

f['t-s0$t-s1'] = f['t-s0'] + '$' + f['t-s1']
f['t-s1$t-s2'] = f['t-s1'] + '$' + f['t-s2']

f['w-b0$w-b1'] = f['w-b0'] + '$' + f['w-b1']
f['w-b1$w-b2'] = f['w-b1'] + '$' + f['w-b2']
```

```
    f['t-b0$t-b1'] = f['t-b0'] + '$' + f['t-b1']
    f['t-b1$t-b2'] = f['t-b1'] + '$' + f['t-b2']

    # =============================================

    # Trigrams
    f['w-s0$w-s1$w-s2'] = f['w-s0$w-s1'] + '$' + f['w-s2']
    f['t-s0$t-s1$t-s2'] = f['t-s0$t-s1'] + '$' + f['t-s2']

    f['w-b0$w-b1$w-b2'] = f['w-b0$w-b1'] + '$' + f['w-b2']
    f['t-b0$t-b1$t-b2'] = f['t-b0$t-b1'] + '$' + f['t-b2']

    f['w-s0$w-lp-s0$w-lp-s1'] = f['w-s0'] + '$' + f['w-lp-s0'] + '$' + f['w-lp-s1']
    f['t-s0$t-lp-s0$t-lp-s1'] = f['t-s0'] + '$' + f['t-lp-s0'] + '$' + f['t-lp-s1']

    f['w-b0$w-lp-b0$w-lp-b1'] = f['w-b0'] + '$' + f['w-lp-b0'] + '$' + f['w-lp-b1']
    f['t-b0$t-lp-b0$t-lp-b1'] = f['t-b0'] + '$' + f['t-lp-b0'] + '$' + f['t-lp-b1']

    return f
```

Now for the final steps, the training and evaluation:

train dependency parser

```
def main():
    train_data = list(iter_ud_dependencies('../../../data/en-ud-train.conllu'))
    test_data = list(iter_ud_dependencies('../../../data/en-ud-dev.conllu'))

    parser = Parser(dependency_features)
    parser.train(train_data, n_iter=5, batch_size=200)

    print("Accuracy: ", parser.evaluate(test_data[:500]))

    p = parser.parse(pos_tag(word_tokenize("I buy green apples")))
    print(p.heads())
```

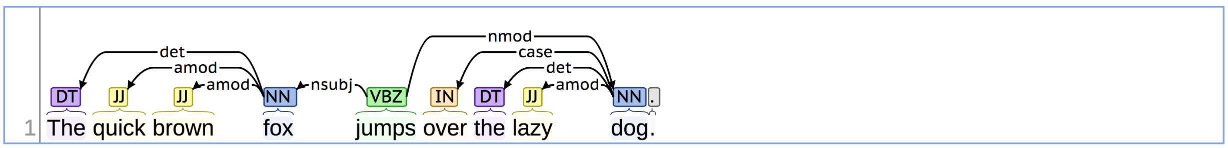
The reason why we're only evaluating on the first 500 sentences from the test set is because our parser isn't particularly fast. It trains quickly, because we take advantage of the vectorized operations, but it is very slow when actually parsing. That's because we need to call the classifier for each and every word because the result is needed for the next word. This is not ideal and there definitely are several solutions for this. The purpose of this solution is not to build the most accurate and efficient parser, but rather to understand the algorithm that drives it.

We've got close to `80%` accuracy:

```
Accuracy:  0.799307530605

# result of parsing: "I buy green apples"
[None, 2, 5, 4, 2, None]
```

Adding Labels to the Parser

In the previous chapter we built one of the most useful tools in modern Natural Language Processing: a dependency parser. There's one more thing to add to it to make it really cool: edge labels. The role of the edges is to describe the type of relationship between the two words: head and dependant. One of the most common labels is `nsubj` and it describes the relationship between a verb and the noun that *is doing the action*. Recall this example?



Stanford Dependency Parser

Here's the complete list of dependency labels and their description: UD labels[46]

Let's create a list with these labels, that we will later use to train our labeler:

List of labels

```
DEPENDENCY_LABELS = [
    "acl",
    "acl:relcl",
    "advcl",
    "advmod",
    "amod",
    "appos",
    "aux",
    "aux:pass",
    "case",
    "cc",
    "cc:preconj",
    "ccomp",
    "compound",
    "compound:prt",
    "conj",
    "cop",
    "csubj",
    "csubj:pass",
    "dep",
    "det",
    "det:predet",
    "discourse",
    "dislocated",
    "expl",
```

[46] http://universaldependencies.org/u/dep/

```
    "fixed",
    "flat",
    "flat:foreign",
    "goeswith",
    "iobj",
    "list",
    "mark",
    "nmod",
    "nmod:npmod",
    "nmod:poss",
    "nmod:tmod",
    "nsubj",
    "nsubj:pass",
    "nummod",
    "obj",
    "obl",
    "obl:npmod",
    "obl:tmod",
    "orphan",
    "parataxis",
    "punct",
    "reparandum",
    "root",
    "vocative",
    "xcomp"
]
```

We'll use most of the code we've written in previous section and create one more model that given an already parsed sentence, adds the labels to the edges. Our new parser will wrap 2 models rather than one. First, we will need to add static method to the `Parser` class that transforms a list of labeled parses into a dataset. This will be fed to our labeler model.

build_labels_dataset static method

```
class Parser(ParserI):

    @staticmethod
    def build_labels_dataset(parses, feature_extractor):
        """ Transform a list of parses to a labels dataset """
        labels_X, labels_y = [], []
        for gold_parse in parses:
            for child, head in enumerate(gold_parse.heads()[1:-1]):
                features = feature_extractor(gold_parse, head, child + 1)

                label = gold_parse.labels()[child + 1]
                labels_X.append(features)
                labels_y.append(label)

        return labels_X, labels_y
```

In the `Parser` constructor method, we're going to initialize another model, responsible for assigning labels:

Parser init

```
def __init__(self, feature_detector, label_feature_detector):
    self.feature_extractor = feature_detector
    self.label_feature_detector = label_feature_detector

    self._vectorizer = FeatureHasher()
    self._model = SGDClassifier(loss='modified_huber')

    self._label_vectorizer = FeatureHasher()
    self._label_model = Perceptron()
```

Learning to Label Dependencies

We need to change the `train_batch` method in order to be able to train the labeler model while training the transition model.

train labeler in train_batch

```
def train_batch(self, gold_parses):
    """ Train the model on a single batch """
    t_X, t_Y = self.build_transition_dataset(
        gold_parses, self.feature_extractor)

    self._model.partial_fit(self._vectorizer.transform(t_X), t_Y,
                            classes=Transitions.ALL)

    l_X, l_Y = self.build_labels_dataset(
        gold_parses, self.label_feature_detector)

    self._label_model.partial_fit(self._label_vectorizer.transform(l_X), l_Y,
                                  classes=DEPENDENCY_LABELS)
```

As planned, we are now going to write a method that given a parsed sentences, adds the labels. We'll call it `label_parse`.

label_parse method

```
def label_parse(self, parse):
    """ Add labels to a dependency parse """
    label_features = []
    for child, head in enumerate(parse.heads()[1:-1]):
        features = self.label_feature_detector(parse, head, child + 1)
        label_features.append(features)

    vectorized_label_features = self._label_vectorizer.transform(label_features)
    predicted_labels = self._label_model.predict(vectorized_label_features)
    parse._labels = [None] + list(predicted_labels) + [None]

    return parse
```

We need to make a small change to the `parse` method, to add the labels just before returning the computed parse:

changes to the parse method

```
def parse(self, sent, *args, **kwargs):
    """ Parse a tagged sentence """
    state = ParserState(DependencyParse(sent))
    while state.stack or (state.buffer_index + 1) < len(state.parse):
        # Extract the features of the current state
        features = self.feature_extractor(state)
        vectorized_features = self._vectorizer.transform([features])

        # Get probabilities for the next transitions
        predictions = self._model.predict_proba(vectorized_features)[0]
        scores = dict(zip(list(self._model.classes_), list(predictions)))

        # Check what moves are actually valid
        valid_moves = state.next_valid()

        # Get the most probable valid mode
        guess = max(valid_moves, key=lambda move: scores[move])

        # apply the transition to the state
        state.apply(guess)

    self.label_parse(state.parse)  # Add labels too ...

    return state.parse
```

We're almost done. Let's tweak the `evaluate` method such that it returns the accuracy of the labeler as well. Here's how:

adapt the evaluate method

```
def evaluate(self, parses):
    correct_heads, correct_labels, total = 0, 0, 0

    for parse in parses:
        predicted_parse = self.parse(parse.tagged_words()[1:-1])

        heads = np.array(parse.heads()[1:-1])
        predicted_heads = np.array(predicted_parse.heads()[1:-1])

        labels = np.array(parse.labels()[1:-1])

        # Relabel the gold parse with what our model would label
        self.label_parse(parse)
        predicted_labels = np.array(parse.labels()[1:-1])

        total += len(heads)
        correct_heads += np.sum(heads == predicted_heads)
        correct_labels += np.sum(labels == predicted_labels)

    return correct_heads / total, correct_labels / total
```

Training our Labelled Dependency Parser

Let's run our training procedure on the labelling parser:

training our labelling parser

```
def main():
    train_data = list(iter_ud_dependencies('../../../data/en-ud-train.conllu'))
    test_data = list(iter_ud_dependencies('../../../data/en-ud-dev.conllu'))

    parser = Parser(dependency_features, edge_label_features)
    parser.train(train_data, n_iter=5, batch_size=200)

    print("Accuracy: ", parser.evaluate(test_data[:250]))

    p = parser.parse(pos_tag(word_tokenize("I buy green apples")))
    print(p.heads())
    print(p.labels())
```

Here are the results:

```
Accuracy:  (0.773558648111332O1, 0.921073558648111136)

# parser.parse(pos_tag(word_tokenize("I buy green apples")))
[None, 2, 5, 4, 2, None]  # heads
[None, 'nsubj', 'root', 'amod', 'obj', None]  # labels
```

Part 5: Build Your Own Chatbot Engine

Chatbots are hot subject right now and for good reason. Almost everybody uses a chat service: Facebook Messenger, Whatsapp, Skype, Telegram, Slack, WeChat, and that's only naming a few. Implementing an in-house chatbot enables companies to build apps on top of an UX that is very familiar to any user.

Great progress has been done in this area and there are a lot of services providing infrastructure for chatbots. The complexity and maturity of the conversational NLP engines varies from one platform to another, but there's a list of the most popular ones at the time of writing:

- ChatFuel[47]
- Microsoft LUIS.ai[48] - Language Understanding Intelligent Service
- Facebook Wit.ai[49] - It can now be used directly from the Messenger Platform[50]
- Google DialogFlow[51] - Previously known as *api.ai*
- Amazon Lex[52]

There are considerably more, but I consider these to be the most popular. The landscape is also constantly evolving so be sure to do some research beforehand if building chatbots is your thing.

In this chapter, we'll be building a toy conversational NLP engine, similar to DialogFlow. Obviously the functionality will be only a teeny-tiny fraction of what DialogFlow has to offer but we're here to understand the underlying principles.

[47] https://chatfuel.com/
[48] https://www.luis.ai/
[49] https://wit.ai/
[50] https://messenger.fb.com/
[51] https://dialogflow.com/
[52] https://aws.amazon.com/lex/

General Architecture

There are many trends and architectures when it comes to building chatbots. Here are a few challenges that one might face:

- Human dialogue is extremely complex. There can be so many conversational threads that are interlaced and only by having domain knowledge can one untangle the threads and can understand what ties into what.
- When it comes to social media and chat apps people use a specific language. It contains a lot of emoticons, a lot of abbreviations, some of them more standardised than others (lol, rotfl, omg, brb, gg), a *loooot* of misspelled words (on purpose or because of typing on phone keyboards).
- Getting the context right is difficult. It is sometimes hard to understand which entity is referring to what: "Matt is with Joe. He is my best friend.", "I want to buy two tickets from Seattle to London and another one to San Francisco".
- Internal memory. How much should a chatbot keep track of?
- Learning from conversations: this is a very tricky one. Remember the whole Tay.ai fiasco[53]?
- Small talk: Should the chatbot be able to entertain the user as a normal person whould do?
- Getting the right information: asking for information from users is tough. After a human's input many followup questions are possible, depending on the business logic. Are we looking of a general solution to this or rather custom implementations that handle some scenarios better?

All of the above are open questions.

This chapter is more of a recap of what's been done and learnt till now. We'll be putting stuff together and will assemble everything in a chatbot platform. Let's state some of the desired features we want for our platform.

Train the Platform via Examples

This is the most important feature. We want to provide a list of examples and the chatbot will learn to adapt to variations of those examples. Given the example: "I want to buy tickets for New York", we decide that the `action` needed is `buy_tickets`

[53] https://www.theverge.com/2016/3/24/11297050/tay-microsoft-chatbot-racist

with parameters: `{"location": "New York"}`. The chatbot should figure out if given the sentence: "I need tickets for Seattle" that the action is `buy_tickets` with parameters `{"location": "Seattle"}`.

Recognize when the user is rude and do basic small talk

Our bot should be able to reply with something like *"That's not a nice thing to say"* when foul language is used. We should also be able to respond to greetings: *Bye*, *See you later* and other basic stuff like that.

Action Handlers

After our platform detects that we need to `buy_tickets`, we need to be able to perform some API calls so that we actually buy the tickets and send them to the user.

I'm going to name this chatbot architecture **The Intent/Entities** architecture. This is something I just made up so don't take it too seriously. It might have another official name or it may not. It is however essentially the same architecture used by DialogFlow.

Here's how this architecture works:

- Every input the chatbot receives is associated to one of the predefined intents the chatbot knows how to handle. By intent we mean exactly what the dictionary says it means: *"intention or purpose. synonyms: aim, purpose, intention, objective, object, goal, target, end;*
- We want the handlers of the intents to be parameterizable. Going back to the example bot that sells tickets, we want to tell the bot for which event we want the tickets for.

I'm going to make a simple and practical analogy to URLs and REST verbs. Let's consider the following HTTP request:

```
POST /tickets/buy?event_id=11145&quantity=2
```

This request would translate in chatbot universe to:

```
I want to purchase two tickets for the Batman movie.
```

After applying the intent detection and entity extraction, this would look like this:

```
I want to purchase [quantity]two[/quantity] tickets for the [event]Batman movie[/event].
INTENT='buy_tickets'
```

If we would delete the noise, we would only be left with something like this:

```
('buy_tickets', {'quantity': 'two', 'event': "Batman movie"})
```

The last form looks like something we can process easily.

Building the Core

In this chapter we're building the underlying framework of our chatbot. We'll not be concerned with the particularities of building a certain chatbot, but a platform we can use to train any chatbot we want.

Here's how we'll be approaching this: * We'll create a training set for the very general capabilities every chatbot should have: recognize greetings, foul language, etc. * We'll build a particular dataset for a specific chatbot we want to build. In our case, the chatbot we want to build is capable of **answering questions about movies**.

Chatbot Base Class and Training Set

Next, we'll implement a `Chatbot` base class that takes the datasets and trains the classifiers for identifying intents and entities. In order to make the chatbot answer with actual information to user queries we need to register handlers for intents. This will be tackled in the next chapter.

Let's start by building the training set. I've organized the training set in 2 parts: some general intents and some movie-specific intents. Here's the dataset I've written:

Intent/Entities dataset

```
MOVIE_EXPERT_TRAINING_SET = {
    # General Intents
    'greetings': [
        ("Hello", {}),
        ("Hello!", {}),
        ("Hi!", {}),
        ("How are you?", {}),
        ("Hi There!", {}),
        ("Hello there!!!", {}),
        ("Hi!", {}),
        ("Ello!", {}),
        ("Hey!", {}),
        ("Hello mate, how are you?", {}),
        ("Good morning", {}),
        ("Good morning!", {}),
        ("mornin'", {}),
    ],
    'bye': [
        ("Bye!", {}),
        ("Bye Bye!", {}),
        ("ByeBye!", {}),
```

```
        ("Bbye!", {}),
        ("See you!", {}),
        ("See you later", {}),
        ("Good bye", {}),
        ("See you soon", {}),
        ("Talk to you later!", {}),
    ],
    'thanks': [
        ("Thanks dude!", {}),
        ("Thank you!", {}),
        ("Thanks", {}),
        ("10x", {}),
        ("10q", {}),
        ("awesome, thanks for your help", {}),
        ("Thank you so much!", {}),
        ("Cheers!", {}),
        ("Cheers, thanks", {}),
        ("many thanks", {}),
    ],
    'rude': [
        ("You're an idiot", {}),
        ("You're such a fucking idiot!", {}),
        ("You're so stupid !!!", {}),
        ("You stupid fuck !!!", {}),
        ("Shut up!", {}),
        ("Shut the fuck up you fuckhead!", {}),
        ("You're such a mother fucker!", {}),
        ("shit!!", {}),
        ("fuck!!", {}),
        ("fuck you!!", {}),
        ("haha, you stupid !", {}),
    ],
    'chit-chat': [
        ("How are you?", {}),
        ("Hey, how are you?", {}),
        ("How's life?", {}),
        ("How you've been?", {}),
        ("How's your day?", {}),
        ("How's life?", {}),
        ("How's business?", {}),
    ],
    'ask-info': [
        ("Tell me more about yourself", {}),
        ("Who are you?", {}),
        ("Can you provide more info?", {}),
        ("Can I get some information?", {}),
        ("Give me more info", {}),
        ("Give me some info please", {}),
        ("I need some info", {}),
        ("I would like to know more please", {}),
        ("Tell me about you", {}),
        ("Let's talk about you", {}),
        ("Who are you?", {}),
    ],

    # Movie specific intents
    'what-director': [
        ("Who directed The Godfather?",
         {'movie': 'The Godfather'}),
        ("Who is the director of The Dark Knight?",
         {'movie': 'The Dark Knight'}),
```

```
        ("Who directed Pulp Fiction?",
         {'movie': 'Pulp Fiction'}),
        ("Who is the director of Lord of the rings?",
         {'movie': 'Lord of the rings'}),
        ("The director of Fight Club is?",
         {'movie': 'Fight Club'}),
        ("director of Forrest Gump is?",
         {'movie': 'Forrest Gump'}),
        ("Who Directed The Matrix?",
         {'movie': 'The Matrix'}),
        ("Who is the director of The Silence of the Lambs?",
         {'movie': 'The Silence of the Lambs'}),
    ],
    'what-release-year': [
        ("In what year was Saving Private Ryan released?",
         {'movie': 'Saving Private Ryan'}),
        ("In what year was Interstellar released?",
         {'movie': 'Interstellar'}),
        ("When was Psycho released?",
         {'movie': 'Psycho'}),
        ("When was Casablanca released?",
         {'movie': 'Casablanca'}),
        ("When was Raiders of the Lost Ark produced?",
         {'movie': 'Raiders of the Lost Ark'}),
        ("What's the release year for The Pianist?",
         {'movie': 'The Pianist'}),
        ("What's the release year for The Matrix?",
         {'movie': 'The Matrix'}),
    ],
    'get-similar-movies': [
        ("What are some similar movies to Interstellar",
         {'movie': 'Interstellar'}),
        ("What are some similar movies to Rear Window ?",
         {'movie': 'Rear Window'}),
        ("Can I get some similar movies to Back to the Future ?",
         {'movie': 'Back to the Future'}),
        ("I want more movies like Back to the Future ?",
         {'movie': 'Back to the Future'}),
        ("I want more movies like Django Unchained ?",
         {'movie': 'Django Unchained'}),
        ("I want more movies similar to Blade Runner ?",
         {'movie': 'Blade Runner'}),
        ("I want to know more movies similar to American Beauty ?",
         {'movie': 'American Beauty'}),
        ("Get me more movies like Citizen Kane ?",
         {'movie': 'Citizen Kane'}),
        ("What are some movies similar to Citizen Kane ?",
         {'movie': 'Citizen Kane'}),
    ],
    'get-latest': [
        ("What are some new movies?", {}),
        ("What are some new released movies?", {}),
        ("What are some new releases?", {}),
        ("Can I get some new movies?", {}),
        ("What's new?", {}),
        ("What are some new and cool movies?", {}),
        ("Tell me about some new movies?", {}),
        ("Tell me about new releases?", {}),
    ],
    'actor-movies': [
        ('In what movies did Ryan Gosling acted?',
```

```
     {'actor': 'Ryan Gosling'}),
    ('in which movies did Selena Gomez play?',
     {'actor': 'Selena Gomez'}),
    ('In which films did Jason Momoa play?',
     {'actor': 'Jason Momoa'}),
    ('What are some movies in which Emilia Clarke acted ?',
     {'actor': 'Emilia Clarke'}),
    ('What are some movies which featured Johnny Depp ?',
     {'actor': 'Johnny Depp'}),
    ('Tell me some movies with Johnny Depp ?',
     {'actor': 'Johnny Depp'}),
    ('Tell me some movies featuring Tom Cruise ?',
     {'actor': 'Tom Cruise'}),
    ('Get me some movies featuring Leonardo DiCaprio ?',
     {'actor': 'Leonardo DiCaprio'}),
    ('Some films with Matt Damon ?',
     {'actor': 'Matt Damon'}),
    ('Some cool films with Keanu Reeves ?',
     {'actor': 'Keanu Reeves'}),
],
'movie-cast': [
    ("What's the cast of Back to the Future ?",
     {'movie': 'Back to the Future'}),
    ("What's the cast of Forrest Gump?",
     {'movie': 'Forrest Gump'}),
    ("Who acted in The Matrix?",
     {'movie': 'The Matrix'}),
    ("Who acted in American Beauty ?",
     {'movie': 'The Matrix'}),
    ("Who played in Django Unchained?",
     {'movie': 'Django Unchained'}),
    ("What are the actors playing in Pulp Fiction?",
     {'movie': 'Pulp Fiction'}),
    ("Give me the cast of Pulp Fiction?",
     {'movie': 'Pulp Fiction'}),
    ("Tell me the full cast of The Silence of the Lambs?",
     {'movie': 'The Silence of the Lambs'}),
    ("Who was casted in Citizen Kane ?",
     {'movie': 'Citizen Kane'}),
    ("Who was casted in Rear Window ?",
     {'movie': 'Rear Window'}),
],
'actor-in-movie': [
    ("Did Morgan Freeman play in The Shawshank Redemption movie?",
     {'actor': 'Morgan Freeman', 'movie': 'The Shawshank Redemption'}),
    ("Did Marlon Brando act in The Godfather?",
     {'actor': 'Marlon Brando', 'movie': 'The Godfather'}),
    ("Was Al Pacino an actor in The Godfather?",
     {'actor': 'Al Pacino', 'movie': 'The Godfather'}),
    ("Is Uma Thurman the actress in Pulp Fiction?",
     {'actor': 'Uma Thurman', 'movie': 'Pulp Fiction'}),
    ("Was Brad Pitt the actor in Fight Club?",
     {'actor': 'Brad Pitt', 'movie': 'Fight Club'}),
    ("Was Ingrid Bergman in the movie Casablanca?",
     {'actor': 'Ingrid Bergman', 'movie': 'Casablanca'}),
    ("Did Russell Crowe star in Gladiator?",
     {'actor': 'Russell Crowe', 'movie': 'Gladiator'}),
    ("Was Sigourney Weaver a star in the Alien movie?",
     {'actor': 'Sigourney Weaver', 'movie': 'Alien'}),
    ("Did Kevin Spacey act in American Beauty?",
     {'actor': 'Kevin Spacey', 'movie': 'American Beauty'}),
```

```
    ]
}
```

As you can notice, we have several intents and two entity types: **movies** and **actors**. The goal is to create an intent classifier (which is nothing more than a text classifier like the ones we've build in chapters 2 and 3) and an entity extractor, like the one we've built in chapter 4.

Training the Chatbot

Let's write a simple intent classifier training routine:

Train intent classifier procedure

```
tokenizer = TweetTokenizer()


def train_intent_model(train_set):
    pipeline = Pipeline(steps=[
        ('vectorizer', CountVectorizer(tokenizer=tokenizer.tokenize)),
        ('classifier', MultinomialNB()),
    ])

    dataset = []
    for intent, samples in train_set.items():
        for sample in samples:
            dataset.append((sample[0], intent))

    random.shuffle(dataset)
    intent_X, intent_y = zip(*dataset)

    pipeline.fit(intent_X, intent_y)
    print("Accuracy on training set:", pipeline.score(intent_X, intent_y))
    return pipeline
```

This is how we will use this procedure:

Training the intent classifier

```
train_intent_model(MOVIE_EXPERT_TRAINING_SET)
```

Training the extractors is a bit trickier. We're going to train one extractor for each entity type. First, we need to create datasets for every entity. We're going to extract the sentences that contain references to the each entity type we want to train an extractors for.

Next thing we need to build is a mapping between intents and extractors. We don't want to apply all the extractors all the time. We want to apply an extractor only to the intents that contain samples with such entities.

Last, but not least, we need to actually train the extractor.

Train entity extractor procedure

```
def sub_list(lst, slst):
    """
    Utility function for getting
    the index of a sublist inside a list
    """

    start_index = 0
    while len(lst) > len(slst):
        if lst[:len(slst)] == slst:
            return start_index

        lst, start_index = lst[1:], start_index + 1

    return None


def mark_entities(tagged_sentence, entity_words, label):
    """
    tagged_sentence: [('Word', 'Tag'), ...]
    entity_words: ['This', 'is', 'an', 'entity']
    label: the entity type

    return a nltk.Tree instance with the entities wrapped in chunks
    """

    iob_tagged = [(w, t, 'O') for w, t in tagged_sentence]

    words = nltk.untag(tagged_sentence)
    start_index = sub_list(words, entity_words)
    if start_index is not None:
        iob_tagged[start_index] = (
            iob_tagged[start_index][0],
            iob_tagged[start_index][1],
            'B-' + label
        )
        for idx in range(1, len(entity_words)):
            iob_tagged[start_index + idx] = (
                iob_tagged[start_index + idx][0],
                iob_tagged[start_index + idx][1],
                'I-' + label
            )

    return nltk.conlltags2tree(iob_tagged)


def build_extractors_datasets(train_set):
    """
    Transform the training set from the original form nltk.Tree organized by entity_type
    {entity_type: [tree1, tree2, ...]}
    """
    tokenizer = TweetTokenizer()
    datasets = defaultdict(list)
    for _, samples in train_set.items():
        for sample in samples:
            words = tokenizer.tokenize(sample[0])
            tagged = nltk.pos_tag(words)
```

```
            for entity_type, entity in sample[1].items():
                entity_words = tokenizer.tokenize(entity)

                tree = mark_entities(tagged, entity_words, entity_type)
                datasets[entity_type].append(tree)

    return datasets


def build_intent_extractor_mapping(train_set):
    mapping = {}
    for intent in train_set:
        mapping[intent] = set({})
        for sample in train_set[intent]:
            mapping[intent].update(list(sample[1].keys()))

    return mapping


def train_extractor(trees):
    def train_scikit_classifier(sentences, feature_detector):

        dataset = feature_detector(sentences)
        featuresets, labels = zip(*dataset)

        scikit_classifier = ScikitClassifier()
        scikit_classifier.train(featuresets, labels)
        return scikit_classifier

    classifier_chunker = ClassifierBasedChunkParser(
        trees,
        chunk_features,
        train_scikit_classifier,
    )

    print("Accuracy on training set:", classifier_chunker.evaluate(trees))
    return classifier_chunker
```

Now, for actually performing the training:

Training the entity extractors

```
extractor_datasets = build_extractors_datasets(MOVIE_EXPERT_TRAINING_SET)
movie_extractor = train_extractor(extractor_datasets['movie'])
actor_extractor = train_extractor(extractor_datasets['actor'])
```

Everything together

Let’s assemble our components in a single chatbot class:

Assemble the chatbot

```
class Chatbot(object):
    def __init__(self, min_intent_confidence=.2):
        self.min_intent_confidence = min_intent_confidence

        self.intent_model = None
        self.entity_types = []
        self.extractors = {}
        self.intent_extractor_mapping = None
        self.handlers = {}

    def train(self, training_set):
        # Train intent model
        self.intent_model = train_intent_model(training_set)

        extractor_datasets = build_extractors_datasets(training_set)

        self.intent_extractor_mapping = build_intent_extractor_mapping(training_set)

        # Train extractor for each entity type
        for entity_type, data in extractor_datasets.items():
            self.extractors[entity_type] = train_extractor(data)

    def predict_intent(self, line):
        probs = self.intent_model.predict_proba([line])[0]
        best_score_index = probs.argmax()
        best_score = probs[best_score_index]

        if best_score < self.min_intent_confidence:
            return None

        return self.intent_model.classes_[best_score_index]

    def predict_entities(self, line, intent):
        tagged = nltk.pos_tag(tokenizer.tokenize(line))

        # Get applicable extractors for the intent
        applicable_extractors = self.intent_extractor_mapping[intent]

        # Apply the extractors and get the entities
        entities = {}

        for extractor_name in applicable_extractors:
            tree = self.extractors[extractor_name].parse(tagged)
            extracted_entities = [t for t in tree if isinstance(t, nltk.Tree)]

            if extracted_entities:
                first_entity = extracted_entities[0]
                entities[extractor_name] = ' '.join(t[0] for t in first_entity)

        return entities

    def handle(self, intent, entities):
        assert intent in self.handlers, "Register a handler for intent: %s" % intent
        return self.handlers[intent](entities)

    def register_handler(self, intent):
        """ Decorator for registering an intent handler """
        def wrapper(handler):
            self.handlers[intent] = handler
```

```
        return wrapper

    def register_default_handler(self):
        """ Decorator for registering the default (fallback) handler """
        def wrapper(handler):
            self.handlers[None] = handler
        return wrapper

    def tell(self, line):
        intent = self.predict_intent(line)

        # In case we haven't registered a handler for this intent
        # let's just pretend we just don't know better
        if intent not in self.handlers:
            intent = None

        if intent is not None:
            entities = self.predict_entities(line, intent)
        else:
            entities = {}

        return self.handle(intent, entities)
```

MovieBot

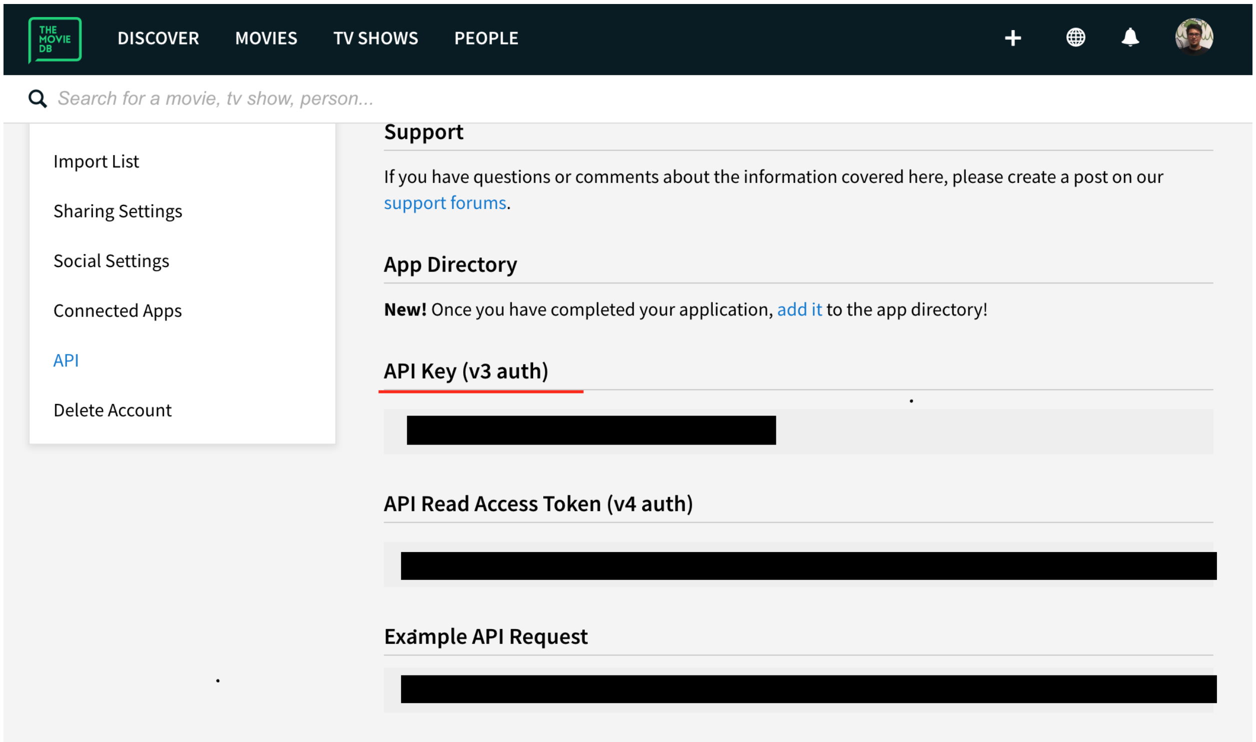
In the previous chapters we discussed about the architecture of our chatbot and we implemented it in the form of a base class named `Chatbot`. We now need to train our chatbot on the dataset we have created and create handlers for every intent.

The purpose of the handlers is to “execute stuff”. After we identified that the user is interested in the latest movies we need to actually fetch the latest movies and assemble an answer to send back.

The Movie DB API

In order to get the movie information we need, we’ll use the The Movie DB[54] API. To be able to use the API you need to create an account and request API access. After you set up API access go check out the documentation of the API here: https://developers.themoviedb.org/3[55]. We won’t use the API directly, but through a library. Go a bit through the documentation to familiarize yourself to the information is available and how to access it. In order to use the API, get your `v3 auth` API key from your API Settings Page[56]:

[54] https://www.themoviedb.org/
[55] https://developers.themoviedb.org/3
[56] https://www.themoviedb.org/settings/api



tmdb v3 auth API key

Let's now install a simple library that handles the API access for TMDB:

```
pip install tmdbsimple
```

Here's how to initialize the TMDB API:

Init TMDB API

```
import tmdbsimple as tmdb
tmdb.API_KEY = 'YOUR-TMDB-V3-API-KEY'
```

Training our chatbot is as easy as:

Init MovieBot

```
bot = Chatbot()
bot.train(MOVIE_EXPERT_TRAINING_SET)
```

Small-Talk Handlers

Before jumping into building handlers that provide movie information from TMDB let's warm up by doing some small-talk ones:

Writing simple handlers

```python
@bot.register_default_handler()
def default_response(_):
    return "Sorry, I can't help you out with that ..."


@bot.register_handler('greetings')
def say_hello(_):
    return random.choice([
        "Hi!",
        "Hi there!",
        "Hello yourself!"
    ])


@bot.register_handler('bye')
def say_bye(_):
    return random.choice([
        "Goodbye to you!",
        "Nice meeting you, bye",
        "Have a good day!",
        "Goodbye"
    ])


@bot.register_handler('thanks')
def say_thanks(_):
    return random.choice([
        "No, thank you!",
        "It's what I do ...",
        "At your service ...",
        "Come back anytime"
    ])


@bot.register_handler('chit-chat')
def make_conversation(_):
    return random.choice([
        "I'm fine, thanks for asking ...",
        "The weather is fine, business is fine, I'm pretty well myself ...",
        "I'm bored",
        "All good, thanks",
    ])


@bot.register_handler('rude')
def be_rude(_):
    return random.choice([
        "That's pretty rude of you. Come back with some manners ...",
        "You're rude, leave me alone.",
        "That's not very nice ...",
        "You're not a nice person, are you?",
    ])


@bot.register_handler('ask-info')
def provide_info(_):
    return random.choice([
        "I know a looot of movies, try me!",
        "I can help you with movies and stuff ...",
```

```
        "I am your father!",
        "I come from the future!",
    ])
```

Simple Handlers

Notice how we've added some *diversity* by selecting a random line out of a number of choices. Also notice that these handlers don't require any parameters. This is not the case for most of the *execution* endpoints that perform real operations. Let's start with the most simple one: the handler associated with the `get-latest` intent. It doesn't require any entities or parameters. It's the perfect opportunity to using the TMDB api:

Writing simple handlers

```
@bot.register_handler('get-latest')
def get_latest(_):
    try:
        latest_movies = tmdb.Movies().now_playing()['results']
        random.shuffle(latest_movies)
        return "Here are some new movies I think you're going to like: %s" % (
            ", ".join(['"' + m['original_title'] + '"' for m in latest_movies[:5]]))
    except KeyError:
        return "I'm not as up to date as I used to be. Sorry."
```

Execution Handlers

Let's now write the more execution handlers. The code is pretty straight-forward, as most of them search for a movie and extract some particular information.

Execution handlers

```
@bot.register_handler('what-director')
def what_director(entities):
    if 'movie' not in entities:
        return "Sorry, I didn't quite catch that."

    search = tmdb.Search()
    search.movie(query=entities['movie'])
    try:
        movie = tmdb.Movies(search.results[0]['id'])
    except (KeyError, IndexError):
        return "That's weird ... I never heard of this movie."

    try:
        director = [p for p in movie.credits()['crew'] if p['job'] == 'Director'][0]
```

```
        return "The director of %s is %s" % (movie.info()['title'], director['name'])
    except (KeyError, IndexError):
        return "That's weird ... I don't know this one."


@bot.register_handler('what-release-year')
def what_release_year(entities):
    if 'movie' not in entities:
        return "Sorry, what movie is that again?"

    search = tmdb.Search()
    search.movie(query=entities['movie'])

    try:
        movie = search.results[0]
    except (KeyError, IndexError):
        return "Is that a real movie or did you just make it up?"

    try:
        return "The release date of %s is %s" % (movie['title'], movie['release_date'])
    except (KeyError, IndexError):
        return "That's weird ... I don't know the release date."


@bot.register_handler('get-similar-movies')
def get_similar_movies(entities):
    if 'movie' not in entities:
        return "Sorry, I didn't quite catch that."

    search = tmdb.Search()
    search.movie(query=entities['movie'])

    try:
        movie = tmdb.Movies(search.results[0]['id'])
    except (KeyError, IndexError):
        return "That's weird ... I never heard of this movie."

    try:
        recommendations = movie.similar_movies()['results']
        random.shuffle(recommendations)
        return "Here are a few ones I think you're going to enjoy: %s" % (
            ", ".join(['"' + m['original_title'] + '"' for m in recommendations[:5]]))
    except (KeyError, IndexError):
        return "That's weird ... I don't know this one."


@bot.register_handler('actor-movies')
def get_actor_movies(entities):
    try:
        actor = entities['actor']
    except KeyError:
        return "Sorry, what actor is that?"

    search = tmdb.Search()
    search.person(query=actor)

    try:
        actor = tmdb.People(search.results[0]['id'])
    except (IndexError, KeyError):
        return "That's weird ... I never heard of him/her."
```

```
    movies = actor.movie_credits()['cast']

    if not movies:
        return "I don't know of any movies featuring him/her."

    random.shuffle(movies)

    return "Ok, here are some movies %s played in: %s" % (
        "he" if actor.info()['gender'] == 2 else "she",
        ", ".join('"%s" in the role of %s' % (m['title'], m['character']) for m in movies[:5])
    )


@bot.register_handler('movie-cast')
def movie_cast(entities):
    if 'movie' not in entities:
        return "Sorry, I didn't quite catch that."

    search = tmdb.Search()
    search.movie(query=entities['movie'])

    try:
        movie = tmdb.Movies(search.results[0]['id'])
    except (IndexError, KeyError):
        return "That's weird ... I never heard of this movie."

    cast = movie.credits()['cast']
    if not cast:
        return "I don't know of any actors playing in this movie ..."

    return "Here are the main actors: %s" % (
        ", ".join('"%s" playing the role of %s' % (m['name'], m['character']) for m in cast[:5]))


@bot.register_handler('actor-in-movie')
def actor_in_movie(entities):
    try:
        actor, movie = entities['actor'], entities['movie']
    except KeyError:
        return "Did who play in what?"

    search = tmdb.Search()
    search.movie(query=movie)

    try:
        movie = tmdb.Movies(search.results[0]['id'])
    except (IndexError, KeyError):
        return "That's weird ... I never heard of this movie."

    search = tmdb.Search()
    search.person(query=actor)

    try:
        actor = tmdb.People(search.results[0]['id'])
    except (IndexError, KeyError):
        return "That's weird ... I never heard of this actor."

    cast = movie.credits()['cast']
    role = [a for a in cast if a['id'] == actor.info()['id']]

    if not role:
```

```
        return 'Nope, I believe "%s" didn\'t play in %s' % (actor.info()['name'], movie.info()['original_title'])

    return 'Yep, "%s" played the role of "%s" in "%s"' % (
        actor.info()['name'], role[0]['character'], movie.info()['original_title']
    )
```

Let's take it for a test run:

The loop

```
while True:
    line = input("You: ")
    print("MovieBot: ", bot.tell(line))
```

Below there is a snippet of an authentic conversation. I didn't pick a staged example that makes our chatbot look perfect.

Chatting

Go ahead and have some fun with your trained bot. Find some examples where it fails and try to edit the training set to fix those scenarios.

MovieBot on Facebook

We've arrived at the end of our journey. We're going to launch our chatbot on Facebook. It is going to be pretty easy. Disclaimer: the Facebook chatbot platform is a bit volatile and stuff changes quite frequently. T he exact steps described here might not be the ones you have to take when you're reading this. The API documentation is luckly comprehensive enough.

Here's we need to do:

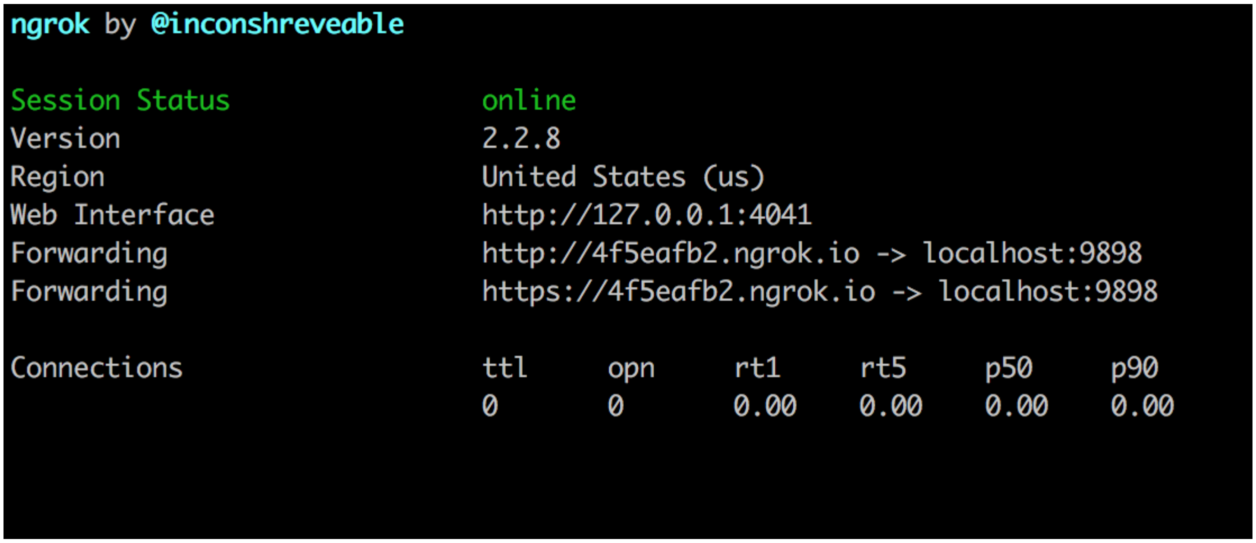
1. Create a Facebook Page
2. Create a Facebook Application
3. Build a Python script that interacts via an API with the Facebook Chatbot API
4. Expose your script on a public IP: In order for Facebook to be able to connect to our bot it needs a public IP. Most likely, you are running your chatbot on localhost, same as me, so you will need to install a tool called **ngrok** that creates a HTTPS a tunnel with a public IP. If this is not your case, use your assigned public IP.

Installing ngrok

Head over to ngrok.com[57] to download and install this utility. Here's how to create a tunnel on port 9898:

```
$ ./ngrok http 9898
```

[57] https://ngrok.com/



Launch ngrok

Notice how ngrok created 2 URLs that are accessible from outside, both for HTTP and HTTPS. Your URLs will be different. Try them out and see what happens. You probably get a message saying that there was no client at the end of the ngrok tunnel. That's the job of our chatbot.

In order to make our chatbot play with the Facebook platform, we need to adhere to some standards:

- Receive requests via https (solved with ngrok)
- Verify a challenge that we've entered on the Facebook dashboard. This way, we know it's indeed Facebook who's hitting our endpoint
- Respond to messages by posting to an unique URL. This URL is built using a unique `FB_PAGE_ACCESS_TOKEN` that Facebook provided.

Let's start by putting sensitive information inside a `.env` file like this:

```
FB_CHALLENGE='YOUR-CHALLENGE-PHRASE'
FB_PAGE_ACCESS_TOKEN='YOUR-ACCESS-TOKEN'
```

You don't have an access token yet but this is the place where you are going to put it. You can come up with whatever challenge phrase you want. Make sure you load the environment variables by running:

```
$ source .env
```

A Facebook chatbot is basically a web application with only one endpoint that Facebook will be calling. If the request method is `GET` then Facebook is doing a webhook verification. If the request method is `POST` than Facebook is sending an event, usually an incoming message. Let's write a simple Flask application for this:

Building the Flask application

```
import os
import json
import requests
from flask import Flask, request
from moviebot import bot

app = Flask(__name__)
MOVIE_BOT_CHALLENGE = os.environ.get('FB_CHALLENGE')
FB_PAGE_ACCESS_TOKEN = os.environ.get('FB_PAGE_ACCESS_TOKEN')


@app.route('/webhook', methods=['GET', 'POST'])
def webhook():
    # Webhook verification
    if request.method == 'GET':
        mode = request.args.get('hub.mode')
        token = request.args.get('hub.verify_token')
        challenge = request.args.get('hub.challenge')

        if mode and token:
            if mode == 'subscribe' and token == MOVIE_BOT_CHALLENGE:
                return challenge
```

In order to run the Flask application, append this line to the file:

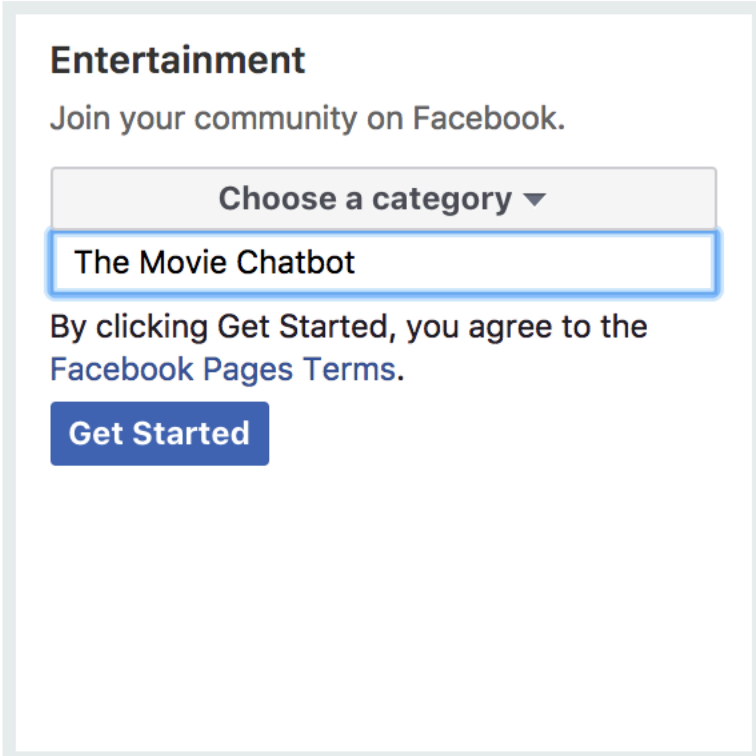
```
app.run('0.0.0.0', 9898)
```

Setting up Facebook

To connect the chatbot to a page, you need to either create a Facebook Page or use an existing one (this is the place where the chatbot will be displayed) and a Facebook Application (this will be the actual chatbot).

In case you never created a Facebook Page before, you can do so here: Create Facebook Page[58] I created an Entertainment Facebook page named "The Movie Chatbot" and chose the category "TV Show" cause nothing else really fitted.

[58] https://www.facebook.com/pages/create/



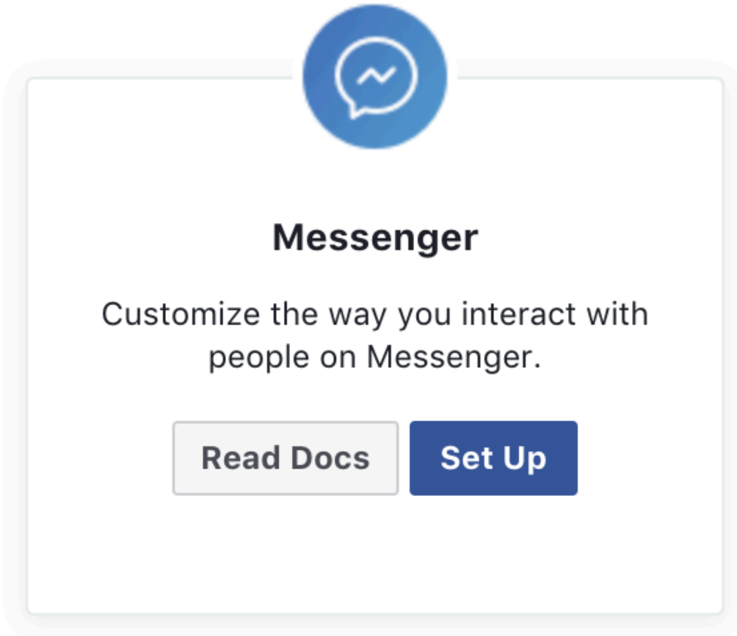
Create Facebook Page

You now have an empty Facebook page:

The new Facebook Page

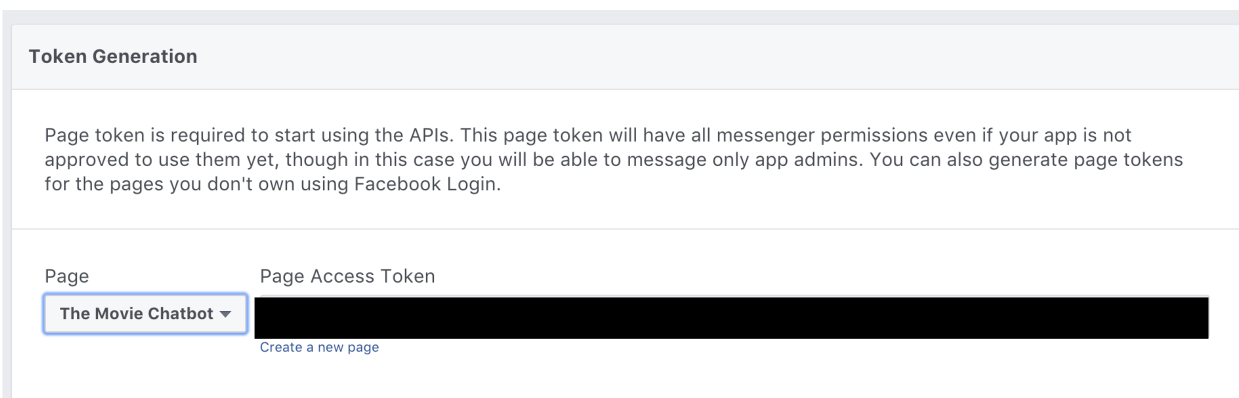
In order to create a Facebook Application, head over to: Facebook Developers[59]. I named mine **TheMovieChatbot**. For the Facebook Application to be a chatbot, in the next screen you need to select the *Messenger* platform:

[59]https://developers.facebook.com/apps/



Facebook Messenger Platform

Head over to the *Messenger* menu item inside your Facebook Application. We now need to connect the application to the Facebook Page:

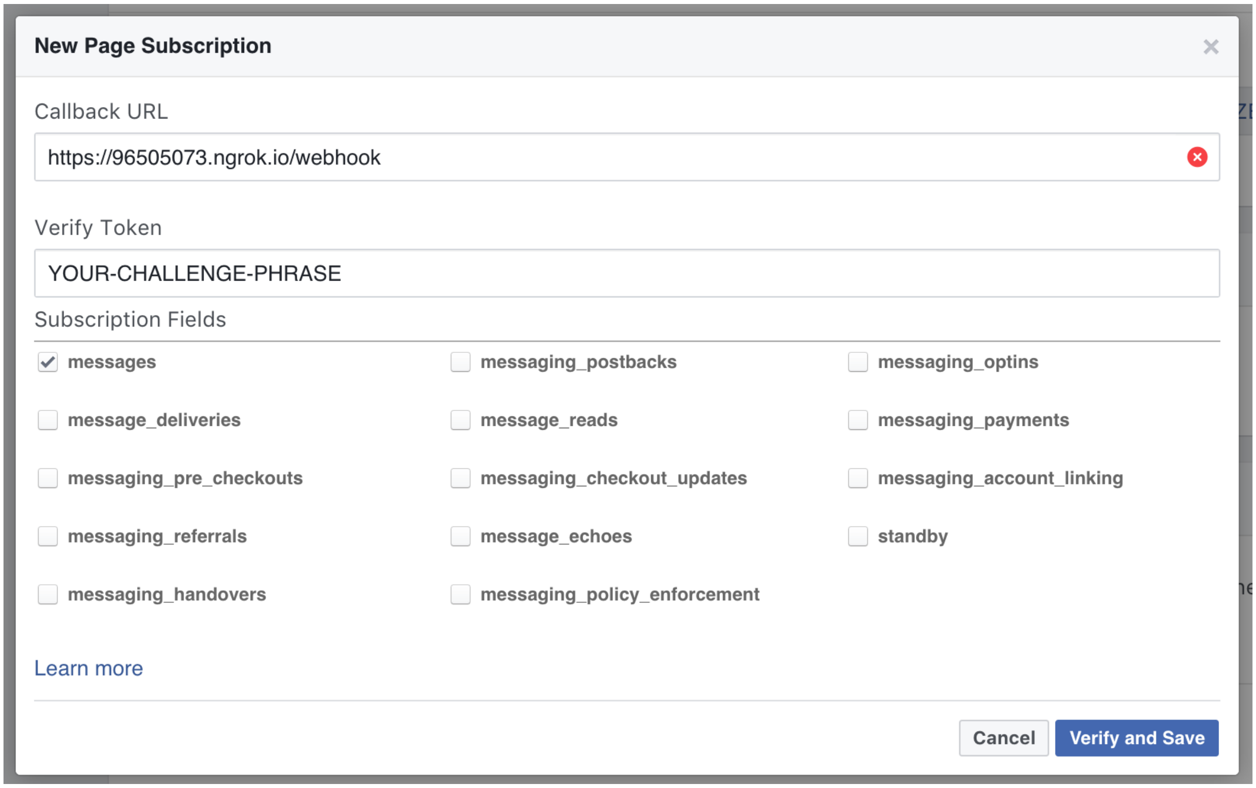


Connect App to Page

Notice how Facebook generated a *Page Access Token* for us. Make sure you copy&paste it inside the `.env` file and reload it:

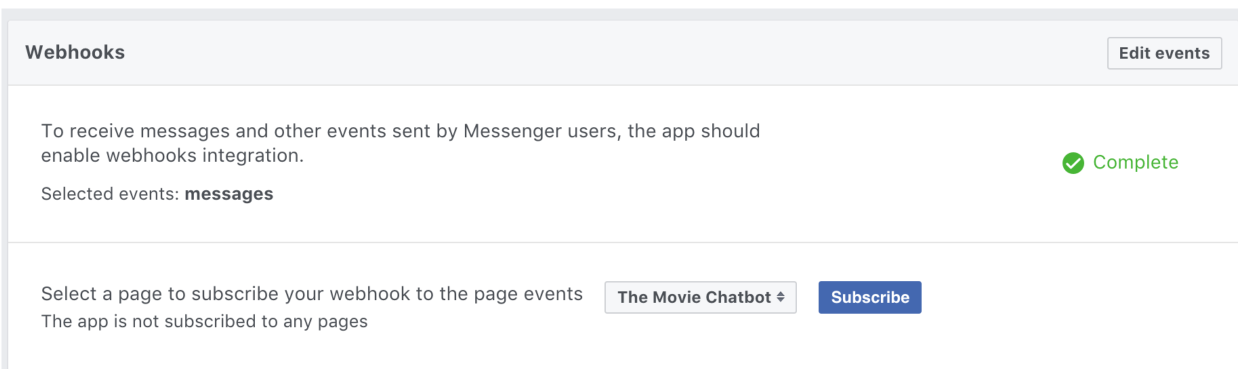
```
$ source .env
```

In the webhook section we now need to register the webhook we created:



Verify Webhook

Let's subscribe to our page to listen for events:



Subscribe to Page

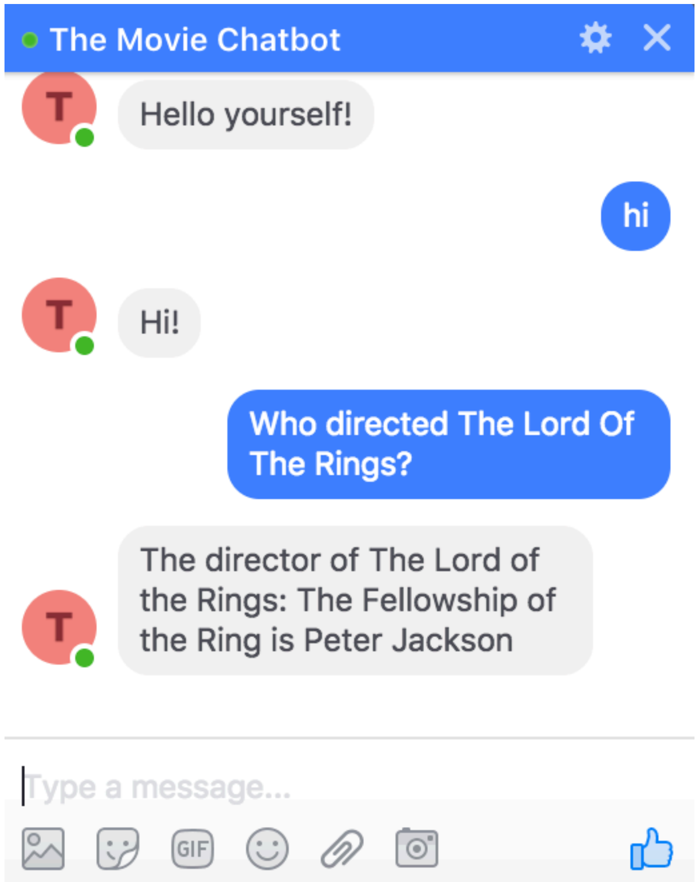
Trying it Out

If everything went smoothly up to this point, when you visit your page and hit the *Send Message* button you will notice that a messenger box appears. You can write stuff to it, but the chatbot won't answer. Let's write the code to handle incoming messages:

Complete Chatbot code

```
@app.route('/webhook', methods=['GET', 'POST'])
def webhook():
    # Webhook verification
    if request.method == 'GET':
        mode = request.args.get('hub.mode')
        token = request.args.get('hub.verify_token')
        challenge = request.args.get('hub.challenge')

        if mode and token:
            if mode == 'subscribe' and token == MOVIE_BOT_CHALLENGE:
                return challenge

    # Incoming message
    elif request.method == 'POST':

        event = request.get_json()
        if event['object'] == 'page':
            for entry in event['entry']:
                try:
                    message_event = entry['messaging'][0]
                    message_handler(message_event['sender']['id'],
                                    message_event['message'])
                except IndexError:
                    pass
        return "ok"


def message_handler(sender_id, message):
    text = message['text']
    answer = bot.tell(text)
    uri = "https://graph.facebook.com/v2.6/me/messages?access_token=%s" % FB_PAGE_ACCESS_TOKEN
    data = {
        'messaging_type': 'RESPONSE',
        'recipient': {'id': sender_id},
        'message': {'text': answer}
    }
    requests.post(uri, data=json.dumps(data),
                  headers={'content-type': 'application/json'})


app.run('0.0.0.0', 9898)
```

And we’re done! Let’s test it out:



Play with TheMovieBot

What Next?

You now know a lot about Natural Language Processing tools, and best of all, you know how to build them from scratch, just in case what's already out there doesn't fit your needs. There's a lot of room for improvements and this is not an exhaustive resource on all things related to Natural Language Processing. The main purpose of the book is to provide a solid basis for getting started with the subject. Please make sure to follow my blog: http://nlpforhackers.io[60] for new content. The plan is to write content there and when I consider I gathered enough feedback I will either update this book or start another one.

What content would you like to see in the book? Send me an email at: *me@bogs.io* or contact me via the form here: https://nlpforhackers.io/contact/[61]

Thank you,

George-Bogdan Ivanov.

[60] http://nlpforhackers.io
[61] https://nlpforhackers.io/contact/