Foreword by Dr. Arwen Griffioen

Generate Tokenizers regular expressions tag NER extract information reduce dimensions Check spelling check grammar analyze sentiment analyze humanness analyze style CNN GAN Search templates FSM MCMC RBM RNN Response feature vector Database Statements responses scroes user profiles Scored statements Possible responses   Natural Language Processing in Action   ii Natural Language Processing in Action Understanding, analyzing, and generating text with Python HOBSON LANE COLE HOWARD HANNES MAX HAPKE M A N N I N G SHELTER ISLAND   For online information and ordering of this and other Manning books, please visit www.manning.com. Manning Publications Co. Acquisitions editor: Brian Sawyer 20 Baldwin Road Development editor: Karen Miller PO Box 761 Technical development editor: René van den Berg Shelter Island, NY 11964 Review editor: Ivan Martinovic´ Production editor: Anthony Calcara Copy editor: Darren Meiss Proofreader: Alyson Brener Technical proofreader: Davide Cadamuro Typesetter and cover designer: Marija Tudor ISBN 9781617294631 Printed in the United States of America 1 2 3 4 5 6 7 8 9 10 ? SP ? 24 23 22 21 20 19

contents

foreword

He quickly became known for his work leveraging the union of machine learning and electrical engineering and, in particular, a strong commitment to having a positive world impact. Throughout his career, this commitment has guided each company and project he has touched, and it was by following this inter- nal compass that he connected with Hobson and Cole, who share similar passion for projects with a strong positive impact. Whether you attribute these words to Voltaire or Uncle Ben, they hold as true today as ever, though perhaps in this age we could rephrase to say, ?With great access to data comes great responsibility.? We trust companies with our data in the hope that it is used to improve our lives. Taking the reader on a clear and well-narrated tour through the core methodologies of natural language processing, the authors begin with tried and true methods, such as TF-IDF, before taking a shallow but deep (yes, I made a pun) dive into deep neural networks for NLP. You have the opportunity to develop a solid understanding, not just of the mechanics of NLP, but the opportunities to generate impactful systems that may one day understand humankind through our language.

preface

It was first introduced on Reddit as the ?SwiftKey game? (https://blog.swiftkey.com/swiftkey-game-winning-is/) in 2013. xv   xvi PREFACE gained confidence, and learned more and more from my mentors and mentees, it seemed like I might be able to build something new and magical myself. 3 If you appreciate the importance of having freely accessible books of natural language, you may want to keep abreast of the international effort to extend copyrights far beyond their original ?use by? date: gutenberg.org (http://www.gutenberg.org) and gutenbergnews.org (http://www.gutenbergnews.org/20150208/copyright-term-extensions-are-looming:). The bots are literally talking to each other and attempting to manipulate each other, while 4 See the web page titled ?Why Banjo Is the Most Important Social Media Company You?ve Never Heard Of? (https://www.inc.com/magazine/201504/will-bourne/banjo-the-gods-eye-view.html). Hopefully you?ve been following the discussion among movers and shakers about the AI Control Problem and the challenge of developing ?Friendly AI.?14 Nick Bostrom,15 9 Duck Duck Go query about NLP (https://duckduckgo.com/?q=Why+is+natural+language+processing+so +important+right+now:) 10 See the Wikipedia article ?Natural language processing? (https://en.wikipedia.org/wiki/Natural_language _processingWikipedia/NLP). And we?re using our collective intelligence to help build and support other semi-intelligent actors (machines).23 We hope that our words will leave their impression in your mind and propagate like a meme through the world of chatbots, infecting others with passion for building prosocial NLP sys- tems.

acknowledgments

These con- tributors came from a vibrant Portland community sustained by organizations like PDX Python, Hack Oregon, Hack University, Civic U, PDX Data Science, Hopester, PyDX, PyLadies, and Total Good. Kudos to Zachary Kent who designed, built, and maintained openchat (PyCon Open Spaces Twitter bot) and Riley Rustad who prototyped its data schema as the book and our skills progressed. Santi Adavani implemented named entity recognition using the Stanford CoreNLP library, developed tutorials for SVD and PCA, and sup- ported us with access to his RocketML HPC framework to train a real-time video description model for people who are blind. Chick Wells cofounded Total Good, developed a clever and entertaining IQ Test for chatbots, and continuously supported us with his devops expertise. NLP experts, like Kyle Gorman, generously shared their time, NLP expertise, code, and precious datasets with us. Chris Gian contributed his NLP project ideas to the examples in this book, and val- iantly took over as instructor for the Civic U Machine Learning class when the teacher bailed halfway through the climb.

Hannes Max Hapke

I owe many thanks to my partner, Whitney, who supported me endlessly in this endeavor. I also would like to thank my fam- ily, especially my parents, who encouraged me to venture out into the world to dis- cover it.

Cole Howard

I would like to thank my wife, Dawn. And my mother, for the freedom to experiment and the encouragement to always be learning.

Roadmap

If you are new to Python and natural language processing, you should first read part 1 and then any of the chapters of part 3 that apply to your interests or on-the-job chal- lenges. If you want to get up to speed on the new NLP capabilities that deep learning enables, you?ll also want to read part 2, in order. And if any of the examples look like they might run on your own text documents, you should put that text into a CSV or text file (one document per line) in the nlpia/src/nlpia/data/ directory.

About the code

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text. Sometimes code is also in bold to high- light code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

liveBook discussion forum

Purchase of Natural Language Processing in Action includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum, go to https://livebook.manning.com/#!/book/natural-language- processing-in-action/discussion.

about the authors

Hannes presented on machine learning at various conferences includ- ing OSCON, Open Source Bridge, and Hack University. He has developed large-scale e-commerce recommendation engines and state-of-the-art neural nets for hyperdimensional machine intelligence systems (deep learning neural nets), which perform at the top of the leader board for the Kaggle competitions. He has presented talks on Convolu- tional Neural Nets, Recurrent Neural Nets, and their roles in natural language processing at the Open Source Bridge Conference and Hack University.

about the cover illustration

The figure on the cover of Natural Language Processing in Action is captioned ?Woman from Kranjska Gora, Slovenia.? This illustration is taken from a recent reprint of Balthasar Hacquet?s Images and Descriptions of Southwestern and Eastern Wends, Illyrians, and Slavs, published by the Ethnographic Museum in Split, Croatia, in 2008. Hacquet (1739?1815) was an Austrian physician and scientist who spent many years studying the botany, geology, and ethnography of the Julian Alps, the mountain range that stretches from northeastern Italy to Slovenia and that is named after Julius Caesar. We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago? brought back to life by the pictures from this collection.

Part 1 Wordy machinesPart 1 30.0

2 CHAPTER Packets of thought (NLP overview) This chapter covers ? What natural language processing (NLP) is ? Why NLP is hard and only recently has become widespread ? When word order and grammar is important and when it can be ignored ? How a chatbot combines many of the tools of NLP ? How to use a regular expression to build the start of a tiny chatbot You are about to embark on an exciting adventure in natural language processing.

1.2 The magic

However, these ?formal? languages?such as early languages Ada, COBOL, and Fortran?were designed to be interpreted (or compiled) only one correct way.

1.2.1 Machines that converse

Natural languages can?t be directly translated into a precise set of mathematical oper- ations, but they do contain information and instructions that can be extracted. Those pieces of information and instruction can be stored, indexed, searched, or immediately 1 Ethnologue is a web-based publication that maintains statistics about natural languages. 2 See the web page titled ?How Google?s Site Crawlers Index Your Site - Google Search? (https://www.google .com/search/howsearchworks/crawling-indexing/). See the ?Mathematical notation? section of the Wikipedia article ?Ambi- guity? (https://en.wikipedia.org/wiki/Ambiguity#Mathematical_notation).

1.2.2 The math

The techniques you?ll learn, however, are powerful enough to create machines that can surpass humans in both accuracy and speed for some surprisingly subtle tasks. For example, you might not have guessed that recognizing sarcasm in an iso- lated Twitter message can be done more accurately by a machine than by a human.5 Don?t worry, humans are still better at recognizing humor and sarcasm within an ongoing dialog, due to our ability to maintain information about the context of a statement. And this book helps you incorporate context (metadata) into your NLP pipeline, in case you want to try your hand at advancing the state of the art. These breakthrough ideas opened up a world of ?semantic? analysis, allowing computers to interpret and store the ?meaning? of statements rather than just word or character counts. Semantic analysis, along with statistics, can help resolve the ambiguity of natural language?the fact that words or phrases often have multiple meanings or interpretations. Speakers and writers of natural languages assume that a human is the one doing the processing (listening or reading), not a machine. So when I say ?good morning?, I assume that you have some knowledge about what makes up a morning, including not only that mornings come before noons and afternoons and evenings but also after midnights. However, we show you techniques in later chap- ters to help machines build ontologies, or knowledge bases, of common sense knowl- edge to help interpret statements that rely on this knowledge.

1.3 Practical applications

Table 1.1 Categorized NLP applications Search Web Documents Autocomplete Editing Spelling Grammar Style Dialog Chatbot Assistant Scheduling Writing Index Concordance Table of contents Email Spam filter Classification Prioritization Text mining Summarization Knowledge extraction Medical diagnoses Law Legal inference Precedent search Subpoena classification News Event detection Fact checking Headline composition Attribution Plagiarism detection Literary forensics Style coaching Sentiment analysis Community morale monitoring Product review triage Customer care Behavior prediction Finance Election forecasting Marketing Creative writing Movie scripts Poetry Song lyrics A search engine can provide more meaningful results if it indexes web pages or docu- ment archives in a way that takes into account the meaning of natural language text.

1.4 Language through a computer?s ?eyes?

This would be equivalent to writing a 8 New York Times, Oct 18, 2016, https://www.nytimes.com/2016/11/18/technology/automated-pro-trump-bots-overwhelmed-pro-clinton-messages-researchers-say.html and MIT Technology Review, Nov 2016, https://www.technologyreview.com/s/602817/how-the-bot-y-politic-influenced-this-election/. 10 See the web page titled ?AI control problem - Wikipedia? (https://en.wikipedia.org/wiki/AI_control_problem). 11 WSJ Blog, March 10, 2017 https://blogs.wsj.com/cio/2017/03/10/homo-deus-author-yuval-noah-harari-says-authority-shifting-from-people-to-ai/.

1.4.1 The language of locks

A combination lock certainly can?t read and understand the textbooks stored inside a school locker, but it can understand the language of locks. Even more importantly, the padlock can tell if a lock ?statement? matches a particularly meaningful statement, the one for which there?s only one correct ?response?: to release the catch holding the U-shaped hasp so you can get into your locker. That?s the reason for this diversion into the mechanical, ?click, whirr?12 language of locks.

1.4.2 Regular expressions

For those who can?t resist trying to understand a bit more about these computer science tools, figure 1.1 shows where Combinational logic Finite-state machine Pushdown automaton Turing machine Figure 1.1 Kinds of automata 13 Stack Exchange went down for 30 minutes on July 20, 2016 when a regex ?crashed? (http://stackstatus.net/ post/147710624694/outage-postmortem-july-20-2016).

1.4.3 A simple chatbot

>>> import re >>> r = "(hi|hello|hey)[ ]*([a-z]*)" >>> re.match(r, 'Hello Rosa', flags=re.IGNORECASE) <_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'> >>> re.match(r, "hi ho, hi ho, it's off to work ...", flags=re.IGNORECASE) <_sre.SRE_Match object; span=(0, 5), match='hi ho'> >>> re.match(r, "hey, what's up", flags=re.IGNORECASE) <_sre.SRE_Match object; span=(0, 3), match='hey> Ignoring the case of text characters is common, to keep the regular expressions simpler. (morn[gin']{0,3}|"\ ... r"afternoon|even[gin']{0,3}))[\s,;:]{1,3}([a-z]{1,20})" >>> re_greeting = re.compile(r, flags=re.IGNORECASE) 14 CHAPTER 1 Packets of thought (NLP overview) >>> re_greeting.match('Hello Rosa') <_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'> >>> re_greeting.match('Hello Rosa').groups() ('Hello', None, None, 'Rosa') >>> re_greeting.match("Good morning Rosa") <_sre.SRE_Match object; span=(0, 17), match="Good morning Rosa"> >>> re_greeting.match("Good Manning Rosa") >>> re_greeting.match('Good evening Rosa Parks').groups() ('Good evening', 'Good ', 'evening', 'Rosa') >>> re_greeting.match("Good Morn'n Rosa") <_sre.SRE_Match object; span=(0, 16), match="Good Morn'n Rosa"> >>> re_greeting.match("yo Rosa") <_sre.SRE_Match object; span=(0, 7), match='yo Rosa'> Notice that this regular expression cannot recognize (match) words with typos. We use Python?s string formatter to create a ?template? for our chatbot response: >>> my_names = set(['rosa', 'rose', 'chatty', 'chatbot', 'bot', ... 'chatterbot']) >>> curt_names = set(['hal', 'you', 'u']) >>> greeter_name = '' >>> match = re_greeting.match(input()) ... >>> if match: ... at_name = match.groups()[-1] We don?t yet know who is chatting with the bot, and we won?t worry about it here.

1.4.4 Another way

And a Python Counter is a special kind of dictionary that bins objects (including strings) and counts them just like we want: Tokenizer a rat A bat and Good Bag of words a and bat Stop A A a Good Rare Rosa bat any Cat bat Bat any Any Bag-of-words [3, 1, 4, 8, 9, 0,..,] vector Figure 1.2 Token sorting tray >>> from collections import Counter >>> Counter("Guten Morgen Rosa".split()) Counter({'Guten': 1, 'Rosa': 1, 'morgen': 1}) >>> Counter("Good morning, Rosa!

1.5 A brief overflight of hyperspace

And when those tokens are each treated as separate, distinct dimensions, there?s no concept that?Good morning, Hobs? has some shared meaning with ?Guten Morgen, Hannes.? We need to create some reduced dimension vector space model of messages so we can label them with a set of continuous (float) values. Well, we simplify our vector dimension questions to things like ?Does it contain the word ?good??? Does it contain the word ?morning?? And so on. The ?notes? for this natural language mechanical player piano are the 26 uppercase and lowercase letters plus any punctuation that the piano must know how to ?play.? The paper roll wouldn?t have to be much wider than for a real player piano, and the number of notes in some long piano songs doesn?t exceed the number of characters in a small document.

1.6 Word order and grammar

Take a look at all these orderings of our ?Good morning Rosa? example: >>> from itertools import permutations >>> [" ".join(combo) for combo in\ ... permutations("Good morning Rosa! ".split(), 3)] ['Good morning Rosa! morning Good'] Now if you tried to interpret each of these strings in isolation (without looking at the others), you?d probably conclude that they all probably had similar intent or mean- ing. Nonetheless, a smart chatbot or clever woman of the 1940s in Bletchley Park would likely respond to any of these six permutations with the same innocuous greeting, ?Good morning my dear General.?

1.7 A chatbot natural language pipeline

The NLP pipeline required to build a dialog engine, or chatbot, is similar to the pipe- line required to build a question answering system described in Taming Text (Manning, 2013).19 However, some of the algorithms listed within the five subsystem blocks may 18 A comparison of the syntax parsing accuracy of SpaCy (93%), SyntaxNet (94%), Stanford?s CoreNLP (90%), and others is available at https://spacy.io/docs/api/. Generate Tokenizers regular expressions tag NER extract information reduce dimensions Check spelling check grammar analyze sentiment analyze humanness analyze style LSTM Search templates FSM MCMC LSTM Response feature vector Database Statements responses scroes user profiles Scored statements Possible responses Figure 1.3 Chatbot recirculating (recurrent) pipeline   24 CHAPTER 1 Packets of thought (NLP overview) Most chatbots will contain elements of all five of these subsystems (the four processing stages as well as the database).

1.8 Processing in depth

Data structure Example Algorithm Applications Cryptography, compression, spelling correction, predictive text, search, dialog (chatbot) Search, stylistics, spam filter, sentiment analysis, word2vec math, semantic search, dialog (chatbot) Spelling and grammar correction, stylistics, dialog (chatbot) Characters Regular expression Tokens POS tagger (FST) Tagged tokens Syntax tree Question answering, stylistics, complex dialog, grammar correction, writing coach Information extractor (FST) Knowledge extraction and inference, medical diagnosis, question answering, game playing morning good Entity relationships Logic compiler (FST) part morning day of Theorem proving, inference, natural language database queries, artificial general intelligence (AGI) .82 Knowledge base name Rosa of Person .87 female .74 English Figure 1.4 Example layers for an NLP pipeline   26 CHAPTER 1 Packets of thought (NLP overview) The bottom two layers (Entity relationships and a Knowledge base) are used to popu- late a database containing information (knowledge) about a particular domain.

1.9 Natural language IQ

Chatbots require all the tools of NLP to work well: Recruiting & match-making Depth Legal advice IQ (Complexity) Finance prediction Sports reporting Siri You are here (NLPIA) Lex Will ChatterBot Breadth Figure 1.5 2D IQ of some natural language processing systems ? Feature extraction (usually to produce a vector space model) ? Information extraction to be able to answer factual questions ? Semantic search to learn from previously recorded natural language text or dialog ? Natural language generation to compose new, meaningful statements Machine learning gives us a way to trick machines into behaving as if we?d spent a life- time programming them with hundreds of complex regular expressions or algo- rithms.

Summary

? Good NLP may help save the world. ? The meaning and intent of words can be deciphered by machines. ? A smart NLP pipeline will be able to deal with ambiguity. ? We can teach machines common sense knowledge without spending a lifetime training them. ? Chatbots can be thought of as semantic search engines.

2 Build your vocabulary (word tokenization)Build your vocabulary 30.0

The single statement ?Don?t!? means ?Don?t you do that!? or ?You, do not do that!? That?s three hidden packets of meaning for a total of five tokens you?d like your machine to know about. 2.1 Challenges (a preview of stemming) As an example of why feature extraction from text is hard, consider stemming?group- ing the various inflections of a word into the same ?bucket? or cluster. Imagine trying to remove verb endings like ?ing? from ?ending? so you?d have a stem called ?end? to represent both words. And you?d like to stem the word ?running? to ?run,? so those two words are treated the same. You wouldn?t want to remove the ?ing? ending from ?sing? or you?d end up with a single- letter ?s.? Or imagine trying to discriminate between a pluralizing ?s? at the end of a word like ?words? and a normal ?s? at the end of words like ?bus? and ?lens.? Do isolated individual letters in a word or parts of a word provide any information at all about that word?s meaning?

2.2 Building your vocabulary with a tokenizer

>>> import numpy as np >>> token_sequence = str.split(sentence) >>> vocab = sorted(set(token_sequence)) >>> ', '.join(vocab) '26., Jefferson, Monticello, Thomas, age, at, began, building, of, the' >>> num_tokens = len(token_sequence) >>> vocab_size = len(vocab) >>> onehot_vectors = np.zeros((num_tokens, ... vocab_size), int) >>> for i, word in enumerate(token_sequence): ... onehot_vectors[i, vocab.index(word)] = 1 >>> ' '.join(vocab) '26. Jefferson Monticello Thomas age at began building of the' >>> onehot_vectors array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0], [0, 1, 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, 1, 0, 0], [0, 0, 1, 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, 1], [0, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]) Sorted lexographically (lexically) so numbers come before letters, and capital letters come before lowercase letters. At the top of that column, the seventh column in the table, you can find the natural lan- guage representation of that word, ?began.? 37 Building your vocabulary with a tokenizer Each row of the table is a binary row vector, and you can see why it?s also called a one-hot vector: all but one of the positions (columns) in a row are 0 or blank. That?s a whole lot of big tables (matrices): Number of rows in the table >>> num_rows = 3000 * 3500 * 15 >>> num_rows 157500000 >>> num_bytes = num_rows * 1000000 >>> num_bytes 157500000000000 >>> num_bytes / 1e9 157500 # gigabytes >>> _ / 1000 157.5 # terabytes Number of bytes, if you use only one byte for each cell in your table In a python interactive console, the variable name "_" is automatically assigned the value of the previous output. 40 CHAPTER 2 Build your vocabulary (word tokenization) Here?s what your single text document, the sentence about Thomas Jefferson, looks like as a binary bag-of-words vector: >>> sentence_bow = {} >>> for token in sentence.split(): ... sentence_bow[token] = 1 >>> sorted(sentence_bow.items()) [('26. All this hand waving about gaps in the vectors and sparse versus dense bags of words should become clear as you add more sentences and their corresponding bag-of-words vectors to your DataFrame (table of vectors corresponding to texts in a corpus): >>> import pandas as pd >>> df = pd.DataFrame(pd.Series(dict([(token, 1) for token in ... sentence.split()])), columns=['sent']).T >>> df 26.

2.2.1 Dot product

The dot product is also called the inner product because the ?inner? dimension of the two vectors (the number of elements in each vector) or matrices (the rows of the first matrix and the columns of the second matrix) must be the same, because that?s where the products happen.

2.2.2 Measuring bag-of-words overlap

Listing 2.6 Overlap of word counts for two bag-of-words vectors >>> df = df.T >>> df.sent0.dot(df.sent1) 0 >>> df.sent0.dot(df.sent2) 1 >>> df.sent0.dot(df.sent3) 1 From this you can tell that one word was used in both sent0 and sent2.

2.2.3 A token improvement

]+', sentence) >>> tokens ['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26', ''] This splits the sentence on whitespace or punctuation that occurs at least once (note the '+' after the closing square bracket in the regular expression). The re.split function goes through each character in the input string (the sec- ond argument, sentence) left to right looking for any matches based on the ?pro- gram? in the regular expression (the first argument, r'[-\s.,;!?]+').])+'') >>> tokens = pattern.split(sentence) >>> tokens[-10:] # just the last 10 tokens ['the', ' ', 'age', ' ', 'of', ' ', '26', '. >>> tokens = pattern.split(sentence) >>> [x for x in tokens if x and x not in '- \t\n.,;!?'] ['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26'] If you want practice with lambda and filter(), use list(filter(lambda x: x if x and x not in '- \t\n.,;!?']+|\S+') >>> tokenizer.tokenize(sentence) ['Thomas', 'Jefferson', 'began', 'building', 'Monticello',   47 Building your vocabulary with a tokenizer 'at', 'the', 'age', 'of', '26', '.'] >>> tokenizer = TreebankWordTokenizer() >>> tokenizer.tokenize(sentence) ['Monticello', 'was', "n't", 'designated', 'as', 'UNESCO', 'World', 'Heritage', 'Site', 'until', '1987', '.']

2.2.4 Extending your vocabulary with n-grams

['Thomas Jefferson', 'Jefferson began', 'began building', 'building Monticello', 'Monticello at', 'at the', 'the age', 'age of', 'of 26'] I bet you can see how this sequence of 2-grams retains a bit more information than if you?d just tokenized the sentence into words. >>> tokens ['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26'] And this is the n-gram tokenizer from nltk in action: >>> from nltk.util import ngrams >>> list(ngrams(tokens, 2)) [('Thomas', 'Jefferson'), ('Jefferson', 'began'), ('began', 'building'), ('building', 'Monticello'), ('Monticello', 'at'), ('at', 'the'), ('the', 'age'), ('age', 'of'), ('of', '26')] >>> list(ngrams(tokens, 3)) [('Thomas', 'Jefferson', 'began'), ('Jefferson', 'began', 'building'), ('began', 'building', 'Monticello'), ('building', 'Monticello', 'at'), ('Monticello', 'at', 'the'), ('at', 'the', 'age'), ('the', 'age', 'of'), ('age', 'of', '26')] In order to be more memory efficient, the ngrams function of the NLTK TIP library returns a Python generator. This will allow the later stages of the pipeline to expect a consistent datatype as input, string sequences: >>> two_grams = list(ngrams(tokens, 2)) >>> [" ".join(x) for x in two_grams]   51 Building your vocabulary with a tokenizer ['Thomas Jefferson', 'Jefferson began', 'began building', 'building Monticello', 'Monticello at', 'at the', 'the age', 'age of', 'of 26'] You might be able to sense a problem here. Here you take a few stop words and ignore them when you iterate through your token list: >>> stop_words = ['a', 'an', 'the', 'on', 'of', 'off', 'this', 'is'] >>> tokens = ['the', 'house', 'is', 'on', 'fire'] >>> tokens_without_stopwords = [x for x in tokens if x not in stop_words] >>> print(tokens_without_stopwords) ['house', 'fire'] You can see that some words carry a lot more meaning than others. Listing 2.8 NLTK list of stop words >>> import nltk >>> nltk.download('stopwords') >>> stop_words = nltk.corpus.stopwords.words('english') >>> len(stop_words) 153 >>> stop_words[:7] ['i', 'me', 'my', 'myself', 'we', 'our', 'ours'] >>> [sw for sw in stopwords if len(sw) == 1] ['i', 'a', 's', 't', 'd', 'm', 'o', 'y'] A document that dwells on the first person is pretty boring, and more importantly for you, has low information content.

2.2.5 Normalizing your vocabulary

But this will pre- vent advanced tokenizers that can split camel case words like ?WordPerfect,? ?FedEx,? or ?stringVariableName.?10 Maybe you want WordPerfect to be it?s own unique thing (token), or maybe you want to reminisce about a more perfect word processing era. 12 The trigram ?cup of joe? (https://en.wiktionary.org/wiki/cup_of_joe) is slang for ?cup of coffee.?   56 CHAPTER 2 Build your vocabulary (word tokenization) this approach prevents the blacksmith connotation of ?smith? being confused with the proper name ?Smith? in a sentence like ?A word smith had a cup of joe.? Even with this careful approach to case normalization, where you lowercase words only at the start of a sentence, you will still introduce capitalization errors for the rare proper nouns that start a sentence. The remainining seven steps are much more complicated because they have to deal with the complicated English spelling rules for the following: ? Step 1a??s? and ?es? endings ? Step 1b??ed,? ?ing,? and ?at? endings ? Step 1c??y? endings ? Step 2??nounifying? endings such as ?ational,? ?tional,? ?ence,? and ?able? ? Step 3?adjective endings such as ?icate,?b ?ful,? and ?alize? ? Step 4?adjective and noun endings such as ?ive,? ?ible,? ?ent,? and ?ism? ? Step 5a?stubborn ?e? endings, still hanging around ? Step 5b?trailing double consonants for which the stem will end in a single ?l? a) This is a trivially abbreviated version of Julia Menchavez?s implementation of porter-stemmer on GitHub (https://github.com/jedijulia/porter-stemmer/blob/master/stemmer.py). 61 Building your vocabulary with a tokenizer >>> nltk.download('wordnet') >>> from nltk.stem import WordNetLemmatizer >>> lemmatizer = WordNetLemmatizer() >>> lemmatizer.lemmatize("better") 'better' >>> lemmatizer.lemmatize("better", pos="a") 'good' >>> lemmatizer.lemmatize("good", pos="a") 'good' >>> lemmatizer.lemmatize("goods", pos="a") 'goods' >>> lemmatizer.lemmatize("goods", pos="n") 'good' >>> lemmatizer.lemmatize("goodness", pos="n") 'goodness' >>> lemmatizer.lemmatize("best", pos="a") 'best' The default part of speech is ?n? for noun. So the word ?best? doesn?t lemmatize to the same root as ?better.? This graph is also missing the connection between ?goodness? and ?good.? A Porter stemmer, on the other hand, would make this connection by blindly stripping off the ?ness? ending of all words: >>> stemmer.stem('goodness') 'good' USE CASES When should you use a lemmatizer or a stemmer?

2.3 Sentiment

:-) :-) :-).? And your algorithm should output -1 for text with negative sentiment like, ?Horrible! Some good and some bad things.? There are two approaches to sentiment analysis: ? A rule-based algorithm composed by a human ? A machine learning model learned from data by a machine The first approach to sentiment analysis uses human-designed rules, sometimes called heuristics, to measure sentiment. A common rule-based approach to sentiment analysis is to find keywords in the text and map each one to numerical scores or weights in a dictionary or ?mapping??a Python dict, for example. A machine learning sentiment model is trained to process input text and output a numerical value for the sentiment you are trying to measure, like positivity or spamminess or trolliness.

2.3.1 VADER?A rule-based sentiment analyzer

>>> from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer >>> sa = SentimentIntensityAnalyzer() >>> sa.lexicon { ... ':(': -1.9, ':)': 2.0, ... 'pls': 0.3, 'plz': 0.3, ... 'great': 3.1, ... } >>> [(tok, score) for tok, score in sa.lexicon.items() ... if " " in tok] [("('}{')", 1.6), ("can't stand", -2.0), ('fed up', -1.8), ('screwed up', -1.5)] >>> sa.polarity_scores(text=\ ... "Python is very readable and it's great for NLP.") {'compound': 0.6249, 'neg': 0.0, 'neu': 0.661, 'pos': 0.339} >>> sa.polarity_scores(text=\ A tokenizer better be good at dealing with punctuation and emoticons (emojis) for VADER to work well.

2.3.2 Naive Bayes

66 CHAPTER 2 Build your vocabulary (word tokenization) >>> movies.head().round(2) sentiment text id 1 2.27 The Rock is destined to be the 21st Century... 2 3.53 The gorgeously elaborate continuation of "... 3 -0.60 Effective but too tepid ... 4 1.47 If you sometimes like to go to the movies t... 5 1.73 Emerges as something rare, an issue movie t... >>> movies.describe().round(2) sentiment count 10605.00 mean 0.00 min -3.88 max 3.94 It looks like movies were rated on a scale from -4 to +4. >>> import pandas as pd >>> pd.set_option('display.width', 75) >>> from nltk.tokenize import casual_tokenize >>> bags_of_words = [] >>> from collections import Counter >>> for text in movies.text: ... bags_of_words.append(Counter(casual_tokenize(text))) >>> df_bows = pd.DataFrame.from_records(bags_of_words) >>> df_bows = df_bows.fillna(0).astype(int) >>> df_bows.shape (10605, 20756) >>> df_bows.head() ! " # $ % & ' ... zone zoning zzzzzzzzz ½ élan ? ? 0 0 0 0 0 0 0 4 ... 0 0 0 0 0 0 0 1 0 0 0 0 0 0 4 ... 0 0 0 0 0 0 0 2 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 4 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 >>> df_bows.head()[list(bags_of_words[0].keys())] The Rock is destined to be ... Van Damme or Steven Segal . >>> from sklearn.naive_bayes import MultinomialNB >>> nb = MultinomialNB() >>> nb = nb.fit(df_bows, movies.sentiment > 0) >>> movies['predicted_sentiment'] =\ ... nb.predict_proba(df_bows) * 8 - 4 >>> movies['error'] = (movies.predicted_sentiment - movies.sentiment).abs() >>> movies.error.mean().round(1) 2.4 >>> movies['sentiment_ispositive'] = (movies.sentiment > 0).astype(int) >>> movies['predicted_ispositiv'] = (movies.predicted_sentiment > 0).astype(int) >>> movies['''sentiment predicted_sentiment sentiment_ispositive\ ... predicted_ispositive'''.split()].head(8) sentiment predicted_sentiment sentiment_ispositive predicted_ispositive id 1 2.266667 4 1 1 2 3.533333 4 1 1 3 -0.600000 -4 0 0 4 1.466667 4 1 1 5 1.733333 4 1 1 6 2.533333 4 1 1 7 2.466667 4 1 1 8 1.266667 -4 1 0 >>> (movies.predicted_ispositive == ... movies.sentiment_ispositive).sum() / len(movies) 0.9344648750589345 The average absolute value of the prediction error (mean absolute error or MAE) is 2.4.

Summary

69 Summary ? Normalization and stemming consolidate words into groups that improve the ?recall? for search engines but reduce precision. ? Lemmatization and customized tokenizers like casual_tokenize() can improve precision and reduce information loss.

Math with words (TF-IDF vectors)

This chapter covers ? Counting words and term frequencies to analyze meaning ? Predicting word occurrence probabilities with Zipf?s Law ? Vector representation of words and how to start using them ? Finding relevant documents from a corpus using inverse document frequencies ? Estimating the similarity of pairs of documents with cosine similarity and Okapi BM25 Having collected and counted words (tokens), and bucketed them into stems or lem- mas, it?s time to do something interesting with them.

3.1 Bag of words

A Python dictionary serves this purpose nicely, and because you want to count the words as well, you can use Counter, as you did in previous chapters: >>> from collections import Counter >>> bag_of_words = Counter(tokens) >>> bag_of_words Counter({'the': 4, 'faster': 3, 'harry': 2, 'got': 1, 'to': 1, 'store': 1, ',': 3, 'would': 1,   73 Bag of words 'get': 1, 'home': 1, '. The Counter object has a handy method, most_common, for just this purpose: >>> bag_of_words.most_common(4) [('the', 4), (',', 3), ('faster', 3), ('harry', 2)] By default, most_common() lists all tokens from most frequent to least, but you?ve limited the list to the top four here. Let?s calculate the term frequency of ?harry? from the Counter object (bag_ of_words) you defined above: The number of unique tokens from your original source >>> times_harry_appears = bag_of_words['harry'] >>> num_unique_words = len(bag_of_words) 1 However, normalized frequency is really a probability, so it should probably not be called frequency. ?Wikipedia Then you?ll assign the text to a variable: >>> from collections import Counter >>> from nltk.tokenize import TreebankWordTokenizer >>> tokenizer = TreebankWordTokenizer() >>> from nlpia.data.loaders import kite_text >>> tokens = tokenizer.tokenize(kite_text.lower()) >>> token_counts = Counter(tokens) >>> token_counts Counter({'the': 26, 'a': 20, 'kite': 16, ',': 15, ...}) kite_text = ?A kite is traditionally ?? as above The TreebankWordTokenizer returns 'kite.' So let?s ditch them for now: >>> import nltk >>> nltk.download('stopwords', quiet=True) True >>> stopwords = nltk.corpus.stopwords.words('english') >>> tokens = [x for x in tokens if x not in stopwords] >>> kite_counts = Counter(tokens) >>> kite_counts Counter({'kite': 16, 'traditionally': 1, 'tethered': 2, 'heavier-than-air': 1, 'craft': 2, 'wing': 5, 'surfaces': 1, 'react': 1, 2 See the web page titled ?spaCy 101: Everything you need to know? (https://spacy.io/usage/spacy-101 #annotations-token).

3.2 Vectorizing

You can do this quickly with >>> document_vector = [] >>> doc_length = len(tokens) >>> for key, value in kite_counts.most_common(): ... document_vector.append(value / doc_length) >>> document_vector [0.07207207207207207, 0.06756756756756757, 0.036036036036036036, ..., 0.0045045045045045045] This list, or vector, is something you can do math on directly. First, let?s look at your lexicon for this corpus containing three documents: >>> doc_tokens = [] >>> for doc in docs: ... doc_tokens += [sorted(tokenizer.tokenize(doc.lower()))] >>> len(doc_tokens[0]) 17 >>> all_doc_tokens = sum(doc_tokens, []) >>> len(all_doc_tokens) 33 >>> lexicon = sorted(set(all_doc_tokens)) >>> len(lexicon) 18 >>> lexicon [',', '. ', 'and', 'as', 'faster', 'get', 'got', 'hairy', 'harry', 'home', 'is', 'jill', 'not', 'store', 'than',   78 CHAPTER 3 Math with words (TF-IDF vectors) 'the', 'to', 'would'] Each of your three document vectors will need to have 18 values, even if the docu- ment for that vector doesn?t contain all 18 words in your lexicon. Some of those token counts in the vector will be zeros, which is what you want: >>> from collections import OrderedDict >>> zero_vector = OrderedDict((token, 0) for token in lexicon) >>> zero_vector OrderedDict([(',', 0), ('. ', 0), ('and', 0), ('as', 0), ('faster', 0), ('get', 0), ('got', 0), ('hairy', 0), ('harry', 0), ('home', 0), ('is', 0), ('jill', 0), ('not', 0), ('store', 0), ('than', 0), ('the', 0), ('to', 0), ('would', 0)]) Now you?ll make copies of that base vector, update the values of the vector for each document, and store them in an array: copy.copy() creates an independent copy, a separate instance of your zero vector, rather than reusing a reference (pointer) to the original object?s memory location. >>> import copy >>> doc_vectors = [] >>> for doc in docs: ... vec = copy.copy(zero_vector) ... tokens = tokenizer.tokenize(doc.lower()) ... token_counts = Counter(tokens) ... for key, value in token_counts.items(): ... vec[key] = value / len(lexicon) ... doc_vectors.append(vec) You have three vectors, one for each document.

3.2.1 Vector spaces

Term frequency vectors in 2D space 0.20 doc_0 ~(0.1, 0.17) 0.15 TF of ?faster? 0.10 ?2 doc_0 ~(0.056, 0.056) 0.05 ?1 doc_2 ~(0.056, 0) 0.00 ?0.05 0.00 0.05 0.10 0.15 0.20 TF of ?harry? Figure 3.3 2D thetas ?0.05   82 CHAPTER 3 Math with words (TF-IDF vectors) In Python this would be a.dot(b) == np.linalg.norm(a) * np.linalg.norm(b) / np.cos(theta) Solving this relationship for cos(theta), you can derive the cosine similarity using cos ? = A · B |A| |B| Or you can do it in pure Python without numpy, as in the following listing.

3.3 Zipf?s Law

85 Zipf?s Law >>> nltk.download('brown') >>> from nltk.corpus import brown >>> brown.words()[:10] ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an', 'investigation', 'of'] >>> brown.tagged_words()[:5] [('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-TL'), ('Jury', 'NN-TL')] >>> len(brown.words()) 1161192 The Brown corpus is about 3MB. ', ... ':', ';', '``', "''", '(', ')', '[', ']')) >>> word_list = (x.lower() for x in brown.words() if x not in puncs) >>> token_counts = Counter(word_list) >>> token_counts.most_common(20) [('the', 69971), ('of', 36412), ('and', 28853), ('to', 26158), ('a', 23195), ('in', 21337), ('that', 10594), ('is', 10109), ('was', 9815), ('he', 9548), ('for', 9489), ('it', 8760), ('with', 7289), ('as', 7253), ('his', 6996), ('on', 6741), ('be', 6377), ('at', 5372), ('by', 5306), ('i', 5164)] A quick glance shows that the word frequencies in the Brown corpus follow the logarithmic relationship Zipf predicted.

3.4 Topic modeling

Wikipedia First let?s get the total word count for each document in your corpus, intro_doc and history_doc: >>> from nlpia.data.loaders import kite_text, kite_history >>> kite_intro = kite_text.lower() >>> intro_tokens = tokenizer.tokenize(kite_intro) >>> kite_history = kite_history.lower() >>> history_tokens = tokenizer.tokenize(kite_history) >>> intro_total = len(intro_tokens) >>> intro_total 363 >>> history_total = len(history_tokens) >>> history_total 297 ?A kite is traditionally ? ?? ?a kite is traditionally ?? Now with a couple tokenized kite documents in hand, let?s look at the term frequency of ?kite? in each document. You?ll store the TFs you find in two dictionaries, one for each document: >>> intro_tf = {} >>> history_tf = {} >>> intro_counts = Counter(intro_tokens) >>> intro_tf['kite'] = intro_counts['kite'] / intro_total >>> history_counts = Counter(history_tokens) >>> history_tf['kite'] = history_counts['kite'] / history_total >>> 'Term Frequency of "kite" in intro is: {:.4f}'.format(intro_tf['kite']) 'Term Frequency of "kite" in intro is: 0.0441' >>> 'Term Frequency of "kite" in history is: {:.4f}'\ ... .format(history_tf['kite']) 'Term Frequency of "kite" in history is: 0.0202' Okay, you have a number twice as large as the other. First, let?s see how those num- bers relate to some other word, say ?and?: >>> intro_tf['and'] = intro_counts['and'] / intro_total >>> history_tf['and'] = history_counts['and'] / history_total >>> print('Term Frequency of "and" in intro is: {:.4f}'\ ... .format(intro_tf['and'])) Term Frequency of "and" in intro is: 0.0275 >>> print('Term Frequency of "and" in history is: {:.4f}'\ ... .format(history_tf['and'])) Term Frequency of "and" in history is: 0.0303   88 CHAPTER 3 Math with words (TF-IDF vectors) Great! Let?s use this ?rarity? measure to weight the term fre- quencies: >>> num_docs_containing_and = 0 >>> for doc in [intro_tokens, history_tokens]: ... if 'and' in doc: ... num_docs_containing_and += 1 similarly for ?kite? and ?China? And let?s grab the TF of ?China? in the two documents: >>> intro_tf['china'] = intro_counts['china'] / intro_total >>> history_tf['china'] = history_counts['china'] / history_total And finally, the IDF for all three.

3.4.1 Return of Zipf

Let?s say, though, you have a corpus of 1 million documents (maybe you?re baby-Google), someone searches for the word ?cat,? and in your 1 mil- lion documents you have exactly 1 document that contains the word ?cat.? The raw IDF of this is 1,000,000 / 1 = 1,000,000 Let?s imagine you have 10 documents with the word ?dog? in them. You?ll want to take the log of the term frequency as well.10 The base of log function isn?t important, because you only want to make the fre- quency distribution uniform, not to scale it within a particular numerical range.11 If you use a base 10 log function, you?ll get: search: cat idf = log(1,000,000/1) = 6 search: dog idf = log(1,000,000/10) = 5 So now you?re weighting the TF results of each more appropriately to their occur- rences in language, in general.

3.4.2 Relevance ranking

>>> document_tfidf_vectors = [] >>> for doc in docs: ... vec = copy.copy(zero_vector) 91 Topic modeling ... tokens = tokenizer.tokenize(doc.lower()) ... token_counts = Counter(tokens) ... ... for key, value in token_counts.items(): ... docs_containing_key = 0 ... for _doc in docs: ... if key in _doc: ... docs_containing_key += 1 ... tf = value / len(lexicon) ... if docs_containing_key: ... idf = len(docs) / docs_containing_key ... else: ... idf = 0 ... vec[key] = tf * idf ... document_tfidf_vectors.append(vec) With this setup, you have K-dimensional vector representation of each document in the corpus. >>> tokens = tokenizer.tokenize(query.lower()) >>> token_counts = Counter(tokens) >>> for key, value in token_counts.items(): ... docs_containing_key = 0 ... for _doc in documents: ... if key in _doc.lower(): ... docs_containing_key += 1 ... if docs_containing_key == 0: ... continue You didn?t find that token in the lexicon, so go to the next key. 92 CHAPTER 3 Math with words (TF-IDF vectors) ... tf = value / len(tokens) ... idf = len(documents) / docs_containing_key ... query_vec[key] = tf * idf >>> cosine_sim(query_vec, document_tfidf_vectors[0]) 0.5235048549676834 >>> cosine_sim(query_vec, document_tfidf_vectors[1]) 0.0 >>> cosine_sim(query_vec, document_tfidf_vectors[2]) 0.0 you can safely say document 0 has the most relevance for your query!

3.4.4 Alternatives

Table 3.1 lists some of the ways you can normalize and smooth your term fre- quency weights.17 Table 3.1 Alternative TF-IDF normalization approaches (Molino 2017) Scheme Definition wij = fij None N nj wij = log(fij) × log( ) TF-IDF N fj wij = log(fij) × log( ) TF-ICF fij N ?nj +0.5 wij = log Okapi BM25 fj fj j fij +0.5 0.5+1.5× +fij fij N (0.5+0.5×) log( ) nj maxf wij = ATC ? fij N ? [(0.5+0.5×) log( )]2 nj maxf N i=1 N nj (log(fij)+1.0)log( ) wij = LTU j fj 0.8+0.2×fj × P(tij |cj) wij = log MI P(tij)P(cj) wij = max(0, MI) PosMI P(tij |cj)?P(tij)P(cj) wij = T-Test ? P(tij)P(cj) x2 See section 4.3.5 of From Distributional to Semantic Similarity (https://www.era.lib.ed.ac.uk/bitstream/handle/1842/563/ IP030023.pdf#subsection.4.3.5) by James Richard Curran fij ×f fi ×fj wij = Lin98a nj N wij = ?1 × log Lin98b log fij +1 log nj +1 wij = Gref94 Search engines (information retrieval systems) match keywords (terms) between que- ries and documents in a corpus.

3.4.6 What?s next

Numbers firmly in hand, in the next chapter you?ll refine those numbers to try to represent the meaning or topic of natural language text instead of only its words.

Summary

? Term frequencies must be weighted by their inverse document frequency to ensure the most important, most meaningful words are given the heft they deserve. ? Zipf?s law can help you predict the frequencies of all sorts of things, including words, characters, and people. ? The rows of a TF-IDF term document matrix can be used as a vector representa- tion of the meanings of those individual words to create a vector space model of word semantics. ? Euclidean distance and similarity between pairs of high dimensional vectors doesn?t adequately represent their similarity for most NLP applications. ? Cosine distance, the amount of ?overlap? between vectors, can be calculated efficiently by just multiplying the elements of normalized vectors together and summing up those products.

Finding meaning in word counts (semantic analysis)

This chapter covers ? Analyzing semantics (meaning) to create topic vectors ? Semantic search using the similarity between topic vectors ? Scalable semantic analysis and semantic search for large corpora ? Using semantic components (topics) as features in your NLP pipeline ? Navigating high-dimensional vector spaces You?ve learned quite a few natural language processing tricks.

4.1 From word counts to topic scores

Formal NLP texts such as the NLP bible by Jurafsky and Martin (https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf#chapter.15) use ?topic vector.? Others, like the authors of Semantic Vector Encoding and Similarity Search (https://arxiv.org/pdf/1706.00957.pdf), use the term ?semantic vector.?

4.1.2 Topic vectors

We call these compact meaning vectors ?word-topic vectors.? We call the document meaning vectors ?document-topic vectors.? You can call either of these vectors ?topic vectors,? as long as you?re clear on what the topic vectors are for, words or documents. You?ll have a topic vector for each word in your vocabulary, and you can use these word topic vectors to compute the topic vector for any document that uses some of those words.

4.1.3 Thought experiment

102 CHAPTER 4 Finding meaning in word counts (semantic analysis) >>> topic['petness'] = (.3 * tfidf['cat'] +\ ... .3 * tfidf['dog'] +\ ... 0 * tfidf['apple'] +\ ... 0 * tfidf['lion'] -\ ... .2 * tfidf['NYC'] +\ ... .2 * tfidf['love']) >>> topic['animalness'] = (.1 * tfidf['cat'] +\ ... .1 * tfidf['dog'] -\ ... .1 * tfidf['apple'] +\ ... .5 * tfidf['lion'] +\ ... .1 * tfidf['NYC'] -\ ... .1 * tfidf['love']) >>> topic['cityness'] = (0 * tfidf['cat'] -\ ... .1 * tfidf['dog'] +\ ... .2 * tfidf['apple'] -\ ... .1 * tfidf['lion'] +\ ... .5 * tfidf['NYC'] +\ ... .1 * tfidf['love']) ?Hand-crafted? weights (.3, .3, 0, 0, -.2, .2) are multiplied by imaginary tfidf values to create topic vectors for your imaginary random document. These vectors of weights would be your word vectors for your six words: >>> word_vector = {} >>> word_vector['cat'] = .3*topic['petness'] +\ ... .1*topic['animalness'] +\ ... 0*topic['cityness'] >>> word_vector['dog'] = .3*topic['petness'] +\ ... .1*topic['animalness'] -\ ... .1*topic['cityness'] >>> word_vector['apple']= 0*topic['petness'] -\ ... .1*topic['animalness'] +\ ... .2*topic['cityness'] >>> word_vector['lion'] = 0*topic['petness'] +\ ... .5*topic['animalness'] -\ ... .1*topic['cityness'] 104 CHAPTER 4 Finding meaning in word counts (semantic analysis) >>> word_vector['NYC'] = -.2*topic['petness'] +\ ... .1*topic['animalness'] +\ ... .5*topic['cityness'] >>> word_vector['love'] = .2*topic['petness'] -\ ... .1*topic['animalness'] +\ ... .1*topic['cityness'] These six topic vectors (shown in Figure 4.1), one for each word, represent the mean- ings of your six words as 3D vectors.

4.1.4 An algorithm for scoring topics

And the video shows a particular algorithm, called SVD, that reorders the words and topics, to put as much of the ?weight? as possible along the diagonal. LSA is an algorithm to analyze your TF-IDF matrix (table of TF-IDF vectors) to gather up words into topics. LSA also optimizes these topics to maintain diversity in the topic dimensions; when you use these new topics instead of the original words, you still capture much of the meaning (semantics) of the documents. If you?ve done machine learning on images or other high- dimensional data, you may have run across a technique called principal component analysis (PCA). PCA, however, is what you say when you?re reducing the dimensionality of images or other tables of numbers, rather than bag-of-words vectors or TF-IDF vectors.

4.1.5 An LDA classifier

'*j) for (i,j) in\ ... zip(range(len(sms)), sms.spam)] >>> sms = pd.DataFrame(sms.values, columns=sms.columns, index=index) >>> sms['spam'] = sms.spam.astype(int) >>> len(sms) 4837 >>> sms.spam.sum() 638 >>> sms.head(6) spam text sms0 0 Go until jurong point, crazy.. So you have 4,837 SMS messages, and 638 of them are labeled with the binary class label ?spam.? Now let?s do our tokenization and TF-IDF vector transformation on all these SMS messages: >>> from sklearn.feature_extraction.text import TfidfVectorizer >>> from nltk.tokenize.casual import casual_tokenize >>> tfidf_model = TfidfVectorizer(tokenizer=casual_tokenize) >>> tfidf_docs = tfidf_model.fit_transform(\ ... raw_documents=sms.text).toarray() >>> tfidf_docs.shape (4837, 9232) >>> sms.spam.sum() 638 The nltk.casual_tokenizer gave you 9,232 words in your vocabulary. >>> mask = sms.spam.astype(bool).values >>> spam_centroid = tfidf_docs[mask].mean(axis=0) >>> ham_centroid = tfidf_docs[~mask].mean(axis=0) Because your TF-IDF vectors are row vectors, you need to make sure numpy computes the mean for each column independently using axis=0. Now you can subtract one centroid from the other to get the line between them: >>> spamminess_score = tfidf_docs.dot(spam_centroid -\ ... ham_centroid) >>> spamminess_score.round(2) array([-0.01, -0.02, 0.04, ..., -0.01, -0. , 0.]) The sklearnMinMaxScaler can do that for you: >>> from sklearn.preprocessing import MinMaxScaler >>> sms['lda_score'] = MinMaxScaler().fit_transform(\ ... spamminess_score.reshape(-1,1)) >>> sms['lda_predict'] = (sms.lda_score > .5).astype(int) >>> sms['spam lda_predict lda_score'.split()].round(2).head(6) spam lda_predict lda_score sms0 0 0 0.23 sms1 0 0 0.18 sms2! This shows you the SMS messages that it labeled as spam that weren?t spam at all (false positives), and the ones that were labeled as ham that should have been labeled spam (false negatives): >>> from pugnlp.stats import Confusion >>> Confusion(sms['spam lda_predict'.split()]) lda_predict 0 1 9 Actually, a Naive Bayes classifier and a logistic regression model are both equivalent to this simple LDA model.

4.2 Latent semantic analysis

Latent semantic analysis is a mathematical technique for finding the ?best? way to linearly transform (rotate and stretch) any set of NLP vectors, like your TF-IDF vectors or bag-of-words vectors.

4.2.1 Your thought experiment made real

Listing 4.2 Topic-word matrix for LSA on 16 short sentences about cats, dogs, and NYC >>> from nlpia.book.examples.ch04_catdog_lsa_3x6x16\ ... import word_topic_vectors >>> word_topic_vectors.T.round(1) cat dog apple lion nyc love top0 -0.6 -0.4 0.5 -0.3 0.4 -0.1 top1 -0.1 -0.3 -0.4 -0.1 0.1 0.8 top2 -0.3 0.8 -0.1 -0.5 0.0 0.1 The rows in this topic-word matrix are the ?word topic vectors? or just ?topic vectors? for each word. SVD, from your linear algebra class, is what LSA uses to create vectors like those in the word-topic matrices just discussed.17 Finally some NLP in action: we now show you how a machine is able to ?play Mad Libs? to understand words.

4.3 Singular value decomposition

Let?s start with a corpus of only 11 documents and a vocabulary of 6 words, similar to what you had in mind for your thought experiment:18 >>> from nlpia.book.examples.ch04_catdog_lsa_sorted\ ... import lsa_models, prettify_tdm >>> bow_svd, tfidf_svd = lsa_models() >>> prettify_tdm(**bow_svd) cat dog apple lion nyc love text 0 1 1 NYC is the Big Apple. You?ll first use SVD on the term-document matrix (the transpose of the document- term matrix above), but it works on TF-IDF matrices or any other vector space model: >>> tdm = bow_svd['tdm'] >>> tdm 0 1 2 3 4 5 6 7 8 9 10 cat 0 0 0 0 0 0 1 1 1 0 1 dog 0 0 0 0 0 0 0 0 0 0 1 apple 1 1 0 1 1 1 0 0 0 0 0 lion 0 0 0 0 0 0 0 1 0 0 0 nyc 1 1 1 1 1 0 0 0 0 1 0 love 0 0 1 0 0 0 0 0 1 1 0 SVD is an algorithm for decomposing any matrix into three ?factors,? three matrices that can be multiplied together to recreate the original matrix. (1988),23 Salton and Lesk (1965),24 and Deer- wester (1990).25 Here?s what SVD (the heart of LSA) looks like in math notation: Wmxn ? Umxp Spxp Vpxn T 19 This is equivalent to the square root of the dot product of two columns (term-document occurrence vectors), but SVD provides you additional information that computing the correlation directly wouldn?t provide.

4.3.1 U?left singular vectors

Listing 4.3 Umxp >>> import numpy as np >>> U, s, Vt = np.linalg.svd(tdm) >>> import pandas as pd >>> pd.DataFrame(U, index=tdm.index).round(2) 0 1 2 3 4 5 cat -0.04 0.83 -0.38 -0.00 0.11 -0.38 dog -0.00 0.21 -0.18 -0.71 -0.39 0.52 apple -0.62 -0.21 -0.51 0.00 0.49 0.27 lion -0.00 0.21 -0.18 0.71 -0.39 0.52 nyc -0.75 -0.00 0.24 -0.00 -0.52 -0.32 love -0.22 0.42 0.69 0.00 0.41 0.37 You?re reusing the tdm term- document matrix from the earlier code sections.

4.3.2 S?singular values

Listing 4.4 Spxp >>> s.round(1) array([3.1, 2.2, 1.8, 1. , 0.8, 0.5]) >>> S = np.zeros((len(U), len(Vt))) >>> pd.np.fill_diagonal(S, s) >>> pd.DataFrame(S).round(1) 0 1 2 3 4 5 6 7 8 9 10 0 3.1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1 0.0 2.2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 2 0.0 0.0 1.8 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 3 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 4 0.0 0.0 0.0 0.0 0.8 0.0 0.0 0.0 0.0 0.0 0.0 5 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 Like the U matrix, your S matrix for your 6-word, 6-topic corpus has six rows (p).

4.3.4 SVD matrix orientation

If you?ve done machine learning with natural language documents before, you may notice that your term-document matrix is ?flipped? (transposed) relative to what you?re used to seeing in scikit-learn and other packages. In the Naive Bayes sentiment model at the end of chapter 2, and the TF-IDF vectors of chapter 3, you created your training set as a document-term matrix. But when you do the SVD linear algebra directly, your matrix needs to be transposed into term-document format.29 29 Actually, within the sklearn.PCA model they leave the document-term matrix unflipped and just flip the SVD matrix math operations.

4.3.5 Truncating the topics

122 CHAPTER 4 Finding meaning in word counts (semantic analysis)   Listing 4.6 Term-document matrix reconstruction error >>> err = [] >>> for numdim in range(len(s), 0, -1): ... S[numdim - 1, numdim - 1] = 0 ... reconstructed_tdm = U.dot(S).dot(Vt) ... err.append(np.sqrt(((\ ... reconstructed_tdm - tdm).values.flatten() ** 2).sum() ... / np.product(tdm.shape))) >>> np.array(err).round(2) array([0.06, 0.12, 0.17, 0.28, 0.39, 0.55]) When you reconstruct a term-document matrix for your 11 documents using the sin- gular vectors, the more you truncate, the more the error grows.

4.4 Principal component analysis

This helps spread out your data and makes any optimization algorithm less likely to get lost in ?half pipes? or ?rivers? of your data that can arise when features in your dataset are correlated with each other.33 Before you apply PCA to real-world, high-dimensional NLP data, let?s take a step back and look at a more visual representation of what PCA and SVD do. 124 CHAPTER 4 Finding meaning in word counts (semantic analysis) You?re going to start with a set of real-world 3D vectors, rather than 10,000+ dimen- sional document-word vectors.

4.4.1 PCA on 3D vectors

>>> df = get_data('pointcloud').sample(1000) >>> pca = PCA(n_components=2) >>> df2d = pd.DataFrame(pca.fit_transform(df), columns=list('xy')) >>> df2d.plot(kind='scatter', x='x', y='y') >>> plt.show() If you run this script, the orientation of your 2D projection may randomly ?flip? left to right, but it never tips or twists to a new angle.

4.4.2 Stop horsing around and get back to NLP

>>> sms = get_data('sms-spam')   127 Principal component analysis >>> index = ['sms{}{}'.format(i, '! '*j) ? for (i,j) in zip(range(len(sms)), sms.spam)] >>> sms.index = index >>> sms.head(6) You?re adding an exclamation mark to the sms message index numbers to make them easier to spot. 1 FreeMsg Hey there darling it's been 3 week's n... Now you can calculate the TF-IDF vectors for each of these messages: >>> from sklearn.feature_extraction.text import TfidfVectorizer >>> from nltk.tokenize.casual import casual_tokenize >>> tfidf = TfidfVectorizer(tokenizer=casual_tokenize) >>> tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray() >>> len(tfidf.vocabulary_) 9232 This centers your vectorized documents (BOW vectors) by subtracting the mean. >>> tfidf_docs = pd.DataFrame(tfidf_docs) >>> tfidf_docs = tfidf_docs - tfidf_docs.mean() >>> tfidf_docs.shape (4837, 9232) >>> sms.spam.sum() 638 The .shape attribute tells you the length of each of the dimensions for any numpy array.

4.4.3 Using PCA for SMS message semantic analysis

You?ve already seen it in action wran- gling 3D horses into a 2D pen; now let?s wrangle your dataset of 9,232-D TF-IDF vec- tors into 16-D topic vectors: >>> from sklearn.decomposition import PCA >>> pca = PCA(n_components=16) >>> pca = pca.fit(tfidf_docs) >>> pca_topic_vectors = pca.transform(tfidf_docs) >>> columns = ['topic{}'.format(i) for i in range(pca.n_components)] >>> pca_topic_vectors = pd.DataFrame(pca_topic_vectors, columns=columns,\ ... index=index) >>> pca_topic_vectors.round(3).head(6) topic0 topic1 topic2 ... topic13 topic14 topic15 sms0 0.201 0.003 0.037 ... -0.026 -0.019 0.039 sms1 0.404 -0.094 -0.078 ... -0.036 0.047 -0.036 35 More on overfitting and generalization in appendix D.   129 Principal component analysis sms2!

4.4.4 Using truncated SVD for SMS message semantic analysis

This is a more direct approach to LSA that bypasses the scikit-learn PCA model so you can see what?s going on inside the PCA wrapper. It can handle sparse matrices, so if you?re working with large datasets you?ll want to use TruncatedSVD instead of PCA anyway.

4.4.5 How well does LSA work for spam classification?

-0.3 -0.1 0.4 -0.1 -0.2 0.4 -0.2 0.4 0.3 1.0 Reading down the ?sms0? column (or across the ?sms0? row), the cosine similarity between ?sms0? and the spam messages (?sms2!,? ?sms5!,? ?sms8!,? ?sms9!?) is signifi- cantly negative.

4.5 Latent Dirichlet allocation (LDiA)

LSA should be your first choice for most topic modeling, semantic search, or content-based recommenda- tion engines.38 Its math is straightforward and efficient, and it produces a linear transformation that can be applied to new batches of natural language without train- ing and with little loss in accuracy. LDiA does a lot of the things you did to create your topic models with LSA (and SVD under the hood), but unlike LSA, LDiA assumes a Dirichlet distribution of word frequencies. LDiA creates a semantic vector space model (like your topic vectors) using an approach similar to how your brain worked during the thought experiment earlier in the chapter. This makes an LDiA topic model much easier to understand, because the words assigned to topics and topics assigned to documents tend to make more sense than for LSA. LDiA assumes that each document is a mixture (linear combination) of some arbi- trary number of topics that you select when you begin training the LDiA model. See ?Comparing LDA and LSA Topic Models for Content-Based Movie Recommendation Systems? by Sonia Bergamaschi and Laura Po (https://www.dbgroup .unimo.it/~po/pubs/LNBI_2015.pdf).

4.5.1 The LDiA idea

The two rolls of the dice represent the Number of words to generate for the document (Poisson distribution) 1 Number of topics to mix together for the document (Dirichlet distribution) 2 After it has these two numbers, the hard part begins, choosing the words for a docu- ment. So all this machine needs is a single parameter for that Poisson distribution (in the dice roll from step 1) that tells it what the ?average? document length should be, and a couple more parameters to define that Dirichlet distribution that sets up the num- ber of topics. For example, for step 1, they could calculate the mean number of words (or n-grams) in all the bags of words for the documents in their corpus; something like this: >>> total_corpus_len = 0 >>> for document_text in sms.text: ... total_corpus_len += len(casual_tokenize(document_text)) >>> mean_document_len = total_corpus_len / len(sms) >>> round(mean_document_len, 2) 21.35 Or, in a one-liner >>> sum([len(casual_tokenize(t)) for t in sms.text]) * 1.

4.5.2 LDiA topic model for SMS messages

Here?s an easy way to compute BOW vectors in scikit-learn: >>> from sklearn.feature_extraction.text import CountVectorizer >>> from nltk.tokenize import casual_tokenize >>> np.random.seed(42) >>> counter = CountVectorizer(tokenizer=casual_tokenize) >>> bow_docs = pd.DataFrame(counter.fit_transform(raw_documents=sms.text)\ ... .toarray(), index=index) >>> column_nums, terms = zip(*sorted(zip(counter.vocabulary_.values(),\ ... counter.vocabulary_.keys()))) >>> bow_docs.columns = terms Let?s double-check that your counts make sense for that first SMS message labeled ?sms0?: >>> sms.loc['sms0'].text 'Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...' >>> bow_docs.loc['sms0'][bow_docs.loc['sms0'] > 0].head() , 1 .. 1 ... 2 amore 1 available 1 Name: sms0, dtype: int64 And here?s how to use LDiA to create topic vectors for your SMS corpus: >>> from sklearn.decomposition import LatentDirichletAllocation as LDiA >>> ldia = LDiA(n_components=16, learning_method='batch') >>> ldia = ldia.fit(bow_docs) >>> ldia.components_.shape (16, 9232) LDiA takes a bit longer than PCA or SVD, especially for a large number of topics and a large number of words in your corpus. And let?s see how they are different from the topic vectors produced by SVD and PCA for those same documents: >>> ldia16_topic_vectors = ldia.transform(bow_docs) >>> ldia16_topic_vectors = pd.DataFrame(ldia16_topic_vectors,\ ... index=index, columns=columns) >>> ldia16_topic_vectors.round(2).head() topic0 topic1 topic2 ... topic13 topic14 topic15 sms0 0.00 0.62 0.00 ... 0.00 0.00 0.00 sms1 0.01 0.01 0.01 ... 0.01 0.01 0.01 sms2!

4.5.3 LDiA + LDA = spam classifier

You?ll use your LDiA topic vectors to train an LDA model again (like you did with your PCA topic vectors): >>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA >>> X_train, X_test, y_train, y_test = ? train_test_split(ldia16_topic_vectors, sms.spam, test_size=0.5, ? random_state=271828) >>> lda = LDA(n_components=1) >>> lda = lda.fit(X_train, y_train) >>> sms['ldia16_spam'] = lda.predict(ldia16_topic_vectors) >>> round(float(lda.score(X_test, y_test)), 2) 0.94 Your ldia_topic_vectors matrix has a determinant close to zero so you will likely get the warning ?Variables are collinear.? This can happen with a small corpus when using LDiA because your topic vectors have a lot of zeros in them and some of your messages could be reproduced as a linear combination of the other message topics. This is where the generalization of LDiA and PCA should help: >>> from sklearn.feature_extraction.text import TfidfVectorizer >>> from nltk.tokenize.casual import casual_tokenize >>> tfidf = TfidfVectorizer(tokenizer=casual_tokenize) >>> tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray() >>> tfidf_docs = tfidf_docs - tfidf_docs.mean(axis=0) >>> X_train, X_test, y_train, y_test = train_test_split(tfidf_docs,\ ... sms.spam.values, test_size=0.5, random_state=271828) >>> lda = LDA(n_components=1) >>> lda = lda.fit(X_train, y_train) >>> round(float(lda.score(X_train, y_train)), 3) 1.0 >>> round(float(lda.score(X_test, y_test)), 3) 0.748 You?re going to ?pretend? that there is only one topic in all the SMS messages, because you?re only interested in a scalar score Fitting an LDA model to all these thousands of features for the ?spamminess? topic.

4.5.4 A fairer comparison: 32 LDiA topics

Let?s try 32 topics (components): >>> ldia32 = LDiA(n_components=32, learning_method='batch') >>> ldia32 = ldia32.fit(bow_docs) >>> ldia32.components_.shape (32, 9232) Now let?s compute your new 32-D topic vectors for all your documents (SMS messages): >>> ldia32_topic_vectors = ldia32.transform(bow_docs) >>> columns32 = ['topic{}'.format(i) for i in range(ldia32.n_components)] >>> ldia32_topic_vectors = pd.DataFrame(ldia32_topic_vectors, index=index,\ ... columns=columns32) >>> ldia32_topic_vectors.round(2).head() topic0 topic1 topic2 ... topic29 topic30 topic31 sms0 0.00 0.5 0.0 ... 0.0 0.0 0.0 sms1 0.00 0.0 0.0 ... 0.0 0.0 0.0 sms2! And here?s your LDA model (classifier) training, this time using 32-D LDiA topic vectors: >>> X_train, X_test, y_train, y_test = ? train_test_split(ldia32_topic_vectors, sms.spam, test_size=0.5, ? random_state=271828) >>> lda = LDA(n_components=1) >>> lda = lda.fit(X_train, y_train) >>> sms['ldia32_spam'] = lda.predict(ldia32_topic_vectors) >>> X_train.shape (2418, 32) >>> round(float(lda.score(X_train, y_train)), 3) 0.924 >>> round(float(lda.score(X_test, y_test)), 3) 0.927 .shape is another way to check the number of dimensions in your topic vectors.

4.6 Distance and similarity

Some of these commonly used examples may be familiar from geometry class or linear algebra, but many others are probably new to you: ? Euclidean or Cartesian distance, or root mean square error (RMSE): 2-norm or L2 ? Squared Euclidean distance, sum of squares distance (SSD): L22 ? Cosine or angular or projected distance: normalized dot product ? Minkowski distance: p-norm or Lp ? Fractional distance, fractional norm: p-norm or Lp for 0 < p < 1 ? City block, Manhattan, or taxicab distance; sum of absolute distance (SAD): 1-norm or L1 ? Jaccard distance, inverse set similarity ? Mahalanobis distance ? Levenshtein or edit distance The variety of ways to calculate distance is a testament to how important it is. In addition to the pairwise distance implementations in Scikit-learn, many others are used in math- ematics specialties such as topology, statistics, and engineering.45 For reference, the fol- lowing listing shows the distances you can find in the sklearn.metrics.pairwise module.46 Listing 4.7 Pairwise distances available in sklearn 'cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan', 'braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule' 45 See Math.NET Numerics for more distance metrics (https://numerics.mathdotnet.com/Distance.html). The angular distance between two vectors is often computed as a fraction of the maximum possible angular separation between two vectors, which is 180 degrees or pi radians.47 As a result, cosine similarity and distance are the reciprocal of each other: >>> import math >>> angular_distance = math.acos(cosine_similarity) / math.pi >>> distance = 1.

4.7 Steering with feedback

Our unsupervised learning of these feature (topic) extraction models didn?t have any data about how ?close? the topic vectors should be to each other. We didn?t allow any ?feedback? about where the topic vectors ended up, or how they were related to each other. By adjusting the distance scores reported to clustering and embedding algorithms, you can ?steer? your vectors so that they minimize some cost function. But we learned quickly that we got much better results when we started ?steering? our topic vectors based on feedback from candidates and account managers responsible for helping them find a job. One way to do this is to calculate the mean difference between your two centroids (like you did for LDA) and add some portion of this ?bias? to all the resume or job description vectors. Topics such as beer on tap at lunch might 49 See the Wikipedia article titled ?Measure (mathematics)? (https://en.wikipedia.org/wiki/Measure_ (mathematics)). 50 See the web page titled ?Superpixel Graph Label Transfer with Learned Distance Metric? (http:// users.cecs.anu.edu.au/~sgould/papers/eccv14-spgraph.pdf).

4.7.1 Linear discriminant analysis

Because you only have a ?spaminess? topic to train on, let?s see how accurate your 1D topic model can be at classifying spam SMS messages: >>> lda = LDA(n_components=1) >>> lda = lda.fit(tfidf_docs, sms.spam) >>> sms['lda_spaminess'] = lda.predict(tfidf_docs) >>> ((sms.spam - sms.lda_spaminess) ** 2. Let?s do some cross validation this time: >>> from sklearn.model_selection import cross_val_score >>> lda = LDA(n_components=1) >>> scores = cross_val_score(lda, tfidf_docs, sms.spam, cv=5) >>> "Accuracy: {:.2f} (+/-{:.2f})".format(scores.mean(), scores.std() * 2) 'Accuracy: 0.76 (+/-0.03)' 51 See the web page titled ?Distance Metric Learning: A Comprehensive Survey? (https://www.cs.cmu.edu/ ~liuy/frame_survey_v2.pdf).

4.8 Topic vector power

This is called ?semantic search,? not to be confused with the ?semantic web.?52 Semantic search is what strong search engines do when they give you documents that don?t contain many of the words in your query, but are exactly what you were looking for. These advanced search engines use LSA topic vectors to tell the difference between a Python package in ?The Cheese Shop? and a python in a Florida pet shop aquarium, while still recognizing its similarity to a ?Ruby gem.?53 Semantic search gives you a tool for finding and generating meaningful text. As Geoffry Hinton says, ?To deal with hyperplanes in a 14-dimensional space, visu- alize a 3D space and say 14 to yourself loudly.? If you read Abbott?s 1884 Flatland when you were young and impressionable, you might be able to do a little bit better than this hand waving. If you?re taking a moment to think deeply about four dimensions, keep in mind that the explosion in complexity you?re trying to wrap your head around is even greater than the complexity growth from 2D to 3D and exponentially greater than the growth in complexity from a 1D world of numbers to a 2D world of triangles, squares, and circles. A 1.5D fractal has infinite length and completely fills a 2D plane while having less than two dimensions!54 But 52 The semantic web is the practice of structuring natural language text with the use of tags in an HTML docu- ment so that the hierarchy of tags and their content provide information about the relationships (web of connections) between elements (text, images, videos) on a web page.

4.8.1 Semantic search

In this chapter, you?ve learned two ways? LSA and LDiA?to compute topic vectors that capture the semantics (mean- ing) of words and documents in a vector. Traditional index- ing approaches work with binary word occurrence vectors, discrete vectors (BOW vectors), sparse continuous vectors (TF-IDF vectors), and low-dimensional continuous vectors (3D GIS data). But high-dimensional continuous vectors, such as topic vectors from LSA or LDiA, are a challenge.58 Inverted indexes work for discrete vectors or binary vectors, like tables of binary or integer word-document vectors, because the index only needs to maintain an entry for each nonzero discrete dimension. Because TF-IDF vectors are sparse, mostly zero, you don?t need an entry in your index for most dimensions for most documents.59 LSA (and LDiA) produce topic vectors that are high-dimensional, continuous, and dense (zeros are rare). In figure 4.6, each row represents a topic vector size (dimensionality), starting with 2 dimensions and working up to 16 dimensions, like the vectors you used earlier for the SMS spam problem. 100th cosine Top 1 Top 2 Top 10 distance correct correct correct Top 100 correct Dimensions 2 .00 TRUE TRUE TRUE TRUE 3 .00 TRUE TRUE TRUE TRUE 4 .00 TRUE TRUE TRUE TRUE 5 .01 TRUE TRUE TRUE TRUE 6 .02 TRUE TRUE TRUE TRUE 7 .02 TRUE TRUE TRUE FALSE 8 .03 TRUE TRUE TRUE FALSE 9 .04 TRUE TRUE TRUE FALSE 10 .05 TRUE TRUE FALSE FALSE 11 .07 TRUE TRUE TRUE FALSE 12 .06 TRUE TRUE FALSE FALSE 13 .09 TRUE TRUE FALSE FALSE 14 .14 TRUE FALSE FALSE FALSE 15 .14 TRUE TRUE FALSE FALSE 16 .09 TRUE TRUE FALSE FALSE Figure 4.6 Semantic search accuracy deteriorates at around 12-D. To find precise semantic matches, you need to find all the closest document topic vectors to a particu- lar query (search) topic vector.

Summary

? No matter how you create your topic vectors, they can be used for semantic search to find documents based on their meaning. ? Topic vectors can be used to predict whether a social post is spam or is likely to be ?liked.? ? Now you know how to sidestep around the curse of dimensionality to find approximate nearest neighbors in your semantic vector space. 63 If you want to learn about faster ways to find a high-dimensional vector?s nearest neighbors, check out appen- dix F, or just use the Spotify annoy package to index your topic vectors.

Part 2 Deeper learning (neural networks)Part 2 30.0

Baby steps with neural networks (perceptrons and backpropagation) This chapter covers ? Learning the history of neural networks ? Stacking perceptrons ? Understanding backpropagation ? Seeing the knobs to turn on neural networks ? Implementing a basic neural network in Keras In recent years, a lot of hype has developed around the promise of neural networks and their ability to classify and identify input data, and more recently the ability of certain network architectures to generate original content.

5.1 Neural networks, the ingredient list

As electrical signals flow into the cell through the dendrites (see figure 5.1) into the nucleus, an electric charge begins to build up. The cell is more ?sensitive? to signals through cer- tain dendrites than others, so it takes less of a signal in those paths to fire the axon. The biology that controls these relationships is most certainly beyond the scope of this book, but the key concept to notice here is the way the cell weights incoming signals when deciding when to fire. Dendrite Axon Nucleus Figure 5.1 Neuron cell 1 Rosenblatt, Frank (1957), ?The perceptron?a perceiving and recognizing automaton.? Report 85-460-1, Cor- nell Aeronautical Laboratory.

5.1.2 A numerical perceptron

In ?normal? machine learning problems, like predicting home prices, your features might be square footage, last sold price, and ZIP code. Or perhaps you?d like to predict the spe- cies of a certain flower using the Iris dataset.2 In that case your features would be petal length, petal width, sepal length, and sepal width. In Rosenblatt?s experiment, the features were the intensity values of each pixel (subsections of the image), one pixel per photo receptor. If you?re familiar with linear regression, then you probably already know where these weights come from.3   2 The Iris dataset is frequently used to introduce machine learning to new students.

5.1.3 Detour through bias

If you want to get deep, think about how you are using a biological neu- ron to read this book about natural language processing to learn about deep learning.4 Inputs Activation Output Xb = 1 y z X0 + ... Axon Xn Dendrite Figure 5.3 A perceptron and a biological neuron And in mathematical terms, the output of your perceptron, denoted f(x), looks like n ? f(x) = 1 if xiwi > threshold else 0 i=0 Equation 5.1 Threshold activation function 4 Natural language understanding (NLU) is a term often used in academic circles to refer to natural language processing when that processing appears to demonstrate that the machine understands natural language text. You can also use the numpy dot function to multiply your two vectors together: >>> import numpy as np >>> example_input = [1, .2, .1, .05, .2] >>> example_weights = [.2, .12, .4, .6, .90] >>> input_vector = np.array(example_input) >>> weights = np.array(example_weights) >>> bias_weight = .2 >>> activation_level = np.dot(input_vector, weights) +\ ... (bias_weight * 1) >>> activation_level 0.674 The multiplication by one (* 1) is just to emphasize that the bias_weight is like all the other weights: it?s multiplied by an input value, only the bias_weight input feature value is always 1. Let?s assume that your earlier example_input should have resulted in a 0 instead: >>> expected_output = 0 >>> new_weights = [] >>> for i, x in enumerate(example_input): ... new_weights.append(weights[i] + (expected_output -\ ... perceptron_output) * x) >>> weights = np.array(new_weights) For example, in the first index above: new_weight = .2 + (0 - 1) * 1 = -0.8 >>> example_weights [0.2, 0.12, 0.4, 0.6, 0.9] >>> weights [-0.8 -0.08 0.3 0.55 0.7] Original weights New weights This process of exposing the network over and over to the same training set can, under the right circumstances, lead to an accurate predictor even on input that the perceptron has never seen. Listing 5.1 OR problem setup >>> sample_data = [[0, 0], # False, False ... [0, 1], # False, True ... [1, 0], # True, False ... [1, 1]] # True, True >>> expected_results = [0, # (False OR False) gives False ... 1, # (False OR True) gives True ... 1, # (True OR False) gives True ... 1] # (True OR True) gives True >>> activation_threshold = 0.5 You need a few tools to get started: numpy just to get used to doing vector (array) mul- tiplication, and random to initialize the weights: >>> from random import random >>> import numpy as np >>> weights = np.random.random(2)/1000 # Small random float 0 < w < .001 >>> weights [5.62332144e-04 7.69468028e-05] You need a bias as well: >>> bias_weight = np.random.random() / 1000 >>> bias_weight 0.0009984699077277136 Then you can pass it through your pipeline and get a prediction for each of your four samples. Listing 5.2 Perceptron random guessing >>> for idx, sample in enumerate(sample_data): ... input_vector = np.array(sample) ... activation_level = np.dot(input_vector, weights) +\ ... (bias_weight * 1) ... if activation_level > activation_threshold: ... perceptron_output = 1 ... else: ... perceptron_output = 0 163 Neural networks, the ingredient list ... print('Predicted {}'.format(perceptron_output)) ... print('Expected: {}'.format(expected_results[idx])) ... print() Predicted 0 Expected: 0 Predicted 0 Expected: 1 Predicted 0 Expected: 1 Predicted 0 Expected: 1 Your random weight values didn?t help your little neuron out that much?one right and three wrong. Listing 5.3 Perceptron learning >>> for iteration_num in range(5): ... correct_answers = 0 ... for idx, sample in enumerate(sample_data): ... input_vector = np.array(sample) ... weights = np.array(weights) ... activation_level = np.dot(input_vector, weights) +\ ... (bias_weight * 1) ... if activation_level > activation_threshold: ... perceptron_output = 1 ... else: ... perceptron_output = 0 ... if perceptron_output == expected_results[idx]: ... correct_answers += 1 ... new_weights = [] ... for i, x in enumerate(sample): ... new_weights.append(weights[i] + (expected_results[idx] -\ ... perceptron_output) * x) ... bias_weight = bias_weight + ((expected_results[idx] -\ ... perceptron_output) * 1) ... weights = np.array(new_weights) ... print('{} correct answers out of 4, for iteration {}'\ ... .format(correct_answers, iteration_num)) 3 correct answers out of 4, for iteration 0 2 correct answers out of 4, for iteration 1 3 correct answers out of 4, for iteration 2 4 correct answers out of 4, for iteration 3 4 correct answers out

5.1.4 Let?s go skiing?the error surface

The goal of training in neural networks, as we stated earlier, is to minimize a cost func- tion by finding the best parameters (weights). From earlier, mean squared error is a common cost function (shown back in equation 5.5). If you imagine plotting the error as a function of the possible weights, given a set of inputs and a set of expected outputs, a point exists where that function is closest to zero.

5.1.5 Off the chair lift, onto the slope

At each epoch, the algorithm is performing gradient descent in trying to minimize the error. And, as in skiing, if these pits are big enough, they can suck you in and you might not reach the bottom of the slope.

5.1.6 Let?s shake things up a bit

Up until now, you have been aggregating the error for all the training examples and skiing down the slope as best you could. This training approach, as described, is batch learning. With this single static surface, if you only head downhill from a random starting point, you could end up in some local minima (divot

5.1.7 Keras: Neural networks in Python

>>> x_train = np.array([[0, 0], ... [0, 1], ... [1, 0], ... [1, 1]]) >>> y_train = np.array([[0], ... [1], ... [1], ... [0]]) >>> model = Sequential() >>> num_neurons = 10 >>> model.add(Dense(num_neurons, input_dim=2)) >>> model.add(Activation('tanh')) >>> model.add(Dense(1)) >>> model.add(Activation('sigmoid')) >>> model.summary() Layer (type) Output Shape Param # === dense_18 (Dense) (None, 10) 30 ____ activation_6 (Activation) (None, 10) 0 ____ dense_19 (Dense) (None, 1) 11 ____ activation_7 (Activation) (None, 1) 0 === Total params: 41.0 Trainable params: 41.0 Non-trainable params: 0.0 Stochastic gradient descent, but there are others x_train is a list of samples of 2D feature vectors used for training. The weights are initialized, and you can use this random state to try to predict from your dataset, but you?ll only get random guesses: >>> model.predict(x_train) [[0.5] [0.43494844] [0.50295198] [0.42517585]] The predict method gives the raw output of the last layer, which would be generated by the sigmoid function in this example. model.fit(x_train, y_train, epochs=100) Epoch 1/100 4/4 [==============================] - 0s - loss: 0.6917 - acc: 0.7500 Epoch 2/100 4/4 [==============================] - 0s - loss: 0.6911 - acc: 0.5000 Epoch 3/100 4/4 [==============================] - 0s - loss: 0.6906 - acc: 0.5000 ... Epoch 100/100 4/4 [==============================] - 0s - loss: 0.6661 - acc: 1.0000   178 CHAPTER 5 Baby steps with neural networks (perceptrons and backpropagation) The network might not converge on the first try.

5.1.9 Normalization: input with style

Say, for a two- bedroom house that last sold for $275,000: input_vec = [2, 275000] As the network tries to learn anything about this data, the weights associated with bed- rooms in the first layer would need to grow huge quickly to compete with the large val- ues associated with price.

Summary

? The amount a weight contributes to a model?s error is directly related to the amount it needs to be updated. ? Neural networks are, at their heart, optimization engines. ? Watch out for pitfalls (local minima) during training by monitoring the gradual reduction in error. ? Keras helps make all of this neural network math accessible.

6 Reasoning with word vectors (Word2vec)Reasoning with word 30.0

They might get better at providing you search results the next time you look for a scientist.1 With word vectors, you can search for words or names that combine the meaning of the words ?woman,? ?Europe,? ?physics,? ?scientist,? and ?famous,? and that would get you close to the token ?Marie Curie? that you?re looking for. And all you have to do to make that happen is add up the word vectors for each of those words that you want to combine: >>> answer_vector = wv['woman'] + wv['Europe'] + wv[physics'] +\ ... wv['scientist'] 1 At least, that?s what it did for us in researching this book.

6.1.1 Analogy questions

Again, Google Search, Bing, and even Duck Duck Go aren?t much help with this one.2 But with word vectors, the solution is as simple as subtracting ?germs? from ?Louis Pas- teur? and then adding in some ?physics?: >>> answer_vector = wv['Louis_Pasteur'] - wv['germs'] + wv['physics'] And if you?re interested in trickier analogies about people in unrelated fields, such as musicians and scientists, you can do that, too: Who is the Marie Curie of music?

6.2 Word vectors

In 2013, once at Google, Mikolov and his teammates released the software for creating these word vectors and called it Word2vec.6 Word2vec learns the meaning of words merely by processing a large corpus of unla- beled text. Instead of trying to train a neural network to learn the target word meanings directly (on the basis of labels for that meaning), you teach the network to predict words near the target word in your sentences. If you want to learn more about unsupervised deep learning models that cre- ate compressed representations of high-dimensional objects like words, search for the term ?autoencoder.?7 They?re also a common way to get started with neural nets, because they can be applied to almost any dataset. I?m sure your nessvector dimensions will be much more fun and useful, like ?trumpness? and ?ghandiness.? 7 See the web page titled ?Unsupervised Feature Learning and Deep Learning Tutorial? (http://ufldl.stanford .edu/tutorial/unsupervised/Autoencoders/).

6.2.1 Vector-oriented reasoning

Word2vec embeddings were four times more accurate (45%) compared to equivalent LSA models (11%) at answering analogy questions like those above.10 The accuracy improvement was so surprising, in fact, that Mikolov?s initial paper was rejected by the International Conference on 8 For those not up on sports, the Portland Timbers and Seattle Sounders are major league soccer teams. You can think of this like a   190 CHAPTER 6 Reasoning with word vectors (Word2vec) Chicago 3 2 Philadelphia San Jose San Diego 1 Phoenix, AZ Los Angeles 0 y ?1 ?2 Austin Jacksonville, FL Houston & Dallas ?5 ?4 ?3 ?2 ?1 0 1 2 x Figure 6.2 Word vectors for ten US cities projected onto a 2D map cartoon map of a popular tourist destination or one of those impressionistic maps you see on bus stop posters. So your machine will be able to generate impres- sionistic maps like the one in figure 6.2 using word vectors you are learning about in this chapter.12 If you?re familiar with these US cities, you might realize that this isn?t an accurate geographic map, but it?s a pretty good semantic map.

6.2.2 How to compute Word2vec representations

192 CHAPTER 6 Reasoning with word vectors (Word2vec) surrounding words wt-2, wt-1 Claude Monet painted the Grand Canal of venice in 1908. surrounding words wt+1, wt+2 = words to be predicted word wt = input word Figure 6.3 Training input and output example for the skip-gram approach output example skip-grams are shown in figure 6.3. For each of the K output nodes, the softmax output value can be calculated using the normalized exponential function: e zj (z)j = ? ? e zk K k =1 If your output vector of a three-neuron output layer looks like this 0.5 v = 0.9 0.2 Equation 6.3 Example 3D vector   193 Word vectors The ?squashed? vector after the softmax activation would look like this: 0.309 (v) = ? 0.461 0.229 Equation 6.4 Example 3D vector after softmax Notice that the sum of these values (rounded to three significant digits) is approxi- mately 1.0, like a probability distribution. One-hot vector ?Monet? Softmax output ?Claude? n hidden neurons Claude 0 0.976 Claude Monet 1 0.002 Monet painted 0 0.001 painted the 0 0.001 the ... ... 1806 0 0.002 1806 One-hot vector ?Monet? Softmax output ?painted? n hidden neurons Claude 0 0.001 Claude Monet 1 0.002 Monet painted 0 0.983 painted ... ... the 0 0.001 the ... 1806 0 0.002 1806 Figure 6.4 Network example for the skip-gram training   194 CHAPTER 6 Reasoning with word vectors (Word2vec) When you look at the structure of the neural network for word embed- NOTE ding, you?ll notice that the implementation looks similar to what you discov- ered in chapter 5. *.55) .12 .44 .02 .32 .23 .55 .06 .32 = .61 Resulting 3-D word vector Figure 6.5 Conversion of one-hot vector to word vector   196 CHAPTER 6 Reasoning with word vectors (Word2vec) CONTINUOUS BAG-OF-WORDS APPROACH In the continuous bag-of-words approach, you?re trying to predict the center word based on the surrounding words (see figures 6.5 and 6.6 and table 6.2). surrounding words wt-2, wt-1 Claude Monet painted the Grand Canal of Venice in 1908. surrounding words wt+1, wt+2 = input words target word wt = word to be predicted Figure 6.6 Training input and output example for the CBOW approach Table 6.2 Ten CBOW 5-grams from sentence about Monet Input word wt-2 Input word wt-1 Input word wt+1 Input word wt+2 Expected output wt Monet painted Claude Claude painted the Monet Claude Monet the Grand painted Monet painted Grand Canal the painted the Canal of Grand the Grand of Venice Canal Grand Canal Venice in of Canal of in 1908 Venice of Venice 1908 in Venice in 1908 Based on the training sets, you can create your multi-hot vectors as inputs and map them to the target word as output. 197 Word vectors   Multi-hot Softmax output n hidden neurons vector ?painted? Claude 1 0.03 Claude Monet 1 0.001 Monet painted 0 0.952 painted ... the 1 0.000 the Grand 1 0.002 Grand ... ... 0.002 1806 1806 0 Figure 6.7 CBOW Word2vec network Continuous bag of words vs. bag of words In previous chapters, we introduced the concept of a bag of words, but how is it dif- ferent than a continuous bag of words?

6.2.3 How to use the gensim.word2vec module

If you?ve already installed the nlpia package,17 you can download a pretrained Word2vec model with the following command: >>> from nlpia.data.loaders import get_data >>> word_vectors = get_data('word2vec') If that doesn?t work for you, or you like to ?roll your own,? you can do a Google search for Word2vec models pretrained on Google News documents.18 After you find and download the model in Google?s original binary format and put it in a local path, you can load it with the gensim package like this: >>> from gensim.models.keyedvectors import KeyedVectors >>> word_vectors = KeyedVectors.load_word2vec_format(\ ... '/path/to/GoogleNews-vectors-negative300.bin.gz', binary=True) Working with word vectors can be memory intensive. In the following example, you?ll load the 200k most common words from the Google News corpus: >>> from gensim.models.keyedvectors import KeyedVectors >>> word_vectors = KeyedVectors.load_word2vec_format(\ ... '/path/to/GoogleNews-vectors-negative300.bin.gz', ... binary=True, limit=200000) 17 See the README file at http://github.com/totalgood/nlpia for installation instructions. That?s why analogies and even zeugmas, odd juxtopositions of multiple meanings within the same word, are no problem:19 >>> word_vectors.most_similar(positive=['cooking', 'potatoes'], topn=5) [('cook', 0.6973530650138855), ('oven_roasting', 0.6754530668258667), ('Slow_cooker', 0.6742032170295715), ('sweet_potatoes', 0.6600279808044434), ('stir_fry_vegetables', 0.6548759341239929)] >>> word_vectors.most_similar(positive=['germany', 'france'], topn=1) [('europe', 0.7222039699554443)] Word vector models also allow you to determine unrelated terms. If you want to perform calculations (such as the famous example king + woman - man = queen, which was the example that got Mikolov and his advisor excited in the first place), you can do that by adding a negative argument to the most_similar method call: >>> word_vectors.most_similar(positive=['king', 'woman'], ... negative=['man'], topn=2) [('queen', 0.7118192315101624), ('monarch', 0.6189674139022827)] 19 Surfaces and Essences: Analogy as the Fuel and Fire of Thinking by Douglas Hoffstadter and Emmanuel Sander makes it clear why machines that can handle analogies and zeugmas are such a big deal.

6.2.4 How to generate your own word vector representations

Your training input should look similar to the following structure: >>> token_list [['to', 'provide', 'early', 'intervention/early', 'childhood', 'special', 'education', 'services', 'to', 'eligible', 'children', 'and', 'their', 'families'], ['essential', 'job', 'functions'], ['participate', 'as', 'a', 'transdisciplinary', 'team', 'member', 'to', 'complete', 'educational', 'assessments', 'for'] ...] To segment sentences and then convert sentences into tokens, you can apply the vari- ous strategies you learned in chapter 2. TRAIN YOUR DOMAIN-SPECIFIC WORD2VEC MODEL Get started by loading the Word2vec module: >>> from gensim.models.word2vec import Word2Vec The training requires a few setup details, shown in the following listing. Listing 6.3 Instantiating a Word2vec model >>> model = Word2Vec(... token_list, ... workers=num_workers, ... size=num_features, ... min_count=min_word_count, ... window=window_size, ... sample=subsampling) Depending on your corpus size and your CPU performance, the training will take a significant amount of time. The following command will discard the unneeded output weights of your neural network: >>> model.init_sims(replace=True) The init_sims method will freeze the model, storing the weights of the hidden layer and discarding the output weights that predict word co-ocurrences. You can save the trained model with the following command and preserve it for later use: >>> model_name = "my_domain_specific_word2vec_model" >>> model.save(model_name) If you want to test your newly trained model, you can use it with the same method you learned in the previous section; use the following listing.

6.2.6 fastText

The new algorithm, which they named fastText, 21 Stanford GloVe Project (https://nlp.stanford.edu/projects/glove/). 23 GloVe: Global Vectors for Word Representation, by Jeffrey Pennington, Richard Socher, and Christopher D. Man- ning: https://nlp.stanford.edu/pubs/glove.pdf.

6.2.7 Word2vec vs. LSA

Even though we didn?t say much about the LSA topic-document vectors in chapter 4, LSA gives you those, too. LSA topic-document 26 See the web page titled ?fastText/pretrained-vectors.md at master? (https://github.com/facebookresearch/ fastText/blob/master/pretrained-vectors.md).

6.2.8 Visualizing word relationships

209 Word vectors As you?ve seen earlier, if you want to retrieve the 300-D vector for a particular word, you can use the square brackets on this KeyedVectors object to .__getitem__() any word or n-gram: >>> wv['Illini'] array([0.15625 , 0.18652344, 0.33203125, 0.55859375, 0.03637695, -0.09375 , -0.05029297, 0.16796875, -0.0625 , 0.09912109, -0.0291748 , 0.39257812, 0.05395508, 0.35351562, -0.02270508, ... Listing 6.7 Distance between ?Illinois? and ?Illini? >>> import numpy as np >>> np.linalg.norm(wv['Illinois'] - wv['Illini']) 3.3653798 >>> cos_similarity = np.dot(wv['Illinois'], wv['Illini']) / (... np.linalg.norm(wv['Illinois']) *\ ... np.linalg.norm(wv['Illini'])) >>> cos_similarity 0.5501352 >>> 1 - cos_similarity 0.4498648 Euclidean distance Cosine similarity is the normalized dot product Cosine distance These distances mean that the words ?Illini? and ?Illinois? are only moderately close to one another in meaning. Listing 6.8 Some US city data >>> from nlpia.data.loaders import get_data >>> cities = get_data('cities') >>> cities.head(1).T geonameid 3039154 name El Tarter 29 The word ?Illini? refers to a group of people, usually football players and fans, rather than a single geographic region like ?Illinois? (where most fans of the ?Fighting Illini? live). Listing 6.9 Some US state data >>> us = cities[(cities.country_code == 'US') &\ ... (cities.admin1_code.notnull())].copy() >>> states = pd.read_csv(\ ... 'http://www.fonz.net/blog/wp-content/uploads/2008/04/states.csv') >>> states = dict(zip(states.Abbreviation, states.State)) >>> us['city'] = us.name.copy() >>> us['st'] = us.admin1_code.copy() >>> us['state'] = us.st.map(states) >>> us[us.columns[-3:]].head() city st state geonameid 4046255 Bay Minette AL Alabama 4046274 Edna TX Texas 4046319 Bayou La Batre AL Alabama 4046332 Henderson TX Texas 4046430 Natalia TX Texas Now you have a full state name for each city in addition to its abbreviation. Let?s check to see which of those state names and city names exist in your Word2vec vocabulary: >>> vocab = pd.np.concatenate([us.city, us.st, us.state]) >>> vocab = np.array([word for word in vocab if word in wv.wv]) >>> vocab[:5] array(['Edna', 'Henderson', 'Natalia', 'Yorktown', 'Brighton']) Even when you only look at United States cities, you?ll find a lot of large cities with the same name, like Portland, Oregon and Portland, Maine. Listing 6.10 Augment city word vectors with US state word vectors >>> city_plus_state = [] >>> for c, state, st in zip(us.city, us.state, us.st): ... if c not in vocab: ... continue ... row = [] ... if state in vocab: ... row.extend(wv[c] + wv[state]) ... else: ... row.extend(wv[c] + wv[st]) ... city_plus_state.append(row) >>> us_300D = pd.DataFrame(city_plus_state) Depending on your corpus, your word relationship can represent different attributes, such as geographical proximity or cultural or economic similarities. >>> from sklearn.decomposition import PCA >>> pca = PCA(n_components=2) >>> us_300D = get_data('cities_us_wordvectors') >>> us_2D = pca.fit_transform(us_300D.iloc[:, :300]) The last column of this DataFrame contains the city name, which is also stored in the DataFrame index. Listing 6.12 Bubble plot of US city word vectors >>> import seaborn >>> from matplotlib import pyplot as plt >>> from nlpia.plots import offline_plotly_scatter_bubble >>> df = get_data('cities_us_wordvectors_pca2_meta') >>> html = offline_plotly_scatter_bubble(... df.sort_values('population', ascending=False)[:350].copy()\ ... .sort_values('population'), ... filename='plotly_scatter_bubble.html', ... x='x', y='y', ... size_col='population', text_col='name', category_col='timezone', ... xscale=None, yscale=None, # 'log' or None ... layout={}, marker={'sizeref': 3000}) {'sizemode': 'area', 'sizeref': 3000} To produce the 2D representations of your 300-D word vectors, you need to use a dimension reduction technique.

6.2.9 Unnatural words

And if Figure 6.9 Decoder rings (left: Hubert Berberich (HubiB) (https://commons.wikimedia.org/wiki/File:CipherDisk2000.jpg), CipherDisk2000, marked as public domain, more details on Wikimedia Commons: https://commons.wikimedia.org/wiki/Template:PD-self; middle: Cory Doctorow (https://www.flickr.com/photos/doctorow/2817314740/in/photostream/), Crypto wedding-ring 2, https://creativecommons.org/licenses/by-sa/2.0/legalcode; right: Sobebunny (https://commons.wikimedia.org/wiki/File:Captain-midnight-decoder.jpg), Captain-midnight-decoder, https://creativecommons.org/licenses/by-sa/3.0/legalcode) 30 See the web page titled ?Caesar cipher? (https://en.wikipedia.org/wiki/Caesar_cipher).

6.2.10 Document similarity with Doc2vec

If you?re running low on RAM, and you know the number of documents TIP ahead of time (your corpus object isn?t an iterator or generator), you might want to use a preallocated numpy array instead of Python list for your training_corpus: training_corpus = np.empty(len(corpus), dtype=object); ? training_corpus[i] = ? Once the Doc2vec model is trained, you can infer document vectors for new, unseen documents by calling infer_vector on the instantiated and trained model: >>> model.infer_vector(simple_preprocess(\ ... 'This is a completely unseen document'), steps=10) Doc2vec requires a ?training? step when inferring new vectors.

Summary

? You?ve learned how word vectors and vector-oriented reasoning can solve some surprisingly subtle problems like analogy questions and nonsynonomy relation- ships between words. ? You can now train Word2vec and other word vector embeddings on the words you use in your applications so that your NLP pipeline isn?t ?polluted? by the GoogleNews meaning of words inherent in most Word2vec pretrained models. ? A PCA projection of geographic word vectors like US city names can reveal the cultural closeness of places that are geographically far apart. ? If you respect sentence boundaries with your n-grams and are efficient at set- ting up word pairs for training, you can greatly improve the accuracy of your latent semantic analysis word embeddings (see chapter 4).

7 Getting words in order with convolutional neural networks (CNNs)Getting words in order with convolutional 30.0

? Building a CNN ? Vectorizing natural language text in a way that suits neural networks ? Training a CNN ? Classifying the sentiment of novel text Language?s true power isn?t in the words themselves, but in the spaces between the words, in the order and combination of words. Understanding the intent beneath the words is a critical skill for an empathetic, emotionally intelligent listener or reader of natural language, be it human or machine.1 Just as in thought and ideas, it?s the connections between 1 International Association of Facilitators Handbook, http://mng.bz/oVWM. The perceptron quickly became the feedforward network (a multilayer perceptron), which led to the develop- ment of new variants: convolutional neural nets and recurrent neural nets, ever more efficient and precise tools to fish patterns out of large datasets. Although neural networks? original design purpose was to enable a machine to learn to quantify input, the field has since grown from just learning clas- sifications and regressions (topic analysis, sentiment analysis) to actually being able to generate novel text based on previously unseen input: translating a new phrase to another language, generating responses to questions not seen before (chatbot, any- one? 220 CHAPTER 7 Getting words in order with convolutional neural networks (CNNs) 7.1 Learning meaning The nature of words and their secrets are most tightly correlated to (after their defini- tion, of course) their relation to each other. The difference between the two is this: in the former, you examine the statement as if written on page?you?re looking for relationships in the position of words; in the latter, you explore it as if spoken?the words and letters become time series data.

7.2 Toolkit

Although a lot of the major players (hi Google and Facebook) have moved to lower-level languages for the implementation of these expensive calculations, the extensive resources poured into early models using Python for development have left their mark. Lasagne (Theano) and Skflow (TensorFlow) are popular options, but we?ll use Keras (https://keras.io/) for its balance of friendly API and versatility.

7.3 Convolutional neural nets

Convolutional neural nets, or CNNs, get their name from the concept of sliding (or con- volving) a small window over the data sample.

7.3.1 Building blocks

A convolutional net, or convnet (yeah that extra n in there is hard to say), achieves its magic not by assigning a weight to each element (say, each pixel of an image), as in a traditional feedforward net; instead it defines a set of filters (also known as kernels) that move across the image. Or it could be the intensity of each pixel in a grayscale image (see figures 7.3 and 7.4), or the intensity in each of the color channels of each pixel in a color image.

7.3.3 Filter composition

z0 Summation Layer output w0 w1 w2 w3 w4 w5 w6 w7 w8 Filter Pairwise multiplication x1 x0 x2 x3 x4 x5 x6 x8 x7 Input data (image) Figure 7.5 Convolutional neural net step In figures 7.5 and 7.6, xi is the value of the pixel at position i and z0 is the output of a ReLU activation function (z_0 = max(sum(x * w), 0) or z0 = max(xi * wj), 0).

7.3.4 Padding

If you start a 3 x 3 filter in the upper-left corner of an input image and stride one pixel at a time across, stop- ping when the rightmost edge of the filter reaches the rightmost edge of the input, the output ?image? will be two pixels narrower than the source input. The downfall of this strategy is that the data in the edge of the original input is undersampled as the interior data points are passed into each filter multiple times, from the overlapped filter positions. 227 Convolutional neural nets The next strategy is known as padding, which consists of adding enough data to the input?s outer edges so that the first real data point is treated just as the innermost data points are. >>> model = Sequential() >>> model.add(Conv1D(filters=16, kernel_size=3, padding='same', activation='relu', strides=1, input_shape=(100, 300))) input_shape is still the shape of your unmodified input.

7.4 Narrow windows indeed

Figure 7.7 1D convolution The cat and dog went to the bodega together .03 .92 .66 .72 .11 .15 .12 .00 .23 .00 .32 .61 .34 .63 .33 .23 .52 .23 Input (word embeddings) .14 .62 .43 .32 .34 .00 .02 .34 .33 .24 .99 .62 .33 .27 .00 .66 .66 .56 .12 .02 .44 .42 .42 .11 .00 .23 .99 .32 .23 .55 .32 .22 .42 .00 .01 .25 w2 w1 w0 w2 w1 w0 w5 w4 w3 w5 w4 w3 Slide (convolve) Filter w8 w7 w6 w8 w7 w6 w9 w11 w10 w9 w11 w10 w12 w14 w13 w12 w14 w13 w15 w16 w17 w15 w16 w17 Aggregate and activation function z[0] = activation_function(sum(w * x[:, i:i+3])) Layer output (for a given filter) z0 z6 z5 z4 z3 z1 z2 Figure 7.8 1D convolution with embeddings

7.4.1 Implementation in Keras: prepping the data

A helper module to handle padding input The base Keras neural network model >>> import numpy as np >>> from keras.preprocessing import sequence >>> from keras.models import Sequential >>> from keras.layers import Dense, Dropout, Activation >>> from keras.layers import Conv1D, GlobalMaxPooling1D The layer objects you?ll pile into the model Your convolution layer, and pooling First download the original dataset from the Stanford AI department (https://ai.stanford.edu/%7eamaas/data/sentiment/). ... """ ... positive_path = os.path.join(filepath, 'pos') ... negative_path = os.path.join(filepath, 'neg') ... pos_label = 1 ... neg_label = 0 ... dataset = [] ... ... for filename in glob.glob(os.path.join(positive_path, '*.txt')): ... with open(filename, 'r') as f: ... dataset.append((pos_label, f.read())) ... ... for filename in glob.glob(os.path.join(negative_path, '*.txt')): ... with open(filename, 'r') as f: ... dataset.append((neg_label, f.read())) ... ... shuffle(dataset) ... ... return dataset The first example document should look something like the following. The first element in the tuple is the target value for sentiment: 1 for positive sentiment, 0 for negative: >>> dataset = pre_process_data('<path to your downloaded file>/aclimdb/train') >>> dataset[0] (1, 'I, as a teenager really enjoyed this movie! Listing 7.4 Vectorizer and tokenizer >>> from nltk.tokenize import TreebankWordTokenizer >>> from gensim.models.keyedvectors import KeyedVectors >>> from nlpia.loaders import get_data >>> word_vectors = get_data('w2v', limit=200000) get_data('w2v') downloads ?GoogleNews-vectors- negative300.bin.gz? to the nlpia.loaders.BIGDATA_PATH directory. >>> def tokenize_and_vectorize(dataset): ... tokenizer = TreebankWordTokenizer() ... vectorized_data = [] ... expected = [] ... for sample in dataset: ... tokens = tokenizer.tokenize(sample[1]) ... sample_vecs = [] ... for token in tokens: ... try: ... sample_vecs.append(word_vectors[token]) ... ... except KeyError: ... pass # No matching token in the Google w2v vocab ... ... vectorized_data.append(sample_vecs) ... ... return vectorized_data Note that you?re throwing away information here. Listing 7.5 Target labels >>> def collect_expected(dataset): ... """ Peel off the target values from the dataset """ ... expected = [] ... for sample in dataset: ... expected.append(sample[0]) ... return expected And then you simply pass your data into those functions: >>> vectorized_data = tokenize_and_vectorize(dataset) >>> expected = collect_expected(dataset) Next you?ll split the prepared data into a training set and a test set. Listing 7.7 CNN parameters How many samples to show the net before backpropagating the error and updating the weights Length of the token vectors you?ll create for passing into the convnet Number of filters you?ll train maxlen = 400 batch_size = 32 embedding_dims = 300 filters = 250 kernel_size = 3 hidden_dims = 250 epochs = 2 The width of the filters; actual filters will each be a matrix of weights of size: embedding_dims x kernel_size, or 50 x 3 in your case Number of neurons in the plain feedforward net at Number of times you?ll pass the entire the end of the chain training dataset through the network In listing 7.7, the kernel_size (filter size or window size) is a scalar TIP value, as opposed to the two-dimensional type filters you had with images. Listing 7.8 Padding and truncating your token sequence >>> def pad_trunc(data, maxlen): ... """ ... For a given dataset pad with zero vectors or truncate to maxlen ... """ ... new_data = [] An astute LiveBook reader (@madara) pointed out this can all be accomplished with a one-liner: [smp[:maxlen] + [[0.]

7.4.2 Convolutional neural network architecture

As with the feed- forward network from chapter 5, Sequential is one of the base classes for neural networks in Keras. In this case, you assume that it?s okay that the output is of smaller dimension than the input, and you set the padding to 'valid'. The kernel (window width) you already set to three tokens in listing 7.7. At each step, you?ll multiply the filter weight times the value in the three tokens it?s looking at (element-wise), sum up those answers, and pass them

7.4.3 Pooling

In an image processor, the pooling region would usually be a 2 x 2 pixel TIP window (and these don?t overlap, like your filters do), but in your 1D convo- lution they would be a 1D window (such as 1 x 2 or 1 x 3). 237 Narrow windows indeed 2D max pooling (2 x 2 window) .15 .12 .00 .23 .33 .23 .52 .23 .33 .52 .00 .02 .34 .33 .66 .66 .00 .66 .66 .56 1D max pooling (1 x 2 window) .33 .99 .01 .52 .99 .00 .01 .16 .33 .52 .23 .23 1D global max pooling .99 .00 .01 .16 .33 .52 .23 .23 .99 Figure 7.9 Pooling layers You have two choices for pooling (see figure 7.9): average and max. But even tossing aside all that good information, your toy model won?t be deterred: >>> model.add(GlobalMaxPooling1D()) Pooling options are GlobalMaxPooling1D(), MaxPooling1D(n), or AvgPooling1D(n), where n is the size of the area to pool and defaults to 2 if not provided.

7.4.4 Dropout

The idea is that on each training pass, if you ?turn off? a certain percentage of the input going to the next layer, randomly chosen on each pass, the model will be less likely to learn the specifics of the training set, ?overfitting,? and instead learn more nuanced representations of the patterns in the data and thereby be able to generalize and make accurate predictions when it sees completely novel data.

7.4.5 The cherry on the sundae

240 CHAPTER 7 Getting words in order with convolutional neural networks (CNNs)   Listing 7.13 Compile the CNN >>> model.compile(loss='binary_crossentropy', ... optimizer='adam', ... metrics=['accuracy']) The loss function is what the network will try to minimize. Listing 7.14 Output layer for categorical variable (word) >>> model.add(Dense(num_classes)) >>> model.add(Activation('sigmoid')) Where num_classes is ? well, you get the picture.

7.4.6 Let?s get to learning (training)

The inte- ger passed in as the argument to seed is unimportant, but as long as it?s consistent, the model will initialize its weights to small values in the same way: >>> import numpy as np >>> np.random.seed(1337) We haven?t seen definitive signs of overfitting; the accuracy improved for both the training and validation sets.

7.4.7 Using the model in a pipeline

Listing 7.17 Loading a saved model >>> from keras.models import model_from_json >>> with open("cnn_model.json", "r") as json_file: ... json_string = json_file.read() >>> model = model_from_json(json_string) >>> model.load_weights('cnn_weights.h5') Let?s make up a sentence with an obvious negative sentiment and see what the net- work has to say about it.

7.4.8 Where do you go from here?

And the filters follow suit and become three-dimensional as well, still a 3 x 3 or 5 x 5 or whatever in the x,y plane, but also three layers deep, resulting in filters that are three pixels wide x three pixels high x three channels deep, which leads to an interesting application in natural lan- guage processing. Your input to the network was a series of words represented as vectors lined up next to each other, 400 (maxlen) words wide x 300 elements long, and you used Word2vec embeddings for the word vectors. However, this analogy breaks down when you realize that the dimensions independent of word embeddings aren?t correlated with each other in the same way that color channels in an image are, so YMMV. Specifi- cally in this case, it?s a representation of the thought and details through the lens of sen- timent analysis, as all the ?learning? that happened was in response to whether the sample was labeled as a positive or negative sentiment. Using the intermediary vector directly from a convolutional neural net isn?t common, but in the coming chapters you?ll see examples from other neural network architectures where the details of that intermedi- ary vector become important, and in some cases are the end goal itself. In many ways, because of the pooling layers and the limits cre- ated by filter size (though you can make your filters large if you wish), you?re throwing away a good deal of information. As you?ve seen, they were able to efficiently detect and predict sentiment over a relatively large dataset, and even though you relied on the Word2vec embeddings, CNNs can perform on much less rich embeddings without mapping the entire language. A lot can depend on the available datasets, but richer models can be achieved by stacking convolutional layers and passing the output of the first set of filters as the ?image? sample into the second set and so on.

Summary

? A convolution is a window sliding over something larger (keeping the focus on a subset of the greater whole). ? Neural networks can treat text just as they treat images and ?see? them.

8 Loopy (recurrent) neural networks (RNNs)Loopy (recurrent) 30.0

The cat and dog went to the bodega together .03 .92 .66 .72 .11 .15 .12 .00 .23 .00 .32 .61 .34 .63 .33 .23 .52 .23 Input (word embeddings) .14 .62 .43 .32 .34 .00 .02 .34 .33 .24 .99 .62 .33 .27 .00 .66 .66 .56 .12 .02 .44 .42 .42 .11 .00 .23 .99 .32 .23 .55 .32 .22 .42 .00 .01 .25 w2 w1 w0 w2 w1 w0 w5 w4 w3 w5 w4 w3 Slide (convolve) Filter w8 w7 w6 w8 w7 w6 w9 w11 w10 w9 w11 w10 w12 w14 w13 w12 w14 w13 w15 w16 w17 w15 w16 w17 Aggregate and activation function f(sum)(wi * xi)) Layer output (for a given filter) z0 z6 z5 z4 z3 z1 z2 Figure 8.1 1D convolution with embeddings   249 The cat and dog went to the bodega together .03 .92 .66 .72 .11 .15 .12 .00 .23 .00 .32 .61 .34 .63 .33 .23 .52 .23 .14 .62 .43 .32 .34 .00 .02 .34 .33 Input (word embeddings) .24 .99 .62 .33 .27 .00 .66 .66 .56 .12 .02 .44 .42 .42 .11 .00 .23 .99 .32 .23 .55 .32 .22 .42 .00 .01 .25 ? w0 w8 w7 w6 w5 w4 w3 w2 w1 w9 w17 w16 w15 w14 w13 w12 w11 w10 WN W2 ? Neuron weights (W1) w18 w19 w20 w21 w22 w23 w24 w25 w26 w27 w35 w34 w33 w32 w31 w30 w29 w28 w36 w43 w42 w41 w40 w39 w38 w37 w44 w45 w46 w47 w48 w49 w50 w51 w52 w53 Activation function f(sum(wi * xi)) Neuron outputs z0 z1 ? zn Activation function Prediction f(sum(wi * xi)) w0 w1 w2 Figure 8.2 Text into a feedforward network Sure, this is a viable model. 254 CHAPTER 8 Loopy (recurrent) neural networks (RNNs)   the arena into sped car clown The Hidden layer Output Associated label Figure 8.7 Data into convolutional network In your recurrent neural net, you pass in the word vector for the first token and get the network?s output.

8.1.1 Backpropagation through time

day Today was a good Hidden layer Hidden layer Hidden layer Hidden layer Hidden layer Hidden layer Ignored output Ignored output Ignored output Ignored output Ignored output Output error = y_true_label - y_output Figure 8.9 Only last output matters here   256 CHAPTER 8 Loopy (recurrent) neural networks (RNNs) With an error for a given sample, you need to figure out which weights to update, and by how much.

8.1.2 When do we update what?

You can figure out the various changes to the weights (as if they were in a bubble) at each time step and then sum up the changes and apply the aggregated changes to each of the weights of the hidden layer as the last step of the learning phase. As the weight update is applied once per data   258 CHAPTER 8 Loopy (recurrent) neural networks (RNNs) sample, the network will settle (assuming it converges) on the weight for that input to that neuron that best handles this task. day Today was a good Hidden layer Hidden layer Hidden layer Hidden layer Hidden layer Hidden layer y0 y1 y2 y3 y4 y5 error = sum([y_true_label[i] - y[i] for i in range(6)]) Figure 8.11 All outputs matter here This process is like the normal backpropagation through time for n time steps. As with a standard feedforward network, you update the weights only after you have calculated the proposed change in the weights for the entire backpropagation step for that input (or set of inputs).

8.1.4 There?s always a catch

Although a recurrent neural net may have relatively fewer weights (parameters) to learn, you can see from figure 8.12 how a recurrent net can quickly get expensive to train, especially for sequences of any significant length, say 10 tokens.

8.1.5 Recurrent neural net with Keras

Listing 8.1 Import all the things >>> import glob >>> import os >>> from random import shuffle >>> from nltk.tokenize import TreebankWordTokenizer >>> from nlpia.loaders import get_data >>> word_vectors = get_data('wv') Then you can build your data preprocessor, which will whip your data into shape, as shown in the following listing. ... """ ... positive_path = os.path.join(filepath, 'pos') ... negative_path = os.path.join(filepath, 'neg') ... pos_label = 1   261 Remembering with recurrent networks ... neg_label = 0 ... dataset = [] ... for filename in glob.glob(os.path.join(positive_path, '*.txt')): ... with open(filename, 'r') as f: ... dataset.append((pos_label, f.read())) ... for filename in glob.glob(os.path.join(negative_path, '*.txt')): ... with open(filename, 'r') as f: ... dataset.append((neg_label, f.read())) ... shuffle(dataset) ... return dataset As before, you can combine your tokenizer and vectorizer into a single function, as shown in the following listing. Listing 8.3 Data tokenizer + vectorizer >>> def tokenize_and_vectorize(dataset): ... tokenizer = TreebankWordTokenizer() ... vectorized_data = [] ... for sample in dataset: ... tokens = tokenizer.tokenize(sample[1]) ... sample_vecs = [] ... for token in tokens: ... try: ... sample_vecs.append(word_vectors[token]) ... except KeyError: ... pass ... vectorized_data.append(sample_vecs) ... return vectorized_data No matching token in the Google w2v vocab And you need to extricate (unzip) the target variable into separate (but correspond- ing) samples, as shown in the following listing. Listing 8.4 Target unzipper >>> def collect_expected(dataset): ... """ Peel off the target values from the dataset """ ... expected = [] ... for sample in dataset: ... expected.append(sample[0]) ... return expected Now that you have all the preprocessing functions assembled, you need to run them on your data, as shown in the following listing. 262 CHAPTER 8 Loopy (recurrent) neural networks (RNNs) >>> x_test = vectorized_data[split_point:] >>> y_test = expected[split_point:] You?ll use the same hyperparameters for this model: 400 tokens per example, batches of 32. Listing 8.7 Load your test and training data >>> import numpy as np >>> x_train = pad_trunc(x_train, maxlen) >>> x_test = pad_trunc(x_test, maxlen) >>> x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims)) >>> y_train = np.array(y_train) >>> x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims)) >>> y_test = np.array(y_test) Now that you have your data back, it?s time to build a model. Listing 8.8 Initialize an empty Keras network >>> from keras.models import Sequential >>> from keras.layers import Dense, Dropout, Flatten, SimpleRNN >>> num_neurons = 50 >>> model = Sequential() And then, as before, the Keras magic handles the complexity of assembling a neural net: you just need to add the recurrent layer you want to your network, as shown in the following listing. Listing 8.9 Add a recurrent layer >>> model.add(SimpleRNN(... num_neurons, return_sequences=True, ... input_shape=(maxlen, embedding_dims)))   263 Remembering with recurrent networks Now the infrastructure is set up to take each input and pass it into a simple recurrent neural net (the not-simple version is in the next chapter), and for each token, gather the output into a vector. Listing 8.10 Add a dropout layer >>> model.add(Dropout(.2)) >>> model.add(Flatten()) >>> model.add(Dense(1, activation='sigmoid')) You requested that the simple RNN return full sequences, but to prevent overfitting you add a Dropout layer to zero out 20% of those inputs, randomly chosen on each input example.

8.2 Putting things together

____ Layer (type) Output Shape Param # === simple_rnn_1 (SimpleRNN) (None, 400, 50) 17550 ____ dropout_1 (Dropout) (None, 400, 50) 0 ____ flatten_1 (Flatten) (None, 20000) 0 ____ dense_1 (Dense) (None, 1) 20001 === Total params: 37,551.0 Trainable params: 37,551.0 Non-trainable params: 0.0 ____ None Pause and look at the number of parameters you?re working with. Each neuron will need 300 weights: 50 * 300 = 15,000 Each neuron also has the bias term, which always has an input value of 1 (that?s what makes it a bias) but has a trainable weight: 15,000 + 50 (bias weights) = 15,050 15,050 weights in the first time step of the first layer.

8.3 Let?s get to learning our past selves

Listing 8.12 Train and save your model >>> model.fit(x_train, y_train, ... batch_size=batch_size, ... epochs=epochs, ... validation_data=(x_test, y_test)) Train on 20000 samples, validate on 5000 samples Epoch 1/2 20000/20000 [==============================] - 215s - loss: 0.5723 - acc: 0.7138 - val_loss: 0.5011 - val_acc: 0.7676 Epoch 2/2 20000/20000 [==============================] - 183s - loss: 0.4196 - acc: 0.8144 - val_loss: 0.4763 - val_acc: 0.7820 >>> model_structure = model.to_json() >>> with open("simplernn_model1.json", "w") as json_file: ... json_file.write(model_structure) >>> model.save_weights("simplernn_weights1.h5") Model saved.

8.4 Hyperparameters

>>> model.summary() ____ Layer (type) Output Shape Param # === simple_rnn_1 (SimpleRNN) (None, 400, 100) 40100 ____ dropout_1 (Dropout) (None, 400, 100) 0 ____ flatten_1 (Flatten) (None, 40000) 0 ____ dense_1 (Dense) (None, 1) 40001 === Total params: 80,101.0 Trainable params: 80,101.0 Non-trainable params: 0.0 ____ Listing 8.15 Train your larger network >>> model.fit(x_train, y_train, ... batch_size=batch_size, ... epochs=epochs, ... validation_data=(x_test, y_test)) Train on 20000 samples, validate on 5000 samples Epoch 1/2 20000/20000 [==============================] - 287s - loss: 0.9063 - acc: 0.6529 - val_loss: 0.5445 - val_acc: 0.7486 Epoch 2/2 20000/20000 [==============================] - 240s - loss: 0.4760 -

8.5 Predicting

Listing 8.16 Crummy weather sentiment >>> sample_1 = "I hate that the dismal weather had me down for so long, when ? will it break! >>> from keras.models import model_from_json >>> with open("simplernn_model1.json", "r") as json_file:

8.5.1 Statefulness

Sometimes you want to remember information from one input sample to the next, not just one-time step (token) to the next within a single sample. If you flip this to True when adding the SimpleRNN layer to your model, the last sample?s last output passes into itself at the next time step along with the first token input, just as it would in the middle of the sample.

8.5.2 Two-way street

Listing 8.17 Build a Bidirectional recurrent network >>> from keras.models import Sequential >>> from keras.layers import SimpleRNN >>> from keras.layers.wrappers import Bidirectional >>> num_neurons = 10 >>> maxlen = 100 >>> embedding_dims = 300 >>> model = Sequential() >>> model.add(Bidirectional(SimpleRNN(... num_neurons, return_sequences=True),\ ... input_shape=(maxlen, embedding_dims)))

8.5.3 What is this thing?

Ahead of the Dense layer you have a vector that is of shape (number of neurons x 1) coming out of the last time step of the Recurrent layer for a given input sequence.

Summary

? In natural language sequences (words or characters), what came before is important to your model?s understanding of the sequence. ? Splitting a natural language statement along the dimension of time (tokens) can help your machine deepen its understanding of natural language. ? Efficiently modeling natural language character sequences wasn?t possible until recurrent neural nets were applied to the task. ? You can model the natural language sequence in a document by passing the sequence of tokens through an RNN backward and forward simultaneously.

9 Improving retention with long short-term memory networksImproving retention with long short-term 30.0

278 CHAPTER 9 Improving retention with long short-term memory networks Output from t-1 Concatenated input Candidate gate (2 elements) Forget gate Output Output gate to t+1 Input at t Memory Figure 9.3 LSTM layer at time step t So let?s take a closer look at one of these cells. Output from t-1 Concatenated input Candidate gate (2 elements) Forget gate Output Output gate to t+1 Input at t Memory Figure 9.4 LSTM layer inputs 280 CHAPTER 9 Improving retention with long short-term memory networks The ?journey? through the cell isn?t a single road; it has branches, and you?ll follow each for a while then back up, progress, branch, and finally come back together for the grand finale of the cell?s output. But an input sequence can easily switch from one noun to another, because an input sequence can be composed of multiple phrases, sentences, Output from time step t-1 50-element vector output from previous time step Concatenated input 350-element vector Forget gate 351 weights (1 for bias) per neuron 17,550 total Input at time step t 1 token (word or character) represented by a 300-element vector Figure 9.5 First stop?the forget gate 281 LSTM or even documents. Output from time step t-1 50-element vector from previous time step Concatenated input 350-element vector Forget gate 351 weights (1 for bias) per neuron 17,550 total Input at time step t 1 token (word or character) represented by 300-element vector Candidate gate (2 elements) Output gate Output to t+1 Memory Figure 9.6 Forget gate 282 CHAPTER 9 Improving retention with long short-term memory networks × Memory vector at t-1 Mask from forget gate at t = New memory vector (elementwise) × .03 .99 = .0297 × = .42 .00 .00 × .01 .14 = .014 ... ... (50 elements total) (50 elements total) (50 elements total) × = .32 .00 .00 Memory vector Mask from at t-1 forget gate at t New memory vector Figure 9.7 Forget gate application The output vector of the forget gate is then a mask of sorts, albeit a porous one, that erases elements of the memory vector.

9.1.1 Backpropagation through time

Listing 9.3 Build a Keras LSTM network >>> from keras.models import Sequential >>> from keras.layers import Dense, Dropout, Flatten, LSTM >>> num_neurons = 50 >>> model = Sequential() >>> model.add(LSTM(num_neurons, return_sequences=True, ... input_shape=(maxlen, embedding_dims))) >>> model.add(Dropout(.2)) >>> model.add(Flatten()) >>> model.add(Dense(1, activation='sigmoid')) >>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy']) >>> model.summary() Layer (type) Output Shape Param # === lstm_2 (LSTM) (None, 400, 50) 70200 ____ dropout_2 (Dropout) (None, 400, 50) 0 ____ flatten_2 (Flatten) (None, 20000) 0 ____ dense_2 (Dense) (None, 1) 20001 === Total params: 90,201.0 Trainable params: 90,201.0 Non-trainable params: 0.0 Keras makes the implementation easy. Listing 9.4 Fit your LSTM model >>> model.fit(x_train, y_train, ... batch_size=batch_size, ... epochs=epochs, ... validation_data=(x_test, y_test)) Train on 20000 samples, validate on 5000 samples Epoch 1/2 20000/20000 [==============================] - 548s - loss: 0.4772 - acc: 0.7736 - val_loss: 0.3694 - val_acc: 0.8412 Epoch 2/2 20000/20000 [==============================] - 583s - loss: 0.3477 - acc: 0.8532 - val_loss: 0.3451 - val_acc: 0.8516 <keras.callbacks.History at 0x145595fd0> Train the model.

9.1.2 Where does the rubber hit the road?

Listing 9.6 Reload your LSTM model >>> from keras.models import model_from_json >>> with open("lstm_model1.json", "r") as json_file: ... json_string = json_file.read() >>> model = model_from_json(json_string) >>> model.load_weights('lstm_weights1.h5') Listing 9.7 Use the model to predict on a sample >>> sample_1 = """I hate that the dismal weather had me down for so long, ... when will it break! >>> vec_list = tokenize_and_vectorize([(1, sample_1)]) >>> test_vec_list = pad_trunc(vec_list, maxlen) Tokenize returns a list of the data (length 1 here). >>> test_vec = np.reshape(test_vec_list, ... (len(test_vec_list), maxlen, embedding_dims)) You pass a dummy value in the first element of the tuple, because your helper expects it from the way you processed the initial data.

9.1.3 Dirty data

Listing 9.8 Optimize the thought vector size >>> def test_len(data, maxlen): ... total_len = truncated = exact = padded = 0 ... for sample in data: ... total_len += len(sample) ... if len(sample) > maxlen: ... truncated += 1 ... elif len(sample) < maxlen: ... padded += 1 ... else: ... exact +=1 ... print('Padded: {}'.format(padded)) ... print('Equal: {}'.format(exact)) ... print('Truncated: {}'.format(truncated)) ... print('Avg length: {}'.format(total_len/len(data))) >>> dataset = pre_process_data('./aclimdb/train') >>> vectorized_data = tokenize_and_vectorize(dataset) >>> test_len(vectorized_data, 400) Padded: 22559 Equal: 12 Truncated: 2429 Avg length: 202.4424 Whoa. Listing 9.9 Optimize LSTM hyperparameters >>> import numpy as np >>> from keras.models import Sequential >>> from keras.layers import Dense, Dropout, Flatten, LSTM >>> maxlen = 200 >>> batch_size = 32 >>> embedding_dims = 300 >>> epochs = 2 >>> num_neurons = 50 >>> dataset = pre_process_data('./aclimdb/train') >>> vectorized_data = tokenize_and_vectorize(dataset) >>> expected = collect_expected(dataset) >>> split_point = int(len(vectorized_data)*.8) >>> x_train = vectorized_data[:split_point] >>> y_train = expected[:split_point] >>> x_test = vectorized_data[split_point:] >>> y_test = expected[split_point:] >>> x_train = pad_trunc(x_train, maxlen) >>> x_test = pad_trunc(x_test, maxlen) All the same code as earlier, but you limit the max length to 200 tokens. 290 CHAPTER 9 Improving retention with long short-term memory networks >>> x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims)) >>> y_train = np.array(y_train) >>> x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims)) >>> y_test = np.array(y_test) Listing 9.10 A more optimally sized LSTM >>> model = Sequential() >>> model.add(LSTM(num_neurons, return_sequences=True, ... input_shape=(maxlen, embedding_dims))) >>> model.add(Dropout(.2)) >>> model.add(Flatten()) >>> model.add(Dense(1, activation='sigmoid')) >>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy']) >>> model.summary() Layer (type) Output Shape Param # === lstm_1 (LSTM) (None, 200, 50) 70200 ____ dropout_1 (Dropout) (None, 200, 50) 0 ____ flatten_1 (Flatten) (None, 10000) 0 ____ dense_1 (Dense) (None, 1) 10001 === Total params: 80,201.0 Trainable params: 80,201.0 Non-trainable params: 0.0 Listing 9.11 Train a smaller LSTM >>> model.fit(x_train, y_train, ... batch_size=batch_size, ... epochs=epochs, ... validation_data=(x_test, y_test)) Train on 20000 samples, validate on 5000 samples Epoch 1/2 20000/20000 [==============================] - 245s - loss: 0.4742 - acc: 0.7760 - val_loss: 0.4235 - val_acc: 0.8010 Epoch 2/2 20000/20000 [==============================] - 203s - loss: 0.3718 - acc: 0.8386 - val_loss: 0.3499 - val_acc: 0.8450 >>> model_structure = model.to_json() >>> with open("lstm_model7.json", "w") as json_file: ... json_file.write(model_structure) >>> model.save_weights("lstm_weights7.h5") Well that trained much faster and the validation accuracy dropped less than half a per- cent (84.5% versus 85.16%).

9.1.4 Back to the dirty data

The list of ?unknowns,? which is basically just words you couldn?t find in the pretrained Word2vec model, is quite extensive. This isn?t the case for the Word2vec embeddings, but many tokens are omitted and they may or may not be important to you. You can use or train a word embedding that has a vector for every last one of your tokens, but doing so is almost always prohibitively expensive.

9.1.5 Words are hard. Letters are easier.

Listing 9.15 Pad and truncated characters >>> def char_pad_trunc(data, maxlen=1500): ... """ We truncate to maxlen or add in PAD tokens """ ... new_dataset = [] ... for sample in data: ... if len(sample) > maxlen: ... new_data = sample[:maxlen] ... elif len(sample) < maxlen: ... pads = maxlen - len(sample) ... new_data = sample + ['PAD'] * pads ... else: ... new_data = sample ... new_dataset.append(new_data) ... return new_dataset You chose maxlen of 1,500 to capture slightly more data than was in the average sam- ple, but you tried to avoid introducing too much noise with PADs. Listing 9.16 Character-based model ?vocabulary? >>> def create_dicts(data): ... """ Modified from Keras LSTM example""" ... chars = set() ... for sample in data: ... chars.update(set(sample)) ... char_indices = dict((c, i) for i, c in enumerate(chars)) ... indices_char = dict((i, c) for i, c in enumerate(chars)) ... return char_indices, indices_char   295 LSTM And then you can use that dictionary to create input vectors of the indices instead of the tokens themselves, as shown in the next two listings. Listing 9.17 One-hot encoder for characters >>> import numpy as np >>> def onehot_encode(dataset, char_indices, maxlen=1500): ... """ ... One-hot encode the tokens ... ... Args: ... dataset list of lists of tokens ... char_indices ... dictionary of {key=character, ... value=index to use encoding vector} ... maxlen int Length of each sample ... Return: ... np array of shape (samples, tokens, encoding length) ... """ ... X = np.zeros((len(dataset), maxlen, len(char_indices.keys()))) ... for i, sentence in enumerate(dataset): ... for t, char in enumerate(sentence): ... X[i, t, char_indices[char]] = 1 ... return X A numpy array of length equal to the number of data samples?each sample will be a number of tokens equal to maxlen, and each token will be a one-hot encoded vector of length equal to the number of characters Listing 9.18 Load and preprocess the IMDB data >>> dataset = pre_process_data('./aclimdb/train') >>> expected = collect_expected(dataset) >>> listified_data = clean_data(dataset) >>> common_length_data = char_pad_trunc(listified_data, maxlen=1500) >>> char_indices, indices_char = create_dicts(common_length_data) >>> encoded_data = onehot_encode(common_length_data, char_indices, 1500) And then you split up your data just like before, as shown in the next two listings. Listing 9.19 Split dataset for training (80%) and testing (20%) >>> split_point = int(len(encoded_data)*.8) >>> x_train = encoded_data[:split_point] >>> y_train = expected[:split_point] >>> x_test = encoded_data[split_point:] >>> y_test = expected[split_point:]   296 CHAPTER 9 Improving retention with long short-term memory networks Listing 9.20 Build a character-based LSTM >>> from keras.models import Sequential >>> from keras.layers import Dense, Dropout, Embedding, Flatten, LSTM >>> num_neurons = 40 >>> maxlen = 1500 >>> model = Sequential() >>> model.add(LSTM(num_neurons, ... return_sequences=True, ... input_shape=(maxlen, len(char_indices.keys())))) >>> model.add(Dropout(.2)) >>> model.add(Flatten()) >>> model.add(Dense(1, activation='sigmoid')) >>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy']) >>> model.summary() Layer (type) Output Shape Param # == lstm_2 (LSTM) (None, 1500, 40) 13920 ____ dropout_2 (Dropout) (None, 1500, 40) 0 ____ flatten_2 (Flatten) (None, 60000) 0 ____ dense_2 (Dense) (None, 1) 60001 == Total params: 73,921.0 Trainable params: 73,921.0 Non-trainable params: 0.0 So you?re getting more efficient at building LSTM models. Listing 9.21 Train a character-based LSTM >>> batch_size = 32 >>> epochs = 10 >>> model.fit(x_train, y_train, ... batch_size=batch_size, ... epochs=epochs, ...

9.1.6 My turn to chat

But you can use this approach to generate lots of text within a given set of parameters (in response to a user?s style, for example), and this larger corpus of novel text could then be indexed and searched as possible responses to a given query. Much like a Markov chain that predicts a sequence?s next word based on the prob- ability of any given word appearing after the 1-gram or 2-gram or n-gram that just occurred, your LSTM model can learn the probability of the next word based on what it just saw, but with the added benefit of memory! This is very similar to the word vector embedding approach you used in chapter 6, only you?re going to train a network on bigrams (2-grams) instead of skip-grams. A word generator model trained this way (see figure 9.10) would work just fine, but you?re going to cut to the chase and go straight down to the character level with the same approach (see figure 9.11).

9.1.7 My turn to speak more clearly

Listing 9.24 Preprocess Shakespeare plays >>> text = '' >>> for txt in gutenberg.fileids(): ... if 'shakespeare' in txt: ... text += gutenberg.raw(txt).lower() >>> chars = sorted(list(set(text))) >>> char_indices = dict((c, i) ... for i, c in enumerate(chars)) >>> indices_char = dict((i, c) ... for i, c in enumerate(chars)) >>> 'corpus length: {} total chars: {}'.format(len(text), len(chars)) 'corpus length: 375542 total chars: 50' Concatenate all Shakespeare plays in the Gutenberg corpus in NLTK. >>> maxlen = 40 >>> step = 3 >>> sentences = [] >>> next_chars = [] >>> for i in range(0, len(text) - maxlen, step): ... sentences.append(text[i: i + maxlen]) ... next_chars.append(text[i + maxlen]) >>> print('nb sequences:', len(sentences)) nb sequences: 125168 Grab a slice of the text. Listing 9.26 One-hot encode the training examples >>> X = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool) >>> y = np.zeros((len(sentences), len(chars)), dtype=np.bool)   303 LSTM >>> for i, sentence in enumerate(sentences): ... for t, char in enumerate(sentence): ... X[i, t, char_indices[char]] = 1 ... y[i, char_indices[next_chars[i]]] = 1 You then one-hot encode each character of each sample in the dataset and store it as the list X. Listing 9.27 Assemble a character-based LSTM model for generating text >>> from keras.models import Sequential >>> from keras.layers import Dense, Activation >>> from keras.layers import LSTM >>> from keras.optimizers import RMSprop >>> model = Sequential() >>> model.add(LSTM(128, ... input_shape=(maxlen, len(chars)))) >>> model.add(Dense(len(chars))) >>> model.add(Activation('softmax')) >>> optimizer = RMSprop(lr=0.01) >>> model.compile(loss='categorical_crossentropy', optimizer=optimizer) >>> model.summary() Layer (type) Output Shape Param # == lstm_1 (LSTM) (None, 128) 91648 ____ dense_1 (Dense) (None, 50) 6450 ____ activation_1 (Activation) (None, 50) 0 == Total params: 98,098.0 Trainable params: 98,098.0 Non-trainable params: 0.0 You use a much wider LSTM layer?128, up from 50. >>> epochs = 6 >>> batch_size = 128 >>> model_structure = model.to_json() >>> with open("shakes_lstm_model.json", "w") as json_file: >>> json_file.write(model_structure) >>> for i in range(5): ... model.fit(X, y, ... batch_size=batch_size, ... epochs=epochs) ... model.save_weights("shakes_lstm_weights_{}.h5".format(i+1)) Epoch 1/6 125168/125168 [=============================] - 266s - loss: 2.0310 Epoch 2/6 125168/125168 [=============================] - 257s - loss: 1.6851 ... 305 LSTM   Listing 9.29 Sampler to generate character sequences >>> import random >>> def sample(preds, temperature=1.0): ... preds = np.asarray(preds).astype('float64') ... preds = np.log(preds) / temperature ... exp_preds = np.exp(preds) ... preds = exp_preds / np.sum(exp_preds) ... probas = np.random.multinomial(1, preds, 1) ... return np.argmax(probas) Because the last layer in the network is a softmax, the output vector will be a proba- bility distribution over all possible outputs of the network. Listing 9.30 Generate three texts with three diversity levels >>> import sys >>> start_index = random.randint(0, len(text) - maxlen - 1) >>> for diversity in [0.2, 0.5, 1.0]: ... print() ... print('----- diversity:', diversity) ... generated = '' ... sentence = text[start_index: start_index + maxlen]   306 CHAPTER 9 Improving retention with long short-term memory networks ... generated += sentence ... print('----- Generating with seed: "' + sentence + '"') ... sys.stdout.write(generated) ... for i in range(400): ... x = np.zeros((1, maxlen, len(chars))) ... for t, char in enumerate(sentence): ... x[0, t, char_indices[char]] = 1.

9.1.9 Other kinds of memory

Listing 9.31 Gated recurrent units in Keras >>> from keras.models import Sequential >>> from keras.layers import GRU >>> model = Sequential() >>> model.add(GRU(num_neurons, return_sequences=True, ... input_shape=X[0].shape)) Another technique is to use an LSTM with peephole connections.

9.1.10 Going deeper

Listing 9.32 Two LSTM layers >>> from keras.models import Sequential >>> from keras.layers import LSTM >>> model = Sequential() >>> model.add(LSTM(num_neurons, return_sequences=True, ... input_shape=X[0].shape)) >>> model.add(LSTM(num_neurons_2, return_sequences=True))

Summary

? Remembering information with memory units enables more accurate and gen- eral models of the sequence. ? Only some new information needs to be retained for the upcoming input, and LSTMs can be trained to find it. ? If you can predict what comes next, you can generate novel text from probabilities. ? Character-based models can more efficiently and successfully learn from small, focused corpora than word-based models.

Sequence-to-sequence models and attention

This chapter covers ? Mapping one text sequence to another with a neural network ? Understanding sequence-to-sequence tasks and how they?re different from the others you?ve learned about ? Using encoder-decoder model architectures for translation and chat ? Training a model to pay attention to what is important in a sequence You now know how to create natural language models and use them for everything from sentiment classification to generating novel text (see chapter 9).

10.1 Encoder-decoder architecture

The first half of an encoder-decoder model is the sequence encoder, a network which turns a sequence, such as natural language text, into a lower-dimensional repre- sentation, such as the thought vector from the end of chapter 9.

10.1.1 Decoding thought

Figure 10.1 Limitations of language modeling   314 CHAPTER 10 Sequence-to-sequence models and attention ?playing? would then need to map to ?Fußball.? Certainly a network could learn these mappings, but the learned representations would have to be hyper-specific to the input, and your dream of a more general language model would go out the window. Sequence-to-sequence networks, sometimes abbreviated with seq2seq, solve this lim- itation by creating an input representation in the form of a thought vector. Sequence- to-sequence models then use that thought vector, sometimes called a context vector, as a starting point to a second network that receives a different set of inputs to gener- ate the output sequence. The thought vector has two parts, each a vector: the output (activation) of the hidden layer of the encoder and the memory state of the LSTM cell for that input example. Figure 10.2 Encoder-decoder sandwich with thought vector meat 2 See the web page titled ?Deep Learning,? (https://www.evl.uic.edu/creativecoding/courses/cs523/slides/ week3/DeepLearning_LeCun.pdf).

10.1.2 Look familiar?

They are a repeat-game-playing neural net that?s trained to 0.2 ? 0.4 0.1 0.7 ? 0.4 0.1 ... 0.0 Output Input Encoder Decoder sequence sequence Thought vector Figure 10.3 Unrolled encoder-decoder

10.1.3 Sequence-to-sequence conversation

It may not be clear how the dialog engine (conversation) problem is related to machine translation, but they?re quite similar. 4 See the web page titled ?Variational Autoencoders Explained? (http://kvfrans.com/variational-autoencoders- explained).

10.1.4 LSTM review

In the last chapter, you learned how an LSTM gives recurrent nets a way to selectively remember and forget patterns of tokens they have ?seen? within a sample document. The input token for each time step passes through the forget and update gates, is mul- tiplied by weights and masks, and then is stored in a memory cell. The network output at that time step (token) is dictated not solely by the input token, but also by a combi- nation of the input and the memory unit?s current state. Importantly, an LSTM shares that token pattern recognizer between documents, because the forget and update gates have weights that are trained as they read many documents. And you learned how to activate these token patterns stored in 5 Also called the ?repeat game,? http://uncyclopedia.wikia.com/wiki/Childish_Repeating_Game.

10.2.1 Preparing your dataset for the sequence-to-sequence training

Figure 10.5 Input and target sequence before preprocessing In addition to the required padding, the output sequence should be annotated with the start and stop tokens, to tell the decoder when the job starts and when it?s done (see figure 10.6). Just keep in mind that you?ll need two versions of the target sequence for training: one that starts with the start token (which you?ll use for the decoder input), and one that starts without the start token (the target sequence the loss function will score for accuracy). Each training example for the sequence-to-sequence model will be a triplet: initial input, expected output (prepended by a start token), and expected output (without the start token). With the initialized state and a start token as input to the decoder network, you?ll then generate the first sequence element (such as a character or word vector) of the output.

10.2.3 Sequence encoder

>>> encoder_inputs = Input(shape=(None, input_vocab_size)) >>> encoder = LSTM(num_neurons, return_state=True) >>> encoder_outputs, state_h, state_c = encoder(encoder_inputs) >>> encoder_states = (state_h, state_c) The first return value of the LSTM layer is the output of the layer.

10.2.4 Thought decoder

You want the decoder to learn to reproduce the tokens of a given input sequence given the state generated by the first piece of the 3-tuple fed into the encoder. <START> Maria spielt Fußball Decoder Decoder Decoder Decoder Thought vector LSTM LSTM LSTM LSTM Maria spielt Fußball <END> Figure 10.9 Thought decoder

10.2.5 Assembling the sequence-to-sequence network

For this sequence-to- sequence network, you?ll pass a list of your inputs to the model. As an output layer, you?re passing the decoder_outputs to the model, which includes the entire model setup you previously defined.

10.3 Training the sequence-to-sequence network

The model must choose between all possible tokens to ?say.? Because you?re predict- ing characters or words rather than binary states, you?ll optimize your loss based on the categorical_crossentropy loss function, rather than the binary_ crossentropy used earlier. Listing 10.4 Train a sequence-to-sequence model in Keras >>> model.compile(optimizer='rmsprop', loss='categorical_crossentropy') >>> model.fit([encoder_input_data, decoder_input_data], decoder_target_data, batch_size=batch_size, epochs=epochs) Setting the loss function to categorical_ crossentropy.

10.3.1 Generate output sequences

Listing 10.5 Decoder for generating text using the generic Keras Model >>> encoder_model = Model(inputs=encoder_inputs, outputs=encoder_states) Here you use the previously defined encoder_inputs and encoder_states; calling the predict method on this model would return the thought vector. Input(shape=(num_neurons,))] >>> decoder_outputs, state_h, state_c = decoder_lstm(... decoder_inputs, initial_state=thought_input) >>> decoder_states = [state_h, state_c] >>> decoder_outputs = decoder_dense(decoder_outputs) Pass the encoder state to the LSTM layer as initial state. >>> decoder_model = Model(... inputs=[decoder_inputs] + thought_input, ... output=[decoder_outputs] + decoder_states) The decoder_inputs and thought_input become the input to the decoder model.

10.4.1 Preparing the corpus for your training

For this example, you?ve preprocessed the dialog corpus by limiting samples to those with fewer than 100 characters, removed odd characters, and only allowed 10 See the web page titled ?Cornell Movie-Dialogs Corpus? (https://www.cs.cornell.edu/~cristian/ Cornell_Movie-Dialogs_Corpus.html).

10.4.2 Building your character dictionary

Similar to the examples from your previous chapters, you need to convert each charac- ter of the input and target texts into one-hot vectors that represent each character. In order to generate the one-hot vectors, you generate token dictionaries (for the input and target text), where every character is mapped to an index. You also generate the reverse dictionary (index to character), which you?ll use during the generation phase to convert the generated index to a character.

10.4.3 Generate one-hot encoded training sets

>>> encoder_input_data = np.zeros((len(input_texts), ... max_encoder_seq_length, input_vocab_size), ... dtype='float32') >>> decoder_input_data = np.zeros((len(input_texts), ... max_decoder_seq_length, output_vocab_size), ... dtype='float32') >>> decoder_target_data = np.zeros((len(input_texts), ... max_decoder_seq_length, output_vocab_size), ... dtype='float32') The training tensors are initialized as zero tensors with shape (num_samples, max_len_sequence, num_unique_tokens_in_vocab). >>> for i, (input_text, target_text) in enumerate(... zip(input_texts, target_texts)): Loop over the training samples; input and target texts need to correspond.

10.4.4 Train your sequence-to-sequence chatbot

>>> encoder_inputs = Input(shape=(None, input_vocab_size)) >>> encoder = LSTM(num_neurons, return_state=True) >>> encoder_outputs, state_h, state_c = encoder(encoder_inputs) >>> encoder_states = [state_h, state_c] >>> decoder_inputs = Input(shape=(None, output_vocab_size)) >>> decoder_lstm = LSTM(num_neurons, return_sequences=True, ... return_state=True) >>> decoder_outputs, _, _ = decoder_lstm(decoder_inputs, ... initial_state=encoder_states) >>> decoder_dense = Dense(output_vocab_size, activation='softmax') >>> decoder_outputs = decoder_dense(decoder_outputs) >>> model = Model([encoder_inputs, decoder_inputs], decoder_outputs) You withhold 10% of the samples for validation tests after each epoch. >>> model.compile(optimizer='rmsprop', loss='categorical_crossentropy', ... metrics=['acc']) >>> model.fit([encoder_input_data, decoder_input_data], ... decoder_target_data, batch_size=batch_size, epochs=epochs, ... validation_split=0.1)

10.4.6 Predicting a sequence

>>> def decode_sequence(input_seq): ... thought = encoder_model.predict(input_seq) ... target_seq = np.zeros((1, 1, output_vocab_size)) ... target_seq[0, 0, target_token_index[stop_token] ...] = 1. Passing the already-generated tokens and the latest state to the decoder to predict the next sequence element ... while not stop_condition: ... output_tokens, h, c = decoder_model.predict(... [target_seq] + thought) ... generated_token_idx = np.argmax(output_tokens[0, -1, :]) ... generated_char = reverse_target_char_index[generated_token_idx] ... generated_sequence += generated_char ... if (generated_char == stop_token or ... len(generated_sequence) > max_decoder_seq_length ...): ... stop_condition = True Setting the stop_condition to True will stop the loop.

10.4.8 Converse with your chatbot

After 100 epochs of training, which took approximately seven and a half hours on an NVIDIA GRID K520 GPU, the trained sequence-to-sequence chatbot was still a bit stubborn and short-spoken. A larger and more general training corpus could change that behavior: >>> response("what is the internet?") Bot Reply (Decoded sentence): it's the best thing i can think of anything. Bot Reply (Decoded sentence): i don't know.

10.5.1 Reduce training complexity with bucketing

Input sequences can have different lengths, which can add a large number of pad tokens to short sequences in your training data. You assign the input sequences to buckets of different lengths, such as all sequences with a length between 5 and 10 tokens, and then use the sequence buckets for your training batches, such as train first with all sequences between 5 and 10 tokens, 10 to 15, and so on.

10.5.2 Paying attention

As with latent semantic analysis introduced in chapter 4, longer input sequences (doc- uments) tend to produce thought vectors that are less precise representations of those documents.

10.6 In the real world

335 In the real world Key sequence-to-sequence applications are ? Chatbot conversations ? Question answering ? Machine translation ? Image captioning ? Visual question answering ? Document summarization As you?ve seen in the previous sections, a dialog system is a common application for NLP. Sequence-to-sequence models are generative, which makes them especially well- suited to conversational dialog systems (chatbots). In contrast, the ?grounding? of knowledge-based dialog systems (discussed in chapter 12) can limit their ability to participate in conversations on top- ics outside their training domains. Besides the Cornell Movie Dialog Corpus, various free and open source training sets are available, such as Deep Mind?s Q&A datasets.15, 16 When you need your dialog system to respond reliably in a specific domain, you?ll need to train it on a corpora of statements from that domain.

Summary

? The encoder model generates a thought vector, a dense, fixed-dimension vector representation of the information in a variable-length input sequence. ? A decoder can use thought vectors to predict (generate) output sequences, including the replies of a chatbot. ? Due to the thought vector representation, the input and the output sequence lengths don?t have to match.

11.1 Named entities and relations

For example, imagine a user says ?Remind me to read aiindex.org on Monday.? You?d like that statement to trigger a calendar entry or alarm for the next Monday after the current date.

11.1.1 A knowledge base

An example of ?self-knowledge? a smart chatbot should keep track of is the history of all the things it has already told someone or the questions it has already asked of the user, so it doesn?t repeat itself. This could be stored in a data structure something like this: ('Stanislav Petrov', 'is-a', 'lieutenant colonel') This is an example of two named entity nodes ('Stanislav Petrov' and 'lieutenant colo- nel') and a relation or connection ('is a') between them in a knowledge graph or knowledge base. military rank is-a Lieutenant Colonel is-a is-member is-a Stanislov organization Petrov Soviet Air Defense Force is-a person Figure 11.1 Stanislav knowledge graph   342 CHAPTER 11 Information extraction (named entity extraction and question answering) For this particular inference or query about Stanislov?s military rank, your knowledge graph would have to already contain facts about militaries and military ranks. And some of that knowledge is instinct, hard-coded into our DNA.2 All kinds of factual relationships exist between things and people, such as ?kind- of,? ?is-used-for,? ?has-a,? ?is-famous-for,? ?was-born,? and ?has-profession.? NELL, the Carnegie Mellon Never Ending Language Learning bot, is focused almost entirely on the task of extracting information about the ?kind-of? relationship. Customer service chatbots, including university TA bots, rely almost exclusively on knowledge bases to generate their replies.3 Question answering systems are great for helping humans find factual information, which frees up human brains to do the things they?re better at, such as attempting to generalize 1 See chapter 4 if you?ve forgotten about how random topic allocation can be.

11.2 Regular patterns

Listing 11.1 Pattern hardcoded in Python >>> def find_greeting(s): ... """ Return greeting str (Hi, etc) if greeting pattern matches """ ... if s[0] == 'H': ... if s[:3] in ['Hi', 'Hi ', 'Hi,', 'Hi!

11.2.1 Regular expressions

Regular expressions define a finite state machine or FSM?a tree of ?if-then? deci- sions about a sequence of symbols, such as the find_greeting() function in listing 11.1. In computer science and mathematics, the word ?grammar? refers to the set of rules that determine whether or not a sequence of symbols is a valid member of a language, often called a computer language or formal language.

11.2.2 Information extraction as ML feature extraction

Plus, as you?ll see here, you can define a compact set of condition checks (a regular expression) to extract key bits of information from a natural language string. Information extraction is just another form of machine learning feature extraction from unstructured natural language data, such as creating a bag of words, or doing PCA on that bag of words. And these patterns and features are still employed in even the most advanced natural language machine learning pipelines, such as Google?s Assistant, Siri, Amazon Alexa, and other state-of-the-art bots. If information is retrieved in real-time, as the chatbot is being queried, this is often called ?search.? Google and other search engines combine these two techniques, querying a knowledge graph (knowl- edge base) and falling back to text search if the necessary facts aren?t found.

11.3.2 Extracting dates

Listing 11.5 Structuring extracted dates >>> dates = [{'mdy': x[0], 'my': x[1], 'm': int(x[2]), 'd': int(x[3]), ... 'y': int(x[4].lstrip('/') or 0), 'c': int(x[5] or 0)} for x in mdy] >>> dates [{'mdy': '12/25/2017', 'my': '12/25', 'm': 12, 'd': 25, 'y': 2017, 'c': 20}, {'mdy': '12/12', 'my': '12/12', 'm': 12, 'd': 12, 'y': 0, 'c': 0}] Even for these simple dates, it?s not possible to design a regex that can resolve all the ambiguities in the second date, ?12/12.? There are ambiguities in the language of dates that only humans can guess at resolving using knowledge about things like Christmas and the intent of the writer of a text. Listing 11.6 Basic context maintenance >>> for i, d in enumerate(dates): ... for k, v in d.items(): ... if not v: ... d[k] = dates[max(i - 1, 0)][k] >>> dates [{'mdy': '12/25/2017', 'my': '12/25', 'm': 12, 'd': 25, 'y': 2017, 'c': 20}, {'mdy': '12/12', 'my': '12/12', 'm': 12, 'd': 12, 'y': 2017, 'c': 20}] >>> from datetime import date >>> datetimes = [date(d['y'], d['m'], d['d']) for d in dates] >>> datetimes [datetime.date(2017, 12, 25), datetime.date(2017, 12, 12)] This works because both the dict and the list are mutable data types. Listing 11.8 Recognizing years >>> yr_19xx = (... r'\b(?P<yr_19xx>' + ... '|'.join('{}'.format(i) for i in range(30, 100)) + ... r')\b' ...) >>> yr_20xx = (... r'\b(?P<yr_20xx>' + ... '|'.join('{:02d}'.format(i) for i in range(10)) + '|' + ... '|'.join('{}'.format(i) for i in range(10, 30)) + 2-digit years 30-99 = 1930-1999 6 See the web page titled ?Year zero? (https://en.wikipedia.org/wiki/Year_zero). 351 Information worth extracting 1- or 2-digit years 01-30 = 2001-2030 First digits of a 3- or 4-digit year such as   the ?1? in ?123 A.D.?   or ?20? in ?2018? ... r')\b' ...) >>> yr_cent = r'\b(?P<yr_cent>' + '|'.join(... '{}'.format(i) for i in range(1, 40)) + r')' >>> yr_ccxx = r'(?P<yr_ccxx>' + '|'.join(... '{:02d}'.format(i) for i in range(0, 100)) + r')\b' >>> yr_xxxx = r'\b(?P<yr_xxxx>(' + yr_cent + ')(' + yr_ccxx + r'))\b' >>> yr = (... r'\b(?P<yr>' + ... yr_19xx + '|' + yr_20xx + '|' + yr_xxxx + ... r')\b' ...) >>> groups = list(re.finditer(... yr, "0, 2000, 01, '08, 99, 1984, 2030/1970 85 47 `66")) >>> full_years = [g['yr'] for g in groups] >>> full_years ['2000', '01', '08', '99', '1984', '2030', '1970', '85', '47', '66'] Last 2 digits of a 3- or 4-digit year such as the ?23? in ?123 A.D.? or ?18? in ?2018? Wow! Listing 11.9 Recognizing month words with regular expressions >>> mon_words = 'January February March April May June July ' \ ... 'August September October November December' >>> mon = (r'\b(' + '|'.join('{}|{}|{}|{}|{:02d}'.format(... m, m[:4], m[:3], i + 1, i + 1) for i, m in ? enumerate(mon_words.split())) + ... r')\b') >>> re.findall(mon, 'January has 31 days, February the 2nd month ? of 12, has 28, except in a Leap Year.')

11.4 Extracting relationships (relations)

So far you?ve looked only at extracting tricky noun instances such as dates and GPS latitude and longitude values. It?s time to tackle the harder problem of extracting knowledge from natural language.

11.4.1 Part-of-speech (POS) tagging

>>> ' '.join(['{}_{}'.format(tok, tok.tag_) for tok in parsed_sent]) 'In_IN 1541_CD Desoto_NNP wrote_VBD in_IN his_PRP$ journal_NN that_IN the_DT Pascagoula_NNP people_NNS   354 CHAPTER 11 Information extraction (named entity extraction and question answering) ranged_VBD as_RB far_RB north_RB as_IN the_DT confluence_NN of_IN the_DT Lea f_NNP and_CC Chickasawhay_NNP rivers_VBZ at_IN 30.4_CD ,_, -88.5_NFP ._.' >>> parsed_sent = en_model(sentence) >>> with open('pascagoula.html', 'w') as f: ... f.write(render(docs=parsed_sent, page=True, ? options=dict(compact=True))) The dependency tree for this short sentence shows that the noun phrase ?the Pasca- goula? is the object of the relationship ?met? for the subject ?Desoto? (see figure 11.2). Listing 11.14 Helper functions for spaCy tagged strings >>> import pandas as pd >>> from collections import OrderedDict   355 Extracting relationships (relations) >>> def token_dict(token): ... return OrderedDict(ORTH=token.orth_, LEMMA=token.lemma_, ... POS=token.pos_, TAG=token.tag_, DEP=token.dep_) >>> def doc_dataframe(doc): ... return pd.DataFrame([token_dict(tok) for tok in doc]) >>> doc_dataframe(en_model("In 1541 Desoto met the Pascagoula.")) >>> matcher = Matcher(en_model.vocab) >>> matcher.add('met', None, pattern) >>> m = matcher(doc) >>> m [(12280034159272152371, 2, 6)] >>> doc[m[0][1]:m[0][2]] Desoto met the Pascagoula   356 CHAPTER 11 Information extraction (named entity extraction and question answering) So you extracted a match from the original sentence from which you created the pat- tern, but what about similar sentences from Wikipedia? >>> m = matcher(doc)[0] >>> m (12280034159272152371, 3, 11) >>> doc[m[1]:m[2]] Lewis and Clark met their first Mandan Chief >>> doc = en_model("On 11 October 1986, Gorbachev and Reagan met at a house") >>> matcher(doc) [] The pattern doesn?t match any substrings of the sentence from Wikipedia. Listing 11.18 Combining multiple patterns for a more robust pattern matcher >>> doc = en_model("On 11 October 1986, Gorbachev and Reagan met at a house") >>> pattern = [{'TAG': 'NNP', 'OP': '+'}, {'LEMMA': 'and'}, {'TAG': 'NNP', 'O P': '+'}, ... {'IS_ALPHA': True, 'OP': '*'}, {'LEMMA': 'meet'}] >>> matcher.add('met', None, pattern) >>> m = matcher(doc) >>> m [(14332210279624491740, 5, 9), (14332210279624491740, 5, 11), (14332210279624491740, 7, 11), (14332210279624491740, 5, 12)] Adds an additional pattern without removing the previous pattern.

11.4.2 Entity name normalization

For example ?Desoto? might be expressed in a particular document in at least five different ways: ? ?de Soto? ? ?Hernando de Soto? ? ?Hernando de Soto (c. 1496/1497?1542), Spanish conquistador? ? https://en.wikipedia.org/wiki/Hernando_de_Soto (a URI) ? A numerical ID for a database of famous and historical people Similarly your normalization algorithm can choose any of these forms.

11.4.4 Word patterns

You can use the spaCy package two different ways to match these patterns in O(1) (constant time) no matter how many patterns you want to match: ? PhraseMatcher for any word/tag sequence patterns9 ? Matcher for POS tag sequence patterns10 To ensure that the new relations found in new sentences are truly analogous to the original seed (example) relationships, you often need to constrain the subject, rela- tion, and object word meanings to be similar to those in the seed sentences.

11.4.5 Segmentation

Document ?chunking? is useful for creating semi-structured data about documents that can make it easier to search, filter, and sort documents for information retrieval. And for information extraction, if you?re extracting relations to build a knowledge base such as NELL or Freebase, you need to break it into parts that are likely to con- tain a fact or two. And grammatically correct English language sentences must contain a subject (noun) and a verb, which means they?ll usually have at least one relation or fact worth extracting. In addition to facilitating information extraction, you can flag some of those state- ments and sentences as being part of a dialog or being suitable for replies in a dialog.

11.4.6 Why won?t split('.!?') work?

Technical text is particularly difficult to segment into sentences, because engineers, scientists, and mathematicians tend to use periods and exclamation points to signify a 12 See the web page titled ?Natural Language Processing: TM-Town? (https://www.tm-town.com/natural -language-processing#golden_rules).

11.4.7 Sentence segmentation with regular expressions

]+)[$]+') >>> examples = get_data('sentences-tm-town') >>> wrong = [] >>> for i, (challenge, text, sents) in enumerate(examples): ... if tuple(regex.split(text)) != tuple(sents): ... print('wrong {}: {}{}'.format(i, text[:50], '...' if len(text) > 50 else '')) ... wrong += [i] >>> len(wrong), len(examples) (61, 61) You?d have to add a lot more ?look-ahead? and ?look-back? to improve the accuracy of a regex sentence segmenter.

11.5 In the real world

Information extraction and question answering systems are used for ? TA assistants for university courses ? Customer service ? Tech support ? Sales ? Software documentation and FAQs Information extraction can be used to extract things such as ? Dates ? Times ? Prices ? Quantities ? Addresses ? Names ? People ? Places ? Apps ? Companies ? Bots ? Relationships ? ?is-a? (kinds of things) ? ?has? (attributes of things) ? ?related-to? Whether information is being parsed from a large corpus or from user input on the fly, being able to extract specific details and store them for later use is critical to the per- formance of a chatbot.

Summary

? A knowledge graph can be built to store relationships between entities. ? Regular expressions are a mini-programming language that can isolate and extract information. ? Part-of-speech tagging allows you to extract relationships between entities men- tioned in a sentence. ? Segmenting sentences requires more than just splitting on periods and excla- mation marks.

12 Getting chatty (dialog engines)Getting chatty 30.0

There are several cash prize competitions, if you think you and your chatbot have the right stuff: ? The Alexa Prize ($3.5M)1 ? Loebner Prize ($7k)2 ? The Winograd Schema Challenge ($27k)3 ? The Marcus Test4 ? The Lovelace Test5 Beyond the pure fun and magic of building a conversational machine, beyond the glory that awaits you if you build a bot that can beat humans at an IQ test, beyond the warm fuzzy feeling of saving the world from malicious hacker botnets, and beyond the wealth that awaits you if you can beat Google and Amazon at their virtual assistant games?the techniques you?ll learn in this chapter will give you the tools you need to get the job done. Some of the NLP skills you?ll use include ? Tokenization, stemming, and lemmatization ? Vector space language models such as bag-of-words vectors or topic vectors (semantic vectors) ? Deeper language representations such as word vectors or LSTM thought vectors ? Sequence-to-sequence translators (from chapter 10) ? Pattern matching (from chapter 11) ? Templates for generating natural language text 1 ?The Alexa Prize,? https://developer.amazon.com/alexaprize.

12.1.1 Modern approaches

Matching patterns in text and populating canned-response templates with informa- tion extracted with those patterns is only one of four modern approaches to building chatbots: ? Pattern matching?Pattern matching and response templates (canned responses) ? Grounding?Logical knowledge graphs and inference on those graphs 6 Wikipedia ?Canned Response,? https://en.wikipedia.org/wiki/Canned_response. Here?s a list of a few of these chatbot applications; you may notice that the more advanced chatbots, such as Siri, Alexa, and Allo, are listed alongside multiple types of problems and applications: ? Question answering?Google Search, Alexa, Siri, Watson ? Virtual assistants?Google Assistant, Alexa, Siri, MS paperclip ? Conversational?Google Assistant, Google Smart Reply, Mitsuki Bot ? Marketing?Twitter bots, blogger bots, Facebook bots, Google Search, Google Assistant, Alexa, Allo ? Customer service?Storefront bots, technical support bots ? Community management?Bonusly, Slackbot ? Therapy?Woebot, Wysa, YourDost, Siri, Allo Can you think of ways to combine the four basic dialog engine types to create chatbots for these seven applications? So the TurboTax wizard can?t really be called a chatbot yet, but it?ll surely be wrapped in a chat interface soon, if the tax- bot AskMyUncleSam takes off.10 Lawyer virtual assistant chatbots have successfully appealed millions of dollars in parking tickets in New York and London.11 And there?s even a United Kingdom law firm where the only interaction you?ll ever have with a lawyer is through a chatbot.12 Lawyers are certainly goal-based virtual assistants, only they?ll do more than set an appointment date: they?ll set you a court date and maybe help you win your case. Chloe gives blind and low-vision people access to a ?visual interpreter for the blind.? During onboard- ing, Chloe can ask customers things such as ?Are you a white cane user?? ?Do you have a guide dog?? and ?Do you have any food allergies or dietary preferences you?d like us to know about?? This is called voice first design, when your app is designed from the 10 Jan 2017, Venture Beat post by AskMyUncleSam: https://venturebeat.com/2017/01/27/how-this-chatbot-powered-by-machine-learning-can-help-with-your-taxes/. 12 Nov 2017, ?Chatbot-based ?firm without lawyers? launched? blog post by Legal Futures: https://www.legalfutures.co.uk/latest-news/chatbot-based-firm-without-lawyers-launched. More and more video games, movies, and TV shows are launched with chatbots on websites promoting them: 17 ? HBO promoted ?Westworld? with ?Aeden.?18 ? Sony promoted ?Resident Evil? with ?Red Queen.?19 ? Disney promoted ?Zootopia? with ?Officer Judy Hopps.?20 ? Universal promoted ?Unfriended? with ?Laura Barnes.? ? Activision promoted ?Call of Duty? with ?Lt. 21 Wikipedia article about the brief ?life? of Microsoft?s Tay chatbot, https://en.wikipedia.org/wiki/Tay_(bot). But because chatbots are active participants, imbued with motivations by developers like you, you shouldn?t dismiss them as merely ?mirrors of society.? Chatbots seem to do more than merely reflect and amplify the best and the worst of us. And the knowledge base used to ?ground? their answers to reality must be kept cur- rent, enabling customer service chatbots to answer questions about orders or products as well as initiate actions such as placing or canceling orders. They must be informative like a question answering 22 March 2014, George Dvorski, ?Why Asimov?s Three Laws of Robotics Can?t Protect Us,? Gizmodo, https:// io9.gizmodo.com/why-asimovs-three-laws-of-robotics-cant-protect-us-1553665410.

12.2 Pattern-matching approach

Because you have a lot of your NLP tools in Python packages already, you can often build much more complex pattern-matching chatbots just by building up the logic for your bot directly in Python and regular expressions or glob patterns.27 At Aira, we developed a simple glob pattern language similar to AIML to define our patterns.

12.2.1 A pattern-matching chatbot with AIML

In AIML (v2.0), here?s how you might define your greeting chatbot from chapter 1.28 Listing 12.1 nlpia/book/examples/greeting.v2.aiml <?xml version="1.0" encoding="UTF-8"?><aiml version="2.0"> <category> <pattern>HI</pattern> <template>Hi!</template> </category> <category> <pattern>[HELLO HI YO YOH YO'] [ROSA ROSE CHATTY CHATBOT BOT CHATTERBOT]</pattern> <template>Hi , How are you?</template> </category> <category> <pattern>[HELLO HI YO YOH YO' 'SUP SUP OK HEY] [HAL YOU U YALL Y'ALL YOUS YOUSE]</pattern> <template>Good one.</template> </category> </aiml> We used some of the new features of AIML 2.0 (by Bot Libre) to make the XML a little more compact and readable. 378 CHAPTER 12 Getting chatty (dialog engines) Listing 12.5 nlpia/data/greeting_step2.aiml <category><pattern>HELO * </pattern><template><srai>HELLO <star/> </srai></template></category> <category><pattern>HI * </pattern><template><srai>HELLO <star/> </srai></template></category> <category><pattern>HIYA * </pattern><template><srai>HELLO <star/> </srai></template></category> <category><pattern>HYA * </pattern><template><srai>HELLO <star/> </srai></template></category> <category><pattern>HY * </pattern><template><srai>HELLO <star/> </srai></template></category> <category><pattern>HEY * </pattern><template><srai>HELLO <star/> </srai></template></category> <category><pattern>WHATS UP * </pattern><template><srai>HELLO <star/> </srai></template></category> <category><pattern>WHAT IS UP * </pattern><template><srai>HELLO <star/> </srai></template></category> If you are writing your own AIML files, don?t forget to include the NOTE <aiml> tags at the beginning and end. Listing 12.7 nlpia/nlpia/data/greeting_step3.aiml <category><pattern>HELLO ROSA </pattern><template> <random> <li>Hi Human!</li> <li>Hello friend</li> <li>Hi pal</li> <li>Hi!</li> <li>Hello!</li> 379 Pattern-matching approach <li>Hello to you too!</li> <li>Greetings Earthling ;)</li> <li>Hey you :)</li> <li>Hey you!</li> </random></template> </category> <category><pattern>HELLO TROLL </pattern><template> <random> <li>Good one, Human.</li> <li>Good one.</li> <li>Nice one, Human.</li> <li>Nice one.</li> <li>Clever.</li> <li>:)</li> </random></template> </category> Now your chatbot doesn?t sound nearly as mechanical (at least at the beginning of a conversation).

12.2.2 A network view of pattern matching

Aira is working on visualization tools to turn AIRS specs into these graph diagrams (see figure 12.2) with ?Could you repeat that?? Bot action User action User action ?What is your name?? STATE-NAME ?Cool? TASK {turn_down_temp} ?Hi {user_reply}! What can I do for you?? ?Koul? <power_on> ?I need a Lyft? USER? ROOT NAME ?{request_lyft} Your Lyft driver, {lyft_name}, should be here by {lyft_eta}? ?What is your name?? ?Cancel? LYFT ?{cancel_lyft} OK, canceling your Lyft driver request.? Figure 12.2 Managing state (context)

12.3 Grounding

Some examples of open knowledge bases you can use to ground your chatbot include ? Wikidata (includes Freebase)31 ? Open Mind Common Sense (ConceptNet)32 ? Cyc33 ? YAGO34 ? DBpedia35 So all you need is a way to query the knowledge base to extract the facts you need to populate a response to a user?s statement. And if the user is asking a factual question that your database might contain, you could translate their natural language question (such as ?Who are you?? or ?What is the 50th state of the United States??) into a knowledge base query to directly retrieve the answer they?re looking for. The SQL-equivalent for a knowledge graph of RDF triples is called SPARQL.37 31 See the web page titled ?Welcome to Wikidata? (https://www.wikidata.org).

12.4.1 The context challenge

For example, what if someone asked your chatbot ?What time is it?? Your chatbot shouldn?t reuse the reply of the human who replied to the best-matched statement in your database. Your chatbot?s personality profile information could be used to resolve ?ties? in the search for matching statements in your database. You need to use one of the other chatbot techniques rather than retrieval if your database of state- ments and replies doesn?t contain a lot of answers to questions such as ?Who are you?? ?Where are you from?? and ?What?s your favorite color?? If you don?t have a lot of profile statement-reply pairs, you?d need to detect any questions about the bot and use a knowl- edge base to ?infer? an appropriate answer for that element of the statement.

12.4.2 Example retrieval-based chatbot

Listing 12.9 ch12_retrieval.py >>> from nlpia.data.loaders import get_data >>> df = get_data('ubuntu_dialog') Downloading ubuntu_dialog requesting URL: https://www.dropbox.com/s/krvi79fbsryytc2/ubuntu_dialog.csv.gz?dl=1 remote size: 296098788 38 ?The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems? by Lowe et al., 2015 https://arxiv.org/abs/1506.08909. Listing 12.11 ch12_retrieval.py >>> import re >>> def split_turns(s, splitter=re.compile('__eot__')): ... for utterance in splitter.split(s): ... utterance = utterance.replace('__eou__', '\n') ... utterance = utterance.replace('__eot__', '').strip() ... if len(utterance): ... yield utterance   388 CHAPTER 12 Getting chatty (dialog engines) Let?s run that split_turns function on a few rows in the DataFrame to see if it makes sense. Listing 12.12 ch12_retrieval.py >>> for i, record in df.head(3).iterrows(): ... statement = list(split_turns(record.Context))[-1] ... reply = list(split_turns(record.Utterance))[-1] ... print('Statement: {}'.format(statement)) ... print() ... print('Reply: {}'.format(reply)) This should print out something like this: Statement: I would prefer to avoid it at this stage. Listing 12.13 ch12_retrieval.py >>> from tqdm import tqdm >>> def preprocess_ubuntu_corpus(df): ... """ ... Split all strings in df.Context and df.Utterance on ... __eot__ (turn) markers ... """ ... statements = [] ... replies = [] ... for i, record in tqdm(df.iterrows()): ... turns = list(split_turns(record.Context)) ... statement = turns[-1] if len(turns) else '\n' ... statements.append(statement) ... turns = list(split_turns(record.Utterance)) You need an if because some of the statements and replies contained only whitespace.

12.4.3 A search-based chatbot

It?s easy to install (just pip install ChatterBot), and it comes with several conversation corpora that you can use to ?train? your chatbot to carry on basic conversations. If you want to be more precise with the personality of your chatbot, you?ll need to create your own corpus in the ChatterBot ?.yml? format. How ChatterBot?s ?training? data (actually just a dialog corpus) is stored in a rela- tional database is shown in the following listing.

12.5 Generative models

But if you recall the sequence-to- sequence models you built in chapter 10, you may recognize them as generative chat- bots. So we don?t go into generative models in any more detail here, but know that many more kinds of generative models are out there.

12.5.1 Chat about NLPIA

The DeepMind folks have provided TensorFlow character sequence-to-sequence language models pretrained on more than 500MB of sentences from CNN and Daily Mail news feeds.44 And if you want to build your own language model, they?ve pro- vided all the sentences in two large datasets as part of their ?reading comprehension? (Q&A) challenge.45 We reused the pretrained text summarization model directly to generate sentences for the ?about this book? NLPIA front matter.

12.5.2 Pros and cons of each approach

Approach Advantages Disadvantages Grammar Easy to get started Training easy to reuse Modular Easily controlled/restrained Limited ?domain? Capability limited by human effort Difficult to debug Rigid, brittle rules Grounding Answers logical questions well Easily controlled/restrained Sounds artificial, mechanical Difficulty with ambiguity Difficulty with common sense Limited by structured data Requires large scale information extraction Requires human curation Retrieval Simple Easy to ?train? Can mimic human dialog Difficult to scale Incoherent personality Ignorant of context Can?t answer factual questions Generative New, creative ways of talking Less human effort Domain limited only by data Context aware Difficult to ?steer? Difficult to train Requires more data (dialog) Requires more processing to train Figure 12.3 Advantages and disadvantages of four chatbot approaches 46 Thank you Kyle Gorman @wellformedness (https://twitter.com/wellformedness) for your 100+ suggestions and bits of clever content for this book.

12.6.1 The Will to succeed

47 We?re building an open source chatbot framework at Aira called aichat to help our users and their friends contribute ?content? to our library of dialog to help and entertain people with blindness and low vision: http://github.com/aira/aichat.

12.7 Design process

How are you?? ? ?How are you?? => ?I?m a stateless bot, so I don?t have an emotional state.? ? ?Who won the 2016 World Series?? => ?Chicago Cubs? ? ?Who won the 2016 World Series?? => ?The Chicago Cubs beat the Cleveland Indians 4 to 3? ? ?What time is it? => ?2:55 pm? ? ?When is my next appointment?? => ?At 3 pm you have a meeting with the sub- ject 'Springboard call'? ? ?When is my next appointment?? => ?At 3 pm you need to help Les with her Data Science course on Springboard? ? ?Who is the President?? => ?Sauli Niinistö? ? ?Who is the President?? => ?Barack Obama? Several valid responses may be possible for any given statement, even for the exact same user and context. Here are some example chatbot user stories that don?t lend themselves well to the template approach: ? ?Where is my home? => ?Your home is 5 minutes away by foot, would you like directions?? ? ?Where am I? => ?You are in SW Portland near Goose Hollow Inn? or ?You are at 2004 SW Jefferson Street? ? ?Who is the President?? => ?Sauli Niinistö? or ?Barack Obama? or ?What coun- try or company ?? ? ?Who won the 2016 World Series?? => ?Chicago Cubs? or ?The Chicago Cubs beat the Cleveland Indians 4 to 3? ? ?What time is it? => ?2:55 pm? or ?2:55 pm, time for your meeting with Joe? or ? And here are some general IQ test questions that are too specific to warrant a pattern- response pair for each variation. Nonetheless, that?s probably how the Mitsuku chatbot was able to get close to the right answer in a recent test by Byron Reese: ? ?Which is larger, a nickel or a dime?? => ?Physically or monetarily?? or ?A nickel is physically larger and heavier but less valuable monetarily.? ? ?Which is larger, the Sun or a nickel?? => ?The Sun, obviously.?50 ? ?What?s a good strategy at Monopoly?? => ?Buy everything you can, and get lucky.? ? ?How should I navigate a corn-row maze?? => ?Keep your hand against one wall of corn and follow it until it becomes an outside wall of the maze.? ? ?Where does sea glass come from?? => ?Garbage? fortunately the polishing of sand and time can sometimes turn human refuse, like broken bottles, into beautiful gemstones.? Even though these cannot be easily translated directly into code, they do translate directly into an automated test set for your NLP pipeline.

12.8.2 Be entertaining

In that sit- uation your chatbot has two choices: 1. admit ignorance, or 2. make up a non sequitur. Your user might enjoy learn- ing a bit about the ?core? of your chatbot if you reveal the size of your database of responses or actions you can handle. The more honest you are the more likely the user is to be kind in return and try to help your chatbot get back on track. Cole Howard found that users would often coax his MNIST-trained handwriting recognizer toward the right answer by redrawing the digits in a more clear way. So for a commercial chatbot, you may want this useless response to be sensational, distracting, flattering, or humorous. And you?ll probably also want to ensure that your responses are randomly selected in a way that a human would consider random.

12.8.6 Getting emotional

Even though Google had access to billions of emails, the paired replies in the Gmail Inbox ?Smart Reply? feature tend to funnel you toward short, generic, bland 52 Humans underestimate the number of repetitions there should be in a random sequence: https://mindmodeling.org/cogsci2014/papers/530/paper530.pdf.

12.9 In the real world

In fact, you?ve probably interacted with such a chat- bot sometime this week: ? Customer service assistants ? Sales assistants ? Marketing (spam) bots ? Toy or companion bots ? Video game AI ? Mobile assistants ? Home automation assistants ? Visual interpreters ? Therapist bots ? Automated email reply suggestions And you?re likely to run across chatbots like the ones you built in this chapter more and more.

Summary

? By combining multiple proven approaches, you can build an intelligent dialog engine. ? Breaking ?ties? between the replies generated by the four main chatbot approaches is one key to intelligence.

13 Scaling up (optimization, parallelization, and batch processing)Scaling up (optimization, parallelization, 30.0

? Batch processing to reduce your memory footprint ? Parallelization to speed up NLP ? Running NLP model training on a GPU In chapter 12, you learned how to use all the tools in your NLP toolbox to build an NLP pipeline capable of carrying on a conversation.

13.2 Optimizing NLP algorithms

Some of the algorithms you?ve looked at in previous chapters have expensive com- plexities, often quadratic O(N2) or higher: ? Compiling a thesaurus of synonyms from word2vec vector similarity ? Clustering web pages based on their topic vectors ? Clustering journal articles or other documents based on topic vectors ? Clustering questions in a Q&A corpus to automatically compose a FAQ All of these NLP challenges fall under the category of indexed search, or k-nearest neighbors (KNN) vector search.

13.2.1 Indexing

This query would find all the ?in Action? Manning titles that start with ?Natural Language.? And there are trigram (trgm) indexes for a lot of data- bases that help you find similar text quickly (in constant time), without even specify- ing a pattern, just specifying a text query that?s similar to what you?re looking for. Conventional database indexes rely on the fact that the objects (documents) they?re indexing are either discrete, sparse, or low dimensional: ? Strings (sequences of characters) are discrete: there are a limited number of characters.

13.2.2 Advanced indexing

Here are some of the Python pack- ages from this competition that have been tested with standard benchmarks for NLP problems at the India Technical University (ITU):1 ? Spotify?s Annoy 2 ? BallTree (using nmslib)3 ? Brute Force using Basic Linear Algebra Subprograms library (BLAS)4 ? Brute Force using Non-Metric Space Library (NMSlib)5 ? Dimension reductiOn and LookuPs on a Hypercube for effIcient Near Neigh- bor (DolphinnPy)6 ? Random Projection Tree Forest (rpforest)7 ? Locality sensitive hashing (datasketch)8 ? Multi-indexing hashing (MIH)9 ? Fast Lookup of Cosine and Other Nearest Neighbors (FALCONN)10 1 ITU comparison of ANN Benchmarks: http://www.itu.dk/people/pagh/SSS/ann-benchmarks/. 2 See the web page titled ?GitHub - spotify/annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk? (https://github.com/spotify/annoy). 3 See the web page titled ?GitHub - nmslib/nmslib: Non-Metric Space Library (NMSLIB): An efficient similarity search library and a toolkit for evaluation of k-NN methods for generic non-metric spaces,? (https:// github.com/searchivarius/nmslib). 5 See the web page titled ?GitHub - nmslib/nmslib: Non-Metric Space Library (NMSLIB): An efficient similarity search library and a toolkit for evaluation of k-NN methods for generic non-metric spaces,? (https:// github.com/searchivarius/NMSLIB). 8 See the web page titled ?GitHub - ekzhu/datasketch: MinHash, LSH, LSH Forest, Weighted MinHash, Hyper- LogLog, HyperLogLog++? (https://github.com/ekzhu/datasketch). 9 See the web page titled ?GitHub - norouzi/mih: Fast exact nearest neighbor search in Hamming distance on binary codes with Multi-index hashing? (https://github.com/norouzi/mih).

13.2.3 Advanced indexing with Annoy

Listing 13.1 Load word2vec vectors >>> from nlpia.loaders import get_data >>> wv = get_data('word2vec') 100%|###############################| 402111/402111 [01:02<00:00, 6455.57it/s] >>> len(wv.vocab), len(wv[next(iter(wv.vocab))]) (3000000, 300) >>> wv.vectors.shape (3000000, 300) If you haven?t already downloaded GoogleNews-vectors-negative300.bin.gz (https://bit.ly/GoogleNews-vectors-negative300) to nlpia/ src/nlpia/bigdata/ then get_data() will download it for you. >>> import numpy as np >>> num_trees = int(np.log(num_words).round(0)) >>> num_trees 15 >>> index.build(num_trees) >>> index.save('Word2vec_euc_index.ann') True >>> w2id = dict(zip(range(len(wv.vocab)), wv.vocab)) round(ln(3000000)) => 15 indexing trees for our 3M vectors?takes a few minutes on a laptop Saves the index to a local file and frees up RAM, but may take several minutes You built 15 trees (approximately the natural log of 3 million), because you have 3 million vectors to search through. >>> wv.vocab['Harry_Potter'].index 9494 >>> wv.vocab['Harry_Potter'].count 2990506 >>> w2id = dict(zip(410 CHAPTER 13 Scaling up (optimization, parallelization, and batch processing) ... wv.vocab, range(len(wv.vocab)))) >>> w2id['Harry_Potter'] 9494 >>> ids = index.get_nns_by_item(... w2id['Harry_Potter'], 11) >>> ids [9494, 32643, 39034, 114813, ..., 113008, 116741, 113955, 350346] >>> [wv.vocab[i] for i in _] >>> [wv.index2word[i] for i in _] ['Harry_Potter', 'Narnia', 'Sherlock_Holmes', 'Lemony_Snicket', 'Spiderwick_Chronicles', 'Unfortunate_Events', 'Prince_Caspian', 'Eragon', 'Sorcerer_Apprentice', 'RL_Stine'] Create a map similar to wv.vocab, mapping the tokens to their index values (integer). Listing 13.6 Top Harry_Potter neighbors with gensim.KeyedVectors index >>> [word for word, similarity in wv.most_similar('Harry_Potter', topn=10)] ['JK_Rowling_Harry_Potter', 'JK_Rowling', 'boy_wizard', 'Deathly_Hallows', 'Half_Blood_Prince', 'Rowling', 'Actor_Rupert_Grint', 'HARRY_Potter', 'wizard_Harry_Potter', 'HARRY_POTTER'] Now that looks like a more relevant top-10 synonym list. Listing 13.7 Build a cosine distance index >>> index_cos = AnnoyIndex(... f=num_dimensions, metric='angular') >>> for i, word in enumerate(wv.index2word): ... if not i % 100000: ... print('{}: {}'.format(i, word)) ... index_cos.add_item(i, wv[word]) 0: </s> 100000: distinctiveness ... 2900000: BOARDED_UP metric='angular' uses the angular (cosine) distance metric to compute your clusters and hashes. Listing 13.9 Harry_Potter neighbors in a cosine distance world >>> ids_cos = index_cos.get_nns_by_item(w2id['Harry_Potter'], 10) >>> ids_cos [9494, 37681, 40544, 41526, 14273, 165465, 32643, 420722, 147151, 28829] >>> [wv.index2word[i] for i in ids_cos] ['Harry_Potter', 'JK_Rowling', 'Deathly_Hallows', 'Half_Blood_Prince', 'Twilight', 'Twilight_saga', 'Narnia', 'Potter_mania', 'Hermione_Granger', 'Da_Vinci_Code'] You?ll not get the same results.

13.2.4 Why use approximate indexes at all?

So you could just throw more RAM and processors at the problem and run some batch training process every night or every weekend to keep your bot?s brain up-to-date.16 Even better, you may be able 16 This is the real-world architecture you used on an N2 document matching problem.

13.2.5 An indexing workaround: discretizing

Listing 13.11 MinMaxScaler for low-dimensional vectors >>> from sklearn.preprocessing import MinMaxScaler >>> real_values = [-1.2, 3.4, 5.6, -7.8, 9.0] >>> >>> scaler = MinMaxScaler() Confine our floats to be between 0.0 and 1.0.

13.3.1 Gensim

As the size and variety of the documents in your corpus grows, you may eventually exceed the RAM capacity of even the largest machines you can rent from a cloud service. Mathema- ticians and computer scientists have had a lot of time to play with it and get it to work out of core, which just means that the objects required to run an algorithm don?t all have to be present in core memory (RAM) at once. Even if you don?t want to parallelize your training pipeline on multiple machines, constant RAM implementations will be required for large datasets. Gensim?s LsiModel is one such out-of-core implementation of singular value decomposition for LSA.18 18 See the web page titled ?gensim: models.lsimodel ? Latent Semantic Indexing? (https://radimrehurek.com/ gensim/models/lsimodel.html).

13.3.2 Graph computing

They contributed research and development time to help Aira and TotalGood paral- lelize our NLP pipelines to assist those who have blindness or low vision: ? Extracting images from videos ? Inference and embedding on pretrained Caffe, PyTorch, Theano, and Tensor- Flow (Keras) models ? SVD on large TF-IDF matrices spanning GB corpora19 19 At SAIS 2008, Santi Adavani explained his optimizations that make SVD faster and more scalable on a Rock- etML HPC platform (databricks.com/speaker/santi-adavani).

13.4.1 Training NLP models on GPUs

GPUs, first introduced in 2007, are designed to parallelize a large number of computational tasks and to access large amounts of memory. They?re designed to handle tasks sequentially at a high speed, and they can access their limited processing mem- ory at a high speed (see figure 13.1). vs. CPU GPU Figure 13.1 Comparison between a CPU and GPU As it turns out, training deep learning models involves various operations that can be parallelized, such as the multiplication of matrices. Similar to graphical animations, which were the initial target market for GPUs, the training of deep learning models is heavily accelerated by parallelized matrix multiplications. If the training is executed on a CPU, each row multiplication 20 Santi Adavani and Vinay Rao (http://www.rocketml.net/) are contributing to the Real-Time Video Descrip- tion project (https://github.com/totalgood/viddesc).

13.4.2 Renting vs. buying

Unless you plan to use your 21 See Apple?s Core ML documentation (https://developer.apple.com/documentation/coreml) or Google?s TensorFlow Lite documentation (https://www.tensorflow.org/mobile/tflite/). 22 See the web page titled ?Keras.js - Run Keras models in the browser? (https://transcranial.github.io/keras-js/ #/).

13.4.3 GPU rental options

GPU options Flexibility started Amazon Web Wide range of GPU options; spot Services (AWS) prices; available in various data cen- ters around the world NVIDIA GRID K520, Tesla M60, Tesla K80, Tesla V100 Medium High Google Cloud Integrates Google Cloud Kubernetes, DialogFlow, Jupyter (colab.research .google.com/notebook) NVIDIA Tesla K80, Medium High Tesla P100 Microsoft Azure Good option if you are using other NVIDIA Tesla K80 Medium High Azure services

13.5 Reducing the memory footprint during model training

Listing 13.12 Error message if your training data exceeds the GPU?s memory Epoch 1/10 Exception in thread Thread-27: Traceback (most recent call last): File "/usr/lib/python2.7/threading.py", line 801, in __bootstrap_inner self.run() File "/usr/lib/python2.7/threading.py", line 754, in run self.__target(*self.__args, **self.__kwargs) File "/usr/local/lib/python2.7/dist-packages/keras/engine/training.py", line 606, in data_generator_task generator_output = next(self._generator) 23 See the web page titled ?TensorFlow Research Cloud? (https://www.tensorflow.org/tfrc/). >>> import numpy as np >>> >>> def training_set_generator(data_store, ... batch_size=32): ... X, Y = [], [] ... while True: ... with open(data_store) as f: ... for i, line in enumerate(f): ... if i % batch_size == 0 and X and Y: ... yield np.array(X), np.array(Y) ... X, Y = [], [] ... x, y = line.split('|') ... X.append(x) ... Y.append(y) >>> >>> data_store = '/path/to/your/data.csv' >>> training_set = training_set_generator(data_store) This opens the training data store and creating the file handler f. Loop over the training data stores content line by line until your entire data has been served as training samples; afterward start from the beginning of the training set. >>> data_store = '/path/to/your/data.csv' >>> model.fit_generator(generator=training_set_generator(data_store, ... batch_size=32),

13.6 Gaining model insights with TensorBoard

If you want to use TensorBoard side-by-side with Keras, you need to install Tensor- Board like any other Python package: pip install tensorboard After the installation is complete, you can now start it up: tensorboard --logdir=/tmp/ After TensorBoard is running, access it in your browser at localhost on port 6006 (http://127.0.0.1:6006) if you train on your laptop or desktop PC.

13.6.1 How to visualize word embeddings

Listing 13.14 Convert an embedding into a TensorBoard projection >>> import os >>> import tensorflow as tf >>> import numpy as np >>> from io import open >>> from tensorflow.contrib.tensorboard.plugins import projector >>> >>> >>> def create_projection(projection_data, ... projection_name='tensorboard_viz', ... path='/tmp/'): ... meta_file = "{}.tsv".format(projection_name) ... vector_dim = len(projection_data[0][1]) ... samples = len(projection_data) ... projection_matrix = np.zeros((samples, vector_dim)) ... ... with open(os.path.join(path, meta_file), 'w') as file_metadata: ... for i, row in enumerate(projection_data): ... label, vector = row[0], row[1] ... projection_matrix[i] = np.array(vector) ... file_metadata.write("{}\n".format(label)) ... ... sess = tf.InteractiveSession() ... ... embedding = tf.Variable(projection_matrix, ... trainable=False, ... name=projection_name) ... tf.global_variables_initializer().run() ... ... saver = tf.train.Saver() ... writer = tf.summary.FileWriter(path, sess.graph) ... ... config = projector.ProjectorConfig() ... embed = config.embeddings.add() The create_projection function takes three arguments: the embedding data, a name for the projection and a path, and where to store the projection files. Once the projection files are created and available to TensorBoard (in your case, TensorBoard expects the files in the tmp directory), head over to TensorBoard in your browser and check out the embedding visualization (see figure 13.5): >>> projection_name = "NLP_in_Action" >>> projection_data = [>>> ('car', [0.34, ..., -0.72]), >>> ... >>> ('toy', [0.46, ..., 0.39]), >>>] >>> create_projection(projection_data, projection_name) Figure 13.5 Visualize word2vec embeddings with Tensorboard.

Summary

? Mastering NLP parallelization can expand your brainpower by giving you a soci- ety of minds?machine clusters to help you think.24 24 Conscious Ants and Human Hives by Peter Watts (https://youtube/v4uwaw_5Q3I?t=45s).

appendix A Your NLP toolsappendix A 30.0

Once you have a package manager installed (or you?re on a developer-friendly OS like Ubuntu that already has one), you can install Anaconda3. This has the added benefit of giving you a package and environment manager that can install a lot of problem- atic packages (such as matplotlib) on a wide range of problematic OSes (like Windows).

A.3 IDE

Now that you have Python 3 and NLPIA on your machine, you only need a good text editor to round out your integrated development environment (IDE).

A.4 Ubuntu package manager

And you may not even need it if you use Anaconda?s package manager conda, as suggested in the NLPIA installation instructions (http://github.com/totalgood/nlpia). We?ve suggested some packages to install 1 That?s you, Steven ?Digital Nomad? Skoczen and Aleck ?The Dude? Landgraf.

A.5.1 A Mac package manager

Homebrew (https://brew.sh) is probably the most popular command-line package manager for Macs among developers. Listing A.4 Install brew $ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/ install/master/install)" 2 See the Homebrew package manager Wikipedia article (https://en.wikipedia.org/wiki/Homebrew_(package _management_software)).

A.5.3 Tuneups

Listing A.6 bash_profile #!/usr/bin/env bash echo "Running customized ~/.bash_profile script: '$0' ...." export HISTFILESIZE=10000000 export HISTSIZE=10000000 # append the history file after each session shopt -s histappend # allow failed commands to be re-edited with Ctrl-R shopt -s histreedit # command substitions are first presented to user before execution shopt -s histverify # store multiline commands in a single history entry shopt -s cmdhist # check the window size after each command and, if necessary, update the valu es of LINES and COLUMNS shopt -s checkwinsize # grep results are colorized

A.6 Windows

But if you install GitGUI on a Windows machine, that gets you a bash prompt and a workable terminal that you can use to run your Python REPL console: Download and install the git installer (https://git-scm.com/download/win). 2 The git installer comes with a version of the bash shell that should work well within Windows, but the git-gui that it installs isn?t very user friendly, especially for begin- ners. Unless you?re using git from the command line (a bash shell within Windows), you should use GitHub Desktop for all your git push/pull/merge needs on Win- dows.

A.7 NLPIA automagic

Fortunately for you, nlpia has some automatic environment provisioning procedures that will download the NLTK, Spacy, Word2vec models, and the data you need for this book. These downloaders will be triggered whenever you call an nlpia wrapper func- tion, like segment_sentences(), that requires any of these datasets or models. So, if you want to customize your environment, the remaining appendices show you how to install and configure the individual pieces you need for a full-featured NLP development environment. 3 Big thanks to Benjamin Berg and Darren Meiss at Manning for figuring this out, and for all the hard work they put into making this book presentable.

appendix B Playful Python and regular expressions

To get the most out of this book, you?ll want to get comfortable with Python. When things don?t work, you?ll need to be able to play around and explore to find a way to make Python do what you want. And even when your code works, playing around may help you uncover cool new ways of doing things or hidden monsters lurking in your code. Hidden errors and edge cases are very common in natural language processing, because there are so many different ways to say things in a language like English. To get playful, just experiment with Python code, like children do. When you press the Tab key, your editor or shell should try to finish your thought by completing the variable, class, function, method, attribute, and path name you started to type. Like man in a Linux shell, help() is your built-in friend in Python. The rest of this Python primer introduces the data structures and functions we use throughout this book so you can start playing with them: ? str and bytes ? ord and chr ? .format() 434

B.1.2 Templates in Python (.format())

This allows you to create dynamic responses with knowledge from a database or the context of a running python program (locals()). 1 There?s no single official Extended ASCII character set, so don?t ever use them for NLP unless you want to confuse your machine trying to learn a general language model.

B.3.1 |?OR

So the regular expression 'Hobson|Cole|Hannes' would match any of the given names (first names) of this book?s authors. Patterns are processed left to right, and ?short circuit? when a match 2 This is only true for strict regular expression syntaxes that don?t look-ahead and look-behind.

B.3.2 ()?Groups

Listing B.2 regex grouping parentheses >>> import re >>> match = re.match(r'(kitt|dogg)y', "doggy") >>> match.group() 'doggy' >>> match.group(0) 'dogg' >>> match.groups() ('dogg',) >>> match = re.match(r'((kitt|dogg)(y))', "doggy") >>> match.groups() ('doggy', 'dogg', 'y') >>> match.group(2) 'y' If you want to capture each part in its own group

B.4 Style

One thing that can help make your code more readable by professionals is to always use the single-quote (') when defining a string intended for a machine, like regular expressions, tags, and labels.

B.5 Mastery

You can do one or two of these a week while reading this book: CodingBat (http://codingbat.com)?Fun challenges in an interactive, web- 1 based Python interpreter Donne Martin?s Coding Challenges (http://github.com/donnemartin/ 2 interactive-coding-challenges)?An open source repository of Jupyter Note- books and Anki flashcards to help you learn algorithms and data 2 DataCamp (http://datacamp.com/community/tutorials)?Pandas and Python 3 tutorials at DataCamp

appendix C Vectors and matrices (linear algebra fundamentals)appendix C Vectors and matrices (linear 30.0

Listing C.1 Create a vector >>> import numpy as np >>> np.array(range(4)) array([0, 1, 2, 3]) 440 441 Vectors >>> np.arange(4) array([0, 1, 2, 3]) >>> x = np.arange(0.5, 4, 1) >>> x array([0.5, 1.5, 2.5, 3.5]) >>> x[1] = 2 >>> x array([0.5, 2, 2.5, 3.5]) >>> x.shape (4,) >>> x.T.shape (4,) An array has some properties that list doesn?t?such as .shape and .T. If you?ve ever heard of a matrix, you?ve probably heard that it can be thought of as an array of row vectors, like this: >>> np.array([range(4), range(4)]) >>> array([[0, 1, 2, 3], [0, 1, 2, 3]]) >>> X = np.array([range(4), range(4)]) >>> X.shape (2, 4) >>> X.T.shape (4, 2) The T property returns the transpose of the matrix. So the following matrix called A >>> A = np.array([[1, 2, 3], [4, 5, 6]]) >>> A array([[1, 2, 3], [4, 5, 6]]) has a transpose of >>> A.T array([[1, 4], [2, 5], [3, 6]]) So if A started out as a collection of row vectors, A.T turns those row vectors into col- umn vectors.

C.1.1 Distances

Listing C.2 Vector difference >>> A array([[1, 2, 3], [4, 5, 6]]) >>> A[0] array([1, 2, 3]) >>> A[1] array([4, 5, 6]) >>> np.diff(A, axis=0) array([[3, 3, 3]] >>> A[1] - A[0] array([3, 3, 3]) That [3, 3, 3] vector gives you exactly the distance along each dimension in your two vectors. Euclidean distance is also called the RSS distance, which stands for the root sum square distance or difference, which means: euclidean_distance = np.sqrt(((vector1 - vector2) ** 2).sum()) Let?s look at Euclidean distance between some vectors from an NLP example in Pat- rick Winston?s AI lecture series.1 Let?s say we have 2D term frequency (bag-of-word) vectors that count the occur- rences of the words ?hack? and ?computer? in articles from two publications, Wired Magazine and Town and Country. Listing C.3 Cosine distance >>> import numpy as np >>> vector_query = np.array([1, 1]) >>> vector_tc = np.array([1, 0]) >>> vector_wired = np.array([5, 6]) >>> normalized_query = vector_query / np.linalg.norm(vector_query) >>> normalized_tc = vector_tc / np.linalg.norm(vector_tc) >>> normalized_wired = vector_wired / np.linalg.norm(vector_wired) >>> normalized_query array([0.70710678, 0.70710678]) >>> normalized_tc array([1., 0.]) >>> normalized_wired array([0.6401844 , 0.76822128]) The cosine similarity between our query TF vector and these other two TF vectors (cosine of the angle between them) is >>> np.dot(normalized_query, normalized_tc) # cosine similarity 0.70710678118654746 >>> np.dot(normalized_query, normalized_wired) # cosine similarity 0.99589320646770374 The cosine distance between our query and these two TF vectors is one minus the cosine similarity. >>> 1 - np.dot(normalized_query, normalized_tc) # cosine distance 0.29289321881345254 >>> 1 - np.dot(normalized_query, normalized_wired) # cosine distance 0.0041067935322962601 This is why cosine similarity is used for TF vectors in NLP: ? It?s easy to compute (just multiplication and addition). Using our made-up magazine vectors from earlier, the Manhattan distance would be: >>> vector_tc = np.array([1, 0]) >>> vector_wired = np.array([5, 6]) >>> np.abs(vector_tc - vector_wired).sum() 10 If your vectors were normalized before calculating Manhattan distance, you?d get a much different distance: >>> normalized_tc = vector_tc / np.linalg.norm(vector_tc) >>> normalized_wired = vector_wired / np.linalg.norm(vector_wired) >>> np.abs(normalized_tc - normalized_wired).sum() 1.128... You might hope this distance metric would stay bounded within some range like 0 to 2, but it won?t.

D.2 How fit is fit?

When working with example data in any model, the given algorithm may do very well at finding patterns in that particular dataset. But given that we already likely know the label of any particular example in the training set (or it wouldn?t be in the training set), that isn?t particu- larly helpful. The real goal is to use those training examples to build a model that will generalize, and be able to correctly label an example that, while similar to members of the training set, is outside of the training set.

D.3 Knowing is half the battle

The solution is to split your labeled data into two and sometimes three datasets: a training set, a validation set, and in some cases a test set. The validation set is a smaller portion of the labeled data we hold out and keep hidden from the model for one round of training. Good performance on the validation set is a first step to verifying that the trained model will perform well in the wild, as novel data comes in. You will often see an 80%/20% or 70%/30% split for training versus validation from a given labeled dataset. The test set is like the validation set?a subset of the labeled training data to run the model against and measure performance. While training the model on the training set, there will be several iterations with var- ious hyperparameters; the final model you choose will be the one that performs the best on the validation set.

D.4 Cross-fit training

You then train your model with k-1 of the folds as a training set and validate it against the 6 Subsamples (k = 6) 5 Subsamples Validation Training run #1 Training Training Training Training Training Training Training run #2 Training Training Training Training Training Training Training Training run #3 Training Training Training Training Training Training Training ... Training run #6 Training Validation Training Training Training Training Training Figure D.2 K-fold cross-validation

D.5.1 Regularization

In any machine learning model, overfitting will eventually come up. The first is regularization, which is a penalization to the learned parameters at each training step. It?s usually, but not always, a factor of the parameters themselves.

D.5.2 Dropout

Listing D.1 A dropout layer in Keras reduces overfitting >>> from keras.models import Sequential >>> from keras.layers import Dropout, LSTM, Flatten, Dense >>> num_neurons = 20 >>> maxlen = 100 >>> embedding_dims = 300 Arbitrary hyperparmeters used as an example

D.6 Imbalanced training sets

If you were to train a model on this dataset, it would be likely that the model would simply learn to predict any given image is a cat, regardless of the input. But totally outside the scope of any particular model, the most likely cause of this fail- ure is the imbalanced training set.

D.6.3 Augmenting your data

The concept of augmentation is to generate novel data, either from perturba- tions of the existing data or generating it from scratch. The famous MNIST dataset is a set of handwritten digits, 0-9 (see figure D.4). AffNIST takes each of the digits and skews, rotates, and scales them in various ways, while maintaining the original labels.

D.7 Performance metrics

The first thing we do when starting a machine learning pipeline is set up a performance metric, such as ?.score()? on any sklearn machine learning model.

D.7.1 Measuring classifier performance

>>> y_true = np.array([0, 0, 0, 1, 1, 1, 1, 1, 1, 1]) >>> y_pred = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0, 0]) >>> true_positives = ((y_pred == y_true) & (y_pred == 1)).sum() >>> true_positives 4 true_positives are the positive class labels (1) that your model got right (correctly labeled 1.) Listing D.4 Count what the model got wrong >>> false_positives = ((y_pred != y_true) & (y_pred == 1)).sum() >>> false_positives 3 >>> false_negatives = ((y_pred != y_true) & (y_pred == 0)).sum() >>> false_negatives 1 false_positives are the negative class examples (0) that were falsely labeled positive by your model (labeled 1 when they should be 0.) Listing D.5 Confusion matrix >>> confusion = [[true_positives, false_positives], ... [false_negatives, true_negatives]] >>> confusion [[4, 3], [1, 2]] >>> import pandas as pd >>> confusion = pd.DataFrame(confusion, columns=[1, 0], index=[1, 0])   456 APPENDIX D Machine learning tools and techniques >>> confusion.index.name = r'pred \ truth' >>> confusion 1 0 pred \ truth 1 4 1 0 3 2 In a confusion matrix, you want to have large numbers along the diagonal (upper left and lower right) and low numbers in the off diagonal (upper right and lower left).

D.8 Pro tips

Once you have the basics down, some simple tricks will help you build good models faster: ? Work with a small random sample of your dataset to get the kinks out of your pipeline. 458 APPENDIX D Machine learning tools and techniques ? Plot high-dimensional data as a raw image to discover shifting across features.1 ? Try PCA on high-dimensional data (LSA on NLP data) when you want to maxi- mize the differences between pairs of vectors (separation). Hyperparameters are all the values that determine HYPERPARAMETER TUNING the performance of your pipeline, including the model type and how it?s configured. Hyperparameters also include the values that govern any preprocessing steps, like the tokenizer type, any list of words that are ignored, the minimum and maximum document frequency for the TF-IDF vocabulary, whether or not to use a lemmatizer, the TF-IDF normalization approach, and so on. When your search gets close to the final model that you think is going to meet your needs, you can increase the dataset size to use as much of the data as you need. The most efficient algorithms for hyperparameter tuning are (from best TIP to worst) Bayesian search 1 Genetic algorithms 2 Random search 3 Multi-resolution grid searches 4 Grid search 5 But any algorithm that lets your computer do this searching at night while you sleep is better than manually guessing new parameters one by one. Discovering this can help you on Kag- gle competitions that hide the source of the data, like the Santander Value Prediction competition (www .kaggle.com/c/santander-value-prediction-challenge/discussion/61394).

appendix E Setting up your AWS GPUappendix E 30.0

The EC2 Dashboard provides summary information about existing EC2 instances (see figure E.2) 459 460 APPENDIX E Setting up your AWS GPU Figure E.1 AWS Management Console Figure E.2 Creating a new AWS instance   461 Steps to create your AWS GPU instance In the EC2 Dashboard, click the blue Launch Instance button to start the 3 instance setup wizard, a sequence of screens where you can configure the vir- tual machine you want to launch. If you?ve returned to the AMI lists on Ama- zon Marketplace, you can click the blue Select button next to the AMI you would like to install on your EC2 instance, which will take you to ?Step 2: Choose an Instance Type? (see figure E.5). The link sends you 18 to the overview of all your EC2 instances, where you?ll see your instance with its state indicated as ?running? or ?initializing.? Figure E.9 Creating a new instance key (or downloading an existing one)   467 Steps to create your AWS GPU instance Figure E.10 AWS launch confirmation Figure E.11 EC2 Dashboard showing the newly created instance   468 APPENDIX E Setting up your AWS GPU You?ll want to record the public IP address for your instance (see figure E.11) 19 alongside the .pem file for the key pair you generated earlier.

E.1.1 Cost control

471 Steps to create your AWS GPU instance Figure E.15 AWS Billing Dashboard   Figure E.16 AWS Budget Console   472 APPENDIX E Setting up your AWS GPU ? Check your EC2 instance summary page for running instances.

F.1.1 Vector space indexes and hashes

Real-value (float) indexes for things like latitude and longitude can?t rely on exact matches, like the index of words at the back of a textbook. For 2D real-valued data, most indexes use 1 Some advanced databases such as PostgreSQL can index higher-dimensional vectors, but efficiency drops quickly with dimensionality.

F.1.2 High-dimensional thinking

Listing F.1 Explore high-dimensional space >>> import pandas as pd >>> import numpy as np >>> from tqdm import tqdm >>> num_vecs = 100000 >>> num_radii = 20 >>> num_dim_list = [2, 4, 8, 18, 32, 64, 128] >>> radii = np.array(list(range(1, num_radii + 1))) >>> radii = radii / len(radii) >>> counts = np.zeros((len(radii), len(num_dims_list))) >>> rand = np.random.rand 477 High-dimensional vectors are different Normalize a table of random row vectors to all have unit length. You can see much of the weirdness in the following table, which shows the density of points in each bounding box as you expand its size bit by bit: >>> df = pd.DataFrame(counts, index=radii, columns=num_dim_list) / num_vecs >>> df = df.round(2) >>> df[df == 0] = '' >>> df 2 4 8 18 32 64 128 0.05 0.10 0.15 0.37 0.20 0.1 1 0.25 1 1 0.30 0.55 1 1 0.35 0.12 0.98 1 1 0.40 0.62 1 1 1 0.45 0.03 0.92 1 1 1 0.50 0.2 0.99 1 1 1 0.55 0.01 0.5 1 1 1 1 0.60 0.08 0.75 1 1 1 1 0.65 0.24 0.89 1 1 1 1 0.70 0.45 0.96 1 1 1 1 0.75 0.12 0.64 0.99 1 1 1 1 0.80 0.25 0.78 1 1 1 1 1 0.85 0.38 0.88 1 1 1 1 1 0.90 0.51 0.94 1 1 1 1 1 0.95 0.67 0.98 1 1 1 1 1 1.00 1 1 1 1 1 1 1 There?s an indexing algorithm called a KD-Tree (https://en.wikipedia.org/wiki/K-d _tree) that attempts to divide up high-dimensional spaces as efficiently as possible to minimize empty bounding boxes.

F.2.1 Locality sensitive hashing

In figure F.1, we constructed 400,000 completely random vectors, each with 200 dimensions (typical for topic vectors for a large corpus). Figure F.1 Semantic search with LSHash You can?t get many search results correct once the number of dimensions gets signifi- cantly above 10 or so. If you?d like to play with this yourself, or try your hand at build- ing a better LSH algorithm, the code for running experiments like this is available in the nlpia package.

F.3 ?Like? prediction

You should probably only use LSA, LDA, and LDiA language models for classification problems where vari- ance maximization (class separability) is helpful: ? Semantic search ? Sentiment analysis ? Spam detection For more subtle discrimination between texts that rely on generalizing from similari- ties in semantic content, you?ll want the most sophisticated NLP tools in your toolbox.

Applications and project ideas

Here are some applications to inspire your own NLP projects: ? Guessing passwords from social network profiles (http://www.sciencemag .org/news/2017/09/artificial-intelligence-just-made-guessing-your-password -whole-lot-easier). ? GitHub - craigboman/gutenberg: Librarian working with project gutenberg data, for NLP and machine learning purposes (https://github.com/craigboman/ gutenberg). ? Time Series Matching: a Multi-filter Approach by Zhihua Wang (https://www .cs.nyu.edu/web/Research/Theses/wang_zhihua.pdf)?Songs, audio clips, and other time series can be discretized and searched with dynamic pro- gramming algorithms analogous to Levenshtein distance.

Courses and tutorials

Here are some good tutorials, demonstrations, and even courseware from renowned university programs, many of which include Python examples: ? Speech and Language Processing (https://web.stanford.edu/~jurafsky/slp3/ed3 book.pdf) by David Jurafsky and James H. Martin?The next book you should read if you?re serious about NLP. They have whole chapters on topics that we largely ignore, like finite state transducers (FSTs), hidden Markhov mod- els (HMMs), part-of-speech (POS) tagging, syntactic parsing, discourse coher- ence, machine translation, summarization, and dialog systems. ? MIT Artificial General Intelligence course 6.S099 (https://agi.mit.edu) led by Lex Fridman Feb 2018?MIT?s free, interactive (public competition!)

Vector space models and semantic search

? Semantic Vector Encoding and Similarity Search Using Fulltext Search Engines (https:// arxiv.org/pdf/1706.00957.pdf)?Jan Rygl et al.

LSTMs and RNNs

The state of the Wikipedia page (and Talk page dis- cussion) on LSTMs is a pretty good indication of the lack of consensus about what LSTM means: ? Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation (https://arxiv.org/pdf/1406.1078.pdf) by Cho et al.?Explains how the contents of the memory cells in an LSTM layer can be used as an embed- ding that can encode variable length sequences and then decode them to a new variable length sequence with a potentially different length, translating or transcoding one sequence into another.

Datasets

Language is the superpower of the human race, and your pipeline should take advantage of it: ? Google?s Dataset Search (http://toolbox.google.com/datasetsearch)?A search engine similar to Google Scholar (http://scholar.google.com), but for data. ? Stanford Datasets (https://nlp.stanford.edu/data/)?Pretrained word2vec and GloVE models, multilingual language models and datasets, multilingual dic- tionaries, lexica, and corpora. ? Pretrained word vector models (https://github.com/3Top/word2vec-api#where-to- get-a-pretrained-model)?The README for a word vector web API provides links to several word vector models, including the 300D Wikipedia GloVE model.

Open source full-text indexers

However, these ?search engines? don?t crawl the web, so you need to provide them with the corpus you want them to index and search: ? Elasticsearch (https://github.com/elastic/elasticsearch)?Open Source, Dis- tributed, RESTful Search Engine.

Less manipulative search engines

And the top search results were often the most objective and useful sites, such as Wikipedia, Stack Exchange, or reputable news articles and blogs: ? Alternatives to Google (https://www.lifehack.org/374487/try-these-15-search-engines-instead-google-for-better-search-results).2 1 Cornell University Networks Course case study, ?Google AdWords Auction - A Second Price Sealed-Bid Auction,? (https://blogs.cornell.edu/info2040/2012/10/27/google-adwords-auction-a-second-price-sealed-bid-auction).

Distributed search engines

It?s just a matter of time before someone decides to contribute code for semantic search into an open source project like Yacy or builds a new distributed search engine capable of LSA: ? Nutch (https://nutch.apache.org/)?Nutch spawned Hadoop and itself became less of a distributed search engine and more of a distributed HPC sys- tem over time. ? Yacy (https://www.yacy.net/en/index.html)?One of the few open source (https://github.com/yacy/yacy_search_server) decentralized, or federated, search engines and web crawlers still actively in use. 3 See the web pages titled ?Distributed search engine,? (https://en.wikipedia.org/wiki/Distributed_search _engine) and ?Distributed Search Engines,? (https://wiki.p2pfoundation.net/Distributed _Search_Engines).

Acronyms

492   glossary GPU?Graphical processing unit The graphics card in a gaming rig, a cryptocurrency mining server, or a machine learning server GRU?Gated recurrent unit A variation of long short-term memory networks with shared parameters to cut computation time HNSW?A graph data structure that enables efficient search (and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs (https://arxiv.org/vc/arxiv/ papers/1603/1603.09320v1.pdf) by Yu A. Malkov and D. A. Yashunin) HPC?High performance computing The study of systems that maximize throughput, usually by parallelizing computation with separate map and reduce computation stages IDE?Integrated development environment A desktop application for software development, such as PyCharm, Eclipse, Atom, or Sub- lime Text 3 IR?Information retrieval The study of document and web search engine algorithms. LSTM?Long short-term memory An enhanced form of a recurrent neural network that maintains a memory of state that itself is trained via backpropagation (see chapter 9) MIH?Multi-index hashing A hashing and indexing approach for high-dimensional dense vectors ML?Machine learning Programming a machine with data rather than hand-coded algorithms MSE?Mean squared error The sum of the square of the difference between the desired output of a machine learning model and the actual output of the model   493 Acronyms NELL?Never Ending Language Learning A Carnegie Mellon knowledge extraction project that has been running continuously for years, scraping web pages and extracting general knowledge about the world (mostly ?IS-A? categorical relationships between terms) NLG?Natural language generation Composing text automatically, algorithmically; one of the most challenging tasks of natural language processing (NLP) NLP?Natural language processing You probably know what this is by now.

Terms

Artificial neural network?A computational graph for machine learning or simulation of a bio- logical neural network (brain) Cell?The memory or state part of an LSTM unit that records a single scalar value and outputs it continuously 4 Dark patterns?Software patterns (usually for a user interface) that are intended to increase revenue but often fail due to ?blowback? because they manipulate your customers into using your product in ways that they don?t intend 4 See the web page titled ?Long short-term memory? (https://en.wikipedia.org/wiki/Long_short-term _memory). The morphology of a token can be found using algorithms in packages like SpaCy that process the token with its context (words around it).5 Net, network, or neural net?Artificial neural network Neuron?A unit in a neural net whose function (such as y = tanh(w.dot(x))) takes multiple inputs and outputs a single scalar value. Subject?The main noun of a sentence?every complete sentence must have a subject (and a predicate) even if the subject is implied, like in the sentence ?Run!? where the implied subject is ?you.? Unit?Neuron or small collection of neurons that perform some more complicated nonlinear function to compute the output.