Data Science from Scratch

SECOND EDITION

First Principles with Python

Joel Grus

Beijing • Boston • Farnham • Sebastopol • Tokyo

DISCLAIMER:

ReadMe website is intended for academic and demonstration purposes only. We’re only showing a preview of the book to respect the author’s copyright. Thank you for your understanding!

- Group 4: The Classified

Data Science from Scratch

by Joel Grus

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Michele Cronin

Production Editor: Deborah Baker

Copy Editor: Rachel Monaghan

Proofreader: Rachel Head

Indexer: Judy McConville

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

April 2015: First Edition

May 2019: Second Edition

Revision History for the Second Edition

> *Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex.*

and represent ideals that we will strive for in our code.

Getting Python

> **NOTE**
>
> As instructions about how to install things can change, while printed books cannot, up-to-date instructions on how to install Python can be found in the book's GitHub repo.
>
> If the ones printed here don't work for you, check those.

You can download Python from Python.org. But if you don't already have Python, I recommend instead installing the Anaconda distribution, which already includes most of the libraries that you need to do data science.

When I wrote the first version of *Data Science from Scratch*, Python 2.7 was still the preferred version of most data scientists. Accordingly, the first edition of the book was based on Python 2.7.

In the last several years, however, pretty much everyone who counts has migrated to Python 3. Recent versions of Python have many features that make it easier to write clean code, and we'll be taking ample advantage of features that are only available in Python 3.6 or later. This means that you should get Python 3.6 or later. (In addition, many useful libraries are ending support for Python 2.7, which is another reason to switch.)

Virtual Environments

Starting in the next chapter, we'll be using the matplotlib library to generate plots and charts. This library is not a core part of Python; you have to install it yourself. Every data science project you do will require some combination of external libraries, sometimes with specific versions that

written about them. We will get into their details the few times we encounter them; here are a few examples of how to use them in Python:

```
import re

re_examples = [                         # All of these are True, because
    not re.match("a", "cat"),               #  'cat' doesn't start with 'a'
    re.search("a", "cat"),                  #  'cat' has an 'a' in it
    not re.search("c", "dog"),              #  'dog' doesn't have a 'c' in it.
    3 == len(re.split("[ab]", "carbs")),    #  Split on a or b to
['c','r','s'].
    "R-D-" == re.sub("[0-9]", "-", "R2D2") #  Replace digits with dashes.
    ]

assert all(re_examples), "all the regex examples should be True"
```

One important thing to note is that `re.match` checks whether the *beginning* of a string matches a regular expression, while `re.search` checks whether *any part* of a string matches a regular expression. At some point you will mix these two up and it will cause you grief.

The official documentation goes into much more detail.

Functional Programming

> **NOTE**
>
> The first edition of this book introduced the Python functions `partial`, `map`, `reduce`, and `filter` at this point. On my journey toward enlightenment I have realized that these functions are best avoided, and their uses in the book have been replaced with list comprehensions, `for` loops, and other, more Pythonic constructs.

zip and Argument Unpacking

Often we will need to *zip* two or more iterables together. The `zip` function transforms multiple iterables into a single iterable of tuples of

> **NOTE**
>
> Using lists as vectors is great for exposition but terrible for performance.
>
> In production code, you would want to use the NumPy library, which includes a high-performance array class with all sorts of arithmetic operations included.

Matrices

A *matrix* is a two-dimensional collection of numbers. We will represent matrices as lists of lists, with each inner list having the same size and representing a *row* of the matrix. If `A` is a matrix, then `A[i][j]` is the element in the *i*th row and the *j*th column. Per mathematical convention, we will frequently use capital letters to represent matrices. For example:

```
# Another type alias
Matrix = List[List[float]]

A = [[1, 2, 3],  # A has 2 rows and 3 columns
     [4, 5, 6]]

B = [[1, 2],     # B has 3 rows and 2 columns
     [3, 4],
     [5, 6]]
```

> **NOTE**
>
> In mathematics, you would usually name the first row of the matrix "row 1" and the first column "column 1." Because we're representing matrices with Python `lists`, which are zero-indexed, we'll call the first row of a matrix "row 0" and the first column "column 0."

Given this list-of-lists representation, the matrix `A` has `len(A)` rows and `len(A[0])` columns, which we consider its `shape`:

```
from typing import Tuple
```

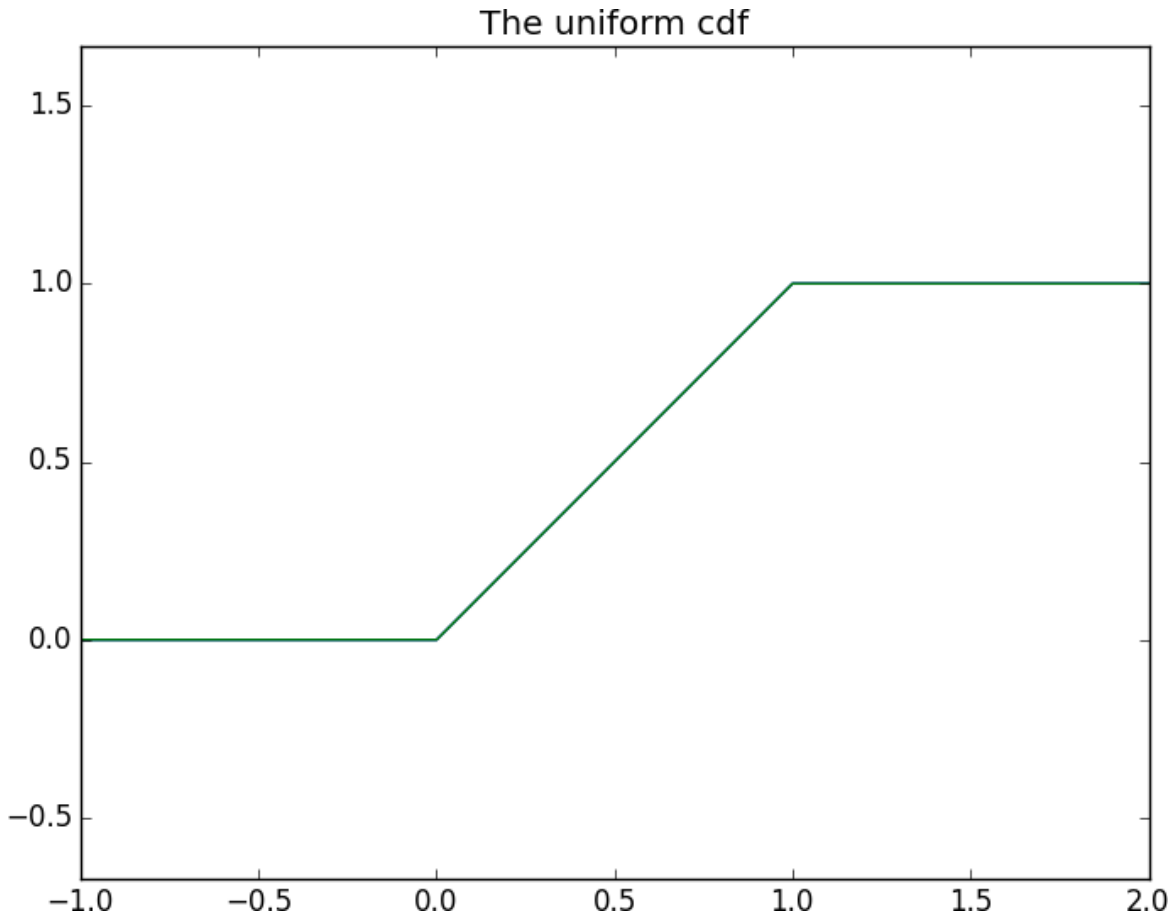

Figure 6-1. The uniform CDF

The Normal Distribution

The normal distribution is the classic bell curve–shaped distribution and is completely determined by two parameters: its mean μ (mu) and its standard deviation σ (sigma). The mean indicates where the bell is centered, and the standard deviation how "wide" it is.

It has the PDF:

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

which we can implement as:

difference quotient for a very small `e`. Figure 8-3 shows the results of one such estimation:

```
xs = range(-10, 11)
actuals = [derivative(x) for x in xs]
estimates = [difference_quotient(square, x, h=0.001) for x in xs]

# plot to show they're basically the same
import matplotlib.pyplot as plt
plt.title("Actual Derivatives vs. Estimates")
plt.plot(xs, actuals, 'rx', label='Actual')       # red  x
plt.plot(xs, estimates, 'b+', label='Estimate')   # blue +
plt.legend(loc=9)
plt.show()
```

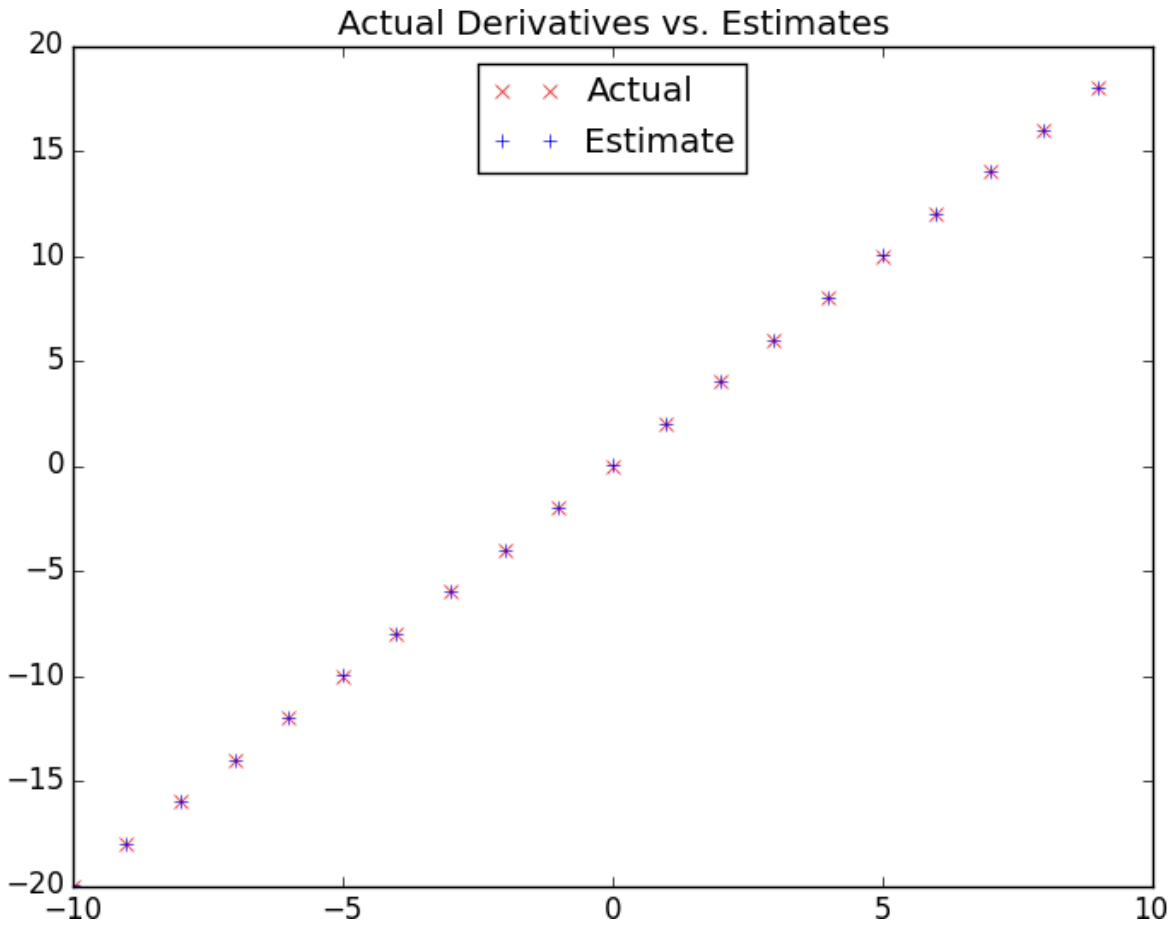

Figure 8-3. Goodness of difference quotient approximation

When `f` is a function of many variables, it has multiple *partial derivatives*, each indicating how `f` changes when we make small changes in just one of

```
                      temp_creds['oauth_token_secret'])
final_step = auth_client.get_authorized_tokens(PIN_CODE)
ACCESS_TOKEN = final_step['oauth_token']
ACCESS_TOKEN_SECRET = final_step['oauth_token_secret']

# And get a new Twython instance using them.
twitter = Twython(CONSUMER_KEY,
                  CONSUMER_SECRET,
                  ACCESS_TOKEN,
                  ACCESS_TOKEN_SECRET)
```

> **TIP**
>
> At this point you may want to consider saving the `ACCESS_TOKEN` and `ACCESS_TOKEN_SECRET` somewhere safe, so that next time you don't have to go through this rigmarole.

Once we have an authenticated `Twython` instance, we can start performing searches:

```
# Search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"]
    text = status["text"]
    print(f"{user}: {text}\n")
```

If you run this, you should get some tweets back like:

```
haithemnyc: Data scientists with the technical savvy &amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization. So easy and Awesome.
```

This isn't that interesting, largely because the Twitter Search API just shows you whatever handful of recent results it feels like. When you're doing data science, more often you want a lot of tweets. This is where the Streaming API is useful. It allows you to connect to (a sample of) the great

```
from scratch.linear_algebra import scalar_multiply

def project(v: Vector, w: Vector) -> Vector:
    """return the projection of v onto the direction w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

If we want to find further components, we first remove the projections from the data:

```
from scratch.linear_algebra import subtract

def remove_projection_from_vector(v: Vector, w: Vector) -> Vector:
    """projects v onto w and subtracts the result from v"""
    return subtract(v, project(v, w))

def remove_projection(data: List[Vector], w: Vector) -> List[Vector]:
    return [remove_projection_from_vector(v, w) for v in data]
```

Because this example dataset is only two-dimensional, after we remove the first component, what's left will be effectively one-dimensional (Figure 10-9).

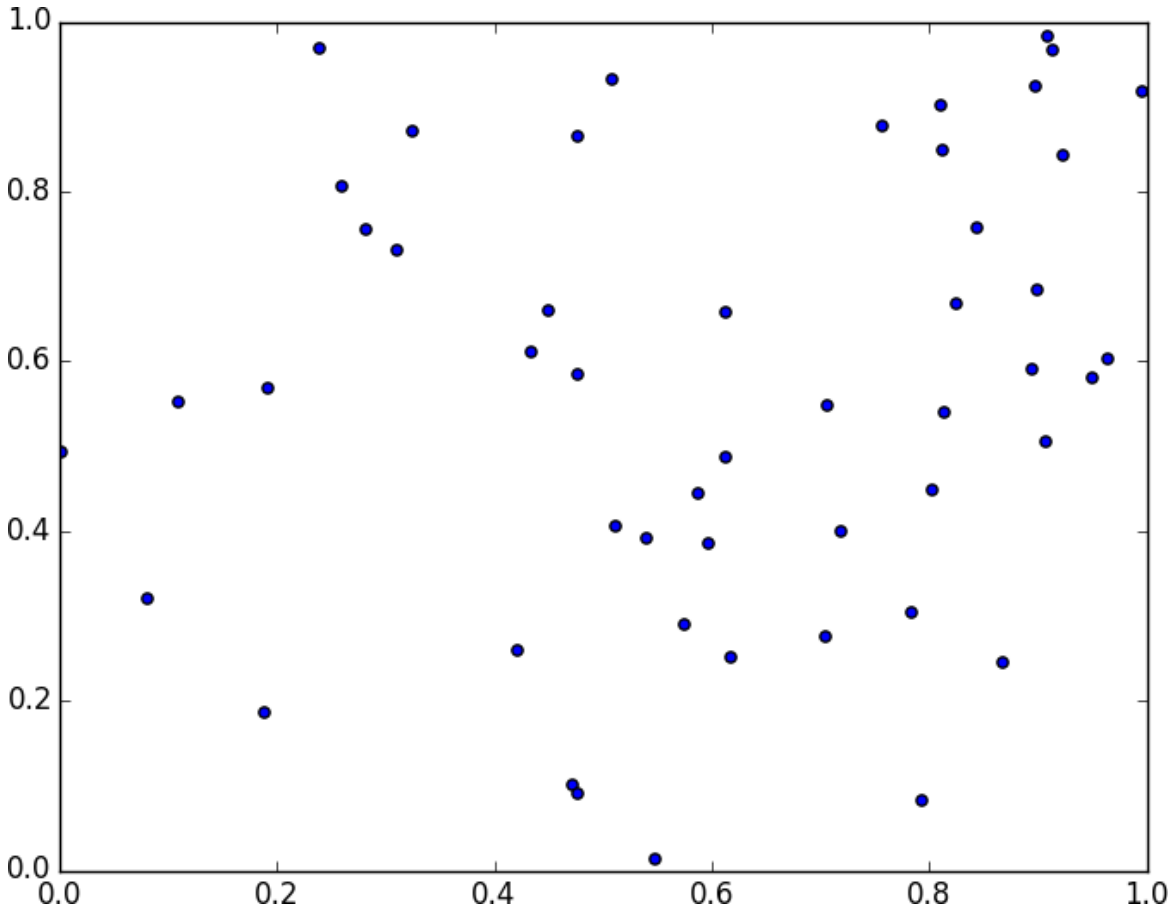

Figure 12-5. Fifty random points in two dimensions

And in three dimensions, less still (Figure 12-6).

matplotlib doesn't graph four dimensions well, so that's as far as we'll go, but you can see already that there are starting to be large empty spaces with no points near them. In more dimensions—unless you get exponentially more data—those large empty spaces represent regions far from all the points you want to use in your predictions.

So if you're trying to use nearest neighbors in higher dimensions, it's probably a good idea to do some kind of dimensionality reduction first.

We can take the same approach to estimating the standard errors of our regression coefficients. We repeatedly take a `bootstrap_sample` of our data and estimate `beta` based on that sample. If the coefficient corresponding to one of the independent variables (say, `num_friends`) doesn't vary much across samples, then we can be confident that our estimate is relatively tight. If the coefficient varies greatly across samples, then we can't be at all confident in our estimate.

The only subtlety is that, before sampling, we'll need to `zip` our `x` data and `y` data to make sure that corresponding values of the independent and dependent variables are sampled together. This means that `bootstrap_sample` will return a list of pairs `(x_i, y_i)`, which we'll need to reassemble into an `x_sample` and a `y_sample`:

```
from typing import Tuple

import datetime

def estimate_sample_beta(pairs: List[Tuple[Vector, float]]):
    x_sample = [x for x, _ in pairs]
    y_sample = [y for _, y in pairs]
    beta = least_squares_fit(x_sample, y_sample, learning_rate, 5000, 25)
    print("bootstrap sample", beta)
    return beta

random.seed(0) # so that you get the same results as me

# This will take a couple of minutes!
bootstrap_betas = bootstrap_statistic(list(zip(inputs, daily_minutes_good)),
                                      estimate_sample_beta,
                                      100)
```

After which we can estimate the standard deviation of each coefficient:

```
bootstrap_standard_errors = [
    standard_deviation([beta[i] for beta in bootstrap_betas])
    for i in range(4)]

print(bootstrap_standard_errors)

# [1.272,    # constant term, actual error = 1.19
```

her level, her preferred language, whether she is active on Twitter, whether she has a PhD, and whether she interviewed well:

```
from typing import NamedTuple, Optional

class Candidate(NamedTuple):
    level: str
    lang: str
    tweets: bool
    phd: bool
    did_well: Optional[bool] = None  # allow unlabeled data

                  #  level     lang     tweets  phd  did_well
inputs = [Candidate('Senior', 'Java',   False, False, False),
          Candidate('Senior', 'Java',   False, True,  False),
          Candidate('Mid',    'Python', False, False, True),
          Candidate('Junior', 'Python', False, False, True),
          Candidate('Junior', 'R',      True,  False, True),
          Candidate('Junior', 'R',      True,  True,  False),
          Candidate('Mid',    'R',      True,  True,  True),
          Candidate('Senior', 'Python', False, False, False),
          Candidate('Senior', 'R',      True,  False, True),
          Candidate('Junior', 'Python', True,  False, True),
          Candidate('Senior', 'Python', True,  True,  True),
          Candidate('Mid',    'Python', False, True,  True),
          Candidate('Mid',    'Java',   True,  False, True),
          Candidate('Junior', 'Python', False, True,  False)
         ]
```

Our tree will consist of *decision nodes* (which ask a question and direct us differently depending on the answer) and *leaf nodes* (which give us a prediction). We will build it using the relatively simple *ID3* algorithm, which operates in the following manner. Let's say we're given some labeled data, and a list of attributes to consider branching on:

- If the data all have the same label, create a leaf node that predicts that label and then stop.
- If the list of attributes is empty (i.e., there are no more possible questions to ask), create a leaf node that predicts the most common label and then stop.

```
    def grads(self) -> Iterable[Tensor]:
        """
        Returns the gradients, in the same order as params().
        """
        return ()
```

The `forward` and `backward` methods will have to be implemented in our concrete subclasses. Once we build a neural net, we'll want to train it using gradient descent, which means we'll want to update each parameter in the network using its gradient. Accordingly, we insist that each layer be able to tell us its parameters and gradients.

Some layers (for example, a layer that applies `sigmoid` to each of its inputs) have no parameters to update, so we provide a default implementation that handles that case.

Let's look at that layer:

```
from scratch.neural_networks import sigmoid

class Sigmoid(Layer):
    def forward(self, input: Tensor) -> Tensor:
        """
        Apply sigmoid to each element of the input tensor,
        and save the results to use in backpropagation.
        """
        self.sigmoids = tensor_apply(sigmoid, input)
        return self.sigmoids

    def backward(self, gradient: Tensor) -> Tensor:
        return tensor_combine(lambda sig, grad: sig * (1 - sig) * grad,
                              self.sigmoids,
                              gradient)
```

There are a couple of things to notice here. One is that during the forward pass we saved the computed sigmoids so that we could use them later in the backward pass. Our layers will typically need to do this sort of thing.

Second, you may be wondering where the `sig * (1 - sig) * grad` comes from. This is just the chain rule from calculus and corresponds to the

Using the `vector_mean` function from Chapter 4, it's pretty simple to create a class that does this.

To start with, we'll create a helper function that measures how many coordinates two vectors differ in. We'll use this to track our training progress:

```
from scratch.linear_algebra import Vector

def num_differences(v1: Vector, v2: Vector) -> int:
    assert len(v1) == len(v2)
    return len([x1 for x1, x2 in zip(v1, v2) if x1 != x2])

assert num_differences([1, 2, 3], [2, 1, 3]) == 2
assert num_differences([1, 2], [1, 2]) == 0
```

We also need a function that, given some vectors and their assignments to clusters, computes the means of the clusters. It may be the case that some cluster has no points assigned to it. We can't take the mean of an empty collection, so in that case we'll just randomly pick one of the points to serve as the "mean" of that cluster:

```
from typing import List
from scratch.linear_algebra import vector_mean

def cluster_means(k: int,
                  inputs: List[Vector],
                  assignments: List[int]) -> List[Vector]:
    # clusters[i] contains the inputs whose assignment is i
    clusters = [[] for i in range(k)]
    for input, assignment in zip(inputs, assignments):
        clusters[assignment].append(input)

    # if a cluster is empty, just use a random point
    return [vector_mean(cluster) if cluster else random.choice(inputs)
            for cluster in clusters]
```

And now we're ready to code up our clusterer. As usual, we'll use `tqdm` to track our progress, but here we don't know how many iterations it will take, so we then use `itertools.count`, which creates an infinite iterable, and we'll `return` out of it when we're done:

```
    for _ in range(num_samples):
        counts[gibbs_sample()][0] += 1
        counts[direct_sample()][1] += 1
    return counts
```

We'll use this technique in the next section.

Topic Modeling

When we built our "Data Scientists You May Know" recommender in Chapter 1, we simply looked for exact matches in people's stated interests.

A more sophisticated approach to understanding our users' interests might try to identify the *topics* that underlie those interests. A technique called *latent Dirichlet allocation* (LDA) is commonly used to identify common topics in a set of documents. We'll apply it to documents that consist of each user's interests.

LDA has some similarities to the Naive Bayes classifier we built in Chapter 13, in that it assumes a probabilistic model for documents. We'll gloss over the hairier mathematical details, but for our purposes the model assumes that:

- There is some fixed number K of topics.
- There is a random variable that assigns each topic an associated probability distribution over words. You should think of this distribution as the probability of seeing word w given topic k.
- There is another random variable that assigns each document a probability distribution over topics. You should think of this distribution as the mixture of topics in document d.
- Each word in a document was generated by first randomly picking a topic (from the document's distribution of topics) and then randomly picking a word (from the topic's distribution of words).

Chapter 22. Network Analysis

> *Your connections to all the things around you literally define who you are.*
>
> —Aaron O'Connell

Many interesting data problems can be fruitfully thought of in terms of *networks*, consisting of *nodes* of some type and the *edges* that join them.

For instance, your Facebook friends form the nodes of a network whose edges are friendship relations. A less obvious example is the World Wide Web itself, with each web page a node and each hyperlink from one page to another an edge.

Facebook friendship is mutual—if I am Facebook friends with you, then necessarily you are friends with me. In this case, we say that the edges are *undirected*. Hyperlinks are not—my website links to *whitehouse.gov*, but (for reasons inexplicable to me) *whitehouse.gov* refuses to link to my website. We call these types of edges *directed*. We'll look at both kinds of networks.

Betweenness Centrality

In Chapter 1, we computed the key connectors in the DataSciencester network by counting the number of friends each user had. Now we have enough machinery to take a look at other approaches. We will use the same network, but now we'll use `NamedTuple`s for the data.

Recall that the network (Figure 22-1) comprised users:

```
from typing import NamedTuple

class User(NamedTuple):
    id: int
    name: str
```

```
4.36 Star Wars (1977)
4.20 Empire Strikes Back, The (1980)
4.01 Return of the Jedi (1983)
```

So let's try to come up with a model to predict these ratings. As a first step, let's split the ratings data into train, validation, and test sets:

```
import random
random.seed(0)
random.shuffle(ratings)

split1 = int(len(ratings) * 0.7)
split2 = int(len(ratings) * 0.85)

train = ratings[:split1]              # 70% of the data
validation = ratings[split1:split2]   # 15% of the data
test = ratings[split2:]               # 15% of the data
```

It's always good to have a simple baseline model and make sure that ours does better than that. Here a simple baseline model might be "predict the average rating." We'll be using mean squared error as our metric, so let's see how the baseline does on our test set:

```
avg_rating = sum(rating.rating for rating in train) / len(train)
baseline_error = sum((rating.rating - avg_rating) ** 2
                     for rating in test) / len(test)

# This is what we hope to do better than
assert 1.26 < baseline_error < 1.27
```

Given our embeddings, the predicted ratings are given by the matrix product of the user embeddings and the movie embeddings. For a given user and movie, that value is just the dot product of the corresponding embeddings.

So let's start by creating the embeddings. We'll represent them as `dicts` where the keys are IDs and the values are vectors, which will allow us to easily retrieve the embedding for a given ID:

Then `wc_reducer` produces the counts for each word:

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

Why MapReduce?

As mentioned earlier, the primary benefit of MapReduce is that it allows us to distribute computations by moving the processing to the data. Imagine we want to word-count across billions of documents.

Our original (non-MapReduce) approach requires the machine doing the processing to have access to every document. This means that the documents all need to either live on that machine or else be transferred to it during processing. More important, it means that the machine can process only one document at a time.

> **NOTE**
>
> Possibly it can process up to a few at a time if it has multiple cores and if the code is rewritten to take advantage of them. But even so, all the documents still have to *get to* that machine.

Imagine now that our billions of documents are scattered across 100 machines. With the right infrastructure (and glossing over some of the details), we can do the following:

- Have each machine run the mapper on its documents, producing lots of key/value pairs.
- Distribute those key/value pairs to a number of "reducing" machines, making sure that the pairs corresponding to any given key all end up on the same machine.
- Have each reducing machine group the pairs by key and then run the reducer on each set of values.