MinHash Tutorial with Python Code

12 Jun 2015

In this post, I'm providing a brief tutorial, along with some example Python code, for applying the MinHash algorithm to compare a large number of documents to one another efficiently.

I first learned about this topic through Stanford's Mining of Massive Datasets ("MMDS") course available for free on Coursera here (Update 11/5/19 - It's no longer on Coursera, but still freely available here). What's especially great about that course is that the authors also provide their textbook online for free! You can find the textbook here, with a separate PDF file for each chapter. Chapter 3 covers the MinHash algorithm, and I'd refer you to that text as a more complete discussion of the topic.

On to the tutorial!

Set Similarity

There is an interesting computing problem that arises in a number of contexts called "set similarity".

Lets say you and I are both subscribers to Netflix, and we've each watched roughly 100 movies on Netflix. The list of movies I've seen is a set, and the list of movies you've seen is another set. To measure the similarity between these two sets, you can use the Jaccard Similarity, which is given by the *intersection* of the sets divided by their *union*. That is, count the number of movies we've both seen, and divide that by the total number of unique movies that we've both collectively seen.

If we've each watched exactly 100 movies, and 50 of those were seen by both of us, then the intersection is 50 and the union is 150, so our Jaccard Similarity is 1/3.

Documents as sets

What seems to be the more common application of "set similarity" is the comparison of documents. One way to represent a document would be to parse it for all of its words, and represent the document as the set of all unique words it contains. In practice, you'd hash the words to integer IDs, and then maintain the set of IDs present in the document.

By representing the documents as sets of words, you could then use the Jaccard Similarity as a measure of how much overlap there is between two documents.

It's important to note that we're not actually extracting any semantic meaning of the documents here, we're simply looking at whether they contain the same words. This technique of comparing documents probably won't work as well, for example, for comparing documents that cover similar concepts but are otherwise completely unique.

Instead, the applications of this technique are found where there's some expectation that the documents will specifically contain a lot of the same words.

One example is aggregating news articles. When the Associated Press releases an article about a particular event, many news agencies will take the AP article, perhaps modify it some, and publish it on their website. A news aggregator needs to recognize that a group of articles are really all based on the same AP article about one particular story. Comparing the web pages using this "similar sets" approach is one way to accomplish this.

Another example is detecting plagiarism. The dataset used in my example code is a large collection of articles, some of which are plagiarisms of each other (where they've been just slightly modified).

You might say that these are all applications of "near-duplicate" detection.

Shingles

A small detail here is that it is more common to parse the document by taking, for example, each possible string of three consecutive words from the document (e.g., "A small detail", "small detail here", "detail here is", etc.) and hashing these strings to integers. This retains a little more of the document structure than just hashing the individual words. This technique of hashing substrings is referred to as "shingling", and each unique string is called a "shingle".

Another shingling technique that's described in the Mining of Massive Datasets textbook is *k*-shingles, where you take each possible sequence of 'k' characters. I'm not clear on the motivation of this approach—it may have to do with the fact that it always produces strings of length 'k', whereas the three-word approach produces variable length strings.

In the example code, I'm using three-word shingles, and it works well.

Problem scale

So far, this all sounds pretty straight forward and manageable. Where it gets interesting is when you look at the compute requirements for doing this for a relatively large number of documents.

Let's say you have a large collection of documents, and you want to find all of the pairs of documents that are near-duplicates of each other. You'd do this by calculating the Jaccard similarity between each pair of documents, and then selecting those with a similarity above some threshold.

To compare each document to every other document requires a lot of comparisons! It's not quite N-squared comparisons, since that would include doing a redundant comparison of 'a' to 'b' and 'b' to 'a', as well as comparing every document to itself.

The number of comparisons required is given by the following formula, which is pronounced "N-choose-2"

$$\binom{N}{2} = \frac{N(N-1)}{2} \approx \frac{N^2}{2}$$

As noted in the equation, a good approximation is N^2 / 2 (This is approximation is equivalent to comparing each document pair only once, but also needlessly comparing each document to itself).

Lets say we have a collection of 1 million documents, and that on average, a PC can calculate the Jaccard similarity between two sets in 1ms per pair.

First, let's calculate the rough number of comparisons required:

$$\binom{10^6}{2} \approx \frac{(10^6)^2}{2} = 500\ billion\ comparisons$$

Next, the amount of time required:

$$5 \times 10^{11} comparisons \times \frac{1 \times 10^{-3} seconds}{1\ comparison} = 500\ million\ seconds \approx 16\ years$$

16 years of compute time! Good luck with that. You'd need 1,000 servers just to get the compute time down to a week. But there's a better way...

MinHash Signatures

The MinHash algorithm will provide us with a fast approximation to the Jaccard Similarity between two sets.

For each set in our data, we are going to calculate a MinHash signature. The MinHash signatures will all have a fixed length, independent of the size of the set. And the signatures will be relatively short—in the example code, they are only 10 components long.

To approximate the Jaccard Similarity between two sets, we will take their MinHash signatures, and simply count the number of components which are equal. If you divide this count by the signature length, you have a pretty good approximation to the Jaccard Similarity between those two sets.

We can compare two MinHash signatures in this way much quicker than we can calculate the intersection and union between two large sets. This is partly because the MinHash signatures tend to be much shorter than the number of shingles in the documents, and partly because the comparison operation is simpler.

In the example code, we have a collection of 10,000 articles which contain, on average, 250 shingles each. Computing the Jaccard similarities directly for all pairs takes 20 minutes on my PC, while generating and comparing the MinHash signatures takes only about 2 minutes and 45 seconds.

MinHash Algorithm

The MinHash algorithm is actually pretty easy to describe if you start with the implementation rather than the intuitive explanation.

The key ingredient to the algorithm is that we have a hash function which takes a 32-bit integer and maps it to a different integer, with no collisions. Put another way, if you took the numbers 0 – (2^32 – 1) and applied this hash function to all of them, you'd get back a list of the same numbers in random order.

To demystify it a bit, here is the definition of the hash function, which takes an input integer 'x':

$$h(x) = (ax + b) \% c$$

The coefficients `a` and `b` are randomly chosen integers less than the maximum value of `x`. `c` is a prime number slightly bigger than the maximum value of `x`.

For different choices of `a` and `b`, this hash function will produce a different random mapping of the values. So we have the ability to "generate" as many of these random hash functions as we want by just picking different values of `a` and `b`.

So here's how you compute the MinHash signature for a given document. Generate, say, 10 random hash functions. Take the first hash function, and apply it to all of the shingle values in a document. Find the minimum hash value produced (hey, "minimum hash", that's the name of the algorithm!) and use it as the first component of the MinHash signature. Now take the second hash function, and again find the minimum resulting hash value, and use this as the second component. And so on.

So if we have 10 random hash functions, we'll get a MinHash signature with 10 values.

We'll use the same 10 hash functions for every document in the dataset and generate their signatures as well. Then we can compare the documents by counting the number of signature components in which they match.

That's it!

Why Does It Work?

The reason MinHash works is that the expected value of the MinHash similarity (the number of matching components divided by the signature length) can be shown to be equal to the Jaccard similarity between the sets. We'll demonstrate this with a simple example involving two small sets:

Set A (32, **3**, 22, 6, **15**, **11**)

Set B (**15**, 30, 7, **11**, 28, **3**, 17)

There are three items in common between the sets (highlighted in bold), and 10 unique items between the two sets. Therefore, these sets have a Jaccard similarity of 3/10.

Let's look first at the probability that, for just a single MinHash signature component, we end up computing the same MinHash value for both sets.

The MinHash calculation can be thought of as taking the union of the two sets, shuffling them in a random order, and selecting the first value in the new sorted order. (Side note: the actual value used in the MinHash signature is not the original item ID, but the hashed value of that ID—but that's not important for our discussion of the probabilities).

So if you take the union of these two sets,

Union (32, **3**, 22, 6, **15**, **11**, 30, 7, 28, 17)

and then randomly shuffle them, what are the odds that one of the bold items ends up first in the list? It's given by the number of common items (3) divided by the total number of items (10), or 3/10, the same as the Jaccard similarity.

The probability that a given MinHash value will come from one of the shared items is equal to the Jaccard similarity.

Now we can go back to look at the full signature. Continuing with our simple example, if we have a MinHash signature with 20 components, on average, how many of those MinHash values would we expect to be in common? The answer is the number of components (20) times the probability of a match (3/10), or 6 components. The expected value of the MinHash similarity, then, would be 6/20 = 3/10, the same as the Jaccard similarity.

The expected value of the MinHash similarity between two sets is equal to their Jaccard similarity.

Example Python Code

You can find my example code on GitHub here.

If you're not familiar with GitHub, fear not. Here's the direct link to the zip file containing all of the code.

Notes on the history of the code

After working through this material in the MMDS course, I played with the Python code from GitHub user rahularora here. It's a complete implementation, but it had a couple important issues (including one fatal bug) that I've addressed.

The most important notes:

- The modulo constant in his random hash function (the variable `c` in the equation above) was not a prime number. If this constant isn't prime, the random hash produces a lot of collisions, and the algorithm doesn't work well.
- I've replaced the example dataset with a much larger one I found here.
- rahularora made use of Python dictionaries to perform the shingle hashing, but this doesn't scale well—it got really slow for a large number of documents / shingles. I've replaced this with CRC32 hashing.

Cite

McCormick, C. (2015, June 12). *MinHash Tutorial with Python Code*. Retrieved from http://www.mccormickml.com

presuming the set of number 0-255...

h(x) = (ax+b)%c

...with a=100, b=1, and c=255 (for convenience, it could be 200, or 150 and the same thing happens)...

h(1) = (100x1+1)%255
= (101)%255
= 101

h(52) = (100x52+1)%255
= (521)%255
= 101

h(103) = (100x103+1)%255 = 101
= (10301)%255
= 101

Also 154, 205, and (though it shouldn't count here) 256.

Making c a prime number (257 would 'work' in this example) stops this.

1 ^ | v • Reply • Share ›

ben ➚ Martin Ericksson • a year ago • edited

Thanks Eddie, that could be, but from what I understand it is when two different inputs are hashed to the same output. Such as two different names "Michael" and "Toby" receive the same index number. https://en.wikipedia.org/wi...

^ | v • Reply • Share ›

Long Nguyen Vu • 2 years ago

by far the best one to get started with MinHash

4 ^ | v • Reply • Share ›

Chris McCormick Mod ➚ Long Nguyen Vu • 2 months ago

Thanks!

^ | v • Reply • Share ›

Tom Pollard • 2 months ago

The MMDS course is apparently not offered on Coursera anymore. It's now available directly from Stanford, here, https://lagunita.stanford.e...

^ | v • Reply • Share ›

Chris McCormick Mod ➚ Tom Pollard • 2 months ago

Thanks Tom, I've updated the post with that link!

^ | v • Reply • Share ›

PK • 7 months ago

"if we have a MinHash signature with 20 components, on average, how many of those MinHash values would we expect to be in common? The answer is the number of components (20) times the probability of a match (3/10), or 6 components. The expected value of the MinHash similarity, then, would be 6/20 = 3/10, the same as the Jaccard similarity."

why are "20" components used here?
if we were to use 10 components, we will end up with 3/10
but if we use 5 components then the answer is not 3/10

also, how do we know probability is 3/10 in minhash? because won't we get the probability only after computing the jaccard similarity? not following here

Reply • Share ›

Chris McCormick Mod ➚ PK • 2 months ago

Hi PK,
Remember that MinHash is only an *approximation* to the Jaccard Similarity, so it will not return the same result. The longer your MinHash signatures are, the more accurate of an approximation it will be... Of course, this also increases the memory and compute requirements.
Thanks,
Chris

Reply • Share ›

Ryan Moulton • 9 months ago

If you'd like to get into more depth (In particular, how to incorporate weights into minhashes), I wrote up a gentle guide to the whole family of algorithms with C++ code here: https://moultano.wordpress....

Reply • Share ›

Siri Aysh • a year ago

Can I get minhash algorithm in Java code?

Reply • Share ›

Pixelarter • a year ago

Isn't this the same as picking a random subset of A, and a random subset of B, and comparing their similarity?

Like if the number of components in the hash is < the number of items in the union set, you are just comparing a subset of A to a subset of B. Which is just an approximation of the Jaccard similarity, less computationally expensive but prone to fail (no match when it should).

While if the number of components in the hash is > the number of items in the union set, then it's still an approximation prone to fail due to being stochastic, but the number of comparisons would exceed computing the Jaccard similarity?

Reply • Share ›

Гриша Кушнир • 2 years ago

Thank you.

Reply • Share ›

Yuqiong Li • 2 years ago

Thanks Chris for the sharing. I'm trying to replicate your code as to learn. Can I ask why do you write this line of code?

Hash the shingle to a 32-bit integer.

crc = binascii.crc32(shingle) & 0xffffffff

Reply • Share ›

Yuqiong Li ➚ Yuqiong Li • 2 years ago

Is this the CRC32 hashing you are referring to? I just am kind of confused why this is a hash function, as the doc suggests binascii is used for "Convert between binary and ASCII"?

Reply • Share ›

ben ➚ Yuqiong Li • a year ago • edited

Converting and between binary and ascii sounds a lot like a hashing operation and apparently it can be used for that. However, it is apparently not secure (I think this means 'not cryptographic'). https://stackoverflow.com/q...

7 Reply • Share ›

Saish Sali • 2 years ago

Amazing! Thanks for such a lovely tutorial :)

Reply • Share ›

Kerem Kurban • 2 years ago

Great explaining but could you give more eloborate explanation on how can we use these algortihm on our respective data. I saw that you tagged each sentences with t#, i did the same for my data. But I don't have a .truth file cuz I don't know the results yet. It would be better if the program is more user-friendly. Thanks

Ps. here is a error I got when I tried this code on my data.

Reply • Share ›

Hassaan Ijaz ➚ Kerem Kurban • a year ago

Hi Kareem, Could you please share the details on how you fixed the error as I am getting it as well. Your help will be very much appreciated!

Reply • Share ›

Kerem Kurban ➚ Hassaan Ijaz
• a year ago • edited

Hi, I forgot how did I fix it. Perhaps I altered some code in runMinHashExample.py. I checked my train file and saw that after adding t# to the articles, I manually deleted some of them. Perhaps its irrelevant. Yet here is the code. I tried and it works. https://we.tl/vrha9a5QMA Ofc you need to change it again to fit file names etc. to your sample

Reply • Share ›

Kerem Kurban ➚ Kerem Kurban • 2 years ago

View — uploads.disquscdn.com

Reply • Share ›

Kerem Kurban ➚ Kerem Kurban • 2 years ago

nvm solved it, thx for the code. if you can give me your reference i can state it when the time comes

Reply • Share ›

Arturo Gutierrez ➚ Kerem Kurban
• 2 years ago

How did you solve that bug? I also have the same error.

Reply • Share ›

Chris McCormick Mod ➚ Kerem Kurban
• 2 years ago • edited

Thanks for the feedback. When you say making the program more user-friendly, I assume you mean making it easier to apply to your own text data? Currently the

Related posts

BERT Research - Ep. 1 - Key Concepts & Sources 11 Nov 2019
GLUE Explained: Understanding BERT Through Benchmarks 05 Nov 2019
Matrix Operations in NumPy vs. Matlab 28 Oct 2019