Text Classification is Your New Secret Weapon

Natural Language Processing is Fun! Part 2

Adam Geitgey
Aug 15, 2018 · 15 min read

This article is part of an on-going series on NLP: Part 1, Part 2, Part 3, Part 4. You can also read a reader-translated version of this article in 普通话.

Giant update: *I've written a new book based on these articles! It not only expands and updates all my articles, but it has tons of brand new content and lots of hands-on coding projects. Check it out now!*

In this series, we are learning how to write programs that can understand text written by humans. In Part 1, we built a Natural Language Processing pipeline where we processed English text by methodically parsing out grammar and structure.

This time, we are going to learn about text classification — the secret weapon that NLP developers use to build cutting edge systems with relatively dumb models.

The kind of results you can get with text classification compared to the development effort is off the charts. Let's go!

Bottoms Up

The NLP pipeline that we set up in Part 1 processes text in a top down way. First we split text into sentences, then we break sentences down into nouns and verbs, then we figure out the relationships between those words, and so on. It's a very logical approach and logic just *feels right*, but logic isn't necessarily the best way to go about extracting data from text.

A lot of user-created content is messy, unstructured and, some might even say, nonsensical:

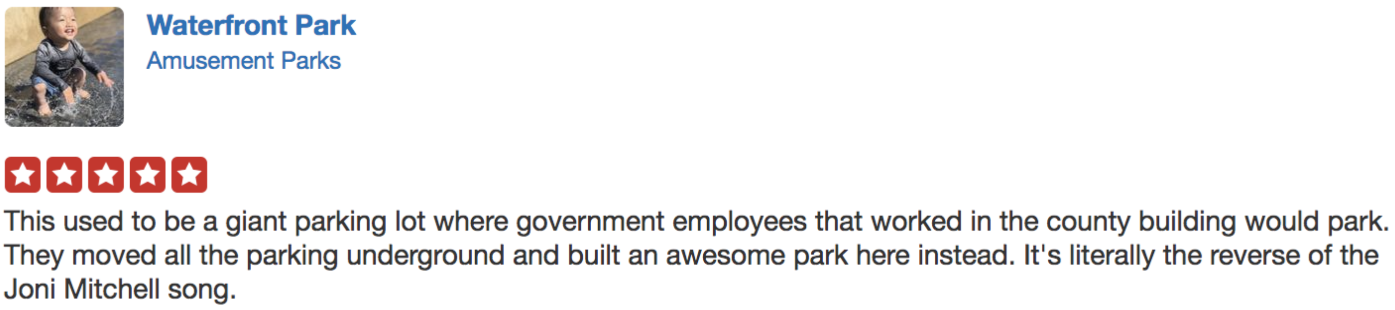
seems like more and more, every young professional should have at all times, a bag to keep papers in

1:18 PM - 13 Feb 2018

Extracting data from messy text by analyzing it’s grammatical structure is very challenging because the text doesn’t follow normal grammatical rules. We can get often get better results using dumber models that work from the bottom up. Instead of analyzing sentence structure and grammar, we’ll just look for statistical patterns in word use.

Using Classification Models to Extract Meaning

Let’s look at user reviews, one of the most common types of online data that you might want to parse with a computer. Here is one of my real Yelp reviews for a public park:



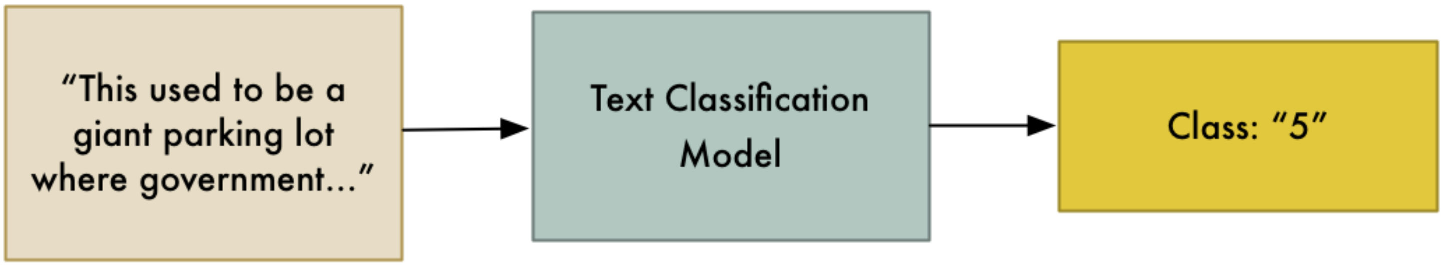
This used to be a giant parking lot where government employees that worked in the county building would park. They moved all the parking underground and built an awesome park here instead. It's literally the reverse of the Joni Mitchell song.

From the screenshot, you can see that I gave the park a 5-star review. But if I had posted this review without a star rating, you would still automatically understand that I liked the park from how I described it.

How can we write a program that can read this text and understand that I liked the park even though I never directly said “I like this park” in the text? The trick is to reframe this complex language understanding task as a simple classification problem.

Let’s set up a simple linear classifier that takes in words. The input to the classifier is the text of the review. The output is one of 5 fixed labels — “1 star”, “2 stars”, “3 stars”, “4 stars”, or “5 stars”.

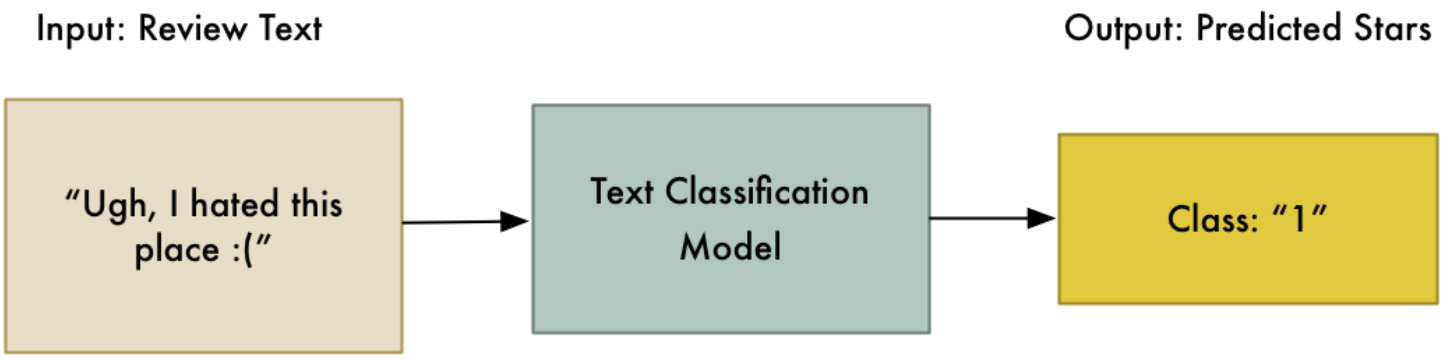
Input: Review Text **Output: Predicted Stars**


"This used to be a giant parking lot where government..."
Text Classification Model
Class: "5"

If the classifier was able to take in the text and reliably predict the correct label, that means it must somehow understand the text enough to extract the overall meaning of whether or not I liked the the park. Of course the model's level of "understanding" is just that it churns some data through a statistical model and gets a most likely answer. It's not similar to human intelligence. But if the end result is the same most of the time, then it doesn't really matter.

To train our text classification model, we'll collect a lot of user reviews of similar places (parks, businesses, landmarks, hotels, whatever we can find…) where the user wrote a text review and assigned a similar star rating. And by *lots*, I mean millions of reviews! Then we'll train the model to predict a star rating based on the corresponding text.

Once the model is trained, we can use it to make predictions for new text. Just pass in a new piece of text and get back a score:



With this simplistic model, we can do all kinds of useful things. For example, we could start a company that analyzes social media trends. Companies would hire us to track how their brand is perceived online and to alert them of negative trends in perception.

To build that, we'd just scan for any tweets that mentioned our customer's business. Then we'd feed all those tweets into the text classification model to predict if each user likes or dislikes the business. Once we have numerical ratings representing each user's feelings, we could track changes of average score over time. We could even automatically trigger an action whenever someone posts something very negative about the business. Free start-up idea!

Why does this work? It seems too simple!

On it’s face, using text classification to *understand* text sounds like magical thinking. With a traditional NLP pipeline, we have to do a lot of work to understand the grammatical structure of text. With a classifier, we’re just throwing huge buckets of text into a wood chipper and hoping for the best. Isn’t human expression more nuanced and complex than that? This is the kind of over-hyping and over simplification that makes machine learning look bad, right?

There’s several reasons why treating text as a classification problem instead of as an understanding problem tends to work really well — even when using relatively simple linear classification models.

First, people constantly create and evolve language. Especially in an online word full of memes and emoji, writing code to reliably parse tweets and user reviews is going to be pretty difficult.

With text classification, the algorithm doesn’t care whether the user wrote standard English, an emoji, or a reference to Goku. The algorithm is looking for statistical relationships between input phrases and outputs. If writing ಠ_ಠ correlates more heavily with 1-star and 2-star reviews, the algorithm will pick that up even though it has no idea what a “look of disapproval” emoticon is. The classifier can still figure out what characters mean in the context of where they appear and how often they contribute to a particular output.

Second, website users don’t always write in the specific language that you expect. An NLP pipeline trained to handle American English is going to fall apart if you give it German text. It’s also going to do poorly if your user decides to write their reviews with Cockney Rhyming Slang — which is still *technically English*.

Again, a classification algorithm doesn’t care what language the text is in as long as it can at least break apart the text into separate words and measure the effects of those words. As long as you give the classifier enough training data to cover a wide range of possible English and German user reviews, it will learn to handle both just fine.

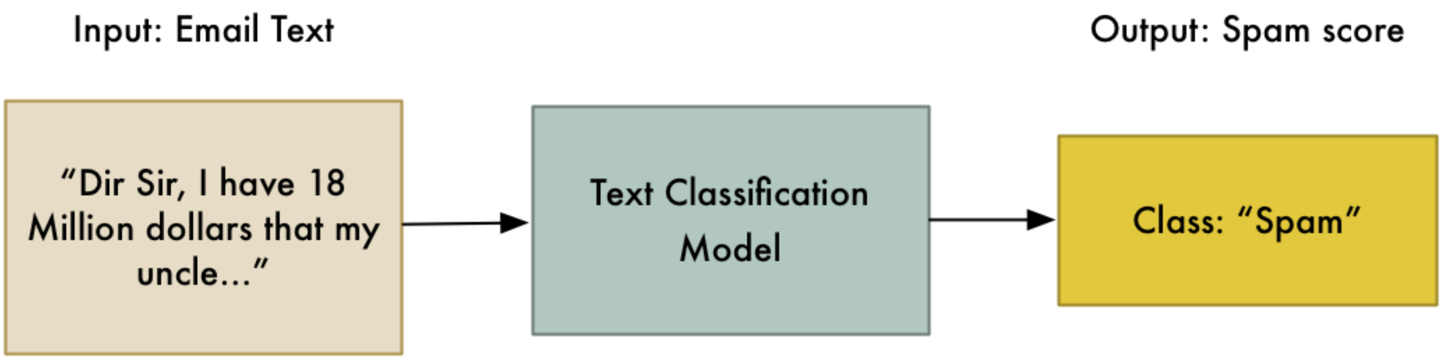
And finally, a big reason that text classification is so great is because it is *fast*. Because linear text classification algorithms are so simple (compared to more complex machine learning models like recurrent neural networks), they can be trained quickly. You can train a linear classifier with gigabytes of text in minutes on a regular laptop. You don't even need any fancy hardware like a GPU. So even if you can get a slightly better accuracy score with a different machine learning algorithm, sometimes the tradeoff isn't worth it. And research has shown that often the accuracy gap is nearly zero anyway.

While text classification models are simple to set up, that's not to say they are always ***easy*** to get working well. The big catch is that you need a lot of training data. If you don't have enough training data to cover the wide range of the ways that people write things, the model won't ever be very accurate. The more training data you can collect, the better the model will perform. The real art of applying text classification well is in finding clever ways of automatically collecting or creating training data.

What can you do with Text Classification?

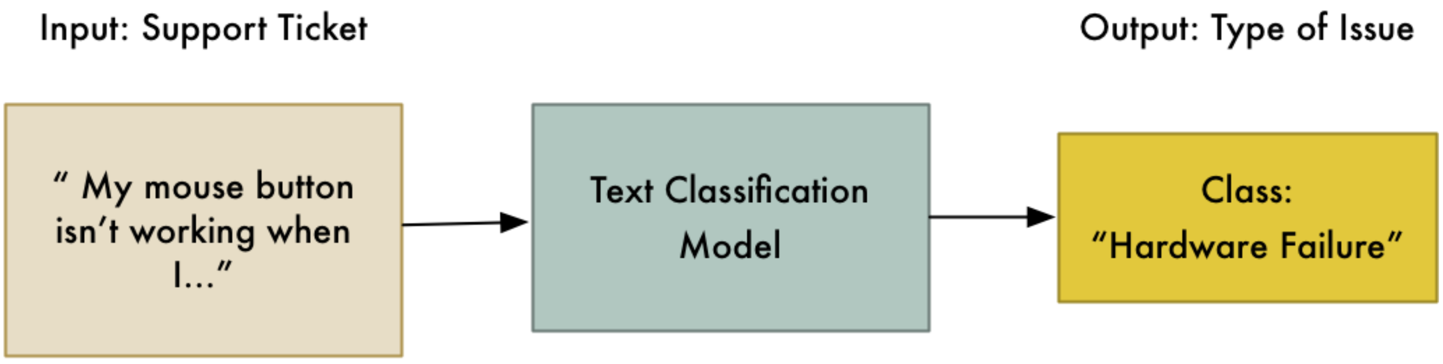
We've seen that we can use text classification to automatically score a user's review text. That's a type of sentiment analysis. Sentiment analysis is where you look at text that a user wrote and you try to figure out if the user is feeling positive or negative.

There's lots of other practical uses of text classification. One that you probably use every day as a consumer without knowing it is the email spam filtering feature built into your email service. If you have a group of real emails marked as "spam" or "not spam", you can use those to train a classification model that automatically flags spam emails in the future:


Input: Email Text
Output: Spam score
"Dir Sir, I have 18 Million dollars that my uncle…"
Text Classification Model
Class: "Spam"

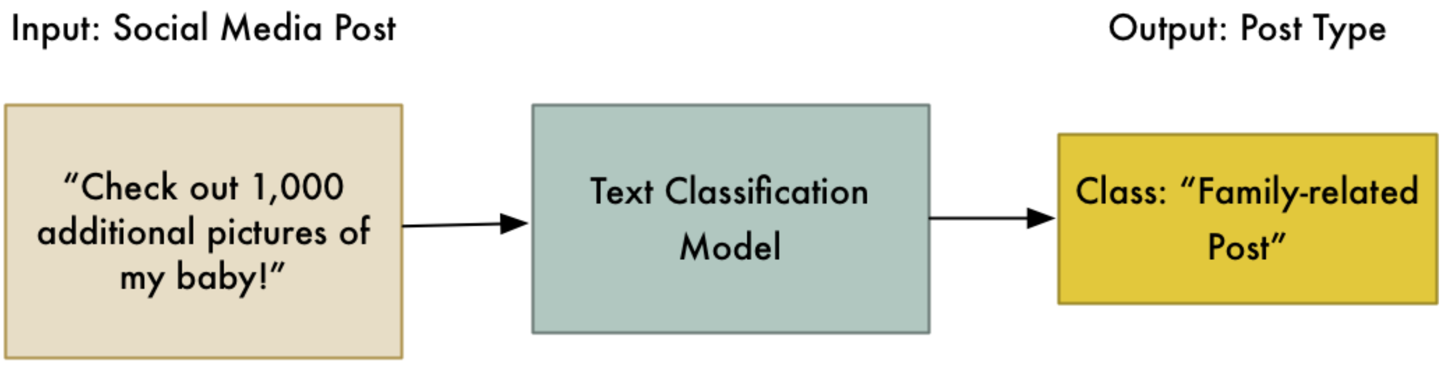
Along the lines of spam filtering, you can also use text classification to identify abusive or obscene content and flag it. A lot of websites use text classification as a first-line defense against abusive users. By also taking the model's confidence score into consideration, you can automatically block the worst offenders while sending the less certain cases to a human moderator to evaluate.

You can expand the idea of filtering beyond spam and abuse. More and more companies use use of text classification to route support tickets. The goal is to parse support questions from users and route them to the right team based on the kind of issue that the user is most likely reporting:


Input: Support Ticket
Output: Type of Issue
" My mouse button isn't working when I..."
Text Classification Model
Class: "Hardware Failure"

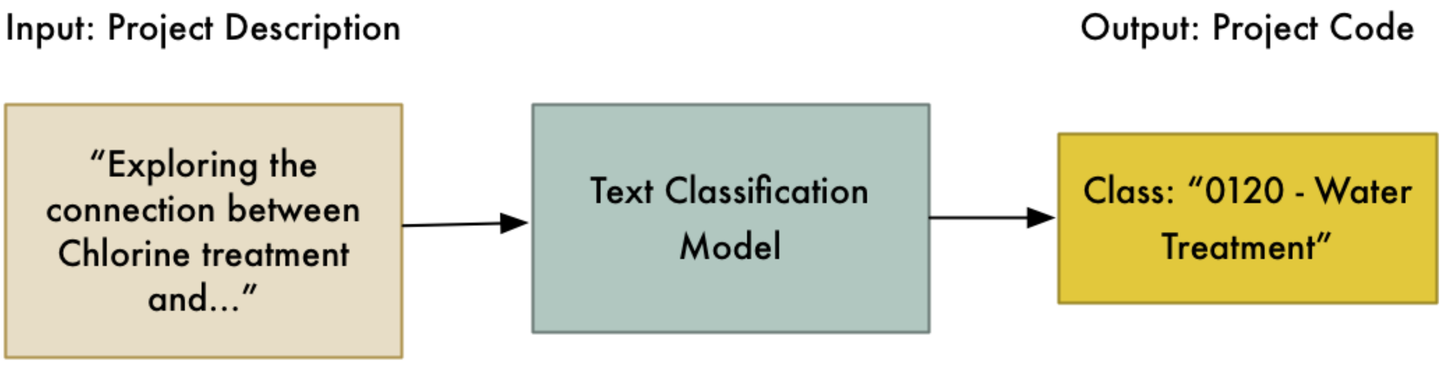
By using classification to automate the busy work of triaging support tickets, the team is freed up to spend more time actually answering questions.

Text classification models can also be used to categorize pretty much anything. You can assume that any time you post on Facebook, behind the scenes it is classifying your post into categories like "family-related" or "related to a scheduled event":


Input: Social Media Post
Output: Post Type
"Check out 1,000 additional pictures of my baby!"
Text Classification Model
Class: "Family-related Post"

That not only helps Facebook know which content to show to which users, but it also lets them track the topics that you are most interested in for advertising purposes.

Classification is also useful for sorting and labeling documents. Imagine that your company has done thousands of consulting projects for clients but that your boss wants them all re-organized according to a new government-mandated project coding system. Instead of reading through every project's summary document and trying to decide which project code is the best match, you could classify a random sampling of them by hand and then build a classification model to automatically code the remaining ones:


Input: Project Description
Output: Project Code
"Exploring the connection between Chlorine treatment and..."
Text Classification Model
Class: "0120 - Water Treatment"

These are just a few ideas. The uses of text classification are endless. You just have to figure out a way to reframe the problem so that the information you are trying to extract from the text can be mapped into a set of discrete output classes.

You can even build systems where one classification model feeds into another classification model. Imagine a user support system where the first classifier guesses the user's language (English or German), the second classifier guesses which team is best suited to handle their request and a third classifier guesses whether or not the user is already upset to choose a ticket priority code. You can get as complex as you want!

Now that you are convinced of the awesomeness of dumb text classification models, let's learn exactly how to build them!

Building the User Review Model with fastText

My favorite tool for building text classification models is Facebook's fastText. It's open source and and you can run it as a command line tool or call it from Python. There great alternatives like Vowpal Wabbit that also work well and are more flexible, but I find fastText easier to use.

You can install fastText by following these instructions.

Step 1: Download Training Data

To build a user review model, we need training data. Luckily, Yelp provides a research dataset of 4.7 million user reviews. You can download it here (but keep in mind that you can't use this data to build commercial applications).

When you download the data, you'll get a 4 gigabyte json file called `reviews.json`. Each line in the file is a json object with data like this:

```
{
  "review_id": "abc123",
  "user_id": "xyy123",
  "business_id": "1234",
  "stars": 5,
  "date":" 2015-01-01",
  "text": "This restaurant is great!",
  "useful":0,
  "funny":0,
  "cool":0
}
```

Step 2: Format and Pre-process Training Data

The first step is to convert this file into the format that fastText expects.

fastText requires a text file with each piece of text on a line by itself. The beginning of each line needs to have a special prefix of `__label__YOURLABEL` that assigns the label to that piece of text.

In other words, our restaurant review data needs to be reformatted like this:

```
__label__5 This restaurant is great!
__label__1 This restaurant is terrible :'(
```

Here's a simple piece of Python code that will read the reviews.json file and write out a text file in fastText format:

Running this creates a new file called `fasttext_dataset.txt` that we can feed into fastText for training. We aren't done yet, though. We still need to do some additional pre-processing.

fastText is totally oblivious to any English language conventions (or the conventions of any other language). As far is it knows, the words `Hello`, `hello` and `hello!` are all totally different words because they aren't exactly the same characters. To fix this, we want to do a quick pass through our text to convert everything to lowercase and to put spaces before punctuation marks. This is called text normalization and it makes it a lot easier for fastText to pick up on statistical patterns in the data.

This means that the text `This restaurant is great!` should become `this restaurant is great !`.

Here's a simple Python function that we can add to our code to do that:

Don't worry, there's a final version of the code below that includes this function.

Step 3: Split the data into a Training set and a Test set

To get an accurate measure of how well our model performs, we need to test it's ability to classify text using text that it didn't see during training. If we test it against the training data, it is like giving it an open book test where it can memorize the answers.

So we need to extract some of the strings from the training data set and keep them in separate test data file. Then we can test the trained model's performance with that held-back data to get a real-world measure of how well the model performs.

Here's a final version of our data parsing code that reads the Yelp dataset, removes any string formatting and writes out separate training and test files. It randomly splits out 90% of the data as test data and 10% as test data:

Run that and you'll have two files, `fasttext_dataset_training.txt` and `fasttext_dataset_test.txt`. Now we are ready to train!

Here's one more tip though: To make your model robust, you will also want to randomize the order of lines in each data file so that the order of the training data doesn't influence the training process. That's not absolutely required in this case since the data from Yelp is already pretty random, but it's definitely worth doing when using your own data.

Step 4: Train the Model

You can train a classifier using the fastText command line tool. You just call `fasttext`, pass in the `supervised` keyword to tell it train a supervised classification model, and then give it the training file and and an output name for the model:

```
fasttext supervised -input fasttext_dataset_training.txt -output
reviews_model
```

It only took 3 minutes to train this model with 580 million words on my laptop. Not bad!

Step 5: Test the Model

Let’s see how accurate the model is by checking it against our test data:

```
fasttext test reviews_model.bin fasttext_dataset_test.txt

N 474292
P@1 0.678
R@1 0.678
```

This means that across 474,292 examples, it guessed the user’s exact star rating 67.8% of the time. Not a bad start.

You can also ask fastText to check how often the correct star rating was in one of it’s Top 2 predictions (i.e. if the model’s top two most likely guesses were “5”, “4” and the real user said “4”):

```
fasttext test reviews_model.bin fasttext_dataset_test.txt 2

N 474292
P@2 0.456
R@2 0.912
```

That means that 91.2% of the time, it recalled the user’s star rating if we check its two best guesses. That’s a good indication that the model is not far off in most cases.

You can also try out the model interactively by running the `fasttext predict` command and then typing in your own reviews. When you hit enter, it will tell you its prediction for each one:

```
fasttext predict reviews_model.bin -
```

```
this is a terrible restaurant . i hate it so much .
__label__1
this is a very good restaurant .
__label__4
this is the best restaurant i have ever tried .
__label__5
```

Important: You have to type in your reviews in all lower case and with spaced our punctuation just like the training data! If you don't format your examples the same way as the training data, the model will do very poorly.

Step 6: Iterate on the model to make it more accurate

With the default training settings, fastText tracks each word independently and doesn't care at all about word order. But when you have a large training data set, you can ask it to take the order of words into consideration by using the `wordNgrams` parameter. That will make it track groups of words instead of just individual words.

For a data set of millions of words, tracking two word pairs (also called *bigrams*) instead of single words is a good starting point for improving the model.

Let's train a new model with the `-wordNgrams 2` parameter and see how it performs:

```
fasttext supervised -input fasttext_dataset_training.txt -output
reviews_model_ngrams -wordNgrams 2
```

This will make training take a bit longer and it will make the model file much larger (since there is now an entry for every two-word pair in the data), but it can be worth it if it gives us higher accuracy.

Once the training completes, you can re-run the test command the same way as before:

```
fasttext test reviews_model_ngrams.bin fasttext_dataset_test.txt
```

For me, using `-wordNgrams 2` got me to 71.2% accuracy on the test set, an improvement of nearly 4%. It also seems to reduce the number of obvious errors that the model makes because now it cares a little bit about the context of each word.

There are other ways to improve your model, too. One of the simplest but most effective ways is skim your training data file by hand and make sure that the preprocessing code is formatting your text in a sane way.

For example, my sample text pre-processing code will turn the common restaurant name `P.F. Chang` into `p . f . chang`. That appears as five separate words to fastText.

If you have cases like that where important words that represent a single concept are getting split up, you can write custom code to fix it. In this case, you might add code to look for common restaurant names and replace them with placeholders like `p_f_chang` so that fastText sees each as a single word.

Step 7: Use your model in your program!

The best part about fastText is that it's easy to call a trained model from any Python program.

There are a few different Python wrappers for fastText that you can use, but I like the official one created by Facebook. You can install it by following these directions.

With that installed, here's the entire code to load the model and use it to automatically score user reviews:

And here's what it looks like when it runs:

```
☆☆☆☆☆ (100% confidence)
This restaurant literally changed my life. This is the best food
I've ever eaten!

☆ (88% confidence)
I hate this place so much. They were mean to me.

☆☆☆ (64% confidence)
I don't know. It was ok, I guess. Not really sure what to say.
```

Those are really good prediction results! And let's see what prediction it would give my Yelp review:

```
☆☆☆☆☆ (58% confidence)
This used to be a giant parking lot where government employees that
worked in the country building would park. They moved all the
```

```
parking underground and built an awesome park here instead. It's
literally the reverse of the Joni Mitchell song.
```

Perfect!

This is why machine learning is so cool. Once we figured out a good way to pose the problem, the algorithm did all the hard work of extracting meaning from the training data. You can then call that model from your code with just a couple of lines of code. And just like that, your program seemingly gains superpowers.

Now go out and build you own text classifier!

. . .

If you liked this article, consider signing up for my Machine Learning is Fun! newsletter:

You can also follow me on Twitter at @ageitgey, email me directly or find me on linkedin. I'd love to hear from you if I can help you or your team with machine learning.

Machine Learning