

Natural Language Processing Is Fun Part 3: Explaining Model Predictions

Using LIME to Peek Inside the Black Box



Adam Geitgey

Jan 2, 2019 · 12 min read

This article is part of an on-going series on NLP: [Part 1](#), [Part 2](#), [Part 3](#), [Part 4](#).

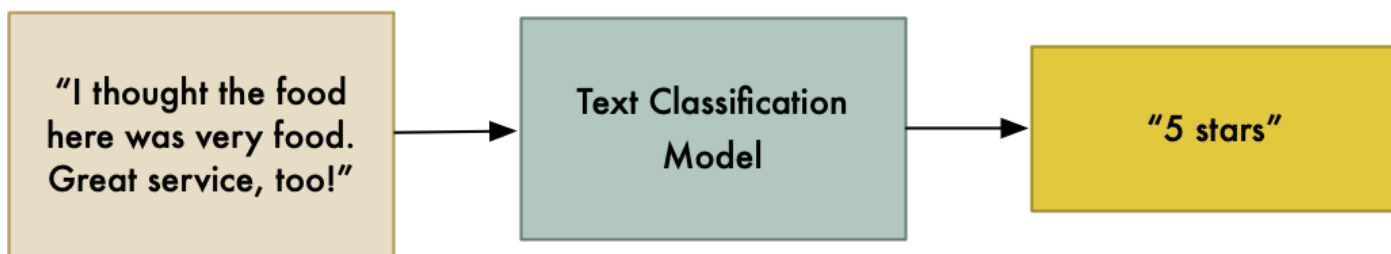
Giant update: *I've written a new book! It expands on my ML articles with tons of brand new content and lots of hands-on coding projects. Check it out now!*

In this series, we are learning how to write programs that understand English text written by humans. In Part 1, we learned how to use an NLP pipeline to understand a sentence by painstakingly picking apart its grammar. In Part 2, we saw that we don't always have to do the hard work of parsing sentence grammar. Instead, we can cheat by framing our question as a text classification problem. If we can classify a piece of text into the correct category, that means we somehow *understand* what the text says.

For example, we can extract meaning from restaurant reviews by training a classifier to predict a star rating (from 1 to 5) based only on the text of the review:

Input: Restaurant Review

Output: Predicted Star Rating



"Understanding" restaurant reviews by classifying the text as "1 star", "2 stars", "3 stars", "4 stars" or "5 stars"

If the classifier can read any restaurant review and reliably assign a star rating that accurately reflects how positively or negatively the person described the restaurant, it proves that we can extract meaning from English text!

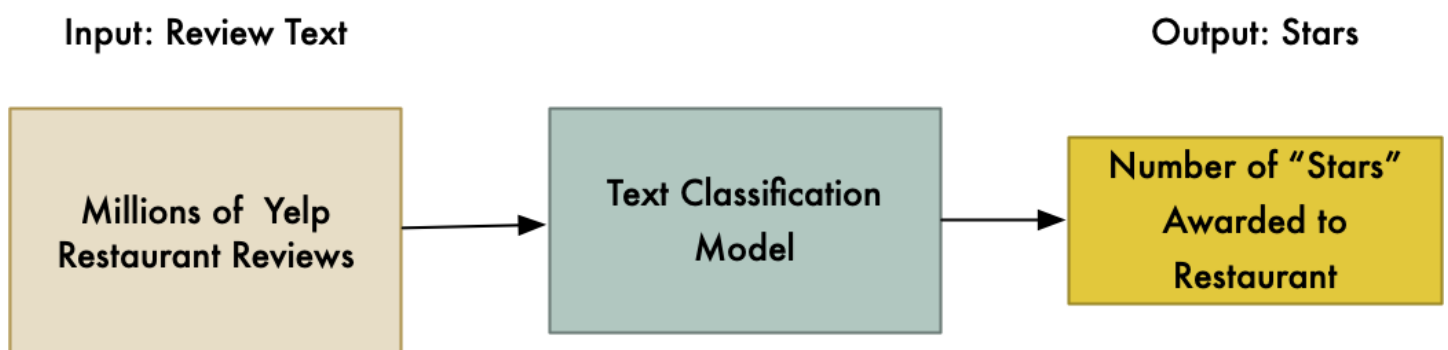
But the downside to using a text classifier is that we usually have no idea *why* it classifies each piece of text the way it does. The classifier is a black box.

The problem of model interpretability has long been a challenge in machine learning. Sometimes having a model that works but not knowing how it works isn't good enough. But thanks to new techniques like **LIME**, we now have ways to see inside the black box. Let's learn how to understand what our models are thinking!

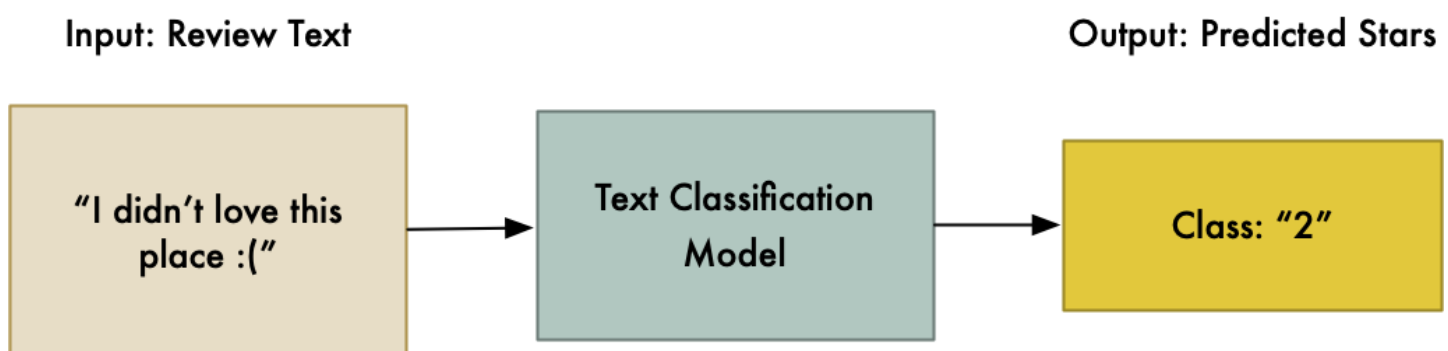
Text Classification is a Cheat Code for Understanding Language

In Part 2, we built a text classifier that can read the text of a restaurant review and predict how much the reviewer liked or didn't like the restaurant.

To do that, we collected millions of restaurant reviews from Yelp.com and then trained a text classifier using Facebook's fastText that could classify each review as either "1 star", "2 stars", "3 stars", "4 stars" or "5 stars":



Then, we used the trained model to read new restaurant reviews and predict how much the user liked the restaurant:



This is almost like a magic power! By leveraging a huge pile of data and an off-the-shelf

classification algorithm, we created a classifier that seems to understand English. And best of all, we achieved all of this without writing a single line of code to parse English sentence structure and grammar.

So, problem solved right? We've built a magic black box that can understand human language. Who cares how it works!

But Who Can You Trust?

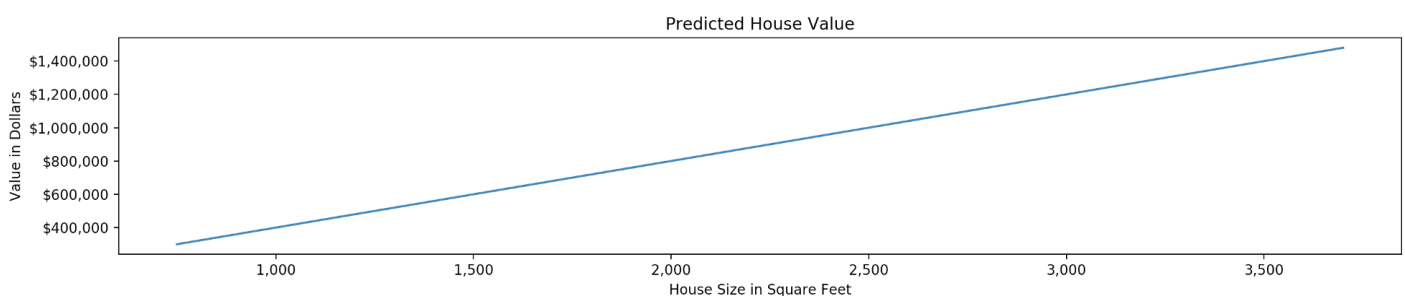
Just because we have a black-box model that seems to work doesn't mean that we should blindly trust it! Classification models tend to find the simplest characteristics in data that they can use to classify the data. Because of this, it is really easy to accidentally create a classifier that appears to work but doesn't really do what you think it does.

For example, maybe users who love a restaurant will tend to write short reviews like "I love this place!" but users who absolutely hate a restaurant will write page after page of complaints because they are so angry. How do we know that our classification model is actually understanding the words in the review and not simply classifying the reviews based on their length?

The answer is that we don't know! Just like a lazy student, it is quite possible that our classifier is taking a shortcut and not actually learning anything useful. To gain trust in the model, we need to be able to understand *why* it made a prediction.

Looking Inside a Complex Machine Learning Model

For some types of machine learning models like linear regression, we can look at the model itself and easily understand why it came up with a prediction. For example, imagine that we had a model that predicts the price of a house only based on the size of the house:



The model predicts a house’s value by taking the size of the house in square feet and multiplying it by a weight of 400. In other words, the model itself is telling us that each square foot of space is worth \$400 in extra value. Linear models like this are very easy to understand since the weights are the explanations. But in many modern machine learning models like fastText, there are just too many variables and weights in the model for a human to comprehend what is happening.

Let’s look at an example of how fastText classifies text to see why it is nearly impossible to comprehend. Let’s have our fastText model to assign a star rating to the sentence “I didn’t love this place :(”.

First, fastText will break the text apart into separate tokens:

Sentence Tokens

i

didn

'

t

love

this


place

:

(


Next, fastText will look up the “meaning” of each of these tokens. When the classifier was first trained, fastText assigned a set of numbers (called a **word vector**) to every word that appeared in the training data at least once. These numbers encode the “meaning” of each word as a point in an imaginary 100-dimensional space. The idea is that words that represent similar concepts will have similar sets of numbers representing them and words that have very different meanings will have very different sets of numbers representing them.

Here is a peek at the actual numbers from fastText that represent the meaning of each token in our sentence:

Word	One hundred numbers that encode the “meaning” of each word 																
i	-0.010	0.118	-0.026	-0.097	0.097	0.078	-0.148	0.064	0.086	-0.056	0.029	0.084	-0.124	0.141	-0.139	0.072	0.013
didn	-0.024	0.029	-0.015	-0.025	-0.017	0.034	-0.082	0.062	-0.020	0.033	0.008	0.001	0.021	-0.022	-0.060	-0.038	-0.006
'	-0.041	-0.178	0.081	0.112	0.101	0.044	-0.114	0.186	0.150	0.049	-0.019	0.125	-0.108	0.156	-0.254	0.025	-0.068
t	0.017	0.049	-0.051	-0.055	0.013	0.052	-0.066	0.027	0.009	-0.025	0.018	0.078	-0.062	0.032	-0.046	0.092	-0.034
love	0.051	0.315	-0.035	-0.191	0.243	0.181	0.026	-0.286	0.201	0.320	-0.242	-0.137	0.189	-0.049	0.242	-0.156	0.063
this	0.068	-0.051	-0.022	0.054	0.142	0.112	0.016	-0.046	0.158	0.054	-0.039	0.155	-0.112	0.177	-0.092	0.134	-0.068
place	-0.016	0.017	-0.032	-0.052	0.022	0.065	-0.034	-0.043	-0.010	0.051	-0.002	-0.002	0.034	-0.002	0.060	0.014	-0.009
:	-0.060	0.003	-0.034	-0.042	-0.059	0.021	-0.118	0.106	-0.059	0.044	0.043	-0.013	0.041	-0.037	-0.085	-0.051	-0.013
(-0.061	-0.062	0.028	0.000	0.036	-0.019	-0.065	0.100	0.062	-0.056	0.028	0.066	-0.100	0.100	-0.205	0.025	-0.040

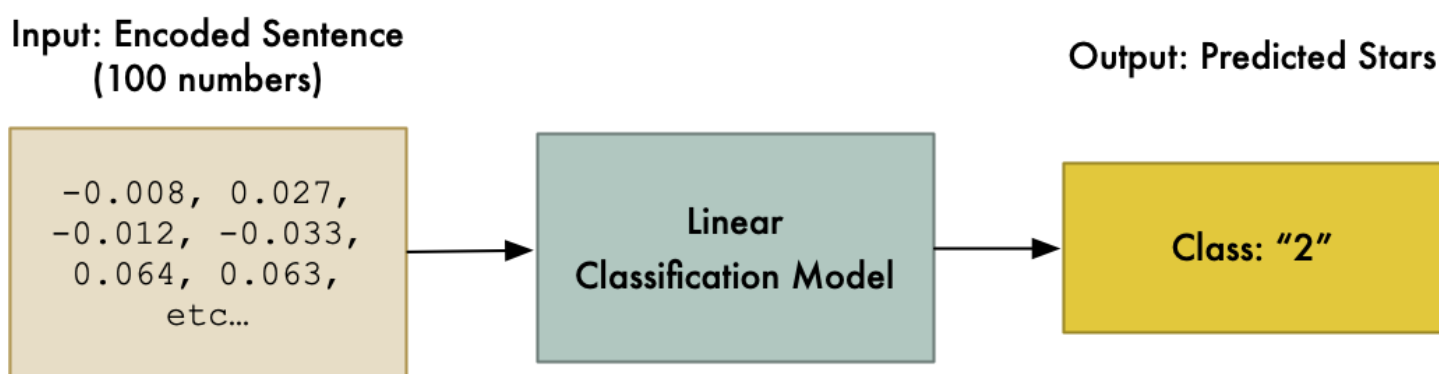
The meaning of each word is encoded by 100 values. Only the first 15 or so are shown here, but imagine the diagram continues to the right to show all 100 values.

Next, fastText will average together the vertical columns of numbers that represent each word to create a 100-number representation of the meaning of the entire sentence called a **document vector**:

Sentence	One hundred numbers that encode the "meaning" of the sentence 													
I didn't love this place : (-0.008	0.027	-0.012	-0.033	0.064	0.063	-0.065	0.019	0.064	0.046	-0.019	0.040	-0.025	

The 100 values that represent the sentence "I didn't love this place" are the average of the 100 values that represented each word. This representation is called a "Document Vector".

Finally, these one hundred values that represent the sentence are used as the input to a linear classification model that assigns the final star rating:



So even though the final rating is assigned by a linear classifier, it is assigning the rating based on 100 seemingly-random numbers that are actually derived from 900 equally inscrutable numbers. It's almost impossible for a human to work backward and understand what each of those numbers represents and why we got the result that we did. Humans just aren't very good at visualizing clusters of millions of points in 100-dimensional spaces.

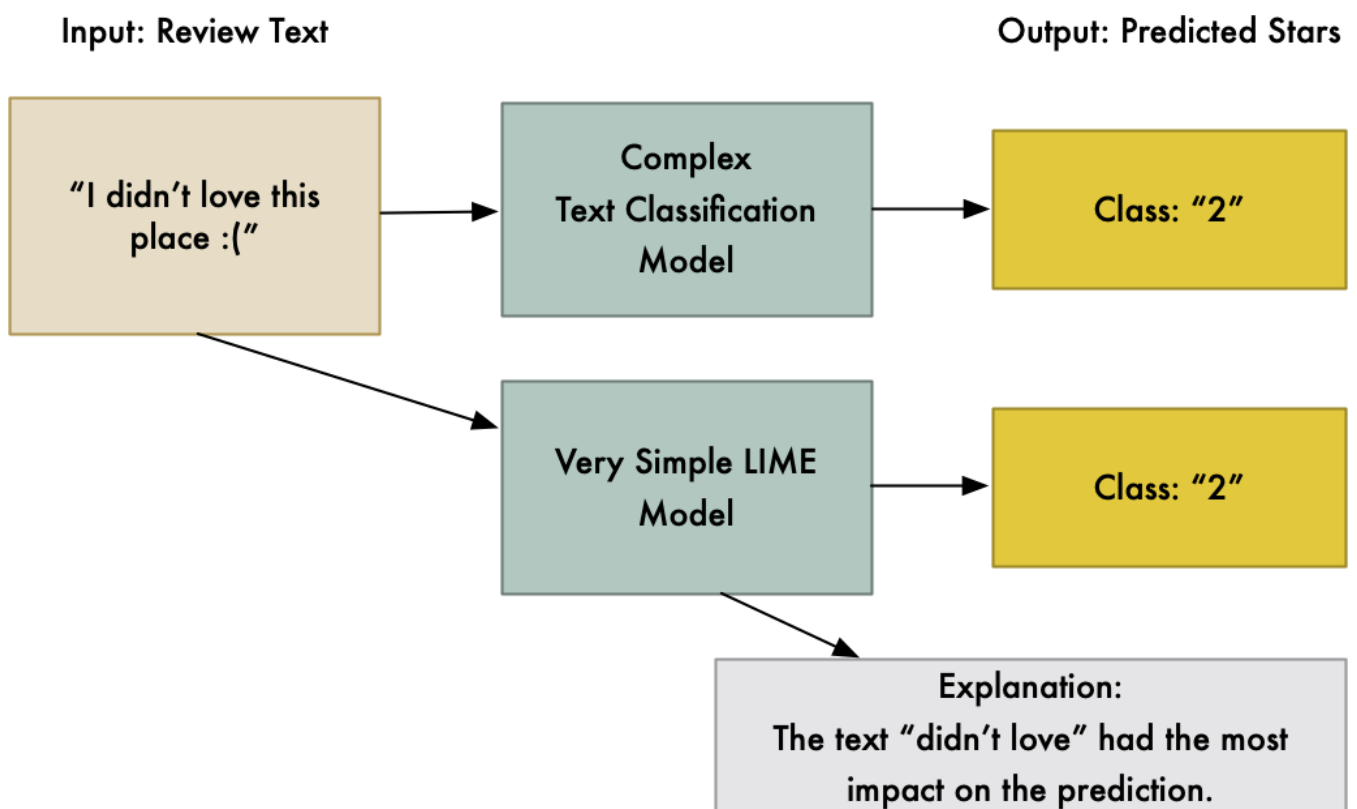
This is a huge problem for machine learning. If we are using a machine learning model to do anything important, we can't blindly trust that it is working correctly. And even if we test the model on a test dataset and get a good evaluation score, we don't know how it will behave on data that is totally different than our test dataset.

So if a model is too complex to understand, how can we possibly explain it's predictions?

We Need a Stunt Double!

In 2016, Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin released a paper titled “Why Should I Trust You?” that introduced a new approach for understanding complex models called *LIME* (short for *Local Interpretable Model-Agnostic Explanations*).

The main idea of LIME is that we can explain how a complex model made a prediction by training a simpler model that mimics it. Then we can use the simpler stand-in model to explain the original model’s prediction:

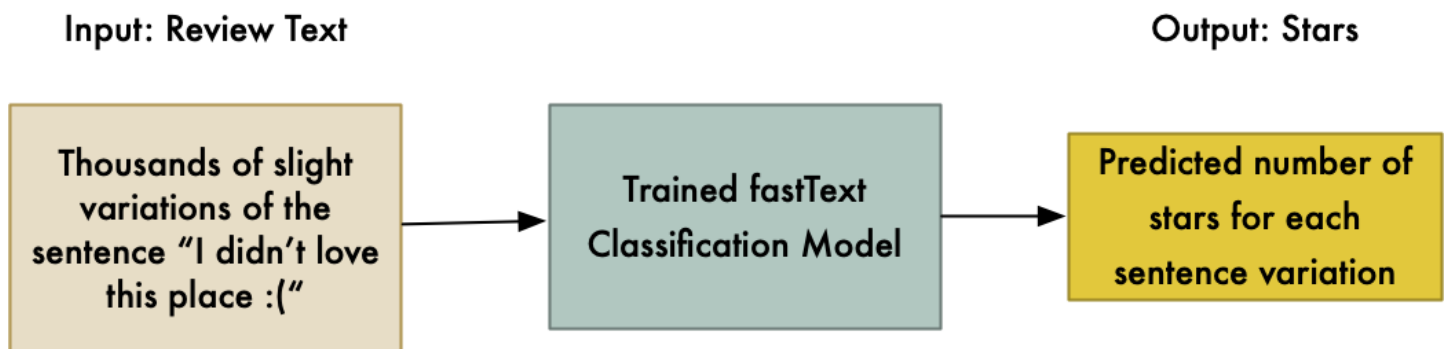


But this raises an obvious question — why does a simple model work as a reliable stand-in for a complex model? The answer is that while the simple model can’t possibly capture all the logic of the complex model for all predictions, it doesn’t need to! As long as the simple model can at least mimic the logic that the complex model used to make one single prediction, that’s all we really need.

The trick to making this work is in how we will train the stand-in model. Instead of training it on the entire dataset that we used to train the original model, we only need to train it on enough data so that it can correctly classify one sentence correctly — “I didn’t love this place :(”. As long as it classifies that one sentence using the same logic as the original model, it will work as a stand-in model to explain the prediction.

To create the data to train the stand-in model, we will run lots of variations of the sentence “I didn’t love this place :(” through the original model:

Making Predictions with the Trained fastText Model

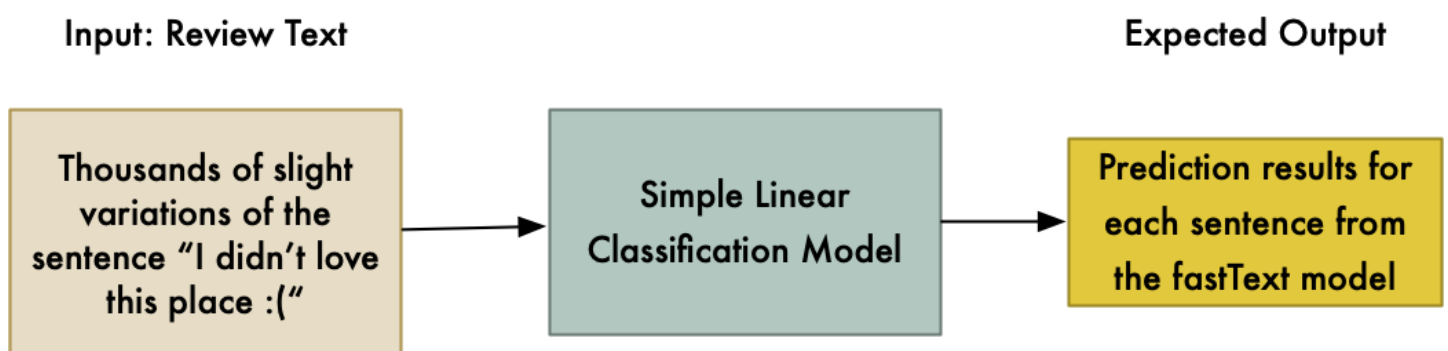


Documenting the logic of the complex fastText Classifier by asking it to make predictions for thousands of slight variations of our sentence “I didn’t love this place :(”

We will ask it to classify the sentence over and over with different words dropped out of the sentence to see how the removal of each word influences the final prediction. By asking the classifier to make predictions for many slight variations of the same sentence, we are essentially documenting how it understands that sentence.

Then we’ll use the sentence variations and classifier prediction results as the training data to train the new stand-in model:

Training the Stand-in Model



The stand-in model is trained to replicate the same predictions that the real model made for all variations of our sentence.

And then we can explain the original prediction by looking at the weights of the simple stand-in model! Because the stand-in model is a simple linear classifier that makes

predictions by assigning a weight to each word, we can simply look at the weights it assigned to each word to understand how each word affected the final prediction.

The prediction explanation we'll create looks something like this:

Word Impact on Prediction of "Two Stars"



The darker the red, the more weight that word had on this sentence getting assigned a "Two Star" rating. We can see that "didn't love" and the frowny face emoticon had a big impact and the other words didn't matter. From this, it seems like the model is valuing the correct words and that we can trust this prediction.

Keep in mind that just because we are explaining this prediction based on how much each word affected the result, that doesn't mean that the fastText model only considered single words. It just means that in this specific case, these words appearing in this context mattered the most. So even though the fastText model is very complex and considers word order in its predictions, the explanation of this single prediction doesn't need to be that complex. That's one of the insights of LIME.

LIME Step-by-Step

Let's walk through a real example of the LIME algorithm to explain a real prediction from our fastText classifier. But to make things a little more interesting, let's expand our review text a little bit. Instead of just the sentence "I didn't love this place :(", let's expand our restaurant review to three sentences:

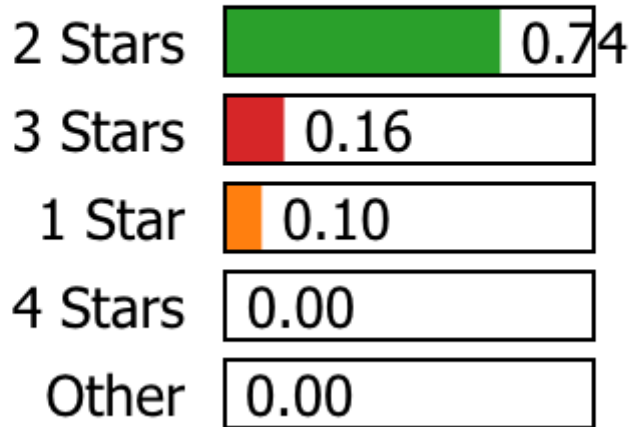
I didn't love this place :(The food wasn't very good and I didn't like the service either. Also, I found a bug in my food.

First, we'll see how many stars our fastText model will assign this review and then we'll use LIME to explain how we got that prediction.

Step 1: Make the Initial Prediction with fastText

The first step is to pass this review into the fastText classifier. Here are the probability scores that we get back from fastText for each possible number of stars:

Prediction probabilities



Great, it gave our review “2 Stars” with a 74% probability of being correct. That seems about right based on the text and how people tend to score Yelp reviews, but we don’t yet know how that prediction was determined. Let’s use LIME to find out!

Step 2: Create Variations of the Review

The first step in LIME is to create thousands of variations of the text where we drop different words from the text. Here are some of the 5,000 variations that LIME will automatically create from our review:

```
i didn't love this place : ( the food wasn't very good and i didn't like the service either . also , i found a bug in my food .  
i __t love this place : _ the food wasn't very good and i didn't like the service either . also , i found a bug in _ food .  
i didn't love this place : ( the food wasn't very good and i didn't like the service either . also , _ found a _ in my food .  
__ __this place ( __ __t __good __didn't __the __either __also __found __my __  
__didn't __place : ( the food wasn't __ __i didn' __like the service either . also , __ found __bug in my food .  
__ __t love __place : ( the food wasn't very good and i __t like the __either . also , __ a bug in my food .  
__ __wasn' __ , __ __food  
i didn't love this place : ( the food __'t very good and i didn't like the service either . also __ i found a _ in _ food .  
i __'t love this __ ( _ food wasn't very _and __didn't like the service either . also __ __bug in __  
i didn' __love this place ( the __wasn't very _and __ __t like the _either . __ __found a _in my food .  
i didn' __place : ( the food __'t very good and i didn't like the service either . also , i found a bug in my food .  
i didn't love this place : ( the food wasn't very good and i didn't like __service either . also , i found a bug in my food .  
i __t love this place : ( the food wasn't very good and i didn't __the __either . also , i found a bug in my __ .  
i __t __place __the food wasn't __ __t like __service either . , __a in __ .
```

Step 2: Make Predictions with the Original Model

The next step is to run these thousands of text variations through our original fastText classification model to see how it classifies each one:

i didn't love this place : (the food wasn't very good and i didn't love the service either . also , i found a bug in my food .	=> 2 Stars
i didn't love this place : (the food wasn't very good and i didn't love the service either . also , i found a bug in my food .	=> 2 Stars
i didn't love this place : (the food wasn't very good and i didn't love the service either . also , i found a bug in my food .	=> 4 Stars
i didn't love this place : (the food wasn't very good and i didn't love the service either . also , i found a bug in my food .	=> 2 Stars
i didn't love this place : (the food wasn't very good and i didn't love the service either . also , i found a bug in my food .	=> 2 Stars
i didn't love this place : (the food wasn't very good and i didn't love the service either . also , i found a bug in my food .	=> 3 Stars
i didn't love this place : (the food wasn't very good and i didn't love the service either . also , i found a bug in my food .	=> 5 Stars
i didn't love this place : (the food wasn't very good and i didn't love the service either . also , i found a bug in my food .	=> 3 Stars

We'll save the prediction result for each text variation to use as training data for the stand-in model.

Step 3: Train the Stand-In Model

The data we created in Step 2 becomes our new training data set. We'll use it to train a linear classifier that mimics the behavior of the original classifier. We'll feed in the same sentence variations and ask the linear classifier to predict the same star ratings.

By default, LIME builds the stand-in linear classifier using the Ridge Regression classification implementation built into the scikit-learn library, but you can configure LIME to use a different classification algorithm if you want. Any kind of linear classifier should work as long as it assigns a single weight to each word in the text.

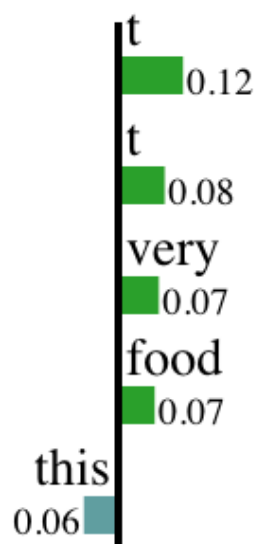
Step 4: Examine the Stand-In Model to Explain the Original Prediction

Now that the stand-in model is trained, we can explain the prediction by looking at which words in the original text have the most weight in the model.

LIME can create a per-word visualization of which words impacted the prediction the most:

NOT 2 Stars

2 Stars



The word "t" had the biggest impact. This is because "t" is from "didn't" and is the signal to the model that you are negating the next word.

On their own, these words don't give us a great picture of what is going on in the model. But LIME can also generate a visualization that color-codes each word based on how much it influenced the prediction positively or negatively:

Text with highlighted words

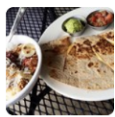
i didn't love this place ; (the food wasn't very good and i didn't like the service either . also , i found a bug in my food .

In this diagram, the green words made the classifier think that the review was more negative while the blue words made the review seem more positive.

We can see that the word “good” on its own is a positive signal but the phrase “food wasn’t very good” is a very negative signal. And the “t” is such a strong negative signal because it negates the word “good”. We can also see that the word “bug” is a negative signal — which makes sense in a restaurant review!

Based on this explanation, we have a much better idea of why the classifier gave this review a low score. And thanks to LIME, we can do the same thing with any other review — including ones that are more complex than this simple example.

Here’s a real restaurant review that I wrote that our fastText model correctly predicts as a four-star review:



Kensington Cafe

\$\$ · Breakfast & Brunch, Bars, American (Traditional)

4141 Adams Ave

San Diego, CA 92116



3 check-ins

This place is pretty great as long as you are expecting the right thing.

This is a comfy little cafe that has good food and great music. However, it's not that big. So if it is really busy, the service might be slow. So if it looks annoying busy, maybe come back later instead of writing a bad review.

They have a really cool 'Smoeres' dessert. They basically scoop out a Sterno gel heater (like caterers use) and dump the flamable gel into a little cast iron mini-grill. Then they give it to you with sticks, marshmallows, grahams and chocolate. You get to roast your own marshmallows indoors over an open flame!

It's the funnest dessert ever, especially when your marshmallow turns into a giant fireball. There's no way the fire marshal would ever approve of this, so don't tell anyone.

Finally, this place is all about Happy Hour. The normal prices are just a little bit too high, but the Happy Hour prices are very reasonable. Go here on any weekday night before 7pm and you'll spend a lot less and be a lot happier for it.

From an NLP standpoint, this review is annoyingly difficult to parse. All the sentences have qualifiers. But our fastText classifier seemed to understand it correctly and correctly predicted four stars. Let’s see why.

Here’s the LIME visualization:

this place is pretty great as long as you are expecting the right thing . this is a comfy little cafe that has good food and great music . however , it ' s not that big . so if it is really busy , the service might be slow . so if it looks annoying busy , maybe come back later instead of writing a bad review . they have a really cool ' smores ' dessert . they basically scoop out a sterno gel heater (like caterers use) and dump the flammable gel into a little cast iron mini-grill . then they give it to you with sticks , marshmallows , grahams and chocolate . you get to roast your own marshmallows indoors over an open flame ! it ' s the funnest dessert ever , especially when your marshmallow turns into a giant fireball . there ' s no way the fire marshal would ever approve of this , so don ' t tell anyone . finally , this place is all about happy hour . the normal prices are just a little bit too high , but the happy hour prices are very reasonable . go here on any weekday night before 7pm and you ' ll spend a lot less and be a lot happier for it .

The default LIME settings use random colors in the visualizations, so in this image, purple is positive. The classifier picked up on positive phrases like “pretty great”, “good food” and “comfy” as signals that this is a positive review. It also picked up on the words “cool” and “reasonable” as positive signals. That all makes sense!

But keep in mind that this explanation is not saying that all that mattered is that the phrases “pretty great” and “good food” appeared in the text. It is saying that those words mattered the most given their context. This is an important distinction to keep in mind when looking at this visualizations.

Try LIME Yourself

You can try this out yourself. After you train the fastText classifier from Part 2, you can use the code below to explain a restaurant review using LIME.

Before you run this code, you'll need to install LIME:

```
pip3 install lime
```

Then you can simply run the code. When it finishes running, a visualization of the prediction should automatically open up in your web browser.

You should be able to apply the exact same code to any fastText classifier. Just update the model path on line 22, the label names on line 34 and the text to explain on line

60. And if your classifier has more than 10 possible classes, increase the number on line 48.

. . .

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](#) [NLP](#) [Naturallanguageprocessing](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app



