How LLMs Work, Explained Without Math

Posted by Miguel Grinberg *on* May 5, 2024 *under* AI

I'm sure you agree that it has become impossible to ignore Generative AI (GenAI), as we are constantly bombarded with mainstream news about Large Language Models (LLMs). Very likely you have tried ChatGPT, maybe even keep it open all the time as an assistant.

A basic question I think a lot of people have about the GenAI revolution is where does the apparent intelligence these models have come from. In this article, I'm going to attempt to explain in simple terms and without using advanced math how generative text models work, to help you think about them as computer algorithms and not as magic.

What Does An LLM Do?

I'll begin by clearing a big misunderstanding people have regarding how Large Language Models work. The assumption that most people make is that these models can answer questions or chat with you, but in reality all they can do is take some text you provide as input and guess what the next word (or more accurately, the next *token*) is going to be.

Let's start to unravel the mystery of LLMs from the tokens.

Tokens

A token is the basic unit of text understood by the LLM. It is convenient to think of tokens as words, but for the LLM the goal is to encode text as efficiently as possible, so in many cases tokens represent sequences of characters that are shorter or longer than whole words. Punctuation symbols and spaces are also represented as tokens, either individually or grouped with other characters.

The complete list of tokens used by an LLM are said to be the LLM's *vocabulary*, since it can be used to express any possible text. The byte pair encoding (BPE) algorithm is commonly used by LLMs to generate a token vocabulary given an input dataset. Just so that you have some

rough idea of scale, the GPT-2 language model, which is open source and can be studied in detail, uses a vocabulary of 50,257 tokens.

Each token in an LLM's vocabulary is given a unique identifier, usually a number. The LLM uses a *tokenizer* to convert between regular text given as a string and an equivalent sequence of tokens, given as a list of token numbers. If you are familiar with Python and want to play with tokens, you can install the `tiktoken` package from OpenAI:

```
$ pip install tiktoken
```

Then try this in a Python prompt:

```
>>> import tiktoken
>>> encoding = tiktoken.encoding_for_model("gpt-2")

>>> encoding.encode("The quick brown fox jumps over the lazy dog.")
[464, 2068, 7586, 21831, 18045, 625, 262, 16931, 3290, 13]

>>> encoding.decode([464, 2068, 7586, 21831, 18045, 625, 262, 16931, 3
'The quick brown fox jumps over the lazy dog.'

>>> encoding.decode([464])
'The'
>>> encoding.decode([2068])
' quick'
>>> encoding.decode([13])
'.'
```

You can see in this experiment that for the GPT-2 language model token 464 represents the word "The", and token 2068 represents the word " quick", including a leading space. This model uses token 13 for the period.

Because tokens are determined algorithmically, you may find strange things, such as these three variants of the word "the", all encoded as different tokens by GPT-2:

```
>>> encoding.encode('The')
[464]
>>> encoding.encode('the')
[1169]
>>> encoding.encode(' the')
[262]
```

The BPE algorithm doesn't always map entire words to tokens. In fact, words that are less frequently used do not get to be their own token and have to be encoded with multiple tokens. Here is an example of a word that this model encodes with two tokens:

```
>>> encoding.encode("Payment")
[19197, 434]

>>> encoding.decode([19197])
'Pay'
>>> encoding.decode([434])
'ment'
```

Next Token Predictions

As I stated above, given some text, a language model makes predictions about what token will follow right after. If it helps to see this with Python pseudo-code, here is how you could run one of these models to get predictions for the next token:

```
predictions = get_token_predictions(['The', ' quick', ' brown', ' fox'
```

The function gets a list of input tokens, which are encoded from the *prompt* provided by the user. In this example I'm assuming words are all individual tokens. To keep things simple I'm using the textual representation of each token, but as you've seen before in reality each token will be passed to the model as a number.

The returned value of this function is a data structure that assigns each token in the vocabulary a probability to follow the input text. If this was based on GPT-2, the return value of the function would be a list of 50,257 floating point numbers, each predicting a probability that the corresponding token will come next.

In the example above you could imagine that a well trained language model will give the token "jumps" a high probability to follow the partial phrase "The quick brown fox" that I used as prompt. Once again assuming a model trained appropriately, you could also imagine that the probability of a random word such as "potato" continuing this phrase is going to be much lower and close to 0.

To be able to produce reasonable predictions, the language model has to go through a *training* process. During training, it is presented with lots

and lots of text to learn from. At the end of the training, the model is able to calculate next token probabilities for a given token sequence using data structures that it has built using all the text that it saw in training.

Is this different from what you expected? I hope this is starting to look less magical now.

Generating Long Text Sequences

Since the model can only predict what the next token is going to be, the only way to make it generate complete sentences is to run the model multiple times in a loop. With each loop iteration a new token is generated, chosen from the returned probabilities. This token is then added to the input that is given to the model on the next iteration of the loop, and this continues until sufficient text has been generated.

Let's look at a more complete Python pseudo-code showing how this would work:

```
def generate_text(prompt, num_tokens, hyperparameters):
    tokens = tokenize(prompt)
    for i in range(num_tokens):
        predictions = get_token_predictions(tokens)
        next_token = select_next_token(predictions, hyperparameters)
        tokens.append(next_token)
    return ''.join(tokens)
```

The `generate_text()` function takes a user prompt as an argument. This could be, for example, a question.

The `tokenize()` helper function converts the prompt to an equivalent list of tokens, using `tiktoken` or a similar library. Inside the for-loop, the `get_token_predictions()` function is where the AI model is called to get the probabilitles for the next token, as in the previous example.

The job of the `select_next_token()` function is to take the next token probabilities (or predictions) and pick the best token to continue the input sequence. The function could just pick the token with the highest probability, which in machine learning is called a *greedy selection*. Better yet, it can pick a token using a random number generator that honors the probabilities returned by the model, and in that way add some variety to the generated text. This will also make the model produce different responses if given the same prompt multiple times.

To make the token selection process even more flexible, the probabilities returned by the LLM can be modified using *hyperparameters*, which are passed to the text generation function as arguments. The hyperparameters allow you to control the "greediness" of the token selection process. If you have used LLMs, you are likely familiar with the `temperature` hyperparameter. With a higher temperature, the token probabilities are flattened out, and this augments the chances of less likely tokens to be selected, with the end result of making the generated text look more creative or unusual. You may have also used two other hyperparameters called `top_p` and `top_k`, which control how many of the highest probable tokens are considered for selection.

Once a token has been selected, the loop iterates and now the model is given an input that includes the new token at the end, and one more token is generated to follow it. The `num_tokens` argument controls how many iterations to run the loop for, or in other words, how much text to generate. The generated text can (and often does) end mid-sentence, because the LLM has no concept of sentences or paragraphs, since it just works on one token at a time. To prevent the generated text from ending in the middle of a sentence, we could consider the `num_tokens` argument as a maximum instead of an exact number of tokens to generate, and in that case we could stop the loop when a period token is generated.

If you've reached this point and understood everything then congratulations, you now know how LLMs work at a high level. Are you interested in more details? In the next section I'll get a bit more technical, while still doing my best to avoid referencing the math that supports this technology, which is quite advanced.

Model Training

Unfortunately, discussing how a model is trained is actually difficult without using math. What I'm going to do is start by showing you a very simple training approach.

Given that the task is to predict tokens that follow other tokens, a simple way to train a model is to get all the pairs of consecutive tokens that appear in the training dataset and build a table of probabilities with them.

Let's do this with a short vocabulary and dataset. Let's say the model's vocabulary has the following five tokens:

```
['I', 'you', 'like', 'apples', 'bananas']
```

To keep this example short and simple, I'm not going to consider spaces or punctuation symbols as tokens.

Let's use a training dataset that is composed of three sentences:

- I like apples
- I like bananas
- you like bananas

We can build a 5x5 table and in each cell write how many times the token representing the row of the cell is followed by the token representing the column. Here is the table built from the three sentences in the dataset:

-	I	you	like	apples	bananas
I			2		
you			1		
like				1	2
apples					
bananas					

Hopefully this is clear. The dataset has two instances of "I like", one instance of "you like", one instance of "like apples" and two of "like bananas".

Now that we know how many times each pair of tokens appeared in the training dataset, we can calculate the probabilities of each token following each other. To do this, we convert the numbers in each row to probabilities. For example, token "like" in the middle row of the table was followed once by "apples" and twice by "bananas". That means that "apples" follows "like" 33.3% of the time, and "bananas" follows it the remaining 66.7%.

Here is the complete table with all the probabilities calculated. Empty cells have a probability of 0%.

-	I	you	like	apples	bananas
I			100%		
you			100%		
like				33.3%	66.7%
apples	25%	25%	25%		25%
bananas	25%	25%	25%	25%	

The rows for "I", "you" and "like" are easy to calculate, but "apples" and "bananas" present a problem because they have no data at all, since the dataset does not have any examples with these tokens being followed by other tokens. Here we have a "hole" in our training, so to make sure that the model produces a prediction even when lacking training, I have decided to split the probabilities for a follow-up token for "apples" and "bananas" evenly across the other four possible tokens, which could obviously generate strange results, but at least the model will not get stuck when it reaches one of these two tokens.

The problem of holes in training data is actually important. In real LLMs the training datasets are very large, so you would not find training holes that are so obvious as in my tiny example above. But smaller, more difficult to detect holes due to low coverage in the training data do exist and are fairly common. The quality of the token predictions the LLM makes in these poorly trained areas can be bad, but often in ways that are difficult to perceive. This is one of the reasons LLMs can sometimes hallucinate, which happens when the generated text reads well, but contains factual errors or inconsistencies.

Using the probabilities table above, you may now imagine how an implementation of the `get_token_predictions()` function would work. In Python pseudo-code it would be something like this:

```
def get_token_predictions(input_tokens):
    last_token = input_tokens[-1]
    return probabilities_table[last_token]
```

Simpler than expected, right? The function accepts a sequence of tokens, which come from the user prompt. It takes the last token in the sequence, and returns the row in the probabilities table that corresponds to that token.

If you were to call this function with `['you', 'like']` as input tokens, for example, the function would return the row for "like", which gives the token "apples" a 33.3% chance of continuing the sentence, and the token "bananas" the other 66.7%. With these probabilities, the `select_next_token()` function shown above should choose "apples" one out of three times.

When the "apples" token is selected as a continuation of "you like", the sentence "you like apples" will be formed. This is an original sentence that did not exist in the training dataset, yet it is perfectly reasonable. Hopefully you are starting to get an idea of how these models can come up with what appears to be original ideas or concepts, just by reusing patterns and stitching together different bits of what they learned in training.

The Context Window

The approach I took in the previous section to train my mini-language model is called a Markov chain.

An issue with this technique is that only one token (the last of the input) is used to make a prediction. Any text that appears before that last token doesn't have any influence when choosing how to continue, so we can say that the *context window* of this solution is equal to one token, which is very small. With such a small context window the model constantly "forgets" its line of thought and jumps from one word to the next without much consistency.

To improve the model's predictions a larger probabilities table can be constructed. To use a context window of two tokens, additional table rows would have to be added with rows that represent all possible sequences of two tokens. With the five tokens I used in the example there would be 25 new rows in the probabilities table each for a pair of tokens, added to the 5 single-token rows that are already there. The model would have to be trained again, this time looking at groups of three tokens in addition to the pairs. Then in each loop iteration of the `get_token_predictions()` function the last two tokens from the input would be used when available, to find the corresponding row in the larger probabilities table.

But a context window of 2 tokens is still insufficient. For the generated text to be consistent with itself and make at least some basic sense, a

much larger context window is needed. Without a large enough context it is impossible for newly generated tokens to relate to concepts or ideas expressed in previous tokens. So what can we do? Increasing the context window to 3 tokens would add 125 additional rows to the probabilities table, and the quality would still be very poor. How large do we need to make the context window?

The open source GPT-2 model from OpenAI uses a context window of 1024 tokens. To be able to implement a context window of this size using Markov chains, each row of the probabilities table would have to represent a sequence that is between 1 and 1024 tokens long. Using the above example vocabulary of 5 tokens, there are 5^{1024} possible sequences that are 1024 tokens long. How many table rows are required to represent this? I did the calculation in a Python session (scroll to the right to see the complete number):

```
>>> pow(5, 1024)
5562684646268003457725581793331010160548039951155829576383318542218011
```

That is a lot of rows! And this is only a portion of the table, since we would also need sequences that are 1023 tokens long, 1022, etc., all the way to 1, because we want to make sure shorter sequences can also be handled when not enough tokens are available in the input. Markov chains are fun to work with, but they do have a big scalability problem.

And a context window of 1024 tokens isn't even that great anymore. With GPT-3, the context window was increased to 2048 tokens, then increased to 4096 in GPT-3.5. GPT-4 started with 8192 tokens, later got increased to 32K, and then again to 128K (that's right, 128,000 tokens!). Models with 1M or larger context windows are starting to appear now, allowing for much better consistency and recall when they make token predictions.

In conclusion, Markov chains allow us to think about the problem of text generation in the right way, but they have big issues that prevent us from considering them as a viable solution.

From Markov Chains to Neural Networks

Obviously we have to forget the idea of having a table of probabilities, since a table for a reasonable context window would require an impossibly large amount of RAM. What we can do is replace the table

with a function that returns an approximation of what the token probabilities would be, generated algorithmically instead of stored as a big table. This is actually something that *neural networks* can do well.

A neural network is a special type of function that takes some inputs, performs some calculations on them, and returns an output. For a language model the inputs are the tokens that represent the prompt, and the output is the list of predicted probabilities for the next token.

I said neural networks are "special" functions. What makes them special is that in addition to the function logic, the calculations they perform on the inputs are controlled by a number of externally defined *parameters*. Initially, the parameters of the network are not known, and as a result, the function produces an output that is completely useless. The training process for the neural network consists in finding the parameters that make the function perform the best when evaluated on the data from the training dataset, with the assumption that if the function works well with the training data it will work comparably well with other data.

During the training process, the parameters are iteratively adjusted in small increments using an algorithm called backpropagation which is heavy on math, so I won't discuss in this article. With each adjustment, the predictions of the neural network are expected to become a tiny bit better. After an update to the parameters, the network is evaluated again against the training dataset, and the results inform the next round of adjustments. This process continues until the function performs good next token predictions on the training dataset.

To help you have an idea of the scale at which neural networks work, consider that the GPT-2 model has about 1.5 billion parameters, and GPT-3 increased the parameter count to 175 billion. GPT-4 is said to have about 1.76 trillion parameters. Training neural networks at this scale with current generation hardware takes a very long time, usually weeks or months.

What is interesting is that because there are so many parameters, all calculated through a lengthy iterative process without human assistance, it is difficult to understand how a model works. A trained LLM is like a black box that is extremely difficult to debug, because most of the "thinking" of the model is hidden in the parameters. Even those who trained it have trouble explaining its inner workings.

Layers, Transformers and Attention

You may be curious to know what mysterious calculations happen inside the neural network function that can, with the help of well tuned parameters, take a list of input tokens and somehow output reasonable probabilities for the token that follows.

A neural network is configured to perform a chain of operations, each called a *layer*. The first layer receives the inputs, and performs some type of transformation on them. The transformed inputs enter the next layer and are transformed once again. This continues until the data reaches the final layer and is transformed one last time, generating the output, or prediction.

Machine learning experts come up with different types of layers that perform mathematical transformations on the input data, and they also figure out ways to organize and group layers so that they achieve a desired result. Some layers are of a general purpose, while others are designed to work on a specific type of input data, such as images or as in the case of LLMs, on tokenized text.

The neural network architecture that is the most popular today for text generation in large language models is called the Transformer. LLMs that use this design are said to be GPTs, or Generative Pre-Trained Transformers.

The distinctive characteristic of transformer models is a layer calculation they perform called Attention, that allows them to derive relationships and patterns between tokens that are in the context window, which are then reflected in the resulting probabilities for the next token.

The Attention mechanism was initially used in language translators, as a way to find which tokens in an input sequence are the most important to extract its meaning. This mechanism gives modern translators the ability to "understand" a sentence at a basic level, by focusing on (or driving "attention" to) the important words or tokens.

Do LLMs Have Intelligence?

By now you may be starting to form an opinion on wether LLMs show some form of intelligence in the way they generate text.

I personally do not see LLMs as having an ability to reason or come up with original thoughts, but that does not mean to say they're useless. Thanks to the clever calculations they perform on the tokens that are in the context window, LLMs are able to pick up on patterns that exist in the user prompt and match them to similar patterns learned during training. The text they generate is formed from bits and pieces of training data for the most part, but the way in which they stitch words (tokens, really) together is highly sophisticated, in many cases producing results that feel original and are useful.

Given the propensity of LLMs to hallucinate, I wouldn't trust any workflow in which the LLM produces output that goes straight to end users without verification by a human.

Will the larger LLMs that are going to appear in the following months or years achieve anything that resembles true intelligence? I feel this isn't going to happen with the GPT architecture due to its many limitations, but who knows, maybe with some future innovations we'll get there.

The End

Thank you for staying with me until the end! I hope I have picked your interested enough for you to decide to continue learning, and eventually facing all that scary math that you cannot avoid if you want to understand every detail. In that case, I can't recommend Andrej Karpathy's Neural Networks: Zero to Hero video series enough.

21 comments

#1 **Vasco** said 4 months ago

Nice article!

On the conclusion about are "Do LLMs Have Inteligence?" is more about what human inteligence is. Assuming we agree that we are made out of matter, one could also say the following sentence about humans:

"I personally do not see Humans as having an ability to reason or come up with original thoughts, but that does not mean to say they're useless. Thanks to the clever calculations they perform (on the tokens) that are in the context window, Humans are able to pick up on patterns that exist in the user prompt and match them to similar patterns learned during training. The text they generate is formed from bits and pieces of training data for the most part, but the way in which they stitch words together is highly sophisticated, in many cases producing results that feel original and useful."

#2 **Rodri** said 4 months ago

A very interesting reading. Thank you for the hard work!

#3 **Miguel Grinberg** said 4 months ago

@Vasco: with all due respect, what you are saying makes absolutely no sense.

#4 **Jay** said 4 months ago

Beautiful and elegant and very useful summary! Well done!

#5 **Rabs** said 4 months ago

Hey, Miguel. Very nice reading. I wonder how all the token prediction are, in fact, related with the conversational aspects of the main LLM models, such as GPT-3 or 4. When you use a

question as a prompt and the model answers you in a very similar way that a human would, are they just completing your question (string of words) with tokens that look like an answer to a question because they are trained with questions and answers a lot, or are there other things happening behind the curtains, such as secondary algorithms (not the token prediction ones) to make the output look like a real conversation?

#6 **Miguel Grinberg** said 4 months ago

@Rabs: The model produces a human-like response because the training dataset has lots and lots of examples of human Q&A. There are no "other things" as you call them, except maybe some added logic to stop at the end of a sentence or paragraph, which the model itself does not really have any knowledge of since it works token by token.

#7 **edo** said 4 months ago

I have a feeling that very soon I will leave Gemini at work to answer questions about the environment, and I will go on a trip right away :-)

#8 **RENATO** said 4 months ago

Congratulations Miguel, as always you are very didactic in your approaches. Today you took off the magic cloak of LLM engines and explained how training is the key to success.

#9 **Eight** said 4 months ago

Would love to see you talk about Knowledge Graphs in the same way, especially since they can "solve" some of the issues like hallucinations / bias.

#10 **Miguel Grinberg** said 4 months ago

@Eight: I assume you mean RAG. I don't really find that much interesting. Maybe it does help reduce hallucinations to some extent, but it does not eliminate them. I think RAG is useful to

expand the knowledge of the LLM with private data, or with events that occurred after the training cutoff date, but it does not make LLMs more (or less) intelligent, and given that hallucinations are still possible it is still risky to use them in customer or end user facing workflows.

#11 **Mark Alex Maidique** said 4 months ago

This is very helpful and insightful. Thank you so much!

#12 **Curious Reader** said 4 months ago

This is a fantastic article but something isn't quite clicking for me. GPT-4 scored within the top 10% on the verbal section of the SAT and can score above average on tests of math and analytical reasoning. If you give GPT-4 a passage it has never seen before it can answer abstract questions about it such as "given the passage, which of these statements would the author most likely agree with" or "which of these, if true would disprove the authors core argument" better than 90% of college bound seniors. How is that possible by just probablistically predicting the next token in the sequence? There's a gap in my understanding here.

#13 **Miguel Grinberg** said 4 months ago

@Curious: what's difficult to understand is the large scale at which these models work. When you say that the model is presented with text it has never seen before, how do you know what the model has seen? The training datasets aren't public, so given a random text we cannot really know if the model has seen something close enough, and chances are that it has. Passing tests or answering general knowledge questions is possible because there's lots of test exercises and Q&As in the training set, so the model can find a similar problem and even if it has different quantities and/or variable or names it can generate the correct response thanks to the attention mechanism, which does a sort of advanced pattern matching.

By all means the way LLMs work is impressive. But it isn't magic. The model produces its answers one token at a time as I explain

in this article, without forming an idea of the answer first like we would do, and without a real understanding beyond token probabilitites.

#14 **Saurav** said 4 months ago

Hi Miguel. Fantastic article as always. Asking a question here connecting LLMs with Flask, given your contributions to the whole Flask community. I see a lot of LLM inference APIs using async servers like FastAPI. Wanted to know your thoughts on this. Do you think Flask is a better choice for APIs that server AI models too? Would love a blog or a tutorial from you focusing on that, as serving AI models has become very relevant these days.

#15 **Miguel Grinberg** said 4 months ago

@Saurav: You typically have the option to use a sync or async API to send queries to LLMs, so which option you take depends on your needs. If you use Flask you can use synchronous calls, if you use FastAPI you can go with the async version. The decision to use sync or async should not be influenced by the LLM, you should use the standard considerations to decide which approach is best for your web application.

#16 **Victor** said 4 months ago

Thanks, this was amazingly insightful for a backend engineer

#17 **Miguel Grinberg** said 4 months ago

@Victor: Just a "backend engineer", huh?

#18 **Jian** said 3 months ago

Deep to the point, Plain and easy to understand, Good article!

#19 **Yoursbest** said 3 months ago

> *Given the propensity of LLMs to hallucinate, I wouldn't trust any workflow in which the LLM produces output that goes straight to end users without verification by a human.*

I cannot agree more.

#20 **OttoGPT** said 2 months ago

Dear Miguel, thank you very much for this great article and comments. I am curious to get your opinion on the question if we could see the "reasoning" as a result of human interaction. The sence of an output will be embodied by a human as long as we are interacting as humans wich these LLM's. In your example "I like banana" could have different meaning for you and me. Are you eating it every day or earn money with selling bananas. Technically we still have a probability output but the way we interact and interpret the answer is creating reasoning on us. This is why it feels intelligent and it is also in some way. Intelligence is usually something that needs ro be embodied in something or somebody, in that case the interacting human. Maybe this is a more philosophical approach. BR

#21 **Miguel Grinberg** said 2 months ago

@OttoGPT: I'm not sure I follow what you are saying. For a human brain of course every idea is relative and has a meaning that depends on personal experience. For an LLM all there is is a large collection of examples from where to pick tokens from, using probabilities and a random number generator. The output of an LLM can be useful, but it is still a computer running a computer algorithm.

«« « » »»

Leave a Comment

Name

Email

Comment

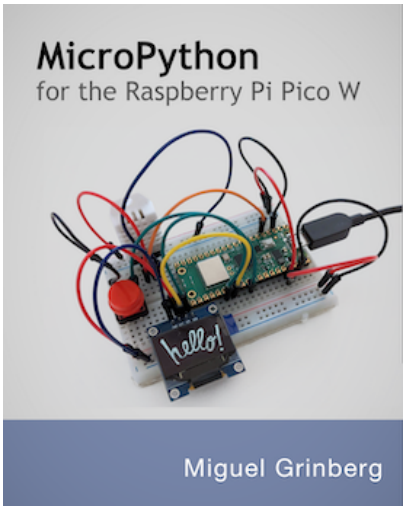
Captcha

I'm not a robot

reCAPTCHA
Privacy - Terms

Submit

MicroPython for the Raspberry Pi Pico W



If you like my MicroPython tutorial series on this blog, you may also like my MicroPython for the Raspberry Pi Pico W book.

Click here to get this Book!

About Miguel

Welcome to my blog!

I'm a software engineer and technical writer, currently living in Drogheda, Ireland.

You can also find me on Twitter, Mastodon, Github, LinkedIn, YouTube, Facebook and Patreon.

Thank you for visiting!

Categories

- AI *2*
- AWS *1*
- Arduino *7*
- Authentication *10*
- Blog *1*
- C++ *5*
- CSS *1*
- Cloud *10*
- Database *22*
- Docker *5*
- Filmmaking *6*
- Flask *126*
- Games *1*
- Heroku *1*
- IoT *8*
- JavaScript *35*
- MicroPython *9*
- Microdot *1*
- Microservices *2*
- Movie Reviews *5*
- OpenStack *1*
- Personal *3*
- Photography *7*
- Product Reviews *2*
- Programming *186*
- Project Management *1*

- Python *168*
- REST *7*
- Rackspace *1*
- Raspberry Pi *8*
- React *18*
- Robotics *6*
- Security *12*
- Video *22*
- WebSocket *2*
- Webcast *3*
- Windows *1*