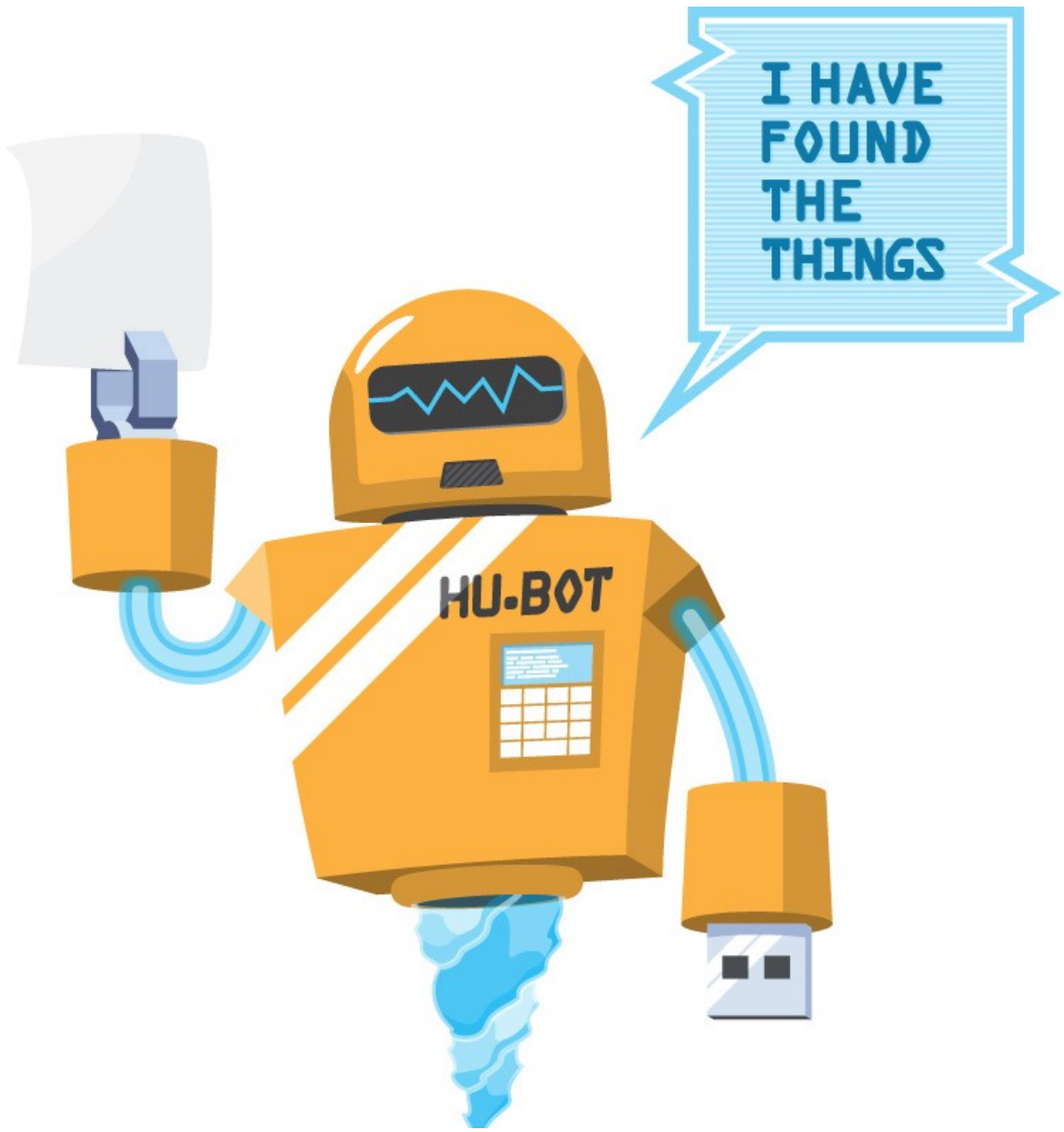
How To Create Natural Language Semantic Search For Arbitrary Objects With Deep Learning

Hamel Husain
May 29, 2018 · 13 min read

An end-to-end example of how to build a system that can search objects semantically.
By Hamel Husain & Ho-Hsiang Wu



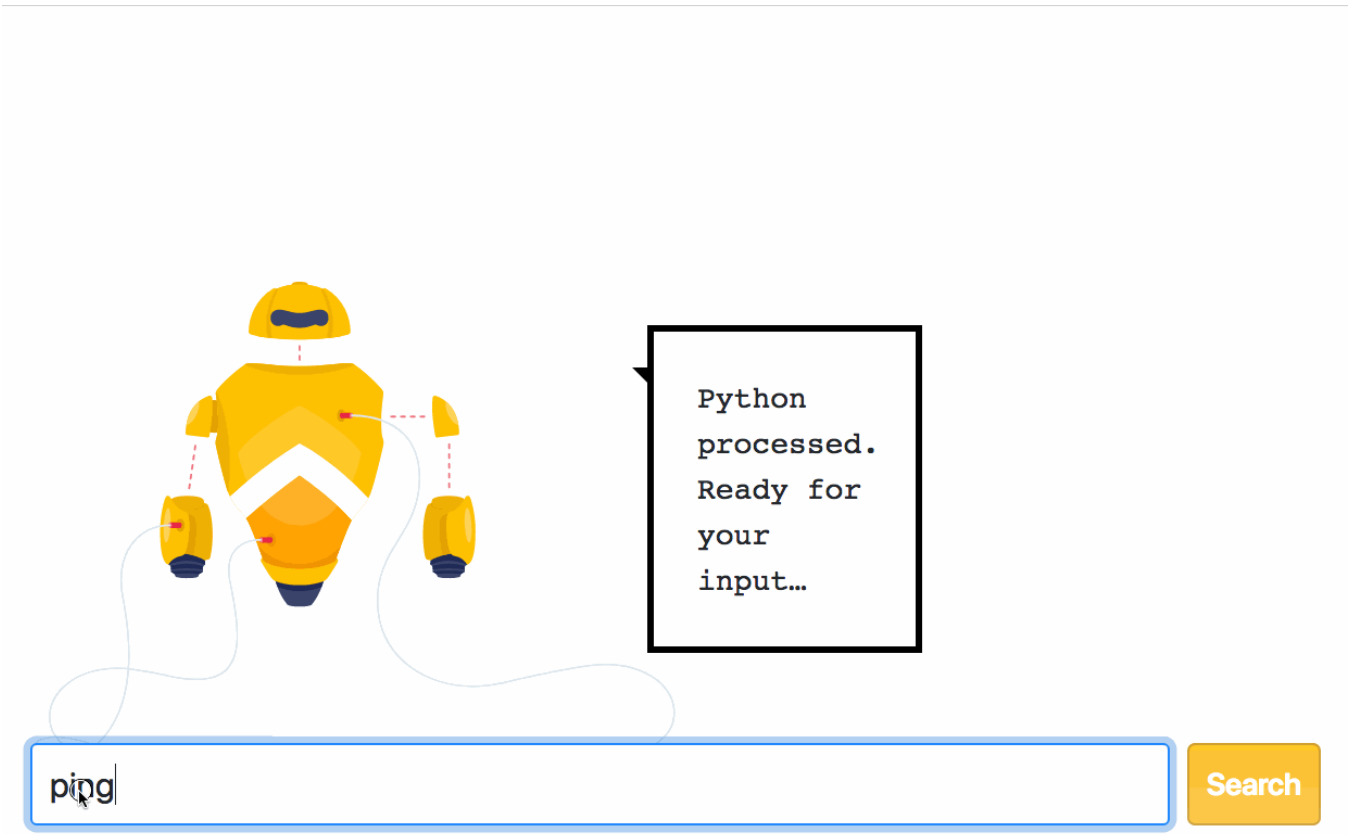
A picture of Hubot.

Motivation:

The power of modern search engines is undeniable: you can summon knowledge from the internet at a moment's notice. Unfortunately, this superpower isn't omnipresent. There are many situations where search is relegated to strict keyword search, or when the objects aren't text, search may not be available. Furthermore, strict keyword search doesn't allow the user to search semantically, which means information is not as discoverable.

Today, we share a reproducible, minimally viable product that illustrates how you can enable semantic search for arbitrary objects! Concretely, we will show you how to create a system that searches python code semantically — but this approach can be generalized to other entities (such as pictures or sound clips).

Why is semantic search so exciting? Consider the below example:

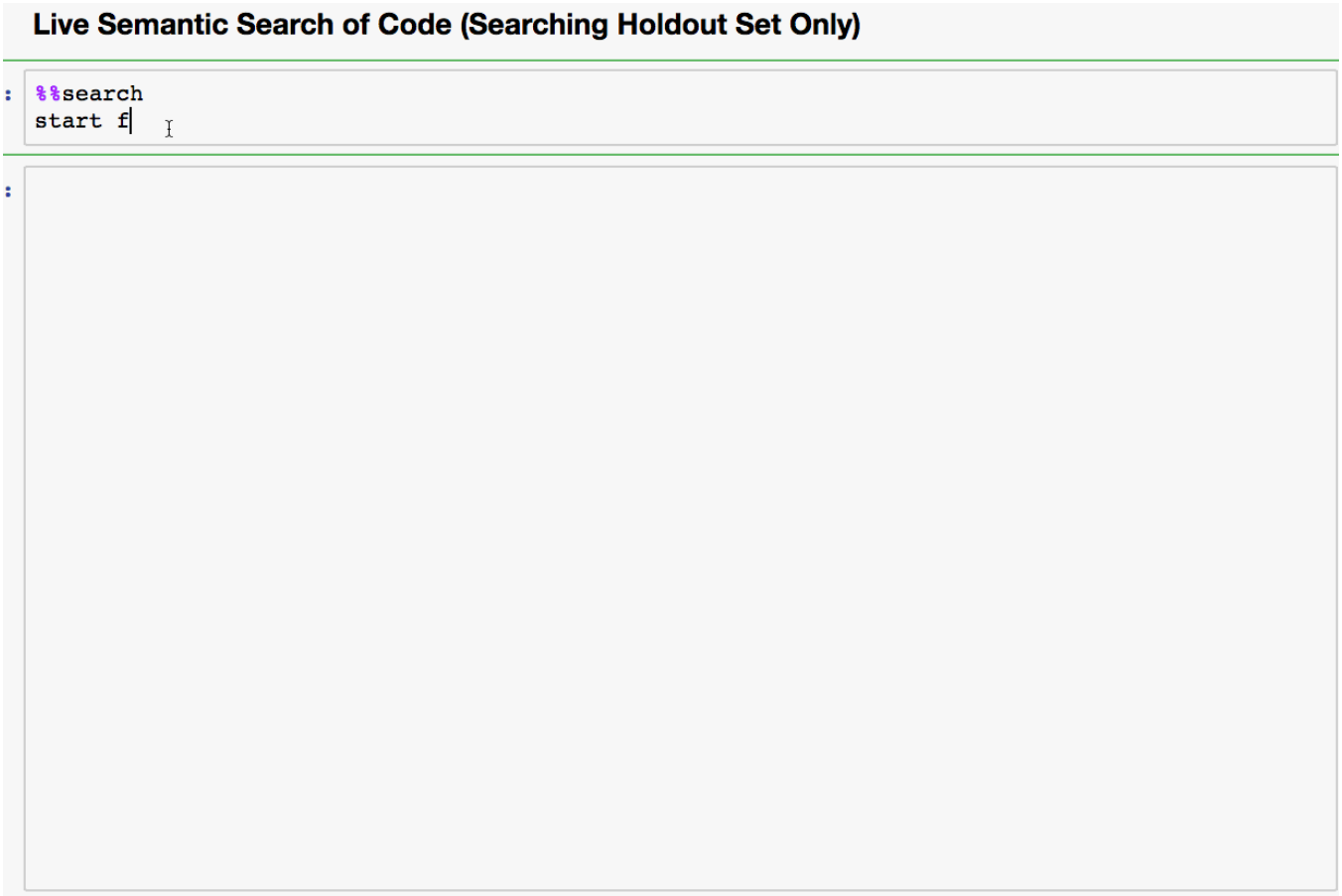


Semantic search at work on python code. *See Disclaimer section below.

The search query presented is "**Ping REST api** and return results". However, the search returns reasonable results even though the code & comments found do not contain the words **Ping, REST** or **api.**

This illustrates the **power of semantic search: we can search content for its *meaning* in addition to keywords**, and maximize the chances the user will find the information they are looking for. The implications of semantic search are profound — for example, such a procedure would allow developers to search for code in repositories even if they are not familiar with the syntax or fail to anticipate the right keywords. More importantly, you can generalize this approach to objects such as pictures, audio and other things that we haven't thought of yet.

If this is not exciting enough, **here is a live demonstration of what you will be able to build by the end of this tutorial**:

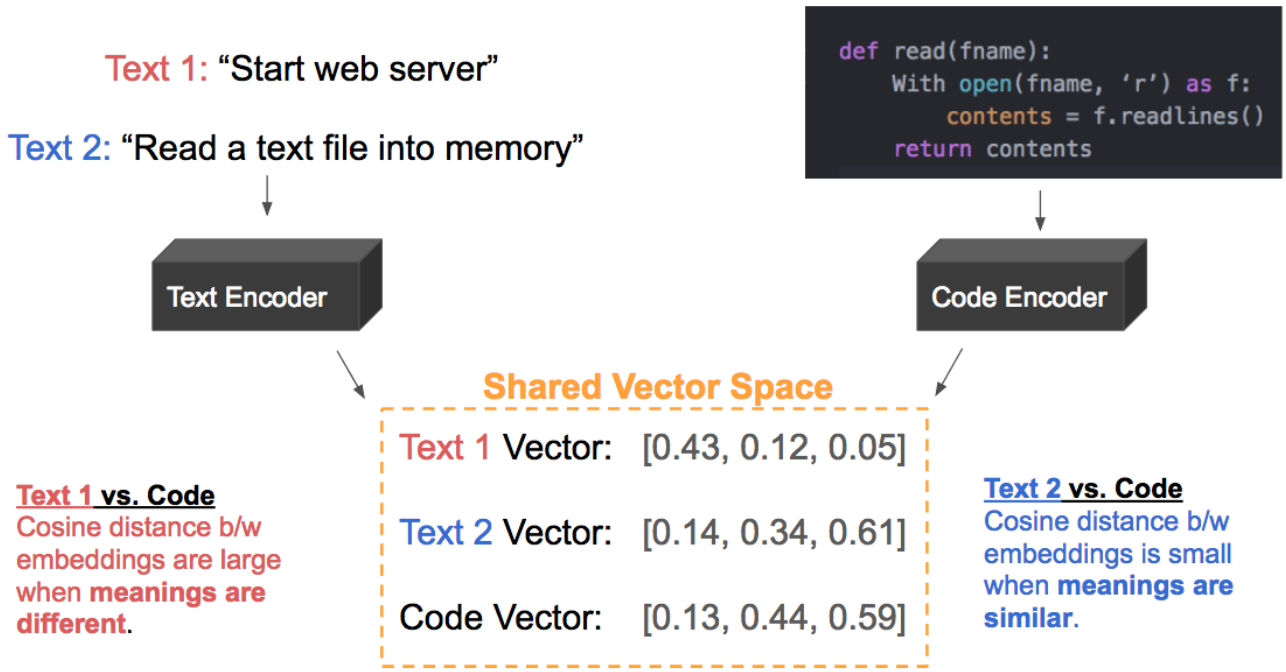


Sometimes I use Jupyter notebooks and custom magic functions to create demonstrations when I cannot build a pretty website. it can be a quick way to interactively demonstrate your work!

Intuition : Construct a Shared Vector-Space

Before diving into the technical details, it is useful to provide you with a high-level intuition of how we will accomplish semantic search. The central idea is to represent

both text and the object we want to search (code) in a shared vector space, as illustrated below:



Example: Text 2 and the **code** should be represented by similar vectors since they are directly related.

The goal is to map code into the vector space of natural language, such that (text, code) pairs that describe the same concept are close neighbors, whereas unrelated (text, code) pairs are further apart, measured by cosine similarity.

There are many ways to accomplish this goal, however, we will demonstrate the approach of taking a pre-trained model that extracts features from code and fine-tuning this model to project latent code features into a vector space of natural language. One warning: We use the term ***vector*** and ***embedding*** interchangeably throughout this tutorial.

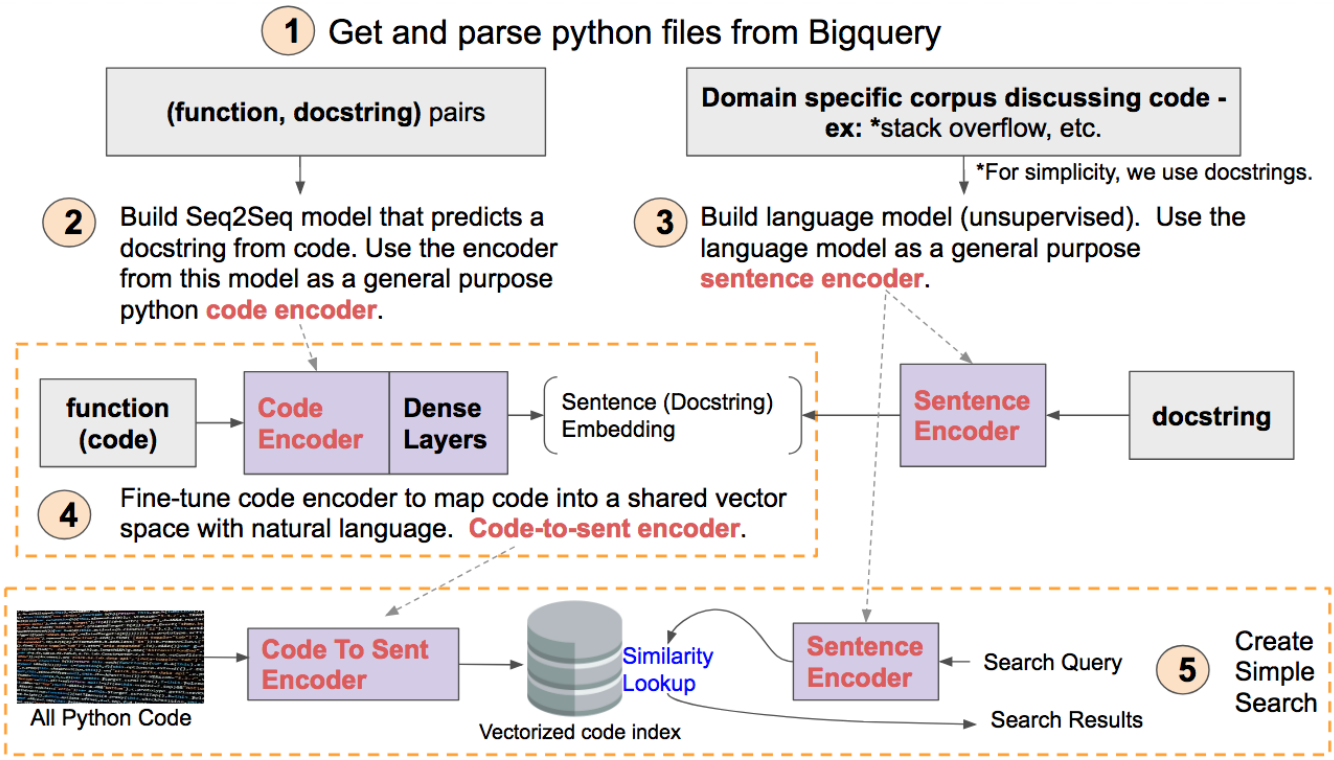
Prerequisites

We recommend familiarity with the following items prior to reading this tutorial:

- *Sequence-to-sequence models:* It will be helpful to review the information presented in a previous tutorial.
- Peruse this paper at a high level and understand the intuition of the approach presented. We draw on similar concepts for what we present here.

Overview:

This tutorial will be broken into 5 concrete steps. These steps are illustrated below and will be a useful reference as you progress throughout the tutorial. After completing the tutorial, it will be useful to revisit this diagram to reinforce how all the steps fit together.



A mind map of this tutorial. Hi-res version available here.

Each step 1–5 corresponds to a Jupyter notebook here. We will explore each step in more detail below.

Part 1 — Acquire and Parse Data:

Part 1 notebook

The folks at Google collect and store data from open-source GitHub repositories on BigQuery. This is a great open dataset for all kinds of interesting data-science projects, including this one! When you sign up for a Google Cloud account, they give you $300 which is more than enough to query the data for this exercise. Getting this data is super convenient, as you can use SQL queries to select what type of files you are looking for as well as other meta-data about repos such as commits, stars, etc.

The steps to acquire this data are outlined in this notebook. Luckily, some awesome people on the Kubeflow team at Google have gone through these steps and have

graciously hosted the data for this exercise, which is also described in this notebook.

After collecting this data, we need to parse these files into (code, docstring) pairs. For this tutorial, one unit of code will be either a top-level function or a method. We want to gather these pairs as training data for a model that will summarize code (more on that later). We also want to strip the code of all comments and only retain the code. This might seem like a daunting task, however, there is an amazing library called **ast** in Python's standard library that can be used to extract functions, methods and, docstrings. We can remove comments from code by converting code into an AST and then back from that representation to code, using the Astor package. Understanding of ASTs or how these tools work is not required for this tutorial, but are very interesting topics!

```
def tokenize_docstring(text):
    """Apply tokenization using spacy to docstrings."""
    tokens = EN.tokenizer(text)
    return [token.text.lower() for token in tokens if not token.is_space]


def tokenize_code(text):
    """A very basic procedure for tokenizing code strings."""
    return RegexpTokenizer(r'\w+').tokenize(text)


def get_function_docstring_pairs(blob):
    """Extract (function/method, docstring) pairs from a given code blob."""
    pairs = []
    try:
        module = ast.parse(blob)
        classes = [node for node in module.body if isinstance(node, ast.ClassDef)]
        functions = [node for node in module.body if isinstance(node, ast.FunctionDef)]
        for _class in classes:
            functions.extend([node for node in _class.body if isinstance(node, ast.Func

        for f in functions:
            source = astor.to_source(f)
            docstring = ast.get_docstring(f) if ast.get_docstring(f) else ''
            function = source.replace(ast.get_docstring(f, clean=False), '') if docstri

            pairs.append((f.name,
                          f.lineno,
                          source,
                          ' '.join(tokenize_code(function)),
                          ' '.join(tokenize_docstring(docstring.split('\n\n')[0]))
```

```
                            ))
    except (AssertionError, MemoryError, SyntaxError, UnicodeEncodeError):
        pass
    return pairs
```

get_code_docstring_pairs.py hosted with ❤️ by GitHub view raw

For more context of how this code is used, see this notebook.

To prepare this data for modeling, we separate the data into train, validation and test sets. We also maintain files (which we name "lineage") to keep track of the original source of each (code, docstring) pair. Finally, we apply the same transforms to code that does not contain a docstring and save that separately, as we will want the ability to search this code as well!

Part 2 — Build a Code Summarizer Using a Seq2Seq Model:

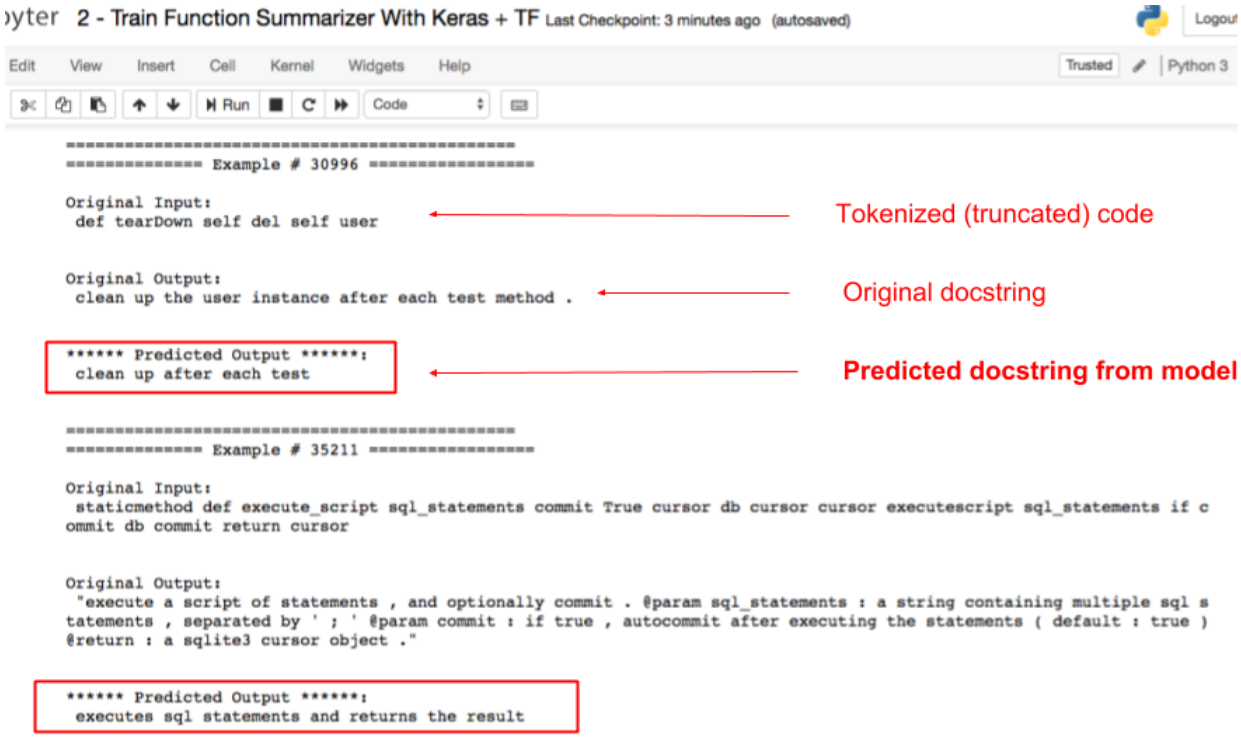
Part 2 notebook

Conceptually, building a sequence-to-sequence model to summarize code is identical to the GitHub issue summarizer we presented previously — instead of issue bodies we use python code, and instead of issue titles, we use docstrings.

However, unlike GitHub issue text, code is not natural language. To fully exploit the information within code, we could introduce domain-specific optimizations like tree-based LSTMs and syntax-aware tokenization. For this tutorial, we are going to keep things simple and treat code like natural language (and still get reasonable results).

Building a function summarizer is a very cool project on its own, but we aren't going to spend too much time focusing on this (but we encourage you to do so!). The entire end-to-end training procedure for this model is described in this notebook. We do not discuss the pre-processing or architecture for this model as it is identical to the issue summarizer.

Our motivation for training this model is not to use it for the task of summarizing code, but rather as a general purpose feature extractor for code. Technically speaking, this step is optional as we are only going through these steps to initialize the model weights for a related downstream task. In a later step, we will extract the encoder from this model and fine tune it for another task. Below is a screenshot of some example outputs of this model:



Sample results from function summarizer on a test set. See notebook here.

We can see that while the results aren't perfect, there is strong evidence that the model has learned to extract some semantic meaning from code, which is our main goal for this task. We can evaluate these models quantitatively using the BLEU metric, which is also discussed in this notebook.

It should be noted that training a seq2seq model to summarize code is not the only technique you can use to build a feature extractor for code. For example, you could also train a GAN and use the discriminator as a feature extractor. However, these other approaches are outside the scope of this tutorial.

Part 3 — Train a Language Model To Encode Natural Language Phrases

Part 3 notebook

Now that we have built a mechanism for representing code as a vector, we need a similar mechanism for encoding natural language phrases like those found in docstrings and search queries.

There are a plethora of general purpose pre-trained models that will generate high-quality embeddings of phrases (also called sentence embeddings). This article provides a great overview of the landscape. For example, Google's universal sentence encoder works very well for many use cases and is available on Tensorflow Hub.

Despite the convenience of these pre-trained models, it can be advantageous to train a model that captures the domain-specific vocabulary and semantics of docstrings. There are many techniques one can use to create sentence embeddings. These range from simple approaches, like averaging word vectors to more sophisticated techniques like those used in the construction of the universal sentence encoder.

For this tutorial, we will leverage a neural language model using an AWD LSTM to generate embeddings for sentences. I know that might sound intimidating, but the wonderful fast.ai library provides abstractions that allow you to leverage this technology without worrying about too many details. Below is a snippet of code that we use to build this model. For more context on how this code works, see this notebook.

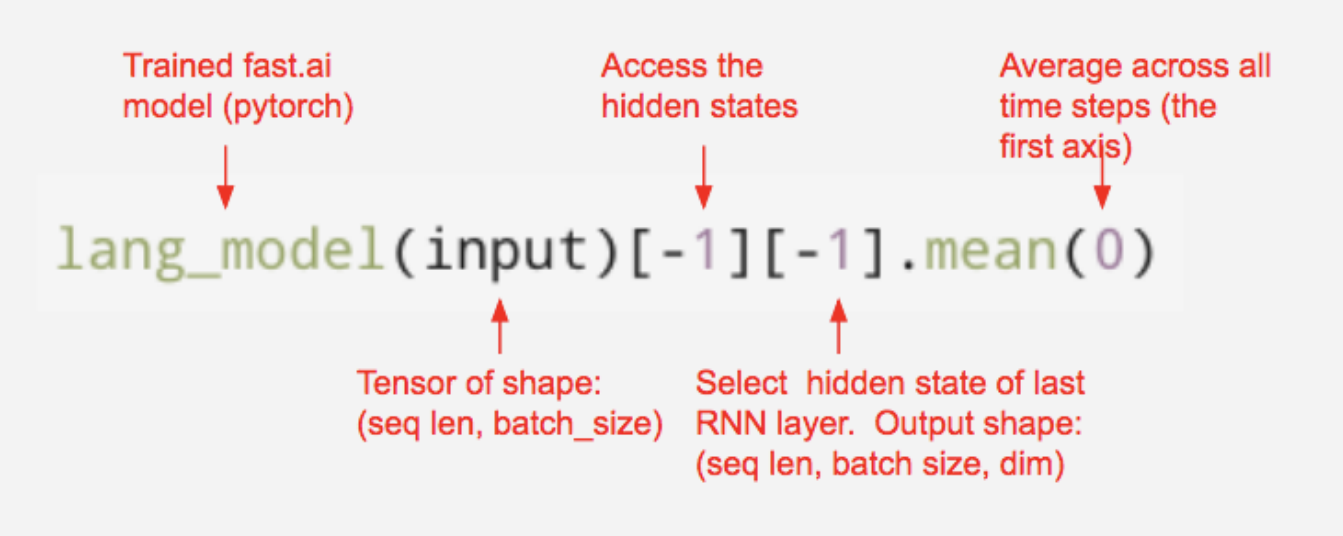
```
# create data loaders
trn_dl = LanguageModelLoader(trn_indexed, bs, bptt)
val_dl = LanguageModelLoader(val_indexed, bs, bptt)

# create lang model data
md = LanguageModelData(mpath, 1, vocab_size, trn_dl, val_dl, bs=bs, bptt=bptt)

# build learner. some hyper-params borrowed from fast.ai examples
opt_fn = partial(optim.Adam, betas=(0.8, 0.99))
drops = np.array([0.25, 0.1, 0.2, 0.02, 0.15]) * 0.7

learner = md.get_model(opt_fn, em_sz, nh, nl,
                       dropouti=drops[0],
                       dropout=drops[1],
                       wdrop=drops[2],
                       dropoute=drops[3],
                       dropouth=drops[4])

# borrowed these parameters from fastai
learner.fit(lr,
            n_cycle=n_cycle,
            wds=wd,
            cycle_len=cycle_len,
            use_clr=(32, 10),
            cycle_mult=cycle_mult,
            best_save_name='langmodel_best')
```

language_model.py hosted with ❤️ by GitHub view raw

Part of the train_lang_model function called in this notebook. Uses fast.ai.

It is important to carefully consider the corpus you use for training when building a language model. Ideally, you want to use a corpus that is of a similar domain to your downstream problem so you can adequately capture the relevant semantics and vocabulary. For example, a great corpus for this problem would be stack overflow data, since that is a forum that contains an extremely rich discussion of code. However, in order to keep this tutorial simple, we re-use the set of docstrings as our corpus. This is sub-optimal as discussions on stack overflow often contain richer semantic information than what is in a one-line docstring. We leave it as an exercise for the reader to examine the impact on the final outcome by using an alternate corpus.

After we train the language model, our next task is to use this model to generate an embedding for each sentence. A common way of doing this is to summarize the hidden states of the language model, such as the concat pooling approach found in this paper. However, to keep things simple we will simply average across all of the hidden states. We can extract the average across hidden states from a fast.ai language model with this line of code:



How to extract a sentence embedding from a fast.ai language model. This pattern is used here.

A good way to evaluate sentence embeddings is to measure the efficacy of these embeddings on downstream tasks like sentiment analysis, textual similarity etc. You can often use general-purpose benchmarks such as the examples outlined here to measure the quality of your embeddings. However, these generalized benchmarks may not be appropriate for this problem since our data is very domain specific. Unfortunately, we have not designed a set of downstream tasks for this domain that we can open source yet. In the absence of such downstream tasks, we can at least sanity check that these embeddings contain semantic information by examining the similarity

between phrases we know should be similar. The below screenshot illustrates some examples where we search the vectorized docstrings for similarity against user-supplied phrases (taken from this notebook):

```
In [86]: se.search('read csv into pandas dataframe')

cosine dist:0.0977
---------------
 load csv or json into pandas dataframe

cosine dist:0.1167
---------------
 reads swn database into a dictionary

cosine dist:0.1187
---------------
 load csv files into a raw object .

In [87]: se.search('train a random forest')

cosine dist:0.0561
---------------
 train a classifier

cosine dist:0.0607
---------------
 train a network

cosine dist:0.0800
---------------
 train a selection class
```

Manual inspection of text similarity as a sanity check. More examples in this notebook.

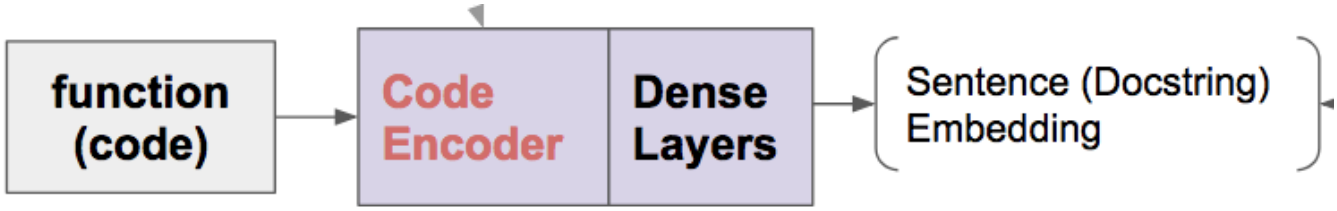
It should be noted that this is only a sanity check — a more rigorous approach is to measure the impact of these embeddings on a variety of downstream tasks and use that to form a more objective opinion about the quality of your embeddings. More discussion about this topic can be found in this notebook.

Part 4 — Train Model To Map Code Vectors Into The Same Vector Space As Natural Language

Part 4 notebook

At this point, it might be useful to revisit the diagram introduced at the beginning of this tutorial to review where you are. In that diagram you will find this representation

of part 4:



A visual representation of the tasks we will perform in Part 4

Most of the pieces for this step come from prior steps in this tutorial. In this step, we will fine-tune the seq2seq model from part 2 to predict docstring embeddings instead of docstrings. Below is code that we use to extract the encoder from the seq2seq model and add dense layers for fine-tuning:

```
# extract encoder
encoder_model = extract_encoder_model(seq2seq_Model)

# Freeze Encoder Model
for l in encoder_model.layers:
    l.trainable = False

#### Build Model Architecture For Fine-Tuning ####
encoder_inputs = Input(shape=(doc_length,), name='Encoder-Input')
enc_out = encoder_model(encoder_inputs)

# first dense layer with batch norm
x = Dense(500, activation='relu')(enc_out)
x = BatchNormalization(name='bn-1')(x)
# dense layer for output
out = Dense(500)(x)

# keras model object
code2emb_model = Model([encoder_inputs], out)
```

code2emb.py hosted with ❤ by GitHub view raw

Build a model that maps code to natural language vector space. For more context, see this notebook.

After we train the frozen version of this model, we unfreeze all layers and train the model for several epochs. This helps fine-tune the model a little more towards this task. You can see the full training procedure in this notebook.

Finally, we want to vectorize the code so we can build a search index. For evaluation purposes, we will also vectorize code that does not contain a docstring in order to see

how well this procedure generalizes to data we have not seen yet. Below is a code snippet (taken from this notebook) that accomplishes this task. Note that we use the ktext library to apply the same pre-processing steps that we learned on the training set to this data.

```
# transform raw docstring input
encinp = enc_pp.transform_parallel(no_docstring_funcs)

# vectorize code using the code2emb model
nodoc_vecs = code2emb_model.predict(encinp, batch_size=20000)
```

vectorize_code.py hosted with ❤ by GitHub view raw

Map code to the vector space of natural language with the code2emb model. For more context, see this notebook.

After collecting the vectorized code, we are ready to proceed to the last and final step!

Part 5 — Create A Semantic Search Tool

Part 5 notebook

In this step, we will build a search index using the artifacts we created in prior steps, which is illustrated below:

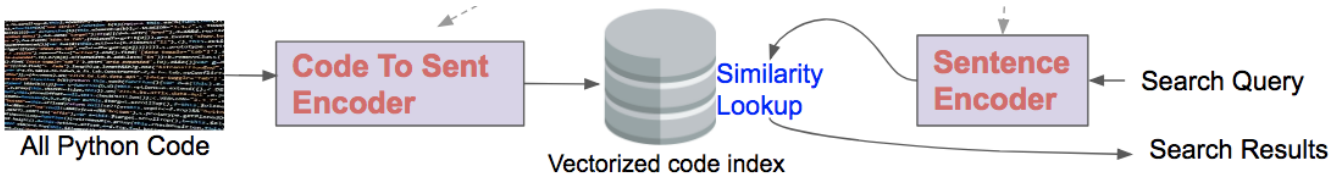


Diagram of Part 5 (extracted from the main diagram presented at the beginning)

In part 4, we vectorized all the code that did not contain any docstrings. The next step is to place these vectors into a search index where nearest neighbors can be quickly retrieved. A good python library for fast nearest neighbors lookups is nmslib. To enjoy fast lookups using nmslib you must precompute the search index like so:

```
search_index = nmslib.init(method='hnsw', space='cosinesimil')
search_index.addDataPointBatch(numpy_vectors)
search_index.createIndex({'post': 2}, print_progress=True)
```

build_search_index.py hosted with ❤ by GitHub view raw

How to create a search index with nmslib.

Now that you have built your search index of code vectors, you need a way to turn a string (query) into a vector. To do this you will use the language model from Part 3. To make this process easy we provided a helper class in lang_model_utils.py called Query2Emb, which is demonstrated in this notebook.

Finally, once we are able to turn strings into query vectors, we can retrieve the nearest neighbors for that vector like so:

```
idxs, dists = self.search_index.knnQuery(query_vector, k=k)
```

The search index will return two items (1) a list of indexes which are integer positions of the nearest neighbors in the dataset (2) distances of these neighbors from your query vector (in this case we defined our index to use cosine distance). Once you have this information, it is straightforward to build semantic search. An example of how you can do this is outlined in the below code:

```
class search_engine:
    """Organizes all the necessary elements we need to make a semantic search tool."""
    def __init__(self,
                 nmslib_index,
                 ref_df,
                 query2emb_func):
        """
        Parameters
        ==========
        nmslib_index : nmslib object
            This is a pre-computed search index.
        ref_df : pandas.DataFrame
            This dataframe contains meta-data for search results.
            must contain the columns 'code' and 'url'
        query2emb_func : callable
            A function that takes as input a string and returns a vector
            that is in the same vector space as what is loaded into the
            search index.

        """
        assert 'url' in ref_df.columns
        assert 'code' in ref_df.columns

        self.search_index = nmslib_index
        self.ref_df = ref_df
        self.query2emb_func = query2emb_func
```

```
27
28      def search(self, str_search, k=2):
29          """
30          Prints the code that are the nearest neighbors (by cosine distance)
31          to the search query.
32
33          Parameters
34          ==========
35          str_search : str
36              a search query.  Ex: "read data into pandas dataframe"
37          k : int
38              the number of nearest neighbors to return.  Defaults to 2.
39
40          """
41          query = self.query2emb_func(str_search)
42          idxs, dists = self.search_index.knnQuery(query, k=k)
43
44          for idx, dist in zip(idxs, dists):
45              code = self.ref_df.iloc[idx].code
46              url = self.ref_df.iloc[idx].url
47              print(f'cosine dist:{dist:.4f}  url: {url}\n---------------\n')
48              print(code)
```

semantic_search_cls.py hosted with ❤️ by GitHub view raw

A class that glues together all the parts we need to build semantic search.

Finally, this notebook shows you how to use the search_engine object above to create an interactive demo that looks like this:

Live Semantic Search of Code (Searching Holdout Set Only)

```
%%search
start f
```

This is the same gif that was presented at the beginning of this tutorial.

Congratulations! You have just learned how to create semantic search. I hope it was worth the journey.

Wait, But You Said Search of Arbitrary Things?

Even though this tutorial describes how to create semantic search for code, you can use similar techniques to search video, audio, and other objects. Instead of using a model that extracts features from code (part 2), you need to train or find a pre-trained model that extracts features from your object of choice. The only prerequisite is that you need a sufficiently large dataset with natural language annotations (such as transcripts for audio, or captions for photos).

We believe you can use the ideas you learned in this tutorial to create your own search and would love to hear from you to see what you create (see the *getting in touch* section below).

Limitations and Omissions

- The techniques discussed in this blog post are simplified and are only scratching the surface of what is possible. What we have presented is a very simple semantic search — however, in order for such a search to be effective, you may have to augment this search with keyword search and additional filters or rules (for example the ability to search a specific repo, user, or organization and other mechanisms to inform relevance).

- There is an opportunity to use domain-specific architectures that take advantage of the structure of code such as tree-lstms. Furthermore, there are other standard tricks like utilizing attention and random teacher forcing that we omitted for simplicity.

- One part we glossed over is how to evaluate search. This is a complicated subject which deserves its own blog post. In order to iterate on this problem effectively, you need an objective way to measure the quality of your search results. This will be the subject of a future blog post.

Get In Touch!

We hope you enjoyed this blog post. Please feel free to get in touch with us:

- Hamel Husain: Twitter, LinkedIn, or GitHub.
- Ho-Hsiang Wu: LinkedIn, GitHub

Resources

- The GitHub repo for this article.
- To make it easier for those trying to reproduce this example, we have packaged all of the dependencies into a Nvidia-Docker container. For those that are not familiar with Docker you might find this post to be helpful. Here is a link to the docker image for this tutorial on Dockerhub.
- My top recommendation for anyone trying to acquire deep learning skills is to take Fast.AI by Jeremy Howard. I learned many of the skills I needed for this blog post there. Additionally, this tutorial leverages the fastai library.
- {Update 9/18/2018}: There is an official GitHub demo of this hosted here: https://experiments.github.com/
- Keep an eye on this book, it is still in early release but goes into useful details on this subject.
- This talk by Avneesh Saluja highlights how Airbnb is researching the use of shared vector spaces to power semantic search of listings as well as other data products.

Thanks

Mockup search UI was designed by Justin Palmer (you can see some of his other work here). Also thanks to the following people for their review and input: Ike Okonkwo, David Shinn, Kam Leung.

Disclaimer

Any ideas or opinions presented in this article are our own. Any ideas or techniques presented do not necessarily foreshadow future products of GitHub. The purpose of this blog is for educational purposes only.

Machine Learning Data Science Deep Learning NLP Towards Data Science