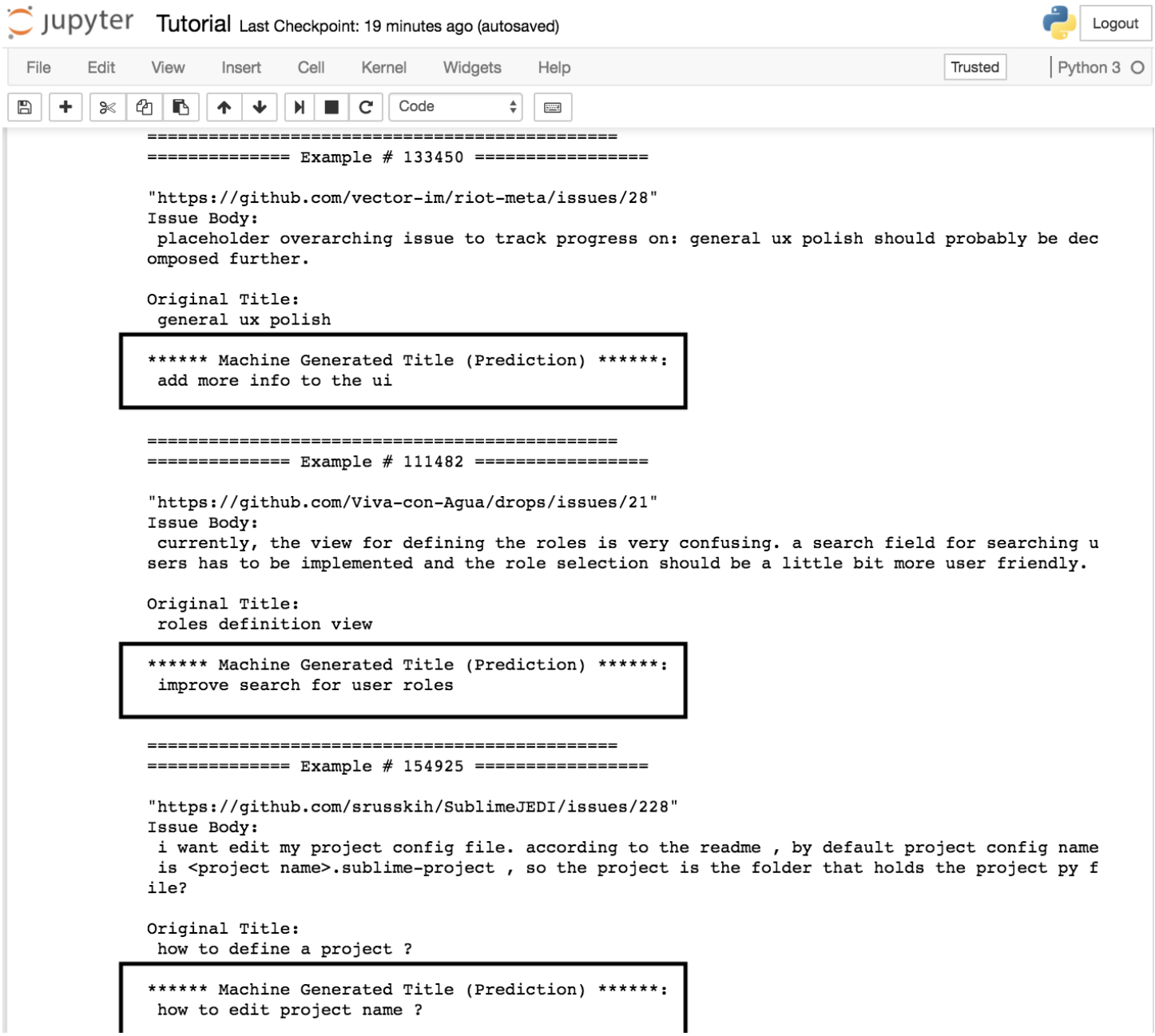
How To Create Data Products That Are Magical Using Sequence-to-Sequence Models

Hamel Husain Follow
Jan 18, 2018 · 17 min read

A tutorial on how to summarize text and generate features from Github Issues using deep learning with Keras and TensorFlow.

By Hamel Husain

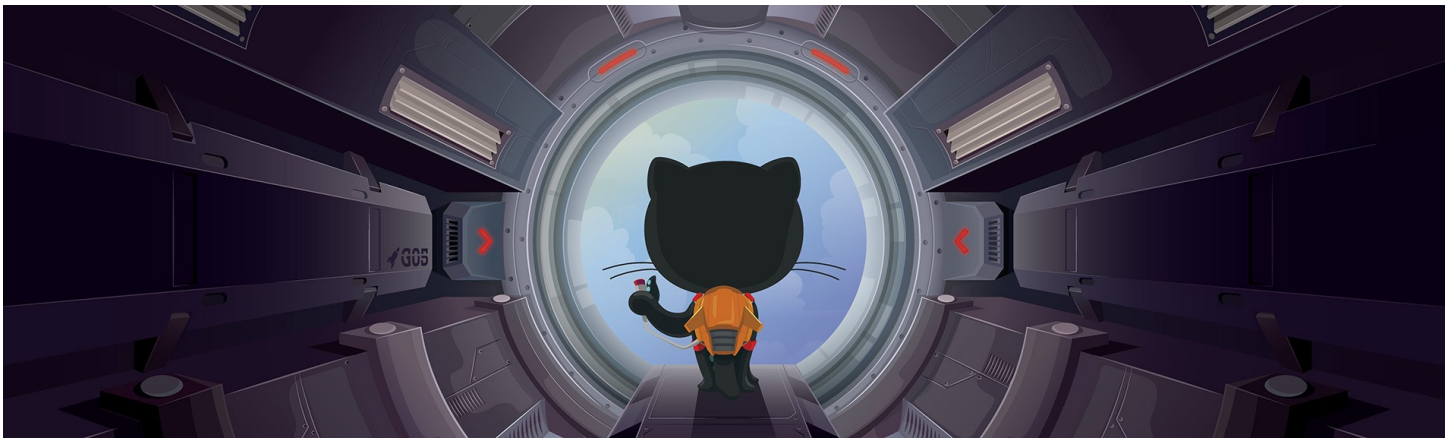
Teaser: Training a model to summarize Github Issues



Predictions are in rectangular boxes.

The above results are randomly selected elements of a holdout set.
Keep reading below, there will be a link to many more examples!

. . .



Github's Octocat

Motivation:

I never imagined I would ever use the word "magical" to describe the output of a machine learning technique. This changed when I was introduced to deep learning, where you can accomplish things like identify objects in pictures or sort two tons of legos. What is more amazing is **you do not need a PhD or years of training** to unleash the power of these techniques on your data. You just need to be comfortable with writing code, high school level math, and patience.

However, there is a dearth of reproducible examples of how deep learning techniques are being used in industry. **Today, I'm going to share with you a reproducible, minimally viable product that illustrates how to to utilize deep learning to create data products from text (Github Issues).**

This tutorial will focus on using sequence to sequence models to summarize text found in Github issues, and will demonstrate the following:

- You don't need to have tons of computing power to achieve sensible results (I am going to use a single GPU).
- You don't need to write lots of code. It's surprising how so few lines of code can produce something so magical.
- Even if you do not want to summarize text, training a model to accomplish this task is a useful for generating features for other tasks.

What I'm going to cover in this post:

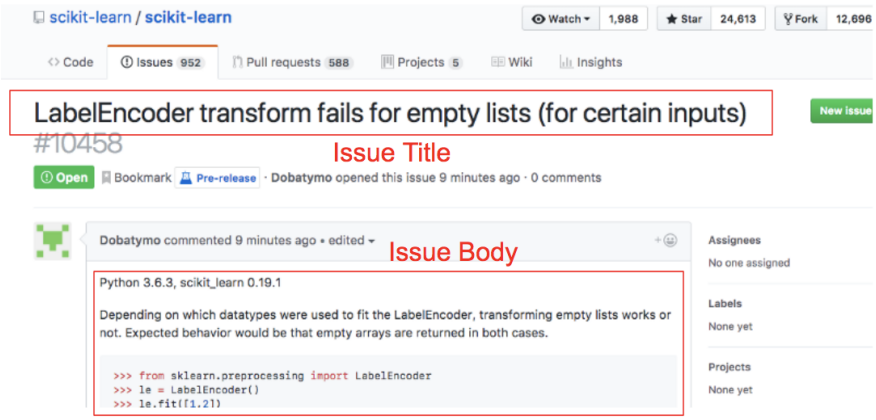
- How to gather the data and prepare it for deep learning.
- How to construct the architecture of a seq2seq model and train the model.
- How to prepare the model for inference, and a discussion and demonstration of various use cases.

My goal is to focus on providing you with an end-to-end example so that you can develop a conceptual model of the workflow, rather than diving very deep into the math. I will provide links along the way to allow you to dig deeper if you so desire.

. . .

Get The Data

If you are not familiar with Github Issues, I highly encourage you to go look at a few before diving in. Specifically, the pieces of data we will be using for this exercise are Github Issue **bodies** and **titles**. An example is below:



https://github.com/scikit-learn/scikit-learn/issues/10458

We will gather many (Issue Title, Issue Body) pairs with the goal of training our model to summarize issues. The idea is that by seeing many examples of issue descriptions and titles a model can learn how to summarize new issues.

The best way to acquire Github data if you do not work at Github is to utilize this wonderful open source project, which is described as:

> *…. a project to **record** the public GitHub timeline, **archive it**, and **make it easily accessible** for further analysis.*

Instructions on querying data from this project is available in the **appendix** of this article. An astute reader of this blog (David Shinn) has gone through all of the steps outlined in the appendix and has hosted the data required for this exercise as a csv file on Kaggle!

You can download the data from this page by clicking on the download link.

. . .

Prepare & Clean The Data

Sometimes, cleaning data is hard work. Image credit.

Keras Text Pre-Processing Primer

Now that we have gathered the data, we need to prepare the data for the modeling. Before jumping into the code, let's warm up with a toy example of two documents:

```
["The quick brown fox jumped over the lazy dog 42 times.",
"The dog is lazy"]
```

Below is a rough outline of the steps I will take in order to pre-processes this raw text:

1. Clean text: in this step, we want to remove or replace specific characters and lower case all the text. This step is discretionary and depends on the size of the data and the specifics of your domain. In this toy example, I lower-case all characters and replace numbers with *number* in the text. In the real data, I handle more scenarios.

```
["the quick brown fox jumped over the lazy dog *number*
times", "the dog is lazy"]
```

3. Tokenize: split each document into a list of words

```
[['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the',
'lazy', 'dog', '*number*', 'times'], ['the', 'dog', 'is',
'lazy']]
```

4. Build vocabulary: You will need to represent each distinct word in your corpus as an integer, which means you will need to build a map of token -> integers. Furthermore, I find it useful to reserve an integer for rare words that occur below a certain threshold as well as 0 for padding (see next step). After you apply a token -> integer mapping, your data might look like this:

```
[[2, 3, 4, 5, 6, 7, 2, 8, 9, 10, 11], [2, 9, 12, 8]]
```

5. Padding: You will have documents that have different lengths. There are many strategies on how to deal with this for deep learning, however for this tutorial I will pad and truncate documents such that they are all transformed to the same length for simplicity. You can decide to pad (with zeros) and truncate your document at the beginning or end, which I will refer to as "pre" and "post" respectively. After pre-padding our toy example, the data might look like this:

```
[[2, 3, 4, 5, 6, 7, 2, 8, 9, 10, 11], [0, 0, 0, 0, 0, 0, 0,
2, 9, 12, 8]]
```

A reasonable way to decide your target document length is to build a histogram of document lengths and choose a sensible number. (Note that the above example has padded the data in front but we could also pad at the end. We will discuss this more in the next section).

Preparing Github Issues Data

For this section, you will want to follow along in this notebook. The data we are working with looks like this:

	issue_url	issue_title	body
222500	"https://github.com/zenoss/ZenPacks.zenoss.Microsoft.Windows/issues/845"	"zwinrmkrb5includedir is not honored"	"i am seeing a persistent issue where setting zwinrmkrb5includedir does not result in correct behavior. if i manually create a krb5.conf file with rdns set to flase, i can successfully model a device, but on subsequent runs modeling fails and the setting disappears. is there a secret to preventing reverse dns lookups?"
264960	"https://github.com/uf-feedback/vertpaleo/issues/38"	"portal usage statistics are almost back"	"thanks to the financial support of the museum of vertebrate zoology at berkeley, we have fixed the issues that were preventing us from logging the vertnet statistics of data use. usage statistics are being collected once again. we are now working on the reporting and visualization of those stats, so that we can bring those back to the natural history collections community in a friendly, useful modality. we expect all of this to be up and running before the end of the year. we apologize for ...
448395	"https://github.com/airesvsg/wp-rest-api-cache/issues/10"	"uncaught exception: serialization of 'closure' is not allowed"	"i'm getting this kind of error just by installing & enabling the plugin: 21-mar-2017 08:46:03 utc php fatal error: uncaught exception: serialization of 'closure' is not allowed in .../wordpress/wp-includes/functions.php:435 stack trace: 0 ... /wordpress/wp-includes/functions.php 435 : serialize object wp_rest_response 1 ... /wordpress/wp-includes/option.php 427 : maybe_serialize object wp_rest_response 2 ... /wordpress/wp-includes/option.php 730 : add_option '_transient_rest...', object wp_...

Pandas dataframe with issue bodies and titles, from this notebook.

We can see there are issue titles and bodies, which we will process separately. I will not be using the URLs for modeling but only as a reference. Note that I have sampled 2M issues from the original 5M in order to make this tutorial tractable for others.

Personally, I find pre-processing text data for deep learning to be extremely repetitive. Keras has good utilities that allow you to do this, however I wanted to parallelize these tasks to increase speed.

The ktext package

I have built a utility called ktext that helps accomplish the pre-processing steps outlined in the previous section. This library is a thin wrapper around keras and spacy text processing utilities, and leverages python process-based-threading to speed things up. It also chains all of the pre-processing steps together and provides a bunch of convenience functions. *Warning: this package is under development so use with caution outside this tutorial (pull requests are welcome!)*. To learn more about how this library works, look at this tutorial (but for now I suggest reading ahead).

To process the **body** data, we will execute this code:

```
from ktext.preprocess import processor
# instantiate data processing object
body_pp = processor(keep_n=8000, padding_maxlen=70)
# process data
```

See full code on in this notebook.

The above code cleans, tokenizes, and applies pre-padding and post-truncating such that each document length is 70 words long. I made decisions about padding length by studying histograms of document length provided by ktext. Furthermore, only the top 8,000 words in the vocabulary are retained and remaining words are set to the index 1 which correspond to rare words (this was an arbitrary choice). It takes one hour for this to run on an AWS p3.2xlarge instance that has 8 cores and 60GB of memory. Below is an example of raw data vs. processed data:

```
12]: print('\noriginal string:\n', train_body_raw[0], '\n')
     print('after pre-processing:\n', train_body_vecs[0], '\n')

original string:
 some of the sds alerts do not have clearing alerts. so it always present in alerting directory. t
hese kinds of alerts should be stored in etcd under /alerting/notify, it never goes to alerting/al
erts directory and it is not displayed under alerts in ui also. these kinds of alerts are notified
via notification channel and deleted via ttl. node_agent should have a logic to handle this in ale
rting framework.

after pre-processing:
 [37 33 39  1  6 17 29 22 13  6  3 36 25  8 34 23  1 15  3 40 26 33  6 35
 11 38 23 18 45  1  4 32  2 25 28 20 42  1  4  6 15  9 25 24 29 16 45  6
 23 44  7  3 40 26 33  6 10 31 46 30 12  9 14 46 43  3  1 35 22  5]
```

Image from this notebook.

The **titles** will be processed almost the same way, but with some subtle differences:

```
# instantiate the pre-processor for titles
title_pp = processor(append_indicators=True, keep_n=450
                     padding_maxlen=12, padding ='post'
# process the titles
```

See full code in this notebook.

This time, we are passing some additional parameters:

- append_indicators=True will append the tokens '_start_' and '_end_' to the start and end of each document, respectively.
- padding='post' means that zero padding will be added to the end of the document instead of default of 'pre'.

The reason for processing the titles in this way is that we want our model to know when the first letter of the title is supposed to occur, and also learn to predict when the end of a phrase should be. This will make more sense in the next section where model architecture is discussed.

. . .

Define The Model Architecture

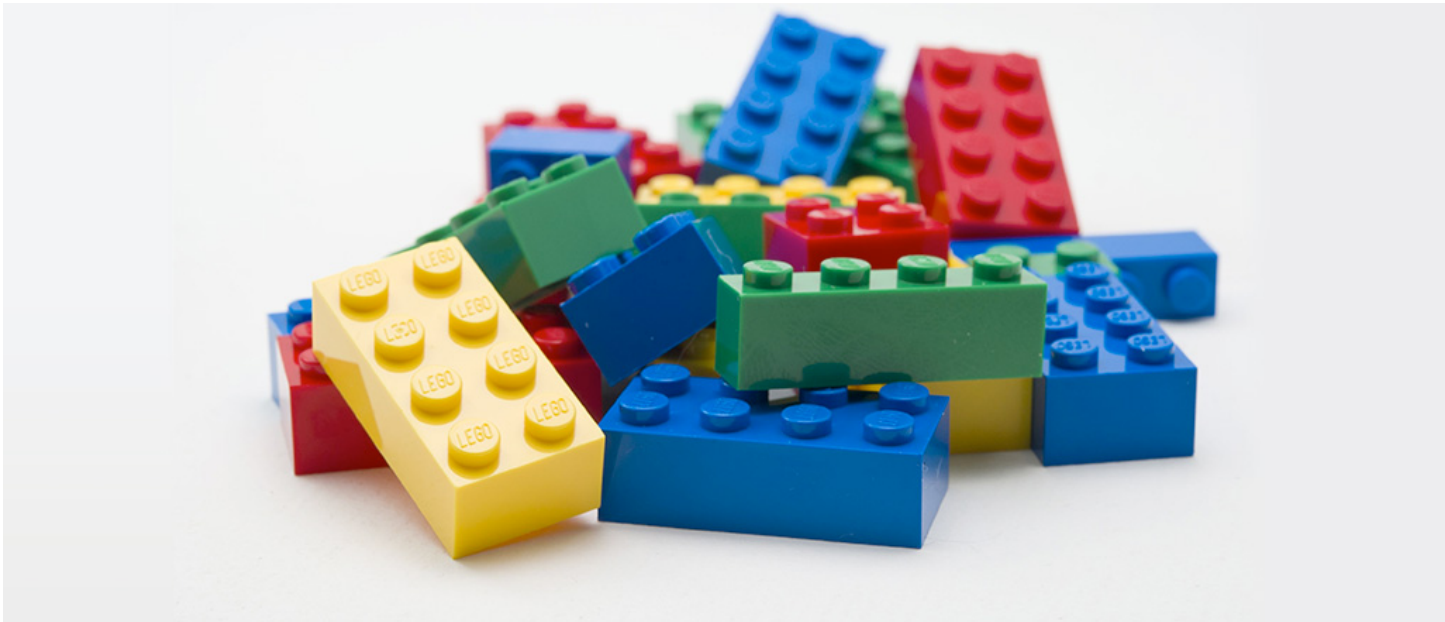


Image credit: https://goo.gl/images/IrVBHB

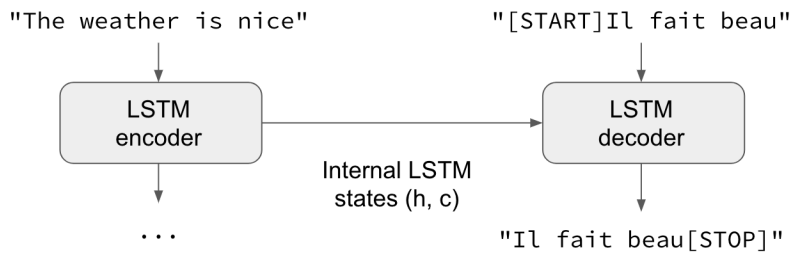
Building a neural network architecture is like stacking lego bricks. For beginners, it can be useful to think of each layer as an API: you send the API some data and then the API returns some data. Thinking of things this way frees you from becoming overwhelmed, and you can build your understanding of things slowly. It is important to understand two concepts:

- the shape of data that each layer expects, and the shape of data the layer will return. (When you stack many layers on top of each other, the input and output shapes must be compatible, like legos).
- conceptually, what will the output(s) of a layer represent? What does the output of a subset of stacked layers represent?

The above two concepts are essential to understanding this tutorial. If you don't feel comfortable with this as you are reading below, I highly recommend watching as many lessons as you need from this MOOC and returning here.

In this tutorial, we will leverage an architecture called a **Sequence to Sequence networks**. Pause reading this blog and carefully read A ten-minute introduction to sequence-to-sequence learning in Keras by Francois Chollet.

Once you finish reading that article, you should conceptually understand the below diagram, which illustrates a network that will take two inputs and have one output:



Credit: https://blog.keras.io/category/tutorials.html

The network we will use for this problem will look very similar to the one in the tutorial described above, and is defined with this code:

```
from keras.models import Model
from keras.layers import Input, LSTM, GRU, Dense, Embe
from keras import optimizers

#arbitrarly set latent dimension for embedding and hid
latent_dim = 300

##### Define Model Architecture ######

########################
#### Encoder Model ####
encoder_inputs = Input(shape=(doc_length,), name='Enco

# Word embeding for encoder (ex: Issue Body)
x = Embedding(num_encoder_tokens,
              latent_dim,
              name='Body-Word-Embedding',
              mask_zero=False)(encoder_inputs)

x = BatchNormalization(name='Encoder-Batchnorm-1')(x)

# We do not need the `encoder_output` just the hidden
_, state_h = GRU(latent_dim, return_state=True, name='

# Encapsulate the encoder as a separate entity so we c
#  encode without decoding if we want to.
encoder_model = Model(inputs=encoder_inputs,
                      outputs=state_h,
                      name='Encoder-Model')

seq2seq_encoder_out = encoder_model(encoder_inputs)

########################
#### Decoder Model ####
decoder_inputs = Input(shape=(None,), name='Decoder-In

# Word Embedding For Decoder (ex: Issue Titles)
dec_emb = Embedding(num_decoder_tokens,
                    latent_dim,
                    name='Decoder-Word-Embedding',
                    mask_zero=False)(decoder_inputs)

dec_bn = BatchNormalization(name='Decoder-Batchnorm-1'

```

For more context, see this notebook.

When you read the above code, you will notice references to the concept of teacher forcing. Teacher forcing is an extremely important mechanism that allows this network to train faster. This is explained better in this post.

Credit: xkcd

You may be wondering—where I came up with the above architecture. I started with publicly available examples and performed lots of experiments. This xkcd comic really describes it best. You will notice that my loss function is *sparse categorical crossentropy* instead of *categorical crossentropy* because this allows me to pass integers as my targets to predict instead of one-hot-encoding my targets, which is more memory efficient.

. . .

Train The Model

We are going to turn the crank of SGD to train our model. Image Credit: https://goo.gl/images/MYrQHk

The code for training the model is fairly straightforward and involves calling the fit method on the model object we defined. We pass additional parameters such as callbacks for logging, the number of epochs, and batch size.

Below is the code we call for training the model, as well as a markdown file that shows the output of running this code. For more context, follow along in this Jupyter notebook.

```
from keras.callbacks import CSVLogger, ModelCheckpoint

#setup callbacks for model logging
script_name_base = 'tutorial_seq2seq'
csv_logger = CSVLogger('{:}.log'.format(script_name_ba
model_checkpoint = ModelCheckpoint('{:}.epoch{{epoch:0
                                   save_best_only=True

# pass arguments to model.fit
batch_size = 1200
epochs = 7
history = seq2seq_Model.fit([encoder_input_data, decod
          batch_size=batch_size,
          epochs=epochs,
          validation_split=0.12, callbacks=[csv_logger
```

train_model.py hosted with ❤️ by GitHub view raw

```
Train on 1584000 samples, validate on 216000 samp
Epoch 1/7
1584000/1584000 [==============] - 411s 259us/st
loss: 2.6989 - val_loss: 2.3833
Epoch 2/7
1584000/1584000 [==============] - 265s 167us/st
loss: 2.2941 - val_loss: 2.3035
Epoch 3/7
```

I trained this model on an AWS p3.2xlarge instance which took approximately 35 minutes to train 7 epochs. In a production scenario, I would probably let such a model train for a longer period of time and leverage additional callbacks for early stopping or for adjusting the learning rate dynamically. However, I found that the training procedure outlined above is sufficient for a minimally viable product.

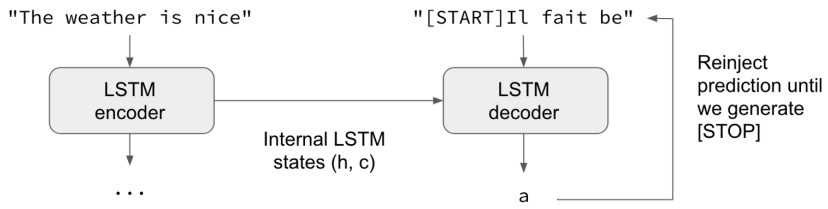
There are significant improvements that can be made by using more advanced learning rate schedules and architecture enhancements, which are discussed towards the end of this article in the “Next Steps” section.

. . .

Prepare The Model For Inference

Image Credit: https://goo.gl/images/8ifMZA

To prepare the model for inference (to make predictions), we have to re-assemble it (with its trained weights intact) such that the decoder uses the last prediction as input rather than being fed the right answer for the previous time step, as illustrated below:



From Keras tutorial on sequence to sequence learning.

If this doesn't make sense, please revisit this tutorial. The decoder is re-assembled using the following code (I've made very verbose comments in the code so you can follow along):

```
def extract_decoder_model(model):
    """
    Extract the decoder from the original model.

    Inputs:
    ------
    model: keras model object

    Returns:
    -------
    A Keras model object with the following inputs and

    Inputs of Keras Model That Is Returned:
    1: the embedding index for the last predicted word
    2: the last hidden state, or in the case of the fi
       from the encoder

    Outputs of Keras Model That Is Returned:
    1.  Prediction (class probabilities) for the next
    2.  The hidden state of the decoder, to be fed bac
        next time step

    Implementation Notes:
    ----------------------
    Must extract relevant layers and reconstruct part
    to allow for different inputs as we are not going
    inference time.

    """
    # the latent dimension is the same throughout the
    # cheat and grab the latent dimension of the embed
    # what is output from the decoder
    latent_dim = model.get_layer('Decoder-Word-Embeddi

    # Reconstruct the input into the decoder
    decoder_inputs = model.get_layer('Decoder-Input').
    dec_emb = model.get_layer('Decoder-Word-Embedding'
    dec_bn = model.get_layer('Decoder-Batchnorm-1')(de

    # Instead of setting the intial state from the enc
```

More helper functions used to make predictions are located in this file. Specifically, the method *generate_issue_title* defines the mechanics of predicting issue titles. I use a greedy next-most-probable word

approach for this tutorial. I would suggest reading that code carefully in order to fully understand how predictions are made.

. . .

A Demonstration Of What This Model Can Do

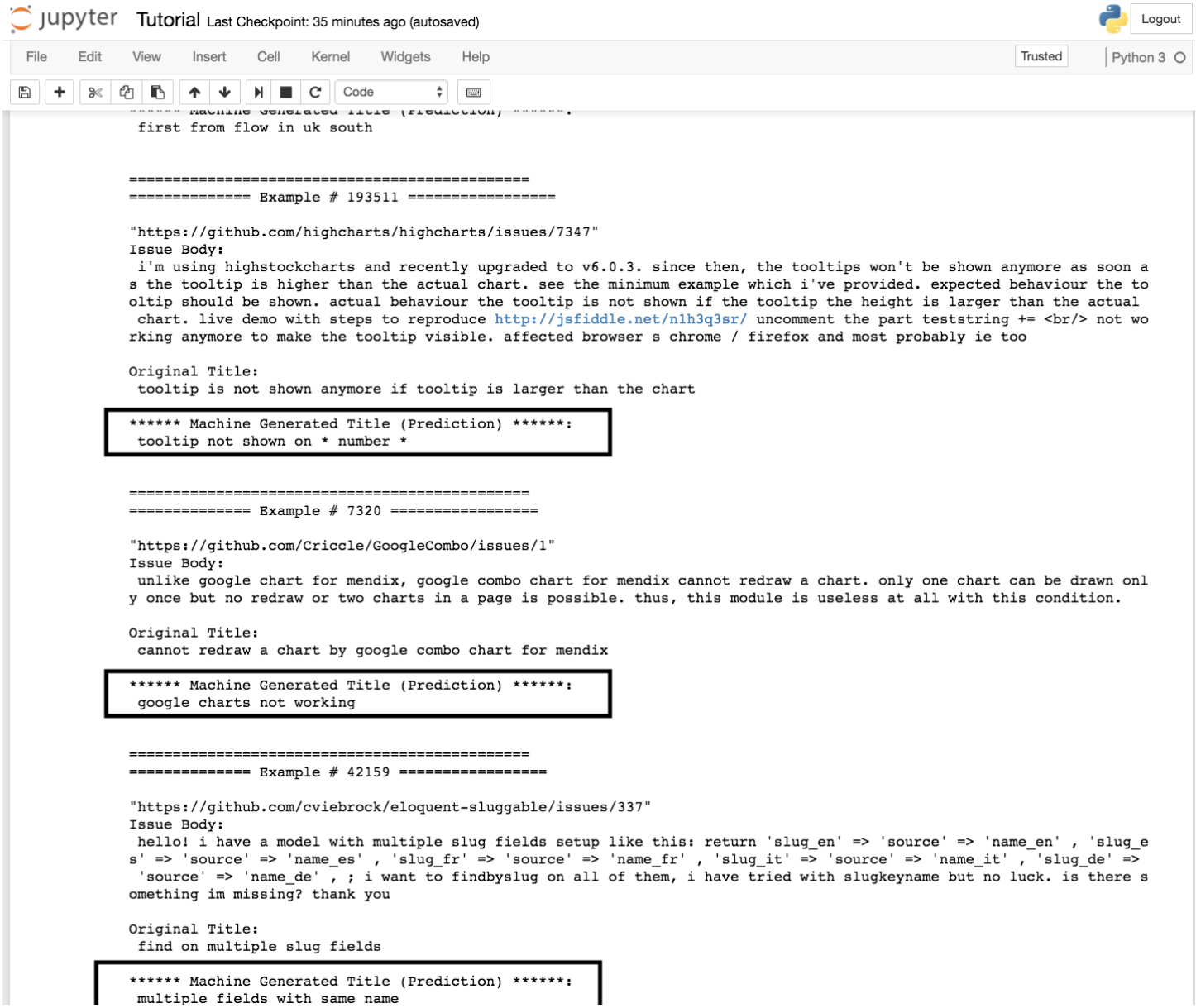
Image credit: https://goo.gl/images/bfPNhR

1. Summarize text, and generate a really good demo out of the box.

In typical classification and regression models, predictions themselves are not that interesting unless accompanied by a heavy dose of visualizations and storytelling. However, if you can train a model to summarize a piece of text in natural language, the predictions themselves are a good way of showing an audience that you have learned to extract meaningful features from the domain—and if the predictions are good, it seems ***magical***.

The ability to summarize text can be a useful data product in itself, for example to auto-suggest titles to users. However, this may not be the most useful part of this model. Other capabilities of this model are discussed in a following section.

Example of text summarization on a holdout set (more examples here):



Predictions are in rectangular boxes. Notebook available on Github here.

2. Extract features that can be used for a plethora of tasks.

Recall that the sequence to sequence model has two components: the encoder and the decoder. The ***encoder*** "encodes" information or

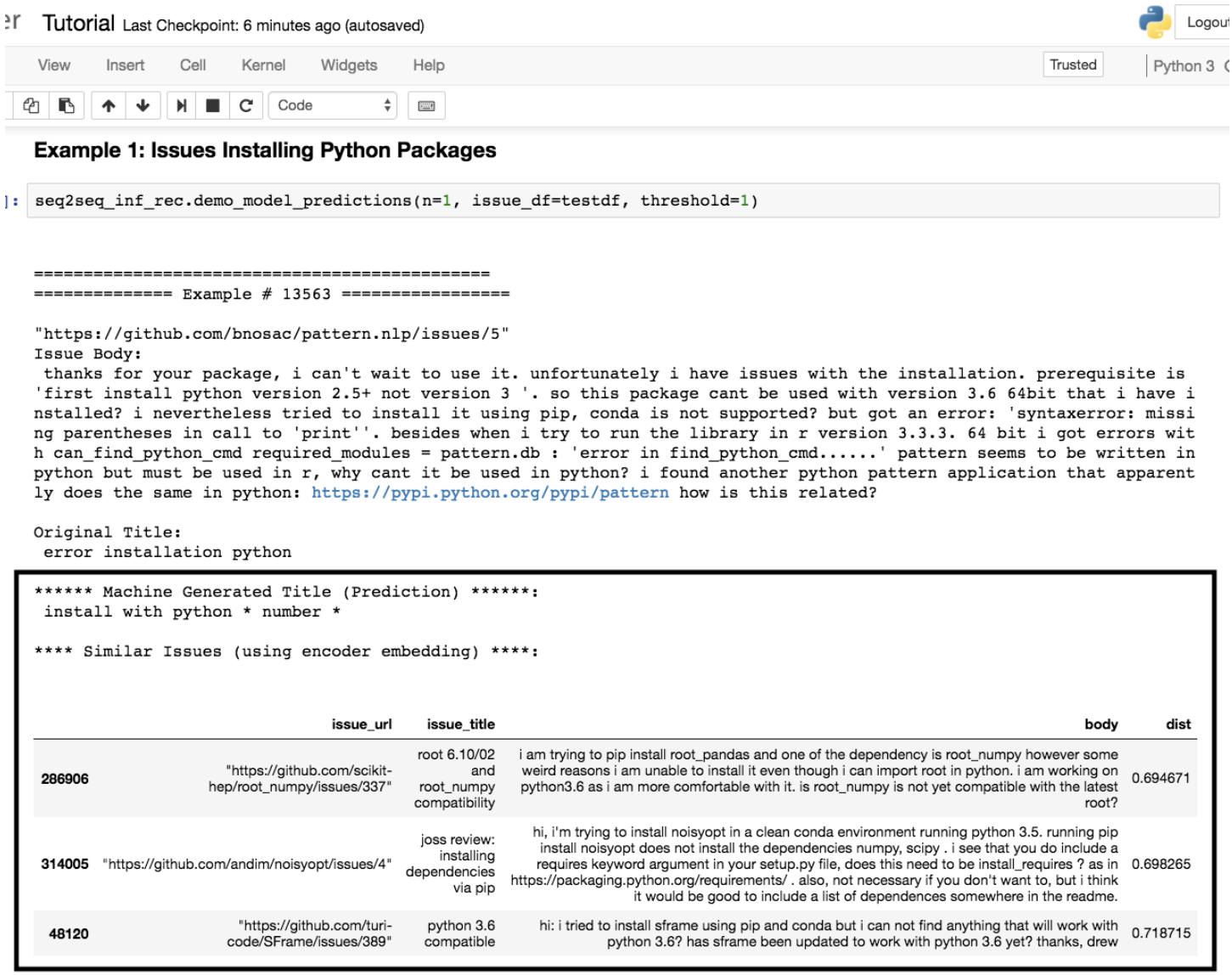
extracts features from the text and presents this information to the decoder, and the ***decoder*** takes that information and attempts to generate a coherent summary in natural language.

In this tutorial, the encoder produces a 300-dimensional vector for each issue. This vector can be used for a variety of machine learning tasks such as:

- Building a recommender system to find similar or duplicate issues.
- Detecting issues that are spam.
- Providing additional features to a regression model that will predict the amount of time an issue will remain open.
- Providing additional features to a classifier to identify which issues represent bugs or vulnerabilities.

It should be noted that there are many ways to extract features from a body of text, and there is no guarantee that these features will be superior for a specific task compared to another method. I have found that it is often useful to combine the features extracted from this approach with other features. However, the main point I want to highlight is **you get these features for free as a side effect of training the model to summarize text!**

Below is an example of this concept at work for recommending similar issues. Because the encoder provides a 300-dimensional vector that describes each issue, it is straightforward to find the nearest neighbors for each issue in vector space. Using the annoy package, **I display the closest neighbors in addition to generating an issue title for several issues**:



	issue_url	issue_title	body	dist
286906	"https://github.com/scikit-hep/root_numpy/issues/337"	root 6.10/02 and root_numpy compatibility	i am trying to pip install root_pandas and one of the dependency is root_numpy however some weird reasons i am unable to install it even though i can import root in python. i am working on python3.6 as i am more comfortable with it. is root_numpy is not yet compatible with the latest root?	0.694671
314005	"https://github.com/andim/noisyopt/issues/4"	joss review: installing dependencies via pip	hi, i'm trying to install noisyopt in a clean conda environment running python 3.5. running pip install noisyopt does not install the dependencies numpy, scipy . i see that you do include a requires keyword argument in your setup.py file, does this need to be install_requires ? as in https://packaging.python.org/requirements/ . also, not necessary if you don't want to, but i think it would be good to include a list of dependences somewhere in the readme.	0.698265
48120	"https://github.com/turi-code/SFrame/issues/389"	python 3.6 compatible	hi: i tried to install sframe using pip and conda but i can not find anything that will work with python 3.6? has sframe been updated to work with python 3.6 yet? thanks, drew	0.718715

Example 2: Issues asking for feature improvements

```
]: seq2seq_inf_rec.demo_model_predictions(n=1, issue_df=testdf, threshold=1)
```

```
================================================
============== Example # 157322 =================

"https://github.com/Chingu-cohorts/devgaido/issues/89"
Issue Body:
 right now, your profile link is https://devgaido.com/profile. this is fine, but it would be really cool if there was
 a way to share your profile with other people. on my portfolio, i have social media buttons to freecodecamp, github,
 ect. without a custom link, i cannot show-off what i have done on devgaido to future employers.

Original Title:
 feature request: sharable profile.

****** Machine Generated Title (Prediction) ******:
 add a link to your profile

**** Similar Issues (using encoder embedding) ****:
```

	issue_url	issue_title	body	dist
250423	"https://github.com/ParabolInc/action/issues/1379"	integrations list view discoverability	issue - enhancement i was initially confused by the link to my account copy; seeing github in the integrations list made me think it had already been set up . i realize now that i had to allow parabol to post as me. i think that link to my account could use a tooltip explaining what link means, and why you'd want to do so. <img width= 728 alt= screen shot 2017-09-29 at 10 52 05 am src= https://user-images.githubusercontent.com/2146312/31024786-2fd39c46-a50e-11e7-9f2a-6d4a5ed2baeb.png >	0.748828
222304	"https://github.com/viosey/hexo-theme-material/issues/166"	allow us to use sns-share for github	i'd love to be able to add a link at the bottom of the page for my github account. however, the sns-share option doesn't currently seem to be able to do this.	0.774398
153327	"https://github.com/tobykurien/GoogleApps/issues/31"	drive provide download ability	sometimes people share files via g drive. provided a link this app can show some info about the files but doesn't show the download button. i hope that it can be fixed and users would be able to download files with this app.	0.778953

Predictions are in rectangular boxes. Notebook available on Github here.

The above two examples illustrate how features extracted by the encoder can be used to find semantically similar issues. For example, you could use these features to inform a recommendation system or another machine learning task as outlined above.

What's even more exciting, is that this doesn't have to be limited to only issues . We can apply the same approach to generating repo titles from README files, or comments and docstrings from code. The possibilities are endless. With the database I introduce you to in the appendix, you can even get this data and try this yourself!

. . .

Model Evaluation

A good way to evaluate the performance of text-summarization models is to use the BLEU score. The code to generate the a BLEU score on this data can be found here. This recent blog post has great explanation of

this metric with some nice visuals. I will leave this as an exercise for the reader.

While I cannot share my best model's BLEU score, I can tell you that there is significant room for improvement on the model I shared in this post. I offer some hints as *next step*s below.

Next Steps

The goal of this blog post was to demonstrate how a Seq2Seq model can be used to produce interesting data products. The model I am actively experimenting with has a different architecture, but the underlying idea is the same. Some useful enhancements that are not presented in this blog post:

- Adding attention layers as well as bi-directional RNNs.
- Stacking more recurrent layers in the encoder and decoder, and tuning the size of various layers.
- Using regularization (ex: Dropout).
- Pre-training word embeddings on the entire corpus of issues.
- A better tokenizer that can handle mixed code and text, as well as better handling of issue templates and other markdown artifacts.
- Training on more data (we only trained the example model on 2M issues in this tutorial, but more data is available).
- For predictions of issue titles, use beam search instead of a greedy next best word approach.
- Exploring the wonderful fastai library built upon PyTorch which has several state of the art techniques included for NLP. This involves switching from Keras to PyTorch.

Some of the above items are more advanced topics, but can easily be learned. For those readers that want to learn more, see the resources section below.

Replicating My Environment : Nvidia-Docker

To make it easier for those trying to run my code, I have packaged all of the dependencies into an Nvidia-Docker container. For those that are not familiar with Docker you might find my post on this subject to be

helpful. Here is a link to the docker image for this tutorial on Dockerhub.

Resources

- This tutorial provides another very cool application of sequence to sequence models!
- Update (4/6/2018): Live demo of this model is available at: http://gh-demo.kubeflow.org/ Copy and paste your private Github issues and try it yourself!
- The Github repo for this article. The notebook is viewable here.
- The place where I have realized the highest return on investment with regards to learning these concepts is the MOOC fast.ai by Jeremy Howard. In the latest version of this MOOC, Jeremy uses PyTorch instead of Keras, and provides high-level abstractions with useful features like cyclical learning rates with random restarts.
- Francois Chollet's blogs, the documentation of the Keras library, and the discussions in the Keras Github repo.
- This kaggle dataset page has the data for this exercise (thanks David Shinn). If you would like to expand this dataset, you can gather a larger dataset by expanding the original query parameters as described in the **appendix**.
- Avneesh Saluja—machine learning scientist at Airbnb, discusses how he uses similar approaches to what is presented here in this excellent talk.

Thanks

Additionally, thanks to those who have reviewed this article and have given me valuable input: David Shinn, Robert Chang and Zachary Deane-Mayer.

Get In Touch!

I hope you enjoyed this blog post. Please feel free to get in touch with me on Twitter, Linkedin, or Github.

Caveats & Disclaimers

Any ideas or opinions presented in this article are my own. Any ideas or techniques presented do not necessarily foreshadow future products of Github.

. . .

Appendix—[Optional] How to acquire Github issue data yourself, from scratch.

Image Credit: https://goo.gl/images/NfaY7t

The easiest way to acquire this data is to use BigQuery. 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. If an astute reader figures out an easier way to obtain this data, please make a note in the comments! This might even make for an excellent Kaggle Dataset which you can earn points for.

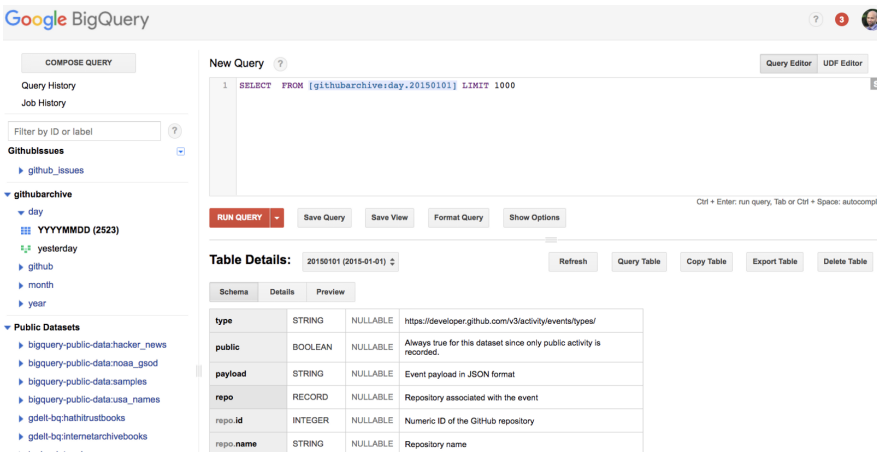
We are going to follow the instructions closely in this link. Please reference this documentation if you get lost. However I'll provide the steps below:

If you don't already have a Google project:

- Login into the Google Developer Console
- Create a project and activate the BigQuery API

Make sure the project you create is linked to your billing account by navigating to the billing console so that you can take advantage of the $300 credit you get when you are a new user (the queries for this exercise cost me $4).

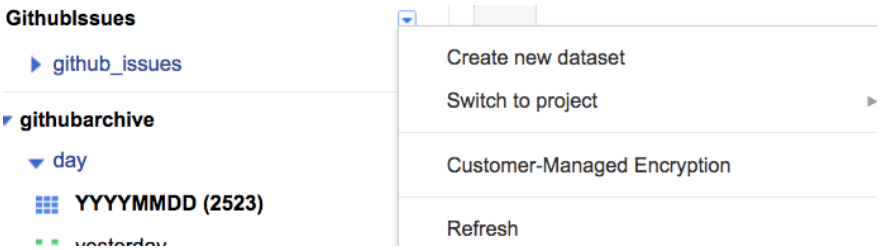
After completing the above steps, you can proceed to querying the data. You can view query console by clicking on this link. On this screen you will want to select the "Query Table" button in the upper right hand corner. Then you will be presented with a screen that looks like this:



The BigQuery Query Editor

Next, you will have to click the "Show Options" button and make sure the "Legacy SQL" check box is NOT selected (it is selected by default).

You will also notice that on the left hand side you will see the name of your project, which I have named *GithubIssues*. Click on the blue drop-down box next to this (as pictured below) and select "Create new dataset", and provide a name for a dataset. You will notice that I have named my dataset *github_issues*. You will need this later.

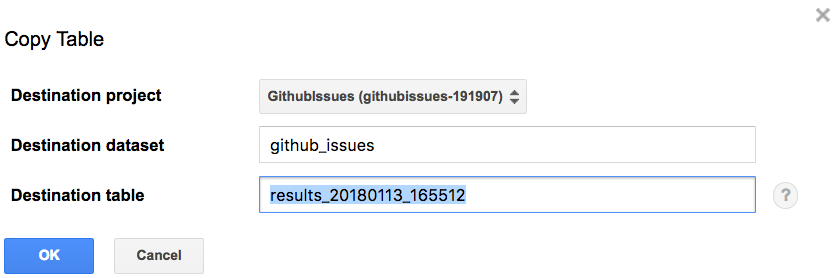


Now, we are ready to get the data we want! Copy and paste the below SQL query into the console and click the red button “Run Query”. Alternatively, you can also click this link. Feel free to study the SQL below, we are simply gathering issue titles and bodies and performing some cleaning of the data while we are at it.

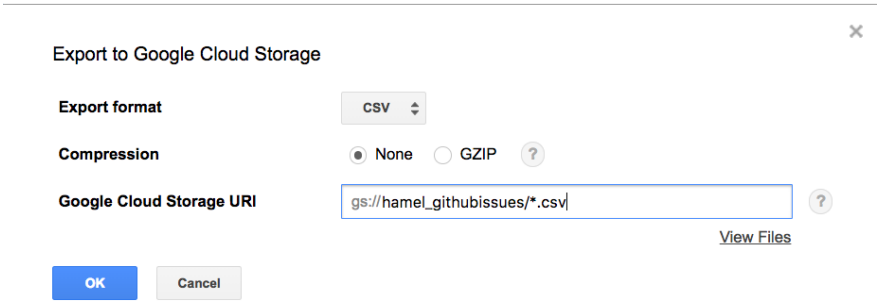
```
SELECT
  url as issue_url
  -- replace more than one white-space character in a
, REGEXP_REPLACE(title, r"\s{2,}", ' ') as issue_title
, REGEXP_REPLACE(body, r"\s{2,}", ' ') as body

FROM(
    SELECT
        JSON_EXTRACT(payload, '$.issue.html_url') as u
        -- extract the title and body removing parenth
      , LOWER(TRIM(REGEXP_REPLACE(JSON_EXTRACT(payload
      , LOWER(TRIM(REGEXP_REPLACE(JSON_EXTRACT(payload
    FROM `githubarchive.day.2017*`
    WHERE
      -- ALL Of 2017
          _TABLE_SUFFIX BETWEEN '0101' and '1231'
      and type="IssuesEvent"
      -- Only want the issue at a specific point other
      and JSON_EXTRACT(payload, '$.action') = "\"opene
) as tbl
```

Query that will return ~5M rows containing (url, title, body) from Github Issues. This file is also available in this repo: https://github.com/hamelsmu/Seq2Seq_Tutorial

After your query finishes, you will have to save it to a Google Cloud Bucket, which is analogous to Amazon S3 storage. To do so, you should click the “Save as Table” button above your query results, which will display the below window:



Select the destination dataset (which is what you created in an earlier step) and press ok. Now navigate to the table you just created in the left-hand pane and select the blue drop down menu and select “Export Table” in which case you will be presented with a window like this:



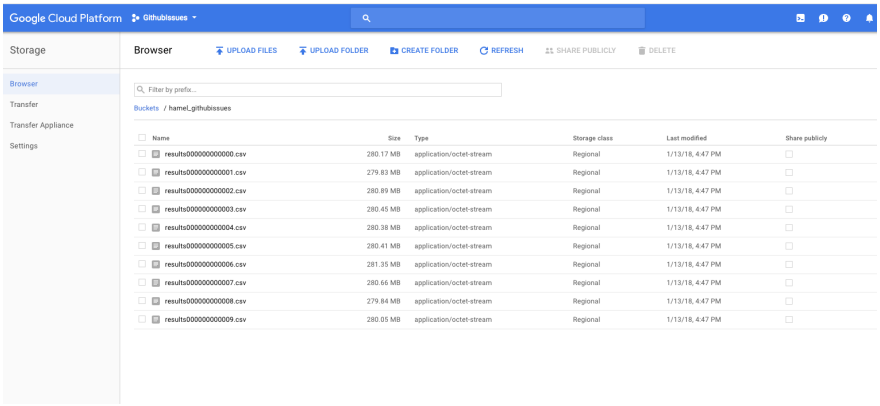
You need to click “View Files” link to create a bucket if you don’t have one.For the Google Cloud Storage URI the syntax is as follows:

gs://bucket_name/destination_filename.csv

However you will have to add a wildcard character as the data is too big to fit into one csv file (the total data is ~ 3GB). For example, the name of my (private) bucket is hamel_githubissues, so the path I put here is

gs://hamel_githubissues/.csv*

Once you do this correctly, you will see a message next to your table name that says (…extracting). This only takes a few minutes. After this is done you can navigate to your Google Cloud storage bucket and you will see the files (will look like this:)



Multi-part csv files that contains the data from our query.

Once you download this data, you have everything you need to complete the rest of this tutorial. You can download this data by simply clicking on each file, or by using the Google Cloud Storage CLI. You can even accomplish this whole process of querying the table by using pandas. I honestly just decided to use the user-interface because I rarely use Google Cloud otherwise.