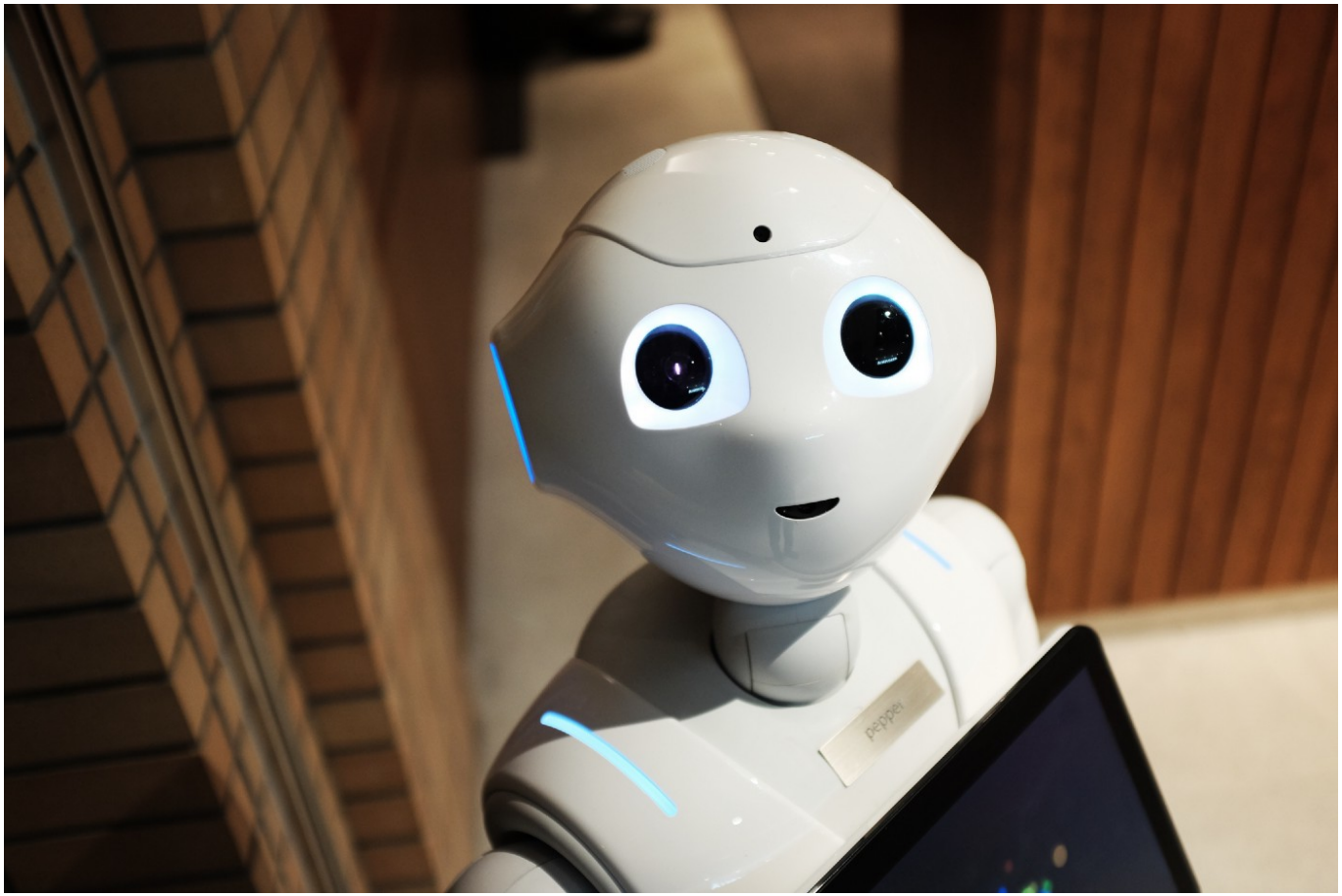


Photo by Alex Knight on Unsplash

HANDS-ON TUTORIALS

The Complete Guide to Building a Chatbot with Deep Learning From Scratch

With spaCy for entity extraction, Keras for intent classification, and more!

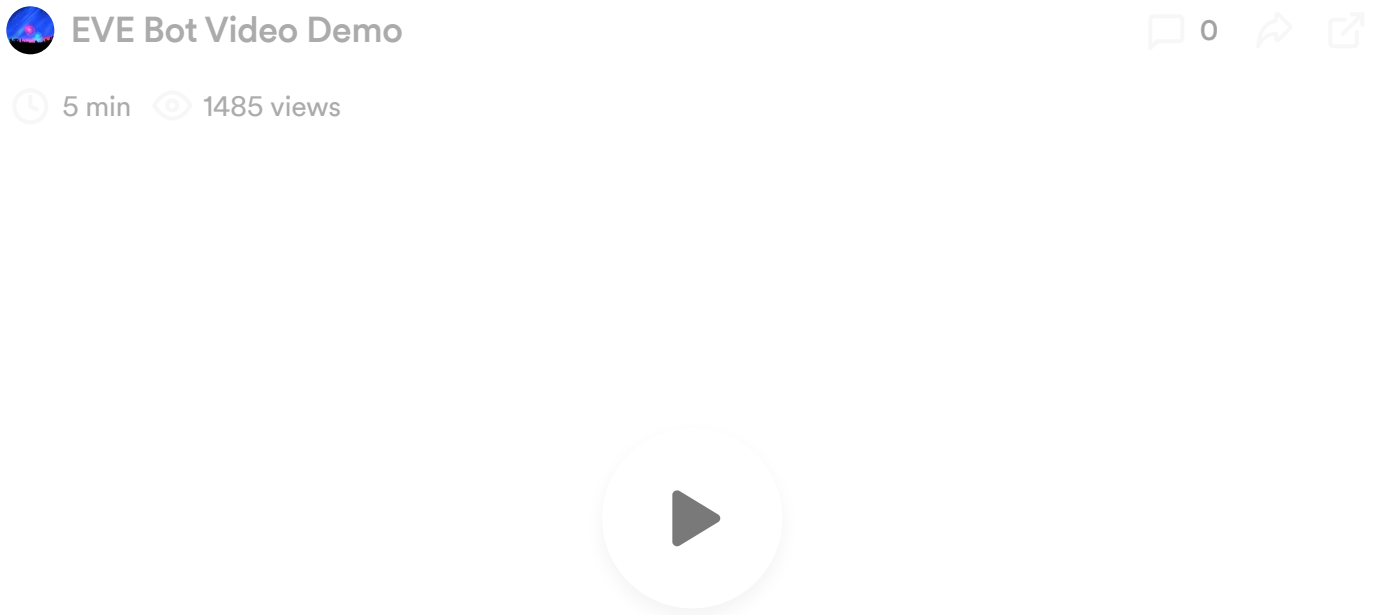
Matthew Evan Taruno Sep 7, 2020 · 24 min read

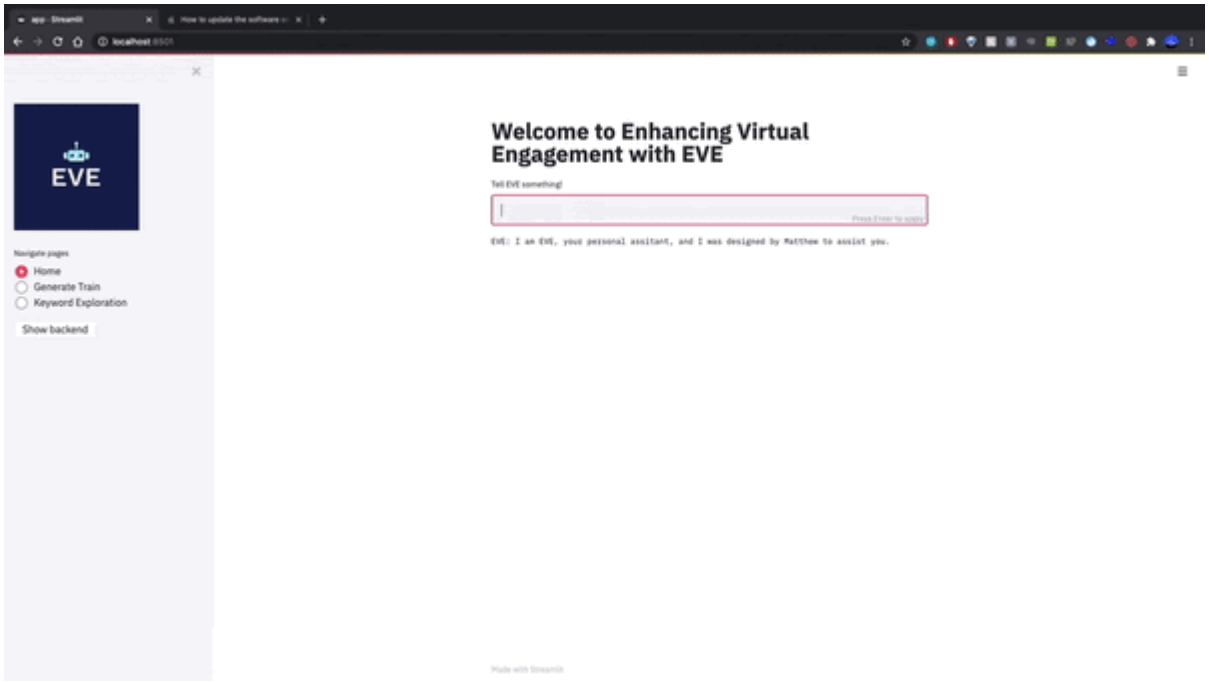
Over the past month, I wanted to look for a project that encompasses the entire data science end-to-end workflow — from the data pipeline, to deep learning, to deployment. It had to be challenging, but not pointlessly so — it still had to be something useful. It took a little ideation and divergent thinking, but when the idea of making a personal assistant came up, it didn't take long for me to settle on it. Conversational assitants are everywhere. Even my university is currently using Dr. Chatbot to track the health status of its members as an effective way to monitor this current pandemic. And it just makes sense: chatbots are faster, easier to interact with, and is super useful especially for things that we just want a fast response on. In this day and age, being able to talk to a bot for help is starting to become the new standard. I personally believe bots are the

future because they just make our lives so much easier. Chatbots are also a key component in Robotic Process Automation.

Now I want to introduce EVE bot, my robot designed to Enhance Virtual Engagement (see what I did there) for the Apple Support team on Twitter. Although this methodology is used to support Apple products, it honestly could be applied to any domain you can think of where a chatbot would be useful.

Here's my demo video for EVE. (And here's my Github repo for this project)





Demo Snippet — Asking EVE how to update my MacBook Pro. Image by Author.

Image by Author.

The demo goes through a high level and less technical overview of my work to keep it short and sweet. But if you want to geek out and learn about how I did it, that's what the rest of this article is exactly for!

I also hope this post would be a guide for those out there who need some structure on how to build your very own bot from scratch — in the sense that you are only using well-known, general purpose packages like Keras and spaCy, and not huge APIs specifically designed for chatbots like the Rasa API.

EVE is a context based bot powered by deep learning. Context-based bots are the step above the simple, keyword-based chatbot you might have seen a long time ago (see: Eliza bot). While I of course did have inspirations and it does have similarities to how it's done in the industry, I offer some approaches that I reasoned myself on how to make a chatbot in 2020.

This method I show in this post utilizes the same logic that powers the chatbots of big companies like Amazon with their Lex conversational AI service.

What's out there?

Before I get in the technical workings, it's important to know at what level of granularity you want to be making a chatbot at. A chatbot is like cooking

spaghetti. You can really start from raw tomatoes, or you can start from canned ones that other people already made for you. It's also similar in that there are many different components — you have:

- **The framework:** where your bot decides how to respond to a customer based on their utterance. You can use higher level tools such as DialogFlow (by Google), Amazon Lex, and Rasa for your framework. These higher level APIs require less work from you compared to the Python based work I am going to show in this post, but you may not be as confident about what's going on in the background. Mine opts for the more white box approach with Tensorflow, spaCy, and Python.
- **Dialogue management:** This is the part of your bot that is responsible for the state and flow of the conversation — it's where you can prompt users for information you need and more.
- **Deployment interface:** You can build an interface with the Messenger API, you can deploy it on WhatsApp Business (with a fee), or really anywhere like your own website or app if you have. I deployed mine on Streamlit as a really quick demo tool.

What should the goal for my chatbot framework be?

Okay, you've decided to make your own framework. Now here's how.

Your goal is two fold — and *both* are important:

1. Entity Extraction
2. Intent Classification

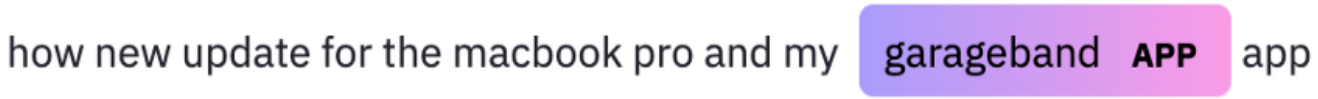
> When starting off making a new bot, this is exactly what you would try to figure out first, because it guides what kind of data you want to collect or generate. I recommend you start off with a base idea of what your intents and entities would be, then iteratively improve upon it as you test it out more and more.

Entity Extraction

Entities are predefined categories of names, organizations, time expressions, quantities, and other general groups of objects that make sense.

Every chatbot would have different sets of entities that should be captured. For a pizza delivery chatbot, you might want to capture the different types of pizza as an entity and delivery location. For this case, *cheese* or *pepperoni* might be the pizza entity and *Cook Street* might be the delivery location entity. In my case, I created an Apple Support bot, so I wanted to capture the hardware and application a user was using.

The hardware entity “macbook pro” is tagged. Other hardware might include iPhone and iPads. Image by Author.



The app entity captured “garageband” is tagged. Other app entities might include Apple Music or FaceTime. Image by Author.

To get these visualizations, displaCy was used, which is spaCy’s visualization tool for Named Entity Recognition (they have more visualizers for other things like dependency parsing as well).

Intent Classification

Intents are simply what the customer intend to do.

> Are they trying to greet you? Are they trying to talk to a representative? Are they challenging if you're a robot? Are they trying to do an update?

Intent classification just means figuring out what the user intent is given a user utterance. Here is a list of all the intents I want to capture in the case of my Eve bot, and a respective user utterance example for each to help you understand what each intent is.

- **Greeting:** Hi!
- **Info:** What's the thinnest MacBook you have available?
- **Forgot password:** I forgot my login details, can you help me recover it?
- **Speak Representative:** Can I talk to a human please
- **Challenge Robot:** Are you even a human
- **Update:** I would like to update my MacBook Pro to the latest OS
- **Payment:** I was charged double for the iPhone X I bought yesterday at Best Buy

- **Location:** Where is the nearest Apple Store located from me?
- **Battery:** My battery keeps on draining and dies in an hour
- **Goodbye:** Thanks Eve, see you later

Intents and **entities** are basically the way we are going to decipher what the customer wants and how to give a good answer back to a customer. I initially thought I only need intents to give an answer without entities, but that leads to a lot of difficulty because you aren't able to be granular in your responses to your customer. And without multi-label classification, where you are assigning multiple class labels to one user input (at the cost of accuracy), it's hard to get personalized responses. Entities go a long way to make your intents just be intents, and personalize the user experience to the details of the user.

With these two goals established, I boiled down my process into five steps that I'll break down one by one in this post:

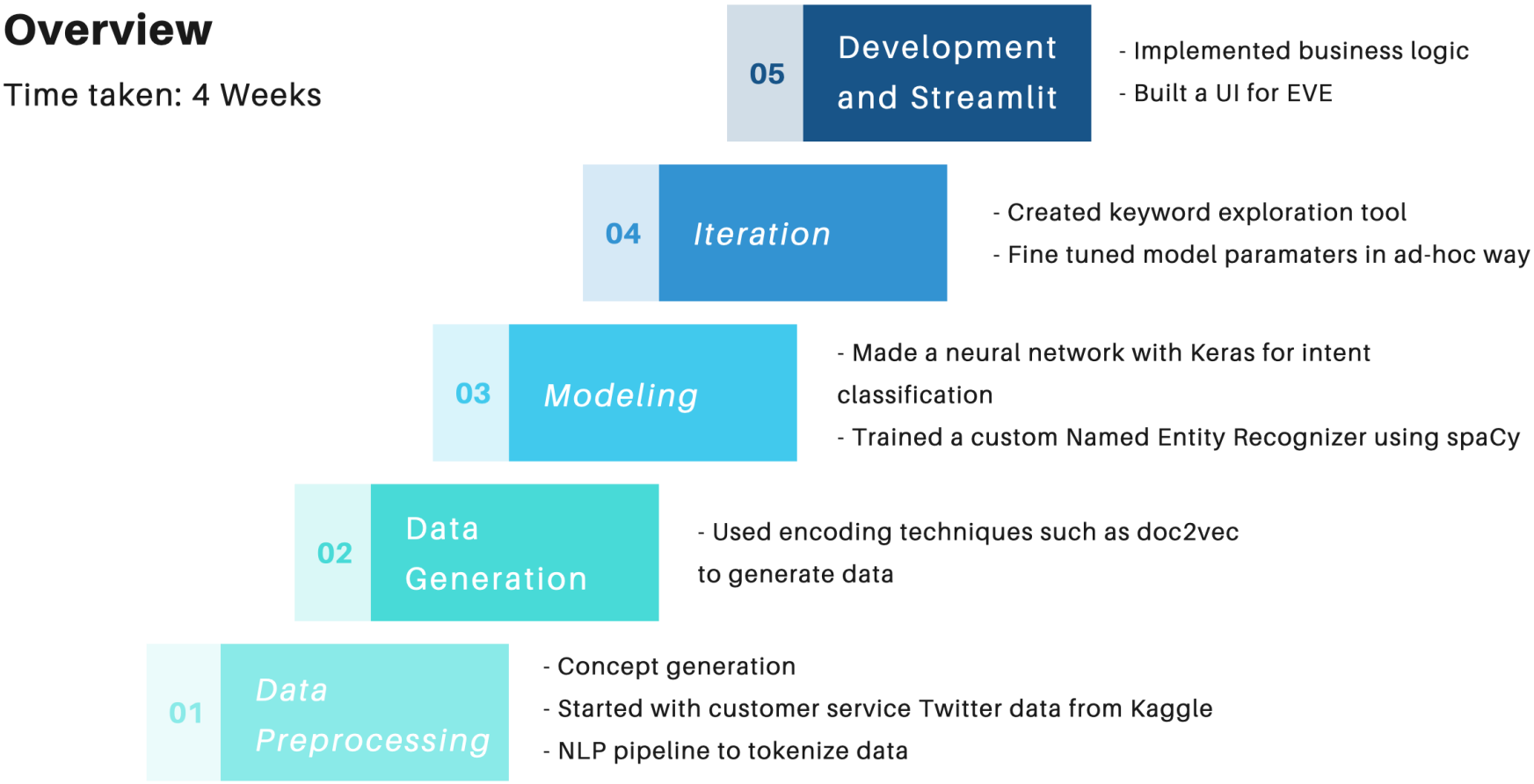


Image by Author.

1. Data Preprocessing

My notebook for data preprocessing is here.

I mention the first step as data preprocessing, but really these 5 steps are not done linearly, because you will be preprocessing your data throughout the entire chatbot creation.

But even before data preprocessing, where on earth do you get your data?

It really depends on the domain of your chatbot.

Data Guidelines

- **You have to find data that best covers as many scenarios that the customer might ask you and that you want to reply to as possible.** The data should contain all the intents you want to be able to answer. This might be a very hard task, but just know your data doesn't have to be perfect, and it could come from multiple sources as long as they are within the same general domain.
- For each intent, you should have a sizable amount of examples so that your bot will be able to learn the nature of that intent.
- If you have really have no data at all, like the one I am thinking of in my next project (which isn't an English chatbot), I am trying to work around it by making a Google sheets form for people to ask questions to my bot. The entire point is to get data that most closely resembles questions that real people are going to ask your bot.

But back to Eve bot, since I am making a Twitter Apple Support robot, I got my data from customer support Tweets on Kaggle. Once you finished getting the right dataset, then you can start to preprocess it. The goal of this initial preprocessing step is to get it ready for our further steps of data generation and modeling.

First, I got my data in a format of inbound and outbound text by some Pandas merge statements. With any sort of customer data, you have to make sure that the data is formatted in a way that separates utterances from the customer to the company (inbound) and from the company to the customer (outbound). **Just be sensitive enough to wrangle the data in such a way where you're left with questions your customer will likely ask you**.

Shortly after, I applied an NLP preprocessing pipeline. Which steps to include depends on your use case (like languages you want to support, how colloquial you want your bot to be, etc.). Mine included:

- Converting to lower case
- Tokenizing using NLTK's Tweet tokenizer
- Removing punctuation and URL links

- Correcting misspellings (leviathans distance)
- Removing stop words
- Expanding contractions
- Removing non-english Tweets with spaCy
- Lemmatization (you can choose stemming as an alternative)
- Removing emojis and numbers (you can leave emojis in if your model can read them in, and if you plan to do emoji analysis)
- Limiting each Tweet length to 50 (for compactness)

I compiled all this steps into one function called `tokenize`. At every preprocessing step, I visualize the lengths of each tokens at the data. I also provide a peek to the head of the data at each step so that it clearly shows what processing is being done at each step.

```
# Initializations

# Punctuations I want to remove, including the empty token
puncts = ['\u200d','?', '....','..','...','','@','#', ',', '.', '"', ':', ')', '(', '-', '!', '
    '•', '~', '£', '·', '_', '{', '}', '©', '^', '®', '`',  '<', '→', '°', '€', '™', '›',  '♥',
    '½', 'à', '…', '“', '★', '”', '–', '●', 'â', '►', '−', '¢', '²', '¬', '░', '¶', '↑', '±', '
    '—', '‹', '─', '▒', ': ', '¼', '⊕', '▼', '▪', '†', '■', '’', '■', '¨', '▄', '♫', '☆', 'é',
    'Ã', '⋅', '‘', '∞', '∙', ') ', '↓', '、', '│', ' (', '»', '，', '♪', '╩', '╚', '³', '・', '╦'
    '¹', '≤', '‡', '√', '!','🅰','🅱']
```

```
10
11  # Using NLTK's stop words corpus
12  stopwords.words('english');
13  stop_words = set(stopwords.words('english'))
14
15  # Found a dictionary of common contractions and colloquial language
16  contraction_colloq_dict = {"btw": "by the way", "ain't": "is not", "aren't": "are not","can't":
17
18  # Initializing the lemmatizer
19  lemmatizer = nltk.stem.WordNetLemmatizer()
20
21  import emoji # For emoji removal step
22
23  import en_core_web_sm
24  try:
25      # Initializing spaCy objects (Don't run this more than once)
26      nlp_cld = spacy.load('en',disable_pipes=["tagger","ner"])
27      language_detector = spacy_cld.LanguageDetector()
28      nlp_cld.add_pipe(language_detector)
29  except ValueError as e:
30      print("The spaCy function was run more than once, but that's okay because it means it was ir
31
32  # My preprocessing functions (defining them here so that I could access them from anywhere in tl
33
34  def visualize_lengths(data,title):
35      '''Visualizing lengths of tokens in each tweet'''
36      lengths = [len(i) for i in data]
37      plt.figure(figsize=(13,6))
38      plt.hist(lengths, bins = 40)
39      plt.title(title)
40      plt.show()
41
42  def remove_from_list(x, stuff_to_remove) -> list:
43          ''' Making a function to remove a list of items from a list'''
```

```
        for item in stuff_to_remove:
            # Making sure to iterate through the entire token
            for i,token in enumerate(x):
                if item == token:
                    del x[i]
        return x

def remove_links(doc):
    return [re.sub(r'^https?:\/\/.*[\r\n]*', '', token, flags=re.MULTILINE) for token in doc]

def correct_spellings(x):
    ''' Takes as input a list and outputs a list of the corrected spelling'''
    corrected_text = []
    for word in x:
        if word in x:
            corrected_text.append(spell.correction(word))
        else:
            corrected_text.append(word)
    return corrected_text

def replace_from_dict(x,dic):
    ''' Making a function to replace all the items in a list based on a dictionary. I made sure
            method to insert the longer-gram replacement as distinct items in the list at that s
    replaced_counter = 0
    for item in dic.items():
        for i, e in enumerate(x):
            if e == item[0]:
                replaced_counter+=1
                    # Inserting the expanded tokens in a way that preserves the order
                del x[i]
                for ix, token in enumerate(item[1].split()):
                    x.insert(i+ix,token)
#         print(f"Amount of words replaced: {replaced_counter}")
```

```
    return x

def only_english(x):
    ''' Making a function that only accepts English by appending True if it is English and Fals
            into a mask. Returns a mask'''
    mask = []
    x = x.apply(" ".join)
    try:
        for i,doc in tqdm(enumerate(nlp_cld.pipe(x, batch_size=512))):
            if 'en' not in doc._.languages or len(doc._.languages) != 1:
                mask.append(False)
            else:
                mask.append(True)
    except Exception as e:
        print(f"Exception:{e}")
    return mask

def get_wordnet_pos(word):
    """Map POS tag to first character lemmatize() accepts"""
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)

def lemmatize_list(x):
    ''' This lemmatizer function should work on a single list of tokenized data'''
#         # Turning list into a string
    x = " ".join(x)
    # Returning a list again
    return [lemmatizer.lemmatize(w, get_wordnet_pos(w)) for w in nltk.word_tokenize(x)]

def extract_emojis(s):
```

```
111         return [c for c in s if c not in emoji.UNICODE_EMOJI]
112
113 def limit_length(x, max_tokens, min_tokens):
114     ''' Inputs a list and drops it out of the document if
115         the document has more than the max and less than the min'''
116     output = x
117     if len(x) > max_tokens:
118         output = np.nan
119     if len(x) <= min_tokens:
120         output = np.nan
121     return output
122
123 def clean_numbers(x):
124     for i,j in enumerate(x):
125         if bool(re.search(r'\d', j)):
126             del x[i]
127     return x
128
129 def validate(func, locals):
130     ''' Validating a function below to accept correct input'''
131     for var, test in func.__annotations__.items():
132         value = locals[var]
133         try:
134             pr=test.__name__+': '+test.__docstring__
135         except AttributeError:
136             pr=test.__name__
137         msg = '{}=={}; Test: {}'.format(var, value, pr)
138         assert test(value), msg
139
140 # End to end tokenizer function
141
142 def my_tokenizer(data: lambda _data: isinstance(_data, pd.Series)) -> 'Cleaned Pandas Series':
143     ''' I am making my own end-to-end tokenizer function for preprocessing that accepts
144     a Pandas Series as input and outputs a preprocessed Pandas Series'''
```

```
145
146     # Making sure input is a series (these are two ways of doing the same thing)
147     assert isinstance(data,pd.Series), 'Input must be a Pandas Series'
148 #     validate(my_tokenizer, locals())
149
150     # 1. Converting all to lower case
151     data = data.str.lower()
152
153     print(f'1. Original shape of data is {data.shape}')
154
155
156     # 2. Tokenizing with NLTK's TweetTokenizer. This limits repeated characters to
157     # three with the reduce lens paramater and strips all the @'s. It also
158     # splits it into 1-gram tokens
159     tknzr = TweetTokenizer(strip_handles = True, reduce_len = True)
160     # Using progress apply to show the progress bar
161     data = data.progress_apply(tknzr.tokenize)
162
163     print(f'2. Tokenized, removed handles, and reduced the length of repeated characters.\
164           \n Shape is still {data.shape}. \n \n Peek: \n {data.head()}')
165     visualize_lengths(data, 'Length of Tokens after Step 2')
166
167     # 3. Removing the punctuation
168
169     data = data.progress_apply(remove_from_list, stuff_to_remove = puncts)
170     print(f'3. Removed empty tokens and punctuation. Shape is still {data.shape}. \n \n Peek: \
171     visualize_lengths(data, 'Length of Tokens after Step 9')
172
173     # 3.1. Removing links
174
175     data = data.progress_apply(remove_links)
176
177     print(f'3. Removed the links. Shape is still {data.shape}. \n \n Peek: \n {data.head()}')
```

```
    visualize_lengths(data, 'Length of Tokens after Step 3')

    # 4. Checking for and correcting mispellings
    spell = SpellChecker()

    # Skipping this step first because it takes way too long
#     data = data.progress_apply(correct_spellings)

    print(f'4. Applied automispelling corrections. Shape is still {data.shape}. \n \n Peek: \n

    # 5. Removing the stop words, utilizing the same remove_from_list function defined above

    data = data.progress_apply(remove_from_list, stuff_to_remove = stop_words)

    print(f'5. Removed the stop words. Shape is still {data.shape}. \n \n Peek: \n {data.head()
    visualize_lengths(data, 'Length of Tokens after Step 4')

    # 6. Expanding contractions and colloquial language

    data = data.progress_apply(replace_from_dict, dic = contraction_colloq_dict)
    print(f'6. Expanded contractions into extra tokens. Shape is still {data.shape}. \n \n Peek
    visualize_lengths(data, 'Length of Tokens after Step 5')

    # 7. Removing non-english Tweets with spaCy

    data = data[only_english(data)]
    print(f'7. Remove all non-english Tweets. Shape is now {data.shape}. Clearly less than befo

    # 8. Lemmatization

    data = data.progress_apply(lemmatize_list)
    print(f'8. Lemmatized the tokens. Shape is still {data.shape}. \n \n Peek: \n {data.head()}
    visualize_lengths(data, 'Length of Tokens after Step 8')
```

```
212
213       # Removing again to make sure I get everything
214       data = data.progress_apply(remove_from_list, stuff_to_remove = puncts)
215
216       # 9.1. Removing emojis -- (UPDATE) in a way that preserves Series indexes
217
218       unique_emojis = [i[0] for i in emoji.UNICODE_EMOJI]
219       data = data.apply(remove_from_list, stuff_to_remove = unique_emojis)
220
221       # 9.2. Removing numbers -- (UPDATE) also in a way that preserves Series indexes
222       # Using nested list comprehension
223       data = data.progress_apply(clean_numbers)
224
225       print(f'9. Removed emojis and numbers. Shape is still {data.shape}. \n \n Peek: \n {data.he
226       visualize_lengths(data, 'Length of Tokens after Step 9')
227
228       # 10. Limiting length of Tweet
229       max_tokens = 50
230       min_tokens = 5
231       data = data.progress_apply(limit_length, min_tokens = min_tokens, max_tokens = max_tokens)
232       # Dropping all nan values, which are the token limits that didn't meet the thresholding requ
233       data = data.dropna()
234       print(f'10. Limited each tweet to a max of {max_tokens} tokens and a min of {min_tokens} to
235       visualize_lengths(data, 'Length of Tokens after Step 10')
236
237       return data
```

tokenizer.py hosted with ♡ by GitHub view raw

My entire tokenizer function.

I started with 106k Apple Suppourt inbound Tweets. This is a histogram of my token lengths before preprocessing this data.

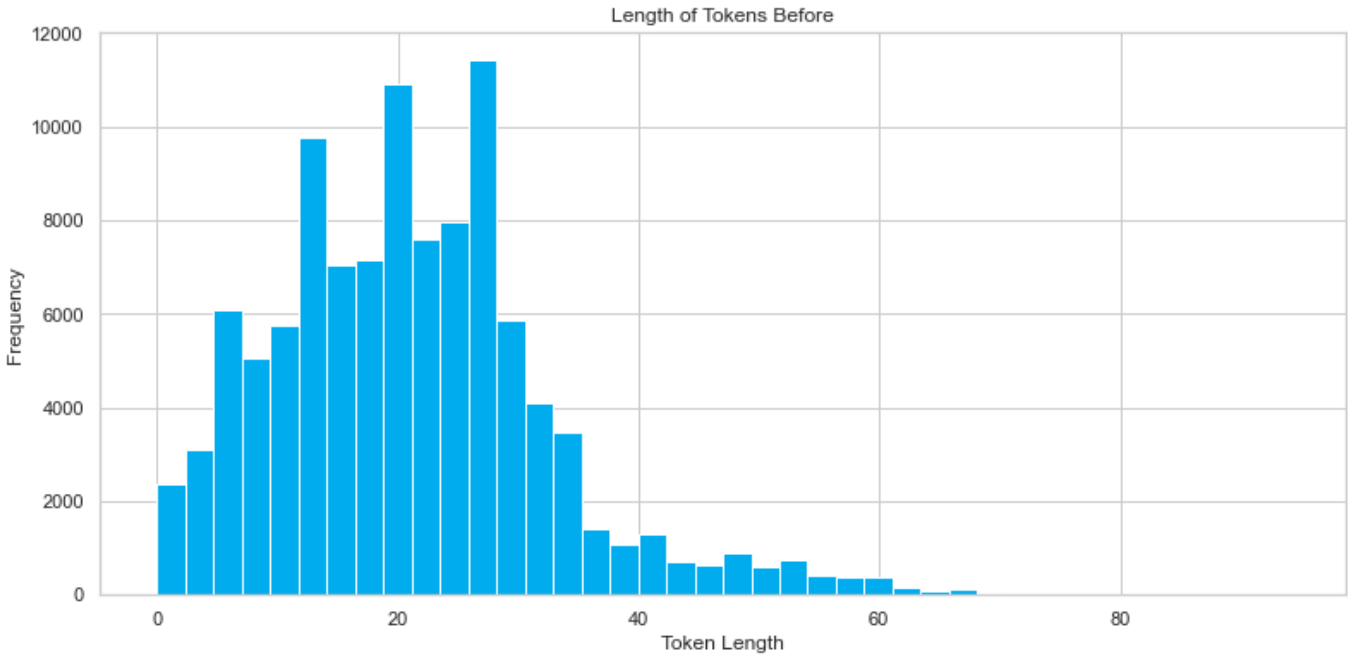


Image by Author.

After step 10, I am left with ~76k Tweets. In general, things like removing stop-words will shift the distribution to the left because we have fewer and fewer tokens at every preprocessing step.

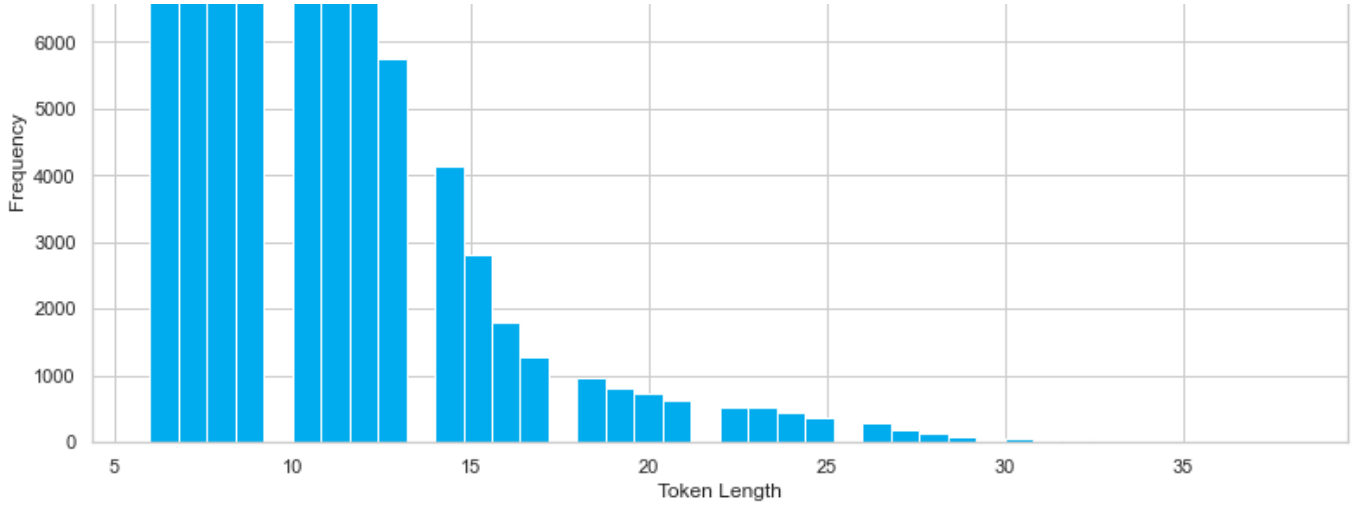


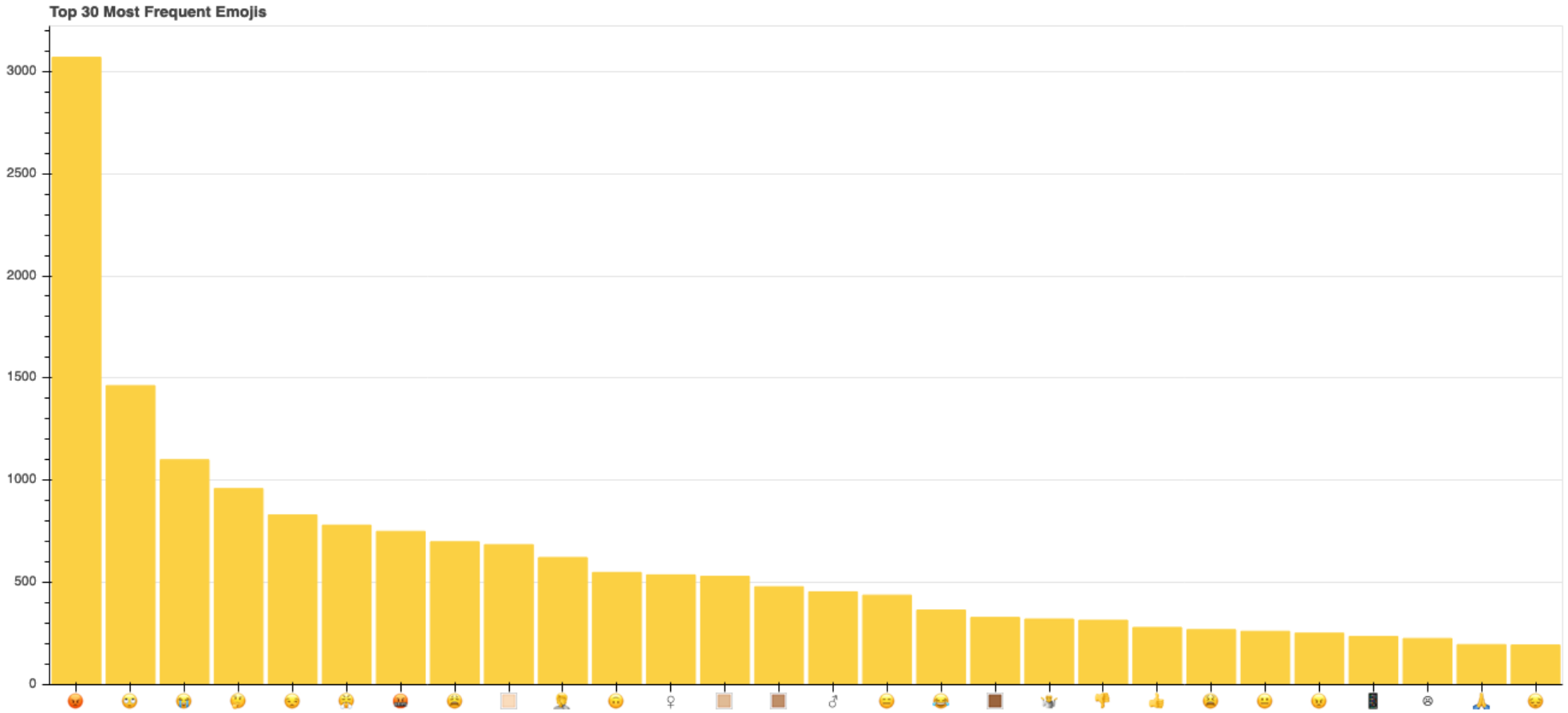
Image by Author.

Here you can see the results of this step. I got my data to go from the Cyan Blue on the left to the Processed Inbound Column in the middle. I also keep the Outbound data on the right in case I need to see how Apple Support responds to their inquiries that will be used for the step where I actually respond to my customers (it's called Natural Language Generation).

	Inbound	Processed Inbound	Outbound
0	@AppleSupport The newest update. I made sure to download it yesterday.	['new', 'update', 'i', 'make', 'sure', 'download', 'yesterday']	@115854 Lets take a closer look into this issue. Select the following link to join us in a DM and we'll go from there. https://t.co/GDrqU22YpT
1	@AppleSupport https://t.co/NV0yucs0lB	['hey', 'anyone', 'else', 'upgraded', 'io', 'issue', 'capital', 'i', 'mail', 'app', 'put']	@115854 We're here for you. Which version of the iOS are you running? Check from Settings > General > About.
2	@AppleSupport Tried resetting my settings .. restarting my phone .. all that	['hello', 'internet', 'someone', 'explain', 'symbol', 'keep', 'appear', 'phone', 'i', 'try', 'type', 'letter', 'i', 'also']	@115855 Let's go to DM for the next steps. DM us here: https://t.co/GDrqU22YpT
3	@AppleSupport This is what it looks like https://t.co/XCQU2l4xUB	['get', 'screenshot', 'say', 'iphonex', 'reserve', 'email', 'say', 'happen']	@115855 Any steps tried since it started last night?
4	@AppleSupport I have an iPhone 7 Plus and yes I do	['thank', 'update', 'phone', 'even', 'slow', 'barely', 'work', 'thank', 'ruin', 'phone']	@115855 That's great it has iOS 11.1 as we can rule out being outdated. Any steps tried since this started? Do you recall when it started?

The head of my Apple Support Twitter data before and after preprocessing. Image by Author.

Finally, as a brief EDA, here are the emojis I have in my dataset — it's interesting to visualize, but I didn't end up using this information for anything that's really useful.



Emojis in my dataset: most of the customers showed negative sentiment (angry and disappointed emojis being the top 2). Image by Author.

2. Data Generation

My complete script for generating my training data is here, but if you want a more step-by-step explanation I have a notebook here as well.

Yes, you read that right — data *generation*. You may be thinking:

> Why do we need to generate data? Why can't we just use the data that we preprocessed in the previous step?

That's a great question. The answer is because the data isn't labeled yet. and intent classification is a supervised learning problem. This means that we need intent labels for every single data point.

If you already have a labelled dataset with all the intents you want to classify, we don't need this step. But more times than not, you won't have that data. That's why we need to do some extra work to add intent labels to our dataset. This is quite an involved process, but I'm sure we can do it.

Here's our goal:

> In this step, we want to group the Tweets together to represent an intent so we can label them. Moreover, for the intents that are not expressed in our data, we either are forced to manually add them in, or find them in another dataset.

For example, my Tweets did not have any Tweet that asked “are you a robot.” This actually makes perfect sense because Twitter Apple Support is answered by a real customer support team, not a chatbot. So in these cases, since there are no documents in out dataset that express an intent for challenging a robot, I manually added examples of this intent in its own group that represents this intent. I explain more on how I did this in Step 4.

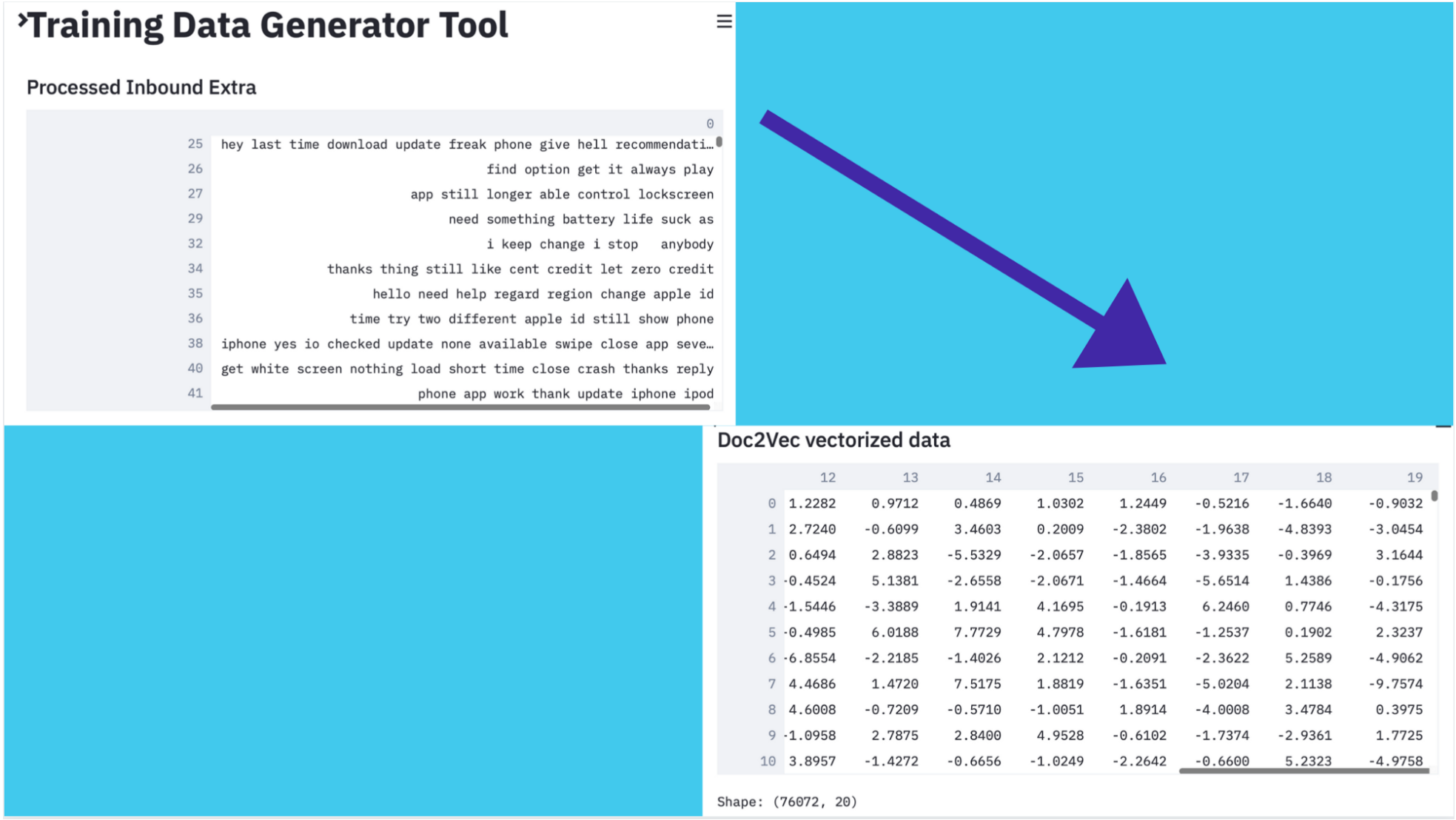
Since I plan to use quite an involved neural network architecture (Bidirectional LSTM) for classifying my intents, I need to generate sufficient examples for each intent. The number I chose is 1000 — I generate 1000 examples for each intent (i.e. 1000 examples for a greeting, 1000 examples of customers who are having trouble with an update, etc.). I pegged every intent to have exactly 1000 examples so that I will not have to worry about class imbalance in the modeling stage later. **In general, for your own bot, the more complex the bot, the more training examples you would need per intent.**

Embedding Techniques

This is where the *how* comes in, how do we find 1000 examples per intent? Well first, we need to know if there are 1000 examples in our dataset of the intent that we want. In order to do this, we need some concept of distance between each Tweet where if two Tweets are deemed “close” to each other,

they should possess the same intent. Likewise, two Tweets that are “further” from each other should be very different in its meaning.

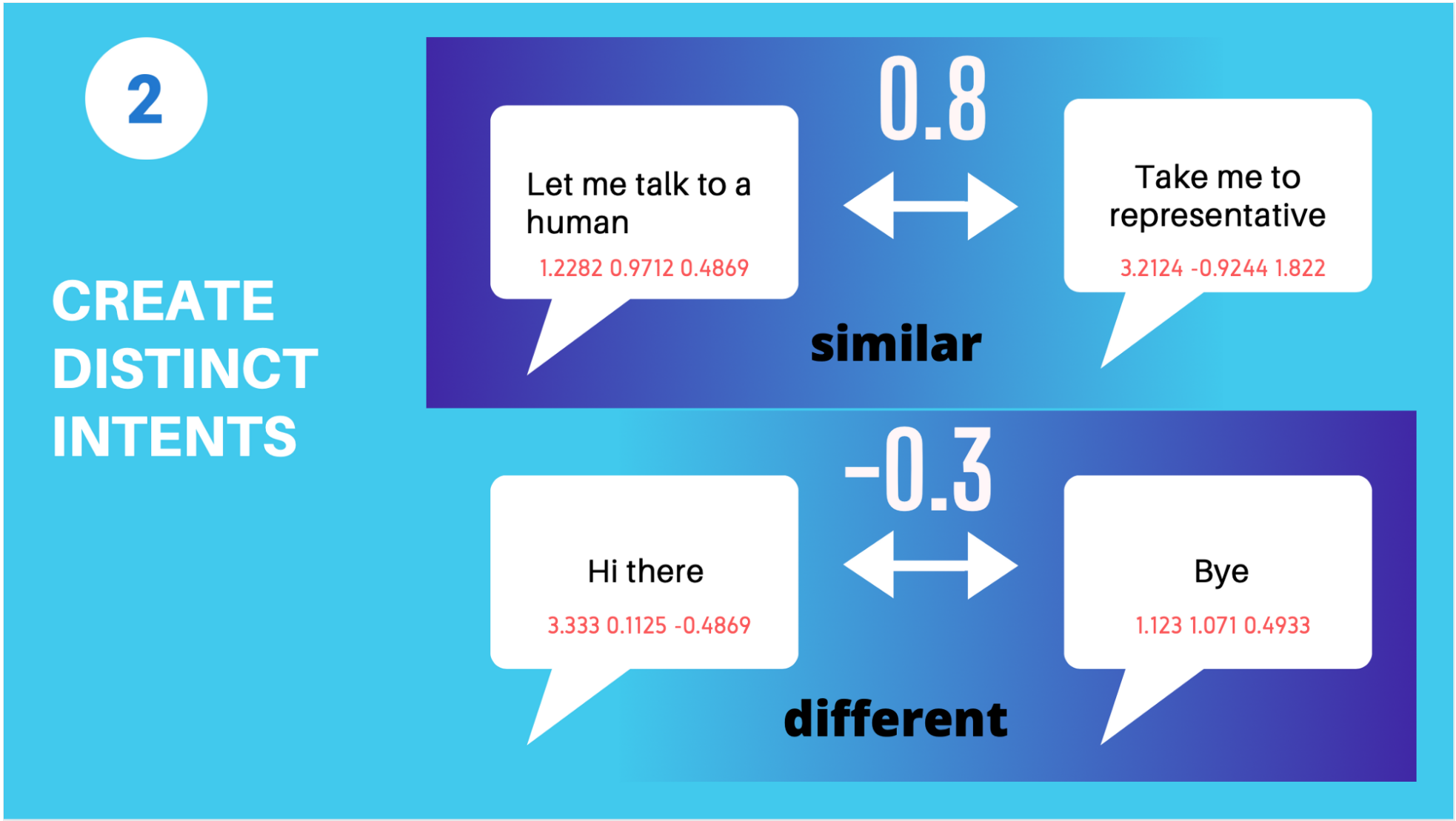
To do this, we use an encoding method known as Doc2Vec. Embedding methods are ways to convert words (or sequences of them) into a numeric representation that could be compared to each other. I created a training data generator tool with Streamlit to convert my Tweets into a 20D Doc2Vec representation of my data where each Tweet can be compared to each other using cosine similarity.



(left) Twitter data (right) Doc2Vec 20D vectorized data. Image by Author.

The following is a diagram to illustrate Doc2Vec can be used to group together similar documents. A document is a sequence of tokens, and a token is a sequence of characters that are grouped together as a useful

semantic unit for processing. For this data generation step, I also experimented with gloVe, but those only vectorize at a per word level, and if you want to try another word embedding (maybe it's more better suited to your domain), make sure that it vectorizes at the document level.



Toy example to show how the distances work. image by Author.

In this toy example, we convert every one of the utterances into 3D vectors (as can be seen in the pink array of 3 numbers below each phrase). Each of these 3D vectors is the numeric representation of that document. For example, “Hi there” is represented numerically as [3.333, 0.1125, -0.4869].

When we compare the *top two* similar meaning Tweets in this toy example (both are asking to talk to a representative), we get a dummy cosine similarity of 0.8. When we compare the bottom two different meaning Tweets (one is a greeting, one is an exit), we get -0.3.

Implementation

As for implementation, I used gensim’s Doc2Vec. You have to train it, and it’s similar to how you would train a neural network (using epochs).

Here's the catch. Before you train your Doc2Vec vectorizer, it's important to already know the intent buckets you want before hand. Once you know what intent buckets you want, you can apply this procedure that aims to get you your top N group of similar-in-meaning Tweets:

- Simply come up with the top keywords you can think of in that intent, then you append that to the end of your training data as a row (so for greeting, that row might be “hi hello hey”).
- The extra rows you add which represent the respective intents are going to be vectorized, which is great news because now you can then compare it to every single other row with cosine similarity.
- Vectorize by training your Doc2Vec vectorizer then fitting it on your data with the extra rows.
- You’ve successfully used keywords to represent an intent, and from this representation, you will find the top 1000 Tweets similar to it to generate your training data for that intent with Gensim’s `model.docvecs.most_similar()` method.

106647	anyone iphone issue phone freeze randomly pulse update io new ve…
0	battery power
1	password account login
2	credit card payment pay
3	update upgrade
4	info information
5	nearest apple location store

The extra rows are 0–5 and they are appended on top of my Tweets training data — 0 represents the battery intent, 1 represents the payment intent, and so on. Image by Author.

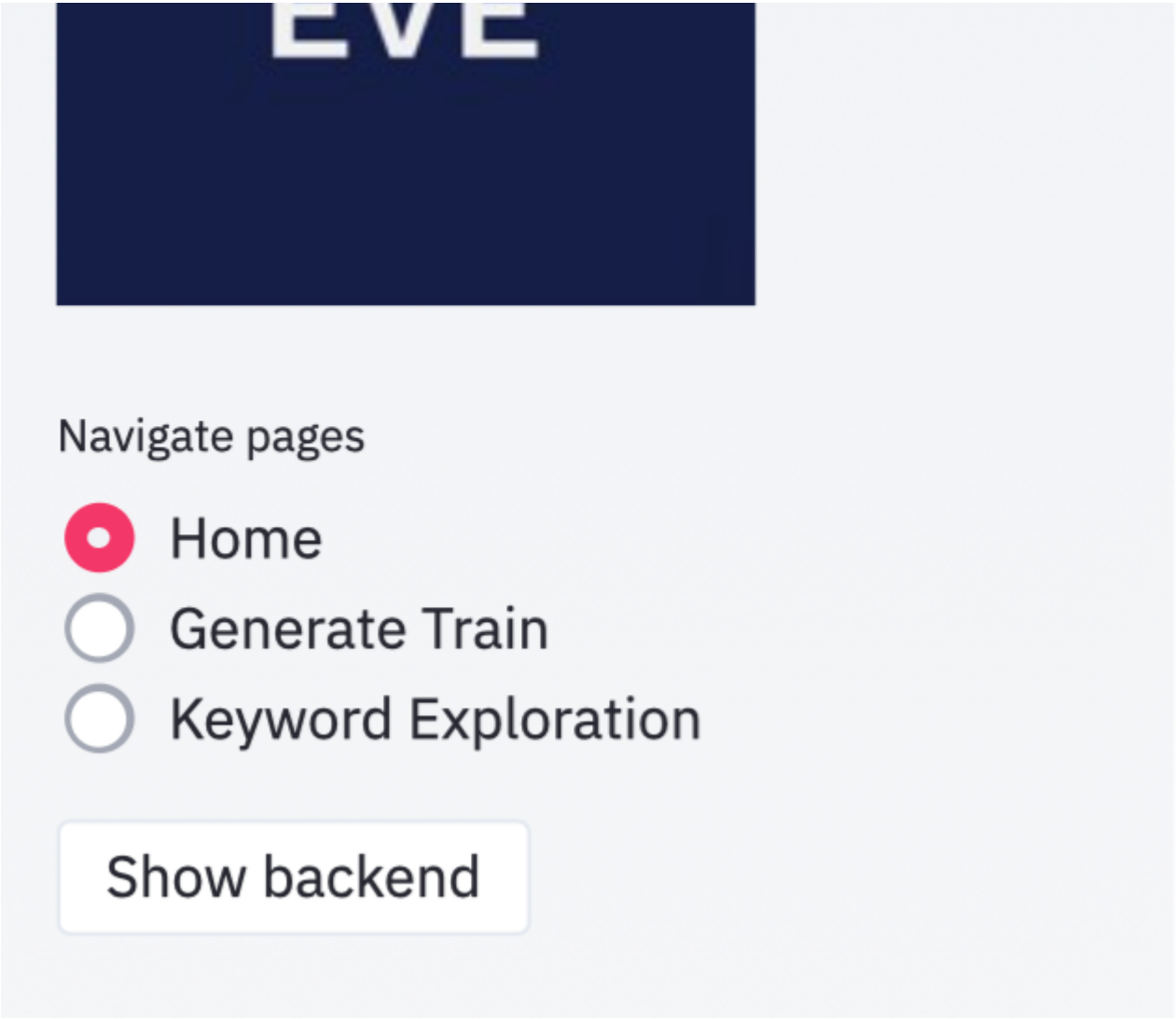
Note that we have to append our new intent keyword representations to the training data before you train your vectorizer because Gensim's implementation could only compare documents that have been put into this Doc2Vec vectorizer as training data. Moreover, it can only access the tags of each Tweet, so I had to do extra work in Python to find the tag of a Tweet given its content.

Once you've generated your data, make sure you store it as two columns "Utterance" and "Intent". Notice that the utterances are stored as a tokenized list. This is something you'll run into a lot and this is okay because you can just convert it to String form with `Series.apply(" ".join)` at any time.

Training data in format to feed into models

	Utterance	Intent
0	['phone', 'battery', '…	battery
1	['forgot', 'my', 'pass…	forgot_password
2	['payment', 'not', 'th…	payment
3	['want', 'update']	update
4	['need', 'information']	info
5	['answer', 'internally…	location
6	['talk', 'human', 'ple…	speak_representative
7	['hi']	greeting
8	['goodbye']	goodbye
9	['robot', 'human']	challenge_robot
10	['new', 'battery', 'up…	battery

This is what the training data format should look like. Image by Author.

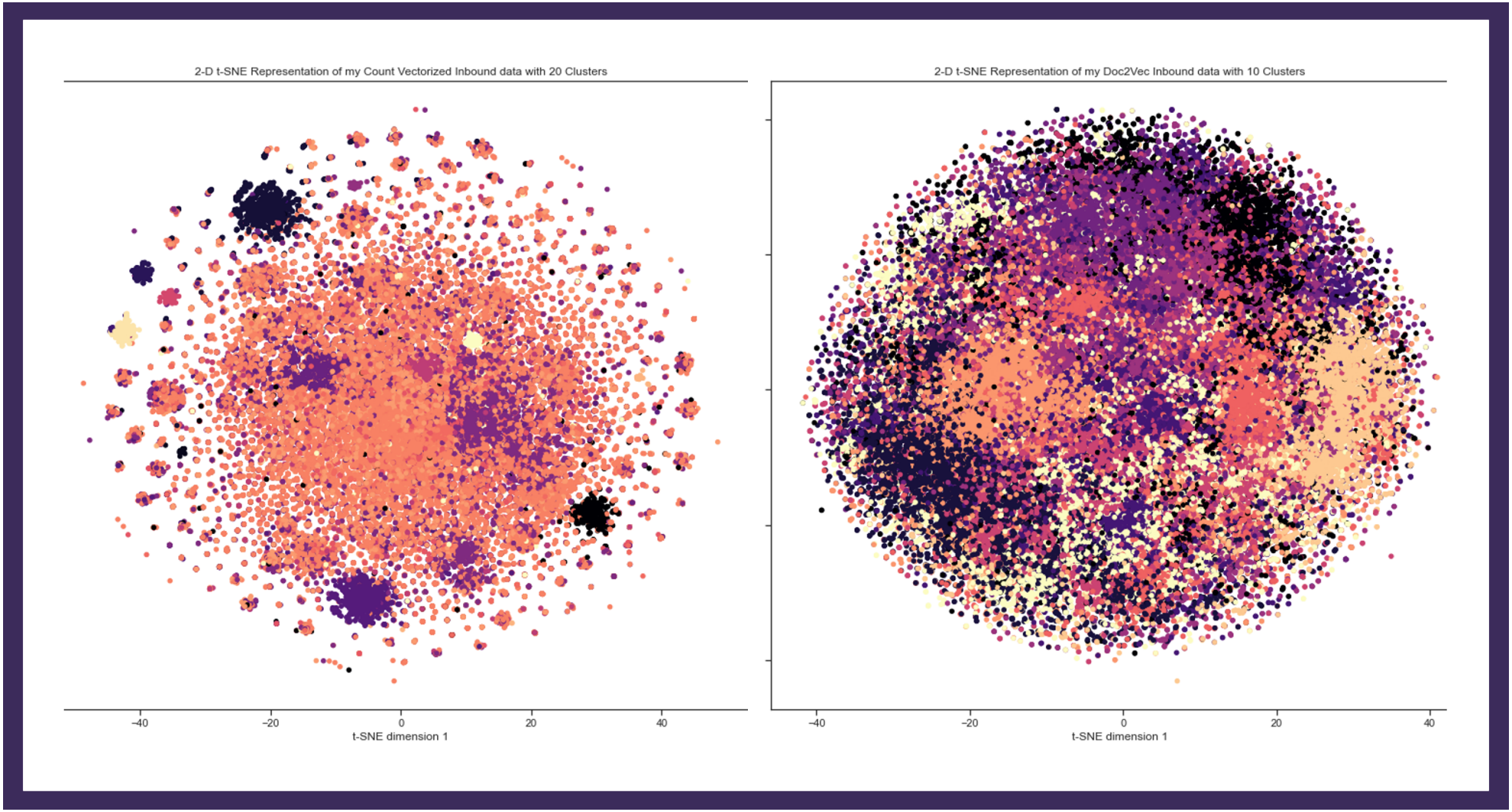


You can access my training data generator tool on the sidebar of my Streamlit app as one of the pages as well. Image by Author.

Some Other Methods I Tried to Add Intent Labels

The first thing I thought of to do was clustering. However, after I tried K-Means, it's obvious that clustering and unsupervised learning generally

yields bad results. The reality is, as good as it is as a technique, it is still an algorithm at the end of the day. You can't come in expecting the algorithm to cluster your data the way you exactly want it to.



t-SNE visualization showing K-Means Clustering failing to work because the data are not in natural clusters. Image by Author.

I also tried word-level embedding techniques like gloVe, but for this data generation step we want something at the document level because we are trying to compare between utterances, not between words in an utterance.

3. Modeling

Intent Classification

My intent classification notebook is here.

With our data labelled, we can finally get to the fun part — actually classifying the intents! I recommend that you don't spend too long trying to get the perfect data beforehand. Try to get to this step at a reasonably fast pace so you can first get a minimum viable product. The idea is to get a result out first to use as a benchmark so we can then iteratively improve upon on data.

There are many ways to do intent classification, Rasa NLU for instance allows you to use many different models such as support vector machines (SVMs), but here I will demonstrate how to do it with a neural network with a bidirectional LSTM architecture.

We are trying to map a user utterance (which is just a sequence of tokens) to one of the N intents that we specify. The data we start with should just have an utterance and an intent ground truth label as the columns. The order of this process is as follows (for the implementation, check out the Intent Classification notebook available at my Github):

1. Train Test Split (should *always* go first, my mentor drilled this point in my head)
2. Keras Tokenizer
3. Label Encode the target variable (intents)
4. Initialize the embedding matrix (I used gloVe embeddings because they had a special variant trained on Twitter data)
5. Initialize the model architecture
6. Initialize the model callbacks (techniques to address overfitting)
7. Fit the model and save it
8. Load the model and save the output (I recommend a dictionary)

Some things to keep in mind for the **embedding layer** (step 4):

- These pre-trained embeddings are essentially how you convert the text that goes into the model into a numeric representation. When you compare the cosine similarities of your numeric representation of different documents, they should have meaningful distances between each other (the cosine similarity between ‘king’ and ‘man’ should be closer than ‘king’ and ‘women’ for example).
- It is really important you choose the right pre-trained embeddings that is appropriate for the domain of your chatbot. If you have a conversational Twitter based chatbot, you probably don’t want embeddings trained on Wikipedia.
- I also recommend you to see if all the vocabulary you want to cover is in your pre-trained embedding file. I checked if my gloVe Twitter embeddings covers Apple specific words like “macbook pro” for example, and fortunately it does.

Checking to see if the word “macbook” is in my gloVe embeddings with grep (turns out it is!). Image by Author.

Some things to keep in mind for the **model architecture** (step 5):

- Make sure the output layer is softmax (if you want to do multi-label classification, then use sigmoid).
- Make sure the output layer has a dimensionality that is same as the number of intents you want to classify, otherwise you will run into shape issues.
- If you don’t label encode, your use of `model.predict()` might be inaccurate because that final dictionary where you output where the keys are the intents and the values are the probabilities of the utterance being that intent wouldn’t be mapped properly.
- When you are deploying your bot, you shouldn’t rerun the model. Rather, I wrote a script that starts by reading in the saved model file and does the predictions from there.

Results

The results are promising. The loss converges to a low level and the accuracy of my model on unseen data is 87%!

Image by Author.

Image by Author.

If you visualize the output of the intent classification, this is what it looks like for the utterance “my battery on my iphone stopped working!”:

Image by Author.

Preventing Overfitting

In order to prevent my model from overfitting, there are also some other settings I set in the form of Keras callbacks:

- Learning rate scheduling — Slowing down the learning rate after it gets past a certain epoch number
- Early stopping — Stopping the training early once the validation loss (or any other parameter you choose) reaches a certain threshold

And finally, after I ran the model, I saved it into an h5 file so I can initialize it later without retraining my model using Model Checkpoint. The code is below:

```
# Initializing checkpoint settings to view progress and save model
filename = 'models/intent_classification_b.h5'

# Learning rate scheduling
# This function keeps the initial learning rate for the first ten
epochs
# and decreases it exponentially after that.
def scheduler(epoch, lr):
    if epoch < 10:
        return lr
    else:
        return lr * tf.math.exp(-0.1)

lr_sched_checkpoint =
tf.keras.callbacks.LearningRateScheduler(scheduler)

# Early stopping
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', min_delta=0, patience=3, verbose=0,
mode='auto',
    baseline=None, restore_best_weights=True
)
```

```
# This saves the best model
checkpoint = ModelCheckpoint(filename, monitor='val_loss', verbose=1,
                             save_best_only=True, mode='min')

# The model you get at the end of it is after 100 epochs, but that
might not have been
# the weights most associated with validation accuracy

# Only save the weights when you model has the lowest val loss. Early
stopping

# Fitting model with all the callbacks above
hist = model.fit(padded_X_train, y_train, epochs = 20, batch_size =
32,
                 validation_data = (padded_X_val, y_val),
                 callbacks = [checkpoint, lr_sched_checkpoint,
early_stopping])
```

Entity Extraction

Here is my complete notebook on entity extraction.

For EVE bot, the goal is to extract Apple-specific keywords that fit under the hardware or application category. Like intent classification, there are many ways to do this — each has its benefits depending for the context. Rasa NLU uses a conditional random field (CRF) model, but for this I will use spaCy's implementation of stochastic gradient descent (SGD).

The first step is to create a dictionary that stores the entity categories you think are relevant to your chatbot. Then you see if spaCy has them by default. More likely than not, they won't. So in that case, you would have to train your own custom spaCy Named Entity Recognition (NER) model. For Apple products, it makes sense for the entities to be what hardware and what application the customer is using. You want to respond to customers who are asking about an iPhone differently than customers who are asking about their Macbook Pro.

```
{'hardware': ['macbook pro',
  'iphone',
  'iphones',
  'mac',
  'ipad',
  'watch',
  'TV',
  'airpods'],
 'apps': ['app store',
  'garageband',
  'books',
  'calendar',
  'podcasts',
  'notes',
  'icloud',
  'music',
  'messages',
  'facetime',
  'catalina',
  'maverick']}
```

Once you stored the entity keywords in the dictionary, you should also have a dataset that essentially just uses these keywords in a sentence. Lucky for me, I already have a large Twitter dataset from Kaggle that I have been using. If you feed in these examples and specify which of the words are the entity keywords, you essentially have a labeled dataset, and spaCy can learn the context from which these words are used in a sentence.

In order to label your dataset, you need to convert your data to spaCy format. This is a sample of how my training data should look like to be able to be fed into spaCy for training your custom NER model using Stochastic Gradient Descent (SGD). We make an offsetter and use spaCy's PhraseMatcher, all in the name of making it easier to make it into this format.

```
TRAIN_DATA = [
('what is the price of polo?', {'entities': [(21, 25, 'PrdName')]}),
('what is the price of ball?', {'entities': [(21, 25, 'PrdName')]}),
('what is the price of jegging?', {'entities': [(21, 28,
'PrdName')]}),
('what is the price of t-shirt?', {'entities': [(21, 28,
'PrdName')]})
]
```

Offsetter

```
# Utility function - converts the output of the PhraseMatcher to
something usable in training

def offsetter(lbl, doc, matchitem):
    ''' Converts word position to string position '''
    one = len(str(doc[0:matchitem[1]]))
    subdoc = doc[matchitem[1]:matchitem[2]]
    two = one + len(str(subdoc))

    # This function was misaligned by a factor of one character, not
sure why, but this is my solution
    if one != 0:
        one += 1
        two += 1
    return (one, two, lbl)


# Example
# offsetter('HARDWARE', nlp('hmm macbooks are great'),
(2271554079456360229, 1, 2)) -> (4, 12, 'HARDWARE')
```

I used this function in my more general function to 'spaCify' a row, a function that takes as input the raw row data and converts it to a tagged version of it spaCy can read in. I had to modify the index positioning to shift by one index on the start, I am not sure why but it worked out well.

Then I also made a function `train_spacy` to feed it into spaCy, which uses the `nlp.update` method to train my NER model. It trains it for the arbitrary number of 20 epochs, where at each epoch the training examples are shuffled beforehand. Try not to choose a number of epochs that are too

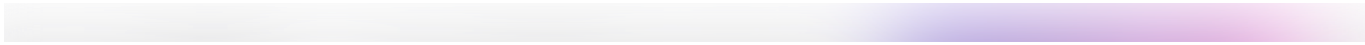
high, otherwise the model might start to ‘forget’ the patterns it has already learned at earlier stages. Since you are minimizing loss with stochastic gradient descent, you can visualize your loss over the epochs.

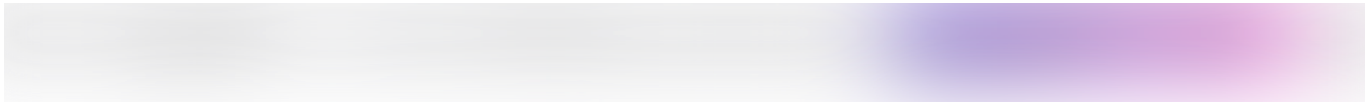
Loss of the hardware entity model at every iteration. Image by Author.

Loss of the app entity model at every iteration. Image by Author.

I did not figure out a way to combine all the different models I trained into a single spaCy pipe object, so I had two separate models serialized into two pickle files. Again, here are the displaCy visualizations I demoed above — it successfully tagged macbook pro and garageband into it's correct entity buckets.

The hardware entity "macbook pro" is tagged. Other hardware might include iPhone and iPads. Image by Author.





The app entity captured “garageband” is tagged. Other app entities might include Apple Music or FaceTime. Image by Author.

From the pickle files you save, you can store all the extracted entities as a list by looping over the ents attribute from the doc object, demonstrated here:

```
def extract_app(user_input, visualize = False):
    # Loading it in
    app_nlp = pickle.load(open("models/app_big_nlp.pkl", "rb"))
    doc = app_nlp(user_input)

    extracted_entities = []

    # These are the objects you can take out
    for ent in doc.ents:
        extracted_entities.append((ent.text, ent.start_char,
ent.end_char, ent.label_))

    return extracted_entities
```

4. Iteration

By iteration, I really just mean improving upon my model. I made it its own stage because this actually takes more time than you may expect. I improve

my model by:

- Choosing better intents and entities
- Improving the quality of my data
- Improving your model architecture

You don't just have to do generate the data the way I did it in step 2. Think of that as one of your toolkits to be able to create your perfect dataset.

> *The goal for the data you feed into your intent classifier is to have each intent be broad ranging (meaning that the intent examples sufficiently exhausts the state space and worlds of what the user might say) and unique from each other.*

That way the neural network is able to make better predictions on user utterances it has never seen before. Here is how I tried to achieve this goal.

Manual Examples

In addition to using Doc2Vec similarity to generate training examples, I also manually added examples in. I started with several examples I can think of, then I looped over these same examples until it meets the 1000 threshold. This makes all the difference in how good your model will be. If you know a

customer is very likely to write something, you should just add it to the training examples.

Keyword Exploration Tuning

How do we choose what intents and examples to include in? To help make a more data informed decision for this, I made a keyword exploration tool that tells you how many Tweets contain that keyword, and gives you a preview of what those Tweets actually are. This is useful to exploring what your customers often ask you and also how to respond to them because we also have outbound data we can take a look at.

Filtering Tweets based on keyword to explore the topics in my data. Image by Author.

I've also made a way to estimate the true distribution of intents or topics in my Twitter data and plot it out. It's quite simple. You start with your intents, then you think of the keywords that represent that intent.

```
{"update":['update'],
"battery": ['battery','power'],
"forgot_password": ['password', 'account', 'login'],
"repair": ['repair', 'fix', 'broken'],
"payment": ['payment'}
```

Image by Author.

I would also encourage you to look at 2, 3, or even 4 combinations of the keywords to see if your data naturally contain Tweets with multiple intents at once. In this following example, you can see that nearly 500 Tweets contain the update, battery, and repair keywords all at once. It's clear that in these Tweets, the customers are looking to fix their battery issue that's potentially caused by their recent update.

Image by Author.

Remember, this is all in the name of making your intent buckets distinct and wide-ranging.

5. Development

As for this development side, this is where you implement business logic that you think suits your context the best. I like to use affirmations like “Did that solve your problem” to reaffirm an intent.

For demo purposes, I used Streamlit. It isn't the ideal place for deploying because it is hard to display conversation history dynamically, but it gets the job done. In reality, you would deploy on one messaging platform. For example, you can use Flask to deploy your chatbot on Facebook Messenger and other platforms. You can also use api.slack.com for integration and can quickly build up your Slack app there.

Conversational interfaces are a whole other topic that has tremendous potential as we go further into the future. And there are many guides out there to knock out your design UX design for these conversational interfaces.

What's Next?

In this article, I essentially show you how to do data generation, intent classification, and entity extraction. These 3 steps are absolutely essential for making a chatbot. However, there is still more to making a chatbot fully functional and feel natural. This mostly lies in how you map the current dialogue state to what actions the chatbot is supposed to take — or in short, *dialogue management*.

The bot needs to learn exactly when to execute actions like to listen and when to ask for essential bits of information if it is needed to answer a

particular intent.

Taking a weather bot as an example, when the user asks about the weather, the bot needs the location to be able to answer that question so that it knows how to make the right API call to retrieve the weather information. So for this specific intent of weather retrieval, it is important to save the location into a slot stored in memory. If the user doesn't mention the location, the bot should ask the user where the user is located. It is unrealistic and inefficient to ask the bot to make API calls for the weather in every city in the world.

I recommend checking out this video and the Rasa documentation to see how Rasa NLU (for Natural Language Understanding) and Rasa Core (for Dialogue Management) modules are used to create an intelligent chatbot. I talk a lot about Rasa because apart from the data generation techniques, I learned my chatbot logic from their masterclass videos and understood it to implement it myself using Python packages. Their framework gives lots of customizability to use different policies and techniques at different stages of the chatbot (such as whether or not to use LSTMs or SVMs for intent classification, to even the more granular details of how to fall back when the bot is not confident in its intent classification).

Also, I would like to use a meta model that controls the dialogue management of my chatbot better. One interesting way is to use a transformer neural network for this (refer to the paper made by Rasa on this, they called it the Transformer Embedding Dialogue Policy). This basically helps you have more natural feeling conversations.

Finally, scaling my chatbot would also be important. This just means expanding the domain of intents and entities that my chatbot would be able to respond to so that it covers the most important areas and edge cases. It's helpful to remember that this framework I have made is transferable to any other chatbot, so I would like to support other languages as well in the future!

Thanks for Reading!

Of course, this is what I learned over approximately the past month from watching NLP lectures, git cloning many Github repos to personally do a hands on dive on how they work, YouTube video scouring, and documentation hunting. So if you have any feedback as for how to improve my chatbot or if there is a better practice compared to my current method,

please do comment or reach out to let me know! I am always striving to make the best product I can deliver and always striving to learn more.

Thanks to Grace and Linda Chen.

Chatbots NLP Named Entity Recognition Editors Pick Hands On Tutorials