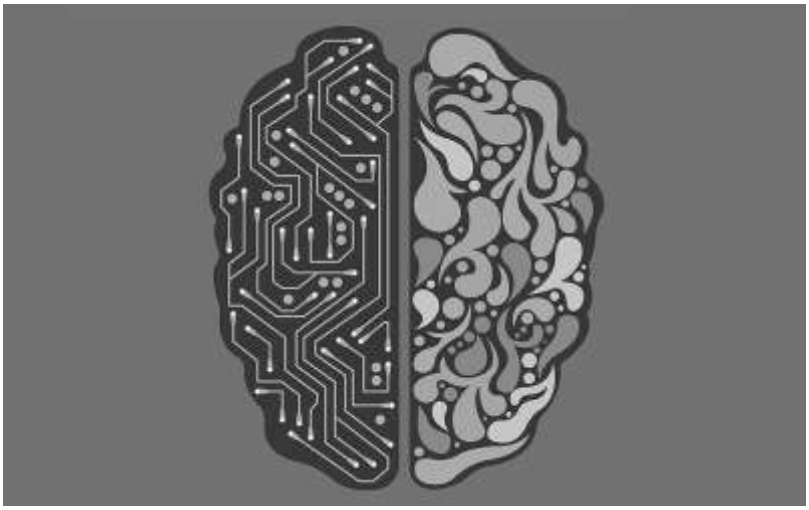
Full Pipeline Project: Python AI for detecting fake news

Johnny Wales Follow
Nov 5 · 22 min read ★



Too many articles on machine learning focus only on modeling. Those crucial middle bits of model building and validation are surely deserving of attention, but I want more — and I hope you do, too. I've written this complete review of my own project, to include data wrangling, the aforementioned model work, and creation of a public interface. If you want to skip right to the "punchline," the finished program can be found at

To find out if this hypothesis is correct, we first need an objectively labeled data set that will give us examples of fake news and examples of real news as provided by professional fact checkers. The data needs to have URLs that point to news articles and a ruling on the truth or fiction of that article. I located 2 data sets that meet this criteria. The first is the interactive media bias chart from Ad Fontes Media. You can access their chart and get the first data set here: https://www.adfontesmedia.com/interactive-media-bias-chart/ The other data set we'll be using is the FakeNewsNet data set available here: https://github.com/KaiDMML/FakeNewsNet. I did some manual changes to the data sets to eliminate PDFs and anything obviously not a URL, and standardized the scoring system by giving a 'fake' rating to all of the items from FakeNewsNet's politifact_fake.csv, and a 'real' rating to all items from the politifact_real.csv file.

Once we have our data sets, our next task will be to combine those data sets into a single database table that we can then use to extract the text we'll be working with. I use the Django ORM for managing a connection to my database. Later, this also makes it easier to provide a public interface to our models, because Django is a full featured MVC web framework. After starting our app via manage.py, I have to create the table via the Django.db models classes:

```
from django.db import models

class ArticleExample(models.Model):
    # This will hold the visible text for this example
    body_text = models.TextField()
    # This bias score is a left-right bias provided by Media Bias
    # Chart, but not used in this project.
    bias_score = models.FloatField()
    bias_class = models.IntegerField()
    # quality_score comes from the Media Bias Chart data
```

Easy, right? Next we go back to Django's manage.py and run makemigrations and migrate to set up our database.

Now we have to get data into this table. This is a process of data wrangling, and is one of the most difficult and time consuming parts of machine learning. In large ML environments, there are data engineers who do little else than getting data sets together.

Let's dig right into this problem. Essentially, we need to:

1. Load up our list of URLs and scores.
2. Load the page from the URL and parse it
3. Store the parsed version along with the score
4. Add a class for any data point that doesn't have one by splitting up the data evenly.

First lets look at everything except the part about parsing the page. For now, you just need to know that SoupStrainer (which we'll develop in a minute) will handle all that for us.

The main part of this program is harvester.py. In the first section of harvester, we have to do some initial setup to allow our django program to run from the command line rather than via a web interface.

```
import os, sys, re, time

proj_path = "/home/jwales/eclipse-workspace/crowdnews/"
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "crowdnews.settings")
sys.path.append(proj_path)
```

```
import pandas as pd
from newsbot.strainer import *
from newsbot.models import *

ss = SoupStrainer()
print("Initializing dictionary…")
ss.init()
```

Next we have the following method for loading the data we got from Politifact:

```
def harvest_Politifact_data():
    print("Ready to harvest Politifact data.")
    input("[Enter to continue, Ctl+C to cancel]>>")
    print("Reading URLs file")

    # Read the data file into a pandas dataframe
    df_csv = pd.read_csv("newsbot/politifact_data.csv",
    error_bad_lines=False, quotechar='"', thousands=',',
    low_memory=False)

    for index, row in df_csv.iterrows():
        print("Attempting URL: " + row['news_url'])
        if(ss.loadAddress(row['news_url'])):
            print("Loaded OK")
    # some of this data loads 404 pages b/c it is a little old,
    # some load login pages. I've found that
    # ignoring anything under 500 characters is a decent
    # strategy for weeding those out.
            if(len(ss.extractText)>500):
                ae = ArticleExample()
                ae.body_text = ss.extractText
                ae.origin_url = row['news_url']
                ae.origin_source = 'politifact data'
                ae.bias_score = 0 # Politifact data doesn't have this
                ae.bias_class = 5 # 5 is 'no data'
                ae.quality_score = row['score']
```

Going through this, you'll see we first used Pandas to read in the CSV file with our URLs and scores. Then we send it off to our parser and save the resulting text in body_text. Then we nap for 1 second to be kind to the websites we're harvesting from. The process for the Media Bias Chart data is similar, we just have to split up the quality_class by each 1/4 of the quality score from media bias chart.

The real meat of the work isn't here yet, though! How are we getting the data from these websites, what does it look like, and how are we parsing it?

For this task, we will need to limit the number of words we're looking at. We want to be sure those words are real English words and we want to only save the word stem, so words like programmer, programming, and program can all be reduced to program. This will limit the size of our eventual training examples and make it manageable on an ordinary PC. This task will be in the class SoupStrainer we encountered in harvester, and I told you we'd explain in a minute. The big moment has arrived!

First is our set of imports. We'll use BeautifulSoup for parsing the HTML, urllib3 for loading the web pages from the net, and PorterStemmer to do the word stemming. Plus we'll need a couple of other things tossed in for wrangling this data:

```
import urllib3, re, string, json, html
from bs4 import BeautifulSoup
from bs4.element import Comment
from urllib3.exceptions import HTTPError
from io import StringIO
from nltk.stem import PorterStemmer
```

Next we'll set up our class and initialize our dictionary from a full dictionary you can get

```
    soup = None
    msgOutput = True

    def init(self):
        with open('newsbot/words_dictionary.json') as json_file:
            self.englishDictionary = json.load(json_file)
```

We are only interested in visible text on the page, so the next step is to build a filter that is capable of detecting which tags are visible and which ones are not:

```
def tag_visible(self, element):
    if element.parent.name in ['style', 'script',
            'head', 'title', 'meta', '[document]']:
        return False
    if isinstance(element, Comment):
        return False
    return True
```

We will use this in our next function, which does the actual loading and parsing. Buckle in, this one is a fun ride.

First, we'll set some things up and be sure it looks at least somewhat like a URL:

```
def loadAddress(self, address):
    self.locToGet = address
    self.haveHeadline = False

    htmatch = re.compile('.*http.*')
    user_agent = {'user-agent': 'Mozilla/5.0 (Windows NT 6.3;
                  rv:36.0) Gecko/20100101 Firefox/36.0'}
    ps = PorterStemmer()
```

```
            "which was derived from " + address)
    try:
        urllib3.disable_warnings(
                urllib3.exceptions.InsecureRequestWarning)
        http = urllib3.PoolManager(2, headers=user_agent)
        r = http.request('GET', self.locToGet)
        self.pageData = r.data
        if(self.msgOutput):
            print("Page data loaded OK")
    except:
        self.errMsg = 'Error on HTTP request'
        if(self.msgOutput):
            print("Problem loading the page")
        return False
```

So far so good, we're just loading up the web page and grabbing the resulting HTML file. Next, we should extract the visible text, strip it of punctuation, ensure it is a real word, then stem it. I left in a commented line showing how you might want to review any words that aren't in the dictionary.

```
self.extractText = ''
self.recHeadline = self.locToGet
self.soup = BeautifulSoup(self.pageData, 'html.parser')
ttexts = self.soup.findAll(text=True)
viz_text = filter(self.tag_visible, ttexts)
allVisText = u"".join(t.strip() for t in viz_text)

for word in allVisText.split():
    canonWord = word.lower()
    canonWord = canonWord.translate(
            str.maketrans('', '', string.punctuation))
    canonWord = canonWord.strip(string.punctuation)
    if(canonWord in self.englishDictionary):
        canonWord = ps.stem(canonWord)
        self.extractText = self.extractText + canonWord + " "
```

appears in the example. Therefore, the next step we need to go through is attaching a unique index for each word in all examples that we've just downloaded and put into our database. So we will need to build a dictionary. The first step there is to build the data models and tables needed for this task. Head back over to models.py, and add the following:

```
class DictEntry(models.Model):
 canonWord = models.TextField()
```

Go out to your terminal and do the makemigrations/migrate dance, and then we are ready to work on building out the dictionary. For that, we need to take a look at the script dictbuilder.py.

Once we have loaded the required django libraries, we build a python dictionary using any words currently in our canonical words dictionary table. This gives us a fast way to see if something is already in the dictionary without having to do a database lookup. If you do this by hitting the database for each individual word test, you will slow down by a factor of at least 10, so this part is definitely worth it. As it happens, we'll be doing this repeatedly throughout the process, so here we will create a file util.py and create a function to return our dictionary of canonical words:

```
def loadCanonDict():
    canonDict = DictEntry.objects.all()
    dictSize = canonDict.count() + 1
    cDict = {}
    for cw in canonDict:
        cDict[cw.canonWord] = cw.pk
```

```
        cwords = ex.body_text.split()
        for cwrd in cwords:
            if(cwrd in cDict.keys()):
                print('.', end='', flush=True)
            else:
                print('X', end='', flush=True)
                nde = DictEntry(canonWord = cwrd)
                nde.save()
                cDict[cwrd] = nde.pk
```

This will assign a primary key ID to each word, and those are going to be numbered sequentially. This way, when we are building an example, we can use the primary key as the column number to update for any particular example, and be sure the same column means the same thing for every example. Note that if you ever need to blow away the dictionary and rebuild, you'll also need to go reset the primary key counter. I had to do that during development, maybe this article will save you that headache!

Stepping back, we now have a data set to work with and we have a way to indicate which words exist in any particular example. We have (in my dataset) 20,870 words and 2,500 unique examples with scores. Now we are ready to begin learning our models. If you're the kind of nerd I am, you're absolutely giddy at this point.

The problem we face is to classify the news into 1 of 4 possible classes, Fake, Dodgy, Seems Legit, and Real. This is a problem for a classifier model. The library of models we're using is scikit-learn. We'll test several models and see which works well. I think this type of problem would be great for a support vector machine or a neural network.

For each article, we now have a copy of the text as stripped-down and stemmed words. Each of those words has a unique ID. We will build each example as a numpy row vector, with each example[n] will be 1 if word ID n appears in the article, and 0 otherwise. Then

```
def buildExampleRow(body_text, cDict):
    dictSize = len(cDict.keys())
    one_ex_vector = np.zeros(dictSize+2)
    cwords = body_text.split()
    for word in cwords:
        if(word in cDict.keys()):
            one_ex_vector[cDict[word]-1] = 1
        else:
            print("This word doesn't exist in the dict:" + word)

    return(one_ex_vector)
```

Next we build a function to load up our examples from the database.

```
def processExamples(qs_Examples, cDict):
    Y_vector = np.zeros(qs_Examples.count(), dtype=np.int8)
    Y_vec_count = 0
    examplesMatrix = None

    for ex in qs_Examples:
        Y_vector[Y_vec_count] = int(ex.quality_class)
        Y_vec_count = Y_vec_count + 1
        if(examplesMatrix is None):
            examplesMatrix = buildExampleRow(ex.body_text, cDict)
        else:
            examplesMatrix = np.vstack(
                [examplesMatrix,
                buildExampleRow(ex.body_text, cDict)])
            print('.', end='', flush=True)

    return( (Y_vector, examplesMatrix))
```

Now we can set up for training quickly and easily in class_learner.py:

It is now time to train and test models. We will break our data into a training and testing set, then we will train our multi-layer perceptron classifier. Easy peasy, as it turns out:

```
X_train, X_test, y_train, y_test = train_test_split(examplesMatrix,
Y_vector, test_size=0.2)

model = MLPClassifier(hidden_layer_sizes=(128,64,32,16,8),
max_iter=2500)

model.fit(X_train, y_train)
```

The great thing about using a powerful library like SciKit is that the basics of this process are the same for several different models. When you want to change models, you can change the line model=MLPClassifier(…) to another model and you will often get a working program.

Now we have a trained model, and we finally get around to finding out if it worked and if we can build something cool from it. To find out, we'll need to do some tests and understand them. Thus begins the process of model validation. For a classification model, we can get a great idea of how we're doing at classifying our news.

```
print("***************")
print("Classification based:")
print(accuracy_score(predictions, y_test))
print(confusion_matrix(predictions, y_test))
print(classification_report(predictions, y_test))
print("***************")
```

```
1            0.51      0.73      0.60        48
             2         0.46      0.51        0.48        87
             3         0.52      0.41        0.46       202
             4         0.63      0.68        0.65       163

accuracy                                0.55       500
   macro avg         0.53      0.58        0.55       500
weighted avg         0.54      0.55        0.54       500

***************
```

Interpreting these numbers, we get:

Accuracy Score, 0.546: This means 54.6% of our predictions were accurate. Since we have 4 classes, a random guess would have produced an accuracy of 25%, so this is actually an improvement.

Confusion matrix: This set of 16 numbers gives us some very interesting information. I have plotted it below in a more readable format. The rows are each for a class predicted class, and each column has the true class. The number at the intersection of the row and column is the number of the real class predicted as each predicted class. So, for instance, the number 44 that you see on the second row, second column is the number of examples where the true class was 2, and the predicted value was also 2. The 28 you see in the next column of the same row is the number of examples that were actually a 2 but our model predicted a 3. Notice that among articles that are actually a 1, there is strong weighting toward the 1/2 end of the scale. Similarly, on the bottom row we have examples that are actually graded a 4, and notice that the last 2 columns are overwhelmingly bigger than the first two. This is good news!

Classification Report:

higher F1 on another model, we should choose it over this one. Accuracy score is also a good single metric, and in practice the two are usually very close.

Earlier, I noted in the confusion matrix that it seems like the classifier is often off by 1. If I also look at the mean absolute error of my model:

```
mae = mean_absolute_error(y_test, predictions)
print("Mean absolute Error: " + str(mae))
```

I will get:

```
Mean absolute Error: 0.54
```

Which is further evidence that, on average, the model is within 1 class of the actual answer. This will inform our further development of our models and our end delivery. If we are going to deliver this model to the end user to give them a clear picture of what our model is thinking, we will need to produce probability estimates. Most models can handle that automatically, the support vector classifier requires the "probability=True" flag to produce a probability estimator when training.

Now that you know how to read a confusion matrix and a classification report, and we've picked out our single metric for model evaluation, we can now test out a whole bunch of different models. In the code available in the repository, I've just commented out calls for all the various models shown here so you can see that scikit allows you to switch learning models by simply changing the model declaration, and everything else remains

Well, I'm pretty surprised that logistic regression took the 2nd spot. Not so surprised about K-Neighbors and Decision Tree, they didn't seem like a good fit for this problem. SVC was the clear winner.

Thinking about our interface

Now we have a clear picture of how well or poorly our models perform. Next we need to see how they perform on live examples. We know that the models tend to be close to correct even if they aren't correct. We next need to know more about whether or not they provide conflicting answers or if they are giving us consistent answers. We'll have to save our top 3 models and build something to test them.

Saving our models

For this part of the task, I created a new file called class_saver.py. This way, I can load the data 1 time, run and test all three top models, and save them using pickle. We'll set up our environment and then put our top three models into an iterable python dictionary:

```
print("Setting up..")
cDict = loadCanonDict()
qs_Examples = ArticleExample.objects.filter(quality_class__lt = 5)
print("Processing examples")
```

Next we will cycle through the chosen models, train them, and decide if we want to keep them. Note that these models have random initializations and running them more than once will produce slightly different results, so you need to take a peek at their statistics to be sure you've got a good one.

```
for fname, model in chosen_models.items():
    print("Working on " + fname)
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    print("Classification report: ")
    print(classification_report(predictions, y_test))
    print("***************")
    dosave = input("Save " + fname + "? ")
    if(dosave == 'y' or dosave == 'Y'):
        print("Saving...")
        pickle.dump(model, open(fname, 'wb'))
        print("Saved!")
    else:
        print("Not saved!")
```

After training your models and saving them, you'll wind up with 3 files. My 3 files were very large, total of about 391MB together.

Now, let's build a command-line interface that will give us our estimates on live examples. Here we can re-use a lot of code. If you're following along in the repository, we're now going to work with classify_news.py.

After we do our usual setup of adding in the required imports and setting up Django for our ORM access, we next need to load up our 3 models designed before.

Next we need to initialize everything for turning an article into an example, as we did before:

```
print("Initializing dictionaries...")
cDict = loadCanonDict()
ss = SoupStrainer()
ss.init()
```

Then we turn our article into an input row for our models:

```
url = input("URL to analyze: ")

print("Attempting URL: " + url)
if(ss.loadAddress(url)):
    articleX = buildExampleRow(ss.extractText, cDict)
else:
    print("Error on URL, exiting")
    exit(0)

articleX = articleX.reshape(1, -1)
```

And finally, when we have a proper row vector set up, we can predict and generate probabilities for each of our models:

```
log_prediction = log_model.predict(articleX)
log_probabilities = log_model.predict_proba(articleX)

svc_prediction = svc_model.predict(articleX)
svc_probabilities = svc_model.predict_proba(articleX)
```

```
print(log_prediction)
print("Probabilities:")
print(log_probabilities)

print("*** MLP ")
print("Prediction on this article is: ")
print(mlp_prediction)
print("Probabilities:")
print(mlp_probabilities)
```

Now I run the classifier. It takes about 1 to 2 seconds load time to unpickle my models and load up dictionaries, which is a little long but still acceptable. If we are rolling this out at scale, we need to find a way to load the models and dictionaries in memory and have them persist there until removed. For our development version, this will still work. If I paste in an article URL I think is real, I get the following output:

```
*** SVC
Prediction on this article is:
[3]
Probabilities:
[[0.01111608 0.0503078 0.70502378 0.23355233]]
*** Logistic
Prediction on this article is:
[3]
Probabilities:
[[5.61033543e-04 5.89780773e-03 7.63196217e-01 2.30344942e-01]]
*** MLP
Prediction on this article is:
[4]
Probabilities:
[[1.18020372e-04 1.93965844e-09 4.88694225e-01 5.11187753e-01]]
```

These probabilities are showing me the probabilities associated with each of my 4

```
*** SVC
Prediction on this article is:
[1]
Probabilities:
[[0.80220529 0.18501285 0.01051862 0.00226324]]
*** Logistic
Prediction on this article is:
[1]
Probabilities:
[[0.53989611 0.45269964 0.0005857 0.00681855]]
*** MLP
Prediction on this article is:
[1]
Probabilities:
[[8.38376936e-01 1.84104358e-03 1.59391877e-01 3.90143317e-04]]
```

Eureka! All three classifiers agree, this article is fake. The probabilities are interesting here. the SVC and Logistic models are very sure on this being on the fake/dodgy end of things. Notice that the MLP gives a 15.9% chance of the article being mostly true.

Finally, I will test an article sent to me by one of my readers who I invited to be a beta tester. He found that it was producing inconsistent results in an earlier version of this project, specifically that the first version of my SVC and MLP models were disagreeing. In this version, it also comes up very controversial:

```
*** SVC
Prediction on this article is:
[3]
Probabilities:
[[0.28233338 0.38757642 0.26880828 0.06128192]]
*** Logistic
Prediction on this article is:
[3]
```

the highest probability. The same basic outcome is seen in the logistic model: Somewhat similar probabilities for 1 and 3, plus a 16.8% chance of 2, and it ultimately predicted a 3. The MLP model is sure this article is a 3. Intriguingly, this article is about the economy, but it is from a dodgy source and uses a lot of very loaded language similar to fake news. I think the classifiers have a hard time because it is about a topic outside the training domain. The articles used in training were generally about US politics rather than about the economy, so that may be why we're seeing this outcome.

Building the web interface

Whew. It has been a long journey, folks. Now we have our 3 models that seem to work pretty well, and we want to deploy them so the public can use them. We know that in many cases where an article should be in class 4, our models usually put most of their weight into class 3 and 4, and that they work similarly on the other end of the scale. We know they sometimes disagree, especially on the finer point of the specific probabilities involved. And we know that the typical user isn't going to install python and a huge batch of libraries to use this tool. We have to make it easy to use, easy to understand, and available to everyone.

Great news, we originally put this puppy together in the Django MVC framework, so from here it is a relatively simple process to build out a web interface.

First, we update the views.py provided by django when we used startapp to start up this whole process.

In this part, we see some familiar code we've worked with before in our command-line interface to set up the models, parse an example, and return our row vector representing the word content of the submitted article:

```
def index(request):

    url = request.GET.get('u')
    if((url is not None) and (len(url) > 5)):
        print("Setting up")
        svc_model = pickle.load(open('newsbot/svc_model.sav', 'rb'))
        mlp_model = pickle.load(open('newsbot/MLPC_model.sav', 'rb'))
        log_model = pickle.load(open('newsbot/log_model.sav', 'rb'))
        cDict = loadCanonDict()
        ss = SoupStrainer()
        ss.init()
        print("Setup complete")
        print("Attempting URL: " + url)
        if(ss.loadAddress(url)):
            articleX = buildExampleRow(ss.extractText, cDict)
        else:
            print("Error on URL, exiting")
            return render(request, 'urlFail.html', {'URL', url})

        articleX = articleX.reshape(1, -1)
```

Shazam, we've got articleX as our article in a numpy array of 1s and 0s that represents the canonical words that appear in the article.

Now we need to get a result from all our models:

```
svc_prediction = svc_model.predict(articleX)
svc_probabilities = svc_model.predict_proba(articleX)

mlp_prediction = mlp_model.predict(articleX)
mlp_probabilities = mlp_model.predict_proba(articleX)

log_prediction = log_model.predict(articleX)
log_probabilities = log_model.predict_proba(articleX)
```

This gives us just 1 comparison to do in our template to decide if we want to display "Seems Real" or "Seems Fake" as our ruling.

```
svc_prb = (svc_probabilities[0][0]*100, svc_probabilities[0][1]*100,
svc_probabilities[0][2]*100, svc_probabilities[0][3]*100)
 svc_totFake = (svc_probabilities[0][0]*100) + (svc_probabilities[0]
[1]*100)
 svc_totReal = (svc_probabilities[0][2]*100) + (svc_probabilities[0]
[3]*100)

 mlp_prb = (mlp_probabilities[0][0]*100, mlp_probabilities[0][1]*100,
mlp_probabilities[0][2]*100, mlp_probabilities[0][3]*100)
 mlp_totFake = (mlp_probabilities[0][0]*100) + (mlp_probabilities[0]
[1]*100)
 mlp_totReal = (mlp_probabilities[0][2]*100) + (mlp_probabilities[0]
[3]*100)

 log_prb = (log_probabilities[0][0]*100, log_probabilities[0][1]*100,
log_probabilities[0][2]*100, log_probabilities[0][3]*100)
 log_totFake = (log_probabilities[0][0]*100) + (log_probabilities[0]
[1]*100)
 log_totReal = (log_probabilities[0][2]*100) + (log_probabilities[0]
[3]*100)
```

Then we want to combine these three models. This will handle the cases of controversial rulings. If 2 of our 3 models are strongly leaning one way, and the other is leaning strongly the other, then averaging them together will settle the dispute. That gives us a probability distribution that we can use as the top-of-page, single prediction that our end user wants: Is it real or fake?

```
fin_prb = ( (((svc_probabilities[0][0]*100)+(mlp_probabilities[0]
[0]*100)+(log_probabilities[0][0]*100))/3)
```

```
context = {'headline':ss.recHeadline, 'words': ss.extractText, 'url'
: url,
 'svc_totFake': svc_totFake,
 'svc_totReal': svc_totReal,
 'svc_prediction': svc_prediction,
 'svc_probabilities': svc_prb,
 'mlp_totFake': mlp_totFake,
 'mlp_totReal': mlp_totReal,
 'mlp_prediction': mlp_prediction,
 'mlp_probabilities': mlp_prb,
 'log_totFake': log_totFake,
 'log_totReal': log_totReal,
 'log_prediction': log_prediction,
 'log_probabilities': log_prb,
 'fin_totFake': fin_totFake,
 'fin_totReal': fin_totReal,
 'fin_probabilities': fin_prb
 }

 return render(request, 'newsbot/results.html', context)
```

And you'll need to add this bit at the end to render the form asking the user to enter a URL if one isn't provided:

```
else:
 return render(request, 'newsbot/urlForm.html')
```

Within the results.html, we just have to create a quick table with each outcome. Here's the one for the final ruling table:

```
<h3 style="text-align: center;">
```

```
 <div class="progress-bar" id="fakeProb_bar" role="progressbar" aria-
valuenow="{{ fin_probabilities.0 }}"
 aria-valuemin="0" aria-valuemax="100" style="width:{{
fin_probabilities.0 }}%"></div>
</div>

 <br>

Probability of Dodgy: {{ fin_probabilities.1|floatformat }}% chance
of Dodgy
<div class="progress">
 <div class="progress-bar" id="MfakeProb_bar" role="progressbar"
aria-valuenow="{{ fin_probabilities.1 }}"
 aria-valuemin="0" aria-valuemax="100" style="width:{{
fin_probabilities.1 }}%"></div>
</div>

 <br>

Probability of Mostly True: {{ fin_probabilities.2|floatformat }}%
chance of Mostly True
<div class="progress">
 <div class="progress-bar" id="MtrueProb_bar" role="progressbar"
aria-valuenow="{{ fin_probabilities.2 }}"
 aria-valuemin="0" aria-valuemax="100" style="width:{{
fin_probabilities.2 }}%"></div>
</div>

 <br>

Probability of True: {{ fin_probabilities.3|floatformat }}% chance of
True
<div class="progress">
 <div class="progress-bar" id="trueProb_bar" role="progressbar" aria-
valuenow="{{ fin_probabilities.3 }}"
 aria-valuemin="0" aria-valuemax="100" style="width:{{
fin_probabilities.3 }}%"></div>
</div>
```

machine. And then we tested it against live data and, much to the amazement of everyone, found that it actually has a sensible answer. Finishing a long project like this gives you a complete picture of the entire machine learning pipeline. This project, long as it was, still has a lot of room for improvements. More ideas include:

Ask the user if they think the bot was right or wrong, and record their answers for possible future training.

Find more data sets or get more data sets from more sources.

Write a crawler to dig through fake news sites gathering more and more examples.

Tune hyperparameters.

Expand domain knowledge to other countries or other subjects.

Use the domain name or headline as a data feature.

Find a way to count the number of advertisements on the page, on the hypothesis that fake news sites usually have more ads.

Now that you've seen the kind of things a classifier can potentially do, especially with a little help interpreting results, the sky is the limit. Thanks for coming, I hope I helped inspire and educate!

)

Machine Learning Python Scikit Learn Naturallanguageprocessing