Building NLP Classifiers Cheaply With Transfer Learning and Weak Supervision

A Step-by-Step Guide for Building an Anti-Semitic Tweet Classifier

Abraham Starosta

Text + Intelligence = Gold… But, how can we mine it cheaply?

Introduction

There is a catch to training state-of-the-art NLP models: their reliance on massive hand-labeled training sets. That's why data labeling is usually the bottleneck in developing NLP applications and keeping them up-to-date. For example, imagine how much it would cost to pay medical specialists to label thousands of electronic health records. In general, having domain experts label thousands of examples is too expensive.

On top of the initial labeling cost, there is another huge cost in keeping models up-to-date with changing contexts in the real-world. Louise Matsakis on Wired explains that the main reasons social media platforms find it so hard to detect hate speech are "shifting contexts, changing circumstances, and disagreements between people." That's mainly because when context changes, we usually have to label thousands of new examples or relabel a big part of our dataset. Again, this is very expensive.

This is an important problem to solve if we want to automate knowledge acquisition from text data, but it's also very hard to solve. Thankfully, new technologies like transfer learning, multitask learning and weak supervision are pushing NLP forward and might be finally converging to offer solutions. **In this blog, I'll walk you through a personal project in which I cheaply built a classifier to detect anti-semitic tweets, with no public dataset available, by combining weak supervision and transfer learning. I**

hope that by the end you'll be able to follow this approach to build your own classifiers relatively cheaply.

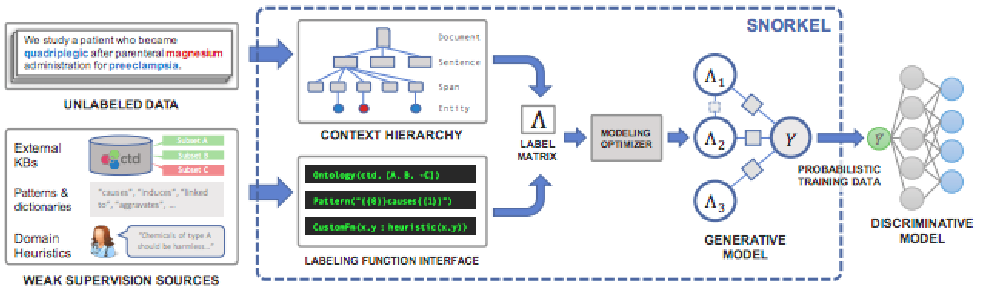
We have 3 steps:

1. Collect a small number of labeled examples (~600)
2. Use weak supervision to build a training set from many unlabeled examples using weak supervision
3. Use a large pre-trained language model for transfer learning

Background

Weak Supervision

Weak supervision (WS) helps us alleviate the **data bottleneck** problem by enabling us to cheaply leverage subject matter expertise to programmatically label millions of data points. More specifically, it's a framework that helps subject matter experts (SMEs) infuse their knowledge into an AI system in the form of hand-written heuristic rules or distant supervision. As an example of WS adding value in the real-world, Google just published a paper in December 2018 describing Snorkel DryBell, an internal tool they built to use WS to build 3 powerful text classifiers in a fraction of the time.



We study a patient who became quadriplegic after parenteral magnesium administration for preeclampsia.
UNLABELED DATA
External KBs
ctd
Subset A
Subset B
Subset C
Patterns & dictionaries
"causes", "induces", "linked to", "aggravates", ...
Domain Heuristics
"Chemicals of type A should be harmless..."
WEAK SUPERVISION SOURCES
SNORKEL
Document
Sentence
Span
Entity
CONTEXT HIERARCHY
Ontology(ctd, [A, B, -C])
Pattern("{{0}}causes{{1}}")
CustomFn(x,y : heuristic(x,y))
LABELING FUNCTION INTERFACE
Λ
LABEL MATRIX
MODELING OPTIMIZER
Λ_1
Λ_2
Λ_3
Y
GENERATIVE MODEL
PROBABILISTIC TRAINING DATA
DISCRIMINATIVE MODEL

Overview of the Data Programming Paradigm with Snorkel

Throughout this project, I used the same general approach Google took: Snorkel (Ratner et al., 2018.) The Stanford Infolab implements the Snorkel framework in this handy Python package called Snorkel Metal. I suggest you go through this tutorial to learn the basic workflow and this one to learn how to get the most out of it.

In Snorkel, the heuristics are called ***Labeling Functions (LFs).*** Here are some common types of LFs:

- **Hard-coded heuristics**: usually regular expressions (regexes)
- **Syntactics:** for instance, Spacy's dependency trees
- **Distant supervision:** external knowledge bases
- **Noisy manual labels:** crowdsourcing
- **External models:** other models with useful signals

After you write your LFs, Snorkel will train a **Label Model** that takes advantage of conflicts between all LFs to estimate their accuracy. Then, when labeling a new data point, each LF will cast a vote: positive, negative, or abstain. Based on these votes and point, each LF will cast a vote: positive, negative, or abstain. Based on those votes and the LF accuracy estimates, the Label Model can programmatically assign **probabilistic labels** to millions of data points. Finally, the goal is to train a classifier that can **generalize beyond our LFs**.

For example, below is the code for an LF I wrote to detect anti-semitic tweets. This LF focuses on catching common conspiracy theories that depict wealthy jews as controlling the media and politics.

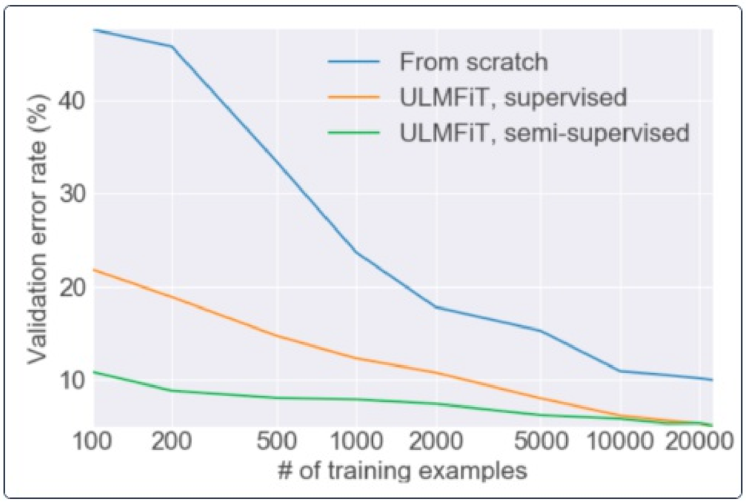
```
# Set voting values.
ABSTAIN = 0
POSITIVE = 1
NEGATIVE = 2

# Detects common conspiracy theories about jews owning the world.
GLOBALISM = r"\b(Soros|Adelson|Rothschild|Khazar)"

def jews_symbols_of_globalism(tweet_text):
    return POSITIVE if re.search(GLOBALISM, tweet_text) else ABSTAIN
```

Key advantages of Weak Supervision:

- **Flexibility:** when we want to update our model, all we need to do is update the LFs, rebuild our training set and retrain our classifier.
- **Improvement in recall:** a discriminative model will be able to generalize beyond the rules in our WS model, thus often giving us a bump in recall.

Transfer Learning and ULMFiT

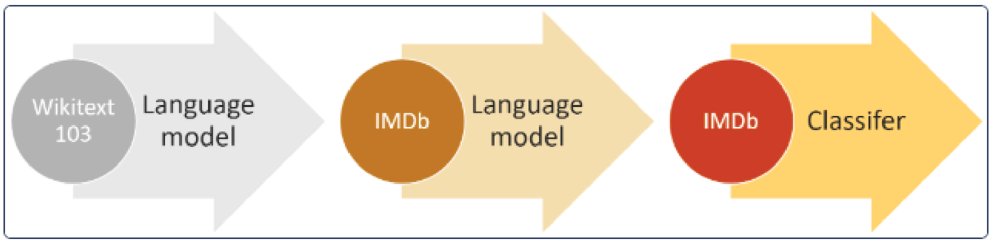
Transfer Learning has greatly impacted computer vision. Using a pre-trained ConvNet on ImageNet as initialization or fine-tuning it to your task at hand has become very common. But, that hadn't translated into NLP until ULMFiT came about.



ULMFiT requires orders of magnitude less data than previous approaches. (Figure 3 from the paper)

ULMFiT

Similar to how computer vision engineers use ConvNets pre-trained on ImageNet, Fastai provides a Universal Language Model, pre-trained on millions of Wikipedia pages, which we can fine-tune to our specific domain space. Then, we can train a text classification model that leverages the LM's learned text representations which can learn with very few examples (up to 100x less data).



High level ULMFiT approach (IMDb example)

Introduction to ULMFiT

ULMFiT consists of 3 stages:

1. Pre-trained an LM on a general purpose corpus (Wikipedia)
2. Fine-tune the LM for the task at hand with a large corpus of unlabeled data points
3. Train a discriminative classifier by fine-tuning it with gradual unfreezing

Key advantages of ULMFiT:

- With only 100 labels, it can match the performance of training from scratch on 100x more data
- Fastai's API is very easy to use. This tutorial is very good
- Produces a Pytorch model we can deploy in production

Step-By-Step Guide for Building an Anti-Semitic Tweet Classifier

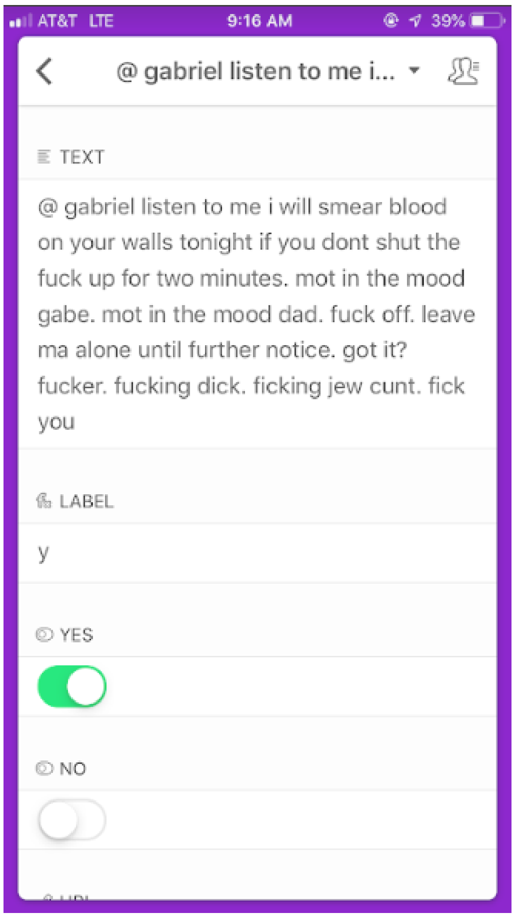
In this section, we'll dive more deeply into the steps I took to build an anti-semitic tweet classifier and I'll share some more general things I learned throughout this process.

First step: Data Collection and Setting a Target

Collecting unlabeled data: The first step is to put together a large set of unlabeled examples (at least 20,000). For the anti-semitic tweet classifier, I downloaded close to 25,000 tweets that mention the word "jew."

Label 600 examples: 600 isn't a large number but I consider this is a good place to start for most tasks because we'll have about 200 examples in each data split. If you already have the labeled examples, then use those! Otherwise, just pick 600 examples at random and label them.

As a labeling tool, you can use Google Sheets. Or, if you're like me and would rather label on your phone, you can use **Airtable.** Airtable is free and it has a slick iPhone app. If you're working in teams, it also lets you easily split the work. I'll probably write another blog focusing on how to use Airtable for labeling. You can just label as you scroll through examples. If you're curious about how I set up Airtable for labeling, feel free to reach out to me.


AT&T LTE 9:16 AM 39%

@ gabriel listen to me i...

TEXT

@ gabriel listen to me i will smear blood on your walls tonight if you dont shut the fuck up for two minutes. mot in the mood gabe. mot in the mood dad. fuck off. leave ma alone until further notice. got it? fucker. fucking dick. ficking jew cunt. fick you

LABEL

y

YES

NO

View of Airtable for Text Labeling

Split data: for the purpose of this project, we'll have a train, test and an LF set. The purpose of the LF set is to both help validate our LFs and to get ideas for new LFs. The test set is used to check the performance of your models (we can't look at it). If you want to do hyperparameter tuning, you would want to label about 200 more samples for a validation set.

I have 24,738 unlabeled tweets (train set), 733 labeled tweets for building LFs, and 438 labeled tweets for testing. So I labeled a total of 1,171, but I realized that was probably too many.

```
DATA_PATH = "../data"
train = pd.read_csv(os.path.join(DATA_PATH, "train_tweets.csv"))
test = pd.read_csv(os.path.join(DATA_PATH, "test_tweets.csv"))
LF_set = pd.read_csv(os.path.join(DATA_PATH, "LF_tweets.csv"))
train.shape, LF_set.shape, test.shape

>> ((24738, 6), (733, 7), (438, 7))
```

Setting our goals: after labeling a few hundred examples you'll have a better idea of how hard the task is and you'll be able to set a goal for yourself. I considered that for anti-semitism classification, it was very important to have high precision so I set myself the goal of getting at least 90% precision and 30% recall.

Second Step: Building a Training Set With Snorkel

Building our Labeling Functions is a pretty hands-on stage, but it will pay off! I expect that if you already have domain knowledge, this should take about a day (and if you don't then it might take a couple days.) Also, **this section is a mix of what I did for my project specifically and some general advice** of how to use Snorkel that you can apply to your own projects.

Since most people haven't used Weak Supervision with Snorkel before, I'll try to explain the approach I took in as much detail as possible. This tutorial is a good way to understand the main ideas, but reading through my workflow will hopefully save you a lot of time of trial and error.

Below is an example of a LF that returns **Positive** if the tweet has one of the common insults against jew. Otherwise, it abstains.

```
# Common insults against jews.
INSULTS = r"\bjew (bitch|shit|crow|fuck|rat|cockroach|ass|bast(a|e)rd)"

def insults(tweet_text):
    return POSITIVE if re.search(INSULTS, tweet_text) else ABSTAIN
```

Here's an example of a LF that returns **Negative** if the tweet's author mentions he or she is jewish, which commonly means the tweet is not anti-semitic.

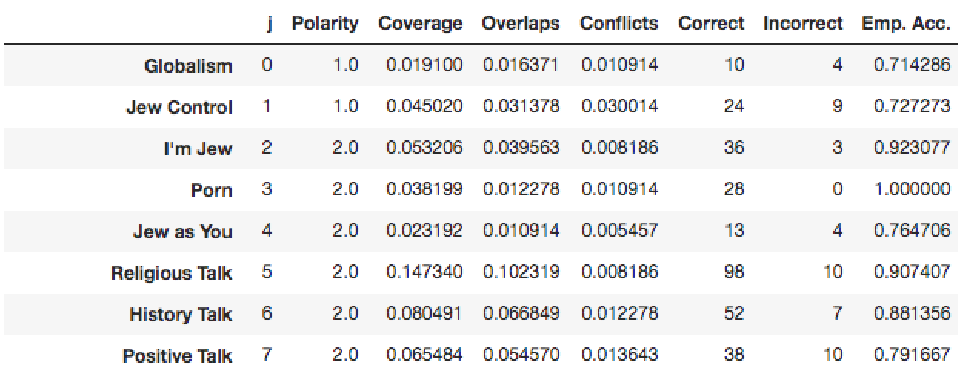
```
# If tweet author is jewish then it's likely not anti-semitic.
JEWISH_AUTHOR = r"((\bI am jew)|(\bas a jew)|(\bborn a jew)"

def jewish_author(tweet_tweet):
    return NEGATIVE if re.search(JEWISH_AUTHOR, tweet_tweet) else ABSTAIN
```

When designing LFs it's important to keep in mind that **we are prioritizing high precision over recall**. Our hope is that the classifier will pick up more patterns, increasing recall. But, don't worry if LFs don't have super high precision or high recall, Snorkel will take care of it.

Once you have some LFs, you just need to build a matrix with a tweet in each row and the LF values in the columns. Snorkel Metal has a very handy util function to display a summary of your LFs.

```
# We build a matrix of LF votes for each tweet
LF_matrix = make_Ls_matrix(LF_set, LFs)

# Get true labels for LF set
Y_LF_set = np.array(LF_set['label'])

display(lf_summary(sparse.csr_matrix(LF_matrix),
                   Y=Y_LF_set,
                   lf_names=LF_names.values()))
```

I have a total of 24 LFs, but here's how the LF summary looks like for a sample of my LFs. Below the table you can find what each column means.



	j	Polarity	Coverage	Overlaps	Conflicts	Correct	Incorrect	Emp. Acc.
Globalism	0	1.0	0.019100	0.016371	0.010914	10	4	0.714286
Jew Control	1	1.0	0.045020	0.031378	0.030014	24	9	0.727273
I'm Jew	2	2.0	0.053206	0.039563	0.008186	36	3	0.923077
Porn	3	2.0	0.038199	0.012278	0.010914	28	0	1.000000
Jew as You	4	2.0	0.023192	0.010914	0.005457	13	4	0.764706
Religious Talk	5	2.0	0.147340	0.102319	0.008186	98	10	0.907407
History Talk	6	2.0	0.080491	0.066849	0.012278	52	7	0.881356
Positive Talk	7	2.0	0.065484	0.054570	0.013643	38	10	0.791667

LF Summary

Column meanings:

- **Emp. Accuracy:** fraction of correct LF predictions. You should make sure this is at least 0.5 for all LFs.
- **Coverage:** % of samples for which at least one LF votes positive or negative. You want to maximize this, while keeping a good accuracy.
- **Polarity:** tells you what values the LF returns.
- **Overlaps & Conflicts:** this tells you how an LF overlaps and conflicts with other LFs. Don't worry about it too much, the Label Model will actually use this to estimate the accuracy for each LF.

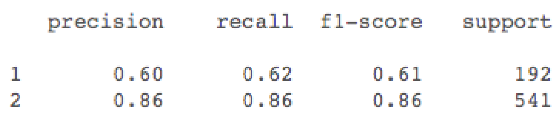
Let's check out our coverage:

```
label_coverage(LF_matrix)
>> 0.8062755798090041
```

That's pretty good!

Now, as a baseline for our weak supervision, we'll evaluate our LFs by using a Majority Label Voter model to predict the classes in our LF set. This just assigns a positive label if most of the LFs are positive, so it's basically assuming that all LFs have the same accuracy.

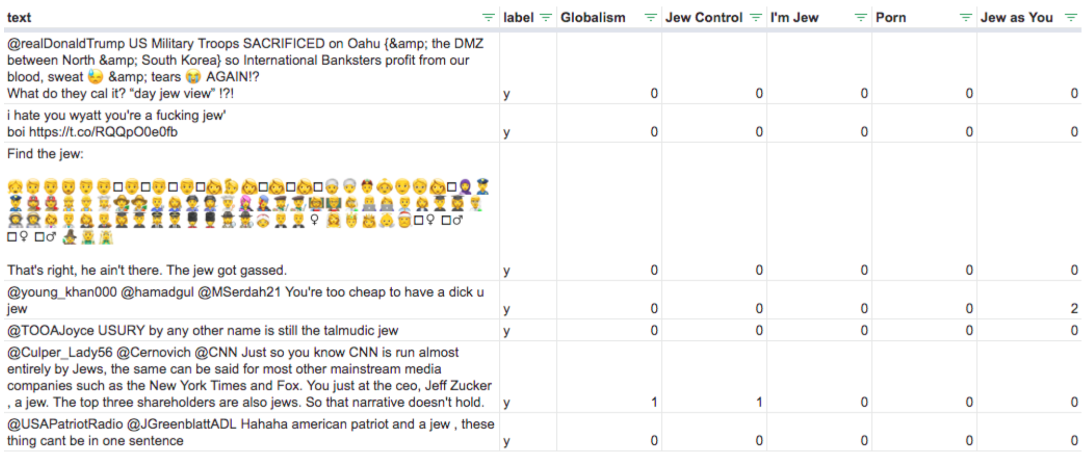
```
from metal.label_model.baselines import MajorityLabelVoter

mv = MajorityLabelVoter()
Y_train_majority_votes = mv.predict(LF_matrix)
print(classification_report(Y_LFs, Y_train_majority_votes))
```



```
     precision    recall  f1-score   support

1         0.60      0.62      0.61       192
2         0.86      0.86      0.86       541
```

Classification Report for Majority Voter Baseline

We can see that we get an F1-score of 0.61 for the positive class (“1”). To improve this, I made a spreadsheet where each row has a tweet, its true label, its assigned label based on each LF. The goal is to find where an LF disagrees with the true label, and fix the LF accordingly.



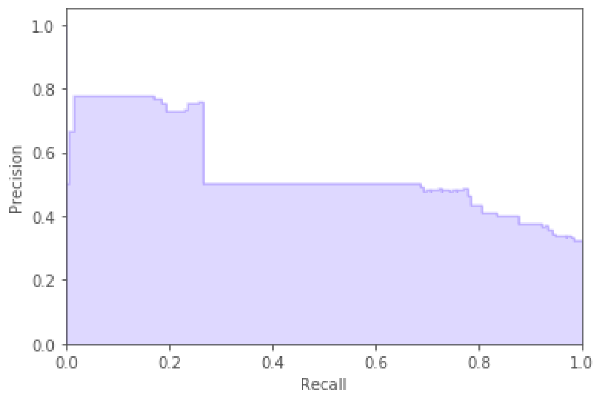
text	label	Globalism	Jew Control	I'm Jew	Porn	Jew as You
@realDonaldTrump US Military Troops SACRIFICED on Oahu {&amp; the DMZ between North &amp; South Korea} so International Banksters profit from our blood, sweat 😓 &amp; tears 😭 AGAIN!? What do they cal it? “day jew view” !?!	y	0	0	0	0	0
i hate you wyatt you're a fucking jew' boi https://t.co/RQQpO0e0fb	y	0	0	0	0	0
Find the jew: [illegible] □♀ □♂ □♀ □♂ [illegible] That's right, he ain't there. The jew got gassed.	y	0	0	0	0	0
@young_khan000 @hamadgul @MSerdah21 You're too cheap to have a dick u jew	y	0	0	0	0	2
@TOOAJoyce USURY by any other name is still the talmudic jew	y	0	0	0	0	0
@Culper_Lady56 @Cernovich @CNN Just so you know CNN is run almost entirely by Jews, the same can be said for most other mainstream media companies such as the New York Times and Fox. You just at the ceo, Jeff Zucker , a jew. The top three shareholders are also jews. So that narrative doesn't hold.	y	1	1	0	0	0
@USAPatriotRadio @JGreenblattADL Hahaha american patriot and a jew , these thing cant be in one sentence	y	0	0	0	0	0

Google Sheet I used for tuning my LFs

After my LFs had about 60% precision and 60% recall, I went ahead and trained the Label Model.

```
Ls_train = make_Ls_matrix(train, LFs)

# You can tune the learning rate and class balance.
label_model = LabelModel(k=2, seed=123)
label_model.train_model(Ls_train, n_epochs=2000, print_every=1000,
                        lr=0.0001,
                        class_balance=np.array([0.2, 0.8]))
```

Now to test the Label Model, I validated it against my test set and plotted a Precision-Recall curve. We can see that we are able to get about 80% precision and 20% recall, which is pretty good. A big advantage of using the Label Model is that we can now tune the prediction probability threshold to get better precision.



Precision-Recall Curve for Label Model

I also validated my Label Model was working by checking the top 100 most anti-semitic tweets in my train set according to the Label Model and making sure it made sense. Now that we are happy with our Label Model, we produce our training labels:

```
# To use all information possible when we fit our classifier, we can # actually combine our hand-labeled LF set with our training set.

Y_train = label_model.predict(Ls_train) + Y_LF_set
```

So, here’s a summary of my WS workflow:

1. Go through the examples in the LF set and identify a new potential LF.
2. Add it to the Label Matrix and check that its accuracy is at least 50%. Try to get the highest accuracy possible, while keeping a good coverage. I grouped different LFs together if they relate to the same topic.
3. Every once in a while you’ll want to use the baseline Majority Vote model (provided in Snorkel Metal) to label your LF set. Update your LFs accordingly to get a pretty good score just with the Majority Vote model.
4. If your Majority Vote model isn’t good enough, then you can fix your LFs or go back to step 1 and repeat.
5. Once your Majority Vote model works, then run your LFs over your Train set. You should have at least 60% coverage.
6. Once this is done, train your Label Model!
7. To validate the Label Model, I ran the Label Model over my Training set and printed the top 100 most anti-semitic tweets and 100 least anti-semitic tweets to make sure it was working correctly.

Now that we have our Label Model, we can compute probabilistic labels for **25 thousand of tweets and use them as a training set**. Now, let’s go ahead and train our classification model!

General Tips for Snorkel:

- On LF accuracy: In the WS step, we’re going for high precision. All of your LFs should have at least 50% accuracy on the LF set. If you can get 75% or more that’s even better.
- On LF coverage: You want to have at least one LF voting positive/negative for at least 65% of our training set. This is called LF Coverage by Snorkel.
- If you’re not a domain expert to start, you’ll get ideas for new LFs as you label your 600 initial data points.

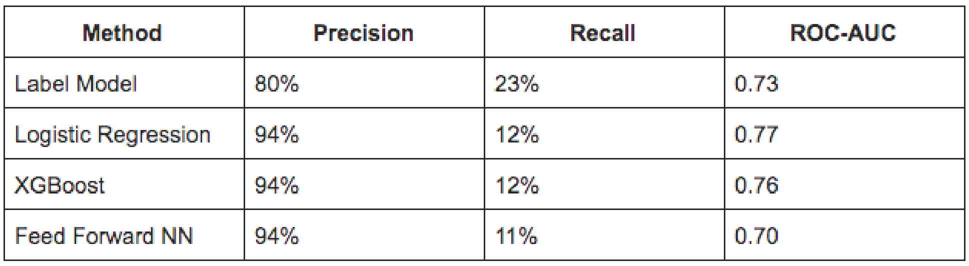
Third Step: Build Classification Model

The last step is to train our classifier to generalize beyond our noisy hand-made rules.

Baselines

We’ll start by setting some baselines**.** I tried to build the best model possible without deep learning. I tried Tf-idf featurization coupled with logistic regression from sklearn, XGBoost, and Feed Forward Neural Networks.

Below are the results. To get these numbers I plotted a Precision-Recall curve against the Development set, and then picked my preferable classification threshold (trying to get a minimum of 90% precision if possible with recall as high as possible).



Method	Precision	Recall	ROC-AUC
Label Model	80%	23%	0.73
Logistic Regression	94%	12%	0.77
XGBoost	94%	12%	0.76
Feed Forward NN	94%	11%	0.70

Baselines

Trying ULMFiT

Once we download the ULM trained on Wikipedia, we need to tune it to tweets since they have a pretty different language. I followed all the steps and code in this awesome blog, and I also used the Twitter Sentiment140 dataset from Kaggle to fine-tune the LM.

We sample 1 million tweets from that dataset randomly, and fine-tune the LM on those tweets. This way, the LM will learn be able to generalize in the twitter domain.

The code below loads the tweets and trains the LM. I used a GPU from Paperspace using the fastai public image, this worked wonders. You can follow these steps to set it up.

```
data_lm = TextLMDataBunch.from_df(train_df=LM_TWEETS,        valid_df=df_test, path="")

learn_lm = language_model_learner(data_lm, pretrained_model=URLs.WT103_1, drop_mult=0.5)
```

We unfreeze all the layers in the LM:

```
learn_lm.unfreeze()
```

We let it run for 20 cycles. I put the cycles in a for loop so that I could save the model after every iteration. I didn’t find a way to do this easily with fastai.

```
for i in range(20):
    learn_lm.fit_one_cycle(cyc_len=1, max_lr=1e-3, moms=(0.8, 0.7))
    learn_lm.save('twitter_lm')
```

Then we should test the LM to make sure it’s making at least a little bit of sense:

```
learn_lm.predict("i hate jews", n_words=10)
>> 'i hate jews are additional for what hello you brother . xxmaj the'
learn_lm.predict("jews", n_words=10)
learn_lm.predict("jews", n_words=10)
>> 'jews out there though probably okay jew back xxbos xxmaj my'
```

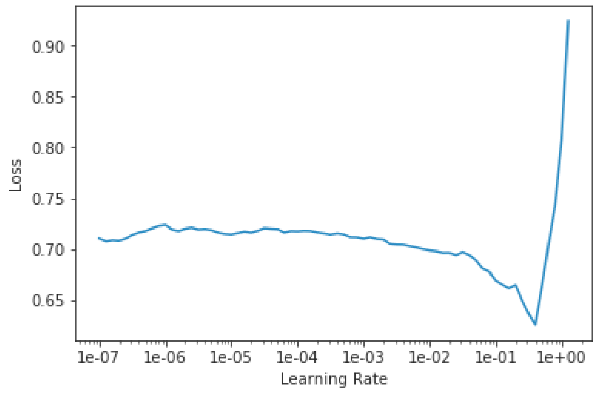
The weird tokens like “xxmaj” are some special tokens that fastai adds that help with text understanding. For example, they add special tokens for capital letters, beginning of a sentence, repeated words, etc. The LM is not really making that much sense, but that’s fine.

Now we’ll train our classifier:

```
# Classifier model data
data_clas = TextClasDataBunch.from_df(path = "",
                                      train_df = df_trn,
                                      valid_df = df_val,
                                      vocab=data_lm.train_ds.vocab,
                                      bs=32,
                                      label_cols=0)

learn = text_classifier_learner(data_clas, drop_mult=0.5)
learn.freeze()
```

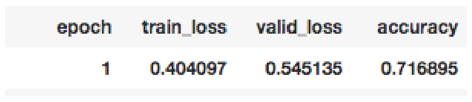
Using fastai’s method for finding a good learning rate:

```
learn.lr_find(start_lr=1e-8, end_lr=1e2)
learn.recorder.plot()
```

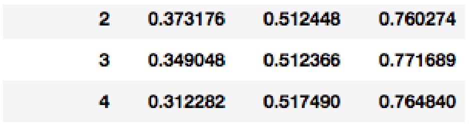


We’ll fine-tune the classifier with gradual unfreezing:

```
learn.fit_one_cycle(cyc_len=1, max_lr=1e-3, moms=(0.8, 0.7))
learn.freeze_to(-2)
learn.fit_one_cycle(1, slice(1e-4,1e-2), moms=(0.8,0.7))
learn.freeze_to(-3)
learn.fit_one_cycle(1, slice(1e-5,5e-3), moms=(0.8,0.7))
learn.unfreeze()
learn.fit_one_cycle(4, slice(1e-5,1e-3), moms=(0.8,0.7))
```



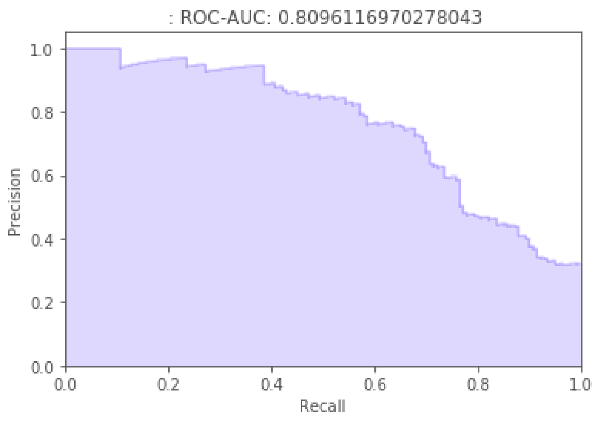
epoch	train_loss	valid_loss	accuracy
1	0.404097	0.545135	0.716895



2	0.373176	0.512448	0.760274
3	0.349048	0.512366	0.771689
4	0.312282	0.517490	0.764840

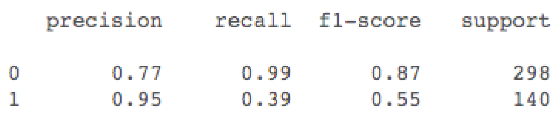
A few training epochs

After fine-tuning, let’s plot our Precision-Recall curve! It was very nice to see this after the first try.



Precision-Recall curve of ULMFiT with Weak Supervision

I picked probability threshold of 0.63, which gives us **95% precision and 39% recall.** This is a very large boost mainly in recall but also in precision.



```
              precision    recall  f1-score   support

           0       0.77      0.99      0.87       298
           1       0.95      0.39      0.55       140
```

Classification Report for ULMFiT Model

Having Fun With Our Model

Below is a pretty cool example of how the model catches that “doesn’t” changes the tweet’s meaning!

```
learn.predict("george soros controls the government")
>> (Category 1, tensor(1), tensor([0.4436, 0.5564]))

learn.predict("george soros doesn't control the government")
>> (Category 0, tensor(0), tensor([0.7151, 0.2849]))
```

Here are some insults against jews:

```
learn.predict("fuck jews")
>> (Category 1, tensor(1), tensor([0.1996, 0.8004]))

learn.predict("dirty jews")
>> (Category 1, tensor(1), tensor([0.4686, 0.5314]))
```

Here is a person calling out anti-semitic tweets:

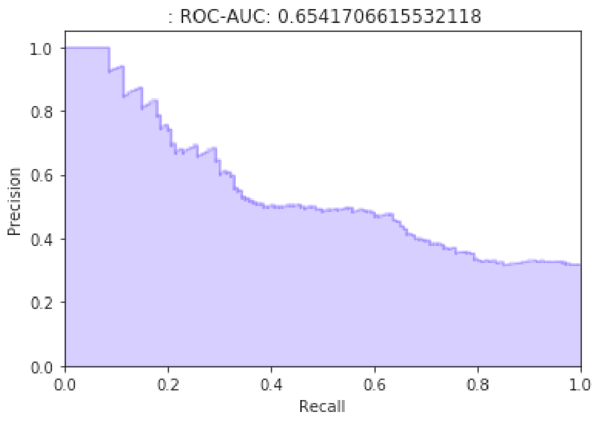
```
learn.predict("Wow. The shocking part is you're proud of offending every serious jew, mocking a religion
and openly being an anti-semite.")
>> (Category 0, tensor(0), tensor([0.9908, 0.0092]))
```

And here are other non anti-semitic tweets:

```
learn.predict("my cousin is a russian jew from ukraine- ▯▯▯ i'm so glad they here")
>> (Category 0, tensor(0), tensor([0.8076, 0.1924]))

learn.predict("at least the stolen election got the temple jew shooter off the drudgereport. I ran out of
tears.")
>> (Category 0, tensor(0), tensor([0.9022, 0.0978]))
```

Does Weak Supervision Actually Help?

I was curious if WS was necessary to obtain this performance, so I ran a little experiment. I ran the same process as before, but without the WS labels, and got this Precision-Recall curve:



Precision-Recall curve of ULMFiT without Weak Supervision

We can see a big drop in recall (we only get about **10% recall** for a 90% precision) and ROC-AUC **(-0.15)**, compared to the previous Precision-Recall curve in which we used our WS labels.

Conclusions

- Weak supervision + ULMFiT helped us hit 95% precision and 39% recall. That was much better than all the baselines, so that was very exciting. I was not expecting that at all.
- This model is very easy to keep up-to-date. There’s no need for relabeling, we just update the LFs and rerun the WS + ULMFiT pipeline.
- Weak supervision makes a big difference by allowing ULMFiT to generalize better.

Next Steps

- I believe we can get the most gains by putting some more effort into my LFs to improve the Weak Supervision model. I would first include LFs based on external knowledge bases like Hatebase’s repository of hate speech patterns. Then, I would write new LFs based on Spacy’s dependency tree parsing.

- We didn’t do any hyperparameter tuning but that could likely help improve both the Label Model and ULMFiT performance.
- We can try different classification models such as fine-tuning BERT or OpenAI’s Transformer.