Building NLP Classifiers Cheaply With Transfer Learning and Weak Supervision

Abraham Starosta
Department of Computer Science
Stanford University
Palo Alto, CA
starosta@stanford.edu
Mentor: Xiaoxue Zang

Abstract

We study the combination of transfer learning and weak supervision to build new text classification models with minimum labeling cost. In our study, we build a high precision classification model to detect anti-semitic tweets based only on their text, with no initial labeled data. By combining weak supervision and transfer learning in an innovative way, we achieve **95%** precision with **39%** recall using less than 1,000 labeled examples. We also observe that building a large training set using weak supervision increases our classification model's recall by **27%** and precision by **5%**.

1 Introduction

There is a catch to training state-of-the-art NLP models: their reliance on massive hand-labeled training sets (Ratner et al. [1]). That's why data labeling is usually the bottleneck in developing NLP applications and keeping them up-to-date. On top of the initial labeling cost, there is a huge cost in keeping models up-to-date with changing contexts in the real-world. 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.

There are multiple techniques that help tackle the data bottleneck problem such as transfer learning, multitask learning, weak supervision, active learning, semi-supervised learning, and zero-shot learning. In the past few years, we have seen an incredible advance in many of these approaches for NLP, pushed by improvements in deep learning.

Weak supervision (WS) is a way to inject expert knowledge into AI systems and build a large training set at a low cost, and it has proved to be very successful. (Bach. S, et al (2018) [10]) describes Snorkel Drybell, a new internal system built by Google to execute weak supervision sources against millions of unlabeled examples. They tested it for 3 tasks, and found that it takes development and labeling costs down by an order of magnitude. In addition, it enabled them to leverage supervision sources that are not available in production, which improves performance of models by 52%. Another example of WS success is shown in (Mallinar. N, et al (2018) [11]), which describes a system built at IBM to make text classification models more cheaply. They call the system SLP, which stands for search, label, and propagate. They also use Snorkel at the learning stage.

After we have built a training set with WS, we need to train a classification model, and this is where transfer learning (TL) is helpful. TL 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 (Howard et al. [2]). ULMFiT offers a pre-trained language model that can be fine-tuned to any specific task. Today, there are newer pre-trained language models that can be fine-tuned such as BERT (Devlin. J, et al (2018) [12]) and OpenAI's Transformer (Vaswani. A, et al (2017) [13].)

Both WS and TL are successful approaches, but the challenge is figuring out how to combine them to train new text classification models as cheaply as possible. This is not just a deep learning problem, but also a worklow problem. In our study, we try to tackle this question.

2 Related Work

2.1 Multitask Learning

(Caruana, R. et al. [5]) defines MTL as a way to improve generalization by using domain information in related tasks; what is learned for each task can help others perform better. It does this by co-learning *N* tasks in parallel, using a shared representation. Although MTL can focus on improving the performance of all tasks at once, the classic approach is to have a main task which we are focusing on optimizing and a set of auxiliary tasks which we hope boost performance for the main task. It is likely that there is an auxiliary task that can help your main task perform better.

2.2 Transfer Learning vs Multitask Learning

It is worth briefly mentioning the relationship between MTL and transfer learning (TL). Transfer learning is somewhat different because given *N - 1* source tasks, its only goal is to transfer this knowledge in order to perform well on the N^{th} target task whereas in MTL, all tasks are trained together. Ultimately, both boil down to solving a multi-objective optimization problem. (Mou. L,. et al [6]) found that MTL is very similar to transfer learning in terms of model performance.

2.3 Zero-Shot Learning

Zero-shot learning is a nascent field that attempts to correctly perform a new task without being trained on any examples of that specific task (Xian. Y, et al (2017) [8].) We can think of it as TL, but without an opportunity to see examples for the target task. Recent work on multitask learning such as DecaNLP (McCann. B, et al (2018) [7]), which has been trained on 10 different NLP tasks, shows zero-shot capabilities. They achieve a strong performance on zero-shot relation extraction by framing all tasks as a question-answering task, which makes it possible to as a new type of question and still get a valid answer.

2.4 Semi-supervised Learning

Semi-supervised learning makes assumptions about the underlying data to leverage unlabeled data. Semi-supervised learning comes from the 60's, and it has recently received more attention because of our data hungry models (Cholaquidis. A, et al (2018) [9]). Commonly, it leverages small sets of labeled data in order build larger training sets. Their study also explains that having a large set of data is like knowing $p(x)$, and its usefulness depends on how much knowing $p(x)$ helps in the inference of $p(y|x)$. Weak supervision also helps build large training sets, but instead of leveraging labeled data, it leverages a variety of lower quality inputs.

3 Approach

3.1 Weak Supervision With Snorkel

Weak Supervision (WS) helps us create training large sets quickly (Ratner et al. [1]). It does so by pooling noisy signals together and programmatically label large training sets. These signals are hand-crafted by domain experts to have a high accuracy. Thus, we can think of WS as a way to efficiently infuse expert knowledge into an AI system.

We have two large motivations to use WS. First, when we want to update our model, all we need to do is update our rules. With that we then automatically rebuild our large training set and finally retrain our discriminative model. Second, a discriminative model will be able to generalize beyond the rules in our WS model, thus often increasing recall.

Snorkel calls WS rules Labeling Functions (LFs), and each LF casts a vote on each unlabeled example: positive, negative, or abstain. and they are represented as Python functions that. LFs in NLP are often comprised of regexes, distant supervision, crowdsourced labels, or external classifiers. After we define all of our LFs, Snorkel will train a generative "Label Model" to learn the accuracy of each LF and which can then give us probabilistic labels to thousands of unlabeled data points.

Snorkel's generative model's goal is to model statistical dependencies between LFs and learn the accuracies for each LF. For example, it should model pairwise dependencies, and even higher-order ones. The model is built as a factor graph, with three factor types ϕ: label propensity, accuracy, and correlations between LFs. To learn the model without access to labeled data, it minimizes the log marginal likelihood given an observed label matrix Λ and parameters w.

$$p_w(\Lambda, Y) = Z_w^{-1} \exp\left(\sum_{i=1}^{m} w^T \phi_i(\Lambda, y_i)\right)$$

$$\hat{w} = \underset{w}{\arg\min} - \log \sum_{Y} p_w(\Lambda, Y)$$

Figure 1: Snorkel's objective to be minimized.

We already defined ϕ, Λ, and w. Snorkel defines Z_w is a normalizing constant, and y_i is the label of the sample i. The objective is optimized by interleaving gradient descent steps with Gibbs sampling. Gibbs Sampling is a Monte Carlo Markov Chain (MCMC) technique used to estimate posterior probabilities by random sampling. Then, Snorkel estimates the labels by computing $\tilde{Y} = p(Y|\Lambda)$.

To summarize, the Snorkel process steps: **1)** we create LFs, train a generative model, **2)** create our probabilistic training data, **3)** and then train a discriminative model.

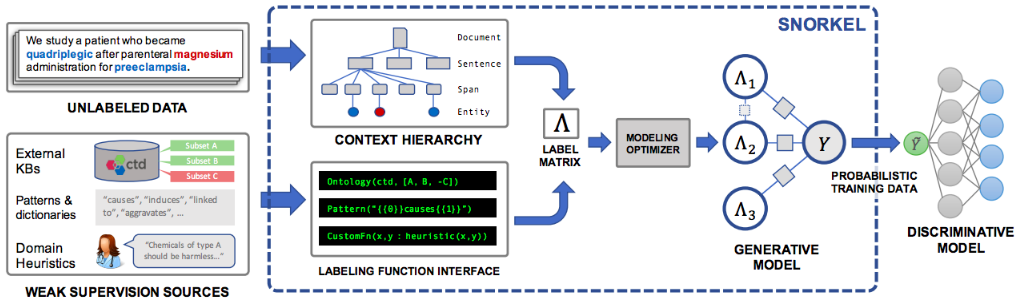

We study a patient who became quadriplegic after parenteral magnesium administration for preeclampsia.
UNLABELED DATA
External KBs
ctd
Subset A
Subset B
Subset C
Patterns & dictionaries
"causes", "induces", "linked to", "aggravates", ...
Domain Heuristics
"Chemicals of type A should be harmless..."
WEAK 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
Λ1
Λ2
Λ3
Y
GENERATIVE MODEL
Ỹ
PROBABILISTIC TRAINING DATA
DISCRIMINATIVE MODEL

Figure 2: Snorkel Metal Weak Supervision Process.

3.2 Transfer Learning With 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).

For its language model, ULMFiT's uses an AvSGD Weight-Dropped LSTM (AWD-LSTM) (Merity et al. [3].) Weight-Dropping uses DropConnect on the recurrent hidden to hidden weight matrices (Wan et al., 2013. [4]) AWD-LSTM has no attention, short-cut connections, or other sophisticated additions. ULMFiT uses AWD-LSTM with an embedding size of 400, 3 layers, 1150 hidden activations per layer, and a BPTT batch size of 70. They apply dropout of 0.4 to layers, 0.3 to RNN layers, 0.4 to input embedding layers, 0.05 to embedding layers, and weight dropout of 0.5 to the RNN hidden-to-hidden matrix.

ULMFiT consists of 3 stages: **1)** pre-train 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. Uses a slanted triangular learning rate (STLR) schedule, **3)** train a discriminative classifier by fine-tuning it with gradual unfreezing.

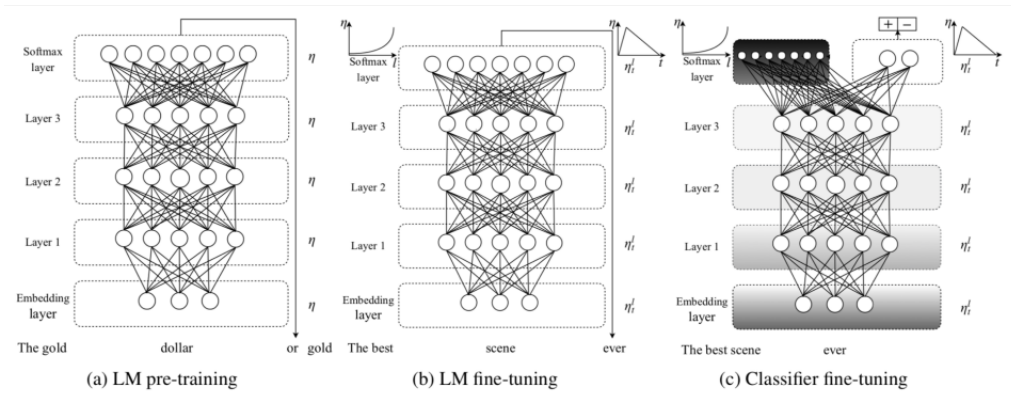


Figure 3: ULMFiT 3 Step Approach.

We believe ULMFiT has three main advantages: **1)** with only 100 labels, it can match the performance of training from scratch on 100x more data, **2)** Fastai's API is relatively easy to use in a single GPU. I use Paperspace's Fastai's public image machine with an 8GB GPU and 30GB RAM, **3)** produces a Pytorch model we can deploy in production.

4 Experiments

4.1 Dataset

We downloaded close to 25,909 tweets that mention the word "jew." We then dedeuplicated them based on text overlap, with a threshold of 80% word overlap. Our goal is to then train a classifier to detect which of those are anti-semitic. 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). We have 24,738 unlabeled tweets (train set), 733 labeled tweets for our LF set, and 438 labeled tweets for testing.

4.2 Building A Weak Supervision Model With Snorkel

The goal of this step is build a large training set. We 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.) Because we believe this is a large part of our innovation, we try to explain our workflow in detail, and we also add implementation details in the Appendix.

Steps:

1. Go through the examples in the LF set and identify a new potential LF. (See Appendix 7.1 for examples.)
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. We grouped different LFs together if they relate to the same topic. (See Appendix 7.2, Figure 10.)
3. Every once in a while we use the baseline Majority Vote model (provided in Snorkel Metal) to label your LF set. We update LFs accordingly to optimize our score with the Majority Vote model.
4. If the Majority Vote model achieves more that 60% precision and 60% recall, we train our Snorkel Label Model. Otherwise, we go back to step 1.
5. To validate the Label Model, we looked at the top 100 most anti-semitic tweets from our Training set.

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. After our LFs had about **60%** precision and **60%** recall, we trained the Label Model with a learning rate of 0.0001, 2000 epochs, and a prior class distribution of 0.8 the negative class and 0.2 for the positive class.

Then, we compute the 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.

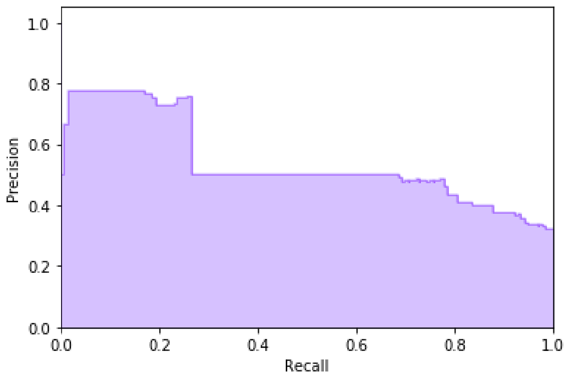


Figure 4: Precision-Recall Curve for Label Model

To validate that the Label Model is working correctly, we computed its predictions on the unlabeled training set and looked at the most and least anti-semitic ones.

Here are some examples of those predictions:

Most anti-semitic:

- He should've just called the owner a cheap Jew instead of hinting at it.
- Offset in other ways? Tell me, Michael. When's the last time the Mets were able to get out of a bad contract? And these cheap Jew owners never eat up money where small market teams like the Braves do.
- So the dude who just died Rothschild , a jew , is in that bloodline who OWNS the media amp; profited the most off slaver…

Least anti-semitic:

- "There is no more "Jew" and "Gentile" racial distinctions. All nations are now a part of Spiritual Israel in Christ. Christ's kingdom is here now in fullness. All (who were a part of the true spiritual) Israel were saved and given the inheritance (Romans 11:26)."
- It's natural. Haredi Jew's are racist and spit on Catholics in Jerusalem. I hear stories on how Christians in Israel are third class citizens there, quite sad.
- The ONLY fight you're in, us to discredit Christians amp; get criticism of Israel confused w REAL Jew haters like you....you're the Matrix, not Neo...you're more NeoNazi though

After this validation, we can use the Label Model to label our 24,738 training examples.

4.3 Classification Model Baselines

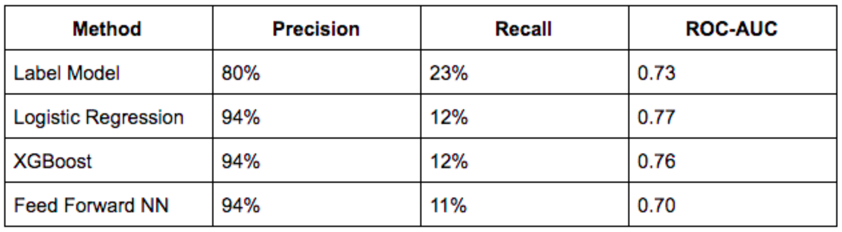
We'll start by setting some baselines. I tried to build the best model possible without deep learning. I tried Tf-idf featurization coupled with logistic regression from sklearn, XGBoost, and Feed Forward Neural Networks.

Tf-idf stands for *term frequency (tf) - inverse document frequency (idf)*, and it's a standard way of converting documents into vectors so that our ML models can make predictions. tf stands for the number of occurrences of term i in the document, and df is the number of documents containing i.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

Figure 5: Tf-idf formula.

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

Figure 6: Baselines.

4.4 Classification Model With Transfer Learning

Once we download the ULM trained on Wikipedia, we need to tune it to tweets since they have a pretty different language.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.

After fine-tuning, we compute the following Precision-Recall curve, which compared to our baselines has an ROC-AUC gain of (**+0.15**).

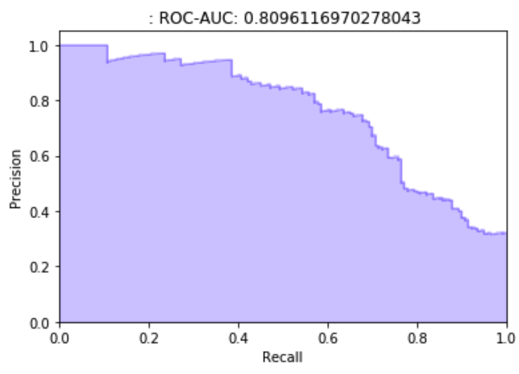


Figure 7: ULMFiT Precision-Recall Curve

We picked a classification 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.

4.5 The Unreasonable Effectiveness of Weak Supervision

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:

We can see a big drop in recall (we only get about **10%** recall for a 9**90%** precision) and ROC-AUC (**-0.15**), compared to the previous Precision-Recall curve in which we used our WS labels.

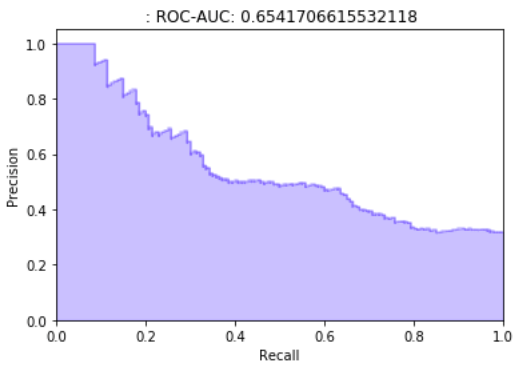


Figure 8: ULMFiT Precision-Recall Curve

5 Analysis

5.1 Examples of Correct Predictions

- Antisemitic: "George Soros controls the government"
- Not Antisemitic: "George Soros doesn't control the government"
- Antisemitic: "f**k jews"
- Antisemitic: "dirty jews"
- Not Antisemitic: "Wow. The shocking part is you're proud of offending every serious jew, mocking a religion and openly being an anti-semite."

5.2 Error Analysis

We collected more tweets that mentioned the word "jew", which had never been seen by the model, and computed the probability of them being anti-semitic. Then, we found 60 false positives.

We observe that **65%** of false positive errors are for tweets that end in the word "jew". For example, the following tweet is labeled as antisemitic: "Then they came for the Jews, and I did not speak out— Because I was not a Jew." **65%** is significally higher than the control **2.1%** of tweets that end in the word "jew" in the whole training set. A hypothesis for this type of errors is that the unidirectional LSTM based language model used in ULMFiT is biased towards the last word in the text. This problem could be ameliorated with attention in the language model. We can explain this effect by the fact that the language model used in ULMFiT has an AWD-LSTM architecture, which is unidirectional. Unidirectional LSTMs have been previously observed to bias their. We found no other pattern in the other 21 false positives.

6 Conclusion

Weak supervision (WS) combined with ULMFiT helped us achieve **95%** precision and **39%** recall for an antisemitic tweet classifier, which was a large improvement over all the baselines, which didn't allow for a recall larger than 12% for a 90% precision. We found that without weak labels, ULMFiT performs very similarly to our baselines, with a decrease in ROC-AUC of **-0.15**. That evidence supports that WS makes a big difference by allowing ULMFiT to generalize better.

To improve our anti-semitism classifier, we plan to experiment with other weak supervision sources such as distant supervision or external models like the Perspective API toxicity scores. In addition, ULMFiT is the best TL approach. There are other language model based TL techniques we can explore such as BERT and OpenAI's Transformer fine-tuning. We hope that the approach presented in our study will help build classification models more cheaply in other domains.

References

[1] Ratner, A. et, al. (2018) Snorkel: Rapid Training Data Creation with Weak Supervision.

[2] Howard, J. et, al. (2018) Universal Language Model Fine-tuning for Text Classification.

[3] Merity, S. et, al. (2017) Regularizing and Optimizing LSTM Language Models.

[4] Wan, M. et, al (2013) Regularization of neural networks using dropconnect.

[5] Caruana, R. et, al (1997) Multitask Learning.

[6] Mou. L,. et al (2016) How Transferable Are Neural Networks in NLP Applications?

[7] McCann. B, et al (2018) The Natural Language Decathlon: Multitask Learning as Question Answering.

[8] Xian. Y, et al (2017) Zero-Shot Learning - A Comprehensive Evaluation of the Good, the Bad and the Ugly.

[9] Cholaquidis. A, et al (2018) On semi-supervised learning.

[10] Bach. S, et al (2018) Snorkel DryBell: A Case Study in Deploying Weak Supervision at Industrial Scale.

[11] Mallinar. N, et al (2018) Bootstrapping Conversational Agents With Weak Supervision.

[12] Devlin. J, et al (2018) BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

[13] Vaswani. A, et al (2017) Attention is all you need.

7 Appendix

7.1 Label Function Examples

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
```

7.2 Designing LFs

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.

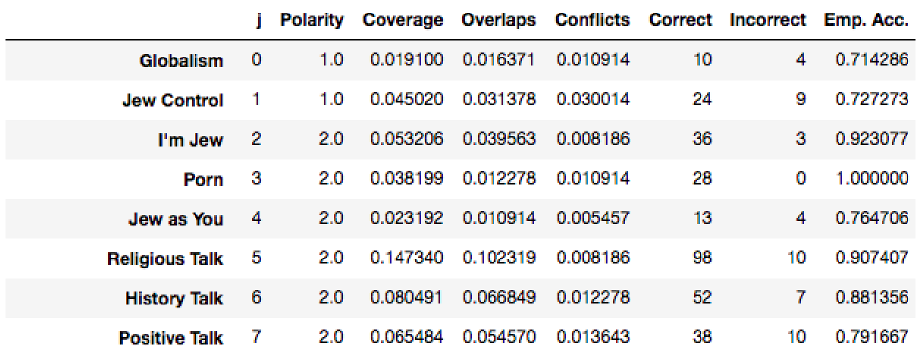
```
# We build a matrix of LF votes for each tweet
LF_matrix = make_Ls_matrix(LF_set, LFs)

# 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()))
```

Figure 9: Code to get a summary of our LFs.

We have a total of 24 LFs, with a total of 80% coverage, meaning that 80% of the data points had at least one LF returning a value other than abstain.

Here's how the LF summary looks like for a sample of my LFs. Below the table you can find what each column means.

Column meanings:



	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

Figure 10: LF Summary.

- **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.

7.3 ULMFiT Usage

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)
```

Figure 11: Loading Language Model.

We unfreeze all the layers in the LM:

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.

Now, we build a classifier and fine-tune it 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))
```

Figure 12: Classifier Fine-Tuning.