Toxic Comment Classification Challenge

Jason Phang
New York University
zp489@nyu.edu

Mihir Rana
New York University
ranamihir@nyu.edu

Kenil Tanna
New York University
kenil@nyu.edu

1 Introduction

For the myriad benefits that the growing adoption of the Internet has brought about, ranging from expanded free education to freer speech to broader access to basic services, the evolution of the Internet of the 21st Century has not been without its growing pains. In recent years, the confluence of the growth of social media, the increasing overlap between Internet culture and mainstream pop culture, and radicalization within pockets of internet sub-communities have brought the issue of toxic Internet subculture and online behavior to the forefront of social consciousness.

In this project, we tackle the Toxic Comment Classification Challenge hosted on on Kaggle. This data-set consists of a large corpus of Wikipedia discussion comments, a portion of which has been labeled into 6 non-exclusive categories of toxicity ("`toxic`", "`severely toxic`", "`obscene`", "`threat`", "`insult`", "`identity-hate`"). The task is a multi-label classification problem, and the goal is to train a machine learning model to automatically categorize each comment into different classes of toxicity. The evaluation metric we use is mean class-wise AUC-ROC score on each target class, which aligns with that used in the competition.

More broadly, this project falls into a large field pertaining to textual classification, within which there has been much prior work. In this short report, we catalog our efforts in applying a broad range of standard text classification models, ranging from the simple and robust (logistic regression) to the cutting edge (`xgboost` [2], `fastText` [6]), and evaluate their effectiveness in classifying and labeling toxic comments.

2 Data [code]

The data set that we are working with has been kindly provided on the Kaggle platform by the Conversation AI team.

For the purposes of evaluation, as we do not have access to either the public leader-board nor private data sets, we will be using the provided training set from the competition and performing our own test/validation-splits. We split the data into training (60%), validation (20%), and test (20%) sets. The column names, their respective data types, along with an example row, are given in Table 1:

	comment_id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
dtype	object	object	integer	integer	integer	integer	integer	integer
example	003217c3eb469ba9	Hi! I am back again!\nLast warning...	1	0	0	1	0	0

Table 1: Sample Data Row

where the first column is the raw text of the comment, and the last 6 columns (targets) are binary depending on whether the comment is classified as that label (1) or not (0).
The total number of rows are 159571. There are no missing values.

For each of the 6 labels, the value counts are given below in Table 2:

	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	144277	157976	151122	159093	151694	158166
1	15294	1595	8449	478	7877	1405

Table 2: Label Distribution

We note that the distribution is not uniform across groups, nor balanced sample between toxic/non-toxic.

The statistics for the length (number of characters) of comments for both cleaned and uncleaned data are summarized below in Table 3:

	mean	std	min	25%	50%	75%	max
comment_length (raw)	394	590	6	96	205	435	5000
comment_length (clean)	227	361	0	54	116	245	6471

Table 3: Comment Length Descriptive Statistics

It is worth nothing that while the `mean` and `std` decreased after cleaning, the `max` increased – which is the case because of some examples when broken down into tokens after lemmatization increase in length, e.g. "hasn't" gives ["have", "not"].

3 Methodology

An overview of the general methodology followed for the Machine Learning models is described below in Figure 1. The specifics of each part, and the methodology for the Deep Learning models are described in detail in the following sections.

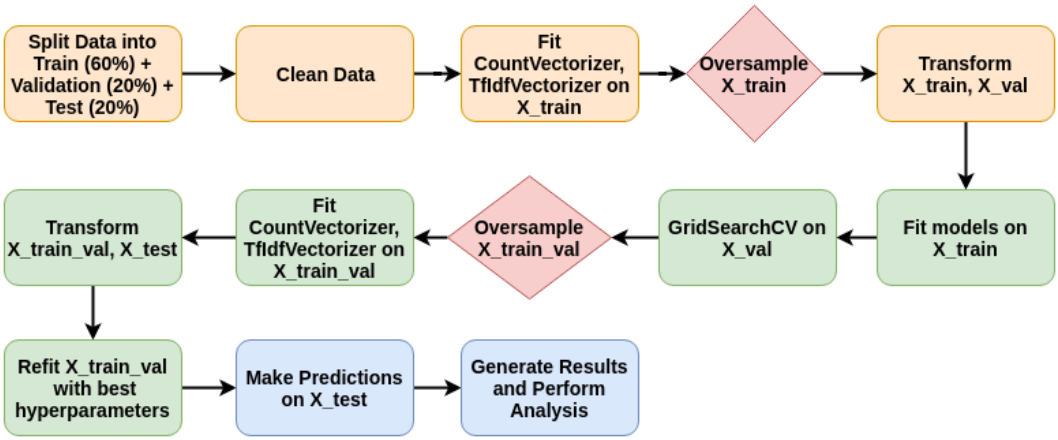

Figure 1: Flowchart of Methodology

The following are the salient characteristics and observations in this methodology:

- Cleaning the data, as expected, produces significantly better results compared to models fitted on uncleaned data.
- Our pipeline provisions for oversampling; however, upon trying Random Over-Sampling on all target columns to bring them to a comparable level, we found that this worsens the results.
- For cross-validation using grid search, we use a predefined split on validation data for evaluation and chose the set of parameters which give the best mean column-wise AUC-ROC score.

3.1 Data Preparation [code]

Note that many of our models require the comments to be pre-tokenized. We follow a standard NLP text preprocessing pipeline of tokenization and sanitization (e.g. removing of stop words, rare words, lower-casing, etc.) using spaCy. In the list of stop-words, negation words (e.g. *not*, *don't*, etc.) were not included as they contribute significantly towards the sentiment of a sentence. Lemmatization of all words was also performed to bring them to their root word (which increased the performance marginally). As mentioned previously, oversampling did not seem to help; moreover, the default splits used by `scikit-learn` preserved the stratification across all target columns well enough as shown in Table 4, and hence in the final modeling, oversampling was not used. Finally, `CountVectorizer` and `TfIdfVectorizer` with an `ngram_range` = 2 and `max_features` = 5000 were used for generating the0 features.

	y_train	y_val	y_train_val	y_test
toxic	9.528	9.551	9.533	9.789
severe_toxic	1.005	0.993	1.002	0.990
obscene	5.321	5.274	5.309	5.239
threat	0.287	0.295	0.289	0.342
insult	4.960	4.860	4.935	4.941
identity_hate	0.902	0.833	0.885	0.862

Table 4: Percentage of 1's in Each Data Set for After Splitting

After preparing the data, **two additional features** were extracted from the data:

- `comment_length`: Length of comment
- `word_length_std`: Standard deviation of length of words in comment

The intent was to investigate if toxic comments tend to be relatively longer/shorter than normal ones, and/or have some unusually long/short words.

3.2 Machine Learning Models [code]

We used the following models in order of increasing complexity. Each model is trained separately for each category of toxic comments, and is fitted using both `CountVectorizer` and `TfIdfVectorizer`:

1. Bernoulli Naive-Bayes
2. Logistic Regression with $l1$ penalty for sparsity
3. Logistic Regression with $l2$ penalty
4. SVM with Naive-Bayes (NBSVM) [10]
5. Random Forests
6. XGBoost [2]
7. Ensembling (Stacking)

Bernoulli Naive-Bayes was used a baseline, and as the results (described below) show, other more sophisticated models outperform it by a large margin, which is why extensive hyper-parameter tuning was not performed for it.

SVM, in its raw form, turned out to be extremely computationally expensive; therefore, we implemented our own `Naive-Bayes SVM` [10], adapted from this discussion on Kaggle.

For Logistic Regression, the regularization parameter (`C`) was tuned; for tree-based models, the number of estimators (`n_estimators`), maximum depth (`max_depth`), maximum features (`max_features`), and additionally for XGBoost, the learning rate (`learning_rate`) and regularization parameter (`reg_lambda`) were tuned using extensive ranges (refer code).

3.3 Deep Learning Models

In the past decade, Deep Learning models have proven stunningly effective in a broad class of domains, including image recognition, text classification, machine translation, and optimal control. We also experiment with a small selection of deep learning methods for our text classification task.

3.3.1 GRU-RNN [code]

Recurrent Neural Networks (RNN) have been shown to be highly effective for a broad range of text and sequence-based applications. For this project, we use a modified variant of the Vanilla Elman RNNs, known as the Gated Recurrent United (GRU) RNN [4]. The GRU is largely similar to its cousin, the Long Short-Term Memory (LSTM) RNN [5] in incorporating multiple gates around its hidden-activation computations, allowing information and gradients to propagate across many time steps.

The setup for the application of RNNs is as follows: First, we split the input text into a sequence of word tokens. Next, we map each word to a corresponding fixed-dimension vector embedding (described below). Finally, we run the sequence of embedding vectors through the RNN, and obtain a final hidden-state. We finally use a simply fully connected layer and softmax operation to compute the probability of that comment being classified into any of the 6 categories. Following the best practices in deep learning for text, we apply a bi-directional RNN, feeding the tokens in both directions via parallel RNNs.

Word embeddings are a methodology for mapping individual word tokens to fixed-dimension vectors, wherein the embedded latent space is constructed to capture pertinent information about each word, including semantics and syntactical role. There are several formulations of word embeddings, including word2vec [8], GloVe [9], and fastText [7]. We chose to use GloVe embeddings because of their easy availability. In addition, there are also several versions of GloVe embeddings trained on different corpora - of especial interest to us is the GloVe embedding trained on Twitter data, which we believed could be better at capturing information about Internet discourse.

There are many potential hyper-parameters for deep learning models. For brevity, we choose to focus our attention on the following:

1. Dimension of hidden activations
2. Using GloVe embeddings trained on Wikipedia, GloVe embeddings trained on Twitter, or randomized embeddings (refer code)

We also run these models on both sanitized and raw comments.

3.3.2 fastText [code]

Beyond the heavy-duty RNN models, there have also been simpler models for text classification trained using deep learning techniques. Here, we focus on **fastText** [7], a highly efficient text classification model based on word embedding techniques[1].

fastText refers both to the word embeddings as well as the text classification model trained on the learned word embeddings. The word embeddings are trained in an unsupervised way based on word co-occurrence statistics as well as sub-word information based on n-grams [1]. The availability of the latter allows fastText to be applied even to out-of-vocabulary words. The corresponding classifier is a linear classifier trained based purely on the bag-of-words, or the sum of the corresponding vectors. Although taking a simple bag-of-words means we lose much of the word-ordering information within the comment, it has been shown that bag-of-word techniques can be surprisingly effective [3] in text classification. While fastText has a native built-in multi-class classification capability, we note that its multi-class classification maps to exclusive labels, which is not appropriate for our use. Hence, we trained 6 individual single-class classifiers to classify our toxic comments.

[1] Strictly speaking, fastText is not a deep learning model as it does not use multiple nested layers of computation. Nevertheless, because word embeddings have been such a key component of many text-based deep learning models, and word embeddings are often trained using similar techniques like back-propagation, it is common to refer to word embeddings, including fastText, as belonging to the pool of deep learning methods.

The most pertinent hyper-parameter for fastText is the word n-grams, which further learns embeddings for word n-grams up to a provided n. Inclusion of additional n-grams is one way to incorporate addition word-order information.

4 Evaluation

4.1 Evaluation Metric

Following the evaluation metric stipulated in the Kaggle competition, we evaluated the proposed models based on the **mean column-wise ROC-AUC** for each category; this is chosen because the labels are highly imbalanced (as is evident in Table 2), and using a metric such as accuracy would not be reflective of the true performance. Thus, we evaluate the average ROC-AUC across each category of toxic comments. More specific category-based evaluation and error analysis are also performed (described in Discussion), and so is the performance on longer and ambiguous sentences.

5 Results

5.1 Machine Learning Models

The ROC-AUC values for each model fitted using `CountVectorizer` *without* and *with* the additional features respectively are shown below:

	Toxic	**Severely Toxic**	**Obscene**	**Threat**	**Insult**	**Identity Hate**	**Mean**
BernoulliNB	0.893	0.887	0.906	0.936	0.953	0.859	0.906
LogisticRegression ($l1$)	0.850	0.863	0.893	0.931	0.919	0.892	0.891
LogisticRegression ($l2$)	0.869	0.920	0.943	0.948	0.920	0.920	0.920
NBSVM	0.897	0.932	0.952	0.939	0.934	0.920	0.929
Random Forests	0.943	0.947	0.965	**0.975**	0.967	0.912	0.951
XGBoost	**0.949**	**0.951**	**0.967**	0.974	**0.974**	**0.924**	**0.956**
Mean	0.900	0.917	0.938	0.950	0.944	0.905	0.926

Table 5: AUC Scores for Machine Learning Models (CountVectorizer) *Without* Added Features

	Toxic	**Severely Toxic**	**Obscene**	**Threat**	**Insult**	**Identity Hate**	**Mean**
BernoulliNB	0.893	0.886	0.905	0.936	0.953	0.859	0.905
LogisticRegression ($l1$)	0.851	0.890	0.897	0.932	0.903	0.870	0.891
LogisticRegression ($l2$)	0.838	0.916	0.943	0.922	0.762	0.856	0.873
NBSVM	0.901	0.934	0.930	0.927	0.922	0.908	0.920
Random Forests	0.939	0.942	0.962	0.970	0.970	0.905	0.948
XGBoost	**0.950**	**0.950**	**0.967**	**0.973**	**0.973**	**0.925**	**0.956**
Mean	0.895	0.920	0.934	0.943	0.914	0.887	0.916

Table 6: AUC Scores for Machine Learning Models (CountVectorizer) *With* Added Features

The same for models fitted using `TfIdfVectorizer` *without* and *with* the additional features respectively are shown below:

	Toxic	**Severely Toxic**	**Obscene**	**Threat**	**Insult**	**Identity Hate**	**Mean**
BernoulliNB	0.893	0.887	0.906	0.936	0.953	0.859	0.906
LogisticRegression ($l1$)	0.943	0.951	0.967	**0.976**	0.962	**0.934**	0.955
LogisticRegression ($l2$)	**0.951**	**0.954**	**0.969**	**0.976**	**0.976**	**0.934**	**0.960**
NBSVM	0.946	0.949	0.966	0.974	0.972	0.933	0.957
Random Forests	0.941	0.943	0.964	0.968	0.956	0.910	0.947
XGBoost	0.945	0.949	0.967	0.970	0.962	0.924	0.953
Mean	0.937	0.939	0.956	0.967	0.963	0.916	0.946

Table 7: AUC Scores for Machine Learning Models (TfIdfVectorizer) *Without* Added Features

	Toxic	Severely Toxic	Obscene	Threat	Insult	Identity Hate	Mean
BernoulliNB	0.893	0.886	0.905	0.936	0.953	0.859	0.905
LogisticRegression ($l1$)	0.944	0.951	0.968	0.977	0.962	**0.935**	0.956
LogisticRegression ($l2$)	**0.951**	**0.955**	**0.970**	**0.979**	**0.974**	0.934	**0.961**
NBSVM	0.927	0.922	0.939	0.911	0.894	0.902	0.916
Random Forests	0.934	0.940	0.961	0.968	0.953	0.902	0.943
XGBoost	0.948	0.948	0.967	0.972	0.960	0.925	0.953
Mean	0.933	0.934	0.952	0.957	0.949	0.909	0.939

Table 8: AUC Scores for Machine Learning Models (TfIdfVectorizer) *With* Added Features

Since the models with configurations mentioned in Table 9 performed the best, we decided to run ensembles on these models. After trying various combinations of all the models, an average of all models except `BernoulliNB` gave the best result, which even surpassed the individual results of individual models:

	Toxic	Severely Toxic	Obscene	Threat	Insult	Identity Hate	Mean
Ensemble	0.953	0.956	0.971	0.978	0.975	0.935	**0.962**

Table 9: AUC Score for Ensembled Model (TfIdfVectorizer) *Without* Added Features

Please note, we have only included all **ROC curve plots** for the `CountVectorizer` and `TfIdfVectorizer` in the Appendix.

5.2 Deep Learning Models

Deep learning models have been shown to obtain good performance when trained end-to-end on raw data. Hence, we train deep learning models on both versions of data, the preprocessed and lemmatized data used in the machine learning models above, and the raw text. We selected the best model hyper-parameters based on the validation set. For the GRU-RNN trained on raw data, we used a single-layer, 256-dimension GRU with 0.2 dropout and the GloVe embeddings trained on Twitter data, whereas for the clean data, we used a single-layer, 128-dimension GRU with 0.2 dropout and the GloVe embeddings trained on Wikipedia and Gigaword data. Both clean and raw fastText models used up to 3-grams of words, and up to 4-grams of characters.

The ROC-AUC values for each model fitted on raw and cleaned data are shown below:

	Toxic	Severely Toxic	Obscene	Threat	Insult	Identity Hate	Mean
GRU-RNN (Raw)	0.922	0.960	0.936	0.952	0.936	0.935	0.940
GRU-RNN (Clean)	**0.962**	**0.984**	**0.984**	**0.969**	**0.973**	**0.972**	**0.974**
fastText (Raw)	0.958	0.969	0.971	0.971	0.960	0.961	0.965
fastText (Clean)	0.959	0.970	0.979	0.955	0.956	0.968	0.964

Table 10: AUC Scores for Deep Learning Models

As before, all **ROC curve plots** for these models are attached in the Appendix.

6 Discussion

In the case of `CountVectorizer` without the two extra features, as is clear, tree-based models beat all other models for all target columns. XGBoost performs the best, with Random Forest being a close second. Also, adding the two features seems to (very) marginally decrease the performance.

When using `TfIdfVectorizer`, thing shake up a little. Linear models outperform the rest of the models, with $l2$ regularized Logistic Regression taking the first place, followed by the $l1$ regularized one.

Similar to this, when we use the additional features, the linear models retain the first two spots, with the tree-based ones not being far behind.

It should be noted that tree-based models (XGBoost, Random Forests) are not far behind in either case for `TfIdfVectorizer`, and more importantly, their performance remains nearly the same as that for `CountVectorizer`. This illustrates the fact that tree-based models are essentially invariant to any form of feature normalization (in this case, TF-IDF normalization).

Moreover, there is a stark difference in the performance of linear models in the case of `TfIdfVectorizer` vis-à-vis `CountVectorizer`, which confirms our knowledge that feature normalization plays a key role for linear models (TF-IDF Vectorization is a form of normalization that penalizes words that occur very frequently across different documents, i.e., sentences).

Interestingly, similar to the case of `CountVectorizer`, for `TfIdfVectorizer`, adding the two extra features marginally decreases the performance (except for Logistic Regression with $l2$ penalty). Upon drawing the feature importance chart for the first 20 features in the case of XGBoost with `TfIdfVectorizer`, we found the two added features to be the most important:

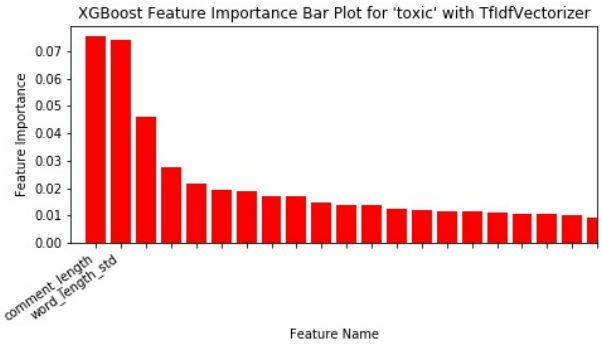

Figure 2: Feature Importance Plot (Top 20 Features)

This illustrates the common misconception that feature importance always correlates with better performance.

Finally, we note that the ensembled model performs the best across all Machine Learning models. As the theory behind the technique suggests, it is presumably because the underlying models used are quite uncorrelated – with some being linear (with some having different forms of regularization, some having different features, e.g. NBSVM), and some being tree-based.

6.1 Deep Learning Models

We were highly impressed by the performance of the deep learning models, which were competitive with or better than even the machine learning models with `TfIdfVectorizer`. The best overall performing model was the GRU-RNN using cleaned data. We were a little surprised that the GRU-RNN slightly under-performed using raw data compared to clean data, although it is still competitive with the rest of the machine learning models. We note that this pattern is consistent across all categories. On the other hand we observed that the fastText models performed similarly on cleaned and raw data. We speculate that the reason for this is that the major difference between cleaned and raw data is the lemmatization of the word tokens. Whereas the GRU-RNN operates on individual word embeddings for discrete word tokens, the fastText model has access to sub-word information via character n-grams (in the case of our models, up to 4-grams of characters). This means that the fastText models would be able to recover the lemmatized word information from sub-words, whereas the GRU-RNN operating on raw data would be unable to do so.

From our analysis of the validation performance, we observed that the hyper-parameters made marginal but not very significant differences in performance. For example, changing between 128 / 256 / 512 hidden nodes led to about a 0.05 difference in validation performance. Similarly, the difference between models use different embeddings also did not make a significant difference. Swapping between embeddings trained on Twitter or Wikipedia+Gigaword also led to about 0.05

difference in validation performance. Our intuition is that given the large corpus of data that both sets of embeddings are trained on, the precise domain of the training corpus does not matter as much assuming sufficiently diverse training data. For the fastText models, we found that increasing the number of character n-grams was somewhat more significant than word n-grams, which made minimal difference in performance. Increasing the number of character n-grams from 0 to 2 and 2 to 4 led to about a 2% improvement in validation performance each time. This is consistent with our speculation above regarding the difference between using clean and raw data. Hence, we conclude that sub-word information can be highly beneficial in improving performance in word classification tasks, and in doing so, we can forego the need for hand-crafted and deliberate feature preprocessing. That said, the best performing model in our analysis is still the GRU-RNN on cleaned data, so there is still some benefit to hand-cleaning data.

6.2 Error Analysis

We also analyze some of the worst errors made by our best model – Logistic Regression with $l2$ penalty. For brevity, we have limited our discussion to the `toxic` label.

One of the sentences in cleaned data was:
`"'fcw check list wp pw mos reliable source source avoid punk'"`
which was originally:
`'"\n\n FCW \n\nCheck the list at WP:PW/MOS. The reliable sources are there as are the sources to avoid.  !!  a Punk !!  "'.`
It was labeled toxic, but our model predicted its probability of toxic = 0.0088 (i.e., non-toxic), which is understandable because in the cleaned data none of the words are clearly negative in nature; words like "punk" change their connotation based on the context. Another instance was the sentence:
`'tno thank mate p s s offe'` which originally was: `'tno thanks mate p i s s offe'`
which actually is toxic, but due to the spaces in between the letters, the model doesn't identify it to be a toxic word and hence it predicts non-toxic.

On the other end of the spectrum, one of the (partial, in the interest of saving space) sentences with cleaned text (and some words being censored by us) was:
`'semi protect edit request 16 february 2015 state f**k hard not love f**k place movie catch feeling fall love kick bed sleep not cuddle shit eventually start catch feeling not catch tell ana dontt́ romance`
while the raw text was:
`"\n\n Semi-protected edit request on 16 February 2015 \n\nHe states"" I f**k hard I dont́ make love""\nThey f**k in various places throughout the movie. She catches feelings. He almost falls in love. He kicks her out of bed to sleep by herself cause he aint́ with that cuddle shit. \nHe too eventually starts catching feelings not before he catches himself and tells Ana "" i dontt́ romance.""`
which wasn't actually "toxic" in nature because it seems as if the person is editing an erotic story, but since it uses a lot of swear words, the model predicts it to be toxic with probability = 0.9801.

Finally, we also explored the relationship between the misclassification rate and `comment_length` to see if our model (presumably) performed better on shorter sentences compared to longer sentences. Interestingly, we didn't find any substantial difference and on average, the models seemed to perform equally well (or worse) on all targets, regardless of their lengths. Since `comment_length` was the most "importance" feature as shown in Figure 2, this once again shows that feature importance scores are often misleading.

References

[1] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *CoRR*, abs/1607.04606, 2016.

[2] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[3] Kyunghyun Cho. Strawman: an ensemble of deep bag-of-ngrams for sentiment analysis. *CoRR*, abs/1707.08939, 2017.

[4] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[6] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[7] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *CoRR*, abs/1607.01759, 2016.

[8] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[9] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[10] Sida Wang and Christopher D. Manning. Baselines and bigrams: Simple, good sentiment and topic classification. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2*, ACL '12, pages 90–94, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics.

7 Appendix

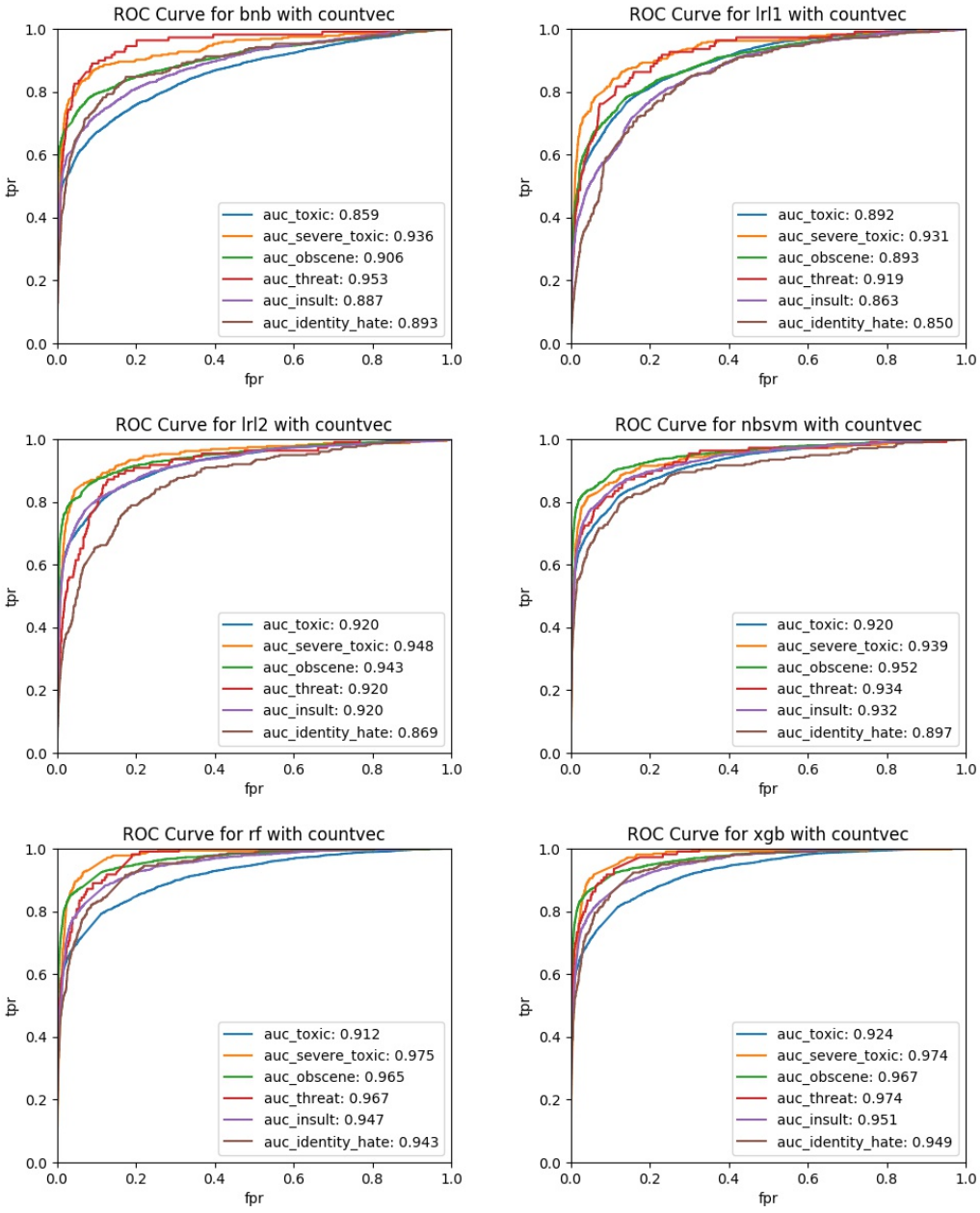

Figure 3: ROC Curve Plots for All Models with `CountVectorizer` *Without* Added Features

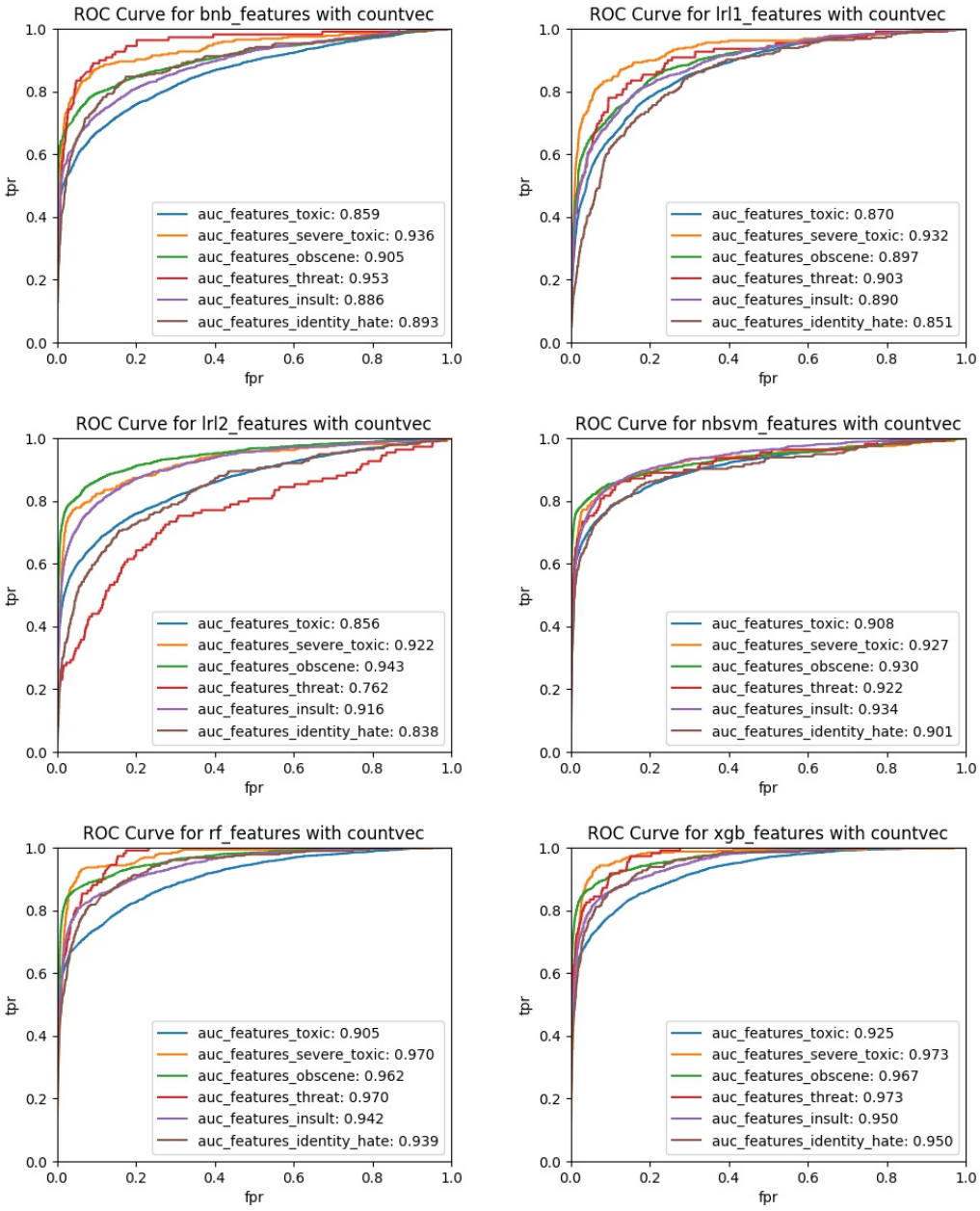

Figure 4: ROC Curve Plots for All Models with `CountVectorizer` *With* Added Features

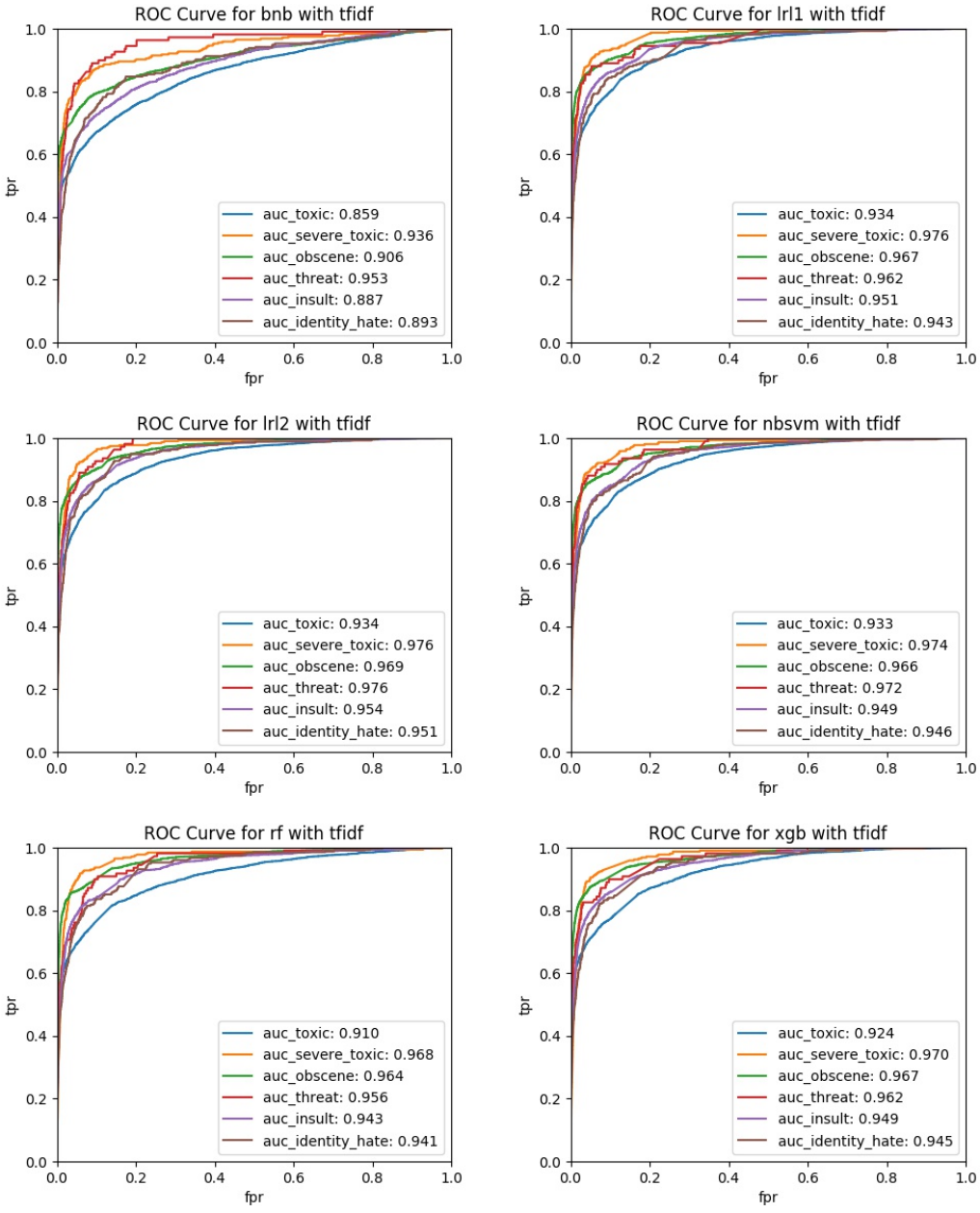

Figure 5: ROC Curve Plots for All Models with `TfIdfVectorizer` *Without* Added Features

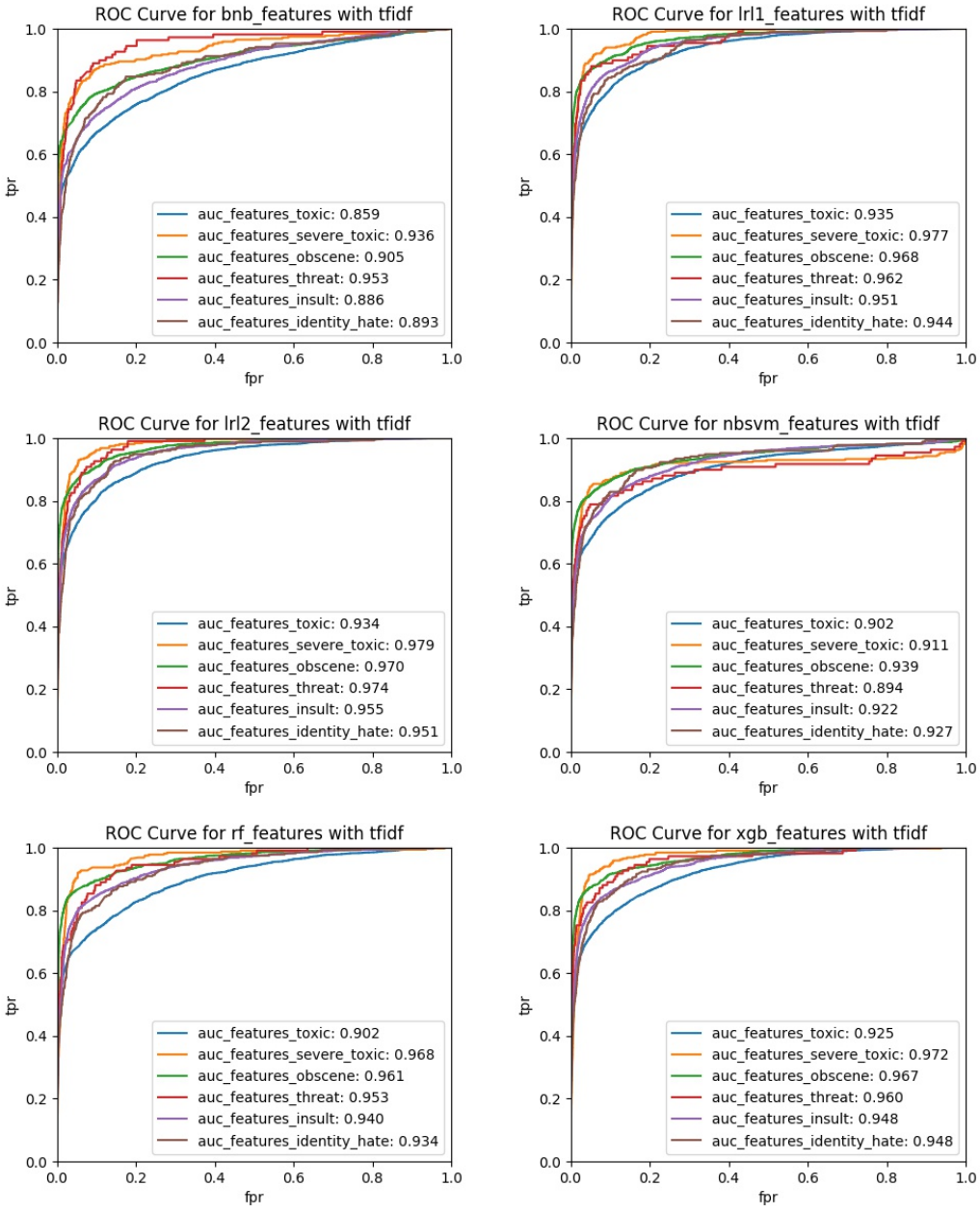

Figure 6: ROC Curve Plots for All Models with `TfIdfVectorizer` *With* Added Features

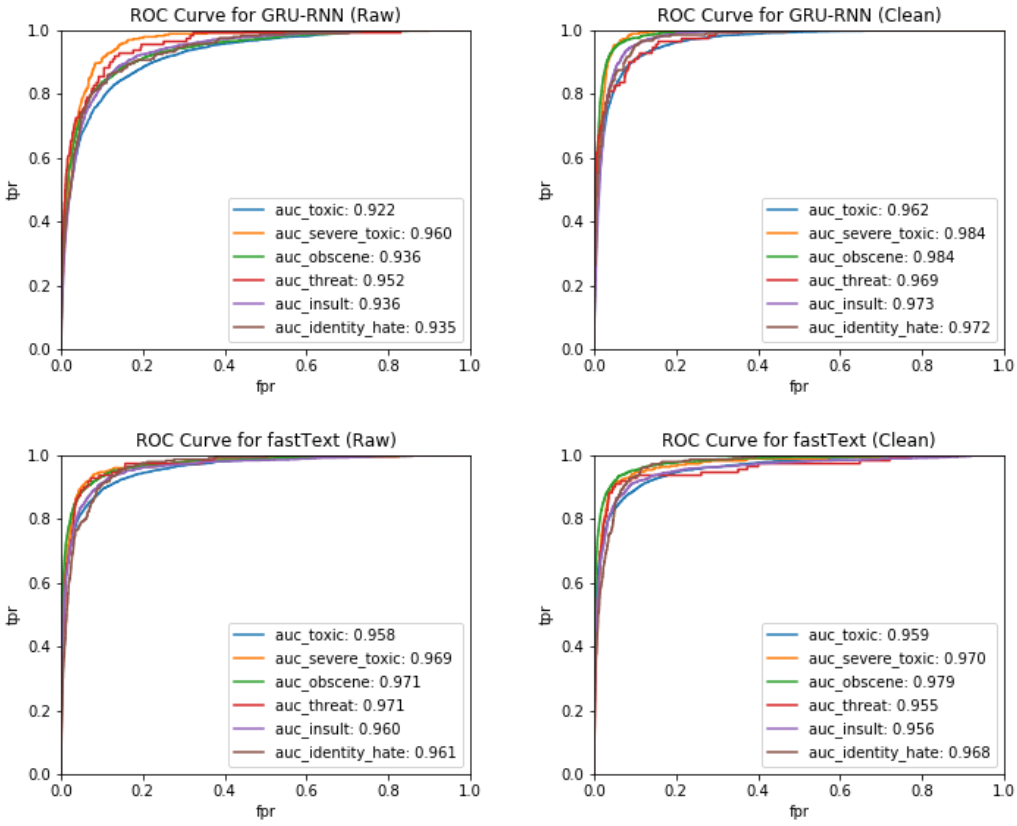

Figure 7: ROC Curve Plots for All Deep Learning Models

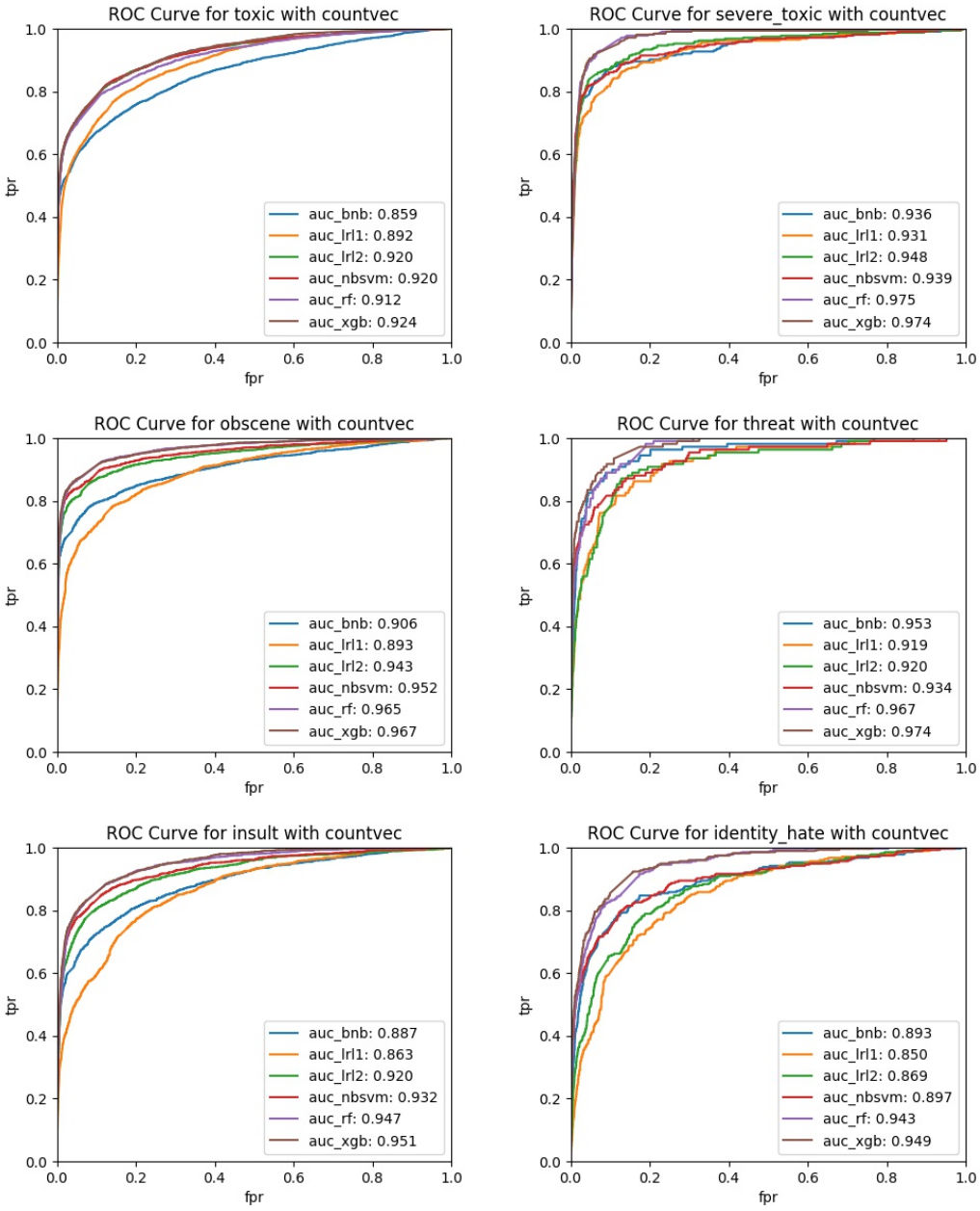

Figure 8: ROC Curve Plots for All Targets with `CountVectorizer` *Without* Added Features

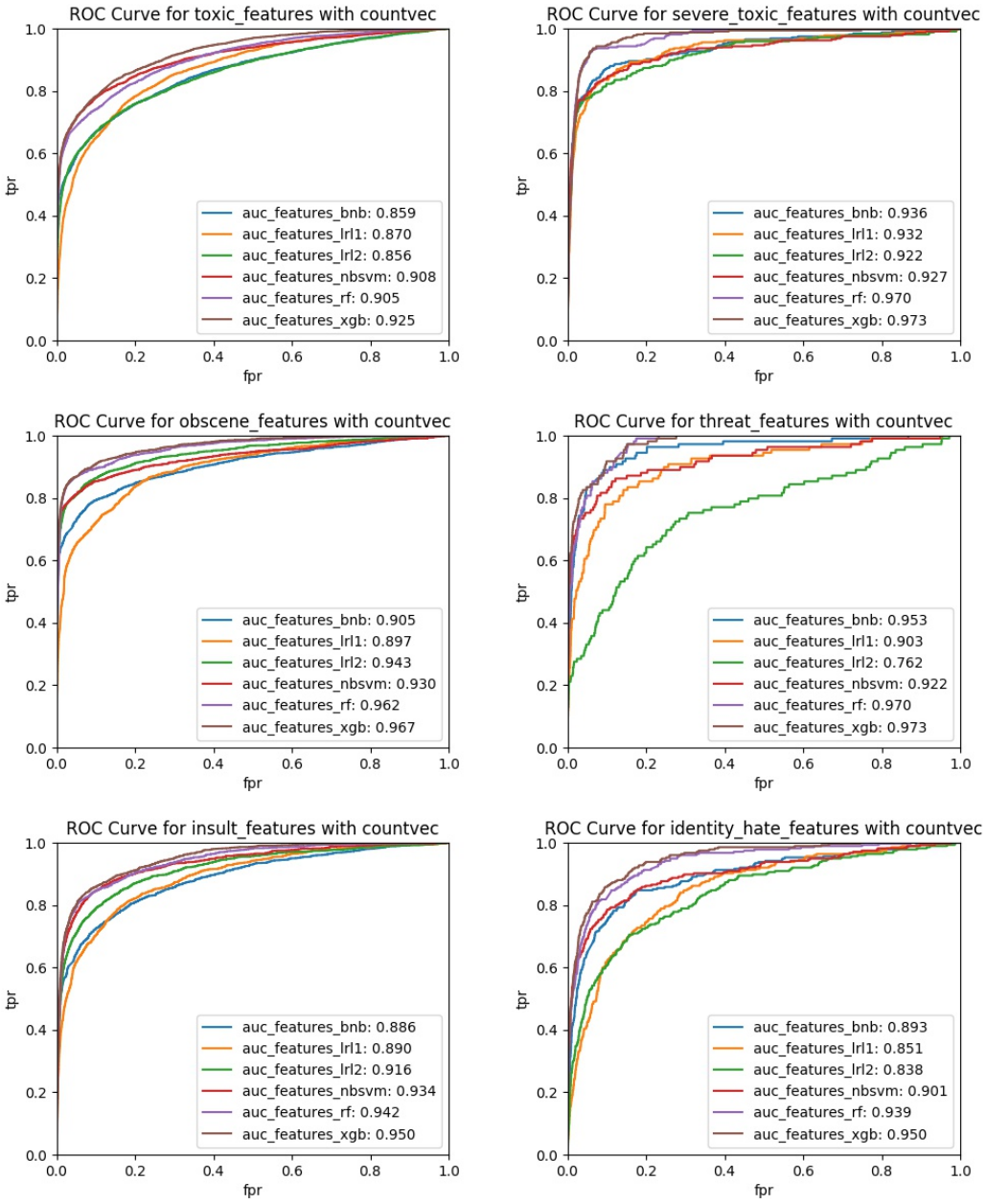

Figure 9: ROC Curve Plots for All Targets with `CountVectorizer` *With* Added Features

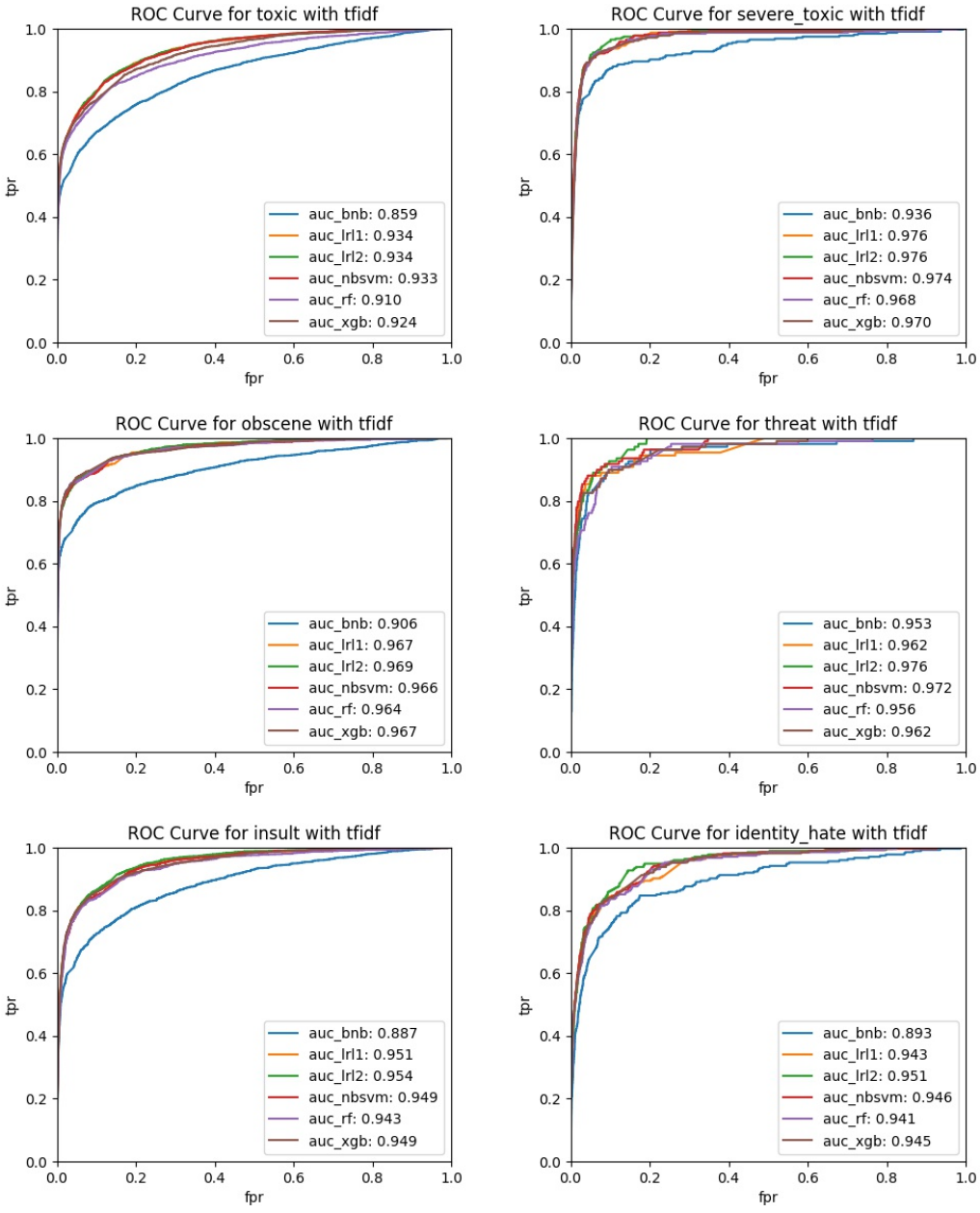

Figure 10: ROC Curve Plots for All Targets with `TfIdfVectorizer` *Without* Added Features

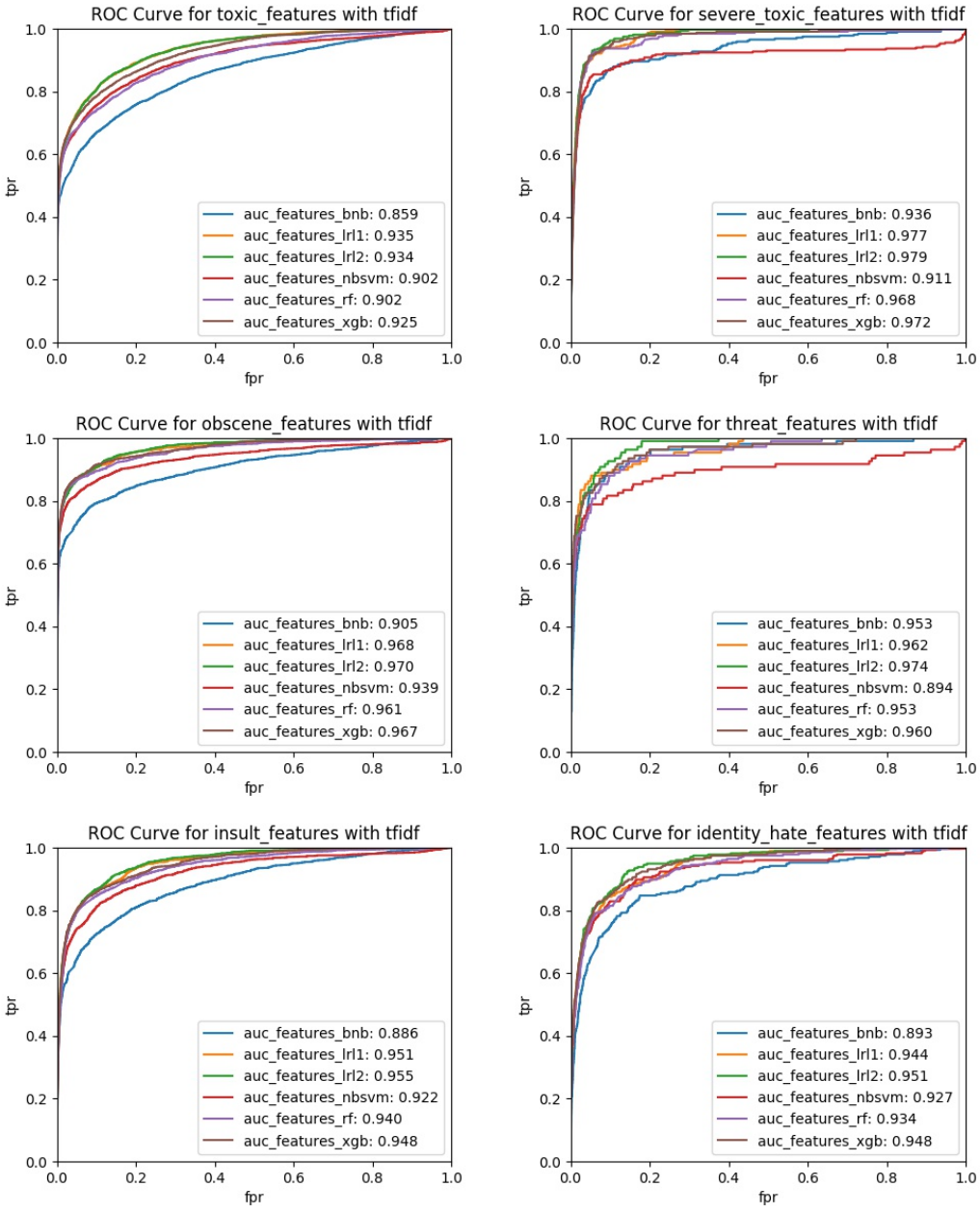

Figure 11: ROC Curve Plots for All Targets with `TfIdfVectorizer` *With* Added Features

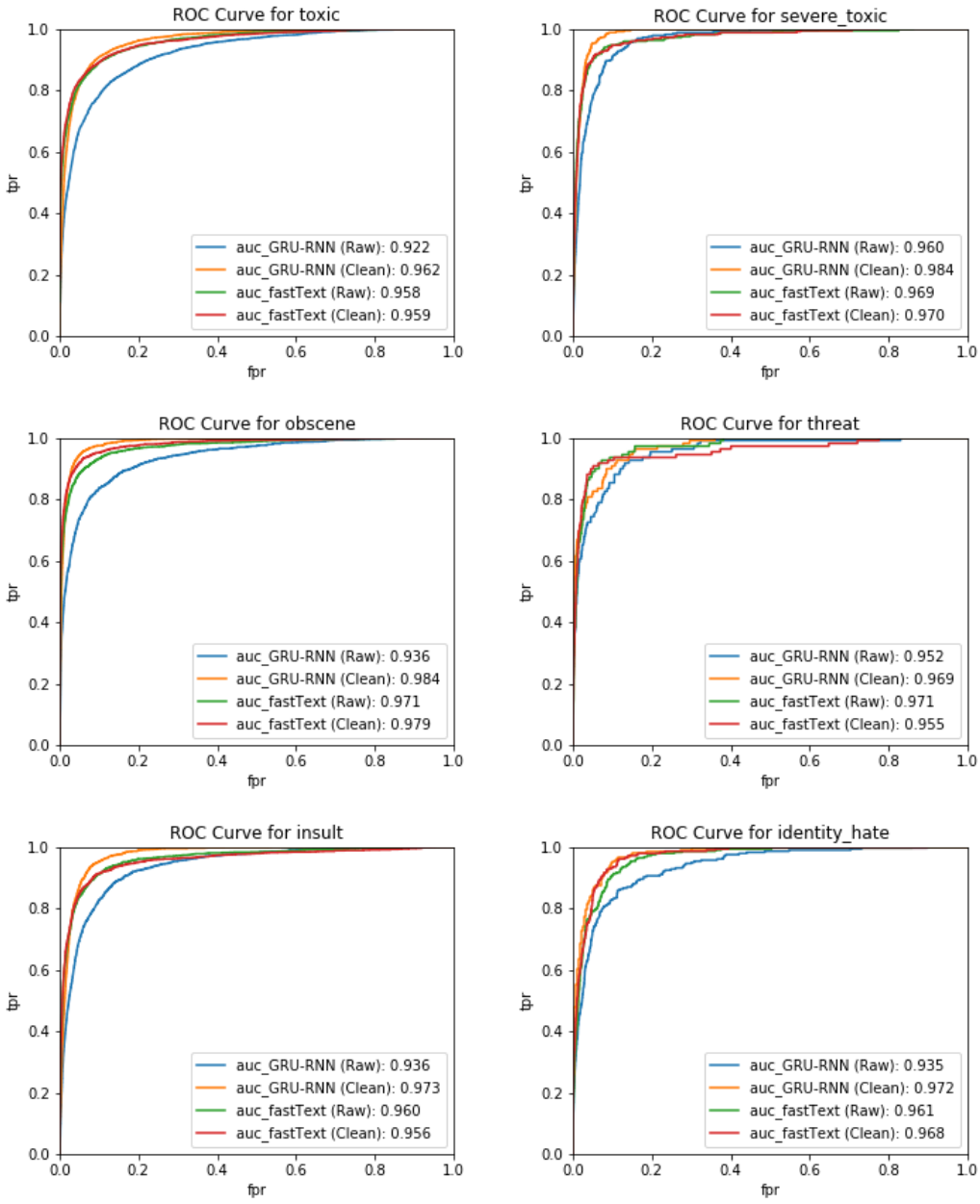

Figure 12: ROC Curve Plots for All Targets with Deep Learning Models