Detecting and Classifying Comment Toxicity at Scale

Yishan Li
MDSAI
University of Waterloo
Waterloo, ON, CDN
a356li@uwaterloo.ca

William Zou
MMath in Data Science
University of Waterloo
Waterloo, ON, CDN
wyzou@uwaterloo.ca

ABSTRACT

Toxic comments are often used in online discussions. We propose ensemble classifiers trained with Wikipedia's Toxic Comment dataset to detect and differentiate different levels of toxic comments. Our model extract n-grams as features and pass term frequency-inverse document frequency (TF-IDF) values to a majority voting ensemble. Our designed model is trained using Apache Spark and leverages its dataframe operations and ML pipeline to scale with big data. Our model is able to detect toxic comments with 93% accuracy and is able to categorize toxic comments into the categories hate_speech, toxic, insult and obscene with around 80% accuracy.

CCS CONCEPTS

• Machine Learning → Information extraction

KEYWORDS

Toxic Text Classification, TF-IDF, n-gram, voting ensemble, machine learning, Apache Spark

1 Introduction

Statistics show that every second, on average, around 6000 tweets are tweeted, which corresponds to over 350,000 tweets per minute, 500 million tweets per day, and 200 billion tweets per year [1]. In the past decade, the sharp increase in toxic content has become a major concern, as it can lead to lowered self-esteem, mental illness, and suicide of the target. Motivated by this, we propose an automated system to detect toxic comments to protect our online space.

In this paper, we propose an ensemble binary classification approach for toxic comment detection and a multilabel classifier based on one-vs-rest architecture for categorizing comments into specific toxic classes. We applied the techniques n-gram and TF-IDF to extract feature representation from the comments. We built 5 models in total: Logistic Regression, Naive Bayes, SVM, Decision Tree, and Gradient Boosted Tree. These feature extraction techniques and models have been used in literature with success [2, 3, 4]. Additionally, the models are simple and quick to train and have been implemented in Apache Spark's ML framework, making them ideal candidates to build a scalable solution on. We performed hyperparameter tuning with 5-fold cross-validation and saved the best model for future evaluation. Afterwards, we performed a comparative performance analysis between the models and the ensemble classifiers that uses the models. We claim the following:

1. Our proposed models are well-suited for toxic detection; however, due to the class imbalance problem, it is unable to categorize toxic comments effectively.
2. The training framework we provide is scalable to big data.
3. The 5 models we investigated have similar accuracy. Decision Tree and Gradient Boosted Tree are the most resource intensive to train and not recommended for a scalable training framework. Using an ensemble method can increase the performance slightly.
4. Optimal performance for toxic detection occurred when we used unigrams and bigrams for feature extraction (compared to higher-ordered n-grams), suggesting that toxic detection is a syntactic problem.
5. Undersampling and oversampling can address the class imbalance issue and improve the precision and recall of the classification of toxic comments, but not to a consumer usable degree.

2 Dataset

The dataset we used is the Wikipedia comments dataset sponsored by Jigsaw [5]. The dataset is labeled by human raters for toxicity behavior, and contains 7 categories in total: toxic, severe_toxic, obscene, threat, insult, identity_hate and clean. Each comment in the dataset can have multiple toxic labels.

As shown in Figure 1, our dataset is highly imbalanced. The severe skewness in the class distribution could influence the performance of our machine learning algorithm negatively, leading the model to ignore the minority class entirely. We combine the classes that have similar semantics in order to remove categories that have small amounts of data. A natural way of combining classes is to group all comments that do not fall under the clean label into one category (Fig 2). A comment that is not labelled as clean is categorized as toxic, otherwise it is labeled as a clean comment.

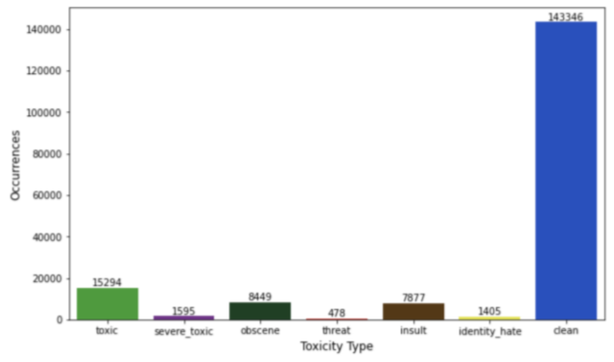


Figure 1: Distribution of toxicity types in the Wikipedia Comments Dataset

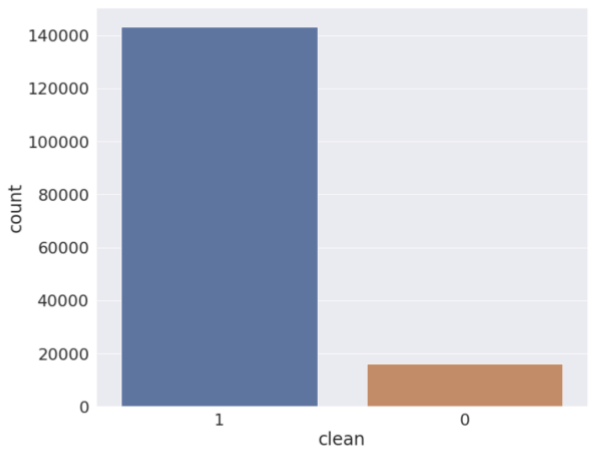


Figure 2: Distribution of clean/unclean data in the Wikipedia Comments Dataset for binary classification task.

To clean and standardize our dataset, we converted all the texts to lower cases, removed special characters and digits from each comment, and adjusted the white space pattern to ensure only one white space is found between words [3]. We then employed the Snowball Stemmer algorithm to reduce the inflectional forms of the words. Finally, we removed English stopwords from the comments. After applying the transformations onto the dataset, we randomly shuffled and split the dataset into an 80% training set and 20% testing set.

3 Feature extraction

During the information retrieval phase, we extracted the n-gram features for each of the comments. The technique TF-IDF is applied to the sequences of n-grams to weigh the importance of the features [2, 3, 6]. $TF(t,d)$ measures how frequently a word t occurs in comment d. This operation is achieved using the PySpark built-in library HashTF, which maps a sequence of the term to the term frequency using a hashing trick.

TF-IDF is a statistical measure that is used to evaluate how important a particular word is to a comment in a large corpus, which is achieved by multiplying IDF and TF. A high TF-IDF score is achieved by a high term frequency in a comment and low frequency of the term among all comments.

We provide a brief mathematical explanation of how TF-IDF works [6]. We denote a term by t, a document by d, and a corpus by D. We first measure the value of $TF(t,d)$ which shows how frequently a term t appears in document d. We then measure the inverse document frequency (IDF) which indicates how much information a word carries. $DF(t,D)$ is the number of documents that contain the term t. Intuitively, IDF could be computed using Equation 1.

$$IDF(t,D) = |D|/DF(t,D) \tag{1}$$

However, certain words have a very low IDF score since they appear in most of the documents and carry little information for the document. So, taking logarithm will rescale the IDF. If a term appears in all documents, this will result in $IDF(t,D) = 0$. As we cannot divide by 0, we smooth the value by adding 1 to the denominator, as shown in Equation 2.

$$IDF(t,D) = \log \frac{|D|+1}{DF(t,D)+1} \tag{2}$$

After obtaining $IDF(t,D)$ and $TF(t,d)$ for word d, we multiply the scores together and obtain a final TF-IDF score. Each word in a particular document contains a TF-IDF score indicating the importance of the word in a comment.

$$TFIDF(t,d,D) = TF(t,d) \times IDF(t,D) \tag{3}$$

4 Model

We considered the following machine learning algorithms used for text classification: Logistic Regression, Naive Bayes, Support Vector Machines, Decision Trees and Gradient Boosted Trees. We trained the model on the training dataset by performing grid search on a set of hyperparameters with 5-fold cross validation. This was trained with unigram feature extraction.

We then created a voting ensemble from these models. Because training trees takes a large amount of memory, we decided to remove it from the voting ensemble since it did not give a significant increase in accuracy.

After tuning the hyperparameters in the models, we evaluated the performance of our voting ensemble with different order n-grams. In addition to model accuracy, we also considered the relative memory usage required to train each of our models.

Model	Accuracy
Logistic Regression	90.92
SVM	91.53
Decision Tree	90.67
Naive Bayes	91.82
Gradient boosted tree	91.65

Table 1: Results of evaluating models on the test set after tuning them with 5-fold cross validation on the training set with unigram feature extraction.

Model	Accuracy
Voting ensemble (with trees)	92.59
Voting ensemble (without trees)	91.86

Table 2: Results of evaluating voting ensemble methods on the test set with methods shown in Table 1.

5 Results

Table 1 shows the result of each model, and Table 2 shows the result of our voting ensemble. After tuning, each model achieved approximately the same accuracy. Most models needed an n-gram feature vector representation of length 1000 and needed less than 10 iterations to converge. A more detailed view of our model parameters can be found in our Gitlab repository in models.

The results for different n-gram models are shown in Table 3 and Figure 3. Bigrams performed the best. However, comparing the testing metrics for unigrams and the testing metrics for bigrams, we conclude that the metrics for bigrams is not significantly better, and we can use the unigram model if our computational resources are limited. These results suggest that whether or not a comment is clean is a syntactic problem rather than a semantic one, and most toxic phrases can be identified with 1 or 2 words. The majority voting technique improved the accuracy of our best performing model (Naive Bayes) by approximately 0.8%.

	Accuracy	Precision	Recall
unigram	92.59	92.58	99.75
bigram	93.05	93.46	99.21
trigram	90.43	91.43	99.12

Table 3: Results of evaluating voting ensemble (without trees) on the test set after varying n-grams feature parameter

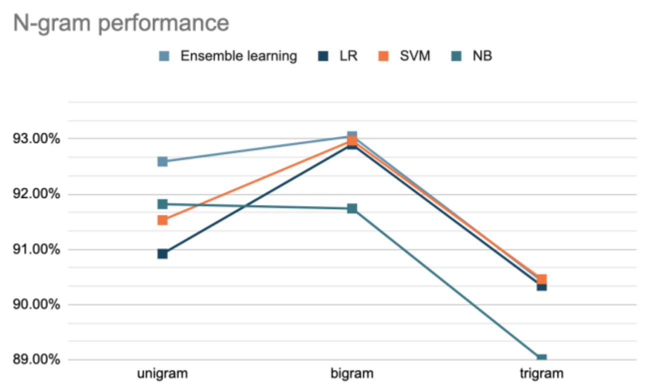


Figure 3: The figure shows comparative analysis of ensemble learning, Logistic Regression, SVM and Naïve Bayes with unigram, bigram and trigram feature extraction

6 Training with Apache Spark

Our model is designed to be trained with Apache Spark. Data is loaded into Spark's dataframe then stemmed using column operations defined with Sparks' User Defined Functions. How this is done is shown in Figure 4. Afterwards, it is passed through Spark's ML pipeline, which applies the tokenization, the stopword removal, the n-gram and TF-IDF feature extraction, and the final fitting of the model.

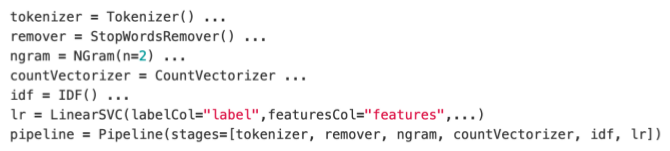
```
stemmer = SnowballStemmer("english")
def stemming(sentence):
    stemSentence = ""
    for word in sentence.split():
        stem = stemmer.stem(word)
        stemSentence += stem
        stemSentence += " "
    stemSentence = stemSentence.strip()
    return stemSentence

stemmer_udf = udf(lambda line: stemming(line), StringType())
train = train.withColumn("comment_text", stemmer_udf("comment_text"))
```

Figure 4: Code to apply stemming to a dataframe using a Spark User Defined Function

```
tokenizer = Tokenizer() ...
remover= StopWordsRemover() ...
hashingTF = HashingTF().setNumFeatures(...) ...
idf = IDF() ...
lr = LinearSVC(labelCol="label", featuresCol="features", ...)
pipeline=Pipeline(stages=[tokenizer, remover, hashingTF, idf, lr])
```

Figure 5: ML pipeline using HashingTF for feature extraction



```
tokenizer = Tokenizer() ...
remover = StopWordsRemover() ...
ngram = NGram(n=2) ...
countVectorizer = CountVectorizer ...
idf = IDF() ...
lr = LinearSVC(labelCol="label",featuresCol="features",...)
pipeline = Pipeline(stages=[tokenizer, remover, ngram, countVectorizer, idf, lr])
```

Figure 6: ML pipeline using n-gram and CountVectorizer for feature extraction

- Stop words removal: Stop words are the most common words in a language. StopWordsRemover takes in a sequence of strings and drops all the predefined stop words in the sequences.
- Feature extraction:
    - TF-IDF feature extraction: we use Spark's HashingTF function to transform tokenized words to feature vector representation, then fit the feature vectors in Spark's IDF estimator to calculate the TF-IDF score. Spark's HashingTF outperforms Spark's CountVectorizer method as the former requires only one pass over the data while the latter requires multiple. A sketch of how to implement this is included in Figure 5.
    - n-gram and TF-IDF feature extraction: we use Spark's n-gram to transform input to sequence of n-grams, then fit the sequence of n-grams to CountVectorizer which aims to convert a collection of input sequence to vectors of token counts. The output of CountVectorizer is fed to the IDF estimator to calculate the TF-IDF score. A sketch of how to implement this is included in Figure 6.

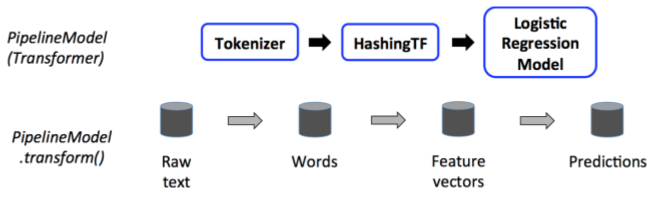

PipelineModel (Transformer)
Tokenizer
HashingTF
Logistic Regression Model
PipelineModel .transform()
Raw text
Words
Feature vectors
Predictions

Figure 7: Apache Spark model for evaluating predictions at test time, taken from Spark's ML pipeline documentation [7]. Tokenizer, HashingTF and the Logistic Regression Model are combined into one pipeline model which can be applied onto raw text to make predictions.

The models are saved to disk and can be loaded as a Spark ML Pipeline and applied as a Spark transformation, allowing for faster evaluation. A workflow of this is shown in Figure 7.

7 Multilabel Classifier

We applied what we learned from our binary classification task to a multilabel classification task. To address the problem of the imbalanced dataset, we combined classes threat, identity_hate and severe_toxic into one class called hate_speech since these categories are the most severe and have the least number of comments. Our model needs to identify comments as toxic, obscene, insult, hate_speech and clean.

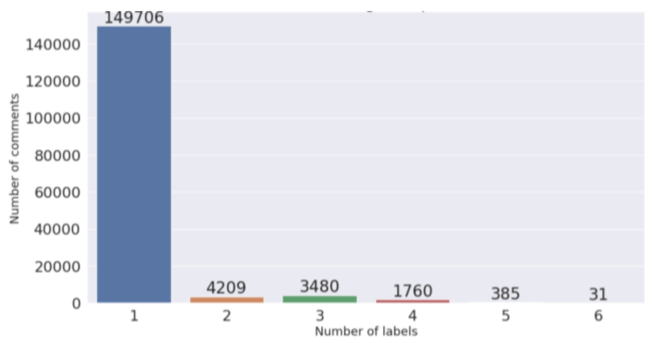


Figure 8: Distribution of number of labels associated with each comment

Since each comment could be labeled into multiple classes (Figure 8), we applied one-vs-rest architecture to classify the toxic comments, which involves splitting the multi-label dataset into several binary classification tasks.

	Accuracy	Precision	Recall
hate_speech	98.06	88.9	3.74
insult	95.94	83.07	20.61
obscene	96.51	92.35	34.27
toxic	92.85	92.31	26.02

Table 4: Results of predicting categories in the test set with tuned voting ensemble (without trees) with unigram feature extraction.

Predicted Class	Actual Class	
	hate_speech	non_hate_speech
hate_speech	0.037	0.000
non_hate_speech	0.963	1.000

Table 5: Confusion matrix for the evaluated test set on the tuned voting ensemble (without trees) with unigram feature extraction.

We approached the problem by training the same voting ensemble in Section 4 to predict each binary classification task. The experiment is conducted using the best set of model parameters saved from Section 5. The results in Table 4 show extremely low recall. This is because even after combining classes, we still do not have enough data in the minority classes, and our model is heavily in favour of predicting clean comments. We can clearly see this in

Table 5. To solve this issue, we tried oversampling and undersampling [8].

7.1 Undersampling

We randomly removed comments from the majority class to make the dataset balanced. We applied our multilabel classifier to the undersampled dataset. Table 7 shows that after undersampling the model has a far better accuracy at predicting hate_speech. We found that undersampling greatly increases the recall of our model but degrades the overall testing accuracy by more than 10% (Table 6). A limitation of undersampling is that the data removed from the majority classes might be considered essential to fitting a robust decision boundary. This technique cannot guarantee to preserve the information-rich data and is likely the main reason that undersampling degrades our model performance. One side benefit to undersampling is that it greatly reduces the resources needed to train our model since it takes only a subset of the original data. As an example, we reduced the amount of data our model trained on by 20 times the original amount for hate_speech.

	Accuracy	Precision	Recall
hate_speech	80.42	80.71	81.49
insult	77.44	73.50	83.70
obscene	74.73	71.32	83.39
toxic	78.21	76.90	80.05

Table 6: Results of predicting categories in the test set with tuned voting ensemble (without trees) with unigram feature extraction after undersampling.

Predicted Class	Actual Class	
	hate_speech	non_hate_speech
hate_speech	0.815	0.207
non_hate_speech	0.185	0.793

Table 7: Confusion matrix for the evaluated test set on the tuned voting ensemble (without trees) with unigram feature extraction after undersampling.

7.2 Oversampling

We randomly duplicated comments in the minority class to make the dataset balanced. We applied our multilabel classifier on the oversampled dataset. Table 9 shows that after oversampling the model has a far better accuracy at predicting hate_speech. Similar to undersampling, oversampling increases the recall of our model but degrades the overall testing accuracy on the dataset by more than 10% (Table 8). Oversampling retains all the information of the original dataset, but randomly duplicating the minority class might cause our model to overfit, causing it to fail to generalize well on the test set. We note that while oversampling has better results than undersampling, a major limitation is that it slows down the training process due to the increase in data. In the future, instead of simply duplicating data in oversampling, we could rely on techniques such as text augmentation, sentence shuffling, and syntax tree manipulation to artificially create comments in the minority classes [9].

	Accuracy	Precision	Recall
hate_speech	85.48	83.39	88.47
insult	76.84	73.009	84.99
obscene	76.36	72.73	94.49
toxic	80.63	79.86	82.28

Table 8: Results of predicting categories in the test set with tuned voting ensemble (without trees) with unigram feature extraction after oversampling.

Predicted Class	Actual Class	
	hate_speech	non_hate_speech
hate_speech	0.885	0.175
non_hate_speech	0.115	0.825

Table 9: Confusion matrix for the evaluated test set on the tuned voting ensemble (without trees) with unigram feature extraction after oversampling.

8 Conclusion

In this paper, we developed a training framework to detect and classify unclean comments by extracting features using n-gram and TF-IDF and passing it to a voting ensemble of Logistic Regression, Naïve Bayes and SVM models. Our model is able to detect unclean comments with 93% accuracy and is able to categorize unclean comments into the categories hate_speech, toxic, insult and obscene with 80% accuracy. By leveraging Apache Spark, our training framework is designed to scale well with the amount of data provided. Optimal performance for toxic detection occurred when we used unigrams and bigrams instead of higher-ordered n-grams for feature extraction, suggesting that toxic detection is a syntactic problem. To improve the recall of the multilabel classifier, we have tried techniques like undersampling and oversampling. The results show that the two techniques improve the recall significantly but degrades the overall test accuracy. To solve the class imbalance problem without compromising the overall performance of our model, we need to increase the minority class with good samples. One way we could accomplish this is by injecting unclean comments from other datasets into our dataset. Additionally, while the difference between clean and unclean data may be largely syntactic, the difference between a threat and an insult could be semantic in nature. This motivates us to explore other pre-trained word embeddings capable of capturing semantic information (i.e., glove, word2vec, BERT). Most of the research we have surveyed shows that using these pre-trained word embeddings boosts the model performance in text analysis tasks [10, 11, 12].

REFERENCES

[1] Tweeter Usage Statistics. Internet Live Stats. Retrieved from: https://www.internetlivestats.com/twitter-statistics/

[2] Gaydhani et al. 2018. Detecting Hate Speech and Offensive Language on Twitter using Machine Learning: An N-gram and TFIDF based Approach. arXiv:1809.08651. Retrieved from https://arxiv.org/pdf/1809.08651.pdf

[3] Fahim Mohammad. 2018. Is preprocessing of text really worth your time for toxic comment classification. Int'l Conf. Artificial Intelligence, ICAI'18,447-453. Retrieved from: https://csce.ucmss.com/cr/books/2018/LFS/CSREA2018/ICA4290.pdf

[4] Zaheri et al. 2020. Toxic Comment Classification. SMU Data Science Review 3, 1, Article 13 (2020), 16 pages. Retrieved from: https://scholar.smu.edu/datasciencereview/vol3/iss1/13/

[5] Ellery Wulczyn, Nithum Thain, Lucas Dixon. 2016. Wikipedia Detox. DOI: https://doi.org/10.6084/m9.figshare.4054689.

[6] Feature Extraction and Transformation - RDD-based API. Apache Spark. Retrieved from: https://spark.apache.org/docs/latest/mllib-feature-extraction.html

[7] ML Pipeline. Apache Spark. Retrieved from: https://spark.apache.org/docs/latest/ml-pipeline.html

[8] Prabhjot Kaur and Anjana Gosain. 2018. Comparing the Behavior of Oversampling and Undersampling Approach of Class Imbalance Learning by Combining Class Imbalance Problem with Noise. ICT Based Innovations, 23-30. Retrieved from: https://www.researchgate.net/publication/320160451_Comparing_the_Behavior_of_Oversampling_and_Undersampling_Approach_of_Class_Imbalance_Learning_by_Combining_Class_Imbalance_Problem_with_Noise

[9] Mnasri Maali. 2019. Text Augmentation for Machine Learning Tasks: How to grow your text datset for classification? Retrieved from: https://medium.com/opla/text-augmentation-for-machine-learning-tasks-how-to-grow-your-text-dataset-for-classification-38a9a207f88d

[10] Manav Kohli, Emily Kuehler, John Palowitch. Paying attention to toxic comments online. Retrieved from: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/reports/6856482.pdf

[11] A. Akshith Sagar, J. Sai Kiran. 2020. Toxic Comment Classification using Natural Language Processing. International Research Journal of Engineering and Technology 7, 6.(June 2020) 6007-6010. Retrieved from: https://www.irjet.net/archives/V7/i6/IRJET-V7I61123.pdf

[12] Pavel, Monirul. 2020. Toxic Comment Classification Implementing Natural Language Processing and Convolutional Neural Networks with Word Embedding Technique. Retrieved from: https://www.researchgate.net/publication/343280364_Toxic_Comment_Classification_Implementing_Natural_Language_Processing_and_Convolutional_Neural_Networks_With_Word_Embedding_Technique