Abusive language detection in games

- Spataru Horia-Stefan -

Introduction:

Abusive online content, including hate speech and harassment, has become a pressing concern across digital platforms. Academics, policy-makers, and tech companies alike recognize the need for effective detection systems. However, the success of such systems hinges on the quality of their training datasets. A fundamental principle in computer science is said to be garbage in, garbage out. In other words, the accuracy and robustness of abusive language classifiers depend on the data used for training.

In this paper, we review publicly available training datasets specifically created for training abusive language classifiers. Our analysis sheds light on what these datasets contain, how they've been annotated, and the formulation of tasks. Additionally, we introduce a dedicated website, hatespeechdata.com, which catalogs abusive language data. As we delve into the challenges and opportunities of open science in this field, we also address the social and ethical risks associated with dataset sharing. Finally, we provide evidence-based recommendations for practitioners embarking on the creation of new abusive content training datasets.

Graph Convolutional Networks (GCNs) for Abusive Language Detection:

Paper: "Abusive Language Detection with Graph Convolutional Networks" by Pushkar Mishra, Marco Del Tredici, Helen Yannakoudakis, and Ekaterina Shutova.

Abstract: This approach goes beyond traditional community-based profiling by modeling both the structure of online communities and the linguistic behavior of users within them. By leveraging graph convolutional networks, it achieves state-of-the-art performance in abusive language detection.

Key Insight: Combining community structure and linguistic features improves model effectiveness.

Unsupervised Domain Adaptation:

Paper: “Unsupervised Domain Adaptation in Cross-corpora Abusive Language Detection”.

Context: Existing models perform well within their training scenarios but struggle when faced with abusive comments from different contexts.

Importance: Adapting to newly collected comments is crucial. Human annotation is time-consuming, so adaptable models are valuable.

Literature Review on Hate Speech Detection:

Paper: “Automatic Hate Speech Detection using Natural Language Processing: A state-of-the-art literature review” .

Abstract: This review highlights the harmful presence of abusive behavior and hate speech on social networks. It provides insights into various approaches and challenges.

Considerations: Understand the nuances of hate speech detection and learn from the existing body of research.

Neural Character-based Composition Models:

Paper: “Neural Character-based Composition Models for Abuse Detection” 4.

Observation: Current state-of-the-art approaches based on recurrent neural networks often struggle with out-of-vocabulary (OOV) words.

Recommendation: Address OOV challenges explicitly to enhance model robustness.

The code:

The python code essentially preprocesses text data, creates TF-IDF natural language processing technique used to evaluate the importance of different words in a sentence, features, trains a logistic regression model, evaluates its performance, and saves predictions for the test set. The final output is a CSV file containing predicted intent class labels for the test data, as well as accuracy analysis in the IDE Terminal.

Load Datasets:

The code reads three CSV files (CONDA_train.csv, CONDA_valid.csv, and CONDA_test.csv) into pandas DataFrames: train_data, valid_data, and test_data.

Preprocess Text:

A function called preprocess_text is defined to preprocess the text data:

It converts text to lowercase.

Removes special characters using regular expressions.

Tokenizes the text into individual words.

The function is applied to the ‘utterance’ column in each dataset, creating a new column called ‘processed_utterance’ containing preprocessed text.

Remove Stopwords:

Stopwords (common words like “the,” “and,” “is,” etc.) are removed from the processed utterances using the NLTK stopwords list.

The resulting cleaned text is stored in the ‘processed_utterance’ column.

Create TF-IDF Features:

A TfidfVectorizer is initialized with a maximum of 1000 features.

It transforms the processed utterances into a sparse matrix of TF-IDF features:

Each row corresponds to a document (utterance).

Each column corresponds to a unique word (feature).

The value in each cell represents the TF-IDF score of the word in that document.

The transformed matrices are stored in X_train, X_valid, and X_test.

Prepare Target Labels:

The target labels ('intentClass') are extracted from the 'train_data' and 'valid_data' DataFrames and stored in y_train and y_valid.

Train a Logistic Regression Model:

A logistic regression model is initialized.

The model is trained using the features in X_train and the corresponding labels in y_train.

Predictions on Validation Set:

The trained model predicts the intent class labels for the validation set (X_valid).

The predicted labels are stored in y_valid_pred.

Evaluate Model Performance:

The code calculates the validation accuracy using accuracy_score.

It also generates a classification report (precision, recall, F1-score, etc.) using classification_report.

Predictions on Test Set:

The trained model predicts the intent class labels for the test set (X_test).

The predicted labels are stored in y_test_pred.

Save Predictions to CSV:

The predicted intent class labels for the test set are added as a new column (‘predicted_intentClass’) in the ‘test_data’ DataFrame.

The updated DataFrame is saved to a CSV file named ‘CONDA_test_predictions.csv’.

Results:

Validation Accuracy:

The overall accuracy of the model on the validation set is approximately 90.54%.

This means that the model correctly predicts the intent class for about 90.54% of the validation examples.

Precision:

Precision measures the proportion of true positive predictions (correctly predicted instances) out of all positive predictions (both true positives and false positives).

For each class (A, E, I, O), precision is calculated as:Precision=True Positives + False PositivesTrue Positives

Recall (Sensitivity):

Recall measures the proportion of true positive predictions out of all actual positive instances (true positives and false negatives).

For each class, recall is calculated as:Recall=True Positives + False NegativesTrue Positives

F1-Score:

F1-score is the harmonic mean of precision and recall.

It balances precision and recall, especially when dealing with imbalanced datasets.

For each class, F1-score is calculated as:F1-Score=2×Precision + RecallPrecision×Recall

Support:

The ‘support’ column indicates the number of instances (examples) for each class in the validation set.

Class Interpretation:

Let’s break down the results for each class:

A (Action):

Precision: 0.82 (82%)

Recall: 0.63 (63%)

F1-score: 0.71

There are 580 instances of class A in the validation set.

E (Explicit):

Precision: 0.91 (91%)

Recall: 0.74 (74%)

F1-score: 0.82

There are 1183 instances of class E.

I (Implicit):

Precision: 0.95 (95%)

Recall: 0.60 (60%)

F1-score: 0.74

There are 582 instances of class I.

O (Other):

Precision: 0.91 (91%)

Recall: 0.98 (98%)

F1-score: 0.94

There are 6629 instances of class O.

Macro Avg and Weighted Avg:

The macro average (macro avg) considers equal weight for each class.

The weighted average (weighted avg) accounts for class imbalance by considering the number of instances in each class.

Summary:

The model performs well overall, with high precision and recall for class O (Other).

Class I (Implicit) has lower recall, indicating that some implicit instances are being missed.

Class A (Action) has relatively lower precision, suggesting that some non-action instances are being misclassified as action.

The weighted average F1-score (0.90) indicates good overall performance.