

# CS 5785 Homework 2

Cornell Tech, Fall 2023

Tian Jin<sup>\*†</sup>

Yufan Zhang<sup>\*</sup>

September 2023

## 1 Binary Classification on Text Data

### 1.1 Basic Statistics of the Data

In this [Kaggle competition](#), we are given a dataset of 7613 tweets as training data and 3263 tweets as test data, each labeled as either a disaster or not. The goal is to build a binary classification model to predict whether a tweet is about a disaster or not. There are approximately 42.97% (3271) of the tweets labeled as disaster and 57.03% (4342) of the tweets labeled as not disaster in the training data. The test data is unlabeled and the prediction results will be evaluated by the accuracy of the prediction. Listing ?? shows the code to get the basic statistics of the dataset.

---

Listing 1: Code to get the basic statistics of the dataset

### 1.2 Splitting the Training Data

We split the training data into training and development sets with a ratio of 0.7:0.3. The training set is used to train the model and the development set is used to evaluate the model. The code to split the training data is shown in Listing ??.

---

Listing 2: Code to split the training data

### 1.3 Data Preprocessing

#### 1.3.1 Handle hashtags

In the first stage of the data preprocessing phase, special attention was given to the handling of hashtags. Hashtags often encapsulate key information but are written concatenated, such as “#World-News”. Since the objective is to extract meaningful features from the text using Bag-of-Words (BoW)

---

<sup>\*</sup>Both authors contributed equally to this research.

and N-Gram models, it is essential to break down these hashtags into their constituent words. For example, the hashtag “#WorldNews” would be split into “World” and “News”. This allows the feature extraction methods to recognize and give appropriate weight to each individual term. Concurrently, the hashtag symbol “#” is removed to ensure that the split words are treated the same way as other words in the tweet. Listing ?? shows the code to handle hashtags.

---

Listing 3: Code to handle hashtags

### 1.3.2 Lowercase all text

The next step in our data preprocessing pipeline is to convert all the text within the tweet data to lowercase. This is done to ensure uniformity and to eliminate any case-based discrepancies that could affect the feature extraction and the subsequent machine learning model. For example, the words “Science,” “SCIENCE,” and “science” may appear differently to a computer algorithm, but they all signify the same concept. This simple yet impactful modification aids in creating a more robust feature set for the machine learning algorithms to leverage. Listing ?? shows the code to lowercase all text.

---

Listing 4: Code to lowercase all text

### 1.3.3 Strip URLs and Mentions

The next step is the removal of URLs and mentions from the tweet text data. URLs often do not contribute meaningful information for text classification in the context of our project, and they may introduce noise that hampers the effectiveness of the machine learning models. Similarly, mentions—indicated by the “@” symbol followed by a username—are usually specific to individual users and do not add generalizable value to the classification task. Therefore, both URLs and mentions are extracted and removed from the tweets during this preprocessing phase. By eliminating these elements, we aim to simplify the text data and focus on the words and phrases that are most relevant for accurate classification. This step enhances the clarity of the dataset, making it easier for the Bag-of-Words and N-Gram models to identify meaningful features. Listing ?? shows the code to strip URLs and mentions.

---

Listing 5: Code to strip URLs and mentions

### 1.3.4 Lemmatization

The subsequent step involves the lemmatization of words using the [Natural Language Toolkit \(nltk\)](#). Lemmatization is the process of converting words into their base or root form, which aids in reducing the dimensionality of the feature space. For example, the words ‘running’, ‘runs’ and ‘ran’ would all be simplified to their lemma, ‘run’. This is particularly important for our project, as we aim to perform text classification using BoW and N-Gram models. Listing ?? shows the code to lemmatize words.

---

Listing 6: Code to lemmatize words

### 1.3.5 Remove punctuation

The next step is the removal of punctuation marks from the tweet text. Punctuation often adds structural complexity but usually does not contribute to the semantic meaning when text data is analyzed using Bag-of-Words and N-Gram models. By stripping the text of the punctuation marks, we create a cleaner, more uniform dataset that is easier to work with. Listing ?? shows the code to remove punctuation.

---

Listing 7: Code to remove punctuation

### 1.3.6 Remove non-alphabetic characters

The next preprocessing step entails the removal of all non-alphabetic characters from the tweet text. This includes numerical digits and any special characters that have not already been eliminated in earlier steps. The rationale behind this action is to focus solely on the words for our text classification task, as we are utilizing Bag-of-Words and N-Gram models for feature extraction. Non-alphabetic characters can introduce noise and complexity into the dataset without contributing meaningful information for classification. Listing ?? shows the code to remove non-alphabetic characters.

---

Listing 8: Code to remove non-alphabetic characters

### 1.3.7 Remove stopwords

The subsequent stage involves the removal of stopwords using the `mltk`. Stopwords are commonly occurring words such as ‘and,’ ‘the,’ and ‘is,’ which, although essential for human communication, usually do not provide significant information for text classification tasks. Given our aim to use Bag-of-Words and N-Gram models for feature extraction, eliminating these high-frequency but low-impact words is beneficial. Listing ?? shows the code to remove stopwords.

---

Listing 9: Code to remove stopwords

### 1.3.8 Streamline data preprocessing pipeline

To streamline the data preprocessing workflow and ensure consistent treatment of all tweet texts, we have encapsulated all the individual preprocessing steps into a single function named ‘`data_preprocess`’. This function takes a `DataFrame` as input and sequentially applies each preprocessing operation, from handling hashtags and converting text to lowercase, to removing mentions, URLs, and stopwords. Other operations such as lemmatization, punctuation removal, and the elimination of non-alphabetic

characters are also conducted in this unified function. The processed DataFrame is then returned, ready for feature extraction and subsequent analysis. By consolidating all preprocessing steps into a single function, we achieve greater efficiency and maintainability to iterate on our machine learning models. Listing ?? shows the code to streamline the data preprocessing pipeline.

---

Listing 10: Code for streamlined data preprocess

## 1.4 Bag of Words Model

In the Bag-of-Words (BoW) model, each tweet is represented by a feature vector  $\mathbf{x}$  of length equal to the size of the vocabulary created from the dataset. The entry at the  $i^{th}$  position of the feature vector is set to 1 if the corresponding  $i^{th}$  vocabulary word appears in the tweet; otherwise, it remains 0. To manage computational and memory efficiency, as well as to eliminate noisy or unreliable features, we implemented a minimum document frequency threshold, termed `min_df`.

To determine the appropriate value for `min_df`, we followed these steps:

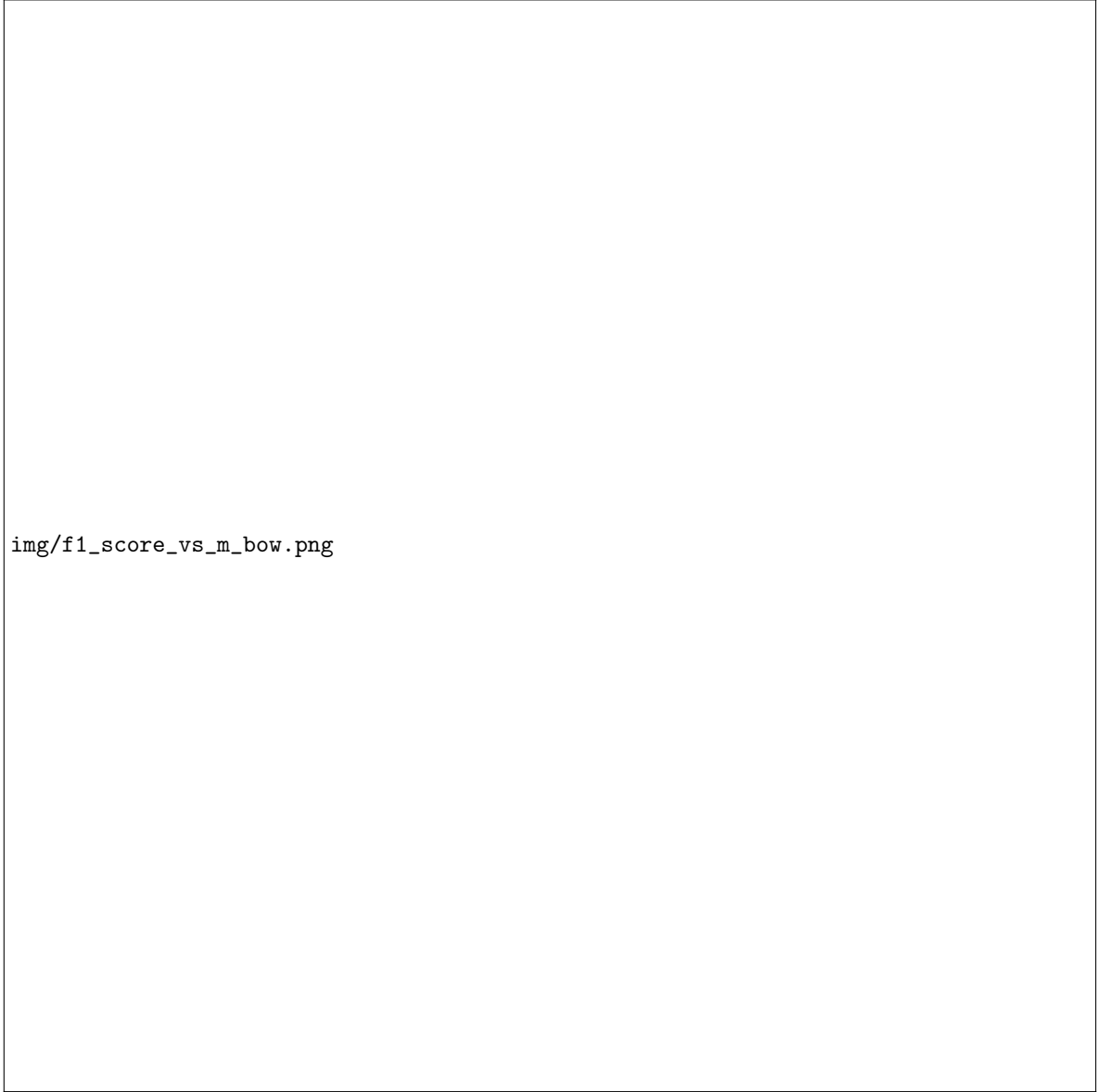
- **Start with a Small Value:** Initially set `min_df` to a 1 to include all terms in the vocabulary.
- **Build and Evaluate the Model:** With this initial `min_df` value, train a logistic regression model with L2 regularization. Evaluate the model's performance (i.e., F1-score) on a validation set.
- **Iterate and Increase `min_df`:** Gradually increase the value of `min_df` and retrain and re-evaluate the model. Keep track of how the model's performance changes.
- **Final Value:** The optimal `min_df` value would be the one resulting in the highest F1-score.

Throughout this iterative process, we noted the effect of increasing `min_df` on the model's performance. As `min_df` increased, the vocabulary size reduced, effectively filtering out terms that appeared in fewer tweets. By closely monitoring the F1-score, we found that the optimal `min_df` value was **3**, at which the model yielded the highest F1-score. As Figure ?? shows, the F1-score increased as `min_df` increased from 1 to 3, but it decreased as `min_df` increased after 3. This indicates that terms appearing in fewer than three tweets may have been adding noise rather than useful information. Therefore, setting `min_df` to 3 provided us with a reliable, efficient feature set for subsequent modeling and analysis. Listing ?? shows the code to implement such an iterative process to find the optimal `min_df` value.

Before going further and implementing the classification models, we first constructed a function named `infer_and_save_result` to streamline the evaluation of various models. This function takes several parameters, including the model name, vectorizer, training and development datasets, and performs several tasks. Initially, it transforms the text data into feature vectors using the supplied vectorizer. Subsequently, it infers the labels for both the training and development sets and calculates their respective macro-averaged F1 scores. Finally, the function saves essential components such as the vectorizer, the trained model, and the F1 scores for both datasets in a dictionary for easy comparison later. Listing ?? shows the code to implement `infer_and_save_result`.

---

Listing 11: Code for iterative process to find the optimal `min_df` value



img/f1\_score\_vs\_m\_bow.png

Figure 1: F1-score vs. min\_df in BoW model

Listing 12: Code for `infer_and_save_result`

## 1.5 Logistic Regression

We employ the results of BoW models to evaluate three logistic regression models: one without regularization, one with L1 regularization, and one with L2 regularization. The resulting performance on training and development sets are shown in Table ?? . We can see that the model with L2 regularization performs the best on the development set, with an F1-score of 0.7955. Listing ?? shows the code to implement the logistic regression models.

Model	F1 Score on Training Set	F1 Score on Dev Set
Logistic Regression (Without Regularization)	0.9813	0.7084
Logistic Regression (L1 Regularization)	0.8718	0.7878
Logistic Regression (L2 Regularization)	0.9066	0.7955

Table 1: F1 Scores for Different Logistic Regression Models with BoW

Listing 13: Code for logistic regression models

After fitting the logistic regression models, we took a closer look at the model utilizing L1 regularization to inspect its weight vector, commonly denoted as  $\theta$ . L1 regularization effectively zeroes out irrelevant features, making it ideal for feature selection. To examine the importance of each feature, we accessed the `coef_` attribute of the trained `LogisticRegression` instance to retrieve the weight vector. We then paired these coefficients with their corresponding feature names and sorted them by their absolute values in descending order.

Feature	Coefficient
hiroshima	3.4721
spill	3.3908
typhoon	3.2693
airport	3.2259
migrant	3.1984
derailment	3.1836
wildfire	3.1072
earthquake	2.9806
debris	2.9161
outbreak	2.8898

Table 2: Top 10 words for predicting a real disaster

The table ?? shows the top 10 words that have the highest absolute coefficients, suggesting that these words are strong indicators for predicting real disasters in tweets. Words like “hiroshima”, “spill,” and “typhoon” are naturally associated with events of serious consequence, which validates the model’s focus on these terms. The presence of such strong, unambiguous indicators in the weight vector underscores the model’s capability to pick out the most significant words for the classification task. This inspection thus provides not just a sanity check for our model but also potentially valuable insights

into the characteristics of tweets that are most indicative of real disasters. Listing ?? shows the code to get the top 10 words for predicting a real disaster.

Listing 14: Code for getting the top 10 words for predicting a real disaster

In addition to the logistic regression models, we also implemented a Bernoulli Naive Bayes classifier with Laplace smoothing. The classifier was built from scratch and applied to the feature vectors generated by our optimized Bag-of-Words (BoW) model. Laplace smoothing was employed to account for possible zero probabilities when estimating the conditional probabilities for each class label. This approach makes the Naive Bayes model more robust, particularly when dealing with features that may not appear in the training set but are present in the development set. Listing ?? shows the code to implement the Bernoulli Naive Bayes classifier.

Listing 15: Code for Bernoulli Naive Bayes classifier

The performance of the Bernoulli Naive Bayes model was evaluated using the F1 score, a balanced measure of precision and recall. On the training set, the model achieved an F1 score of 0.8481. When applied to the development set, the F1 score was 0.7963, which suggests that the model generalizes well to unseen data. These results validate the effectiveness of Bernoulli Naive Bayes when paired with an optimized BoW representation, making it a viable alternative to logistic regression for this classification task. Table ?? shows the results of the Bernoulli Naive Bayes model.

Model	F1 Score on Training Set	F1 Score on Dev Set
Bernoulli Naive Bayes on BoW	0.8481	0.7963

Table 3: F1 Scores for Bernoulli Naive Bayes Model with BoW

## 1.6 Model Comparison

Based on the F1 scores, the Bernoulli Naive Bayes model with an F1 score of 0.7963 on the development set performed comparably to the Logistic Regression model with L2 Regularization, which had an F1 score of 0.7955 on the development set. Both models performed well in predicting whether a tweet is related to a real disaster or not, but the **Naive Bayes model slightly outperformed the logistic models** on the development set, despite having a lower F1 score on the training set. This suggests that the **Naive Bayes model may generalize better to unseen data** in this specific task.

Regarding the pros and cons of generative versus discriminative models: generative models like Naive Bayes are often easier to train and require fewer data to make reasonable predictions. They model the underlying probability distribution of the data, which can be an advantage when you need to generate new samples. However, **they make strong assumptions about feature independence**. Discriminative models like Logistic Regression, on the other hand, focus on the boundary between classes and often achieve better performance when there is sufficient data. However, they may require more data to generalize well and are computationally more intensive to train.

Naive Bayes assumes that all features are conditionally independent given the class label. This is often **not the case in natural language processing tasks**, where word order and context can be important.

Logistic regression does not make this independence assumption, instead, it models the probability that a given instance belongs to a particular category. Given these considerations, it appears valid and efficient to use a Bernoulli Naive Bayes classifier for natural language texts, especially when we have a smaller dataset, including this one.

## 1.7 N-gram Model

After exploring feature extraction using the Bag-of-Words (BoW) model, we extended our investigation to incorporate N-Grams, specifically 2-Grams, into our feature vectors. Similar to the BoW model, we first determined the optimal value for `min_df` for the 2-Gram `CountVectorizer`. This was achieved by running a logistic regression model on feature vectors generated with various `min_df` values and monitoring the F1 score on the development set. Figure ?? shows the F1 score for different `min_df` values. After this empirical evaluation, the optimal `min_df` was identified as 2. Listing ?? shows the code to find the optimal `min_df` value for the 2-Gram `CountVectorizer`.

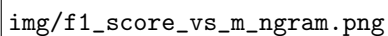


Figure 2: F1-score vs. `min_df` in 2-Gram model



Listing 16: Code for finding the optimal `min_df` value for the 2-Gram CountVectorizer

Utilizing this optimized `min_df`, we employed scikit-learn’s CountVectorizer to generate 2-Gram feature vectors. Subsequently, we assessed the performance of three logistic regression models—without regularization, with L1 regularization, and with L2 regularization—alongside a Bernoulli Naive Bayes model on these feature vectors. The code snippets similar to the ones shown in Listing ??, although omitted here for brevity, encapsulate the core logic for this part of the experimentation. Upon evaluation, the performance metrics revealed a varied outcome. Table ?? shows the results of the N-Gram models.

Model	F1 Score on Training Set	F1 Score on Dev Set
Logistic Regression (Without Regularization)	0.8281	0.6950
Logistic Regression (L1 Regularization)	0.7395	0.6670
Logistic Regression (L2 Regularization)	0.7913	0.6910
Bernoulli Naive Bayes	0.7216	0.6430

Table 4: F1 Scores for Different Models with 2-Gram

This exercise highlights that while N-Gram features can capture more context than individual words alone, they did not necessarily improve classification performance in our case, particularly when compared to the BoW model. It suggests that, for this specific task, simpler BoW features may suffice.

## 1.8 Performance on the Test Set

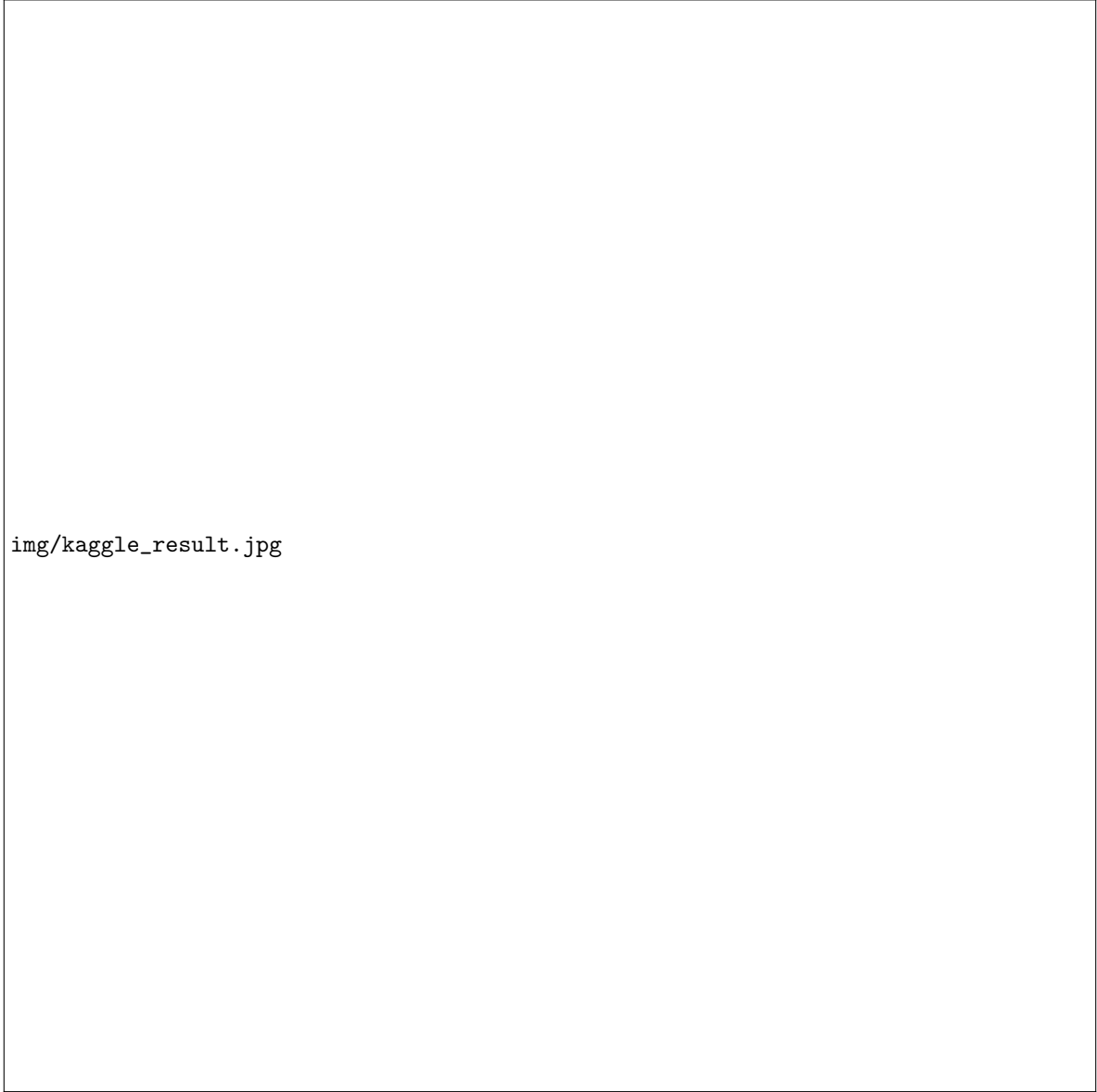
Upon concluding the series of experiments on feature extraction methods and classifiers, we identified the Bernoulli Naive Bayes model with Bag-of-Words (BoW) features as the top-performing model. Accordingly, we re-trained this model using the entire training dataset and then tested its performance on the Kaggle test set. The result was an F1 score of 0.79895, which positioned us at rank 477 out of 987 on the Kaggle leaderboard. Figure ?? shows the Kaggle score.

Interestingly, the performance on the Kaggle test set slightly exceeded our expectations based on the development set results, where the F1 score was 0.7963. There are several factors that could account for this discrepancy. One possible explanation is that the Kaggle test set might contain examples that are more aligned with the feature representations learned by our model. Additionally, using the entire training set to re-train the model could have provided it with a more comprehensive understanding of the feature space, thus boosting its ability to generalize well to unseen data. This slight increase in performance on the Kaggle test set can be considered as a positive indicator of the model’s reliability and generalization capability.

## 2 Written Exercises

### 2.1 Naive Bayes with Binary Features

### 2.2 Categorical Naive Bayes



img/kaggle\_result.jpg

Figure 3: Kaggle score