CS 5785 Homework 2

Cornell Tech

Tian Jin[*][†] Yufan Zhang[*][‡]

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 1 shows the code to get the basic statistics of the dataset.

```python
def load_data_from_csv(filename):
    return pd.read_csv(os.path.join(DATA_PATH, filename), index_col=0)

df = load_data_from_csv("train.csv")
X_test = load_data_from_csv("test.csv")

# Amount of data
print("The number of data points:")
print(f"training data \t {len(df)} \t test data \t {len(X_test)}\n")

# Percentage of classes
print("Percentage of training tweets are of real disasters:")
print(f"{round(df['target'].value_counts()[1]/len(df)*100, 2)}%")
print("Percentage of training tweets are not of real disasters:")
print(f"{round(df['target'].value_counts()[0]/len(df)*100, 2)}%")
```

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

[*]Both authors contributed equally to this project.
[†]tj299@cornell.edu
[‡]yz2894@cornell.edu

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 2.

```python
# Split the training data w/o sklearn
def train_test_split(X, y, test_size=0.2, random_state=42):
    np.random.seed(random_state)
    shuffled_indices = np.random.permutation(len(X))
    test_set_size = int(len(X) * test_size)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return (
        X.iloc[train_indices],
        X.iloc[test_indices],
        y.iloc[train_indices],
        y.iloc[test_indices],
    )

X = drop_features(df, ["target"])
y = keep_features(df, ["target"])

X_train, X_dev, y_train, y_dev = train_test_split(X, y, test_size=.3)

print(f"X_train: {X_train.shape} \t y_train: {y_train.shape}")
print(f"X_dev: {X_dev.shape} \t y_dev: {y_dev.shape}")

# X_train: (5330, 1)          y_train: (5330, 1)
# X_dev: (2283, 1)              y_dev: (2283, 1)
```

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 "#WorldNews". Since the objective is to extract meaningful features from the text using Bag-of-Words (BoW) 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 3 shows the code to handle hashtags.

```python
# Handle hashtags
def handle_hashtags(df):
    def split_hashtag_into_words(text):
        # Helper function to split words in hashtag
        return " ".join(re.findall(r"[A-Z]?[a-z]+|[A-Z]+(?=[A-Z]|$)", text))

    df["text"] = df["text"].apply(
        lambda x: re.sub(
            r"#(\S+)", lambda y: split_hashtag_into_words(y.group(1)), x
        )
    )
    return df
```

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 4 shows the code to lowercase all text.

```python
# Lowercase all text
def convert_to_lowercase(df):
    df["text"] = df["text"].str.lower()
    return df
```

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 5 shows the code to strip URLs and mentions.

```python
# Strip @ and urls.
def strip_mentions_and_urls(df):
    def helper(text):
        # Remove @mentions
        text = re.sub(r"@\w+", "", text)
        # Remove URLs
        text = re.sub(r"http\S+", "", text)
        return text

    df["text"] = df["text"].apply(helper)
    return df
```

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 6 shows the code to lemmatize words.

```python
# Lemmatize all the words
def lemmatize_words(df):
    lemmatizer = WordNetLemmatizer()
    df["text"] = df["text"].apply(
        lambda x: " ".join(
            [lemmatizer.lemmatize(word) for word in word_tokenize(x)]
        )
    )
    return df
```

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 7 shows the code to remove punctuation.

```python
# Strip punctuation
def strip_punctuation(df):
    df["text"] = df["text"].str.replace(r'[^\w\s]', "", regex=True)
    return df
```

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 8 shows the code to remove non-alphabetic characters.

```python
# Remove latin characters and digits
def remove_latin_and_digits(df):
    df["text"] = df["text"].str.replace(r"[^a-zA-Z\s]", "", regex=True)
    return df
```

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 9 shows the code to remove stopwords.

```python
# Strip the stop words, e.g., "the", "and", "or"
def strip_stopwords(df):
    stop = stopwords.words("english")
    df["text"] = df["text"].apply(
        lambda x: " ".join(x for x in x.split() if x not in stop)
    )
    return df
```

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 10 shows the code to streamline the data preprocessing pipeline.

```python
# Combine all the preprocessing steps into a single function
def data_preprocess(df):
    df = handle_hashtags(df)
    df = convert_to_lowercase(df)
    df = strip_mentions_and_urls(df)
    df = lemmatize_words(df)
    df = strip_punctuation(df)
    df = remove_latin_and_digits(df)
    df = strip_stopwords(df)
    return df
```

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 1 shows, the F1-score increased as `min_df` increased

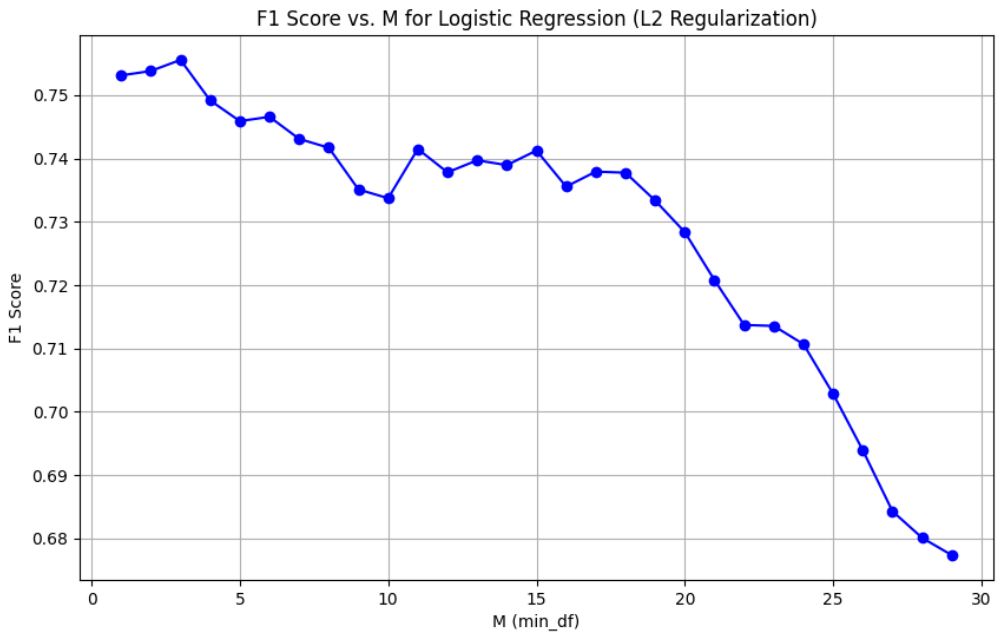


Figure 1: F1-score vs. `min_df` in BoW model

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 11 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 12 shows the code to implement `infer_and_save_result`.

```python
# Find the M with the highest F1 score
# Initialize empty lists to store values
M_values = []
F1_scores = []

# Find the M with the highest F1 score
for M in range(1, 30, 1):
    # Initialize the CountVectorizer with binary=True and min_df=M
    vectorizer = CountVectorizer(binary=True, min_df=M)

    # Vectorize the X_train
    X_train_bow_matrix = vectorizer.fit_transform(X_train['text'])
    X_train_bow = pd.DataFrame(
        X_train_bow_matrix.toarray(), columns=vectorizer.get_feature_names_out()
    )

    # Initialize and train the model
    model = LogisticRegression(penalty="l2", max_iter=1000)
    model.fit(X_train_bow, y_train)

    # Predict the labels of the validation set
    y_pred = model.predict(vectorizer.transform(X_dev['text']))
    score = f1_score(y_dev, y_pred)

    # Append M and its corresponding F1 score to the lists
    M_values.append(M)
    F1_scores.append(score)


# Create a plot
plt.figure(figsize=(10, 6))
plt.plot(M_values, F1_scores, marker='o', linestyle='-', color='b')
plt.title("F1 Score vs. M for Logistic Regression (L2 Regularization)")
plt.xlabel("M (min_df)")
plt.ylabel("F1 Score")
plt.grid(True)
plt.savefig(os.path.join(IMG_PATH, "f1_score_vs_m_bow.png"))
plt.show()

# Get the index of the M with the highest F1 score
best_M = np.argmax(F1_scores)
print(f"Best value for M: {M_values[best_M]}")

# Best value for M: 3
```

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

```python
# Record trained models and their results
models = dict()

def infer_and_save_result(
    model_name, vectorizer, model, X_train, y_train, X_dev, y_dev
):
    print(model_name)
    X_train_vec = vectorizer.transform(X_train['text'])
    X_train_vec = pd.DataFrame(
        X_train_vec.toarray(), columns=vectorizer.get_feature_names_out()
    )
    X_dev_vec = vectorizer.transform(X_dev['text'])
    X_dev_vec = pd.DataFrame(
        X_dev_vec.toarray(), columns=vectorizer.get_feature_names_out()
    )

    # Infer the training set
    y_pred_train = model.predict(X_train_vec)
    # Calculate the F1 score
    f1_train = f1_score(y_train, y_pred_train, average="macro")
    print("F1 score on training set: {:.4f}".format(f1_train))

    # Infer the dev set
    y_pred_dev = model.predict(X_dev_vec)
    # Calculate the F1 score
    f1_dev = f1_score(y_dev, y_pred_dev, average="macro")
    print("F1 score on dev set: {:.4f}".format(f1_dev))

    # Save the result
    models[model_name] = {
        "vectorizer": vectorizer,
        "model": model,
        "f1_train": f1_train,
        "f1_dev": f1_dev,
    }
```

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 1. We can see that the model with L2 regularization performs the best on the development set, with an F1-score of 0.7955. Listing 13 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

```python
# Train a logistic regression without regularization
model = LogisticRegression(penalty="none", max_iter=1000)
model.fit(X_train_bow, y_train)

infer_and_save_result(
    "Logistic Regression (Without Regularization) with BoW",
    vectorizer, model,
    X_train, y_train, X_dev, y_dev,
)

# Train a logistic regression with L1 regularization
model = LogisticRegression(penalty="l1", max_iter=1000)
model.fit(X_train_bow, y_train)

infer_and_save_result(
    "Logistic Regression (L1 Regularization) with BoW",
    vectorizer, model,
    X_train, y_train, X_dev, y_dev,
)

# Train a logistic regression with L2 regularization
model = LogisticRegression(penalty="l2", max_iter=1000)
model.fit(X_train_bow, y_train)

infer_and_save_result(
    "Logistic Regression (L2 Regularization) with BoW",
    vectorizer, model,
    X_train, y_train, X_dev, y_dev,
)
```

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 θ. 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.

The table 2 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

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

"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 14 shows the code to get the top 10 words for predicting a real disaster.

```python
# Get the L1 regularization logistic regression model
model_dict = models["Logistic Regression (L1 Regularization) with BoW"]
model = model_dict["model"]
fea_names = model_dict["vectorizer"].get_feature_names_out()

# Get the weight vector of the classifier
coefficients = model.coef_[0]

# Pair feature names with their coefficients
fea_coef = zip(fea_names, coefficients)

# Sort the features by their absolute coefficients in descending order
sorted_features = sorted(fea_coef, key=lambda x: abs(x[1]), reverse=True)

# Print the most important words/features
num_top_words = 10  # You can adjust this to show more or fewer top words
print(f"Top {num_top_words} words for predicting a real disaster:")
for feature, coef in sorted_features[:num_top_words]:
    print(f"{feature}: {coef}")
```

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

1.6 Bernoulli Naive Bayes

The Bernoulli Naive Bayes classifier with Laplace smoothing classifier was built from scratch and applied to the feature vectors generated by our optimized 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 15 shows the code to implement the Bernoulli Naive Bayes classifier.

```python
class BernoulliNaiveBayes:
    def __init__(self, alpha=1.0):
        self.alpha = alpha  # Laplace smoothing parameter
        self.d = None       # Number of features
        self.K = None       # Unique class labels
        self.psis = None    # Parameters of the Bernoulli distributions
        self.phis = None    # Class priors

    def fit(self, X, y):
        self.K = np.unique(y).shape[0]
        n = X.shape[0]  # Number of data points
        self.d = X.shape[1]
        self.psis = np.zeros([self.K, self.d])
        self.phis = np.zeros([self.K])

        for k in range(self.K):     # Update the parameters
            X_k = X[np.array(y_train == k)[:, 0]]
            self.psis[k] = (X_k.sum(axis=0) + self.alpha) / float(
                X_k.shape[0] + 2 * self.alpha
            )
            self.phis[k] = X_k.shape[0] / float(n)

    def predict(self, x):
        n, d = x.shape
        x = np.reshape(x, (1, n, self.d))
        psis = np.reshape(self.psis, (self.K, 1, d))

        psis = psis.clip(1e-14, 1 - 1e-14)  # clip probabilities to avoid log(0)

        # compute log-probabilities
        logpy = np.log(self.phis).reshape([self.K, 1])
        logpxy = x * np.log(psis) + (1 - x) * np.log(1 - psis)
        logpyx = logpxy.sum(axis=2) + logpy

        return logpyx.argmax(axis=0).flatten()

# Make predictions on the dev set
bayes = BernoulliNaiveBayes()
bayes.fit(X_train_bow, y_train)
infer_and_save_result(
    "Bernoulli Naive Bayes with BoW",
    vectorizer, bayes, X_train, y_train, X_dev, y_dev,
)
```

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 3 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

1.7 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.

1.8 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 2 shows the F1 score for different `min_df` values. After this empirical evaluation, the optimal `min_df` was identified as 2. Listing 16 shows the code to find the optimal `min_df` value for the 2-Gram `CountVectorizer`.

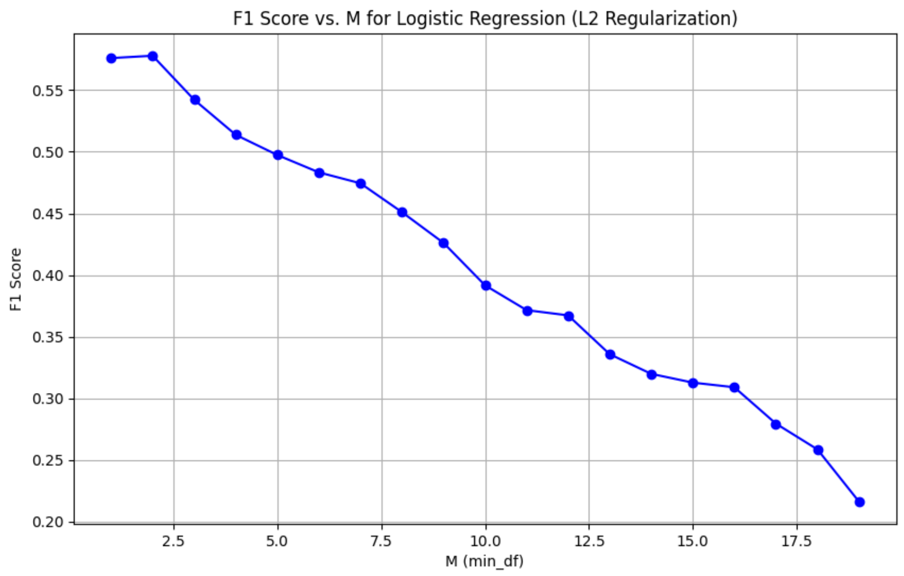


Figure 2: F1-score vs. `min_df` in 2-Gram model

```python
M_values = []
F1_scores = []

# Find the M with the highest F1 score
for M in range(1, 20, 1):
    # CountVectorizer with min_df=M
    vectorizer = CountVectorizer(binary=True, min_df=M, ngram_range=(2, 2))
    X_train_bow_matrix = vectorizer.fit_transform(X_train['text']).toarray()

    # Initialize and train the model
    model = LogisticRegression(penalty="l2", max_iter=1000)
    model.fit(X_train_bow, y_train)

    # Predict the labels of the validation set
    y_pred = model.predict(vectorizer.transform(X_dev['text']))
    score = f1_score(y_dev, y_pred)

    M_values.append(M)
    F1_scores.append(score)

plt.figure(figsize=(10, 6))
plt.plot(M_values, F1_scores, marker='o', linestyle='-', color='b')
plt.title("F1 Score vs. M for Logistic Regression (L2 Regularization)")
plt.xlabel("M (min_df)")
plt.ylabel("F1 Score")
plt.savefig(os.path.join(IMG_PATH, "f1_score_vs_m_ngram.png"))
```

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 13, 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 4 shows the results of the N-Gram models.

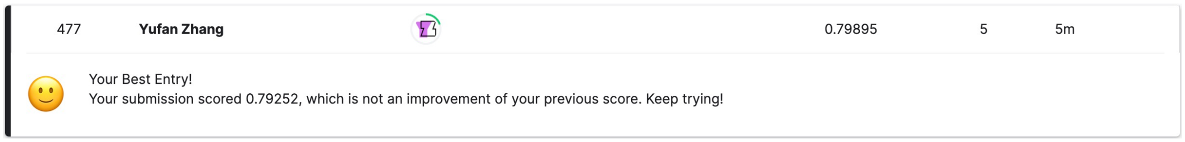
Model	**F1 Score on Training Set**	**F1 Score on Dev Set**
Logistic Regression (Without Regularization)	0.8281	0.6950
Logistic Regression (L1 Regularization)	0.7395	0.6670
Logistic Regression (L2 Regularization)	0.7913	0.6910
Bernoulli Naive Bayes	0.7216	0.6430

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.

1.9 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 3 shows the Kaggle score.


477 **Yufan Zhang** 0.79895 5 5m

Your Best Entry!
Your submission scored 0.79252, which is not an improvement of your previous score. Keep trying!

Figure 3: 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

(a) Please elaborate in this context what is the Naive Bayes assumption.

Naive Bayes classifier is a probabilistic classifier based on probability models that apply Bayes' theorem and incorporate strong independence assumptions. In this context, the Naive Bayes assumption is that the individual probabilities of students biking or skiing are independent of each other, whether they are Master's or PhD students. Mathematically, the assumption can be represented as:

$$P(x_1, x_2|y) = P(x_1|y) \times P(x_2|y) \tag{1}$$

(b) With the Naive Bayes assumption, find the probability of a student in this group who neither bikes or skis being a Master's student.

Given the definition, the probability of a student in this group who neither bikes or skis being a Master's student can be represented as:

$$P(y=0|x_1=0, x_2=0) = \frac{P(x_1=0, x_2=0|y=0) \times P(y=0)}{P(x_1=0, x_2=0)} \tag{2}$$

By Naive Bayes assumption, the calculation can be further extended as follows:

$$\begin{aligned} P(y=0|x_1=0, x_2=0) &= \frac{P(x_1=0, x_2=0|y=0) \times P(y=0)}{P(x_1=0, x_2=0|y=0) \times P(y=0) + P(x_1=0, x_2=0|y=1) \times P(y=1)} \\ &= \frac{P(x_1=0|y=0) \times P(x_2=0|y=0) \times P(y=0)}{P(x_1=0|y=0) \times P(x_2=0|y=0) \times P(y=0) + P(x_1=0|y=1) \times P(x_2=0|y=1) \times P(y=1)} \end{aligned} \tag{3}$$

Then we plug in the given numbers to compute the probability:

$$P(y=0|x_1=0, x_2=0) = \frac{\frac{30-8}{30} \times \frac{30-9}{30} \times \frac{30}{80}}{\frac{30-8}{30} \times \frac{30-9}{30} \times \frac{30}{80} + \frac{50-30}{50} \times \frac{50-12}{50} \times \frac{50}{80}} = \frac{77}{153}$$

(c) Suppose we know that every PhD who skis also bikes. Does it make sense to still assume that probability of biking and skiing are conditionally independent for a PhD student? If not, how would your answer to part (b) change with this knowledge (you can still assume probability of biking and skiing are conditionally independent for a Master's student)?

If every PhD who skis also bikes, then the independence assumption doesn't make sense, meaning that

$$P(x_1=0, x_2=0|y=1) = P(x_1=0|y=1) \times P(x_2=0|y=1)$$

no longer holds. In this case, to find the desired probability in part (b), i.e., the probability of a student in this group who neither bikes or skis being a Master's student, we follow (3) and get:

$$\begin{aligned} P(y=0|x_1=0, x_2=0) &= \frac{P(x_1=0, x_2=0|y=0) \times P(y=0)}{P(x_1=0, x_2=0|y=0) \times P(y=0) + P(x_1=0, x_2=0|y=1) \times P(y=1)} \\ &= \frac{P(x_1=0|y=0) \times P(x_2=0|y=0) \times P(y=0)}{P(x_1=0|y=0) \times P(x_2=0|y=0) \times P(y=0) + P(x_1=0, x_2=0|y=1) \times P(y=1)} \end{aligned}$$

$$= \frac{\frac{30-8}{30} \times \frac{30-9}{30} \times \frac{30}{80}}{\frac{30-8}{30} \times \frac{30-9}{30} \times \frac{30}{80} + \frac{20}{50} \times \frac{50}{80}} = \frac{77}{177}$$

2.2 Categorical Naive Bayes

(a) Show that the maximum likelihood estimate for the parameters ϕ_k is $\phi_k^* = \frac{n_k}{n}$, where n_k is the number of data points with class k.

We first obtain the log-likelihood function:

$$l(\phi) = \sum_{i=1}^{n} \log(\phi_{y^{(i)}}) = \sum_{k=1}^{K} n_k(\log \phi_k)$$

where n_k is the number of data points with class k.

In order to maximize the log-likelihood, we apply a Lagrange multiplier λ and get:

$$L(\phi, \lambda) = \sum_{i=1}^{n} \log(\phi_{y^{(i)}}) + \lambda \left(\sum_{k=1}^{K} \phi_k - 1 \right)$$

We then take the partial derivative with respect to ϕ_k and set it to zero:

$$\frac{\partial L}{\partial \phi_k} = \frac{n_k}{\phi_k} - \lambda = 0$$

$$\phi_k = \frac{n_k}{\lambda}$$

By definition of the categorical distribution, we have $\sum_{k=1}^{K} \phi_k = 1 = \sum_{k=1}^{K} \frac{n_k}{\lambda}$ as our inherent constraint, from which we get $\lambda = n$.

Therefore, we have our maximum likelihood estimate for the parameter as $\phi_k^* = \frac{n_k}{n}$.

(b) Show that the maximum likelihood estimate for the parameters ψ_{jkl} is $\psi_{jkl^*} = \frac{n_{jkl}}{n_k}$, where n_{jkl} is the number of data points with class k for which the j-th feature equals l.

Similarly, we apply a Lagrange multiplier to maximize the log-likelihood, and get:

$$L = \sum_{i:y^{(i)}=k} \log(\psi_{jx_j^{(i)}k}) + \lambda \left(1 - \sum_{l=1}^{L} \psi_{jkl} \right)$$

We then take the partial derivative with respect to ψ_{jkl} and set it to zero:

$$\frac{\partial L}{\partial \psi_{jkl}} = \frac{n_{jkl}}{\psi_{jkl}} - \lambda = 0$$

$$\psi_{jkl} = \frac{n_{jkl}}{\lambda}$$

Using the constraint that $\sum_{l=1}^{L} \psi_{jkl} = 1 = \sum_{l=1}^{L} \frac{n_{jkl}}{\lambda}$, we get $\lambda = n_k$.

Therefore, we have our maximum likelihood estimate for the parameter as $\psi_{jkl}^* = \frac{n_{jkl}}{n_k}$.