Applied Natural Language Processing
2023 S1
Assignment 1: Building a sentiment analysis system with Naive Bayes

Akide Liu
STUDENT NO. 1743748

April 16, 2023

Report submitted for **COMP SCI 4817** at the Faculty of Sciences, Engineering and Technology, University of Adelaide

In submitting this work I am indicating that I have read the University's Academic Integrity Policy. I declare that all material in this assessment is my own work except where there is clear acknowledgement and reference to the work of others.

I give permission for this work to be reproduced and submitted to other academic staff for educational purposes.

In this report, first-person pronouns such as "WE" or "I" have been used interchangeably to refer to the **author's own work**, in order to maintain a professional and academic tone.

OPTIONAL: I give permission this work to be reproduced and provided to future students as an exemplar report.

1 Text pre-processing technique

1.1 Stop words removal

```
1  def remove_stop_words(row):
2      # tokenize the sentence
3      tokens = word_tokenize(row)
4      # remove stop words
5      tokens = [w for w in tokens if not w in nltk.corpus.stopwords.words('english')]
6      # join the tokens
7      tokens = ' '.join(tokens)
8      # update the dataframe
9      return tokens
10
11 # multiple-threads processing
12 track_parallel_progress(remove_stop_words, df['review'].to_list(), nproc=10)
```

Figure 1: Python code for Stop Wrods Removal

Stop words represent frequently occurring words in a language that lack substantial semantic or pragmatic meaning. Consequently, they are often considered dispensable in natural language processing tasks such as sentiment analysis [9]. Examples of stop words in English include "a", "an", "the", "in", "on", "is", "and", and "of".

Eliminating stop words from text offers numerous benefits, including noise reduction, improved efficiency, enhanced accuracy of analysis, and decreased memory and computational resource consumption. This process involves identifying stop words within the text and subsequently removing them. In this study, we employed the pre-defined English stop words set available through the Natural Language Toolkit (NLTK), as illustrated in Code Block 1 on line 5. The results of this process are presented in the accompanying notebook, where we observed a significant reduction in token count from an initial size of 900 to 580 within a single text record.

We executed the stop words removal operation on a Pandas data frame, primarily using a CPU device. To optimize efficiency, we introduced a parallel execution mechanism (Code Block 1 on line 12) that leverages multiple threads within contemporary computing systems. We set the number of threads utilized to 10, resulting in a linear acceleration effect. Specifically, parallelization accelerated processing time by approximately 10 times for the current setting.

1.2 Stemming

Stemming is a technique employed in natural language processing to reduce a word to its root form or stem [5]. The aim of stemming is to simplify text data analysis by reducing inflected or derived words to a shared base form. For instance, the words "run", "running", and "runner" can all be stemmed to the root form "run".

Incorporating stemming in natural language processing tasks offers several advantages. It can minimize vocabulary size, consequently decreasing memory and computational resource requirements for analysis. Additionally, stemming can enhance analysis accuracy by reducing the number of unique words to be considered. Furthermore, stemming proves helpful in information retrieval tasks, as it allows searches for documents containing variations of a word to yield identical results.

In this study, we employed the Porter stemming algorithm available through the Natural Language Toolkit (NLTK) to stem words in the text, as demonstrated in Code Block 2

line 5. The results of this process are presented in the accompanying notebook, where we observed a notable reduction in token count, as inflected or derived words were condensed to their common base forms.

```
1  def stemming(row):
2      # tokenize the sentence
3      tokens = word_tokenize(row)
4      # stemming
5      tokens = [nltk.stem.PorterStemmer().stem(w) for w in tokens]
6      # join the tokens
7      tokens = ' '.join(tokens)
8      # update the dataframe
9      return tokens
10
11 # multiple-threads processing
12 track_parallel_progress(remove_stop_words, df['review'].to_list(), nproc=10)
```

Figure 2: Python code for Stemming

1.3 Byte Pair Encoding (BPE)

Byte Pair Encoding (BPE) [2] is a subword-based tokenizer used in natural language processing (NLP) to compress data by substituting the most frequently occurring pair of consecutive bytes of data with a byte that does not appear in the data. BPE is a prevalent data compression algorithm in NLP for constructing language models, including GPT-2 [10], RoBERTa [8], XLM [6], and FlauBERT [7]. The primary goal of BPE is to represent the corpus with the fewest number of tokens possible by decomposing infrequent words into two or more subword tokens while representing the most common words in the vocabulary as single tokens.

The BPE algorithm begins with a pre-tokenized corpus, and each word is divided into characters, including a special end token that signifies a word boundary. The algorithm then identifies the most frequently occurring byte pair and combines them into a new token. This process continues until the token limit size or iteration limit is reached.

We implemented the BPE algorithm by following the Hugging Face Tutorial [3, 4], which involves the subsequent steps. Code samples for the implementation can be found in the advanced notebook. In this section, we selected the first two text entries in the dataset data frame as a sample. To reduce computational overhead, we employed a subset for exploration purposes. The full version can be accessed by invoking a predefined BPE algorithm for batch preprocessing, as outlined in section 3.

- Define a BPE Tokenizer for pre-tokenization. In this case, we employed GPT-2.
- Calculate the frequency of each word and the base vocabulary formed by all characters used in the corpus.
- Compute the frequency of each pair and define a new function capable of merging split pairs.
- To tokenize new text, pre-tokenize it, split it, and apply all the learned merge rules.

2 Nave Bayes Classifier & evaluation

2.1 Training

The Nave Bayes Classifier [11] is a supervised learning algorithm employed for classification tasks. This parameterized classifier necessitates the estimation of two types of parameters: word likelihood and category likelihood. Word likelihood represents the probability of observing a specific word given a category, whereas category likelihood indicates the probability of observing a category in the training data.

To estimate these parameters, we utilize a training set comprising labeled examples. For each category, we count the occurrence frequency of each word and the total number of words within that category. This information enables us to calculate the word likelihood for each word in each category using the following formula:

$$P\left(w_k \mid c\right) = \frac{\text{count}\left(w_k, c\right)}{\sum_w \text{count}(w, c)} \tag{1}$$

In this equation, $\text{count}\left(w_k, c\right)$ denotes the number of times word w_k appears in category c, while $\sum_w \text{count}(w, c)$ represents the total number of words in category c.

Category likelihood is the probability of observing a category in the training data. We estimate this by counting the number of documents in each category and dividing it by the total number of documents:

$$P(c) = \frac{N_c}{N_{doc}} \tag{2}$$

Here, N_c refers to the number of documents in category c, and N_{doc} signifies the total number of documents in the training set.

After estimating these parameters, we can utilize them to classify new, unlabeled examples. We compute the likelihood of each category given the observed words in the example using Bayes' theorem and select the category with the highest likelihood as the predicted category.

```
def train(self, X_train, y_train):
    # Calculate the count of each class and each word in each class
    self._build_class_table(y_train)
    for x, y in tqdm(zip(X_train, y_train)):
      # Increment the count of the current class
        self.class_counts[y] += 1
        for word in x.split():
            if word not in self.class_word_counts[y]:
                # Initialize the count of the current word in the current class
                self.class_word_counts[y][word] = 0
            # Increment the count of the current word in the current class
            self.class_word_counts[y][word] += 1
            # Add the current word to the vocabulary
            self.vocabulary.add(word)
```

Figure 3: Python code for Naive Bayes Classifier Training

As depicted in Code Block 3, the training process of the Naive Bayes Classifier (NBC) begins by feeding the training data, denoted by `X_train` and `y_train`, to the `train` method. Then, a class counting table is constructed by enumerating the unique categories. In the present dataset, the class counting table is of size 2, as the dataset comprises only positive and negative categories.

Subsequently, each document and its corresponding label are iterated over, and a bag of words representation is constructed while counting the frequency of each word. To optimize efficiency, upon encountering a new word, there are two possible scenarios: either the word exists in the categorical counting table, in which case its corresponding counter is incremented by one, or the word is absent from the table, and we initialize its counter to zero. Finally, we append the word to the vocabulary using the `self.vocabulary` attribute.

In the training stage, we do not calculate the probability and likelihood of the full dataset to reduce computational overhead. Because that test set sample might not exist in the training set.

2.2 Prediction

Upon accumulating the necessary word count frequencies and the number of categories in a cache, the prediction can be executed based on the following calculations: Formally, designate the class variable as y, and let the dependent feature vector comprise $x_1, \ldots, x_n$. Bayes' theorem articulates the subsequent relationship:

$$P\left(y \mid x_1, \ldots, x_n\right) = \frac{P(y) P\left(x_1, \ldots, x_n \mid y\right)}{P\left(x_1, \ldots, x_n\right)} \tag{3}$$

The naive conditional independence assumption posits that

$$P\left(x_i \mid y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\right) = P\left(x_i \mid y\right) \tag{4}$$

for all i. This simplifies the relationship to:

$$P\left(y \mid x_1, \ldots, x_n\right) = \frac{P(y) \prod_{i=1}^{n} P\left(x_i \mid y\right)}{P\left(x_1, \ldots, x_n\right)} \tag{5}$$

As $P(x_1, \ldots, x_n)$ remains constant given the input, the classification rule becomes:

$$\begin{aligned} P\left(y \mid x_1, \ldots, x_n\right) &\propto P(y) \prod_{i=1}^{n} P\left(x_i \mid y\right) \\ &\Downarrow \\ \hat{y} &= \arg\max_y P(y) \prod_{i=1}^{n} P\left(x_i \mid y\right), \end{aligned} \tag{6}$$

To estimate $P(y)$ and $P(x_i \mid y)$, Maximum A Posteriori (MAP) estimation is employed. The former is derived from the relative frequency of class y in the training set.

In Code Block 4, the Naive Bayes classifier prediction process is illustrated. At Line 11, we compute the prior probability $P(y)$ as the ratio of the number of samples in a specific category to the total samples in the dataset. Additionally, we applied to add one smoothing to avoid zero probability which one word $P(w_k \mid c) = 0$. In Line 17, we calculate the conditional probability $P\left(x_i \mid y\right)$, considering the frequency of words in the test instance, and accumulate the product of these probabilities. Lastly, in Line 6, we perform the $argmax_y$ operation to determine the most probable category for the given test instance.

The code implementation adheres to the foundational concepts of the Naive Bayes classifier, capitalizing on its probabilistic nature to generate predictions.

```python
def predict(self, X_test):
    # Calculate the probability of each class for each test instance
    predictions = []
    for x in X_test:
        scores = [self.calculate_score(x, c) for c in self.classes]
        best_class = list(self.classes)[np.argmax(scores)]
        predictions.append(best_class)
    return predictions
def calculate_score(self, x, c):
    # Calculate the score of a test instance for a given class
    score = self.class_counts[c] / sum(self.class_counts.values())
    for word in x.split():
        if word in self.class_word_counts[c]:
            count = self.class_word_counts[c][word]
        else:
            count = 0
        score *= (count + 1) / (self.class_counts[c] + len(self.vocabulary))
    return np.log(score) if score > 0 else 0 # avoid underflow
```

Figure 4: Python code for Naive Bayes Classifier Prediction

2.3 Evaluation

2.3.1 Evaluation Metrics

The performance of a classification model can be assessed using various evaluation metrics, including accuracy, precision, recall, and F1 score. These metrics are defined as follows:

1. Accuracy: The proportion of correctly classified instances out of the total instances.

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1\left(\hat{y}_i = y_i\right) \tag{7}$$

2. Precision: The proportion of true positive instances among those predicted as positive.

$$\text{Rrecision } = \frac{tp}{tp + fp} \tag{8}$$

3. Recall (Sensitivity): The proportion of true positive instances among the actual positive instances.

$$\text{Recall } = \frac{tp}{tp + fn} \tag{9}$$

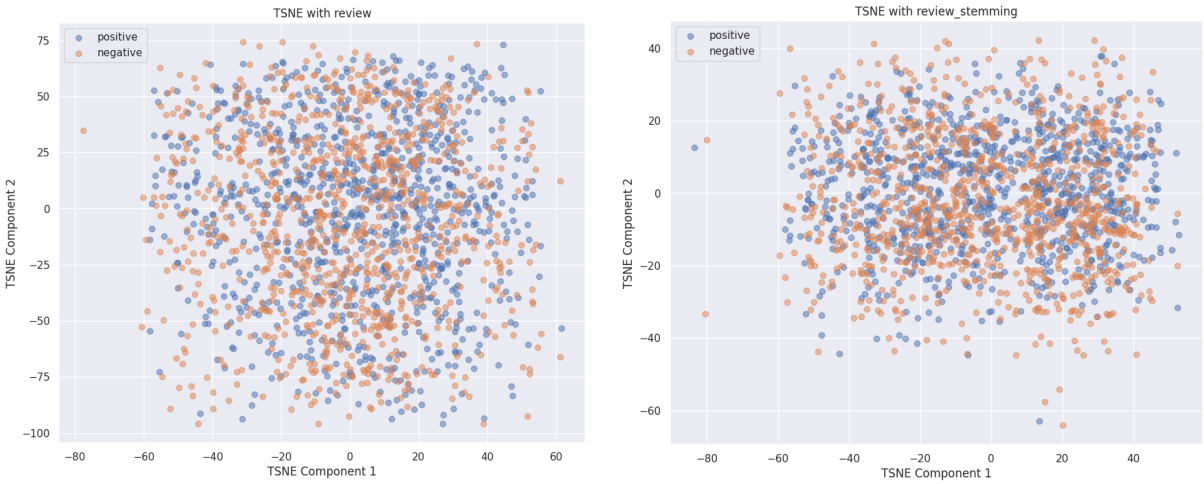
4. F1 Score: The harmonic mean of precision and recall, used as a balanced measure of both metrics.

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{10}$$

2.3.2 Performance Comparison

	Accuracy	Precision	Recall	F1 Score
w/o preprocessing	0.6189	0.7061	0.6189	0.5738
w preprocessing	0.7206	0.7531	0.7206	0.7113

Table 1: Performance Comparison Between w/o Preprocessing and w Preprocessing.



(a) w/o Preprocessing

(b) w Preprocessing

Figure 5: T-SNE Visualization Comparison for input text features space, we select 1000 samples per category for T-SNE analysis.

Table 1 illustrates a comparative analysis of a text classification model's performance with and without preprocessing. The preprocessing procedure encompasses the removal of stop words and the application of stemming to textual input. As a result, employing preprocessing techniques, such as stop word elimination and stemming, yields a substantial enhancement in the text classification model's performance across all evaluation metrics, including accuracy, precision, recall, and F1 score.

Figure 9 demonstrates that, in the absence of preprocessing, text features are more dispersed, potentially increasing the complexity for the classifier. However, by implementing stemming and removing stop words, the text features coalesce, even in the absence of distinct boundaries. Consequently, the classifier exhibits improved performance within this feature space.

3 Impact factor Investigation for classifier

3.1 Pre-trained GPT-2 BPE Tokenizer

The use of Byte Pair Encoding (BPE) as a tokenizer has been implemented in GPT-2, a language model developed by Radford et al. [10]. BPE is widely acknowledged as a practical solution for language modeling as it provides a middle ground between character and word-level approaches, effectively interpolating between frequent symbol sequences at the word level and infrequent symbol sequences at the character level.

In order to evaluate the efficacy of this approach, we conducted experiments with a pre-trained GPT-2 tokenizer in conjunction with basic pre-processing. The results in table 2 demonstrate that this enhancement led to an increase in classifier performance, with accuracy rising from 72.06% to 74.56%, and f1 score metrics improving from 71.13% to 74.13%.

Methods	Accuracy	Precision	Recall	F1 Score
Baseline	0.7206	0.7531	0.7206	0.7113
Pre-trained BPE	0.7456	0.7630	0.7456	0.7413
Fine-tune BPE	0.7672	0.7758	0.7672	0.7654
K-Smoothing(k=0.1)	0.7879	0.7919	0.7879	0.7872
Optimized NBC(k=4.2)	0.8203	0.8217	0.8203	0.8201
BERT Base [1]	0.9210	-	-	-
DistillBERT Base [12]	0.9405	-	-	-
Roberta Base [8]	0.9468	-	-	-

Table 2: Performance comparison for different enhancement techniques, The baseline methodology adopted in this research corresponds to the basic tokenization approach outlined in Section 2.3. The GPT-2 pre-trained tokenizer was utilized for the pre-trained byte pair encoding technique. To ensure a fair comparison, the BERT, DistillBert, and Roberta models were trained under the same hyperparameters and for two epochs.

3.2 Fine-tuned BPE Tokenizer

While the GPT-2 tokenizer is pre-trained on a large dataset, there may still be significant domain gaps between the production dataset and the database used in this assignment. Therefore, we fine-tuned another BPE tokenizer to evaluate its performance. By training the BPE tokenizer with a few iterations on the target dataset, we obtained promising results. Table 2 shows that after fine-tuning the BPE tokenizer on the target dataset, the accuracy increased from 74.56% to 76.72%, and f1 score metrics improved from 74.13% to 76.54% compared to the pre-trained BPE method.

3.3 K-Smoothing

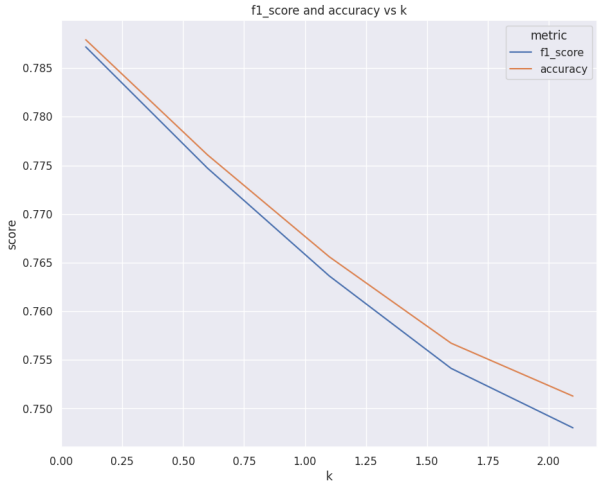


Figure 6: F1 score,accuracy vs k

k	f1_score	accuracy
0.1	**0.787186**	**0.78792**
0.6	0.774712	0.77608
1.1	0.763631	0.7656
1.6	0.75413	0.75672
2.1	0.748019	0.75128

Figure 7: F1 score and accuracy for K-Smoothing.

$$P(w_i) = \frac{\text{count}(w_i) + k}{N + Vk} \tag{11}$$

K-smoothing, also known as Laplace smoothing or add-k smoothing, is a technique used to improve the performance of the Naive Bayes classifier. As shown in the equation above, that add k to the words frequency and multiple k to the amount of vocabularies has following benefits :

1. *Handling zero probabilities*: by additional a non-zero positive k, we can avoid the problem that word in test set not existing in the training set which resulting 0 probability.

2. *Regularization*: By adding a small constant k to the feature-class counts, K-smoothing reduces the impact of rare feature-class combinations, making the model more robust and less likely to overfit.

3. *Smoothing effect*: By adding a k value redistributing probabilities across feature-class combinations. In another hand, when assign k to unseen words, K-smoothing helps create a more balanced distribution of probabilities, improving the overall performance of the Naive Bayes classifier.

As illustrated in Figure 6 and Table 7, the performance of the Naive Bayes classifier is optimal when the smallest value of k is employed. It is observed that the classifier's performance deteriorates as the value of k increases. A plausible explanation for this phenomenon may be attributed to the intrinsic property of the Naive Bayes classifier, where probabilities of feature values conditional on a class are multiplied to compute the joint probability for the respective class. Nevertheless, multiplying numerous probabilities can potentially lead to numerical challenges, particularly when confronted with minuscule probability values. The cumulative product of several small probabilities can become exceedingly small, thus resulting in underflow errors or imprecisions in floating-point arithmetic.

3.4 Optimized Naive Beyes Classifer

```
1  class OptimizedKSmoothingNaiveBayesClassifier(KSmoothingNaiveBayesClassifier):
2      def __init__(self, k=1):
3          super().__init__()
4          self.k = k
5
6      def calculate_score(self, x, c):
7          # Calculate the score of a test instance for a given class
8          score = np.log(self.class_counts[c])
9          for word in x.split():
10             if word in self.class_word_counts[c]:
11                 count = self.class_word_counts[c][word]
12             else:
13                 count = 0
14             score += np.log(count + self.k) / (self.class_counts[c] + self.k * len(self.vocabulary)
                   )
15         return score
```

Figure 8: Python code for multinomial naive Bayes classifier

We address the numerical issues observed in section 3.3, we use log space and addition instead of multiplication. The logarithm function has the property that:

$$\log(a * b) = \log(a) + \log(b) \tag{12}$$

Using this property, we can convert the multiplication of probabilities into the addition of logarithms of probabilities:

$$\begin{aligned} &\log\left(P\left(f_1 \mid C\right) * P\left(f_2 \mid C\right) * \cdots * P\left(f_n \mid C\right)\right) = \\ &\log\left(P\left(f_1 \mid C\right)\right) + \log\left(P\left(f_2 \mid C\right)\right) + \cdots + \log\left(P\left(f_n \mid C\right)\right) \end{aligned} \tag{13}$$

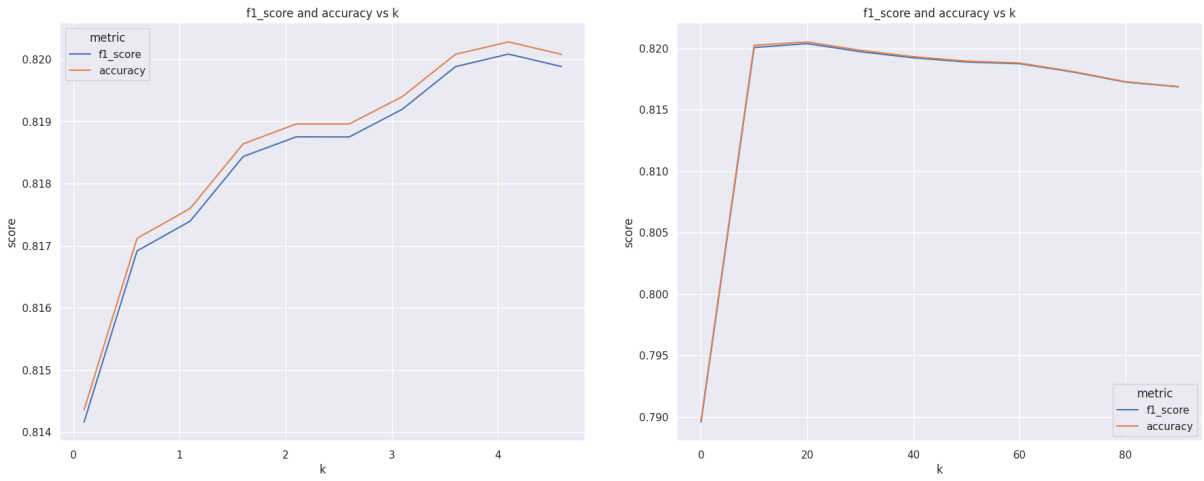
where f_i represents the feature values and C represents the class.
based on the above preliminary, we have introduce an **Optimized NBC** also termed Multinomial naive Bayes. Given that original naive Bayes as following :

$$p\left(\mathbf{x} \mid C_k\right) = \frac{\left(\sum_{i=1}^{n} x_i\right)!}{\prod_{i=1}^{n} x_i!} \prod_{i=1}^{n} p_{ki}^{x_i} \text{ where } p_{ki} := p\left(x_i \mid C_k\right) \tag{14}$$

The multinomial naive Bayes classifier becomes a linear classifier when expressed in log-space:

$$\begin{aligned} \log p\left(C_k \mid \mathbf{x}\right) &\propto \log\left(p\left(C_k\right) \prod_{i=1}^{n} p_{ki} x_i\right) \\ &= \log p\left(C_k\right) + \sum_{i=1}^{n} x_i \cdot \log p_{ki} \end{aligned} \tag{15}$$

In the Code Block 8 Line 14, we implemented the equation 15 for optimized NBC. After the optimized NBC, the performance raised in a large margin that from 78.79% accuracy to 82.03% accuracy and also improved across all metrics.3.



(a) k in range of 0.1-5.1 (b) k in range of 10-100

Figure 9: K-Smoothing analysis with different K values.

Upon examining the aforementioned figure, it has been determined that the optimal value for k is 4.1, within the range of 0.1 to 5.1. The performance exhibits a declining trend when $k > 5.1$. In the right figure, it can be observed that the performance consistently deteriorates within the range of 10 to 100. The potential issues contributing to this phenomenon are as follows:

1. *Increased bias*: As the value of k escalates, probability estimates tend to become more uniform, subsequently introducing a greater degree of bias into the model.

Consequently, the classifier's ability to accurately represent the true data distribution may be compromised.

2. *Attenuation of informative features*: A larger value of k results in the dilution of observed frequencies, thereby diminishing the significance of informative features in the classifier's decision-making process.

3. *Decreased sensitivity to rare events*: With an augmented k value, the influence of rare events on the overall probability estimates is relatively diminished. As Naive Bayes classifiers typically rely on such events to make accurate classifications, performance may suffer as a result.

4. *Over-smoothing*: The adoption of a larger k value can induce over-smoothing of the data, thereby inhibiting the classifier's ability to discern important patterns and nuances. This occurrence may adversely impact the classifier's performance.

3.5 Deep Learning Enhanced Classifier

Table 2 presents a comparison of the performance of traditional machine learning (ML) methods and deep learning (DL) methods, specifically BERT Base, DistillBERT Base, and Roberta Base, on an unspecified NLP task. The table reveals that the DL-based models considerably outperform the traditional ML-based models in terms of accuracy.

BERT [1], introduced by Google in 2018, is a transformer-based DL model pre-trained on a large corpus of text data and fine-tuned for a specific NLP task. It employs a bidirectional approach to consider both the left and right context of a word in order to understand its meaning. DistillBERT [12], introduced by Hugging Face in 2019, is a compressed version of BERT that leverages knowledge distillation to improve efficiency. Roberta [8], another transformer-based model introduced by Facebook AI in 2019, employs an enhanced training approach that allows learning from additional unlabeled data compared to BERT, leading to better performance.

The highest accuracy achieved by the ML-based models was 82.03% using an optimized NBC model, while BERT Base reached an accuracy of 92.10%, DistillBERT Base attained an accuracy of 94.05%, and Roberta Base achieved an accuracy of 94.68%. This demonstrates that DL-based models offer a significant improvement over traditional ML-based methods.

However, it is important to note that the fine-tuning process for these pre-trained transformer models entails a high computational cost. For example, our experiments used one node composed of 4× NVIDIA A100 GPUs to fine-tune these models. We set a global batch size of 256, and the fine-tuning process took approximately 5 minutes per epoch, consuming 144 GB VRAM.

In conclusion, DL-based models such as BERT Base, DistillBERT Base, and Roberta Base have shown superior performance to traditional ML-based methods in various NLP tasks. These models represent a significant advancement in the field of NLP and continue to push the boundaries of state-of-the-art performance.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[2] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, 1994.

[3] Huggingface. Byte-pair encoding tokenization - hugging face course, 04 2023.

[4] Chetna Khanna. Byte-pair encoding: Subword-based tokenization algorithm, 08 2021.

[5] Divya Khyani, BS Siddhartha, NM Niveditha, and BM Divya. An interpretation of lemmatization and stemming in natural language processing. *Journal of University of Shanghai for Science and Technology*, 22(10):350–357, 2021.

[6] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[7] Hang Le, Loïc Vial, Jibril Frej, Vincent Segonne, Maximin Coavoux, Benjamin Lecouteux, Alexandre Allauzen, Benoit Crabbé, Laurent Besacier, and Didier Schwab. Flaubert: Unsupervised language model pre-training for french. *arXiv preprint arXiv:1912.05372*, 2019.

[8] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[9] Hans Peter Luhn. The automatic creation of literature abstracts. *IBM Journal of research and development*, 2(2):159–165, 1958.

[10] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[11] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[12] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.