VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMMING INTERGRATION PROJECT (CO3101)

Report

Multi-label text classification

Advisor: Mr. Tran Tuan Anh
Students: Kang Dong Giang - 2152060
Mai Ton Dang Khanh - 2152122
Tran Ngoc Oanh - 2053312
Tran Hoang Khoi Tuan - 2012359

HO CHI MINH CITY, DECEMBER 2023

Contents

1 Introduction

1.1 Problem Statement

Multi-label sentiment analysis, also known as opinion mining, involves the intricate task of discerning multiple sentiments expressed in a given text, categorizing them as positive, negative, or neutral. In the era of social media, where copious amounts of unstructured data are generated daily, sentiment analysis plays a crucial role in understanding public opinions on diverse subjects, spanning from product reviews to political campaigns.

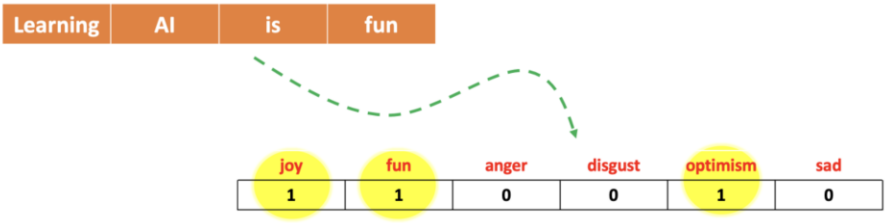
Traditional methods for sentiment analysis, such as TF-IDF, Naive Bayes, and Support Vector Machines (SVM), have been widely employed but face limitations in effectively capturing the nuanced subtleties of human language. These rule-based and lexicon-based approaches fall short when dealing with the complexity inherent in sentiment expression.

In contrast, neural networks within the realm of machine learning have emerged as powerful tools capable of learning intricate patterns from data. In this project, our focus will be on exploring various neural network architectures alongside traditional methods like TF-IDF, Naive Bayes, and SVM to enhance multi-label sentiment analysis.

The development of a robust sentiment analysis system, incorporating both traditional methods and neural networks, holds the potential to relieve the burden on human efforts required to discern and categorize various sentiments in a review. This approach provides a more comprehensive overview of sentiments expressed in our application's comments, offering a nuanced understanding beyond a simple binary positive or negative classification.

1.2 Project Statement

In the case of Multi-label, a text will be labeled with n classes (n is the number of classes in the problem) by assigning values 0, 1 (corresponding to yes or no) to each position of the text. class in the label.



Thus, the Input/Output of the problem will be:

- Input: A piece of text (string).
- Output: Vector of n elements (including real numbers with values in the range (0, 1))

In this project, we will solve a problem about Multi-label Text Classification through Multi-label Sentiment Analysis. In particular, the task is to predict the labels contained in a Tweet.

Dataset

We use the Multilabel Sentiment Analaysis dataset (SemEval-2018 Task 1), which has been divided into 3 sets of train/val/test as csv files. The following is the information of the first 5 rows of the **train.csv** data table:

	ID	Tweet	anger	anticipation	disgust	fear	joy	love	optimism	pessimism	sadness	surprise	trust
0	2017-En-21441	"Worry is a down payment on a problem you may ...	False	True	False	False	False	False	True	False	False	False	True
1	2017-En-31535	Whatever you decide to do make sure it makes y...	False	False	False	False	True	True	True	False	False	False	False
2	2017-En-21068	@Max_Kellerman it also helps that the majorit...	True	False	True	False	True	False	True	False	False	False	False
3	2017-En-31436	Accept the challenges so that you can literall...	False	False	False	False	True	False	True	False	False	False	False
4	2017-En-22195	My roommate: it's okay that we can't spell bec...	True	False	True	False	False	False	False	False	False	False	False

Based on this, it can be easily seen that the dataset has a total of 11 classes, including: **anger, anticipation, digust, fear, joy, love, optimism, pessimism, sadness, surprise, trust.**

In this project we will experiment 5 models, including:

1. **TF-IDF + Naive Bayes:** Utilizing the TF-IDF vectorization technique coupled with the Naive Bayes classifier provides a solid baseline. Naive Bayes is known for its efficiency and simplicity, making it well-suited for text classification tasks.
2. **TF-IDF + SVM:** SVMs excel in handling high-dimensional data and are particularly effective in scenarios where clear decision boundaries are crucial.
3. **BiLSTM:** BiLSTM is capable of understanding the sequential nature of textual data.
4. **BERT + CNN:** Integrating BERT embeddings with CNN introduces a powerful combination of pre-trained contextual embeddings and the ability to capture hierarchical features. This model is expected to excel in extracting intricate patterns.
5. **BERT + LSTM:** Merging BERT embeddings with LSTM enhances the model's capacity to retain information over longer sequences.

After all compare the result to see which models exhibit superior performance in capturing the nuanced emotional expressions present in our dataset.

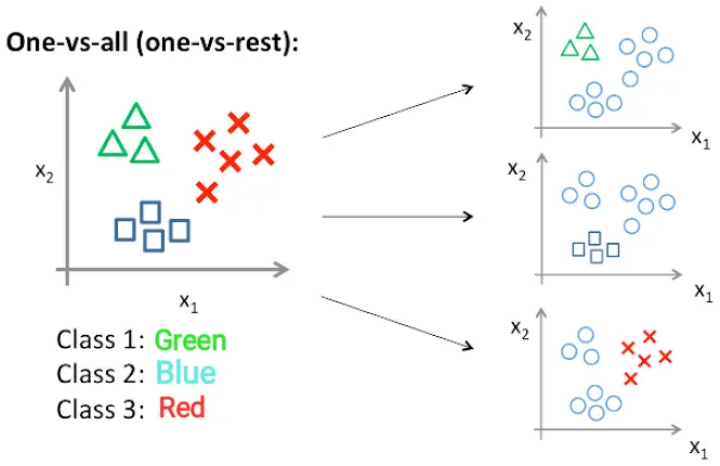
2 Approaches

2.1 Multi-label techniques

2.1.1 OnevsRest

Traditional two-class and multi-class problems can both be cast into multi-label ones by restricting each instance to have only one label. On the other hand, the generality of multi-label problems inevitably makes it more difficult to learn. An intuitive approach to solving multi-label problem is to decompose it into multiple independent binary classification problems (one per category).

In an "one-to-rest" strategy, one could build multiple independent classifiers and, for an unseen instance, choose the class for which the confidence is maximized.



2.1.2 Binary Relevance

In this case an ensemble of single-label binary classifiers is trained, one for each class. Each classifier predicts either the membership or the non-membership of one class. The union of all classes that were predicted is taken as the multi-label output. This is a simple approach but does not work well when there's dependencies between the labels.

$L = \{l_1, l_2, l_3, l_4\}$

EX#	Label set
1	$\{l_1, l_2\}$
2	$\{l_2, l_4\}$
3	$\{l_1, l_3, l_4\}$
4	$\{l_4\}$

EX#	l_1
1	true
2	false
3	true
4	false

EX#	l_2
1	true
2	true
3	false
4	false

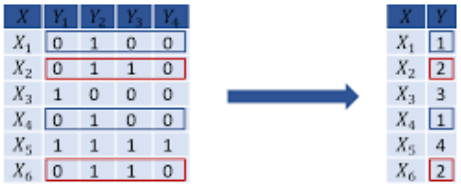
EX#	l_3
1	false
2	false
3	true
4	false

EX#	l_4
1	false
2	true
3	true
4	true

2.1.3 Label Powerset

This approach does take possible correlations between class labels into account. More commonly this approach is called the label-powerset method, because it considers each member of the power set of labels in the training set as a single label.

However when the number of classes increases the number of distinct label combinations can grow exponentially. This easily leads to combinatorial explosion and thus computational infeasibility.



X	Y_1	Y_2	Y_3	Y_4
X_1	0	1	0	0
X_2	0	1	1	0
X_3	1	0	0	0
X_4	0	1	0	0
X_5	1	1	1	1
X_6	0	1	1	0

X	Y
X_1	1
X_2	2
X_3	3
X_4	1
X_5	4
X_6	2

2.2 Feature Extraction

2.2.1 What are embeddings

Embeddings are vector representations of words or phrases that capture semantic relationships between them. In the context of natural language processing, these embeddings are numerical representations generated through techniques like Word2Vec, GloVe, or more sophisticated models like BERT and GPT. Each word or phrase is mapped to a vector in a high-dimensional space, and these vectors carry semantic information about the words.

2.2.2 Use of Embeddings

1. **Semantic Information:** Embeddings encode semantic relationships between words. Words with similar meanings are closer together in the embedding space, allowing models to understand and capture semantic nuances.
2. **Dimensionality Reduction:** Embeddings provide a more compact representation compared to traditional one-hot encodings. Instead of using a sparse and high-dimensional representation (as in one-hot encoding), embeddings offer dense, lower-dimensional vectors that capture meaningful relationships.
3. **Generalization:** Embeddings enable models to generalize better to unseen data. The model can learn from the relationships encoded in the embeddings and apply this knowledge to words or phrases not present in the training data.

We can also use a one-hot vector as our embedding, however, there are some reasons why we should considering of using other embeddings than one-hot:

1. Dimensionality Efficiency: Embeddings have lower dimensionality compared to one-hot encodings.
2. Semantic Information: Embeddings capture semantic relationships, allowing models to understand the contextual meaning of words.
3. Reduced Sparsity: One-hot encodings result in sparse vectors, with the majority of elements being zero.

2.2.3 Bag-of-words (BOW)

Bag of words is a Natural Language Processing technique of text modeling. It is a representation of text that describes the occurrence of words within a document.

A simplified explanation of how BoW works

- Step 1: Go through all the words in the above text and make a list of all of the words in our model vocabulary.

- Step 2: Count the presence of each word and mark 0 for the absence
- Step 3: Writing the frequencies in the vector form

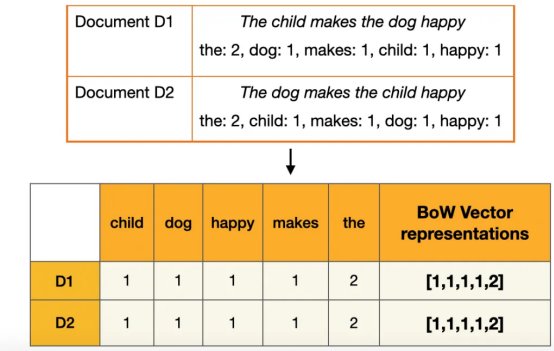


Figure 1. Bag of words

Although Bag-of-Words is quite efficient and easy to implement, there are still some disadvantages to this technique which are given below:

- The model ignores the location information of the word. The location information is a piece of very important information in the text. For example "today is off" and "Is today off", have the exact same vector representation in the BoW model.
- Bag of word models doesn't respect the semantics of the word. For example, the words 'soccer' and 'football' are often used in the same context. However, the vectors corresponding to these words are quite different in the bag of words model. The problem becomes more serious while modeling sentences. Ex: "Buy used cars" and "Purchase old automobiles" are represented by totally different vectors in the Bag-of-words model.
- The range of vocabulary is a big issue faced by the Bag-of-Words model. For example, if the model comes across a new word it has not seen yet, rather we say a rare, but informative word like Biblioklept(means one who steals books). The BoW model will probably end up ignoring this word as this word has not been seen by the model yet.

2.2.4 Term Frequency - Inverse Document Frequency (TF-IDF)

Another way to transfer the text data to the numerical data is **Term Frequency - Inverse Document Frequency**. It uses the two following definition:

Firstly, **Term Frequency**: describe the number of times the term appears in a document compared to the total number of words in the document.

$$TF = \frac{\text{number of times the term appears in the document}}{\text{total number of terms in the document}} \tag{1}$$

Next, **Inverse Document Frequency**: reflects the proportion of documents in the corpus that contain the term. Words unique to a small percentage of documents (e.g., technical jargon terms) receive higher importance values than words common across all documents (e.g., a, the, and).

$$IDF = \log\left(\frac{\text{number of the documents in the corpus}}{\text{number of documents in the corpus contain the term}}\right) \tag{2}$$

The **TF-IDF** of a term is calculated by multiplying TF and IDF scores.

$$TF - IDF = TF * IDF \tag{3}$$

It reflects the importance of the term within the specific document and its distinctiveness across the corpus. A high TF-IDF score suggests that a term is both frequent in the document and rare in the corpus, making it more significant.

TF-IDF is widely used in information retrieval, search engines, and document ranking to find documents that are most relevant to a query. It is also employed in text classification, sentiment analysis, document clustering, and topic modeling. Moreover, TF-IDF can help extract keywords, key phrases, and features from text data.

2.2.5 BERT

BERT stands for **Bidirectional Encoder Representations from Transformers**. It utilizes the infamous transformer architecture to learn contextual word representations, capturing the dynamic relationships between words in a sentence, paragraph, or even entire document. This allows BERT's embeddings to perform some NLP tasks that were once challenge for machines, like:

- Question Answering: Accurately answering questions based on a given context, even if the answer isn't explicitly stated.
- Sentiment Analysis: Understanding the emotional tone of text, beyond simple keyword identification.
- Text Summarization: Condensing the key points of a document while preserving its meaning.

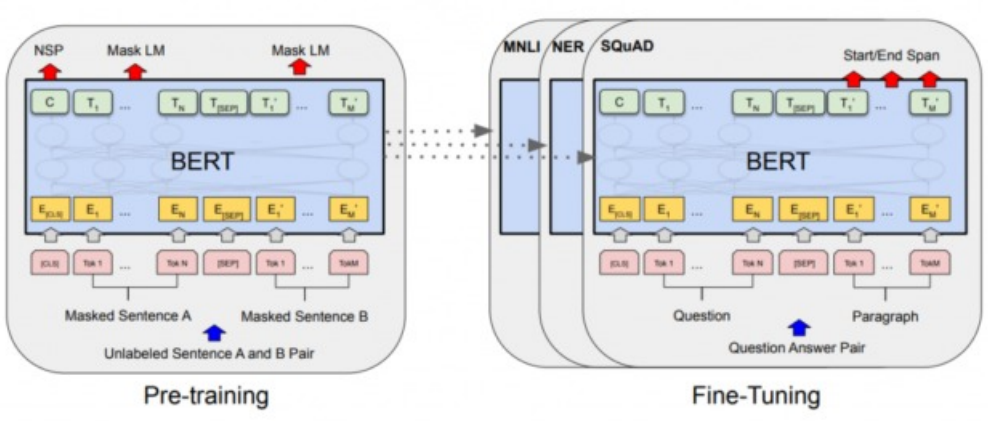


Figure 2. BERT model architecture

The core of BERT's magic lies in its training process and the mathematical formula behind its transformer architecture. BERT's training objectives include:

1. Masked Language Modeling (MLM): Imagine randomly masking words in a sentence and asking BERT to predict the missing words. This forces BERT to attend to the surrounding context and learn how words interact with each other. Mathematically, this is represented by a softmax function that calculates the probability of each word appearing in the masked position, given the context of the remaining words.

2. Next Sentence Prediction (NSP): BERT is trained to determine whether two sentences are consecutive or not. This helps it understand the flow of information and long-range dependencies between sentences. NSP is typical

Feature	CBOW Embeddings	BERT Embeddings
Context	Local	Contextual
Long-range dependencies	Limited	Captures long-range dependencies
Computation efficiency	Efficient	Less efficient
Suitable tasks	Local word relationships	Semantic understanding, context awareness

Bảng 1: CBOW and BERT

Based on the comparison provided above, to enhance accuracy, we will attempt to use BERT to obtain embeddings for our text in deep learning approaches.

2.3 Machine Learning approach

2.3.1 Naive Bayes

The Naive Bayes classifier is a supervised machine learning algorithm, which is used for classification tasks, like text classification. It is also part of a family of generative learning algorithms, meaning that it seeks to model the distribution of inputs of a given class or category. Unlike discriminative classifiers, like logistic regression, it does not learn which features are most important to differentiate between classes.

It is based on Bayes' Theorem.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{4}$$

Naive Bayes calculates probabilities to determine the likelihood of a data instance belonging to a particular class, and it uses these probabilities for classification

$$P(C_k|x_1, x_2, \ldots, x_n) = P(C_k) \cdot \prod_{i=1}^{n} P(x_i|C_k) \tag{5}$$

Naive Bayes assumes that predictors in a Naive Bayes model are conditionally independent, or unrelated to any of the other features in the model. It also assumes that all features contribute equally to the outcome. While these assumptions are often violated in real-world scenarios. Despite this unrealistic independence assumption, the classification algorithm performs well, particularly with small sample sizes.

The most popular types differ based on the distributions of the feature values. Some of these include:

- Gaussian Naive Bayes (GaussianNB): This is a variant of the Naïve Bayes classifier, which is used with Gaussian distributions—i.e. normal distributions—and continuous variables. This model is fitted by finding the mean and standard deviation of each class.
- Multinomial Naive Bayes (MultinomialNB): This type of Naïve Bayes classifier assumes that the features are from multinomial distributions. This variant is useful when using discrete data, such as frequency counts, and it is typically applied within natural language processing use cases, like spam classification.
- Bernoulli Naive Bayes (BernoulliNB): This is another variant of the Naïve Bayes classifier, which is used with Boolean variables—that is, variables with two values, such as True and False or 1 and 0.

Naive Bayes approach for our project

Because we want to apply Naive Bayes to multi-label text classification, we will use OneVsRestClassifier to change the problem to multiple binary classifications. And for each classification, we apply MultinomialNB.

Steps to build the model:

- Import libraries and load Data
- Preprocessing data, which includes removing some unnecessary characters and stopwords, lemmatizing, data splitting, and one hot encoding
- Vectorize data using TF-IDF
- Train model and tune hyperparameters. We use OneVsRestClassifier associated with MultiNomialNB to classify multi-label. Then we tune the "estimator_alpha" using GridSearchCV
- Evaluate model on Validation and Test Sets

In conclusion, the code demonstrates a comprehensive pipeline for multilabel text classification, encompassing data preprocessing, model training, hyperparameter tuning, and evaluation. It leverages the One-vs-Rest strategy to handle multiple labels and uses TF-IDF vectorization for feature representation. The hyperparameters are tuned using GridSearchCV to optimize the model's performance on the validation set.

2.3.2 Support Vector Machine

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection. For this problem, we can classify text by using this method.

What is the goal of the Support Vector Machine (SVM)?

The goal of a support vector machine is to find the optimal separating hyperplane which maximizes the margin of the training data.

The first thing we can see from this definition, is that a SVM needs training data. Which means it is a supervised learning algorithm. It is also important to know that SVM is a classification algorithm. Which means we will use it to predict if something belongs to a particular class.

In SVM algorithm, we want to find an optimal hyperplane that separates the two or more class.

What is the hyper-plane?

- In the 2-D space, it is a line that separates two region.

$$b + w_1 * X_1 + w_2 * X_2 = 0 \tag{6}$$

- In the 3-D space, it is a plane that separates two spaces.

$$b + w_1 * X_1 + w_2 * X_2 + w_3 * X_3 = 0 \tag{7}$$

- In the m-dimensional space, It is a hyper0plane that separates two class in two higher-space

$$b + w_1 * X_1 + w_2 * X_2 + w_3 * X_3 ... + w_m * X_m = 0 \tag{8}$$

For calculating the distance, using formula:

$$\frac{|w^T x_0 + b|}{\|w\|_2} \tag{9}$$

$$\|w\|_2 = \sqrt{\sum_{i=1}^{d} w_i^2} \tag{10}$$

With **d** is the number of dimension in the space.

We use the term "Margin" which represents the distance from every point in one class to the hyper plane. The bigger Margin are, the more correctly the training data was classified. Moreover, Maximising Margin can help us generalize better with unseen data. Therefore, SVM is also called as the Maximum Margin Classifier.

Firstly, we begin with the **Hard Margin Classifier** which is the easiest one to begin with:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \tag{11}$$

Constraints:

$$y_i(w \cdot x_i + b) \geq 1, \quad \forall i = 1, 2, ..., n \tag{12}$$

However, there are some drawbacks when using Hard Margin Classifier:

- Non-linearly Separable Data: Hard margin SVM assumes that the data is linearly separable, which means that it can find a hyperplane that perfectly separates the classes without any error. However, this is rarely the case in real-world scenarios, where data sets are often not linearly separable due to noise and outliers.
- Over-fitting: Because hard margin SVM seeks a solution with no classification errors, it can lead to over-fitting, especially when the data set has noise. The classifier may be too tightly fit to the training data, resulting in poor generalization to unseen data.
- Sensitivity to Outliers: Hard margin classifiers are extremely sensitive to outliers. A single outlier can dramatically change the position of the decision boundary, leading to a model that does not perform well on the overall data set.
- No Flexibility for Overlapping Classes: In many practical classification tasks, the classes can have a certain degree of overlap due to the inherent variability of the data. Hard margin SVM cannot handle such cases as it does not allow any mis-classifications.

With these problem, Soft Margin Classifier can be appropriate choices.
Soft Margin Classifier is the Classifier can detect the outliers by adding the term

$$C \sum_{n=1}^{N} \xi_n \tag{13}$$

In the context of Support Vector Machines (SVM), the constant C is a positive value and ξ represents the set of slack variables $\{\xi_1, \xi_2, \ldots, \xi_N\}$.

The constant C is utilized to balance the trade-off between maximizing the margin and minimizing mis-classification. The value of C can either be predefined by the programmer or determined via cross-validation.

$$(w, b, \xi) = \arg\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C \sum_{n=1}^{N} \xi_n \tag{14}$$

Subject to:

$$1 - \xi_n - y_n(w^T x_n + b) \leq 0, \forall n = 1, 2, \ldots, N \tag{15}$$

$$-\xi_n \leq 0, \forall n = 1, 2, \ldots, N \tag{16}$$

If the constant C is small, the high or low slack variables will not significantly affect the value of the objective function, and the algorithm will adjust to make the norm of $\|w\|^2$ as small as possible, which corresponds to the largest possible margin, leading to a reduction in the sum of the slack variables $\sum_{n=1}^{N} \xi_n$. Conversely, if C is large, the algorithm will focus on reducing $\sum_{n=1}^{N} \xi_n$ to minimize the objective function value. In scenarios where C is very large and the two classes are linearly separable, we obtain $\sum_{n=1}^{N} \xi_n = 0$, indicating that there are no misclassifications, corresponding to a solution for Hard Margin SVM. Thus, Hard Margin SVM is a particular instance of Soft Margin SVM.

For the data nearly linearly separate, we can use Soft Margin Classifier. Nevertheless, with the data sets have other shape or not linearly separate, Kernel SVM can change the data into the higher-dimensional space, to make it linearly separate.

The dual formulation of the Support Vector Machine optimization problem is given by:

$$\begin{aligned} \lambda &= \arg\max_{\lambda} \left(\sum_{n=1}^{N} \lambda_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \lambda_n \lambda_m y_n y_m k(x_n, x_m) \right) \\ \text{subject to:} \quad & \sum_{n=1}^{N} \lambda_n y_n = 0 \\ & 0 \leq \lambda_n \leq C, \quad \forall n = 1, 2, \ldots, N \end{aligned} \tag{17}$$

And the decision function is:

$$f(x) = \sum_{m \in S} \lambda_m y_m k(x_m, x) + \frac{1}{N_M} \sum_{n \in M} \left(y_n - \sum_{m \in S} \lambda_m y_m k(x_m, x_n) \right) \tag{18}$$

SVM approach for our project

We do apply the SVM similarly the Naive Bayes:

Steps to build the model:

- Import libraries and load Data
- Preprocessing data, which includes removing some unnecessary characters and stopwords, lemmatizing, data splitting, and one hot encoding
- Vectorize data using TF-IDF
- Train model and tune hyperparameters. We use OneVsRestClassifier, then tune the "estimator_alpha" using GridSearchCV
- Evaluate model on Validation and Test Sets

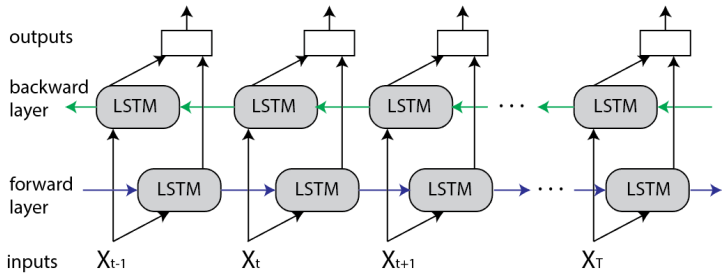
In conclusion, the code demonstrates a comprehensive pipeline for multilabel text classification, encompassing data preprocessing, model training, hyperparameter tuning, and evaluation. It leverages the One-vs-Rest strategy to handle multiple labels and uses TF-IDF vectorization for feature representation. The hyperparameters are tuned using GridSearchCV to optimize the model's performance on the validation set.

2.4 Deep Learning approach

2.4.1 BiLSTM

In the realm of multi-label text classification, where the complexity of understanding diverse aspects of textual data prevails, the **Bidirectional Long Short-Term Memory (BiLSTM)** model emerges as a formidable solution. The inherent challenges posed by sequences of words, coupled with the need to capture intricate dependencies, find an effective ally in the BiLSTM architecture.

Considering the multi-layered demands of multi-label text classification, the strengths of the BiLSTM model become not just advantageous but imperative. Its bidirectional processing, memory retention mechanisms, and contextual understanding make it adept at handling the intricate web of labels embedded within diverse texts. Therefore I decided to employ the BiLSTM model for this task, which is grounded in its proven efficacy in navigating the complexities of sequential data, ensuring a nuanced and comprehensive approach to multi-label text classification challenges.



2.4.2 BERT with CNN and LSTM

2.4.2.a Embedding implementation

When utilizing BERT embedding, certain configurations need to be addressed. When opting for phoBERT embedding, two choices are available:

1. **Using BERT as an Embedding Layer**: In this option, BERT serves as the embedding layer in our model. Consequently, BERT undergoes training during the backpropagation process.

2. **Using BERT Output as Embedding**: In this alternative, we initially pass our raw text through the phoBERT model. Subsequently, we extract the model's output and employ it as our text embedding. This implies that we do not train BERT during our classification task.

Due to limited available training data and insufficient computing resources, we have chosen the second option. This entails utilizing only the embeddings output from BERT without training it specifically for our task.

However, opting for the second option introduces a significant challenge regarding truncation and padding for sentences of varying lengths. Although the padding option in the BERT tokenizer can be employed, an issue arises. Despite padding the input with zeros before being fed into the model, the values change after passing through the model. These values could be disregarded if BERT were trained concurrently with our classification task. Nevertheless, since we have chosen option 2, a different approach to padding is required to prevent these values from introducing noise.

Therefore, a new padding technique is necessary. Initially, we will pass the raw text through the phoBERT embedding. After obtaining the embedding, we will pad the sequence to the maximum length we have set, which is 256, corresponding to the maximum length of the BERT model.

Example:
Max length: 8
Text: I love going to school
Approach 1: padding -> model
Before: [2,5,6,4,3]
After padding: [2,5,6,4,3,0,0,0]
Embeddings: [1,2,3,4,5,6,7,8]
Approach 2: model -> padding (CHOSEN)
Before: [2,5,6,4,3]
Embeddings: [1,2,3,4,5]
After padding: [1,2,3,4,5,0,0,0]

2.4.2.b CNN model

We will implement a CNN model in PyTorch. Three convolutional layers will be employed, each with a distinct filter size: 3, 4, and 5, corresponding to 3-gram, 4-gram, and 5-gram structures. The number of filters for each layer will be set to 100. The pooling layer will utilize maxpooling1D. After obtaining the results from the pooling layer, we will concatenate the output from each of the three convolutional models into a single hidden state with a size of (batch_size, number of filters * 3). Subsequently, we will pass this hidden state through a feed-forward layer and apply softmax to classify it into one of the three classes.

2.4.2.c LSTM model

First variant: Taking the mean

The LSTM network structure comprises two LSTM layers, succeeded by a fully connected layer and a softmax layer. The initial LSTM layer receives the input sequence and produces a

```python
class CNNnet(nn.Module):
  def __init__(self):
    super().__init__()
    self.conv_0 = nn.Conv1d(in_features, num_filters, kernel_sizes[0])
    self.pool_0 = nn.MaxPool1d(seq_len - kernel_sizes[0] + 1, stride = 1)

    self.conv_1 = nn.Conv1d(in_features, num_filters, kernel_sizes[1])
    self.pool_1 = nn.MaxPool1d(seq_len - kernel_sizes[1] + 1, stride = 1)

    self.conv_2 = nn.Conv1d(in_features, num_filters, kernel_sizes[2])
    self.pool_2 = nn.MaxPool1d(seq_len - kernel_sizes[2] + 1 , stride = 1)

    self.flatten = nn.Flatten()

    self.dropout = nn.Dropout(drop)

    self.dense_0 = nn.Linear(num_filters * 3, 3)

  def forward(self, x):
    x_0 = F.relu(self.pool_0(self.conv_0(x)))
    x_1 = F.relu(self.pool_1(self.conv_1(x)))
    x_2 = F.relu(self.pool_2(self.conv_2(x)))
    merged_tensor = self.flatten(torch.cat([x_0, x_1, x_2], dim = 1))
    merged_tensor = self.dropout(merged_tensor)
    x = F.softmax(self.dense_0(merged_tensor))
    return x
```

sequence of hidden states. The subsequent LSTM layer takes the hidden states from the first layer as input and generates a final hidden state. Subsequently, we calculate the mean of the logits derived from all the hidden states. The fully connected layer receives the final hidden state

```python
class LSTMnet(nn.Module):
  def __init__(self):
    super().__init__()
    self.lstm_0 = nn.LSTMCell(input_size = in_features, hidden_size = hidden_size)
    self.lstm_1 = nn.LSTMCell(input_size = hidden_size, hidden_size = hidden_size_2)
    self.dropout = nn.Dropout(drop)
    self.dense_0 = nn.Linear(hidden_size_2, 3)
  def forward(self, x):
    x = torch.permute(x, (2,0,1))
    output = []
    for i in range(x.size()[0]):
        hx, cx = self.lstm_0(x[i])
        output.append(hx)
    x = torch.stack(output, dim=0) #256, 4, 512
    output = []
    for i in range(x.size()[0]):
        hx, cx = self.lstm_1(x[i])
        output.append(hx)
    x = torch.stack(output, dim=0) #256, 4, 128
    x = torch.mean(x, dim = 0)

    x =  self.dense_0(x)
    x = F.softmax(x)
    return x
```

as input and produces a vector of logits. Finally, the softmax layer takes the mean logits vector as input and generates a probability distribution across the three output classes.

Second variant: Taking the last hidden state

The LSTM network structure comprises two LSTM layers, succeeded by a fully connected layer and a softmax layer. The initial LSTM layer receives the input sequence and produces a sequence of hidden states. The subsequent LSTM layer takes the hidden states from the first layer as input and generates a final hidden state. Subsequently, we get the logits from only the

last hidden states. The fully connected layer receives the final hidden state as input and produces

```
class LSTMnet(nn.Module):
  def __init__(self):
    super().__init__()
    self.lstm_0 = nn.LSTMCell(input_size = in_features, hidden_size = hidden_size)
    self.lstm_1 = nn.LSTMCell(input_size = hidden_size, hidden_size = hidden_size_2)
    self.dropout = nn.Dropout(drop)
    self.dense_0 = nn.Linear(hidden_size_2, 3)
  def forward(self, x):
    x = torch.permute(x, (2,0,1))
    output = []
    for i in range(x.size()[0]):
        hx, cx = self.lstm_0(x[i])
        output.append(hx)
    x = torch.stack(output, dim=0) #256, 4, 512
    output = []
    for i in range(x.size()[0]):
        hx, cx = self.lstm_1(x[i])
        output.append(hx)
    x = torch.stack(output, dim=0) #256, 4, 128
    x = torch.tensor(output[-1])
    x =  F.softmax(self.dense_0(x))
    return x
```

a vector of logits. Finally, the softmax layer takes the mean logits vector as input and generates a probability distribution across the three output classes.

3 Experiments

3.1 Machine Learning model

3.1.1 TF-IDF + Naive Bayes

Use the TF-IDF vectorizer to vectorize the Tweet:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
X_train_vectorized = vectorizer.fit_transform(x_train)
X_val_vectorized = vectorizer.transform(x_val)
X_test_vectorized = vectorizer.transform(x_test)
```

Then create and fit the Naive Bayes model to the training set:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.multiclass import OneVsRestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, accuracy_score, classification_report

# Create a Multinomial Naive Bayes classifier
nb_classifier = MultinomialNB()

# Wrap the classifier with OneVsRestClassifier
ovr_classifier = OneVsRestClassifier(nb_classifier)

# Train the model
ovr_classifier.fit(X_train_vectorized, y_train)

param_grid = {'estimator__alpha': [0.1, 0.5, 1.0, 1.5, 2.0]}

# Use accuracy as the scoring metric for GridSearchCV
scorer = make_scorer(accuracy_score)

# Create a GridSearchCV object
grid_search = GridSearchCV(ovr_classifier, param_grid, scoring=scorer, cv=5)
```

```

# Fit the model on the training set
grid_search.fit(X_train_vectorized, y_train)

# Get the best hyperparameters from the grid search
best_alpha = grid_search.best_params_['estimator__alpha']
print(f"Best alpha: {best_alpha}")

# Make predictions on the validation set using the best hyperparameters
val_predictions = grid_search.predict(X_val_vectorized)

# Evaluate the model on the validation set
val_accuracy = accuracy_score(y_val, val_predictions)
print(f"Validation Accuracy with Best Hyperparameters: {val_accuracy:.2f}")

# Make predictions on the test set using the best hyperparameters
test_predictions = grid_search.predict(X_test_vectorized)

# Evaluate the model on the test set
test_accuracy = accuracy_score(y_test, test_predictions)
print(f"Test Accuracy with Best Hyperparameters: {test_accuracy:.2f}")

# Print classification report for the test set
print("Classification Report for Test Set:")
print(classification_report(y_test, test_predictions))
```

The validation and accuracy result:

```
Best alpha: 0.5
Validation Accuracy with Best Hyperparameters: 0.11
Test Accuracy with Best Hyperparameters: 0.12
Classification Report for Test Set:
              precision    recall  f1-score   support

           0       0.79      0.45      0.57      1101
           1       0.00      0.00      0.00       425
           2       0.72      0.40      0.51      1099
           3       0.98      0.09      0.17       485
           4       0.90      0.48      0.63      1442
           5       1.00      0.01      0.02       516
           6       0.80      0.22      0.34      1143
           7       1.00      0.01      0.01       375
           8       0.83      0.17      0.29       960
           9       0.00      0.00      0.00       170
          10       0.00      0.00      0.00       153

   micro avg       0.82      0.27      0.40      7869
   macro avg       0.64      0.17      0.23      7869
weighted avg       0.77      0.27      0.36      7869
 samples avg       0.44      0.28      0.33      7869
```

Figure 3. Classification Report

In the Table above, we show the precision, recall, and f1-score for each class (label).

The accuracy of the validation set and test set is very low (0.11 and 0.12). The reason is that Naive Bayes methods are based on the assumption that features are conditionally independent. Moreover, it can be overfitting because the train set is not representative of a true distribution of the dataset, leading to poor generalization performance

3.1.2 TF-IDF + Support Vector Machine

Just changing Naive Bayes model into SVM model for training and testing:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.model_selection import GridSearchCV
```

```python
from sklearn.metrics import make_scorer, accuracy_score, classification_report
from sklearn.svm import SVC

# Create an SVM classifier with a linear kernel
svm_classifier = SVC(kernel='linear', probability=True)

# Wrap the classifier with OneVsRestClassifier
ovr_classifier = OneVsRestClassifier(svm_classifier)

# Train the model
ovr_classifier.fit(X_train_vectorized, y_train)

# Adjust the hyperparameter grid for the SVM
param_grid = {
    'estimator__C': [0.1, 1, 10],  # Example values for C
    'estimator__kernel': ['linear']  # Only 'linear' is needed since we already chose a linear kernel above
}

# Use accuracy as the scoring metric for GridSearchCV
scorer = make_scorer(accuracy_score)

# Create a GridSearchCV object
grid_search = GridSearchCV(ovr_classifier, param_grid, scoring=scorer, cv=5)

# Fit the model on the training set
grid_search.fit(X_train_vectorized, y_train)

# Get the best hyperparameters from the grid search
best_params = grid_search.best_params_
print(f"Best parameters: {best_params}")

# Make predictions on the validation set using the best hyperparameters
val_predictions = grid_search.predict(X_val_vectorized)

# Evaluate the model on the validation set
val_accuracy = accuracy_score(y_val, val_predictions)
print(f"Validation Accuracy with Best Hyperparameters: {val_accuracy:.2f}")

# Make predictions on the test set using the best hyperparameters
test_predictions = grid_search.predict(X_test_vectorized)

# Evaluate the model on the test set
test_accuracy = accuracy_score(y_test, test_predictions)
print(f"Test Accuracy with Best Hyperparameters: {test_accuracy:.2f}")

# Print classification report for the test set
print("Classification Report for Test Set:")
print(classification_report(y_test, test_predictions))
```

```
Best parameters: {'estimator__C': 1, 'estimator__kernel': 'linear'}
Validation Accuracy with Best Hyperparameters: 0.19
Test Accuracy with Best Hyperparameters: 0.20
Classification Report for Test Set:
              precision    recall  f1-score   support

           0       0.74      0.59      0.66      1101
           1       0.50      0.02      0.04       425
           2       0.68      0.55      0.61      1099
           3       0.83      0.55      0.66       485
           4       0.85      0.66      0.74      1442
           5       0.75      0.31      0.44       516
           6       0.70      0.48      0.57      1143
           7       0.67      0.06      0.11       375
           8       0.79      0.40      0.53       960
           9       0.45      0.05      0.09       170
          10       0.00      0.00      0.00       153

   micro avg       0.76      0.46      0.57      7869
   macro avg       0.63      0.33      0.40      7869
weighted avg       0.72      0.46      0.54      7869
 samples avg       0.63      0.48      0.51      7869
```

Figure 4. TF-IDF + SVM result

Overall, The accuracy of the validation set and test set is higher than Naive Bayes but still low (19% and 20%).The Precision. The precision, recall and the F1-score are also higher than the result of Naive Bayes. .It represents that SVM is the better algorithm than Naive Bayes.

Some classes, like 0, 2, 4, and 8, have relatively higher F1-scores, suggesting better performance. Class 1 and 9 have particularly low recall and f1-scores, indicating the model is struggling to correctly identify these classes. Class 10 has very low precision and f1-score but isn't represented in the micro/macro/weighted averages, possibly due to an error or it being an outlier.

3.1.3 Comparison

- Model Complexity:
  - Naive Bayes (NB): The results show a best alpha of 0.5 with validation and test accuracies of 0.11 and 0.12, respectively. The Naive Bayes model, given its simplicity, had a relatively low performance, possibly due to its feature independence assumption.
  - The SVM with a linear kernel (estimator___C: 1) achieved validation and test accuracies of 0.19 and 0.20, respectively, suggesting a better fit for the data complexity.
- Training Time:
  - NB: Likely to have been faster to train, as reflected by the simpler model structure.
  - SVM: Potentially took longer due to the need to compute the optimal hyperplane, but the use of a linear kernel might have mitigated some of the computational intensity.
- Prediction Time:
  - NB: Predictions with Naive Bayes are generally fast since the model involves straightforward calculations.
  - SVM: SVM predictions can be fast, especially in scenarios where the decision boundary is well-defined. However, in high-dimensional spaces, it may require more computational resources.
- Handling Imbalanced Data:

  - NB:May not have performed well with imbalanced data. This is reflected in the classification report, where certain classes have a recall of 0.00, indicating a failure to identify those classes.
  - SVM: Shows some ability to handle imbalance, as indicated by improved recall scores for some classes, but still struggles with certain classes (class 1 and class 9, for instance).

- Interpretability:
  - NB: Generally interpretable with a clear understanding of how features contribute to predictions.
  - SVM: Less interpretable, particularly as it does not directly provide probability estimates for class membership without enabling the $probability = True$ parameter.

- Robustness to Noisy Data:
  - SVM: SVM models can be less interpretable, especially when using complex kernel functions. Understanding the impact of individual features may be more challenging.
  - SVM: SVM may be less robust to noisy data, and outliers can affect the position of the decision boundary.

- Scalability:
  - NB: Naive Bayes is often considered scalable, especially for text classification tasks.
  - SVM: SVM can be less scalable, particularly with large datasets or when the number of features is high.

- Hyperparameter Sensitivity:
  - NB: Naive Bayes has fewer hyperparameters, making it less sensitive to tuning.
  - SVM: While effective in text classification, the model's performance was sub-optimal in this instance, possibly due to the dataset's complexity or imbalance.

- Effectiveness on Text Data:
  - NB: While effective in text classification, the model's performance was sub-optimal in this instance, possibly due to the dataset's complexity or imbalance.
  - SVM: With TF-IDF, the SVM showed improved effectiveness, likely benefiting from its ability to construct a more nuanced decision boundary.

In conclusion, the SVM model, even with a simple linear kernel, outperformed the Naive Bayes in this particular dataset, as evidenced by higher accuracy and f1-scores across most classes. The SVM's ability to handle complex patterns and its flexibility in dealing with imbalanced data likely contributed to its better performance despite being computationally more intensive and less interpretable. Both models have their use cases, but for this specific text classification task, SVM with a linear kernel seems to be a more suitable choice.

3.2 Deep Learning model

3.2.1 BiLSTM

- Input Layer: Receives a vector with data type 'string'. (Shape = (1,)).
- Text Vectorization Layer: From the input string, perform text vectorization using the layer tf.keras.layers.TextVectorization:

```
text_vectorization_layer = tf.keras.layers.TextVectorization(
    max_tokens=MAX_FEATURES, # Vocabulary size
    output_mode='int', # The token value is the index of the word in vocab
    output_sequence_length=MAX_SEQ_LEN # Maximum number of tokens in 1 vector
)

train_text = train_ds.map(lambda text, labels: text) # Call `content` of all data samples in the training set
text_vectorization_layer.adapt(train_text) # Build vectorization layer based on training data
```

Build the model initialization function:

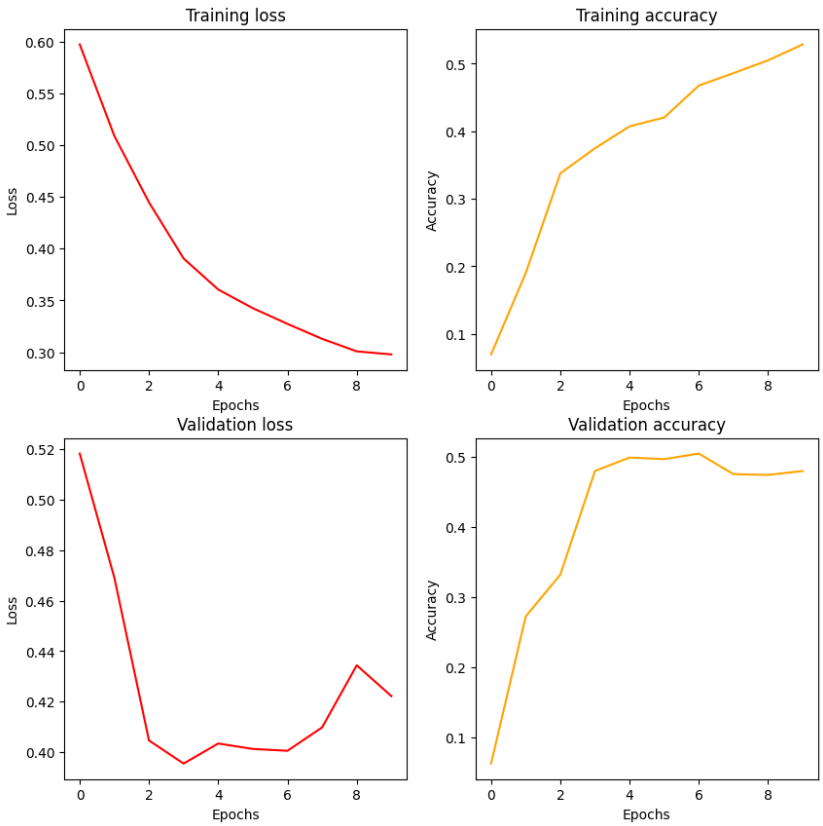
```
# Build the model initialization function
def build_model(max_features, max_seq_len, embedding_dims, n_classes):
    model = tf.keras.Sequential([
        # Input layer (receives a string)
        tf.keras.Input(shape=(1,), dtype='string', name='input_layer'),

        # Text Vectorization Layer declared above
        text_vectorization_layer,

        # Embedding Layer (convert tokens into vectors)
        tf.keras.layers.Embedding(input_dim=max_features+1,
                                  output_dim=embedding_dims,
                                  embeddings_initializer=tf.random_uniform_initializer(seed=RANDOM_SEED),
                                  mask_zero=True,
                                  name='embedding_layer'),

        # Bi-LSTM Layer 1
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,
                                    return_sequences=True,
                                    kernel_initializer=tf.initializers.GlorotUniform(seed=RANDOM_SEED)),
                                    name='bilstm_layer_1'),

        # Bi-LSTM Layer 2
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,
                                    return_sequences=True,
                                    kernel_initializer=tf.initializers.GlorotUniform(seed=RANDOM_SEED)),
                                    name='bilstm_layer_2'),

        # Bi-LSTM Layer 3
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,
                                    return_sequences=False,
                                    kernel_initializer=tf.initializers.GlorotUniform(seed=RANDOM_SEED)),
                                    name='bilstm_layer_3'),

        # Dropout Layer 1
        tf.keras.layers.Dropout(0.2),

```

```
        # Fully-connected Layer 1
        tf.keras.layers.Dense(64,
                              activation='relu',
                              kernel_initializer=tf.initializers.GlorotUniform(
    seed=RANDOM_SEED),
                              name='fl_layer_1'),

        tf.keras.layers.Dropout(0.3),

        # Fully-connected Layer 2
        tf.keras.layers.Dense(32,
                              activation='relu',
                              kernel_initializer=tf.initializers.GlorotUniform(
    seed=RANDOM_SEED),
                              name='fl_layer_2'),

        # Dropout Layer 2
        tf.keras.layers.Dropout(0.3),

        # Output Layer
        tf.keras.layers.Dense(n_classes,
                              activation='sigmoid',
                              kernel_initializer=tf.initializers.GlorotUniform(
    seed=RANDOM_SEED),
                              name='output_layer')
    ],
    name='bilstm_model')

    return model
```

In there:

- **Embedding Layer:** From the output vector of the text vectorization layer, We put it into the embedding layer. The main function of this layer is to represent integer elements in the input vector into vectors with the same length (embedding dims).
- **BiLSTM Layer 1:** From the output vector of the embedding layer, We put in the first BiLSTM layer with 64 units. Note that it is necessary to return the output vector of the entire timestep at this layer so that it can be used as input of the next BiLSTM layer.
- **BiLSTM Layer 2:** The second BiLSTM layer with 64 units, returns the hidden state of the entire timestep.
- **BiLSTM Layer 3:** The final BiLSTM layer with 64 units, with this layer I will only return the output vector at the last timestep.
- **Dropout Layer 1:** Dropout layer with drop rate = 0.2.
- **Fully-connected Layer 1:** From the output vector of the last layer RNN, I introduce a first fully-connected layer with 64 nodes with activation function 'relu'.
- **Dropout Layer 2:** Dropout layer with drop rate = 0.3.
- **Fully-connected Layer 2**: The second fully-connected layer with 32 nodes with activation function 'relu'
- **Dropout Layer 3:** Dropout layer with drop rate = 0.3.
- **Output Layer:** Fully-connected layer with 11 nodes (representing 11 classes) and activation function is 'sigmoid'.

Visualize training results: We can visualize the model's performance during the learning process by plotting loss and accuracy graphs on the training set and val during the training process.



In summary, the outcomes following 10 epochs of training on a biLSTM model for a multi-label text classification task reveal a training loss of 0.3 and a training accuracy of 0.5. Simultaneously, the validation loss and accuracy are recorded at 0.42 and 0.5, respectively. These metrics suggest that the model is achieving a moderate fit to the training data, as indicated by the relatively low training loss. However, the validation results, with comparable loss and accuracy, indicate a potential challenge in generalizing the model's knowledge to unseen instances.

3.2.2 BERT

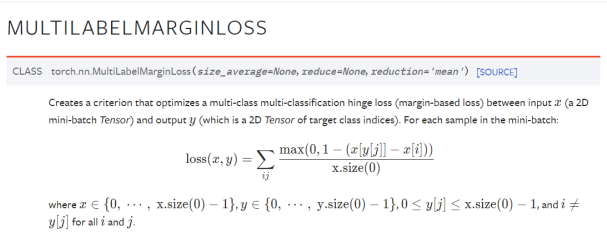
```
import torch
from transformers import AutoModel, AutoTokenizer
from tqdm import tqdm

phobert = AutoModel.from_pretrained("bert-base-uncased")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
```

For the BERT model, we utilize the pretrained version of BERT. Specifically, we opt for the bert-base-uncased variant, aligning with our preprocessing step of lowercasing every tweet in our dataset.

To enhance accuracy, we implement the embedding method described in section 2.3.2.a.

After passing each sentence through BERT to obtain their embeddings, we then feed the processed data into our model and training loop.

The loss function employed is the multilabel margin loss, and its formula can be observed in the figure below.



MULTILABELMARGINLOSS

CLASS torch.nn.MultiLabelMarginLoss(*size_average=None, reduce=None, reduction='mean'*) [SOURCE]

Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input x (a 2D mini-batch *Tensor*) and output y (which is a 2D *Tensor* of target class indices). For each sample in the mini-batch:

$$\text{loss}(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{\text{x.size}(0)}$$

where $x \in \{0, \cdots, \text{x.size}(0) - 1\}, y \in \{0, \cdots, \text{y.size}(0) - 1\}, 0 \leq y[j] \leq \text{x.size}(0) - 1$, and $i \neq y[j]$ for all i and j.

Figure 5. Multilabel margin loss

The architectures of the two mentioned models are:

CNN model:

```
import torch.nn as nn
import torch.nn.functional as F
in_features = 768
kernel_sizes = [3,4,5]
num_filters = 256
drop = 0.2
seq_len = 128
class CNNnet(nn.Module):
  def __init__(self):
    super().__init__()
    self.conv_0 = nn.Conv1d(in_features, num_filters, kernel_sizes[0])
    self.pool_0 = nn.MaxPool1d(seq_len - kernel_sizes[0] + 1, stride = 1)

    self.conv_1 = nn.Conv1d(in_features, num_filters, kernel_sizes[1])
    self.pool_1 = nn.MaxPool1d(seq_len - kernel_sizes[1] + 1, stride = 1)

    self.conv_2 = nn.Conv1d(in_features, num_filters, kernel_sizes[2])
    self.pool_2 = nn.MaxPool1d(seq_len - kernel_sizes[2] + 1 , stride = 1)

    self.flatten = nn.Flatten()

    self.dropout = nn.Dropout(drop)

    self.dense_0 = nn.Linear(num_filters * 3, 11)

  def forward(self, x):
    x_0 = F.relu(self.pool_0(self.conv_0(x)))
    x_1 = F.relu(self.pool_1(self.conv_1(x)))
    x_2 = F.relu(self.pool_2(self.conv_2(x)))
    merged_tensor = self.flatten(torch.cat([x_0, x_1, x_2], dim = 1))
    merged_tensor = self.dropout(merged_tensor)
    x = F.relu(self.dense_0(merged_tensor))
    return x
```

This CNN employs three distinct kernel sizes to capture various sequence lengths within our tweets. Each kernel is accompanied by its own pooling layer.

Subsequently, the outputs from the three pooling layers are concatenated and flattened before being fed into a dense layer to compute the desired logits.

To constrain the logits within the [0, 1] range, the rectified linear unit (ReLU) function is utilized instead of the sigmoid function, aiming to enhance gradient flow.

LSTM model:

```
import torch.nn as nn
import torch.nn.functional as F
```

```
in_features = 768
kernel_sizes = [3,4,5]
hidden_size = 128
hidden_size_2 = 256
drop = 0
seq_len = 128
class LSTMnet(nn.Module):
  def __init__(self):
    super().__init__()
    self.lstm_0 = nn.LSTMCell(input_size = in_features, hidden_size = hidden_size)
    self.lstm_1 = nn.LSTMCell(input_size = hidden_size, hidden_size =
    hidden_size_2)
    self.dropout = nn.Dropout(drop)
    self.dense_0 = nn.Linear(hidden_size_2, 11)
  def forward(self, x):
    x = torch.permute(x, (2,0,1))
    output = []
    for i in range(x.size()[0]):
        hx, cx = self.lstm_0(x[i])
        output.append(hx)
    x = torch.stack(output, dim=0) #256, 4, 512
    for i in range(x.size()[0]):
        hx, cx = self.lstm_1(x[i])
        output.append(hx)
    x = torch.stack(output, dim=0) #256, 4, 12
    x = torch.mean(x, dim = 0)
    x =  F.relu(self.dense_0(x))
    return x
```

Conducting a comparative analysis between variant 1 and 2 of the LSTM structure, we have chosen to employ the mean instead of solely relying on the last hidden state.

In the LSTM architecture, we utilize two LSTM cells to process the tweet. In the final layer, we compute the mean of each state and subsequently feed the calculated result into a dense layer to obtain our desired output.

Additionally, in the LSTM framework, we incorporate ReLU as our activation function.

Result:

CNN: There are some errors in this logging process, the actual training loop starts at epoch

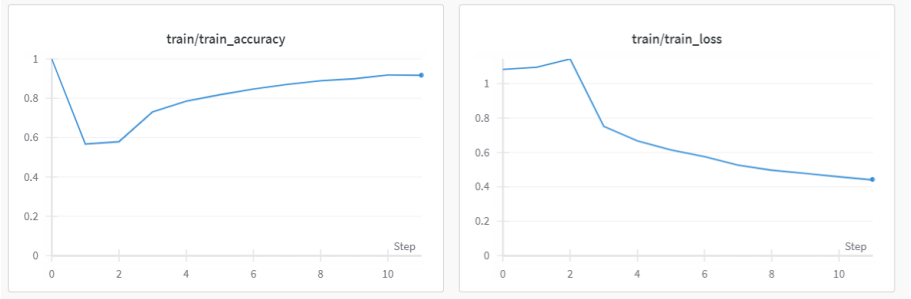


Figure 6. Train accuracy and loss using CNN architecture

one. We using 10 epochs to train the model and the best accuracy on test set is 80%

LSTM: In the LSTM model architecture, we employ 10 epochs for the training process; however, the achieved result is only approximately 70

In summary, concerning this multi-label sentiment analysis task, our CNN architecture has proven to be superior to the LSTM model. This superiority may be attributed to the quality of

Figure 7. Test accuracy and loss using CNN architecture

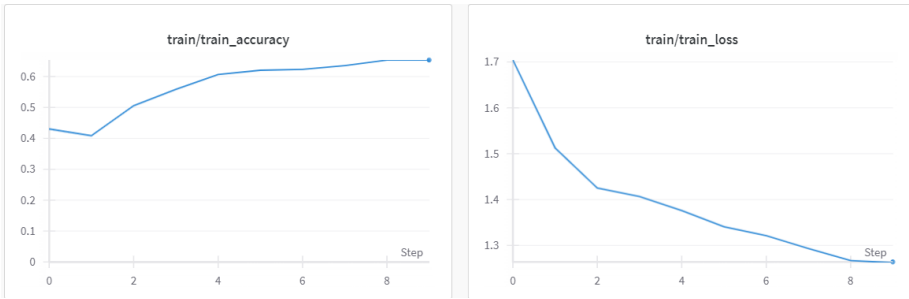


Figure 8. Train accuracy and loss using LSTM (mean) architecture

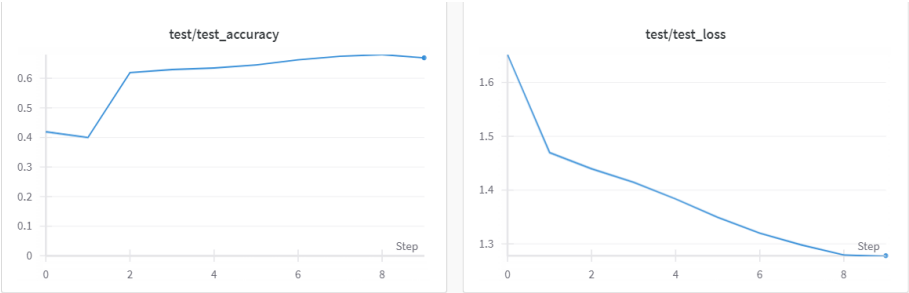


Figure 9. Test accuracy and loss using LSTM (mean) architecture

the data, as the CNN assists in capturing local information more effectively, resulting in fewer disturbances and better outcomes.

3.2.3 Comparison

The baseline biLSTM model achieved an accuracy of 0.5, while BERT with LSTM yielded an accuracy of 0.7, surpassing the biLSTM baseline. This improvement suggests that leveraging BERT's contextual embeddings, combined with the sequential learning capabilities of LSTM, contributes to a more understanding of the text, leading to enhanced classification accuracy. The most substantial advancement is observed with the BERT with CNN model, achieving an accuracy of 0.8. The integration of BERT's contextual embeddings with the parallel processing strength of CNNs appears to be particularly effective in capturing intricate patterns within the text, resulting in a higher accuracy compared to both the biLSTM and BERT with LSTM model.

4 Conclusion

4.1 Our achievement

Through this project, we have achieved:

- Shows that the TF-IDF + Naive Bayes and TF-IDF + SVM models were outperform by the deep learning models.
- The biLSTM model demonstrated how recurrent neural networks excel in capturing sequential dependencies within text, surpassing traditional methods.
- BERT + CNN and BERT + LSTM models emphasize BERT's transformative impact on understanding and categorizing emotions in text through pre-trained contextual embeddings.
- Explored diverse multi-label techniques, traditional machine learning models, and advanced deep learning architectures
- A deeper understanding of the concepts of convolutional neural networks (CNNs) and recurrent neural networks (RNNs) for natural language processing (NLP) tasks.
- An understanding of the different hyperparameters involved in training CNN and RNN models, and their impact on model performance.
- Experience in text data pre-processing tasks.
- Experience in training machine learning models such as Naive Bayes and SVM. Also built and trained a BiLSTM model using TensorFlow.
- Hands-on experience in building and training CNN and LSTM models for text classification using PyTorch.

4.2 Difficulties

However, we still face some difficulties:

- Time and Resource constraint: as this is the time we face our final, so we do not have too much time to fully spend on this project. Also, we do not have enough resource (RAM, GPU, etc) to train the model, even though we have already tried Kaggle and used the GPU supplied by the platform
- Knowledge barrier: there are many knowlegde that is still quite new to us when start this project,hence, it takes us a huge amount of time to learn the necessary stuff for this project.

Due to the difficulties listed above, there are many things that we are not able to cover in this project. Hence, we will reserved them as our future work.

4.3 Future work

Our future work:

- Parameter-tuning: the parameters used in the project are still hand-crafted and can be better if using the correct parameter-tuning method.

- Training with more epochs: to me, we think 10 epochs is somehow sufficient but not enough. Training the models with more epochs may give us better results and more certainty.
- Preprocessing better: The data still have a lot of noise and that may worsen the performance of our model, hence better preprocessing flow may help.

References

[1] *Introducing MLLib's One-vs-rest Classifier* https://antonhaugen.medium.com/introducing-mllibs-one-vs-rest-classifier-402eeab22493

[2] *Deep dive into multi-label classification..! (With detailed Case Study)* https://towardsdatascience.com/journey-to-the-center-of-multi-label-classification-384c40229bff

[3] *SemEval-2018 Task 1: Affect in Tweets (AIT-2018)* https://competitions.codalab.org/competitions/17751

[4] *A Survey of Sentiment Analysis: Approaches, Datasets, and Future Research* https://www.mdpi.com/2076-3417/13/7/4550

[5] *bert-base-uncased-emotion* https://huggingface.co/bhadresh-savani/bert-base-uncased-emotion