COMP 4211 Project Report
Nature Language Processing: Sentiment Analysis on Mental Health Problem Classification

CHAN, Chun Hin (20853893) chchanec@connect.ust.hk
LAW, Hui Nok (20860535) hnlaw@connect.ust.hk

Youtube link: https://youtu.be/Z-gUddy29Cs

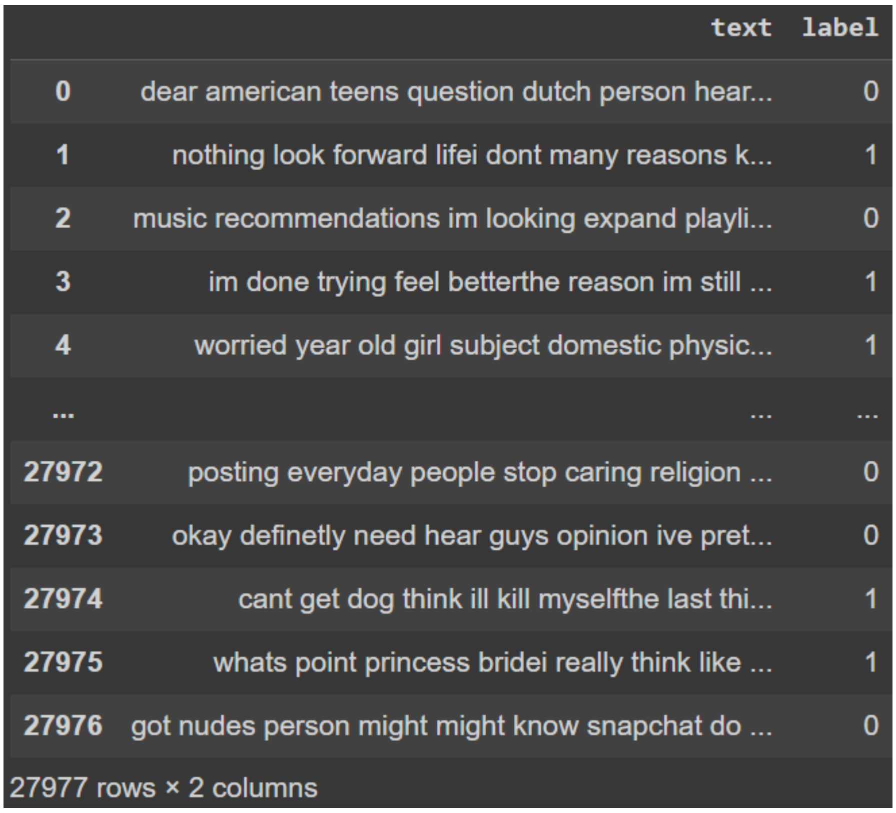
Description of the dataset

Source:

The dataset is obtained from Kaggle in the following website:
https://www.kaggle.com/datasets/reihanenamdari/mental-health-corpus

The name of the dataset is called "Mental Health Corpus".

Structure of the dataset:



	text	label
0	dear american teens question dutch person hear...	0
1	nothing look forward lifei dont many reasons k...	1
2	music recommendations im looking expand playli...	0
3	im done trying feel betterthe reason im still ...	1
4	worried year old girl subject domestic physic...	1
...	...	...
27972	posting everyday people stop caring religion ...	0
27973	okay definetly need hear guys opinion ive pret...	0
27974	cant get dog think ill kill myselfthe last thi...	1
27975	whats point princess bridei really think like ...	1
27976	got nudes person might might know snapchat do ...	0

27977 rows × 2 columns

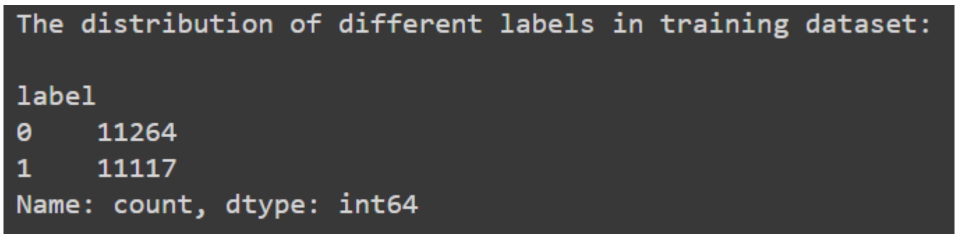
In total, there are 27977 data samples in the dataset. There are 2 columns in the dataset.

The first column (text) contains the comments. They are of string data type.

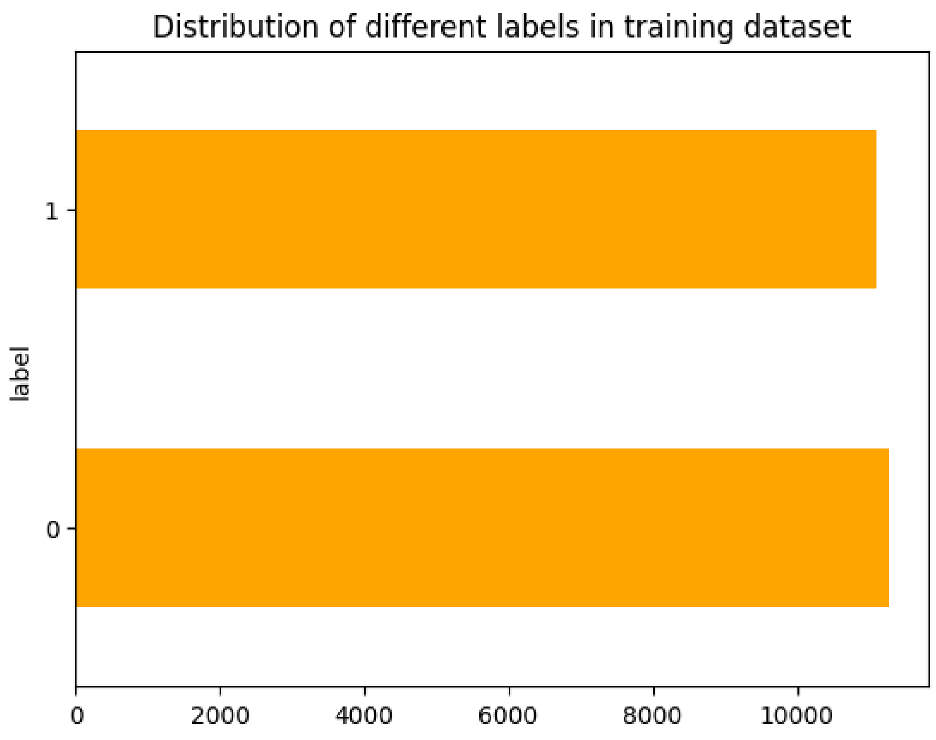
The second column (label) contains the corresponding labels of the comment. Label 0 means that the comment is not poisonous and the corresponding commenter does not have explicit potential mental health illness or issues; whereas label 1 indicates that the comment is poisonous and the corresponding commenter may have potential mental health illness or issues, such that the psychiatrists should pay more attention on these commenter and offer appropriate therapy to them.

Distribution of different labels:

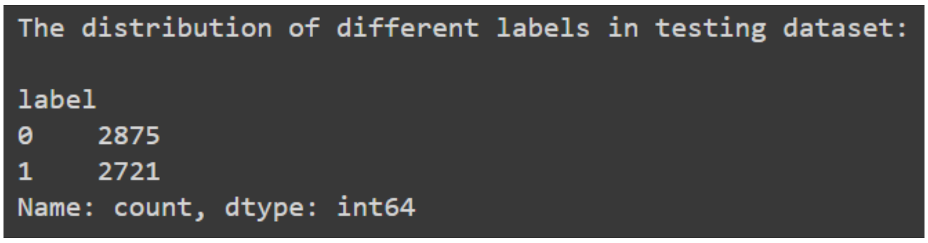
We have split the dataset into training dataset and testing dataset.



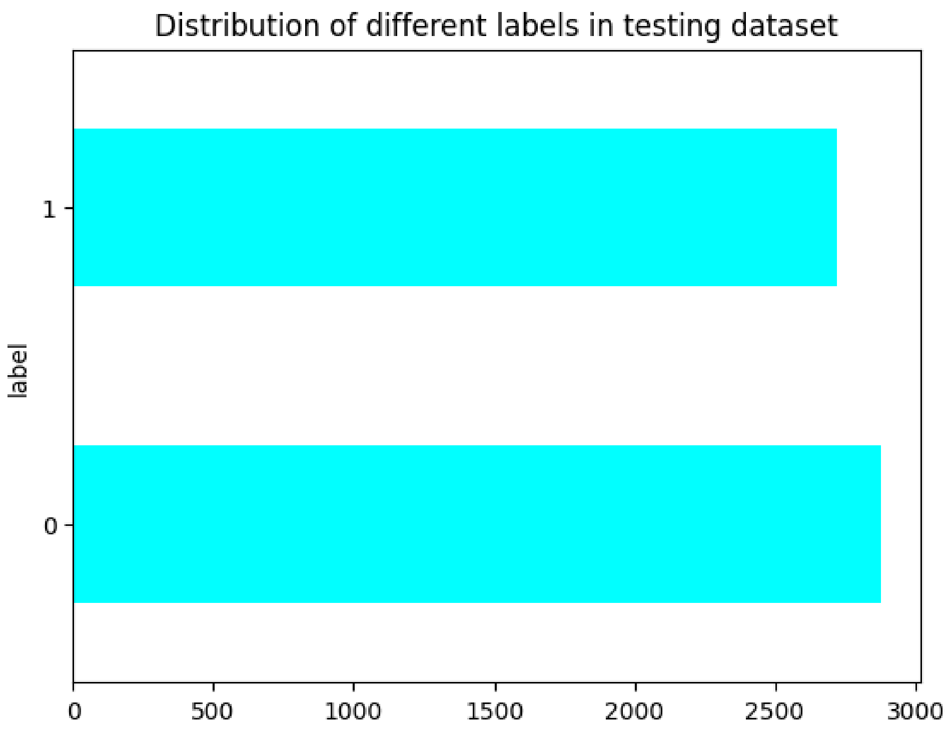
```
The distribution of different labels in training dataset:

label
0    11264
1    11117
Name: count, dtype: int64
```



For the training dataset, 11264 of the data samples are of label 0, while 11117 of the data samples are of label 1.



```
The distribution of different labels in testing dataset:

label
0    2875
1    2721
Name: count, dtype: int64
```



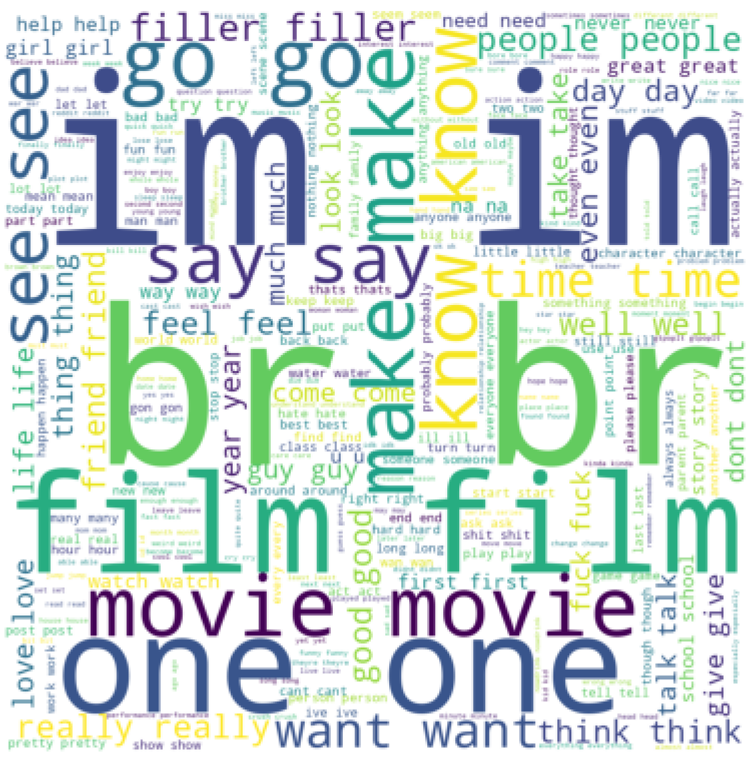
For the testing dataset, 2875 of the data samples are of label 0, while 2721 of the data samples are of label 1.

Therefore, by considering both training and testing datasets, we have 14139 of the data samples are of label 0, while 13838 of the data samples are of label 1. Therefore, the distribution of different labels is quite evenly distributed.

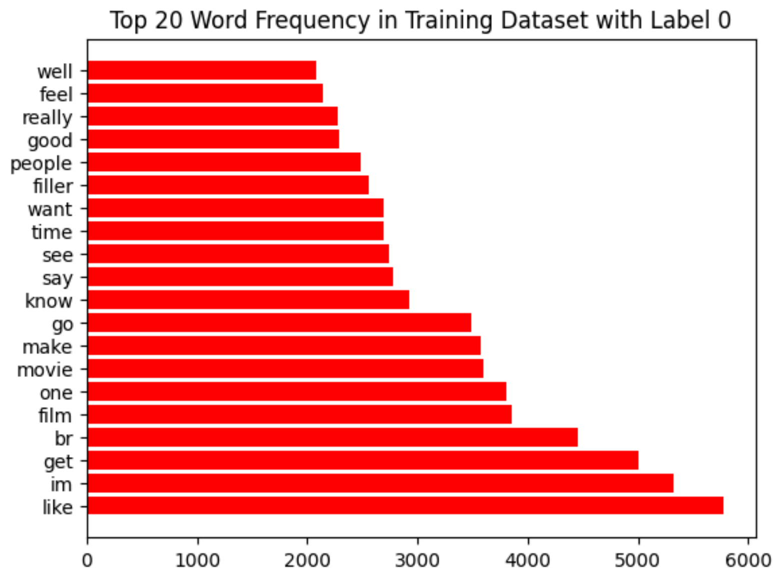
<u>Most frequently occurred words with different labels:</u>

For training dataset:

The word cloud below shows the most frequently occurring words with label 0.



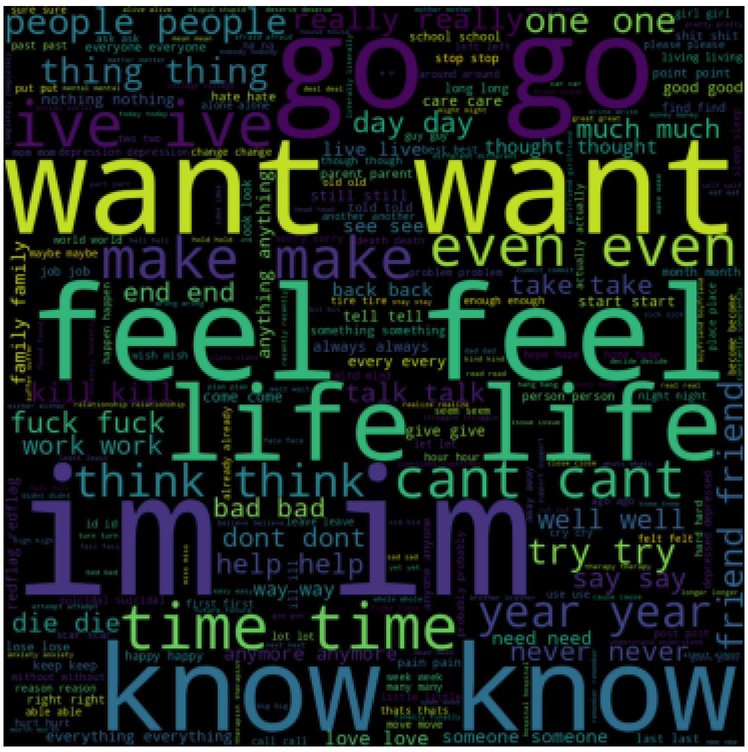
In specific, here is the top 20 most frequently occurred words with label 0:



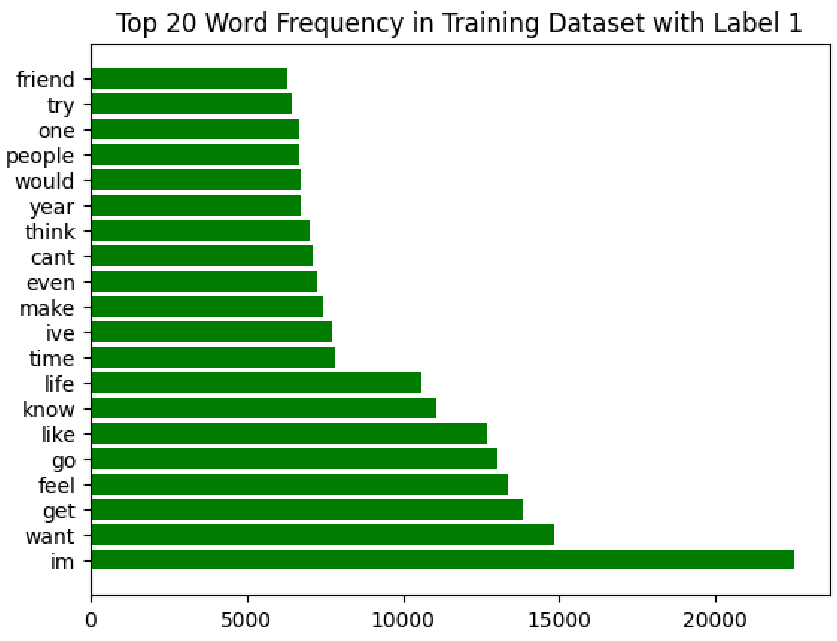
like	5777
im	5326
get	5004
br	4453

film	3851
one	3800
movie	3592
make	3576
go	3485
know	2921
say	2783
see	2740
time	2697
want	2691
filter	2553
people	2487
good	2286
really	2275
feel	2138
well	2085

The word cloud below shows the most frequently occurring words with label 1.



In specific, here is the top 20 most frequently occurred words with label 1:

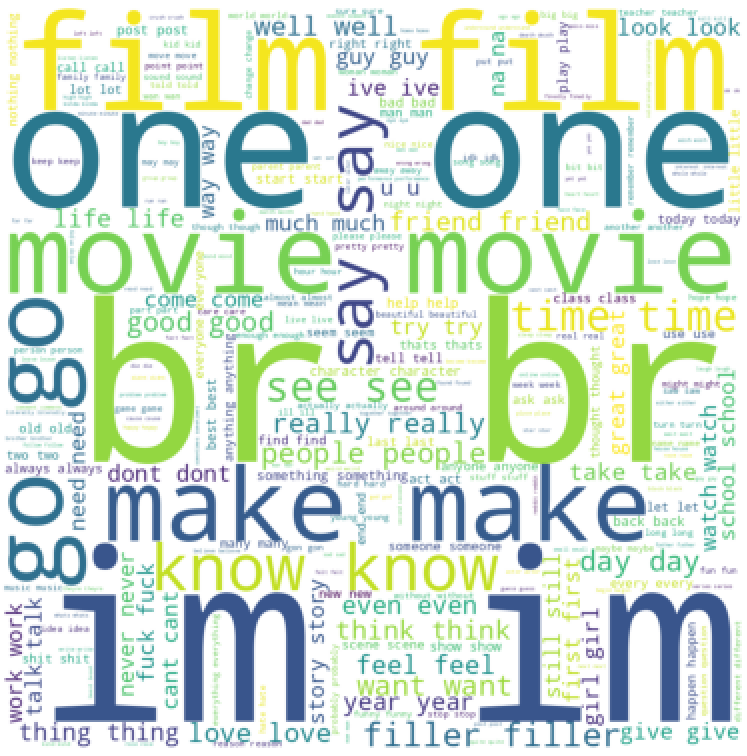


im	22520
want	14818
get	13856

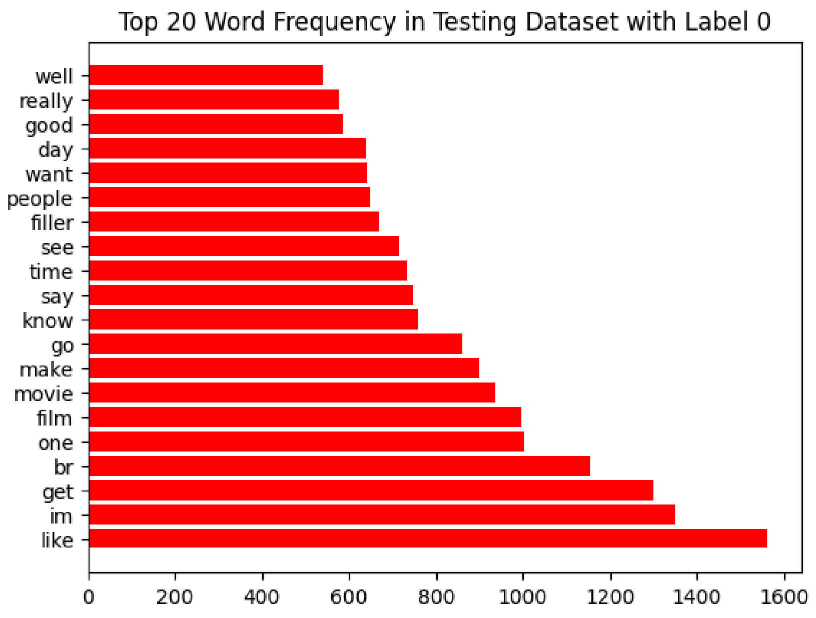
feel	13362
go	13031
like	12712
know	11091
life	10611
time	7813
ive	7744
make	7426
even	7249
cant	7095
think	6999
year	6740
would	6726
people	6694
one	6688
try	6448
friend	6286

For testing dataset:

The word cloud below shows the most frequently occurring words with label 0.



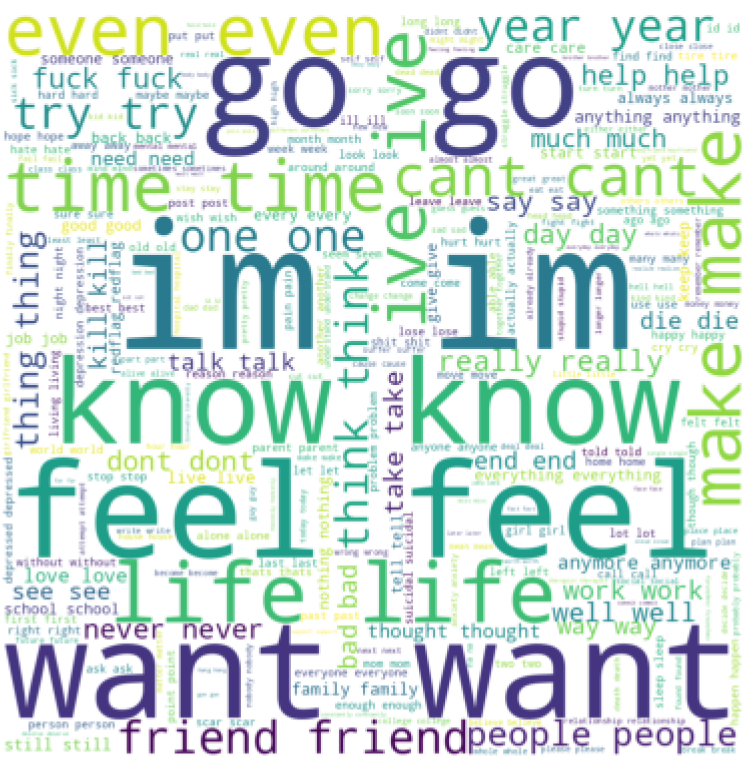
In specific, here is the top 20 most frequently occurred words with label 0:



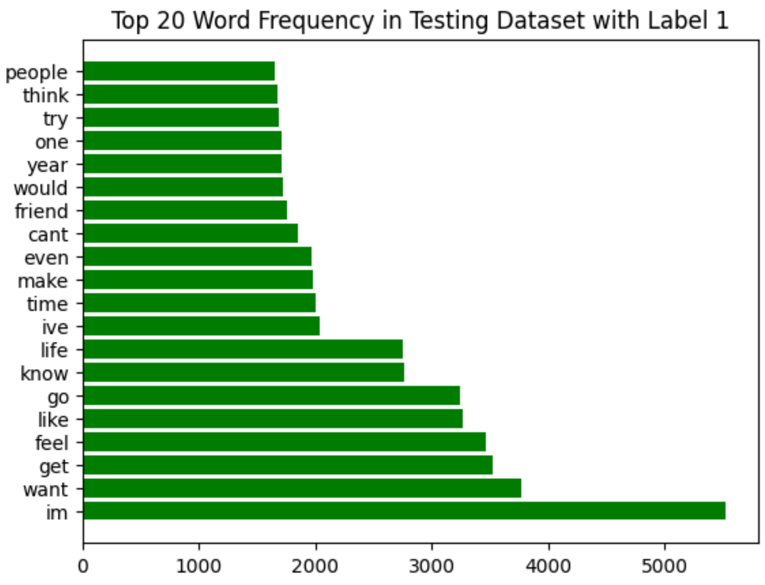
like	1562
im	1350
get	1299

br	1155
one	1001
film	997
movie	935
make	899
go	859
know	758
say	748
time	733
see	714
filler	668
people	647
want	642
day	639
good	585
really	577
well	539

The word cloud below shows the most frequently occurring words with label 1.



In specific, here is the top 20 most frequently occurred words with label 1:



im	5524
want	3762
get	3520
feel	3464

like	3265
go	3244
know	2767
life	2747
ive	2035
time	2006
make	1982
even	1965
cant	1846
friend	1752
would	1719
year	1714
one	1704
try	1680
think	1678
people	1656

Description of the preprocessing

We have several preprocessing functions for the sentences. They are lowercase, tokenization, removing punctuations, removing numbers, stemming, lemmatization, filtering stopwords, n-gram, building corpus, getting one-hot vector and getting index vector.

Lowercase:
This function is used to convert all string sentences or a list of words into lowercase. This can ensure all words in the later process will not be treated as the same word because one letter in the word is upper case (e.g. “Apple” should be treated as the same word as “apple” because they have the same meaning in semantic). This can help reduce the dimensionality of the data in building the corpus and the stopwords removal is only based on lowercase words.

Example output:

```
String - Before:  Hello, we are dealing with the data preprocessing. : Enjoy! 1 2 45
String - After:  hello, we are dealing with the data preprocessing. : enjoy! 1 2 45

List - Before:  ['Hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
List - After:  ['hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'enjoy', '!', '1', '2', '45']
```

Tokenization:
This function utilizes the nltk library to perform the tokenization. It can break the sentence into separate words, punctuations, or numbers, which can help further data preprocessing such as stemming and removing punctuations since they require a single token to process.

Example output:

```
Before tokenization:  Hello, we are dealing with the data preprocessing. : Enjoy! 1 2 45

After tokenization:  ['Hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
```

Removing punctuations:
This function is used to remove all punctuations after tokenization. Punctuation usually does not add too much value to the semantics of the text. Removing them can reduce the complexity of the data without losing much information.

Example output:

```
Before filtering punctuations:  ['Hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
After filtering punctuations:  ['Hello', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', 'Enjoy', '1', '2', '45']
```

Removing numbers:
This function is used to remove all numerical tokens after tokenization. In our task for classifying mental health problems, the number usually does not play a significant role in the classification. Removing them can reduce the complexity of the data without losing much information.

Example output:

```
Before filtering number:  ['Hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
After filtering number:  ['Hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!']
```

Stemming:
This function is used to convert different forms of the words to become the original form. It tries its best to convert the words back to the root form by its algorithm. The reason for "trying its best" is that the algorithm will recognize some pattern for the words and remove the "potential" external stuff (e.g. removing the 's' or 'es' first). But the problem of stemming will sometimes make the words meaningless such as "this" will be converted to "thi".

Example output:

```
Before stemming:  ['Hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
After stemming:  ['hello', ',', 'we', 'are', 'deal', 'with', 'the', 'data', 'preprocess', '.', ':', 'enjoy', '!', '1', '2', '45']
```

Lemmatization:
Since the stemming will sometimes make the words meaningless, lemmatization is a better choice for converting the words back to their root form based on their part of speech.

Example output:

```
Before lemmatization:  ['Hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
After lemmatization:  ['Hello', ',', 'we', 'be', 'deal', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
```

A little comparison between stemming and lemmatization:
Speed: Stemming is much faster than lemmatization because it only needs to follow the rules and directly perform the pruning of the words. However, lemmatization needs to understand the content and each word's part of speech.

Output Quality: As mentioned above, stemming may sometimes give meaningless words as the outcome but lemmatization will not.

Use Cases: Stemming may be more useful in query retrieval scenarios while lemmatization performs better in tasks that require a deeper understanding of the language, like text analysis. Therefore, we utilize the lemmatization for the data preprocessing later on instead of stemming.

Filtering stopwords:
This function is used to remove the stopwords in sentences such as "is", "the". The reason for removing them is that they usually do not carry any meaningful information so removing them can help the reducing of the dimensionality without affecting the sentences meaning.

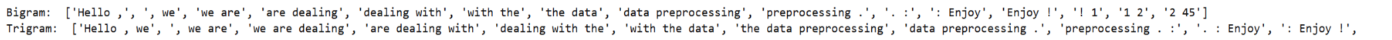
Example output:

```
Before filtering stopwords:  ['Hello', ',', 'we', 'are', 'dealing', 'with', 'the', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
After filtering stopwords:  ['Hello', ',', 'dealing', 'data', 'preprocessing', '.', ':', 'Enjoy', '!', '1', '2', '45']
```

N-gram:

This function is used to help capture the context and semantic information by considering a sequence of ‘n’ words together. This can be particularly useful when the order of words is important to the meaning of the text. When n = 1 (Monogram), the words “not” and “good” will be separately considered which may provide a positive meaning of that sentence. However, the actual semantic meaning can be extracted when n = 2 (Bigram) which gives “not good” in the negative emotion.

Example output:

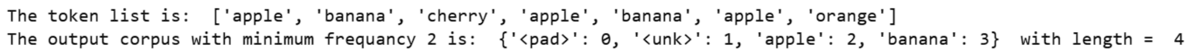


```
Bigram:  ['Hello ,', ', we', 'we are', 'are dealing', 'dealing with', 'with the', 'the data', 'data preprocessing', 'preprocessing .', '. :', ': Enjoy', 'Enjoy !', '! 1', '1 2', '2 45']
Trigram:  ['Hello , we', ', we are', 'we are dealing', 'are dealing with', 'dealing with the', 'with the data', 'the data preprocessing', 'data preprocessing .', 'preprocessing . :', '. : Enjoy', ': Enjoy !',
```

Building corpus:
This function is used to build the corpus for all words with certain criteria. We can restrict the minimum and the maximum frequency of the words that will be added to the corpus. Also, we can limit the size of the corpus to reduce memory usage with a sufficient number of words. The output will be a dictionary which assigns each word with a numerical integer value (the index of the word).

The label <pad> = 0 is used when we want to set all sentences to the same length but the original sentence does not have enough words, so we will place 0 for the padding. Also, not all words can be able to store in the corpus, so the <unk> = 1 is for the unknown words when performing the index encoding.

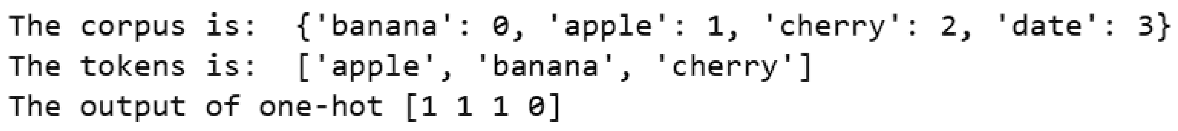
Example output:



```
The token list is:  ['apple', 'banana', 'cherry', 'apple', 'banana', 'apple', 'orange']
The output corpus with minimum frequancy 2 is:  {'<pad>': 0, '<unk>': 1, 'apple': 2, 'banana': 3}  with length =  4
```

Getting one-hot vector:
This function is used to construct the one-hot encoding of the sentence based on the corpus. We will construct a vector which has the same length as the number of elements in the corpus. If the sentence contains those words, the corresponding place to the corpus in the vector will change to 1 to indicate that we have that word in the sentence.

Example output:



```
The corpus is:  {'banana': 0, 'apple': 1, 'cherry': 2, 'date': 3}
The tokens is:  ['apple', 'banana', 'cherry']
The output of one-hot [1 1 1 0]
```

Getting index vector:
This function is used to construct the index vector of the sentence based on the index number in the corpus. We can set the length of the vector we want (i.e. how many words are considered in the first n place in the sentence) and we will place the index number in the corresponding place in the sentence.

Example output:

```
The corpus is:  {'banana': 0, 'apple': 1, 'cherry': 2, 'date': 3}
The tokens is:  ['apple', 'banana', 'cherry']
The output of index vector with length 2 [1 0]
The output of index vector with length 4 [1 0 2 0]
The output of index vector with length 6 [1 0 2 0 0 0]
```

Conclusion of the preprocessing:
We will perform lowercase, tokenization, removing punctuations, removing numbers, lemmatization, filtering stopwords, n-gram and building corpus to be the basic preprocessing step to the model.

For the one-hot vector or index vector, we will only perform either one based on the model we want. For example, the index model and CNN model will have the embedding layer so we utilize the index vector, but the MLP one-hot model will use the one-hot vector.

Description the novelty of the application and Machine learning task(s) performed on the dataset

We have changed the dataset used in this project while leaving the machine learning tasks we would like to do remain unchanged.

First, let's talk about the reason for changing the dataset. Initially, we are using a dataset that is related to news title and corresponding news category classification. However, after we have consulted the opinions from our instructor Professor Yeung, he suggested that news classification is not novel enough as news classification has been widely investigated by many researchers. After further research by us, we have changed the dataset that is related to classifying whether a person has potential mental health issues based on their comments. We have noticed that applying sentiment classification to discover patients with potential mental health illnesses before further therapy is a novel topic that so far there not many researches have done on that before. Hence, we decided to change to using the current dataset.

As the topic of our project suggests, we would perform binary classification on the dataset to classify whether a person has potential mental health issues. This is particularly useful in Hong Kong as recently there are some news that discovered that many people who have mental health illnesses do not receive adequate support from psychiatrists, as it is very hard to identify whether a person has mental health illnesses. Therefore, by utilizing the powerfulness of machine learning techniques, we hope to use binary classification and different model architectures to discover potential patients with mental health problems, so as to help those patients receive appropriate therapy as soon as possible.

In this project, we would use different data preprocessing techniques and model architectures to classify the potential patients based on the words they express.

Description of the hardware and software computing environment

Except for training the two transformers (DistilBERT and XLNet), the running of the remaining codes are done in Google Colaboratory using CPU runtime. Here is the computing environment of Google Colaboratory:

RAM: 12.7 GB RAM
ROM: 107.7 GB Disk memory
CPU: Intel(R) Xeon(R) CPU @ 2.20GHz

For the training of the two transformers (DistilBERT and XLNet), the running of the codes are done in Google Colaboratory using GPU runtime. Here is the computing environment of Google Colaboratory:

RAM: 12.7 GB RAM
ROM: 107.7 GB Disk memory
CPU: Intel(R) Xeon(R) CPU @ 2.20GHz
GPU: Tesla T4 GPU

Machine learning method with its performance

According to the Data, the TP is bottom right corner and the TN is top left corner in the confusion matrix

1. Naive Bayes Classifier

Before the Naive Bayes Rule, we should have a brief discussion on the Bayes Rule first. The Bayes' Rule was named in the 18th century by mathematician Thomas Bayes. This rule describes how to update a belief given some evidence. The equation is as follows and E is evidence, B is belief, P(x) is the probability of x.

$$P(B|E) = P(B) \times \frac{P(E|B)}{P(E)}$$

The Bayes' Rule can be more generalized to more than one belief which are the classes that we want to classify and more than one evidence which is the word in each sentence to help us to perform the classification. The equation is as follows:

$$\begin{aligned} P(B_i|E) &= \frac{P(B_i)P(E|B_i)}{P(E|B_1)P(B_1) + P(E|B_2)P(B_2) + P(E|B_3)P(B_3) + \cdots + P(E|B_n)P(B_n)} \\ &= \frac{P(B_i)P((e_1, e_2, e_3, \ldots, e_d)|B_i)}{\sum_{j=1}^{n} P((e_1, e_2, e_3, \ldots, e_d)|B_j)P(B_j)} \end{aligned}$$

After introducing the Bayes Rule with its general form, the word "Naive" gives further assumptions to the Bayes Rule which assumes each piece of evidence is independent and equally contributes to the classes. To fit into our problem, the independent means each word will not affect other words' appearance. Equal contribution means different words no matter if it is a general word or with a specific meaning in mental health, they will have the same weight to contribute to the class. Based on the properties of Naive, we can derive the equation as follows:

$$\begin{aligned} P(B_i|E) &= \frac{P(B_i)P((e_1, e_2, e_3, \ldots, e_d)|B_i)}{\sum_{j=1}^{n} P((e_1, e_2, e_3, \ldots, e_d)|B_j)P(B_j)} \\ &= \frac{P(B_i)P(e_1|B_i)P(e_2|B_i)P(e_3|B_i)\ldots P(e_d|B_i)}{\sum_{j=1}^{n} P(e_1|B_j)P(e_2|B_j)P(e_3|B_j)\ldots P(e_d|B_j)P(B_j)} \end{aligned}$$

The aim of the task is to perform the classification instead of knowing the actual probability of all beliefs with the given evidence. Therefore, we can realize that the denominator will be the same in the calculation which means the right to consider which class it belongs to is only by the nominator. As a result, we can only take the argmax from the nominator to extract the corresponding class. The equation can be stated as follows:

$$B_{NB} = argmax_{B_i} P(B_i)(P(e_1|B_i)P(e_2|B_i)P(e_3|B_i) \cdots P(e_d|B_i))$$

As the lecture describes the problem of lots of floating point numbers, which are less than 1, multiplied together will cause the problem of floating point underflow. Therefore, to prevent this problem, the calculation will also perform all computations by summing logs of probabilities instead of directly multiplying all probabilities. The final equation for the classification becomes:

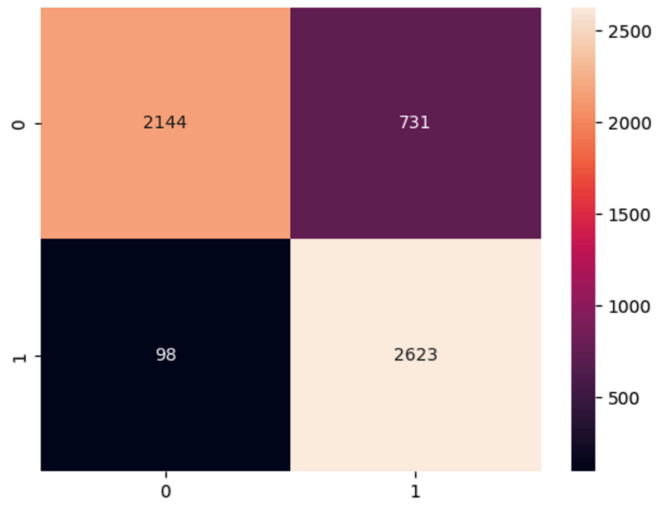
$$B_{NB} = argmax_{B_i} \left(logP(B_i) + \sum_{n=1}^{d} logP(e_n|B_i) \right)$$

Another problem with this approach is that if such a word does not appear in the training data, then the probability of that evidence will be 0 and it leads to the entire probability which is used for classification will be always 0. It will make the prediction always predict to another class if that word appears or just randomly pick one if both predicted probabilities are 0. To resolve this problem, we can apply the approach that adding the count of each word by 1 which is called Laplace Correction so that the problem can be prevented.
After the description of the machine learning approach, let's talk about the implementation with its parameter setting and performance.

The implementation is as follows. First, we utilize CountVectorizer() to get the counting number of each word in each sentence with an assigned index number to it. Afterwards, we pass through the data to MultinomialNB with two settings, one is setting alpha = 0 which means we will not perform the Laplace Correction (without smoothing), and another one is setting alpha = 1 by default which means we will perform Laplace Correction (with smoothing) by adding 1 to each word counting.

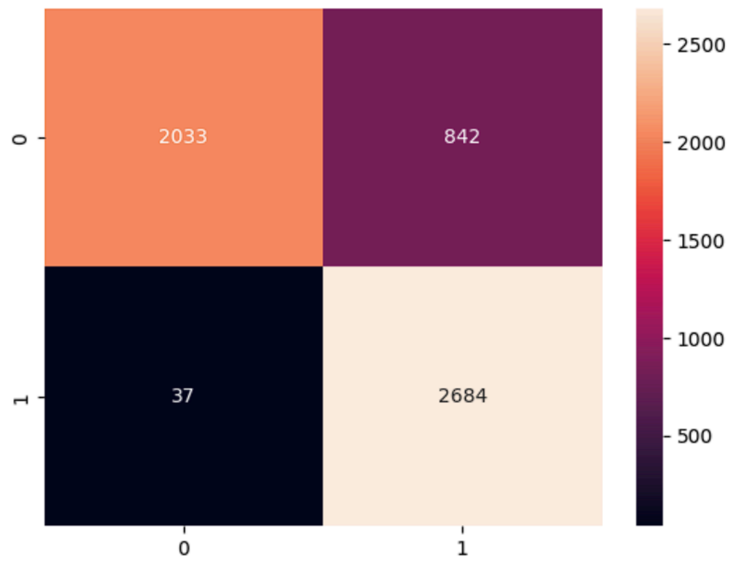
The performance of alpha = 0:

```
Naive Bayes Accuracy without smoothing:  0.8518584703359543
Precision: 0.782051282051282
Recall: 0.9639838294744579
F1 Score: 0.8635390946502058
```



The performance of alpha = 1 as default:

```
Naive Bayes Accuracy with smoothing:  0.8429235167977126
Precision: 0.7612024957458877
Recall: 0.9864020580668872
F1 Score: 0.859292460380983
```



We will analyze the insight by using alpha = 0 and alpha = 1 with its performance evaluation in the next part.

2. One-Hot Model by MLP

In text classification, encoding is required to transform words or sentences into a numerical format, enabling computers to comprehend their meaning. This numerical representation allows machine learning models to process and analyze the text accurately by extracting the information in vector form. As the name stated, we will first perform the one-hot encoding of the sentence and then use MLP for the model design.

The implementation procedure is as follows. First, we will build the corpus and set the number of words saved in it before encoding by using the ***Building corpus*** preprocessing function. We have tried two different lengths of the corpus which are 100 and 1000 and we will explain the resulting difference of them in the next part. Afterwards, use ***Getting one-hot vector*** to construct the one-hot vector for each sentence in the training, validation and testing data.

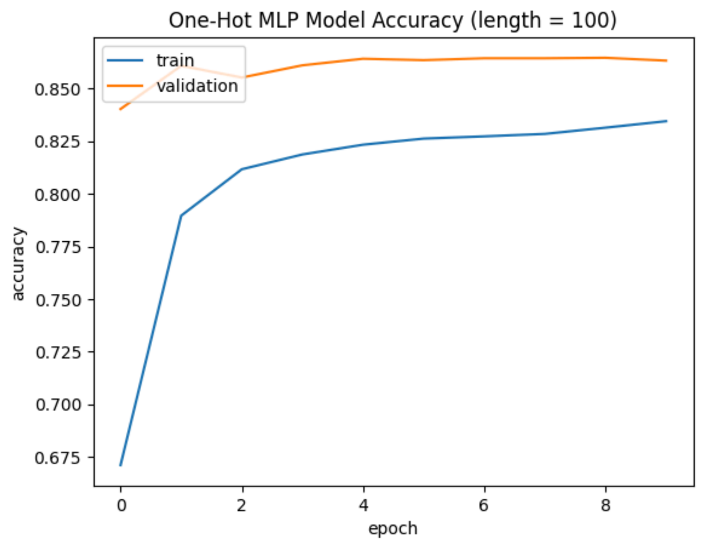
The model setting in the vector length of 100 is as follows:

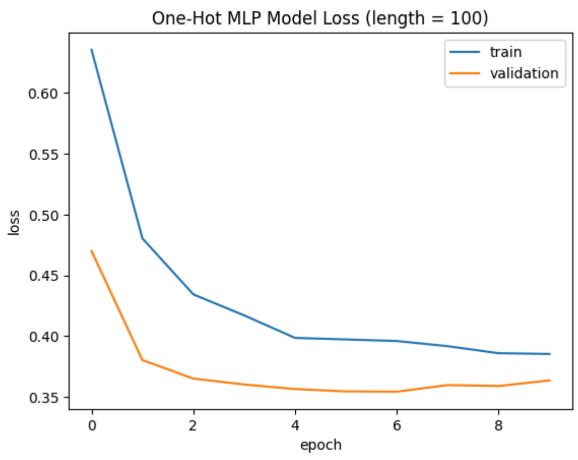
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_5 (Flatten)         (None, 100)               0

 dense_67 (Dense)            (None, 16)                1616

 dropout_24 (Dropout)        (None, 16)                0

 dense_68 (Dense)            (None, 8)                 136

 dense_69 (Dense)            (None, 8)                 72

 dropout_25 (Dropout)        (None, 8)                 0

 dense_70 (Dense)            (None, 4)                 36

 dense_71 (Dense)            (None, 2)                 10

=================================================================
Total params: 1870 (7.30 KB)
Trainable params: 1870 (7.30 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

First, we use the flattened layer to flatten the input sequence since the input is in a 2D vector but the MLP Dense Layer requires a 1D vector as the input. Then followed by a Dense layer with 16 hidden units and a Dropout layer with parameter 0.5. After that, two Dense layers with both 8 hidden units and a Dropout layer with parameter 0.5. Last, a Dense layer with 4 hidden units and a Dense layer with 2 hidden units. Except for the last Dense layer uses the "softmax" activation function, other Dense layers use "ReLU".

For the training setting, the optimizer is using "adam", the loss is using "binary_crossentropy" since we are classifying two labels only and the metrics is using "accuracy". Also, we set the batch size as 64 and we will run 10 epochs for the training.

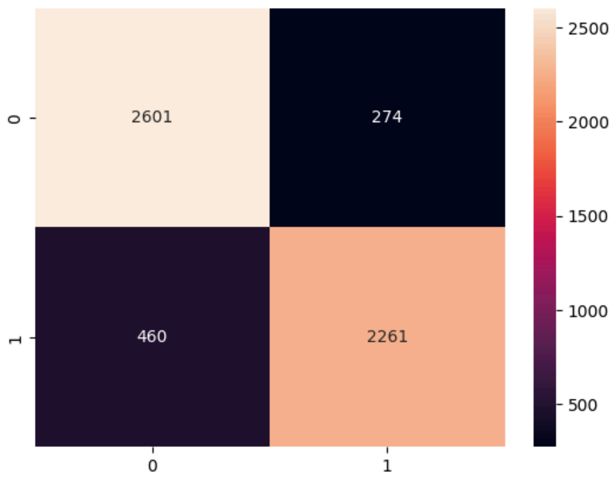
The training and validation accuracy with loss is the graph as follows:





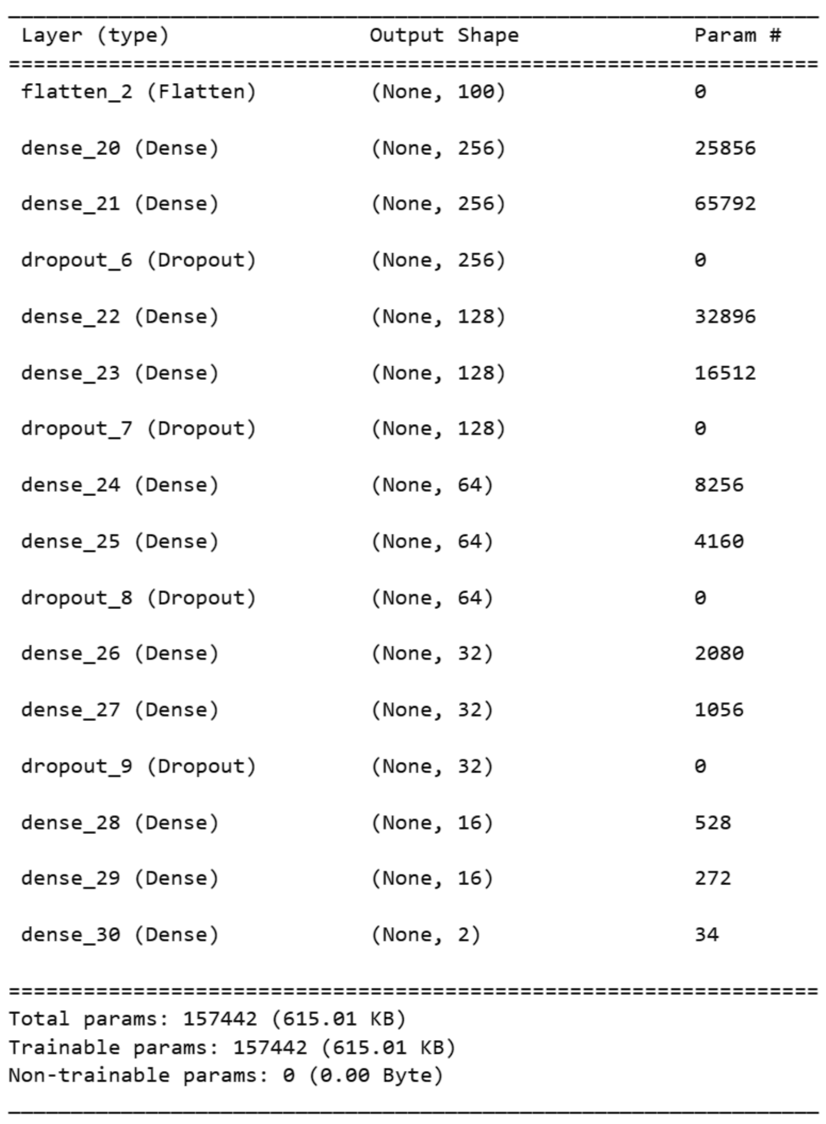
The training accuracy is increasing while loss is decreasing. The accuracy validation set is quite stable and the loss is decreasing.

The testing performance is as follows:



The accuracy = 86.88%, the precision = 89.19%, the recall = 83.09%, the F1 score = 86.04%

Another more complex model setting in vector length of 100 is as follows:

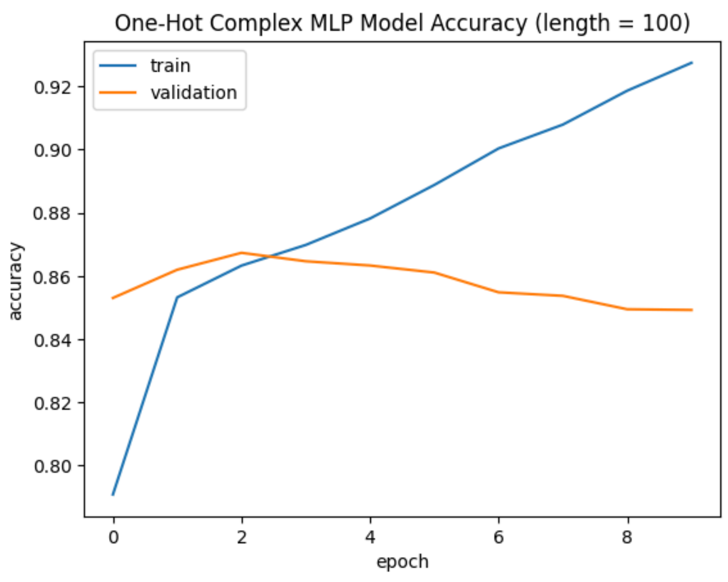


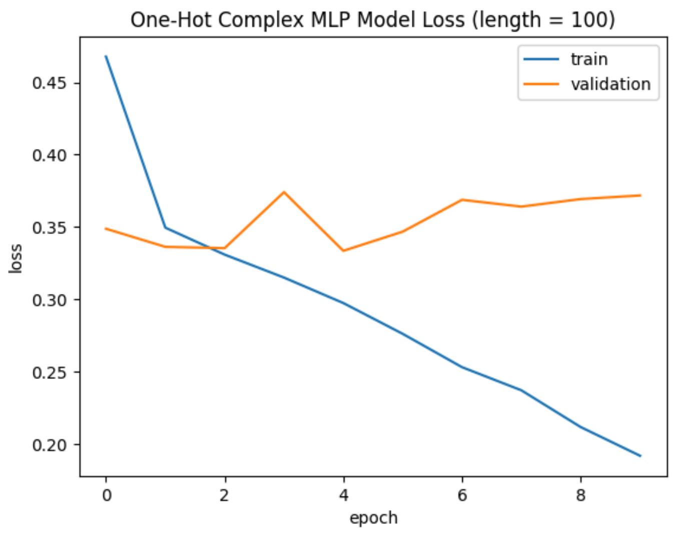
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_2 (Flatten)         (None, 100)               0

 dense_20 (Dense)            (None, 256)               25856

 dense_21 (Dense)            (None, 256)               65792

 dropout_6 (Dropout)         (None, 256)               0

 dense_22 (Dense)            (None, 128)               32896

 dense_23 (Dense)            (None, 128)               16512

 dropout_7 (Dropout)         (None, 128)               0

 dense_24 (Dense)            (None, 64)                8256

 dense_25 (Dense)            (None, 64)                4160

 dropout_8 (Dropout)         (None, 64)                0

 dense_26 (Dense)            (None, 32)                2080

 dense_27 (Dense)            (None, 32)                1056

 dropout_9 (Dropout)         (None, 32)                0

 dense_28 (Dense)            (None, 16)                528

 dense_29 (Dense)            (None, 16)                272

 dense_30 (Dense)            (None, 2)                 34

=================================================================
Total params: 157442 (615.01 KB)
Trainable params: 157442 (615.01 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

First, we use the flattened layer to flatten the input sequence. Then followed by two Dense layer with 256 hidden units and a Dropout layer with parameter 0.5. Then followed by two Dense layer with 128 hidden units and a Dropout layer with parameter 0.5. Then followed by two Dense layer with 64 hidden units and a Dropout layer with parameter 0.5. Then followed by two Dense layer with 32 hidden units and a Dropout layer with parameter 0.5. Then followed by two Dense layer with 16 hidden units and a Dense layer with 2 hidden units. Except the last layer is using "softmax", other Dense layers are using "”ReLU" as activation function.

For the training setting, the optimizer is using "adam", the loss is using "binary_crossentropy" since we are classifying two labels only and the metrics is using "accuracy". Also, we set the batch size as 64 and we will run 10 epochs for the training.

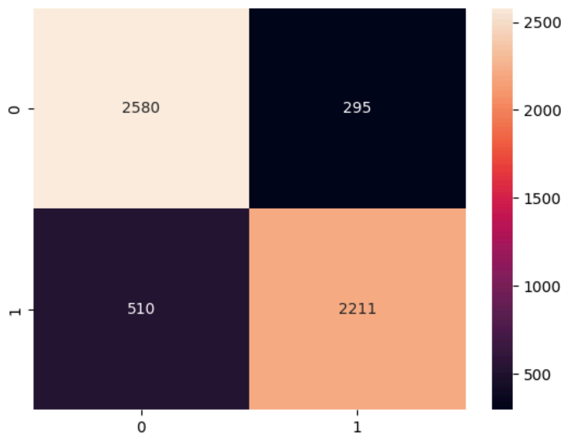
The training and validation accuracy with loss is the graph as follows:





The training accuracy is increasing while loss is decreasing. The validation accuracy and loss is stable.

The testing performance is as follows:



The accuracy = 85.61%, the precision = 88.23%, the recall = 81.26%, the F1 score = 84.60%

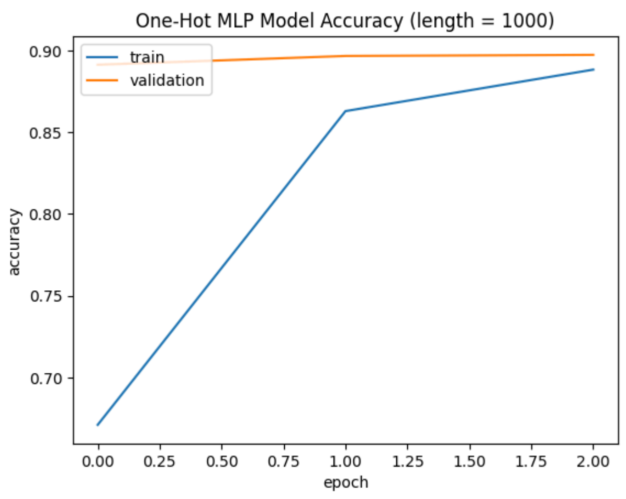
The model setting in the vector length of 1000 is as follows:

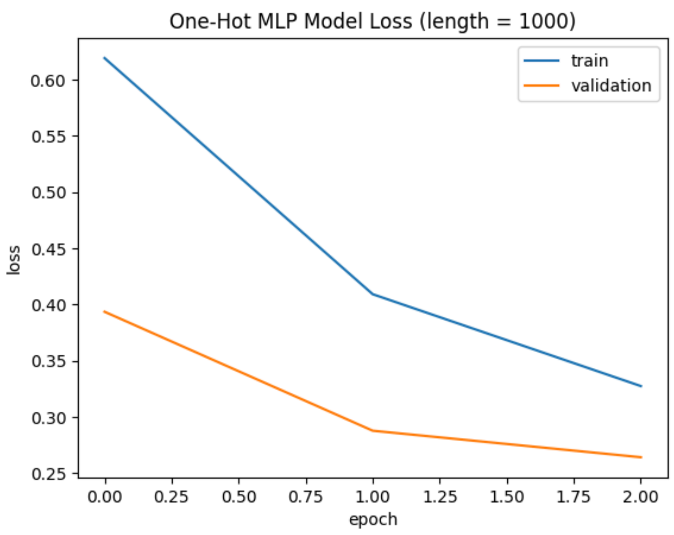
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 flatten_2 (Flatten)         (None, 1000)              0         
                                                                 
 dense_30 (Dense)            (None, 16)                16016     
                                                                 
 dropout_8 (Dropout)         (None, 16)                0         
                                                                 
 dense_31 (Dense)            (None, 8)                 136       
                                                                 
 dense_32 (Dense)            (None, 8)                 72        
                                                                 
 dropout_9 (Dropout)         (None, 8)                 0         
                                                                 
 dense_33 (Dense)            (None, 4)                 36        
                                                                 
 dense_34 (Dense)            (None, 2)                 10        
                                                                 
=================================================================
Total params: 16270 (63.55 KB)
Trainable params: 16270 (63.55 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Same as previously, we need to flatten the input data first. Then followed by a Dense layer with 16 hidden units and a Dropout layer with parameter 0.5. After that, two Dense layers with both 8 hidden units and a Dropout layer with parameter 0.5. Last, a Dense layer with 4 hidden units and a Dense layer with 2 hidden units. Except for the last Dense layer uses the "softmax" activation function, other Dense layers use "ReLU".

For the training setting, the optimizer is using "adam", the loss is using "binary_crossentropy" since we are classifying two labels only and the metrics is using "accuracy". Also, we set the batch size as 64 and we will run 3 epochs for the training.

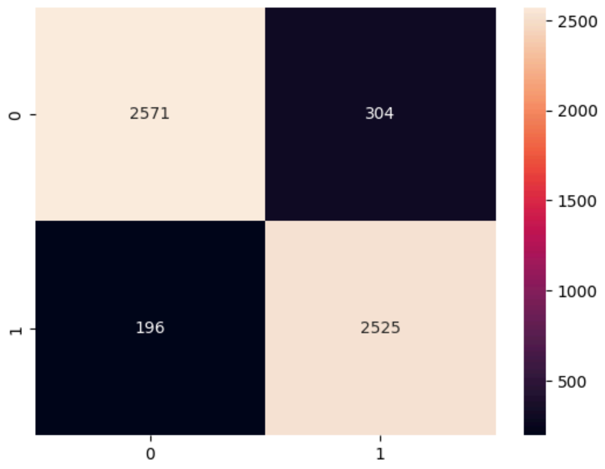
The training and validation accuracy with loss is the graph as follows:





The training accuracy is increasing while loss is decreasing. The accuracy validation set is quite stable and the loss is decreasing.

The testing performance is as follows:



The accuracy = 91.07%, the precision = 89.25%, the recall = 92.80%, the F1 score = 90.99%

3. Index Model by RNN

Similar to the previous model, we also need to encode the text sentences before passing through the model. However, we do not use the one-hot encoding anymore, we change to use the index vector to construct the input sequence (We will discuss the difference in the next part of the comparison). The approach of building the index vector is first we also build the corpus but without limiting the number of words stored inside by the ***Building corpus*** preprocessing function. Then, using the ***Getting index vector*** function to get the sentence vector based on their index in the corpus form the first 100 words in each sentence for both training and testing data.

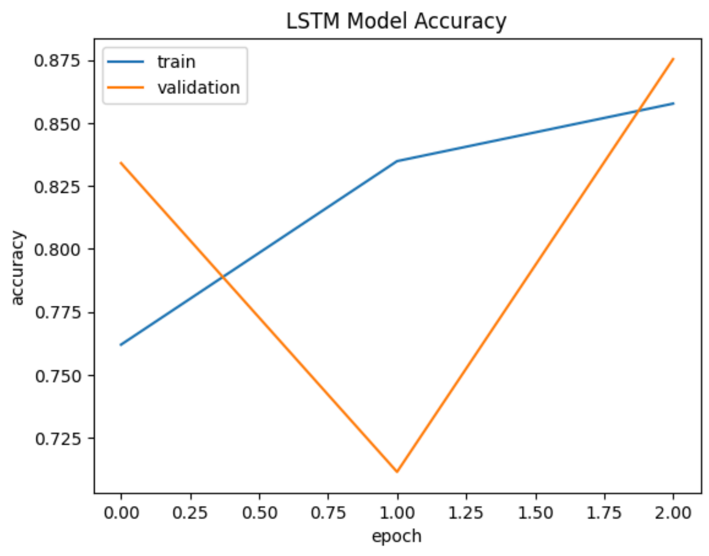
The model design is as follows:

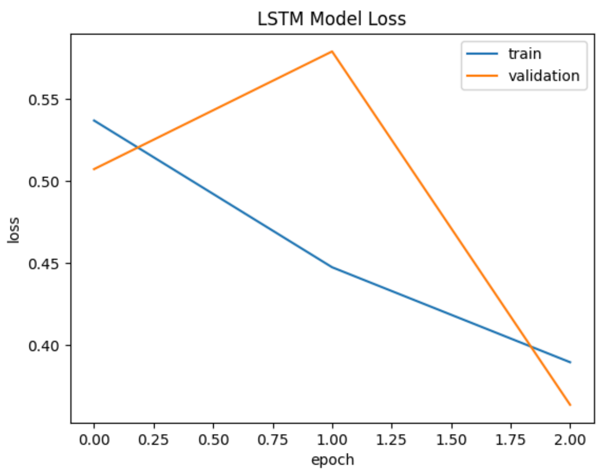
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_10 (Embedding)    (None, None, 32)          1729696

 lstm_26 (LSTM)              (None, None, 16)          3136

 dropout_22 (Dropout)        (None, None, 16)          0

 lstm_27 (LSTM)              (None, 8)                 800

 dense_65 (Dense)            (None, 2)                 18

=================================================================
Total params: 1733650 (6.61 MB)
Trainable params: 1733650 (6.61 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

First, we will have an embedding layer which is to learn a better representation of each word in the corpus. The input_dim parameter is set as the length of the corpus and the output_dim is set as 32 which means we would like to encode each word into 32 dimensionality for representation. Following that is the LSTM layer which has 16 units in the layer with return_sequences = True to keep all the units output for further use. Then we have a Dropout layer with parameter 0.5. After that, we have one more LSTM layer with 8 units but the return_sequences sets as False as default which means we only need the last unit's output and drop others. Following that is the Dense layer with 2 units and a "softmax" activation function for classification.

For the training setting, the optimizer is using "adam", the loss is using "binary_crossentropy" since we are classifying two labels only and the metrics is using "accuracy". Also, we set the batch size as 64 and we will run 3 epochs for the training.

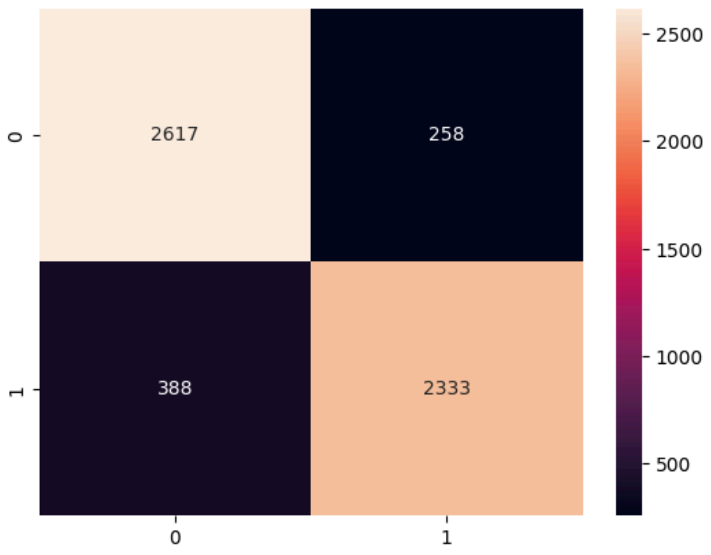
The training and validation accuracy with loss is the graph as follows:





The training accuracy is increasing while loss is decreasing. Also similar case in the validation set.

The testing performance is as follows:

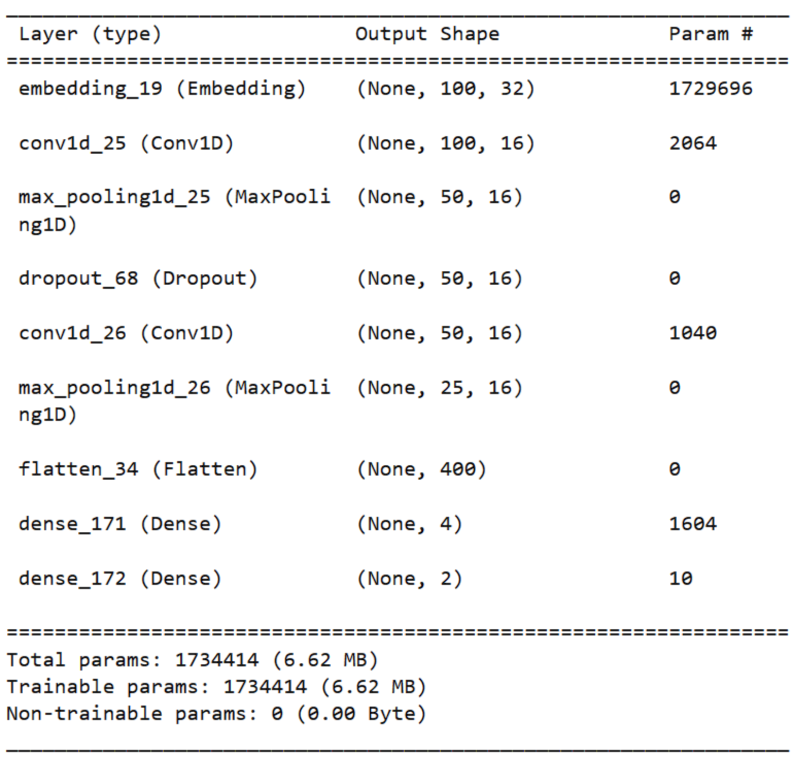


The accuracy = 88.46%, the precision = 90.04%, the recall = 85.74%, the F1 score = 87.84%

4. CNN Model

As with the previous model, the index encoding method is the same. The approach of building the index vector is first we also build the corpus but without limiting the number of words stored inside by the ***Building corpus*** preprocessing function. Then, using the ***Getting index vector*** function to get the sentence vector based on their index in the corpus form the first 100 words in each sentence for both training and testing data.

The model design is as follows:

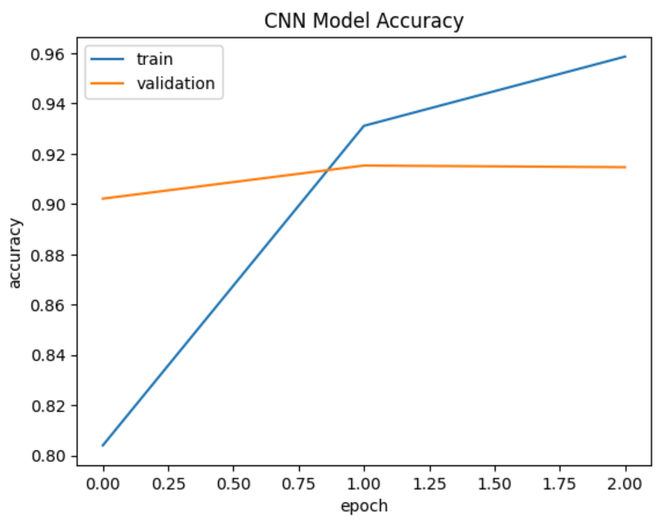


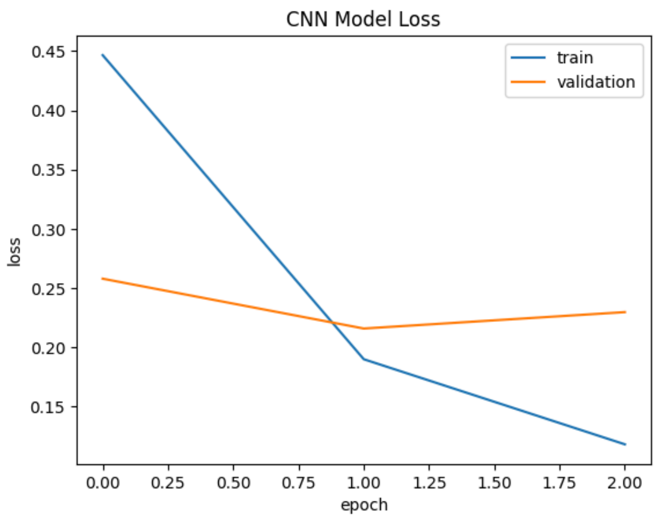
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_19 (Embedding)    (None, 100, 32)           1729696

 conv1d_25 (Conv1D)          (None, 100, 16)           2064

 max_pooling1d_25 (MaxPooli  (None, 50, 16)            0
 ng1D)

 dropout_68 (Dropout)        (None, 50, 16)            0

 conv1d_26 (Conv1D)          (None, 50, 16)            1040

 max_pooling1d_26 (MaxPooli  (None, 25, 16)            0
 ng1D)

 flatten_34 (Flatten)        (None, 400)               0

 dense_171 (Dense)           (None, 4)                 1604

 dense_172 (Dense)           (None, 2)                 10

=================================================================
Total params: 1734414 (6.62 MB)
Trainable params: 1734414 (6.62 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

First, we will have an embedding layer which is to learn a better representation of each word in the corpus. The input_dim parameter is set as the length of the corpus and the output_dim is set as 32 which means we would like to encode each word into 32 dimensionality for representation. Also, we need to use the Dense layer with “ReLU” as the activation, so we need to set the input_length = 100 at the embedding layer. After the embedding layer, we have a Conv1D layer with filters = 16, kernel_size = 4, padding = 'same', activation = 'relu'. Then we have a MaxPooling1D layer with pooling size = 2 and a Dropout layer with parameter = 0.5. Afterwards, we have a Conv1D layer with filters = 16, kernel_size = 4, padding = 'same', activation = 'relu'. Then we have a MaxPooling1D layer with pooling size = 2 and a Dropout layer with parameter = 0.5. Then we have a flattened layer to prepare the fully connected layer. We have a Dense layer with hidden units = 4 and the activation function is “ReLU”. Last we have a Dense layer with hidden units = 2 and the activation function is “softmax” for classification.

For the training setting, the optimizer is using “adam”, the loss is using “binary_crossentropy” since we are classifying two labels only and the metrics is using “accuracy”. Also, we set the batch size as 64 and we will run 3 epochs for the training.

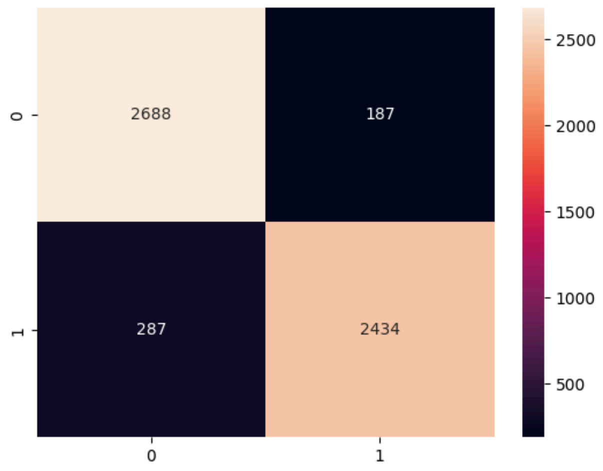
The training and validation accuracy with loss is the graph as follows:





The training accuracy is increasing while loss is decreasing. Also similar case in the validation set but with less variation.

The testing performance is as follows:



The accuracy = 91.53%, the precision = 92.87%, the recall = 89.45%, the F1 score = 91.13%

5. DistilBERT Model

The “Distil” stands for distillation which means this model is a simple version of the original BERT model. As the prior knowledge taught in the lecture on the BERT model, now let’s dive into the details of the DistilBERT Model.

Firstly, about the architecture of the DistilBERT model. It also needs an embedding vector for each word and the positional encoding. Afterwards, the input will be passed into the encoder which is similar to the original BERT model and the DistilBERT model only contains 6 encoders. The architecture of the encoder is the same as the BERT model which contains the self-attention, add & normalize, and feed-forward neural network. After passing through all encoders, the first input special [CLS] token will be passed to a Dense Layer with a softmax or sigmoid activation function (depending on the task) to perform our downstream classification task.

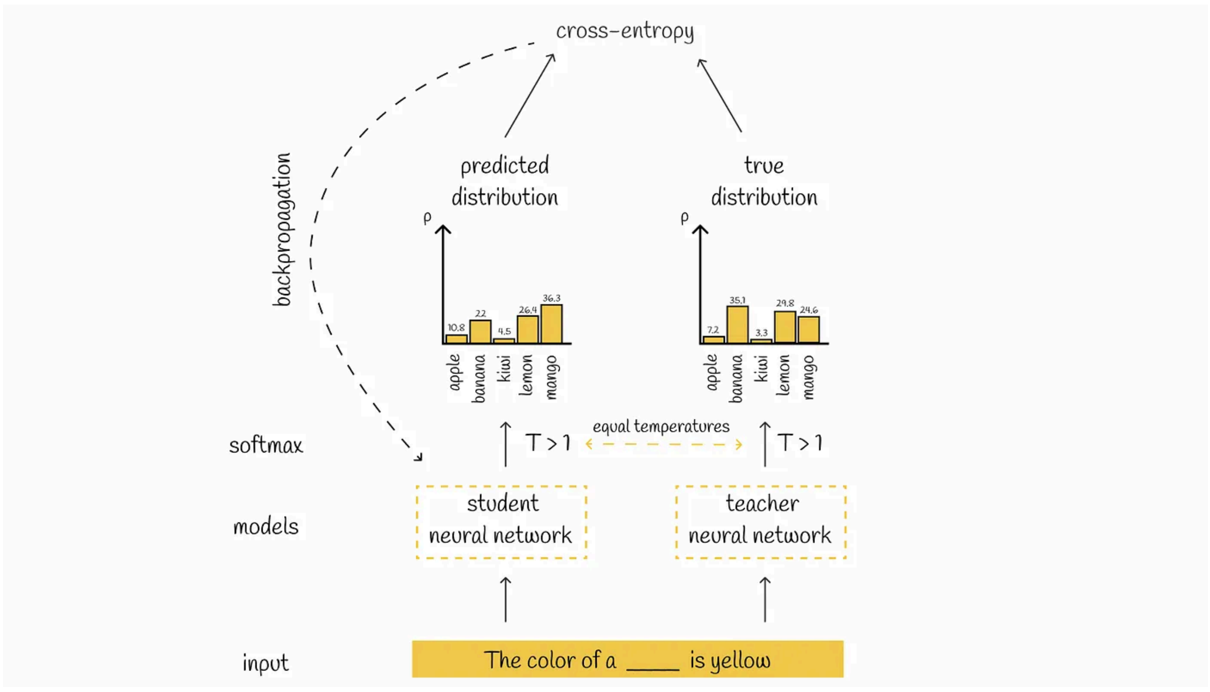
Before training the DistilBERT model, we already had the original BERT model. First, we will utilize the pre-trained BERT model’s weight to initialize the DistilBERT model’s weight. The approach is as follows, for the first encoder in the DistilBERT model, it will pick either layer 1 or 2 weights from the BERT model, for the second encoder in the DistilBERT model, it will pick either layer 3 or 4 weights from the BERT model, and so on until the end of the 6th encoder.

Then, we need to train the DistilBERT model with 3 loss functions which are distillation loss, masked language modeling loss, and cosine embedding loss. Since we want the DistilBERT model similar to the BERT model, we can treat the BERT model as a “teacher” and the DistilBERT model as a “student”.

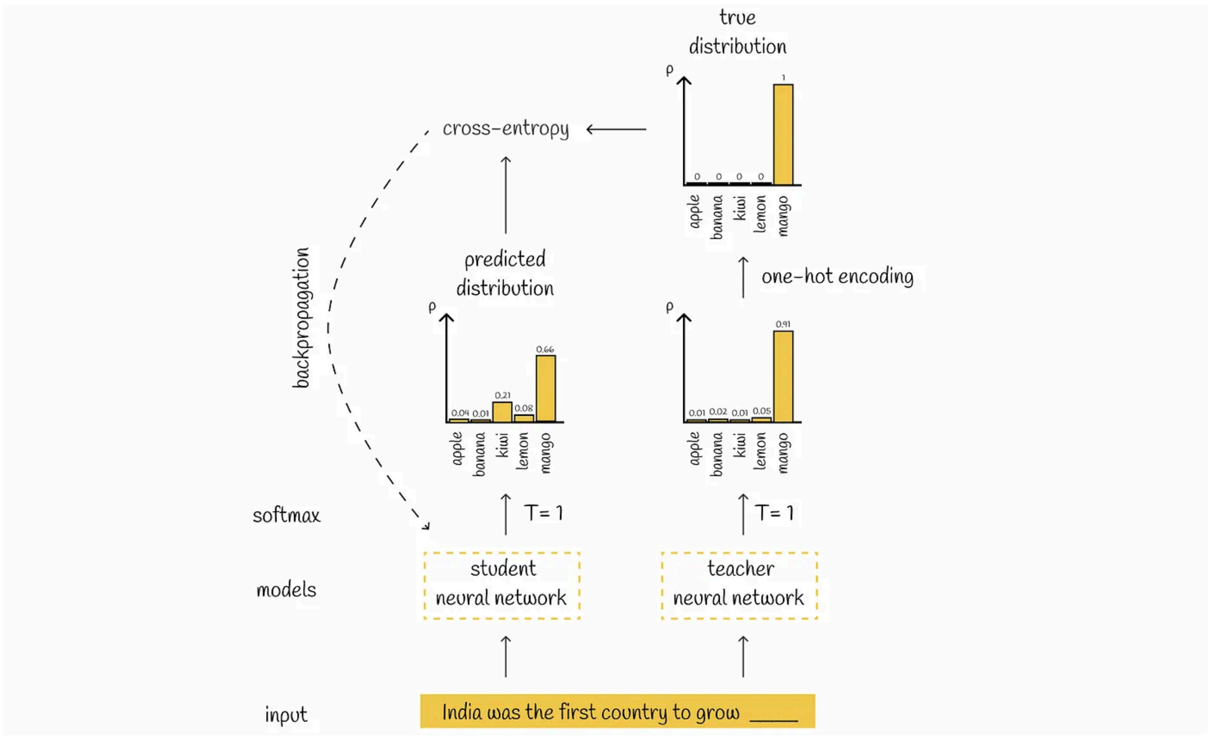
The first distillation loss is (cross-entropy loss):

$$L_{ce} = \sum_i t_i * \log(s_i)$$

The ti is the predicted output from the BERT model after the softmax and the si is the predicted output from the DistilBERT model after the softmax. The function of this loss is to help the DistilBERT model learn from the BERT model, which is the student and teacher relationship.



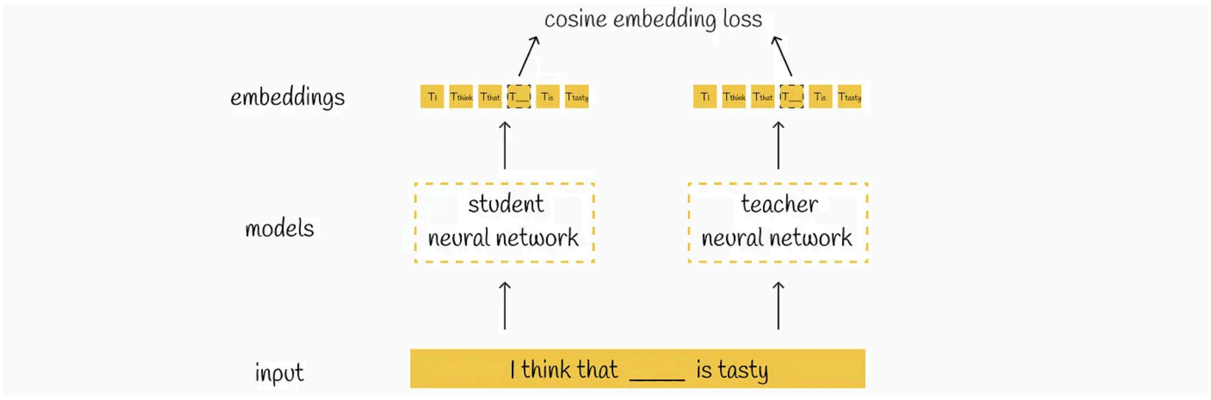
The second masked language modeling loss is the same as the BERT model which uses the cross-entropy loss for the predicted masked word and the original encoded text word. It can be done by randomly masking 15% of the input and predicting the masked words using the DistilBERT model. For this loss, the function is to help the DistilBERT model learn the bidirectional representation of the sentence.



The last loss is called cosine embedding loss.

$$L_{cos}(x, y) = 1 - \cos(x, y) = 1 - \frac{x \cdot y}{\|x\| \cdot \|y\|}$$

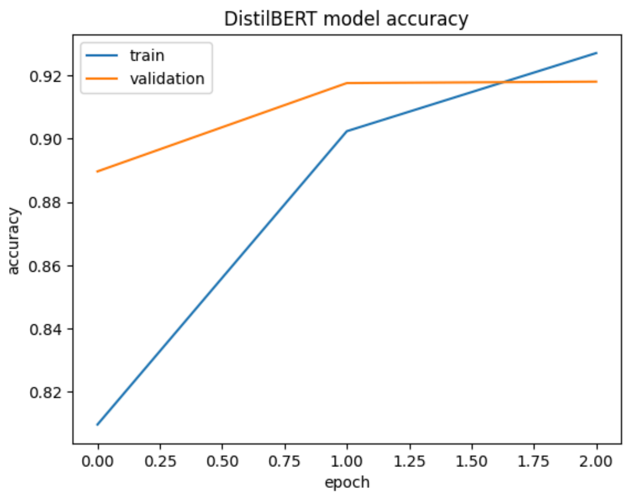
The x and y are the predicted output after the BERT model's encoders and the DistilBERT model's encoders respectively. It can help the DistilBERT model construct the embedding and mimic the behaviour of the BERT model as closely as possible.

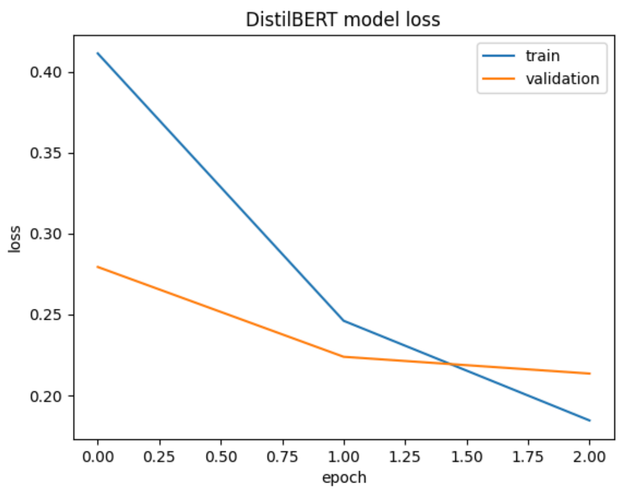


As a result, these three loss functions help the DistilBERT model learn to perform similar behaviour to the BERT model and learn from the ground truth label to get the correct result.

To implement the model, first, we load the DistilBERT model from the tenserflow library and then add a Dense layer with sigmoid and one unit for performing the classification task. For the training setting, the optimizer is using "adam", the loss is using "binary_crossentropy" since we are classifying two labels only and the metrics is using "accuracy". Also, we set the batch size as 64 and we will run 3 epochs for the training.

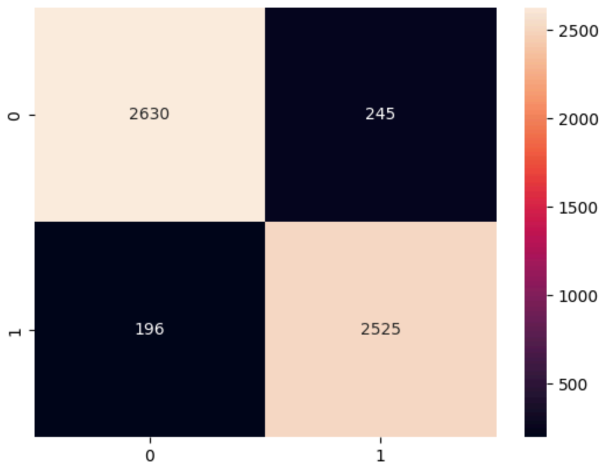
The training and validation accuracy with loss is the graph as follows:





The training accuracy is increasing while loss is decreasing. Also similar case in the validation set but with less variation.

The testing performance is as follows:



The accuracy = 92.12%, the precision = 91.16%, the recall = 92.80%, the F1 score = 91.97%

6. XLNet

XLNet was proposed by Google in June of 2019. XLNet utilizes the best things from autoencoding (AE) and autoregressive (AR) language modeling during its model pretraining, while avoiding the weaknesses of both AE and AR. Other than that, XLNet utilizes four new techniques of (i) permutation language modeling, (ii) two-stream self-attention, (iii) relative positional encoding scheme and (iv) segment recurrence mechanism to boost its performance.

First, let's talk about the pros and cons of AE. The good thing about AE is that it can perform denoising. The first model that uses AE for denoising is BERT. Since BERT has been discussed during lecture, in a nutshell, during the model pre-training, denoising using AE can help to predict the randomly masked token based on the context before and after the masked words (i.e. bidirectional context dependency). However, AE has 2 drawbacks. The first drawback is that it assumes independence. Notice that the ≈ symbol in the following log-likelihood equation:

$$\max_{\theta} \quad \log p_\theta(\bar{\mathbf{x}} \mid \hat{\mathbf{x}}) \approx \sum_{t=1}^{T} m_t \log p_\theta(x_t \mid \hat{\mathbf{x}}) = \sum_{t=1}^{T} m_t \log \frac{\exp\left(H_\theta(\hat{\mathbf{x}})_t^\top e(x_t)\right)}{\sum_{x'} \exp\left(H_\theta(\hat{\mathbf{x}})_t^\top e(x')\right)}$$

where masked tokens is $\overline{x}$, not masked token is $\widehat{x}$, $m_t = 1$ indicates x_t is masked, and H_θ is a Transformer that maps a length-T text sequence **x** into a sequence of hidden vectors $H_\theta(x) = [H_\theta(x)_1, H_\theta(x)_2, \cdots, H_\theta(x)_T]$.

This shows that all masked tokens $\overline{x}$ are separately predicted as the joining conditional probability $p(\overline{x}|\widehat{x})$ is assumed to be independent to each other. (Note: This can be solved by AR language modeling as it does not have such independent assumption.)

The second problem is the input noise problem. Note that BERT contains [MASK] that should never appear in the finetuning task, which leads to a pretrain-finetune discrepancy. (Note: this can be solved by AR language modeling as it does not depend on any masked input words and would not have such an issue.)

Second, let's talk about the pros and cons of AR language modeling. The good thing about AR language modeling is that its pretraining tasks are done by maximizing the likelihood under the forward autoregressive factorization using the following equation:

$$\max_{\theta} \quad \log p_\theta(\mathbf{x}) = \sum_{t=1}^{T} \log p_\theta(x_t \mid \mathbf{x}_{<t}) = \sum_{t=1}^{T} \log \frac{\exp\left(h_\theta(\mathbf{x}_{1:t-1})^\top e(x_t)\right)}{\sum_{x'} \exp\left(h_\theta(\mathbf{x}_{1:t-1})^\top e(x')\right)}$$

where $h_\theta(\mathbf{x}_{1:t-1})$ is a context representation produced by neural models and e(x) denotes the embedding of x given a text sequence $\mathbf{x} = [x_1, \ldots, x_T]$.

In other words, given a word sequence $[x_1, x_2, \ldots, x_t]$, in the pretraining, it uses $[x_1]$to predict x_2, $[x_1, x_2]$ to predict x_3, and all the way to $[x_1, x_2, \ldots, x_{t-1}]$ to predict x_t. The only drawback is that it only uses the context on the left side of the word token, which cannot utilizes the context on the right side of the word token. (Note: This is solved by the bidirectional context dependency in AE.)

Now, let's dive into the detail explanation of the three new techniques:

(i) **permutation language modeling**

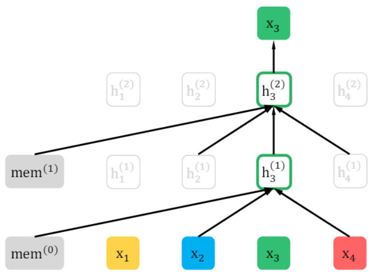
Permutation language modeling is aimed to collect bidirectional context information while preventing the independence assumption and the pretrain-finetune discrepancy problems.

This technique is borrowed from orderless NADE. The key idea is the use of permutation. If we have a sequence **x** that has a length of T, then there would be T! number of different orderings to perform AR factorization, which allows bidirectional context capturing. This technique can be generalized into the following equation:

$$\max_{\theta} \quad \mathbb{E}_{\mathbf{z}\sim\mathcal{Z}_T}\left[\sum_{t=1}^{T}\log p_\theta(x_{z_t} \mid \mathbf{x}_{\mathbf{z}_{<t}})\right]$$

where Z_T is the set of all possible permutations of the length-T index sequence [1, 2, . . . , T], z_t is the t-th element of a permutation $\mathbf{z}\in Z_T$ and $\mathbf{z}_{<t}$ is the first t-1 elements of a permutation $\mathbf{z}\in Z_T$.

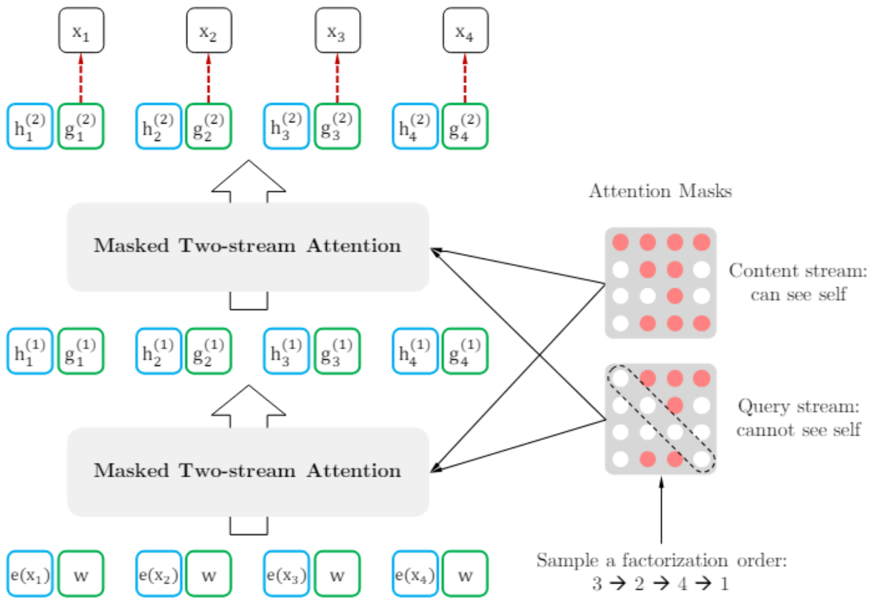
Here is the detailed explanation on how permutation language modeling can capture the bidirectional context without using any masked tokens. For a given text sequence **x**, it would sample a factorization order **z** at one time and decompose the likelihood $p_\theta(\mathbf{x})$ according to factorization orders during training. Therefore, x_t has seen every possible element $x_i \neq x_t$ in the sequence. Below is an example to demonstrate the working mechanism:



Factorization order: 2 → 4 → 3 → 1

Let say we have a sequence $[x_1, x_2, x_3, x_4]$ initially. After we do a round of permutation, we get a new sequence $[x_2, x_4, x_3, x_1]$. Assume that we randomly chose x_3 as the prediction target, you would notice that x_3 does not only able to attend to the context on the left x_2, but also the context on the right x_4, as x_3 can attend to $[x_2, x_4, x_3]$. Similarly, x_2 can attend to $[x_2]$, x_4 can attend to $[x_2, x_4]$ and x_1 can attend to $[x_2, x_4, x_3, x_1]$. Note that mem$^{(0)}$ is the architecture of Transformer-XL, and permutation would not change the actual attention matrix, but just the tokens that would be attend to after the permutation.

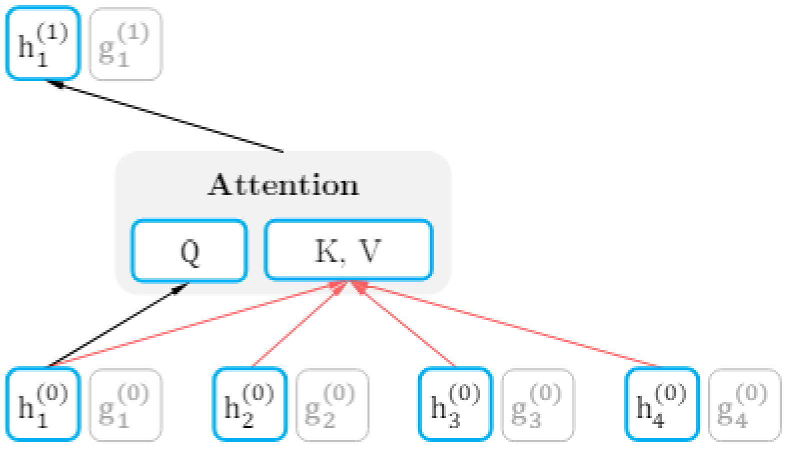
(ii) **<u>two-stream self-attention</u>**


Masked Two-stream Attention
Masked Two-stream Attention
Attention Masks
Content stream: can see self
Query stream: cannot see self
Sample a factorization order: 3 → 2 → 4 → 1

(c)

Two-stream self-attention is used to predict the location of a word token and the bidirectional context relationship. There are two streams, which are content stream and query stream respectively.

<u>Content stream:</u>



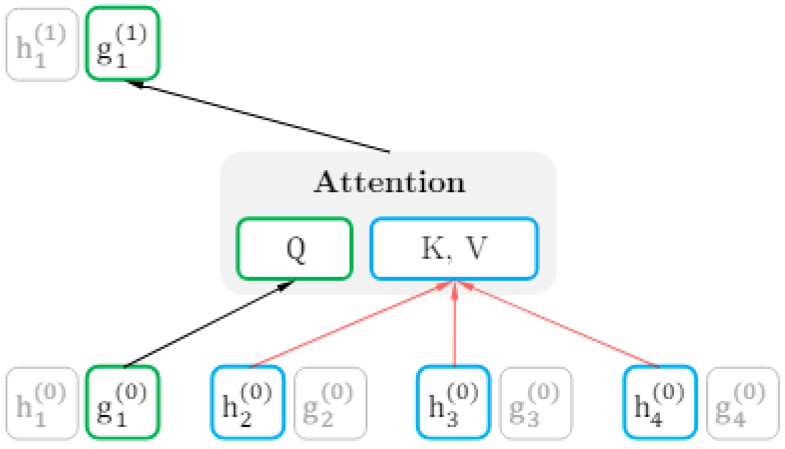
(a)

$$h_{z_t}^{(m)} \leftarrow \text{Attention}(\text{Q} = h_{z_t}^{(m-1)}, \text{KV} = \mathbf{h}_{\mathbf{z}_{<t}}^{(m-1)}; \theta), \quad \text{(content stream: use both } z_t \text{ and } x_{z_t}\text{)}$$

Content stream is used to learn the bidirectional context. Basically, content stream is identical to the standard self-attention. The content representation h is similar to a standard hidden state in transformer, which contains context and the context information.

Let us take the above figure as an example. In the first layer, h_1 is Q, and h_1 to h_4 as both K and V. h_1 would perform attention with x_1 to x_4. This gives the attention weight and multiply with V to get the representation of h_1 in the second layer.

Query stream:



(b)

$$g_{z_t}^{(m)} \leftarrow \text{Attention}(\text{Q} = g_{z_t}^{(m-1)}, \text{KV} = \mathbf{h}_{\mathbf{z}_{<t}}^{(m-1)}; \theta), \quad \text{(query stream: use } z_t \text{ but cannot see } x_{z_t})$$

Query stream is used to replace the [MASK] token in BERT. The query representation g contains context information and position, but not the content.

Let us take the figure above as an example. In the first layer, g_1 is Q, and h_2 to h_4 is K and V. After using the attention equation above, it would make g_1 attend to h_2 to h_4.

(iii) **relative positional encoding scheme** (from Transformer-XL)

Relative positional encoding scheme is a scheme to tackle the weakness of absolute positional encoding that does not take the relationship between a word position relative to another word position into account. Under relative positional encoding, apart from randomly initializing the word embeddings in each position, we also create the pair-wise embedding matrix of size [T, 2*T - 1]. Each row's index represents the word that we are concerned about, and each column's index represents the relative positional distance from the word we are concerned about to other words (both before and after the concerned word).

Here is a simple example to demonstrate this technique. Let us consider a simple sentence "I like cats and dogs".

Now, after relative positional encoding, we get a 2-dimensional vector encoding for each word as follows:

I	[0.3, 0.7]
like	[0.2, 0.9]
cats	[0.4, 0.5]

and	[0.1, 0.3]
dogs	[0.8, 0.4]

After we compute a relative positional encoding based on their distance from each other in the sequence, we would get relative positional distance in 2-dimension. The smaller the distance, the higher their relationship.

Relative position of “like” relative to “I”	[0.1, 0.2]
Relative position of “cats” relative to “I”	[0.2, 0.4]
Relative position of “and” relative to “I”	[0.3, 0.7]
Relative position of “dogs” relative to “I”	[0.4, 0.9]
Relative position of “cats” relative to “like”	[0.1, 0.1]
Relative position of “and” relative to “like”	[0.3, 0.2]
Relative position of “dogs” relative to “like”	[0.1, 0.1]
Relative position of “and” relative to “cats”	[0.3, 0.4]
Relative position of “dogs” relative to “cats”	[0.3, 0.6]
Relative position of “dogs” relative to “and”	[0.3, 0.5]

As you can see, since “like” and “cats”, “like” and “dogs” have much closer relationships, so their distance vector is smaller than other pairwise vector distances. This is how relative positional encoding works.

(iv) **segment recurrence mechanism** (from Transformer-XL)

Segment recurrence mechanism is used to allow each sentence segment from the past can be used with a new sentence segment, so as to avoid the same fixed-length context fragmentation embodiment.

The equation for updating the attention with memory for the next segment **x** is:

$$h_{z_t}^{(m)} \leftarrow \text{Attention}(\mathrm{Q} = h_{z_t}^{(m-1)}, \mathrm{KV} = \left[\tilde{\mathbf{h}}^{(m-1)}, \mathbf{h}_{\mathbf{z}_{\leq t}}^{(m-1)}\right]; \theta)$$

where [., .] denotes concatenation along the sequence dimension.

Let say we have two segments extracted from a long sequence **s**, which are

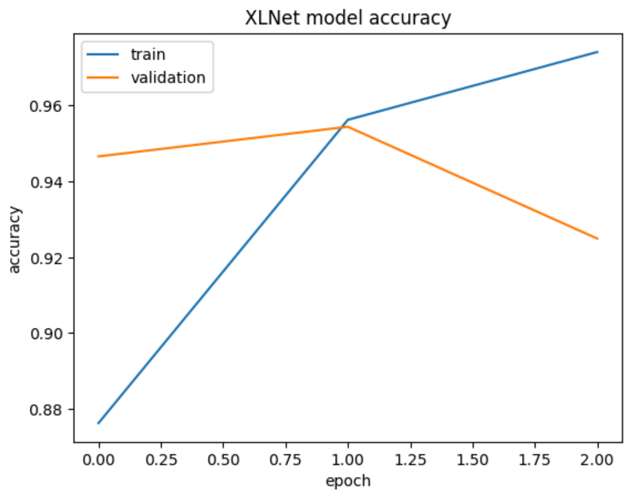
$$\tilde{\mathbf{x}} = \mathbf{s}_{1:T} \text{ and } \mathbf{x} = \mathbf{s}_{T+1:2T}$$

Let say $\tilde{\mathbf{z}}$ and z be the permutations of [1 … T] and [T + 1 … 2T] respectively. Then, based on the permutation $\tilde{\mathbf{z}}$, we process the first segment and cache the obtained content representations $\tilde{\mathbf{h}}^{(m)}$ for each layer m. Hence, the next segment **x**, the attention update with the memory can be written as the above equation. Note that positional encodings depends on actual positions in original sequence of words only. This means the above attention updating equation is independent of $\tilde{\mathbf{z}}$ after the representations of $\tilde{\mathbf{h}}^{(m)}$ is obtained. This eventually means without knowing the permutation order of the previous segment, we can cache and reuse the memory. It is expected that the model can learn to use the memory over all permutation orders of the last segment of sequence, and the query stream can be computed identically.

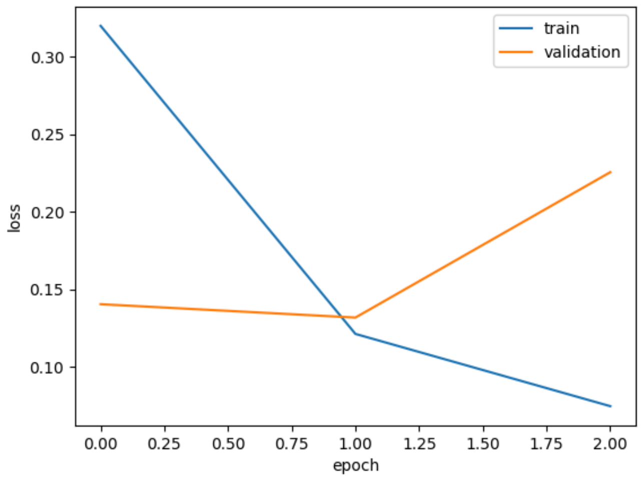
At last, below is the parameter settings when using XLNet:

- XLNet tokenizer: xlnet-base-cased
- XLNet model: TFXLNet (xlnet-base-cased)
- Epochs: 3
- Batch size: 64
- Maximum input sequence length: 128

The training and validation accuracy with loss is the graph as follows:

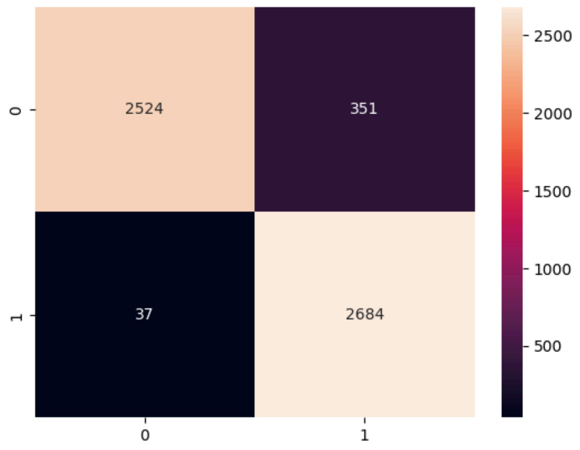


XLNet model Loss



The training accuracy is increasing while loss is decreasing. Also similar case in the validation set but in the third epoch has a slight variation.

The testing performance is as follows:



The accuracy = 93.07%, the precision = 88.43%, the recall = 98.64%, the F1 score = 93.26%

Experiments and results

Since the performance and the visualization details are stated in the previous part, so in this part will mainly focus on the analysis by the result and with different model advantages and disadvantages.

Our group has performed three experiments with our data so that we can have a more comprehensive understanding of the dataset, how the MLP model will be affected by the length of the one-hot encoding, and we want to compare different models' performance to have a better understanding of different models with its strength and weakness. Finally, based on the comparison, we want to draw a conclusion on which model is better for mental health classification in the medical aspect.

1. First Experiment on Naive Bayes

Description:

In the traditional aspect of evaluating the distribution of the word in training data and testing data by the frequency count and histogram. Moreover, we can use loops to know how similar the words that appear in training data will likely appear in testing data. However, it is extremely time-consuming to loop through it since the number of words in the corpus is easily to be more than a hundred thousand. Also, in the previous course trial, the Naive Bayes performs extremely fast. Therefore, our group would like to utilize the drawbacks of the Naive Bayes classifier without Laplace Correction which will make the probability become 0 if the words in the testing data do not appear in the training data.

We are expecting the result can be extracted from the Naive Bayes classifier with or without Laplace Correction's accuracy.

Experiment:
Independent variable: the Laplace Correction value (ie. alpha = 0 means don't perform and alpha = 1 means adding 1 count to each word to obtain no zero probability which is Laplace Correction)

Dependent Variable: The accuracy of both Naive Bayes classifier which has or hasn't Laplace Correction.

Fixed Condition: The training data and testing splitting in both cases are the same and CountVectorizer() performs the counting of each word for preparing the probability in the Naive Bayes Classifier.

Process: We utilize MultinomialNB() with one set alpha to 0 and another one set alpha to 1 to perform the testing. Then we fit the same training data into both of them and use two models to predict on same testing data.

Result:
As in the previous section stated, the accuracy without Laplace Correction is 0.8518584703359543 and the accuracy with Laplace Correction is 0.8429235167977126. The difference is less than 1% in both accuracy. And the time usage for both trials is 1.5413191318511963 seconds which is obviously faster than looping the corpus in O(n^2) time where n is easily more than a hundred thousand.

Also, from the first part of the report, we know that the most frequent 20 of each label (we have two labels so in total of 40). Then the probability of the most frequent word appearing in the testing data which is also in the most frequent of the training data is 39/40 = 97.5%.

According to the research “Heart disease prediction using Naive Bayes algorithm and Laplace Smoothing technique” draws the conclusion that if a number of testing data do not appear in the training, after applying the Laplace Correction will have a significant increase in the accuracy. Therefore, in contrast, our accuracy is extremely similar and based on the alignment of the most frequent word result. We can conclude that utilizing the Naive Bayes with or without Laplace Correction, if the accuracy is similar which implies the data in both training and testing is also similar, if the accuracy has a significant improvement after applying Laplace Correction which implies there are more variations in the training and testing data.

As a result, we can perform Naive Bayes with or without Laplace Correction to quickly estimate is there exists a large variation or not in both training and testing data.

2. The impact of embedding sequence length

Description:

As prior knowledge, with a shorter length of the one-hot encoding will reduce the information and the dimensionality which may lower the accuracy but have a less usage of the RAM.

However, we are interested in examining how the model will be affected or limited by a shorter encoding. We are also curious that will a more complex model can improve the performance while the information is less.

Experiment:

As in the previous session mentioned, we have 3 different models in the “One-Hot model with MLP”.

Independent Variable: Length of the encoding or model architecture

Dependent Variable: The accuracy, precision, recall and F1 score for the three models

Process: In the first trial, we fixed the length of the one-hot encoding to be 100. Then we have one less complex MLP and one more complex MLP model for training and comparison. Afterwards, we fixed the MLP model to be the one with less complexity and performed the comparison with one using an encoding length is 100 and another one with an encoding length is 1000.

Result:
For both less complex and more complex models with encoding length = 100, the performance is accuracy = 86.88%, the precision = 89.19%, the recall = 83.09%, F1 score =

86.04% and accuracy = 85.61%, the precision = 88.23%, the recall = 81.26%, the F1 score = 84.60% respectively. We can see that even though we increase the capability of the model, we still cannot improve any metrics. Also, in the complex model, the validation accuracy and loss have slightly decreased and increased respectively during training which means there may be over overfitting problem.

For both are less complex MLP but with different lengths of the encoding 100 and 1000. The performance is accuracy = 86.88%, the precision = 89.19%, the recall = 83.09%, F1 score = 86.04% and The accuracy = 91.07%, the precision = 89.25%, the recall = 92.80%, the F1 score = 90.99% respectively. We can see that 3 metrics out of 4 have a great improvement even though we are using the less complex model. It appears that a longer encoding allows the model to capture more nuanced patterns and information in the data, leading to better performance.

As a result, based on the two comparisons, we can realize that the length of the encoding will be a bottleneck of the performance. It can be concluded that the length of the encoding is a critical factor in determining the performance of the MLP model.

3. Model Comparison and Best Model for Mental Health Classification

Description:

Since there are so many models that can perform the same task, as our group already used 6 of them, they are the Naive Bayes Classifier, One-Hot MLP model, Index RNN model, CNN model, distilBERT model and XLNet. Therefore, after the implementation and testing, our group would like to compare all models' benefits and drawbacks and compare their performance with the trade-off of the drawbacks which can help us understand different models better.

Experiment:

Independent Variable: 6 different models

Dependent Variable: The computational power (time and memory usage), model performance (accuracy, precision, recall and F1 score)

Process: We built all models as the last part stated with their performance. So we can directly jump to the result part.

Result:

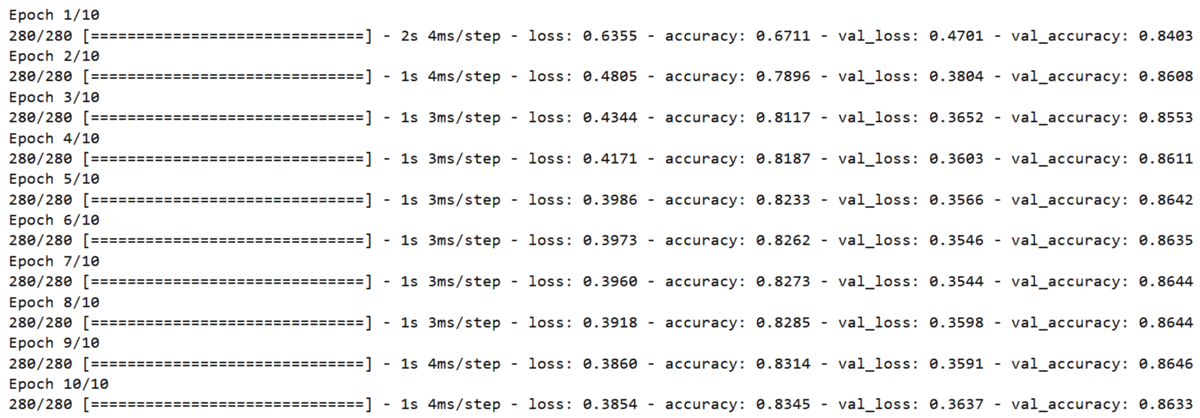
1. Naive Bayes Classifier
    a) Advantages

In the computational aspect, it is extremely quick since for both with and without Laplace Correction models only takes 1.5413191318511963 seconds.

b) Disadvantages
As mentioned before, the “Naive” approach assumes each word will have an equal contribution and independent probability. Therefore, This is usually not the case in the NLP task since in the language, it is obvious that words are not independent and indifferent sentences, some words will have a higher contribution to the emotional expression. Also, the meaning of the words will be changed based on the words near them or the words will have different implicatures such as “bad” is negative but “not bad” is positive which also dramatically affects the meaning.

c) Performance
The accuracy 84.29%, precision 76.12%, recall 98.64%, F1 score 85.93%
Since Naive Bayes with Laplace Correction approach is usually used, so we will only consider this performance.

2. One-Hot MLP model (We will only consider the length = 100 encoding)
a) Advantages
The computation time is low, as the image below



```
Epoch 1/10
280/280 [==============================] - 2s 4ms/step - loss: 0.6355 - accuracy: 0.6711 - val_loss: 0.4701 - val_accuracy: 0.8403
Epoch 2/10
280/280 [==============================] - 1s 4ms/step - loss: 0.4805 - accuracy: 0.7896 - val_loss: 0.3804 - val_accuracy: 0.8608
Epoch 3/10
280/280 [==============================] - 1s 3ms/step - loss: 0.4344 - accuracy: 0.8117 - val_loss: 0.3652 - val_accuracy: 0.8553
Epoch 4/10
280/280 [==============================] - 1s 3ms/step - loss: 0.4171 - accuracy: 0.8187 - val_loss: 0.3603 - val_accuracy: 0.8611
Epoch 5/10
280/280 [==============================] - 1s 3ms/step - loss: 0.3986 - accuracy: 0.8233 - val_loss: 0.3566 - val_accuracy: 0.8642
Epoch 6/10
280/280 [==============================] - 1s 3ms/step - loss: 0.3973 - accuracy: 0.8262 - val_loss: 0.3546 - val_accuracy: 0.8635
Epoch 7/10
280/280 [==============================] - 1s 3ms/step - loss: 0.3960 - accuracy: 0.8273 - val_loss: 0.3544 - val_accuracy: 0.8644
Epoch 8/10
280/280 [==============================] - 1s 3ms/step - loss: 0.3918 - accuracy: 0.8285 - val_loss: 0.3598 - val_accuracy: 0.8644
Epoch 9/10
280/280 [==============================] - 1s 4ms/step - loss: 0.3860 - accuracy: 0.8314 - val_loss: 0.3591 - val_accuracy: 0.8646
Epoch 10/10
280/280 [==============================] - 1s 4ms/step - loss: 0.3854 - accuracy: 0.8345 - val_loss: 0.3637 - val_accuracy: 0.8633
```

For each epoch, each only takes 1 to 2 seconds since the Dense layer just takes simple mathematical operation and the back-propagate is more straight forward than other models.

b) Disadvantages
Since the Dense layer is used in the MLP, based on the property of the Dense layer, all the previous layer’s input will be fully connected to the layer’s units. Therefore, it implies that it treated every input as independent which is the similar drawbacks stated in the Naive Bayes classifier.

The second disadvantage is one-hot encoding requires lots of RAM usage, we are not able to use the entire corpus to perform the encoding for each sentence.

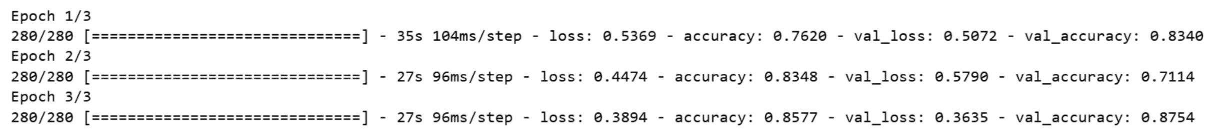
c) Performance
   The accuracy = 86.88%, the precision = 89.19%, the recall = 83.09%, the F1 score = 86.04%

3. Index RNN model
   a) Advantages
      Firstly, we have an embedding layer before the LSTM layer, so each word can be represented better instead of using the one-hot or only their index in the corpus since the words with relative meanings will have closer vectors in representation.

      Secondly, we are utilizing the LSTM layer which can capture the long and short term memory, so each word in the sentences can be related which can be better for the semantic understanding.

      Thirdly, LSTM can prevent the vanishing gradient problem because the LSTM contains a gate mechanism which can control the gradient and the flow of information.

   b) Disadvantages
      The model requires more computational power since we will pass the data in sequential and the backpropagation is more complicated in the LSTM units. We can see the training time is around 27 to 32 seconds for each epoch.



```
Epoch 1/3
280/280 [==============================] - 35s 104ms/step - loss: 0.5369 - accuracy: 0.7620 - val_loss: 0.5072 - val_accuracy: 0.8340
Epoch 2/3
280/280 [==============================] - 27s 96ms/step - loss: 0.4474 - accuracy: 0.8348 - val_loss: 0.5790 - val_accuracy: 0.7114
Epoch 3/3
280/280 [==============================] - 27s 96ms/step - loss: 0.3894 - accuracy: 0.8577 - val_loss: 0.3635 - val_accuracy: 0.8754
```

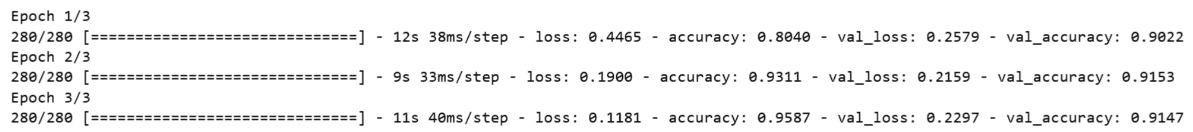
      Secondly, the interpretation is limited. Since the LSTM is like a “black box” and it is hard to understand the interpretation in it.

   c) Performance
      The accuracy = 88.46%, the precision = 90.04%, the recall = 85.74%, the F1 score = 87.84%

4. CNN Model
   a) Advantages
      Firstly, same as the index RNN model, the CNN model starts with an embedding layer which can have a better representation of each word.

      Secondly, due to the property of spatial capture of convolution. CNN can have a high accuracy when the testing data is similar to the training data since it can easily capture the pattern of the sentences. Also, this property can help the CNN model robust to noise even if the text data is distorted or corrupted.

Thirdly, the computational power is relatively small. For each epoch takes around 10 seconds.



```
Epoch 1/3
280/280 [==============================] - 12s 38ms/step - loss: 0.4465 - accuracy: 0.8040 - val_loss: 0.2579 - val_accuracy: 0.9022
Epoch 2/3
280/280 [==============================] - 9s 33ms/step - loss: 0.1900 - accuracy: 0.9311 - val_loss: 0.2159 - val_accuracy: 0.9153
Epoch 3/3
280/280 [==============================] - 11s 40ms/step - loss: 0.1181 - accuracy: 0.9587 - val_loss: 0.2297 - val_accuracy: 0.9147
```

b) Disadvantages
The CNN model will not perform well in adversarial attacks and not general enough. As the advantages stated, the CNN is capturing the pattern of the sentences which means that it does not learn any semantic meaning from the sentences. Once the usage of the model to a new pattern of sentences with the same meaning as in the training data. It may also not be able to give a good performance.

c) Performance
The accuracy = 91.53%, the precision = 92.87%, the recall = 89.45%, the F1 score = 91.13%

5. DistilBERT

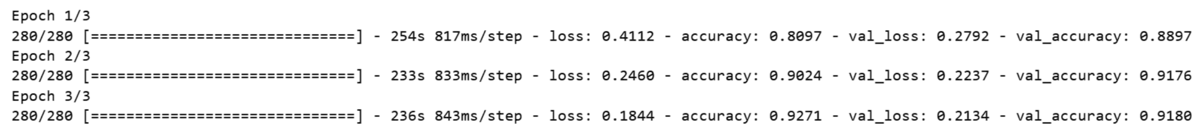
a) Comparison with original BERT
According to the architecture simplify, the research paper stated that DistilBERT is 60% faster than the BERT model. Also, DistilBERT has 44M parameters fewer than the BERT model. Even though the DistilBERT has much improvement in space and time, it can still maintain 97% performance of the original BERT model. Therefore, we would like to try it out.

b) Advantages
Firstly, DistilBERT can have a contextual understanding of the sentences since there is the self-attention mechanism inside each encoder which can make the model understand the relationship of words.

Secondly, DistilBERT can have a bidirectional understanding of the context since it uses the mask for prediction during training which can enhance the ability of the understanding.

Thirdly, as the BERT model, there is a pre-trained model for distilBERT which can significantly reduce the converge time to achieve a good performance.

c) Disadvantages
The computational power of the distilBERT is also relatively high. For each epoch training, it requires around 4 minutes.



```
Epoch 1/3
280/280 [==============================] - 254s 817ms/step - loss: 0.4112 - accuracy: 0.8097 - val_loss: 0.2792 - val_accuracy: 0.8897
Epoch 2/3
280/280 [==============================] - 233s 833ms/step - loss: 0.2460 - accuracy: 0.9024 - val_loss: 0.2237 - val_accuracy: 0.9176
Epoch 3/3
280/280 [==============================] - 236s 843ms/step - loss: 0.1844 - accuracy: 0.9271 - val_loss: 0.2134 - val_accuracy: 0.9180
```

d) Performance
   The accuracy = 92.12%, the precision = 91.16%, the recall = 92.80%, the F1 score = 91.97%

6. XLNet Model
   a) Advantages
      Since the XLNet is somehow like the combination of BERT and Transform-XL, so I would like to compare it with BERT and Transform-X more to see the enhancement.

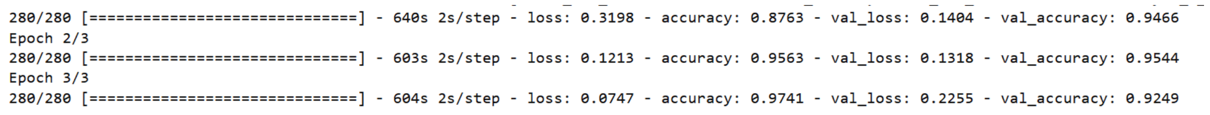
      First, the XLNet construct by permutation language modeling which can take advantage from both AR and AE (the mask mechanism in BERT). Therefore, it is better than Transform-XL because of both directional understanding and it is also better than BERT since it can overcome the limitation that neglects the dependency of the mask position.

      Secondly, XLNet is using the relative positional encoding scheme but the BERT uses the absolute positional encoding. Therefore, the relationship of the words in the sentences in XLNet can have a better representation of the positional information.

      Thirdly, the two-stream self-attention mechanism in XLNet is better than the self-attention mechanism in BERT since for the prediction it uses the query which can hide the content of the current position and there is the content stream that can help for the other prediction.

      Fourthly, there is a pre-trained model for XLNet which can significantly reduce the converge time to achieve a good performance.

   b) Disadvantages
      The drawbacks of the model also require a high computational power. It takes around 10 minutes for each epoch.



```
280/280 [==============================] - 640s 2s/step - loss: 0.3198 - accuracy: 0.8763 - val_loss: 0.1404 - val_accuracy: 0.9466
Epoch 2/3
280/280 [==============================] - 603s 2s/step - loss: 0.1213 - accuracy: 0.9563 - val_loss: 0.1318 - val_accuracy: 0.9544
Epoch 3/3
280/280 [==============================] - 604s 2s/step - loss: 0.0747 - accuracy: 0.9741 - val_loss: 0.2255 - val_accuracy: 0.9249
```

   c) Performance
      The accuracy = 93.07%, the precision = 88.43%, the recall = 98.64%, the F1 score = 93.26%

After drawing out all the advantages and disadvantages of each model with its performance, we can now consider which model is better for medical judgment.

Firstly, from the analysis of the data, we know that the data in both training and testing data have similar patterns and words. However, in the real world, the human language is ambiguous which means it is hard to have similar patterns or utilize the same words in their speaking. Therefore, we will eliminate those models that are not able to capture the contextual meaning which are Naive Bayes, One-Hot MLP model, and CNN model.

The remaining models are Index RNN, distilBERT and XLNet. Since the accuracy for the three models is similar but the recall metric of the data is much more important than the other metrics since a high recall means that there are less people who have mental health problems but are classified as having no problems. If lower precision means more people without problems but classified as having mental problems.

From the medical perspective, we only want to focus on the recall since we want to find out as many people as we can if they have problems and for the people who are without problems but classified as having problems, we can still further examine and remove them from the list.

According to this aspect, we can see that the recall of XLNet is 98.64%, distilBERT is 92.80% and the Index RNN model is 85.74%. Therefore, even though the computational time is the highest but we think the trade-off is fair with its recall rate.

As a result, the XLNet is the best model that we examine to solve this mental health classification problem.

Division of labour in teamwork

Report Part:

Description of the dataset — CHAN, Chun Hin
Description of the preprocessing — LAW, Hui Nok
Description of the machine learning task(s) performed on the dataset — CHAN, Chun Hin
Description of the hardware and software computing environment — CHAN, Chun Hin
Machine learning method with its performance (Naive Bayes, One-Hot MLP model, Index RNN model, CNN model, distilBERT) — LAW, Hui Nok
Machine learning method with its performance (XLNet) — CHAN, Chun Hin
Experiments and results — LAW, Hui Nok
Division of labour in teamwork — LAW, Hui Nok

Coding:

Colab Computer Seeting — CHAN, Chun Hin
Loading Data To Colab — LAW, Hui Nok
Utility — CHAN, Chun Hin & LAW, Hui Nok
Set Seed — CHAN, Chun Hin
Function For Data Preprocessing with Demo — CHAN, Chun Hin

Data Reading and Splitting — LAW, Hui Nok
Data Distribution — CHAN, Chun Hin
Data Processing for Naive Bayes — LAW, Hui Nok
Naive Bayes — LAW, Hui Nok
Data Processing for the below model — LAW, Hui Nok
One Hot MLP Model — LAW, Hui Nok
Index RNN Model — LAW, Hui Nok
CNN Model — LAW, Hui Nok
DistilBert Model — CHAN, Chun Hin
XLNet Model — CHAN, Chun Hin
Save the model and back to drive — LAW, Hui Nok

Overall distribution: CHAN, Chun Hin = 50%, LAW, Hui Nok = 50%

Material that accessed

Cherian, V., & Bindu, M. S. (2017). Heart disease prediction using Naive Bayes algorithm and Laplace Smoothing technique. *Int. J. Comput. Sci. Trends Technol*, *5*(2), 68-73.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., Salakhutdinov, R. (2019). Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. *arXiv:1901.02860*

Efimov, V. (2023, October 9). Large language models: DistilBERT — smaller, faster, cheaper and lighter. *Medium*. https://towardsdatascience.com/distilbert-11c8810d29fc

Kianyew, N. (2024, January 7). What is Relative Positional Encoding. *Medium*. https://medium.com/@ngiengkianyew/what-is-relative-positional-encoding-7e2fbaa3b510

Pei, C. T. (2021, December 13). DistilBERT — 更小更快的BERT模型 - NLP-Trend-and-Review - Medium. *Medium*. https://medium.com/nlp-tsupei/distilbert-%E6%9B%B4%E5%B0%8F%E6%9B%B4%E5%BF%AB%E7%9A%84bert%E6%A8%A1%E5%9E%8B-eec345d17230

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

Tsoi, Y. C., Xiao, H. R., Naive Bayes Classifier. https://course.cse.ust.hk/comp2211/notes/3-naive-bayes-full.pdf

Xiao, M. (2020, May 5). Understanding Language using XLNet with autoregressive pre-training. *Medium*. https://medium.com/@zxiao2015/understanding-language-using-xlnet-with-autoregressive-pre-training-9c86e5bea443

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., & Le, Q. V. (2019). XLNet: Generalized Autoregressive Pretraining for Language Understanding. *Advances in neural information processing systems*, *32*.