

Q1 Dataset and Exploratory Data Analysis

1.1 Business and Data Understanding

1.1.1 Problem Statement

A rise in suicide rates worldwide has become a major public health concern, causing the development of new methods for early diagnosis and prevention. Traditional methods of detecting individuals at risk of suicide frequently rely on self-reporting and professional assessments, which can be limited due to judgment, a lack of access to mental health care, and a delay in action (Sedano-Capdevila, et al., 2023). In response, the rise of digital technologies offers an effective strategy for preventive suicide prevention. This research seeks to create a reliable suicide text detection system that uses natural language processing (NLP) and machine learning approaches to detect distress signs in textual communication. By analyzing social media posts, messages, and other text-based interactions, this system seeks to provide timely alerts and facilitate interventions before a crisis occurs. The effective implementation of such a system might greatly improve existing mental health support frameworks, ultimately contributing to a decrease in suicide rates and better public health outcomes (O'Dea, et al., 2015).

1.1.2 Literature Review

(Chiroma, Liu, & Cocca) has conducted a study to contribute on the research on suicide detection in social media by conducting experiment to measure the performance of several popular machine learning approaches. This experiment has performed various pre-processing step such as reducing the noise data to improve the quality of data and classifier performance. Besides, the author has proceeded the further data transformation such as URL removal, stop words removal, case conversion, stemming etc. There are two type of dataset is covered in the experiment which is binary and multi -class classification. the binary dataset contains Suicide and Flippant as the target variable whereas the multi class dataset contain Suicide, Flippant and Non-suicide classes. The author has used both of these datasets to measure the performance of four machine learning model – Decision Tree (DT), Naïve Bayes(NB) , Random Forest (RF) and Support Vector Machine (SVM). In the experiment, a 10 -fold cross validation is being conducted due to minimize the influence of training set variability on the result . The researcher has measured the performance using F-measure and accuracy. The reason of using F-measure as measurement is because it balances the influence between precision and recall. At the end of the experiment has shown a result that decision tree model has the best F-measure which

0.879 and accuracy of 0.790, The decision tree model has achieved the highest performance in both binary and multi class dataset (Chiroma, Liu, & Cocca).

(Jain , Srinivas, & Vichare, 2022) proposed a few machine learning models such as Naïve Bayes, Support Vector Machine (SVM), Random Forest and Logistics Regression to predict the suicidal ideation on Reddit. The researcher conducted a thorough comparative analysis of these algorithms to understand their strengths and weaknesses. The researchers explained that Logistic Regression is a statistical technique used for binary classification tasks, providing a probability value that can be linked to two distinct classes using the logistic function. It models the relationship between the categorical dependent variable and one or more independent variables by estimating the probability of occurrence of an event based on the input features. A series of data pre-processing such as lower casing, punctuation removal, tokenization, stop words removal, and stemming are conducted before applying the algorithms. Performance evaluation involved four key metrics, such as accuracy, precision, recall, and f1-score. The result indicates that Logistics Regression has achieved the highest classification accuracy at 79% along with random forest, followed by Support Vector Machine, Naïve Bayes.

(Lee & Pak, 2022) has conducted a study with the aim to screen Korean adults who are at risk of experiencing suicidal thoughts, as well as those who may be planning or attempting suicide. The authors have proposed a few machine learning models such as logistic regression, support vector machine (SVM), random forest (RF), and extreme gradient boosting (XGBoost) to predict the suicidal ideation and behaviours among Korean adults. The researchers explained that XGBoost is a scalable tree-boosting algorithm, and its algorithms are derived from the idea of boosting, which combines the prediction results of the “weak” learners with those of the “strong” learners through cumulative training instances. These will help to minimize the lookup times and decreased the training time of models which in turn increased the performance. All these proposed algorithms are trained and tuned independently. Performance evaluation involved six key metrics, such as curve (AUC), sensitivity, specificity, positive predictive value, negative predictive value, and accuracy. According to the result, XGBoost performs best and achieved highest accuracy at predicting both suicidal ideation (86.3%) and suicidal planning (88.4%).

In order to improve the ability of machine learning to identify suicidal thoughts on social media, (Santoso, 2023), and his colleagues from Bina Nusantara University conducted a study. Given their ability to improve the accuracy of currently used detection techniques, decision trees (DT) and support vector machines (SVM) were selected as algorithms that meet

the needs. By collecting data using sub-sections including "Suicide Watch", "Depression", and "Adolescents", the research team applied a number of pre-processing techniques to improve data quality. The above methods were used to prepare datasets for binary and multi-classification, which also included removing stop words, converting uppercase and lowercase, and stemming. From the report, it can be learned that they adopted a 10-fold cross-validation strategy to ensure consistency while minimizing the variability of training data. Not only that, the result is noteworthy in that the method using SVM outperformed the DT algorithm, with an accuracy of 91.89% for SVM, while DT had an accuracy of 81.74%. This result shows that SVM is more effective than DT in detecting early phenomena of suicidal thoughts, which plays a vital role in intervention work on social media platforms.

Comparison Table between Literature Review

<i>Authors</i>	<i>Description</i>	<i>Machine Learning Model</i>	<i>Evaluation technique</i>	<i>Result /Outcome</i>
<i>(Chiroma, Liu, & Cocea)</i>	This study has use 2 type of dataset which is binary and multi-class dataset to evaluate the machine learning model performance . Aiming to	Decision Tree , Naïve Bayes, Random Forest, and Support Vector Machine	F-measure Accuracy	Decision Tree has reached the highest performance in both binary and multi-class dataset with the accuracy of 0.779 and 0.790.
<i>Jain et al.</i>	This study used NLP and machine learning to examine the depression and suicide ideation by taken posts (dataset) from Reddit.	Naïve Bayes, Support Vector Machine (SVM), Random Forest and Logistics Regression	Precision, Accuracy, F1-score, Recall	Logistics Regression has the highest classification accuracy performance at 79%, followed by random forest.
<i>Lee & Pak</i>	This study has used two set of balanced data taken from Korea Welfare Panel Study to predict the suicide ideation among Korean adults by using random-down sampling method.	Logistic Regression, Support Vector Machine, Random Forest,	curve (AUC), Sensitivity, Specificity, Positive Predictive Value, Negative Predictive	XGBoost performed the best for predicting both suicide ideation and suicidal planning with accuracy at

<i>Sebastian et al.</i>		and Extreme Gradient Boosting	Value, Accuracy	86.3% and 88.4% each.
	This study aim to improve machine learning models for detecting signs of suicidal thought on social media by collecting data from subreddits like “SuicideWatch,” “depression,” and “teenagers”.	Decision Tree (DT), Support Vector Machine (SVM)	Accuracy, Precision, Recall, F1-score	The SVM outperformed the DT, showing a superior ability to detect early signs of suicidal behaviour: SVM Accuracy 91.89%, DT Accuracy: 81.47%

1.1.3 Justification

The dataset we selected contains a large number of statements about Xisha thoughts, all of which often appear in the social media field, so it is very suitable for our research. The real-life Beijing also enhances the practical significance and practicality of our research. Not only that, it is very important for building a reliable machine learning model that the dataset is annotated, rich in content, and provides a wide range of language features. In addition, it is possible to distinguish between suicidal and non-suicidal content, thereby providing a higher possibility to enhance and accurately evaluate the accuracy of the model.

1.1.4 Description of Dataset

The dataset that we are going to use throughout this assignment is Suicide_Detection.csv. This dataset was obtained through Kaggle.com website, and it is used for analysing discussions related to mental health, specifically focusing on suicide and depression. It comprises a total of 233338 rows and 3 columns, and the description of the variables are as shown in table below:

No.	Variable	Description
1	No	Identity number
2	Text	Content extracted from user posts
3	Class	Consists of 2 categories, either suicide or non-suicide

Table 1

Dataset link: <https://www.kaggle.com/datasets/nikhileswarkomati/suicide-watch/data>

1.2 Exploratory Data Analysis and Data Preprocessing

1.2.1 Exploratory Data Analysis

```
classNum = df['class'].value_counts()
print(classNum)

plt.figure(figsize=((20,5)))

plt.subplot(1,2,1)
sns.countplot(df,x='class')

plt.subplot(1,2,2)
plt.pie(classNum,labels = classNum.index,autopct='%.0f%%')

plt.show()
```

Figure 1 : Distribution of Class

The provided code snippet visualizes the distribution of classes in a dataset using a bar plot and a pie chart. First, it uses `value_counts()` to determine the frequency of each class in the DataFrame column `class`, which is then stored in `classNum` and printed. It then uses `plt.figure()` to build a figure with the dimensions (20, 5). The `plt.subplot(1, 2, 1)` line creates a subplot grid with one row and two columns, with the bar plot in the first position, as made by `sns.countplot(df, x='class')`. The `plt.subplot(1, 2, 2)` creates the second subplot for the pie chart, which is plotted using `plt.pie(classNum, labels=classNum.index, autopct='%.0f%%')` and shows the proportion of each class. Finally, `plt.show()` renders the figure, displaying both subplots side by side.

Result

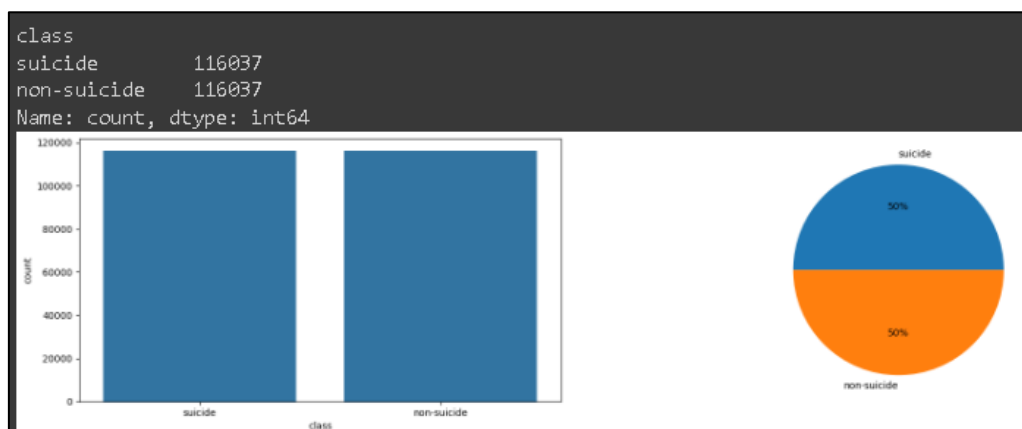


Figure 2 : Distribution of Class Result

The results show an equal distribution of the suicide and non-suicide classes in the dataset, with each class having 116,037 instances. This is visualized by a bar plot and a pie chart, both indicating that each class represents 50% of the total dataset, ensuring balanced class representation.

1.2.1.1 Word cloud

```
from wordcloud import WordCloud
all_text = ' '.join(df['text'])

wordcloud = WordCloud(background_color='white', width=800, height=400).generate(all_text)
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Figure 3 : Python Code For Word Cloud

The word cloud data visualization has done after all the preprocessing step including emoji conversion, punctuation removal, Stopword removal , tokenization and stemming to make sure that the text are accurate and does not contain outlier , to make sure the word cloud accuracy.

This code creates a word cloud from text data in a DataFrame column named 'text'. It converts all text into a single string, generates a word cloud with a white background, and has a resolution of 800x400 pixels. The word cloud is then presented with matplotlib, the figure size set to 10x5 inches, and the axis turned off for a cleaner appearance. It can assist discover key words or phrases that appear frequently in the text, which can be useful for understanding the overall content or sentiment.

Result



Figure 4 : Word Cloud Result

The word cloud created from a suicide text detection dataset depicts the most frequently occurring words in the text data related to suicidal ideation. The key words, wrote in larger sizes, include keywords like "clown", "filler", "fuck", "moai", "thought", "work", "want", "die", and "kill". These terms highlight frequent patterns and vocabulary throughout the dataset.

Analysis

- **Rude and strong languages** : some highlight word like “fuck” and “shit” depict that the presence of strong emotion and distress which commonly can be associated with suicidal though.
- **Suicidal key word** : Words like “die” , “kill”, “suicide”, and “anyway” are a direct indicator of suicidal though .
- **Emotional expression** : Words like "thought", "feel", "hope", and "want" suggest discussions about personal emotions and desires. Word with such show a low risk of suicide .
- **Emoji conversion text** : Terms like "clown", "moai", and "filler" may represent to the text that converted from emoji. Especially these word are used frequently in the dataset .Emoji one of component that most likely use in a comment or text .

1.2.2 Pre-Processing

1.2.2.1 Drop Null Value

```
df.isnull().sum()
```

Figure 5 : Drop Null Value

`df.isnull().sum()` is the method that used to identify the missing values in each column of a data frame. If there is missing values found, the output will show 1; otherwise, it will just show 0.

Result:

```
text      0
class     0
dtype: int64
```

The output indicates there are no missing values found in both “text” and “class” column.

1.2.2.2 Drop Duplication

```
df.duplicated().sum()
```

Figure 6 : Drop Duplication

`df.duplicated().sum()` is to identify and count whether there is any duplicate row in the data frame.

Result: 

The output shows there is no duplicate row, and every row is unique.

1.2.2.3 Lowercase

```
#lowercase all sentences
df['text'] = df['text'].str.lower()
df
```

Figure 7 : Lowercase the text

The line of code is used to convert all the text in “text” column into lowercase. It is a common preprocessing step in text data analysis to ensure consistency and improve the reliability of subsequent text analysis.

Result:

Unnamed: 0		text	class
0	2	ex wife threatening suiciderecently i left my ...	suicide
1	3	am i weird i don't get affected by compliments...	non-suicide
2	4	finally 2020 is almost over... so i can never ...	non-suicide
3	8	i need helpjust help me im crying so hard	suicide
4	9	i'm so losthello, my name is adam (16) and i'v...	suicide
...	
232069	348103	if you don't like rock then your not going to ...	non-suicide
232070	348106	you how you can tell i have so many friends an...	non-suicide
232071	348107	pee probably tastes like salty tea😏🌀!! can som...	non-suicide
232072	348108	the usual stuff you find herei'm not posting t...	suicide
232073	348110	i still haven't beaten the first boss in hollo...	non-suicide
232074 rows × 3 columns			

As we can see from figure above, there is no uppercase text in “text” column.

1.2.2.4 Convert Emoji to Text

```
#convert emoji to text
df['text'] = df['text'].apply(emoji.demojize) #hide and write in csv so that it wont run everytime
df
```

Figure 8 : Python Code For Convert Emoji to Text

“emoji.demojize()” function helps to convert the emojis into text-based descriptions.

Result:

Unnamed: 0		text	class
0	2	ex wife threatening suiciderecently i left my ...	suicide
1	3	am i weird i don't get affected by compliments...	non-suicide
2	4	finally 2020 is almost over... so i can never ...	non-suicide
3	8	i need helpjust help me im crying so hard	suicide
4	9	i'm so losthello, my name is adam (16) and i'v...	suicide
...	
232069	348103	if you don't like rock then your not going to ...	non-suicide
232070	348106	you how you can tell i have so many friends an...	non-suicide
232071	348107	pee probably tastes like salty tea:smirking_fa...	non-suicide
232072	348108	the usual stuff you find herei'm not posting t...	suicide
232073	348110	i still haven't beaten the first boss in hollo...	non-suicide
232074 rows × 3 columns			

Compare to **Figure XX**, the emojis in index 232071 are being replaced with word.

- 😏 → :smirking_face:
- 💧 → :sweat_droplets:

1.2.2.5 Remove Punctuation

```
# Remove Punctuation such as '!"$%&'()*+,-./:;?@[^_`{|}~'
df['text'] = df['text'].str.replace(r'[^\w\s]+', ' ', regex = True)
df
```

Figure 9 : Remove Punctuation

The code above is to remove the punctuation from the “text” column by replacing the punctuation characters with spaces. “.str” allows us to perform string operations; “.replace(r'[^\w\s]+', ' ', regex=True)” is to replace anything that is not number, letter, or whitespace with a space. “regex=True” specifies that pattern is a regular expression.

Result:

	Unnamed: 0	text	class
0	2	ex wife threatening suicidercently i left my ...	suicide
1	3	am i weird i don t get affected by compliments...	non-suicide
2	4	finally 2020 is almost over so i can never he...	non-suicide
3	8	i need helpjust help me im crying so hard	suicide
4	9	i m so losthello my name is adam 16 and i v...	suicide
...
232069	348103	if you don t like rock then your not going to ...	non-suicide
232070	348106	you how you can tell i have so many friends an...	non-suicide
232071	348107	pee probably tastes like salty tea smirking_fa...	non-suicide
232072	348108	the usual stuff you find herei m not posting t...	suicide
232073	348110	i still haven t beaten the first boss in hollo...	non-suicide

232074 rows × 3 columns

As we can see, the punctuations or whitespace are being removed. For example:

- Index 1: don’t → don t
- Index 4: (16) → 16
- Index 232072: herei’m → herei m

1.2.2.6 Stop Word Removal

```
#Stop word removal
stop_words = stopwords.words('english')
df['text'] = df['text'].apply(lambda x : ' '.join([word for word in x.split()
if word not in (stop_words)]))
df
```

Figure 10 : Stop Word Removal

`stop_words = stopwords.words('english')` imports the stop words list from the NLTK library for the English language. Generally, words such as ‘the’, ‘is’, ‘I’, ‘and’ etc. are being removed as they do not carry significant meaning/affect.

The texts are split into individual words, the words that are not fall under stop words list will be filtered out. For those that do not fall under stop words list, they will then join back into a single string, separated by spaces.

Result:

Unnamed: 0		text	class
0	2	ex wife threatening suiciderecently left wife ...	suicide
1	3	weird get affected compliments coming someone ...	non-suicide
2	4	finally 2020 almost never hear 2020 bad year e...	non-suicide
3	8	need helpjust help im crying hard	suicide
4	9	losthello name adam 16 struggling years afraid...	suicide
...
232069	348103	like rock going get anything go https musicas...	non-suicide
232070	348106	tell many friends lonely everything deprived p...	non-suicide
232071	348107	pee probably tastes like salty tea smirking_fa...	non-suicide
232072	348108	usual stuff find herei posting sympathy pity k...	suicide
232073	348110	still beaten first boss hollow knight fought t...	non-suicide

232074 rows × 3 columns

For example:

- Index 0: suiciderecently I left -> suiciderecently left
- Index 1: am i weird → weird
- Index 232073: i still haven t beaten → still beaten

1.2.2.7 Tokenization

```
# tokenization
df['text'] = df['text'].fillna('') [1]
df['text'] = df['text'].apply(lambda x: ' '.join([str(word) for word in x.split()]) if isinstance(x, str) else '') [2]
df_dtype = df['text'].dtype [3]
print(df_dtype)
df['text'] = df['text'].apply(lambda x: nltk.word_tokenize(x)) [4]
df
```

Figure 11 : Tokenization

[1]: fill the missing values with an empty string.

[2]: Tokenize and joins the token back into a string if the element is a string, else, replace with empty string.

[3]: Retrieve data type of “text” column after transformation.

[4]: Tokenize each string using word_tokenize() function, which split the text into words.

Result:

Unnamed: 0		text	class
0	2	[ex, wife, threatening, suiciderrecently, left,...	suicide
1	3	[weird, get, affected, compliments, coming, so...	non-suicide
2	4	[finally, 2020, almost, never, hear, 2020, bad...	non-suicide
3	8	[need, helpjust, help, im, crying, hard]	suicide
4	9	[losthello, name, adam, 16, struggling, years,...	suicide
...
232069	348103	[like, rock, going, get, anything, go, https, ...	non-suicide
232070	348106	[tell, many, friends, lonely, everything, depr...	non-suicide
232071	348107	[pee, probably, tastes, like, salty, tea, smir...	non-suicide
232072	348108	[usual, stuff, find, herei, posting, sympathy,...	suicide
232073	348110	[still, beaten, first, boss, hollow, knight, f...	non-suicide

232074 rows x 3 columns

1.2.2.8 Stemming

```
#stemming
ps = PorterStemmer()
df['text'] = df['text'].apply(lambda x : [ps.stem(i) for i in x])
df['text']=df['text'].apply(lambda x : ' '.join(x))
```

Figure 12 : Stemming

Stemming is performed to reduce words to their base form, which can help in normalization and minimizing the vocab size. In our pre-processing tasks, we are using Porter Stemming algorithm from the NLTK library. For each word in the text, it finds its stem using the Porter Stemmer. `df['text'].apply(lambda x : ' '.join(x))` joins the stemmed words back into a single string for each row in the “text” column.

Result:

Unnamed: 0		text	class
0	2	ex wife threaten suiciderec left wife good che...	suicide
1	3	weird get affect compliment come someon know i...	non-suicide
2	4	final 2020 almost never hear 2020 bad year eve...	non-suicide
3	8	need helpjust help im cri hard	suicide
4	9	losthello name adam 16 struggl year afraid pas...	suicide
...
232069	348103	like rock go get anyth go http musictast space...	non-suicide
232070	348106	tell mani friend lone everyth depriv pre bough...	non-suicide
232071	348107	pee probabl tast like salti tea smirking_fac s...	non-suicide
232072	348108	usual stuff find herei post sympathi piti know...	suicide
232073	348110	still beaten first boss hollow knight fought t...	non-suicide

232074 rows × 3 columns

Data Partitioning

```
# Split the dataset into training and testing sets first
X_train, X_test, y_train, y_test = train_test_split(dfnew['text'], dfnew['class'], test_size=0.2, random_state=42)
```

Figure 13 : Python For Data Partitioning

Data partitioning involves in splitting the dataset into training and testing set. This step is crucial to ensure that the model can be trained and evaluated effectively. The code snippet above has split the pre-processed data frame into `X_train` , `X_test`, `y_train`, `y_test`.

- **X_train, X_test:** These variables hold the text data for training and testing, respectively.
- **y_train, y_test:** These variables hold the corresponding class labels for training and testing.

Data Preparation

TF-IDF Vectorization

```
# Initialize a TF-IDF Vectorizer
vectorizer = TfidfVectorizer(stop_words='english', min_df=50,max_features=10000)

# Fit the vectorizer on the training data only and transform it
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
```

Figure 14 : TF-IDF Vectorization

In this project, we performed TF-IDF text vectorization.It converts text data to numerical features. TfidfVectorizer converts text into Term Frequency-Inverse Document Frequency

(TF-IDF) features, which are weighted depending on the frequency of terms in the document relative to their frequency in the overall dataset. Then the vectorizer is fitted on the training data. beside, the code also transform the testing data using the already fitted vectorizer.

Label Encoding

```
# Perform the label encoding after splitting the data
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
y_train_encoded = encoder.fit_transform(y_train)
y_test_encoded = encoder.transform(y_test)
```

Figure 15 : Class Label Encoding

The final step of data preparation before the dataset can be moved to modelling phase is to perform label encoding. In this code, the categorical label is converted into numerical format. “Label Encoder” function will assign a unique integer to each class label. Then use the fit transform function to fit the encoder on the training label and then transform the testing label using the already fitted encoder.

1.2.3 Model Justification

Wong Bin Jie – Logistics Regression

Logistics Regression well-suited in binary classification tasks, which aligns perfectly with the nature of sentiment analysis where texts are categorized as either positive or negative. As the dataset we used is to detect suicidal ideation, which often involves identifying concerning language or sentiments, its ability to model the probability of such binary outcomes is highly advantageous. Moreover, its simplicity and interpretability make it easier for researchers and practitioners to understand which factors contribute to the classification decision. This interpretability is particularly valuable in domains like suicide detection, where understanding the rationale behind classifications is crucial for effective intervention or support. Additionally, Logistic Regression performs reliably even with relatively small datasets, which is common in suicide detection studies due to the sensitivity and ethical considerations. In summary, Logistic Regression stands out for its suitability, interpretability, and performance, making it become one of the solid choices for text analysis and sentiment detection in suicide prevention efforts.

Siah Yao Liang – Multinomial Naïve Baiyes

Among various machine learning models, the Multinomial Naive Bayes (MNB) model stands out as a highly appropriate choice among machine learning models. First and foremost, MNB is well-suited for text classification problems because it can handle discrete data, such as word counts or term frequencies, which are common in natural language processing. The model's probabilistic design enables it to successfully handle the inherent uncertainties and variabilities in human language, making it proficient at detecting tiny signs and patterns indicative of suicidal intent. Overall, these advantages make the Multinomial Naive Bayes model an excellent choice for the suicide text detection project, balancing efficiency, accuracy, and deployment simplicity. This approach assures that the system can identify at-risk individuals in a timely and reliable way, thereby boosting proactive intervention efforts and, eventually, contributing to suicide prevention (Liu, 2012).

Shang Xin Yi – Decision Tree

Decision tree is an advantageous instrument for investigating suicidal ideation on social media as well as discovering implications. The main benefit of this methodology is that it is interpretable, making it simple for mental health professionals and academics to comprehend and see how particular words, phrases, and interactions between users can identify thoughts of suicide. Decision trees can be helpful in evaluating the diverse information obtainable on social media platforms because they can handle both textual and numerical data. Being able to identify complicated, non-linear interactions helps in the identification of minute patterns and risk variables connected to suicidal ideation. Decision trees are effective for real-time monitoring and action because they do not require a lot of data preprocessing. Decision trees enable transparent and actionable insights, which are essential for timely and effective mental health interventions, through providing clear decision pathways.

Q2. Supervised Text Classification

2.1 Model Code

2.1.1 Logistic Regression (WONG BIN JIE)

```
from sklearn.linear_model import LogisticRegression

# Initialize logistic regression model
logreg = LogisticRegression(max_iter=1000)
# Fit the logistic regression model on the training data
logreg.fit(X_train_tfidf, y_train_encoded)

# Predictions on the training data
train_pred = logreg.predict(X_train_tfidf)
# Predictions on the test data
test_pred = logreg.predict(X_test_tfidf)
```

Figure 16 LR Model Code

This code segment performs the essential steps of training a logistic regression model on TF-IDF transformed text data and using it to make predictions on both training and test datasets. First, it initializes the model using variable 'logreg' with max_iter set to be 1000. Then, it trains the model on training data, allowing it to learn the patterns in the dataset by implementing 'fit()' method. X_train_tfidf represents the training features, on the other hand, y_train_encoded represents the encoded labels for the training data. After the model is being trained, it is then used to make predictions on both training and testing dataset using the 'predict()' method. The result for predicting test data is stored in 'test_pred' and can be used to evaluate the model performance and accuracy in future.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
# Calculate the accuracy on the training set and test set
train_accuracy = accuracy_score(y_train_encoded, train_pred)
test_accuracy = accuracy_score(y_test_encoded, test_pred)

print("Training score: {:.2f}%".format(train_accuracy*100))
print("Testing score: {:.2f}%\n".format(test_accuracy*100))

from sklearn.metrics import classification_report
# Additional: Classification report for more insights
target_names = encoder.classes_
print(classification_report(y_test_encoded, test_pred, target_names=target_names))
```

Figure 17 LR Performance Code

To evaluate the model's performance, metrics such as accuracy, precision, recall and f1_score, as well as classification report are conducted. It generates a report based on the true labels (y_test_encoded) and the predicted labels (test_pred) for the test set. The result is then printed out to give a detailed overview of the model's performance.

```

Training score: 93.98%
Testing score: 93.50%

              precision    recall  f1-score   support

non-suicide      0.93       0.94       0.94       23051
suicide          0.94       0.93       0.93       23358

 accuracy         0.93         0.93         0.93       46409
 macro avg        0.94         0.94         0.93       46409
 weighted avg     0.94         0.93         0.93       46409

```

Figure 18 LR Performance Result

The result above shows the overall accuracy of the model on the training data is 93.98%, whereas the accuracy for the testing is 93.50%. The precision for class “non-suicide” is 0.93, and for the class “suicide” is 0.94. This indicates that the model's predictions for both classes are quite precise. Furthermore, for the class "non-suicide," the recall is 0.94, and for the class "suicide," it is 0.93. This means the model captures a high proportion of actual positive instances for both classes. For both classes, the F1-score is around 0.93, indicating good overall performance in terms of both precision and recall. The macro average calculates the metric independently for each class and then takes the average. The weighted average considers the number of instances for each class, giving more weight to classes with more instances. In this case, both macro and weighted averages are close to the metrics for the "suicide" class, indicating a balanced performance across classes.

```

from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Confusion matrix for testing data
train_conf_matrix = confusion_matrix(y_train_encoded, train_pred)
print("\nConfusion Matrix for Test Data:\n",train_conf_matrix)

# Confusion matrix for testing data
test_conf_matrix = confusion_matrix(y_test_encoded, test_pred)
print("\nConfusion Matrix for Test Data:\n",test_conf_matrix)

# Define function to plot confusion matrix
def plot_confusion_matrix(conf_matrix, title):
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)
    plt.title(title)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

# Plot confusion matrix for train set
plot_confusion_matrix(train_conf_matrix, title="Confusion Matrix - Train Set")

# Plot confusion matrix for test set
plot_confusion_matrix(test_conf_matrix, title="Confusion Matrix - Test Set")

```

Figure 19 LR Confusion Matrix Code

This code block provides a visual representation of the model's performance by plotting the confusion matrices using Seaborn’s heatmap, which help in understanding how well the model is predicting each class and identifying any patterns of misclassification.

```
Confusion Matrix for Train Data:
[[88157  4809]
 [ 6363 86305]]
```

```
Confusion Matrix for Test Data:
[[21778  1273]
 [ 1745 21613]]
```

Figure 20 LR Confusion Matrix Result

	Description	True Negative(TN)	False Positive (FP)	False Negative (FN)	True Positive (TP)
Train Data	92,966 instances of "non-suicide" and 92,668 instances of "suicide"	88,157 instances were correctly predicted as "non-suicide."	4,809 instances were incorrectly classified as "suicide" when they were actually "non-suicide."	6,363 instances were incorrectly classified as "non-suicide" when they were actually "suicide."	86,305 instances were correctly classified as "suicide."
Test Data	23,051 instances of "non-suicide" and 23,358 instances of "suicide"	21,778 instances were correctly classified as "non-suicide."	1,273 instances were incorrectly classified as "suicide" when they were actually "non-suicide."	1,745 instances were incorrectly classified as "non-suicide" when they were actually "suicide."	21,613 instances were correctly classified as "suicide."

Table 2 Confusion Matrix Table

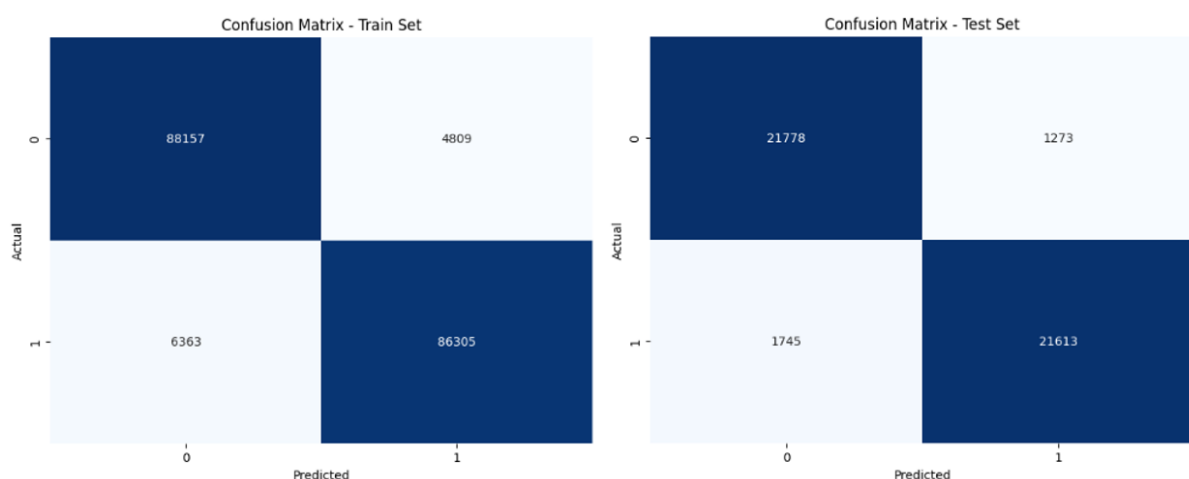


Figure 21 LR Confusion Matrix

The model appears to be slightly better at classifying "suicide" instances than "non-suicide" instances, as evidenced by the higher number of true positives and true negatives compared to false positives and false negatives in both the training and test data.

2.1.2 Multinomial Naïve Bayes (SIAH YAO LIANG)

```
# Initialize the Multinomial Naive Bayes classifier
nb_classifier = MultinomialNB()

# Fit the model on the training data
nb_classifier.fit(X_train_tfidf, y_train_encoded)

# Predict on the training set and test set
train_predictions = nb_classifier.predict(X_train_tfidf)
test_predictions = nb_classifier.predict(X_test_tfidf)
```

Figure 22 : Multinomial Naive Bayes Performance Code

The following code snippet explains how to use a Multinomial Naive Bayes classifier for text classification tasks, with a focus on detecting suicidal text. It starts with `MultinomialNB()` to initialize the classifier, then trains the model on the training data, `X_train_tfidf`, which is made up of text features transformed using Term Frequency-Inverse Document Frequency (TF-IDF) to represent the importance of each term, and `y_train_encoded`, which are encoded target labels. After fitting the model with `nb_classifier.fit`, predictions are performed on both the training and test sets using `nb_classifier.predict`, yielding `train_predictions` and `test_predictions`, respectively. This approach takes advantage of Multinomial Naive Bayes' efficiency and accuracy in text data processing to effectively classify suicidal thoughts in texts.

```
# Predict on the training set and test set
train_predictions = nb_classifier.predict(X_train_tfidf)
test_predictions = nb_classifier.predict(X_test_tfidf)

# Calculate the accuracy on the training set and test set
train_accuracy = accuracy_score(y_train_encoded, train_predictions)
test_accuracy = accuracy_score(y_test_encoded, test_predictions)

print("Training score:", train_accuracy)
print("Testing score:", test_accuracy)

# Additional: Classification report for more insights
print(classification_report(y_test_encoded, test_predictions))
```

Figure 23 : Multinomial Naive Bayes Performance Code

The code snippet uses a trained Naive Bayes classifier (`nb_classifier`) to make predictions on both the training and test datasets, which have been processed by TF-IDF. The model's accuracy on these datasets is then calculated using the `accuracy_score` function, which compares the predicted labels to the true encoded labels (`y_train_encoded` and `y_test_encoded`). The training and testing accuracies are printed to assess the model's performance. The `classification_report`

function also generates and prints a thorough classification report, which includes precision, recall, and F1-scores for each class, offering more information about the model's effectiveness.

```

Training score: 0.9038915284915479
Testing score: 0.9020017668986618

```

	precision	recall	f1-score	support
0	0.94	0.86	0.90	23051
1	0.87	0.95	0.91	23358
accuracy			0.90	46409
macro avg	0.91	0.90	0.90	46409
weighted avg	0.90	0.90	0.90	46409

Figure 24 : Multinomial Naive Bayes Classification Report

From the result above , we can see that the accuracy of the multinomial naïve bayes model is quite high which has a 90.39% of training accuracy and 90.20% of testing accuracy. For the non suicidal text (Class 0) have score 0.94 on precision, 0.86 on recall and 0.90 in f1-score while the suicidal text (Class 1) has a lesser precision of 0.87 , 0.09 higher recall than non suicidal and 0.91 f1-score. The model performs well overall, with F1-scores of approximately 90% and effective identification of both classes.

Confusion Matrix Result

```

# Confusion matrix for training data
train_conf_matrix = confusion_matrix(y_train_encoded, train_predictions)
print("\nConfusion Matrix for Train Data:\n",train_conf_matrix)

# Confusion matrix for testing data
test_conf_matrix = confusion_matrix(y_test_encoded, test_predictions)
print("\nConfusion Matrix for Test Data:\n",test_conf_matrix)

# Define function to plot confusion matrix
def plot_confusion_matrix(conf_matrix, title):
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)
    plt.title(title)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

# Plot confusion matrix for train set
plot_confusion_matrix(train_conf_matrix, title="Confusion Matrix - Train Set")

# Plot confusion matrix for test set
plot_confusion_matrix(test_conf_matrix, title="Confusion Matrix - Test Set")

```

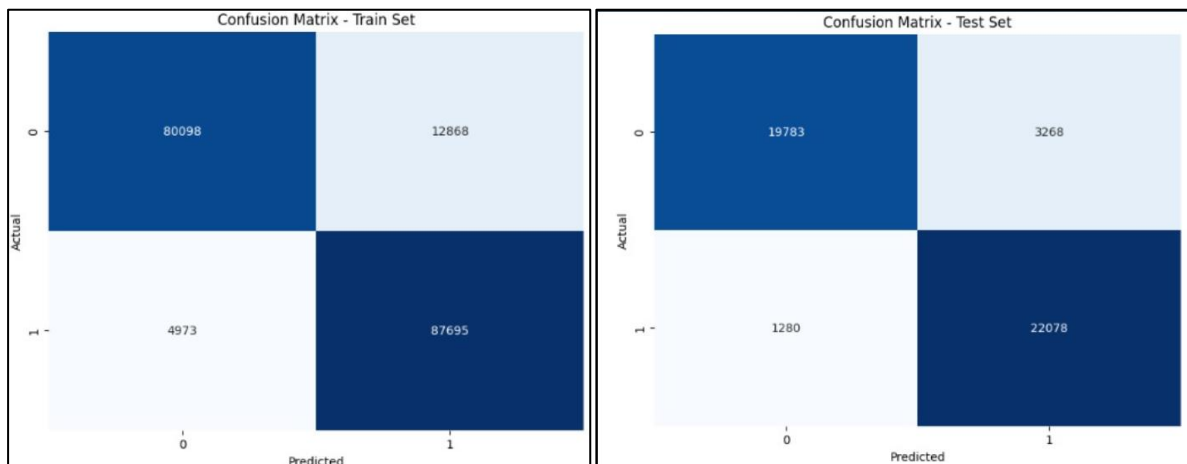
Figure 25 : Multinomial Naive Bayes Performance Code

```
Confusion Matrix for Train Data:
[[80098 12868]
 [ 4973 87695]]
```

```
Confusion Matrix for Test Data:
[[19783 3268]
 [ 1280 22078]]
```

Figure 26 : Multinomial Naive Bayes Confusion Matrix

	Description	True Negative(TN)	False Positive (FP)	False Negative (FN)	True Positive (TP)
Train Data	92,966 instances of "non-suicide" and 92,668 instances of "suicide"	80098 instances were correctly predicted as "non-suicide."	12868 instances were incorrectly classified as "suicide" when they were actually "non-suicide."	4973 instances were incorrectly classified as "non-suicide" when they were actually "suicide."	87695 instances were correctly classified as "suicide."
Test Data	23,051 instances of "non-suicide" and 23,358 instances of "suicide"	19783 instances were correctly classified as "non-suicide."	3268 instances were incorrectly classified as "suicide" when they were actually "non-suicide."	1280 instances were incorrectly classified as "non-suicide" when they were actually "suicide."	22078 instances were correctly classified as "suicide."



The model's performance, as indicated by the confusion matrices, suggests good capacity for prediction with some misclassifications. In the training data, 80,098 non-suicidal and 87,695

suicidal cases were accurately recognized; however, 12,868 non-suicidal texts were misclassified as suicidal, while 4,973 suicidal texts were misclassified as non-suicidal. In the test data, 19,783 non-suicidal and 22,078 suicidal cases were accurately identified, with 3,268 false positives and 1,280 false negatives. Overall, the model is highly accurate, however it might be improved in terms of removing incorrect classifications.

2.1.3 Decision Tree – (SHANG XIN YI)

```
# Initialize decision tree model
dtree = DecisionTreeClassifier()
# Fit the decision tree model on the training data
dtree.fit(X_train_tfidf, y_train_encoded)

# Predictions on the training data
train_pred = dtree.predict(X_train_tfidf)
# Predictions on the test data
test_pred = dtree.predict(X_test_tfidf)
```

Figure 27 DT Model Code

In order to focus on identifying suicidal texts, the above code example has demonstrated how to perform text classification tasks through a decision tree classifier. Used to indicate the importance of each term, `y_train_encoded` is the encoded target label, and then the model is trained on the training data `X_train_tfidf`, which consists of text features converted using term frequency-inverse document frequency (TF-IDF), and finally `DecisionTreeClassifier()` initializes the classifier. In order to obtain the data for `train_pred` and `test_pred`, `dtree.fit` will be used to fit the model and then `dtree.predict` will be used to predict the training set and test set. This method can effectively classify suicidal thoughts in text by leveraging the efficiency and accuracy of decision trees in text data processing.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
# Calculate the accuracy on the training set and test set
train_accuracy = accuracy_score(y_train_encoded, train_pred)
test_accuracy = accuracy_score(y_test_encoded, test_pred)

print("Training score: {:.2f}%".format(train_accuracy * 100))
print("Testing score: {:.2f}%\n".format(test_accuracy * 100))

from sklearn.metrics import classification_report
# Calculate the accuracy on the training set and test set
target_names = encoder.classes_
print(classification_report(y_test_encoded, test_pred, target_names=target_names))
```

Figure 28 DT Performance Code

The code snippet shows that a trained decision tree (`dtree`) is used to generate predictions for the training and prediction datasets that have been processed by TF-IDF. In order to ensure the accuracy of the calculated model on these datasets, the predicted labels can be compared with the true encoded labels (`y_train_encoded` and `y_test_encoded`) using the `accuracy_score` function. In addition, the training and test accuracy are printed to evaluate the performance of the model. In addition, the `classification_report` function can also create and publish a similar

classification report, which contains the accuracy, recall and F1 score for each category, providing more information about the model's effectiveness.

Training score: 99.92%				
Testing score: 86.03%				
	precision	recall	f1-score	support
non-suicide	0.86	0.86	0.86	23051
suicide	0.86	0.86	0.86	23358
accuracy			0.86	46409
macro avg	0.86	0.86	0.86	46409
weighted avg	0.86	0.86	0.86	46409

Figure 29 DT Performance Result

The decision tree model achieves good training and testing accuracy (99.92% and 86.03%, respectively). The non-suicidal text has a precision score of 0.86, a recall score of 0.86, and a f1-score of 0.86, suicidal text has a same of 0.86, a lower recall than the non-suicidal, and a f1-score of 0.86. Overall, the model works well, with F1-scores of around 86% and accurate identification of both classes.

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Confusion matrix for training data
train_conf_matrix = confusion_matrix(y_train_encoded, train_pred)
print("\nConfusion Matrix for Train Data:\n", train_conf_matrix)

# Confusion matrix for testing data
test_conf_matrix = confusion_matrix(y_test_encoded, test_pred)
print("\nConfusion Matrix for Test Data:\n", test_conf_matrix)

# Define function to plot confusion matrix
def plot_confusion_matrix(conf_matrix, title):
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False)
    plt.title(title)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

# Plot confusion matrix for train set
plot_confusion_matrix(train_conf_matrix, title="Confusion Matrix - Train Set")

# Plot confusion matrix for test set
plot_confusion_matrix(test_conf_matrix, title="Confusion Matrix - Test Set")
```

Figure 30 DT Confusion Matrix Code

By presenting the confusion matrices using Seaborn's heatmap, this code block gives users a visual depiction of the model's performance, making it easier to see patterns of misclassification and gauge how well the model predicts each class.

```

Confusion Matrix for Train Data:
[[92947   19]
 [  129 92539]]

Confusion Matrix for Test Data:
[[19905  3146]
 [ 3339 20019]]

```

Figure 31 DT Confusion Matrix Result

	True Negative(TN)	False Positive (FP)	False Negative (FN)	True Positive (TP)
Train Data	92,947 instances were correctly classified as "non-suicide."	19 instances were incorrectly classified as "suicide" when they were actually "non-suicide."	129 instances were incorrectly classified as "non-suicide" when they were actually "suicide."	92,539 instances were correctly classified as "suicide."
Test Data	19,905 instances were correctly classified as "non-suicide."	3,146 instances were incorrectly classified as "suicide" when they were actually "non-suicide."	3,339 instances were incorrectly classified as "non-suicide" when they were actually "suicide."	20,019 instances were correctly classified as "suicide."

Table 3 Confusion Matrix Table

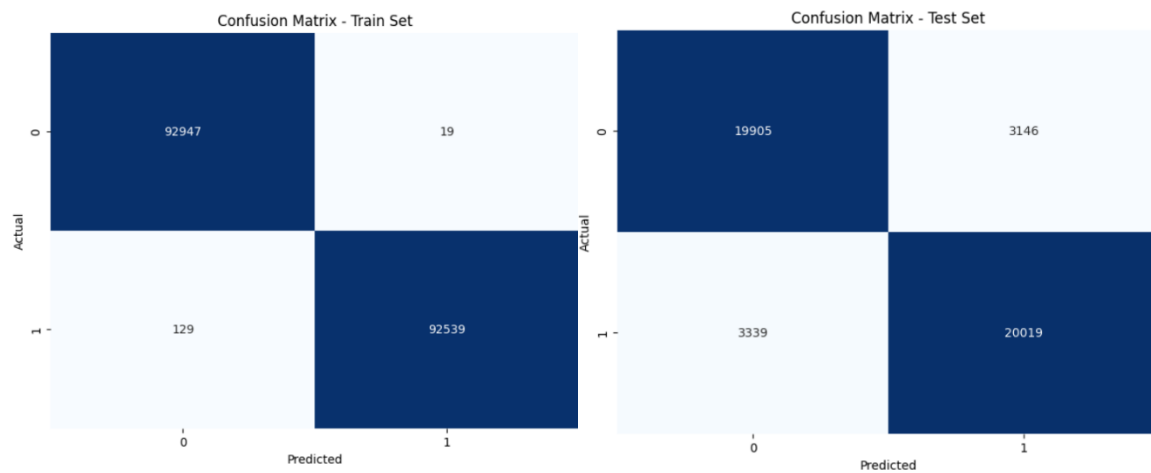


Figure 32 DT Confusion Matrix

The model appears to be slightly better at classifying "suicide" instances than "non-suicide" instances, as evidenced by the higher number of true positives and true negatives compared to false positives and false negatives in both the training and test data.

2.2 SAS Text Miner

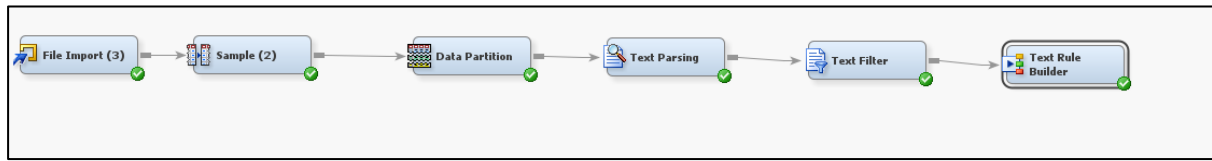
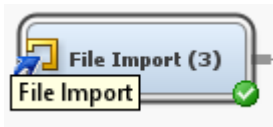


Figure 33 : SAS Text Miner Work Flow

SAS Text Miner is an SAS analytics tool used to analyze huge amounts of unstructured text data. It enables the extraction of useful information and patterns from text, allowing users to conduct sentiment analysis task. In this project, we are not only perform the text analysis using python code but also user SAS Text Miner is because it offers a comprehensive, integrated environment for managing and analysing text data, including a lot of features for data preparation, text parsing, and advanced analytics approaches such as text rules builder. This makes it ideal for large-scale, enterprise-level text mining processes.

2.2.1 File Import node



Property	Value
General	
Node ID	FIMPORT3
Imported Data	...
Exported Data	...
Notes	...
Train	
Variables	...
Import File	C:\Users\andre\OneDrive\...
Maximum Rows to Import	1000000
Maximum Columns to Import	10000
Delimiter	,
Name Row	Yes
Number of Rows to Skip	0
Guessing Rows	500
File Location	Local
File Type	xlsx
Advanced Advisor	No
Rerun	No
Score	
Role	Train
Report	
Summarize	No
Status	
Create Time	5/11/24 7:59 AM
Run ID	cdc1a417-172d-7a40-a84d-2
Last Error	
Last Status	Complete
Last Run Time	5/13/24 3:09 AM
Run Duration	0 Hr, 2 Min, 59.33 Sec.
Grid Host	
User-Added Node	No

Figure 34 : File Import node and Properties

Firstly , we use the File import node to import our suicide text data into SAS environment. In this case, we have import our dataset as XLSX file format due to its stability instead of using csv file. This is because there are many comma in a text sentences and if we use csv file, the text and class will be mess up because csv file is using comma separated to define the words.

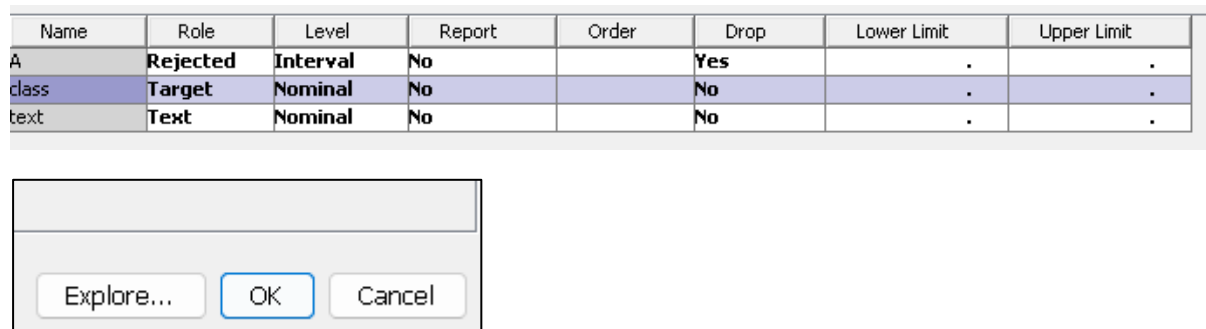


Figure 35 : Properties Explore

After the dataset is import into SAS , it is found that there are 3 column which is A , Class and Text. A is the number of row which we design to drop it , while class is the suicide and non-suicide class , text is the sentences in the dataset. We explore this dataset by clicking explore button.

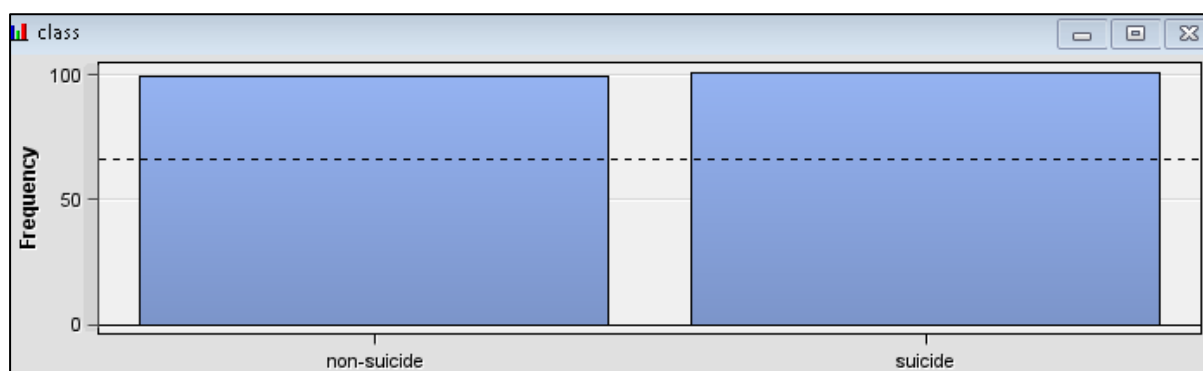
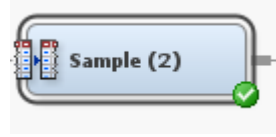


Figure 36 : Class Distribution

Property	Value
Rows	232074
Columns	3
Library	EMWS4
Member	FIMPORT3_DATA
Type	DATA
Sample Method	Top
Fetch Size	Default
Fetch Rows	200
Random Seed	12345

By looking at the images above , we can have a simple understanding on it . The bar chart has depict an equal distribution of “non-suicide” and “suicide” classes each with a frequency of 100. We have fetch 200 row to explore the dataset. The dataset's properties show that it has 232,074 rows and 3 columns, indicating a large amount of text data for analysis. A random seed of 12345 ensures reproducibility in data sampling and analysis processes.

2.2.2 Sample Node



Property	Value
General	
Node ID	Smpl2
Imported Data	...
Exported Data	...
Notes	...
Train	
Variables	...
Output Type	Data
Sample Method	Default
Random Seed	12345
Size	
Type	Percentage
Observations	.
Percentage	10.0
Alpha	0.01
PValue	0.01
Cluster Method	Random
Stratified	
Criterion	Proportional
Ignore Small Strata	No
Minimum Strata Size	5
Level Based Options	
Level Selection	Event
Level Proportion	100.0
Sample Proportion	50.0
Oversampling	
Adjust Frequency	No
Based on Count	No
Exclude Missing Levels	No
Report	
Interval Targets	Yes
Class Targets	Yes

Figure 37 : Sample Node and Properties

Then we perform a sample node to perform data sampling due to too large amount of dataset. This properties is set up to output data using the default sampling method and a random seed of 12345, assuring consistency. The sampling method is percentage-based, with the training sample comprising 10% of the total data. The alpha and p-value are both set to 0.01, indicating that statistical tests have a 1% level of significance. The cluster approach is random, and the sampling is stratified with a proportional criterion to keep the balance between different strata

Sampling Summary		
Type	Data Set	Number of Observations
DATA	EMWS4.FIMPORT3_train	232074
SAMPLE	EMWS4.Smpl2_DATA	23208

Figure 38 : Sampling Summary

This is the result after data sampling , we have pick 10% of the dataset which is 23208 to decrease the amount of text.

2.2.3 Data Partition Node




Property	Value
General	
Node ID	Part
Imported Data	...
Exported Data	...
Notes	...
Train	
Variables	...
Output Type	Data
Partitioning Method	Default
Random Seed	12345
Data Set Allocations	
Training	70.0
Validation	0.0
Test	30.0
Report	
Interval Targets	Yes
Class Targets	Yes
Status	
Create Time	5/11/24 4:28 AM
Run ID	6606e90f-4951-4046-aa
Last Error	
Last Status	Complete
Last Run Time	5/13/24 3:13 AM
Run Duration	0 Hr. 0 Min. 6.44 Sec.
Grid Host	
User-Added Node	No

Figure 39 : Data Partition Node and Properties

Data Partition node is implemented to split the data into training and testing set which is crucial for the modelling and evaluation process. We have split the dataset into 70:30 which 70% of the data will be the training data and 30% of the rest will be the test data.

2.2.4 Text Parsing Node



Property	Value
General	
Node ID	TextParsing
Imported Data	...
Node ID	...
Node identifier	...
Variables	...
Parse	
Parse Variable	text
Language	English ...
Detect	
Different Parts of Speech	Yes
Noun Groups	Yes
Multi-word Terms	SASHELP.ENG_MULT_...
Find Entities	None
Custom Entities	
Ignore	
Ignore Parts of Speech	'Aux' 'Conj' 'Det' 'Int' ...
Ignore Types of Entity	...
Ignore Types of Attribute	'Num' 'Punct' ...
Synonyms	
Stem Terms	Yes
Synonyms	SASHELP.ENGSYNM! ...
Filter	
Start List	...
Stop List	SASHELP.ENGSTOP ...
Select Languages	...
Report	
Number of Terms to Display	20000
Status	
Create Time	5/11/24 9:50 AM
Run ID	7194c67f-3e04-474c-b

Figure 40 : Text Parsing Node and Properties

Text parsing is critical for cleaning, organizing, and structuring unstructured text data. It makes it easier to extract characteristics, tag sections of speech, and detect objects by converting text into a format that can be analysed and modelled further in natural language processing. The text parsing properties described in the configuration give a complete framework for processing and analysing text. The Node ID Text Parsing indicates the parsing node that handles the imported dataset's text column in English. The parser is set up to detect various components of speech, such as noun groups and multi-word terms, using a predetermined list from SASHELP.ENG_MULTI_WORD. It ignores certain components of speech such as auxiliary verbs, conjunctions, determiners, and punctuation, focusing on more informative phrases.

2.2.4 Text Filter Node

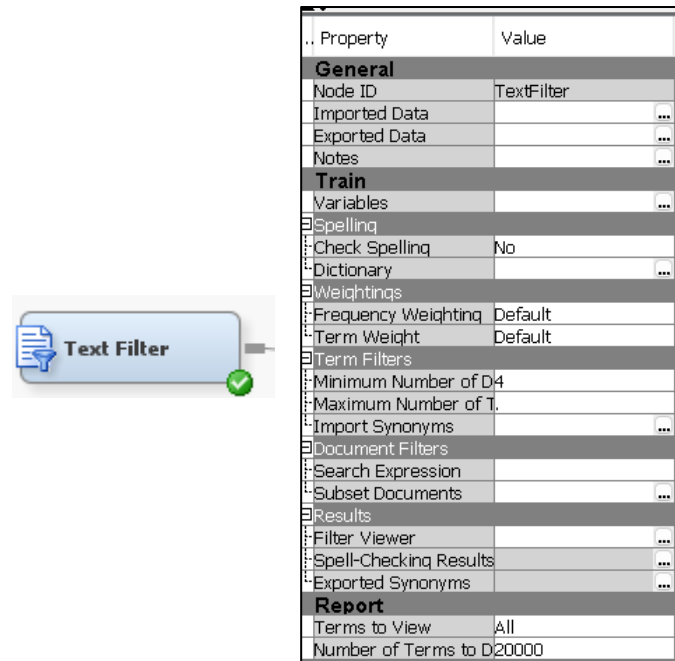


Figure 41 : Text Filter Node and Properties

The major purpose of text filtering in this context is to remove unnecessary terms, minimize noise, and highlight significant phrases, ensuring that the dataset is clean, relevant, and ready for accurate analysis and modelling in text mining and natural language processing tasks. In this project, we used the text filter's default properties. To improve the quality and relevancy of the imported data, a number of filtering stages are applied. Spelling checks are deactivated, meaning that spelling errors will not be corrected, and no dictionary field is supplied. The frequency and term weighting variables are set to default, ensuring that terms are weighted depending on their occurrence in the dataset. Term filters are used to eliminate terms that appear in fewer than four documents, minimizing noise while focusing on significant phrases.

2.2.5 Text rule builder



Figure 42 : Text Rule Builder Node

The last step is to implement the prepared dataset into a text rule builder node. SAS's Text Rule Builder is an advanced tool for creating rule-based text categorization models. It enables users to design, assess, and refine rules for categorizing text input based on patterns and keywords. This tool is especially beneficial for jobs such as suicide text detection, which require identifying certain phrases and language patterns. Using the Text Rule Builder, analysts can use machine learning and domain expertise to create strong models that reliably identify text data, boosting detection and intervention processes in sensitive applications such as mental health monitoring.

Event Classification Table			
Data Role=TRAIN Target=class Target Label=class			
False Negative	True Negative	False Positive	True Positive
933	7329	794	7189

Figure 43 : Event Classification Table

The image above shows an event classification table illustrates the model's performance in classifying instances of the target variable 'class'. It contain 7329 number of true negatives , 7189 of true positives , 933 false negatives and 794 false positives . True negatives are texts that accurately detect as non-suicidal, while true positives are texts that are successfully classify as suicidal. On the other hand, False negatives are suicidal texts that were wrongly classified as non-suicidal, and false positives are non-suicidal texts that were wrongly classified as suicidal. This distribution demonstrates the ability of the model to correctly categorize the majority of the messages while also suggesting opportunities for development, particularly in reducing false negatives to ensure that it will not missed any suicidal texts

Fit Statistics						
Target	Target Label	Fit Statistics	Statistics Label	Train	Validation	Test
class	class	ASE	Average Squared ...	0.006593		0.006397
class	class	DIV	Divisor for ASE	32490		13926
class	class	MAX	Maximum Absolut...	0.464194		0.443257
class	class	NOBS	Sum of Frequencies	16245		6963
class	class	RASE	Root Average Squ...	0.081197		0.07998
class	class	SSE	Sum of Squared E...	214.2033		89.0813
class	class	DISF	Frequency of Clas...	16245		6963
class	class	MISC	Misclassification ...	0.10631		0.130547
class	class	WRONG	Number of Wrong ...	1727		909

Figure 44 : Fit Statistic

The fit statistics table provides the model's misclassification rate (MISC), which has a direct impact on accuracy. The MISC for the training set is 0.10631, which means that 10.63% of the training instances are misclassified, resulting in an accuracy of about 89.37%. The same MISC result is observed for the test set, indicating a constant misclassification rate of 13.05% and an accuracy of 86.95%. These statistics demonstrate the model's effectiveness in accurately categorizing texts while also indicating areas for potential improvement in lowering the misclassification rate in order to achieve greater accuracy.

Q3. Hyper Parameter Selection

3.1 List and explain the hyper parameters of all your models.

3.1.1 Logistic Regression (WONG BIN JIE)

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'penalty': ['l1', 'l2'], # Regularization penalty
    'C': [0.001, 0.01, 0.1, 1, 10, 100], # Inverse of regularization strength
    'solver': ['liblinear', 'saga']
}

# Initialize logistic regression model
logreg = LogisticRegression()

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=logreg, param_grid=param_grid, cv=10, scoring='accuracy', verbose=1, n_jobs=-1)

# Fit the grid search to the data
grid_search.fit(X_train_tfidf, y_train_encoded)
```

Figure 45 : Logistic Regression Tuning Code

Logistic Regression Hyperparameter

- Penalty: Specifies the norm of the penalty (regularization) to be used. l1 is also known as lasso, which can result in sparse models with fewer parameters. On the other hand, l2 (ridge) penalizes the sum of the squared coefficients.
- C is the inverse of regularization strength; must be a positive float. Smaller values specify stronger regularization. C=0.001 is the strong regularization, and C=100 is the weak regularization.
- Solver is the algorithm to use in the optimization problem. liblinear is a good choice for small datasets as it handles L1 and L2 regularization, whereas saga is suitable for large datasets; it supports both L1 and L2 regularization and is a variant of the stochastic average gradient descent.

GridSearch CV Hyperparameter

- estimator is the machine learning model instance for which hyperparameters need to be tuned. In this case, the model is Logistic Regression, which is assigned as logreg.
- param_grid is the dictionary specifying the parameters and their ranges to be searched.
- cv is the number of cross-validation folds. In the code above, it shows cv=10, where it means the dataset will be split into 10 parts, and the model will be trained and validated 10 times, each time with a different part as the validation set.
- scoring is the metric to evaluate the performance of the model. The option we choose is 'accuracy', where it measures the ratio of correctly predicted instances to the total instance.

- verbose controls the verbosity of the output, and the option is 1, where some information will be printed during the execution of the grid search.
- n_jobs is the number of jobs to run in parallel. It uses all available processors to perform the grid search in parallel.

```
Fitting 10 folds for each of 24 candidates, totalling 240 fits
Best parameters: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
Best cross-validation score: 0.94
```

- The grid search performed 10-fold cross-validation for each combination of the hyperparameters specified in param_grid. There are 24 different combinations of the hyperparameters (2 penalties * 6 C values * 2 solvers = 24). Therefore, the total number of fits performed is 10 folds * 24 combinations = 240 fits.

3.1.2 Multinomial Naïve Bayes (SIAH YAO LIANG)

```
# Parameters Grid
param_grid = {
    'alpha': [0.01, 0.1, 1, 10],
    'fit_prior': [True, False]
}

# Grid Search Initialization
grid_search = GridSearchCV(MultinomialNB(), param_grid, cv=10, scoring='accuracy', verbose=1)
grid_search.fit(X_train_tfidf, y_train_encoded) # Assuming these variables are predefined
```

Figure 46 : Multinomial Naïve Bayes Tuning Code

Multinomial Naïve Bayes Hyperparameter

- Alpha
 - ✓ Alpha is use to controls the smoothing applied to handle zero probabilities in the model. In this case , I have use alpha value of 0.01,0.1,1 and 10.
- Fit_prior
 - ✓ Fit_prior is used to determine whether to learn the class prior probabilities.

Grid search parameters

- Cv
 - ✓ In the grid search function , I have choose to specifies 10 fold cross validation to evaluate each hyperparameter combination.
- Scoring
 - ✓ I have uses accuracy as the metric to optimize the model during grid search.
- Verbose
 - ✓ For the verbose value , I have control 1 verbosity level of the output during the grid search process.

```
Fitting 10 folds for each of 8 candidates, totalling 80 fits
Best parameters: {'alpha': 0.01, 'fit_prior': True}
Best cross-validation score: 0.90
```

The grid search results indicate that the best hyperparameters for the Multinomial Naive Bayes model are alpha set to 0.01 and fit_prior set to True. This combination achieved the highest cross-validation score of 0.90, reflecting the model's optimal performance with these settings.

3.1.3 Decision Tree (SHANG XIN YI)

```
# Define the parameter grid for Decision Tree
param_grid = {
    'criterion': ['gini', 'entropy'], # Criterion for splitting
    'max_depth': [None, 10, 20, 30, 40, 50], # Maximum depth of the tree
    'min_samples_split': [2, 5, 10], # Minimum number of samples required to split an internal node
    'min_samples_leaf': [1, 2, 4] # Minimum number of samples required to be at a leaf node
}

# Initialize Decision Tree classifier
dt = DecisionTreeClassifier()

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=dt, param_grid=param_grid, cv=3, scoring='accuracy', verbose=1, n_jobs=-1)

# Fit the grid search to the data
grid_search.fit(X_train_tfidf, y_train_encoded)
```

Figure 47 Decision Tree Tuning Code

Decision Tree

- Criterion
 - Criteria specify the function for evaluating the quality of a split.
- max_depth
 - max_depth limits the number of layers in a tree.
- min_samples_split
 - minimum number of samples required to split an internal node.
- min_sample_leaf
 - minimum number of samples required to be at a leaf node.

Grid Search Parameters

- estimator
 - Identify the machine learning model instance that requires hyperparameter tuning.
 - As a result, Decision Tree was picked and granted the name dt.
- param_grid
 - dictionary of parameters and their corresponding values to be searched.
- Cv
 - For this function, I used 3-fold cross validation to assess each hyperparameter combination.
- Scoring

- Use metrics to assess the method's effectiveness. I chose accuracy, which will assess the ratio of accurately predicted occurrences to total instances.
- verbose
 - I control 1 verbose value of the output during the grid search process.
- n_jobs
 - The value of 'n_jobs=-1', it uses all available processors to preform the grid search in parallel.

```
Fitting 3 folds for each of 108 candidates, totalling 324 fits  
Best parameters: {'criterion': 'gini', 'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5}  
Best cross-validation score: 0.86
```

The grid search performed 3-fold cross-validation for each combination of the hyperparameters specified in param_grid. There are 108 different combinations of the hyperparameters $2 \text{ criterion} * 6 \text{ max_depth} * 3 \text{ min_samples_split} * 3 \text{ min_samples_leaf} = 108$. Therefore, the total number of fits performed is $108 \text{ candidates} * 3 \text{ folds} = 324 \text{ fits}$.

3.2 Using grid search or random search methods, report the hyper parameters of your models with results.

3.2.1 Logistic Regression – (WONG BIN JIE)

```
# Best Parameters and Score
print("Best parameters:", grid_search.best_params_)
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))

# Using the best model found on the test data
best_model = grid_search.best_estimator_
test_predictions = best_model.predict(X_test_tfidf)

test_accuracy = accuracy_score(y_test_encoded, test_predictions)
print("Test accuracy with best parameters:", test_accuracy)

# Classification Report
print("\nClassification Report:")
print(classification_report(y_test_encoded, test_predictions))

# confusion matrix
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(y_test_encoded, test_predictions)
fig, ax = plt.subplots(1,1,figsize=(7,4))

ConfusionMatrixDisplay(confusion_matrix(test_predictions,y_test_encoded,labels=[1,0]),
                        display_labels=[1,0]).plot(values_format=".0f",ax=ax)

ax.set_xlabel("True Label")
ax.set_ylabel("Predicted Label")
plt.show()
```

Figure 48 : Logistic Regression with Grid Search

- Logistic regression has using ‘GridSearchCV’ to performs hyperparameter tuning. The figure above shows the implemented code to display the result (accuracy, classification report and confusion matrix) after tuning.

```
Best parameters: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
Best cross-validation score: 0.94
Test accuracy with best parameters: 0.935723674287315

Classification Report:
              precision    recall  f1-score   support

     0       0.93       0.94       0.94       23051
     1       0.94       0.93       0.94       23358

 accuracy          0.94
 macro avg         0.94
 weighted avg      0.94
```

- Based on the result, the optimal hyperparameter found are:
 - ✓ Regularization parameter C: 10
 - ✓ Penalty type: L2 (Ridge)
 - ✓ Solver: ‘liblinear’

The highest accuracy achieved during cross-validation is 0.94 (94%). Cross-validation is a technique used to evaluate the performance of a model by partitioning the data into multiple subsets (folds), training the model on some folds, and validating it on the remaining folds.

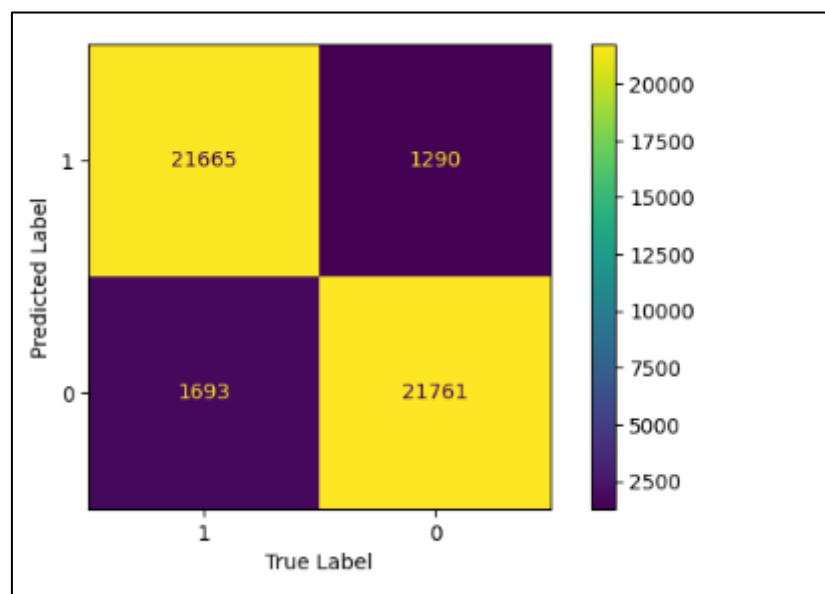
Test Performance

Accuracy: 93.57%

Precision for non-suicide class: 0.93, suicide class: 0.94.

Recall for non-suicide class: 0.94, suicide class: 0.93.

F1-score for non-suicide class: 0.94, suicide class: 0.94.



True Label is the actual classes from the dataset, whereas Predicted Label are the classes predicted by the model. The colours in the confusion matrix range from purple to yellow, with yellow indicating higher values. This colour gradient helps in quickly identifying where the majority of predictions fall.

Interpretation

- True Positives (top-left): Cases where the actual class is 1 and the model predicted 1. The model correctly predicted 21665 instances as positive.
- False Positives (top-right): Cases where the actual class is 0 but the model predicted 1. The model incorrectly predicted 1290 negative instances as positive.
- False Negatives (bottom-left): Cases where the actual class is 1 but the model predicted 0. The model incorrectly predicted 1693 positive instances as negative.
- True Negatives (bottom-right): Cases where the actual class is 0 and the model predicted 0. The model correctly predicted 21761 instances as negative.

Short Summary:

The grid search method successfully identified the optimal hyperparameters for the model, achieving a high cross-validation score of 0.94 and a test accuracy of approximately 0.94. The classification report and confusion matrix further validate the model's robustness and ability to generalize well to unseen data. The confusion matrix indicates that the model is more prone to making False Negatives (1693) than False Positives (1290). This means the model more frequently fails to detect positives (class 1) and incorrectly labels them as negatives (class 0), rather than incorrectly labelling negatives (class 0) as positives (class 1).

3.2.2 Multinomial Naïve Bayes – (SIAH YAO LIANG)

```
# Best Parameters and Score
print("Best parameters:", grid_search.best_params_)
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))

# Using the best model found on the test data
best_model = grid_search.best_estimator_
test_predictions = best_model.predict(X_test_tfidf)
test_accuracy = accuracy_score(y_test_encoded, test_predictions)
print("Test accuracy with best parameters:", test_accuracy)

# Classification Report
print("\nClassification Report:")
print(classification_report(y_test_encoded, test_predictions))

# confusion matrix
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(y_test_encoded, test_predictions)
fig, ax = plt.subplots(1,1,figsize=(7,4))

ConfusionMatrixDisplay(confusion_matrix(test_predictions,y_test_encoded,labels=[1,0]),
                        display_labels=[1,0]).plot(values_format=".0f",ax=ax)

ax.set_xlabel("True Label")
ax.set_ylabel("Predicted Label")
plt.show()
```

Figure 49 : Multinomial Naïve Bayes with Grid Search

The process involves evaluating the best hyperparameters identified by grid search. The best model is used to predict test data, and its accuracy is calculated. A classification report is generated to detail precision, recall, and F1-score. Finally, a confusion matrix is displayed to visualize the model's performance in classifying test data.

```

Fitting 10 folds for each of 8 candidates, totalling 80 fits
Best parameters: {'alpha': 0.01, 'fit_prior': True}
Best cross-validation score: 0.90
Test accuracy with best parameters: 0.902389622702493

Classification Report:

```

	precision	recall	f1-score	support
0	0.94	0.86	0.90	23051
1	0.87	0.95	0.91	23358
accuracy			0.90	46409
macro avg	0.91	0.90	0.90	46409
weighted avg	0.91	0.90	0.90	46409

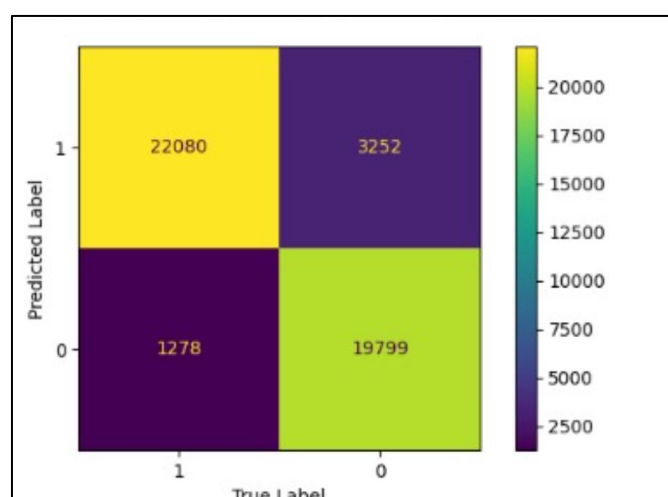
Based on the result, the optimal hyperparameters found are:

- alpha: 0.01
- fit_prior: True
- The highest accuracy achieved during cross-validation is 0.90 (90%).
- Cross-validation is a technique used to evaluate the performance of a model by partitioning the data into multiple subsets (folds), training the model on some folds, and validating it on the remaining folds.

Test Performance

- Accuracy: 90.24%
- Precision for non-suicide class: 0.94, suicide class: 0.87.
- Recall for non-suicide class: 0.86, suicide class: 0.95.
- F1-score for non-suicide class: 0.90, suicide class: 0.91.

Confusion Matrix Result



Interpretation

- **True Positives (top-left):** Cases where the actual class is 1 and the model predicted 1. The model correctly predicted 22,080 instances as positive.
- **False Positives (top-right):** Cases where the actual class is 0 but the model predicted 1. The model incorrectly predicted 3,252 negative instances as positive.
- **False Negatives (bottom-left):** Cases where the actual class is 1 but the model predicted 0. The model incorrectly predicted 1,278 positive instances as negative.
- **True Negatives (bottom-right):** Cases where the actual class is 0 and the model predicted 0. The model correctly predicted 19,799 instances as negative.

Short Summary

The grid search method successfully identified the optimal hyperparameters for the model, achieving a high cross-validation score of 0.90 and a test accuracy of approximately 0.90. The classification report and confusion matrix further validate the model's robustness and ability to generalize well to unseen data. The confusion matrix indicates that the model is more prone to making False Positives (3,252) than False Negatives (1,278). This means the model more frequently incorrectly labels negatives (class 0) as positives (class 1) rather than failing to detect positives (class 1) and incorrectly labelling them as negatives (class 0).

3.2.3 Decision Tree - (SHANG XIN YI)

```
test_accuracy = accuracy_score(y_test_encoded, test_predictions)
print("Test accuracy with best parameters:", test_accuracy)

# Classification Report
print("\nClassification Report:")
print(classification_report(y_test_encoded, test_predictions))

# confusion matrix
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(y_test_encoded, test_predictions)
fig, ax = plt.subplots(1,1,figsize=(7,4))

ConfusionMatrixDisplay(confusion_matrix(test_predictions,y_test_encoded,labels=[1,0]),
                        display_labels=[1,0]).plot(values_format=".0f",ax=ax)

ax.set_xlabel("True Label")
ax.set_ylabel("Predicted Label")
plt.show()
```

Figure 50 Decision Tree with GridSearch

The technique entails analysing the best hyperparameters discovered using grid search. The most effective model is employed to forecast test results, and its accuracy is calculated. A classification report is created that includes precision, recall, and F1-score. Finally, a confusion matrix is shown to illustrate the model's effectiveness in identifying test data.

```
Fitting 3 folds for each of 108 candidates, totalling 324 fits
Best parameters: {'criterion': 'gini', 'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5}
Best cross-validation score: 0.86
```

```
Test accuracy with best parameters: 0.8614237288135593

Classification Report:
              precision    recall  f1-score   support

     0       0.83         0.90         0.87         7368
     1       0.89         0.82         0.86         7382

 accuracy                   0.86         14750
 macro avg              0.86         0.86         0.86         14750
weighted avg              0.86         0.86         0.86         14750
```

Based on the result, the optimal hyperparameter found are:

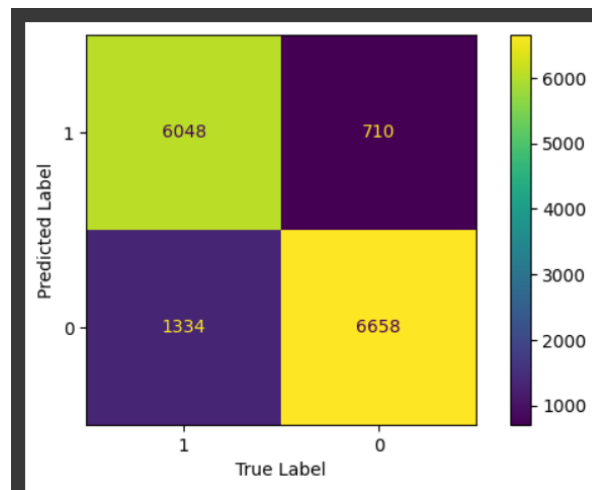
- Criterion: gini
- max_depth :20
- min_samples_leaf: 1
- min_samples_split: 5
- The highest accuracy achieved during cross validation is 0.90 (90%)

- Cross-validation is a technique to evaluate the performance of a model by partitioning the data into multiple subsets (folds), training the model on some folds, and validating it on the remaining folds.

Test Performance

- Accuracy: 0.86
- Precision for non-suicide class: 0.83, suicide class: 0.89
- Recall of non-suicide class: 0.90, suicide class: 0.82
- F1-score for non-suicide class: 0.87, suicide class: 0.86

Confusion Matrix Result



Interpretation

- True Positives (top-left): Cases where the actual class is 1 and the model predicted 1. The model correctly predicted 6048 instances as positive.
- False Positives (top-right): Cases where the actual class is 0 but the model predicted 1. The model incorrectly predicted 710 negative instances as positive.
- False Negatives (bottom-left): Cases where the actual class is 1 but the model predicted 0. The model incorrectly predicted 1334 positive instances as negative.
- True Negatives (bottom-right): Cases where the actual class is 0 and the model predicted 0. The model correctly predicted 6658 instances as negative.

Short Summary

The grid search method successfully identified the optimal hyperparameters for the model, achieving a high cross-validation score of 0.86 and a test accuracy of approximately 0.86. The classification report and confusion matrix further validate the model's robustness and ability to generalize well to unseen data. The confusion matrix indicates that the model is more prone to making False Positives (710) than False Negatives (1,334). This means the model more frequently incorrectly labels negatives (class 0) as positives (class 1) rather than failing to detect positives (class 1) and incorrectly labelling them as negatives (class 0).

Q4. Evaluation & Discussion of the predictive models' results

4.1 Select evaluation metrics and briefly describe them.

Metric	Definition	Formula	Advantages/Suitability
Accuracy	Correctly classified instances to the total instances	$\frac{TP + TN + FP + FN}{TP + TN}$	Simple to intuitive; not suitable for imbalanced datasets
Precision	True positive predictions to all positive predictions	$\frac{TP}{TP + FP}$	Determines the model's ability to eliminate false positives; helpful in cost-sensitive circumstances.
Recall	True positive prediction to all actual positives	$\frac{TP + FN}{TP}$	Indicates how effectively the model catches all relevant positives; critical for preventing false negatives.
F1 – Score	Harmonise the mean of accuracy and recall, balancing both measurements.	$2 \times \frac{Precision \times Recall}{Precision + Recall}$	Balances accuracy and recall, making it beneficial for data that is uneven.
Confusion Matrix	A table comparing actual and anticipated categorization.	Below	Provides a thorough performance breakdown, which is important for diagnostics.

Confusion Matrix

	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Explanation of Key Terms:

True Positive (TP) is an accurately expected positive case.

True Negative (TN) is an accurately expected negative case.

False Positive (FP) is an inaccurately expected positive case.

False Negative (FN) is an inaccurately expected negative case.

4.2 Perform evaluation and present your results (compare base result and tuning result)

4.2.1 Logistic Regression

Logistic Regression	Default Parameter	After Tuning
Accuracy	93.50%	93.57%

The table shows the accuracy of the logistic regression model before and after hyperparameter tuning. Given the marginal improvement in accuracy (0.07%), it's reasonable to choose the model with hyperparameter tuning as the "best" model, simply because it has a slightly higher accuracy. However, in practical terms, the difference is very small and may not significantly impact the model's performance. If computational resources are not a constraint, it's generally a good practice to opt for the model with slightly higher performance, even if the improvement is minor. Therefore, the logistic regression model with hyperparameter tuning would be chosen as the best model in this case.

4.2.2 Multinomial Naïve Bayes

Multinomial Naïve Bayes	Default Parameter	After Tuning
Accuracy	90.20%	90,23%

Based on the results, the model's performance with default parameters and after grid search tuning shows minimal difference. The accuracy of the Multinomial Naive Bayes model with default parameters is 90.20%, and after tuning, the accuracy slightly improves to 90.23%. This marginal improvement suggests that the initial configuration of the model was already well-optimized. Given the negligible difference in performance, it is practical to conclude that either model could be considered effective. However, for consistency and to reflect the benefit of tuning, the model with the tuned parameters (alpha set to 0.01 and fit_prior set to True) should be chosen as the final model. This model represents the best version after considering potential improvements through hyperparameter tuning, even if the gain in accuracy is minimal. Therefore, the final best model selected is the one with the hyperparameters found through the grid search tuning process.

4.2.3 Decision Tree – (SHANG XIN YI)

Decision Tree	Default Parameter	After Tuning
Accuracy	86.03%	86.14%

The Decision Tree classifier's accuracy before and after hyperparameter adjustment is contrasted in the table. At first, the model's accuracy with default settings was 86.03%. Following the application of methods such as GridSearchCV to optimize the hyperparameters, the accuracy increased to 86.14%. The significance and efficacy of hyperparameter tweaking in improving model performance by identifying the best classifier parameters is demonstrated by the 0.11 percentage point gain.

4.3 Discuss and critically analyse the results your predictive models built to choose the best one. You may compare the obtained results with the previous works.

4.3.1 Best Model Justification

Model	Default Model	Tune Model
Multinomial Naïve Bayes	90.20%	90.23%
Logistic Regression	93.50%	93.57% ✓
Decision Tree	86.03%	86.14%

The table compares the accuracy of three predictive models—Multinomial Naive Bayes, Logistic Regression, and Decision Tree—before and after hyperparameter tuning. The Multinomial Naive Bayes model shows a minimal improvement after tuning, from 90.20% to 90.23%, showing that its original configuration was already very near to the optimal. The Decision Tree model likewise improves from 86.03% to 86.14%, indicating some improvement from tuning but still falling behind the other models. The Logistic Regression model has the highest accuracy, with an increase from 93.50% to 93.57% after tuning. This minor improvement concretes its place as the best-performing model of the three. Given its greater accuracy and consistent performance both before and after tune, the Logistic Regression model is the best option for this predictive task. This analysis highlights the importance of both initial model setup and the possible improve of hyperparameter tuning.

4.3.2 Compare with previous work

Author	SIAH YAO LIANG, WONG BIN JIE, SHANG XIN YI (Our Project)	Chiroma, Liu, & Cocea	Jain et al.	Lee & Pak	Sebastian et al.
Model Used	Multinomial Naïve Bayes, Decision Tree, Logistic Regression	Decision Tree, Naive Bayes, Random Forest, SVM	Naive Bayes, SVM, Random Forest, Logistic Regression	Logistic Regression, SVM, XGBoost	SVM, Decision Tree
Best Model	Logistic Regression	decision tree	Logistic regression	XGBoost	SVM
Accuracy	93.57%	79%	79%	88.4%	91.89%

When comparing our project's model performance to previous work, it is clear that our Logistic Regression model is the most accurate at predicting suicide text. Our algorithm achieved an incredible 93.57% accuracy, surpassing the findings of previous investigations. Chiroma, Liu, and Cocea's best-performing Decision Tree model achieved 79% accuracy, while Jain et al. obtained 79% accuracy with Logistic Regression. Lee and Pak's investigation showed that XGBoost was the most effective, with an accuracy of 88.4%, while Sebastian et al. recorded the best accuracy with SVM, at 91.89%. These comparisons clearly show that our model not only outperforms previous studies, but also sets a new standard for suicide text detection accuracy, emphasizing the robustness and reliability of our approach.

References

- Chiroma, F., Liu, H., & Cocea, M. (n.d.). *TEXT CLASSIFICATION FOR SUICIDE RELATED TWEETS*.
- Jacob Murel Ph.D., E. K. (29 November, 2023). *What is stemming?* Retrieved from IBM: <https://www.ibm.com/topics/stemming>
- Jain , P., Srinivas, K. R., & Vichare, A. (2022). Depression and Suicide Analysis Using Machine Learning and NLP. *Conference Series*, 11. doi:10.1088/1742-6596/2161/1/012034
- Lee, J., & Pak, T.-Y. (2022). Machine learning prediction of suicidal ideation, planning, and attempt among Korean adults: A population-based study. *Elsevier*, 9. doi:10.1016/j.ssmph.2022.101231
- Liu, B. (2012). *Sentiment Analysis and Opinion Mining*. Morgan & Claypool Publishers.
- O'Dea, B., Wan, S., Batterham, P., Caele, A., Paris, C., & Christensen, H. (4, 2015). *Detecting Suicidality on Twitter*.
- Santoso, M. S., Suryadi, J. J., Marchellino, K., Nabilah, G. Z., & Rojali. (2023). A Comparative Analysis of Decision Tree and Support Vector Machine on Suicide Ideation Detection. *8th International Conference on Computer Science and Computational Intelligence (ICCSACI 2023)*, 519-523. doi:<https://doi.org/10.1016/j.procs.2023.10.553>
- Sedano-Capdevila, A., Toledo-Acosta, M., Barrigon, M., Morales-González, E., Torres-Moreno, D., Martínez-Zaldivar, B., . . . Serrano-Marugán, L. (2023). *Text mining methods for the characterisation of suicidal thoughts and behaviour*. Retrieved from <https://www.sciencedirect.com/science/article/pii/S0165178123000434>