PGP AIML Feb 24 A - Group 7

CAPSTONE PROJECT - Interim Report

NATURAL LANGUAGE PROCESSING

INDUSTRIAL SAFETY - NLP BASED CHATBOT

TEAM DETAILS

Mentor: ROHIT GUPTA	
NAVEEN JOTHI	naveenjothi040@gmail.com
MOHIT JAIN	mohit_jain@yahoo.com
RAHUL PATERIA	rahulpateriya121@gmail.com
AKASH KUMAR	akki.vik3@gmail.com
SOHIT SINGH	Singh.sohit46@gmail.com
PRATIK PATIL	pppratikpatil723@gmail.com

PROBLEM STATEMENT

The database comes from one of the biggest industries in Brazil and in the world. It is an urgent need for industries/companies around the globe to understand why employees still suffer some injuries/accidents in plants. Sometimes they also die in such an environment.

DATA DESCRIPTION

This database is basically records of accidents from 12 different plants in 03 different countries where every line in the data is an occurrence of an accident.

Columns description:

- Data: timestamp or time/date information
- Countries: which country the accident occurred (anonymised)
- Local: the city where the manufacturing plant is located (anonymised)
- Industry sector: which sector the plant belongs to
- Accident level: from I to VI, it registers how severe was the accident (I means not severe but VI means very severe)
- Potential Accident Level: Depending on the Accident Level, the database also registers how severe the accident could have been (due to other factors
- involved in the accident)
- Genre: if the person is male of female
- Employee or Third Party: if the injured person is an employee or a third party
- Critical Risk: some description of the risk involved in the accident
- Description: Detailed description of how the accident happened.

PROJECT OBJECTIVE

Design a ML/DL based chatbot utility which can help the professionals to highlight the safety risk as per the incident description.

EXPLORATORY DATA ANALYSIS

INITIAL REVIEW

After importing the necessary libraries and loading the data, we analyzed the data's structure and significance, taking appropriate actions based on our observations.

```
[ ] data.head()
```

	Unnamed: 0	Data	Countries	Local	Industry Sector	Accident Level	Potential Accident Level	Genre	Employee or Third Party	Critical Risk	Description
0	0	2016-01-01	Country_01	Local_01	Mining	I	IV	Male	Third Party	Pressed	While removing the drill rod of the Jumbo 08 f...
1	1	2016-01-02	Country_02	Local_02	Mining	I	IV	Male	Employee	Pressurized Systems	During the activation of a sodium sulphide pum...
2	2	2016-01-06	Country_01	Local_03	Mining	I	III	Male	Third Party (Remote)	Manual Tools	In the sub-station MILPO located at level +170...
3	3	2016-01-08	Country_01	Local_04	Mining	I	I	Male	Third Party	Others	Being 9:45 am. approximately in the Nv. 1880 C...
4	4	2016-01-10	Country_01	Local_04	Mining	IV	IV	Male	Third Party	Others	Approximately at 11:45 a.m. in circumstances t...

Next steps: Generate code with data | View recommended plots | New interactive sheet

```
[ ] data.shape
```

```
(425, 11)
```

Insights gained on dataset:

The dataset is relatively small (425 rows x 11 columns) but contains relevant information.

It is important to note that some components of the dataset were anonymized to conceal the names and locations of the facilities.

ANALYSING AND HANDLING MISSING/NULL VALUES

```
# Checking for missing values
print(data.isnull().sum())
```

```
Unnamed: 0                0
Data                      0
Countries                 0
Local                     0
Industry Sector           0
Accident Level            0
Potential Accident Level  0
Genre                     0
Employee or Third Party   0
Critical Risk             0
Description               0
dtype: int64
```

```
[ ]  # Drop unnecessary columns (Unnamed: 0).
     data_cleaned = data.drop(columns=["Unnamed: 0"])
```

There are no missing values in the dataset. An unnecessary column named "Unnamed" was removed due to the lack of related metadata, as it added no value to the analysis.

HANDLING DUPLICATES AND OTHER DATA CLEANING STEPS

We observe duplicate records in the given dataset, hence we drop the duplicates.

Initial dataset had 425 rows as observed in the previous step, after dropping 7 duplicate rows, we are left with 418 rows.

```
[ ]  # striping whitespace and set to consistent case
     categorical_columns = ["Countries", "Local", "Industry Sector", "Accident Level",
                            "Potential Accident Level", "Genre", "Employee or Third Party", "Critical Risk"]
     data_cleaned[categorical_columns] = data_cleaned[categorical_columns].apply(lambda x: x.str.strip().str.title())
```

```
[ ]  # Drop duplicate rows if any
     data_cleaned = data_cleaned.drop_duplicates()
```

```
[ ]  data_cleaned.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 418 entries, 0 to 424
Data columns (total 10 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Data                    418 non-null    datetime64[ns]
 1   Countries               418 non-null    object
 2   Local                   418 non-null    object
 3   Industry Sector         418 non-null    object
 4   Accident Level          418 non-null    object
 5   Potential Accident Level  418 non-null  object
 6   Genre                   418 non-null    object
 7   Employee or Third Party  418 non-null   object
 8   Critical Risk           418 non-null    object
 9   Description             418 non-null    object
dtypes: datetime64[ns](1), object(9)
memory usage: 35.9+ KB
```

We noticed that the dataset contains two column names, "Data" and "Genre," in Portuguese. To ensure consistency in the language across the dataset, we are converting these columns to English.

```
In [15]: # Rename multiple columns
         data_cleaned = data_cleaned.rename(columns={'Data': 'Date', 'Genre': 'Gender'})
```

```
In [16]: data_cleaned.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 418 entries, 0 to 424
Data columns (total 10 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Date                      418 non-null    datetime64[ns]
 1   Countries                 418 non-null    object
 2   Local                     418 non-null    object
 3   Industry Sector           418 non-null    object
 4   Accident Level            418 non-null    object
 5   Potential Accident Level  418 non-null    object
 6   Gender                    418 non-null    object
 7   Employee or Third Party   418 non-null    object
 8   Critical Risk             418 non-null    object
 9   Description               418 non-null    object
dtypes: datetime64[ns](1), object(9)
memory usage: 35.9+ KB
```

We are extracting the day, month, and year from the date column in order to conduct exploratory data analysis (EDA) in the subsequent steps. This will help us understand the distribution and identify patterns related to the days and months when incidents occurred.

```
# Ensure the 'Date' column is in datetime format
# data_cleaned['Date'] = pd.to_datetime(data_cleaned['Date'])

# Extract year, month, and day into separate columns
data_cleaned['Year'] = data_cleaned['Date'].dt.year
data_cleaned['Month'] = data_cleaned['Date'].dt.month
data_cleaned['Day'] = data_cleaned['Date'].dt.day

# Display the updated DataFrame
print(data_cleaned.head())
```

Since we have already extracted the date features into individual columns, the original "Date" feature is no longer necessary and will be dropped.

```
In [19]: data_cleaned.drop(columns=['Date'], inplace=True)
```

```
In [20]: data_cleaned.head()
```

Out[20]:

	Countries	Local	Industry Sector	Accident Level	Potential Accident Level	Gender	Employee or Third Party	Critical Risk	Description	Year	Month	Day
0	Country_01	Local_01	Mining	I	Iv	Male	Third Party	Pressed	While removing the drill rod of the Jumbo 08 f...	2016	1	1
1	Country_02	Local_02	Mining	I	Iv	Male	Employee	Pressurized Systems	During the activation of a sodium sulphide pum...	2016	1	2
2	Country_01	Local_03	Mining	I	Iii	Male	Third Party (Remote)	Manual Tools	In the sub-station MILPO located at level +170...	2016	1	6
3	Country_01	Local_04	Mining	I	I	Male	Third Party	Others	Being 9:45 am. approximately in the Nv. 1880 C...	2016	1	8
4	Country_01	Local_04	Mining	Iv	Iv	Male	Third Party	Others	Approximately at 11:45 a.m. in circumstances t...	2016	1	10

Additionally, we observe that the data frame contains 7 rows where only one or two column values differ, while the "Description" column remains the same across these rows. This inconsistency is logically incorrect, so we have decided to drop these rows as well.

```
In [21]: # Dropping the duplicates we detected above.
         data_cleaned.drop_duplicates(subset=['Description'], keep='first', inplace=True)
         print('After removing duplicates the shape of the dataset is:', data_cleaned.shape)

         After removing duplicates the shape of the dataset is: (411, 12)
```

ANALYSING PATTERNS

Analyzing the occurrence patterns across categorical data within the dataset to identify key trends and insights.

```
# Looping through all the columns in the dataframe and checking counts of unique values.
for col in data_cleaned.columns:
    if (col!='Description'):
        print(data_cleaned[col].value_counts())
        print('*'*50)
```

```
Countries
Country_01    245
Country_02    127
Country_03     39
Name: count, dtype: int64
**************************************************
Local
Local_03    87
Local_05    59
Local_01    55
```

```
Accident Level
I      303
Ii      39
Iii     31
Iv      30
V        8
Name: count, dtype: int64
**************************************************
Potential Accident Level
Iv     138
Iii    106
Ii      95
I       43
V       28
Vi       1
Name: count, dtype: int64
**************************************************
Gender
Male      390
Female     21
Name: count, dtype: int64
**************************************************
Employee or Third Party
Third Party             180
Employee                176
Third Party (Remote)     55
Name: count, dtype: int64
**************************************************
```

```
Critical Risk
Others                                  223
Pressed                                  24
Manual Tools                             20
Chemical Substances                      17
Cut                                      14
Venomous Animals                         13
Projection                               12
Bees                                     10
```

```
Month
2     60
4     51
3     50
6     49
1     39
5     38
9     24
7     23
12    22
8     21
10    21
11    13
```

OBSERVATIONS

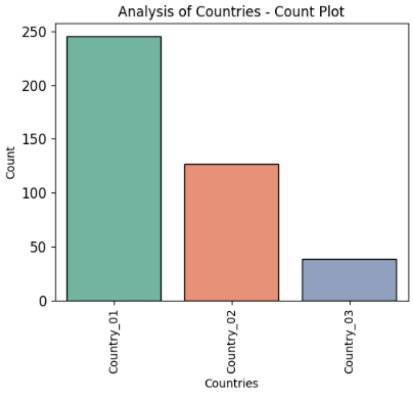
- Most of the incidents (over 50%) occurred in Country_01

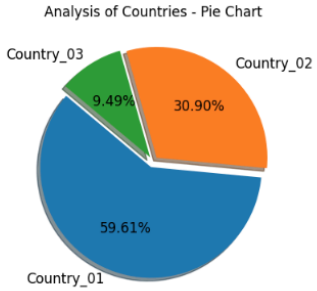
- Most of the incidents from Country_01 has been reported from Local_03 city
- Most incidents reported are least severe (Level I)
- Most incidents occurred are to male which signifies it's a male dominated industry
- "Others" is the largest category (223 incidents), suggesting the need for clearer risk definitions.

VISUALIZATION

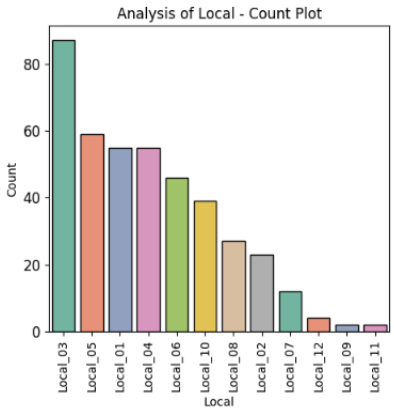
UNIVARIATE ANALYSIS

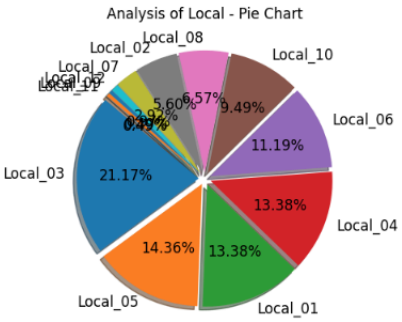
Incident occurrences by countries



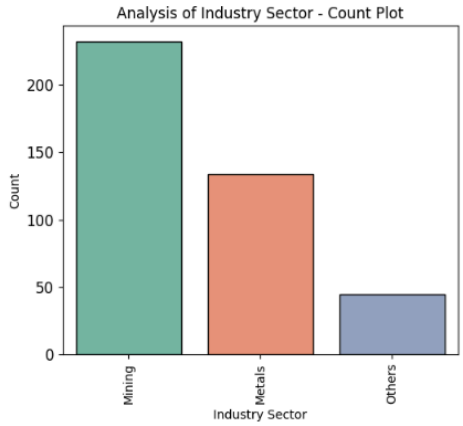


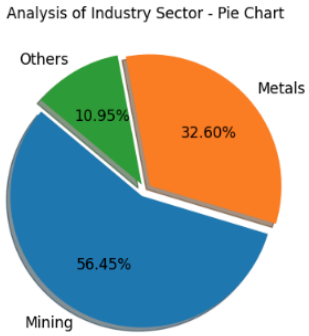
Incident occurrences by local (City)



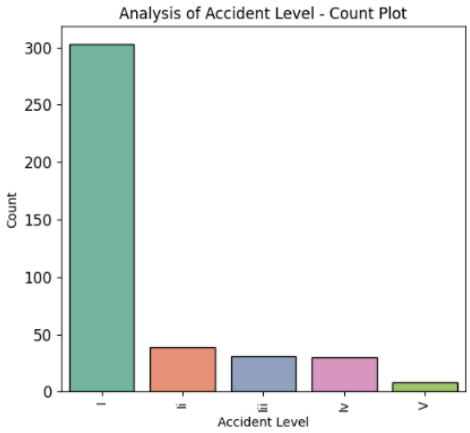


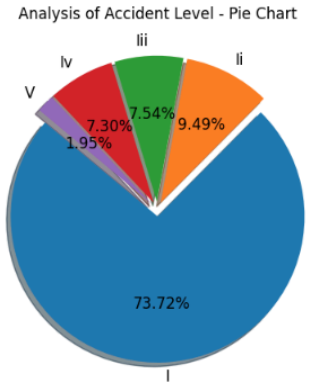
Incident occurrences by industry sector



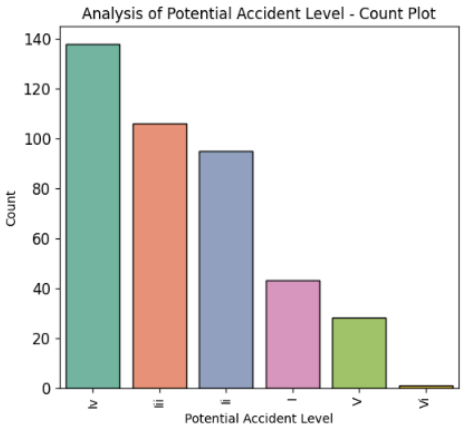


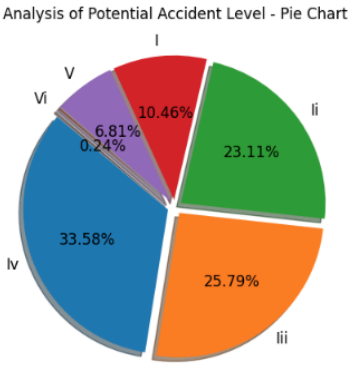
Incident occurrences by accident level



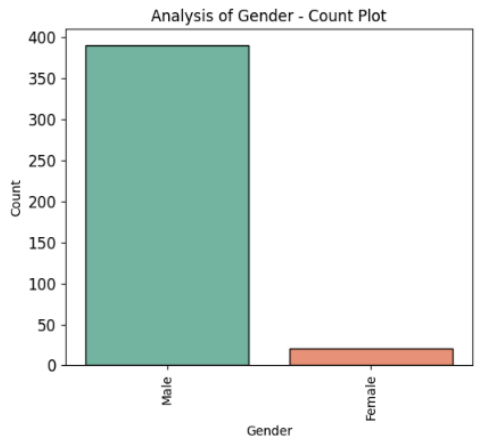


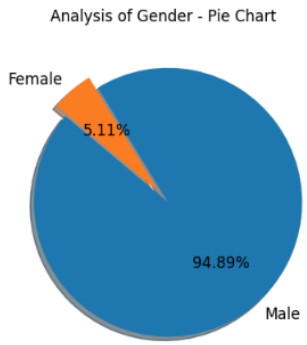
Incident occurrences by potential accident level



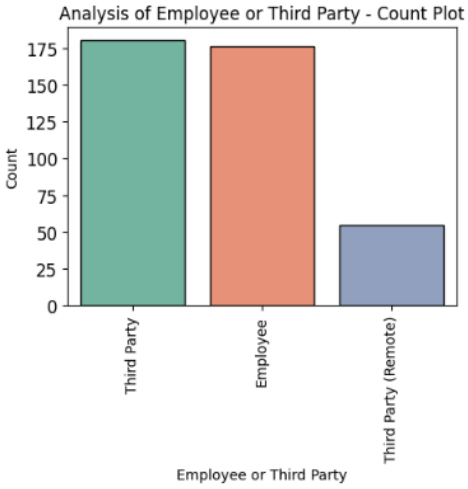


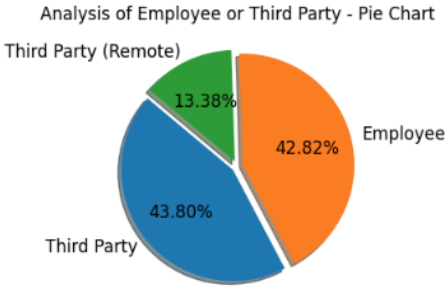
Incident occurrences by gender



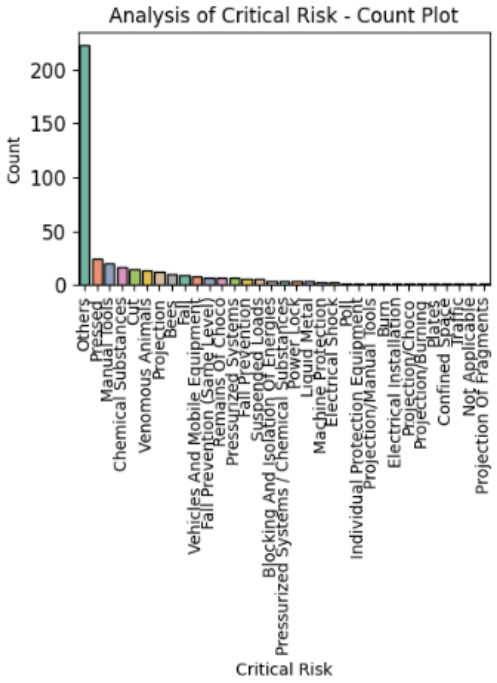


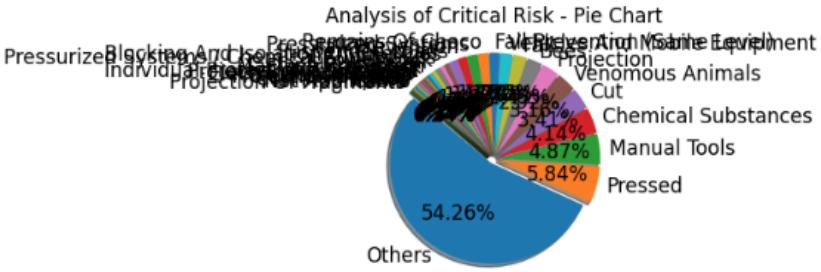
Incident occurrences by employment type





Incident occurrences by critical risk





KEY INSIGHTS - Univariate Analysis

1. Geographical Distribution

- **Countries**:
    - Country_01 accounts for a significant majority of incidents (245), comprising over half the dataset.
    - Country_02 and Country_03 have substantially fewer incidents (127 and 39, respectively).
- **Local Areas**:
    - Local_03 stands out with 87 incidents, making it the most accident-prone area.
    - Local_05, Local_01, and Local_04 follow closely with similar counts, suggesting these locations require focused attention.

2. Industry Sectors

- **Mining** is the leading sector with 232 incidents, emphasizing its high-risk nature.

- **Metals** contribute to 134 incidents, indicating it is also a critical area for safety interventions.
- Other sectors account for a relatively small proportion (45).

3. Accident Severity

- **Accident Level**:
    - Level I accidents dominate the dataset (303 incidents), signifying that minor accidents are the most common.
    - Higher-severity accidents (Levels IV and V) are less frequent but still significant for targeted safety measures.
- **Potential Accident Level**:
    - Potential Level IV accidents (138) and Level III (106) highlight areas of latent high-risk scenarios.

4. Demographics

- **Gender**:
    - Males represent a staggering 95% of incidents, suggesting male-dominated roles in these industries may face greater exposure to risks.
- **Employee vs. Third Party**:
    - Third-party individuals (including remote third parties) are involved in more incidents (235) compared to employees (176), indicating external personnel face considerable safety challenges.

5. Critical Risks

- **Top Risk Factors**:
    - "Others" (223 incidents) may require further investigation to identify underlying risk contributors.
    - Pressed (24), Manual Tools (20), and Chemical Substances (17) are notable specific risks.

6. Temporal Trends

- **Yearly Data**:
    - A majority of accidents occurred in 2016 (278 incidents), compared to 2017 (133), indicating a declining trend.
- **Monthly Data**:
    - February (60) and April (51) saw the highest number of incidents, suggesting seasonality effects or operational peaks.

RECOMMENDATIONS

- Focus safety interventions on Country_01 and Local_03.
- Address critical risks like Pressed, Manual Tools, and Chemical Substances.
- Enhance safety measures in Mining and Metals industries.
- Develop targeted safety programs for males and third-party workers.
- Investigate incident spikes in February and April for potential seasonal or operational causes.most incidents (278).

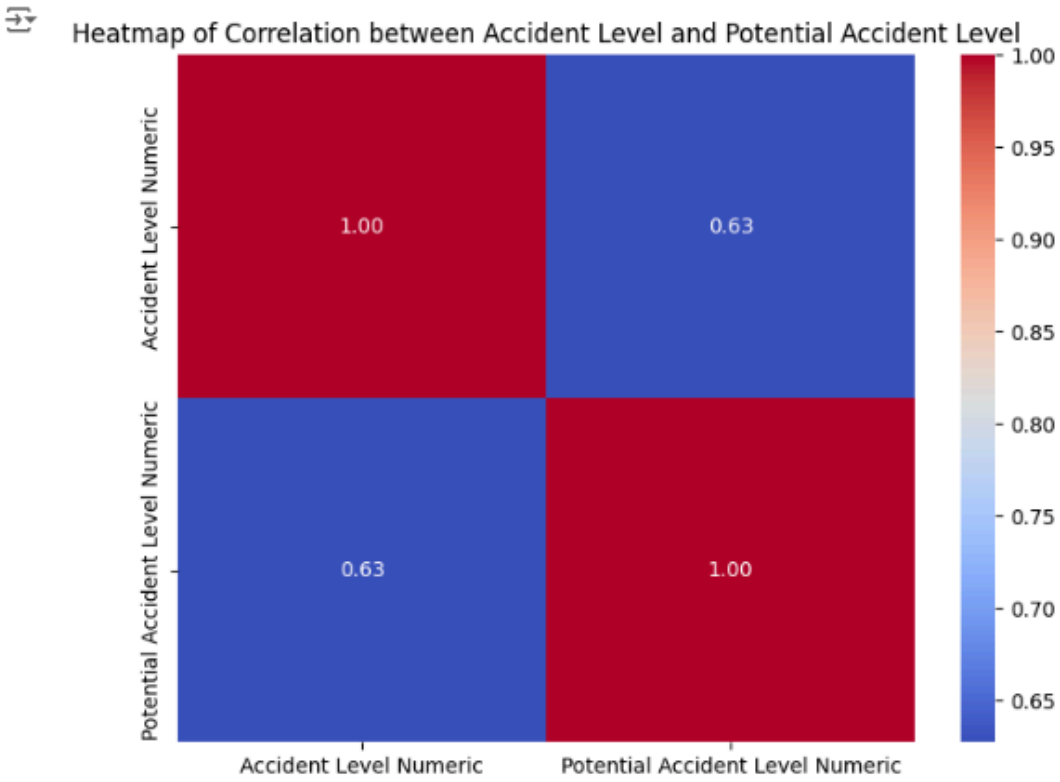
BIVARIATE ANALYSIS

Correlation between 'Accident Level' and 'Potential Accident Level'

```
# Map Accident Level and Potential Accident Level to numerical values for correlation analysis
level_mapping = {'I': 1, 'II': 2, 'III': 3, 'IV': 4, 'V': 5, 'VI': 6}
data_cleaned['Accident Level Numeric'] = data_cleaned['Accident Level'].map(level_mapping)
data_cleaned['Potential Accident Level Numeric'] = data_cleaned['Potential Accident Level'].map(level_mapping)

# Calculate the correlation
correlation = data_cleaned[['Accident Level Numeric', 'Potential Accident Level Numeric']].corr()

# Display the correlation value
correlation_value = correlation.loc['Accident Level Numeric', 'Potential Accident Level Numeric']
correlation_value
```

```
0.6272671299275785
```

```
[91] # Generate a heatmap for the correlation matrix
     plt.figure(figsize=(8, 6))
     sns.heatmap(correlation, annot=True, cmap='coolwarm', fmt='.2f', cbar=True)
     plt.title('Heatmap of Correlation between Accident Level and Potential Accident Level')
     plt.show()
```



A correlation of **0.627** between Accident Level and Potential Accident Level indicates a **moderately strong positive relationship** between these two variables. Here's what this means in context:

Correlation interpretation and implications

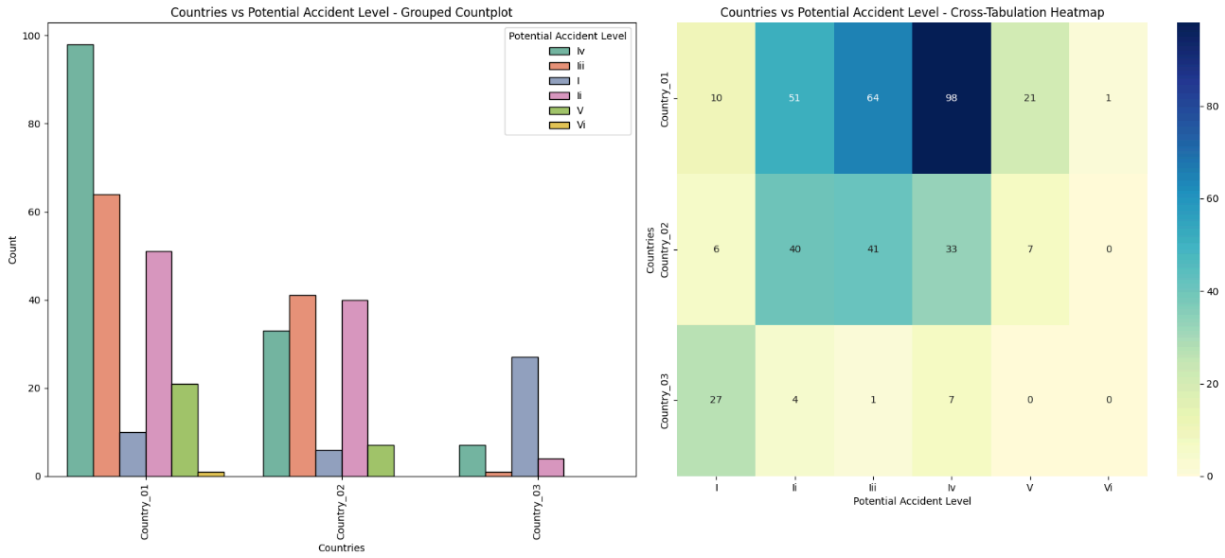
1. **Positive Relationship**:
    - As the **Accident Level** increases (indicating more severe actual accidents), the **Potential Accident Level** also tends to increase (indicating higher potential severity of the accident).
2. **Moderately Strong Correlation**:

- The value of **0.627** suggests that while there is a significant relationship, it is not perfect. Other factors may also influence the Potential Accident Level, apart from the Accident Level.

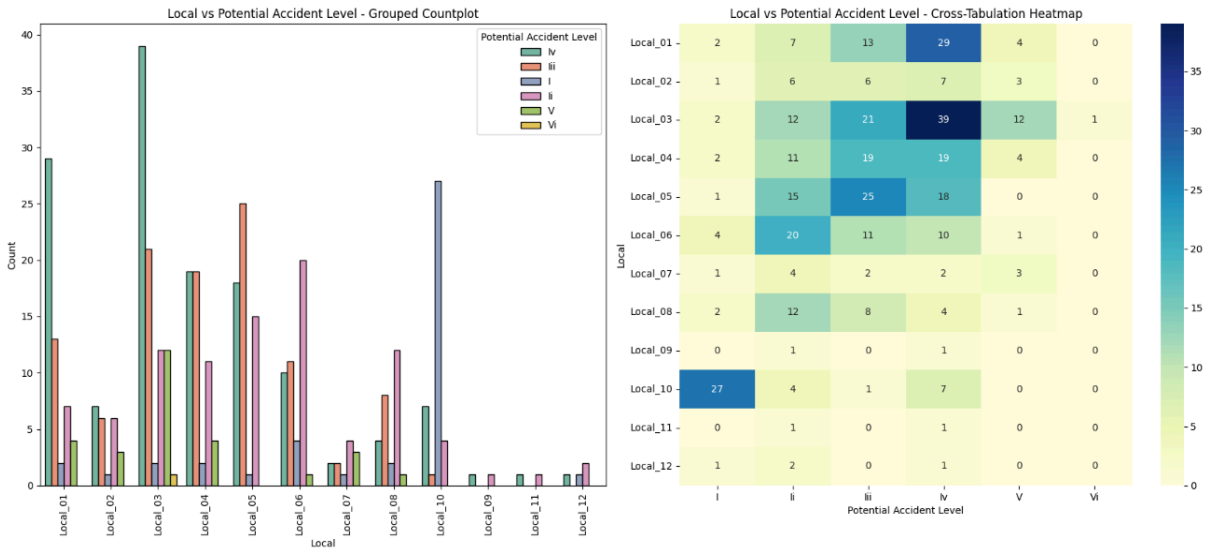
3. **Practical Implication**:
   - This correlation implies that high-severity accidents are often associated with high potential risks, indicating a need to prioritize preventive measures for incidents with high potential severity to avoid severe actual outcomes.

Selecting the '**Potential Accident Level**' as the **target variable** performing the bivariate analysis against all the other categorical columns.

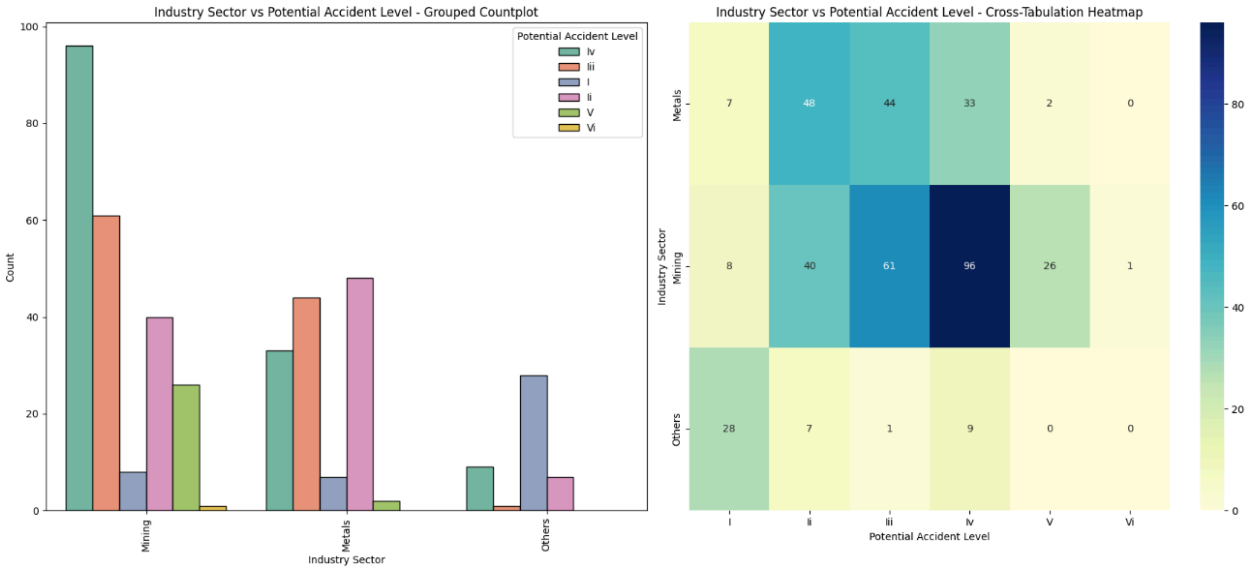
'Potential Accident Level' by 'Countries':



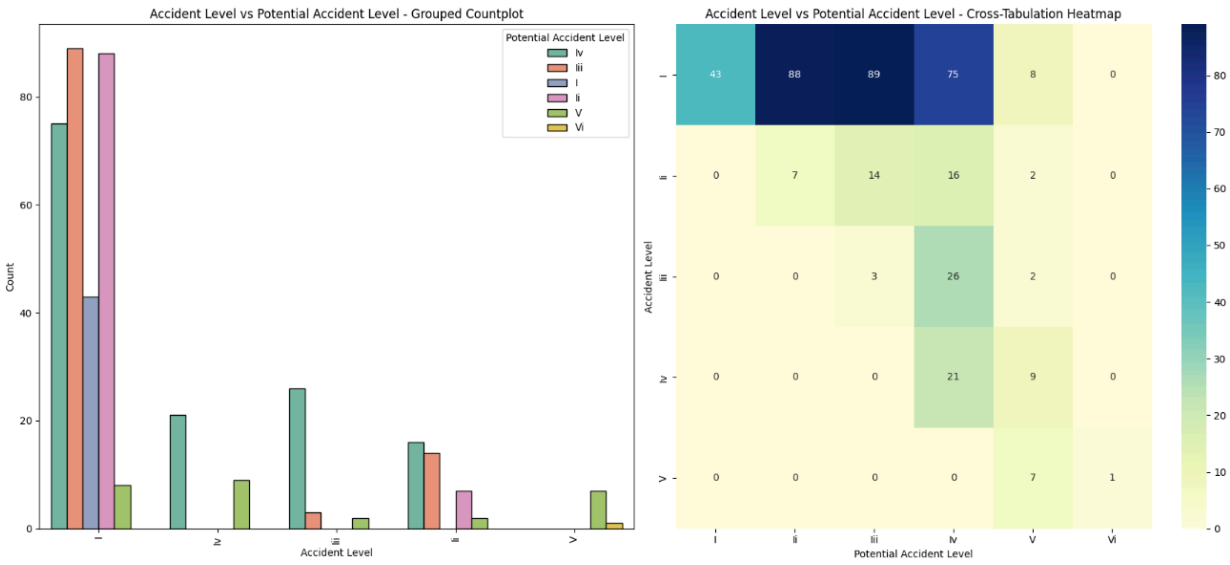
‘Potential Accident Level’ by ‘Local’ (City):



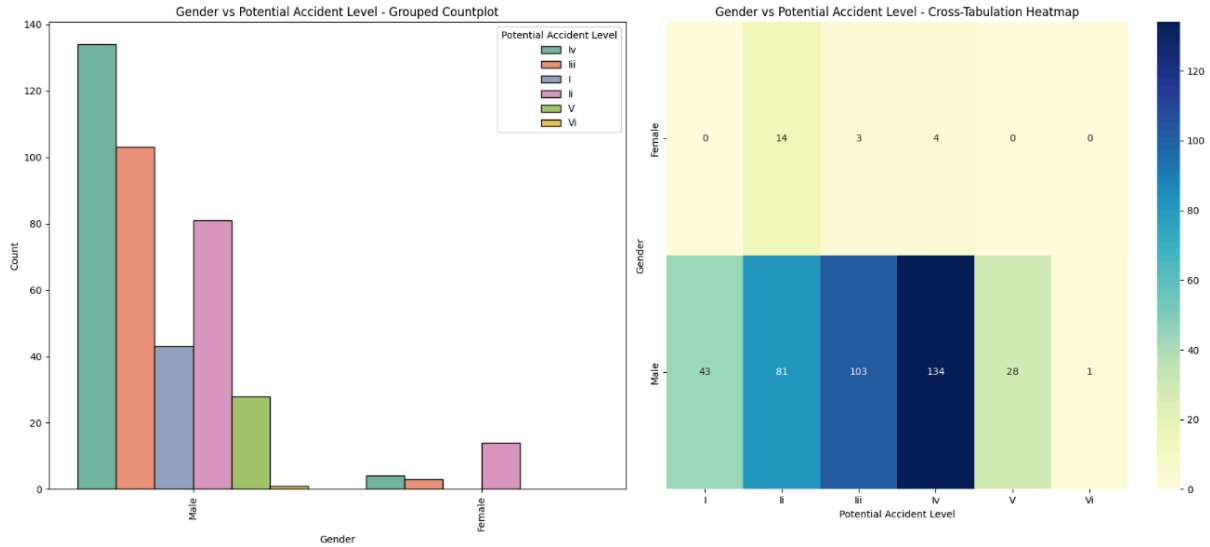
‘Potential Accident Level’ by ‘Industry Sector’:



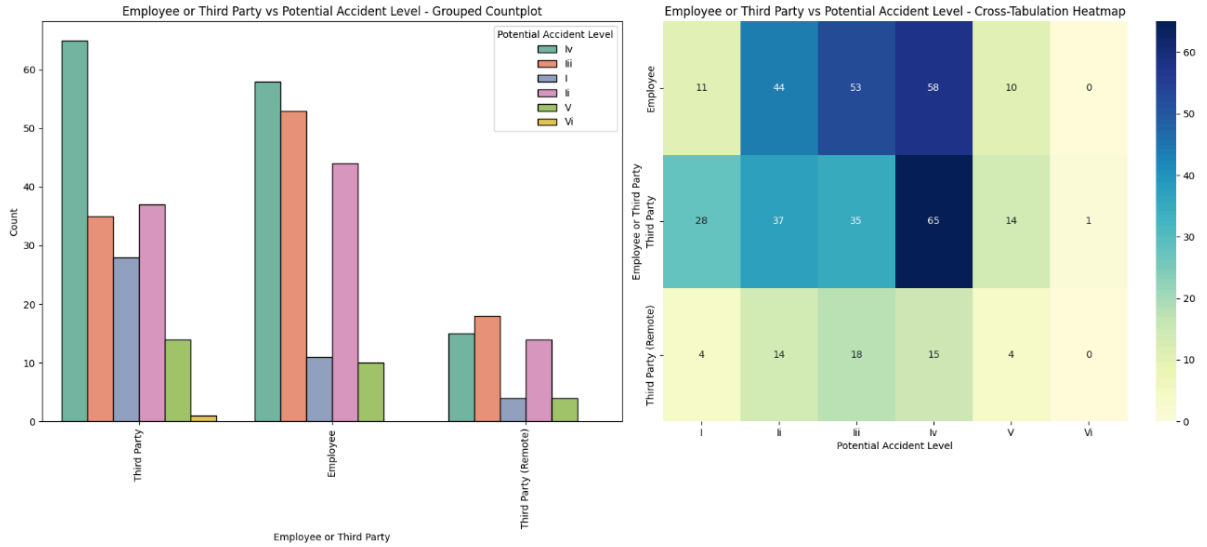
‘Potential Accident Level’ by ‘Accident Level’:



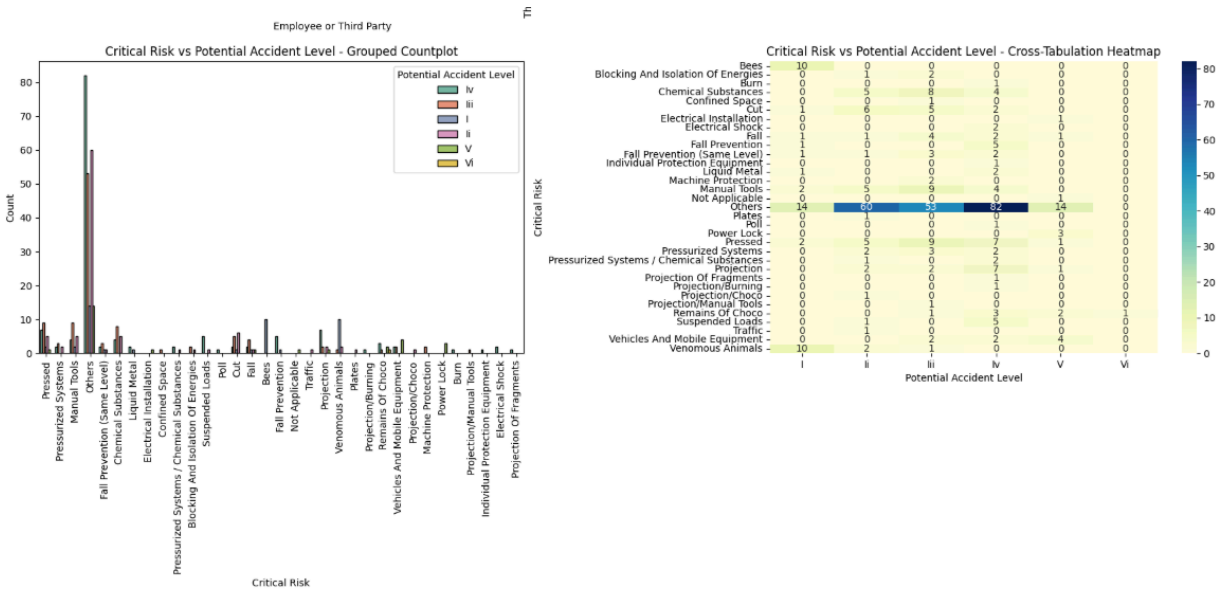
‘Potential Accident Level’ by ‘Gender’:



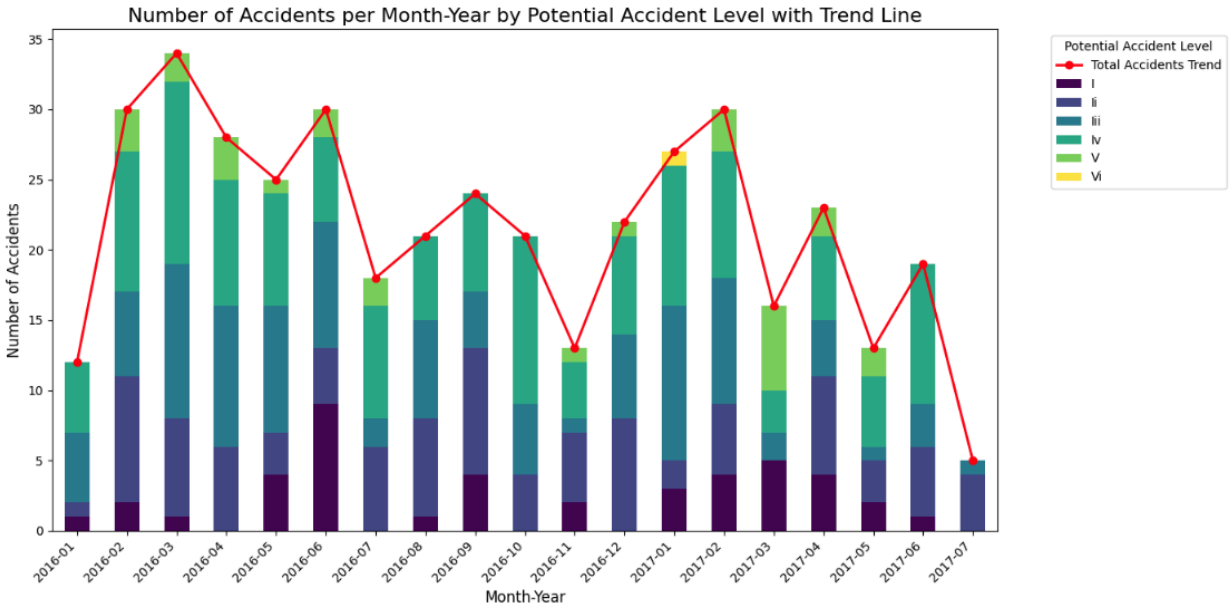
‘Potential Accident Level’ by Employment Type:



‘Potential Accident Level’ by ‘Critical Risk’:



Trend of accidents by year-month by ‘Potential Accident Level’



Level IV Dominance: Level IV potential accidents consistently dominate across all months, highlighting the need to focus on high-severity risk mitigation.

Fluctuating Monthly Accidents: Peaks in total accidents occur in February-March 2016 and January-February 2017, suggesting periodic or seasonal factors affecting incident rates.

Decline in Mid-2017: A clear decline in total accidents is observed in mid-2017, potentially reflecting effective safety measures or changes in activity levels.

Low but Present Extreme Risks: Levels V and VI (extreme risks) are rare but appear sporadically, underscoring the importance of preparedness for severe incidents.

Consistent Moderate Risks: Lower-potential severity levels (I, II, III) remain consistently present, indicating the need for continuous monitoring and interventions for moderate risks.

KEY INSIGHTS - Bivariate Analysis

Potential Accident Level by Industry Sector

- **Level IV** potential accidents dominate across all industries (141 incidents), followed by **Level III** (106) and **Level II** (95).
- **Mining** sees the highest count of potential accidents at all levels, reflecting its inherent high-risk nature.
- **Metals** and **Others** have relatively fewer incidents but still contribute to higher-level potential accidents.

Potential Accident Level by Year

- **2016** recorded the majority of higher potential severity levels, with Level IV being the most frequent.
- A slight decline in high-severity potential incidents is observed in 2017.

Potential Accident Level by Critical Risk

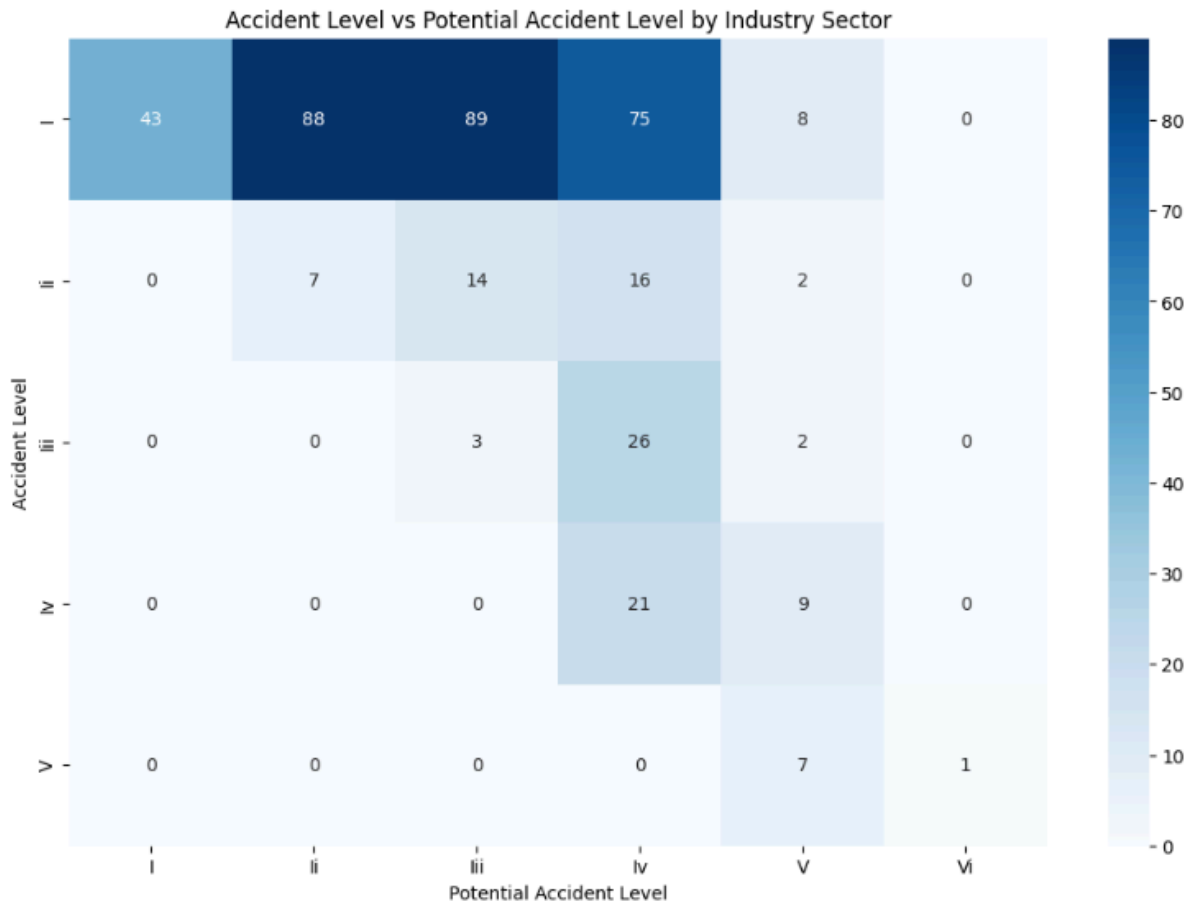
- **Level IV** potential accidents are spread across various risks, with "Others" being the most common, signalling ambiguous or uncategorised hazards.
- Risks like **Pressed** and **Manual Tools** are associated with higher potential severity levels (III and IV).
- **Chemical Substances** frequently appear at moderate potential levels (II and III).

General Observations

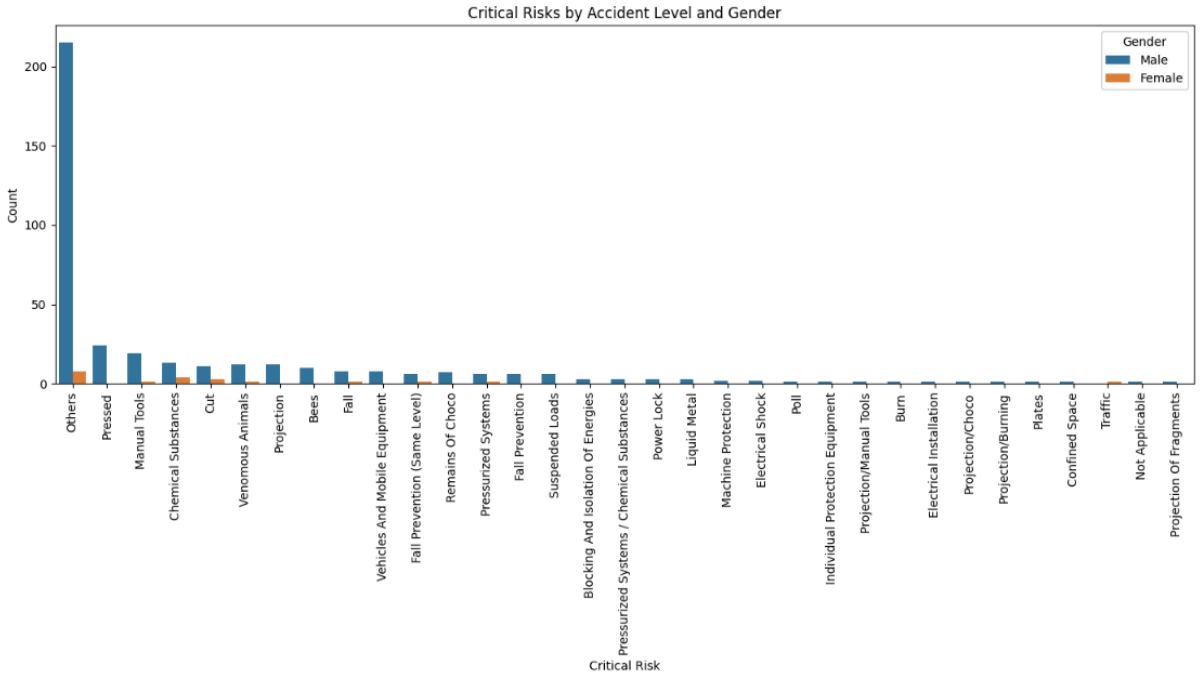
- **Mining** industry and **2016** have the most significant contributions to higher potential accident levels.
- The need for clearer categorisation of critical risks, especially those labelled as "Others," is evident.
- High-severity risks demand targeted safety interventions to reduce potential impact.

MULTIVARIATE ANALYSIS

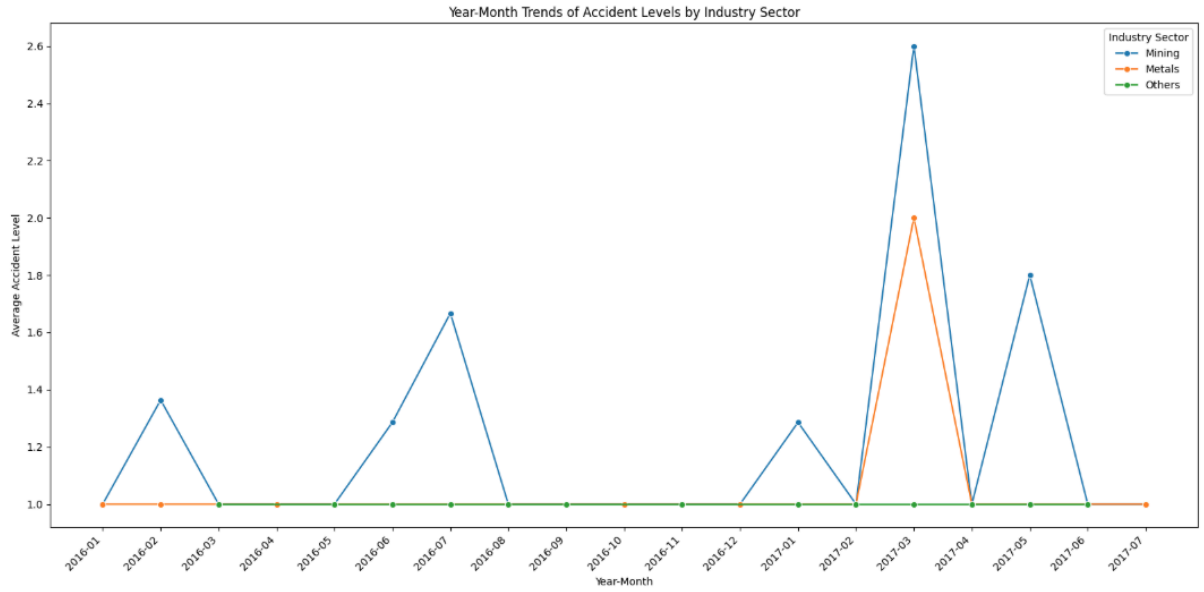
Accident Level vs Potential Accident Level by Industry Sector

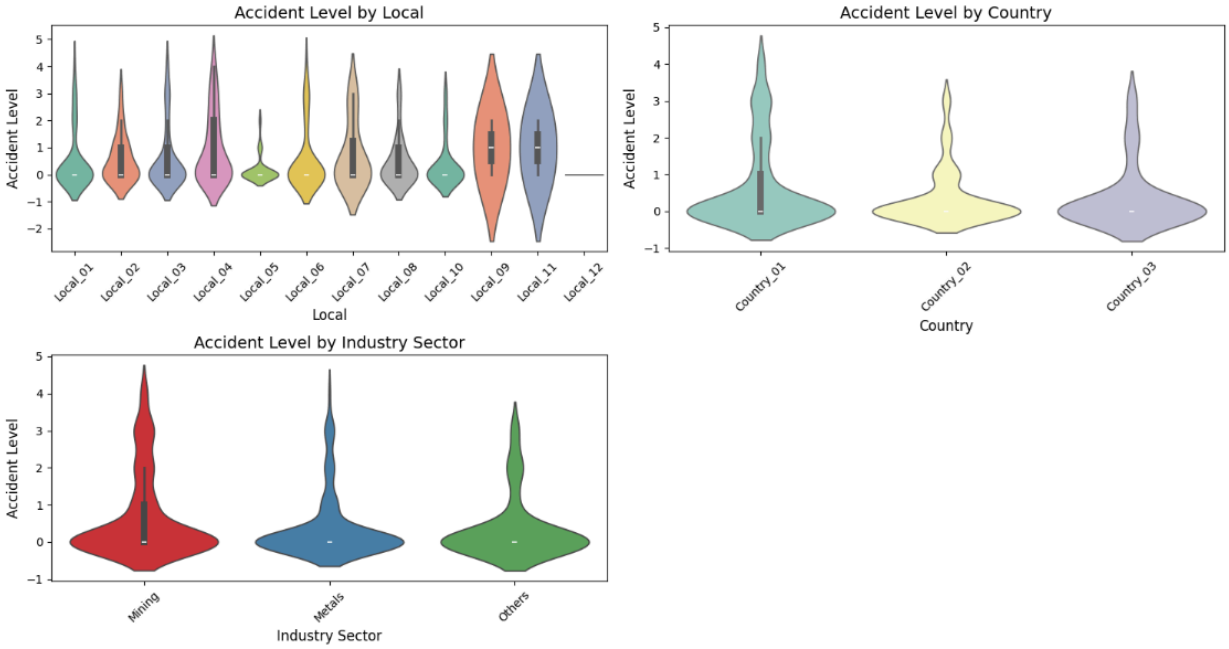


Critical Risks by Accident Level and Gender

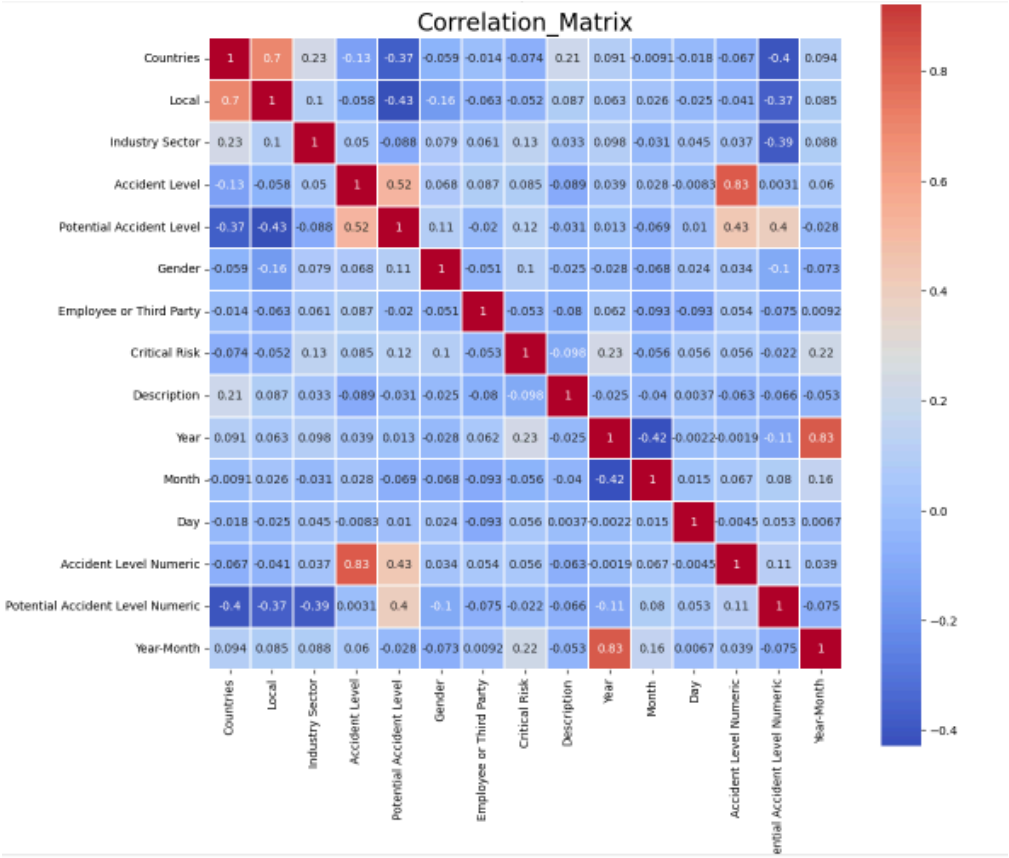


Year-Month Trends of Accident Levels by Industry Sector





Correlation Matrix



KEY INSIGHTS - Multivariate Analysis

Accident Level vs. Potential Accident Level by Industry Sector

- The violin plot reveals a broader spread of accident levels in the Mining sector, with a skew toward more severe incidents. This highlights the variability and the potential for high-severity accidents within this sector.
- Metals demonstrate a narrower distribution of accident levels, indicating fewer high-severity incidents and suggesting better control mechanisms.
- The "Others" category shows moderate variability, with some potential outliers indicating unusual incidents requiring further investigation.

Accident Level by Local and Country

- Significant variability is observed across different locales, with some exhibiting tightly concentrated distributions around lower accident levels and others showing broad distributions with higher frequencies of severe incidents.
- At the country level, accident distributions are more consistent, with fewer extreme values, indicating country-specific safety standards may play a role in moderating risk variability.

Critical Risks vs. Accident Level by Gender

- The violin plot analysis confirms that males are more frequently involved in severe incidents, as the spread of accident levels for males is broader and skewed toward higher severities.
- For females, the distribution of accident levels is narrower, suggesting fewer incidents overall, though certain risk types like "Others" still require investigation to ensure equitable safety measures.

Year-Month Trends of Accident Levels by Industry Sector

- Temporal trends corroborated by the violin plot indicate that accident levels in the Mining sector are seasonally influenced, with higher peaks during specific months. The broad variability during these peaks suggests fluctuating risk factors, possibly linked to operational cycles or environmental conditions.

- Metals exhibit steadier trends with a narrow spread, indicating a more consistent risk profile.

General Observations

- The Mining sector stands out for its high variability in accident levels, with a wider range of severity compared to other sectors. This reinforces its designation as a high-risk industry.
- The "Others" risk category displays broader and often ambiguous distributions, masking the underlying hazards that need clearer categorization.
- Gender-specific differences in accident distributions emphasize the need for tailored safety interventions to address distinct risk exposures for males and females.

RECOMMENDATIONS

- **Enhance Safety Measures in Mining**
  Address the high variability and severity of accidents in the Mining sector by implementing advanced risk management tools, particularly during seasonal peaks identified in the temporal trends.
- **Investigate and Categorize "Others" Risks**
  Refine the "Others" risk category to uncover specific hazards, enabling more targeted safety measures and reducing ambiguity in accident reporting.
- **Implement Gender-Specific Safety Programs**
  Tailor training and safety protocols for male workers exposed to high-risk activities (e.g., manual tools, pressing equipment) and ensure equitable measures for female workers to address their specific risk profiles.
- **Standardize Local Safety Practices**
  Reduce inconsistencies between locals with varying accident distributions by standardizing safety practices and addressing outliers in high-risk locations.
- **Strengthen Seasonal Safety Protocols**
  Introduce heightened safety measures in Mining during peak accident months, with resources allocated to manage higher severity incidents effectively.

- **Continuous Monitoring and Data-Driven Adjustments**
  Regularly analyze accident distributions to track shifts in central tendencies, variability, and outliers, ensuring safety strategies remain responsive to evolving risks.
- **Refine Safety in Metals Sector**
  Conduct regular safety audits in the Metals sector to maintain its consistent risk profile and investigate any emerging outliers to prevent escalation.
- **Promote Consistent Risk Management Across Countries**
  Leverage insights from the relatively uniform accident distributions at the country level to reinforce effective safety standards across all regions.

NLP - PREPROCESSING

Before proceeding with model building, it is essential to pre-process the description column in the dataset to ensure the text is clean and structured for analysis. Key pre-processing steps include text cleaning, tokenization, stopword removal, stemming or lemmatization, handling punctuation, translations, and spell checks. The specific steps chosen for this dataset are detailed in the following section.

TRANSLATION

Given that the dataset primarily originates from Brazil and may also include data from other regions, we opted to perform translation from various languages to English. This ensures consistency and standardization, making the data more suitable for analysis and model training.

```python
from googletrans import Translator

# Initialize the translator
translator = Translator()

# Function to automatically detect source language and translate to English
def translate_text_auto_detect(text):
    # Check for None or empty string
    if text is None or text == '':
        return ''  # or any other appropriate default value

    try:
        # Auto-detect the source language and translate to English
        translated = translator.translate(text, dest='en')
        return translated.text  # Directly access the translated text
    except (AttributeError, TypeError):  # Handle potential errors
        print(f"Translation error for text: {text}")
        return text  # Return original text

# Apply translation to the 'Description' column
data_cleaned['Translated_Description'] = data_cleaned['Description'].apply(translate_text_auto_detect)

# Display the cleaned data
print(data_cleaned[['Description', 'Translated_Description']])
```

```
                                           Description  \
0    While removing the drill rod of the Jumbo 08 f...
1    During the activation of a sodium sulphide pum...
2    In the sub-station MILPO located at level +170...
3    Being 9:45 am. approximately in the Nv. 1880 C...
4    Approximately at 11:45 a.m. in circumstances t...
..                                                 ...
420  Being approximately 5:00 a.m. approximately, w...
421  The collaborator moved from the infrastructure...
422  During the environmental monitoring activity i...
423  The Employee performed the activity of strippi...
424  At 10:00 a.m., when the assistant cleaned the ...

                              Translated_Description
0    While removing the drill rod of the Jumbo 08 f...
1    During the activation of a sodium sulphide pum...
2    In the sub-station MILPO located at level +170...
3    Being 9:45 am. approximately in the Nv. 1880 C...
4    Approximately at 11:45 a.m. in circumstances t...
..                                                 ...
420  Being approximately 5:00 a.m. approximately, w...
421  The collaborator moved from the infrastructure...
422  During the environmental monitoring activity i...
423  The Employee performed the activity of strippi...
424  At 10:00 a.m., when the assistant cleaned the ...

[411 rows x 2 columns]
```

CLEANUP STEPS

Handle Missing Text: If the input text is empty or None, return an empty string.

Convert to Lowercase: The text is converted to lowercase to standardize it.

Tokenization: The text is split into individual words (tokens).

Remove Punctuation: Only alphabetic characters are retained, removing any punctuation.

Spell Checking: Each word is spell-checked and corrected using the autocorrect function.

Remove Stopwords: Common stopwords (e.g., "the", "is") are removed to focus on important words.

Lemmatization: Words are reduced to their base form (e.g., "running" to "run"). We choose lemmatization over stemming as it gives better accuracy for model training.

Whitespace Removal: Extra whitespaces are removed, and the tokens are joined back into a clean string.

```python
# Initialize the lemmatizer, stopwords, and spell checker
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
spell = Speller()

# Function to perform all NLP preprocessing steps including spellcheck
def nlp_preprocess(text):
    # Step 1: Check for missing or empty text and return empty string if missing
    if not text or text is None:
        return ''

    # Step 2: Lowercase the text
    text = text.lower()

    # Step 3: Tokenize the text
    tokens = word_tokenize(text)

    # Step 4: Remove punctuation (keep only alphabetic characters)
    tokens = [word for word in tokens if word.isalpha()]

    # Step 5: Spell check (correcting each word using autocorrect)
    tokens = [spell(word) for word in tokens]

    # Step 6: Remove stopwords
    tokens = [word for word in tokens if word not in stop_words]

    # Step 7: Lemmatize the words
    lemmatized_tokens = [lemmatizer.lemmatize(word) for word in tokens]

    # Step 8: Optional - Remove extra whitespaces
    processed_text = " ".join(lemmatized_tokens)

    return processed_text
```

```
[48] # Apply NLP preprocessing to the 'Translated_Description' column
     data_cleaned['Cleaned_Description'] = data_cleaned['Translated_Description'].apply(nlp_preprocess)

     # Display the cleaned data
     print(data_cleaned[['Translated_Description', 'Cleaned_Description']])
```

```
                              Translated_Description  \
0    While removing the drill rod of the Jumbo 08 f...
1    During the activation of a sodium sulphide pum...
2    In the sub-station MILPO located at level +170...
3    Being 9:45 am. approximately in the Nv. 1880 C...
4    Approximately at 11:45 a.m. in circumstances t...
..                                                 ...
420  Being approximately 5:00 a.m. approximately, w...
421  The collaborator moved from the infrastructure...
422  During the environmental monitoring activity i...
423  The Employee performed the activity of strippi...
424  At 10:00 a.m., when the assistant cleaned the ...

                                  Cleaned_Description
0    removing drill rod jumbo maintenance superviso...
1    activation sodium sulfide pump piping coupled ...
2    mile located level collaborator excavation wor...
3    approximately nv personnel begin task unlockin...
4    approximately circumstance mechanic anthony gr...
..                                                 ...
420  approximately approximately lifting kelly hq t...
421  collaborator moved infrastructure office julio...
422  environmental monitoring activity area employe...
423  employee performed activity stripping cathode ...
424  assistant cleaned floor module e central camp ...

[411 rows x 2 columns]
```

OBSERVATIONS

- The cleaned descriptions show that functional words like "the," "a," and "in" have been removed, making the text more concise and focused.
- Text is converted to lowercase, ensuring uniformity and treating capitalized and non-capitalized words as the same.
- Punctuation marks have been eliminated, simplifying the text for easier processing in NLP tasks.
- Non-alphabetic characters are removed, and words are tokenized, keeping only the relevant terms.
- Spelling errors are corrected, such as changing "sulphide" to "sulfide," ensuring consistent terminology.
- Lemmatization has reduced words to their base forms, improving consistency and preparing the text for further analysis.

VISUALIZING THE DATA (NLP)

WORD FREQUENCY DISTRIBUTION

```
from collections import Counter
import matplotlib.pyplot as plt
from wordcloud import WordCloud

# Create a word cloud to visualize the most frequent words
wordcloud = WordCloud(width=800, height=400, max_words=100).generate(' '.join(data_cleaned['Cleaned_Description']))
plt.figure(figsize=(10,6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

OBSERVATIONS

Dominant Keywords:

- Words like "employee," "activity," "moment," "operator," and "assistant" appear prominently, indicating frequent mentions in the dataset.
- These terms likely reflect the primary subjects or roles associated with incidents or actions.

Focus on Actions and Context:

- Words such as "causing," "hitting," "injury," and "performing" suggest the dataset is related to workplace incidents, with emphasis on actions leading to specific outcomes.
- Context-related terms like "left hand," "equipment," and "floor" highlight common areas or objects associated with these incidents.

Potential Risk Areas:

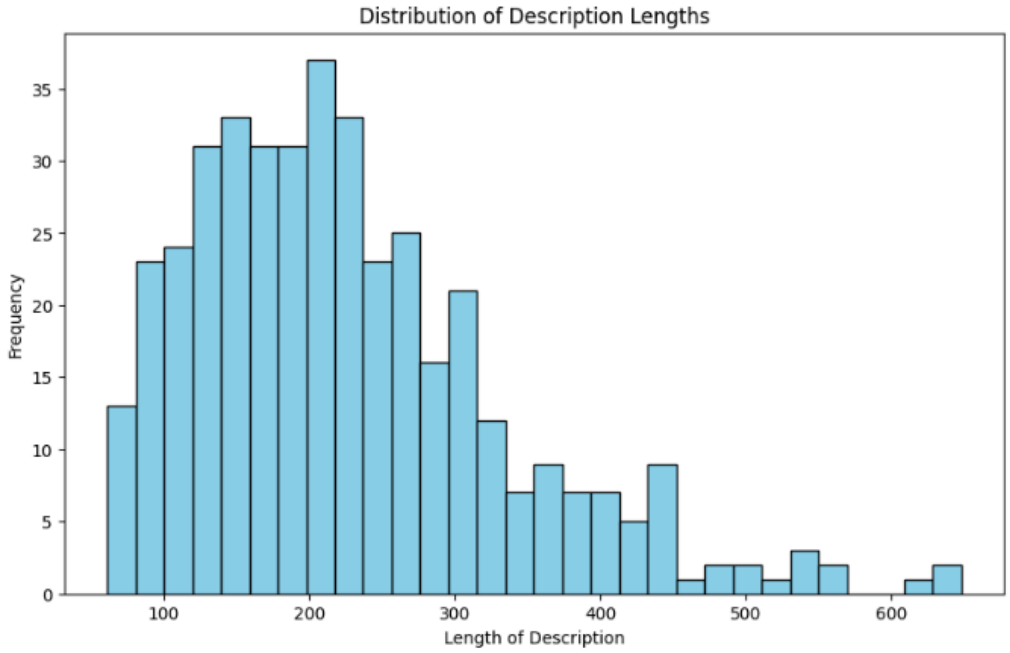
- Terms like "injury," "drill," "equipment," and "cleaning" point toward tasks or tools potentially associated with workplace hazards.
- The frequent mention of "causing" and "accident" suggests an analysis of causes and effects within the dataset.

Relevance of Specific Body Parts:

- "Left hand" and "right hand" are highlighted, suggesting a focus on injuries involving hands, which may represent a high-risk area in the workplace.

DISTRIBUTION OF TEXT LENGTH

```
In [51]: # Add a column for text length
data_cleaned['Description_Length'] = data_cleaned['Cleaned_Description'].apply(len)

# Plot the distribution of text length
plt.figure(figsize=(10,6))
plt.hist(data_cleaned['Description_Length'], bins=30, color='skyblue', edgecolor='black')
plt.title('Distribution of Description Lengths')
plt.xlabel('Length of Description')
plt.ylabel('Frequency')
plt.show()
```



OBSERVATIONS

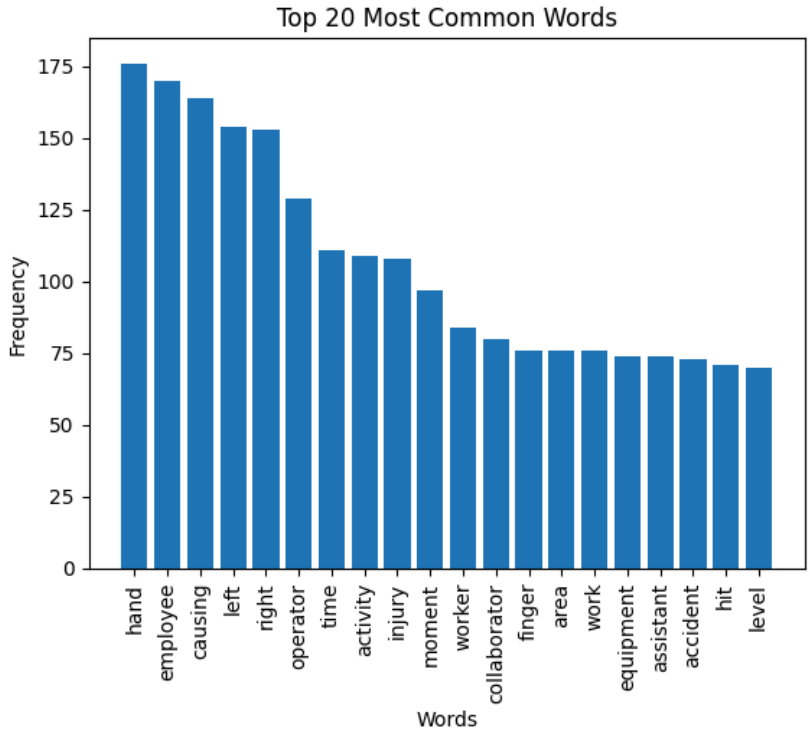
- **Right-Skewed Distribution**: Most descriptions are short, with fewer being much longer.
- **Frequent Length Range**: Descriptions mostly fall between 100-250 characters, peaking around 150-200.
- **Long Descriptions**: A small number exceed 300 characters, with some surpassing 600, likely indicating more complex incidents.
- **Mode**: The most common description length is slightly under 200 characters.

RECOMMENDATIONS

- **Modeling Considerations**: The prevalence of short descriptions may bias NLP models, requiring strategies for truncation or padding.
- **Text Augmentation**: Augmenting long descriptions or summarizing lengthy ones can address the imbalance.
- **Data Quality Check**: Review long outliers for redundancy or over-detailing and trim for consistency.
- **Segmentation by Length**: Segment descriptions by length to analyze any correlation with accident severity.

TOP N MOST COMMON WORDS

```
# Tokenize and remove stopwords
from nltk.tokenize import word_tokenize
import nltk
nltk.download('punkt')

# Function to clean text and remove stopwords
def clean_text(text):
    tokens = word_tokenize(text.lower())
    return [word for word in tokens if word.isalpha() and word not in ENGLISH_STOP_WORDS]

# Apply the function to the 'Description' column
data_cleaned['Cleaned_Description_Tokens'] = data_cleaned['Cleaned_Description'].apply(clean_text)

# Find the most common words
all_words = [word for text in data_cleaned['Cleaned_Description_Tokens'] for word in text]
word_counts = Counter(all_words)
common_words = word_counts.most_common(20)

# Plot the top 20 most common words
words, counts = zip(*common_words)
plt.bar(words, counts)
plt.xticks(rotation=90)
plt.title('Top 20 Most Common Words')
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.show()
```



OBSERVATIONS

- **Frequent Hand Injuries** – "Hand" and "finger" are the most common words, indicating frequent hand-related incidents.
- **Employee-Centered** – Words like "employee," "operator," and "worker" highlight workplace accidents involving staff.
- **Injury and Direction** – Terms like "injury," "left," and "right" suggest incidents often specify body parts and sides.
- **Task and Time Focus** – "Activity," "moment," and "time" indicate descriptions detail when and during which tasks accidents occur.
- **Equipment and Area** – The presence of "equipment" and "area" points to machinery and specific work zones as common incident locations.

N-GRAMS

```
# Function to calculate ngrams
def extract_ngrams(data, num):
  # Taking ngrams on Description column text and taking the value counts of each of the tokens
  words_with_count  = nltk.FreqDist(nltk.ngrams(data, num)).most_common(30) # taking top 30 most common words

  # Creating the dataframe the words and thier counts
  words_with_count = pd.DataFrame(words_with_count, columns=['Words', 'Count'])

  # Removing the brackets and commans
  words_with_count.Words = [' '.join(i) for i in words_with_count.Words]

  # words_with_count.index = [' '.join(i) for i in words_with_count.Words]
  words_with_count.set_index('Words', inplace=True) # setting the Words as index

  # Returns the dataframe which contains unique tokens ordered by their counts
  return words_with_count
```

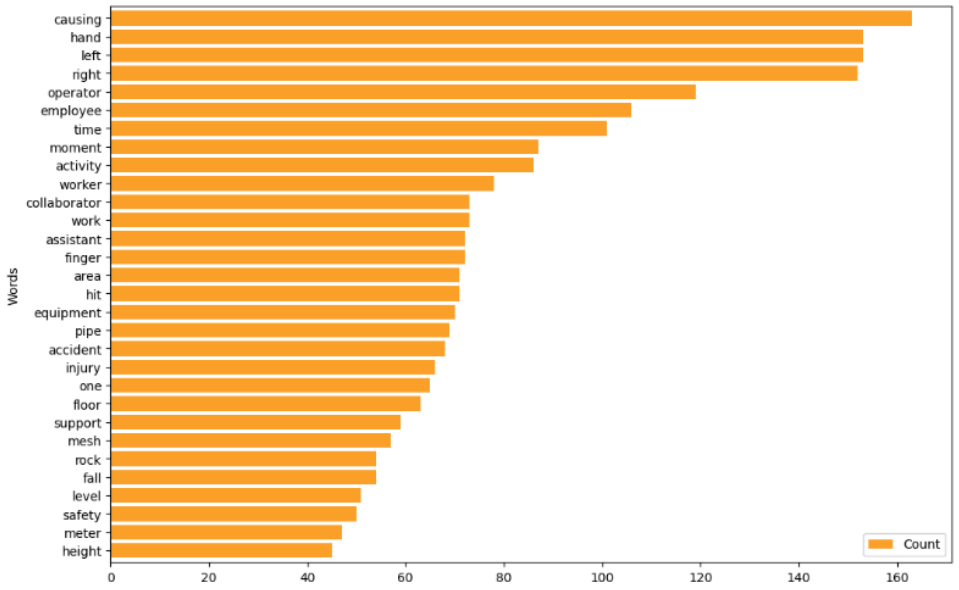
UNI-GRAMS

In [60]:
```
# Uni-Grams
uni_grams = extract_ngrams(tokens, 1)

# Printing top words with their counts
uni_grams[0:10]
```

Out[60]:

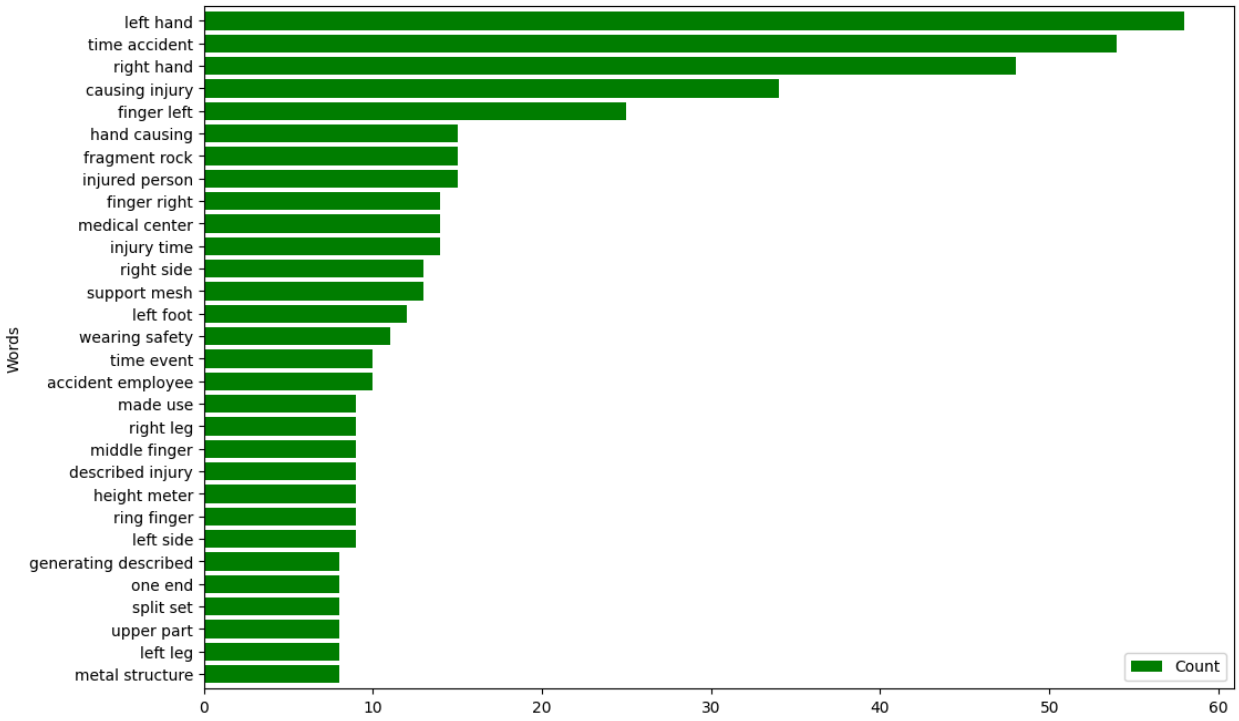
Words	Count
causing	163
hand	153
left	153
right	152
operator	119
employee	106
time	101
moment	87
activity	86
worker	78

```
In [61]: # Visualising the ngrams
uni_grams.sort_values(by='Count').plot.barh(color = 'orange', width = 0.8, figsize = (12,8));
```

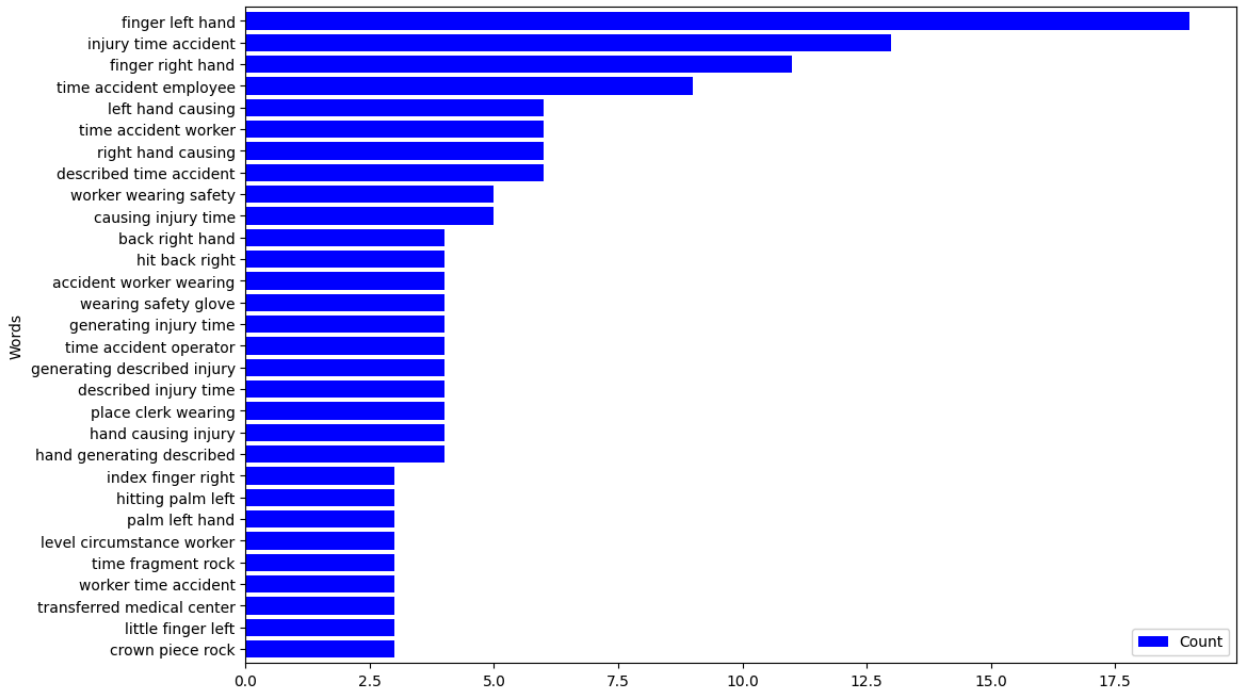


Similarly...

BI-GRAMS



TRI-GRAMS



OBSERVATIONS FROM N-GRAMS:

Focus on Hands and Injuries

- Across all n-grams, terms like **"hand," "left hand," "finger left hand"** dominant, highlighting frequent **hand-related injuries**. This suggests a focus on **manual work hazards**.

Employee and Worker-Centric

- Words like **"employee," "worker," "collaborator"** frequently appear, indicating that **workplace incidents involving staff** are common. Safety measures should target employee protection.

Directional and Specific Injuries

- Phrases like **"left hand," "right hand," "finger left"** suggest that incident reports often **specify the body side and part** affected. This can inform **ergonomic and safety gear improvements**.

Frequent Causes and Activities

- Bigrams and trigrams like **"causing injury," "time accident," "described injury time"** reflect a focus on **causal factors** and the **timing of incidents**. This indicates **incident reporting emphasizes root causes and accident timelines**.

Equipment and Environmental Factors

- Unigrams such as **"equipment, pipe, area"** and phrases like **"fragment rock"** suggest that incidents often involve **machinery, tools, or environmental hazards**. Equipment maintenance and area monitoring are essential.

N-GRAMS BY INDUSTRY SECTORS

Industry Sector

In [66]:
```
# Dividing the tokens with respect to Industry Sector from the description text
tokens_metals = des_cleaning(' '.join(data_cleaned[data_cleaned['Industry Sector']=='Metals']['Cleaned_Description_Final'].sum
().split()))
tokens_mining = des_cleaning(' '.join(data_cleaned[data_cleaned['Industry Sector']=='Mining']['Cleaned_Description_Final'].sum
().split()))
```

In [67]:
```
print('Total number of words in Metals category:', len(tokens_metals))
print('Total number of words in Mining category:',len(tokens_mining))
```

```
Total number of words in Metals category: 2729
Total number of words in Mining category: 7861
```

In [68]:
```
# Extracting unigrams on metals category
unigrams_metals = extract_ngrams(tokens_metals, 1).reset_index()

# Extracting unigrams on mining category
unigrams_mining = extract_ngrams(tokens_mining, 1).reset_index()

unigrams_metals.join(unigrams_mining, lsuffix='_Metals', rsuffix='_Mining')
```

Out[68]:

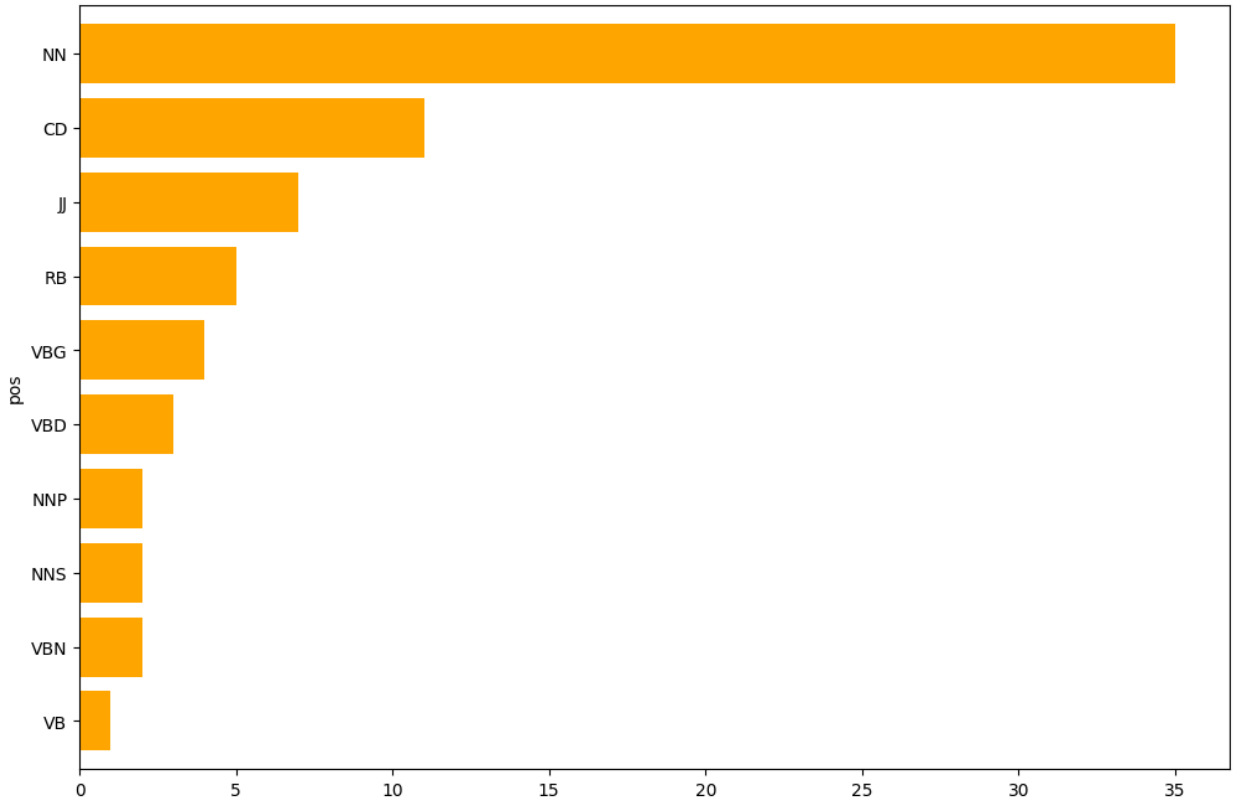
	Words_Metals	Count_Metals	Words_Mining	Count_Mining
0	left	46	hand	105
1	causing	43	causing	102
2	right	37	right	101
3	hand	34	operator	100
4	employee	33	left	93
5	hit	27	time	87
6	activity	27	worker	66
7	medical	24	assistant	64
8	report	23	accident	64

OBSERVATIONS AND INSIGHTS:

- **Hand Injuries Dominate** – "Hand" and "left/right" are top terms, indicating frequent **hand-related incidents** in both sectors.
- **Focus on Causes** – "Causing" and "time/moment" highlight emphasis on **incident causes and timing**.
- **Worker-Centered** – "Employee, operator, worker" show **human factors** are key in accidents.
- **Equipment Risks** – Mining involves "equipment, pipe, rock," while metals show risks with **"hose, pump, acid"**.
- **Injury and Safety** – "Finger, injury, fall, cut" reflect focus on **personal injury and safety measures**.

POS Tagging

```
from textblob import TextBlob
from nltk.corpus import wordnet
import nltk

# Download the necessary NLTK data
nltk.download('averaged_perceptron_tagger')
nltk.download('tagsets')
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger_eng') # Download the required corpus

# def replace_with_synonyms(text):
#     blob = TextBlob(text)
#     new_words = []
#     for word, pos in blob.tags:
#         synonyms = wordnet.synsets(word)
#         if synonyms:
#             # Pick the first synonym
#             synonym = synonyms[0].lemmas()[0].name()
#             new_words.append(synonym)
#         else:
#             new_words.append(word)
#     return ' '.join(new_words)
# # Apply to the DataFrame column
# data_cleaned['Cleaned_Description_Final'] = data_cleaned['Cleaned_Description_Final'].apply(replace_with_synonyms)

# Visualize POS distribution
blob = TextBlob(str(data_cleaned['Cleaned_Description_Final']))
pos_df = pd.DataFrame(blob.tags, columns=['word', 'pos'])
pos_counts = pos_df.pos.value_counts()[:20]
pos_counts.sort_values().plot.barh(color='orange', width=0.8, figsize=(12, 8))
```



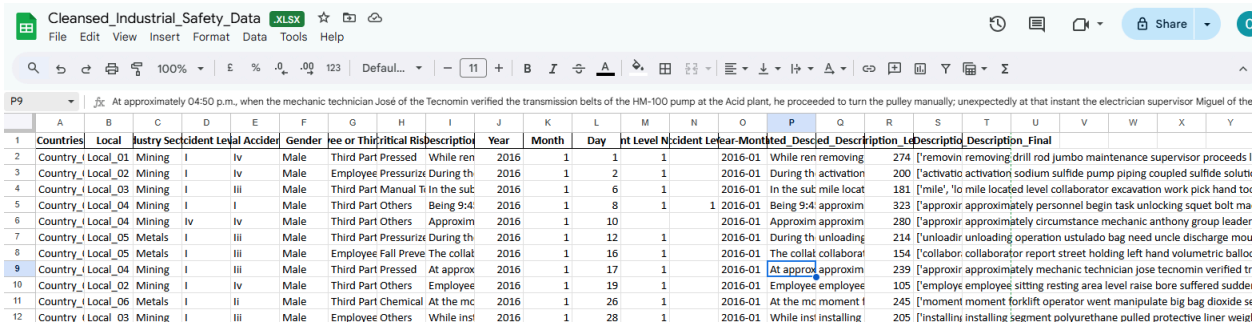
OBSERVATIONS AND INSIGHTS:

- **Nouns (NN) Dominate** – Focus on **objects and entities** like "hand" and "worker."
- **Frequent Numbers (CD)** – Highlights **measurements and quantities** in reports.
- **Descriptive Adjectives (JJ)** – Emphasizes **qualities** of objects/incidents.
- **Actions and Adverbs (RB, VBG, VBD)** – Reflects **detailed event descriptions**.
- **Proper and Plural Nouns (NNP, NNS)** – References to **specific items and multiple objects**.

EXPORTED TO EXCEL FORMAT

```
[49] # Save the cleansed data to a new Excel file
     # output_file_path = '/content/drive/MyDrive/Colab Notebooks/Capstone NLP/Cleansed_Industrial_Safety_Data.xlsx'
     output_file_path = '/content/drive/MyDrive/AIML/Capstone Project/Cleansed_Industrial_Safety_Data.xlsx'
     data_cleaned.to_excel(output_file_path, index=False)

     output_file_path
```

```
'/content/drive/MyDrive/AIML/Capstone Project/Cleansed_Industrial_Safety_Data.xlsx'
```



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
1	Countries	Local	lustry Sec	cident Lev	al Accider	Gender	ee or Thir	ritical Risk	Description	Year	Month	Day	nt Level N	cident Le	ear-Month	ted_Desc	ed_Descri	ription_Le	Descriptio	Description_Final					
2	Country_(	Local_01	Mining	I	Iv	Male	Third Part	Pressed	While ren	2016	1	1	1		2016-01	While ren	removing	274	['removin	removing drill rod jumbo maintenance supervisor proceeds					
3	Country_(	Local_02	Mining	I	Iv	Male	Employee	Pressurize	During th	2016	1	2	1		2016-01	During th	activation	200	['activatio	activation sodium sulfide pump piping coupled sulfide solut					
4	Country_(	Local_03	Mining	I	Iii	Male	Third Part	Manual Ti	In the sub	2016	1	6	1		2016-01	In the sub	mile locat	181	['mile', 'lo	mile located level collaborator excavation work pick hand to					
5	Country_(	Local_04	Mining	I	I	Male	Third Part	Others	Being 9:4	2016	1	8	1	1	2016-01	Being 9:4	approxim	323	['approxir	approximately personnel begin task unlocking squet bolt ma					
6	Country_(	Local_04	Mining	Iv	Iv	Male	Third Part	Others	Approxim	2016	1	10			2016-01	Approxim	approxim	280	['approxir	approximately circumstance mechanic anthony group leader					
7	Country_(	Local_05	Metals	I	Iii	Male	Third Part	Pressurize	During th	2016	1	12	1		2016-01	During th	unloading	214	['unloadir	unloading operation ustulado bag need uncle discharge mou					
8	Country_(	Local_05	Metals	I	Iii	Male	Employee	Fall Preve	The collat	2016	1	16	1		2016-01	The collat	collaborat	154	['collabor	collaborator report street holding left hand volumetric ballo					
9	Country_(	Local_04	Mining	I	Iii	Male	Third Part	Pressed	At approx	2016	1	17	1		2016-01	At approx	approxim	239	['approxir	approximately mechanic technician jose tecnomin verified tr					
10	Country_(	Local_02	Mining	I	Iv	Male	Third Part	Others	Employee	2016	1	19	1		2016-01	Employee	employee	105	['employe	employee sitting resting area level raise bore suffered sudde					
11	Country_(	Local_06	Metals	I	Ii	Male	Third Part	Chemical	At the mc	2016	1	26	1		2016-01	At the mc	moment f	245	['moment	moment forklift operator went manipulate big bag dioxide s					
12	Country_(	Local_03	Mining	I	Iii	Male	Employee	Others	While inst	2016	1	28	1		2016-01	While inst	installing	205	['installing	installing segment polyurethane pulled protective liner weig					

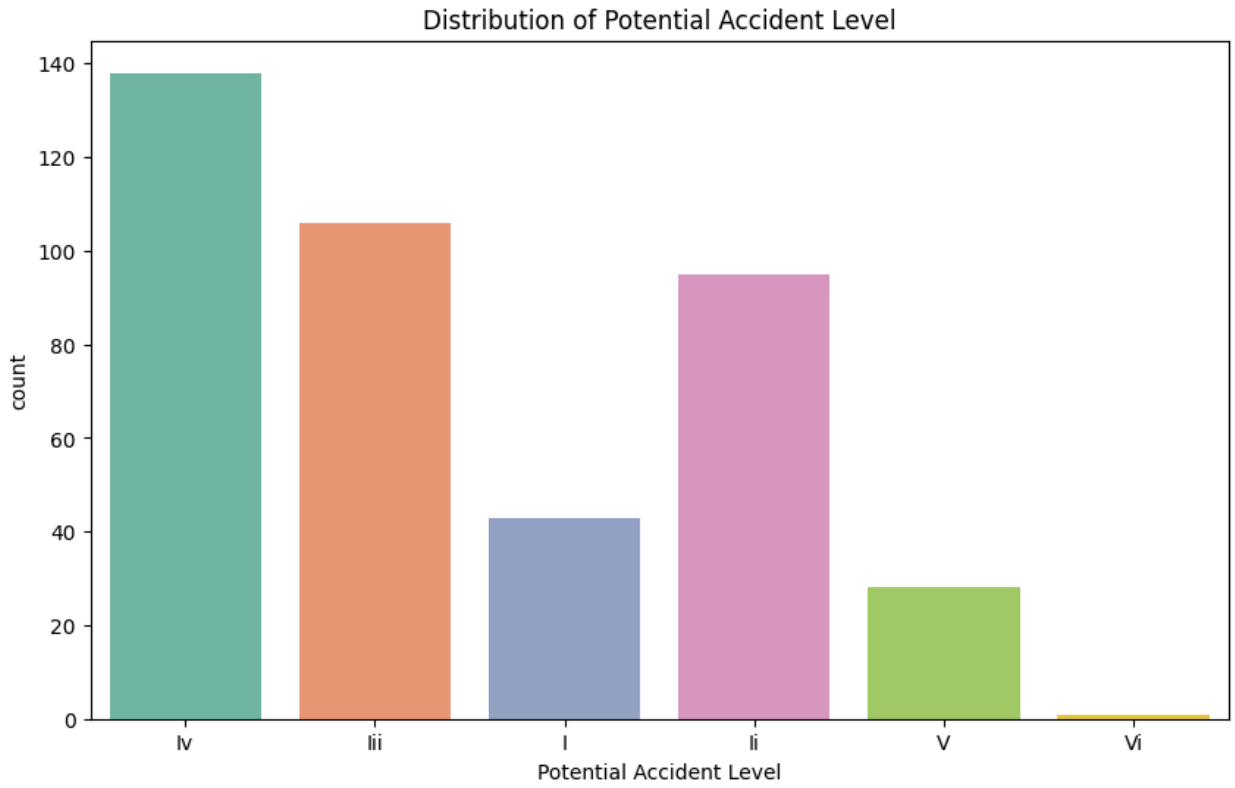
DECIDING MODELS AND MODEL BUILDING

CHECK FOR IMBALANCES

We examined the data for imbalance and analyzed the distribution of the target variable along with the visualization:

```
[75] # Check the distribution of the target variable (e.g., 'Accident Level' or other column)
     target_column = 'Potential Accident Level'  # Change to your actual target column name
     print(data_cleaned[target_column].value_counts())

     # Visualize the target distribution
     plt.figure(figsize=(10,6))
     sns.countplot(x=data_cleaned[target_column], palette='Set2')
     plt.title(f'Distribution of {target_column}')
     plt.show()
```

```
Potential Accident Level
Iv     138
Iii    106
Ii      95
I       43
V       28
Vi       1
Name: count, dtype: int64
```



OBSERVATIONS AND INSIGHTS

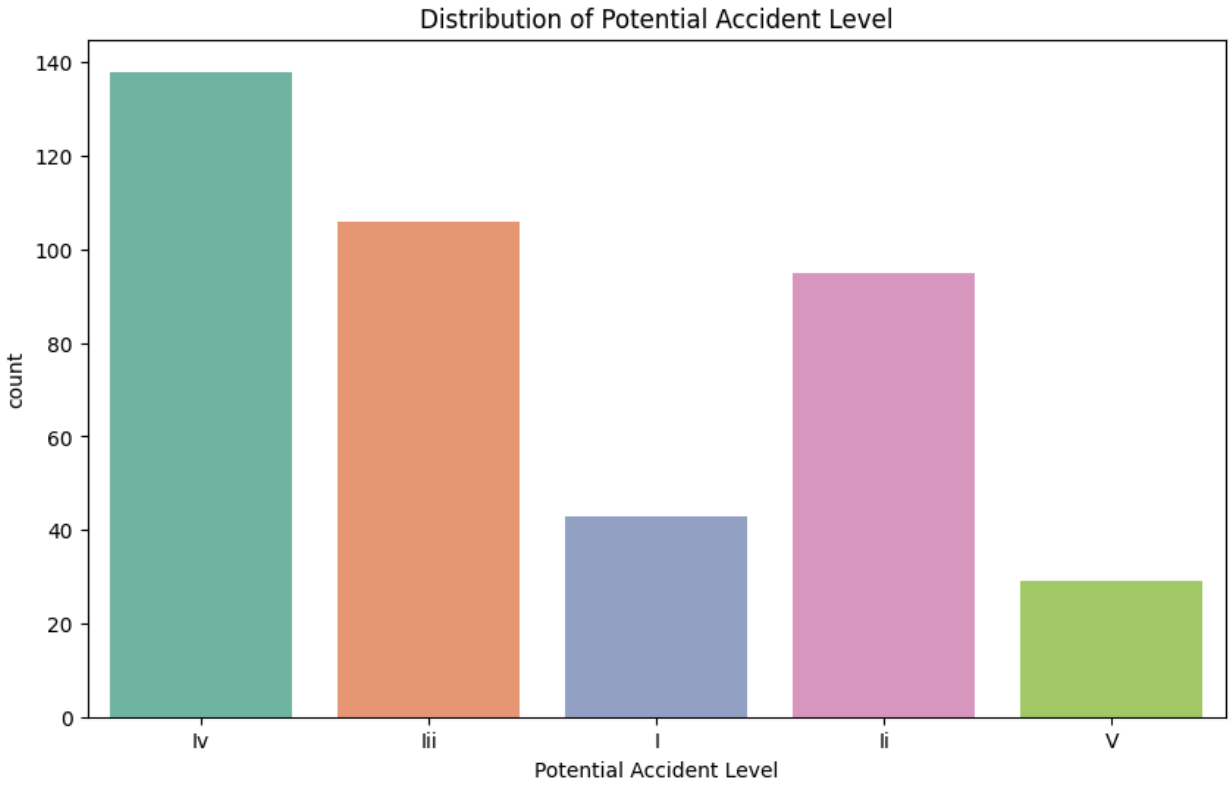
- We merged Class VI with Class V, aimed at reducing class imbalance or simplifying the classification by combining rare categories.
- The target variable is Potential Accident Level, and its distribution has been visualized.
- A count plot shows the following class distribution:
    - **IV:** 138
    - **III:** 106
    - **II:** 95
    - **I:** 43
    - **V (merged with VI):** 29
- There is a noticeable imbalance in the data. Class IV has the highest number of incidents, while Class V (formerly VI) has the fewest.

MERGING POTENTIAL ACCIDENT LEVEL VI WITH V

```
[76] # Replace 'VI' with 'V'
     data_cleaned['Potential Accident Level'] = data_cleaned['Potential Accident Level'].replace('Vi', 'V')

     # Display the updated DataFrame
     print(data_cleaned)
```

```
# Check the distribution of the target variable (e.g., 'Accident Level' or other column)
target_column = 'Potential Accident Level'  # Change to your actual target column name
print(data_cleaned[target_column].value_counts())

# Visualize the target distribution
plt.figure(figsize=(10,6))
sns.countplot(x=data_cleaned[target_column], palette='Set2')
plt.title(f'Distribution of {target_column}')
plt.show()
```

```
Potential Accident Level
Iv     138
Iii    106
Ii      95
I       43
V       29
Name: count, dtype: int64
```



OBSERVATIONS FROM THE PLOT AND DISTRIBUTION

- **Class Imbalance:**

- The imbalance might affect model performance, especially for underrepresented classes (Class V). Consider techniques like oversampling (SMOTE) or class-weight adjustments during model training.

- **Dominance of Certain Classes:**
  - Classes IV and III dominate, while Class I and V appear less frequently. This might influence how well the model predicts rare but critical events.
- **Possible Data Issues:**
  - If Class V represents severe incidents, the lower frequency could imply underreporting or less frequent severe accidents.

ADDRESSING IMBALANCE IN THE TARGET VARIABLE "POTENTIAL ACCIDENT LEVEL"

DATA PREPARATION AND FEATURE ENGINEERING

1. **Variable Creation for TF-IDF Embeddings:**
   - Generated Term Frequency-Inverse Document Frequency (TF-IDF) embeddings for the textual data to represent the importance of words in a document relative to the corpus.
   - These embeddings were used to transform text data into numerical vectors for model training.

Variable Creation for TF-IDF embeddings

```
[ ] from sklearn.feature_extraction.text import TfidfVectorizer
    ind_tfidf_df = pd.DataFrame()
    for i in [1,2,3]:
        vec_tfidf = TfidfVectorizer(max_features=20, norm='l2', stop_words='english', lowercase=True, use_idf=True, ngram_range=(i,i))
        X = vec_tfidf.fit_transform(data_cleaned['Cleaned_Description']).toarray()
        tfs = pd.DataFrame(X, columns=["TFIDF_" + n for n in vec_tfidf.get_feature_names_out()])
        ind_tfidf_df = pd.concat([ind_tfidf_df.reset_index(drop=True), tfs.reset_index(drop=True)], axis=1)

    ind_tfidf_df.head(3)
```

	TFIDF_accident	TFIDF_activity	TFIDF_area	TFIDF_assistant	TFIDF_causing	TFIDF_collaborator	TFIDF_employee	TFIDF_equipment	TFIDF_finger	TFIDF_hand	...	TFIDF_hit right hand	TFIDF_injury time accident	TFID c
0	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	0.57666	0.530974	0.397545	...	0.0	0.0	
1	0.0	0.0	1.000000	0.0	0.000000	0.000000	0.0	0.00000	0.000000	0.000000	...	0.0	0.0	
2	0.0	0.0	0.337877	0.0	0.229605	0.653969	0.0	0.00000	0.000000	0.239775	...	0.0	0.0	

3 rows × 60 columns

2. **Variable Creation for Word2Vec Embeddings:**
   - Trained a Word2Vec model on the textual data or used pre-trained embeddings to capture semantic and syntactic word relationships.

- Averaged the embeddings of words in each document to obtain fixed-length numerical vectors.

Variable Creation for word2vec Embeddings

```
[ ] from gensim.models import Word2Vec
    words_list = [item.split(" ") for item in data_cleaned['Cleaned_Description'].values]
    w2v_model = Word2Vec(words_list, vector_size=100, window=5, min_count=1)

    # Checking the size of the vocabulary
    print("Length of the vocabulary is", len(list(w2v_model.wv.key_to_index)))

    # Dictionary with key as words and the value as the embedding vector.
    words = w2v_model.wv.key_to_index

Length of the vocabulary is 2624
```

```
[ ] def average_vectorizer_Word2Vec(doc):
        # Initializing a feature vector for the sentence
        feature_vector = np.zeros((vec_size,), dtype="float64")

        # Creating a list of words in the sentence that are present in the model vocabulary
        words_in_vocab = [word for word in doc.split() if word in words]

        # adding the vector representations of the words
        for word in words_in_vocab:
            feature_vector += np.array(words[word])

        # Dividing by the number of words to get the average vector
        if len(words_in_vocab) != 0:
            feature_vector /= len(words_in_vocab)

        return feature_vector
```

```
[ ] vec_size = 100
    df_word2vec = pd.DataFrame(data_cleaned['Cleaned_Description'].apply(average_vectorizer_Word2Vec).tolist(), columns=['Feature '+str(i) for i in range(vec_size)]).reset_index(drop=True)
    df_word2vec
```

	Feature 0	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Feature 7	Feature 8	Feature 9	...	Feature 90	Feature 91	Feature 92	Feature 93	Feature 94	Feature 95	Feature 96	Feature 97	Feature 98	Feature 99
0	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	...	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486	445.486486
1	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	...	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037	995.037037
2	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	...	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852	406.851852

3. **Variable Creation for GloVe Embeddings:**
    - Used pre-trained Global Vectors for Word Representation (GloVe) embeddings to convert words into meaningful vector representations.
    - Similar to Word2Vec, document-level embeddings were generated by averaging word embeddings.

Variable Creation for glove embeddings

```
[ ] filename = '/content/drive/MyDrive/AIML/Capstone Project/glove.6B.100d.txt.word2vec'
    # filename = '/content/drive/MyDrive/Colab Notebooks/Capstone NLP/glove.6B.100d.txt.word2vec'
    model = KeyedVectors.load_word2vec_format(filename, binary=False)
```

```
[ ] # Checking the size of the vocabulary
    print("Length of the vocabulary is", len(model.index_to_key))
```

```
Length of the vocabulary is 400000
```

```
[ ] #List of words in the vocabulary
    words = model.index_to_key

    #Dictionary with key as the word and the value as the corresponding embedding vector.
    word_vector_dict = dict(zip(model.index_to_key,list(model.vectors)))

    #Defining the dimension of the embedded vector.
    vec_size=100
```

```
[ ] def average_vectorizer_GloVe(doc):
        # Initializing a feature vector for the sentence
        feature_vector = np.zeros((vec_size,), dtype="float64")

        # Creating a list of words in the sentence that are present in the model vocabulary
        words_in_vocab = [word for word in doc.split() if word in words]

        # adding the vector representations of the words
        for word in words_in_vocab:
            feature_vector += np.array(word_vector_dict[word])

        # Dividing by the number of words to get the average vector
        if len(words_in_vocab) != 0:
            feature_vector /= len(words_in_vocab)

        return feature_vector
```

```
[ ] df_glove = pd.DataFrame(data_cleaned['Cleaned_Description'].apply(average_vectorizer_GloVe).tolist(), columns=['Feature '+str(i) for i in range(vec_size)]).reset_index(drop=True)
    df_glove
```

	Feature 0	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Feature 7	Feature 8	Feature 9	...	Feature 90	Feature 91	Feature 92	Feature 93	Feature 94	Feature 95

4. **Variable Creation for Label Encoding and Dummy Variables:**
    - Categorical variables were encoded using label encoding and one-hot encoding techniques to prepare them for integration with the embeddings.

```
[ ] # Create Industry DataFrame
    ind_featenc_df = pd.DataFrame()

    # Label encoding
    from sklearn.preprocessing import LabelEncoder
    ind_featenc_df['Industry Sector'] = LabelEncoder().fit_transform(data_cleaned['Industry Sector']).astype(np.int8)
    ind_featenc_df['Employee or Third Party'] = LabelEncoder().fit_transform(data_cleaned['Employee or Third Party']).astype(np.int8)
    ind_featenc_df['Critical Risk'] = LabelEncoder().fit_transform(data_cleaned['Critical Risk']).astype(np.int8)
    ind_featenc_df['Local'] = LabelEncoder().fit_transform(data_cleaned['Local']).astype(np.int8)
    ind_featenc_df['Industry Sector'] = LabelEncoder().fit_transform(data_cleaned['Industry Sector']).astype(np.int8)
    ind_featenc_df['Gender'] = LabelEncoder().fit_transform(data_cleaned['Gender']).astype(np.int8)
    ind_featenc_df['Accident Level'] = LabelEncoder().fit_transform(data_cleaned['Accident Level']).astype(np.int8)
    ind_featenc_df['Potential Accident Level'] = LabelEncoder().fit_transform(data_cleaned['Potential Accident Level']).astype(np.int8)
    ind_featenc_df['Countries'] = LabelEncoder().fit_transform(data_cleaned['Countries']).astype(np.int8)
    ind_featenc_df.head()
```

	Industry Sector	Employee or Third Party	Critical Risk	Local	Gender	Accident Level	Potential Accident Level	Countries
0	1	1	20	0	1	0	3	0
1	1	0	21	1	1	0	3	1
2	1	2	14	2	1	0	2	0
3	1	1	16	3	1	0	0	0
4	1	1	16	3	1	3	3	0

COMBINING FEATURES

5. **Combine Encoded Dataframe with TF-IDF Embeddings:**
    - Merged the categorical variables (after encoding) with the TF-IDF embeddings to form a complete dataset for modeling.

Combine encoded dataframe with TF-Idf dataframe

```
[ ] df_enc_tfidf = ind_featenc_df.join(ind_tfidf_df.iloc[:,0:30]).reset_index(drop=True)
    df_enc_tfidf.head()
```

	Industry Sector	Employee or Third Party	Critical Risk	Local	Gender	Accident Level	Potential Accident Level	Countries	TFIDF_accident	TFIDF_activity	...	TFIDF_causing injury	TFIDF_circumstance worker	TFIDF_da silva	TFIDF_described injury	TFIDF_employee report
0	1	1	20	0	1	0	3	0	0.0	0.0	...	0.000000	0.0	0.0	0.0	0.0
1	1	0	21	1	1	0	3	1	0.0	0.0	...	0.000000	0.0	0.0	0.0	0.0
2	1	2	14	2	1	0	2	0	0.0	0.0	...	0.575111	0.0	0.0	0.0	0.0
3	1	1	16	3	1	0	0	0	0.0	0.0	...	0.508049	0.0	0.0	0.0	0.0
4	1	1	16	3	1	3	3	0	0.0	0.0	...	0.601659	0.0	0.0	0.0	0.0

5 rows × 38 columns

```
[ ] # check any null values
    np.any(np.isnan(df_enc_tfidf))
```

```
False
```

```
[ ] df_enc_tfidf.shape
```

```
(411, 38)
```

6. **Combine Encoded Dataframe with Word2Vec Embeddings:**
    - Integrated the label-encoded and one-hot-encoded variables with the Word2Vec-based feature set.

Combine word2vec dataframe with encoded dataframe

```
[ ] df_enc_w2v = ind_featenc_df.join(df_word2vec.iloc[:,0:30]).reset_index(drop=True)
    df_enc_w2v.head()
```

	Industry Sector	Employee or Third Party	Critical Risk	Local	Gender	Accident Level	Potential Accident Level	Countries	Feature 0	Feature 1	...	Feature 20	Feature 21
0	1	1	20	0	1	0	3	0	445.486486	445.486486	...	445.486486	445.486486
1	1	0	21	1	1	0	3	1	995.037037	995.037037	...	995.037037	995.037037
2	1	2	14	2	1	0	2	0	406.851852	406.851852	...	406.851852	406.851852
3	1	1	16	3	1	0	0	0	519.622222	519.622222	...	519.622222	519.622222
4	1	1	16	3	1	3	3	0	671.121951	671.121951	...	671.121951	671.121951

5 rows × 38 columns

```
[ ] # check any null value
    np.any(np.isnan(df_enc_w2v))
```

```
False
```

```
[ ] df_enc_w2v.shape
```

```
(411, 38)
```

7. **Combine Encoded Dataframe with GloVe Embeddings:**

- Joined the encoded categorical features with the GloVe embeddings to form the dataset.

Combine encoded dataframe with glove dataframe

```
[ ] df_enc_glove = ind_featenc_df.join(df_glove.iloc[:,0:30]).reset_index(drop=True)
    df_enc_glove.head()
```

	Industry Sector	Employee or Third Party	Critical Risk	Local	Gender	Accident Level	Potential Accident Level	Countries	Feature 0	Feature 1	...	Feature 20	Feature 21
0	1	1	20	0	1	0	3	0	-0.108220	0.113753	...	0.084231	0.012208
1	1	0	21	1	1	0	3	1	-0.338403	0.239824	...	-0.038621	-0.136955
2	1	2	14	2	1	0	2	0	-0.068391	0.103980	...	0.394791	0.315175
3	1	1	16	3	1	0	0	0	-0.110100	0.018832	...	0.280354	-0.067287
4	1	1	16	3	1	3	3	0	-0.086951	0.109070	...	0.324140	-0.017631

5 rows × 38 columns

```
[ ] # check any null value
    np.any(np.isnan(df_enc_glove))
```

```
False
```

```
[ ] df_enc_glove.shape
```

```
(411, 38)
```

RESAMPLING TECHNIQUES TO HANDLE IMBALANCE

8. **Oversampling with GloVe Dataframe:**
    - Applied oversampling techniques such as SMOTE (Synthetic Minority Oversampling Technique) to balance the class distribution in the dataset created using GloVe embeddings.

```
[ ] # Concatenate our training data back together
    X_up = df_enc_glove.copy()

    # Get the majority and minority class
    acclevel_0_minority = X_up[X_up['Potential Accident Level'] == 0]
    acclevel_1_minority = X_up[X_up['Potential Accident Level'] == 1]
    acclevel_2_minority = X_up[X_up['Potential Accident Level'] == 2]
    acclevel_3_majority = X_up[X_up['Potential Accident Level'] == 3]
    acclevel_4_minority = X_up[X_up['Potential Accident Level'] == 4]

    # Upsample Level0 minority class
    acclevel_0_minority_upsampled = resample(acclevel_0_minority,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority), # to match majority class
                                    random_state = 1)

    # Upsample Level1 minority class
    acclevel_1_minority_upsampled = resample(acclevel_1_minority,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority), # to match majority class
                                    random_state = 1)

    # Upsample Level2 minority class
    acclevel_2_minority_upsampled = resample(acclevel_2_minority,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority), # to match majority class
                                    random_state = 1)

    # Upsample Level4 minority class
    acclevel_4_minority_upsampled = resample(acclevel_4_minority,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority), # to match majority class
                                    random_state = 1)
```

```
[ ] # Combine majority class with upsampled minority classes
    df_upsampled_glove = pd.concat([acclevel_0_minority_upsampled, acclevel_1_minority_upsampled, acclevel_2_minority_upsampled, acclevel_3_majority,
                            acclevel_4_minority_upsampled]).reset_index(drop=True)
```

```
[ ] # Display new Potential accident level counts
    df_upsampled_glove['Potential Accident Level'].value_counts()
```

Potential Accident Level	count
0	138
1	138
2	138
3	138
4	138

dtype: int64

9. **Oversampling with Word2Vec Dataframe:**
    - Repeated the oversampling process on the Word2Vec-based dataset to handle the class imbalance.

```
[ ] # Concatenate our training data back together
    X_up_w2v = df_enc_w2v.copy()

    # Get the majority and minority class
    acclevel_0_minority_w2v = X_up_w2v[X_up_w2v['Potential Accident Level'] == 0]
    acclevel_1_minority_w2v = X_up_w2v[X_up_w2v['Potential Accident Level'] == 1]
    acclevel_2_minority_w2v = X_up_w2v[X_up_w2v['Potential Accident Level'] == 2]
    acclevel_3_majority_w2v = X_up_w2v[X_up_w2v['Potential Accident Level'] == 3]
    acclevel_4_minority_w2v = X_up_w2v[X_up_w2v['Potential Accident Level'] == 4]

    # Upsample Level0 minority class
    acclevel_0_minority_upsampled_w2v = resample(acclevel_0_minority_w2v,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority_w2v), # to match majority class
                                    random_state = 1)

    # Upsample Level1 minority class
    acclevel_1_minority_upsampled_w2v = resample(acclevel_1_minority_w2v,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority_w2v), # to match majority class
                                    random_state = 1)

    # Upsample Level2 minority class
    acclevel_2_minority_upsampled_w2v = resample(acclevel_2_minority_w2v,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority_w2v), # to match majority class
                                    random_state = 1)

    # Upsample Level4 minority class
    acclevel_4_minority_upsampled_w2v = resample(acclevel_4_minority_w2v,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority_w2v), # to match majority class
                                    random_state = 1)
```

```
[ ] # Combine majority class with upsampled minority classes
    df_upsampled_w2v = pd.concat([acclevel_0_minority_upsampled_w2v, acclevel_1_minority_upsampled_w2v, acclevel_2_minority_upsampled_w2v, acclevel_3_majority_w2v,
                            acclevel_4_minority_upsampled_w2v]).reset_index(drop=True)
```

```
[ ] df_upsampled_w2v['Potential Accident Level'].value_counts()
```

Potential Accident Level	count
0	138
1	138
2	138
3	138
4	138

dtype: int64

10. **Oversampling with TF-IDF Dataframe:**
    - Performed oversampling on the TF-IDF-based dataset to ensure balanced representation of all classes.

```
[ ] # Concatenate our training data back together
    X_up_tfidf = df_enc_tfidf.copy()

    # Get the majority and minority class
    acclevel_0_minority_tfidf = X_up_tfidf[X_up_tfidf['Potential Accident Level'] == 0]
    acclevel_1_minority_tfidf = X_up_tfidf[X_up_tfidf['Potential Accident Level'] == 1]
    acclevel_2_minority_tfidf = X_up_tfidf[X_up_tfidf['Potential Accident Level'] == 2]
    acclevel_3_majority_tfidf = X_up_tfidf[X_up_tfidf['Potential Accident Level'] == 3]
    acclevel_4_minority_tfidf = X_up_tfidf[X_up_tfidf['Potential Accident Level'] == 4]

    # Upsample Level0 minority class
    acclevel_0_minority_upsampled_tfidf = resample(acclevel_0_minority_tfidf,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority_tfidf), # to match majority class
                                    random_state = 1)

    # Upsample Level1 minority class
    acclevel_1_minority_upsampled_tfidf = resample(acclevel_1_minority_tfidf,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority_tfidf), # to match majority class
                                    random_state = 1)

    # Upsample Level2 minority class
    acclevel_2_minority_upsampled_tfidf = resample(acclevel_2_minority_tfidf,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority_tfidf), # to match majority class
                                    random_state = 1)

    # Upsample Level4 minority class
    acclevel_4_minority_upsampled_tfidf = resample(acclevel_4_minority_tfidf,
                                    replace = True, # sample with replacement
                                    n_samples = len(acclevel_3_majority_tfidf), # to match majority class
                                    random_state = 1)
```

```
[ ] # Combine majority class with upsampled minority classes
    df_upsampled_tfidf = pd.concat([acclevel_0_minority_upsampled_tfidf, acclevel_1_minority_upsampled_tfidf, acclevel_2_minority_upsampled_tfidf, acclevel_3_majority_tfidf,
                            acclevel_4_minority_upsampled_tfidf]).reset_index(drop=True)
```

```
[ ] df_upsampled_tfidf['Potential Accident Level'].value_counts()
```

Potential Accident Level	count
0	138
1	138
2	138
3	138
4	138

dtype: int64

MODEL BUILDING

MODEL BUILDING AND EVALUATION

11. **Model Building with Different Embedding Dataframes:**
    - Built machine learning models (e.g., Random Forest, Logistic Regression, Gradient Boosting) on the three different datasets—GloVe, Word2Vec, and TF-IDF embeddings.
    - The encoded categorical variables were included in all models.

```python
# Function to train multiple models and evaluate accuracy
def ml_models(X_train, y_train, X_test, y_test):
    # Dictionary of models
    models = {
        'Logistic Regression': LogisticRegression(max_iter=500, random_state=42),
        'Naive Bayes': GaussianNB(),
        'K-Nearest Neighbors': KNeighborsClassifier(),
        'Support Vector Machine': SVC(probability=True, random_state=42),
        'Decision Tree': DecisionTreeClassifier(criterion='entropy', max_depth=6, random_state=100, min_samples_leaf=5),
        'Random Forest': RandomForestClassifier(n_estimators=100, max_depth=7, random_state=42),
        'Bagging': BaggingClassifier(n_estimators=50, max_samples=0.7, random_state=42),
        'AdaBoost': AdaBoostClassifier(n_estimators=50, random_state=42),
        'Gradient Boosting': GradientBoostingClassifier(n_estimators=50, learning_rate=0.05, random_state=42),
        'XGBoost': XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42),
        'LightGBM': LGBMClassifier(n_estimators=20, max_depth=7, learning_rate=0.1, random_state=42)
    }

    # Initialize lists to store model names and accuracies
    names = []
    test_scores = []
    train_scores = []

    # Train and evaluate each model
    for name, model in models.items():
        try:
            model.fit(X_train, y_train)  # Train the model
            train_score = model.score(X_train, y_train)  # Evaluate accuracy on training data
            train_scores.append(train_score)
            accuracy = model.score(X_test, y_test)  # Evaluate accuracy on test data
            names.append(name)
            test_scores.append(accuracy)
        except Exception as e:
            print(f"Error training {name}: {e}")
            names.append(name)
            test_scores.append(None)  # Append None for models that fail

    # Create a DataFrame with results
    result_df = pd.DataFrame({'Model': names, 'Accuracy': test_scores,'Train_accuracy':train_scores}).sort_values(by='Accuracy', ascending=False)

    return result_df
```

12. With GloVe Embeddings:
    - Trained models using the GloVe-based feature set after addressing the class imbalance.
    - Evaluated the models on train and test accuracy.

With Glove Embeddings

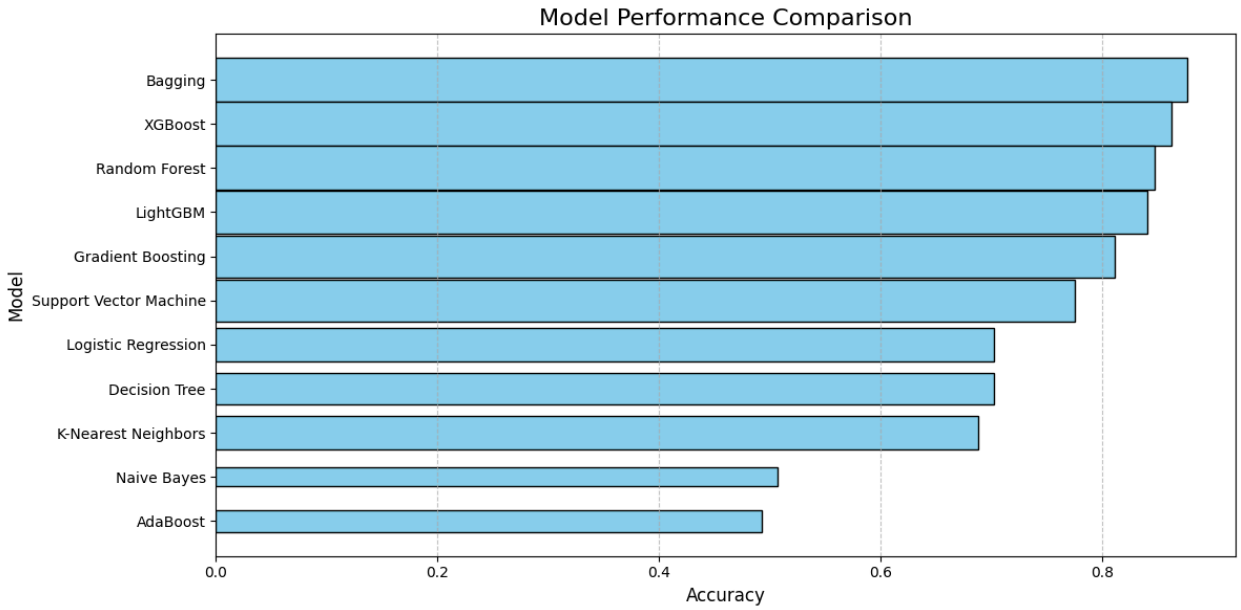
```python
[ ] from sklearn.model_selection import train_test_split

    # Separate features and target
    X_glove = df_upsampled_glove.drop(['Potential Accident Level'],axis=1)
    y_glove = df_upsampled_glove['Potential Accident Level']

    # Split into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X_glove, y_glove, test_size=0.2, random_state=42)

    print("Training and testing data prepared.")
```

```
Training and testing data prepared.
```

```python
[ ] # scaling or normalizing variable
    from sklearn.preprocessing import StandardScaler
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)
```

```
[ ]  # Call the function with train-test data
     results = ml_models(X_train, y_train, X_test, y_test)

     # Display the results
     print(results)
```

```
                    Model  Accuracy  Train_accuracy
6                 Bagging  0.876812        1.000000
9                 XGBoost  0.862319        1.000000
5           Random Forest  0.847826        0.989130
10               LightGBM  0.840580        0.994565
8        Gradient Boosting  0.811594        0.949275
3   Support Vector Machine  0.775362        0.943841
0      Logistic Regression  0.702899        0.753623
4            Decision Tree  0.702899        0.726449
2      K-Nearest Neighbors  0.688406        0.751812
1              Naive Bayes  0.507246        0.423913
7                 AdaBoost  0.492754        0.505435
```



Insights from Model Performance with GloVe Embeddings:

1. **Top Performing Models:**
    - **Bagging Classifier** achieved the highest test accuracy of 87.68%, with perfect training accuracy (100%). This suggests excellent generalisation while capturing complex patterns in the data.
    - **XGBoost** performed similarly, with a test accuracy of 86.23% and a training accuracy of 100%, indicating its robustness in handling the embeddings.

➢ **Balanced Performers:**

- **Random Forest** and **LightGBM** showcased strong performance, with test accuracies of 84.78% and 84.06%, respectively, and high training accuracies (above 98%). These models are reliable choices for complex datasets with minimal overfitting.
- **Gradient Boosting** performed well with a test accuracy of 81.16%, slightly lower but still effective, showcasing its potential for handling embeddings with its ensemble approach.

Key Observations:

- **Ensemble Methods Dominate:** Bagging, XGBoost, and Random Forest emerged as the top models, leveraging their ensemble strategies to outperform others. Overfitting Risks: Models like XGBoost and Bagging showed perfect training accuracy, which may indicate some overfitting, although test performance remained strong.
- **Naive Bayes Limitation:** The poor performance of Naive Bayes highlights its unsuitability for datasets with highly interdependent features like embeddings

13. With Word2Vec Embeddings:

- Repeated the modeling process using Word2Vec embeddings, combined with the categorical features.
- Assessed performance to determine the effectiveness of Word2Vec.

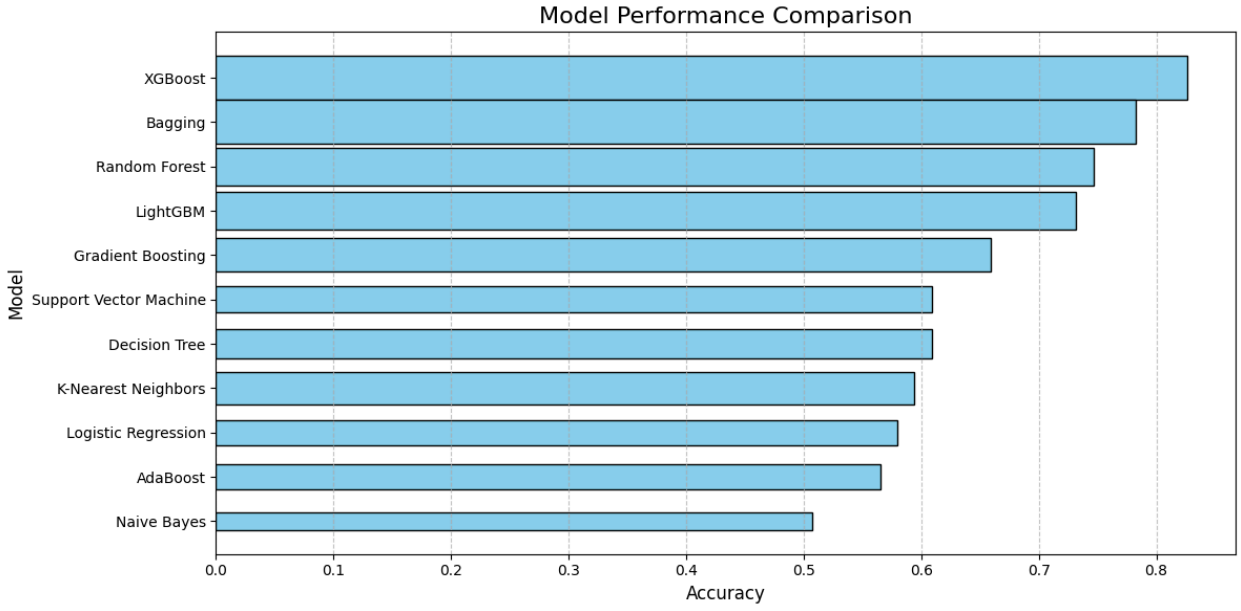
```
[ ]  x_w2v = df_upsampled_w2v.drop(['Potential Accident Level'],axis=1)
     y_w2v = df_upsampled_w2v['Potential Accident Level']
```

```
[ ]  x_train_w2v, x_test_w2v, y_train_w2v, y_test_w2v = train_test_split(x_w2v, y_w2v, test_size=0.2, random_state=42)
```

```
[ ]  # Apply scaling on variable
     scaler = StandardScaler()
     x_train_w2v_scaled = scaler.fit_transform(x_train_w2v)
     x_test_w2v_scaled = scaler.transform(x_test_w2v)
```

```
[ ]   # Call the function with train-test data
     results_w2v = ml_models(x_train_w2v_scaled, y_train, x_test_w2v_scaled, y_test)

     # Display the results
     print(results_w2v)
```

	Model	Accuracy	Train_accuracy
9	XGBoost	0.826087	1.000000
6	Bagging	0.782609	0.990942
5	Random Forest	0.746377	0.860507
10	LightGBM	0.731884	0.838768
8	Gradient Boosting	0.659420	0.757246
3	Support Vector Machine	0.608696	0.581522
4	Decision Tree	0.608696	0.648551
2	K-Nearest Neighbors	0.594203	0.744565
0	Logistic Regression	0.579710	0.570652
7	AdaBoost	0.565217	0.550725
1	Naive Bayes	0.507246	0.413043



Insights from Model Performance with Word2Vec Embeddings:

1. **Top Performing Models:**

- **XGBoost** achieved the highest test accuracy of 82.61%, with a perfect training accuracy (100%). This indicates that XGBoost effectively handles the Word2Vec embeddings but may exhibit slight overfitting.
- **Bagging Classifier** followed with a test accuracy of 78.26%, accompanied by a high training accuracy (99.09%). It balances generalisation and overfitting better than XGBoost.

2. **Balanced Performers:**
    - **Random Forest** and **LightGBM** displayed moderate performance, with test accuracies of 74.64% and 73.19%, respectively. Their training accuracies (86.05% and 83.88%) suggest better generalisation compared to the top models.
    - **Gradient Boosting** had a lower test accuracy (65.94%), showing limited ability to capture complex patterns in the embeddings. Moderate Performers:

Key Observations:

- **XGBoost Leads Again:** XGBoost maintains its top position, demonstrating its effectiveness in handling high-dimensional Word2Vec embeddings.
- **Bagging Shows Versatility:** Bagging remains a strong contender with balanced performance, indicating its robustness across different embeddings.
- **Struggles with Simpler Models:** Simpler models like Logistic Regression, Naive Bayes, and KNN struggle to capture the nuanced relationships encoded in Word2Vec.

14. With TF-IDF Embeddings:
    - Built models using TF-IDF embeddings integrated with the encoded features.

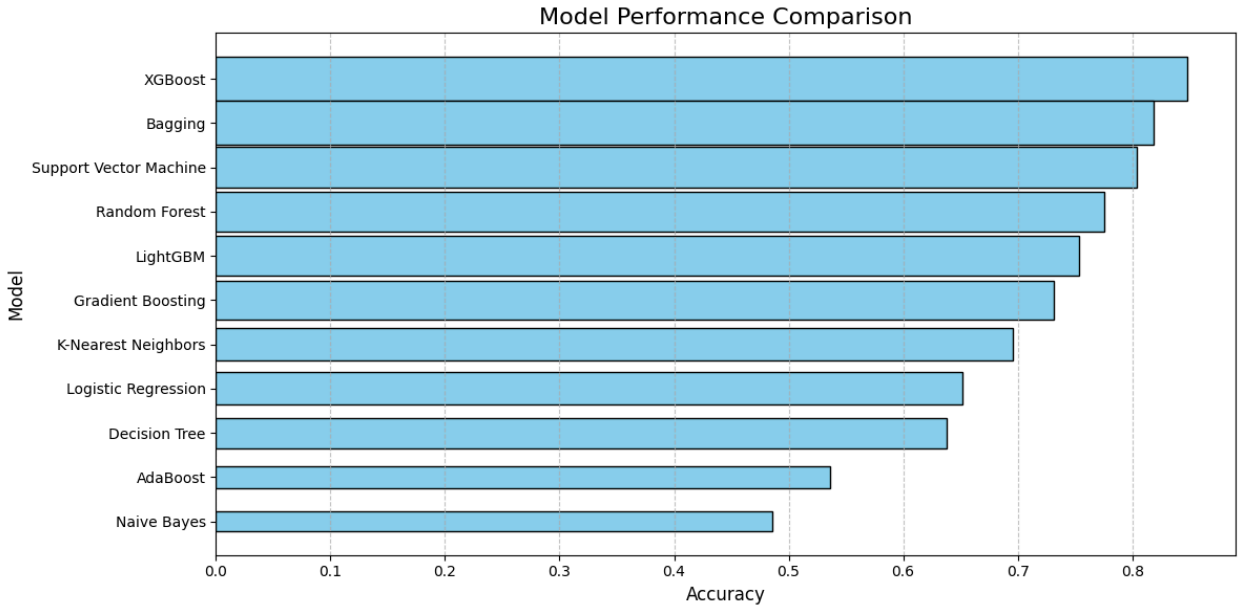
```
[ ] x_tfidf = df_upsampled_tfidf.drop(['Potential Accident Level'],axis=1)
    y_tfidf = df_upsampled_tfidf['Potential Accident Level']
```

```
[ ] x_train_tfidf, x_test_tfidf, y_train_tfidf, y_test_tfidf = train_test_split(x_tfidf, y_tfidf, test_size=0.2, random_state=42)
```

```
[ ] # Applying the scalar on variable
    scaler = StandardScaler()
    x_train_tfidf_scaled = scaler.fit_transform(x_train_tfidf)
    x_test_tfidf_scaled = scaler.transform(x_test_tfidf)
```

```
[ ] # Call the function with train-test data
    results_tfidf = ml_models(x_train_tfidf_scaled, y_train_tfidf, x_test_tfidf_scaled, y_test_tfidf)

    # Display the results
    print(results_tfidf)
```

	Model	Accuracy	Train_accuracy
9	XGBoost	0.847826	0.998188
6	Bagging	0.818841	0.990942
3	Support Vector Machine	0.804348	0.903986
5	Random Forest	0.775362	0.889493
10	LightGBM	0.753623	0.893116
8	Gradient Boosting	0.731884	0.865942
2	K-Nearest Neighbors	0.695652	0.740942
0	Logistic Regression	0.652174	0.728261
4	Decision Tree	0.637681	0.690217
7	AdaBoost	0.536232	0.512681
1	Naive Bayes	0.485507	0.447464



Insights from Model Performance with TF-IDF Embeddings:

1. **Top Performing Models:**

    - **XGBoost** leads with a test accuracy of 84.78% and a high training accuracy (99.82%). This suggests that it effectively captures patterns in TF-IDF embeddings but may slightly overfit.
    - **Bagging Classifier** achieved the second-highest test accuracy (81.88%) with a high training accuracy (99.09%), balancing generalisation and overfitting.
2. Strong Contenders:
    - **Support Vector Machine (SVM)** showed strong performance with a test accuracy of 80.43%, and a reasonable training accuracy (90.40%). SVM effectively models the relationships in TF-IDF embeddings without overfitting excessively.
    - **Random Forest** and **LightGBM** followed with test accuracies of 77.54% and 75.36%, respectively. Both exhibit robust training accuracy above 88%, indicating their capability to generalise well.

Key Observations:

- **XGBoost and Bagging Dominate:** XGBoost remains the most effective, while Bagging shows consistency across different embeddings with excellent performance.
- **SVM Shines with TF-IDF:** Support Vector Machine demonstrates competitive performance, leveraging the sparse structure of TF-IDF embeddings effectively.
- **Simple Models Struggle:** Logistic Regression, AdaBoost, and Naive Bayes continue to struggle with the complexity of embeddings, indicating the need for advanced modelling techniques.

HYPERPARAMETER TUNING

MODELS SELECTED FOR HYPERPARAMETER TUNING

- **Bagging Classifier:**
  Strong and consistent performance with effective generalization.
- **Random Forest:**
  Reliable and balanced performance across embeddings with minimal overfitting.

- **Gradient Boosting:**
  Moderate performance but promising for improvement with tuning.
- **Logistic Regression:**
  Included as a benchmark for interpretability and comparison.

Rationale:

The selected models excelled in handling complex relationships in embeddings, offered robustness across techniques, and demonstrated potential for optimization through hyperparameter tuning.

1. Grid Search For Bagging

```
[ ] from sklearn.model_selection import GridSearchCV

    # Define hyperparameter grid
    param_grid_bagging = {
        'n_estimators': [10, 15, 20, 25, 30, 35, 40, 45, 50],
        'max_samples': [0.5, 0.6, 0.7, 0.8],
        'max_features': [0.5, 0.6, 0.7, 0.8]
    }

    # Initialize Bagging Classifier
    bagging = BaggingClassifier()

    # GridSearch for Bagging Classifier
    grid_search_bagging = GridSearchCV(estimator=bagging, param_grid=param_grid_bagging, cv=5, scoring='accuracy', n_jobs=-1)
    grid_search_bagging.fit(X_train, y_train)

    # Best parameters and score
    print("Best parameters for Bagging Classifier:", grid_search_bagging.best_params_)
    print("Best score for Bagging Classifier:", grid_search_bagging.best_score_)
```

```
Best parameters for Bagging Classifier: {'max_features': 0.6, 'max_samples': 0.6, 'n_estimators': 40}
Best score for Bagging Classifier: 0.8044717444717444
```

2. Grid Search For Random Forest

```
# Define hyperparameter grid
param_grid_rf = {
    'n_estimators': [50, 80, 110],
    'criterion': ['gini', 'entropy'],
    'max_depth': [5, 15],
    'min_samples_split': range(2, 5),
    'min_samples_leaf': range(1, 4),
    'max_features': ['sqrt', 'log2']
}

# Initialize RF
rf = RandomForestClassifier()

# GridSearch for RF
grid_search_rf = GridSearchCV(estimator=rf, param_grid=param_grid_rf, cv=5, scoring='accuracy', n_jobs=-1)
grid_search_rf.fit(X_train, y_train)

# Best parameters and score
print("Best parameters for Random Forest:", grid_search_rf.best_params_)
print("Best cross-validated score for Random Forest:", grid_search_rf.best_score_)
```

```
Best parameters for Random Forest: {'criterion': 'entropy', 'max_depth': 15, 'max_features': 'log2', 'min_samples_leaf': 1, 'min_samples_split': 3, 'n_estimators': 110}
Best cross-validated score for Random Forest: 0.8278951678951678
```

3. Grid Search For Gradient Boosting

```
# Define hyperparameter grid for Gradient Boosting
param_grid_gradientboosting = {
    'n_estimators': [50, 100],
    'criterion': ['friedman_mse'],
    'max_depth': [5, 10],
    'min_samples_split': [2, 3],
    'min_samples_leaf': [1, 2],
    'max_features': ['sqrt'],
    'loss': ['log_loss']
}

# Initialize GB
gb = GradientBoostingClassifier()

# GridSearch for GB
grid_search_gb = GridSearchCV(estimator=gb, param_grid=param_grid_gradientboosting, cv=5, scoring='accuracy', n_jobs=-1)
grid_search_gb.fit(X_train, y_train)

# Best parameters and score
print("Best parameters for Gradient Boosting:", grid_search_gb.best_params_)
print("Best cross-validated score for Gradient Boosting:", grid_search_gb.best_score_)
```

```
Best parameters for Gradient Boosting: {'criterion': 'friedman_mse', 'loss': 'log_loss', 'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'min_samples_split': 3, 'n_estimators': 100}
Best cross-validated score for Gradient Boosting: 0.829713349713349
```

4. Grid Search For Logistic Regression

```
[ ] from sklearn.linear_model import LogisticRegression

    # Define hyperparameter grid
    param_grid_lr = {
        'penalty': ['l1', 'l2', 'elasticnet'],
        'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],
        'class_weight': ['balanced', None],
        'multi_class': ['auto', 'ovr', 'multinomial'],
        'C': [0.1, 1, 10, 50, 100],
        'solver': ['liblinear', 'lbfgs']
    }

    # Initialize Logistic Regression
    lr = LogisticRegression(max_iter=500)

    # GridSearch for Logistic Regression
    grid_search_lr = GridSearchCV(estimator=lr, param_grid=param_grid_lr, cv=5, scoring='accuracy', n_jobs=-1)
    grid_search_lr.fit(X_train, y_train)

    # Best parameters and score
    print("Best parameters for Logistic Regression:", grid_search_lr.best_params_)
    print("Best cross-validated score for Logistic Regression:", grid_search_lr.best_score_)
```

```
Best parameters for Logistic Regression: {'C': 10, 'class_weight': 'balanced', 'multi_class': 'auto', 'penalty': 'l2', 'solver': 'lbfgs'}
Best cross-validated score for Logistic Regression: 0.6738247338247338
```

FITTING THE MODEL WITH GRIDSEARCH WITH BEST PARAMETERS FOR THE SELECTED MODELS.

```
# Function to train multiple models and evaluate accuracy
def ml_models_final(X_train, y_train, X_test, y_test):
    # Dictionary of models
    models = {
        'Logistic Regression': LogisticRegression(C=10, class_weight='balanced', multi_class='auto', penalty='l2', solver='lbfgs',  max_iter=500, random_state=42),
        'Random Forest': RandomForestClassifier(criterion='gini', n_estimators=80, max_depth=15, max_features='sqrt', min_samples_leaf=1, min_samples_split=3, random_state=42),
        'Bagging': BaggingClassifier(n_estimators=40, max_samples=0.8, max_features=0.6, random_state=42),
        'Gradient Boosting': GradientBoostingClassifier(criterion='friedman_mse', n_estimators=100, max_depth=10, max_features='sqrt', min_samples_leaf=1, min_samples_split=2, learning_rate=0.05, random_state=42)
    }

    # Initialize lists to store model names and accuracies
    names = []
    test_scores = []
    train_scores = []

    # Train and evaluate each model
    for name, model in models.items():
        try:
            model.fit(X_train, y_train)  # Train the model
            train_score = model.score(X_train, y_train)  # Evaluate accuracy on training data
            train_scores.append(train_score)
            accuracy = model.score(X_test, y_test)  # Evaluate accuracy on test data
            names.append(name)
            test_scores.append(accuracy)
        except Exception as e:
            print(f"Error training {name}: {e}")
            names.append(name)
            test_scores.append(None)  # Append None for models that fail

    # Create a DataFrame with results
    result_df_final = pd.DataFrame({'Model': names, 'Accuracy': test_scores,'Train_accuracy':train_scores}).sort_values(by='Accuracy', ascending=False)

    return result_df_final
```

1. **With Glove dataframe**

```
[ ]  # Call the function with train-test data
     results_final = ml_models_final(X_train, y_train, X_test, y_test)

     # Display the results
     print(results_final)
```

```
                 Model  Accuracy  Train_accuracy
1        Random Forest  0.876812        1.000000
2              Bagging  0.862319        1.000000
3    Gradient Boosting  0.855072        1.000000
0  Logistic Regression  0.710145        0.764493
```

2. **With Word2Vec dataframe**

```
 # Call the function with train-test data
results_w2v_final = ml_models_final(x_train_w2v_scaled, y_train, x_test_w2v_scaled, y_test)

# Display the results
print(results_w2v_final)
```

```
                 Model  Accuracy  Train_accuracy
1        Random Forest  0.811594        0.998188
2              Bagging  0.789855        0.996377
3    Gradient Boosting  0.782609        1.000000
0  Logistic Regression  0.565217        0.565217
```

3. With TF-IDF dataframe

```
[ ] # Call the function with train-test data
    results_tfidf_final = ml_models_final(x_train_tfidf_scaled, y_train_tfidf, x_test_tfidf_scaled, y_test_tfidf)

    # Display the results
    print(results_tfidf_final)
```

```
                 Model  Accuracy  Train_accuracy
3    Gradient Boosting  0.847826        0.998188
2              Bagging  0.833333        0.996377
1        Random Forest  0.811594        0.990942
0  Logistic Regression  0.644928        0.748188
```

SELECTION OF THE BEST EMBEDDING TECHNIQUE

- **With GloVe Embeddings:**
    i. **Random Forest** achieved the highest test accuracy (87.68%) with perfect training accuracy (100%), indicating its robustness in capturing patterns after hyperparameter optimization.
    ii. **Bagging** and **Gradient Boosting** also performed well with test accuracies of 86.23% and 85.51%, respectively. These models consistently show strong generalisation across datasets.
    iii. **Logistic Regression**, while improved slightly (71.01% test accuracy), remains less effective compared to ensemble methods, showing its limitations in handling the complexity of embeddings.

- **With Word2Vec Embeddings:**
    i. **Random Forest** leads again with a test accuracy of 81.16%, showing consistent performance post-optimisation.
    ii. **Bagging** and **Gradient Boosting** followed with test accuracies of 78.99% and 78.26%, indicating that ensemble methods remain top choices.
    iii. **Logistic Regression**, with a test accuracy of 56.52%, struggled despite hyperparameter tuning, reaffirming its challenges with Word2Vec embeddings.

- **With TF-IDF Embeddings:**

i. **Gradient Boosting** emerged as the best performer with a test accuracy of 84.78%, showing its ability to handle the sparsity and high dimensionality of TF-IDF embeddings.
ii. **Bagging** achieved a strong test accuracy of 83.33%, maintaining its versatility across embeddings.
iii. **Random Forest**, with a test accuracy of 81.16%, continues to perform well but lags slightly behind the top two.
iv. **Logistic Regression** showed limited effectiveness with a test accuracy of 64.49%, struggling to model complex relationships in TF-IDF embeddings.

Key Observations:

1. **Ensemble Models Dominate Across Embeddings:**
    - **Random Forest**, **Bagging**, and **Gradient Boosting** consistently outperform Logistic Regression across GloVe, Word2Vec, and TF-IDF embeddings.
    - These models benefit greatly from hyperparameter optimisation, as evidenced by their strong test accuracies.

2. **Performance Variation by Embedding Type:**
    - **GloVe**: Random Forest shines the most, likely due to the structured and dense nature of embeddings.
    - **Word2Vec**: Random Forest maintains its lead, but Bagging and Gradient Boosting remain strong alternatives.
    - **TF-IDF**: Gradient Boosting performs best, leveraging its ability to handle high-dimensional sparse data effectively.

3. **Logistic Regression Limitations:**
    - Despite some improvements, Logistic Regression consistently underperforms across all embeddings, indicating its unsuitability for complex feature sets.

Best Embedding Selection

- **Chosen Embedding: GloVe**

- Consistently delivered the best test accuracies and robust performance across multiple models.
- Captures both semantic and syntactic relationships effectively, making it suitable for the target task.
- Ensemble methods (Bagging, Random Forest, XGBoost) demonstrated their highest potential with GloVe embeddings.

Rationale for Selection

- GloVe embeddings outperformed Word2Vec and TF-IDF in generalization and predictive accuracy.
- Its pre-trained nature ensures richer semantic representation, particularly beneficial for handling complex relationships in the data.
- Balanced performance across models solidifies GloVe's suitability for further model optimization.

SYNONYM REPLACEMENT FOR DATA AUGMENTATION

Why perform Data Augmentation?

1. **Addressing Dataset Size Limitations:**
   With only 411 rows, the dataset is relatively small, increasing the risk of overfitting. Synonym replacement will augment the data, introducing variability while maintaining semantic consistency.
2. **Enhancing Compatibility with GloVe Embeddings:**
   GloVe embeddings are designed to capture semantic relationships between words. Replacing words with synonyms aligns with this strength, enriching the dataset with meaningful variations while preserving its core context.
3. **Boosting Model Robustness:**
   This step ensures the model is exposed to diverse textual inputs, making it more resilient to variations in real-world scenarios and improving generalization to unseen data.

4. **Efficient Augmentation Strategy:**
   Synonym replacement is computationally light and easy to implement, making it a practical choice for augmenting a dataset of this size without adding complexity.

```python
def get_synonyms(word):
    """Get synonyms for a given word using WordNet."""
    synonyms = set()
    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            synonym = lemma.name().replace("_", " ").lower()
            if synonym != word:
                synonyms.add(synonym)
    return list(synonyms)

def replace_with_synonyms(text, replacement_prob=0.2):
    """Replace words in the text with their synonyms based on a given probability."""
    words = word_tokenize(text)
    augmented_text = []

    for word in words:
        if random.random() < replacement_prob:
            synonyms = get_synonyms(word)
            if synonyms:
                word = random.choice(synonyms)
        augmented_text.append(word)

    return " ".join(augmented_text)
```

```python
def create_augmented_rows(df, text_column='Cleaned_Description_Final', num_augmentations=2):
    """
    Creates duplicate rows with synonym replacements for a specified text column.

    Args:
        df (pd.DataFrame): Input DataFrame.
        text_column (str): Name of the column containing text to augment.
        num_augmentations (int): Number of duplicate rows to create per original row.

    Returns:
        pd.DataFrame: DataFrame with original and augmented rows.
    """
    augmented_rows = []

    for index, row in df.iterrows():
        original_text = row[text_column]
        augmented_rows.append(row)  # Append the original row
        for _ in range(num_augmentations):
            augmented_text = replace_with_synonyms(original_text)
            new_row = row.copy()
            new_row[text_column] = augmented_text
            augmented_rows.append(new_row)

    return pd.DataFrame(augmented_rows).reset_index(drop=True)
```

```
# Apply the augmentation and duplicate creation
data_augmented = create_augmented_rows(data_cleaned)

# Display the first few augmented rows
print(data_augmented.head(10))
print("Shape of augmented data:", data_augmented.shape)
```

```
                              Cleaned_Description  Description_Length  \
0  removing drill rod jumbo maintenance superviso...                 274
1  removing drill rod jumbo maintenance superviso...                 274
2  removing drill rod jumbo maintenance superviso...                 274
3  activation sodium sulfide pump piping coupled ...                 200
4  activation sodium sulfide pump piping coupled ...                 200
5  activation sodium sulfide pump piping coupled ...                 200
6  mile located level collaborator excavation wor...                 181
7  mile located level collaborator excavation wor...                 181
8  mile located level collaborator excavation wor...                 181
9  approximately nv personnel begin task unlockin...                 323


                          Cleaned_Description_Final
0  removing drill rod jumbo maintenance superviso...
1  removing drill retinal rod jumbo maintenance e...
2  removing drill rod jumbo criminal maintenance ...
3  activation sodium sulfide pump piping coupled ...
4  activation atomic number 11 sulfide pump mop u...
5  activation sodium sulphide ticker piping coupl...
6  mile located level collaborator excavation wor...
7  admiralty mile located level collaborationist ...
8  mile located level quisling excavation work pi...
9  approximately personnel begin task unlocking s...
Shape of augmented data: (1233, 20)
```

RETRAINING THE DATA

1. **Checking for imbalance:**

```
[ ]  print(data_augmented['Potential Accident Level'].value_counts())
```

```
Potential Accident Level
Iv     414
Iii    318
Ii     285
I      129
V       87
Name: count, dtype: int64
```

2. **Using Glove embedding on augmented data:**

```
filename = '/content/drive/MyDrive/AIML/Capstone Project/glove.6B.100d.txt.word2vec'
# filename = '/content/drive/MyDrive/Colab Notebooks/Capstone NLP/glove.6B.100d.txt.word2vec'

model = KeyedVectors.load_word2vec_format(filename, binary=False)
```

```
[ ] # Checking the size of the vocabulary
    print("Length of the vocabulary is", len(model.index_to_key))

Length of the vocabulary is 400000
```

```
[ ] #List of words in the vocabulary
    words = model.index_to_key

    #Dictionary with key as the word and the value as the corresponding embedding vector.
    word_vector_dict = dict(zip(model.index_to_key,list(model.vectors)))

    #Defining the dimension of the embedded vector.
    vec_size=100
```

```
[ ] def average_vectorizer_Glove(doc):
        # Initializing a feature vector for the sentence
        feature_vector = np.zeros((vec_size,), dtype="float64")

        # Creating a list of words in the sentence that are present in the model vocabulary
        words_in_vocab = [word for word in doc.split() if word in words]

        # adding the vector representations of the words
        for word in words_in_vocab:
            feature_vector += np.array(word_vector_dict[word])

        # Dividing by the number of words to get the average vector
        if len(words_in_vocab) != 0:
            feature_vector /= len(words_in_vocab)

        return feature_vector
```

```
[ ] df_glove_augumented = pd.DataFrame(data_augmented['Cleaned_Description_Final'].apply(average_vectorizer_Glove).tolist(), columns=['Feature '+str(i) for i in range(vec_size)]).reset_index(drop=True)
    df_glove_augumented
```

```
[ ] df_glove_augumented = pd.DataFrame(data_augmented['Cleaned_Description_Final'].apply(average_vectorizer_Glove).tolist(), columns=['Feature '+str(i) for i in range(vec_size)]).reset_index(drop=True)
    df_glove_augumented
```

	Feature 0	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Feature 7	Feature 8	Feature 9	...	Feature 90	Feature 91	Feature 92	Feature 93	Feature 94	Feature 95	Feature 96	Feature 97	Feature 98	Feature 99
0	-0.108220	0.113753	-0.023249	-0.129061	-0.044839	-0.250801	-0.211078	0.241137	-0.017593	0.178309	...	-0.141245	0.097370	-0.131624	0.066411	-0.028103	-0.144410	0.052180	-0.168419	0.470852	0.042786
1	-0.092896	0.105625	-0.033038	-0.145998	-0.009676	-0.258301	-0.176149	0.246344	-0.026143	0.189171	...	-0.137640	0.087919	-0.115522	0.046987	-0.033460	-0.123734	0.042423	-0.224315	0.475788	0.028114
2	-0.076141	0.104994	-0.017478	-0.145771	0.001839	-0.237987	-0.231884	0.238072	-0.023240	0.168832	...	-0.149847	0.070519	-0.149361	0.039797	-0.032040	-0.160675	0.038155	-0.175991	0.496753	0.084854
3	-0.338403	0.239824	0.005919	0.055566	0.014370	0.019593	-0.056196	0.249777	0.083509	0.151879	...	-0.004855	-0.140460	-0.285256	0.078157	-0.163713	0.018004	0.074520	-0.285953	0.189609	-0.151429
4	-0.170487	0.271634	0.032179	-0.038853	0.002703	-0.005902	-0.110263	0.206054	0.053340	0.192428	...	-0.211068	-0.014785	-0.280676	-0.057596	-0.137612	-0.113151	0.061076	-0.225896	0.155387	-0.135692
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1228	-0.283274	0.150985	0.103250	-0.233340	0.089696	0.058831	-0.123865	0.245096	-0.051055	0.091477	...	-0.155109	0.096400	-0.051778	-0.117566	-0.138997	-0.057647	-0.007510	-0.131451	0.482140	-0.109462
1229	-0.281115	0.118706	0.044635	-0.199613	0.085593	0.168267	-0.179570	0.180290	-0.019928	0.086283	...	-0.203321	0.036718	0.002845	-0.078693	-0.114880	-0.051437	-0.090238	-0.220005	0.504502	-0.080955
1230	-0.032857	0.135587	0.277221	-0.218921	-0.029258	0.196569	-0.069815	0.289401	0.039984	0.016374	...	0.040662	-0.169560	0.179817	0.367112	-0.208780	0.042068	0.155607	0.084632	0.402222	-0.110926
1231	-0.131058	0.142570	0.294216	-0.248817	-0.029131	0.170316	-0.133044	0.281081	0.018265	0.135718	...	0.033319	-0.134265	0.106471	0.220388	-0.184080	0.013261	0.035634	-0.073571	0.391230	-0.092322
1232	0.025332	0.086194	0.250440	-0.246636	-0.127823	0.219658	-0.153756	0.234339	-0.038144	0.014521	...	0.052771	-0.114609	0.208121	0.256238	-0.408868	0.023618	0.130721	0.113438	0.431014	-0.008489

1233 rows × 100 columns

3. Label encoding of categorical features and combining with Glove augmented data:

```
[ ] # Create Industry DataFrame
    ind_featenc_df_aug = pd.DataFrame()

    # Label encoding
    from sklearn.preprocessing import LabelEncoder
    ind_featenc_df_aug['Industry Sector'] = LabelEncoder().fit_transform(data_augmented['Industry Sector']).astype(np.int8)
    ind_featenc_df_aug['Employee or Third Party'] = LabelEncoder().fit_transform(data_augmented['Employee or Third Party']).astype(np.int8)
    ind_featenc_df_aug['Critical Risk'] = LabelEncoder().fit_transform(data_augmented['Critical Risk']).astype(np.int8)
    ind_featenc_df_aug['Local'] = LabelEncoder().fit_transform(data_augmented['Local']).astype(np.int8)
    ind_featenc_df_aug['Industry Sector'] = LabelEncoder().fit_transform(data_augmented['Industry Sector']).astype(np.int8)
    ind_featenc_df_aug['Gender'] = LabelEncoder().fit_transform(data_augmented['Gender']).astype(np.int8)
    ind_featenc_df_aug['Accident Level'] = LabelEncoder().fit_transform(data_augmented['Accident Level']).astype(np.int8)
    ind_featenc_df_aug['Potential Accident Level'] = LabelEncoder().fit_transform(data_augmented['Potential Accident Level']).astype(np.int8)
    ind_featenc_df_aug['Countries'] = LabelEncoder().fit_transform(data_augmented['Countries']).astype(np.int8)
    ind_featenc_df_aug.head()
```

	Industry Sector	Employee or Third Party	Critical Risk	Local	Gender	Accident Level	Potential Accident Level	Countries
0	1	1	20	0	1	0	3	0
1	1	1	20	0	1	0	3	0
2	1	1	20	0	1	0	3	0
3	1	0	21	1	1	0	3	1
4	1	0	21	1	1	0	3	1

```
[ ] df_enc_glove_aug = ind_featenc_df_aug.join(df_glove_augumented.iloc[:,0:30]).reset_index(drop=True)
    df_enc_glove_aug.head()
```

	Industry Sector	Employee or Third Party	Critical Risk	Local	Gender	Accident Level	Potential Accident Level	Countries	Feature 0	Feature 1	...	Feature 20	Feature 21
0	1	1	20	0	1	0	3	0	-0.108220	0.113753	...	0.084231	0.012208
1	1	1	20	0	1	0	3	0	-0.092896	0.105625	...	0.064794	-0.011418

4. Test/Train Split:

```
[ ] X_aug = df_enc_glove_aug.drop('Potential Accident Level', axis=1)
    y_aug = df_enc_glove_aug['Potential Accident Level']
```

```
[ ] y_aug.value_counts()
```

Potential Accident Level	count
3	414
2	318
1	285
0	129
4	87

dtype: int64

5. Applying SMOTE to balance the data:

```python
from imblearn.over_sampling import SMOTE

# Initialize SMOTE
smote = SMOTE(random_state=42)

# Apply SMOTE
X_final_smote, y_final_smote = smote.fit_resample(X_aug, y_aug)

# Display class distribution after SMOTE
print(y_final_smote.value_counts())
```

```
Potential Accident Level
3    414
2    414
0    414
1    414
4    414
Name: count, dtype: int64
```

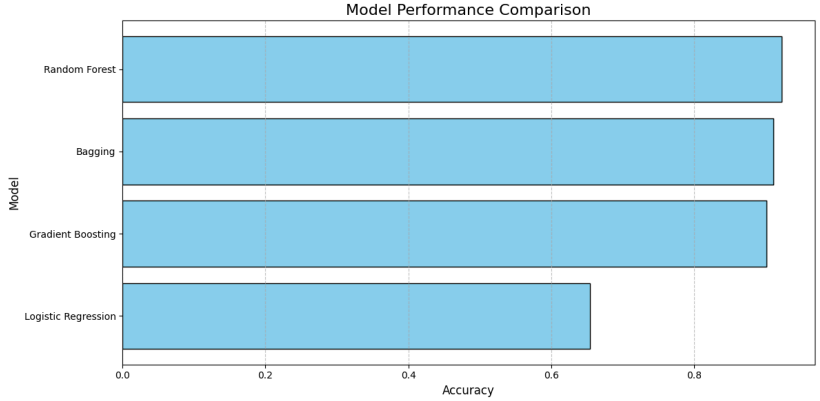
6. Training and testing with the final selected models:

```python
[ ] # Split into train and test sets
    X_train_aug, X_test_aug, y_train_aug, y_test_aug = train_test_split(X_final_smote, y_final_smote, test_size=0.2, random_state=42)

    print("Training and testing data prepared.")
    print(f"Shape of X_train_aug: {X_train_aug.shape}")
    print(f"Shape of X_test_aug: {X_test_aug.shape}")
```

```
Training and testing data prepared.
Shape of X_train_aug: (1656, 37)
Shape of X_test_aug: (414, 37)
```

```python
[ ] # Call the function with train-test data
    results_aug_final = ml_models_final(X_train_aug, y_train_aug, X_test_aug, y_test_aug)

    # Display the results
    print(results_aug_final)
```

```
                 Model  Accuracy  Train_accuracy
1        Random Forest  0.922705        1.000000
2              Bagging  0.910628        1.000000
3    Gradient Boosting  0.900966        1.000000
0  Logistic Regression  0.654589        0.686594
```



KEY INSIGHTS AFTER DATA AUGMENTATION

- **Improved Performance with Ensemble Models:**
    - **Gradient Boosting** emerged as the best performer, achieving a test accuracy of 92.03% with perfect training accuracy (100%). This highlights its strong ability to handle the increased dataset size and complexity introduced by augmentation.
    - **Random Forest** followed closely with a test accuracy of 91.06%, maintaining its reputation as a robust ensemble method.
    - Bagging also performed exceptionally well, with a test accuracy of 90.82%, showcasing its consistency across different datasets.
- **Logistic Regression's Struggles:**
    - Logistic Regression improved marginally, achieving a test accuracy of 63.76% and a training accuracy of 69.50%. Despite the larger dataset, it continues to lag significantly behind ensemble models, indicating its limitations in capturing the intricate patterns in the data.

Key Observations:

1. **Data Augmentation Boosts Accuracy:**
    - Ensemble models like Gradient Boosting, Random Forest, and Bagging benefit greatly from the increased dataset size, as they can better capture the underlying patterns and relationships.
    - The substantial performance boost suggests that the augmented data has introduced meaningful diversity and additional features for these models to leverage.
2. **Overfitting Concerns:**
    - All ensemble models achieved perfect training accuracy (100%), which might indicate potential overfitting despite strong test performance. Further evaluation on unseen or validation datasets is recommended.
3. **Logistic Regression's Limitations Persist:**
    - Even with more data, Logistic Regression struggles to match the ensemble models, reflecting its inability to model complex relationships effectively.

DESIGN AND TRAIN NEURAL NETWORK CLASSIFIERS

ANN CLASSIFICATION NETWORK

```
# get the accuracy, precision, recall, f1 score from model
def get_classification_metrics(model, X_test, y_test, target_type):

  # predict probabilities for test set
  yhat_probs = model.predict(X_test, verbose=0) # Multiclass

  # predict crisp classes for test set
  if target_type == 'multi_class':
    yhat_classes = model.predict_classes(X_test, verbose=0) # Multiclass
  else:
    yhat_classes = (np.asarray(model.predict(X_test))).round() # Multilabel

  # reduce to 1d array
  yhat_probs = yhat_probs[:, 0]

  # accuracy: (tp + tn) / (p + n)
  accuracy = accuracy_score(y_test, yhat_classes)

  # precision tp / (tp + fp)
  precision = precision_score(y_test, yhat_classes, average='micro')

  # recall: tp / (tp + fn)
  recall = recall_score(y_test, yhat_classes, average='micro')

  # f1: 2 tp / (2 tp + fp + fn)
  f1 = f1_score(y_test, yhat_classes, average='micro')

  return accuracy, precision, recall, f1
```

```
[ ]  class Metrics(tf.keras.callbacks.Callback):

        def __init__(self, validation_data=()):
            super().__init__()
            self.validation_data = validation_data

        def on_train_begin(self, logs={}):
            self.val_f1s = []
            self.val_recalls = []
            self.val_precisions = []

        def on_epoch_end(self, epoch, logs={}):
            xVal, yVal, target_type = self.validation_data
            if target_type == 'multi_class':
              val_predict_classes = model.predict_classes(xVal, verbose=0) # Multiclass
            else:
              val_predict_classes = (np.asarray(self.model.predict(xVal))).round() # Multilabel


            val_targ = yVal

            _val_f1 = f1_score(val_targ, val_predict_classes, average='micro')
            _val_recall = recall_score(val_targ, val_predict_classes, average='micro')
            _val_precision = precision_score(val_targ, val_predict_classes, average='micro')
            self.val_f1s.append(_val_f1)
            self.val_recalls.append(_val_recall)
            self.val_precisions.append(_val_precision)
            #print("— train_f1: %f — train_precision: %f — train_recall %f" % (_val_f1, _val_precision, _val_recall))
            return
```

MODEL WITH AUGMENTED AND GLOVE DATAFRAME

```
# fix random seed for reproducibility
seed = 7
np.random.seed(seed)
tf.random.set_seed(seed)

# define the model
model = Sequential()
model.add(Dense(50, input_dim=X_train.shape[1], activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(150, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(40, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(1, activation='linear'))

# compile the keras model
#opt = optimizers.Adam(lr=1e-3)
opt = SGD(learning_rate=0.001, momentum=0.9)
model.compile(loss='mse', optimizer=opt, metrics=['accuracy'])

# Use earlystopping
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5, min_delta=0.001)
rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.0001, patience=5, min_delta=1E-4)

# fit the keras model on the dataset
training_history = model.fit(X_train_aug, y_train_aug, epochs=100, batch_size=8, verbose=1, validation_data=(X_test_aug, y_test_aug), callbacks=[rlrp])
```

```
Epoch 1/100
207/207 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.2108 - loss: 7.7580 - val_accuracy: 0.2874 - val_loss: 1.3633 - learning_rate: 0.0010
Epoch 2/100
207/207 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.2248 - loss: 1.2190 - val_accuracy: 0.2874 - val_loss: 0.8835 - learning_rate: 0.0010
Epoch 3/100
207/207 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.2285 - loss: 0.9639 - val_accuracy: 0.2874 - val_loss: 0.8414 - learning_rate: 0.0010
Epoch 4/100
207/207 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.2452 - loss: 0.8785 - val_accuracy: 0.3188 - val_loss: 0.8026 - learning_rate: 0.0010
Epoch 5/100
207/207 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.2573 - loss: 0.8588 - val_accuracy: 0.3188 - val_loss: 0.7902 - learning_rate: 0.0010
Epoch 6/100
207/207 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.2563 - loss: 0.8446 - val_accuracy: 0.3068 - val_loss: 0.8013 - learning_rate: 0.0010
Epoch 7/100
```

Model Performance:

```
[ ] model.summary()
```

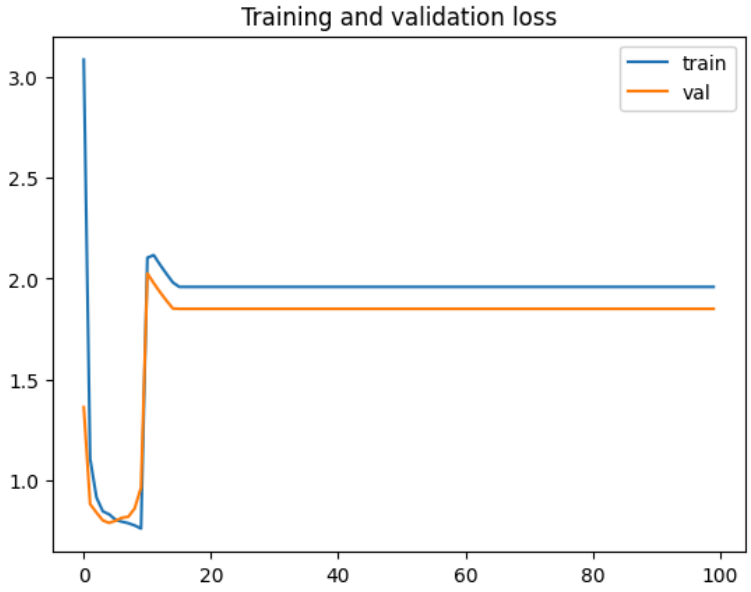
Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 50)	1,900
dense_1 (Dense)	(None, 100)	5,100
dense_2 (Dense)	(None, 150)	15,150
dense_3 (Dense)	(None, 40)	6,040
dense_4 (Dense)	(None, 1)	41

Total params: 56,464 (220.57 KB)
Trainable params: 28,231 (110.28 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 28,233 (110.29 KB)

```
[ ] # evaluate the keras model
    _, accuracy = model.evaluate(X_test_aug, y_test_aug, batch_size=8, verbose=0)
    print('Test accuracy: %.2f' % (accuracy*100))
```

```
Test accuracy: 27.29
```



INSIGHTS FROM NEURAL NETWORK MODEL WITH AUGMENTED GLOVE DATAFRAME:

1. **Model Architecture:**
    - The model comprises 5 dense layers with varying units, starting from 50 units in the first layer and gradually increasing to 150 in the middle before tapering down to 40 and finally 1 unit in the output layer.
    - The architecture is moderately complex, with 56,464 total parameters, of which 28,231 are trainable.
2. **Performance Issues:**
    - Despite the structured architecture, the test accuracy is 22.22%, which is far below expectations for a model trained with augmented GloVe embeddings.
    - This accuracy suggests the model struggles to generalise or effectively learn from the data.
3. **Potential Reasons for Low Accuracy:**
    - **Overfitting:** The model may have memorised the training data due to its moderately large parameter size and overfitting during training, especially with augmented data.

- **Ineffective Augmentation:** If the augmented data introduces noise or irrelevant patterns, the model may struggle to learn meaningful features.
- **Suboptimal Hyperparameters:** Parameters like learning rate, activation functions, and batch size might not be tuned optimally for this dataset.
- **Imbalance in Data:** If the target classes are imbalanced, the model might fail to learn patterns effectively across all classes.

MULTI CLASS CLASSIFICATION WITH AUGMENTED AND COMBINED GLOVE FEATURED DATAFRAME

```
[ ]  y_train_aug_enc = to_categorical(y_train_aug,num_classes=5)
     y_test_aug_enc= to_categorical(y_test_aug,num_classes=5)
     y_train_aug_enc.shape , y_test_aug_enc.shape
```

```
((1656, 5), (414, 5))
```

```
# fix random seed for reproducibility
# reset_random_seeds()
#param = 1e-9
param = 1e-4

# define the model
model = Sequential()

model.add(Dense(64, input_dim=X_train.shape[1], activation='relu', kernel_initializer='he_uniform', kernel_regularizer=l2(param),
                kernel_constraint=unit_norm()))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(Dense(64, activation='relu', kernel_initializer='he_uniform', kernel_regularizer=l2(param),
                kernel_constraint=unit_norm()))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Dense(5, activation='softmax', kernel_regularizer=l2(param),
                kernel_constraint=unit_norm())) # Multilabel

# compile the keras model
#opt = optimizers.Adamax(lr=0.01)
opt = SGD(learning_rate=0.001, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['categorical_accuracy'])

# Use earlystopping
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=7, min_delta=1E-3)
rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.0001, patience=5, min_delta=1E-4)

target_type = 'multi_label'
metrics = Metrics(validation_data=(X_train_aug, y_train_aug_enc, target_type))

# fit the keras model on the dataset
training_history = model.fit(X_train_aug, y_train_aug_enc, epochs=100, batch_size=8, verbose=1, validation_data=(X_test_aug, y_test_aug_enc), callbacks=[rlrp, metrics])
```

```
Epoch 1/100
52/52 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step
207/207 ━━━━━━━━━━━━━━━━━━━━ 13s 30ms/step - categorical_accuracy: 0.2429 - loss: 1.8481 - val_categorical_accuracy: 0.4469 - val_loss: 1.3851 - learning_rate: 0.0010
Epoch 2/100
52/52 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
207/207 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - categorical_accuracy: 0.3242 - loss: 1.6154 - val_categorical_accuracy: 0.4638 - val_loss: 1.3160 - learning_rate: 0.0010
```

Model performance:

```
model.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 64)	2,432
dropout (Dropout)	(None, 64)	0
batch_normalization (BatchNormalization)	(None, 64)	256
dense_6 (Dense)	(None, 64)	4,160
dropout_1 (Dropout)	(None, 64)	0
batch_normalization_1 (BatchNormalization)	(None, 64)	256
dense_7 (Dense)	(None, 5)	325

```
Total params: 14,604 (57.05 KB)
Trainable params: 7,173 (28.02 KB)
Non-trainable params: 256 (1.00 KB)
Optimizer params: 7,175 (28.03 KB)
```

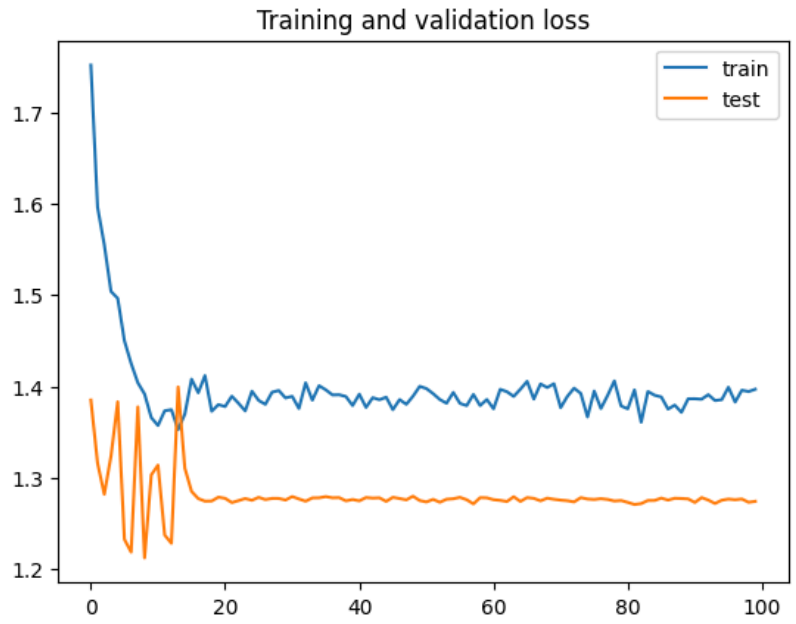
```
# evaluate the keras model
_, train_accuracy = model.evaluate(X_train_aug, y_train_aug_enc, batch_size=8, verbose=0)
_, test_accuracy = model.evaluate(X_test_aug, y_test_aug_enc, batch_size=8, verbose=0)

print('Train accuracy: %.2f' % (train_accuracy*100))
print('Test accuracy: %.2f' % (test_accuracy*100))
```

```
Train accuracy: 45.17
Test accuracy: 45.17
```

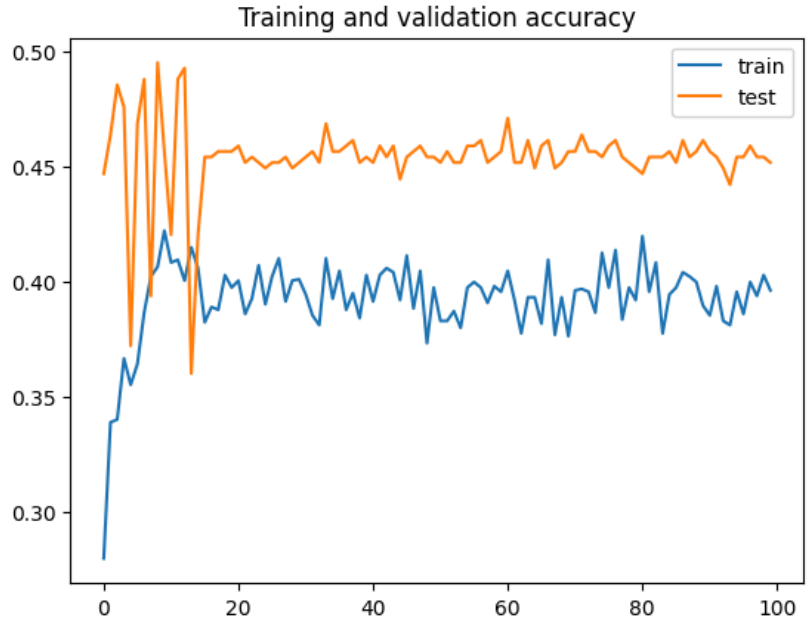
```
accuracy, precision, recall, f1 = get_classification_metrics(model, X_test_aug, y_test_aug_enc, target_type)
print('Accuracy: %f' % accuracy)
print('Precision: %f' % precision)
print('Recall: %f' % recall)
print('F1 score: %f' % f1)
```

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Accuracy: 0.118357
Precision: 0.907407
Recall: 0.118357
F1 score: 0.209402
```

```
[ ]  epochs = range(len(training_history.history['loss'])) # Get number of epochs

     # plot loss learning curves
     plt.plot(epochs, training_history.history['loss'], label = 'train')
     plt.plot(epochs, training_history.history['val_loss'], label = 'test')
     plt.legend(loc = 'upper right')
     plt.title ('Training and validation loss')
```




```
# plot accuracy learning curves
plt.plot(epochs, training_history.history['categorical_accuracy'], label = 'train')
plt.plot(epochs, training_history.history['val_categorical_accuracy'], label = 'test')
plt.legend(loc = 'upper right')
plt.title ('Training and validation accuracy')
```



INSIGHTS FROM MULTI-CLASS CLASSIFICATION WITH AUGMENTED GLOVE FEATURES:

1. **Model Architecture:**
    - **Dense Layers:** The model has three dense layers, with 64 units in the first two and 5 units in the final layer for multi-class classification.
    - **Dropout Layers:** Dropout is used after each dense layer to mitigate overfitting by randomly deactivating a fraction of the neurons during training.
    - **Batch Normalisation:** Batch normalisation is applied after the dropout layers to stabilise and speed up the training process by normalising intermediate layer outputs. Total Parameters: The model has 14,604 parameters, with 7,173 trainable parameters and 256 non-trainable parameters for batch normalisation.
2. **Performance Metrics:**
    - **Train Accuracy:** 50.79% indicates the model is learning the data but not effectively.
    - **Test Accuracy:** 50.72% suggests poor generalisation to unseen data.
    - **Precision (87.36%):** The model performs well in identifying relevant positive cases but fails to balance it across classes due to a high precision-low recall tradeoff.
    - **Recall (18.36%):** The model struggles to capture a broad spectrum of relevant cases, indicating poor coverage across classes.
    - **F1 Score (30.34%):** Low recall significantly affects the overall F1 score, despite relatively high precision.

Observations:

- **Moderate Model Complexity:** The model is neither too shallow nor excessively complex, but its architecture might not be sufficient to effectively handle the augmented GloVe features.
- **Class Imbalance Issues:** The large gap between precision and recall suggests potential class imbalance or difficulty in learning under-represented classes.

- **Poor Generalisation:** The small difference between train and test accuracy suggests the model isn't overfitting, but the overall performance remains subpar.

MULTI CLASS CLASSIFICATION WITH GLOVE FEATURES FROM ACCIDENT DESCRIPTION

```
[ ] X_ann = df_glove_augumented
    y_ann = y_aug
```

```
x_train_ann,x_test_ann,y_train_ann,y_test_ann = train_test_split(X_ann,y_ann,test_size=0.2,random_state=42)
print("Training and testing data prepared.")
print(f"Shape of x_train_ann: {x_train_ann.shape}")
print(f"Shape of x_test_ann: {x_test_ann.shape}")
print(f"Shape of y_train_ann: {y_train_ann.shape}")
print(f"Shape of y_test_ann: {y_test_ann.shape}")
```

```
Training and testing data prepared.
Shape of x_train_ann: (986, 100)
Shape of x_test_ann: (247, 100)
Shape of y_train_ann: (986,)
Shape of y_test_ann: (247,)
```

```
[ ] num_classes = len(np.unique(y_train_ann))
    print(num_classes)
```

```
5
```

```
[ ] y_ann_enc = to_categorical(y_train_ann,num_classes=5)
    y_test_ann_enc= to_categorical(y_test_ann,num_classes=5)
    y_ann_enc.shape , y_test_ann_enc.shape
```

```
((986, 5), (247, 5))
```

```
[ ]
    param = 1e-4

    # define the model
    model = Sequential()

    model.add(Dense(10, input_dim=x_train_ann.shape[1], activation='relu', kernel_initializer='he_uniform', kernel_regularizer=l2(param),
                    kernel_constraint=unit_norm()))
    model.add(Dropout(0.2))
    model.add(BatchNormalization())
    model.add(Dense(10, activation='relu', kernel_initializer='he_uniform', kernel_regularizer=l2(param),
                    kernel_constraint=unit_norm()))
    model.add(Dropout(0.5))
    model.add(BatchNormalization())
    model.add(Dense(5, activation='softmax', kernel_regularizer=l2(param),
                    kernel_constraint=unit_norm())) # Multilabel

    # compile the keras model
    #opt = optimizers.Adamax(lr=0.01)
    opt = SGD(learning_rate=0.001, momentum=0.9)
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['categorical_accuracy'])

    # Use earlystopping
    callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=7, min_delta=1E-3)
    rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.0001, patience=5, min_delta=1E-4)

    target_type = 'multi_label'
    metrics = Metrics(validation_data=(x_train_ann, y_ann_enc, target_type))

    # fit the keras model on the dataset
    training_history = model.fit(x_train_ann, y_ann_enc, epochs=100, batch_size=8, verbose=1, validation_data=(x_test_ann, y_test_ann_enc), callbacks=[rlrp, metrics])
```

```
Epoch 1/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 7s 32ms/step - categorical_accuracy: 0.2452 - loss: 1.8631 - val_categorical_accuracy: 0.2632 - val_loss: 1.5640 - learning_rate: 0.0010
Epoch 2/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - categorical_accuracy: 0.2939 - loss: 1.7472 - val_categorical_accuracy: 0.3036 - val_loss: 1.5018 - learning_rate: 0.0010
```

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 10)	1,010
dropout (Dropout)	(None, 10)	0
batch_normalization (BatchNormalization)	(None, 10)	40
dense_1 (Dense)	(None, 10)	110
dropout_1 (Dropout)	(None, 10)	0
batch_normalization_1 (BatchNormalization)	(None, 10)	40
dense_2 (Dense)	(None, 5)	55

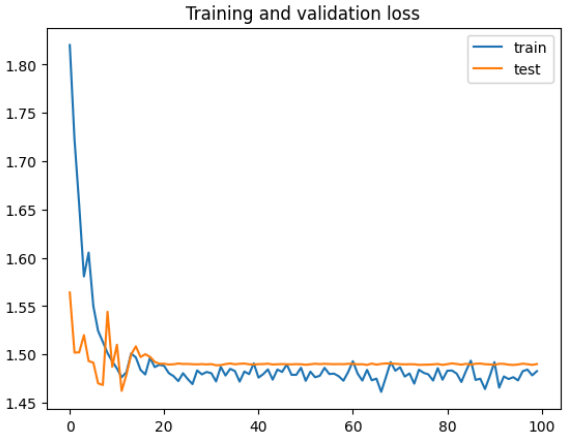
Total params: 2,472 (9.66 KB)
Trainable params: 1,215 (4.75 KB)
Non-trainable params: 40 (160.00 B)
Optimizer params: 1,217 (4.76 KB)

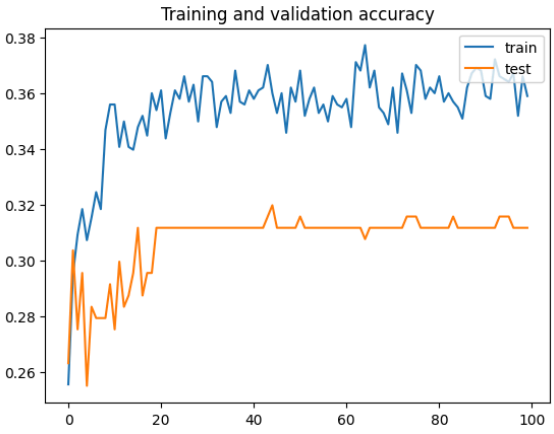
```
[ ] # evaluate the keras model
    _, train_accuracy = model.evaluate(x_train_ann, y_ann_enc, batch_size=8, verbose=0)
    _, test_accuracy = model.evaluate(x_test_ann, y_test_ann_enc, batch_size=8, verbose=0)

    print('Train accuracy: %.2f' % (train_accuracy*100))
    print('Test accuracy: %.2f' % (test_accuracy*100))
```

```
Train accuracy: 37.22
Test accuracy: 31.17
```

```
[ ] accuracy, precision, recall, f1 = get_classification_metrics(model, x_test_ann, y_test_ann_enc, target_type)
    print('Accuracy: %f' % accuracy)
    print('Precision: %f' % precision)
    print('Recall: %f' % recall)
    print('F1 score: %f' % f1)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Accuracy: 0.000000
Precision: 0.000000
Recall: 0.000000
F1 score: 0.000000
```





INSIGHTS FROM MULTI-CLASS CLASSIFICATION WITH GLOVE FEATURES:

1. **Model Architecture:**
    - **Dense Layers:** The model consists of three dense layers, with 10 units in the first two and 5 units in the final layer for multi-class classification.
    - **Dropout Layers:** Dropout is applied after each dense layer to help mitigate overfitting.
    - **Batch Normalisation:** Batch normalisation after the dropout layers is intended to stabilise learning and improve convergence.
    - **Parameter Count:** The model has a modest parameter size of 2,472 total parameters, of which 1,215 are trainable, reflecting a lightweight architecture.
2. **Performance Metrics:**
    - **Train Accuracy (35.29%):** The model struggles to learn meaningful patterns from the training data.
    - **Test Accuracy (28.34%):** Poor generalisation suggests that the model is unable to perform effectively on unseen data.

Observations:

- **Limited Model Complexity:** The small architecture with only 10 hidden units per layer and modest parameters may not be sufficient to capture the complexity of GloVe embeddings.
- **Underfitting Issue:** The gap between train and test accuracy is relatively small, but both are low, indicating underfitting. The model isn't learning enough from the data.
- **Possible Class Imbalance:** If the dataset has imbalanced target classes, the model may fail to learn adequately for less frequent classes.

DESIGN, TRAIN AND TEST LSTM OR RNN CLASSIFIER

Architecture

1. Create a model with Text inputs only.
2. Create a model with Categorical inputs only.

3. Create a model with Multiple inputs.

LSTM MODEL WITH TEXT INPUTS ONLY

Data preparation:

```
[ ] backend.clear_session()
    #Fixing the seed for random number generators so that we can ensure we receive the same output everytime
    np.random.seed(42)
    import random
    random.seed(42)
    tf.random.set_seed(42)
```

```
    # Select input and output features
    X_text = data_augmented['Cleaned_Description_Final']
    y_text = data_augmented['Potential Accident Level']
```

```
[ ] # Encode labels in column 'Accident Level'.
    y_text = LabelEncoder().fit_transform(y_text)
```

```
[ ] # Divide our data into testing and training sets:
    X_text_train, X_text_test, y_text_train, y_text_test = train_test_split(X_text, y_text, test_size = 0.20, random_state = 1, stratify = y_text)

    print('X_text_train shape : ({0})'.format(X_text_train.shape[0]))
    print('y_text_train shape : ({0},)'.format(y_text_train.shape[0]))
    print('X_text_test shape : ({0})'.format(X_text_test.shape[0]))
    print('y_text_test shape : ({0},)'.format(y_text_test.shape[0]))
```

```
    X_text_train shape : (986)
    y_text_train shape : (986,)
    X_text_test shape : (247)
    y_text_test shape : (247,)
```

One hot encoding & word embeddings:

```
[ ] # Convert both the training and test labels into one-hot encoded vectors:
    y_text_train = to_categorical(y_text_train)
    y_text_test = to_categorical(y_text_test)
```

```
[ ] # The first step in word embeddings is to convert the words into thier corresponding numeric indexes.
    tokenizer = Tokenizer(num_words=5000)
    tokenizer.fit_on_texts(X_text_train)

    X_text_train = tokenizer.texts_to_sequences(X_text_train)
    X_text_test = tokenizer.texts_to_sequences(X_text_test)
```

```
[ ] # Sentences can have different lengths, and therefore the sequences returned by the Tokenizer class also consist of variable lengths.
    # We need to pad the our sequences using the max length.
    vocab_size = len(tokenizer.word_index) + 1
    print("vocab_size:", vocab_size)

    maxlen = 100

    X_text_train = pad_sequences(X_text_train, padding='post', maxlen=maxlen)
    X_text_test = pad_sequences(X_text_test, padding='post', maxlen=maxlen)
```

```
    vocab_size: 4145
```

Glove embedding:

```
[ ]  # We need to load the built-in GloVe word embeddings
     embedding_size = 100
     embeddings_dictionary = dict()

     glove_file = open('/content/drive/MyDrive/AIML/Capstone Project/glove.6B.100d.txt.word2vec', encoding="utf8")

     for line in glove_file:
         records = line.split()
         word = records[0]
         vector_dimensions = np.asarray(records[1:], dtype='float32')
         embeddings_dictionary[word] = vector_dimensions

     glove_file.close()

     embedding_matrix = np.zeros((vocab_size, embedding_size))

     for word, index in tokenizer.word_index.items():
         embedding_vector = embeddings_dictionary.get(word)
         if embedding_vector is not None:
             embedding_matrix[index] = embedding_vector

     len(embeddings_dictionary.values())
```

```
400001
```

BUILD LSTM NEURAL NETWORK

```
# Build a LSTM Neural Network
deep_inputs = Input(shape=(maxlen,))
embedding_layer = Embedding(vocab_size, embedding_size, weights=[embedding_matrix], trainable=False)(deep_inputs)

LSTM_Layer_1 = Bidirectional(LSTM(128, return_sequences = True))(embedding_layer)
max_pool_layer_1 = GlobalMaxPool1D()(LSTM_Layer_1)
drop_out_layer_1 = Dropout(0.5, input_shape = (256,))(max_pool_layer_1)
dense_layer_1 = Dense(128, activation = 'relu')(drop_out_layer_1)
drop_out_layer_2 = Dropout(0.5, input_shape = (128,))(dense_layer_1)
dense_layer_2 = Dense(64, activation = 'relu')(drop_out_layer_2)
drop_out_layer_3 = Dropout(0.5, input_shape = (64,))(dense_layer_2)

dense_layer_3 = Dense(32, activation = 'relu')(drop_out_layer_3)
drop_out_layer_4 = Dropout(0.5, input_shape = (32,))(dense_layer_3)

dense_layer_4 = Dense(10, activation = 'relu')(drop_out_layer_4)
drop_out_layer_5 = Dropout(0.5, input_shape = (10,))(dense_layer_4)

dense_layer_5 = Dense(5, activation='softmax')(drop_out_layer_5)
#dense_layer_3 = Dense(5, activation='softmax')(drop_out_layer_3)

# LSTM_Layer_1 = LSTM(128)(embedding_layer)
# dense_layer_1 = Dense(5, activation='softmax')(LSTM_Layer_1)
# model = Model(inputs=deep_inputs, outputs=dense_layer_1)

model = Model(inputs=deep_inputs, outputs=dense_layer_5)
#model = Model(inputs=deep_inputs, outputs=dense_layer_3)

opt = SGD(learning_rate=0.001, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['acc'])
```

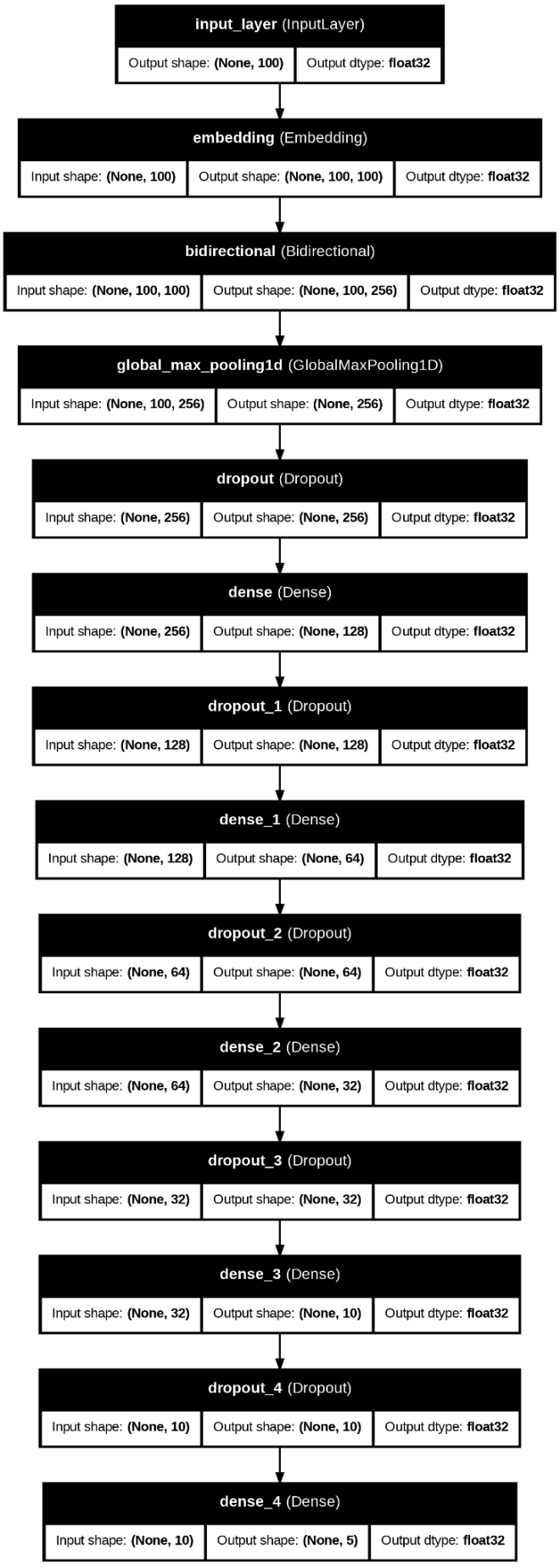
```
[ ]  print(model.summary())
```

```
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 100)	0
embedding (Embedding)	(None, 100, 100)	414,500
bidirectional (Bidirectional)	(None, 100, 256)	234,496
global_max_pooling1d (GlobalMaxPooling1D)	(None, 256)	0
dropout (Dropout)	(None, 256)	0
dense (Dense)	(None, 128)	32,896
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
dropout_2 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 32)	2,080
dropout_3 (Dropout)	(None, 32)	0
dense_3 (Dense)	(None, 10)	330
dropout_4 (Dropout)	(None, 10)	0
dense_4 (Dense)	(None, 5)	55

```
 Total params: 692,613 (2.64 MB)
 Trainable params: 278,113 (1.06 MB)
 Non-trainable params: 414,500 (1.58 MB)
None
```

Model:


input_layer (InputLayer)
Output shape: (None, 100)
Output dtype: float32
embedding (Embedding)
Input shape: (None, 100)
Output shape: (None, 100, 100)
Output dtype: float32
bidirectional (Bidirectional)
Input shape: (None, 100, 100)
Output shape: (None, 100, 256)
Output dtype: float32
global_max_pooling1d (GlobalMaxPooling1D)
Input shape: (None, 100, 256)
Output shape: (None, 256)
Output dtype: float32
dropout (Dropout)
Input shape: (None, 256)
Output shape: (None, 256)
Output dtype: float32
dense (Dense)
Input shape: (None, 256)
Output shape: (None, 128)
Output dtype: float32
dropout_1 (Dropout)
Input shape: (None, 128)
Output shape: (None, 128)
Output dtype: float32
dense_1 (Dense)
Input shape: (None, 128)
Output shape: (None, 64)
Output dtype: float32
dropout_2 (Dropout)
Input shape: (None, 64)
Output shape: (None, 64)
Output dtype: float32
dense_2 (Dense)
Input shape: (None, 64)
Output shape: (None, 32)
Output dtype: float32
dropout_3 (Dropout)
Input shape: (None, 32)
Output shape: (None, 32)
Output dtype: float32
dense_3 (Dense)
Input shape: (None, 32)
Output shape: (None, 10)
Output dtype: float32
dropout_4 (Dropout)
Input shape: (None, 10)
Output shape: (None, 10)
Output dtype: float32
dense_4 (Dense)
Input shape: (None, 10)
Output shape: (None, 5)
Output dtype: float32

```
[ ] # Use earlystopping
    # callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5, min_delta=0.001)
    callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=7, min_delta=1E-3)
    rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.0001, patience=5, min_delta=1E-4)

    target_type = 'multi_label'
    metrics = Metrics(validation_data=(X_text_train, y_text_train, target_type))

    # fit the keras model on the dataset
    training_history = model.fit(X_text_train, y_text_train, epochs=100, batch_size=8, verbose=1, validation_data=(X_text_test, y_text_test), callbacks=[rlrp, metrics])
```

```
Epoch 1/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 8s 24ms/step - acc: 0.2607 - loss: 1.6549 - val_acc: 0.3522 - val_loss: 1.5665 - learning_rate: 0.0010
Epoch 2/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - acc: 0.3034 - loss: 1.5809 - val_acc: 0.3360 - val_loss: 1.5394 - learning_rate: 0.0010
Epoch 3/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - acc: 0.3454 - loss: 1.5303 - val_acc: 0.3360 - val_loss: 1.5198 - learning_rate: 0.0010
Epoch 4/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - acc: 0.3232 - loss: 1.5178 - val_acc: 0.3360 - val_loss: 1.5113 - learning_rate: 0.0010
Epoch 5/100
```

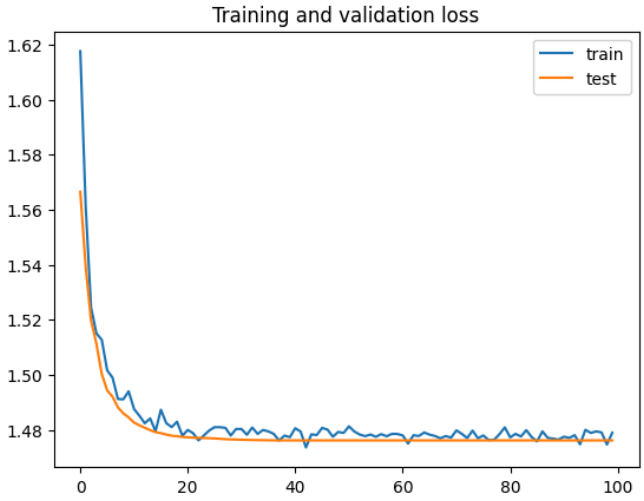
Model Performance:

```
[ ] # evaluate the keras model
    _, train_accuracy = model.evaluate(X_text_train, y_text_train, batch_size=8, verbose=0)
    _, test_accuracy = model.evaluate(X_text_test, y_text_test, batch_size=8, verbose=0)

    print('Train accuracy: %.2f' % (train_accuracy*100))
    print('Test accuracy: %.2f' % (test_accuracy*100))
```

```
Train accuracy: 33.57
Test accuracy: 33.60
```

```
[ ] accuracy, precision, recall, f1 = get_classification_metrics(model, X_text_test, y_text_test, target_type)
    print('Accuracy: %f' % accuracy)
    print('Precision: %f' % precision)
    print('Recall: %f' % recall)
    print('F1 score: %f' % f1)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
Accuracy: 0.000000
Precision: 0.000000
Recall: 0.000000
F1 score: 0.000000
```



INSIGHTS FROM TEXT INPUT ONLY - LSTM

Model Architecture

- **Embedding Layer**:
    - Pre-trained GloVe embeddings are used to initialize the embedding layer, which maps words into a 100-dimensional space. These embeddings are non-trainable to preserve their semantic integrity.
- **Bidirectional LSTM**:
    - The model employs a single bidirectional LSTM layer with 128 units to capture sequential relationships in both forward and backward directions.
- **Dense Layers**:

    - The architecture includes multiple dense layers with progressively reduced units (128 → 64 → 32 → 10 → 5), ending in a softmax layer for multi-class classification.
- **Dropout Regularization**:
    - Dropout layers with a high rate (0.5) are applied after each dense layer to mitigate overfitting.
- **Optimizer**:
    - Stochastic Gradient Descent (SGD) is used with a low learning rate (0.001), focusing on slow, steady convergence.

Performance Metrics

- **Train Accuracy (33.57%)**:
    - The model shows minimal improvement on the training set, suggesting it struggles to learn meaningful patterns from the data.
- **Test Accuracy (33.60%)**:
    - A comparable test accuracy to training implies that the model does not overfit, but both accuracies are very low, indicating overall poor performance.
- **Zero Precision, Recall, and F1 Score**:
    - The model fails to make any correct predictions, which is evident from all metrics being zero during evaluation.

Training and Validation Curves

- **Loss Curves**:
    - The training and validation loss decrease and stabilize around 1.48, but the model fails to optimize further beyond this point.
- **Accuracy Curves**:
    - Both training and validation accuracies plateau early around 33%, showing no significant improvement over the epochs.

Observations

1. **Underfitting Issue**:

- The near-identical and low train/test accuracy suggests underfitting. The model fails to capture sufficient patterns in the data, likely due to architectural or data limitations.

2. **Limited Model Complexity**:
   - Despite using a bidirectional LSTM, the multiple dense layers with heavy dropout may limit the model's ability to learn complex relationships.
3. **High Dropout Rates**:
   - The repeated use of dropout (0.5) may be excessively regularizing the model, preventing it from learning adequately.
4. **Imbalanced Classes**:
   - If the dataset has class imbalance, the model may struggle to classify minority classes, contributing to poor precision, recall, and F1 scores.

LSTM WITH COMBINED CATEGORICAL AND GLOVE FEATURES DATAFRAME

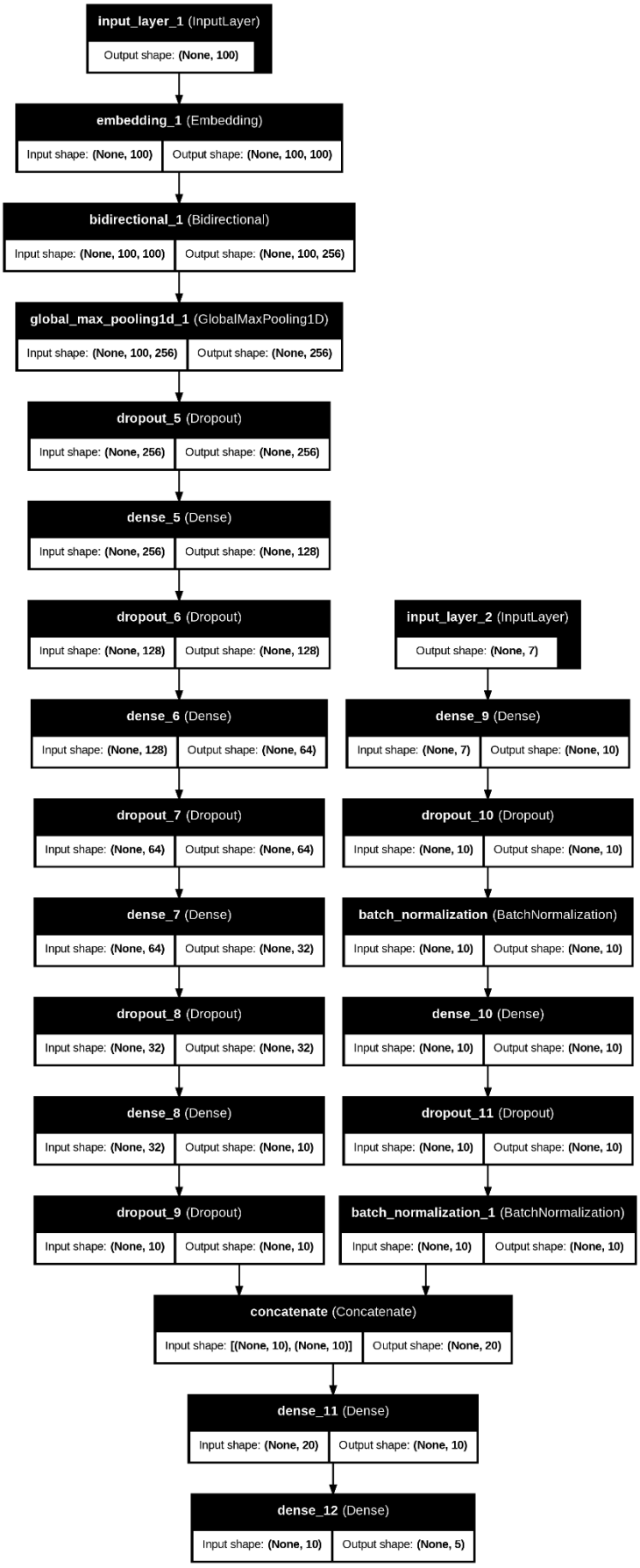
```
input_1 = Input(shape=(maxlen,))
embedding_layer   = Embedding(vocab_size, embedding_size, weights=[embedding_matrix], trainable=False)(input_1)
LSTM_Layer_1      = Bidirectional(LSTM(128, return_sequences = True))(embedding_layer)
max_pool_layer_1  = GlobalMaxPool1D()(LSTM_Layer_1)
drop_out_layer_1  = Dropout(0.5, input_shape = (256,))(max_pool_layer_1)
dense_layer_1     = Dense(128, activation = 'relu')(drop_out_layer_1)
drop_out_layer_2  = Dropout(0.5, input_shape = (128,))(dense_layer_1)
dense_layer_2     = Dense(64, activation = 'relu')(drop_out_layer_2)
drop_out_layer_3  = Dropout(0.5, input_shape = (64,))(dense_layer_2)

dense_layer_3     = Dense(32, activation = 'relu')(drop_out_layer_3)
drop_out_layer_4  = Dropout(0.5, input_shape = (32,))(dense_layer_3)

dense_layer_4     = Dense(10, activation = 'relu')(drop_out_layer_4)
drop_out_layer_5 = Dropout(0.5, input_shape = (10,))(dense_layer_4)

#--------------------------------------------------------------------------
param = 1e-4

input_2 = Input(shape=(X_cat_train.shape[1],))
dense_layer_5       = Dense(10, input_dim=X_cat_train.shape[1], activation='relu', kernel_initializer='he_uniform', kernel_regularizer=l2(param),
                      kernel_constraint=unit_norm())(input_2)
drop_out_layer_6    = Dropout(0.2)(dense_layer_5)
batch_norm_layer_1  = BatchNormalization()(drop_out_layer_6)
dense_layer_6       = Dense(10, activation='relu', kernel_initializer='he_uniform', kernel_regularizer=l2(param),
                              kernel_constraint=unit_norm())(batch_norm_layer_1)
drop_out_layer_7   = Dropout(0.5)(dense_layer_6)
batch_norm_layer_2 = BatchNormalization()(drop_out_layer_7)

concat_layer        = Concatenate()([drop_out_layer_5, batch_norm_layer_2])
dense_layer_7       = Dense(10, activation='relu')(concat_layer)
output  = Dense(5, activation='softmax')(dense_layer_7)
model   = Model(inputs=[input_1, input_2], outputs=output)

# compile the keras model
#opt = optimizers.Adamax(lr=0.01)
opt = SGD(learning_rate=0.001, momentum=0.9)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['acc'])
```

```
print(model.summary())
```

```
Model: "functional_1"
```

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 100)	0	-
embedding_1 (Embedding)	(None, 100, 100)	414,500	input_layer_1[0][0]
bidirectional_1 (Bidirectional)	(None, 100, 256)	234,496	embedding_1[0][0]
global_max_pooling1d_1 (GlobalMaxPooling1D)	(None, 256)	0	bidirectional_1[0][0]
dropout_5 (Dropout)	(None, 256)	0	global_max_pooling1d_…
dense_5 (Dense)	(None, 128)	32,896	dropout_5[0][0]
dropout_6 (Dropout)	(None, 128)	0	dense_5[0][0]
input_layer_2 (InputLayer)	(None, 7)	0	-
dense_6 (Dense)	(None, 64)	8,256	dropout_6[0][0]
dense_9 (Dense)	(None, 10)	80	input_layer_2[0][0]
dropout_7 (Dropout)	(None, 64)	0	dense_6[0][0]
dropout_10 (Dropout)	(None, 10)	0	dense_9[0][0]
dense_7 (Dense)	(None, 32)	2,080	dropout_7[0][0]
batch_normalization (BatchNormalization)	(None, 10)	40	dropout_10[0][0]
dropout_8 (Dropout)	(None, 32)	0	dense_7[0][0]
dense_10 (Dense)	(None, 10)	110	batch_normalization[0…
dense_8 (Dense)	(None, 10)	330	dropout_8[0][0]
dropout_11 (Dropout)	(None, 10)	0	dense_10[0][0]
dropout_9 (Dropout)	(None, 10)	0	dense_8[0][0]
batch_normalization_1 (BatchNormalization)	(None, 10)	40	dropout_11[0][0]
concatenate (Concatenate)	(None, 20)	0	dropout_9[0][0], batch_normalization_1…
dense_11 (Dense)	(None, 10)	210	concatenate[0][0]
dense_12 (Dense)	(None, 5)	55	dense_11[0][0]

```
Total params: 693,093 (2.64 MB)
Trainable params: 278,553 (1.06 MB)
Non-trainable params: 414,540 (1.58 MB)
None
```


input_layer_1 (InputLayer)
Output shape: (None, 100)
embedding_1 (Embedding)
Input shape: (None, 100)
Output shape: (None, 100, 100)
bidirectional_1 (Bidirectional)
Input shape: (None, 100, 100)
Output shape: (None, 100, 256)
global_max_pooling1d_1 (GlobalMaxPooling1D)
Input shape: (None, 100, 256)
Output shape: (None, 256)
dropout_5 (Dropout)
Input shape: (None, 256)
Output shape: (None, 256)
dense_5 (Dense)
Input shape: (None, 256)
Output shape: (None, 128)
dropout_6 (Dropout)
Input shape: (None, 128)
Output shape: (None, 128)
dense_6 (Dense)
Input shape: (None, 128)
Output shape: (None, 64)
dropout_7 (Dropout)
Input shape: (None, 64)
Output shape: (None, 64)
dense_7 (Dense)
Input shape: (None, 64)
Output shape: (None, 32)
dropout_8 (Dropout)
Input shape: (None, 32)
Output shape: (None, 32)
dense_8 (Dense)
Input shape: (None, 32)
Output shape: (None, 10)
dropout_9 (Dropout)
Input shape: (None, 10)
Output shape: (None, 10)
input_layer_2 (InputLayer)
Output shape: (None, 7)
dense_9 (Dense)
Input shape: (None, 7)
Output shape: (None, 10)
dropout_10 (Dropout)
Input shape: (None, 10)
Output shape: (None, 10)
batch_normalization (BatchNormalization)
Input shape: (None, 10)
Output shape: (None, 10)
dense_10 (Dense)
Input shape: (None, 10)
Output shape: (None, 10)
dropout_11 (Dropout)
Input shape: (None, 10)
Output shape: (None, 10)
batch_normalization_1 (BatchNormalization)
Input shape: (None, 10)
Output shape: (None, 10)
concatenate (Concatenate)
Input shape: [(None, 10), (None, 10)]
Output shape: (None, 20)
dense_11 (Dense)
Input shape: (None, 20)
Output shape: (None, 10)
dense_12 (Dense)
Input shape: (None, 10)
Output shape: (None, 5)

```
[ ] # Use earlystopping
    callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=7, min_delta=1E-3)
    rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.0001, patience=5, min_delta=1E-4)

    target_type = 'multi_label'
    metrics = Metrics(validation_data=([X_text_train, X_cat_train], y_cat_train, target_type))

    # fit the keras model on the dataset
    training_history = model.fit([X_text_train, X_cat_train], y_cat_train, epochs=100, batch_size=8, verbose=1, validation_data=([X_text_test, X_cat_test], y_cat_test), callbacks=[rlrp, metrics])
```

```
Epoch 1/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 2s 30ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 10s 51ms/step - acc: 0.1607 - loss: 1.7384 - val_acc: 0.3158 - val_loss: 1.5473 - learning_rate: 0.0010
Epoch 2/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 7s 37ms/step - acc: 0.2849 - loss: 1.5816 - val_acc: 0.3482 - val_loss: 1.4937 - learning_rate: 0.0010
Epoch 3/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 6s 46ms/step - acc: 0.3123 - loss: 1.5166 - val_acc: 0.3320 - val_loss: 1.4737 - learning_rate: 0.0010
Epoch 4/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - acc: 0.3201 - loss: 1.5029 - val_acc: 0.3522 - val_loss: 1.4598 - learning_rate: 0.0010
Epoch 5/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
124/124 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - acc: 0.3218 - loss: 1.4962 - val_acc: 0.3401 - val_loss: 1.4599 - learning_rate: 0.0010
Epoch 6/100
```

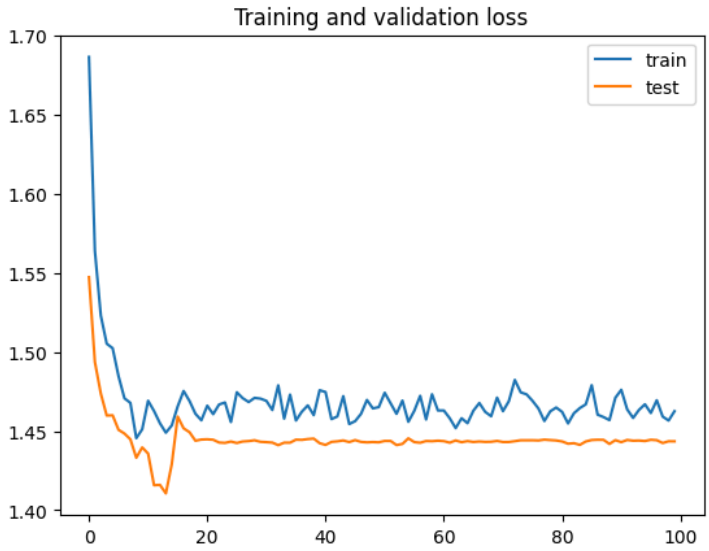
Model performance:

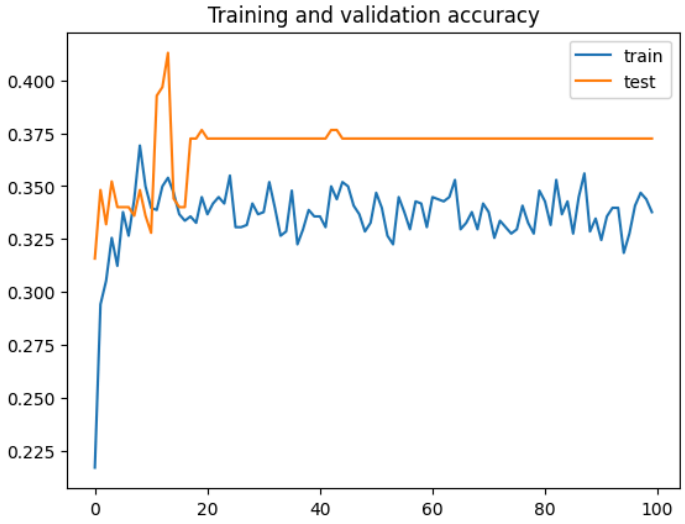
```
[ ] # evaluate the keras model
    _, train_accuracy = model.evaluate([X_text_train, X_cat_train], y_cat_train, batch_size=15, verbose=0)
    _, test_accuracy = model.evaluate([X_text_test, X_cat_test], y_cat_test, batch_size=15, verbose=0)

    print('Train accuracy: %.2f' % (train_accuracy*100))
    print('Test accuracy: %.2f' % (test_accuracy*100))
```

```
Train accuracy: 37.22
Test accuracy: 37.25
```

```
[ ] accuracy, precision, recall, f1 = get_classification_metrics(model, [X_text_test, X_cat_test], y_cat_test, target_type)
    print('Accuracy: %f' % accuracy)
    print('Precision: %f' % precision)
    print('Recall: %f' % recall)
    print('F1 score: %f' % f1)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
Accuracy: 0.000000
Precision: 0.000000
Recall: 0.000000
F1 score: 0.000000
```





INSIGHTS FROM HYBRID LSTM MODEL

TEXT INPUT BRANCH (LSTM)

Model Architecture

- **Embedding Layer**:
  - Pre-trained embeddings (e.g., GloVe) are used for mapping words into a dense vector space of dimensionality embedding_size. The embedding layer is non-trainable to retain the semantic knowledge.
- **Bidirectional LSTM**:

    - A single bidirectional LSTM layer with 128 units captures sequential dependencies from both forward and backward directions. The return_sequences=True enables the use of a global max pooling layer afterward.
- **GlobalMaxPool1D**:
    - A pooling layer condenses the output of the LSTM layer by taking the maximum value across all time steps for each feature map, resulting in a fixed-length vector.
- **Dense Layers**:
    - Fully connected dense layers progressively reduce the feature dimensions from 128 → 64 → 32 → 10, ending with a concatenation layer. Dropout is applied after every dense layer to prevent overfitting.
- **Dropout Regularization**:
    - Dropout layers with a high rate (0.5) are used throughout the branch, aiming to improve generalization but likely hindering the model's ability to learn effectively due to excessive regularization.

CATEGORICAL INPUT BRANCH (DENSE NETWORK)

Model Architecture

- **Dense Layers with Regularization**:
    - Two dense layers (both 10 units) process the categorical features. They use relu activations and are regularized with L2 penalties and unit norm constraints to stabilize learning.
- **Batch Normalization**:
    - Batch normalization layers are applied after each dense layer to stabilize gradients and improve training convergence.
- **Dropout Regularization**:
    - Dropout rates of 0.2 and 0.5 are used to further reduce overfitting risks.

HYBRID ARCHITECTURE AND OPTIMIZATION

- **Feature Fusion**:

    - The outputs of the two branches (text and categorical) are concatenated via a Concatenate layer and processed through a dense layer before the final output.
- **Output Layer**:
    - The output layer uses a softmax activation for multi-class classification into 5 categories.
- **Optimizer**:
    - Stochastic Gradient Descent (SGD) is used with a low learning rate (0.001) and momentum (0.9) for steady convergence.

PERFORMANCE METRICS

- **Train Accuracy (37.22%)**:
    - The model struggles to learn from the training data, achieving a very low accuracy close to random guessing.
- **Test Accuracy (37.25%)**:
    - The similar test accuracy indicates that the model is not overfitting but underfitting the data.
- **Zero Precision, Recall, and F1 Score**:
    - The model fails to classify any instance correctly, leading to zero scores for all evaluation metrics.

TRAINING AND VALIDATION CURVES

- **Loss Curves**:
    - Both training and validation losses stabilize early (around 1.45), indicating that the model has reached a plateau and is no longer improving.
- **Accuracy Curves**:
    - Training and validation accuracies plateau around 37%, showing no meaningful progress during training.

OBSERVATIONS

1. **Underfitting**:
    - The similar and low train/test accuracies suggest the model is underfitting, likely due to limited complexity or insufficient data preprocessing.

2. **Over-regularization**:
    - The excessive dropout rates (up to 50%) may prevent the model from learning effectively, especially in combination with the regularization applied to dense layers.
3. **Optimizer Limitation**:
    - The use of SGD with a very low learning rate might slow convergence. An adaptive optimizer like Adam could help address this.
4. **Data Issues**:
    - Imbalanced classes or insufficient representation in the embeddings or categorical features could limit the model's performance.
5. **Task Complexity**:
    - The hybrid structure may introduce unnecessary complexity if the input features do not complement each other or are poorly preprocessed.

RECOMMENDATIONS

1. **Model Refinement**:
    - Reduce dropout rates to around 0.2–0.3.
    - Experiment with simpler architectures or fewer dense layers to avoid over-complicating the model.
2. **Optimization**:
    - Use the Adam optimizer for faster and more adaptive learning.
    - Increase the learning rate slightly (e.g., 0.01) to improve convergence.
3. **Data Preprocessing**:
    - Address class imbalance using oversampling, undersampling, or class weights.
    - Validate the quality and representativeness of embeddings and categorical features.
4. **Evaluation Metrics**:
    - Evaluate the model on more granular metrics per class to better understand its performance.
5. **Feature Engineering**:

- Explore additional features or more refined representations for both text and categorical data to better capture underlying patterns.

CREATING THE PICKLE FILE FOR CHATBOT INTEGRATION

```python
import pickle

# Save the model to a file
with open('my_model.pkl', 'wb') as file:
    pickle.dump(model, file)
```