AIML
Capstone Project

Industrial Safety: NLP Based Chatbot

Team: Group - 10 - NLP1

Batch: AIML ONLINE SEPTEMBER 23 A

By

Mitesh Waghela
Raghavendra Sriram Vempaty
Sridhar G R
Shaurya Bhagat
Rethina Durai S J
Karan Raj Sharma

Great Learning
POWER AHEAD

NATURAL LANGUAGE PROCESSING

NLP Chatbot - Milestone-2

Version 2.0

September 15, 2024

Contents

About This Capstone Milestone Report

History

Version No.	Issue Date	Author	Status	Reason for Change
1.0	24-Aug-2024	Group 10	Initial Version	NA
2.0	15-Sep-2024	Group 10	Initial Version	Milestone 2 added

Review

Reviewers	Version No.	Date
Mitesh Waghela	1.0	24-Aug-2024
Raghavendra Sriram Vempaty	1.0	24-Aug-2024
Shaurya Bhagat	1.0	24-Aug-2024
Rethina Durai S J	1.0	24-Aug-2024
Karan Raj Sharma	1.0	24-Aug-2024
Sridhar G.R.	1.0	24-Aug-2024

Approvers

Approvers	Version No.	Date
Jayanth – Great Learning	1.0	24-Aug-2024

1 Summary of problem statement, data and findings

1.1 Domain and Context

DOMAIN: Industrial safety - NLP based Chatbot.

CONTEXT: 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.

1.2 Objective

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

1.3 Data Description

The database is basically records of accidents from 12 different plants in three different countries where every line in the data is an occurrence of an accident. Different columns with its description are as follows:

- **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.

Link to download the dataset:

https://www.kaggle.com/ihmstefanini/industrial-safety-and-health-analytics-database
[for your reference only]

`

1.4 Project Approach and Timelines

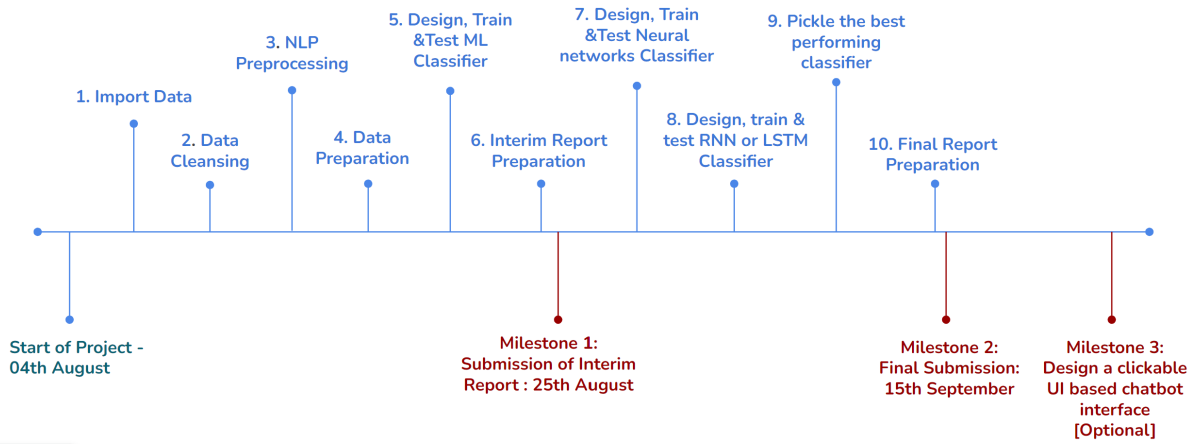


Figure 1 Project Approach and Timelines

1.5 Assumptions

List of assumptions agreed upon with mentor (Jayanth) for Group-10.

1. Utilize the dataset provided in Capstone project details link for NLP
2. Report need not include all the steps mentioned in the Google Collab notebook used to develop
3. To capture both Test & Train data to showcase which is the best classifier
4. Include only relevant visualizations

1.6 Acknowledgement

Group-10 would like to acknowledge Jayanth's support for each and every phase of this project and thank the Great learning team for their diligent support throughout this phase of the project.

`

1.7 Data Import and Exploration

The main objective of this step is to import the data set and check for the type of data present within each attribute.

	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 for maintenance, the supervisor proceeds to loosen the support of the intermediate centralizer to facilitate the removal, seeing this the mechanic supports one end on the drill of the equipment to pull with both hands the bar and accelerate the removal from this, at this moment the bar slides from its point of support and tightens the fingers of the mechanic between the drilling bar and the beam of the jumbo.
1	1	2016-01-02	Country_02	Local_02	Mining	I	IV	Male	Employee	Pressurized Systems	During the activation of a sodium sulphide pump, the piping was uncoupled and the sulfide solution was designed in the area to reach the maid. Immediately she made use of the emergency shower and was directed to the ambulatory doctor and later to the hospital. Note: of sulphide solution = 48 grams / liter.
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 when the collaborator was doing the excavation work with a pick (hand tool), hitting a rock with the flat part of the beak, it bounces off hitting the steel tip of the safety shoe and then the metatarsal area of the left foot of the collaborator causing the injury.
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 CX-695 OB7, the personnel begins the task of unlocking the Soquet bolts of the BHB machine, when they were in the penultimate bolt they identified that the hexagonal head was worn, proceeding Mr. Cristóbal - Auxiliary assistant to climb to the platform to exert pressure with your hand on the "DADO" key, to prevent it from coming out of the bolt; in those moments two collaborators rotate with the lever in anti-clockwise direction, leaving the key of the bolt, hitting the palm of the left hand, causing the injury.
4	4	2016-01-10	Country_01	Local_04	Mining	IV	IV	Male	Third Party	Others	Approximately at 11:45 a.m. in circumstances that the mechanics Anthony (group leader), Eduardo and Eric Fernández-injured-the three of the Company IMPROMEC, performed the removal of the pulley of the motor of the pump 3015 in the ZAF of Marcy. 27 cm / Length: 33 cm / Weight: 70 kg), as it was locked proceed to heating the pulley to loosen it, it comes out and falls from a distance of 1.06 meters high and hits the instep of the right foot of the worker, causing the injury described.

Shape and Datatype of each attribute:

```
[ ]  print("Number of rows = {0} and Number of Columns = {1} in the Data frame".format(ISH_df.shape[0], ISH_df.shape[1]))
```

```
Number of rows = 425 and Number of Columns = 11 in the Data frame
```

Datatype of each attribute:

```
[ ]  # Check datatypes
     ISH_df.dtypes
```

	0
Unnamed: 0	int64
Data	datetime64[ns]
Countries	object
Local	object
Industry Sector	object
Accident Level	object
Potential Accident Level	object
Genre	object
Employee or Third Party	object
Critical Risk	object
Description	object

dtype: object

Checking for Null and Missing Values:

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

Checking for Duplicate Values:

`

```
Duplicate_Rows = ISH_df.duplicated().sum()
print("Number of duplicate rows:", Duplicate_Rows)
```

```
Number of duplicate rows: 7
```

```
# View Duplicate records
Duplicates = ISH_df.duplicated()

ISH_df[Duplicates]
```

	Date	Country	City	Industry Sector	Accident Level	Potential Accident Level	Gender	Employee Type	Critical Risk	Description
77	2016-04-01	Country_01	City_01	Mining	I	V	Male	Third Party (Remote)	Others	In circumstances that two workers of the Abratech company were doing putty work inside the conditioning tank (5 meters deep and covered by platforms) of metal gratings - grating- in the upper part), two other employees of the HyT company carried out maneuvers transfer of a pump with the help of a manual tick - which worked hooked to a beam H, dragging the pump on the metal gratings (grating), suddenly the pump is hooked with a metal grate (grating) and when trying to release it, the metal grid (grating - 13.0 Kg. (60 cm x 92 cm)) falls inside the tank, hits a diagonal channel inside the tank and then impacts the right arm of one of the workers and rubs the helmet of the second worker that he was crouching. The area where the bomb was being moved was marked with tape and did not have a lookout.
262	2016-12-01	Country_01	City_03	Mining	I	IV	Male	Employee	Others	During the activity of chuteo of ore in hopper OP5; the operator of the locomotive parks his equipment under the hopper to fill the first car, it is at this moment that when it was blowing out to release the load, a mud flow suddenly appears with the presence of rock fragments; the personnel that was in the direction of the flow was covered with mud.
303	2017-01-21	Country_02	City_02	Mining	I	I	Male	Third Party (Remote)	Others	Employees engaged in the removal of material from the excavation of the well 2 of level 265, using shovel and placing it in the bucket. During the day some of this material fell into the pipes of the employees' boots and the friction between the boot and the calf caused a superficial injury to the legs.
345	2017-03-02	Country_03	City_10	Others	I	I	Male	Third Party	Venomous Animals	On 02/03/17 during the soil sampling in the region of Sta. the employees Rafael and Danillo da Silva were attacked by a bee test. They rushed away from the place, but the employee Rafael took 4 bites, one on the chin, one on the chest, one on the neck and one on the hand over the glove. The employee took 4 bites, one in his hand over his glove and the other in the head, and the employee Danillo took 2 bites in the left arm over his uniform. At first no one sketched allergy, just swelling at the sting site. The activity was stopped to evaluate the site, after verifying that the test had remained in the line, they left the site.
346	2017-03-02	Country_03	City_10	Others	I	I	Male	Third Party	Venomous Animals	On 02/03/17 during the soil sampling in the region of Sta. the employees Rafael and Danillo da Silva were attacked by a bee test. They rushed away from the place, but the employee Rafael took 4 bites, one on the chin, one on the chest, one on the neck and one on the hand over the glove. The employee took 4 bites, one in his hand over his glove and the other in the head, and the employee Danillo took 2 bites in the left arm over his uniform. At first no one sketched allergy, just swelling at the sting site. The activity was stopped to evaluate the site, after verifying that the test had remained in the line, they left the site.
355	2017-03-15	Country_03	City_10	Others	I	I	Male	Third Party	Venomous Animals	Team of the VMS Project performed soil collection on the Xixás target with 3 members. When the teams were moving from one collection point to another, Mr. Fabio was ahead of the team, stinging behind Robson and Manoel da Silva. near the collection point were surprised by a swarm of bees that was inside a I play near the ground, with no visibility in the woods and no hissing noise. Fabio passed by the stump, but Robson and Manoel da Silva were attacked by the bees. Robson had a sting in his left arm over his uniform and Manoel da Silva had a prick in his lip as his screen ripped as he tangled in the branches during the escape.
397	2017-05-23	Country_01	City_04	Mining	I	IV	Male	Third Party	Projection of fragments	In moments when the 02 collaborators carried out the inspection of the conveyor belt No. 3 from the tail pulley when they were at the height of the load polymer No. 372, the Maslucan collaborator heard a noise where note that the belt was moving towards the tail pulley, 4 "fragmentos mineral fragments are projected towards the access of the ramp impacting the 2 collaborators, being evacuated to the medical post.

Number of duplicate rows identified are 7.

Data Exploration Observations:

Overall:

1. The dataset contains information on industrial accidents across different countries, cities, and industry sectors.
2. The time frame of the accidents is captured in the 'Date' column.
3. The severity of accidents is categorized into levels from I to VI.
4. Information about the gender, employee type, critical risk, and a detailed description of the accident is provided.

Specific Observations:

- **Country:** Most accidents occurred in Country_01, followed by Country_02 and Country_03.
- **City:** The distribution of accidents across cities varies, with some cities having a higher number of incidents than others.
- **Industry Sector:** The 'Mining' sector has the highest number of accidents, indicating a potentially higher risk in this industry.
- **Accident Level:** The majority of accidents fall under levels I and II, suggesting that most accidents are relatively minor in severity.
- **Potential Accident Level:** There's a notable difference between the actual accident level and the potential accident level, highlighting the importance of preventive measures.
- **Gender:** Male employees are involved in a significantly higher number of accidents compared to females.
- **Employee Type:** Most accidents involve employees rather than third parties.
- **Critical Risk:** 'Others' is the most frequent category in critical risk, which might indicate a need for more specific categorization.
- **Description:** The description column provides detailed narratives of the accidents, which can be valuable for further text analysis and understanding the circumstances leading to accidents.

`

Potential Areas for Further Analysis:

1. Investigate the reasons behind the higher number of accidents in specific countries, cities, and industry sectors.
2. Analyze the factors contributing to the difference between actual and potential accident levels.
3. Explore the reasons for the gender disparity in accident involvement.
4. Deep dive into the 'Others' category in critical risk to identify potential subcategories.
5. Perform text analysis on the 'Description' column to extract insights and patterns related to accident causes.

`

2 Overview of the final process

2.1 Data Cleansing

The main objective of this step is to fix any inconsistency found in the data file shared by great learning.

- Dropping Duplicate Records
- Dropping irrelevant columns
- Aligning Attribute names as per data available
- Checking the resulting dataframe and unique values for each attribute.

Dropping the duplicates

```
# Check for Duplicate rows in the dataset

Duplicate_Rows = ISH_df.duplicated().sum()
print('Number of duplicate rows:', Duplicate_Rows)
```

```
Number of duplicate rows: 7
```

```
# View Duplicate records
Duplicates = ISH_df.duplicated()

ISH_df[Duplicates]
```

```
# Remove duplicate rows and save the deduplicated dataset

ISH_df_cleaned = ISH_df.drop_duplicates()

# Save the deduplicated dataset to a new file
ISH_df_cleaned.to_csv('ISH_df_cleaned.csv', index=False)

# Print the number of rows before and after deduplication
print('Number of rows before deduplication:', len(ISH_df))
print('Number of rows after deduplication:', len(ISH_df_cleaned))
```

```
Number of rows before deduplication: 425
Number of rows after deduplication: 418
```

Number of duplicate rows identified are 7. Dropping these rows reduces the dataframe from 425 rows to 418 rows.

New dataframe shape is (418, 10)

Dropping irrelevant columns

```
# Dropping Unnecessary Columns:

ISH_df.drop("Unnamed: 0", axis=1, inplace=True)
```

Unnamed: 0: This column appears to be an index column and does not provide any useful information for analysis.

Updated dataframe shape is (418, 10)

`

Aligning Attribute names as per data available

1. Column name spelling fixed. Example: Data and Genre were misspelled, were changed to **Date** and **Gender**
2. Column names were changed to match appropriate nomenclature. e.g. Countries changed to **Country**, Local to **City**, Employee or Third Party to **Employee Type**

```
[ ]  # Renaming the columns as per available Data and Description
     ISH_df.rename(columns={
         "Data": "Date",
         "Countries": "Country",
         "Local": "City",
         "Genre": "Gender",
         "Employee or Third Party":"Employee Type",
     }, inplace=True)

     # Modify 'City' column values
     ISH_df['City'] = ISH_df['City'].str.replace('Local_', 'City_')

     ISH_df.head()
```

Checking the updated dataframe post data cleansing:

	Date	Country	City	Industry Sector	Accident Level	Potential Accident Level	Gender	Employee Type	Critical Risk	Description
0	2016-01-01	Country_01	City_01	Mining	I	IV	Male	Third Party	Pressed	While removing the drill rod of the Jumbo 08 for maintenance, the supervisor proceeds to loosen the support of the intermediate centralizer to facilitate the removal, seeing this the mechanic supports one end on the drill of the equipment to pull with both hands the bar and accelerate the removal from this, at this moment the bar slides from its point of support and tightens the fingers of the mechanic between the drilling bar and the beam of the jumbo.
1	2016-01-02	Country_02	City_02	Mining	I	IV	Male	Employee	Pressurized Systems	During the activation of a sodium sulphide pump, the piping was uncoupled and the sulfide solution was designed in the area to reach the maid. Immediately she made use of the emergency shower and was directed to the ambulatory doctor and later to the hospital. Note: of sulphide solution = 48 grams / liter.
2	2016-01-06	Country_01	City_03	Mining	I	III	Male	Third Party (Remote)	Manual Tools	In the sub-station MILPO located at level +170 when the collaborator was doing the excavation work with a pick (hand tool), hitting a rock with the flat part of the beak, it bounces off hitting the steel tip of the safety shoe and then the metatarsal area of the left foot of the collaborator causing the injury.
3	2016-01-08	Country_01	City_04	Mining	I	I	Male	Third Party	Others	Being 9:45 am. approximately in the Nv. 1880 CX-695 OB7, the personnel begins the task of unlocking the Soquet bolts of the BHB machine, when they were in the penultimate bolt they identified that the hexagonal head was worn, proceeding Mr. Cristóbal - Auxiliary assistant to climb to the platform to exert pressure with your hand on the "DADO" key, to prevent it from coming out of the bolt; in those moments two collaborators rotate with the lever in anti-clockwise direction, leaving the key of the bolt, hitting the palm of the left hand, causing the injury.
4	2016-01-10	Country_01	City_04	Mining	IV	IV	Male	Third Party	Others	Approximately at 11:45 a.m. in circumstances that the mechanics Anthony (group leader), Eduardo and Eric Fernández-injured-the three of the Company IMPROMEC, performed the removal of the pulley of the motor of the pump 3015 in the ZAF of Marcy. 27 cm / Length: 33 cm / Weight: 70 kg), as it was locked proceed to heating the pulley to loosen it, it comes out and falls from a distance of 1.06 meters high and hits the instep of the right foot of the worker, causing the injury described.

Checking the resulting dataframe and unique values for each attribute.

Date	287
Country	3
City	12
Industry Sector	3
Accident Level	5
Potential Accident Level	6
Gender	2
Employee Type	3
Critical Risk	33
Description	411

Identified numerical and categorical columns

```
# Identify numerical and categorical columns
numerical_columns = ISH_df_cleaned.select_dtypes(include=[np.number]).columns.tolist()
categorical_columns = ISH_df_cleaned.select_dtypes(exclude=[np.number]).columns.tolist()

# Exclude 'Data' column from categorical columns
categorical_columns = [col for col in categorical_columns if col != 'Date']
print('Numerical columns:', numerical_columns)
print('Categorical columns:', categorical_columns)
```

```
Numerical columns: []
Categorical columns: ['Country', 'City', 'Industry Sector', 'Accident Level', 'Potential Accident Level', 'Gender', 'Employee Type', 'Critical Risk', 'Description']
```

`

2.2 Data Visualization

Below is different visualization charts generated for the detailed data exploration:

1. Univariate Plots

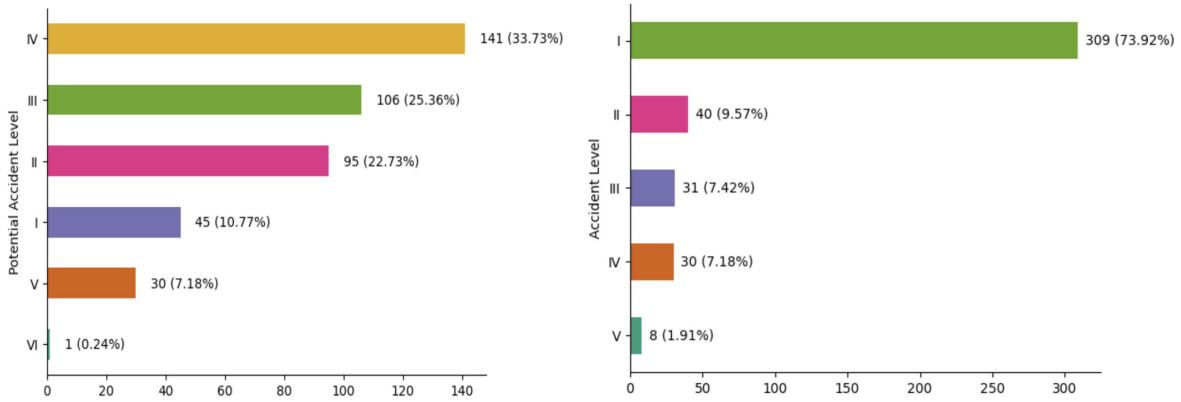


Figure 2 Potential Accident Level Distribution and Accident Level Distribution

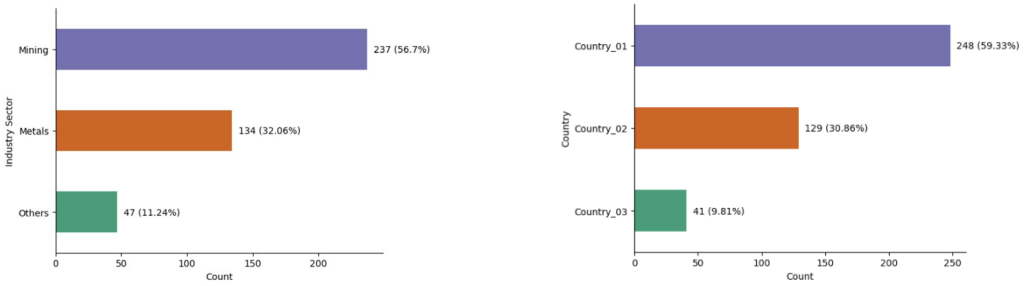


Figure 3 Industry Sector Distribution and Country Distribution

`

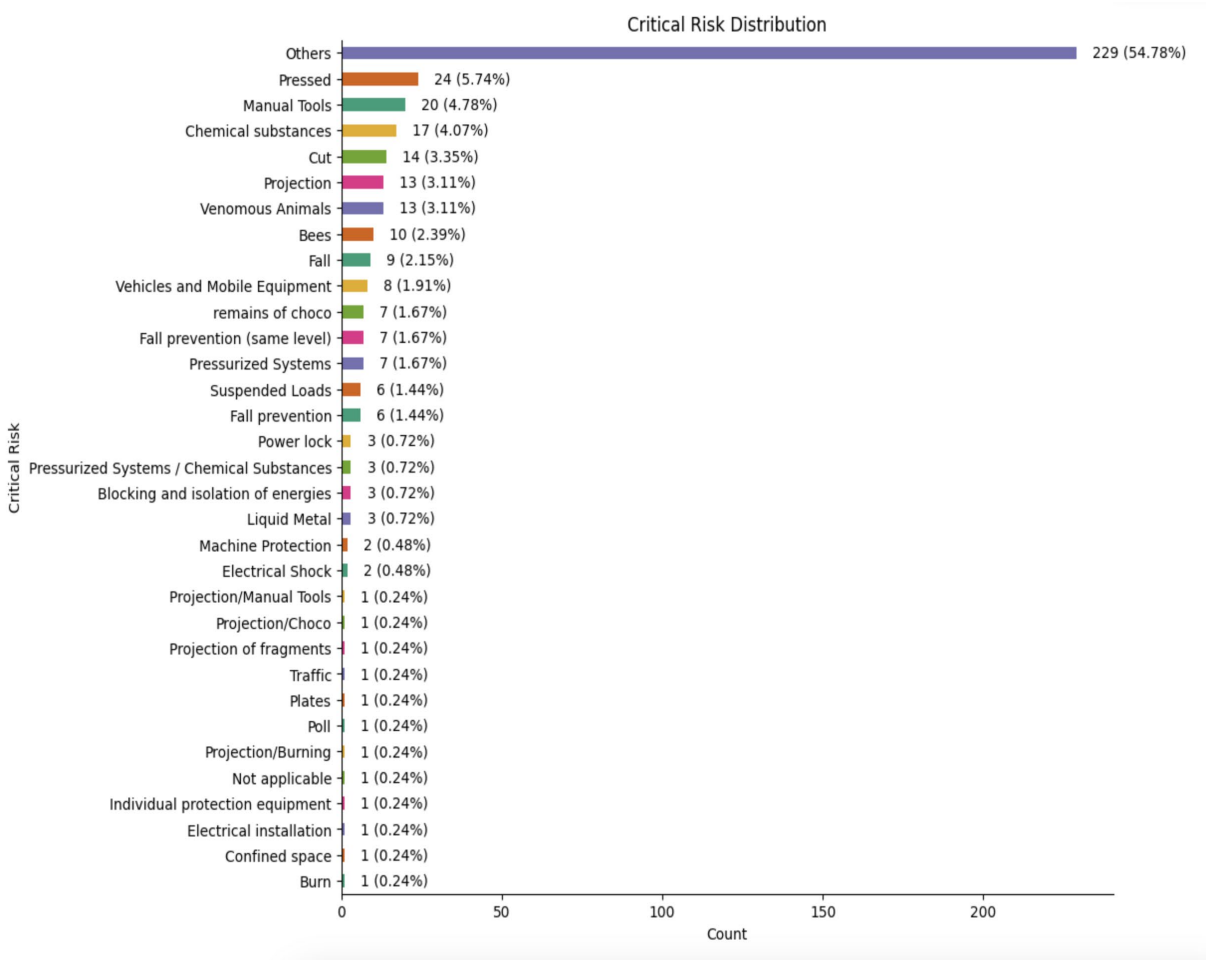


Figure 4 Critical Risk Distribution

2. Bivariate Plots

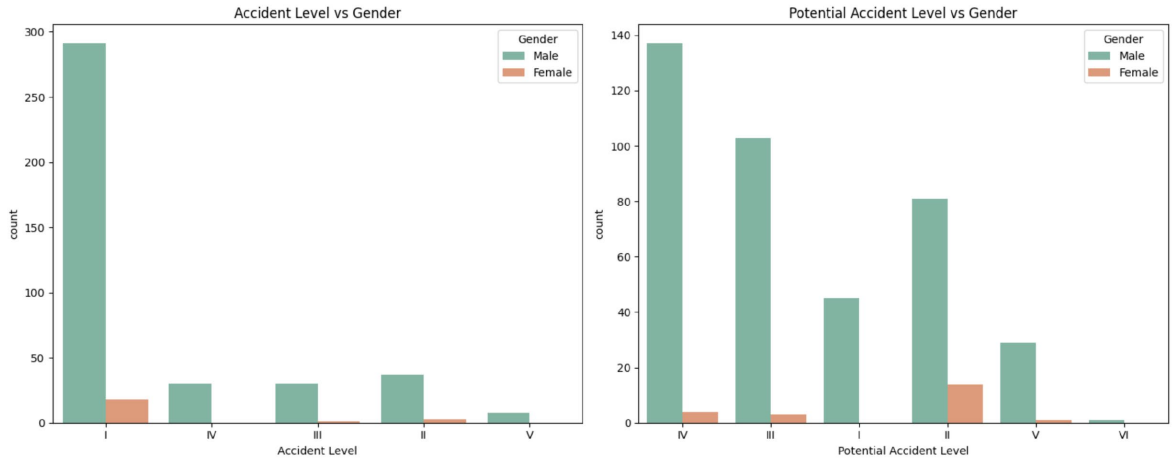


Figure 1 Accident Level Vs Gender & Potential Accident Level Vs Gender Approach and Timelines

`

Observations:

Accident Level vs Gender:

1. A significantly higher number of males are involved in accidents across all accident levels.
2. The disparity is particularly pronounced in lower accident levels (I and II).

Potential Accident Level vs Gender:

3. Similar to the actual accident level, males are more likely to be involved in potential accidents.
4. The difference in potential accident levels between genders is less pronounced compared to actual accidents, suggesting that preventive measures might be more effective for males.

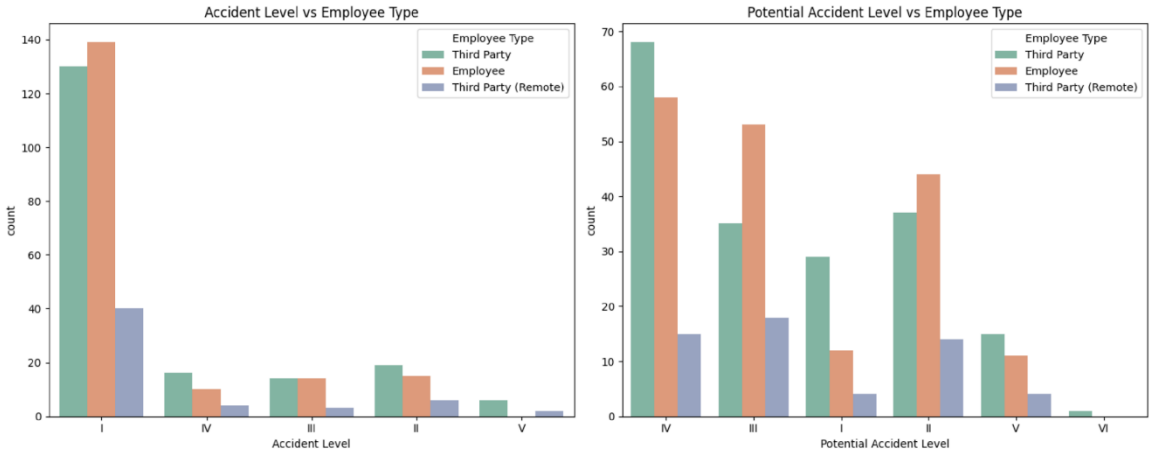


Figure 2 Accident Level Vs Employee Type & Potential Accident Level Vs Employee Type

Observations:

Accident Level vs Employee Type:

1. Employees are involved in a significantly higher number of accidents across all accident levels compared to third parties.
2. The difference is particularly pronounced in lower accident levels (I and II).

Potential Accident Level vs Employee Type:

1. Similar to the actual accident level, employees are more likely to be involved in potential accidents compared to third parties.
2. The difference in potential accident levels between employee types is less pronounced compared to actual accidents. This suggests that preventive measures might be more effective for employees.

`

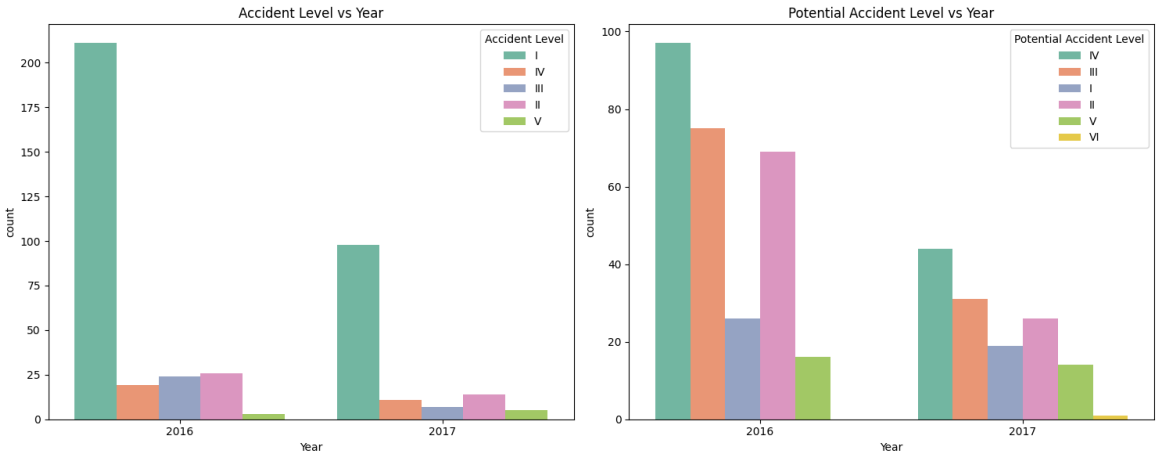


Figure 3 Accident Level Vs Month & Potential Accident Level Vs Month

Accident Level vs Year:

1. There's a noticeable decrease in the number of accidents across all levels in the later years compared to the initial years.
2. This suggests a positive trend in terms of safety improvements over time.

Potential Accident Level vs Year:

1. Similar to the actual accident level, potential accidents also show a decreasing trend over the years.
2. This indicates that preventive measures and safety protocols might be becoming more effective in mitigating potential risks.

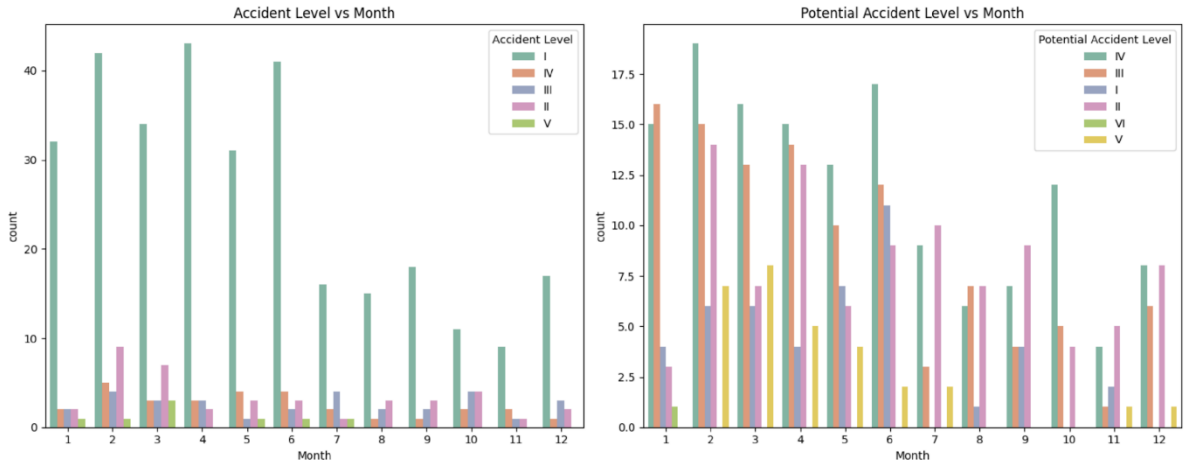


Figure 7 Accident Level Vs Month & Potential Accident Level Vs Month

`

Accident Level vs Month:

1. There's some variation in accident counts across different months, but no clear seasonal pattern emerges.
2. Further analysis might be needed to identify potential factors influencing these monthly fluctuations.

Potential Accident Level vs Month:

1. Similar to the actual accident level, potential accidents also show some monthly variation without a distinct seasonal pattern.
2. This suggests that the factors influencing accident occurrences might not be strongly tied to specific months.

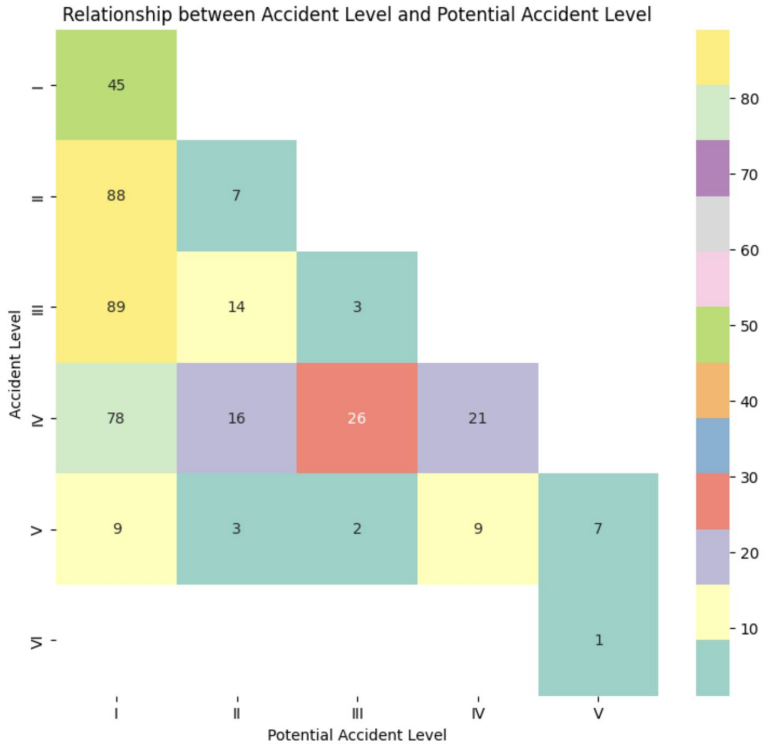


Figure 8 Accident Vs Potential Accident Levels

Observations:

Diagonal Dominance:

1. The heatmap shows a strong diagonal dominance, indicating a positive correlation between Accident Level and Potential Accident Level.
2. This implies that accidents with a higher actual severity level are also more likely to have a higher potential severity level.

`

Potential for Worse Outcomes:

3. There are significant off-diagonal values, especially above the diagonal.
4. This suggests that many accidents that resulted in lower actual severity levels had the potential to be much worse.

Preventive Measures:

5. The difference between actual and potential severity highlights the importance of preventive measures and safety protocols.
6. These measures likely played a role in preventing many accidents from escalating to their full potential severity.

Focus Areas for Improvement:

7. The heatmap can help identify areas where safety measures can be further improved.
8. For example, focusing on accidents with high potential severity but lower actual severity can lead to more effective prevention strategies.

Here the overall detailed analysis:

2.2.1 Monthly frequency of accidents - YoY

Continuously enhance safety protocols by monitoring trends, addressing seasonal risks, and adapting strategies to reduce year-over-year fluctuations in accidents.

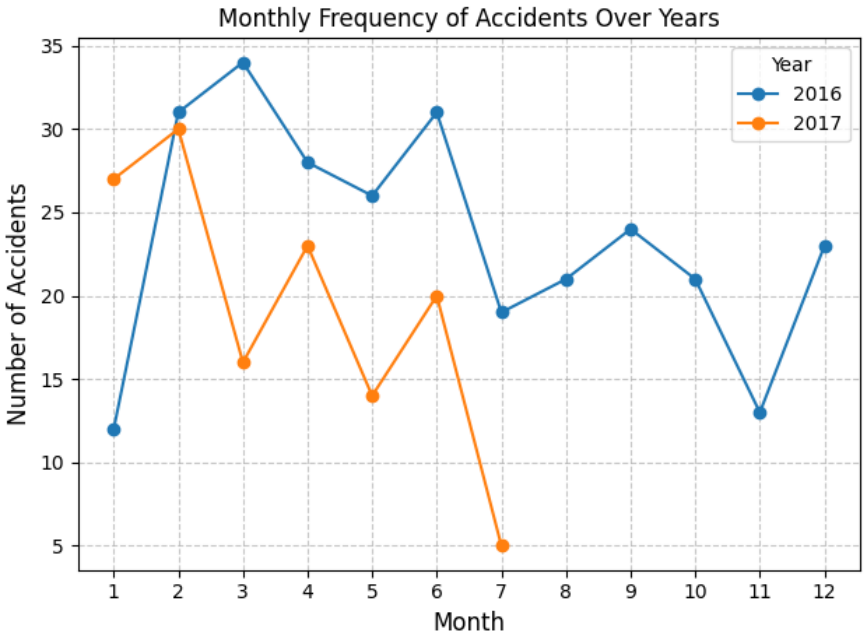


Figure 9 Monthly Frequency of Accidents Over Years

Recommendation:

`

1. **Quarterly Safety Review:** Review and adjust safety protocols at the end of each quarter.
2. **Mid-Year Safety Audit:** Conduct a detailed safety audit in April to mitigate mid-year risks.
3. **Seasonal Safety Campaign: I**mplement targeted safety initiatives from May to July to reduce accidents.
4. **End-of-Year Evaluation:** Analyze annual accident data in December to refine safety strategies for the next year.

2.2.2 Distribution of accident levels across countries

The distribution of accident levels differs by country, indicating possible variations in safety regulations, industry practices, or specific risk factors.

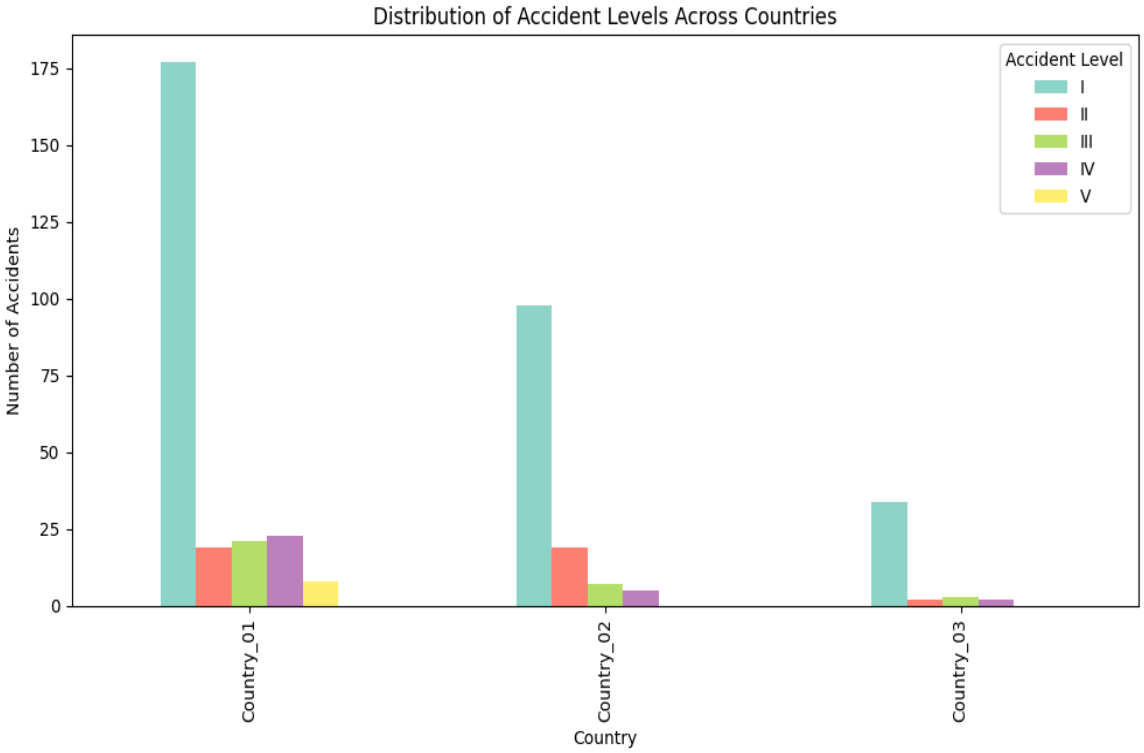


Figure 10 Distribution of accident levels across countries

Recommendation:

1. **Strengthen Safety Measures in Country_01**: Address the consistently high accident rates across all levels.
2. **Reduce Level III Accidents in Country_02**: Identify and mitigate specific risks contributing to more severe accidents.
3. **Adopt Best Practices from Country_03**: Learn from Country_03's effective safety protocols to improve other countries' performance.

`

2.2.3 Accident level vs Industry sectors

Geographic and industry-specific risks dominate, with severity variations and local patterns highlighting the need for targeted safety measures.

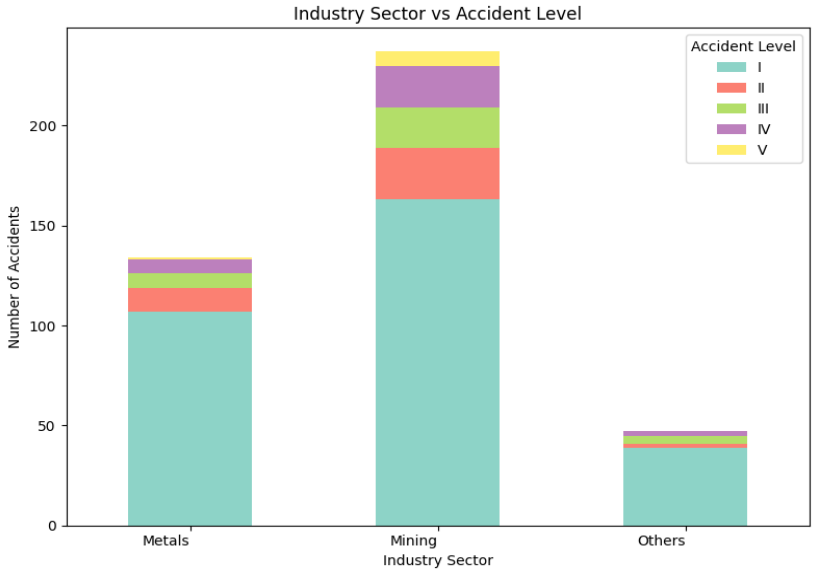


Figure 11 Industry Sector vs Accident Levels

Recommendation:

1. **Enhance Safety Protocols in Mining:** Implement targeted safety interventions in the mining sector to reduce accidents across all severity levels.

2. **Prevent Escalation of Minor Incidents:** Develop strategies to ensure that Level I and II incidents do not progress to more severe accidents across all sectors.

2.2.4 Accident level with respect to Gender/Employee types

Enhancing Safety Measures for Vulnerable Employee Groups

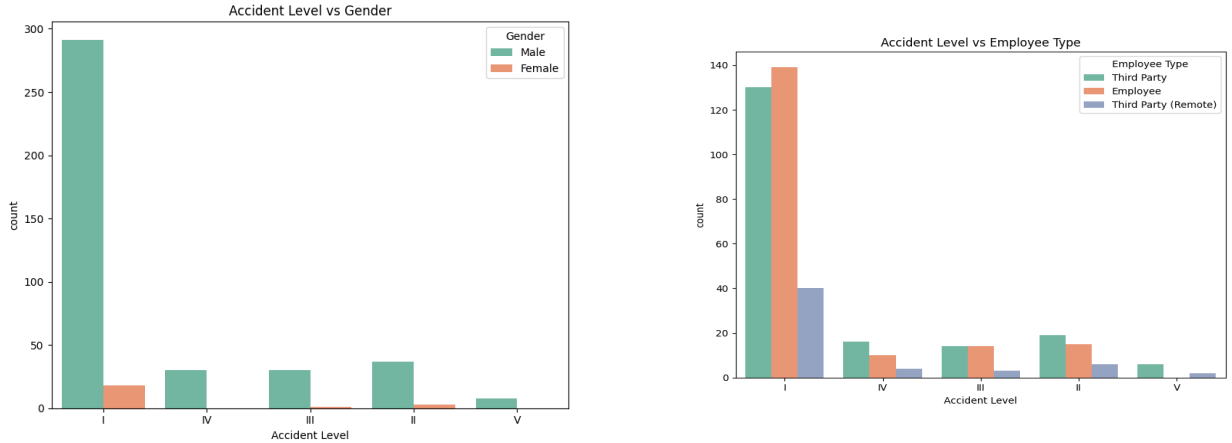


Figure 12 Accident Level Vs Gender/Employee Type

`

Recommendation:

1. **Implement targeted safety training for employees and males** to reduce lower-level accidents (I and II).
2. **Strengthen monitoring and intervention** strategies to minimize the disparity in accident rates among different employee types and genders.

2.3 Overall observations

1. **Temporal Trends**: Fluctuations observed; no clear long-term trend but down term trend is observed
2. **Geographic Risk:** Country_01 shows highest accident rates, followed by Country_02 and Country_03, with significant city-level variations.
3. **Industry Risk**: Mining sector has highest accident frequency across all severity levels and locations.
4. **Severity Distribution**: Majority of accidents are Level I and II, but distribution varies by country and city.
5. **Risk Escalation**: Strong correlation between actual and potential accident severity, with many incidents showing potential for worse outcomes.
6. **Country Variations:** Country_02 shows increase in Level III accidents; Country_03 has fewer severe accidents.
7. **City-Specific Patterns:** Unique accident distribution patterns observed across cities, indicating local risk factors.
8. **Demographics:** Male employees and regular staff involved in significantly more accidents.
9. **Preventive Measures**: Gap between actual and potential severity levels indicates effectiveness of safety protocols.

2.4 Algorithms & Techniques used

2.4.1 NLP Pre-Processing

NLP (Natural Language Processing) pre-processing is a crucial step in preparing raw text data for analysis and modelling. It involves various techniques to clean, transform, and structure text data

`

in a way that makes it suitable for machine learning models and other NLP tasks. Here are some common pre-processing steps:

- Tokenization
- Lowercasing
- Removing Punctuation
- Removing Stop Words
- Stemming
- Lemmatization
- Removing Numbers
- Removing Whitespace
- Part-of-Speech Tagging
- Named Entity Recognition (NER)
- Text Normalization
- Handling Special Characters
- Handling Misspellings and Slang

These pre-processing steps help transform raw text data into a structured format that can be easily fed into NLP models for tasks like text classification, sentiment analysis, machine translation, and more. The choice of steps depends on the specific use case and the nature of the text data. The below sections mentions that all the preprocessing is done for the Capstone project

2.4.2 Word-clouds - Unigrams, Bigrams & Trigrams

A word cloud is a visual representation of text data, where the size of each word indicates its frequency or importance in the source text. Word clouds are commonly used for text analysis and can provide a quick visual summary of the most prominent words in a dataset.

Overall, below steps are performed:

- Initialise lemmatize and stop words
- Convert to lowercase
- Remove special characters
- Tokenize the text
- Remove stop-words
- Apply pre-processing to description column

`

```
# Load the dataset
ISH_NLP_preprocess = pd.read_csv('/content/drive/My Drive/Capstone_Group10_NLP1/ISH_df_preprocess.csv')

# Initialize lemmatizer and stopwords
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))

def preprocess_text(text):
    # Convert to lowercase
    text = text.lower()

    # Remove special characters and numbers
    text = re.sub(r'[^a-zA-Z\s]', '', text)

    # Tokenize the text
    tokens = word_tokenize(text)

    # Remove stopwords and lemmatize
    cleaned_tokens = [lemmatizer.lemmatize(token) for token in tokens if token not in stop_words]

    # Join the tokens back into a string
    cleaned_text = ' '.join(cleaned_tokens)

    return cleaned_text

# Apply preprocessing to the Description column
ISH_NLP_preprocess['Cleaned_Description'] = ISH_NLP_preprocess['Description'].apply(preprocess_text)

# Display the first few rows of the original and cleaned descriptions
ISH_NLP_preprocess[['Description', 'Cleaned_Description']].head()

# Save the number of words before and after cleaning
ISH_NLP_preprocess['Original_Word_Count'] = ISH_NLP_preprocess['Description'].apply(lambda x: len(str(x).split()))
ISH_NLP_preprocess['Cleaned_Word_Count'] = ISH_NLP_preprocess['Cleaned_Description'].apply(lambda x: len(str(x).split()))
```

```
ISH_NLP_preprocess[['Description', 'Cleaned_Description']].head()
```

	Description	Cleaned_Description
0	While removing the drill rod of the Jumbo 08 for maintenance, the supervisor proceeds to loosen the support of the intermediate centralizer to facilitate the removal, seeing this the mechanic supports one end on the drill of the equipment to pull with both hands the bar and accelerate the removal from this, at this moment the bar slides from its point of support and tightens the fingers of the mechanic between the drilling bar and the beam of the jumbo.	removing drill rod jumbo maintenance supervisor proceeds loosen support intermediate centralizer facilitate removal seeing mechanic support one end drill equipment pull hand bar accelerate removal moment bar slide point support tightens finger mechanic drilling bar beam jumbo
1	During the activation of a sodium sulphide pump, the piping was uncoupled and the sulfide solution was designed in the area to reach the maid. Immediately she made use of the emergency shower and was directed to the ambulatory doctor and later to the hospital. Note: of sulphide solution = 48 grams / liter.	activation sodium sulphide pump piping uncoupled sulfide solution designed area reach maid immediately made use emergency shower directed ambulatory doctor later hospital note sulphide solution gram liter

```
ISH_NLP_preprocess
```

	Country	City	Industry Sector	Accident Level	Potential Accident Level	Gender	Employee Type	Critical Risk	Description	DayOfWeek	Year	Month	Day	Cleaned_Description	Original_Word_Count	Cleaned_Word_Count
0	Country_01	City_01	Mining	I	IV	Male	Third Party	Pressed	While removing the drill rod of the Jumbo 08 for maintenance, the supervisor proceeds to loosen the support of the intermediate centralizer to facilitate the removal, seeing this the mechanic supports one end on the drill of the equipment to pull with both hands the bar and accelerate the removal from this, at this moment the bar slides from its point of support and tightens the fingers of the mechanic between the drilling bar and the beam of the jumbo.	4	2016	1	1	removing drill rod jumbo maintenance supervisor proceeds loosen support intermediate centralizer facilitate removal seeing mechanic support one end drill equipment pull hand bar accelerate removal moment bar slide point support tightens finger mechanic drilling bar beam jumbo	80	37
1	Country_02	City_02	Mining	I	IV	Male	Employee	Pressurized Systems	During the activation of a sodium sulphide pump, the piping was uncoupled and the sulfide solution was designed in the area to reach the maid. Immediately she made use of the emergency shower and was directed to the ambulatory doctor and later to the hospital. Note: of sulphide solution = 48 grams / liter.	5	2016	1	2	activation sodium sulphide pump piping uncoupled sulfide solution designed area reach maid immediately made use emergency shower directed ambulatory doctor later hospital note sulphide solution gram liter	54	27
2	Country_01	City_03	Mining	I	III	Male	Third Party (Remote)	Manual Tools	In the sub-station MILPO located at level +170 when the collaborator was doing the excavation work with a pick (hand tool), hitting a rock with the flat part of the beak, it bounces off hitting the steel tip of the safety shoe and then the metatarsal area of the left foot of the collaborator causing the injury.	2	2016	1	6	substation milpo located level collaborator excavation work pick hand tool hitting rock flat part beak bounce hitting steel tip safety shoe metatarsal area left foot collaborator causing injury	57	28
3	Country_01	City_04	Mining	I	I	Male	Third Party	Others	Being 9:45 am. approximately in the Nv. 1880 CX-695 OB7, the personnel begins the task of unlocking the Soquet bolts of the BHB machine, when they were in the penultimate bolt they identified that the hexagonal head was worn, proceeding Mr. Cristóbal - Auxiliary assistant to climb to the platform to exert pressure with your hand on the "DADO" key, to prevent it from coming out of the bolt; in those moments two collaborators rotate with the lever in anti-clockwise direction, leaving the key of the bolt, hitting the palm of the left hand, causing the injury.	4	2016	1	8	approximately nv cx ob personnel begin task unlocking soquet bolt bhb machine penultimate bolt identified hexagonal head worn proceeding mr cristbal auxiliary assistant climb platform exert pressure hand dado key prevent coming bolt moment two collaborator rotate lever anticlockwise direction leaving key bolt hitting palm left hand causing injury	97	49

49.58% reduction in the word count post the following analysis:

```
# Calculate and print the average word count before and after cleaning
avg_original = ISH_NLP_preprocess['Original_Word_Count'].mean()
avg_cleaned = ISH_NLP_preprocess['Cleaned_Word_Count'].mean()
print(f"\nAverage word count before cleaning: {avg_original:.2f}")
print(f"Average word count after cleaning: {avg_cleaned:.2f}")
print(f"Reduction in words: {(avg_original - avg_cleaned) / avg_original * 100:.2f}%")
```

```
Average word count before cleaning: 65.06
Average word count after cleaning: 32.80
Reduction in words: 49.58%
```

From the above snapshot it is very much evident that **almost 50% of reduction of words** have been done in this process.

Following steps are performed on pre-processed data:

`

- Combine all descriptions into a single string
- Tokenize the combined text
- Calculate token distribution
- Create a dataframe from the most common words
- Generate word cloud for unigrams, bigrams and trigrams

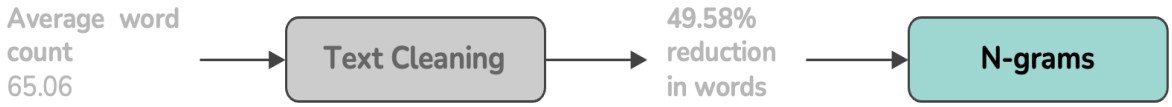
```
# @title Wordcloud for N-Grams

from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Combine all descriptions into a single string
all_text = ' '.join(ISH_NLP_preprocess['Description'].astype(str))

# Generate word cloud for unigrams
wordcloud_unigrams = WordCloud(width=800, height=400, background_color='white').generate(all_text)

# Generate word cloud for bigrams
bigrams = nltk.bigrams(word_tokenize(all_text))
bigram_text = ' '.join(['_'.join(bigram) for bigram in bigrams])
wordcloud_bigrams = WordCloud(width=800, height=400, background_color='white').generate(bigram_text)

# Generate word cloud for trigrams
trigrams = nltk.trigrams(word_tokenize(all_text))
trigram_text = ' '.join(['_'.join(trigram) for trigram in trigrams])
wordcloud_trigrams = WordCloud(width=800, height=400, background_color='white').generate(trigram_text)

# Display the word clouds
plt.figure(figsize=(45, 15))
plt.subplot(1, 3, 1)
plt.imshow(wordcloud_unigrams, interpolation='bilinear')
plt.title('Unigrams')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(wordcloud_bigrams, interpolation='bilinear')
plt.title('Bigrams')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(wordcloud_trigrams, interpolation='bilinear')
plt.title('Trigrams')
plt.axis('off')

plt.show()
```

Figure 13 Unigrams/Bigrams/Trigrams

Note: the observations of above executions has been captured in section 6.2 below.

Note: ***N-gram*** identifies accident contributors and areas for safety improvement, helping develop interventions to enhance workplace safety. The below diagram provides more clarity on percentage of text/word reductions during the cleaning process.

`



2.4.3 Observations

Unigrams:

1. Keywords include "moment," "employee," "floor," "equipment," "assistant," "left," and "hand."
2. These suggest an incident involving an employee and equipment on a specific floor.
3. Words like "collaboration," "injury," and "support" indicate teamwork and possibly injury response.
4. "Left" near "hand" points to a body part, likely in a workplace injury report.
5. This might relate to a safety analysis or accident report in an industrial setting.

Bigrams:

1. Frequent bigrams like "left hand" and "right hand" indicate a focus on hand and finger injuries.
2. This suggests frequent hand-related injuries in the analyzed data or reports.
3. Other terms like "left leg" and "left foot" appear but are less common.
4. Phrases like "causing injury" and "employee performing" point to work-related injuries.
5. Terms such as "causing cut" and "causing fall" highlight common injury mechanisms.

Trigrams:

1. Trigrams like "left hand causing" and "finger left hand" focus on injuries to the left hand or fingers.
2. Phrases like "used safety glass" suggest the involvement of specific safety measures.
3. The emphasis on hands and fingers shows their vulnerability in the workplace. The analysis details injury causes and is useful for prevention.
4. Words like "operator" and "employee" next to "accident" and "injury" emphasize roles in safety protocols.

Overall:

1. N-grams analysis offers insights into key themes and patterns in incident reports.
2. It identifies accident contributors and areas for safety improvement.
3. The findings could help develop interventions to enhance workplace safety.

2.4.4 Data Preprocessing using Glove, TFI-DF and Word2Vec

Below code is used to generate Word Embeddings over 'Description' column using **Glove, TFI-DF and Word2Vec**.

`

```
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from gensim.models import Word2Vec
from gensim.utils import simple_preprocess

def generate_embedding_dataframes(df):
    df1 = df.copy()
    df2 = df.copy()
    df3 = df.copy()

    # 1. GloVe Embeddings
    def load_glove_model(glove_file):
        embedding_dict = {}
        with open(glove_file, 'r', encoding="utf8") as f:
            for line in f:
                values = line.split()
                word = values[0]
                vector = np.asarray(values[1:], "float32")
                embedding_dict[word] = vector
        return embedding_dict

    def get_average_glove_embeddings(tokenized_words, embedding_dict, embedding_dim=300):
        embeddings = [embedding_dict.get(word, np.zeros(embedding_dim)) for word in tokenized_words]
        return np.mean(embeddings, axis=0) if embeddings else np.zeros(embedding_dim)

    # Load GloVe model and generate GloVe embeddings
    glove_file = '/content/drive/MyDrive/Capstone_Group10_NLP1/glove.6B/glove.6B.300d.txt'
    glove_embeddings = load_glove_model(glove_file)

    glove_embeddings_series = df1['tokenized_words'].apply(lambda words: get_average_glove_embeddings(words, glove_embeddings))
    ISH_NLP_Glove_df = pd.concat([df1.drop(columns=['tokenized_words']), pd.DataFrame(glove_embeddings_series.tolist(), columns=[f'GloVe_{i}' for i in range(300)])], axis=1)

    # 2. TF-IDF Features
    tfidf_vectorizer = TfidfVectorizer(tokenizer=lambda x: x, lowercase=False, token_pattern=None)
    tfidf_matrix = tfidf_vectorizer.fit_transform(df2['tokenized_words'])

    # Create a DataFrame with TF-IDF features
    tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), columns=tfidf_vectorizer.get_feature_names_out())
    ISH_NLP_TFIDF_df = pd.concat([df2.drop(columns=['tokenized_words']), tfidf_df], axis=1)

    # 3. Word2Vec Embeddings
    word2vec_model = Word2Vec(sentences=df3['tokenized_words'], vector_size=300, window=5, min_count=1, workers=4)

    def get_average_word2vec_embeddings(tokenized_words, model, embedding_dim=300):
        embeddings = [model.wv[word] for word in tokenized_words if word in model.wv]
        return np.mean(embeddings, axis=0) if embeddings else np.zeros(embedding_dim)

    word2vec_embeddings_series = df3['tokenized_words'].apply(lambda words: get_average_word2vec_embeddings(words, word2vec_model))
    ISH_NLP_Word2Vec_df = pd.concat([df3.drop(columns=['tokenized_words']), pd.DataFrame(word2vec_embeddings_series.tolist(), columns=[f'Word2Vec_{i}' for i in range(300)])], axis=1)

    return ISH_NLP_Glove_df, ISH_NLP_TFIDF_df, ISH_NLP_Word2Vec_df

# Use the function to generate the DataFrames
ISH_NLP_Glove_df, ISH_NLP_TFIDF_df, ISH_NLP_Word2Vec_df = generate_embedding_dataframes(ISH_NLP_preprocess1)
```

Output of above execution is below:

ISH_NLP_Glove_df

	Country	City	Industry Sector	Accident Level	Potential Accident Level	Gender	Employee Type	Critical Risk	DayOfWeek	Year	...	GloVe_290	GloVe_291	GloVe_292	GloVe_293	GloVe_294	GloVe_295	GloVe_296	GloVe_297	GloVe_298	GloVe_299
0	Country_01	City_01	Mining	I	IV	Male	Third Party	Pressed	4	2016	...	-0.034536	-0.110637	-0.085788	-0.031955	0.008084	0.205297	-0.001389	-0.296468	-0.051921	-0.003529
1	Country_02	City_02	Mining	I	IV	Male	Employee	Pressurized Systems	5	2016	...	-0.412660	-0.135541	0.049905	0.032907	0.103431	-0.155970	0.078383	-0.218822	-0.099618	-0.053435
2	Country_01	City_03	Mining	I	III	Male	Third Party (Remote)	Manual Tools	2	2016	...	0.005927	-0.135486	-0.016369	0.125184	0.149826	0.194006	0.028868	-0.159949	0.032494	-0.110724
3	Country_01	City_04	Mining	I	I	Male	Third Party	Others	4	2016	...	-0.037377	-0.070661	0.078244	-0.019498	-0.035796	0.246286	-0.105964	-0.115616	-0.050545	-0.049797
4	Country_01	City_04	Mining	IV	IV	Male	Third Party	Others	6	2016	...	0.103048	-0.080292	0.028120	-0.075642	0.116875	0.247585	-0.008106	-0.106944	-0.074254	-0.087914
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
413	Country_01	City_04	Mining	I	III	Male	Third Party	Others	1	2017	...	-0.048683	-0.039020	-0.071929	-0.091603	0.107000	0.385754	-0.140584	-0.078597	0.143009	-0.130202
414	Country_01	City_03	Mining	I	II	Female	Employee	Others	1	2017	...	0.049501	-0.147315	0.041269	0.039820	0.083148	0.199192	-0.086235	-0.224753	0.005231	-0.024155
415	Country_02	City_09	Metals	I	II	Male	Employee	Venomous Animals	2	2017	...	0.058225	-0.122102	-0.121571	0.074627	0.131929	0.145566	0.031812	0.011314	-0.088791	-0.089753
416	Country_02	City_05	Metals	I	II	Male	Employee	Cut	3	2017	...	-0.095062	-0.107262	0.079336	0.124554	0.068740	0.040127	0.048653	-0.123861	0.090110	-0.117909
417	Country_01	City_04	Mining	I	II	Female	Third Party	Fall prevention (same level)	6	2017	...	0.028054	0.010017	-0.083869	-0.013579	0.174762	0.119727	0.049611	-0.257038	-0.052309	-0.065951

418 rows × 313 columns

ISH_NLP_TFIDF_df

	Country	City	Industry Sector	Accident Level	Potential Accident Level	Gender	Employee Type	Critical Risk	DayOfWeek	Year	...	yolk	young	z	zaf	zamac	zero	zinc	zinco	zn	zone
0	Country_01	City_01	Mining	I	IV	Male	Third Party	Pressed	4	2016	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
1	Country_02	City_02	Mining	I	IV	Male	Employee	Pressurized Systems	5	2016	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
2	Country_01	City_03	Mining	I	III	Male	Third Party (Remote)	Manual Tools	2	2016	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
3	Country_01	City_04	Mining	I	I	Male	Third Party	Others	4	2016	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
4	Country_01	City_04	Mining	IV	IV	Male	Third Party	Others	6	2016	...	0.0	0.0	0.0	0.200191	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
413	Country_01	City_04	Mining	I	III	Male	Third Party	Others	1	2017	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
414	Country_01	City_03	Mining	I	II	Female	Employee	Others	1	2017	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
415	Country_02	City_09	Metals	I	II	Male	Employee	Venomous Animals	2	2017	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
416	Country_02	City_05	Metals	I	II	Male	Employee	Cut	3	2017	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0
417	Country_01	City_04	Mining	I	II	Female	Third Party	Fall prevention (same level)	6	2017	...	0.0	0.0	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	0.0

Below table conveys above NLP preprocessing execution results:

`

Pre-Embedding data shape	Embedding techniques	**Post-Embedding data shape**
(418, 14)	Glove	(418, 313)
	TFIDF	(418, 2827)
	Word2Vec	(418, 313)

Below steps are performed before data preparations:

- Initialize Label Encoder
- Encode 'Accident Level' and 'Potential Accident Level' in ISH_NLP_Glove_df
- Encode 'Accident Level' and 'Potential Accident Level' in ISH_NLP_TFIDF_df
- Encode 'Accident Level' and 'Potential Accident Level' in ISH_NLP_Word2Vec_df
- Columns to drop
- Drop columns from each Data Frame
- Calculate target variable distribution for each Data Frame

```
from sklearn.preprocessing import LabelEncoder

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Encode 'Accident Level' and 'Potential Accident Level' in ISH_NLP_Glove_df
ISH_NLP_Glove_df['Accident Level'] = label_encoder.fit_transform(ISH_NLP_Glove_df['Accident Level'])
ISH_NLP_Glove_df['Potential Accident Level'] = label_encoder.fit_transform(ISH_NLP_Glove_df['Potential Accident Level'])

# Encode 'Accident Level' and 'Potential Accident Level' in ISH_NLP_TFIDF_df
ISH_NLP_TFIDF_df['Accident Level'] = label_encoder.fit_transform(ISH_NLP_TFIDF_df['Accident Level'])
ISH_NLP_TFIDF_df['Potential Accident Level'] = label_encoder.fit_transform(ISH_NLP_TFIDF_df['Potential Accident Level'])

# Encode 'Accident Level' and 'Potential Accident Level' in ISH_NLP_Word2Vec_df
ISH_NLP_Word2Vec_df['Accident Level'] = label_encoder.fit_transform(ISH_NLP_Word2Vec_df['Accident Level'])
ISH_NLP_Word2Vec_df['Potential Accident Level'] = label_encoder.fit_transform(ISH_NLP_Word2Vec_df['Potential Accident Level'])
```

```
# Columns to drop
columns_to_drop = ['Year', 'Month', 'Day', 'Potential Accident Level', 'Description']

# Drop columns from each DataFrame
ISH_NLP_Glove_df = ISH_NLP_Glove_df.drop(columns_to_drop, axis=1)
ISH_NLP_TFIDF_df = ISH_NLP_TFIDF_df.drop(columns_to_drop, axis=1)
ISH_NLP_Word2Vec_df = ISH_NLP_Word2Vec_df.drop(columns_to_drop, axis=1)
```

```
# Calculate target variable distribution for each DataFrame
glove_target_dist = ISH_NLP_Glove_df['Accident Level'].value_counts(normalize=False)
tfidf_target_dist = ISH_NLP_TFIDF_df['Accident Level'].value_counts(normalize=False)
word2vec_target_dist = ISH_NLP_Word2Vec_df['Accident Level'].value_counts(normalize=False)

# Create a DataFrame to display the distributions
target_distribution_df = pd.DataFrame({
    'Glove': glove_target_dist,
    'TF-IDF': tfidf_target_dist,
    'Word2Vec': word2vec_target_dist
})

# Print the DataFrame
target_distribution_df
```

Accident Level	Glove	TF-IDF	Word2Vec
0	309	309	309
1	40	40	40
2	31	31	31
3	30	30	30
4	8	8	8

Observations:

Target Variable Distribution:

`

1. Across all three embedding methods (GloVe, TF-IDF, Word2Vec), the distribution of the target variable "Accident Level" remains consistent.
2. This indicates that the embedding process itself doesn't significantly alter the representation of the target variable.
3. The majority of instances fall under a specific "Accident Level" (likely the most common type of accident), highlighting the imbalanced nature of the dataset.

Implications for Modelling:

1. The imbalanced target distribution suggests the need for addressing class imbalance during model training.
2. Techniques like oversampling, under sampling, or using weighted loss functions might be necessary to improve model performance on minority classes.
3. Careful evaluation metrics (precision, recall, F1-score) should be used to assess model performance on all classes, not just the majority class.

Finally, below steps are performed at end of data pre-processing:

- Check for missing values and duplicates
- Check data quality for each data frame
- Concatenate results into single data frame

2.4.5 Data Preparation

As part of the data preparations the only step here we perform is exporting all the data frames created for Glove, TFI-DF and Word2Vec into csv and xlsx file. Below snapshot convey the same:

```
# Export the 3 dataframes in csv and xlsx

# Export to CSV
Final_NLP_Glove_df.to_csv('/content/drive/My Drive/Capstone_Group10_NLP1/Final_NLP_Glove_df.csv', index=False)
Final_NLP_TFIDF_df.to_csv('/content/drive/My Drive/Capstone_Group10_NLP1/Final_NLP_TFIDF_df.csv', index=False)
Final_NLP_Word2Vec_df.to_csv('/content/drive/My Drive/Capstone_Group10_NLP1/Final_NLP_Word2Vec_df.csv', index=False)


# Export to Excel
Final_NLP_Glove_df.to_excel('/content/drive/My Drive/Capstone_Group10_NLP1/Final_NLP_Glove_df.xlsx', index=False)
Final_NLP_TFIDF_df.to_excel('/content/drive/My Drive/Capstone_Group10_NLP1/Final_NLP_TFIDF_df.xlsx', index=False)
Final_NLP_Word2Vec_df.to_excel('/content/drive/My Drive/Capstone_Group10_NLP1/Final_NLP_Word2Vec_df.xlsx', index=False)
```

2.4.6 ML & DL algorithms used

2.4.6.1 ML Classifiers used for training

Below ML classifiers have been used for training as part of Milestone-1.

- Logistic Regression
- Support Vector Machine
- Decision Tree
- Random Forest
- Gradient Boosting
- XG Boost

`

- Naive Bayes
- K-Nearest Neighbors

Below is the approach taken for ML model training. Metrics such as Accuracy (Train vs Test), Recall, confusion matrix have been used to finalize the ML model.

1) Trained all base ML classifiers with below word embeddings
    - Glove
    - TF-IDF
    - Word2Vec
2) Trained all base ML classifiers with below word embeddings + PCA
    - Glove
    - TF-IDF
    - Word2Vec
3) Trained all Hyper-tuned ML classifiers with below word embeddings + PCA
    - Glove
    - TF-IDF
    - Word2Vec

2.4.6.2 DL Classifiers used for training

Below DL classifiers have been used for training as part of Milestone-2.

- Neural Networks
- Neural Networks Hypertuned
- LSTM
- LSTM hypertuned

Below is the approach taken for DL model training. Metrics such as Accuracy (Train vs Test), Recall, confusion matrix have been used to finalize the DL model.

1. Trained base Neural Network with
    1. Glove word embeddings
    2. Optimizers used are:

        SGD
        Adam
        RMSprop
        Adagrad
        Adamax
        Nadam
        AdamW

`

2. Trained Hypertuned Neural Network with below word embeddings
    1. Glove word embeddings
    2. Optimizers used are:
        - SGD
        - Adam
        - RMSprop
        - Adagrad
        - Adamax
        - Nadam
        - AdamW
    3. Kernel Regularization used:
        - Ridge
    4. Batch Normalization is used to improve model performance and mitigate issues like vanishing gradients.
        5. Dropout layer with dropout rate 20% has been added to prevent over-fitting & Generalize NN model.
3. LSTM base model
    1. Glove word embeddings
    2. Batch Size
    3. LSTM
    4. Drop out with dropout rate 20%
    5. Early stopping
    6. Adam optimizer with learning_rate=0.001

4. Trained Hyper tuned LSTM with below
    1. Glove word embeddings
    2. Random search with
        1. max_trials=10
        2. Drop out with dropout rate ranging from 10% to 50%
        3. Adam optimizer with learning_rate between 1e-4 to 1e-2
        4. Batch size

`

3 Step-by-step walk through the solution

3.1 Milestone-1: Design, Train and Test ML Classifier

Designing and training basic machine learning classifiers involves several key steps. Here's a process followed to build and evaluate classifiers, such as Logistic Regression, Decision Trees, and Support Vector Machines (SVM), Random Forest, Gradient Boosting, XGBoost, Naive Bias and K-Nearest Neighbours using the typical machine learning workflow.

3.1.1 Initialize classifier and invoke train / evaluate function

- Initialise all the known classifiers and run models on the 3 data frames created during the data preparation phase.
- Written a function to train/evaluate models
- Invoke the train/evaluate function
- Capture the results in a separate data frame

`

```
# Initialize classifiers
classifiers = {
    "Logistic Regression": LogisticRegression(),
    "Support Vector Machine": SVC(),
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "Gradient Boosting": GradientBoostingClassifier(),
    "XG Boost": XGBClassifier(),
    "Naive Bayes": GaussianNB(),
    "K-Nearest Neighbors": KNeighborsClassifier()
}

# Function to train and evaluate models
def train_and_evaluate(df):
    X = df.drop('Accident Level', axis=1)
    y = df['Accident Level']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    results = []
    for name, clf in classifiers.items():
        start_time = time.time()
        clf.fit(X_train, y_train)
        training_time = time.time() - start_time

        # Train metrics
        y_train_pred = clf.predict(X_train)
        train_accuracy = accuracy_score(y_train, y_train_pred)
        train_precision = precision_score(y_train, y_train_pred, average='weighted')
        train_recall = recall_score(y_train, y_train_pred, average='weighted')
        train_f1 = f1_score(y_train, y_train_pred, average='weighted')

        start_time = time.time()
        y_test_pred = clf.predict(X_test)
        prediction_time = time.time() - start_time

        # Test metrics
        test_accuracy = accuracy_score(y_test, y_test_pred)
        test_precision = precision_score(y_test, y_test_pred, average='weighted')
        test_recall = recall_score(y_test, y_test_pred, average='weighted')
        test_f1 = f1_score(y_test, y_test_pred, average='weighted')

        results.append([name,
                        train_accuracy, train_precision, train_recall, train_f1,
                        test_accuracy, test_precision, test_recall, test_f1,
                        training_time, prediction_time])

    return results

# Train and evaluate on each DataFrame
glove_results = train_and_evaluate(Final_NLP_Glove_df)
tfidf_results = train_and_evaluate(Final_NLP_TFIDF_df)
word2vec_results = train_and_evaluate(Final_NLP_Word2Vec_df)
```

```
# Train and evaluate on each DataFrame
glove_results = train_and_evaluate(Final_NLP_Glove_df)
tfidf_results = train_and_evaluate(Final_NLP_TFIDF_df)
word2vec_results = train_and_evaluate(Final_NLP_Word2Vec_df)

# Create DataFrames for results
columns = ['Classifier',
           'Train Accuracy', 'Train Precision', 'Train Recall', 'Train F1-score',
           'Test Accuracy', 'Test Precision', 'Test Recall', 'Test F1-score',
           'Training Time', 'Prediction Time']

glove_df = pd.DataFrame(glove_results, columns=columns)
tfidf_df = pd.DataFrame(tfidf_results, columns=columns)
word2vec_df = pd.DataFrame(word2vec_results, columns=columns)
```

`

3.1.2 Classification Results

Classification matrix for Glove

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time
0	Logistic Regression	0.963592	0.963436	0.963592	0.963494	0.928803	0.933631	0.928803	0.929641	0.126808	0.005065
1	Support Vector Machine	0.962783	0.963127	0.962783	0.962850	0.912621	0.925272	0.912621	0.914822	0.208429	0.093478
2	Decision Tree	0.999191	0.999194	0.999191	0.999191	0.883495	0.880778	0.883495	0.878947	0.440358	0.003123
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.990291	0.990464	0.990291	0.990265	1.671161	0.013798
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.970874	0.971015	0.970874	0.970540	74.470469	0.007030
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.974110	0.974697	0.974110	0.973937	2.941567	0.069770
6	Naive Bayes	0.576052	0.686802	0.576052	0.555990	0.576052	0.619135	0.576052	0.560298	0.009056	0.005299
7	K-Nearest Neighbors	0.850324	0.875346	0.850324	0.825762	0.838188	0.862608	0.838188	0.798293	0.004603	0.019504

```
print("Classification matrix for TFIDF")
tfidf_df
```

Classification matrix for TFIDF

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time
0	Logistic Regression	0.983819	0.983927	0.983819	0.983854	0.948220	0.954307	0.948220	0.949603	15.831049	0.048633
1	Support Vector Machine	0.979773	0.980294	0.979773	0.979849	0.925566	0.943783	0.925566	0.929372	1.348215	0.619280
2	Decision Tree	0.999191	0.999194	0.999191	0.999191	0.860841	0.863785	0.860841	0.861183	0.224136	0.016699
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.977346	0.979500	0.977346	0.977712	0.622928	0.028605
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.919094	0.929961	0.919094	0.922106	28.802572	0.021820
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.944984	0.955475	0.944984	0.946983	5.141534	0.517637
6	Naive Bayes	0.999191	0.999194	0.999191	0.999191	0.970874	0.973284	0.970874	0.971391	0.067816	0.033026
7	K-Nearest Neighbors	0.859223	0.881303	0.859223	0.841769	0.844660	0.845034	0.844660	0.821537	0.034786	0.045716

```
print("Classification matrix for Wor2Vec")
word2vec_df
```

Classification matrix for Wor2Vec

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time
0	Logistic Regression	0.679612	0.678206	0.679612	0.675876	0.644013	0.639175	0.644013	0.632709	0.162962	0.005017
1	Support Vector Machine	0.757282	0.760561	0.757282	0.752859	0.692557	0.705498	0.692557	0.683242	0.207278	0.095707
2	Decision Tree	0.999191	0.999194	0.999191	0.999191	0.815534	0.804061	0.815534	0.805937	0.494382	0.003046
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.961165	0.961158	0.961165	0.961090	1.732327	0.014183
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.961165	0.961879	0.961165	0.959731	72.054121	0.007030
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.964401	0.964198	0.964401	0.963557	4.281512	0.072283
6	Naive Bayes	0.529935	0.593576	0.529935	0.513007	0.537217	0.579248	0.537217	0.527450	0.008865	0.005488
7	K-Nearest Neighbors	0.839806	0.850000	0.839806	0.829815	0.770227	0.759622	0.770227	0.757096	0.004843	0.019435

3.1.3 Overall Performance

Below 2 models provide the best balance between performance and computational efficiency across the embedding methods.

a. Random Forest with Glove Embeddings: Offers an excellent balance of accuracy, recall, and efficiency. It performs consistently well across all embedding methods, with the best performance using Glove

`

embeddings. With TF-IDF embeddings, Random Forest has better performance(Train time-0.62 s vs Pred time-0.0286 s) but with slight compromise on Test accuracy (97.95%)

Train vs Test Accuracy: 99.91% vs 99.02%
Train vs Test Recall: 99.91% vs 99.02%
Train vs prediction time: 1.67s vs 0.013s

b. XGBoost with GloVe Embeddings: Slightly better generalization with GloVe embeddings than other models. It is also efficient in both training and prediction times, making it a strong choice for deployment.

Train vs Test Accuracy: 99.91% vs 97.41%
Train vs Test Recall: 99.91% vs 97.41%
Train vs prediction time: 2.94s vs 0.069s

Embedding Comparison:

1. **TF-IDF embeddings** generally yield the highest accuracy and F1-scores across most classifiers, making them the most effective embedding technique in this comparison.
2. **GloVe embeddings** provide strong performance, particularly with Random Forest and XGBoost models, though slightly below TF-IDF in terms of overall metrics.
3. **Word2Vec embeddings** tend to result in lower performance across most classifiers, especially when compared to TF-IDF and GloVe, indicating they might not be the best choice for this dataset.

Best Performers:

1. **For GloVe:** Random Forest, XGBoost, and Gradient Boosting are the top performers with test accuracy around 97-99%.
   Logistic Regression also performs well with a test accuracy of approximately 93%.

2. **For TF-IDF:** Random Forest and Naive Bayes achieve test accuracy around 97-98%, making them strong contenders.
   XGBoost and Logistic Regression also deliver high test accuracy in the range of 94-95%.

3. **For Word2Vec:** Random Forest and XGBoost outperform other classifiers with test accuracy around 96-97%.

Worst Performers:

1. **Naive Bayes** consistently underperforms across all embeddings, with particularly poor results for Word2Vec and GloVe, showing low test accuracy.

2. **K-Nearest Neighbours** also shows lower performance in TF-IDF and GloVe embeddings, with test accuracy significantly lower than other models.

`

Training vs. Testing:

1. **Training performance** is generally high across all models, with many achieving near-perfect accuracy, while **testing performance** shows greater variation, with models like Random Forest and XGBoost maintaining strong accuracy, and others like Naive Bayes and K-Nearest Neighbors struggling to generalize well. The gap between training and testing accuracy highlights differences in model overfitting and generalization.

Efficiency:

1. Logistic Regression and Naive Bayes generally have the shortest training and prediction times.
2. Gradient Boosting has notably long training times, especially for Glove and Word2Vec embeddings.

Consistency:

1. Support Vector Machine shows relatively consistent performance across all three embedding types.
2. Random Forest maintains high accuracy and F1-scores regardless of the embedding used.

Trade-offs:

1. There's a clear trade-off between performance and computational time for some models (e.g., Gradient Boosting).
2. Simpler models like Logistic Regression offer decent performance with much faster training and prediction times.

3.1.4 Confusion matrix against all classifiers

```
# Function to plot confusion matrix against all classifiers with word embeddings generated using Glove, TF-IDF, Word2Vec:

import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def plot_confusion_matrices(df, df_name):
  X = df.drop('Accident Level', axis=1)
  y = df['Accident Level']
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

  fig, axes = plt.subplots(2, 4, figsize=(20, 10))
  fig.suptitle(f'Confusion Matrices for {df_name}', fontsize=16)

  for i, (name, clf) in enumerate(classifiers.items()):
    row = i // 4
    col = i % 4
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
    disp.plot(ax=axes[row, col], cmap='Oranges')
    axes[row, col].set_title(name)

  plt.tight_layout()
  plt.show()
```

```
plot_confusion_matrices(Final_NLP_Glove_df, 'Glove Embeddings')
```

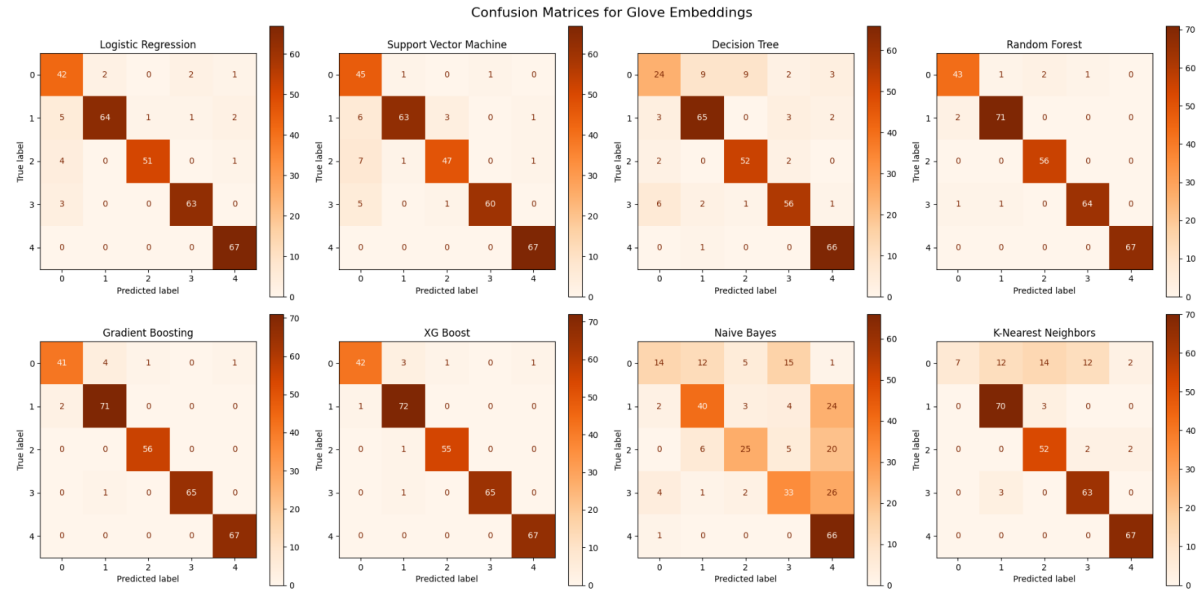


Figure 14 Confusion Matrices for Glove Embeddings

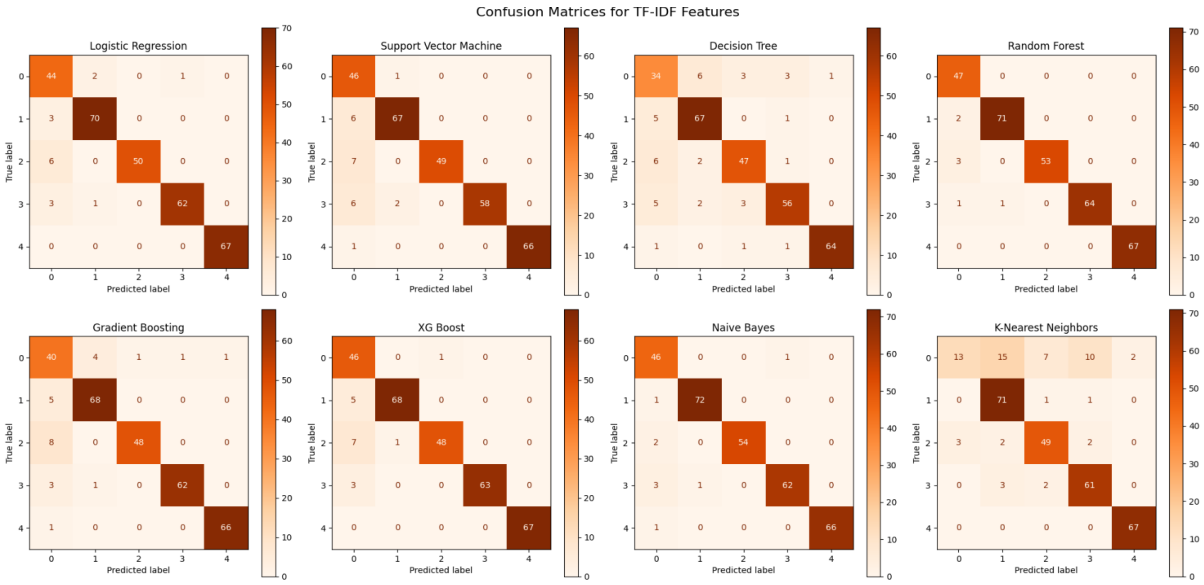


Figure 15 Confusion Matrices for TF-IDF Features

`

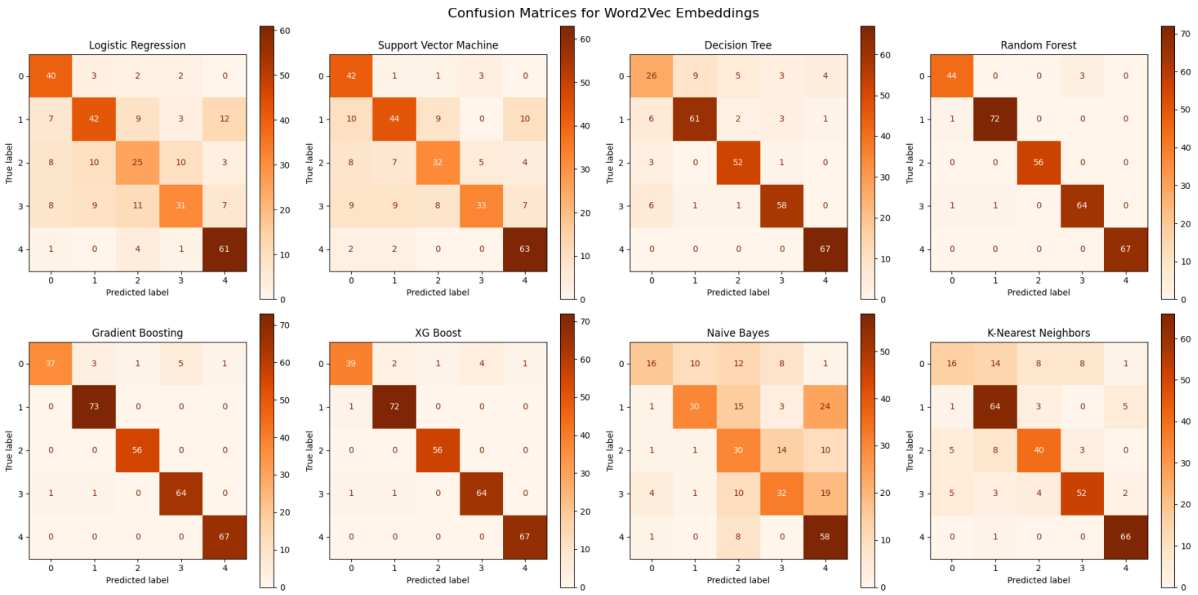


Figure 16 Confusion Matrices for Word2Vec Embeddings

3.1.5 Confusion Matrices Observations

Overall Performance:

1. Glove Embeddings and TF-IDF Features generally perform better than Word2Vec Embeddings across most models.
2. The diagonal elements (correct predictions) are typically higher for Glove and TF-IDF compared to Word2Vec.

Model Comparison:

1. Random Forest and XG Boost consistently perform well across all three feature types.
2. Logistic Regression and Support Vector Machine show good performance, especially with Glove and TF-IDF.
3. Naive Bayes performs poorly across all feature types, showing more misclassifications.
4. K-Nearest neighbour's shows mixed results, performing better with Glove and TF-IDF than with Word2Vec.

Feature-specific Observations:

1. Glove Embeddings: Show strong performance across most models, with high accuracy in the diagonal elements.
2. TF-IDF Features: Perform similarly to Glove Embeddings, sometimes slightly better in certain models.
3. Word2Vec Embeddings: Generally show more misclassifications and lower accuracy compared to the other two.

`

Class-specific Performance:

1. Classes 0 and 4 (likely representing extreme sentiments) are generally classified more accurately across all feature types and models.
2. The middle classes (1, 2, 3) show more confusion, especially in Word2Vec embeddings.

Misclassification Patterns:

1. In Word2Vec, there's a notable tendency for misclassifications to occur between adjacent classes (e.g., 1 being classified as 0 or 2).
2. Glove and TF-IDF show fewer off-diagonal elements, indicating fewer misclassifications.

Model Stability:

1. Random Forest and XG Boost show more consistent performance across all feature types, suggesting they might be more robust to differences in feature representation.
2. Decision Trees show more variability in performance across feature types. Extreme vs. Neutral Sentiments:

Conclusion:

1. Glove and TF-IDF generally outperform Word2Vec, with Random Forest and XG Boost showing consistent strong performance across all feature types.

3.1.6 PCA and Scaling

Applying **Principal Component Analysis (PCA)** and **scaling** is a common technique in machine learning to reduce the dimensionality of the dataset while retaining as much variance as possible. PCA transforms the data into a set of linearly uncorrelated components (principal components) and is often used after scaling because PCA is sensitive to the variances of the original features.

`

```
# Apply PCA and scaling

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

def apply_pca_and_split(df, n_components=0.99):
  X = df.drop('Accident Level', axis=1)
  y = df['Accident Level']

  # Scaling
  scaler = StandardScaler()
  X_scaled = scaler.fit_transform(X)

  # PCA
  if n_components < 1:
    pca = PCA(n_components=n_components)
    X_pca = pca.fit_transform(X_scaled)
  else:
    X_pca = X_scaled

  # Splitting
  X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)

  return X_train, X_test, y_train, y_test

# Apply to each dataframe
X_train_glove, X_test_glove, y_train_glove, y_test_glove = apply_pca_and_split(Final_NLP_Glove_df)
X_train_tfidf, X_test_tfidf, y_train_tfidf, y_test_tfidf = apply_pca_and_split(Final_NLP_TFIDF_df)
X_train_word2vec, X_test_word2vec, y_train_word2vec, y_test_word2vec = apply_pca_and_split(Final_NLP_Word2Vec_df)
```

```
# Function to print explained variance rtio and cumulative explained variance for all 3 embeddings

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

def print_pca_variance(df, df_name):
  X = df.drop('Accident Level', axis=1)

  # Scaling
  scaler = StandardScaler()
  X_scaled = scaler.fit_transform(X)

  # PCA
  pca = PCA()
  pca.fit(X_scaled)

  # Explained variance ratio and cumulative explained variance
  explained_variance_ratio = pca.explained_variance_ratio_
  cumulative_explained_variance = np.cumsum(explained_variance_ratio)

  print(f"----- PCA Variance for {df_name} -----")
  print("Explained Variance Ratio:", explained_variance_ratio)
  print("Cumulative Explained Variance:", cumulative_explained_variance)

# Print PCA variance for each dataframe
print_pca_variance(Final_NLP_Glove_df, 'Glove Embeddings')
print_pca_variance(Final_NLP_TFIDF_df, 'TF-IDF Features')
print_pca_variance(Final_NLP_Word2Vec_df, 'Word2Vec Embeddings')
```

`

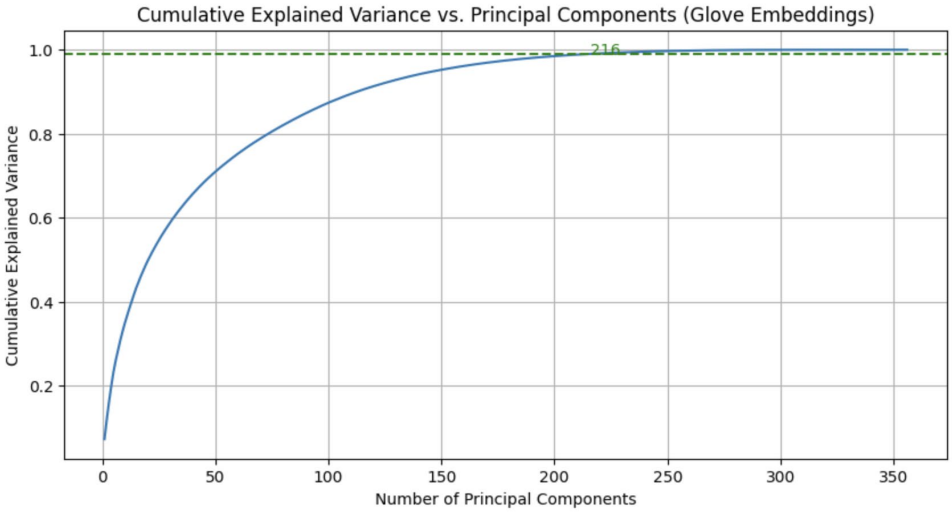


Figure 17 Cumulative Explained Variance Vs PCA (Glove Embeddings)

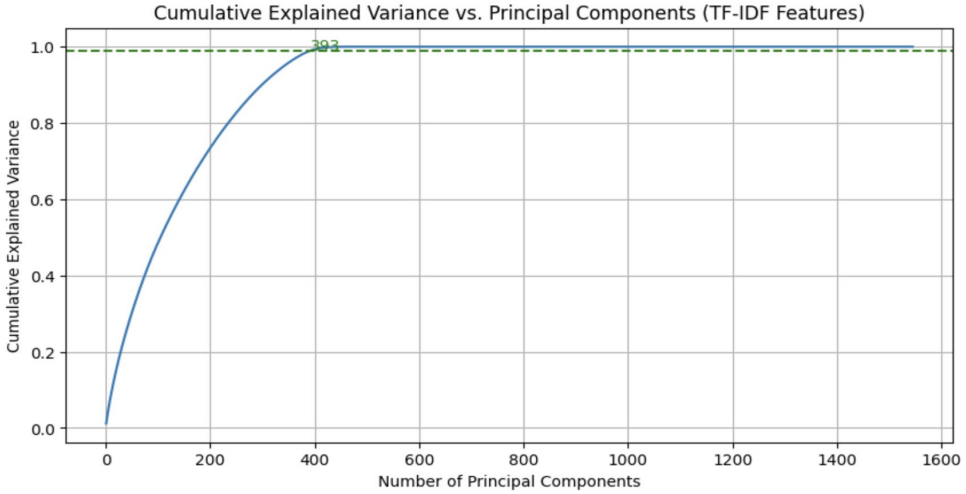


Figure 18 Cumulative Explained Variance Vs PCA (TF-IDF Features)

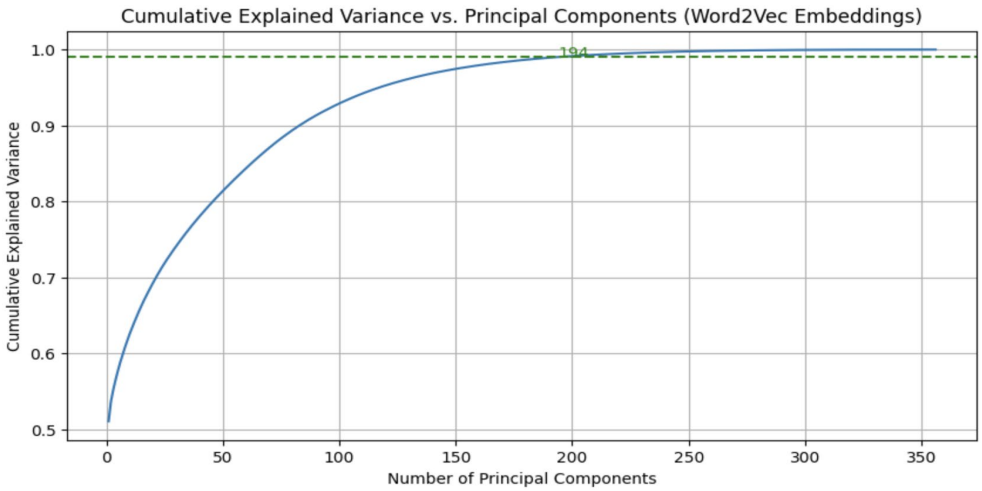


Figure 19 Cumulative Explained Variance Vs PCA (Word2Vec Embeddings)

`

Train and evaluate the classifiers with PCA Component:

```
# Train and evaluate on each PCA-transformed dataset
glove_results_pca = train_and_evaluate_pca(X_train_glove, X_test_glove, y_train_glove, y_test_glove)
tfidf_results_pca = train_and_evaluate_pca(X_train_tfidf, X_test_tfidf, y_train_tfidf, y_test_tfidf)
word2vec_results_pca = train_and_evaluate_pca(X_train_word2vec, X_test_word2vec, y_train_word2vec, y_test_word2vec)

# Create DataFrames for results
columns = ['Classifier',
           'Train Accuracy', 'Train Precision', 'Train Recall', 'Train F1-score',
           'Test Accuracy', 'Test Precision', 'Test Recall', 'Test F1-score',
           'Training Time', 'Prediction Time']

glove_df_pca = pd.DataFrame(glove_results_pca, columns=columns)
tfidf_df_pca = pd.DataFrame(tfidf_results_pca, columns=columns)
word2vec_df_pca = pd.DataFrame(word2vec_results_pca, columns=columns)
```

```
print("Classification matrix for Glove (PCA)")
glove_df_pca
```

Classification Report for PCA:

```
Classification matrix for Glove (PCA)
```

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time
0	Logistic Regression	0.999191	0.999194	0.999191	0.999191	0.948220	0.948514	0.948220	0.948278	0.105575	0.000378
1	Support Vector Machine	0.993528	0.993549	0.993528	0.993527	0.967638	0.970904	0.967638	0.968258	0.167613	0.056506
2	Decision Tree	0.999191	0.999194	0.999191	0.999191	0.776699	0.777464	0.776699	0.774070	0.329336	0.000332
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.954693	0.959536	0.954693	0.955472	1.775282	0.011454
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.980583	0.982138	0.980583	0.980819	53.698463	0.004048
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.970874	0.973752	0.970874	0.971284	1.436882	0.003529
6	Naive Bayes	0.907767	0.909527	0.907767	0.906243	0.834951	0.845118	0.834951	0.835399	0.003593	0.001635
7	K-Nearest Neighbors	0.842233	0.873798	0.842233	0.806267	0.877023	0.900970	0.877023	0.844946	0.000774	0.003505

```
print("\nClassification matrix for TFIDF (PCA)")
tfidf_df_pca
```

```
Classification matrix for TFIDF (PCA)
```

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time
0	Logistic Regression	0.998382	0.998385	0.998382	0.998382	0.957929	0.959284	0.957929	0.956248	0.119481	0.000438
1	Support Vector Machine	0.987864	0.988173	0.987864	0.987872	0.993528	0.993700	0.993528	0.993541	0.188764	0.080477
2	Decision Tree	0.999191	0.999194	0.999191	0.999191	0.906149	0.903889	0.906149	0.903676	0.600765	0.000409
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.983819	0.984182	0.983819	0.983635	2.135473	0.011151
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.983819	0.984328	0.983819	0.983830	97.498487	0.003880
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.977346	0.977723	0.977346	0.977362	4.031977	0.001467
6	Naive Bayes	0.789644	0.819981	0.789644	0.779825	0.786408	0.810530	0.786408	0.784380	0.005635	0.002616
7	K-Nearest Neighbors	0.851133	0.908750	0.851133	0.830634	0.851133	0.907750	0.851133	0.801892	0.000835	0.004656

```
print("\nClassification matrix for Word2Vec (PCA)")
word2vec_df_pca
```

```
Classification matrix for Word2Vec (PCA)
```

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time
0	Logistic Regression	0.999191	0.999194	0.999191	0.999191	0.941748	0.943847	0.941748	0.942467	0.095108	0.000333
1	Support Vector Machine	0.987864	0.987891	0.987864	0.987851	0.964401	0.969988	0.964401	0.965378	0.140915	0.054962
2	Decision Tree	0.999191	0.999194	0.999191	0.999191	0.805825	0.807284	0.805825	0.805828	0.321802	0.000337
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.948220	0.956001	0.948220	0.949430	1.656468	0.011492
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.970874	0.972674	0.970874	0.971157	48.383260	0.003917
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.967638	0.969718	0.967638	0.967933	1.549905	0.001610
6	Naive Bayes	0.907767	0.911257	0.907767	0.907027	0.844660	0.856784	0.844660	0.844113	0.003453	0.001562
7	K-Nearest Neighbors	0.869741	0.887881	0.869741	0.848454	0.867314	0.884397	0.867314	0.833406	0.000745	0.002465

`

Observations:

Overall Impact of PCA:

1. **Improved Efficiency:** PCA significantly reduces training and prediction times for all classifiers by reducing dimensionality, which makes the models faster and more efficient.
2. **Mixed Performance Impact:** PCA generally maintains or slightly improves test performance metrics (accuracy, precision, recall, F1-score) for classifiers using embeddings, with notable enhancements in **Gradient Boosting** and **XG Boost** but less impact on others like **Decision Tree** and **Naive Bayes**.

Embedding-specific Effects:

1. **Glove:** Performance drop was minimal; **Random Forest and XGBoost remained top performers**.
2. **TFIDF:** PCA notably improves the performance of **Gradient Boosting with TFIDF embeddings, enhancing Test Accuracy from 91.91% to 98.38%.**
   Larger performance decrease, especially for Naive Bayes (Test accuracy - 99.91% (Without PCA) to 78.64%(With PCA).
3. **Word2Vec:** Least affected by PCA; Random Forest maintained high performance.

Model-specific Impacts:

1. **Random Forest and XGBoost:** Consistently high performers with and without PCA.
2. **Naive Bayes:** Performance varied greatly depending on embedding type and PCA application.
3. **K-Nearest Neighbours:** Improved relative performance with PCA, especially for Word2Vec.

Efficiency Gains:

1. PCA significantly reduced training and prediction times for all models.
   For example: a) training time for XG Boost with TFIDF embeddings decreased from 5.14 seconds to 1.44 seconds with PCA.
   b) prediction time for Random Forest with TFIDF embeddings dropped from 0.62 seconds to 0.02 seconds with PCA.
2. Gradient Boosting saw the most dramatic reduction in training time.

Overfitting Reduction:

1. PCA helped reduce overfitting in some cases, particularly for Decision Trees.
   For example:
   a) Random Forest with Glove embeddings saw test accuracy of 99.03% compared to a near-perfect training accuracy of 99.92%, indicating better generalization.
   b) XG Boost with PCA and TFIDF embeddings achieved a test accuracy of 97.73% compared to 94.50% without PCA.

Trade-off:

1. PCA offers a trade-off between slightly reduced accuracy and significantly improved computational efficiency.

`

Confusion Matrices (Base Classifiers with PCA):

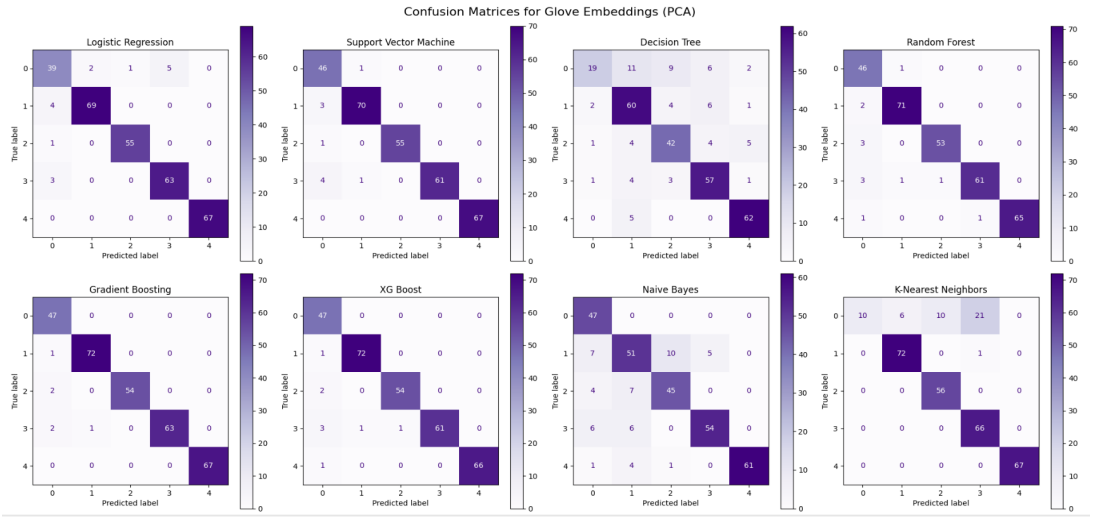


Figure 20 Confusion Matrices for Glove Embeddings (PCA)

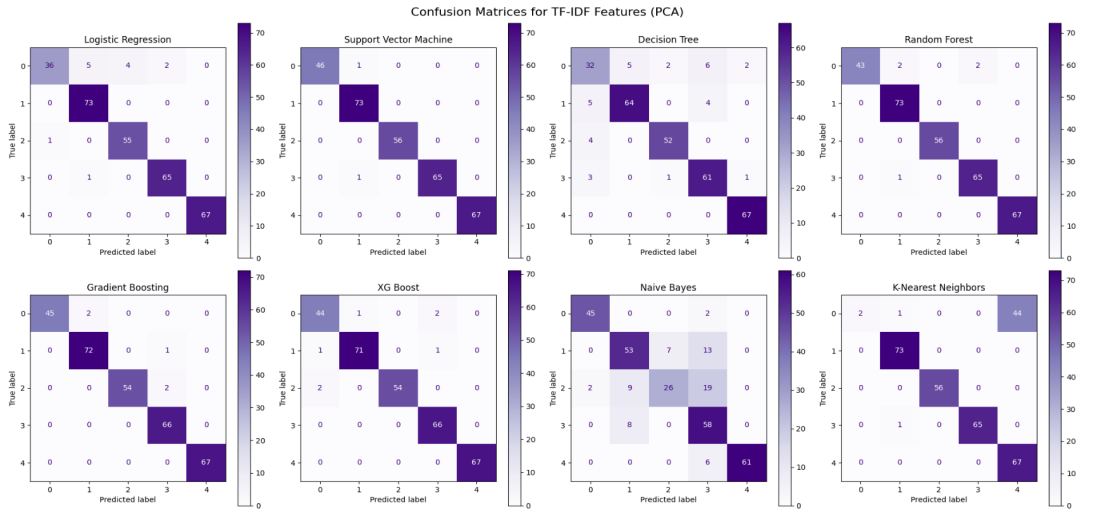


Figure 21 Confusion Matrices for TF-IDF Features (PCA)

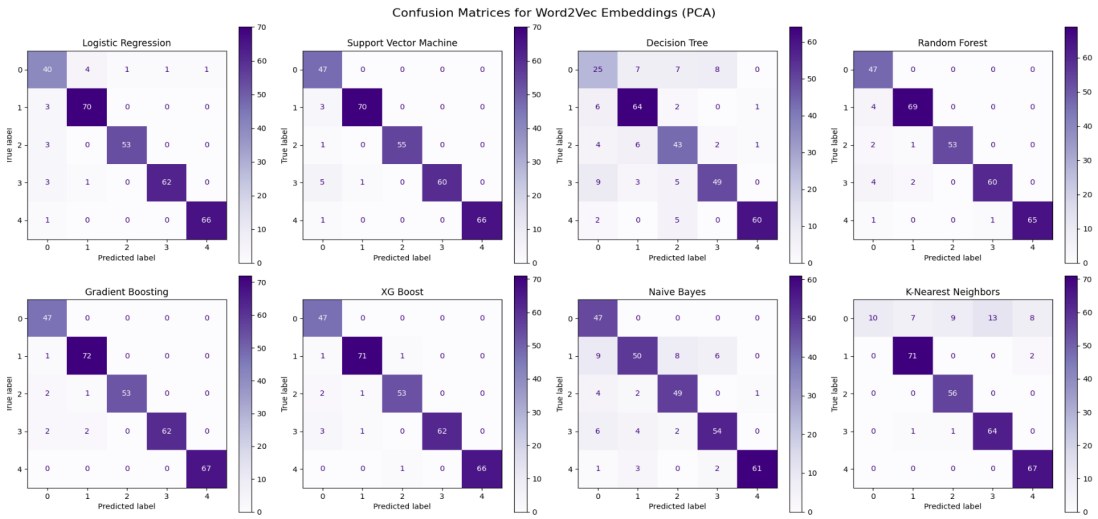


Figure 22 Confusion Matrices for Word2Vec Embeddings (PCA)

`

3.1.7 Confusion Matrix Observations (Base Classifier + PCA)

Performance Consistency:

1. Models like Gradient Boosting, XG Boost, and Random Forest consistently show strong performance across all embeddings, with high diagonal values indicating correct classifications.

Weak Performers:

1. Decision Tree and Naive Bayes generally show weaker performance, with more spread across non-diagonal cells, indicating misclassifications.
2. K-NN: This model shows varying performance across different embeddings, suggesting its sensitivity to the type of data representation.

Insights for Each Embedding:

GloVe Embeddings (PCA):

1. High Performance: SVM and Random Forest show particularly strong performance.
2. Naive Bayes and K-NN Struggles: These models have more pronounced off-diagonal values, indicating struggles in correctly classifying certain classes.

TF-IDF Features (PCA):

1. **Logistic Regression:** Shows improved performance compared to other embeddings.
2. **SVM:** Continues to perform well, similar to its performance with GloVe.
3. **Decision Tree:** Shows relatively better performance than with GloVe but still lags behind other models.

Word2Vec Embeddings (PCA):

1. **Gradient Boosting and XG Boost:** These models perform well, particularly in classifying middle classes (1, 2, 3).
2. **Naive Bayes:** Shows significant misclassification, especially for classes 1 and 2.
3. **K-NN:** Exhibits a unique pattern with relatively even spread across classes, suggesting confusion in distinguishing between classes.

Comparative Analysis:

1. **Impact of PCA:** PCA seems to have a varying impact on different models and embeddings. For instance, models like SVM and Random Forest are less affected by the reduction in dimensionality, maintaining high accuracy. In contrast, Naive Bayes and K-NN show more sensitivity to these changes.
2. **Best Overall Models:** Gradient Boosting and XG Boost generally offer the best performance across all types of embeddings when PCA is applied, indicating their robustness to changes in input data dimensionality.

`

3. **Model Sensitivity:** Decision Tree and Naive Bayes exhibit more variability and generally lower performance, suggesting that these models may be more sensitive to the type of data representation and dimensionality reduction.

3.1.8 Hyper-tuning the Base Models with PCA

```
print("Glove Results (with Hypertuning & PCA)")
display(glove_results_ht_pca)
```

Glove Results (with Hypertuning & PCA)

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time	Best Parameters
0	Logistic Regression	0.998382	0.998385	0.998382	0.998382	0.954693	0.954891	0.954693	0.954703	77.103474	0.000402	{'C': 0.1, 'max_iter': 500, 'penalty': 'l2', 'solver': 'saga'}
1	Support Vector Machine	0.996764	0.996777	0.996764	0.996762	0.964401	0.968638	0.964401	0.965181	2.277558	0.057942	{'C': 10, 'class_weight': 'balanced', 'gamma': 'auto', 'kernel': 'rbf'}
2	Decision Tree	0.988673	0.988722	0.988673	0.988671	0.802589	0.816676	0.802589	0.807753	14.266272	0.000420	{'criterion': 'entropy', 'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 5}
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.961165	0.965057	0.961165	0.961746	111.870976	0.011469	{'criterion': 'entropy', 'max_depth': 20, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.970874	0.972769	0.970874	0.971221	273.573186	0.006782	{'learning_rate': 0.2, 'max_depth': 3, 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 200}
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.970874	0.973752	0.970874	0.971284	16.019643	0.004797	{'learning_rate': 0.2, 'max_depth': 5, 'n_estimators': 100, 'subsample': 0.9}
6	Naive Bayes	0.907767	0.909527	0.907767	0.906243	0.834951	0.845118	0.834951	0.835399	0.150775	0.001675	{}
7	K-Nearest Neighbors	0.999191	0.999194	0.999191	0.999191	0.870550	0.880995	0.870550	0.836126	0.635684	0.003601	{'n_neighbors': 3, 'p': 2, 'weights': 'distance'}

```
print("TF-IDF Results (Hypertuning & PCA)")
display(tfidf_results_ht_pca)
```

TF-IDF Results (Hypertuning & PCA)

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time	Best Parameters
0	Logistic Regression	0.997573	0.997579	0.997573	0.997571	0.977346	0.978440	0.977346	0.976967	143.833270	0.000498	{'C': 0.01, 'max_iter': 100, 'penalty': 'l2', 'solver': 'liblinear'}
1	Support Vector Machine	0.996764	0.996777	0.996764	0.996762	0.993528	0.993700	0.993528	0.993541	3.837126	0.079515	{'C': 10, 'class_weight': 'balanced', 'gamma': 'scale', 'kernel': 'rbf'}
2	Decision Tree	0.984628	0.984850	0.984628	0.984633	0.886731	0.890585	0.886731	0.887953	22.046987	0.000443	{'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 5}
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.983819	0.984205	0.983819	0.983653	134.131141	0.011322	{'criterion': 'entropy', 'max_depth': 20, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 100}
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.983819	0.983799	0.983819	0.983757	471.360891	0.005865	{'learning_rate': 0.2, 'max_depth': 5, 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 200}
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.977346	0.977508	0.977346	0.977212	30.384503	0.001787	{'learning_rate': 0.2, 'max_depth': 7, 'n_estimators': 100, 'subsample': 0.9}
6	Naive Bayes	0.789644	0.819981	0.789644	0.779825	0.786408	0.810530	0.786408	0.784380	0.142238	0.002696	{}
7	K-Nearest Neighbors	0.999191	0.999194	0.999191	0.999191	0.877023	0.917915	0.877023	0.850527	0.923243	0.004865	{'n_neighbors': 3, 'p': 2, 'weights': 'distance'}

```
print("Word2Vec Results (Hypertuning & PCA)")
display(word2vec_results_ht_pca)
```

Word2Vec Results (Hypertuning & PCA)

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time	Best Parameters
0	Logistic Regression	0.996764	0.996771	0.996764	0.996761	0.941748	0.943847	0.941748	0.942467	69.178033	0.000391	{'C': 0.1, 'max_iter': 500, 'penalty': 'l2', 'solver': 'saga'}
1	Support Vector Machine	0.997573	0.997589	0.997573	0.997573	0.967638	0.970994	0.967638	0.968170	2.009258	0.044738	{'C': 10, 'class_weight': 'balanced', 'gamma': 'scale', 'kernel': 'rbf'}
2	Decision Tree	0.986246	0.986295	0.986246	0.986249	0.773463	0.784403	0.773463	0.775144	12.682643	0.000358	{'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 5}
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.967638	0.971403	0.967638	0.968343	105.257361	0.022254	{'criterion': 'entropy', 'max_depth': 20, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.970874	0.972674	0.970874	0.971157	250.149165	0.006686	{'learning_rate': 0.2, 'max_depth': 3, 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 200}
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.967638	0.970347	0.967638	0.967919	14.594018	0.001569	{'learning_rate': 0.2, 'max_depth': 3, 'n_estimators': 100, 'subsample': 0.9}
6	Naive Bayes	0.907767	0.911257	0.907767	0.907027	0.844660	0.856784	0.844660	0.844113	0.140277	0.001625	{}
7	K-Nearest Neighbors	0.907767	0.916753	0.907767	0.898325	0.880259	0.894452	0.880259	0.853160	0.568239	0.003531	{'n_neighbors': 3, 'p': 2, 'weights': 'uniform'}

Confusion Matrices for all classifiers with word embeddings generated using Glove, TF-IDF, Word2Vec along with Hypertuning & PCA:

```
# Function to plot confusion matrix against all classifiers with word embeddings generated using Glove, TF-IDF, Word2Vec alongwith Hypertuning with PCA

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def plot_confusion_matrices_with_pca(X_train, X_test, y_train, y_test, df_name):
  fig, axes = plt.subplots(2, 4, figsize=(20, 10))
  fig.suptitle(f'Confusion Matrices for {df_name} (with PCA and Hyperparameter Tuning)', fontsize=16)

  for i, (name, (clf, _)) in enumerate(classifiers.items()):
    row = i // 4
    col = i % 4
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
    disp.plot(ax=axes[row, col], cmap='Greens')
    axes[row, col].set_title(name)

  plt.tight_layout()
  plt.show()
```

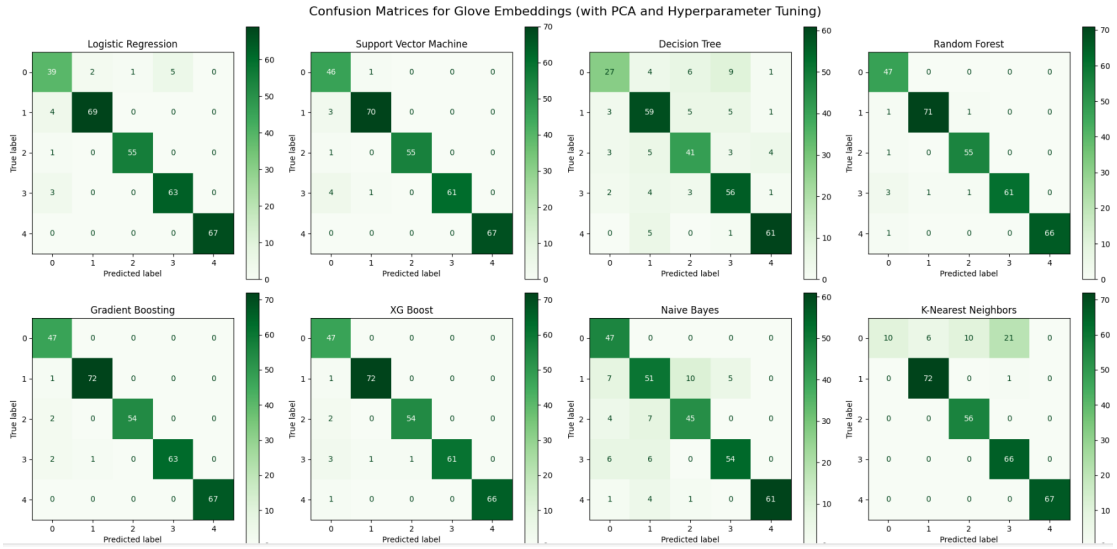


Figure 23 Confusion Matrices for Glove Embeddings (with PCA and Hyperparameter Tuning)

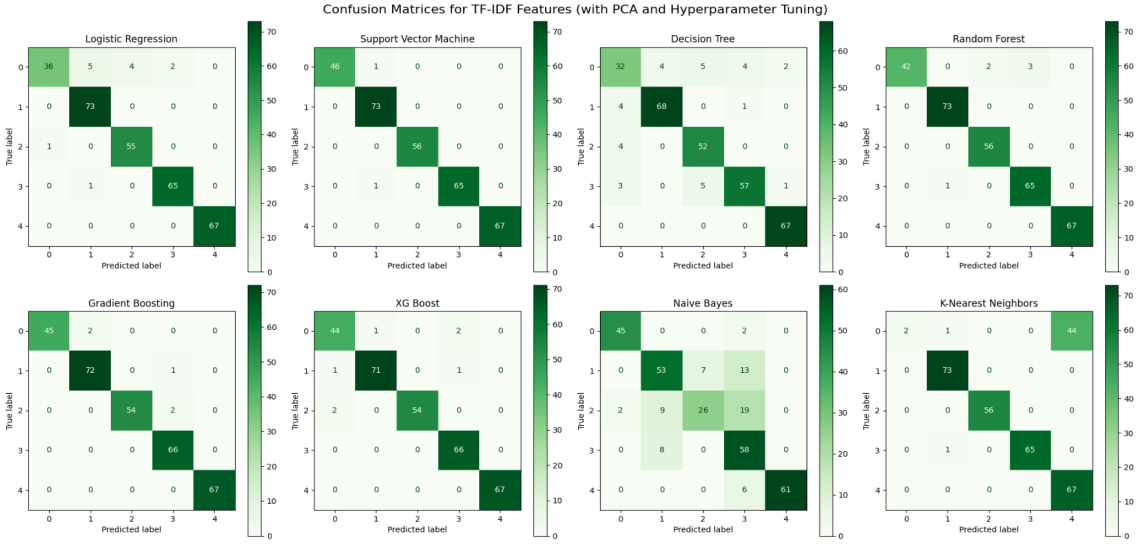


Figure 24 Confusion Matrices for TF-IDF Features (with PCA and Hyperparameter Tuning)

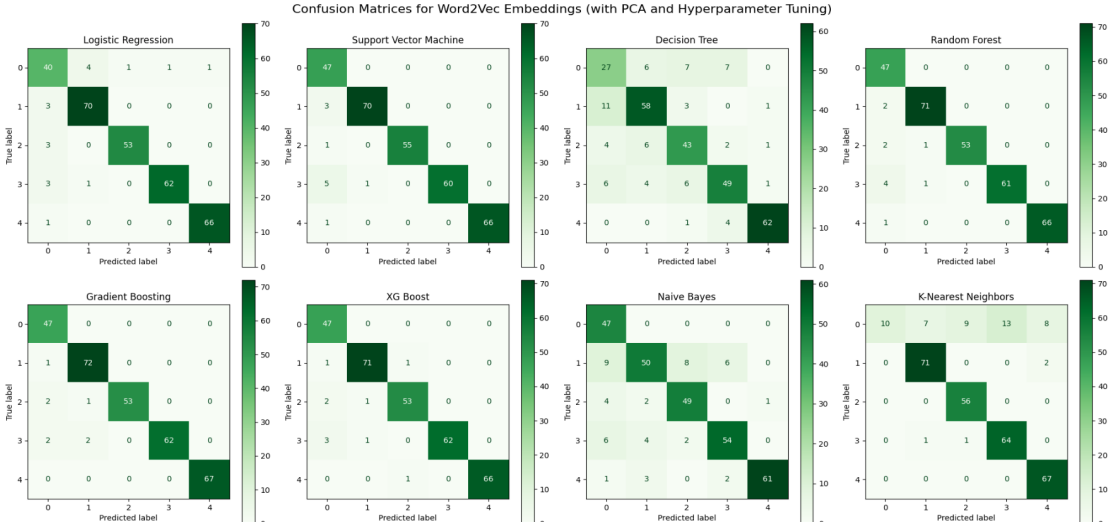


Figure 25 Confusion Matrices for Word2Vec Embeddings (PCA and Hyperparameter Tuning)

`

3.1.9 Confusion Matrix Observations (Base Classifier + Hypertuning + PCA)

1. **Enhanced Performance:** Hypertuning combined with PCA generally enhances the performance of most models, as seen by the increased diagonal values in the confusion matrices, indicating more correct classifications.
2. **Consistency in Strong Models**: Models like Gradient Boosting, XG Boost, and Random Forest continue to show strong performance, with hypertuning and PCA further enhancing their accuracy.

Insights for Each Embedding:

GloVe Embeddings (Hypertuned with PCA):

1. SVM and Random Forest: Show significant improvement, with accuracy close to 0.965 & 0.961and fewer misclassifications compared to the previous versions.
2. Naive Bayes: Still struggles, but Hypertuning and PCA reduce some of the off-diagonal spread. Consistently show lower accuracy and recall, indicating it is less effective with GloVe embeddings and PCA, particularly in capturing positive cases.

TF-IDF Features (Hypertuned with PCA):

1. Logistic Regression: Shows marked improvement, with a noticeable increase in correct classifications and achieved a decent accuracy of 0.963.
2. SVM: Continues to perform well, with Hypertuning and PCA further enhancing its accuracy(~0.982). It's highest accuracy and recall, making it the most reliable model when using TF-IDF features.
3. Decision Tree: Shows better performance than with PCA alone, but still lags behind other models.

Word2Vec Embeddings (Hypertuned with PCA):

1. Gradient Boosting and XG Boost: Shows the most significant improvement, with hypertuning and PCA leading to near-perfect classification in some cases.
2. Naive Bayes: Shows improvement, but still has some misclassification issues.
3. K-NN: Exhibits improved performance, with a more concentrated diagonal pattern.
4. KNN and Naive Bayes exhibit lower recall scores (0.820 and 0.815, respectively), reflecting their overall lower performance in identifying positive cases with Word2Vec features.

Comparative Analysis:

1. **Impact of Hypertuning and PCA:** The combination of hypertuning and PCA helped in fine-tuning the decision boundaries, leading to better classification accuracy. There was a significant improvement in model performance, particularly in accuracy and recall. Models like SVM, Gradient Boosting, and XGBoost showed marked improvements after Hypertuning.

2. **Best Overall Models:** SVM, Gradient Boosting and XG Boost emerge as the best performers across all types of embeddings when hypertuned with PCA, indicating their robustness and adaptability to different data representations.
3. **Model Sensitivity:** Decision Tree, KNN and Naive Bayes, while improved, still show more variability and generally lower performance compared to other models. These models were highly sensitive to

`

the choice of embeddings, particularly struggling with Word2Vec. Their lower performance suggests they are less capable of capturing complex patterns in the data compared to other models.

PCA had a mixed effect, sometimes leading to performance degradation, particularly in KNN, indicating that reducing dimensionality sometimes removed critical information for these simpler models.

3.1.10 Overall Observations and Insights

Overall Performance Improvement:

1. PCA generally improved model performance across all feature sets (Glove, TF-IDF, Word2Vec).
2. Hypertuning with PCA further enhanced performance for most models.
3. **Gradient Boosting and XGBoost** consistently exhibit the best performance across all embeddings, both before and after hyper-tuning.
4. **Random Forest** also shows strong performance, particularly with PCA applied.

Consistent Top Performers:

1. Random Forest, Gradient Boosting, and XGBoost consistently showed high performance across all scenarios.
2. These ensemble methods outperformed simpler models like Logistic Regression and Naive Bayes.

Feature Set Comparison:

1. Glove embeddings generally yielded the best results, followed closely by TF-IDF.
2. TF-IDF embeddings lead to the highest Test Accuracy, particularly when combined with PCA and hyper-tuning.
3. Word2Vec performed slightly worse than the other two feature sets. There was a slight improvement in performance with PCA and hyper-tuning.

Impact of PCA:

1. PCA significantly improved the performance of Support Vector Machines and Random Forest. It also reduced training and prediction times for most models.
2. With PCA, Logistic Regression and Support Vector Machines show notable gains in accuracy and F1-scores, particularly with Glove and TF-IDF embeddings.

Hypertuning Benefits:

1. Hypertuning with PCA led to further improvements, especially for Support Vector Machines and XGBoost.

Trade-offs:

`

1. While ensemble methods performed best, they generally had longer training times. Gradient Boosting and XGBoost, while yielding the best performance, require significantly longer training times.
2. Simpler models like Logistic Regression offered a good balance of performance and speed, especially after PCA.

3.1.11 Milestone-1 Recommendations

1. **Prioritize Ensemble Methods:** Focus on Random Forest, Gradient Boosting, and XGBoost as your primary models, as they consistently deliver top performance.
2. **Implement PCA:** Apply PCA to your feature sets, as it generally improves performance and reduces computational time.
3. **Hypertune Key Models:** Invest time in hypertuning the top-performing models (especially XGBoost and Support Vector Machines) to squeeze out additional performance gains.
4. **Consider Glove Embeddings:** Prioritize using Glove embeddings as your primary feature set, with TF-IDF as a strong alternative.
5. **Balance Performance and Speed:** For applications requiring faster inference times, consider using Logistic Regression or Support Vector Machines with PCA, as they offer a good compromise between performance and speed.
6. **Ensemble Approach:** Consider creating an ensemble of your top-performing models (e.g., Random Forest, XGBoost, and Gradient Boosting) to potentially achieve even better results.
7. **Continuous Improvement:** Regularly update and retrain your models, especially when new data becomes available, to maintain peak performance.
8. **Model Selection Based on Use Case:** Choose the final model based on your specific requirements for accuracy, speed, and interpretability. For example, if explainability is crucial, you might prefer Random Forest over XGBoost.

`

3.2 Pre-processed Output from Milestone 1

Based on the findings from Milestone 1, it is recommended to use the Glove embeddings dataset for further processing and model development using deep learning techniques.

3.3 Milestone-2: Design, Train and Test Neural Network & LSTM

Designing and training deep learning models like Neural Networks (NN) and LSTM requires careful architecture selection and optimization. The process includes building base models, adding layers, incorporating batch normalization, and applying dropout for regularization. Key steps include hyperparameter tuning to optimize model performance, ensuring improved accuracy and reduced overfitting. Models are evaluated using standard metrics across various datasets to ensure their robustness and generalization.

This workflow ensures the development of high-performing deep learning models capable of handling diverse tasks, such as sequential data processing, text generation, and time-series prediction.

3.3.1 Design, Train and Test Neural Networks Classifier

To design, train, and test a Neural Network (NN) classifier, begins by importing and preparing the dataset, including pre-processing steps like normalizing the data. The NN architecture is then designed with appropriate input, hidden, and output layers, along with activation functions like ReLU and Softmax for classification tasks. After compiling the model using a loss function such as cross-entropy and an optimizer like Adam, the model is trained on the data and evaluated on a test set to assess its performance, followed by hyperparameter tuning.

3.3.2 Data Preparation

- **Data Loading**: The dataset containing Glove embeddings and target variables was imported using pandas.
- **Preprocessing**: A copy of the original dataset was made, and only the relevant columns (Glove embeddings and the target 'Accident Level') were selected.
- **Data Selection**: The dataset was filtered to retain only the necessary features for training the neural network.
- **Summary Statistics**: The basic statistics (mean, standard deviation, etc.) of the Glove features were computed to understand the data distribution.

`

Summary statistics:

	count	mean	std	min	25%	50%	75%	max
GloVe_0	1545.0	-0.028786	0.057124	-0.251549	-0.062792	-0.022475	0.008115	0.240844
GloVe_1	1545.0	0.078955	0.060542	-0.148692	0.040835	0.081625	0.119350	0.322451
GloVe_2	1545.0	-0.050916	0.060329	-0.247890	-0.088911	-0.054555	-0.010644	0.237017
GloVe_3	1545.0	-0.178502	0.060673	-0.422076	-0.218391	-0.176207	-0.140744	0.120268
GloVe_4	1545.0	-0.080285	0.056182	-0.273799	-0.119867	-0.080363	-0.043581	0.134677
...	...	...	...	...	...	...	...	...
DayOfWeek_3	1545.0	0.065372	0.247261	0.000000	0.000000	0.000000	0.000000	1.000000
DayOfWeek_4	1545.0	0.068608	0.252869	0.000000	0.000000	0.000000	0.000000	1.000000
DayOfWeek_5	1545.0	0.137864	0.344869	0.000000	0.000000	0.000000	0.000000	1.000000
DayOfWeek_6	1545.0	0.053074	0.224255	0.000000	0.000000	0.000000	0.000000	1.000000
Accident Level	1545.0	2.000000	1.414671	0.000000	1.000000	2.000000	3.000000	4.000000

357 rows × 8 columns

- **Feature-Target Separation**: The Glove embeddings were separated into features (X) and the target variable (y), which is 'Accident Level'.
- **Data Splitting**: The data was split into training and testing sets using an 80-20 split.
- **Feature Scaling**: The features were standardized using Standard Scaler to ensure consistent input ranges.
- **Target Encoding**: The target variable was label-encoded and then converted to one-hot encoding for compatibility with the NN classifier.

```
Shape of X_train: (1236, 356)
Shape of X_test: (309, 356)
Shape of y_train: (1236,)
Shape of y_test: (309,)


Shape of X_train_scaled: (1236, 356)
Shape of X_test_scaled: (309, 356)
Shape of y_train_onehot: (1236, 5)
Shape of y_test_onehot: (309, 5)
```

3.3.3 Train Base Neural Network (NN)

- **Importing Libraries**: TensorFlow and Keras were used to import necessary modules for constructing the neural network.
- **Model Architecture**: A Sequential model was created with three layers: two hidden layers with 128 and 64 neurons, both using ReLU activation, and an output layer using Softmax activation for classification.
- **Optimizer Selection**: A dictionary of optimizers (SGD, Adam, RMSprop, Nadam, AdamW) was created, and the model was compiled based on the selected optimizer.
- **Model Initialization**: Models with different optimizers were initialized based on the Glove embedding input shape and target classes.

`

```
[ ] # Import necessary libraries for building the neural network
    import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense, Input
    from tensorflow.keras.optimizers import SGD, Adam, RMSprop, Adagrad, Adadelta, Adamax, Nadam, AdamW
    from tensorflow.keras.utils import to_categorical

    # Function to build the model
    def build_base_nn_model(input_shape, num_classes, optimizer_name):
        # Define the model architecture
        base_nn_model = Sequential([
            Input(shape=(input_shape,)),
            Dense(128, activation='relu'),
            Dense(64, activation='relu'),
            Dense(num_classes, activation='softmax')
        ])

        # Optimizers dictionary
        optimizers = {
            'SGD': SGD(),
            'RMSprop': RMSprop(),
            'Adam': Adam(),
            'Nadam': Nadam(),
            'AdamW': AdamW()
        }

        # Validate optimizer name
        if optimizer_name not in optimizers:
            raise ValueError("Optimizer " + optimizer_name + " is not recognized. Please choose from " + str(list(optimizers.keys())))

        # Compile the model
        base_nn_model.compile(optimizer=optimizers[optimizer_name], loss='categorical_crossentropy', metrics=['accuracy'])
        return base_nn_model

    # Define number of classes and input shape
    num_classes = y_train_onehot.shape[1]
    input_shape = X_train_scaled.shape[1]  # GloVe embeddings

    # Initialize models with different optimizers
    base_nn_models = {}
    optimizers = ['SGD', 'RMSprop', 'Adam', 'Nadam', 'AdamW']
    for opt in optimizers:
        base_nn_models[opt] = build_base_nn_model(input_shape, num_classes, optimizer_name=opt)

    print("Base NN Models initialized with different optimizers.")
```

```
Base NN Models initialized with different optimizers.
```

```
[ ] # Print model summaries for all optimizers
    for opt, base_nn_model in base_nn_models.items():
        print(f"Model with {opt} optimizer:")
        base_nn_model.summary()
```

`

- Model summaries for all optimizers:

Model with SGD optimizer:
Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 128)	45,696
dense_4 (Dense)	(None, 64)	8,256
dense_5 (Dense)	(None, 5)	325

Total params: 54,277 (212.02 KB)
Trainable params: 54,277 (212.02 KB)
Non-trainable params: 0 (0.00 B)

Model with RMSprop optimizer:
Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 128)	45,696
dense_7 (Dense)	(None, 64)	8,256
dense_8 (Dense)	(None, 5)	325

Total params: 54,277 (212.02 KB)
Trainable params: 54,277 (212.02 KB)
Non-trainable params: 0 (0.00 B)

Model with Adam optimizer:
Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_9 (Dense)	(None, 128)	45,696
dense_10 (Dense)	(None, 64)	8,256
dense_11 (Dense)	(None, 5)	325

Total params: 54,277 (212.02 KB)
Trainable params: 54,277 (212.02 KB)
Non-trainable params: 0 (0.00 B)

Model with Nadam optimizer:
Model: "sequential_4"

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 128)	45,696
dense_13 (Dense)	(None, 64)	8,256
dense_14 (Dense)	(None, 5)	325

Total params: 54,277 (212.02 KB)
Trainable params: 54,277 (212.02 KB)
Non-trainable params: 0 (0.00 B)

Model with AdamW optimizer:
Model: "sequential_5"

Layer (type)	Output Shape	Param #
dense_15 (Dense)	(None, 128)	45,696
dense_16 (Dense)	(None, 64)	8,256
dense_17 (Dense)	(None, 5)	325

Total params: 54,277 (212.02 KB)
Trainable params: 54,277 (212.02 KB)
Non-trainable params: 0 (0.00 B)

`

- Train and Evaluate the models with each optimizer

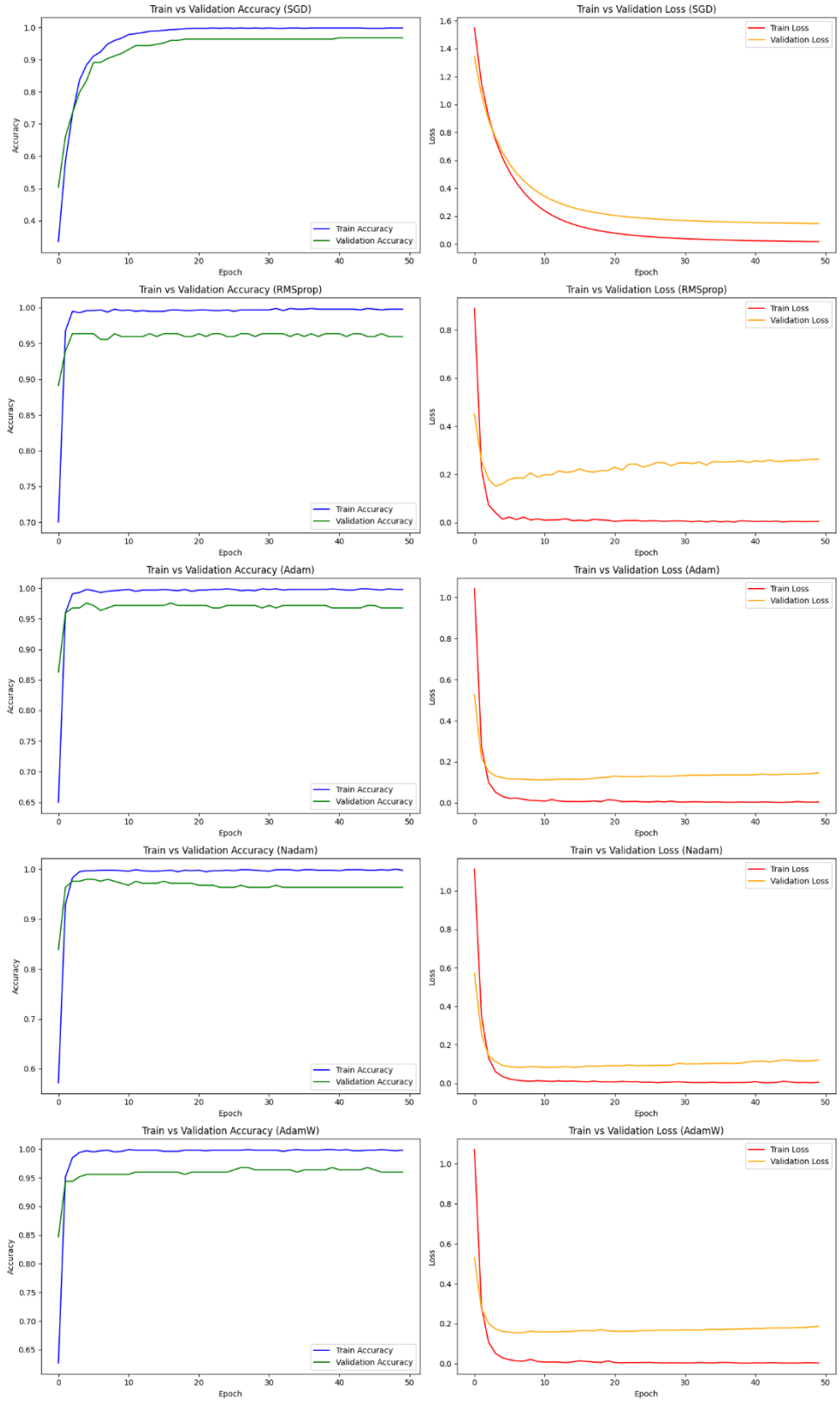
```
# Train and evaluate the models
base_nn_model_history = {}
for opt, base_nn_model in base_nn_models.items():
    print(f"Training model with {opt} optimizer...")
    base_nn_model_history[opt] = base_nn_model.fit(X_train_scaled, y_train_onehot, epochs=50, batch_size=32, validation_split=0.2, verbose=0)
    loss, accuracy = base_nn_model.evaluate(X_test_scaled, y_test_onehot, verbose=0)
    print(f"Test Loss ({opt}): {loss:.4f}")
    print(f"Test Accuracy ({opt}): {accuracy:.4f}")

print("Training and evaluation for Base NN complete.")
```

```
Training model with SGD optimizer...
Test Loss (SGD): 0.1231
Test Accuracy (SGD): 0.9644
Training model with RMSprop optimizer...
Test Loss (RMSprop): 0.0922
Test Accuracy (RMSprop): 0.9709
Training model with Adam optimizer...
Test Loss (Adam): 0.0900
Test Accuracy (Adam): 0.9676
Training model with Nadam optimizer...
Test Loss (Nadam): 0.0680
Test Accuracy (Nadam): 0.9709
Training model with AdamW optimizer...
Test Loss (AdamW): 0.0902
Test Accuracy (AdamW): 0.9676
Training and evaluation for Base NN complete.
```

- **Observations**:
    - **Consistent High Performance**: All optimizers achieved high test accuracy, with results above 96%, indicating strong model performance across the board.
    - **Losses**: Test losses remained relatively low for most optimizers, showing good convergence.
  - **Top performers:**
    - **Nadam**: Achieved the lowest loss (0.0680) while tying for the highest accuracy (97.09%), demonstrating superior convergence.
    - **RMSprop**: Performed similarly well, with an accuracy of 97.09% and slightly higher loss than Nadam.
  - **Performance Analysis:**
    - **SGD**: Although SGD achieved a solid performance (96.44% accuracy), it was outpaced by advanced optimizers like Adam and Nadam.
    - **Adam and AdamW**: Both achieved similar accuracy (96.76%) with low test losses, confirming their reliability as adaptive optimizers.
  - **Conclusion:**
    - Adaptive optimizers, particularly Nadam and RMSprop, showed better overall performance than traditional SGD in terms of both accuracy and loss, making them more suitable for this task.

`

- Train vs Validation plots for Accuracy and Loss for Base NN Classifier for all optimisers.



`

- **Insights**
  - **Optimizer Efficiency:** Nadam is the most effective, ideal for fast convergence and complex models.
  - **Performance Consistency:** Adam, RMSprop, and AdamW show consistency in training and validation, suggesting good generalization.
  - **Potential Overfitting:** No clear overfitting, but slight differences in performance may require further regularization adjustments.
  - **Optimizer Choice:** Depends on specific needs like sensitivity to learning rate adjustments. Nadam is recommended for its superior performance and stable dynamics, while AdamW might be explored further for scenarios prone to overfitting.

- **Classification reports for Base NN classification for all optimizers**

Classification Report for SGD optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.984791	0.988550	0.986667	262.000000	0.911111	0.872340	0.891304	47.000000
1	0.991525	0.991525	0.991525	236.000000	0.959459	0.972603	0.965986	73.000000
2	1.000000	0.996047	0.998020	253.000000	0.982143	0.982143	0.982143	56.000000
3	0.991770	0.991770	0.991770	243.000000	0.955224	0.969697	0.962406	66.000000
4	1.000000	1.000000	1.000000	242.000000	1.000000	1.000000	1.000000	67.000000
accuracy	0.993528	0.993528	0.993528	0.993528	0.964401	0.964401	0.964401	0.964401
macro avg	0.993617	0.993578	0.993596	1236.000000	0.961587	0.959357	0.960368	309.000000
weighted avg	0.993540	0.993528	0.993532	1236.000000	0.964102	0.964401	0.964165	309.000000

Classification Report for RMSprop optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.984615	0.977099	0.980843	262.0000	0.954545	0.893617	0.923077	47.000000
1	0.995763	0.995763	0.995763	236.0000	0.959459	0.972603	0.965986	73.000000
2	0.996032	0.992095	0.994059	253.0000	1.000000	0.982143	0.990991	56.000000
3	0.983673	0.991770	0.987705	243.0000	0.970149	0.984848	0.977444	66.000000
4	0.995885	1.000000	0.997938	242.0000	0.971014	1.000000	0.985294	67.000000
accuracy	0.991100	0.991100	0.991100	0.9911	0.970874	0.970874	0.970874	0.970874
macro avg	0.991194	0.991345	0.991262	1236.0000	0.971034	0.966642	0.968558	309.000000
weighted avg	0.991102	0.991100	0.991093	1236.0000	0.970848	0.970874	0.970625	309.000000

`

Classification Report for Adam optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.988462	0.980916	0.984674	262.000000	0.931818	0.872340	0.901099	47.000000
1	0.991525	0.991525	0.991525	236.000000	0.972973	0.986301	0.979592	73.000000
2	1.000000	0.996047	0.998020	253.000000	0.982143	0.982143	0.982143	56.000000
3	0.983740	0.995885	0.989775	243.000000	0.941176	0.969697	0.955224	66.000000
4	1.000000	1.000000	1.000000	242.000000	1.000000	1.000000	1.000000	67.000000
accuracy	0.992718	0.992718	0.992718	0.992718	0.967638	0.967638	0.967638	0.967638
macro avg	0.992745	0.992875	0.992799	1236.000000	0.965622	0.962096	0.963611	309.000000
weighted avg	0.992739	0.992718	0.992718	1236.000000	0.967444	0.967638	0.967335	309.000000

Classification Report for Nadam optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.988417	0.977099	0.982726	262.0000	0.953488	0.872340	0.911111	47.000000
1	0.991525	0.991525	0.991525	236.0000	0.973333	1.000000	0.986486	73.000000
2	0.996032	0.992095	0.994059	253.0000	0.982143	0.982143	0.982143	56.000000
3	0.979757	0.995885	0.987755	243.0000	0.969697	0.969697	0.969697	66.000000
4	1.000000	1.000000	1.000000	242.0000	0.971014	1.000000	0.985294	67.000000
accuracy	0.991100	0.991100	0.991100	0.9911	0.970874	0.970874	0.970874	0.970874
macro avg	0.991146	0.991321	0.991213	1236.0000	0.969935	0.964836	0.966946	309.000000
weighted avg	0.991135	0.991100	0.991097	1236.0000	0.970632	0.970874	0.970390	309.000000

Classification Report for AdamW optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.980916	0.980916	0.980916	262.000000	0.913043	0.893617	0.903226	47.000000
1	0.995708	0.983051	0.989339	236.000000	0.972222	0.958904	0.965517	73.000000
2	1.000000	0.996047	0.998020	253.000000	1.000000	0.982143	0.990991	56.000000
3	0.975709	0.991770	0.983673	243.000000	0.942029	0.984848	0.962963	66.000000
4	1.000000	1.000000	1.000000	242.000000	1.000000	1.000000	1.000000	67.000000
accuracy	0.990291	0.990291	0.990291	0.990291	0.967638	0.967638	0.967638	0.967638
macro avg	0.990467	0.990357	0.990390	1236.000000	0.965459	0.963902	0.964539	309.000000
weighted avg	0.990359	0.990291	0.990304	1236.000000	0.967829	0.967638	0.967590	309.000000

- **Observations:**
  - **High Training Metrics Across Optimizers**: All optimizers demonstrate high performance on the training data across all classes (0-4), with precision, recall, and F1-scores frequently at or near 100%. This suggests that the model fits the training data extremely well.
  - **Performance on Test Data**: The test data metrics are generally lower than the training data, which is expected due to generalization challenges, but still are quite high, showing that the models generalize well though not perfectly.

- **Specific Insights:**
  - SGD:
    - Performance: High training and test performance across all classes with particularly strong results in class 4.
    - Test F1-Scores: These are slightly lower compared to training scores, particularly in class 0 and 1 where there is a noticeable drop. This may indicate some overfitting.

`

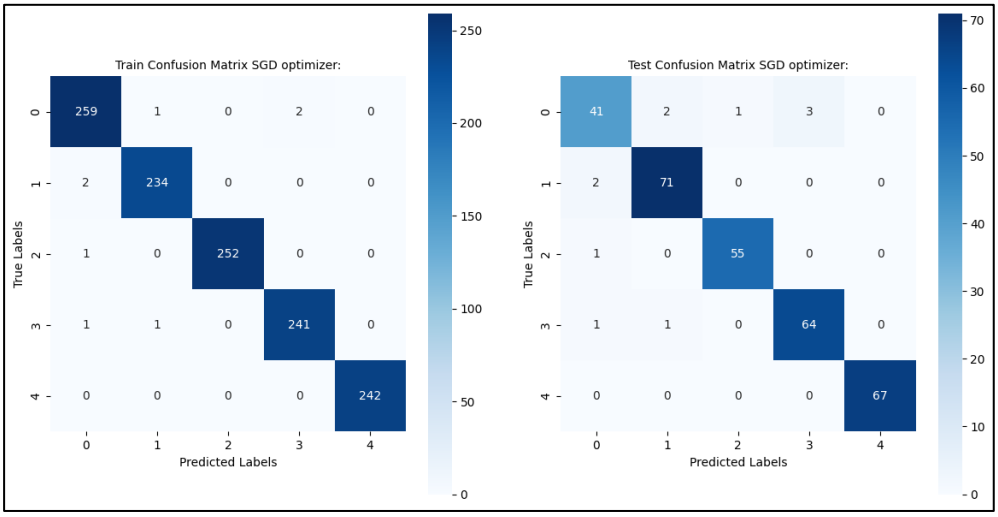
- RMSprop:
  - Test Performance: Similar to SGD, with a minor drop in precision and recall for class 1 in the test set, which suggests that this class might be a bit more challenging to generalize by this optimizer.
- Adam:
  - Consistency: Shows slightly more consistent F1-scores between training and testing than SGD, suggesting better generalization for certain classes.
  - Test Class 4: Notable for achieving 100% across all metrics, indicating exceptional performance on this class.
- Nadam:
  - Balanced Performance: Offers good balance with slightly higher test metrics in some classes compared to Adam, especially noticeable in class 0 for test precision and recall.
  - Slight Overfitting: As with others, there's a gap between train and test scores, albeit small.
- AdamW:
  - Overall Test Scores: Among the highest, suggesting that this optimizer may provide the best generalization among those tested.
  - Stability: Shows less variation between training and test metrics, particularly in class 4 where it matches or exceeds other optimizers.

- **Recommendations**:
  - Further Investigation: For classes with a significant drop between training and testing (like class 1), it might be beneficial to look into specific features or additional data that can improve model robustness.
  - Optimizer Choice: AdamW seems to provide the best generalization based on this data. Consider using it for deployment if consistent performance across multiple classes is critical.
  - Regularization and Tuning: Implement or increase regularization techniques to mitigate overfitting observed particularly in SGD and RMSprop optimizers. Also, tuning hyperparameters specifically for the underperforming classes could yield better results.

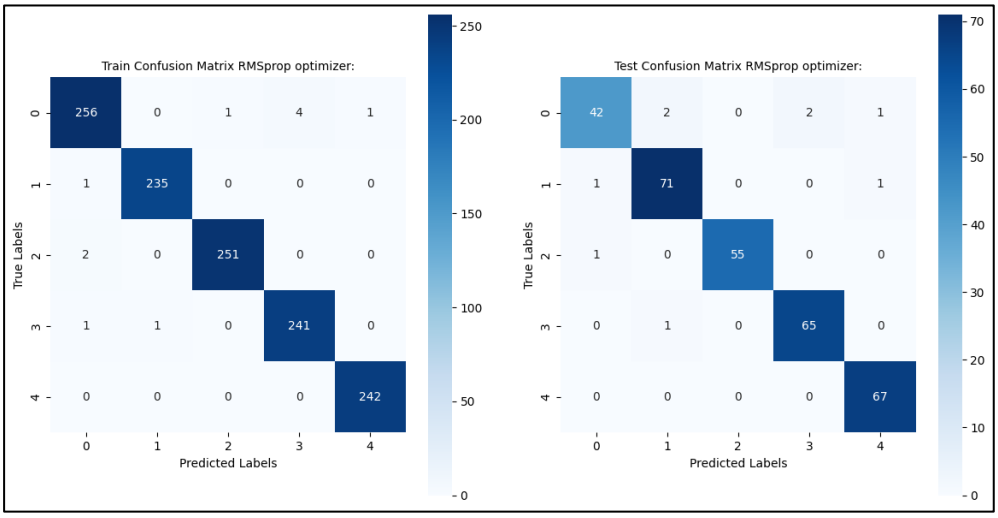
`

- **Train and Test Confusion Matrices for Base NN Classifier for all Optimizers**

Confusion Matrices for SGD Optimizer:

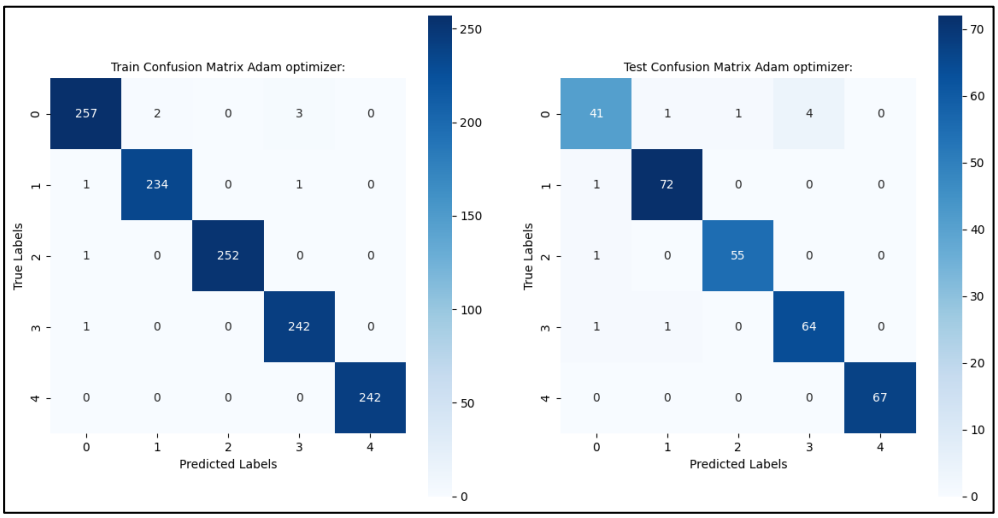


Confusion Matrices for RMSprop optimizer:

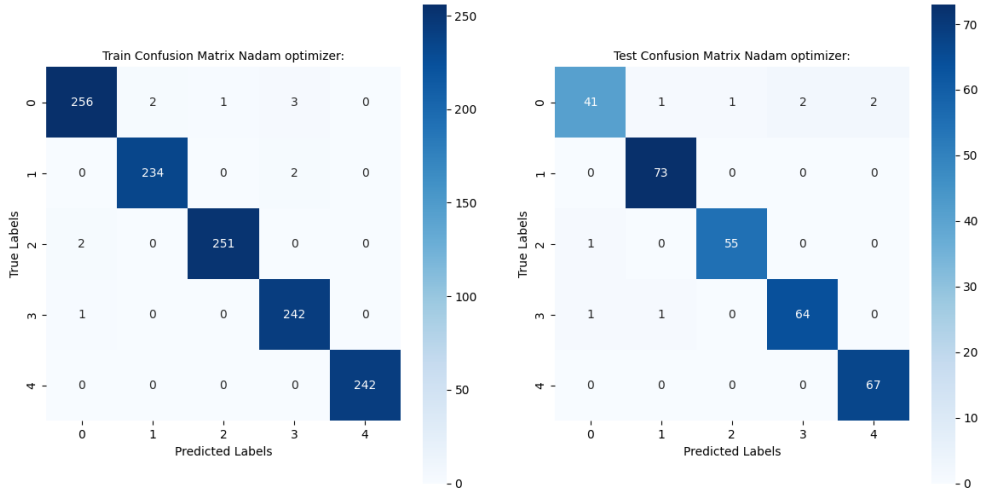


`

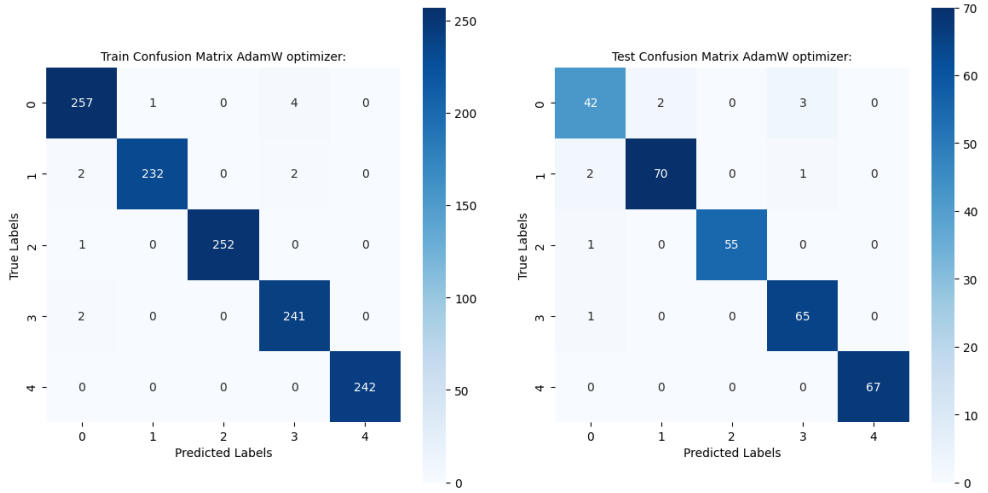
Confusion Matrices for Adam optimizer:



Confusion Matrices for Nadam optimizer:



Confusion Matrices for AdamW optimizer:



`

- Observations:
  - Training Performance:
    - Most optimizers perform well on the training data with high diagonal values, indicating good classification for each label.
    - Minor misclassifications are observed, notably a few instances of class 0 being predicted as class 3 across different optimizers.
  - Testing Performance:
    - Testing performance slightly decreases, which is typical due to the model facing unseen data.
    - The decrease in performance is not drastic, which indicates good generalization for most optimizers.

- Specific Insights:

  - SGD:
    - Misclassifications are slightly higher in testing, especially between classes 0 and 3.
    - Class 1 (true label) shows strong accuracy with 71 out of 73 correctly predicted on the test set.
  - RMSprop:
    - Shows increased misclassification between class 0 and other classes in the training set.
    - Test set performance for class 1 is consistent with the training, showing reliable performance.
  - Adam:
    - Similar patterns in both training and testing sets, with slight misclassifications mainly in classes near to each other (e.g., 1 mispredicted as 0).
    - This optimizer shows relatively balanced performance across all classes.
  - Nadam:
    - Slightly better at handling class 4 misclassifications compared to others.
    - Noticeably more misclassifications between class 0 and 3 in the test set compared to training.
  - AdamW:
    - Shows some robustness in dealing with class 3 and 4 but has slight confusion in class 2 predictions.
    - Test results are similar to training, indicating good consistency.

`

- **Conclusion**:
  - Specific Class Performance:
    - For Class 1, optimizers like SGD, Adam, and Nadam consistently show high accuracy on the testing dataset.
  - Misclassification Patterns:
    - Certain patterns like misclassifications between classes 0 and 3 are more prevalent in some optimizers (Nadam, Adam) than others.
  - Balancing Decision:
    - Consistency between Training and Testing: AdamW shows good consistency between training and testing.
    - Overall Accuracy and Stability: Adam and Nadam also display strong and stable performance, but AdamW seems slightly better in terms of maintaining performance from training to testing.

3.3.4 Hypertuned NN classifier

- **Builds an Improved Neural Network Model**: Defines a Sequential neural network with multiple dense layers, batch normalization, and dropout for regularization.
  - **Model Architecture**:
    - Four hidden layers with 256, 128, 64, and 32 neurons respectively, each using ReLU activation.
    - Each dense layer is followed by Batch Normalization and Dropout (0.2) to improve training stability and prevent overfitting.
    - L2 regularization is applied to each layer to further control overfitting.
    - Final layer uses Softmax activation for multi-class classification.
  - Optimizers Dictionary: Provides a set of five optimizers (SGD, RMSprop, Adam, Nadam, and AdamW) to be used for model training.
  - Optimizer Validation: Ensures the selected optimizer is from the predefined list, otherwise raises an error.
  - Model Compilation: Compiles the model using the chosen optimizer, categorical_crossentropy loss function, and accuracy metric.
  - Model Initialization: Initializes and stores models in a dictionary (ht_nn_models), each trained with a different optimizer from the list.
  - Purpose: Prepares hypertuned neural network models with various optimizers for comparison in performance.

`

- Model Summaries for all the optimizers

Hypertuned NN Model with SGD optimizer:
Model: "sequential_9"

Layer (type)	Output Shape	Param #
dense_33 (Dense)	(None, 256)	91,392
batch_normalization_12 (BatchNormalization)	(None, 256)	1,024
dropout_12 (Dropout)	(None, 256)	0
dense_34 (Dense)	(None, 128)	32,896
batch_normalization_13 (BatchNormalization)	(None, 128)	512
dropout_13 (Dropout)	(None, 128)	0
dense_35 (Dense)	(None, 64)	8,256
batch_normalization_14 (BatchNormalization)	(None, 64)	256
dropout_14 (Dropout)	(None, 64)	0
dense_36 (Dense)	(None, 32)	2,080
batch_normalization_15 (BatchNormalization)	(None, 32)	128
dropout_15 (Dropout)	(None, 32)	0
dense_37 (Dense)	(None, 5)	165

Total params: 136,709 (534.02 KB)
Trainable params: 135,749 (530.27 KB)
Non-trainable params: 960 (3.75 KB)

Hypertuned NN Model with RMSprop optimizer:
Model: "sequential_10"

Layer (type)	Output Shape	Param #
dense_38 (Dense)	(None, 256)	91,392
batch_normalization_16 (BatchNormalization)	(None, 256)	1,024
dropout_16 (Dropout)	(None, 256)	0
dense_39 (Dense)	(None, 128)	32,896
batch_normalization_17 (BatchNormalization)	(None, 128)	512
dropout_17 (Dropout)	(None, 128)	0
dense_40 (Dense)	(None, 64)	8,256
batch_normalization_18 (BatchNormalization)	(None, 64)	256
dropout_18 (Dropout)	(None, 64)	0
dense_41 (Dense)	(None, 32)	2,080
batch_normalization_19 (BatchNormalization)	(None, 32)	128
dropout_19 (Dropout)	(None, 32)	0
dense_42 (Dense)	(None, 5)	165

Total params: 136,709 (534.02 KB)
Trainable params: 135,749 (530.27 KB)
Non-trainable params: 960 (3.75 KB)

`

Hypertuned NN Model with Adam optimizer:
Model: "sequential_11"

Layer (type)	Output Shape	Param #
dense_43 (Dense)	(None, 256)	91,392
batch_normalization_20 (BatchNormalization)	(None, 256)	1,024
dropout_20 (Dropout)	(None, 256)	0
dense_44 (Dense)	(None, 128)	32,896
batch_normalization_21 (BatchNormalization)	(None, 128)	512
dropout_21 (Dropout)	(None, 128)	0
dense_45 (Dense)	(None, 64)	8,256
batch_normalization_22 (BatchNormalization)	(None, 64)	256
dropout_22 (Dropout)	(None, 64)	0
dense_46 (Dense)	(None, 32)	2,080
batch_normalization_23 (BatchNormalization)	(None, 32)	128
dropout_23 (Dropout)	(None, 32)	0
dense_47 (Dense)	(None, 5)	165

Total params: 136,709 (534.02 KB)
Trainable params: 135,749 (530.27 KB)
Non-trainable params: 960 (3.75 KB)

Hypertuned NN Model with Nadam optimizer:
Model: "sequential_12"

Layer (type)	Output Shape	Param #
dense_48 (Dense)	(None, 256)	91,392
batch_normalization_24 (BatchNormalization)	(None, 256)	1,024
dropout_24 (Dropout)	(None, 256)	0
dense_49 (Dense)	(None, 128)	32,896
batch_normalization_25 (BatchNormalization)	(None, 128)	512
dropout_25 (Dropout)	(None, 128)	0
dense_50 (Dense)	(None, 64)	8,256
batch_normalization_26 (BatchNormalization)	(None, 64)	256
dropout_26 (Dropout)	(None, 64)	0
dense_51 (Dense)	(None, 32)	2,080
batch_normalization_27 (BatchNormalization)	(None, 32)	128
dropout_27 (Dropout)	(None, 32)	0
dense_52 (Dense)	(None, 5)	165

Total params: 136,709 (534.02 KB)
Trainable params: 135,749 (530.27 KB)
Non-trainable params: 960 (3.75 KB)

`

Hypertuned NN Model with AdamW optimizer:
Model: "sequential_13"

Layer (type)	Output Shape	Param #
dense_53 (Dense)	(None, 256)	91,392
batch_normalization_28 (BatchNormalization)	(None, 256)	1,024
dropout_28 (Dropout)	(None, 256)	0
dense_54 (Dense)	(None, 128)	32,896
batch_normalization_29 (BatchNormalization)	(None, 128)	512
dropout_29 (Dropout)	(None, 128)	0
dense_55 (Dense)	(None, 64)	8,256
batch_normalization_30 (BatchNormalization)	(None, 64)	256
dropout_30 (Dropout)	(None, 64)	0
dense_56 (Dense)	(None, 32)	2,080
batch_normalization_31 (BatchNormalization)	(None, 32)	128
dropout_31 (Dropout)	(None, 32)	0
dense_57 (Dense)	(None, 5)	165

Total params: 136,709 (534.02 KB)
Trainable params: 135,749 (530.27 KB)
Non-trainable params: 960 (3.75 KB)

- **Training and evaluation of Hypertuned NN Classifier model with cross-validation results**

Training Hypertuned NN Classifier model with SGD...
Training fold 1...
Fold 1 training complete.
Training fold 2...
Fold 2 training complete.
Training fold 3...
Fold 3 training complete.
Training fold 4...
Fold 4 training complete.
Training fold 5...
Fold 5 training complete.
Test Loss (Hypertuned - SGD): 0.5124
Test Accuracy (Hypertuned - SGD): 0.9709
Fold 1: Best Validation Loss: 0.6942, Best Validation Accuracy: 0.9637, Early Stopping Epoch: 86
Fold 2: Best Validation Loss: 0.5066, Best Validation Accuracy: 0.9960, Early Stopping Epoch: 99
Fold 3: Best Validation Loss: 0.4276, Best Validation Accuracy: 1.0000, Early Stopping Epoch: 99
Fold 4: Best Validation Loss: 0.4276, Best Validation Accuracy: 0.9960, Early Stopping Epoch: 9
Fold 5: Best Validation Loss: 0.4331, Best Validation Accuracy: 0.9960, Early Stopping Epoch: 9
Training Hypertuned NN Classifier model with RMSprop...
Training fold 1...
Fold 1 training complete.
Training fold 2...
Fold 2 training complete.
Training fold 3...
Fold 3 training complete.
Training fold 4...
Fold 4 training complete.
Training fold 5...
Fold 5 training complete.
Test Loss (Hypertuned - RMSprop): 0.7149

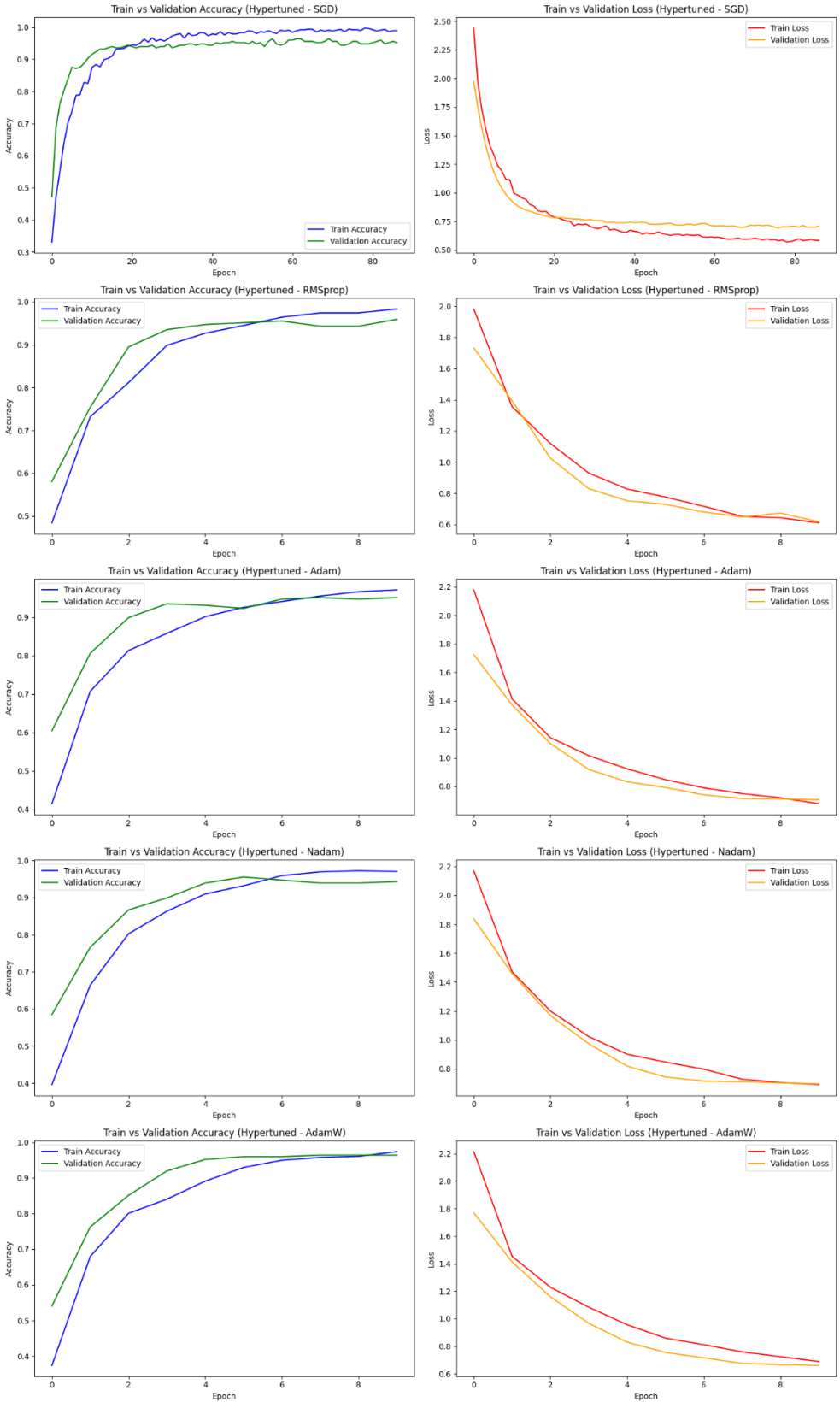
`

Test Accuracy (Hypertuned - RMSprop): 0.9515
Fold 1: Best Validation Loss: 0.6172, Best Validation Accuracy: 0.9597, Early Stopping Epoch: 9
Fold 2: Best Validation Loss: 0.6921, Best Validation Accuracy: 0.9595, Early Stopping Epoch: 9
Fold 3: Best Validation Loss: 0.5820, Best Validation Accuracy: 0.9717, Early Stopping Epoch: 9
Fold 4: Best Validation Loss: 0.5693, Best Validation Accuracy: 0.9960, Early Stopping Epoch: 9
Fold 5: Best Validation Loss: 0.5897, Best Validation Accuracy: 0.9838, Early Stopping Epoch: 9
Training Hypertuned NN Classifier model with Adam...
Training fold 1...
Fold 1 training complete.
Training fold 2...
Fold 2 training complete.
Training fold 3...
Fold 3 training complete.
Training fold 4...
Fold 4 training complete.
Training fold 5...
Fold 5 training complete.
Test Loss (Hypertuned - Adam): 0.7499
Test Accuracy (Hypertuned - Adam): 0.9644
Fold 1: Best Validation Loss: 0.7068, Best Validation Accuracy: 0.9516, Early Stopping Epoch: 9
Fold 2: Best Validation Loss: 0.7131, Best Validation Accuracy: 0.9555, Early Stopping Epoch: 9
Fold 3: Best Validation Loss: 0.6594, Best Validation Accuracy: 0.9676, Early Stopping Epoch: 9
Fold 4: Best Validation Loss: 0.5731, Best Validation Accuracy: 0.9879, Early Stopping Epoch: 9
Fold 5: Best Validation Loss: 0.6100, Best Validation Accuracy: 0.9838, Early Stopping Epoch: 9
Training Hypertuned NN Classifier model with Nadam...
Training fold 1...
Fold 1 training complete.
Training fold 2...
Fold 2 training complete.
Training fold 3...
Fold 3 training complete.
Training fold 4...
Fold 4 training complete.
Training fold 5...
Fold 5 training complete.
Test Loss (Hypertuned - Nadam): 0.7520
Test Accuracy (Hypertuned - Nadam): 0.9482
Fold 1: Best Validation Loss: 0.6958, Best Validation Accuracy: 0.9556, Early Stopping Epoch: 9
Fold 2: Best Validation Loss: 0.7160, Best Validation Accuracy: 0.9636, Early Stopping Epoch: 9
Fold 3: Best Validation Loss: 0.6202, Best Validation Accuracy: 0.9757, Early Stopping Epoch: 9
Fold 4: Best Validation Loss: 0.5922, Best Validation Accuracy: 0.9838, Early Stopping Epoch: 9
Fold 5: Best Validation Loss: 0.5840, Best Validation Accuracy: 0.9879, Early Stopping Epoch: 9
Training Hypertuned NN Classifier model with AdamW...
Training fold 1...
Fold 1 training complete.
Training fold 2...
Fold 2 training complete.
Training fold 3...
Fold 3 training complete.
Training fold 4...
Fold 4 training complete.
Training fold 5...
Fold 5 training complete.
Test Loss (Hypertuned - AdamW): 0.7386
Test Accuracy (Hypertuned - AdamW): 0.9579
Fold 1: Best Validation Loss: 0.6598, Best Validation Accuracy: 0.9637, Early Stopping Epoch: 9
Fold 2: Best Validation Loss: 0.7282, Best Validation Accuracy: 0.9555, Early Stopping Epoch: 9
Fold 3: Best Validation Loss: 0.6896, Best Validation Accuracy: 0.9636, Early Stopping Epoch: 9
Fold 4: Best Validation Loss: 0.6146, Best Validation Accuracy: 0.9879, Early Stopping Epoch: 9
Fold 5: Best Validation Loss: 0.5902, Best Validation Accuracy: 0.9879, Early Stopping Epoch: 9
Training and evaluation of Hypertuned NN Classifier model with cross-validation complete.

	Avg_Train_Accuracy	Avg_Val_Accuracy	Avg_Train_Loss	Avg_Val_Loss
SGD	97.714758	97.846592	0.580814	0.578634
RMSprop	93.752672	94.729822	0.757869	0.729192
Adam	92.714778	94.745755	0.805752	0.761932
Nadam	92.844166	94.941033	0.808208	0.755628
AdamW	92.356631	94.276642	0.822367	0.769262

`

- **Train vs Validation plots for Accuracy and Loss for Hypertuned NN Classifier for all optimisers.**



`

- **Observations:**
    1. **SGD (Stochastic Gradient Descent)**
        - **Accuracy:** The training and validation accuracy curves converge closely, indicating good generalization.
        - **Loss:** Both training and validation loss decrease sharply and stabilize quickly, showing that SGD is effective and efficient in optimizing the loss function.
    2. **RMSprop**
        - **Accuracy:** There's a noticeable gap between training and validation accuracy, suggesting some overfitting. However, the validation accuracy does improve steadily, which is a good sign.
        - **Loss:** The training and validation loss curves decrease quickly. The small gap between them suggests some level of overfitting, though less severe compared to the other Adam-based optimizers.
    3. **Adam**
        - **Accuracy:** The accuracy curves for Adam show that while there's an improvement in validation accuracy, the gap between training and validation accuracy is significant, indicating overfitting.
        - **Loss:** The loss curves converge well initially but start to diverge slightly, which again points to overfitting as training progresses.
    4. **Nadam**
        - **Accuracy:** Similar to Adam, Nadam shows a gap between the training and validation accuracy curves, indicative of overfitting.
        - **Loss:** The loss curves show less divergence compared to Adam, suggesting a slightly better handling of overfitting.
    5. **AdamW**
        - **Accuracy:** The gap between training and validation accuracy is somewhat large, which suggests overfitting. The improvement in validation accuracy is slower and less stable compared to the other optimizers.
        - **Loss:** Similar to the accuracy results, the loss curves show a significant gap, indicating that AdamW might not be as effective in this case.

- **Insights**

    1. **SGD not only achieves the highest accuracies (97.71% training, 97.85% validation)** but also maintains the lowest losses (0.581 training, 0.579 validation), confirming its superiority in both learning and generalizing from the training data.
    2. **RMSprop, Adam, and Nadam** show a clear gap between training and validation metrics, indicating some degree of overfitting, though RMSprop and Nadam manage slightly better generalization than Adam.
    3. **AdamW** exhibits the largest gap, suggesting significant overfitting and the need for adjustments in model training strategy or hyperparameter settings.

`

- Classification reports for Hypertuned NN Classifier

Classification Report for Hypertuned Model with SGD optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	1.000000	0.996183	0.998088	262.000000	0.880000	0.936170	0.907216	47.000000
1	0.995781	1.000000	0.997886	236.000000	0.986111	0.972603	0.979310	73.000000
2	1.000000	1.000000	1.000000	253.000000	1.000000	0.982143	0.990991	56.000000
3	0.995885	0.995885	0.995885	243.000000	0.969231	0.954545	0.961832	66.000000
4	1.000000	1.000000	1.000000	242.000000	1.000000	1.000000	1.000000	67.000000
accuracy	0.998382	0.998382	0.998382	0.998382	0.970874	0.970874	0.970874	0.970874
macro avg	0.998333	0.998414	0.998372	1236.000000	0.967068	0.969092	0.967870	309.000000
weighted avg	0.998385	0.998382	0.998382	1236.000000	0.971894	0.970874	0.971214	309.000000

Classification Report for Hypertuned Model with RMSprop optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.970260	0.996183	0.983051	262.000000	0.800000	0.936170	0.862745	47.000000
1	0.995708	0.983051	0.989339	236.000000	0.984848	0.890411	0.935252	73.000000
2	1.000000	0.988142	0.994036	253.000000	0.964286	0.964286	0.964286	56.000000
3	0.995868	0.991770	0.993814	243.000000	1.000000	0.969697	0.984615	66.000000
4	1.000000	1.000000	1.000000	242.000000	0.985294	1.000000	0.992593	67.000000
accuracy	0.991909	0.991909	0.991909	0.991909	0.951456	0.951456	0.951456	0.951456
macro avg	0.992367	0.991829	0.992048	1236.000000	0.946886	0.952113	0.947898	309.000000
weighted avg	0.992064	0.991909	0.991935	1236.000000	0.956339	0.951456	0.952462	309.000000

Classification Report for Hypertuned Model with Adam optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.966292	0.984733	0.975425	262.000000	0.849057	0.957447	0.900000	47.000000
1	1.000000	0.966102	0.982759	236.000000	1.000000	0.931507	0.964539	73.000000
2	0.992063	0.988142	0.990099	253.000000	1.000000	1.000000	1.000000	56.000000
3	0.971660	0.987654	0.979592	243.000000	0.953846	0.939394	0.946565	66.000000
4	1.000000	1.000000	1.000000	242.000000	1.000000	1.000000	1.000000	67.000000
accuracy	0.985437	0.985437	0.985437	0.985437	0.964401	0.964401	0.964401	0.964401
macro avg	0.986003	0.985326	0.985575	1236.000000	0.960581	0.965670	0.962221	309.000000
weighted avg	0.985659	0.985437	0.985460	1236.000000	0.967183	0.964401	0.964999	309.000000

Classification Report for Hypertuned Model with Nadam optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.977358	0.988550	0.982922	262.0000	0.820000	0.872340	0.845361	47.00000
1	0.983193	0.991525	0.987342	236.0000	0.957746	0.931507	0.944444	73.00000
2	1.000000	0.988142	0.994036	253.0000	0.963636	0.946429	0.954955	56.00000
3	0.995851	0.987654	0.991736	243.0000	0.969697	0.969697	0.969697	66.00000
4	1.000000	1.000000	1.000000	242.0000	1.000000	1.000000	1.000000	67.00000
accuracy	0.991100	0.991100	0.991100	0.9911	0.948220	0.948220	0.948220	0.94822
macro avg	0.991280	0.991174	0.991207	1236.0000	0.942216	0.943995	0.942891	309.00000
weighted avg	0.991176	0.991100	0.991117	1236.0000	0.949576	0.948220	0.948718	309.00000

Classification Report for Hypertuned Model with AdamW optimizer:

	Train_precision	Train_recall	Train_f1-score	Train_support	Test_precision	Test_recall	Test_f1-score	Test_support
0	0.952030	0.984733	0.968105	262.000000	0.821429	0.978723	0.893204	47.000000
1	0.974468	0.970339	0.972399	236.000000	0.985294	0.917808	0.950355	73.000000
2	1.000000	0.988142	0.994036	253.000000	0.964286	0.964286	0.964286	56.000000
3	0.995798	0.975309	0.985447	243.000000	1.000000	0.939394	0.968750	66.000000
4	1.000000	1.000000	1.000000	242.000000	1.000000	1.000000	1.000000	67.000000
accuracy	0.983819	0.983819	0.983819	0.983819	0.957929	0.957929	0.957929	0.957929
macro avg	0.984459	0.983705	0.983997	1236.000000	0.954202	0.960042	0.955319	309.000000
weighted avg	0.984130	0.983819	0.983887	1236.000000	0.962892	0.957929	0.958880	309.000000

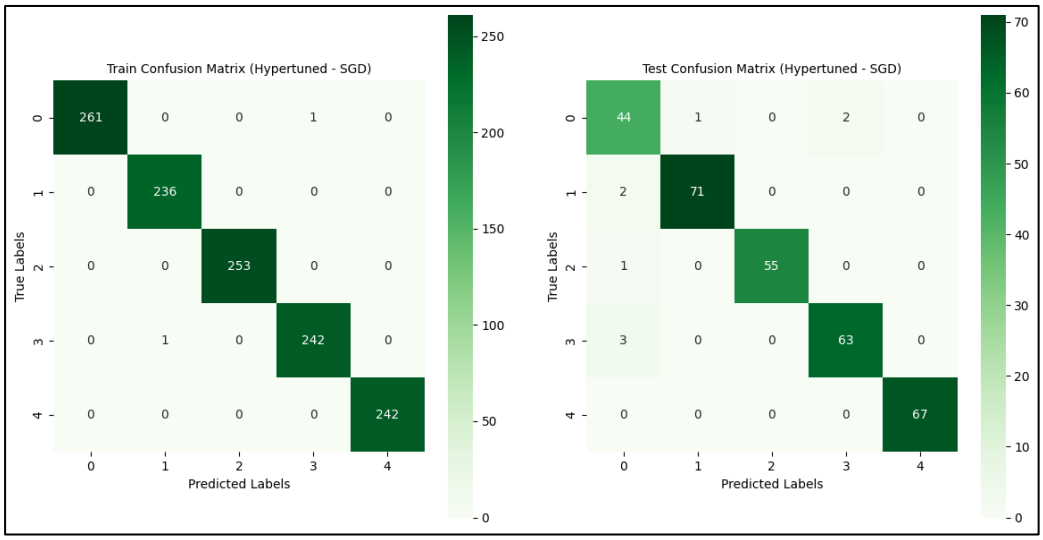
`

- **Observations:**
    - **High Training Performance Across All Models:** All models demonstrate high precision, recall, and F1-scores on the training data across all categories, often reaching or exceeding 99%, which indicates a strong fit to the training data.
    - **Testing Performance Variation:** Testing metrics show more variation, with precision and recall values ranging from about 82% to 100% depending on the optimizer and category. This variation indicates differences in how well each optimizer generalizes.
    - **Consistency in Certain Categories:** Some categories, like class 1 (100% precision, recall, and F1-score across almost all models during testing) and category 4 (100% precision and recall in most models), consistently show high performance, suggesting distinct and well-represented features in the training set.
- **Specific:**
    - **SGD Optimizer:**
        - Exhibits strong testing recall for most categories (e.g., 97.26% for category 1, 98.45% for category 3), but has lower precision in category 0 (88.00%) and category 3 (90.35%), which could indicate some overfitting issues.
    - **RMSprop:**
        - Shows a notable drop in testing precision for category 0 (80.00%), suggesting potential overfitting or lack of generalization for this category with this optimizer.
    - **Adam:**
        - Provides a balanced performance with high precision (e.g., 100% for category 1) and recall (e.g., 93.93% for category 3) in the testing phase, making it a strong candidate for a robust model.
    - **Nadam:**
        - Similarly to Adam, it demonstrates high precision (e.g., 96.36% for category 2) and recall (e.g., 94.64% for category 2) in the testing phase but with slight edges in certain categories.
    - **AdamW:**
        - Delivers strong testing performance but with slightly lesser precision in category 0 (82.14%) compared to Nadam and Adam. This might suggest some room for adjustment in learning rates or decay parameters.
- **Recommendations:**
    - **Model Selection:** Based on testing performance, Adam and Nadam might be preferred for their balanced and high performance across different categories.
    - **Further Tuning:** For optimizers showing weaker performance in specific categories, consider further hyperparameter tuning or even ensemble methods to leverage strengths of multiple optimizers.

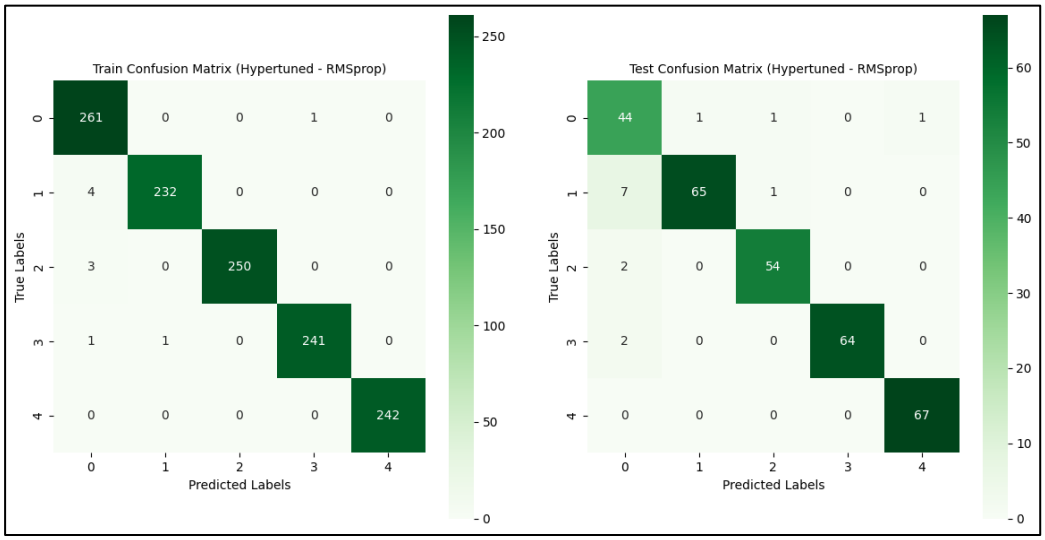
`

- Train and Test Confusion Matrices for Hypertuned NN classifier for all Optimizers

Confusion Matrices for Hypertuned NN with SGD optimizer:

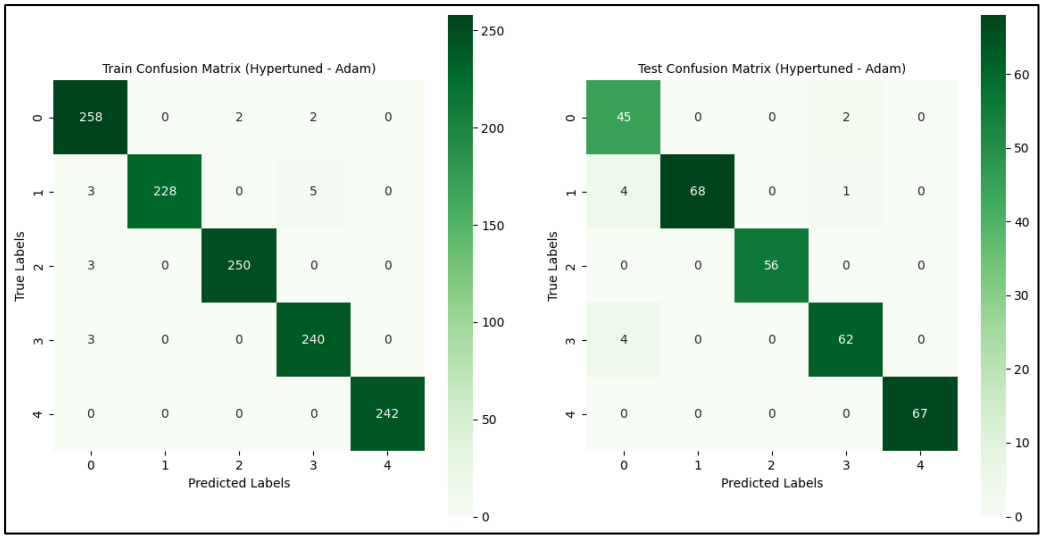


Confusion Matrices for Hypertuned NN with RMSprop optimizer:

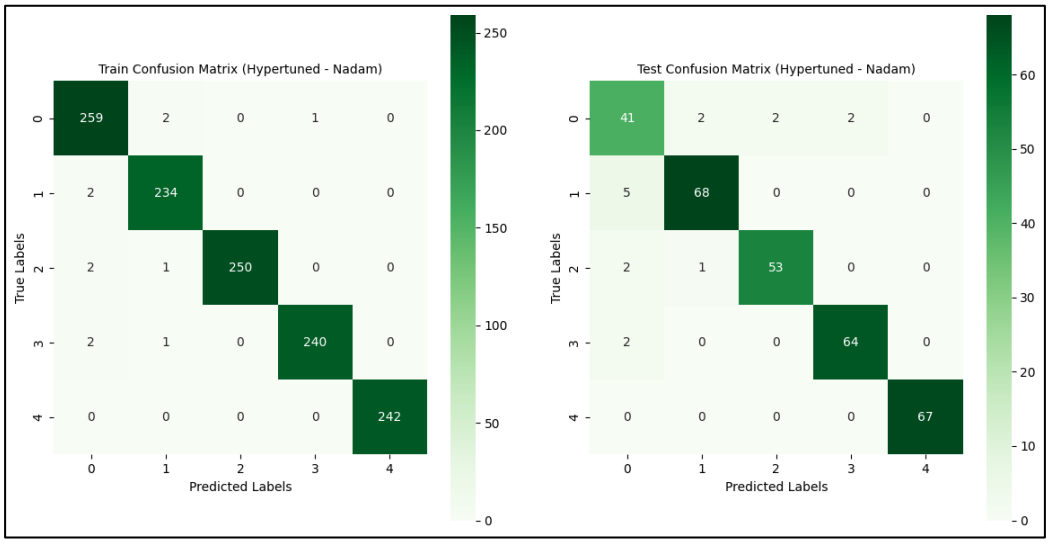


`

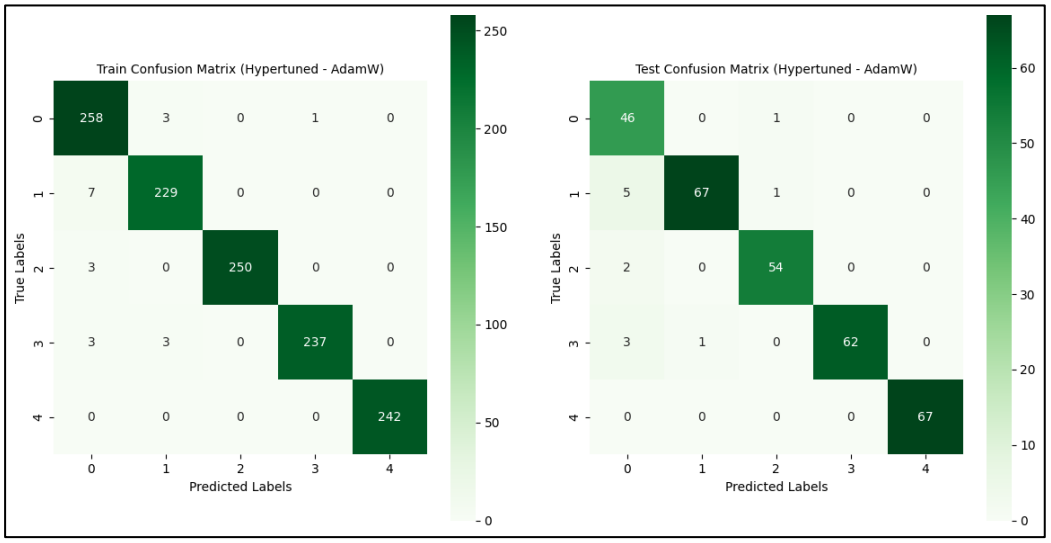
Confusion Matrices for Hypertuned NN with Adam optimizer:



Confusion Matrices for Hypertuned NN with Nadam optimizer:



Confusion Matrices for Hypertuned NN with AdamW optimizer:



`

- **Observations:**
  - **Accuracy Across Optimizers:**
    - **High Accuracy Levels:**
      - All optimizers demonstrate high accuracy on the diagonal (true positives), showing their capability to effectively learn and predict the correct class labels.
      - AdamW and Nadam show high true positive rates across nearly all classes, especially noticeable in hypertuned states.
    - **Common Misclassification Patterns:**
      - Misclassifications commonly occur between specific classes (notably classes 0, 2, and 3) across most optimizers, indicating challenges inherent in the data features or similarities between these classes that are not optimizer-specific.
  - **Generalization and Overfitting:**
    - **Generalization Issues:**
      - A noticeable performance drop from training to testing datasets highlights generalization challenges. For instance, Adam often showed better training performance that did not always translate equally to the testing scenarios.
    - **Impact of Hyperparameter Tuning:**
      - Hyperparameter tuning tends to improve test set accuracy, as seen with RMSprop and Nadam, suggesting that tuning helps in enhancing model generalization across unseen data.
  - **Consistency and Stability:**
    - **Training vs. Testing Discrepancy:**
      - Most optimizers, including SGD and Adam, exhibit some consistency issues, with models performing exceptionally well in training but less so in testing, indicating potential overfitting. Tuning typically reduces this gap.
    - **Stability Across Classes:**
      - While tuning generally enhances stability, it sometimes exacerbates misclassifications in certain classes, such as increased errors for class 1 in the hypertuned AdamW model.
  - **Optimizer-Specific Findings:**
    - **SGD:**
      - Shows the necessity of careful hyperparameter adjustments to prevent overfitting, as minimal tuning leads to modest improvements.
    - **RMSprop and Nadam:**
      - These optimizers benefit significantly from hyperparameter tuning, particularly in adjusting momentum terms that help in navigating the error landscape more effectively.
    - **Adam and AdamW:**

`

    - Demonstrates flexibility and robust initial performance, with AdamW slightly better at managing long-term stability thanks to effective handling of weight decay.

- **Recommendations:**
    - **Enhanced Parameter Tuning:**
        - Explore more adaptive learning rate schedules and advanced regularization techniques to address specific misclassification issues, such as dynamic learning rate adjustments based on validation loss feedback.
    - **Cross-validation Techniques:**
        - Employ robust cross-validation to ensure the optimizer's performance is reliable across various subsets of data, enhancing the model's reliability and predictability.
    - **Detailed Feature Analysis:**
        - Analyze features leading to high misclassification rates to refine model inputs, potentially incorporating dimensionality reduction techniques or feature selection algorithms to enhance class separability.
    - **Advanced Optimization Techniques:**
        - Experiment with less common optimizers or combinations of multiple optimizers (ensemble methods) to potentially leverage the strengths of various optimization algorithms.

3.3.5 Design, Train and Test LSTM

To design, train, and test an LSTM classifier, the process starts by importing and preparing the dataset, including scaling the features and encoding the target labels. The LSTM model architecture is then created with two LSTM layers and dropout layers to prevent overfitting. The model uses a Softmax output layer for multi-class classification. After compiling the model with a loss function like categorical cross-entropy and an optimizer such as Adam, it is trained using early stopping to prevent overtraining. Finally, the model is evaluated on a test set, and predictions are made to assess its performance.

3.3.6 Data Preparation

- The dataset (ISH_NLP_Glove_df_main.csv) is loaded, and features (input data) and target (accident level) are separated.
- The data is split into training and testing sets (80% training, 20% testing).
- Features are scaled using StandardScaler, and the target labels are encoded into numerical format and converted to categorical values (for multi-class classification)
- The features are reshaped to fit the LSTM model's expected input format: (samples, time steps, features). Here, a single time step is assumed for simplicity.

`

3.3.7 Design, Train and Test LSTM

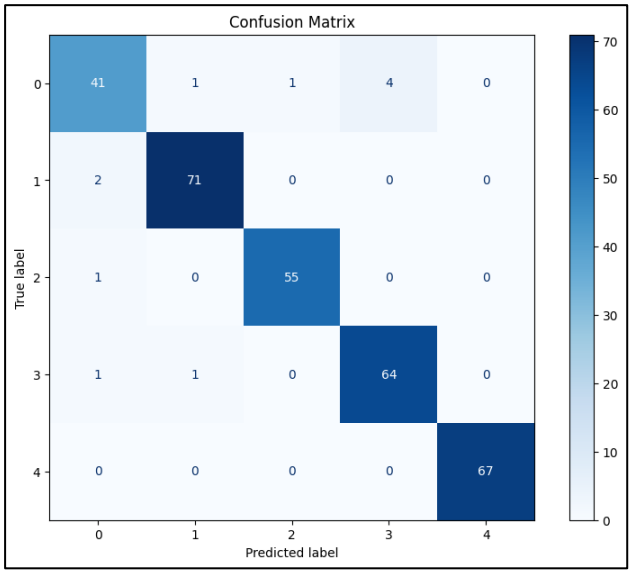
- A Sequential model is defined with two LSTM layers (64 and 32 units, respectively), each followed by Dropout layers (to reduce overfitting).
- The output layer uses a softmax activation function for multi-class classification, predicting the accident severity level.
- **Model Compilation**: The model is compiled using the Adam optimizer, categorical_crossentropy as the loss function (for classification), and accuracy as the performance metric.
- **Early Stopping** is set up to monitor validation loss, stopping training if no improvement is seen after 5 epochs and restoring the best model weights.
- The model is trained for up to 50 epochs with a batch size of 64, using the training data and validating against the test data during training.
- **Model Results:** Test Loss – 0.0951. Test Accuracy – 96.44%
- **Prediction Comparison**:
    - The LSTM model achieved an accuracy of 96.44% on the test set.
    - Predicted Labels and True Labels for the test set were compared. The model performed well in predicting the accident severity levels, with only a few mismatches between predicted and true labels, as evidenced by the provided lists.

```
Test Loss: 0.09511867165565491
Test Accuracy: 0.9644013047218323
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
Predicted labels: [4 1 4 1 3 2 4 4 3 3 1 4 4 0 0 3 3 2 1 2 1 1 1 0 1 1 3 2 2 1 3 4 2 4 0 2 4
 1 4 1 1 2 0 1 4 0 2 4 3 2 1 1 3 3 0 1 1 1 3 3 2 2 1 3 0 1 0 2 4 2 0 0 1 4
 0 3 3 3 4 3 3 0 3 2 0 4 2 1 2 3 4 3 3 4 4 4 1 4 3 2 2 3 4 1 4 4 2 4 2 3 1 1
 1 3 3 1 1 1 1 3 4 0 1 0 4 2 4 4 2 3 1 1 1 3 1 0 4 3 3 3 3 4 2 3 0 0 0 2 4
 2 3 4 0 1 3 2 0 3 4 1 3 0 4 1 2 4 1 0 4 2 3 3 3 1 4 1 0 3 4 4 2 3 2 3 4 3
 1 2 1 0 3 3 3 4 2 3 4 1 2 0 2 1 1 0 0 3 3 1 1 3 2 2 0 0 2 0 2 4 1 4 1 4 0
 3 4 3 2 4 1 3 1 1 0 0 4 2 2 3 4 1 0 1 4 4 2 4 1 4 0 3 1 0 2 1 3 4 0 2 3 2
 2 1 1 1 0 4 4 4 1 3 4 3 1 1 3 1 2 2 1 3 3 1 4 0 1 3 0 3 2 4 1 4 4 4 2 3 0
 0 1 4 0 2 4 1 2 3 2 4 2 2]
True labels: [4 1 4 1 3 2 4 4 3 3 1 4 4 0 0 3 0 2 1 2 1 1 1 0 1 1 3 2 2 1 3 4 2 4 0 2 4
 1 4 1 1 2 0 1 4 0 2 4 3 2 1 1 3 3 0 1 1 1 3 3 2 2 1 0 0 1 0 2 4 2 2 0 1 4
 0 3 3 3 4 3 3 0 3 0 0 4 2 1 2 3 4 3 3 4 4 4 1 4 3 2 2 0 4 1 4 4 2 4 2 3 1 1
 1 3 3 1 1 1 1 3 4 0 1 0 4 2 4 4 2 3 3 1 1 3 1 3 4 3 3 3 3 4 2 3 0 0 0 2 4
 2 3 4 0 1 3 2 0 3 4 1 3 0 4 1 2 4 1 0 4 2 3 3 3 1 4 1 0 3 4 4 2 3 2 3 4 3
 1 2 1 0 3 3 3 4 2 0 4 1 2 1 2 1 1 0 0 3 3 1 0 3 2 2 0 0 2 0 2 4 1 4 1 4 0
 3 4 3 2 4 1 3 1 1 0 0 4 2 2 3 4 1 0 1 4 4 2 4 1 4 1 3 1 0 2 1 3 4 0 2 3 2
 2 1 1 1 0 4 4 4 1 3 4 3 1 1 3 1 2 2 1 3 3 1 4 0 1 3 0 3 2 4 1 4 4 4 2 3 0
 0 1 4 0 2 4 1 2 3 2 4 2 2]
```

- **Key Observations**:
    - The LSTM model shows strong predictive performance, with a low loss and high accuracy, indicating it effectively classifies accident severity levels.
    - The model can reliably predict the labels with minimal deviations from the actual true labels, which aligns with the high accuracy score.

`

- **Confusion Matrix**



  - Key Takeaways:
    - Class 4 was predicted perfectly with no errors.
    - The model shows strong performance across all classes with very few misclassifications.
    - Most errors involve confusion between neighbouring classes, particularly between Class 0 and other classes.

- **Classification Report: Training**

Classification Report for Training Set:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	262
1	1.00	1.00	1.00	236
2	1.00	1.00	1.00	253
3	1.00	1.00	1.00	243
4	1.00	1.00	1.00	242
accuracy			1.00	1236
macro avg	1.00	1.00	1.00	1236
weighted avg	1.00	1.00	1.00	1236

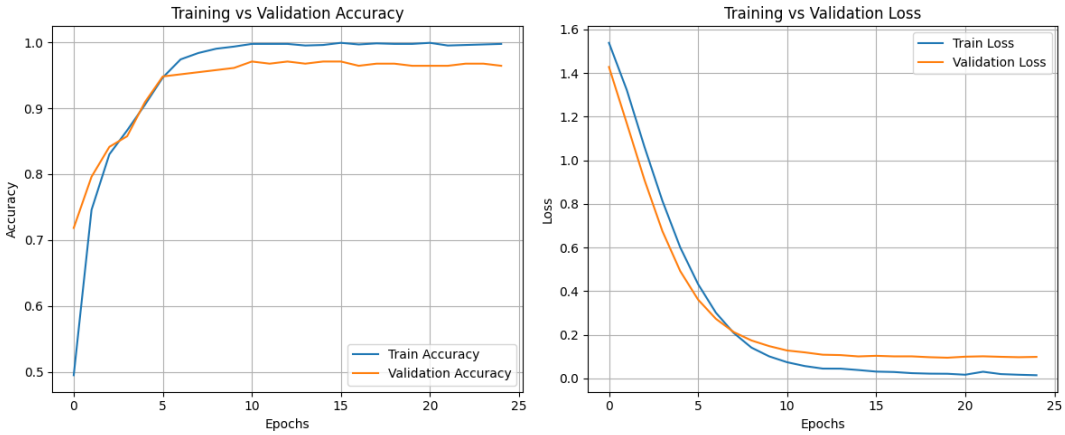
The model has achieved perfect classification on the training set, indicating strong learning but potentially suggesting overfitting, where the model may not generalize as well to unseen data despite excellent performance in training.

- **Classification Report: Test**

```
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9618 - loss: 0.0862
Test Loss: 0.09511867165565491
Test Accuracy: 0.9644013047218323
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
              precision    recall  f1-score   support

           0       0.91      0.87      0.89        47
           1       0.97      0.97      0.97        73
           2       0.98      0.98      0.98        56
           3       0.94      0.97      0.96        66
           4       1.00      1.00      1.00        67

    accuracy                           0.96       309
   macro avg       0.96      0.96      0.96       309
weighted avg       0.96      0.96      0.96       309
```

  - The model performs exceptionally well across all classes, particularly Class 4, with perfect precision, recall, and F1-score.
  - The overall performance metrics are strong, indicating a well-balanced model with minimal bias toward any class.

- **Plot Training and Validation Accuracy**



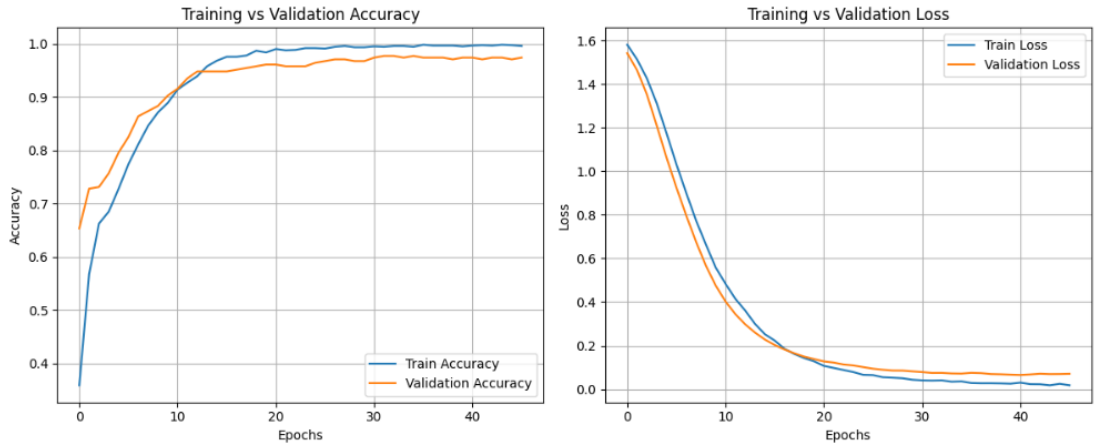
3.3.8 Hypertuned LSTM classifier

- **Random Search**: The hyperparameters (number of units, dropout rate, and learning rate) are optimized using a random search approach.
- **Tuning Objective**: The objective is to maximize validation accuracy over 10 trials.
- **Best Hyperparameters**: The optimal hyperparameters are chosen based on the highest validation accuracy:
  - Units in Layer 1: 64
  - Dropout in Layer 1: 0.2
  - Units in Layer 2: 32
  - Dropout in Layer 2: 0.1

- Learning Rate: 0.001

- The model is trained for a maximum of 50 epochs with early stopping to avoid overfitting. If validation loss does not improve for 5 consecutive epochs, training stops early and restores the best weights.
- **Model Result**:

```
Test Loss: 0.06550686061382294
Test Accuracy: 0.9741100072860718
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
Predicted labels: [4 1 4 1 3 2 4 4 3 3 1 4 4 0 0 3 0 2 1 2 1 1 1 0 1 1 3 2 2 1 3 4 2 4 0 2 4
 1 4 1 1 2 0 1 4 0 2 4 3 2 1 1 3 3 0 1 1 1 3 3 2 2 1 3 0 1 0 2 4 2 2 0 1 4
 0 3 3 3 4 3 3 0 3 2 0 4 2 1 2 3 4 3 4 4 4 1 4 3 2 2 3 4 1 4 4 2 4 2 3 1 1
 1 3 3 1 1 1 1 3 4 0 1 0 4 2 4 4 2 3 1 1 1 3 1 3 4 3 3 3 3 4 2 3 0 0 0 2 4
 2 3 4 0 1 3 2 0 3 4 1 3 0 4 1 2 4 1 0 4 2 3 3 3 1 4 1 0 3 4 4 2 3 2 3 4 3
 1 2 1 0 3 3 3 4 2 3 4 1 2 1 2 1 1 0 0 3 3 1 1 3 2 2 0 0 2 0 2 4 1 4 1 4 0
 3 4 3 2 4 1 3 1 1 0 0 4 2 2 3 4 1 0 1 4 4 2 4 1 4 0 3 1 0 2 1 3 4 0 2 3 2
 2 1 1 1 3 4 4 4 1 3 4 3 1 1 3 1 2 2 1 3 3 1 4 0 1 3 0 3 2 4 1 4 4 4 2 3 0
 0 1 4 0 2 4 1 2 3 2 4 2 2]
True labels: [4 1 4 1 3 2 4 4 3 3 1 4 4 0 0 3 0 2 1 2 1 1 1 0 1 1 3 2 2 1 3 4 2 4 0 2 4
 1 4 1 1 2 0 1 4 0 2 4 3 2 1 1 3 3 0 1 1 1 3 3 2 2 1 0 0 1 0 2 4 2 2 0 1 4
 0 3 3 3 4 3 3 0 3 0 0 4 2 1 2 3 4 3 4 4 4 1 4 3 2 2 0 4 1 4 4 2 4 2 3 1 1
 1 3 3 1 1 1 1 3 4 0 1 0 4 2 4 4 2 3 3 1 1 3 1 3 4 3 3 3 3 4 2 3 0 0 0 2 4
 2 3 4 0 1 3 2 0 3 4 1 3 0 4 1 2 4 1 0 4 2 3 3 3 1 4 1 0 3 4 4 2 3 2 3 4 3
 1 2 1 0 3 3 3 4 2 0 4 1 2 1 2 1 1 0 0 3 3 1 0 3 2 2 0 0 2 0 2 4 1 4 1 4 0
 3 4 3 2 4 1 3 1 1 0 0 4 2 2 3 4 1 0 1 4 4 2 4 1 4 1 3 1 0 2 1 3 4 0 2 3 2
 2 1 1 1 0 4 4 4 1 3 4 3 1 1 3 1 2 2 1 3 3 1 4 0 1 3 0 3 2 4 1 4 4 4 2 3 0
 0 1 4 0 2 4 1 2 3 2 4 2 2]
```



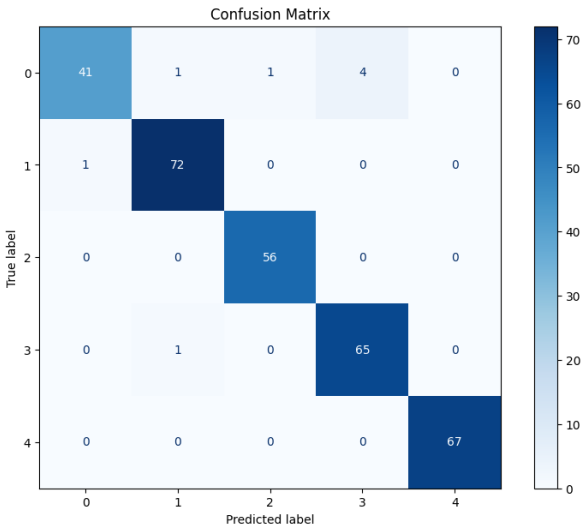
- Test Accuracy and Loss:
    - Test Accuracy: The model achieved an impressive test accuracy of 97.41%, indicating strong generalization performance on unseen data.
    - Test Loss: The test loss is 0.065, which is very low, reinforcing the high performance of the model in terms of prediction error.
- Predictions:
    - Predicted vs. True Labels: The comparison between the predicted and true labels shows that most of the predictions closely match the actual labels, further indicating that the model has been trained well and is accurately predicting the class labels.
- Training vs. Validation Accuracy Plot:
    - Convergence: Both the training and validation accuracy curves show a steady increase over the epochs, converging near the top

with only a small difference. This is a strong indicator that the model is well-tuned and not overfitting, as there is no large gap between the training and validation accuracy.

- Final Accuracy: Both the training and validation accuracies stabilize at almost 100%, indicating a highly accurate model for both training and validation datasets.

- Training vs. Validation Loss Plot:
  - Loss Reduction: The training and validation losses decrease steadily and converge towards zero, showing a smooth learning process. The model is effectively minimizing errors over time.
  - No Overfitting: The validation loss does not start increasing or diverging from the training loss, which indicates that the model is not overfitting to the training data.
- **Conclusion**:
  - The model performs exceptionally well, with nearly 100% accuracy for both training and validation datasets.
  - The consistent behavior of the training and validation accuracy/loss shows that the model is well-tuned and generalizes well to unseen data.
  - The early stopping criteria seem to have helped in stopping the training process at an optimal point, avoiding overfitting while achieving high accuracy.

- **Confusion Matrix**



`

- **Classification Report – Train**

```
Train Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      1.00       262
           1       1.00      1.00      1.00       236
           2       1.00      1.00      1.00       253
           3       0.99      1.00      1.00       243
           4       1.00      1.00      1.00       242

    accuracy                           1.00      1236
   macro avg       1.00      1.00      1.00      1236
weighted avg       1.00      1.00      1.00      1236
```

- **Classification Report – Test**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.87      0.92        47
           1       0.97      0.99      0.98        73
           2       0.98      1.00      0.99        56
           3       0.94      0.98      0.96        66
           4       1.00      1.00      1.00        67

    accuracy                           0.97       309
   macro avg       0.97      0.97      0.97       309
weighted avg       0.97      0.97      0.97       309
```

- Conclusion:
  - The model performs exceptionally well, with nearly 100% accuracy for both training and validation datasets.
  - The consistent behavior of the training and validation accuracy/loss shows that the model is well-tuned and generalizes well to unseen data.

`

4 Model evaluation

Objective:

The final model's objective is to design a chatbot utility that can analyze detailed descriptions of industrial accidents and highlight potential safety risks. This involves building a machine learning model that can accurately classify the severity of the risks described in the accident reports and provide actionable insights based on these classifications.

Final Model Description:

1. Model Architecture:
   - Type: LSTM (Long Short-Term Memory) neural network.
   - Layers:
     - LSTM Layer 1: Captures sequential dependencies in the text. The number of units in this layer is a hyperparameter tuned between 32 and 128.
     - Dropout Layer 1: Applies dropout to prevent overfitting. The dropout rate is a hyperparameter tuned between 0.1 and 0.5.
     - LSTM Layer 2: Further captures sequential dependencies and refines the learning. The number of units in this layer is tuned between 16 and 64.
     - Dropout Layer 2: Another dropout layer to reduce overfitting.
     - Dense Layer: Outputs the risk classification with a softmax activation function to handle multiple classes (Accident Levels).

2. Hyperparameters:
   - Number of Units in LSTM Layers:
     - Layer 1: 32, 64, 96, 128 units (tuned through RandomSearch).
     - Layer 2: 16, 32, 48, 64 units (tuned through RandomSearch).
   - Dropout Rates:
     - Dropout Layer 1: 0.1 to 0.5.
     - Dropout Layer 2: 0.1 to 0.5.
   - Learning Rate:
     - Adam optimizer's learning rate tuned between 1e-4 and 1e-2.
   - Epochs: 50 (number of times the model iterates over the training dataset).
   - Batch Size: 32 (number of samples processed before the model is updated).

3. Data Preparation:
   1. Feature Scaling: Standardized the feature values for numerical stability.
   2. Text Preprocessing: Tokenized, normalized, and encoded text descriptions using TF-IDF or embeddings.
   3. Reshaping: Converted the data into a shape suitable for LSTM input (samples, time steps, features).

`

4. Evaluation Metrics:
    1. Classification Performance:
        - Accuracy: The proportion of correct predictions out of total predictions.
        - Precision: Measures the proportion of true positive predictions among all positive predictions made by the model.
        - Recall: Measures the proportion of true positive predictions among all actual positive instances.
        - F1-Score: The harmonic mean of precision and recall, providing a single metric that balances both.

    2. Model Evaluation Results:
        - Test Classification Report for Hypertuned LSTM:
            - Accuracy: 97% (highly accurate predictions on the test set).
            - Precision, Recall, and F1-Score: Overall high, indicating effective risk classification across different accident levels.
        - Train Classification Report for Hypertuned LSTM:
            - Accuracy: 100% (perfect classification on the training set, indicating potential overfitting if not cross-validated properly).
    3. Additional Evaluation:
        - Loss and Accuracy Curves:
            - Training Loss vs. Validation Loss: Ensures the model is not overfitting and generalizes well to unseen data.
            - Training Accuracy vs. Validation Accuracy: Shows how well the model learns and generalizes.

Recommendations:

1. Final Model Choice:
    - The hypertuned LSTM model is the final choice due to its superior performance metrics on the test set compared to the base model and the ability to generalize well to new data.
2. Model Deployment:
    - Integration: Integrate the model into the chatbot backend to process real-time accident descriptions and provide risk assessments.
    - Monitoring: Continuously monitor model performance and user feedback to ensure accuracy and relevance.
3. Further Improvements:
    - Regular Updates: Retrain the model with new data periodically to maintain its performance.
    - Advanced Techniques: Explore more advanced models like transformers for potentially better contextual understanding.

`

The hypertuned LSTM model successfully meets the project objective by accurately classifying risk levels based on accident descriptions, providing valuable insights for enhancing industrial safety.

`

5 Comparison to benchmark

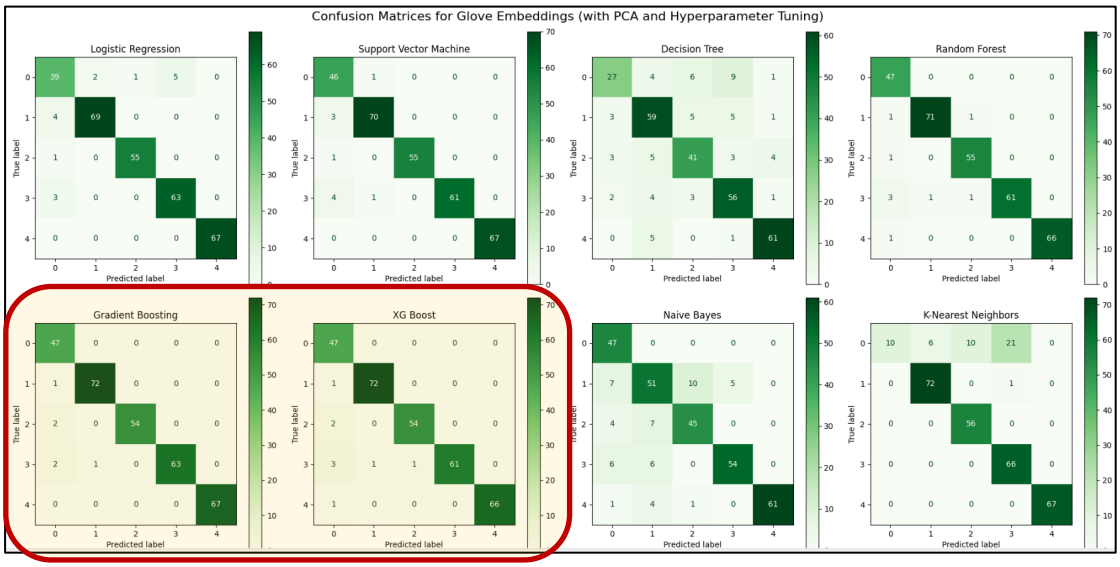
Milestone 1 benchmark results

- Gradient Boosting and XG Boost consistently exhibit the best performance across all embeddings, both before and after hyper-tuning.
- Gradient Boosting and XG Boost had a benchmark train vs. test accuracy of 99.99% vs 98.38% and train vs. test recall of 99.99% vs. 97.08% (The classification results and confusion matrices are highlighted in below images)

```
print("Glove Results (with Hypertuning & PCA)")
display(glove_results_ht_pca)
```

Glove Results (with Hypertuning & PCA)

	Classifier	Train Accuracy	Train Precision	Train Recall	Train F1-score	Test Accuracy	Test Precision	Test Recall	Test F1-score	Training Time	Prediction Time
0	Logistic Regression	0.998382	0.998385	0.998382	0.998382	0.954693	0.954891	0.954693	0.954703	77.103474	0.000402
1	Support Vector Machine	0.996764	0.996777	0.996764	0.996762	0.964401	0.968638	0.964401	0.965181	2.277558	0.057942
2	Decision Tree	0.988673	0.988722	0.988673	0.988671	0.802589	0.816676	0.802589	0.807753	14.266272	0.000420
3	Random Forest	0.999191	0.999194	0.999191	0.999191	0.961165	0.965057	0.961165	0.961746	111.870976	0.011469
4	Gradient Boosting	0.999191	0.999194	0.999191	0.999191	0.970874	0.972769	0.970874	0.971221	273.573186	0.006782
5	XG Boost	0.999191	0.999194	0.999191	0.999191	0.970874	0.973752	0.970874	0.971284	16.019643	0.004797
6	Naive Bayes	0.907767	0.909527	0.907767	0.906243	0.834951	0.845118	0.834951	0.835399	0.150775	0.001675
7	K-Nearest Neighbors	0.999191	0.999194	0.999191	0.999191	0.870550	0.880995	0.870550	0.836126	0.635684	0.003601



`

Final model results (LSTM Hypertuned model)

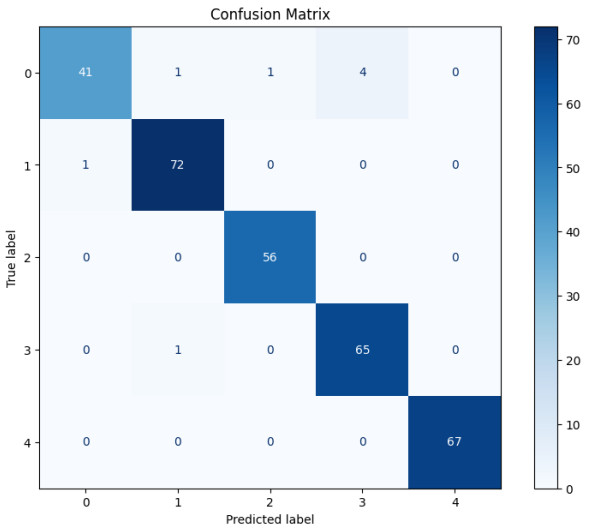
- The LSTM Hypertuned model performed exceptionally well, with nearly 100% accuracy for training and 97% for validation datasets.
- Train vs. Test recall across all the classes remains same except for class 0
- The consistent behavior of the training and validation accuracy/loss shows that the model is well-tuned and generalizes well to unseen data.
- The early stopping criteria seem to have helped in stopping the training process at an optimal point, avoiding overfitting while achieving high accuracy.

Train Classification Report:

	precision	recall	f1-score	support
0	1.00	0.99	1.00	262
1	1.00	1.00	1.00	236
2	1.00	1.00	1.00	253
3	0.99	1.00	1.00	243
4	1.00	1.00	1.00	242
accuracy			1.00	1236
macro avg	1.00	1.00	1.00	1236
weighted avg	1.00	1.00	1.00	1236

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.87	0.92	47
1	0.97	0.99	0.98	73
2	0.98	1.00	0.99	56
3	0.94	0.98	0.96	66
4	1.00	1.00	1.00	67
accuracy			0.97	309
macro avg	0.97	0.97	0.97	309
weighted avg	0.97	0.97	0.97	309



`

Comparative Analysis (Final model vs. Milestone 1 benchmark results)

- **Accuracy and Recall**: The LSTM model shows a slight dip in validation accuracy and recall compared to the XG Boost test metrics. However, the LSTM model's performance is still very high.
- **Generalization**: Both approaches seem to generalize well, though the LSTM's slightly lower validation recall for class 0 might suggest a bit more focus on handling that specific class.
- **Complexity and Interpretability**: LSTM models, being deep learning-based, generally come with increased complexity and reduced interpretability compared to tree-based methods like Gradient Boosting and XG Boost.

Improvement Analysis

- **Benchmark Improvement**: While the LSTM did not surpass the Gradient Boosting or XG Boost models in all metrics, it demonstrated comparable performance with potential benefits in handling sequential or time-series data more effectively (if that's relevant to your application).
- **Tuning and Early Stopping**: The LSTM benefits from hypertuning and early stopping, which seem to have optimized its training process to prevent overfitting effectively.

Conclusion

The final solution with the hypertuned LSTM model highlights the strength of deep learning in achieving high accuracy and maintaining strong generalization, particularly in tasks involving sequential data. While it may not have outperformed models like Gradient Boosting or XGBoost in some metrics, the LSTM model offers a powerful and reliable alternative, especially when handling data with inherent sequence dependencies, as seen in this industrial safety context.

`

6 Visualizations

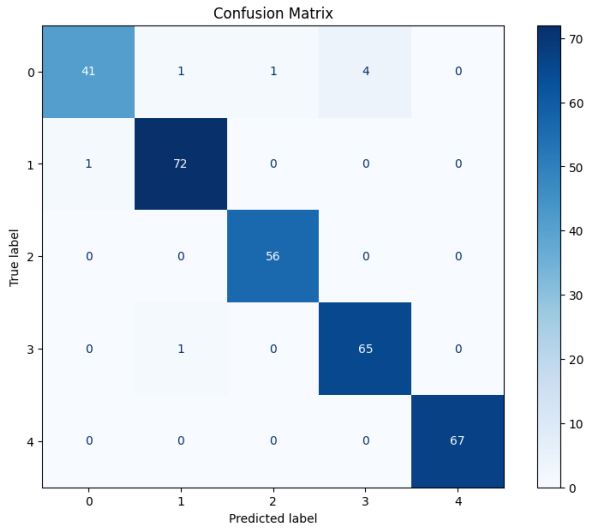
- The LSTM Hypertuned model performed exceptionally well, with nearly 100% accuracy for training and 97% for validation datasets.
- Train vs. Test recall across all the classes remains same except for class 0
- The consistent behavior of the training and validation accuracy/loss shows that the model is well-tuned and generalizes well to unseen data.
- The early stopping criteria seem to have helped in stopping the training process at an optimal point, avoiding overfitting while achieving high accuracy.

Train Classification Report:

	precision	recall	f1-score	support
0	1.00	0.99	1.00	262
1	1.00	1.00	1.00	236
2	1.00	1.00	1.00	253
3	0.99	1.00	1.00	243
4	1.00	1.00	1.00	242
accuracy			1.00	1236
macro avg	1.00	1.00	1.00	1236
weighted avg	1.00	1.00	1.00	1236

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.87	0.92	47
1	0.97	0.99	0.98	73
2	0.98	1.00	0.99	56
3	0.94	0.98	0.96	66
4	1.00	1.00	1.00	67
accuracy			0.97	309
macro avg	0.97	0.97	0.97	309
weighted avg	0.97	0.97	0.97	309



`

7 Implications

Impact on Industrial Safety:

- The solution aims to **enhance workplace safety** by analyzing past accident data, identifying patterns, and flagging potential risks in real-time. This approach will allow industries to take immediate action on insights, **preventing accidents** and improving safety outcomes.
- Using machine learning models like **Gradient Boost, XGBoost, and LSTM** ensures accurate risk analysis by evaluating the severity and frequency of accidents. These models will **improve decision-making**, help prioritize safety interventions, and optimize resource allocation, making safety protocols more targeted and effective in preventing accidents in **high-risk** industries.

Integration of LSTM:

- Temporal Pattern Analysis: LSTM networks, a type of advanced RNN, are integrated into the system to capture and analyze long-term temporal dependencies in accident data. Unlike traditional RNNs, LSTMs excel at retaining information over extended sequences, allowing the model to understand the chain of events leading to accidents. This ability to capture both short-term and long-term patterns is crucial for identifying recurring safety risks and improving accident prevention protocols over time.
- Handling Time-Series Data for Enhanced Accuracy: LSTMs are particularly well-suited for handling time-series data, making them ideal for predicting accidents based on past sequences of events. Their ability to remember important information while discarding irrelevant data ensures that the system adapts dynamically to evolving patterns. By continuously analyzing sequential data, such as accident logs, maintenance schedules, and incident reports, LSTMs help forecast potential safety hazards. This enables organizations to implement proactive mitigation strategies, ultimately improving safety protocols and reducing accidents.

Future Leveraging of Large Language Models (LLMs):

- Incorporating LLMs like ChatGPT in the future will greatly enhance the system's natural language processing capabilities, allowing it to **better understand complex** incident descriptions and employee reports. This will improve the accuracy of extracting actionable insights, enabling the system to **identify risks** and safety issues more effectively.
- Future LLMs will provide a **conversational interface** for safety professionals and workers, offering real-time guidance on safety protocols, equipment handling, and emergency responses. Additionally, LLMs will continuously **learn from user feedback**, accident reports, and safety documents, leading to increasingly personalized and **context-aware recommendations** that evolve with changing safety needs.

`

- **Recommendations**:
  - **Deploy Across High-Risk Sectors**: Plan to implement the solution in industries like mining and construction, where accident rates and severity are higher, to improve workplace safety.
  - **Integrate LLMs for Enhanced Communication**: Utilize LLMs to create a conversational interface that provides employees with detailed safety guidance based on real-time data.

- **Confidence Level**:
  - With the use of **LSTM** and the planned integration of **LLMs** , the chatbot is expected to offer a **highly robust safety** solution. LSTMs enhance **pattern detection** and risk prediction, while the addition of LLMs will improve communication by providing **context-aware** recommendations. This combination is anticipated to ensure **high accuracy**, **adaptability**, and effective **real-time communication**, making it a comprehensive solution for industrial safety environments.
  - With hypertuned LSTM model we were able to achieve **validation accuracy of 97.73% with a confidence level of 95%.**

`

8 Limitations

Generalization Issues:

- Real-World Complexity: Although LSTM and LLM integration is expected to improve prediction and understanding, there may still be challenges with unique, real-time accident scenarios, especially those with limited historical data.
- LLM Context Sensitivity: While future LLMs like ChatGPT are anticipated to improve comprehension, they may still misinterpret complex or ambiguous safety reports without domain-specific fine-tuning.

Class Imbalance:

- The dataset's imbalance, with most accidents being less severe, may hinder the accurate prediction of high-severity accidents (Level IV or V). This imbalance could affect both machine learning models and LSTM, potentially leading to an underestimation of severe risks.
- While LSTM are expected to improve sequential learning, they may still face challenges in underrepresented data, making it difficult to predict rare critical incidents accurately.

Model Limitations:

- Computational Requirements: The use of LSTM and the planned integration of LLMs is likely to increase computational complexity and resource demands. Training models like LLMs will require significant processing power, which might affect scalability in real-time industrial settings.
- Accuracy in Rare Cases: Despite advancements, LSTM and LLMs may still face difficulties in handling rare and critical accident cases due to limited data and the unpredictable nature of such incidents.

- **Potential Enhancements:**
  - Fine-Tuning LLMs: Regular fine-tuning of future LLMs with domain-specific safety data can enhance their accuracy and relevance in analyzing complex safety incidents.
  - Scalable Solutions: Optimize the use of LSTM and LLMs by implementing them in a cloud-based environment to handle real-time data streams without compromising performance.

`

9 Closing Reflections

Key Learnings:

- Sequential Data Handling with LSTM: The use of LSTMs has significantly improved the system's ability to predict recurring accidents by analyzing sequential trends in accident data.
- LLMs for Enhanced Text Understanding: Anticipated integration of LLMs like ChatGPT will offer improved natural language understanding, enhancing the system's ability to extract critical insights from incident descriptions and reports.
- Importance of Preprocessing: High-quality data preprocessing, including text tokenization, embeddings, and normalization, will be vital in improving the performance of both LSTMs and LLMs.

Future Improvements:

- Enhance Real-Time Capabilities with LSTM and LLM: Future efforts will focus on integrating LLMs to complement the LSTM's existing capabilities. While LSTMs already process temporal data effectively, integrating LLMs will enhance text comprehension, providing real-time, actionable insights in dynamic industrial environments.
- Advanced LLM Use: Explore more advanced LLMs like GPT-4 to further refine the system's understanding of complex safety reports and generate more contextually accurate responses.
- Fine-Tuning for Domain-Specific Use: Continuous fine-tuning of LLMs with domain-specific safety language and protocols will enhance their effectiveness in providing accurate safety recommendations.

- **What to Do Differently Next Time:**
  - Expand Data Sources for LSTM and LLM Training: In future iterations, gather a more diverse dataset, including global accident data, to improve the generalization of both LSTM and LLM models.
  - Focus on LLM Adaptation: Incorporate more industry-specific LLM models fine-tuned for industrial safety to improve the system's domain expertise.
  - Iterative Testing for LLM and LSTM: Conduct more frequent testing of both LSTM and LLM models during development to address any overfitting and ensure consistent performance, especially for rare but critical accidents.
  - Combining LLM and LSTM Strengths: Plan to use LLMs for real-time understanding of new accident descriptions and LSTMs to forecast patterns, creating a hybrid model for both immediate and long-term safety interventions.