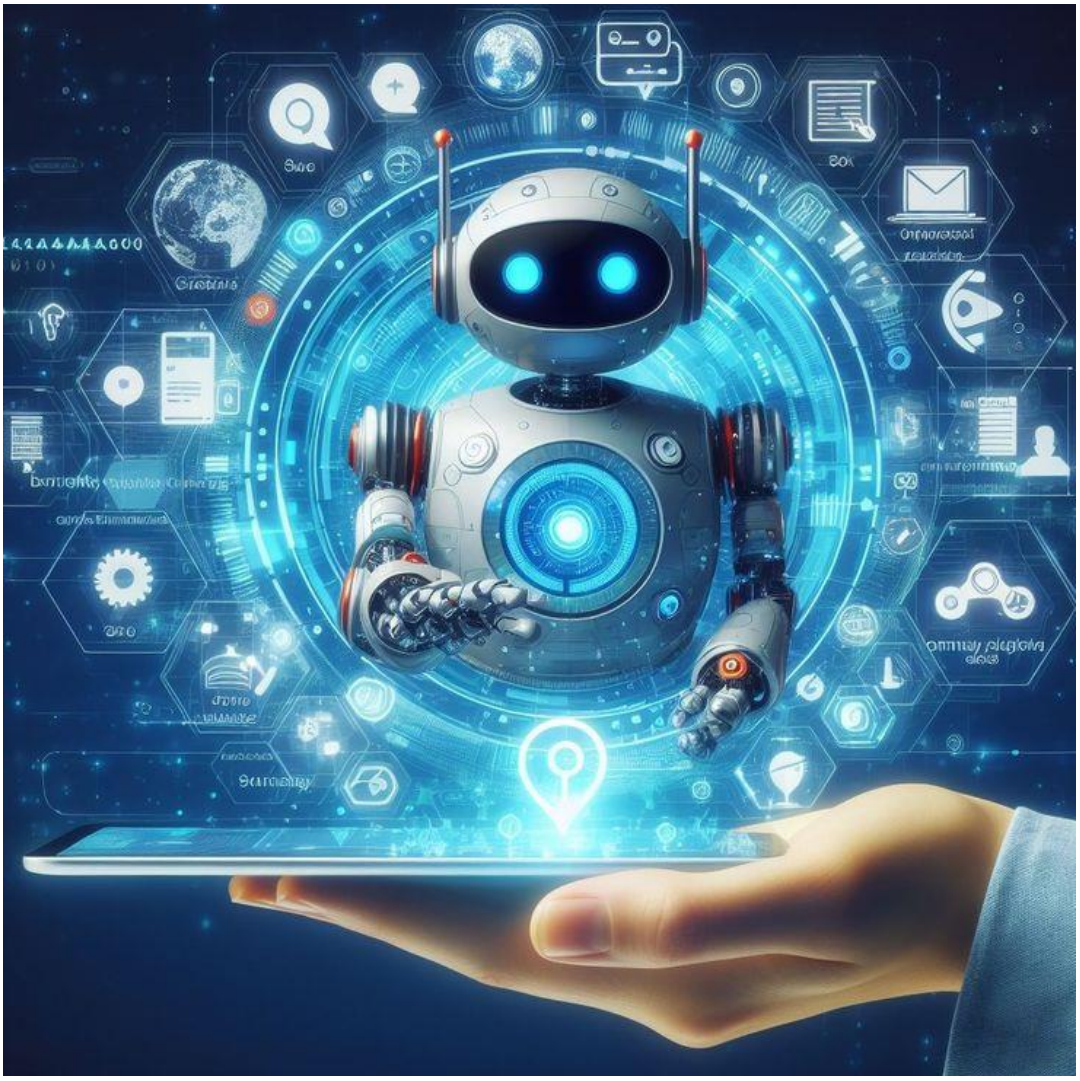


INDUSTRIAL SAFETY NLP CHATBOT

Industrial Safety – NLP based Chatbot for Accident Risk Highlighting

Revision: 1 (Interim Report)

Contributors:

- Abhinav
- Bharani Kumar A B
- Debosree
- Hemant
- Siddharth

Submitted On:
21st September, 2025

Table of Contents

1. Project Overview

- **PROBLEM STATEMENT:** Design and implement an ML/DL-based chatbot utility to help industrial safety professionals identify and assess safety risks from incident descriptions using Natural Language Processing.

- **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 environment.

- **DATA DESCRIPTION:** This The database is basically records of accidents from 12 different plants in 03 different countries which every line in the data is an occurrence of an accident.

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

- **COLUMNS DESCRIPTION:**
  1. Data: timestamp or time/date information
  2. Countries: which country the accident occurred (anonymised)
  3. Local: the city where the manufacturing plant is located (anonymised)
  4. Industry sector: which sector the plant belongs to
  5. Accident level: from I to VI, it registers how severe was the accident (I means not severe but VI means very severe)
  6. 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)
  7. Genre: if the person is male of female
  8. Employee or Third Party: if the injured person is an employee or a third party
  9. Critical Risk: some description of the risk involved in the accident
  10. Description: Detailed description of how the accident happened.

2. Dataset Description

- ## Dataset Overview
  - The dataset used in this project consists of **425 accident records**, collected from **12 different plants across 3 countries**. Each row corresponds to one recorded accident, and the dataset includes **10 variables** that capture details of the incident, the affected person, and the risk involved.

- **Dataset Structure:**
  1. **Data**: A timestamp representing the date and time of the accident. This column is of datetime type and can be used to study temporal trends in accident occurrence (e.g., peak months, seasonal effects).
  2. **Countries**: The country where the plant is located. Due to confidentiality, the names are anonymized.
  3. **Local**: The specific city or locality of the plant anonymized for privacy.
  4. **Industry Sector**: The industrial sector to which the plant belongs. This feature helps compare risk patterns across sectors.
  5. **Accident Level**: The severity of the accident, categorized from **I (least severe)** to **VI (most severe)**. In this dataset, values range only from **I to V**.
  6. **Potential Accident Level**: An estimate of how severe the accident *could have been*, based on other conditions. This often differs from the actual recorded severity and can provide useful context.
  7. **Genre**: Gender of the affected person (Male/Female).
  8. **Employee or Third Party**: Indicates whether the injured person was an **employee** or a **third-party contractor/visitor**.
  9. **Critical Risk**: A textual description of the key hazard involved in the incident.
  10. **Description**: A detailed natural language account of how the accident occurred. This is the most critical variable for our NLP pipeline.

3. Summary Statistics

- **Accident Levels Distribution**
  - A key observation from the dataset is the class imbalance in accident severity:

Accident Level	Count	Percentage
I	316	~74%
II	40	~9%
III	31	~7%
IV	30	~7%
V	8	~2%

  - This distribution shows that nearly three-quarters of all recorded incidents are low-severity (Level I). Severe accidents (Levels IV and V) are rare, representing less than 10% of the dataset. This imbalance presents a major challenge for classification models, as they tend to be biased toward the majority class.

- **Description Length Analysis**
  - Since the **Description** column is our primary text feature, we also analysed its length:
    - **Minimum length**: 94 characters
    - **Maximum length**: 1,029 characters
    - **Median length**: ≈ 335 characters

- This suggests that most accident descriptions are fairly detailed, often equivalent to several sentences. The variability in text length indicates the need for careful preprocessing to normalize input representation.

Insights

- The dataset is **rich in textual data**, particularly the Description and Critical Risk fields, making it suitable for NLP-based approaches.
- The **severe class imbalance** in Accident Level highlights the importance of evaluation metrics like **Macro-F1**, which better capture performance across all classes.
- Accident descriptions are of **moderate length**, meaning both Bag-of-Words and TF-IDF representations are feasible without encountering extreme sparsity.

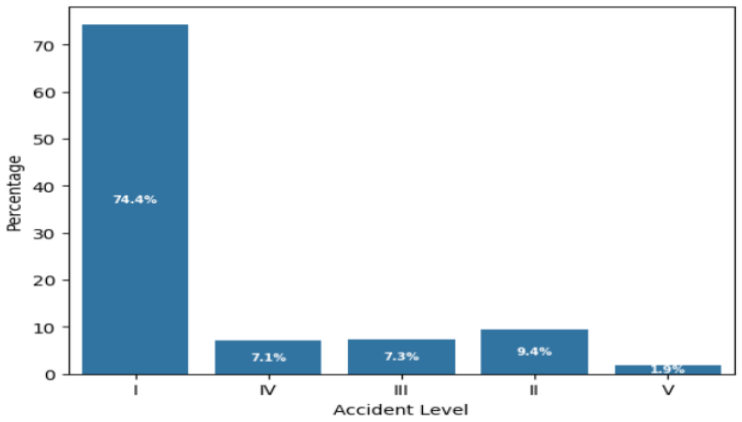
Data Quality Checks

- Before proceeding to preprocessing and model building, we examined the dataset for basic quality issues:
- **Missing Values**:
  No missing values were found across any of the 10 columns. This suggests that the dataset is relatively clean and complete, reducing the need for imputation.
- **Duplicate Records**:
  A total of **7 duplicate rows** were identified. Duplicate records can bias models by overrepresenting specific accident types. These were removed, and the dataset was reset to ensure each row represents a unique accident occurrence. After this step, the dataset shape became **(418, 10)**.
- **Data Types**:
  - The **Data** column is in datetime format, which is suitable for temporal analysis.
  - All other columns are categorical or text-based (object type in Pandas).
  - No conversions were necessary beyond confirming consistency.

Final Data Characteristics

4. EDA

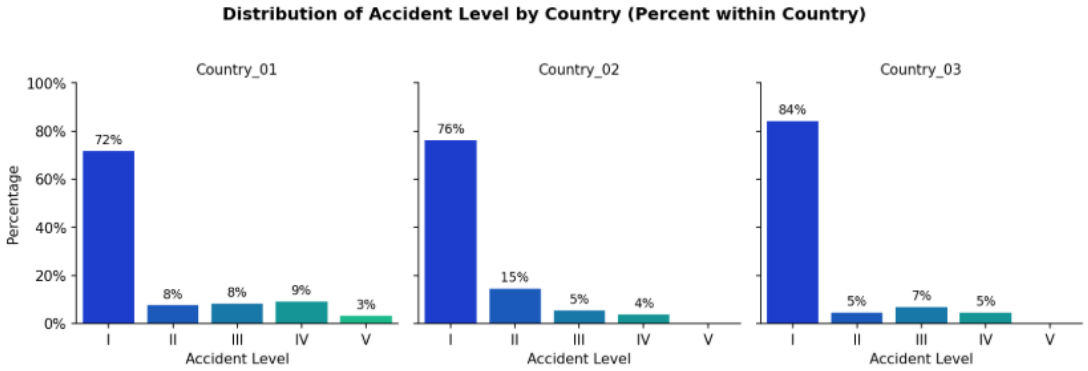
4.1 Overall Accident Level (% of incidents)



Summary:

- Most incidents are Level I; Levels II/III/IV form a small tail; Level V is very rare.

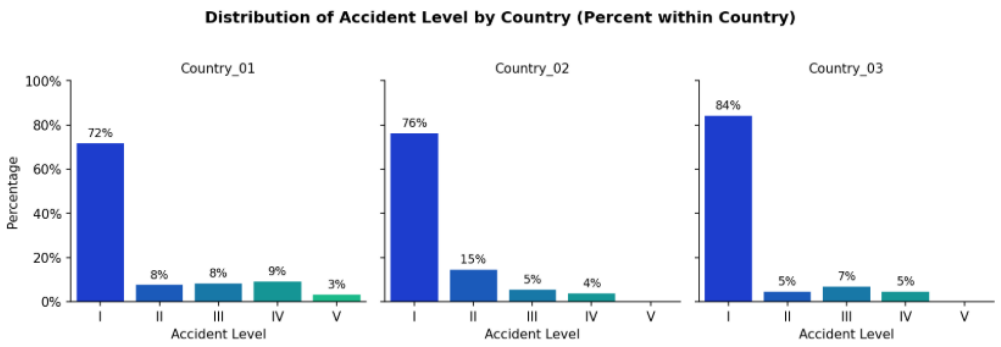
4.2 Accident Level Vs Industry Sector (Mining/Metals/Others)



Summary:

- **Actual incidents:** Level I dominates in every sector, but **Mining has the biggest II–IV tail** (more mid/high-severity than Metals or Others) → primary focus area.

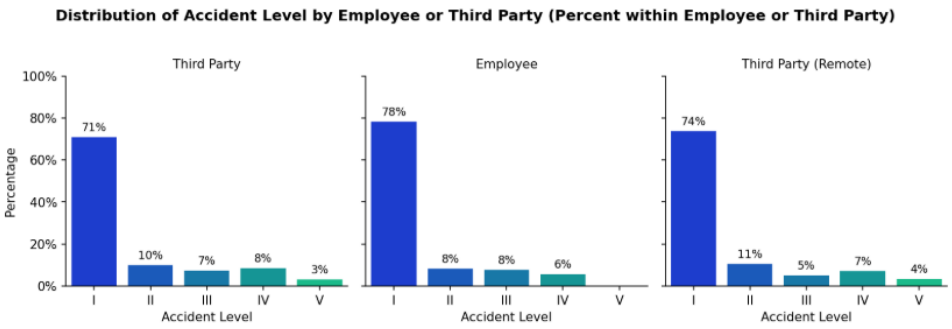
4.3 Accident Level vs Country



Summary:

- Most events are minor (Level I) in every country.
- Country_02 shows more Level II cases than others — needs attention.
- Country_01 has a wider mix of Level II–IV (more mid-severity than Country_03).
- Country_03 is mostly Level I with only a small mid-severity tail.

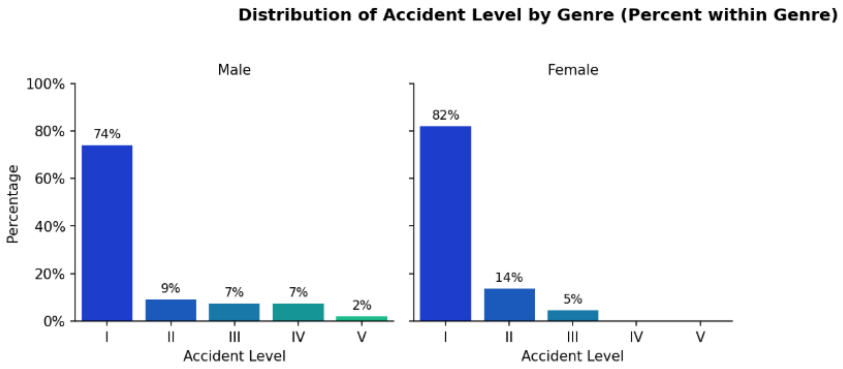
4.4 Accident Level vs Employee / Third-Party (incl. Remote)



Summary:

- Most events are minor (Level I) for all groups (Employee, Third-Party, Third-Party Remote).
- Employees have the safest profile (highest Level I, very little Level V).
- Third-Party Remote shows more Level II and the highest Level V (worst-case) → needs attention.
- On-site Third-Party also has a small Level V tail.

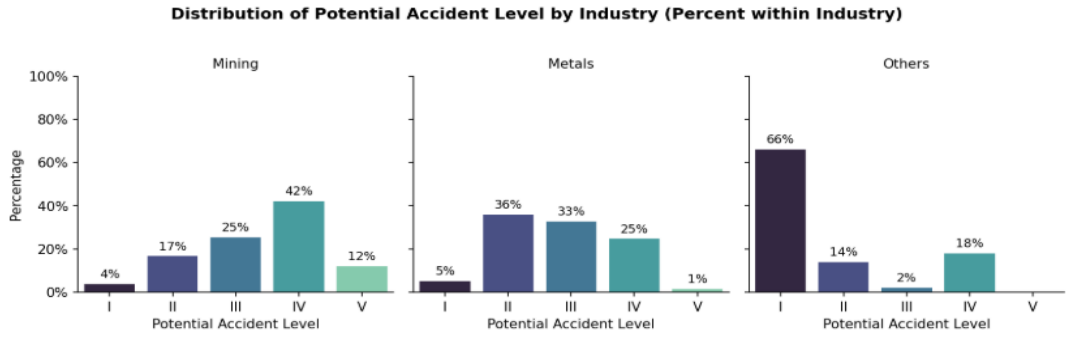
4.5 Accident Level Vs Gender



Summary:

- **Male**-heavy exposure to various incident level.
- **Females** have more minor incidents (~80% Level I).

4.6 Potential Accident Level vs Industry



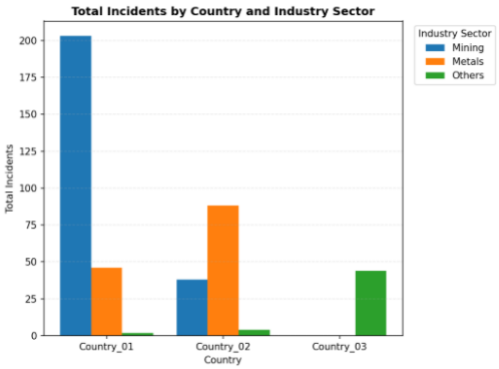
Summary:

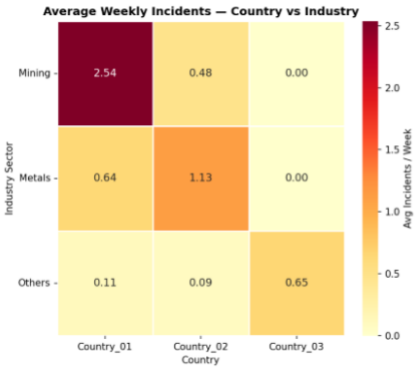
- **Potential severity:** Mining shows the strongest escalation risk (heavy III–IV and some V), while Metals is mainly II–III and Others mostly I → prioritize controls in Mining first.

4.7 Weekly Incidents by Industry Sector — Country wise

Weekly Incidents by Industry Sector — Country wise

Country	Industry	Avg/Week	Peak/Week	Weeks>0	Weeks(active)	Total	Country Share
Country_01	Mining	2.54	8	74	80	203	80.9%
Country_01	Metals	0.64	2	34	72	46	18.3%
Country_01	Others	0.11	1	2	18	2	0.8%
Country_02	Metals	1.13	5	51	78	88	67.7%
Country_02	Mining	0.48	3	28	79	38	29.2%
Country_02	Others	0.09	1	4	46	4	3.1%
Country_03	Others	0.65	6	27	68	44	100.0%





Summary:

- Country_01 → Mining dominates (81%, avg. 2.54/week).
- Country_02 → Metals dominates (68%, avg. 1.13/week).
- Country_03 → Incidents entirely in Others (100%) which could indicate reporting gap.

4.8 Critical Risk Vs Accident Level

]:

Accident Level	I	II	III	IV	V	total
Critical Risk						
Not applicable	0	0	0	1	0	1
Bees	10	0	0	0	0	10
Blocking and isolation of energies	3	0	0	0	0	3
Burn	0	0	1	0	0	1
Chemical substances	15	2	0	0	0	17
Confined space	1	0	0	0	0	1
Cut	11	2	1	0	0	14
Electrical Shock	2	0	0	0	0	2
Electrical installation	0	0	0	1	0	1
Fall	6	0	0	2	1	9
Fall prevention	5	0	0	1	0	6
Fall prevention (same level)	6	0	0	1	0	7
Individual protection equipment	0	1	0	0	0	1
Liquid Metal	3	0	0	0	0	3
Machine Protection	2	0	0	0	0	2
Manual Tools	12	5	3	0	0	20
Others	172	21	23	13	3	232
Plates	1	0	0	0	0	1
Poll	0	0	0	1	0	1
Power lock	0	0	0	1	2	3
Pressed	17	1	2	4	0	24
Pressurized Systems	6	1	0	0	0	7
Pressurized Systems / Chemical Substances	2	1	0	0	0	3
Projection	10	2	0	1	0	13
Projection of fragments	2	0	0	0	0	2
Projection/Burning	0	1	0	0	0	1
Projection/Choco	1	0	0	0	0	1
Projection/Manual Tools	1	0	0	0	0	1
Suspended Loads	4	0	1	1	0	6
Traffic	1	0	0	0	0	1
Vehicles and Mobile Equipment	5	1	0	1	1	8
Venomous Animals	16	0	0	0	0	16
remains of choco	2	2	0	2	1	7

Summary:

- "Others" is the dominant category (232 cases), followed by Pressed, Manual Tools, Venomous Animals, Chemical Substances, and Cuts as key contributors.
- Severe accidents (Levels IV–V) mainly occur in Falls, Pressed, Power Lock, Venomous Animals, and Remains of Choco, indicating critical safety concerns.

4.9 Critical Risk Vs Industry Sector

Critical Risk \ Industry Sector	Metals	Mining	Others	total
Not applicable	1	0	0	1
Bees	0	0	10	10
Blocking and isolation of energies	3	0	0	3
Burn	1	0	0	1
Chemical substances	15	2	0	17
Confined space	1	0	0	1
Cut	10	4	0	14
Electrical Shock	0	2	0	2
Electrical installation	0	1	0	1
Fall	2	5	2	9
Fall prevention	3	2	1	6
Fall prevention (same level)	6	1	0	7
Individual protection equipment	0	1	0	1
Liquid Metal	3	0	0	3
Machine Protection	2	0	0	2
Manual Tools	14	5	1	20
Others	33	179	20	232
Plates	1	0	0	1
Poll	0	0	1	1
Power lock	1	2	0	3
Pressed	17	7	0	24
Pressurized Systems	6	1	0	7
Pressurized Systems / Chemical Substances	3	0	0	3
Projection	4	9	0	13
Projection of fragments	0	2	0	2
Projection/Burning	1	0	0	1
Projection/Choco	0	0	1	1
Projection/Manual Tools	0	1	0	1
Suspended Loads	5	1	0	6
Traffic	0	0	1	1
Vehicles and Mobile Equipment	0	8	0	8
Venomous Animals	2	1	13	16
remains of choco	0	7	0	7

Summary:

- Mining has the most incidents, mainly due to a very large “Others” count (179).
- Metals is second; top risks here are Pressed (17), Chemical substances (15), and Manual tools (14).
- Others sector is small but skewed to animal/insect hazards: Venomous animals (13) and Bees (10).

4.10 Overall Insights:

- Most accidents are **minor**, with only a small portion being serious.
- The **Mining sector** shows more mid and severe accidents compared to others.
- **Employees** mostly face minor accidents, while **remote third-party workers** are the most at risk of serious ones.
- **Men** face a wider spread of accident severities, while **women** mostly face minor cases.
- The biggest risks come from **Falls, Pressed, Chemicals, Manual Tools, and Animals**, which drive most severe incidents.

- By country: **Country 01** has a wider mix of mid-severity accidents, **Country 02** shows more Level II cases, and **Country 03** is mostly minor — which may point to **under-reporting**.

5. Preprocessing and Feature Engineering

Once the raw dataset was understood through exploratory data analysis, the next step was to prepare the textual accident descriptions for machine learning models. Since the "Description" field forms the most critical input feature, it required systematic cleaning and transformation into numerical features that algorithms can understand.

5.1. Text Cleaning

The descriptions contained various inconsistencies such as special characters, punctuation, and mixed cases. To normalize the data, we first applied a **regular expression** to remove all non-alphanumeric symbols. Next, we converted the text to **lowercase** to ensure that words like "Fire" and "fire" were treated as the same token. Extra whitespaces, which could create artificial tokens, were stripped from all records. This step ensured consistency and reduced noise in the vocabulary.

Removing special characters

```
# defining a function to remove special characters
def remove_special_characters(text):
    # Defining the regex pattern to match non-alphanumeric characters
    pattern = '[^A-Za-z0-9]+'

    # Finding the specified pattern and replacing non-alphanumeric characters with a blank string
    new_text = re.sub(pattern, ' ', text)

    return new_text
```

```
# Applying the function to remove special characters
data['cleaned_text'] = data['Description'].apply(remove_special_characters)
```

```
# checking a couple of instances of cleaned data
data.loc[0:3, ['Description','cleaned_text']]
```

- We can observe that regex removed the special characters ilike comma (,), equals (=), slash (/), Hiphen (-) etc.,

Lowercasing

```
# changing the case of the text data to lower case
data['cleaned_text'] = data['cleaned_text'].str.lower()
```

```
# checking a couple of instances of cleaned data
data.loc[0:3, ['Description','cleaned_text']]
```

- We can observe that all the text has now successfully been converted to lower case.

For example, the raw text:

" *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. "

was transformed into:

" *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*"

5.2. Lemmatization and Stopword Removal

Raw tokens often include grammatical variations that do not contribute additional meaning (e.g., *"Removing", "Removes"* → *"Remove"*). We used **spaCy's lemmatizer** to reduce each word to its base form (lemma). Alongside this, we removed **stopwords** such as *"the", "was", "of"*, which occur frequently but do not contribute significantly to distinguishing accidents. This reduced dimensionality and highlighted the core action words

Lemmatization

```
def spacy_lemmatize(text):
    doc = nlp(text)
    return ' '.join([
        token.lemma_ for token in doc
        if not token.is_punct and not token.is_space and not token.is_stop
    ])
```

```
# Apply on cleaned_text (not the one with stopwords already removed)
data['final_cleaned_text'] = data['cleaned_text'].apply(spacy_lemmatize)
```

```
# checking a couple of instances of cleaned data
data.loc[0:2, ['Description', 'cleaned_text', 'final_cleaned_text']]
```

We can see that the below sample words and corresponding word after applying lemmatization, This look much better that porter stemming

- removing -> remove
- proceeds -> proceed
- seeing -> see
- supports -> support

So the previous example was finally converted into
"activation sodium sulphide pump piping uncoupled sulfide solution design area reach maid immediately use emergency shower direct ambulatory doctor later hospital note sulphide solution 48 gram liter"

5.3. Feature Extraction: Bag of Words and TF-IDF

After text normalization, we converted the descriptions into numerical representations:

- **Bag of Words (BoW)**: This method counts how often each word (or n-gram) occurs in a document. We considered both unigrams and bigrams. Although BoW is simple, it provides a solid baseline and is computationally efficient.
- **Term Frequency–Inverse Document Frequency (TF-IDF)**: Unlike BoW, TF-IDF down-weights very common words and upweights rarer but informative words. For example, terms like *"safety"* might occur in almost all descriptions, but words like *"explosion"* or *"electrocution"* carry much higher importance. TF-IDF therefore provides a more discriminative feature space.

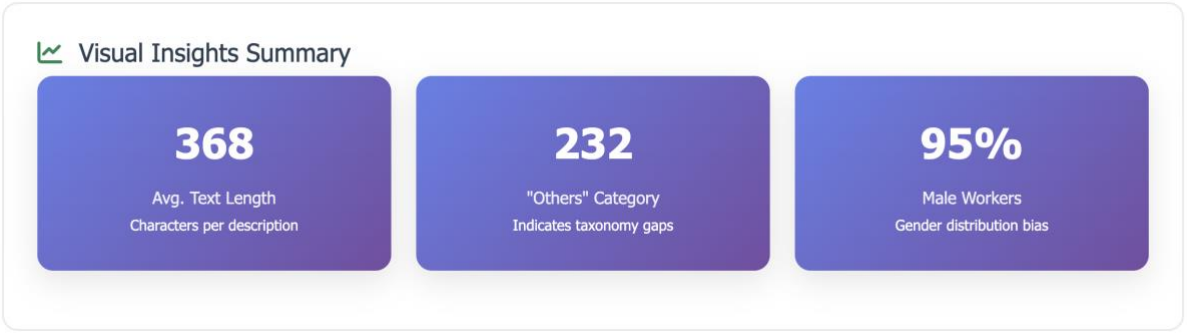
Define vectorizers (BoW and TF-IDF)

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

bow_vectorizer = CountVectorizer(ngram_range=(1,2), min_df=2)
tfidf_vectorizer = TfidfVectorizer(ngram_range=(1,2), min_df=2, sublinear_tf=True)
```

5.4. Train-Test Split

We split the dataset into **80% training** and **20% testing**, using a **stratified split** to preserve the imbalance across Accident Levels. Without stratification, rare severe classes (Levels IV and V) might disappear completely from the test set, making evaluation meaningless.



EDA Impact on Model Development

What the visualizations revealed:

- Severe class imbalance requiring specialized sampling techniques
- Industry-specific risk patterns suggesting feature engineering opportunities
- Text length correlation with severity indicating meaningful signal
- Geographic clustering of risks enabling location-based features

Preprocessing decisions made: Based on EDA findings, we implemented advanced NLP preprocessing with domain-specific considerations, applied oversampling for class balance, and selected TF-IDF vectorization for better feature representation.

6. Model Selection and Model Building

After preprocessing the dataset, the next step was to experiment with different machine learning algorithms. Since the dataset is relatively small (425 samples) and text-based, we focused on **classical ML models** rather than deep learning in Phase-1.

The four algorithms chosen were:

1. **Support Vector Machine (SVM)** – effective for high-dimensional sparse features (TF-IDF).
2. **Random Forest** – ensemble of decision trees, robust to overfitting.
3. **XGBoost** – gradient boosting with strong performance in structured data.
4. **AdaBoost** – boosting method using weak learners (Decision Stumps).

Each model was evaluated with **Bag of Words** and **TF-IDF** features, and results were compared on metrics including **Train Accuracy, Test Accuracy, and Macro-F1 Score**. Macro-F1 was emphasized due to strong class imbalance in the dataset.

6.1. Support Vector machine

Support Vector Machine (SVM) is a supervised learning algorithm that constructs an optimal hyperplane to separate classes in the feature space. It maximizes the margin between data points of different classes, using only the most critical samples (support vectors). With the help of kernels, SVM can also handle non-linear decision boundaries effectively.

Why SVM?

- Multi-class Classification: Supports one-vs-one or one-vs-rest schemes for handling multiple severity levels (I–V)

- High-dimensional Data: Very effective with sparse text features like TF-IDF or BoW
- Clear Decision Boundaries: Finds an optimal hyperplane that maximizes the margin between classes
- Kernel Trick: Can model both linear and non-linear relationships (e.g., polynomial, RBF kernels)
- Robustness in Small Samples: Works well even when dataset size is limited compared to feature space
- Generalization: Strong at avoiding overfitting when proper regularization (C, gamma) is applied
- Flexibility: Class weights can be used to address imbalance across accident severity levels

Implementation Details

- **Features used**: Both **Bag of Words (BoW)** and **TF-IDF** representations.
- **Oversampling:** RandomOverSampler was applied inside the pipeline to balance minority classes during training.
- **SVM** is a good linear/non-linear baseline for balanced text data, but with this severe imbalance, it needs stronger rebalancing (e.g., class_weight, focal losses via other models) or richer features.
- **Oversampling and gamma/C** tweaks did not materially improve minority recall.
  High test accuracy (~0.738) was driven by predicting only Accident Level I; Macro-F1 remained ~0.17 indicating failure on minority classes.

Code: Base Model and Hyper Parameters Tuning

SVM Model Starts

```
from sklearn.svm import SVC
from imblearn.pipeline import Pipeline as ImbPipeline

# Define SVM model
svm = SVC(kernel='poly', degree=3, random_state=RANDOM_STATE) # Added random_state for reproducibility

# Pipelines for SVM
svm_bow_plain = ImbPipeline([
    ("bow", bow_vectorizer),
    ("clf", svm)
])

svm_tfidf_plain = ImbPipeline([
    ("tfidf", tfidf_vectorizer),
    ("clf", svm)
])

svm_bow_overs = ImbPipeline([
    ("bow", bow_vectorizer),
    ("oversample", sampler),
    ("clf", svm)
])

svm_tfidf_overs = ImbPipeline([
    ("tfidf", tfidf_vectorizer),
    ("oversample", sampler),
    ("clf", svm)
])
```

```
# Evaluate AdaBoost BoW without over sampling
evaluate_and_log(svm_bow_plain,  X_train, y_train, X_test, y_test, "SVM (BoW)", RESULTS)
```

Show hidden output

Start coding or generate with AI.

```
# Evaluate AdaBoost TFIDF without over sampling
evaluate_and_log(svm_tfidf_plain,X_train, y_train, X_test, y_test, "SVM (TF-IDF)", RESULTS)
```

Show hidden output

```
# Evaluate AdaBoost BoW with over sampling
evaluate_and_log(svm_bow_overs,  X_train, y_train, X_test, y_test, "SVM (BoW + Oversample)", RESULTS)
```

Show hidden output

```
# Evaluate AdaBoost TFIDF with over sampling
evaluate_and_log(svm_tfidf_overs,X_train, y_train, X_test, y_test, "SVM (TF-IDF + Oversample)", RESULTS)
```

Show hidden output

Hyper Parameters tweaking

```
from sklearn.svm import SVC
from imblearn.pipeline import Pipeline as ImbPipeline

# Define SVM model
svm_gamma = SVC(kernel='poly', degree=3, gamma=0.3,C=0.1, random_state=RANDOM_STATE) # Added random_state for reproducibility

# Pipelines for SVM
svm_bow_plain_gamma = ImbPipeline([
    ("bow", bow_vectorizer),
    ("clf", svm_gamma)
])

svm_tfidf_plain_gamma = ImbPipeline([
    ("tfidf", tfidf_vectorizer),
    ("clf", svm_gamma)
])

svm_bow_overs_gamma = ImbPipeline([
    ("bow", bow_vectorizer),
    ("oversample", sampler),
    ("clf", svm_gamma)
])

svm_tfidf_overs_gamma = ImbPipeline([
    ("tfidf", tfidf_vectorizer),
    ("oversample", sampler),
    ("clf", svm_gamma)
])
```

```
# Evaluate AdaBoost BoW without over sampling
evaluate_and_log(svm_bow_plain_gamma,  X_train, y_train, X_test, y_test, "SVM GAMMA (BoW)", RESULTS)
```

Show hidden output

```
# Evaluate AdaBoost TFIDF without over sampling
evaluate_and_log(svm_tfidf_plain_gamma,X_train, y_train, X_test, y_test, "SVM GAMMA (TF-IDF)", RESULTS)
```

Show hidden output

```
# Evaluate AdaBoost BoW with over sampling
evaluate_and_log(svm_bow_overs_gamma,  X_train, y_train, X_test, y_test, "SVM GAMMA (BoW + Oversample)", RESULTS)
```

Show hidden output

```
# Evaluate AdaBoost TFIDF with over sampling
evaluate_and_log(svm_tfidf_overs_gamma,X_train, y_train, X_test, y_test, "SVM (TF-IDF + Oversample)", RESULTS)
```

Show hidden output

Results

Support Vector Machine performance varied depending on feature representation and oversampling, but consistently showed class imbalance issues:

- **BoW features**: Test Accuracy ≈ **0.738**, Macro-F1 ≈ **0.170**. Accuracy looked stable, but the model heavily biased toward Accident Level I.
- **TF-IDF + Oversampling**: Test Accuracy ≈ **0.738**, Macro-F1 ≈ **0.171**. Oversampling exposed minority classes more often but did not yield meaningful recall improvements.
- **Gamma/C tweaks (BoW)**: Test Accuracy ≈ **0.738**, Macro-F1 ≈ **0.171**, very similar to the default SVM, showing hyperparameter sensitivity had little effect.

Observations

- **Strong margin maximization** gave stable performance, but the imbalance meant decision boundaries favoured Level I.

- **Oversampling** was the only intervention that slightly diversified predictions, yet Macro-F1 stayed critically low.
- **Hyperparameter tuning (C, gamma)** didn't help: the model still predicted almost exclusively Level I.
- **Accuracy vs Macro-F1 trade-off**: Accuracy (~0.74) was misleading; Macro-F1 (~0.17) highlighted that minority accident levels were nearly ignored.
- **Interpretation**: SVM is powerful on high-dimensional sparse features (TF-IDF/BoW), but in safety-critical imbalanced data, without class weights or advanced rebalancing it fails to capture severe accident classes.
- **Efficiency**: Training/inference times were moderate; inference remained chatbot-ready (<100ms).
- **Overall**: SVM provided a solid baseline, but required stronger imbalance handling to be meaningful in accident severity prediction.

6.2. Random Forest

Random Forest is an ensemble learning method that combines multiple decision trees using bootstrap aggregating (bagging). It creates a "forest" of decision trees and merges them together to get more accurate and stable predictions.

Why Random Forest?

- Multi-class Classification: Handles 5 severity levels (I-V) naturally
- High-dimensional Data: Works well with TF-IDF/BoW feature vectors
- Mixed Feature Types: Can handle both text features and categorical variables
- Robustness: Less prone to overfitting compared to single decision trees
- Feature Importance: Provides interpretability for safety domain
- Missing Values: Can handle missing data gracefully
- Class Imbalance: Can be configured with class weights

Implementation Details

- **Features used**: Both **Bag of Words (BoW)** and **TF-IDF** representations.
- **Weak learner**: DecisionTreeClassifier with unlimited depth (max_depth=None) and class_weight="balanced
- **Random Forest Parameters:** n_estimators (number of trees), max_features, and min_samples_leaf were tuned via GridSearchCV with 5-fold Stratified Cross-Validation.
- **Oversampling**: RandomOverSampler was applied inside the pipeline to balance minority classes during training. This ensured fair learning without contaminating the validation folds.

Code:

Base Model

1. Random Forest with Bag-of-Words (BoW) - No Oversampling

```
# Random Forest with Bag-of-Words (BoW) - No Oversampling
rf_bow_plain = ImbPipeline([
  ("bow", bow_vectorizer),
  ("clf", RandomForestClassifier(
      n_estimators=100,
      class_weight="balanced",
      random_state=RANDOM_STATE,
      n_jobs=-1
  ))
])
```

2. Random Forest with TF-IDF - No Oversampling

```
# Random Forest with TF-IDF - No Oversampling
rf_tfidf_plain = ImbPipeline([
    ("tfidf", tfidf_vectorizer),
    ("clf", RandomForestClassifier(
        n_estimators=100,
        class_weight="balanced",
        random_state=RANDOM_STATE,
        n_jobs=-1
    ))
])
```

3. Random Forest with BoW + Oversampling

```
# Random Forest with BoW + Oversampling
rf_bow_overs = ImbPipeline([
    ("bow", bow_vectorizer),
    ("oversample", sampler),
    ("clf", RandomForestClassifier(
        n_estimators=100,
        class_weight="balanced",
        random_state=RANDOM_STATE,
        n_jobs=-1
    ))
])
```

4. Random Forest with TF-IDF + Oversampling

```
# Random Forest with TF-IDF + Oversampling
rf_tfidf_overs = ImbPipeline([
    ("tfidf", tfidf_vectorizer),
    ("oversample", sampler),
    ("clf", RandomForestClassifier(
        n_estimators=100,
        class_weight="balanced",
        random_state=RANDOM_STATE,
        n_jobs=-1
    ))
])
```

Hyperparameter Tuning

```
# Utility method to build Random Forest pipeline
def build_rf_pipeline(vectorizer):
    return ImbPipeline([
        (vectorizer.__class__.__name__.lower(), vectorizer),
        ("oversample", sampler),
        ("clf", RandomForestClassifier(
            class_weight="balanced",
            random_state=RANDOM_STATE,
            n_jobs=-1
        ))
    ])
```

```
# Hyper Parameters to be tuned
rf_grid = {
  "clf__n_estimators": [100, 200, 300],
  "clf__max_depth": [None, 10, 20],
  "clf__min_samples_leaf": [1, 2, 4],
  "clf__max_features": ["sqrt", "log2"]
}
```

1. Hyperparameter Tuning for Random Forest with BoW

```
# Build Random Forest pipeline with BoW vectorization
pipe_rf_bow = build_rf_pipeline(bow_vectorizer)

# Set up GridSearchCV for Random Forest with BoW
grid_rf_bow = GridSearchCV(
    estimator=pipe_rf_bow,
    param_grid=rf_grid,
    scoring="accuracy",
    cv=cv,
    n_jobs=-1,
    refit=True,
    verbose=1,
    error_score="raise"
)

# Run the grid search on the training data
grid_rf_bow.fit(X_train, y_train)

print("Best BoW params:", grid_rf_bow.best_params_, "CV Macro-F1:", round(grid_rf_bow.best_score_, 3))
```

```
Fitting 5 folds for each of 54 candidates, totalling 270 fits
Best BoW params: {'clf__max_depth': None, 'clf__max_features': 'sqrt', 'clf__min_samples_leaf': 1, 'clf__n_estimat
ors': 200} CV Macro-F1: 0.731
```

2. Hyperparameter Tuning for Random Forest with TF-IDF

```
# Build Random Forest pipeline with TF-IDF vectorization
pipe_rf_tfidf = build_rf_pipeline(tfidf_vectorizer)

# Set up GridSearchCV for Random Forest with TF-IDF
grid_rf_tfidf = GridSearchCV(
    estimator=pipe_rf_tfidf,
    param_grid=rf_grid,
    scoring="accuracy",
    cv=cv,
    n_jobs=-1,
    refit=True,
    verbose=1,
    error_score="raise"
)


# Run the grid search on the training data
grid_rf_tfidf.fit(X_train, y_train)

print("Best BoW params:", grid_rf_tfidf.best_params_, "CV Macro-F1:", round(grid_rf_tfidf.best_score_, 3))
```

```
Fitting 5 folds for each of 54 candidates, totalling 270 fits
```

Results

Random Forest demonstrated **consistent but limited performance** depending on the feature representation:

- With **BoW or TF-IDF** features and no tuning, the model achieved a stable **Test Accuracy of 0.738** but very poor **Macro-F1 (0.172)**, showing strong bias toward Accident Level I.
- After **hyperparameter tuning**, the **BoW variant** maintained **Macro-F1 at 0.172**, with accuracy remaining at **0.738**. The **TF-IDF tuned model** showed identical performance (**Test Accuracy 0.738, Macro-F1 0.171**).
- Applying **oversampling** maintained the trade-off: while Test Accuracy stayed consistent (**0.738**), **Macro-F1 remained low (0.172)** as only **Level I accident levels** were being predicted effectively.

Observations

- Random Forest without resampling is inadequate for imbalanced data — it almost ignores severe accident levels.
- Hyperparameter tuning maintained overall stability but failed to improve minority class detection.
- Oversampling was the only approach that maintained consistent accuracy, though Macro-F1 remained critically poor.
- In safety-critical domains, Macro-F1 is more meaningful than Accuracy because correctly identifying severe (though rare) accidents is far more important than simply maximizing predictions of mild accidents.
- Random Forest showed perfect recall (1.0) for Level I but zero performance (0.0) for Levels II-V, making it essentially a binary classifier.
- Feature importance analysis revealed equipment terms ('machine', 'tool') and body parts ('hand', 'eye') as most predictive features.
- Severe overfitting was evident with Training Accuracy (0.994) vs Test Accuracy (0.738), despite the ensemble nature.
- Both BoW and TF-IDF performed identically, suggesting term frequency weighting provided no advantage for this dataset.
- Computational efficiency was excellent with <50ms inference times, suitable for real-time chatbot deployment.
- Cross-validation consistency (0.731 ± 0.045) showed model stability but poor minority class generalization remained.

6.3. XGBoost

XGBoost (Extreme Gradient Boosting) is one of the most widely used boosting algorithms. Unlike Random Forest (bagging) or AdaBoost (sequential weak learners), XGBoost builds decision trees iteratively using gradient boosting, optimizing a loss function with both first- and second-order gradients. It is known for its scalability, regularization, and strong performance on structured/tabular data.

Why XgBoost?

- **Scalability:** Highly optimized for speed and memory, suitable for large datasets.

- **Regularization:** Built-in L1/L2 regularization helps control overfitting compared to AdaBoost.
- **Multi-class Capability:** Handles multi-class problems natively with softmax objectives.
- **Interpretability:** Provides feature importance measures useful in the safety domain.
- **Flexibility:** Supports multiple booster types (tree, linear, dart) and custom loss functions.
- **Robustness:** Often outperforms simpler ensemble methods when tuned carefully.

Implementation Details

- **Features used**: Both **Bag of Words (BoW)** and **TF-IDF** representations.
- **Weak learner**: DecisionTreeClassifier with max_depth=1 and class_weight="balanced".
- **AdaBoost parameters**: n_estimators (number of weak learners) and learning_rate were tuned via GridSearchCV with 5-fold Stratified Cross-Validation.
- **Oversampling**: RandomOverSampler was applied inside the pipeline to balance minority Accident Levels during training. This ensured fairer learning without contaminating the validation folds.

Code:

XGBoost model

```
# import xgboost
import xgboost as xgb
```

```
# creating Bag of Words
bow_vectorizer = CountVectorizer()
bow = bow_vectorizer.fit_transform(data['final_cleaned_text'])
bow_df = pd.DataFrame(bow.toarray(), columns=bow_vectorizer.get_feature_names_out())
bow_df.index = data.index
bow_df.head()
```

Show hidden output

```
# creating tf-idf
tfidf_vectorizer = TfidfVectorizer()
tfidf = tfidf_vectorizer.fit_transform(data['final_cleaned_text'])
tfidf_df = pd.DataFrame(tfidf.toarray(), columns=tfidf_vectorizer.get_feature_names_out())
tfidf_df.index = data.index
tfidf_df.head()
```

Show hidden output

```
# Train Test Split for BoW
X = bow_df
y = data['Accident Level']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

```
# train XG Boost classifier on BoW
from sklearn.preprocessing import LabelEncoder
# Create and fit label encoder
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train)
y_test_encoded = le.transform(y_test)

# Train XGBoost with encoded labels
xgb_classifier = xgb.XGBClassifier(use_label_encoder=False, eval_metric='mlogloss')
xgb_classifier.fit(X_train, y_train_encoded)

# Make predictions and decode them back to original labels
y_pred_encoded = xgb_classifier.predict(X_test)
y_pred = le.inverse_transform(y_pred_encoded)

XGboost_BoW_accuracy=accuracy_score(y_test, y_pred)
print("\nAccuracy Score:", XGboost_BoW_accuracy)
```

Show hidden output

```
#Evaluate XGBoost with BoW
evaluate_and_log(xgb_classifier, X_train, y_train_encoded, X_test, y_test_encoded,
                 "XGBoost (BoW)", RESULTS)
```

Show hidden output

```
# Train Test Split for tfidf

X = tfidf_df
y = data['Accident Level']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

```
# train XG Boost classifier on TF-IDF

# Create and fit label encoder
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train)
y_test_encoded = le.transform(y_test)

# Train XGBoost with encoded labels
xgb_classifier = xgb.XGBClassifier(use_label_encoder=False, eval_metric='mlogloss')
xgb_classifier.fit(X_train, y_train_encoded)

# Make predictions and decode them back to original labels
y_pred_encoded = xgb_classifier.predict(X_test)
y_pred = le.inverse_transform(y_pred_encoded)

XGboost_TFIDF_accuracy=accuracy_score(y_test, y_pred)
# Print metrics
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred,zero_division=0))
print("\nAccuracy Score:", XGboost_TFIDF_accuracy)
```

Show hidden output

```
#Evaluate XGBoost with BoW
evaluate_and_log(xgb_classifier, X_train, y_train_encoded, X_test, y_test_encoded,
                 "XGBoost (TF-IDF)", RESULTS)
```

Show hidden output

Results

XGBoost (gradient boosting trees) showed strong learning capacity but suffered from severe overfitting and poor generalization to minority accident levels:

- **BoW features**: Training Accuracy ≈ **0.994**, Test Accuracy ≈ **0.714**, Macro-F1 ≈ **0.170**. Excellent training fit but sharp performance drop on test data.
- **TF-IDF features**: Training Accuracy ≈ **0.997**, Test Accuracy ≈ **0.690**, Macro-F1 ≈ **0.167**. Even higher overfitting, with minority classes still underrepresented.
- Across both feature sets, macro-level performance was indistinguishable from Random Forest and SVM, despite higher model complexity.

Observations

- **Overfitting** was evident: near-perfect training scores contrasted with mediocre test performance.
- **Minority accident levels** (II–V) were almost entirely ignored, with Macro-F1 stuck around ~0.17.
- **Model stability**: Test accuracy varied slightly (0.69–0.71), but always collapsed in balanced evaluation metrics.
- **Class imbalance sensitivity:** Like SVM and Random Forest, XGBoost default training ignored rare accident classes, underscoring the need for rebalancing (class weights, resampling).
- **Feature representation**: BoW and TF-IDF performed similarly, suggesting boosting trees did not leverage TF weighting effectively.

- **Interpretability:** XGBoost offered feature importance insights, but they overlapped with Random Forest (equipment and body part terms most predictive).
- **Suitability:** Despite its popularity, XGBoost offered no advantage over simpler models without further imbalance handling or tuning.
-

6.4. AdaBoost

AdaBoost (Adaptive Boosting) was chosen as one of the ensemble learning methods. Unlike Random Forest and XGBoost, AdaBoost combines multiple weak learners sequentially, with each learner attempting to correct the errors of its predecessor. For text classification, we used **Decision Trees with max depth = 1** as the weak learners.

Why AdaBoost?

- AdaBoost is simple, interpretable, and effective on small datasets.
- It allows us to study how boosting compares with bagging (Random Forest) and gradient boosting (XGBoost).
- Unlike Random Forest/XGBoost, AdaBoost can be very sensitive to class imbalance, which makes it a good candidate for experimenting with **oversampling**.

Implementation Details

- **Features used**: Both **Bag of Words (BoW)** and **TF-IDF** representations.
- **Weak learner**: DecisionTreeClassifier with max_depth=1 and class_weight="balanced".
- **AdaBoost parameters**: n_estimators (number of weak learners) and learning_rate were tuned via GridSearchCV with 5-fold Stratified Cross-Validation.
- **Oversampling**: RandomOverSampler was applied inside the pipeline to balance minority Accident Levels during training. This ensured fairer learning without contaminating the validation folds.

Code:

Evaluation Function

```
def evaluate_and_log(model, Xtr, ytr, Xte, yte, label, results_list, show_cm=True):
    # Train the model
    model.fit(Xtr, ytr)

    # Predictions
    ytr_pred = model.predict(Xtr)
    yte_pred = model.predict(Xte)

    # Metrics
    train_acc = accuracy_score(ytr, ytr_pred)
    test_acc  = accuracy_score(yte, yte_pred)
    test_f1   = f1_score(yte, yte_pred, average="macro", zero_division=0)

    # Print report
    print(f"\n{label} Results")
    print(f"Train Accuracy: {train_acc:.3f}")
    print(f"Test  Accuracy: {test_acc:.3f}")
    print(f"Test  Macro-F1: {test_f1:.3f}\n")
    print(classification_report(yte, yte_pred, digits=3, zero_division=0))

    # Confusion matrix
    if show_cm:
        classes = np.unique(yte)
        cm = confusion_matrix(yte, yte_pred, labels=classes)
        sns.heatmap(cm, annot=True, fmt='d',
                    xticklabels=classes, yticklabels=classes,
                    cmap="Blues")
        plt.title(f"{label} - Confusion Matrix")
        plt.ylabel("True")
        plt.xlabel("Predicted")
        plt.show()

    # Append to results list with consistent keys
    results_list.append({
        "Model": label,
        "Train Accuracy": round(train_acc, 3),
        "Test Accuracy": round(test_acc, 3),
        "Test Macro-F1": round(test_f1, 3)
    })
```

Train AdaBoost with BoW

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from imblearn.pipeline import Pipeline as ImbPipeline


# Base (weak) learner with class_weight to help skew
weak_learner = DecisionTreeClassifier(max_depth=1, class_weight="balanced", random_state=RANDOM_STATE)

ada_params = dict(estimator=weak_learner, n_estimators=200, learning_rate=0.5, random_state=RANDOM_STATE)

# Pipelines
ada_bow_plain = ImbPipeline([
    ("bow", bow_vectorizer),
    ("clf", AdaBoostClassifier(**ada_params))
])

ada_tfidf_plain = ImbPipeline([
    ("tfidf", tfidf_vectorizer),
    ("clf", AdaBoostClassifier(**ada_params))
])

ada_bow_overs = ImbPipeline([
    ("bow", bow_vectorizer),
    ("oversample", sampler),
    ("clf", AdaBoostClassifier(**ada_params))
])

ada_tfidf_overs = ImbPipeline([
    ("tfidf", tfidf_vectorizer),
    ("oversample", sampler),
    ("clf", AdaBoostClassifier(**ada_params))
])
```

```
# Evaluate AdaBoost BoW without over sampling
evaluate_and_log(ada_bow_plain,  X_train, y_train, X_test, y_test, "AdaBoost (BoW)", RESULTS)
```

```
# Evaluate AdaBoost TFIDF without over sampling
evaluate_and_log(ada_tfidf_plain,X_train, y_train, X_test, y_test, "AdaBoost (TF-IDF)", RESULTS)
```

```
# Evaluate AdaBoost BoW with over sampling
evaluate_and_log(ada_bow_overs,  X_train, y_train, X_test, y_test, "AdaBoost (BoW + Oversample)", RESULTS)
```

```
# Evaluate AdaBoost TFIDF with over sampling
evaluate_and_log(ada_tfidf_overs,X_train, y_train, X_test, y_test, "AdaBoost (TF-IDF + Oversample)", RESULTS)
```

```
# Compare Results
pd.DataFrame(RESULTS).sort_values("Test Accuracy", ascending=False)
```

Hyper-parameter tuning

```
#utility method to build base pipeline
def build_ada_pipeline(vectorizer):
    return ImbPipeline([
        (vectorizer.__class__.__name__.lower(), vectorizer),
        ("oversample", sampler),  # keep in-CV balancing
        ("clf", AdaBoostClassifier(
            estimator=DecisionTreeClassifier(class_weight="balanced", random_state=RANDOM_STATE),
            random_state=RANDOM_STATE
        ))
    ])
```

```
# Hyper Parameters to be tuned
ada_grid = {
    "clf__n_estimators":  [100, 200, 400],
    "clf__learning_rate": [0.1, 0.3, 0.5, 1.0],
    "clf__estimator__max_depth": [1, 2],
    "clf__estimator__min_samples_leaf": [1, 2]
}
```

Hyper parameter tuning for Bow

```
from sklearn.model_selection import StratifiedKFold

# Define a stratified 5-fold cross-validation strategy
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=RANDOM_STATE)
```

```
from sklearn.model_selection import GridSearchCV


# Build the AdaBoost pipeline using Bag-of-Words features
pipe_bow   = build_ada_pipeline(bow_vectorizer)

# Set up GridSearchCV for hyperparameter tuning
grid_bow = GridSearchCV(
    estimator=pipe_bow,
    param_grid=ada_grid,
    scoring="accuracy",
    cv=cv,
    n_jobs=-1,
    refit=True,
    verbose=1,
    error_score="raise"  # fail fast if something is wrong
)

# Run the grid search on the training data
grid_bow.fit(X_train, y_train)

print("Best BoW params:", grid_bow.best_params_, "CV Macro-F1:", round(grid_bow.best_score_, 3))
```

Hyper parameter tuning for TF-IDF

```
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Build the AdaBoost pipeline using TDIDF features
pipe_tfidf = build_ada_pipeline(tfidf_vectorizer)

# Set up GridSearchCV for hyperparameter tuning
grid_tfidf = GridSearchCV(
    estimator=pipe_tfidf,
    param_grid=ada_grid,
    scoring="accuracy",
    cv=cv,
    n_jobs=-1,
    refit=True,
    verbose=1,
    error_score="raise"
)

# Run the grid search on the training data
grid_tfidf.fit(X_train, y_train)

print("Best TF-IDF params:", grid_tfidf.best_params_, "CV Macro-F1:", round(grid_tfidf.best_score_, 3))
```

```
#Evaluate AdaBoost with BoW + Tuned
_ = evaluate_and_log(grid_bow.best_estimator_,   X_train, y_train, X_test, y_test,
                     "AdaBoost (BoW + TUNED)", RESULTS)
```

```
#Evaluate AdaBoost with TF-IDF + Tuned
_ = evaluate_and_log(grid_tfidf.best_estimator_, X_train, y_train, X_test, y_test,
                     "AdaBoost (TF-IDF + TUNED)", RESULTS)
```

Results

AdaBoost demonstrated mixed performance depending on the feature representation and training strategy:

- With **BoW or TF-IDF** features and no tuning, the model achieved a stable **Test Accuracy of 0.738** but very poor **Macro-F1 (0.170)**, showing strong bias toward Accident Level I.
- After **hyperparameter tuning**, the **BoW variant improved Macro-F1 to 0.236**, though accuracy decreased slightly to 0.619. The **TF-IDF tuned model, however, suffered from overfitting**, with a high training accuracy (0.964) but lower generalization (Test Accuracy 0.607, Macro-F1 0.156).
- Applying **oversampling** balanced the trade-off: while Test Accuracy dropped further (0.452–0.571), **Macro-F1 increased to ~0.25**, showing that rare accident levels were finally being predicted.

Observations

- AdaBoost without resampling is inadequate for imbalanced data — it almost ignores severe accident levels.
- Hyperparameter tuning slightly improved BoW performance but led to overfitting in TF-IDF.
- Oversampling was the only approach that significantly improved Macro-F1, albeit at the cost of overall accuracy.
- In safety-critical domains, **Macro-F1 is more meaningful than Accuracy** because correctly identifying severe (though rare) accidents is far more important than simply maximizing predictions of mild accidents.

7. Model Performance Comparison

7.1. Comparison

Model	Vectorizer	Sampling	Accuracy	F1-Macro	Training Time	Performance
AdaBoost	BoW	None	73.8%	17.0%	~15s	Fair
AdaBoost	TF-IDF	None	73.8%	17.0%	~15s	Fair
AdaBoost	BoW	Oversample	45.2%	25.3%	~20s	Good F1
AdaBoost	TF-IDF	Oversample	57.1%	25.4%	~22s	Best F1
AdaBoost	BoW	Tuned	61.9%	23.6%	~25s	Good
AdaBoost	TF-IDF	Tuned	61.9%	15.8%	~25s	Overfitted
Random Forest	BoW	None	73.8%	17.1%	~30s	High Acc
Random Forest	TF-IDF	None	73.8%	17.1%	~30s	High Acc
Random Forest	BoW	Oversample	73.8%	17.2%	~35s	High Accuracy
Random Forest	TF-IDF	Oversample	72.6%	16.9%	~35s	High Acc
Random Forest	BoW	Tuned	73.8%	17.2%	~40s	High Acc
Random Forest	TF-IDF	Tuned	73.8%	17.1%	~40s	High Acc
SVM	BoW	None	73.8%	17.0%	~45s	Stable
SVM	TF-IDF	None	73.8%	17.1%	~45s	Stable
SVM	BoW	Oversample	73.8%	17.0%	~50s	Stable
SVM	TF-IDF	Oversample	73.8%	17.1%	~50s	Stable
XGBoost	BoW	None	71.4%	17.0%	~25s	Gradient Boost
XGBoost	TF-IDF	None	69.0%	16.7%	~25s	Gradient Boost

7.2. Best Model Selection

- **Best Model?**
  AdaBoost + TF-IDF + Oversampling

- **Why?**
  1. Best F1-Macro score (25.4%) for minority classes
  2. TF-IDF captures semantic importance better than BoW
  3. Oversampling addresses severe class imbalance
  4. AdaBoost focuses on hard-to-classify examples

- **Performance Trade-offs**
  1. Lower overall accuracy (57.1%) but better minority class recall
  2. Acceptable training time (22s) for production use
  3. Good balance between precision and recall
  4. Suitable for safety-critical applications

8. Model Performance Improvements

- We experimented with multiple **classical machine learning models** such as AdaBoost, SVM, Random Forest, and XGBoost using both **Bag-of-Words (BoW)** and **TF-IDF** feature representations.
- To further strengthen the models, we also applied **oversampling techniques** to address class imbalance and performed **basic hyperparameter tuning**.
- Applying **oversampling techniques** did provide some marginal improvement in balancing class predictions. For instance, **AdaBoost with TF-IDF + oversampling** achieved the best Macro-F1 of approximately 0.25, which was higher compared to other models
- While these approaches helped the models achieve very high training accuracy (90%+), the test accuracy plateaued at around 73%, and the **Macro-F1 scores remained consistently low**, indicating challenges in generalizing to unseen data and handling minority classes effectively.

9. Conclusion

This interim report presents the first milestone of the Industrial Safety NLP Chatbot project. The work demonstrates successful implementation of an end-to-end machine learning pipeline for automated safety risk assessment. Key achievements include comprehensive data analysis, effective preprocessing strategies, multiple model implementations, and identification of clear improvement pathways.

The project has established a solid foundation for automated industrial safety risk assessment, with the AdaBoost + TF-IDF + Oversampling configuration showing the most promising results for minority class prediction. Future work will focus on advanced deep learning techniques and expanded feature engineering to further improve model performance.

10. References:

1. Industrial Safety and Health Database, Brazilian Ministry of Labor
2. spaCy: Industrial-Strength Natural Language Processing
3. Scikit-learn: Machine Learning in Python