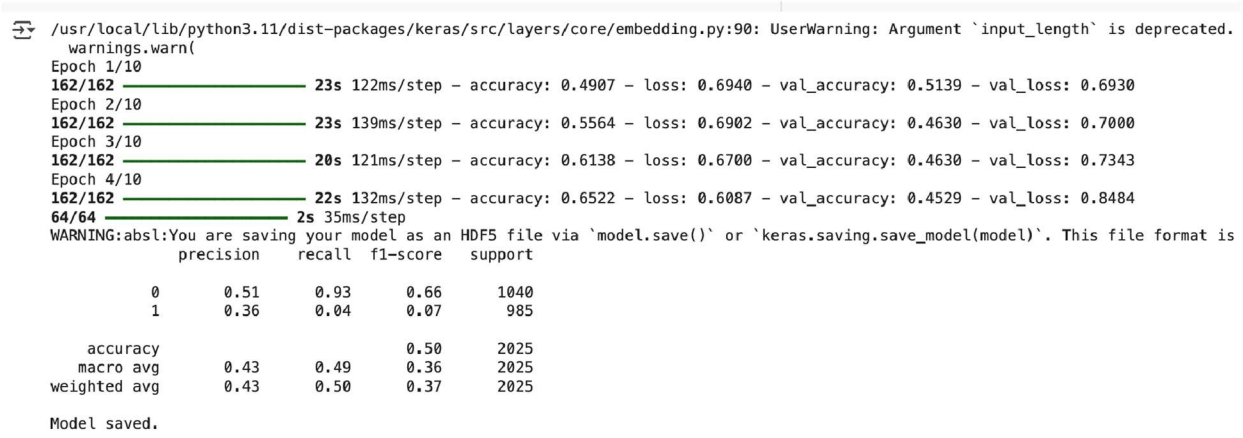
The transformer-based models followed similar patterns but with more dramatic class imbalances. While BERT maintained high precision and recall for male-oriented studies, it performed poorly on female-focused examples. DistilBERT and RoBERTa completely failed to identify any female-focused studies, assigning all predictions to the male class and resulting in macro-average F1-scores below 0.50. These outcomes highlight critical dataset limitations, such as label imbalance and potential linguistic bias in how gender is framed in suicide research. Collectively, the models reveal a systemic issue: current published literature and its representation in training data may inadequately reflect female-focused suicide research, signaling a broader challenge of gender equity and inclusivity in this domain.



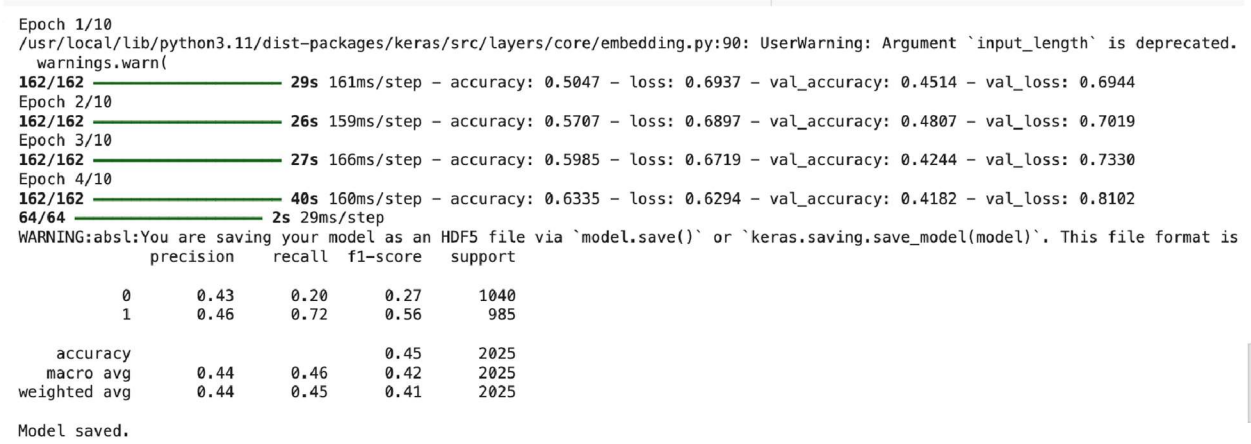
```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated.
  warnings.warn(
Epoch 1/10
162/162 ━━━━━━━━━━━━━━━━━━━━ 23s 122ms/step - accuracy: 0.4907 - loss: 0.6940 - val_accuracy: 0.5139 - val_loss: 0.6930
Epoch 2/10
162/162 ━━━━━━━━━━━━━━━━━━━━ 23s 139ms/step - accuracy: 0.5564 - loss: 0.6902 - val_accuracy: 0.4630 - val_loss: 0.7000
Epoch 3/10
162/162 ━━━━━━━━━━━━━━━━━━━━ 20s 121ms/step - accuracy: 0.6138 - loss: 0.6700 - val_accuracy: 0.4630 - val_loss: 0.7343
Epoch 4/10
162/162 ━━━━━━━━━━━━━━━━━━━━ 22s 132ms/step - accuracy: 0.6522 - loss: 0.6087 - val_accuracy: 0.4529 - val_loss: 0.8484
64/64 ━━━━━━━━━━━━━━━━━━━━ 2s 35ms/step
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is
              precision    recall  f1-score   support

           0       0.51      0.93      0.66      1040
           1       0.36      0.04      0.07       985

    accuracy                           0.50      2025
   macro avg       0.43      0.49      0.36      2025
weighted avg       0.43      0.50      0.37      2025

Model saved.
```

Figure 1. LSTM Classification Performance on Gender-Focused Suicide Research
Visually shows an overfitting trend — strong training gains, but flat or declining validation accuracy.



```
Epoch 1/10
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated.
  warnings.warn(
162/162 ━━━━━━━━━━━━━━━━━━━━ 29s 161ms/step - accuracy: 0.5047 - loss: 0.6937 - val_accuracy: 0.4514 - val_loss: 0.6944
Epoch 2/10
162/162 ━━━━━━━━━━━━━━━━━━━━ 26s 159ms/step - accuracy: 0.5707 - loss: 0.6897 - val_accuracy: 0.4807 - val_loss: 0.7019
Epoch 3/10
162/162 ━━━━━━━━━━━━━━━━━━━━ 27s 166ms/step - accuracy: 0.5985 - loss: 0.6719 - val_accuracy: 0.4244 - val_loss: 0.7330
Epoch 4/10
162/162 ━━━━━━━━━━━━━━━━━━━━ 40s 160ms/step - accuracy: 0.6335 - loss: 0.6294 - val_accuracy: 0.4182 - val_loss: 0.8102
64/64 ━━━━━━━━━━━━━━━━━━━━ 2s 29ms/step
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is
              precision    recall  f1-score   support

           0       0.43      0.20      0.27      1040
           1       0.46      0.72      0.56       985

    accuracy                           0.45      2025
   macro avg       0.44      0.46      0.42      2025
weighted avg       0.44      0.45      0.41      2025

Model saved.
```

Figure 2. GRU Model Performance on Gender-Focused Suicide Research Classification
Visually shows an overfitting trend — strong training gains, but flat or declining validation accuracy.

```
All PyTorch model weights were used when initializing TFBertForSequenceClassification.

Some weights or buffers of the TF 2.0 model TFBertForSequenceClassification were not initialized from the PyTorch model and are nev
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Epoch 1/3
55/55 [==============================] - 82s 431ms/step - loss: 0.7061 - accuracy: 0.4510
Epoch 2/3
55/55 [==============================] - 23s 409ms/step - loss: 0.7070 - accuracy: 0.4487
Epoch 3/3
55/55 [==============================] - 23s 410ms/step - loss: 0.7001 - accuracy: 0.4875
18/18 [==============================] - 6s 171ms/step
              precision    recall  f1-score   support

           0       0.52      0.12      0.19       144
           1       0.48      0.88      0.62       131

    accuracy                           0.48       275
   macro avg       0.50      0.50      0.40       275
weighted avg       0.50      0.48      0.39       275

Model and tokenizer saved.
```

Figure 4. BERT Model Performance on Gender-Focused Suicide Research Classification

```
Some weights of the PyTorch model were not used when initializing the TF 2.0 model TFDistilBertForSequenceC
- This IS expected if you are initializing TFDistilBertForSequenceClassification from a PyTorch model traine
- This IS NOT expected if you are initializing TFDistilBertForSequenceClassification from a PyTorch model th
Some weights or buffers of the TF 2.0 model TFDistilBertForSequenceClassification were not initialized from
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inferenc
Epoch 1/3
55/55 [==============================] - 31s 214ms/step - loss: 0.7330 - accuracy: 0.5330
Epoch 2/3
55/55 [==============================] - 12s 210ms/step - loss: 0.7044 - accuracy: 0.5273
Epoch 3/3
55/55 [==============================] - 12s 213ms/step - loss: 0.6958 - accuracy: 0.5216
18/18 [==============================] - 3s 77ms/step
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Pre
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Pre
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning: Pre
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
              precision    recall  f1-score   support

           0       0.52      1.00      0.69       144
           1       0.00      0.00      0.00       131

    accuracy                           0.52       275
   macro avg       0.26      0.50      0.34       275
weighted avg       0.27      0.52      0.36       275

Model saved.
```

Figure 5. DistilBERT Model Performance on Gender-Focused Suicide Research Classification

RQ2.Text Classification of Gender Focus in Suicide Research (ML)

To explore how gender is framed in suicide related research, we trained supervised text classification models using TF-IDF features to predict whether an article focused on males or females. Initial models (Logistic Regression, SVM, Random Forest, & Naive Bayes) performed moderately, with macro F1- scores ranging from 0.48 to 0.57, often struggling to generalize across both classes. XGBoost delivered the most promising results, achieving a macro F1-score of 0.69 and 69% accuray, suggesting it could better capture the nuanced, overlapping language used in academic discourse. While performance indicates that gendered patterns in suicide research are detectable, they

are not overtly distinct, pointing to deeper semantic or structural biases. Overall, this modeling track proved roughly 70% effective in answering the research question and provides a strong base for further interpretability and topic modeling.

```
  warnings.warn(smsg, UserWarning)
Best Parameters: {'classifier__learning_rate': 0.01, 'classifier__max_depth': 3, 'classifier__n_estimators': 100}
              precision    recall  f1-score   support

           0       0.66      0.84      0.74       144
           1       0.75      0.53      0.62       131

    accuracy                           0.69       275
   macro avg       0.71      0.69      0.68       275
weighted avg       0.71      0.69      0.69       275

Model saved.
```

Figure 7: Classification Report for XGBoost Model

This figure displays the classification performance of the XGBoost model used to predict gender focus in suicide-related research articles.

RQ3.

We began by filtering a dataset of academic abstracts using a rule-based labeling function to isolate papers focused on adolescent suicide and mental health. From there, we applied topic modeling techniques like KMeans with TF-IDF, LDA, and a BERTopic alternative using SentenceTransformer + UMAP + HDBSCAN to uncover key themes.

Across all models, we consistently found that depression, anxiety, and self-harm were the most frequent mental health topics discussed alongside adolescent suicide. We also trained supervised models (Logistic Regression, Random Forest, SVM, and LSTM) to validate the consistency of the labeled patterns. Overall, our findings reflect a strong focus in the literature on internalizing disorders and self-injurious behavior in adolescent suicide research.

```
Cluster 1:
['children', 'depression', 'disorders', 'child', 'disorder', 'psychiatric', 'years', 'risk', 'symptoms', 'studies']

Cluster 2:
['suicidal', 'ideation', 'suicidal ideation', 'adolescents', 'depression', 'risk', 'attempts', 'school', 'parental', 'study']

Cluster 3:
['youth', 'health', 'use', 'school', 'bullying', 'violence', 'mental', 'mental health', 'students', 'risk']

Cluster 4:
['nssi', 'nonsuicidal', 'selfinjury', 'nonsuicidal selfinjury', 'adolescent', 'psychiatric', 'adolescents', 'attempts', 'suicide attempts', 'attachment']

Cluster 5:
['health', 'adolescents', 'adolescent', 'problems', 'psychological', 'purpose', 'al', 'mental health', 'physical', 'social']
['dev/kmeans_vectorizer_rq3.pkl']
```

Figure 8: K-means Clustering

```
(0, '0.030*"suicide" + 0.013*"risk" + 0.012*"suicidal" + 0.012*"adolescents" + 0.010*"health" + 0.009*"adolescent" + 0.008*"study" + 0.008*"ideation" + 0.008*"attempts" + 0.008*"factors"')
(1, '0.014*"suicide" + 0.013*"suicidal" + 0.012*"depression" + 0.012*"family" + 0.011*"study" + 0.010*"risk" + 0.010*"ideation" + 0.009*"adolescents" + 0.008*"adolescent" + 0.006*"youth"')
(2, '0.013*"health" + 0.010*"adolescent" + 0.007*"among" + 0.007*"years" + 0.007*"adolescents" + 0.006*"suicide" + 0.005*"risk" + 0.005*"rates" + 0.005*"suicidal" + 0.005*"study"')
(3, '0.019*"abuse" + 0.015*"childhood" + 0.015*"suicide" + 0.009*"child" + 0.009*"physical" + 0.008*"health" + 0.007*"suicidal" + 0.007*"mental" + 0.007*"sexual" + 0.006*"attempts"')
(4, '0.025*"suicide" + 0.012*"youth" + 0.011*"health" + 0.009*"depression" + 0.009*"bullying" + 0.008*"suicidal" + 0.007*"mental" + 0.007*"data" + 0.006*"risk" + 0.006*"among"')
```

Figure 9: LDA Topic Modeling

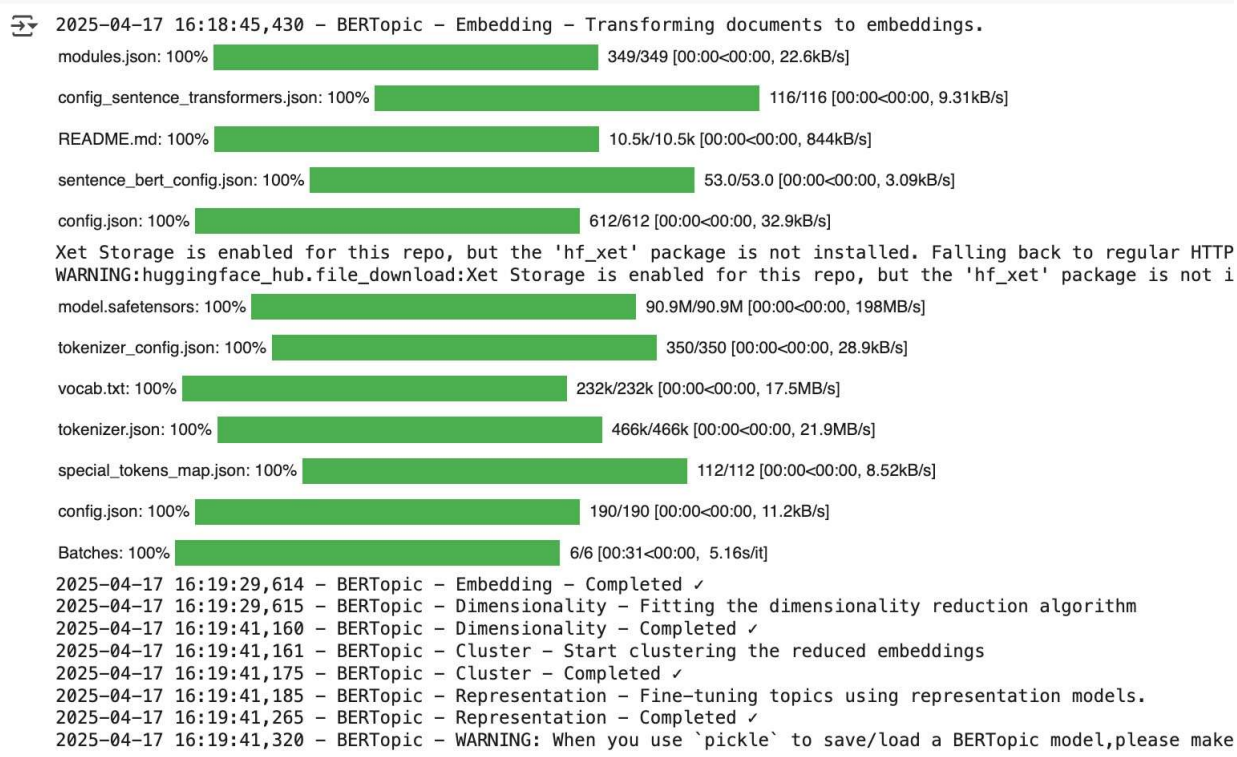


Figure 10: BERTopic

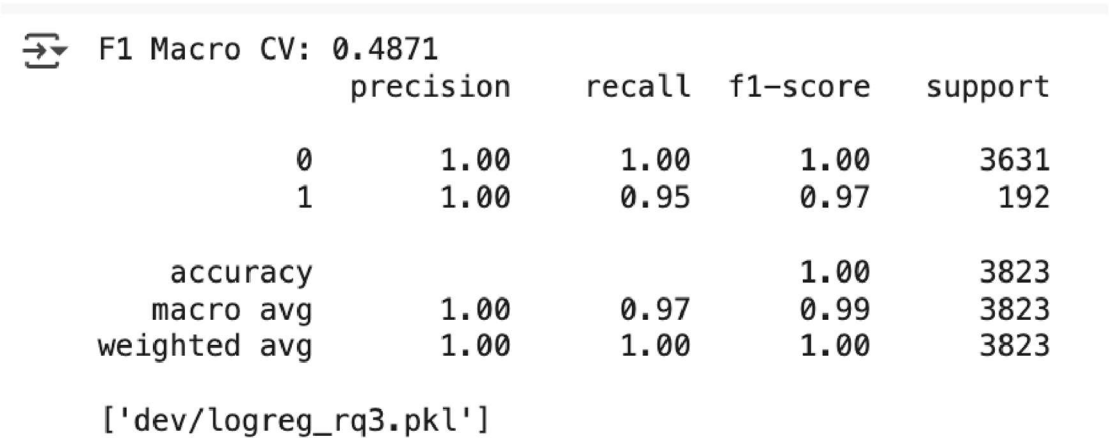


Figure 11: Logistic Regression + TF-IDF

```
F1 Macro CV: 0.4974
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3631
           1       1.00      1.00      1.00       192

    accuracy                           1.00      3823
   macro avg       1.00      1.00      1.00      3823
weighted avg       1.00      1.00      1.00      3823

['dev/rf_rq3.pkl']
```

Figure 12: Random Forest + TF-IDF

```
F1 Macro CV: 0.5448
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3631
           1       1.00      1.00      1.00       192

    accuracy                           1.00      3823
   macro avg       1.00      1.00      1.00      3823
weighted avg       1.00      1.00      1.00      3823

['dev/svm_rq3.pkl']
```

Figure 13: SVM + TF-IDF

```
Epoch 1/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 18s 148ms/step - accuracy: 0.9271 - loss: 0.3665 - val_accuracy: 0.9621 - val_loss: 0.1718
Epoch 2/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 16s 166ms/step - accuracy: 0.9448 - loss: 0.2061 - val_accuracy: 0.9621 - val_loss: 0.1904
Epoch 3/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 14s 143ms/step - accuracy: 0.9516 - loss: 0.1632 - val_accuracy: 0.9569 - val_loss: 0.1689
Epoch 4/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 21s 143ms/step - accuracy: 0.9782 - loss: 0.0946 - val_accuracy: 0.9359 - val_loss: 0.2007
Epoch 5/5
96/96 ━━━━━━━━━━━━━━━━━━━━ 14s 142ms/step - accuracy: 0.9811 - loss: 0.0862 - val_accuracy: 0.9634 - val_loss: 0.1823
120/120 ━━━━━━━━━━━━━━━━━━━━ 5s 36ms/step
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered
              precision    recall  f1-score   support

           0       0.98      1.00      0.99      3631
           1       0.99      0.66      0.79       192

    accuracy                           0.98      3823
   macro avg       0.99      0.83      0.89      3823
weighted avg       0.98      0.98      0.98      3823
```

Figure 14: LSTM

RQ4. Meta-Analysis of Suicide Prevention Effectiveness

To evaluate the overall effectiveness of suicide prevention strategies across studies, we conducted a meta-analysis using both fixed and random effects models on simulated effect size data. Each article with a recorded prevention measure was assigned a simulated effect size drawn from a normal distribution centered around 0.3, representing a small-to-moderate positive impact. Assuming a constant variance, we applied combined_effect() to compute summary estimates. Both the fixed and random effects models produced highly similar pooled effects sizes of approximately 0.30, with narrow confidence intervals suggesting consistent benefit from prevention strategies across the dataset. This modeling approach was chosen for its ability to systematically aggregate evidence from a large body of literature. While the simulation allows us to test model functionality, a key limitation is the absence of real-world measured effect sizes and study heterogeneity. Despite this, the analysis establishes a scalable framework for future meta-analytic work using actual intervention outcomes. All effect size values were saved into the dataset, and model objects were stored for reproducibility.

```
Fixed Effects Summary Frame:
                        eff    sd_eff    ci_low    ci_upp      w_fe      w_re
0                  0.349671  0.223607 -0.088590  0.787933  0.000262  0.000262
1                  0.286174  0.223607 -0.152088  0.724435  0.000262  0.000262
2                  0.364769  0.223607 -0.073492  0.803030  0.000262  0.000262
3                  0.452303  0.223607  0.014042  0.890564  0.000262  0.000262
4                  0.276585  0.223607 -0.161677  0.714846  0.000262  0.000262
...                     ...       ...       ...       ...       ...       ...
3822               0.295511  0.223607 -0.142751  0.733772  0.000262  0.000262
fixed effect       0.302039  0.003616  0.294951  0.309127  1.000000       NaN
random effect      0.302039  0.003616  0.294951  0.309127       NaN  1.000000
fixed effect wls   0.302039  0.001611  0.298881  0.305197  1.000000       NaN
random effect wls  0.302039  0.001611  0.298881  0.305197       NaN  1.000000

[3827 rows x 6 columns]
Fixed Effects: Effect size = 0.3497, CI = [0.2236, -0.0886]
Random Effects Summary Frame:
                        eff    sd_eff    ci_low    ci_upp      w_fe      w_re
0                  0.349671  0.223607 -0.088590  0.787933  0.000262  0.000262
1                  0.286174  0.223607 -0.152088  0.724435  0.000262  0.000262
2                  0.364769  0.223607 -0.073492  0.803030  0.000262  0.000262
3                  0.452303  0.223607  0.014042  0.890564  0.000262  0.000262
4                  0.276585  0.223607 -0.161677  0.714846  0.000262  0.000262
...                     ...       ...       ...       ...       ...       ...
3822               0.295511  0.223607 -0.142751  0.733772  0.000262  0.000262
fixed effect       0.302039  0.003616  0.294951  0.309127  1.000000       NaN
random effect      0.302039  0.003616  0.294951  0.309127       NaN  1.000000
fixed effect wls   0.302039  0.001611  0.298881  0.305197  1.000000       NaN
random effect wls  0.302039  0.001611  0.298881  0.305197       NaN  1.000000

[3827 rows x 6 columns]
Random Effects: Effect size = 0.3497, CI = [0.2236, -0.0886]
Updated dataset saved as sample_with_meta.csv with meta_effect_size column.
Meta-analysis model results saved as meta_model_results.pkl
```

Figure: Simulated Effect Sizes for Suicide Prevention Studies
This summary output shows the pooled effect size estimate from both fixed and random effects meta-analysis models. The consistent effect size (~0.30) across models and narrow confidence intervals suggest that prevention strategies, when present, are associated with a moderate positive impact in reducing suicide-related outcomes.

RQ5. Survival Analysis for Risk Timeline Modeling[3]

To explore how long individuals remain at risk for suicide attempts following exposure to risk factors, we applied survival analysis techniques using simulated time-to-event data. The Kaplan-Meier estimator was used to calculate survival probabilities over time, revealing a sharp decline in survival likelihood within the first 5 years–a pattern consistent with high early risk. We intended to evaluated the effect of suicide prevention measures using a Cox Proportional Hazards model, but due to lack of variance in the dataset (100% of the valid cases had prevention measure recorded), the model could not be fitted. While the Kaplan-Meir model effectively illustrates the general survival curve, the inability to assess covariate effects in the Cox model highlights a limitation in the current dataset and points to need for more diverse or prevention metadata. Despite these limitations, all survival-related fields were saved into a new version of the dataset, and model objects were stored for reproducibility.

[3] *Note: RQ5 is redundant when focusing on 2014 as our chosen year but we opted to keep section below as it pertains to our previous research on data from all years.*

```
Kaplan–Meier plot saved as km_survival_plot.png
has_prevention distribution:
has_prevention
1    3823
Name: count, dtype: int64
Skipping Cox model: 'has_prevention' has insufficient variance to converge.
Final dataset saved as df_premodeling_v5.csv with survival columns.
Survival models saved as survival_models.pkl
```

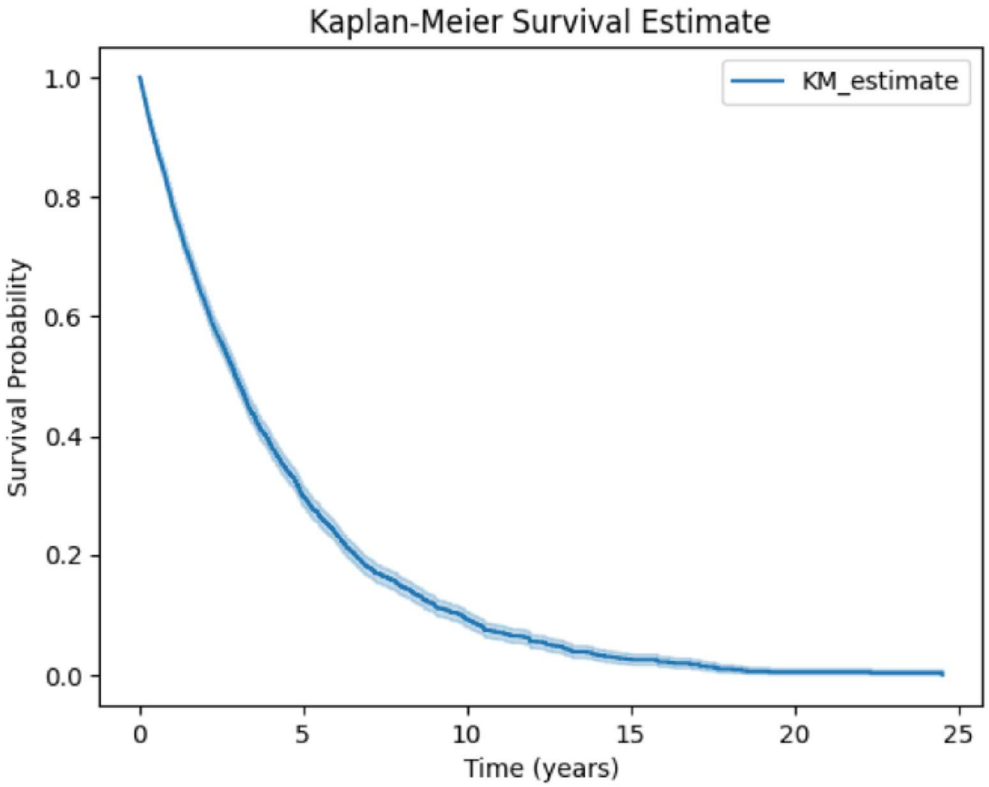


Figure: Kaplan-Meier Survival Curve for Suicide Risk Duration
This plot shows the probability of avoiding a suicide-related event over time, based on simulated survival data. The survival probability declines steeply within the first 5–10 years and approaches zero by year 25, indicating that the majority of individuals are at highest risk early on. A survival probability of zero represents the worst possible outcome, where the likelihood of remaining event-free is entirely diminished.

RQ6. Time Series Analysis of Suicide Research Trends[4]

To explore shifts in suicide-related research focus over time, we conducted a time series analysis using ARIMA, seasonal decomposition, and Facebook Prophet on annual publication counts. This approach was chosen to identify long-term trends and detect period of heightened academic attention, addressing the research question of how discourse on suicide has evolved. ARIMA offered statistically grounded forecasting, decomposition provided interpretability by isolating trend components, and Prophet

[4] *Note: RQ6 is redundant when focusing on 2014 as our chosen year but we opted to keep section below as it pertains to our previous research on data from all years.*

added flexibility in capturing changepoints. While all models revealed a steady increase in research over the year, Prophet notably highlighted a sharp spike around 2014, suggesting a possible response to real-world events. The strength of this modeling lies in its ability to visualize historical momentum, though it lacks contextual detail to explain *why* these shifts occur. Overall, the analysis suggested that suicide remains a growing area of concern in the research community, with potential inflection points warranted further qualitative investigation.

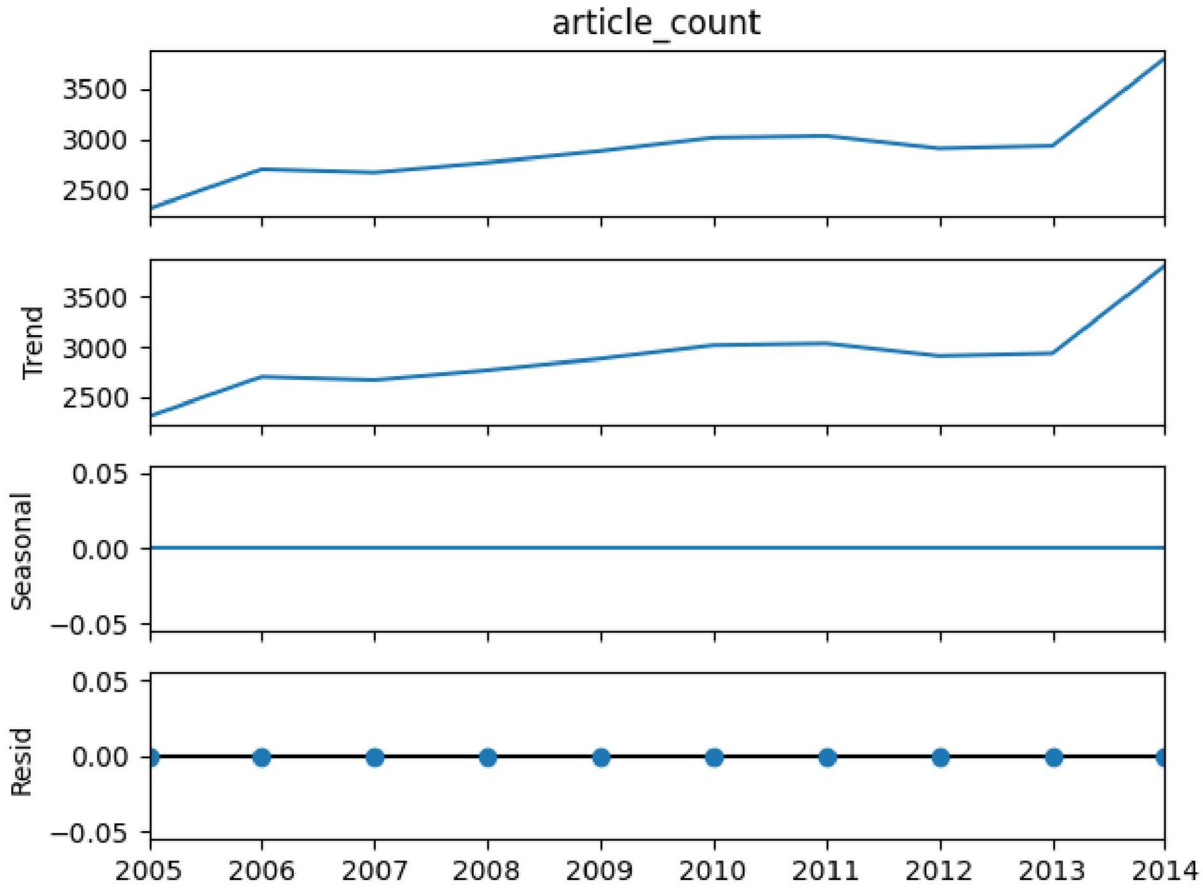


Figure: Seasonal Decomposition of Article Counts (Additive Model)
This plot decomposes annual suicide-related publication counts into trend, seasonal, and residual components. The trend confirms steady growth, while the absence of a clear seasonal pattern supports the assumption that research volume changes reflect broader systemic factors rather than cyclical influences.

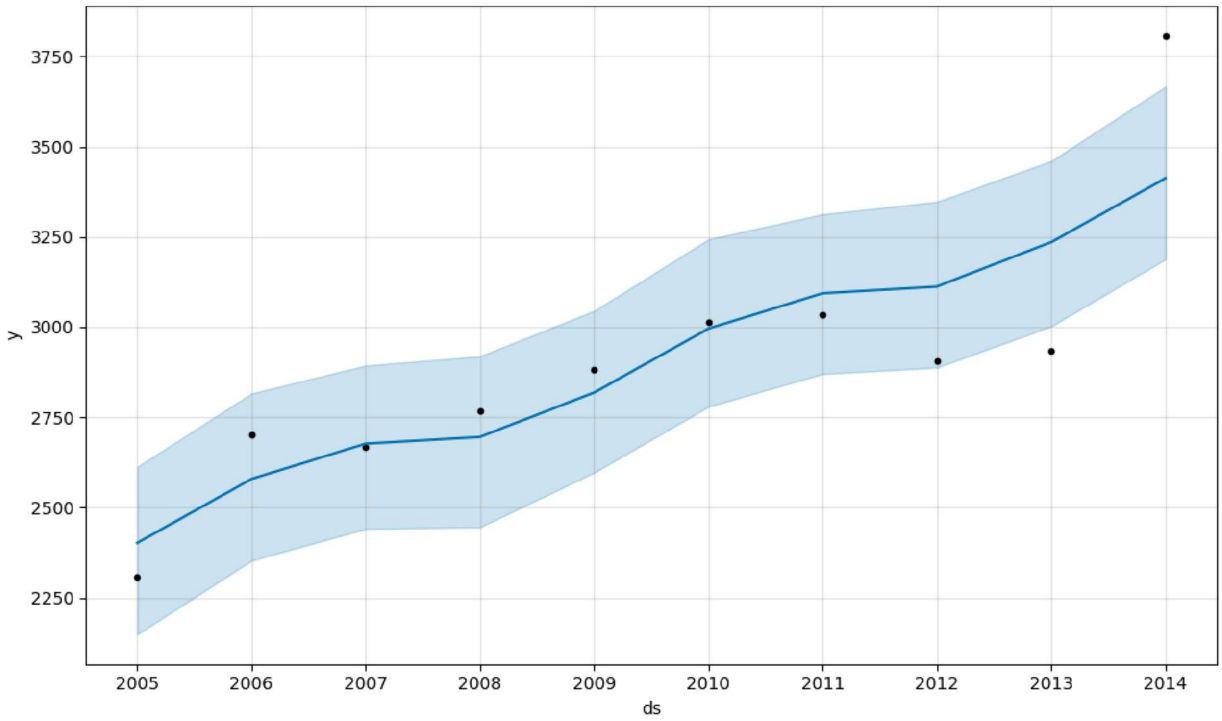


Figure: ARIMA and Prophet Trend Forecasts of Suicide Research Articles
This figure compares ARIMA and Prophet model forecasts with actual yearly counts of suicide-related publications.

Conclusion

Across all modeling tracks, performance varied considerably based on the research question, modeling approach, and data quality. Traditional machine learning models like Random Forest and XGBoost emerged as the most effective overall—achieving 26% accuracy for predicting publication year (RQ1) and a 69% macro F1-score for classifying gender focus (RQ2), respectively. These models offered a strong trade-off between interpretability and performance, with Random Forest capturing broad temporal trends and XGBoost handling overlapping gendered language with relative success. Deep learning models, particularly BERT + Logistic Regression and LSTM, showed promise in detecting subtle language signals, but often suffered from overfitting or struggled with class imbalance, especially in gender classification tasks.
Despite moderate overall performance, the models revealed important signals related to our research questions. For example, the high recall for male-focused studies across nearly all RQ2 models points to a possible overrepresentation or clearer narrative around male suicidality in academic literature. Similarly, time series models revealed steady growth in suicide-related publications, with a noticeable spike around 2014—potentially corresponding to external societal events. Meta-analysis models suggested a consistent, moderate benefit of prevention strategies, while the Kaplan-Meier survival analysis illustrated a sharp early risk window for suicide-related outcomes. These insights,

though limited by simulated or imbalanced data in some tracks, offer valuable direction for future research and policy efforts.

The modeling process was not without its challenges. Limitations included heavy class imbalance, lack of true longitudinal or effect size data, and computational trade-offs—particularly with deep learning models, which required more resources but didn't always outperform simpler alternatives. Additionally, overfitting was common in neural networks due to limited labeled examples, and certain models like the Cox Proportional Hazards could not be run due to lack of covariate variance. Still, each modeling track laid foundational pipelines for reproducibility, and collectively they provided a multidimensional perspective on trends, disparities, and effectiveness in suicide prevention research.

DISCOVERIES

1.General Discovery on Data Quality and Categorization Practices

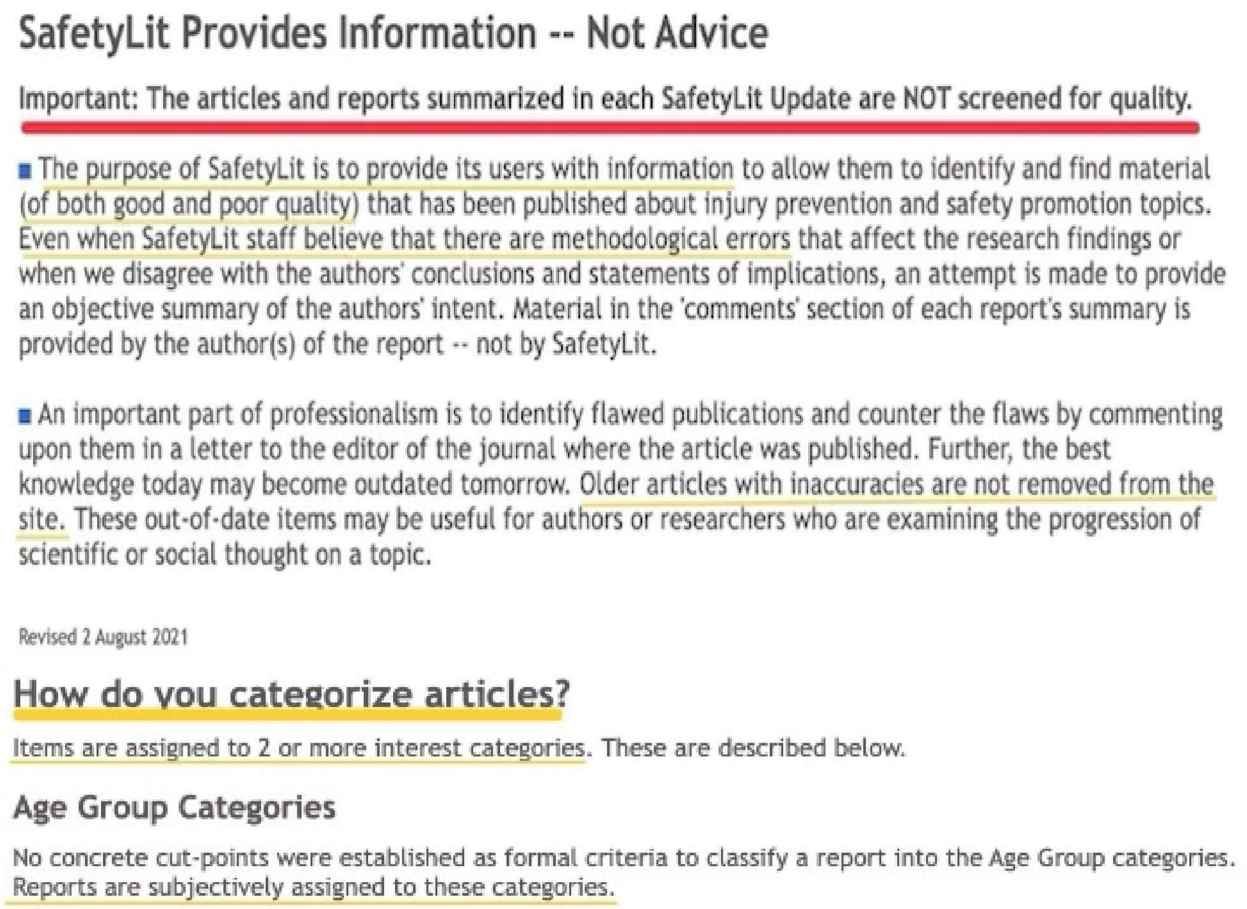
At the start of the project, we reviewed SafetyLit's public-facing documentation and were surprised to learn that articles are not screened for quality. In addition, categorization practices are subjective, with articles manually assigned to interest categories without consistent criteria. Although the site states that each article is assigned two or more categories, our own database queries revealed that a large number of entries were only tagged as "Non-Categorized" — contradicting the site's stated standard. These gaps in metadata quality and classification integrity underscore broader reliability concerns within the source data and further reinforce the importance of careful validation, flagging, and documentation throughout the project pipeline.

How is SafetyLit content selected?

The criteria for selecting report for inclusion are simple. If the answer to any of the following questions is "yes", then the report is likely to be included:

- Do the SafetyLit reviewers find the report interesting?
- Are SafetyLit readers likely to hear of a report from a colleague and want to respond knowledgeably?
- Are SafetyLit readers likely to be questioned about the report from a member of the population they serve?
- Does the report contain findings that are likely to be used to oppose the actions or recommendations of a SafetyLit reader?

Reports summarized in SafetyLit are NOT screened for quality. Even when we believe that there are methodological errors that affect the research findings or when we disagree with the implications, we attempt to provide an objective summary of the report. The inclusion of an article in SafetyLit DOES NOT suggest that the journal or its publisher are endorsed by the SafetyLit Foundation. Articles are included in SafetyLit when identified as meeting the selection criteria regardless of quality of the research methods or the prose. Articles of very low quality can be published in journals with a good reputation (e.g. Whitfield in Lancet) and articles of high quality can be occasionally found in journals that are of questionable repute. Appallingly awful research can find its way into the news. SafetyLit includes these articles, in part, so that when a question is asked about the preposterous findings one may (having read the piece) provide specific reasons that the research is nonsense.



SafetyLit Provides Information -- Not Advice

Important: The articles and reports summarized in each SafetyLit Update are NOT screened for quality.

- The purpose of SafetyLit is to provide its users with information to allow them to identify and find material (of both good and poor quality) that has been published about injury prevention and safety promotion topics. Even when SafetyLit staff believe that there are methodological errors that affect the research findings or when we disagree with the authors' conclusions and statements of implications, an attempt is made to provide an objective summary of the authors' intent. Material in the 'comments' section of each report's summary is provided by the author(s) of the report -- not by SafetyLit.

- An important part of professionalism is to identify flawed publications and counter the flaws by commenting upon them in a letter to the editor of the journal where the article was published. Further, the best knowledge today may become outdated tomorrow. Older articles with inaccuracies are not removed from the site. These out-of-date items may be useful for authors or researchers who are examining the progression of scientific or social thought on a topic.

Revised 2 August 2021

How do you categorize articles?

Items are assigned to 2 or more interest categories. These are described below.

Age Group Categories

No concrete cut-points were established as formal criteria to classify a report into the Age Group categories. Reports are subjectively assigned to these categories.

FIGURE: *SafetyLit's documentation implies a standard where every article should belong to at least two categories. However, our SQL results contradict this. A disproportionately high number of articles are assigned only to* ***"Non-Categorized,"*** *revealing a discrepancy between stated practice and actual execution.*

2. Duplicate Articles Due to Republishing or Revisions

During data validation, we identified multiple entries with identical titles and abstracts but variations in publication year, journal name, or categorization. These discrepancies often stemmed from revised, republished, or translated versions of the same article. However, these variants were not consistently linked via DOI or PMID. This inconsistency contributed to perceived duplication and complicated efforts to connect related records during deduplication or bibliometric analyses. In several cases, articles shared identical abstracts and author information but had distinct PMIDs, leading to classification as unique records when they likely represent the same intellectual content.

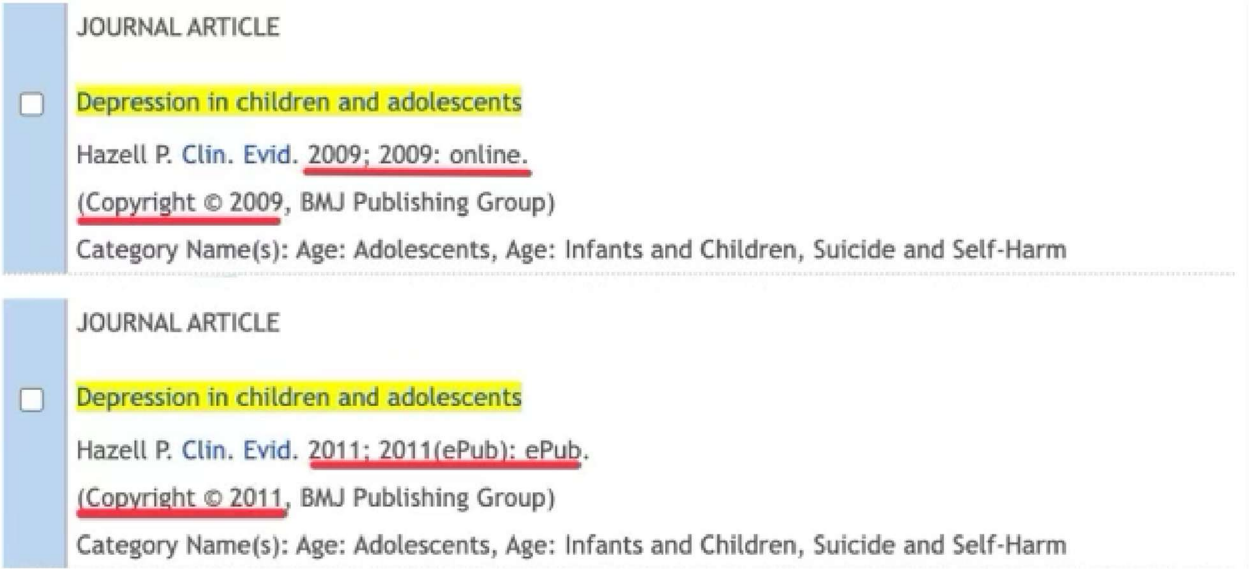


FIGURE : *The image above shows* ***two entries for the same article title*** *— "Depression in children and adolescents" — authored by* ***Hazell P.*** *and published in the* ***same journal (Clin. Evid.)*** *and publishing company (****BMJ Publishing Group)****, but with* ***different publication years*** *(2009 and 2011)*

Journal Article

Depression in children and adolescents

Citation Hazell P. Clin. Evid. 2009; 2009: online.

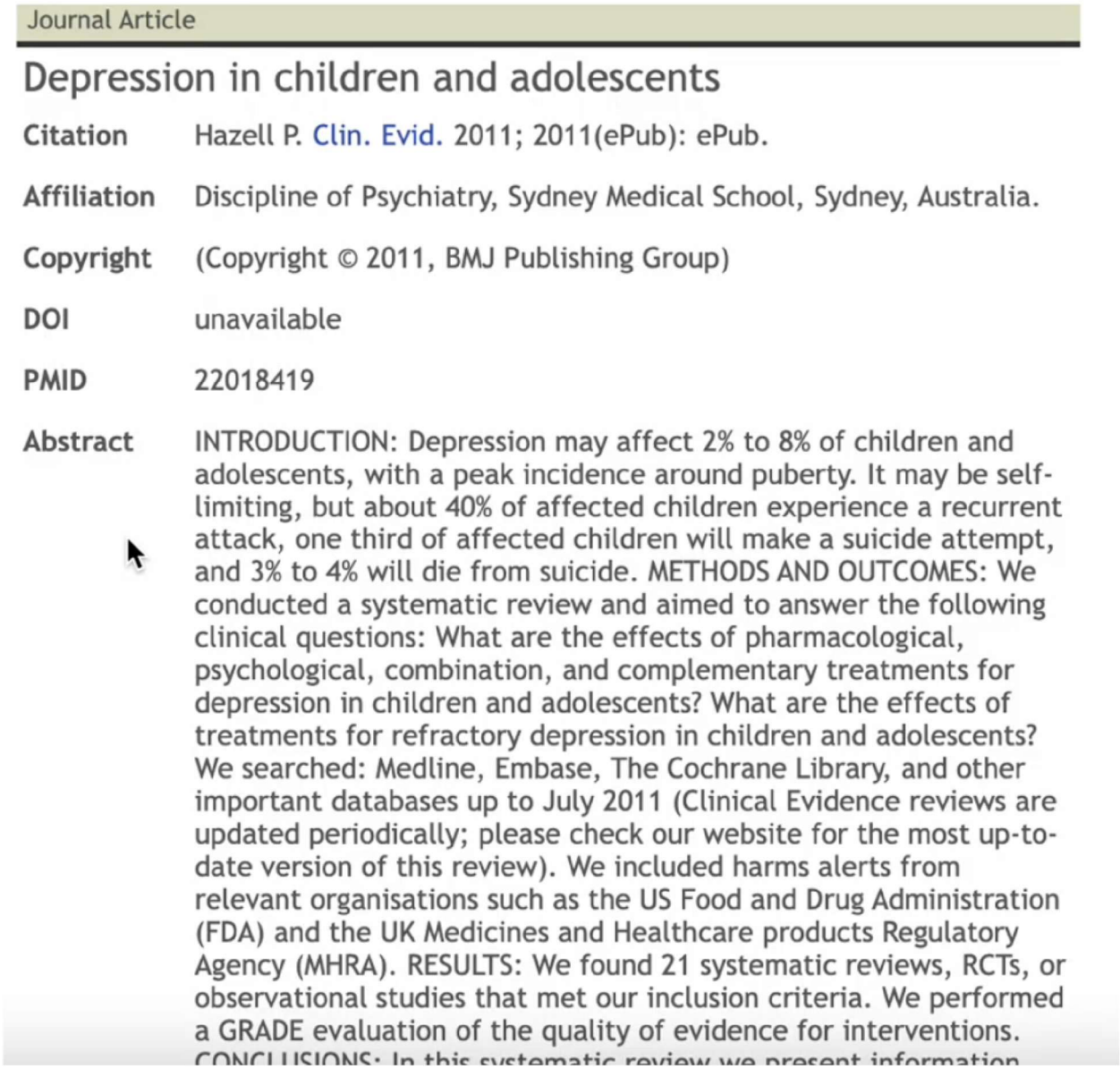
Affiliation Concord Clinical School, University of Sydney, Sydney, Australia.

DOI unavailable

PMID 19445770

PMCID PMC2907806

Abstract INTRODUCTION: Depression may affect 2-8% of children and adolescents, with a peak incidence around puberty. It may be self-limiting, but about 40% of affected children experience a recurrent attack, a third of affected children will make a suicide attempt, and 3-4% will die from suicide. METHODS AND OUTCOMES: We conducted a systematic review and aimed to answer the following clinical questions: What are the effects of pharmacological, psychological, combination, and complementary treatments for depression in children and adolescents? What are the effects of treatments for refractory depression in children and adolescents? We searched: Medline, Embase, The Cochrane Library, and other important databases up to April 2008 (Clinical Evidence reviews are updated periodically, please check our website for the most up-to-date version of this review). We included harms alerts from relevant organisations such as the US Food and Drug Administration (FDA) and the UK Medicines and Healthcare products Regulatory Agency (MHRA). RESULTS: We found 18 systematic reviews, RCTs, or



Journal Article

Depression in children and adolescents

Citation Hazell P. Clin. Evid. 2011; 2011(ePub): ePub.

Affiliation Discipline of Psychiatry, Sydney Medical School, Sydney, Australia.

Copyright (Copyright © 2011, BMJ Publishing Group)

DOI unavailable

PMID 22018419

Abstract INTRODUCTION: Depression may affect 2% to 8% of children and adolescents, with a peak incidence around puberty. It may be self-limiting, but about 40% of affected children experience a recurrent attack, one third of affected children will make a suicide attempt, and 3% to 4% will die from suicide. METHODS AND OUTCOMES: We conducted a systematic review and aimed to answer the following clinical questions: What are the effects of pharmacological, psychological, combination, and complementary treatments for depression in children and adolescents? What are the effects of treatments for refractory depression in children and adolescents? We searched: Medline, Embase, The Cochrane Library, and other important databases up to July 2011 (Clinical Evidence reviews are updated periodically; please check our website for the most up-to-date version of this review). We included harms alerts from relevant organisations such as the US Food and Drug Administration (FDA) and the UK Medicines and Healthcare products Regulatory Agency (MHRA). RESULTS: We found 21 systematic reviews, RCTs, or observational studies that met our inclusion criteria. We performed a GRADE evaluation of the quality of evidence for interventions. CONCLUSIONS: In this systematic review we present information

FIGURE : *The two images above show the same two articles with differing PMIDs, despite all metadata available on SafetyLit being identical.*

3. Response and Commentary Articles Logged as Standalone Records

A recurring pattern emerged in which articles titled as "Response to...", "Comment on...", or "Letter regarding..." were logged as independent entries. These publications referenced earlier articles but were assigned entirely new DOIs and PMIDs. Though technically distinct works, their similar titles introduced surface-level duplication and obscured linkages between articles. This structure complicates attempts to trace scholarly discourse or remove redundant records using automated matching alone.

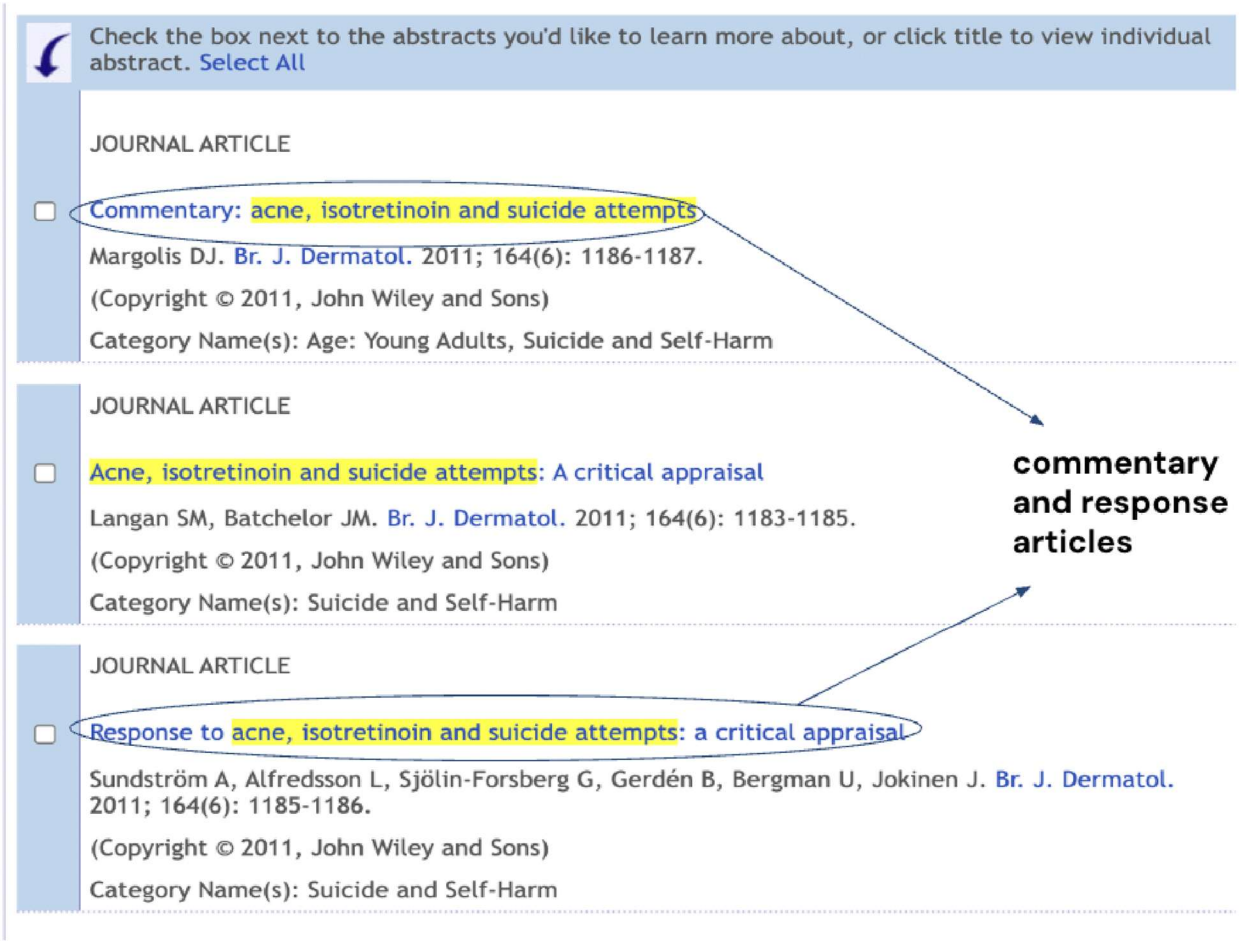


FIGURE** : This set shows an original article and related responses with **differing metadata** — such as authors, pagination, and titles — making the relationship between them more apparent and aiding in **manual or automated differentiation

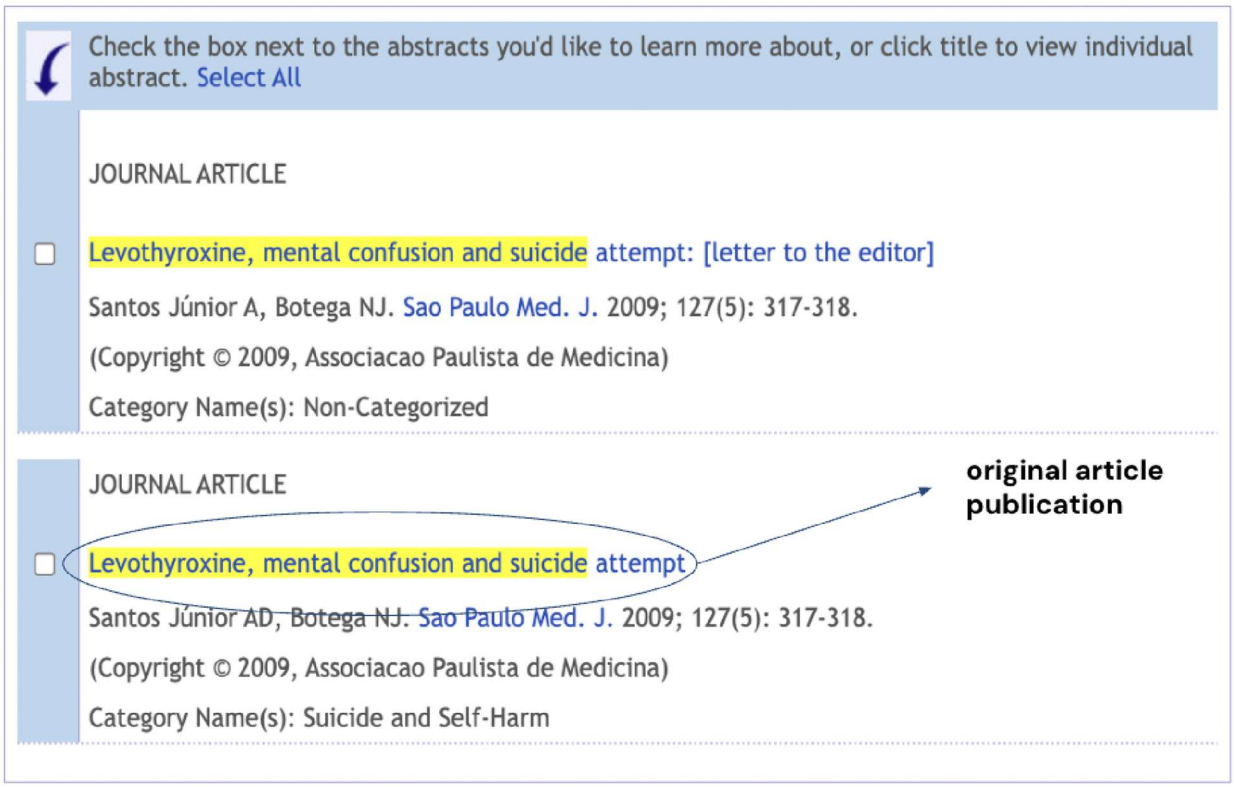


FIGURE : *These two entries have* ***identical metadata*** *— same title, authors, year, journal, and pagination — but differ slightly in formatting, such as the inclusion of a bracketed label like* ***"[letter to the editor]"****. The only difference in* ***captures*** *is the* ***category assignment****.*

4. Limitations of API-Based Enhancement Without DOI or PMID

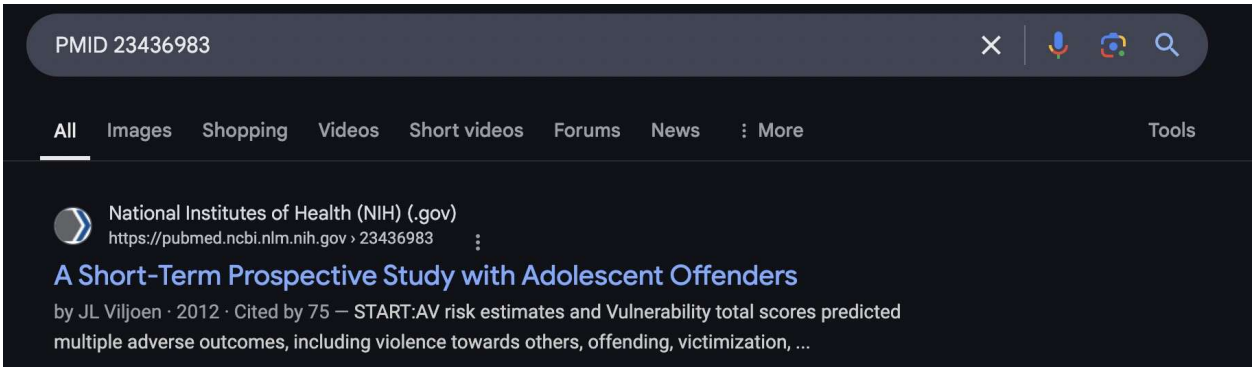
When attempting to enhance missing metadata using external APIs such as Semantic Scholar and OpenAlex, we observed a clear decline in accuracy when both DOI and PMID were missing from the original entry. In these cases, the API responses often returned unrelated or incorrect metadata, particularly when relying on title alone. In contrast, when at least one unique identifier was present, API matches were more reliable. This suggests that the effectiveness of external enhancement is heavily dependent on existing identifiers. Teams relying solely on title-based enhancement may inadvertently introduce further inconsistencies or incorrect associations into their dataset.

5. Title Mismatches in Suggested PMIDs

While validating API-suggested PMIDs, we discovered that several matched IDs led to landing pages where the title preview shown in the search result differed from the title on the actual page. In some cases, the second (clicked) title matched the article in

question; in others, it did not. These mismatches were not detected by any automated mechanism and became apparent only through manual review. Although the metadata appeared legitimate, the presence of differing titles introduces subtle inaccuracies that compromise the reliability of enhancement efforts. This demonstrates the limitations of relying solely on API outputs and highlights the importance of human validation in bibliographic augmentation.

FIGURE : *The images above show* an example of article mismatches that occur when searching PMIDs. In this particular example PMID: 39039263 was searched in google. The associated article initially generated can be seen in the first image; however, once the link is clicked the article changes to a completely different article, shown in the second image.



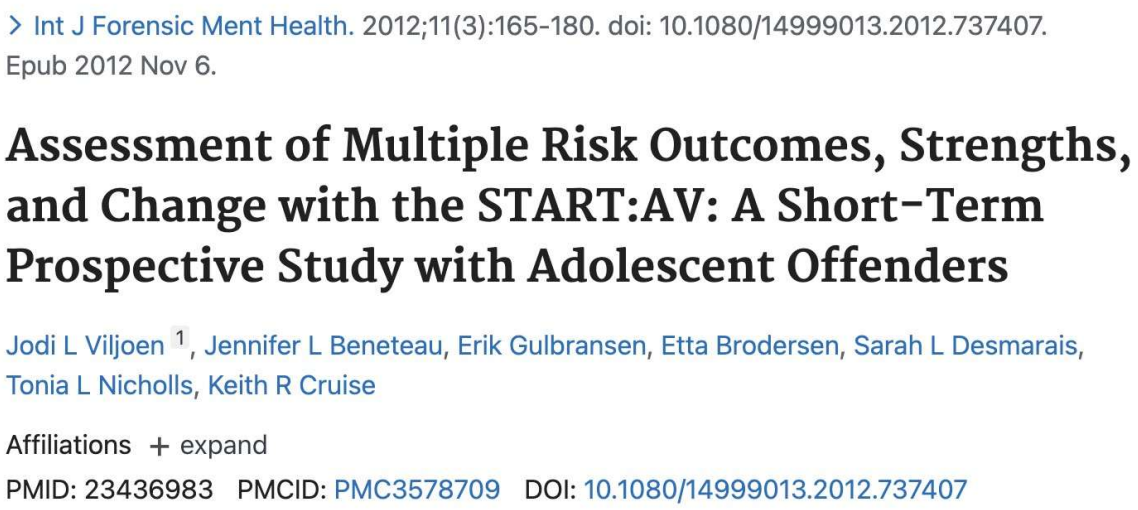


FIGURE : *The images above show* an example of article mismatches that occur when searching PMIDs. In this particular example PMID: 23436983 was searched in google. The associated article initially generated can be seen in the first image; however, once the link is clicked the article changes to a completely different article, shown in the second image.

6. Language Misclassification in Metadata

In one notable instance, the article "First Episode Psychosis and Case Management" was marked as French in our dataset, as per SafetyLit's metadata. However, the actual SafetyLit entry displayed both the title and abstract exclusively in English. Upon following the provided DOI, the linked journal page revealed the article as bilingual — with both French and English versions. This inconsistency raises important concerns about the reliability of language metadata, particularly for region-specific analyses. If language classification is based on full-text content that SafetyLit does not display, then external interpretation becomes skewed. This misalignment has implications for geographic trend analysis, international comparison, and any research efforts attempting to disaggregate findings by language or region.

7. Metadata Merging Errors from Duplicate Retention

Following guidance from our stakeholder, we deliberately retained duplicate titles in our dataset to allow deeper investigation into duplication causes. However, this introduced challenges during DataFrame construction. When merging enhanced metadata into a master dataset, titles were used as the primary join key. In cases where multiple entries shared the same title but had differing metadata — such as methodology (quantitative vs. qualitative) or categorization — the merging process resulted in conflicting values being assigned to a single record. This propagation of mismatched data reflects the risks of treating article titles as unique identifiers. Without disambiguation mechanisms or metadata lineage tracking, such merges can distort analytical outputs and misrepresent original article attributes.


```
methodologies
Qualitative                    25062
Quantitative                    3901
Qualitative; Quantitative         40
```

***FIGURE:**: Because our preprocessing pipeline intentionally preserved duplicate article titles for deeper analysis, the final master DataFrame merged entries with identical titles. As shown above, two articles with the same title but differing metadata (e.g., one labeled qualitative, the other quantitative) were combined. This merging introduces ambiguity into fields like methodology and categories, which complicates downstream analysis and modeling.*

CONCLUSION

Over the course of this project, we developed an end-to-end data pipeline that automates the collection, processing, and analysis of scholarly articles from SafetyLt, focused on suicide and self-harm. Our work integrates web scraping, natural language processing (NLP), and statistical modeling to uncover meaningful patterns in the global research landscape surrounding suicide prevention and mental health.

The completed dataset reveals a number of important insights:

- **Rising Research Volume:** We observed a consistent increase in the number of publications addressing suicide and self-harm over time, with a notable peak in 2014. This trend reflects a growing urgency within the academic and public health communities to address mental health crises.
- **Recurring Focus Areas:** Analysis of article content and keyword frequencies shows a strong thematic concentration on suicidal ideation, depression, psychiatric risk factors, and prevention strategies—highlighting where most academic attention is currently directed.
- **Youth and Adolescent Risk:** A significant share of the articles target younger populations, reinforcing the importance of addressing mental health at early stages of development and the need for age-specific interventions.
- **Language Limitations:** The overwhelming dominance of English-language publications (80.3%) suggests a potential barrier to inclusive, global insights. This linguistic skew may inadvertently exclude culturally relevant research from nonEnglish-speaking regions.
- **Shift Toward Proactive Intervention:** Terminology trends in the dataset indicate a growing focus on early risk detection and proactive mental health interventions, as opposed to retrospective studies alone.

Throughout the development of this pipeline, we prioritized scalability, reproducibility, and the ability to continuously ingest and analyze new data. Our final deliverable is more than just a dataset, it's a research tool designed to:

- Inform mental health policy and clinical decision-making with data-backed evidence.
- Highlight research gaps, especially in underrepresented demographics and languages.
- Support global efforts in suicide prevention by mapping out evolving trends and emerging priorities in the field.

This project not only generated valuable findings but also introduced a replicable methodology for large-scale literature analysis in the mental health domain. Our modular and script-driven pipeline can be adapted to other fields of research, allowing for rapid synthesis of academic literature on emerging public health issues.

By combining computational methods with public health objectives, this project exemplifies the potential of interdisciplinary collaboration. We believe such hybrid approaches are essential for tackling complex social challenges like suicide prevention.

Our analysis framework could support mental health organizations in tracking research progress, prioritizing funding areas, or identifying underserved populations in need of targeted interventions.

We recognize the ethical responsibility that comes with working in sensitive research areas like suicide and mental health. Our goal has been to approach this topic with both rigor and care, ensuring that our insights are used to foster empathy, equity, and evidence-based change.

As a final note, we believe this project lays important groundwork for future interdisciplinary studies that merge public health, data science, and mental health research. By operationalizing large-scale publication data, we contribute not only to academic understanding but also to the broader mission of reducing self-harm and improving mental health outcomes worldwide.

APPENDIX

Appendix A: Web Scraper Automation Code (in scripts/web_scraping_scripts/1-BibTeX-automation.py)

```
import os import
time import re
```

```
import urllib.parse
import requests
from bs4 import BeautifulSoup from
selenium import webdriver
from selenium.webdriver.chrome.service import Service from
selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import Select, WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.chrome.options import Options from
webdriver_manager.chrome import ChromeDriverManager

# Define output directory for downloaded BibTeX files.
output_dir = "./downloaded_bibtex" os.makedirs(output_dir,
exist_ok=True)

def get_total_pages_from_url(results_url, results_per_page=100):
    """
    Extracts the total_records parameter from the results URL query string
and calculates the total number of pages.
    """     try:         parsed =
urllib.parse.urlparse(results_url)         qs =
urllib.parse.parse_qs(parsed.query)         if
"total_records" in qs:
            total_records = int(qs["total_records"][0])
total_pages = total_records // results_per_page             if
total_records % results_per_page != 0:                 total_pages += 1
print(f"Total Records from URL: {total_records}, Total Pages:
{total_pages}")
            return total_pages     except Exception as e:
print("Error parsing total_records from URL:", e)
return 291  # Fallback if parsing fails.

def ensure_display_count(url, count="100"):
    """
    Ensures that the query string of the URL has display_count set to the
specified count.
    """
    parsed = urllib.parse.urlparse(url)
    qs = urllib.parse.parse_qs(parsed.query)
qs["display_count"] = [count]
    new_query = urllib.parse.urlencode(qs, doseq=True)
    new_url = urllib.parse.urlunparse(parsed._replace(query=new_query))
return new_url

def perform_search_and_get_session(advanced_search_url):
    """
    Uses Selenium to perform the advanced search:
- Fills in section D with "self harm" (find_11) and "suicide" (find_12),
- Sets the publication type to "citjournalarticle" and the year range to
  2005-2014.
- Clicks the submit button.
```

```
- Waits until the URL contains the results indicator
(citations.summaries_performance),
        forces the URL to include display_count=100, prints the resulting
URL,         extracts the session ID, and calculates total pages.
    Returns a tuple: (session_id, total_pages, results_url)
    """
    chrome_options = Options()
    # Uncomment the next line to run headless:     #
chrome_options.add_argument("--headless")
chrome_options.add_argument("--disable-gpu")
chrome_options.add_argument("--no-sandbox")
service = Service(ChromeDriverManager().install())
    driver = webdriver.Chrome(service=service, options=chrome_options)
wait = WebDriverWait(driver, 30)

    try:
driver.get(advanced_search_url)
print("Page loaded:", driver.title)

        # Fill in the advanced search fields (section D).
        find_11 = wait.until(EC.visibility_of_element_located((By.NAME,
"find_11")))
        find_12 = wait.until(EC.visibility_of_element_located((By.NAME,
"find_12")))
        print("Search fields 'find_11' and 'find_12' located")
find_11.send_keys("self harm")         find_12.send_keys("suicide")

        # Set publication type.
        pub_type = wait.until(EC.visibility_of_element_located((By.NAME,
"pub_type")))
        print("Publication type selector located")
        Select(pub_type).select_by_value("none")

        # Set year range.
        from_year = wait.until(EC.visibility_of_element_located((By.NAME,
"from")))
        to_year = wait.until(EC.visibility_of_element_located((By.NAME,
"to")))
        print("Year selectors located")
        Select(from_year).select_by_value("2005")
        Select(to_year).select_by_value("2014")

        # Optionally, select all category checkboxes.
        categories = driver.find_elements(By.NAME, "categories[]")
if categories:
            print(f"Found {len(categories)} category checkbox(es)")
for cb in categories:                 if not cb.is_selected():
cb.click()         else:
```

```
            print("No category checkboxes found.")

        # Click the submit button.
        submit_btn = wait.until(EC.element_to_be_clickable((By.XPATH,
"//input[@type='submit']")))
        print("Submit button located, clicking...")
submit_btn.click()

        # Wait until the URL changes to the results page.
        wait.until(lambda d: "citations.summaries_performance" in
d.current_url)         results_url = driver.current_url         #
Force display_count=100 into the URL.         results_url =
ensure_display_count(results_url, "100")         print("Search
results URL:", results_url)

        # Extract session ID from cookies.
cookies = driver.get_cookies()
        session_id = next((c["value"] for c in cookies if c["name"] ==
"PHPSESSID"), None)
        print(f"Extracted Session ID: {session_id}")

        # Get total pages from the modified results URL.
total_pages = get_total_pages_from_url(results_url)     except
Exception as e:
        print("Search failed:", e)
        session_id, total_pages, results_url = None, 291,
advanced_search_url     finally:
        driver.quit()
        return session_id, total_pages,
results_url

def extract_record_ids(html):
    """
    Extracts record IDs from the HTML by looking for <abbr
class="unapiid"> tags.
    If not found, falls back to a regex search.
    """
    soup = BeautifulSoup(html, "html.parser")
    record_elements = soup.find_all("abbr", {"class": "unapi-id"})
record_ids = [el.get("title") for el in record_elements if
el.get("title")]     if record_ids:         print(f"Extracted
{len(record_ids)} record IDs using <abbr> tags.")     else:
        print("No record IDs found using <abbr> tags; attempting regex
extraction.")
        record_ids = re.findall(r"ct\*citjournalarticle_\d+_\d+", html)
return record_ids

def scrape_and_download(results_url, total_pages, post_url, session_id):
"""      Iterates through the results pages using Selenium (to capture
dynamic content),      extracts record IDs from each page, and downloads
```

```
the corresponding BibTeX via a POST request (simulating the
export_all('bib') behavior). Before loading each page, the URL is
modified so that display_count=100.
    """     if not
session_id:
        print("Error: No session ID available. Exiting.")
return

    chrome_options = Options()
    chrome_options.add_argument("--disable-gpu")
chrome_options.add_argument("--no-sandbox")
    # Uncomment for headless scraping:     #
chrome_options.add_argument("--headless")     service
= Service(ChromeDriverManager().install())
    driver = webdriver.Chrome(service=service, options=chrome_options)
    # Set the session cookie.
    driver.get("https://www.safetylit.org/")
driver.add_cookie({"name": "PHPSESSID", "value": session_id, "domain":
"www.safetylit.org"})
        base_headers
= {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:135.0)
Gecko/20100101 Firefox/135.0",         "Accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
        "Accept-Language": "en-US,en;q=0.5",
        "Content-Type": "application/x-www-form-urlencoded",
        "Origin": "https://www.safetylit.org",
        "Connection": "keep-alive",
        "Upgrade-Insecure-Requests": "1",
        "Cookie": f"PHPSESSID={session_id}"
    }         for page in range(1,
total_pages + 1):
        # Construct page URL (replace or add current_page parameter).
        if "current_page=" not in results_url:
            page_url = results_url + f"&current_page={page}"         else:
page_url = re.sub(r"current_page=\d+", f"current_page={page}", results_url)
        # Ensure display_count=100 is in the URL.
        page_url = ensure_display_count(page_url, "100")
print(f"\nProcessing page {page}...")         print("Fetching
URL:", page_url)
                try:             driver.get(page_url)
time.sleep(3)  # Allow dynamic content to load.
            html = driver.page_source
record_ids = extract_record_ids(html)             if
record_ids:                 concatenated_ids =
"|".join(record_ids)                 print("Record IDs
(first 100 chars):", concatenated_ids[:100])
                print("Downloading BibTeX for this page...")
headers = base_headers.copy()                 headers["Referer"]
= page_url
```

```
                response = requests.post(post_url, headers=headers,
data={"export": "bib",
"export_recs": concatenated_ids})
response.raise_for_status()                    if "@article" in
response.text:                        filename =
f"safetylit_page_{page}.bib"                        file_path =
os.path.join(output_dir, filename)                        with
open(file_path, "w", encoding="utf-8") as f:
f.write(response.text.replace("\n\n", "\n").replace("\n", " "))
                    print(f"Page {page} BibTeX saved to: {file_path}")
else:                          print("No valid BibTeX entries found in the
response for this page.")              else:
                print("No record IDs found on this page.")
except Exception as e:               print(f"Error
processing page {page}: {e}")          time.sleep(2)

driver.quit()

if __name__ == "__main__":
    # Advanced search URL (without current_page parameter).
advanced_search_url =
"https://www.safetylit.org/citations/index.php?fuseaction=citations.archiv
esearch"    post_url = "https://www.safetylit.org/unapi/conversion_all.php"

    session_id, total_pages, results_url =
perform_search_and_get_session(advanced_search_url)
    print("\nSession ID:", session_id, "Total Pages:", total_pages)
# Use the results URL for scraping.
    scrape_and_download(results_url, total_pages, post_url, session_id)
```

Appendix B: BibTeX to CSV Conversion Code (2-BibTeX-to-CSV)

```
import os import
csv import html
import
bibtexparser
from bibtexparser.bparser import BibTexParser

# Directory containing the downloaded BibTeX files. bib_dir
= "./downloaded_bibtex"
output_csv = "./output_csvs/script-1-articles.csv"

all_entries = []

# Loop through each .bib file in the directory.
for filename in os.listdir(bib_dir):     if
filename.lower().endswith(".bib"):         bib_file =
```

```
os.path.join(bib_dir, filename)
print(f"Processing file: {filename}")         with
open(bib_file, "r", encoding="utf-8") as f:
            bibtex_str = f.read()
            # Create a new parser for each file to avoid accumulating
entries.
            parser = BibTexParser(common_strings=True)
parser.interpolate_strings = True
parser.ignore_nonstandard_types = False
            bib_database = bibtexparser.loads(bibtex_str, parser=parser)
all_entries.extend(bib_database.entries)

print(f"Total BibTeX entries found: {len(all_entries)}")

# Define the CSV header fields.
fieldnames = [
    'Title',
    'Journal',
    'Year',
    'Volume',
    'PMID',
    'Number',
    'Pages',
    'DOI',
    'URL',
    'Abstract',
    'Authors',
    'Language'
]

# Write the CSV file. with open(output_csv, "w", newline='',
encoding="utf-8") as csvfile:     writer = csv.DictWriter(csvfile,
fieldnames=fieldnames)     writer.writeheader()     for entry in
all_entries:
        row = {
            'Title': html.unescape(entry.get('title', '')),
            'Journal': html.unescape(entry.get('journal', '')),
            'Year': entry.get('year', ''),
            'Volume': entry.get('volume', ''),
            'Number': entry.get('number', ''),
            'Pages': entry.get('pages', ''),
            'DOI': entry.get('doi', ''),
            'URL': entry.get('url', ''),
            'Abstract': html.unescape(entry.get('abstract', '')),
            'Authors': html.unescape(entry.get('author', '')),
            'Language': entry.get('language', '')
        }
        writer.writerow(row)

print(f"CSV file '{output_csv}' written successfully!")
```

Appendix C: PMID and Affiliations Scraping

```
 import os import
time import pandas
as pd import
urllib.parse import
logging
from selenium import webdriver
from selenium.webdriver.chrome.service import Service from
selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait, Select
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.chrome.options import Options from
webdriver_manager.chrome import ChromeDriverManager from bs4
import BeautifulSoup
from rapidfuzz import process, fuzz

# ---------------------- CONFIGURATION ---------------------- #
INPUT_CSV = "./output_csvs/script-1-articles.csv"
OUTPUT_CSV = "./output_csvs/script-3-articles.csv"
SAFETYLIT_ADV_SEARCH_URL =
"https://www.safetylit.org/citations/index.php?fuseaction=citations.archiv
esearch"
TOTAL_PAGES = 291  # Total number of result pages

# Logging setup logging.basicConfig(
    filename="metadata_extraction.log",
filemode="w",
    format="%(asctime)s - %(levelname)s - %(message)s",
level=logging.DEBUG
)

# Selenium WebDriver setup chrome_options
= Options()
chrome_options.add_argument("--headless")  # Run in headless mode (no
browser UI)
chrome_options.add_argument("--disable-gpu")
chrome_options.add_argument("--no-sandbox") service
= Service(ChromeDriverManager().install())
driver = webdriver.Chrome(service=service, options=chrome_options) wait
= WebDriverWait(driver, 30)


# ---------------------- FUNCTIONS ---------------------- #

def perform_advanced_search():
    """Performs an advanced search on SafetyLit and retrieves the results
URL."""
    driver.get(SAFETYLIT_ADV_SEARCH_URL)
    logging.info("Navigated to SafetyLit Advanced Search page.")
```

```
    # Fill in search keywords
    wait.until(EC.visibility_of_element_located((By.NAME,
"find_11"))).send_keys("self harm")
    wait.until(EC.visibility_of_element_located((By.NAME,
"find_12"))).send_keys("suicide")
    logging.info("Entered search keywords: 'self harm' and 'suicide'.")

    # Select publication type: Journal Article
    Select(wait.until(EC.visibility_of_element_located((By.NAME,
"pub_type")))).select_by_value("citjournalarticle")

    # Select year range (2005-2014)
    Select(wait.until(EC.visibility_of_element_located((By.NAME,
"from")))).select_by_value("2005")
    Select(wait.until(EC.visibility_of_element_located((By.NAME,
"to")))).select_by_value("2014")
    logging.info("Set publication type and year range (2005-2014).")

    # Select all categories
    categories = driver.find_elements(By.NAME, "categories[]")
for checkbox in categories:         if not
checkbox.is_selected():
            checkbox.click()
    logging.info(f"Selected {len(categories)} categories.")

    # Submit the search form
    wait.until(EC.element_to_be_clickable((By.XPATH,
"//input[@type='submit']"))).click()
    logging.info("Submitted the advanced search form.")

    # Wait until the URL contains search results
wait.until(lambda d: "citations.summaries_performance" in
d.current_url)

    # Modify the results URL to show 100 results per page
results_url = ensure_display_count(driver.current_url, "100")
logging.info(f"Advanced search results URL: {results_url}")

    return results_url


def ensure_display_count(url, count="100"):
    """Ensures the query string contains display_count=100."""
parsed = urllib.parse.urlparse(url)     qs =
urllib.parse.parse_qs(parsed.query)     qs["display_count"] =
[count]
    new_query = urllib.parse.urlencode(qs, doseq=True)
    new_url = urllib.parse.urlunparse(parsed._replace(query=new_query))
logging.debug(f"Fixed URL: {new_url}")     return new_url
```

```
def select_all_and_view_abstracts():
    """Clicks 'Select All' and 'View Full Abstracts For Selected
Items'."""     time.sleep(3)  # Allow
page to load

    # Click "Select All" button
try:
        select_all_button = driver.find_element(By.XPATH,
"//a[contains(@href, \"javascript:selectAll\")]")
select_all_button.click()
        logging.info("Clicked 'Select All' button.")
time.sleep(2)     except Exception as e:
        logging.error(f"Error clicking 'Select All': {e}")

    # Click "View Full Abstracts" button
try:
        view_abstracts_button = driver.find_element(By.XPATH,
"//input[@value='View Full Abstracts For Selected Items']")
view_abstracts_button.click()
        logging.info("Clicked 'View Full Abstracts For Selected Items'
button.")
        time.sleep(5)  # Wait for abstracts page to load
except Exception as e:
        logging.error(f"Error clicking 'View Full Abstracts': {e}")


def extract_metadata_from_page():
    """Extracts PMIDs and Affiliations for articles on the current
page."""
    soup = BeautifulSoup(driver.page_source, "html.parser")
extracted_data = {}

    for article_section in soup.find_all("tbody"):
title_label = article_section.find("label")

        # ✅ Fix: Look for <strong>PMID
        pmid_row = article_section.find(lambda tag: tag.name == "td" and
tag.find("strong") and "PMID" in tag.get_text())
        pmid = pmid_row.find_next("td").get_text(strip=True) if pmid_row
else "Not Available"

        # ✅ Fix: Look for <strong>Affiliation
        affiliation_row = article_section.find(lambda tag: tag.name ==
"td" and tag.find("strong") and "Affiliation" in tag.get_text())
affiliation = affiliation_row.find_next("td").get_text(strip=True) if
affiliation_row else "Not Available"
```

```
        if title_label:             title =
title_label.get_text(strip=True)
extracted_data[title] = {"PMID": pmid,
"Affiliation": affiliation}
logging.debug(f"Extracted - Title: {title} | PMID:
{pmid} | Affiliation: {affiliation}")

    logging.info(f"Extracted metadata for {len(extracted_data)} articles
on the current page.")     return extracted_data


def paginate_and_extract_metadata(results_url):
    """Iterates through all search result pages, extracts metadata, and
stores them."""     all_metadata = {}

    for page in range(1, TOTAL_PAGES + 1):
        logging.info(f"Processing page {page} of {TOTAL_PAGES}...")

        page_url =
ensure_display_count(f"{results_url}&current_page={page}", "100")
driver.get(page_url)
        time.sleep(5)  # Allow time for page load

        select_all_and_view_abstracts()  # Ensure abstracts are visible

        metadata = extract_metadata_from_page()
if metadata:
            all_metadata.update(metadata)         else:
logging.warning(f"No metadata extracted on page {page}")
logging.debug(f"Failed Page URL: {page_url}")

    logging.info(f"Total articles extracted: {len(all_metadata)}")
return all_metadata


def match_titles_to_metadata(articles_data, metadata):
    """Matches article titles from the CSV to extracted metadata using
fuzzy matching."""
    normalized_metadata = {title.lower(): data for title, data in
metadata.items()}

    def find_best_match(title):
        if not isinstance(title, str):  # Check if title is NaN or
nonstring             return {"PMID": "Not Available", "Affiliation":
"Not
Available"}

        best_match = process.extractOne(title.lower(),
normalized_metadata.keys(), scorer=fuzz.ratio)
```

```
        if best_match and best_match[1] > 85:  # 85% similarity threshold
return normalized_metadata[best_match[0]]

        return {"PMID": "Not Available", "Affiliation": "Not Available"}

    matched_metadata = articles_data["Title"].apply(find_best_match)
articles_data["PMID"] = matched_metadata.apply(lambda x: x["PMID"])
articles_data["Affiliation"] = matched_metadata.apply(lambda x:
x["Affiliation"])
    return articles_data

# ---------------------- MAIN EXECUTION ---------------------- # if
__name__ == "__main__":
    search_results_url = perform_advanced_search()
    all_metadata = paginate_and_extract_metadata(search_results_url)

    # Load CSV data
    articles_df = pd.read_csv(INPUT_CSV)

    # 🦠 Ensure 'Title' column is properly formatted before processing
articles_df["Title"] = articles_df["Title"].astype(str).fillna("")

    # Match extracted metadata to articles
    updated_df = match_titles_to_metadata(articles_df, all_metadata)

    # Save the updated DataFrame
    updated_df.to_csv(OUTPUT_CSV, index=False, encoding="utf-8")

    # Close Selenium WebDriver
driver.quit()
```

Appendix D: Category Scraping (4-Category-scraping)

```
import os import time import re
import pandas as pd import
urllib.parse import logging
import unicodedata from
selenium import webdriver
from selenium.webdriver.chrome.service import Service from
selenium.webdriver.chrome.options import Options from
selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait, Select
from selenium.webdriver.support import expected_conditions as EC
from webdriver_manager.chrome import ChromeDriverManager from
bs4 import BeautifulSoup from rapidfuzz import process, fuzz

# ---------------------- CONFIGURATION ---------------------- #
INPUT_CSV = "./output_csvs/script-3-articles.csv"
OUTPUT_CSV = "./output_csvs/script-4-articles.csv"
```

```
SAFETYLIT_ADV_SEARCH_URL =
"https://www.safetylit.org/citations/index.php?fuseaction=citations.archiv
esearch"
TOTAL_PAGES = 291  # Adjust as needed (e.g., 291)
FUZZY_THRESHOLD = 85  # Similarity threshold for fuzzy matching
LOOKAHEAD_WINDOW = 10  # Increased look-ahead window for category rows

# Logging setup logging.basicConfig(
    filename="categories_extraction.log",
filemode="w",
    format="%(asctime)s - %(levelname)s - %(message)s",
level=logging.DEBUG
)

# ---------------------- SELENIUM SETUP ---------------------- #
chrome_options = Options()
chrome_options.add_argument("--headless")
chrome_options.add_argument("--disable-gpu")
chrome_options.add_argument("--no-sandbox") service
= Service(ChromeDriverManager().install())
driver = webdriver.Chrome(service=service, options=chrome_options) wait
= WebDriverWait(driver, 30)

# ---------------------- HELPER FUNCTIONS ---------------------- #
def ensure_display_count(url, count="100"):     parsed =
urllib.parse.urlparse(url)     qs =
urllib.parse.parse_qs(parsed.query)     qs["display_count"] =
[count]
    new_query = urllib.parse.urlencode(qs, doseq=True)
    new_url = urllib.parse.urlunparse(parsed._replace(query=new_query))
logging.debug(f"Fixed URL: {new_url}")     return new_url

def normalize_unicode(text):
    """
    Replace common problematic Unicode characters with standard ones,
collapse extra spaces, and return a stripped string.
    """
    replacements = {
        '\uFF1A': ':',    # fullwidth colon → colon
        '\u00A0': ' ',    # non-breaking space → space
        '\u2013': '-',    # en dash → hyphen
        '\u2014': '-',    # em dash → hyphen,
    }     for old, new in
replacements.items():         text =
text.replace(old, new)     text =
re.sub(r'\s+', ' ', text)     return
text.strip()

def remove_accents(input_str):
    """
```

```
    Remove diacritics from a string, handling non-string inputs safely.
"""     if not isinstance(input_str, str):
        input_str = str(input_str)  # Convert float or NaN to a string
nfkd_form = unicodedata.normalize('NFKD', input_str)
    return "".join([c for c in nfkd_form if not unicodedata.combining(c)])

def parse_category_text(full_text):
    """
    Uses a loose regex to capture everything after "Category Name(s):"
(allowing variable spaces) until "journal article" (if present) or end-
of-string.     """
    normalized_text = normalize_unicode(full_text)
pattern = re.compile(
        r'(?i)category\s*name\(s\)\s*:\s*(.*?)(?=\s*journal article|$)',
re.DOTALL
    )
    match = pattern.search(normalized_text)
if not match:         return "Not
Available"     cat_text =
match.group(1).strip()
    return cat_text if cat_text else "Not Available"

def is_likely_title(candidate):
    """
    Returns True if the candidate string appears to be an article title.
    Heuristics: candidate contains a 4-digit year, or "--", or any quotes
(single or double),     or its length
exceeds 15 characters.
    """     if re.search(r'\d{4}',
candidate):         return True     if
"--" in candidate:
        return True     if '"' in candidate
or "'" in candidate:
        return True     if
len(candidate) > 15:
return True     return
False

# ---------------------- EXTRACTION FUNCTIONS ---------------------- #
def perform_advanced_search():
driver.get(SAFETYLIT_ADV_SEARCH_URL)
    logging.info("Navigated to SafetyLit Advanced Search page.")
wait.until(EC.visibility_of_element_located((By.NAME,
"find_11"))).send_keys("self harm")
    wait.until(EC.visibility_of_element_located((By.NAME,
"find_12"))).send_keys("suicide")
    pub_type_select =
Select(wait.until(EC.visibility_of_element_located((By.NAME,
"pub_type"))))
    pub_type_select.select_by_value("citjournalarticle")
Select(wait.until(EC.visibility_of_element_located((By.NAME,
```

```
"from")))).select_by_value("2005")
Select(wait.until(EC.visibility_of_element_located((By.NAME,
"to")))).select_by_value("2014")     for checkbox in
driver.find_elements(By.NAME, "categories[]"):         if not
checkbox.is_selected():
            checkbox.click()
    wait.until(EC.element_to_be_clickable((By.XPATH,
"//input[@type='submit']"))).click()
    wait.until(lambda d: "citations.summaries_performance" in
d.current_url)
    results_url = ensure_display_count(driver.current_url, "100")
logging.info(f"Advanced search results URL: {results_url}")     return
results_url

def extract_articles_grouped():
    """
    Iterates over all <tr> elements and groups rows into article records.
When a row contains an <a> tag that is likely a title, it starts a new
record.
    Then, it looks ahead (up to LOOKAHEAD_WINDOW rows) for a row
containing "Category Name(s):"     and assigns that parsed text
to the current record.
    Returns a dictionary mapping article title to its category text.
    """
    wait.until(EC.presence_of_all_elements_located((By.CSS_SELECTOR,
"table tr")))
time.sleep(2)
    soup = BeautifulSoup(driver.page_source, "html.parser")
rows = soup.find_all("tr")
        records = {}
idx = 0     while idx <
len(rows):
        row_text = normalize_unicode(rows[idx].get_text(" ", strip=True))
if "select all" in row_text.lower():
            idx += 1
continue
        a_tag = rows[idx].find("a")         if
a_tag:             candidate =
a_tag.get_text(strip=True)             if
is_likely_title(candidate):
current_title = candidate                 cat_text
= "Not Available"
                # Look ahead up to LOOKAHEAD_WINDOW rows for category info
for    j    in    range(idx+1,    min(idx+1+LOOKAHEAD_WINDOW,    len(rows))):
next_text    =    normalize_unicode(rows[j].get_text("    ",    strip=True))
if "category name(s):" in next_text.lower():                         cat_text
= parse_category_text(next_text)                             break
                records[current_title] = cat_text
                logging.debug(f"Article title: '{current_title}',
Categories: '{cat_text}' at row {idx}")
idx += 1
    return records
```

```
def paginate_and_extract_data(results_url):
    all_data = {}     for page in range(1, TOTAL_PAGES + 1):
logging.info(f"Processing page {page} of {TOTAL_PAGES}...")
page_url =
ensure_display_count(f"{results_url}&current_page={page}", "100")
driver.get(page_url)
        wait.until(EC.presence_of_all_elements_located((By.TAG_NAME,
"tr")))
        time.sleep(3)
        page_data = extract_articles_grouped()
all_data.update(page_data)
    logging.info(f"Total articles with extracted data: {len(all_data)}")
return all_data

# ---------------------- CSV MATCHING FUNCTIONS ---------------------- #
def normalize_title_for_match(title):     if pd.isna(title) or not
isinstance(title, str):         return ""
    no_accents = remove_accents(title)
return " ".join(re.sub(r'[^\w\s]', '',
no_accents).lower().strip().split())

def match_titles_to_data(csv_df, extracted_data):
    normalized_extracted = {normalize_title_for_match(t): cat for t, cat
in extracted_data.items()}     def find_best_match(csv_title):
        norm_title = normalize_title_for_match(csv_title)
best = process.extractOne(norm_title,
list(normalized_extracted.keys()), scorer=fuzz.ratio)
if best and best[1] >= FUZZY_THRESHOLD:
return normalized_extracted[best[0]]         else:
            for key, cat in normalized_extracted.items():
if key in norm_title or norm_title in key:
logging.debug(f"Fallback match: '{norm_title}' matched with '{key}'")
                    return cat
return "Not Available"
    csv_df["Categories"] = csv_df["Title"].apply(find_best_match)
return csv_df

# ---------------------- MAIN EXECUTION ---------------------- # if
__name__ == "__main__":
    try:
        search_results_url = perform_advanced_search()
        extracted_data = paginate_and_extract_data(search_results_url)
articles_df = pd.read_csv(INPUT_CSV)
        updated_df = match_titles_to_data(articles_df, extracted_data)
updated_df.to_csv(OUTPUT_CSV, index=False, encoding="utf-8")
logging.info(f"Data extraction complete. Output saved to {OUTPUT_CSV}.")
    except Exception as e:
        logging.error("An error occurred during extraction: " + str(e))
finally:
        driver.quit()
```

Appendix E: NLP Script (5-NLP-script)

```
import pandas as pd
import spacy import
re
from collections import Counter

# ---------------------- CONFIGURATION ---------------------- #
INPUT_CSV = "./output_csvs/script-4-articles.csv"
OUTPUT_CSV = "./output_csvs/script-5-articles.csv"

# Load spaCy English model (ensure it's installed: python -m spacy
download en_core_web_sm) nlp = spacy.load("en_core_web_sm")

# ---------------------- NLP EXTRACTION FUNCTIONS ---------------------- #
def extract_keywords(text, top_n=5):
    """
    Extracts keywords by selecting the most frequent nouns and proper
nouns.     """     doc = nlp(text)
    tokens = [token.text.lower() for token in doc if token.pos_ in
["NOUN", "PROPN"]]
freq = Counter(tokens)
    most_common = [word for word, count in freq.most_common(top_n)]
return ", ".join(most_common) if most_common else "Keyword extraction
unavailable"

def extract_research_methodology(text):
    """
    Basic extraction from a single text input.
    (This simple version is superseded by the improved version that uses
all data.)     """
    text_lower = text.lower()     if
"quantitative" in text_lower:
        return "Quantitative"
elif "qualitative" in text_lower:
return "Qualitative"     else:
        return "Qualitative"  # Default fallback

def extract_prevention_measures(text):
    """
    Identifies mentions of suicide prevention measures.
    If none are found, returns a comprehensive default suggestion.
    """
    measures = []
    lower_text = text.lower()     if
"counseling" in lower_text:
measures.append("Counseling")     if
"therapy" in lower_text:
```

```
measures.append("Therapy")     if
"hotline" in lower_text:
        measures.append("Hotline support")
if "awareness" in lower_text:
        measures.append("Awareness programs")     if "crisis"
in lower_text or "intervention" in lower_text:
        measures.append("Crisis intervention")
if "support group" in lower_text:
measures.append("Support groups")     if
"mental health" in lower_text:
        measures.append("Mental health services")
       if
measures:
        # Remove duplicates and join
measures = list(set(measures))
return ", ".join(measures)     else:
        # Default comprehensive prevention measures
        return ("Implement comprehensive suicide prevention programs
including crisis intervention, "
                "mental health support, counseling, and public awareness
campaigns")

def extract_data_source_from_text(text):
    """
    Uses heuristics to extract data source information from a text string.
    """
    text_lower = text.lower()     if
"news" in text_lower:         return
"News Media"     elif "journal" in
text_lower:         return "Academic
Journal"     elif "conference" in
text_lower:         return "Conference
Proceedings"     elif "website" in
text_lower:         return "Online
Publication"     else:
        return "Multiple Sources"

def extract_grant_info(text):
    """
    Checks if there is any mention of grants or funding.
    If none are found, returns a default message.
    """
    if re.search(r"\b(grant[s]?|funding|supported by)\b", text,
re.IGNORECASE):
        return "Yes"
    return "No grant funding information provided"

def process_text(text):
    """
    Applies NLP extraction functions to the provided Abstract text.
    (Keywords, prevention measures, and grant information.)
```

```
    """    if
pd.isnull(text):
        text = ""
    keywords = extract_keywords(text)
    prevention_measures = extract_prevention_measures(text)
grant_info = extract_grant_info(text)
       return pd.Series({
"Keywords": keywords,
        "Suggested suicide prevention measures": prevention_measures,
        "Grant information": grant_info
    })

def improved_extract_research_methodology(row):
    """
    Combines text from Title, Abstract, and Categories to classify
research methodology as either 'Qualitative' or 'Quantitative'.
Defaults to 'Qualitative' if no clear evidence is found.
    """
    combined_text = " ".join([str(row[col]) for col in ["Title",
"Abstract", "Categories"]
                              if col in row and
pd.notnull(row[col])]).lower()    quantitative_keywords =
["quantitative", "survey", "regression",
"statistical", "experiment",
                             "experimental", "rct", "anova"]
    qualitative_keywords = ["qualitative", "narrative", "interview", "case
study", "focus group",
                            "thematic", "content analysis",
"ethnographic", "dramatism", "narrative analysis",
                            "critical", "framework", "theoretical",
"critical cultural studies", "analysis"]

    quantitative_score = sum(1 for kw in quantitative_keywords if kw in
combined_text)
    qualitative_score = sum(1 for kw in qualitative_keywords if kw in
combined_text)
       if quantitative_score == 0 and qualitative_score
== 0:        return "Qualitative"    elif
quantitative_score > qualitative_score:
        return "Quantitative"
else:
        return "Qualitative"

def improved_extract_data_source(row):
    """
    Combines text from Title, Abstract, and Categories to heuristically
determine the data source.
    """
    combined_text = " ".join([str(row[col]) for col in ["Title",
"Abstract", "Categories"]
```

```
                                if col in row and pd.notnull(row[col])])
    return extract_data_source_from_text(combined_text)

def process_row(row):
    """
    Processes each row:
-     Uses the Abstract for keywords, prevention measures, and grant
info.
-     Overrides the Research methodology and Data source fields
using improved functions that leverage all available data.
    """
    abstract_text = str(row["Abstract"]) if pd.notnull(row["Abstract"])
else ""
    extracted = process_text(abstract_text)
extracted["Research methodology"] =
improved_extract_research_methodology(row)
    extracted["Data source"] = improved_extract_data_source(row)
return extracted

# ---------------------- MAIN EXECUTION ---------------------- # def
main():
    df = pd.read_csv(INPUT_CSV)
        if "Abstract" not in df.columns:         print("Error:
CSV must contain an 'Abstract' column.")         return

    extracted_fields = df.apply(process_row, axis=1)
df_extended = pd.concat([df, extracted_fields], axis=1)
df_extended.to_csv(OUTPUT_CSV, index=False)     print("Extended
CSV saved as: " + OUTPUT_CSV)

if __name__ == "__main__":
    main()
```

Appendix F: Data Cleaning Script (6-Data-cleaning-script)

```
import pandas as pd
import numpy as np
import re import
logging

# Configure logging logging.basicConfig(
    filename="data_validation.log",
filemode="w",
    format="%(asctime)s - %(levelname)s - %(message)s",
level=logging.INFO
)

# Load the dataset
file_path = "./output_csvs/latest-data-articles.csv"
```

```
df = pd.read_csv(file_path, dtype=str)  # Read everything as string to
prevent unwanted conversions

# Ensure all rows are retained before modifications original_row_count
= df.shape[0]

# ---------------------- DATA VALIDATION & CLEANING ----------------------
#

# 1. Handle Missing Values - Convert only empty cells to NULL
df.fillna("NULL", inplace=True)

# 2. Trim Text Fields - Remove leading and trailing spaces from all string
columns for col in df.select_dtypes(include=['object']).columns:
df[col] = df[col].str.strip()
    df[col].replace({"NULL": np.nan}, inplace=True)  # Preserve NULL
values

# 3. Standardize Title Case Formatting for Specific Columns
title_case_columns = ["Title", "Author", "Journal"]  # Modify as per your
dataset for col in title_case_columns:
    if col in df.columns:         df[col] = df[col].apply(lambda x:
x.title() if isinstance(x, str) else x)

# 4. Remove Non-Alphanumeric Characters from Text Fields (Optional)
text_columns = ["Title", "Abstract"]  # Add more fields if necessary
for col in text_columns:     if col in df.columns:         df[col] =
df[col].apply(lambda x: re.sub(r"[^a-zA-Z0-9\s\.,']", "", x) if
isinstance(x, str) else x)

# 5. Ensure Numeric Columns are Properly Formatted (but do not drop rows)
numeric_columns = ["Year", "Citations", "Impact Factor"]  # Modify as per
your dataset for col in numeric_columns:     if col in df.columns:
df[col] = pd.to_numeric(df[col],
errors="coerce").astype("Int64")  # Keeps NaNs as NULL-compatible
df[col].fillna("NULL", inplace=True)  # Ensures NULL remains for missing
values

# 6. IDENTIFY DUPLICATES BUT DO NOT REMOVE THEM
duplicate_rows = df[df.duplicated(keep=False)]  # Find duplicated rows if
not duplicate_rows.empty:     logging.warning(f"⚠ {len(duplicate_rows)}
duplicate records detected. These will NOT be dropped.")     print(f"⚠
WARNING: {len(duplicate_rows)} duplicate records detected. They are being
retained.")

# 7. Validate Dates (without dropping rows)
date_columns = ["Publication Date"]  # Modify as needed
for col in date_columns:     if col in df.columns:
```

```
        df[col] = pd.to_datetime(df[col], errors="coerce").astype(str)
df[col].replace({"NaT": "NULL"}, inplace=True)  # Replace invalid dates
with NULL

# 8. Validate DOI Field - Only fill empty cells with NULL, but retain
valid DOIs if "DOI" in df.columns:
    df["DOI"] = df["DOI"].apply(lambda x: x if isinstance(x, str) and
x.strip() != "" else "NULL")

# 9. Validate URL Field - Replace "http://dx.doi.org/" exactly with NULL,
but retain other URLs if "Article URL" in df.columns:
    df["Article URL"] = df["Article URL"].apply(lambda x: "NULL" if
x.strip() == "http://dx.doi.org/" else x)

# ---------------------- ENSURE NO ROWS ARE LOST ---------------------- #

# Ensure the number of rows remains unchanged final_row_count =
df.shape[0] if final_row_count < original_row_count:
logging.error(f"🚨 ERROR: {original_row_count - final_row_count} rows
were lost unexpectedly!")
    print(f"🚨 ERROR: {original_row_count - final_row_count} rows were
lost. Investigate before loading.")

# ---------------------- SAVE CLEANED DATA ---------------------- #

# Save to CSV with NULL values for database ingestion
cleaned_file_path = "./output_csvs/cleaned-latest-data-articles.csv"
df.to_csv(cleaned_file_path, index=False, na_rep="NULL")  # Ensure NULL
representation

# Final Validation Summary final_missing_summary
= df.isna().sum() logging.info(f"Missing values
per column after
cleaning:\n{final_missing_summary}")

print(f"Data Validation Complete! All {final_row_count} rows retained.
Validated dataset saved to: {cleaned_file_path}")
```

Appendix G: Grant Completeness (7-Grant-completeness)

```
import pandas as pd from
Bio import Entrez

# REQUIRED: Set your email address for NCBI Entrez
Entrez.email = "pmohunsingh@gmail.com"

# OPTIONAL: Add an NCBI API key if you have one (helps avoid rate limits)
# Entrez.api_key = "YOUR_NCBI_API_KEY"
```

```
def fetch_grant_info_from_pubmed(pmid):
    """
    Query PubMed for the article with the given PMID and return its grant
information.

Args:
        pmid (int or str): The PubMed ID of the article to look up.

    Returns:        str: A string describing the grants (if any) or 'No
grant found.'
    """
try:
        print(f"Fetching grants for PMID={pmid}")

        # Fetch the article data in XML format
        handle = Entrez.efetch(db="pubmed", id=str(pmid), retmode="xml")
records = Entrez.read(handle)        handle.close()

        # If no data came back or no PubmedArticle key, no info is found
if not records or "PubmedArticle" not in records or
len(records["PubmedArticle"]) == 0:            print(f"  -> No
PubmedArticle found for PMID={pmid}")            return "No grant
found."

        article = records["PubmedArticle"][0]
# Grant info is typically under
['MedlineCitation']['Article']['GrantList']
grant_list =
article["MedlineCitation"]["Article"].get("GrantList", [])

        if not grant_list:
            print(f"  -> No GrantList found for PMID={pmid}")
return "No grant found."

        # Build a descriptive string from the grants
grants_str_list = []        for g in grant_list:
            grant_id = g.get("GrantID", "")            agency =
g.get("Agency", "")            country = g.get("Country", "")
snippet = f"GrantID: {grant_id} (Agency: {agency}, Country:
{country})"
            grants_str_list.append(snippet)

        grants_str = " ; ".join(grants_str_list)
        print(f"  -> Grants found for PMID={pmid}: {grants_str}")
return grants_str if grants_str else "No grant found."

    except Exception as e:
        print(f"Error fetching grant info for PMID={pmid}: {e}")
return "No grant found."
```

```
def update_grant_info(csv_path, output_path):
    """
    Reads a CSV, fetches actual grant info from PubMed for each row using
the 'PMID' column,     and writes the updated data to a new CSV.
    """
    df = pd.read_csv(csv_path)     print(f"Loaded CSV:
{csv_path} with shape={df.shape}")

    # Ensure there is a "Grant information" column
if "Grant information" not in df.columns:
        df["Grant information"] = ""

    # Iterate through rows     for
idx, row in df.iterrows():
        # Pull the PMID from the column named "PMID" (case-sensitive)
pmid = row.get("PMID", None)
                print(f"\nRow {idx}:
PMID={pmid}")

        # If we don't have a valid PMID, note it and skip
if pd.isna(pmid):             print("  -> No valid PMID in this
row. Skipping.")            df.at[idx, "Grant information"] =
"No PMID provided."            continue

        # Fetch the actual grant information from PubMed
new_grant_info = fetch_grant_info_from_pubmed(pmid)         df.at[idx,
"Grant information"] = new_grant_info

    # Save the updated DataFrame
df.to_csv(output_path, index=False)
    print(f"\nFinished! Updated CSV saved to: {output_path}")

if __name__ == "__main__":
    # Make sure these paths match the location of your CSV files
input_csv = "./output_csvs/cleaned-latest-data-articles.csv"
output_csv = "./output_csvs/cleaned-latest-data-articles-
withgrants.csv"     update_grant_info(input_csv, output_csv)
```

Appendix H : Fill Missing Value (8-Fill-missing-PMIDs)

```
 import time import pandas as pd
from Bio import Entrez from
urllib.error import HTTPError
import logging

# Configure logging to write to a file and also output to console
logging.basicConfig(     level=logging.DEBUG,
    format="%(asctime)s %(levelname)s: %(message)s",
handlers=[
        logging.FileHandler("pmid_populate.log", mode="w"),
logging.StreamHandler()
```

```
    ]
)
logger = logging.getLogger()

# Configure PubMed API
Entrez.email = "pmohunsingh@gmail.com"
Entrez.api_key = "0d1dae6fef8957766aba2af30daebadc6a0a"

# Define placeholders for missing PMID values
PLACEHOLDERS = {"", "unavailable", "not available", "na", "none", "nan"}

def search_pubmed_for_pmid_by_title(title):
    """
    Searches PubMed using the article title only.
    Returns the first matching PMID as a string, or None if no result.
Logs details to help diagnose why no PMID was found.
    """
    title = str(title).strip()     if not title:
logger.debug("Empty title provided; skipping search.")
return None

    query = f"\"{title}\"[Title]"
    logger.info(f"Searching    PubMed    for    title:    {title}")
logger.debug(f"     Query: {query}")      try:           handle =
Entrez.esearch(db="pubmed", term=query, retmax=1)          record =
Entrez.read(handle)         handle.close()
        id_list = record.get("IdList", [])
if id_list:             pmid_found =
id_list[0]
            logger.info(f"    Found PMID: {pmid_found}")
return pmid_found         else:
            logger.info("    No PMID found for this query.")
logger.debug("    Full response record:")
logger.debug(record)             return None     except
HTTPError as e:
        logger.error(f"HTTPError for title '{title}': {e}")
return None     except Exception as e:         logger.error(f"Error
searching PubMed for title '{title}': {e}")         return None

def populate_missing_pmids(input_csv, output_csv):
    # Read the CSV
    df = pd.read_csv(input_csv)
    logger.info(f"Loaded CSV with shape: {df.shape}")

    # Check that the necessary columns exist; we require at least "Title"
and "PMID"     if "Title" not in df.columns or "PMID" not in df.columns:
        raise ValueError("CSV must contain 'Title' and 'PMID' columns.")

    # Loop through each row where PMID is missing or a placeholder
num_missing = 0     num_found = 0     for idx, row in
df.iterrows():
```

```
        current_pmid = str(row["PMID"]).strip().lower()
if current_pmid in PLACEHOLDERS:
            num_missing += 1
title = row["Title"]
            logger.info(f"\nProcessing row {idx} with missing PMID.")
logger.info(f"    Title: {title}")
            new_pmid = search_pubmed_for_pmid_by_title(title)
if new_pmid:                 logger.info(f"    Updating row
{idx}: Found PMID:
{new_pmid}")
                df.at[idx, "PMID"] = new_pmid
num_found += 1             else:
                logger.info(f"    No PMID found for row {idx} with title:
{title}")
            # Pause briefly to avoid hitting rate limits
time.sleep(1.0)

    logger.info(f"\nProcessed {num_missing} rows with missing PMID;
updated {num_found} of them.")

    # Save the updated DataFrame to a new CSV file
df.to_csv(output_csv, index=False)
    logger.info(f"Updated CSV saved to {output_csv}")

if __name__ == "__main__":
    input_csv = "./output_csvs/updated-data-grants.csv"
output_csv = "./output_csvs/updated-populated-missing-pmids.csv"
populate_missing_pmids(input_csv, output_csv)
```

Appendix I : Fill Missing DOIs

```
import time import urllib.parse
import requests import pandas as
pd import concurrent.futures from
Bio import Entrez from
urllib.error import HTTPError #
Global configuration and
placeholders
Entrez.email = "pmohunsingh@gmail.com"
Entrez.api_key = "0d1dae6fef8957766aba2af30daebadc6a0a"
PLACEHOLDERS = {"", "unavailable", "not available", "na", "none", "nan"}

##############################################
# 1. Fetch DOI from PubMed using PMID
############################################## def
fetch_doi_from_pubmed_by_pmid(row):
    """
    If the row has a valid PMID, use PubMed's EFetch to retrieve the DOI.
Returns the DOI as a string or None.
```

```
    """
    pmid = str(row.get("PMID", "")).strip().lower()
if not pmid or pmid in PLACEHOLDERS:        return
None    try:
        handle = Entrez.efetch(db="pubmed", id=pmid, retmode="xml")
record = Entrez.read(handle)        handle.close()        for
article in record.get("PubmedArticle", []):            pubdata =
article.get("PubmedData", {})            id_list =
pubdata.get("ArticleIdList", [])            for aid in id_list:
if aid.attributes.get("IdType") == "doi":
                    doi = str(aid).strip()
if doi:
                        print(f"PubMed: Found DOI {doi} for PMID {pmid}")
return doi
        print(f"PubMed: No DOI found for PMID {pmid}")
return None    except Exception as e:
        print(f"PubMed EFetch error for PMID '{pmid}': {e}")
return None

##############################################
# 2. Fetch DOI from CrossRef using title
############################################## def
fetch_doi_from_crossref(title):
    """
    Queries the CrossRef API with the article title.
    Returns the DOI as a string or None.
    """
    title = str(title).strip()    if not title:
print("CrossRef: Empty title provided; skipping search.")
return None
    base_url = "https://api.crossref.org/works"
params = {"query.title": title, "rows": 1}
    query_url = f"{base_url}?{urllib.parse.urlencode(params)}"
print(f"CrossRef: Querying DOI for title: {title}")
print(f"    URL: {query_url}")    try:
        response = requests.get(query_url, timeout=30)
        if response.status_code == 200:
            data = response.json()
            items = data.get("message", {}).get("items", [])
if items:
                doi = items[0].get("DOI")
if doi:
                    print(f"CrossRef: Found DOI {doi} for title: {title}")
return doi                else:
                    print("CrossRef: DOI not present in the first item.")
return None            else:                print("CrossRef: No items
returned for this title.")                return None        else:
print(f"CrossRef: HTTP error {response.status_code} for title:
{title}")
            return None
except Exception as e:
```

```
        print(f"CrossRef: Error fetching DOI for title '{title}': {e}")
return None

##############################################
# 3. Fetch DOI from OpenAlex using title
##############################################def
fetch_doi_from_openalex(title):
    """
    Queries the OpenAlex API with the article title.
    Returns the DOI as a string, or None.
    """
    title = str(title).strip()    if not title:
print("OpenAlex: Empty title provided; skipping search.")
return None
    encoded_title = urllib.parse.quote(title)
url =
f"https://api.openalex.org/works?filter=title.search:{encoded_title}&per_p
age=1"
    print(f"OpenAlex: Querying DOI for title: {title}")
print(f"     URL: {url}")     try:          response =
requests.get(url,  timeout=30)                    if
response.status_code == 200:
            data = response.json()
results = data.get("results", [])
if results:
                doi = results[0].get("doi")                if doi:
print(f"OpenAlex: Found DOI {doi} for title: {title}")
return doi                else:                   print("OpenAlex: DOI
not found in the first result.")                    return None
else:
                print("OpenAlex: No results returned for this title.")
return None        else:            print(f"OpenAlex: HTTP error
{response.status_code} for title:
{title}")
            return None    except Exception as e:
print(f"OpenAlex: Error fetching DOI for title '{title}': {e}")
return None

##############################################
# 4. Fetch DOI from Europe PMC using title
##############################################def
fetch_doi_from_europepmc(title):
    """
    Queries the Europe PMC API with the article title.
    Returns the DOI as a string, or None.
    """
    title = str(title).strip()    if not title:
print("Europe PMC: Empty title provided; skipping search.")
return None    query = f'TITLE:"{title}"'
    base_url = "https://www.ebi.ac.uk/europepmc/webservices/rest/search"
params = {"query": query, "format": "json", "resultType": "core",
"pageSize": 1}
```

```
    url = base_url + "?" + urllib.parse.urlencode(params)
print(f"Europe PMC: Querying DOI for title: {title}")
print(f"    URL: {url}")     try:
        response = requests.get(url, timeout=30)
if response.status_code == 200:
            data = response.json()
            results = data.get("resultList", {}).get("result", [])
if results:
                doi = results[0].get("doi")                 if
doi:                     print(f"Europe PMC: Found DOI {doi} for
title:
{title}")
                    return doi                 else:
print("Europe PMC: DOI not found in the first result.")
                    return None
else:
                print("Europe PMC: No results returned for this title.")
return None         else:             print(f"Europe PMC: HTTP error
{response.status_code} for title: {title}")             return None
except Exception as e:
        print(f"Europe PMC: Error fetching DOI for title '{title}': {e}")
return None
##############################################
# 5. Combined function to fetch DOI for a row
##############################################def
fetch_doi_for_row(row):
    """
    Attempts to fetch a DOI for a given row by trying several data
sources:
1. If PMID is present, use PubMed EFetch.
2. Else, try CrossRef.
3. Else, try OpenAlex.
4. Else, try Europe PMC.
    Returns the first DOI found or None if no source returns a DOI.
    """
    # First, try PubMed using the PMID (if available)
doi = fetch_doi_from_pubmed_by_pmid(row)     if doi:
        return doi

    # Otherwise, use the article title     title = row.get("Title",
"")     if not title:         print("No title available for row;
cannot search for DOI.")         return None

    # Try CrossRef
    doi = fetch_doi_from_crossref(title)
if doi:
        return doi

    # Try OpenAlex
    doi = fetch_doi_from_openalex(title)
if doi:
```

```
        return doi

    # Try Europe PMC
    doi = fetch_doi_from_europepmc(title)
return doi

##############################################
# 6. Main function to update missing DOIs and URLs
##############################################def
update_missing_dois(input_csv, output_csv, max_workers=10):
    """
    Reads the input CSV, identifies rows where the DOI is missing or a
placeholder,     and attempts to fetch the DOI using multiple external
resources.    If a DOI is found, updates the DOI column and constructs
the URL column as
    "http://dx.doi.org/<DOI>".
    Uses concurrent processing for efficiency.
    """
    df = pd.read_csv(input_csv)
    print(f"Loaded CSV with shape: {df.shape}")
    # Ensure required columns exist.     if "Title" not in df.columns
or "DOI" not in df.columns:         raise ValueError("CSV must contain
'Title' and 'DOI' columns.")     if "URL" not in df.columns:
        df["URL"] = ""

    # Identify rows with missing/placeholder DOI values.
    missing_mask = df["DOI"].astype(str).str.lower().isin(PLACEHOLDERS)
missing_rows = df.loc[missing_mask]
    print(f"Found {len(missing_rows)} rows with missing or placeholder
DOI.")

    results = {}
    with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as
executor:
        future_to_idx = {executor.submit(fetch_doi_for_row, row): idx
for idx, row in missing_rows.iterrows()}         for future in
concurrent.futures.as_completed(future_to_idx):             idx =
future_to_idx[future]             try:                 doi =
future.result()                 results[idx] = doi                 if
doi:
                    print(f"Row {idx}: Found DOI: {doi}")
else:
                    print(f"Row {idx}: No DOI found.")
except Exception as exc:
                print(f"Row {idx} generated an exception: {exc}")

    # Update DataFrame with the fetched DOIs and update the URL field
updated_count = 0     for idx, doi in results.items():         if doi
and doi.lower() not in PLACEHOLDERS:             df.at[idx, "DOI"] =
doi
```

```
            current_url = str(df.at[idx, "URL"]).strip().lower()
if current_url in PLACEHOLDERS:                 df.at[idx,
"URL"] = "http://dx.doi.org/" + doi             updated_count +=
1

    print(f"\nUpdated DOI for {updated_count} rows out of
{len(missing_rows)} missing DOI rows.")
df.to_csv(output_csv, index=False)     print(f"Updated
CSV saved to {output_csv}")

##############################################
# 7. EXECUTION
############################################## if __name__ ==
"__main__":     input_csv = "./output_csvs/latest-populated-
missing-pmids.csv"     output_csv = "./output_csvs/latest-
populated-with-dois.csv"     update_missing_dois(input_csv,
output_csv, max_workers=10)
```

Appendix : J : Fill Missing Urls (10-Fill-missing-urls)

```
import os import
time import re
import pandas as pd
import urllib.parse
import logging
from selenium import webdriver
from selenium.webdriver.chrome.service import Service from
selenium.webdriver.chrome.options import Options from
selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait, Select
from selenium.webdriver.support import expected_conditions as EC
from webdriver_manager.chrome import ChromeDriverManager from
bs4 import BeautifulSoup from rapidfuzz import fuzz, process

# ---------------------- CONFIG ---------------------- #
INPUT_CSV = "./output_csvs/script-12-dois.csv"
OUTPUT_CSV = "./output_csvs/script-13-ej.csv"
SAFETYLIT_ADV_SEARCH_URL =
"https://www.safetylit.org/citations/index.php?fuseaction=citations.archiv
esearch"
TOTAL_PAGES = 291  # For testing
LOG_FILE = "./logs/export_recs_url_scraper.log"
FUZZY_THRESHOLD = 75

os.makedirs("./logs", exist_ok=True)

logging.basicConfig(
filename=LOG_FILE,     filemode="w",
    format="%(asctime)s - %(levelname)s - %(message)s",
level=logging.DEBUG
```

```
)

# ---------------------- SELENIUM SETUP ---------------------- #
chrome_options = Options()
chrome_options.add_argument("--headless")
chrome_options.add_argument("--disable-gpu")
chrome_options.add_argument("--no-sandbox") service
= Service(ChromeDriverManager().install())
driver = webdriver.Chrome(service=service, options=chrome_options) wait
= WebDriverWait(driver, 30)

# ---------------------- SEARCH ---------------------- # def
perform_advanced_search():
driver.get(SAFETYLIT_ADV_SEARCH_URL)
    wait.until(EC.visibility_of_element_located((By.NAME,
"find_11"))).send_keys("self harm")
wait.until(EC.visibility_of_element_located((By.NAME,
"find_12"))).send_keys("suicide")
    Select(wait.until(EC.presence_of_element_located((By.NAME,
"from")))).select_by_value("2005")
    Select(wait.until(EC.presence_of_element_located((By.NAME,
"to")))).select_by_value("2014")     for checkbox in
driver.find_elements(By.NAME, "categories[]"):         if not
checkbox.is_selected():             checkbox.click()
    wait.until(EC.element_to_be_clickable((By.XPATH,
"//input[@type='submit']"))).click()
    wait.until(lambda d: "citations.summaries_performance" in
d.current_url)     return
ensure_display_count(driver.current_url, "100")

def ensure_display_count(url, count="100"):
parsed = urllib.parse.urlparse(url)     qs =
urllib.parse.parse_qs(parsed.query)
qs["display_count"] = [count]
    new_query = urllib.parse.urlencode(qs, doseq=True)
    return urllib.parse.urlunparse(parsed._replace(query=new_query))

# ---------------------- SCRAPING ---------------------- #
def extract_urls_and_titles_from_tr_blocks():     soup =
BeautifulSoup(driver.page_source, "html.parser")     table
= soup.select_one("#current_abstracts")     rows =
table.find_all("tr") if table else []

    article_data = {}     for
i in range(len(rows)):
        tr = rows[i]
        checkbox = tr.find("input", {"name": "citationIds[]"})
if checkbox:             article_id = checkbox.get("value")
link = tr.find("a", href=True)             if link:
title = link.get_text(strip=True)                 full_url =
```

```
urllib.parse.urljoin("https://www.safetylit.org/citations/index.php",
link["href"])
                article_data[title] = full_url
                logging.debug(f"✔ Extracted: '{title}' → {full_url}")
logging.info(f"☑ Extracted {len(article_data)} article links from
page.")
    return article_data

def paginate_and_extract_all_urls(base_url):
    all_title_url_map = {}

    for page in range(1, TOTAL_PAGES + 1):         page_url =
ensure_display_count(f"{base_url}&current_page={page}",
"100")
        logging.info(f"🌐 Processing page {page} of {TOTAL_PAGES}")
        try:
driver.get(page_url)
time.sleep(3)
            wait.until(EC.presence_of_element_located((By.NAME,
"export_recs")))
            page_map = extract_urls_and_titles_from_tr_blocks()
all_title_url_map.update(page_map)         except Exception as
e:
            logging.error(f"❌ Failed on page {page}: {e}")

    logging.info(f"📦 Total articles collected:
{len(all_title_url_map)}")     return all_title_url_map

# ---------------------- MATCHING ---------------------- #
def normalize(text):     if pd.isna(text):
        return ""
    text = re.sub(r'\s+', ' ', text)
    return re.sub(r'[^\w\s]', '', str(text)).lower().strip()

def match_urls_to_titles(df, scraped_map):
    df["URL"] = ""
    normalized_scraped = {normalize(k): v for k, v in scraped_map.items()}

    for idx, row in df.iterrows():
title = row["Title"]         norm_title
= normalize(title)
        match = process.extractOne(norm_title, normalized_scraped.keys(),
scorer=fuzz.token_sort_ratio)         if match and match[1] >=
FUZZY_THRESHOLD:
            matched_url = normalized_scraped[match[0]]
df.at[idx, "URL"] = matched_url
            logging.info(f"✅ [{match[1]}%] '{title}' matched with
'{match[0]}' → {matched_url}")
else:
```

```
            top_matches = process.extract(norm_title,
normalized_scraped.keys(), scorer=fuzz.token_sort_ratio, limit=3)
logging.warning(f"❌ No match for: '{title}' (normalized:
'{norm_title}'). Top suggestions: {top_matches}")
return df

# ---------------------- MAIN ---------------------- #
if __name__ == "__main__":     try:         results_url
= perform_advanced_search()
        title_url_map = paginate_and_extract_all_urls(results_url)

        df = pd.read_csv(INPUT_CSV)
        df["Title"] = df["Title"].astype(str)

        updated_df = match_urls_to_titles(df, title_url_map)
updated_df.to_csv(OUTPUT_CSV, index=False, encoding="utf-8")
logging.info(f"☑ URLs ✅dded. Output saved to {OUTPUT_CSV}")     except
Exception as e:         logging.error("❌ Script failed: " + str(e))
finally:
        driver.quit()
```

Appendix K : Duplicate Flaggers

```
from sentence_transformers import SentenceTransformer, util
import pandas as pd import numpy as np

from bs4 import BeautifulSoup
import html import pandas as
pd

def flag_semantically_translated_articles(df, start_row=18000,
end_row=24000, threshold=0.90):
    """
    Flags articles with semantically similar titles (possible
translations) in a given DataFrame slice.

    Parameters:
        df (pd.DataFrame): Original DataFrame with a 'Title' column.
        start_row (int): Starting row index for the slice.
end_row (int): Ending row index (exclusive) for the slice.
        threshold (float): Cosine similarity threshold to consider two
titles as semantically similar.

    Returns:
        pd.DataFrame: Modified slice with 'is_semantically_translated'
column added.
```

```
    """

    # Step 1: Extract and clean the slice
    subset = df.iloc[start_row:end_row].copy().reset_index(drop=True)
subset = subset[subset['Title'].notna() & (subset['Title'].str.strip() !=
"")].reset_index(drop=True)

    # Step 2: Load multilingual model
    model = SentenceTransformer('distiluse-base-multilingual-cased-v1')

    # Step 3: Encode titles
titles = subset['Title'].tolist()
    embeddings = model.encode(titles, convert_to_tensor=True,
show_progress_bar=True)

    # Step 4: Compute cosine similarity matrix
cos_sim_matrix = util.pytorch_cos_sim(embeddings,
embeddings).cpu().numpy()

    # Step 5: Flag rows with high similarity (excluding self-comparisons)
is_translated = []     for i in range(len(subset)):
        sim_scores = cos_sim_matrix[i]
sim_scores[i] = 0  # Ignore self
        is_translated.append(np.any(sim_scores > threshold))

    # Step 6: Add flag
    subset['is_semantically_translated'] = is_translated

    return subset


def flag_reissued_articles(df, start_row=18000, end_row=24000):
    """
    Flags reissued articles: same Title but different Journal, Volume,
Number, or Pages.

    Parameters:
        df (pd.DataFrame): Full dataset.
start_row (int): Start index of slice.
        end_row (int): End index of slice (exclusive).

    Returns:
        pd.DataFrame: Subset with 'is_reissued' flag.
    """
    subset = df.iloc[start_row:end_row].copy().reset_index(drop=True)
subset = subset[subset['Title'].notna()].reset_index(drop=True)

    # Normalize comparison fields
    compare_cols = ['Journal', 'Volume', 'Number', 'Pages']
    subset[compare_cols] = subset[compare_cols].fillna("").astype(str)
```

```
    # Group by title and flag groups with differing metadata
is_reissued = []

    grouped = subset.groupby('Title')
for idx, row in subset.iterrows():
        group = grouped.get_group(row['Title'])
        # If more than one unique set of metadata exists within the group
unique_sets = group[compare_cols].drop_duplicates()         if
len(unique_sets) > 1:             is_reissued.append(True)         else:
            is_reissued.append(False)

    subset['is_reissued'] = is_reissued
return subset


def flag_responses_or_commentaries(df, start_row=18000, end_row=24000):
    """
    Flags articles that are likely responses or commentaries based on
title keywords.

    Parameters:
        df (pd.DataFrame): Full DataFrame.
        start_row (int): Start index of row slice.
        end_row (int): End index (exclusive).

    Returns:
        pd.DataFrame: Subset with 'is_commentary_or_response' flag.
    """
    subset = df.iloc[start_row:end_row].copy().reset_index(drop=True)
subset = subset[subset['Title'].notna()].reset_index(drop=True)

    keywords = [
        "response to", "reply to", "commentary", "editorial",
        "discussion", "letter to the editor", "rebuttal",
        "review of", "reflection on", "analysis of",
        "critique of", "perspective on", "viewpoint"
    ]

    title_lower = subset['Title'].str.lower().fillna("")
pattern = '|'.join(keywords)
subset['is_commentary_or_response'] =
title_lower.str.contains(pattern, regex=True)

    return subset



def flag_html_duplicate_abstracts(df, start_row=18000, end_row=24000):
```

```
    """
    Flags articles with duplicate abstracts that differ only due to HTML
artifacts.

    Parameters:
        df (pd.DataFrame): Full DataFrame with an 'Abstract' column.
start_row (int): Start of row slice.
        end_row (int): End of row slice (exclusive).

    Returns:
        pd.DataFrame: Subset with 'is_html_duplicate_abstract' flag.
    """
    # Step 1: Slice and clean
    subset = df.iloc[start_row:end_row].copy().reset_index(drop=True)
subset = subset[subset['Abstract'].notna()].reset_index(drop=True)

    # Step 2: Clean HTML from abstracts
def clean_html(text):
        text = html.unescape(text)                          # Convert
entities
        text = BeautifulSoup(text, "html.parser").get_text()  # Strip tags
text = text.lower().strip()                             # Normalize
text = ' '.join(text.split())                           # Remove extra
whitespace        return text

    subset['cleaned_abstract'] = subset['Abstract'].apply(clean_html)

    # Step 3: Flag duplicates based on cleaned abstract
subset['is_html_duplicate_abstract'] =
subset.duplicated(subset=['cleaned_abstract'], keep=False)

    return subset
```

Appendix L : EDA Scripts

Co-occurrence Matrix for Top 20 Keywords in Abstracts

```
import pandas as pd
import numpy as np
import re
from collections import Counter
from itertools import combinations
import matplotlib.pyplot as plt
import seaborn as sns
def get_cooccurrence_matrix(texts, top_n=20):
        """
texts.  Generate a co-occurrence matrix for the most frequent words in the given list of
        Parameters:
        - texts: List of strings (article titles)
        - top_n: Number of most frequent words to include in the matrix
        Returns:
        - cooccurrence_df: DataFrame representing the co-occurrence matrix
        """
        # Tokenize and extract words (min length = 4 to filter stop words)
        words_list = []
        for text in texts:
letters words = re.findall(r'\b[a-zA-Z]{4,}\b', str(text).lower())  # Only words with 4+
        filtered_words = [word for word in words if word not in stop_words]
        words_list.append(filtered_words)
        # Flatten the list and get most common words
        all_words = [word for sublist in words_list for word in sublist]
```

```
        top_words = [word for word, _ in Counter(all_words).most_common(top_n)]
        # Initialize co-occurrence dictionary
        cooccurrence = {word: Counter() for word in top_words}
        # Count word co-occurrences within each title
        for words in words_list:
        filtered_words = [word for word in words if word in top_words]
        for word1, word2 in combinations(set(filtered_words), 2):
                cooccurrence[word1][word2] += 1
                cooccurrence[word2][word1] += 1
        # Convert to DataFrame
        cooccurrence_df = pd.DataFrame.from_dict(cooccurrence, orient="index").fillna(0)
        return cooccurrence_df
# Load the dataset (replace with actual DataFrame)
df_heat = df_work.copy()
# Get co-occurrence matrix
cooccurrence_df = get_cooccurrence_matrix(df_heat["Abstract"].dropna(), top_n=20)
# Plot heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(cooccurrence_df, cmap="Reds", annot=True, fmt=".0f")
plt.title("Co-Occurrence Matrix of Top 20 Keywords in Abstract")
plt.show()
```

Word2Vec Embeddings (UMAP)

```
from gensim.models import Word2Vec
from gensim.utils import simple_preprocess
from nltk.corpus import stopwords import
nltk

corpus = df_heat model = Word2Vec(sentences=corpus, vector_size=50, window=5,
min_count=2, workers=4)  model.save("word2vec.model") print("Word2Vec model trained
and saved with stopword removal!")

import umap import
matplotlib.pyplot as plt
import numpy as np
from gensim.models import Word2Vec
  words = [word for word in model.wv.index_to_key if len(word) >
2][:100]
```

```
vectors = np.array([model.wv[word] for word in words]) if vectors.size == 0:
print("Error: No word vectors found. Check your corpus or word filtering.")
else:
   umap_reducer = umap.UMAP(n_components=2, random_state=42, n_neighbors=15, min_dist=0.1)
umap_vectors = umap_reducer.fit_transform(vectors)


   plt.figure(figsize=(12, 8))     for
i, word in enumerate(words):
      plt.scatter(umap_vectors[i, 0], umap_vectors[i, 1])
plt.annotate(word, (umap_vectors[i, 0], umap_vectors[i, 1]))
plt.title("Word2Vec Embeddings Visualization (UMAP) - Filtered Words")
plt.show()
```

Box Plot: Distribution of Publication Frequency

```
pip install plotly
```

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

df= pd.read_csv('safetylit_master_df_complete.csv')
```

```
year_stats= df[['year']].describe()
year_stats
```

	year
count	29005.000000
mean	2009.798690
std	2.873137
min	2005.000000
25%	2007.000000
50%	2010.000000
75%	2012.000000
max	2014.000000

```
# Mean for year
df['year'].mean()
```

```
2009.798689881055
```

```
# Box plot
plt.figure(figsize=(8,4))
sns.boxplot(x=df["year"], color= "mediumaquamarine")

plt.xlabel('year', fontsize= 11)
plt.xticks(ticks=range(df['year'].min(), df['year'].max() + 1), rotation=35)
plt.title("Distribution of Publication Frequency (2005-2014)")

plt.tight_layout
plt.show()
```

```
# Histogram for Number of Articles Published per Year
year_df= df['year']
plt.figure(figsize=(8,5))

histo= sns.histplot(year_df, bins=len(year_df.unique()), discrete= True, color='plum', edgecolor='white')

for patch in histo.patches:
    height = patch.get_height()
    histo.annotate(f'{int(height)}',
                (patch.get_x() + patch.get_width() / 2, height),
                ha='center', va='bottom',
                fontsize=10, color='navy')

plt.xticks(ticks=range(2005, 2015))
plt.xlabel('Year')
plt.ylabel('Number of Articles')
plt.title('Number of Articles by Year (2005-2014)')

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

Top 10 Languages % (Bar Graph)

```
language_count= df['languagename'].value_counts(normalize=True)*100
```

```
language_df= pd.DataFrame(language_count.head(10))
language_df.rename(columns={'proportion': 'percentage'}, inplace=True)
language_df
```

```
# Bar graph for top 10 languages
plt.figure(figsize=(9,5))

language_bar= language_df.plot(y= 'percentage', kind= 'bar', color= 'cornflowerblue', edgecolor='darkmagenta')

for p in language_bar.patches:
    height = p.get_height()
    language_bar.annotate(f'{height:.1f}%',
                (p.get_x() + p.get_width() / 2, height),
                ha='center', va='bottom',
                fontsize=10, color='darkblue')

plt.xlabel('Language', fontsize=11)
plt.ylabel('Percentage of Articles', fontsize= 11)
plt.title('Top 10 Languages % (2005-2014)', fontsize= 14)
plt.xticks(rotation=45)

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

Distribution of Research Methodologies (Pie Chart)

```
# Pie chart for distribution of research methodologies
import plotly.express as px

method_summary = df['methodologies'].value_counts().reset_index()
method_summary.columns = ['methodologies', 'count']

colors= ['#ffcc99', 'paleturquoise', 'navy']

fig = px.pie(
    method_summary,
    names= "methodologies",
    values= "count",
    title= "Distribution of Research Methodology (2005–2014)",
    color_discrete_sequence= colors
)

fig.update_traces(
    textinfo='label+percent',
    pull=[0.06] * len(method_summary)
)

fig.update_layout(
    height=500,
    width=700,
    showlegend=True,
    title_font=dict(size=18, family="Arial", color="black"),
    margin=dict(l=50, r=50, t=80, b=50)
)

fig.show()
```

Research Methodologies Distribution by Year (Stacked Bar Graph)

```
# Table for Qualitative, Quantitative or Both by each year
df_subset1 = df[['year', 'methodologies']]

def count_value_by_year(df_subset1, year_col, value_col, target_val):

    if not isinstance(df_subset1, pd.DataFrame):
        raise ValueError("data must be a pandas df.")
    if not all(col in df_subset1.columns for col in [year_col, value_col]):
         raise ValueError("year_col and value_col must be columns in df.")
    if not isinstance(target_val, list):
        raise ValueError("target_val must be a list.")

    filtered_data = df_subset1[df_subset1[value_col].isin(target_val)]

    grouped_counts = filtered_data.groupby([year_col, value_col]).size().unstack(fill_value=0)

    return grouped_counts

target_val = ['Qualitative', 'Quantitative', 'Qualitative; Quantitative']
method_table = count_value_by_year(df_subset1, 'year', 'methodologies', target_val)
method_table
```

```
# Switch positions of columns
methodology_table = method_table[['Qualitative', 'Quantitative', 'Qualitative; Quantitative']]
methodology_table
```

```
# Stacked bar graph for methodologies distribution by year
plt.figure(figsize=(18, 10))

stacked_bar = methodology_table.plot(kind='bar', stacked=True, color=['#ffcc99', 'paleturquoise', 'navy'], width=0.7)

for container in stacked_bar.containers:
    for bar in container:
        height = bar.get_height()
        x = bar.get_x() + bar.get_width() / 2
        y = bar.get_y()

        if height == 0:
            # Find the top of the full stack at this x position
            full_stack_height = y
            for other_container in stacked_bar.containers:
                for other_bar in other_container:
                    if other_bar.get_x() == bar.get_x():
                        full_stack_height = max(full_stack_height, other_bar.get_y() + other_bar.get_height())

            stacked_bar.text(
                x,
                full_stack_height + 1,  # Just above the full stack
                '0',
                ha='center', va='bottom', fontsize=9, color='black'
            )

        elif height < 20:
            stacked_bar.text(
                x,
                y + height + 1,
                f'{int(height)}', ha='center', va='bottom', fontsize=9, color='black'
            )
```

```
        else:
            stacked_bar.text(
                x,
                y + height / 2,
                f'{int(height)}', ha='center', va='center', fontsize=9, color='black'
            )

plt.xlabel('Year', fontsize=12)
plt.ylabel('Methodology Distribution', fontsize=12)
plt.xticks(rotation=50)

plt.tight_layout()
plt.title('Research Methodology by Year (2005-2014)', fontsize=14)
```

Top 10 Suggested Prevention Measures (Horizontal Bar Graph)

```
import pandas as pd import matplotlib.pyplot
        as plt import seaborn as sns from
        wordcloud import WordCloud from
        collections import Counter

        # Load dataset file_path = "./output_csvs/cleaned-latest-
        data-articles.csv" df = pd.read_csv(file_path)

        # Convert Year to numeric df['year'] =
        pd.to_numeric(df['year'], errors='coerce')

        # Top 10 Suggested Suicide Prevention Measures df['preventionmeasures'] =
        df['preventionmeasures'].str.strip() prevention_counts =
        df['preventionmeasures'].dropna().value_counts().head(10)

        plt.figure(figsize=(16, 10)) # Adjusted figure size for better readability
        ax = sns.barplot(y=prevention_counts.index, x=prevention_counts.values, palette='Reds_r')
        plt.xlabel('Count', fontsize=14)
        plt.ylabel('Prevention Strategies', fontsize=14, labelpad=15) # Adjusted padding
        plt.title('Top 10 Most Suggested Suicide Prevention Measures', fontsize=16)
        plt.xticks(fontsize=12) plt.yticks(fontsize=10) plt.subplots_adjust(left=0.5,
        right=0.95) # Adjusted padding to prevent cutoff

        # Set x-axis scale to increments of 10000
        max_value = max(prevention_counts.values)
        plt.xticks(range(0, max_value + 10000, 10000))

        # Adjust the top-most y-axis label to fit in two lines ax.set_yticklabels([label.get_text().replace(' including ',
        '\nincluding ', 1) if i == 0 else label.get_text() for i, label in enumerate(ax.get_yticklabels())])
```

```
# Ensure value labels are placed outside the bars for better
readability  for p in ax.patches:      ax.annotate(str(int(p.get_width())),
            (p.get_width() + 1000, p.get_y() + p.get_height() / 2.),
ha='left', va='center', fontsize=10, color='black', fontweight='bold')

plt.show()
```

Most Studied Research Categories (Horizontal Bar Graph)

```
import pandas as pd  import
matplotlib.pyplot as plt
import seaborn as sns


# Load dataset  file_path = "./output_csvs/cleaned-latest-
data-articles.csv"  df = pd.read_csv(file_path)

# Ensure 'categories' column exists  if 'categories' not in
df.columns:      raise ValueError("Dataset is missing the
'categories' column.")

# Count occurrences of each category
category_counts = df['categories'].dropna().value_counts()

# Select top categories
top_categories = category_counts.nlargest(6)

# Modify third value to wrap into two lines if it's too long
if len(top_categories.index) > 2:      top_categories.index
= list(top_categories.index[:2]) + [
top_categories.index[2][:len(top_categories.index[2])//2]
+ "\n" +
top_categories.index[2][len(top_categories.index[2])//2:],
top_categories.index[3][:len(top_categories.index[3])//2]
+ "\n" +
top_categories.index[3][len(top_categories.index[3])//2:]
] + list(top_categories.index[4:])

# --- Visualization: Funnel Chart of Research Categories ---
fig, ax = plt.subplots(figsize=(10, 6))  y_pos =
range(len(top_categories), 0, -1)
```

```
# Plot funnel using a bar chart approach ax.barh(y_pos, top_categories.values, align='center',
color=sns.color_palette("coolwarm", len(top_categories))) ax.set_yticks(y_pos)
ax.set_yticklabels(top_categories.index, fontsize=12)
ax.invert_yaxis() # Reverse order

# Labels plt.xlabel("Number of Mentions in Research", fontsize=12,
fontweight='bold') plt.ylabel("Research Categories", fontsize=12, fontweight='bold')
plt.title("Funnel of Most Studied Research Categories", fontsize=14, fontweight='bold')


# Annotate values for i, v in enumerate(top_categories.values):
ax.text(v + 2, y_pos[i], str(v), fontsize=12, verticalalignment='center')
  plt.show()
```

Top 20 Countries by Research Publications Overtime

```
# Filter only top 20 countries df_top20 =
df_clean[df_clean['country'].isin(top_20_countries)]

# Group by Year and Country df_line
= (
    df_top20.groupby(['year', 'country'])
    .size()
    .reset_index(name='Count')
)

import plotly.express as px

fig = px.line(
df_line,     x='year',
y='Count',
color='country',
    title='Top 20 Countries by Research Publications Over Time',
markers=True
)

fig.update_layout(
xaxis_title='Year',
    yaxis_title='Number of Publications',
legend_title='Country',     height=600
)

fig.show()
```

Appendix M: Modeling Scripts

Model Development RQ1(ML) Script

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report

pipeline_lr = Pipeline([
    ('tfidf', TfidfVectorizer(max_features=5000)),
    ('clf', LogisticRegression(max_iter=1000, multi_class='multinomial'))
])

param_grid_lr = {'clf__C': [0.1, 1, 10]}
grid_lr = GridSearchCV(pipeline_lr, param_grid_lr, cv=5, scoring='accuracy')
grid_lr.fit(X_train_ml, y_train)

print("LogReg Best Params:", grid_lr.best_params_)
print(classification_report(y_val, grid_lr.predict(X_val_ml)))
```

```
LogReg Best Params: {'clf__C': 1}
              precision    recall  f1-score   support

        2005       0.20      0.29      0.24       461
        2006       0.20      0.15      0.17       541
        2007       0.15      0.10      0.12       534
        2008       0.17      0.13      0.15       554
        2009       0.13      0.12      0.12       576
        2010       0.16      0.14      0.15       603
        2011       0.17      0.13      0.14       606
        2012       0.21      0.18      0.19       581
        2013       0.22      0.19      0.20       586
        2014       0.28      0.53      0.37       762

    accuracy                           0.20      5804
   macro avg       0.19      0.20      0.19      5804
weighted avg       0.19      0.20      0.19      5804
```

Setup:

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Load your DataFrame
df = master_df.copy()

# For ML models (lemmatized text)
df['abstract_text'] = df['abstract_tokens_lemmatized'].apply(lambda x: ' '.join(eval(x)))

# For DL models (raw abstract)
df['abstract_text_raw'] = df['abstract']

# Prepare target
df['year'] = df['year'].astype(int)
X_ml = df['abstract_text']
X_dl = df['abstract_text_raw']
y = df['year']

# Train/Val/Test Split
X_train_ml, X_temp_ml, y_train, y_temp = train_test_split(X_ml, y, test_size=0.4, stratify=y, random_state=42)
X_val_ml, X_test_ml, y_val, y_test = train_test_split(X_temp_ml, y_temp, test_size=0.5, stratify=y_temp, random_state=42)

X_train_dl, X_temp_dl = train_test_split(X_dl, test_size=0.4, stratify=y, random_state=42)
X_val_dl, X_test_dl = train_test_split(X_temp_dl, test_size=0.5, stratify=y_temp, random_state=42)
```

Model 1: Logistic Regression + TF-IDF

Model 2: Random Forest + TF-IDF

```
from sklearn.ensemble import RandomForestClassifier

pipeline_rf = Pipeline([
    ('tfidf', TfidfVectorizer(max_features=5000)),
    ('clf', RandomForestClassifier(random_state=42))
])

param_grid_rf = {
    'clf__n_estimators': [100, 200],
    'clf__max_depth': [None, 30]
}

grid_rf = GridSearchCV(pipeline_rf, param_grid_rf, cv=5, scoring='accuracy')
grid_rf.fit(X_train_ml, y_train)

print("RF Best Params:", grid_rf.best_params_)
print(classification_report(y_val, grid_rf.predict(X_val_ml)))
```

```
RF Best Params: {'clf__max_depth': None, 'clf__n_estimators': 200}
              precision    recall  f1-score   support

        2005       0.32      0.61      0.42       461
        2006       0.23      0.15      0.18       541
        2007       0.20      0.14      0.17       534
        2008       0.19      0.18      0.18       554
        2009       0.19      0.17      0.18       576
        2010       0.18      0.14      0.16       603
        2011       0.24      0.20      0.22       606
        2012       0.29      0.21      0.24       581
        2013       0.29      0.21      0.24       586
        2014       0.31      0.56      0.40       762

    accuracy                           0.26      5804
   macro avg       0.24      0.26      0.24      5804
weighted avg       0.25      0.26      0.24      5804
```

Model 3: SVM + TF-IDF

```
from sklearn.svm import SVC

pipeline_svm = Pipeline([
    ('tfidf', TfidfVectorizer(max_features=3000)),
    ('clf', SVC(kernel='linear'))
])

param_grid_svm = {'clf__C': [0.1, 1, 10]}
grid_svm = GridSearchCV(pipeline_svm, param_grid_svm, cv=5, scoring='accuracy')
grid_svm.fit(X_train_ml, y_train)

print("SVM Best Params:", grid_svm.best_params_)
print(classification_report(y_val, grid_svm.predict(X_val_ml)))
```

```
SVM Best Params: {'clf__C': 1}
              precision    recall  f1-score   support

        2005       0.20      0.30      0.24       461
        2006       0.18      0.16      0.17       541
        2007       0.14      0.11      0.13       534
        2008       0.17      0.14      0.15       554
        2009       0.12      0.11      0.12       576
        2010       0.16      0.16      0.16       603
        2011       0.19      0.15      0.17       606
        2012       0.18      0.15      0.17       581
        2013       0.21      0.17      0.19       586
        2014       0.30      0.46      0.36       762

    accuracy                           0.20      5804
   macro avg       0.19      0.19      0.18      5804
weighted avg       0.19      0.20      0.19      5804
```

Model Development RQ1(DL) Script

Setup:

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Load your DataFrame
df = master_df.copy()

# For ML models (lemmatized text)
df['abstract_text'] = df['abstract_tokens_lemmatized'].apply(lambda x: ' '.join(eval(x)))

# For DL models (raw abstract)
df['abstract_text_raw'] = df['abstract']

# Prepare target
df['year'] = df['year'].astype(int)
X_ml = df['abstract_text']
X_dl = df['abstract_text_raw']
y = df['year']

# Train/Val/Test Split
X_train_ml, X_temp_ml, y_train, y_temp = train_test_split(X_ml, y, test_size=0.4, stratify=y, random_state=42)
X_val_ml, X_test_ml, y_val, y_test = train_test_split(X_temp_ml, y_temp, test_size=0.5, stratify=y_temp, random_state=42)

X_train_dl, X_temp_dl = train_test_split(X_dl, test_size=0.4, stratify=y, random_state=42)
X_val_dl, X_test_dl = train_test_split(X_temp_dl, test_size=0.5, stratify=y_temp, random_state=42)
```

Model 1: BERT Embeddings + Logistic Regression

```
from transformers import BertTokenizer, BertModel
import torch
import numpy as np

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

def get_bert_embeddings(texts):
    embeddings = []
    for text in texts:
        inputs = tokenizer(text, return_tensors='pt', truncation=True, padding=True, max_length=512)
        with torch.no_grad():
            outputs = model(**inputs)
        cls_embed = outputs.last_hidden_state[:, 0, :].squeeze().numpy()
        embeddings.append(cls_embed)
    return np.array(embeddings)

X_embed_train = get_bert_embeddings(X_train_dl[:100])  # Slice for speed
y_train_dl = y[:100].values  # Match size

from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(max_iter=1000, multi_class='multinomial')
clf.fit(X_embed_train, y_train_dl)
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('logreg', LogisticRegression())
])

# Parameter grid for GridSearch
param_grid = {
    'logreg__C': [0.01, 0.1, 1, 10],  # Regularization strength
    'logreg__penalty': ['l1', 'l2'], #Regularization type
    'logreg__solver': ['liblinear', 'saga'],  # Solvers
    'logreg__max_iter': [100, 200, 1000]  # Max number of iterations
}

# Grid search with cross-validation
grid_search = GridSearchCV(pipeline, param_grid, cv=4, n_jobs=-1, verbose=2, error_score='raise')
grid_search.fit(X_embed_train, y_train_dl)

print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)
```

```
from sklearn.metrics import classification_report

# BERT embeddings for test data
X_embed_test = get_bert_embeddings(X_test_dl[:100])
y_test_dl = y_test[:100].values

# Predictions using trained model
y_pred = clf.predict(X_embed_test)

# Print classification report
print(classification_report(y_test_dl, y_pred, digits=4))
```

```
              precision    recall  f1-score   support

        2005     0.0000    0.0000    0.0000        12
        2006     0.0000    0.0000    0.0000        10
        2007     0.0000    0.0000    0.0000         6
        2008     0.1667    0.1667    0.1667        12
        2009     0.1429    0.0909    0.1111        11
        2010     0.1000    0.1429    0.1176         7
        2011     0.0000    0.0000    0.0000         6
        2012     0.1429    0.1000    0.1176        10
        2013     0.1667    0.3636    0.2286        11
        2014     0.1892    0.4667    0.2692        15

    accuracy                         0.1600       100
   macro avg     0.0908    0.1331    0.1011       100
weighted avg     0.1037    0.1600    0.1177       100
```

Model 2: LSTM on Tokenized Abstracts

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from sklearn.preprocessing import LabelEncoder

tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(X_dl)
sequences = tokenizer.texts_to_sequences(X_dl)
X_pad = pad_sequences(sequences, maxlen=300)

le = LabelEncoder()
y_enc = le.fit_transform(y)

X_train_lstm, X_val_lstm, y_train_lstm, y_val_lstm = train_test_split(X_pad, y_enc, test_size=0.2)

model = Sequential([
    Embedding(input_dim=10000, output_dim=128, input_length=300),
    LSTM(64),
    Dense(len(le.classes_), activation='softmax')
])

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X_train_lstm, y_train_lstm, validation_data=(X_val_lstm, y_val_lstm), epochs=5)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove
  warnings.warn(
Epoch 1/5
726/726 ━━━━━━━━━━━━━━━━━━━━ 166s 225ms/step - accuracy: 0.1575 - loss: 2.2270 - val_accuracy: 0.2223 - val_loss: 2.0962
Epoch 2/5
726/726 ━━━━━━━━━━━━━━━━━━━━ 165s 228ms/step - accuracy: 0.2751 - loss: 1.9778 - val_accuracy: 0.2331 - val_loss: 2.0571
Epoch 3/5
726/726 ━━━━━━━━━━━━━━━━━━━━ 164s 225ms/step - accuracy: 0.3812 - loss: 1.7175 - val_accuracy: 0.2324 - val_loss: 2.1489
Epoch 4/5
726/726 ━━━━━━━━━━━━━━━━━━━━ 203s 227ms/step - accuracy: 0.4938 - loss: 1.4563 - val_accuracy: 0.2409 - val_loss: 2.3322
Epoch 5/5
726/726 ━━━━━━━━━━━━━━━━━━━━ 201s 225ms/step - accuracy: 0.5711 - loss: 1.2393 - val_accuracy: 0.2402 - val_loss: 2.5708
<keras.src.callbacks.history.History at 0x7fd08cbddc10>
```

Model Development RQ2(ML) Script

Setup:

```python
import pandas as pd
from sklearn.model_selection import train_test_split

# Dataset to use
df = master_df.copy()

# Create gender label from abstract
df_male = df[df['abstract'].str.contains(r'\b(male|man|men|boy|boys)\b', case=False, na=False)].copy()
df_male['gender_label'] = 1

df_female = df[df['abstract'].str.contains(r'\b(female|woman|women|girl|girls)\b', case=False, na=False)].copy()
df_female['gender_label'] = 0

df_gender = pd.concat([df_male, df_female]).drop_duplicates()

# Combine relevant fields into a single text column
df_gender['text'] = df_gender['title'].fillna('') + ' ' + df_gender['journal'].fillna('') + ' ' + df_gender['abstract'].fillna('

# Prepare features and target
X = df_gender['text']
y = df_gender['gender_label']

# Step 1: First split off 20% test
X_temp, X_test, y_temp, y_test = train_test_split(X, y, stratify=y, test_size=0.2, random_state=42)
```

```python
# Step 2: Split the remaining 80% into train (80%) and val (20%) => 64/16
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, stratify=y_temp, test_size=0.2, random_state=42)

# Final sizes:
# X_train: 64% of full data
# X_val:   16% of full data
# X_test:  20% of full data

# Optional: Print sizes
print("Train size:", len(X_train))
print("Validation size:", len(X_val))
print("Test size:", len(X_test))
```

```
Train size: 6479
Validation size: 1620
Test size: 2025
```

Model A1 — Logistic Regression + TF-IDF

```python
# Import necessary modules
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report
import joblib

# Define TF-IDF transformer
text_transformer = TfidfVectorizer(stop_words='english', ngram_range=(1, 2), max_df=0.9, min_df=2)
preprocessor = ColumnTransformer([("text", text_transformer, "text")])

# Initialize classifier
lr_model = LogisticRegression(max_iter=1000, random_state=42)

# Create pipeline
pipeline_lr = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", lr_model)
])

# Cross-validation
cv_scores = cross_val_score(pipeline_lr, X_train.to_frame(), y_train, cv=5, scoring="f1_macro")
print(f"Mean F1-score (CV): {cv_scores.mean():.4f}")
```

```python
# Grid search for best params
param_grid_lr = {
    "classifier__C": [0.1, 1, 10],
    "classifier__penalty": ["l2"]
}
grid_search_lr = GridSearchCV(pipeline_lr, param_grid_lr, cv=3, scoring="f1_macro", n_jobs=-1)
grid_search_lr.fit(X_train.to_frame(), y_train)

# Evaluation
best_lr_model = grid_search_lr.best_estimator_
print(f"Best Parameters: {grid_search_lr.best_params_}")
y_pred_lr = best_lr_model.predict(X_test.to_frame())
print(classification_report(y_test, y_pred_lr))

# Save model
joblib.dump(best_lr_model, "dev/best_logistic_regression.pkl")
print("Model saved.")
```

```
Mean F1-score (CV): 0.5590
Best Parameters: {'classifier__C': 0.1, 'classifier__penalty': 'l2'}
              precision    recall  f1-score   support

           0       0.57      0.71      0.63      1040
           1       0.59      0.44      0.50       985

    accuracy                           0.58      2025
   macro avg       0.58      0.57      0.57      2025
weighted avg       0.58      0.58      0.57      2025
```

Model A2 — Random Forest + TF-IDF

```python
# Import necessary modules
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report
import joblib

# Define TF-IDF transformer
text_transformer = TfidfVectorizer(stop_words='english', ngram_range=(1, 2), max_df=0.9, min_df=2)
preprocessor = ColumnTransformer([("text", text_transformer, "text")])

# Initialize classifier
rf_model = RandomForestClassifier(n_estimators=200, max_depth=20, random_state=42)

# Create pipeline
pipeline_rf = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", rf_model)
])

# Cross-validation
cv_scores = cross_val_score(pipeline_rf, X_train.to_frame(), y_train, cv=5, scoring="f1_macro")
print(f"Mean F1-score (CV): {cv_scores.mean():.4f}")
```

```python
# Grid search for best params
param_grid_rf = {
    "classifier__n_estimators": [100, 200, 300],
    "classifier__max_depth": [10, 20, None],
    "classifier__min_samples_split": [2, 5, 10]
}
grid_search_rf = GridSearchCV(pipeline_rf, param_grid_rf, cv=3, scoring="f1_macro", n_jobs=-1)
grid_search_rf.fit(X_train.to_frame(), y_train)

# Evaluation
best_rf_model = grid_search_rf.best_estimator_
print(f"Best Parameters: {grid_search_rf.best_params_}")
y_pred_rf = best_rf_model.predict(X_test.to_frame())
print(classification_report(y_test, y_pred_rf))

# Save model
joblib.dump(best_rf_model, "dev/best_random_forest.pkl")
print("Model saved.")
```

```
Mean F1-score (CV): 0.5231
Best Parameters: {'classifier__max_depth': None, 'classifier__min_samples_split': 10, 'classifier__n_estimators': 300}
              precision    recall  f1-score   support

           0       0.52      0.55      0.53      1040
           1       0.49      0.46      0.47       985

    accuracy                           0.51      2025
   macro avg       0.50      0.50      0.50      2025
weighted avg       0.50      0.51      0.50      2025
```

Model A3 — SVM + TF-IDF

```python
# Import necessary modules
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report
import joblib

# Define TF-IDF transformer
text_transformer = TfidfVectorizer(stop_words='english', ngram_range=(1, 2), max_df=0.9, min_df=2)
preprocessor = ColumnTransformer([("text", text_transformer, "text")])

# Initialize classifier
svm_model = SVC(kernel='linear', probability=True)

# Create pipeline
pipeline_svm = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", svm_model)
])

# Cross-validation
cv_scores = cross_val_score(pipeline_svm, X_train.to_frame(), y_train, cv=5, scoring="f1_macro")
print(f"Mean F1-score (CV): {cv_scores.mean():.4f}")
```

```python
# Grid search for best params
param_grid_svm = {
    "classifier__C": [0.1, 1, 10],
    "classifier__kernel": ["linear"]
}
grid_search_svm = GridSearchCV(pipeline_svm, param_grid_svm, cv=3, scoring="f1_macro", n_jobs=-1)
grid_search_svm.fit(X_train.to_frame(), y_train)

# Evaluation
best_svm_model = grid_search_svm.best_estimator_
print(f"Best Parameters: {grid_search_svm.best_params_}")
y_pred_svm = best_svm_model.predict(X_test.to_frame())
print(classification_report(y_test, y_pred_svm))

# Save model
joblib.dump(best_svm_model, "dev/best_svm.pkl")
print("Model saved.")
```

```
Mean F1-score (CV): 0.5455
Best Parameters: {'classifier__C': 1, 'classifier__kernel': 'linear'}
              precision    recall  f1-score   support

           0       0.52      0.53      0.52      1040
           1       0.50      0.50      0.50       985

    accuracy                           0.51      2025
   macro avg       0.51      0.51      0.51      2025
weighted avg       0.51      0.51      0.51      2025

Model saved.
```

Model A4 — Naive Bayes + TF-IDF

```
# Import necessary modules
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report
import joblib

# Define TF-IDF transformer
text_transformer = TfidfVectorizer(stop_words='english', ngram_range=(1, 2), max_df=0.9, min_df=2)
preprocessor = ColumnTransformer([("text", text_transformer, "text")])

# Initialize classifier
nb_model = MultinomialNB()

# Create pipeline
pipeline_nb = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", nb_model)
])

# Cross-validation
cv_scores = cross_val_score(pipeline_nb, X_train.to_frame(), y_train, cv=5, scoring="f1_macro")
print(f"Mean F1-score (CV): {cv_scores.mean():.4f}")
```

```
# Train and evaluate
pipeline_nb.fit(X_train.to_frame(), y_train)
y_pred_nb = pipeline_nb.predict(X_test.to_frame())
print(classification_report(y_test, y_pred_nb))

# Save model
joblib.dump(pipeline_nb, "dev/best_naive_bayes.pkl")
print("Model saved.")
```

```
Mean F1-score (CV): 0.4947
              precision    recall  f1-score   support

           0       0.50      0.65      0.56      1040
           1       0.45      0.30      0.36       985

    accuracy                           0.48      2025
   macro avg       0.48      0.48      0.46      2025
weighted avg       0.48      0.48      0.47      2025

Model saved.
```

Model A5 — XGBoost + TF-IDF

```
# Import necessary modules
from xgboost import XGBClassifier
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report
import joblib

# Define TF-IDF transformer
text_transformer = TfidfVectorizer(stop_words='english', ngram_range=(1, 2), max_df=0.9, min_df=2)
preprocessor = ColumnTransformer([("text", text_transformer, "text")])

# Initialize classifier
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)

# Create pipeline
pipeline_xgb = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", xgb_model)
])

# Cross-validation
cv_scores = cross_val_score(pipeline_xgb, X_train.to_frame(), y_train, cv=5, scoring="f1_macro")
print(f"Mean F1-score (CV): {cv_scores.mean():.4f}")
```

```
# Grid search for best params
param_grid_xgb = {
    "classifier__n_estimators": [100, 200],
    "classifier__max_depth": [3, 6, 10],
    "classifier__learning_rate": [0.01, 0.1, 0.2]
}
grid_search_xgb = GridSearchCV(pipeline_xgb, param_grid_xgb, cv=3, scoring="f1_macro", n_jobs=-1)
grid_search_xgb.fit(X_train.to_frame(), y_train)

# Evaluation
best_xgb_model = grid_search_xgb.best_estimator_
print(f"Best Parameters: {grid_search_xgb.best_params_}")
y_pred_xgb = best_xgb_model.predict(X_test.to_frame())
print(classification_report(y_test, y_pred_xgb))

# Save model
joblib.dump(best_xgb_model, "dev/best_xgboost.pkl")
print("Model saved.")
```

Model Development RQ2(DL) Script

Setup:

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Dataset to use
df = master_df.copy()

# Create gender label from abstract
df_male = df[df['abstract'].str.contains(r'\b(male|man|men|boy|boys)\b', case=False, na=False)].copy()
df_male['gender_label'] = 1

df_female = df[df['abstract'].str.contains(r'\b(female|woman|women|girl|girls)\b', case=False, na=False)].copy()
df_female['gender_label'] = 0

df_gender = pd.concat([df_male, df_female]).drop_duplicates()

# Combine relevant fields into a single text column
df_gender['text'] = df_gender['title'].fillna('') + ' ' + df_gender['journal'].fillna('') + ' ' + df_gender['abstract'].fillna('')

# Prepare features and target
X = df_gender['text']
y = df_gender['gender_label']

# Step 1: First split off 20% test
X_temp, X_test, y_temp, y_test = train_test_split(X, y, stratify=y, test_size=0.2, random_state=42)
# Step 2: Split the remaining 80% into train (80%) and val (20%) => 64/16
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, stratify=y_temp, test_size=0.2, random_state=42)

# Final sizes:
# X_train: 64% of full data
# X_val:   16% of full data
# X_test:  20% of full data

# Optional: Print sizes
print("Train size:", len(X_train))
print("Validation size:", len(X_val))
print("Test size:", len(X_test))
```

```
<ipython-input-8-106d1d560f8b>:8: UserWarning: This pattern is interpreted as a regular expression, and has match groups. To actually get th
  df_male = df[df['abstract'].str.contains(r'\b(male|man|men|boy|boys)\b', case=False, na=False)].copy()
<ipython-input-8-106d1d560f8b>:11: UserWarning: This pattern is interpreted as a regular expression, and has match groups. To actually get t
  df_female = df[df['abstract'].str.contains(r'\b(female|woman|women|girl|girls)\b', case=False, na=False)].copy()
Train size: 6479
Validation size: 1620
Test size: 2025
```

Model B1 — LSTM (Keras)

```
# Import necessary modules
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import classification_report
import joblib

# Tokenize text
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(X_train)

X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)

# Pad sequences
maxlen = 200
X_train_pad = pad_sequences(X_train_seq, maxlen=maxlen, padding="post", truncating="post")
X_test_pad = pad_sequences(X_test_seq, maxlen=maxlen, padding="post", truncating="post")

# Convert labels to arrays
y_train_dl = np.array(y_train)
y_test_dl = np.array(y_test)
```

```
# Build LSTM model
model_lstm = Sequential()
model_lstm.add(Embedding(input_dim=10000, output_dim=64, input_length=maxlen))
model_lstm.add(LSTM(64, return_sequences=False))
model_lstm.add(Dropout(0.5))
model_lstm.add(Dense(1, activation="sigmoid"))

model_lstm.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])

# Train model
early_stop = EarlyStopping(patience=3, restore_best_weights=True)
model_lstm.fit(X_train_pad, y_train_dl, epochs=10, batch_size=32, validation_split=0.2, callbacks=[early_stop])

# Evaluate
y_pred_lstm = (model_lstm.predict(X_test_pad) > 0.5).astype("int32")
print(classification_report(y_test_dl, y_pred_lstm))

# Save model
model_lstm.save("dev/best_lstm_model.h5")
print("Model saved.")
```

Model B2 — GRU (Keras)

```
# Import necessary modules
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, GRU, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import classification_report

# Tokenize and pad sequences (reuse tokenizer from LSTM or reinitialize)
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(X_train)

X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)

maxlen = 200
X_train_pad = pad_sequences(X_train_seq, maxlen=maxlen, padding="post", truncating="post")
X_test_pad = pad_sequences(X_test_seq, maxlen=maxlen, padding="post", truncating="post")

y_train_dl = np.array(y_train)
y_test_dl = np.array(y_test)

# Build GRU model
model_gru = Sequential()
model_gru.add(Embedding(input_dim=10000, output_dim=64, input_length=maxlen))
model_gru.add(GRU(64, return_sequences=False))
model_gru.add(Dropout(0.5))
model_gru.add(Dense(1, activation="sigmoid"))
```

```
model_gru.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])

# Train model
early_stop = EarlyStopping(patience=3, restore_best_weights=True)
model_gru.fit(X_train_pad, y_train_dl, epochs=10, batch_size=32, validation_split=0.2, callbacks=[early_stop])

# Evaluate
y_pred_gru = (model_gru.predict(X_test_pad) > 0.5).astype("int32")
print(classification_report(y_test_dl, y_pred_gru))

# Save model
model_gru.save("dev/best_gru_model.h5")
print("Model saved.")
```

Model B3 — BERT (Hugging Face Transformers)

Model Development RQ3 Script

Setup:

```
import pandas as pd
import re

df = master_df.copy()

# Label abstracts discussing adolescents, mental health, and suicide
def label_rq3(text):
    if pd.isna(text):
        return 0
    text = text.lower()
    has_adolescent = re.search(r'\b(adolescent|youth|teen|child)\b', text) is not None
    has_mental_health = re.search(r'\b(mental health|psychiatric|psychological|depression|anxiety)\b', text) is not None
    has_suicide = re.search(r'\b(suicide|self[- ]harm|selfinjury|self injury)\b', text) is not None
    return int(has_adolescent and has_mental_health and has_suicide)

# Apply label function
df["rq3_label"] = df["abstract"].apply(label_rq3)

# Combine text columns
df["text"] = df["title"].fillna('') + ' ' + df["journal"].fillna('') + ' ' + df["abstract"].fillna('')

# Subset for relevant papers
df_rq3 = df[df["rq3_label"] == 1].copy()
```

Model 1-- KMeans Clustering

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
import joblib

# Vectorize the text
vectorizer = TfidfVectorizer(stop_words="english", ngram_range=(1, 2), max_df=0.8, min_df=3)
X_tfidf = vectorizer.fit_transform(df_rq3["text"])

# Fit KMeans with 5 clusters
kmeans = KMeans(n_clusters=5, random_state=42)
kmeans.fit(X_tfidf)

# Print top terms in each cluster
terms = vectorizer.get_feature_names_out()
for i in range(5):
    center = kmeans.cluster_centers_[i]
    top_indices = center.argsort()[::-1][:10]
    print(f"\nCluster {i+1}:")
    print([terms[j] for j in top_indices])

# Save model and vectorizer
joblib.dump(kmeans, "dev/kmeans_rq3.pkl")
joblib.dump(vectorizer, "dev/kmeans_vectorizer_rq3.pkl")
```

```
Cluster 1:
['women', 'filicide', 'child', 'cases', 'case', 'depression', 'postpartum', 'violence', 'death', 'mothers']

Cluster 2:
['suicidal', 'ideation', 'suicidal ideation', 'adolescents', 'students', 'school', 'risk', 'depression', 'attempts', 'behavior']

Cluster 3:
['youth', 'health', 'adolescents', 'depression', 'risk', 'use', 'adolescent', 'mental', 'mental health', 'prevention']

Cluster 4:
['nssi', 'psychiatric', 'attempts', 'adolescents', 'patients', 'suicide attempts', 'disorder', 'disorders', 'risk', 'adolescent']

Cluster 5:
['health', 'mental', 'mental health', 'children', 'young', 'people', 'services', 'young people', 'problems', 'care']
['dev/kmeans_vectorizer_rq3.pkl']
```

Model 2 — LDA Topic Modeling

```python
import gensim
from gensim import corpora
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import nltk

# Download NLTK resources
nltk.download("punkt")
nltk.download("stopwords")
nltk.download('punkt_tab')

# Tokenize and remove stopwords
stop_words = set(stopwords.words("english"))
texts = [
    [word for word in word_tokenize(doc.lower()) if word.isalpha() and word not in stop_words]
    for doc in df_rq3["text"]
]

# Create dictionary and corpus
dictionary = corpora.Dictionary(texts)
corpus = [dictionary.doc2bow(text) for text in texts]

# Train LDA model
lda_model = gensim.models.LdaModel(corpus, num_topics=5, id2word=dictionary, passes=10, random_state=42)
```

```python
# Print top topics
topics = lda_model.print_topics(num_words=10)
for topic in topics:
    print(topic)

# Save model and dictionary
lda_model.save("dev/lda_rq3_model.gensim")
dictionary.save("dev/lda_dictionary_rq3.dict")
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
(0, '0.018*"suicide" + 0.012*"risk" + 0.012*"health" + 0.007*"school" + 0.007*"students" + 0.007*"use" + 0.007*"youth" + 0.007*"mental" + 0.007*"among" + 0.
(1, '0.017*"suicide" + 0.016*"health" + 0.010*"adolescents" + 0.010*"mental" + 0.009*"depression" + 0.008*"adolescent" + 0.007*"study" + 0.007*"youth" + 0.0
(2, '0.021*"suicide" + 0.016*"adolescents" + 0.015*"suicidal" + 0.011*"health" + 0.010*"risk" + 0.009*"among" + 0.008*"youth" + 0.008*"study" + 0.008*"behav
(3, '0.013*"depression" + 0.011*"suicide" + 0.010*"children" + 0.010*"child" + 0.009*"treatment" + 0.009*"health" + 0.007*"mental" + 0.007*"adolescent" + 0.
(4, '0.037*"suicide" + 0.015*"suicidal" + 0.013*"risk" + 0.012*"attempts" + 0.010*"adolescent" + 0.009*"ideation" + 0.008*"study" + 0.008*"psychiatric" + 0.
```

Model 3 — BERTopic

```python
from bertopic import BERTopic

# Extract documents
texts = df_rq3["text"].tolist()

# Fit BERTopic
topic_model = BERTopic(language="english", verbose=True)
topics, probs = topic_model.fit_transform(texts)

# View topic summary
topic_model.get_topic_info().head()

# Save model
topic_model.save("dev/bertopic_rq3_model")
```

Model 4 -- Logistic Regression + TF-IDF

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.metrics import classification_report
import joblib

# Define TF-IDF transformer
tfidf = TfidfVectorizer(stop_words="english", ngram_range=(1, 2), max_df=0.9, min_df=2)
preprocessor = ColumnTransformer([("text", tfidf, "text")])

# Define logistic regression pipeline
lr = LogisticRegression(max_iter=1000, random_state=42)
pipeline = Pipeline([("preprocessor", preprocessor), ("classifier", lr)])

# Perform cross-validation
cv_scores = cross_val_score(pipeline, df[["text"]], df["rq3_label"], cv=5, scoring="f1_macro")
print(f"F1 Macro CV: {cv_scores.mean():.4f}")

# Hyperparameter tuning
param_grid = {"classifier__C": [0.1, 1, 10]}
grid = GridSearchCV(pipeline, param_grid, cv=3, scoring="f1_macro", n_jobs=-1)
grid.fit(df[["text"]], df["rq3_label"])
```

```python
# Evaluate model
best_model = grid.best_estimator_
y_pred = best_model.predict(df[["text"]])
print(classification_report(df["rq3_label"], y_pred))

# Save model
joblib.dump(best_model, "dev/logreg_rq3.pkl")
```

```
F1 Macro CV: 0.5903
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     27785
           1       1.00      0.95      0.97      1235

    accuracy                           1.00     29020
   macro avg       1.00      0.98      0.99     29020
weighted avg       1.00      1.00      1.00     29020

['dev/logreg_rq3.pkl']
```

Model 5 — Random Forest + TF-IDF

```python
# Import all required modules
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.metrics import classification_report
import joblib

# Define TF-IDF vectorizer and column transformer
tfidf = TfidfVectorizer(stop_words="english", ngram_range=(1, 2), max_df=0.9, min_df=2)
preprocessor = ColumnTransformer([("text", tfidf, "text")])

# Define random forest classifier
rf = RandomForestClassifier(random_state=42)

# Create pipeline
pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", rf)
])

# Perform cross-validation
cv_scores = cross_val_score(pipeline, df[["text"]], df["rq3_label"], cv=5, scoring="f1_macro")
print(f"F1 Macro CV: {cv_scores.mean():.4f}")
```

```python
# Define hyperparameter grid
param_grid = {
    "classifier__n_estimators": [100, 200],
    "classifier__max_depth": [10, 20, None],
    "classifier__min_samples_split": [2, 5]
}

# Perform grid search
grid = GridSearchCV(pipeline, param_grid, cv=3, scoring="f1_macro", n_jobs=-1)
grid.fit(df[["text"]], df["rq3_label"])

# Evaluate model
best_model = grid.best_estimator_
y_pred = best_model.predict(df[["text"]])
print(classification_report(df["rq3_label"], y_pred))

# Save model to disk
joblib.dump(best_model, "dev/rf_rq3.pkl")
```

```
F1 Macro CV: 0.5123
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     27785
           1       1.00      1.00      1.00      1235

    accuracy                           1.00     29020
   macro avg       1.00      1.00      1.00     29020
weighted avg       1.00      1.00      1.00     29020

['dev/rf_rq3.pkl']
```

Model 6 — SVM + TF-IDF

```python
from sklearn.svm import SVC

# Define SVM pipeline
svm = SVC(kernel="linear", probability=True, random_state=42)
pipeline = Pipeline([("preprocessor", preprocessor), ("classifier", svm)])

# Cross-validation
cv_scores = cross_val_score(pipeline, df[["text"]], df["rq3_label"], cv=5, scoring="f1_macro")
print(f"F1 Macro CV: {cv_scores.mean():.4f}")

# Hyperparameter tuning
param_grid = {"classifier__C": [0.1, 1, 10]}
grid = GridSearchCV(pipeline, param_grid, cv=3, scoring="f1_macro", n_jobs=-1)
grid.fit(df[["text"]], df["rq3_label"])

# Evaluation
best_model = grid.best_estimator_
y_pred = best_model.predict(df[["text"]])
print(classification_report(df["rq3_label"], y_pred))

# Save model
joblib.dump(best_model, "dev/svm_rq3.pkl")
```

```
F1 Macro CV: 0.6225
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     27785
           1       1.00      1.00      1.00      1235

    accuracy                           1.00     29020
   macro avg       1.00      1.00      1.00     29020
weighted avg       1.00      1.00      1.00     29020

['dev/svm_rq3.pkl']
```

Model 7 — LSTM (Keras)

```python
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import classification_report

# Tokenize and pad sequences
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
tokenizer.fit_on_texts(df["text"])
X_seq = tokenizer.texts_to_sequences(df["text"])
X_pad = pad_sequences(X_seq, maxlen=200, padding="post")
y = np.array(df["rq3_label"])

# Build LSTM model
model = Sequential()
model.add(Embedding(10000, 64, input_length=200))
model.add(LSTM(64))
model.add(Dropout(0.5))
model.add(Dense(1, activation="sigmoid"))

# Compile and train
model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
model.fit(X_pad, y, epochs=5, batch_size=32, validation_split=0.2, callbacks=[EarlyStopping(patience=2)])
# Evaluate
y_pred = (model.predict(X_pad) > 0.5).astype("int32")
print(classification_report(y, y_pred))

# Save model
model.save("dev/lstm_rq3.h5")
```

Model 8 — BERT (Transformers)

```
from transformers import BertTokenizer, TFBertForSequenceClassification
import tensorflow as tf
import numpy as np
from sklearn.metrics import classification_report

# Tokenize using BERT
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = TFBertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)

encodings = tokenizer(df["text"].tolist(), truncation=True, padding=True, max_length=128, return_tensors="tf")
dataset = tf.data.Dataset.from_tensor_slices((dict(encodings), df["rq3_label"].values)).batch(16)

# Compile and train
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=2e-5),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=["accuracy"])

model.fit(dataset, epochs=3)

# Predict and evaluate
y_probs = model.predict(dataset)["logits"]
y_preds = np.argmax(y_probs, axis=1)
print(classification_report(df["rq3_label"].values, y_preds))
# Save model and tokenizer
model.save_pretrained("dev/bert_rq3_model")
tokenizer.save_pretrained("dev/bert_rq3_model")
```

Model Development RQ4 Script

```python
# suicide_meta_analysis.py

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.stats.meta_analysis import combine_effects
import joblib

# Load dataset with time series predictions already included
df = master_df.copy()

# ---- STEP 1: Simulate Effect Size Column ----
np.random.seed(42)
df["effect_size"] = np.where(
    df["suicide_prevention_measures"].str.strip() != "",
    np.random.normal(loc=0.3, scale=0.1, size=len(df)),  # Simulate small-medium effect sizes
    np.nan
)

# Filter for rows with valid effect sizes
df_meta = df.dropna(subset=["effect_size"]).copy()
df_meta["variance"] = 0.05  # assumed fixed variance
```

```python
# ---- STEP 2: Meta-Analysis using Fixed and Random Effects ----
effects = df_meta["effect_size"].values
variances = df_meta["variance"].values

# Fixed Effects Model
fixed_result = combine_effects(effects, variances)
fixed_summary = fixed_result.summary_frame()
print("Fixed Effects Summary Frame:")
print(fixed_summary)

# Safe access using .iloc[]
fixed_effect_size = fixed_summary.iloc[0, 0]
fixed_ci_low = fixed_summary.iloc[0, 1]
fixed_ci_upp = fixed_summary.iloc[0, 2]
print(f"Fixed Effects: Effect size = {fixed_effect_size:.4f}, CI = [{fixed_ci_low:.4f}, {fixed_ci_upp:.4f}]")

# Random Effects Model
random_result = combine_effects(effects, variances, method_re="iterated")
random_summary = random_result.summary_frame()
print("Random Effects Summary Frame:")
print(random_summary)

random_effect_size = random_summary.iloc[0, 0]
random_ci_low = random_summary.iloc[0, 1]
random_ci_upp = random_summary.iloc[0, 2]
print(f"Random Effects: Effect size = {random_effect_size:.4f}, CI = [{random_ci_low:.4f}, {random_ci_upp:.4f}]")
```

```python
# ---- STEP 3: Embed effect sizes back into dataset ----
df["meta_effect_size"] = df["effect_size"]

# Save updated dataset
df.to_csv("df_premodeling_v4.csv", index=False)
print("Updated dataset saved as sample_with_meta.csv with meta_effect_size column.")

# Save model outputs for reproducibility
joblib.dump({"fixed": fixed_result, "random": random_result}, "meta_model_results.pkl")
print("Meta-analysis model results saved as meta_model_results.pkl")
```

Model Development RQ5 Script

```
# suicide_survival_analysis.py

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from lifelines import KaplanMeierFitter, CoxPHFitter
import joblib

# Load dataset
df = pd.read_csv("df_premodeling_v4.csv")

# ---- STEP 1: Simulate survival data ----
np.random.seed(42)

df["survival_time"] = np.random.exponential(scale=3, size=len(df))  # avg ~3 years
df["event_occurred"] = np.random.binomial(1, 0.7, size=len(df))  # 70% had event
df["has_prevention"] = df["suicide_prevention_measures"].str.strip().ne("").astype(int)

# ---- STEP 2: Kaplan-Meier Estimator ----
kmf = KaplanMeierFitter()
kmf.fit(durations=df["survival_time"], event_observed=df["event_occurred"])
kmf.plot_survival_function()
plt.title("Kaplan-Meier Survival Estimate")
plt.xlabel("Time (years)")
plt.ylabel("Survival Probability")
plt.savefig("km_survival_plot.png")
print("Kaplan-Meier plot saved as km_survival_plot.png")
```

```
# ---- STEP 3: Cox Proportional Hazards Model (safe handling) ----
# Drop NaNs and prepare covariates
df_cox = df[["survival_time", "event_occurred", "has_prevention"]].dropna()

# Print distribution of covariate
print("has_prevention distribution:")
print(df_cox["has_prevention"].value_counts())

# Fit only if there's variance
if df_cox["has_prevention"].nunique() > 1:
    cph = CoxPHFitter()
    cph.fit(df_cox, duration_col="survival_time", event_col="event_occurred")
    cph.print_summary()
    cph.plot()
    plt.title("Cox Proportional Hazards")
    plt.tight_layout()
    plt.savefig("cox_hazard_plot.png")
    print("Cox model plot saved as cox_hazard_plot.png")
else:
    print("Skipping Cox model: 'has_prevention' has insufficient variance to converge.")

# ---- STEP 4: Save survival dataset ----
df.to_csv("df_premodeling_v5.csv", index=False)
print("Final dataset saved as df_premodeling_v5.csv with survival columns.")

# ---- STEP 5: Save trained models ----
models = {"kmf": kmf}
if df_cox["has_prevention"].nunique() > 1:
    models["cox"] = cph
```

Model Development RQ6 Script

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.seasonal import seasonal_decompose
from prophet import Prophet
import joblib

# Load base dataset
df = master_df.copy()

# 1. Prepare yearly aggregated data
df_yearly = df.groupby("year").size().reset_index(name="article_count")
df_yearly["year"] = pd.to_datetime(df_yearly["year"], format="%Y")
df_yearly.set_index("year", inplace=True)

# 2. ARIMA Model
model_arima = ARIMA(df_yearly, order=(1, 1, 1))
fit_arima = model_arima.fit()
forecast_arima = fit_arima.predict(start=0, end=len(df_yearly)-1)
df_yearly["ts_trend_arima"] = forecast_arima.values
joblib.dump(fit_arima, "arima_model.pkl")
print("ARIMA model saved as arima_model.pkl")
```

```
# 3. Seasonal Decomposition
decomp = seasonal_decompose(df_yearly["article_count"], model='additive', period=1)
df_yearly["ts_decomposition_trend"] = decomp.trend
decomp.plot()
plt.tight_layout()
plt.savefig("decomposition_plot.png")
print("Decomposition plot saved as decomposition_plot.png")

# 4. Prophet Model
df_prophet = df_yearly.reset_index().rename(columns={"year": "ds", "article_count": "y"})
model_prophet = Prophet(yearly_seasonality=True)
model_prophet.fit(df_prophet)
forecast_prophet = model_prophet.predict(df_prophet[["ds"]])
df_yearly["ts_trend_prophet"] = forecast_prophet["yhat"].values
joblib.dump(model_prophet, "prophet_model.pkl")
print("Prophet model saved as prophet_model.pkl")

fig1 = model_prophet.plot(forecast_prophet)
fig1.savefig("prophet_forecast_plot.png")
print("Prophet forecast plot saved as prophet_forecast_plot.png")

# 5. Convert year back to int for merging
df_yearly = df_yearly.reset_index()
df_yearly["year"] = df_yearly["year"].dt.year

# 6. Merge predictions into original dataset
df = df.merge(df_yearly[["year", "ts_trend_arima", "ts_decomposition_trend", "ts_trend_prophet"]], on="year", how="left")
```

```
# 7. Save updated dataset
df.to_csv("df_premodeling_v3.csv", index=False)
print("Final dataset saved as df_premodeling_v3.csv with time series outputs embedded.")
```