Gun Incidence Analysis

PG7

Bi Yuying
Cai Zhenxin
Khoo Tina
Lee Hui En
Ruvvenesh Ramesh
Tan Teck Hwe Damaen

Presentation Contents

01 Overview Of Dataset

- Guns Incidence Dataset Intro
- Dataset characteristics

02 Data Preprocessing

- Data Cleaning
- Data Visualisations
- Feature Engineering

03 Model 1 - Decision Tree

- Implementing Decision Tree
- Hyperparameter Tuning
- Optimization Techniques
- Sampling Techniques
- Choosing The Best Model
- Plotting The Decision Tree

04 Model 2 - Logistic Regression

- Implementing Logistic Regression
- Hyperparameter Tuning
- Optimization Techniques
- Sampling Techniques
- Choosing The Best Model

05 Question Analysis

- Answering the four questions
- Concluding and evaluating outcomes

01

Overview Of Dataset

Introduction To Guns Incidence Dataset

First 5 rows of dataset:

	S.No.	Year	Month	Date	Reason	Education	Sex	Age	Race	Hispanic	Place of incident	Police involvement
0	1	2017	1	6/1/17	Suicide	Bachelors	Male	35.0	Asian/Pacific Islander	100	Home	0
1	2	2017	1	19/1/17	Suicide	Some college	Female	22.0	White	100	Street	0
2	3	2017	1	1/1/17	Suicide	Bachelors	Male	61.0	White	100	Other specified	0
3	4	2017	2	6/2/17	Suicide	Bachelors	Male	65.0	White	100	Home	0
4	5	2017	2	9/2/17	Suicide	High School	Male	32.0	White	100	Other specified	0

Dataset contains:

- 100798 rows
- 12 columns

Our objective is to identify the causes of death from guns based on multiple factors, including education level, race, sex, location of incident, and other relevant features.

Dataset Characteristics

Important Features To Consider

Education

- Bachelors
- High school
- Less than high school
- Some College

Sex

- Female
- Male

Age

- Range of values from 20 to 60

Race

- Asian/Pacific Islander
- Black
- Hispanic
- Native American/Alaskan
- White

Place Of Incident

- Home
- Street
- Other Specified
- Other Unspecified

and many more

Year/Month/Date

- Year: 2017 to 2019
- Month: 1 (Jan) to 12 (Dec)
- Date: Format of DD/MM/YYYY

Problem Statement & Motivation

Problem Statement

We want to accurately predict the occurrence of fatal gun violence incidents by analysing patterns and trends in the dataset.

Motivations

Gun injuries and deaths are a significant public health problem in the United States. Analysing these trends in gun-related incidents can help in avoiding such scenarios.

Furthermore, being able to characterise those who are likely to be involved in such incidents allows for early interventions, reducing the mortality rate for such incidents.

Motivating Questions

1. Which age, race or gender groups are most affected by suicide incidents?

2. Is there a significant racial disparity in gun violence incidents?

3. Do the locations where these incidents happen (at home or on the street) make a difference in the type of fatal event?

4. Are certain gun crimes more likely to occur in certain months or years?

02

Data Preprocessing

Data Cleaning

Step 1: Removing NA/missing values

Education	Age	Place of incident
1422 NA's	**18** NA's	**1384** NA's

Number of rows after removal

100798 rows -> **98015** rows

(2783 rows removed)

Step 2: Removing duplicate rows

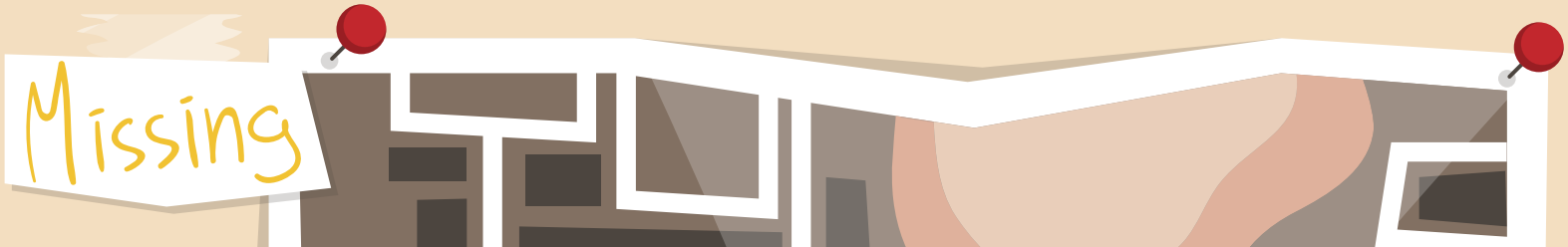
```
# Check the total number of duplicates in the dataset
number_duplicates = df.duplicated().sum()
print(f'No. of duplicate rows: {number_duplicates}')
```

```
No. of duplicate rows: 3096
```

We observed **3096 duplicated rows**

98015 rows -> **94919** rows

(3096 rows removed)



Data Cleaning

Step 3: Convert to correct data type for date column

Date column of type **object**

(Incorrect)

Date column of type **datetime**

(Correct)

```
df['Date'] = pd.to_datetime(df['Date'], format='%d/%m/%y')
```

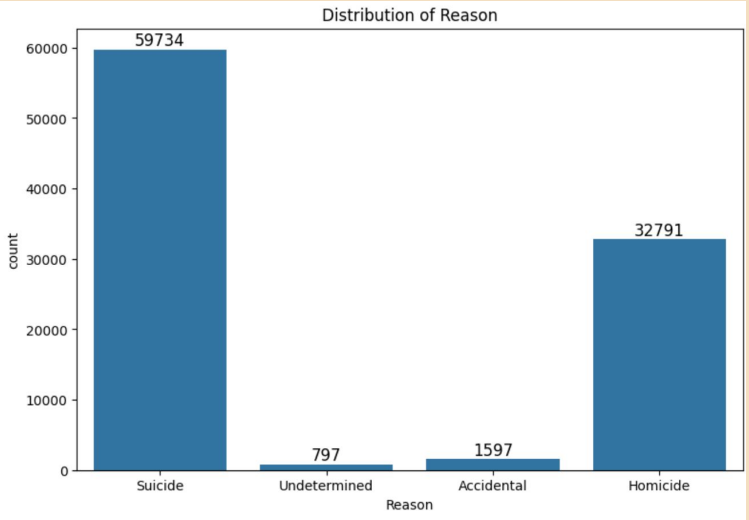
Step 4: Removing unnecessary and irrelevant columns

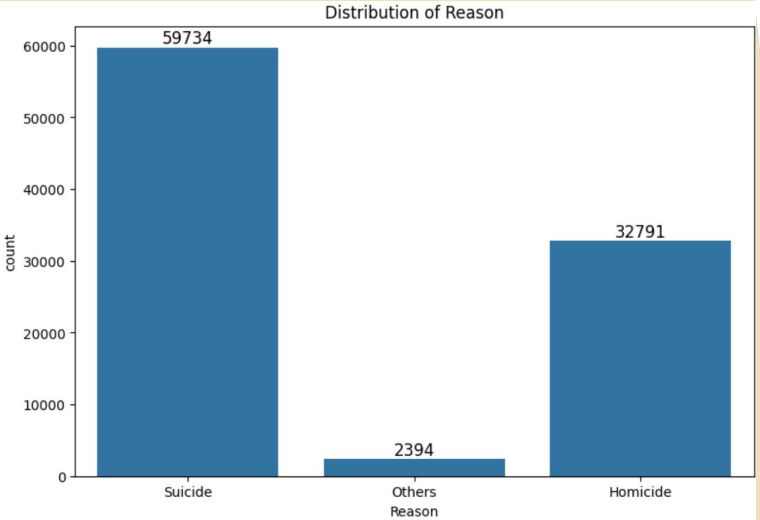
- Dropped "**Year**" and "**Month**" columns
  -> Information on Year and Month can be found in the "**Date**" column

- Dropped "**Hispanic**" column
  -> **Irrelevant feature**, does not make sense to have random numbers

Data Visualisations

Visualisation 1: Distribution of target variable - **Reason**





- **59734** Suicide cases
- **797** Undetermined cases
- **1597** Accidental cases
- **32791** Homicide cases

Few Underdetermined and Accidental cases

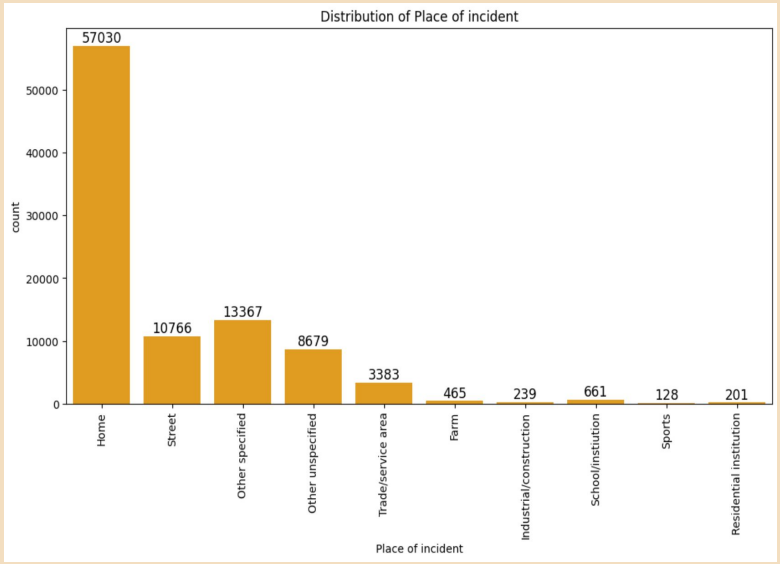
Merge into category called Others

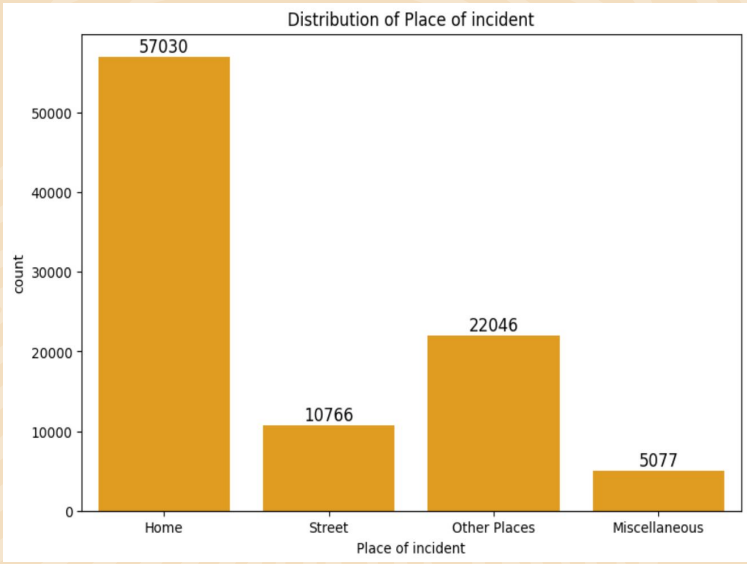
- **59734** Suicide cases
- **2394** Others cases
- **32791** Homicide cases

11:15 pm

Data Visualisations

Visualisation 2: Distribution of feature - **Place Of Incident**



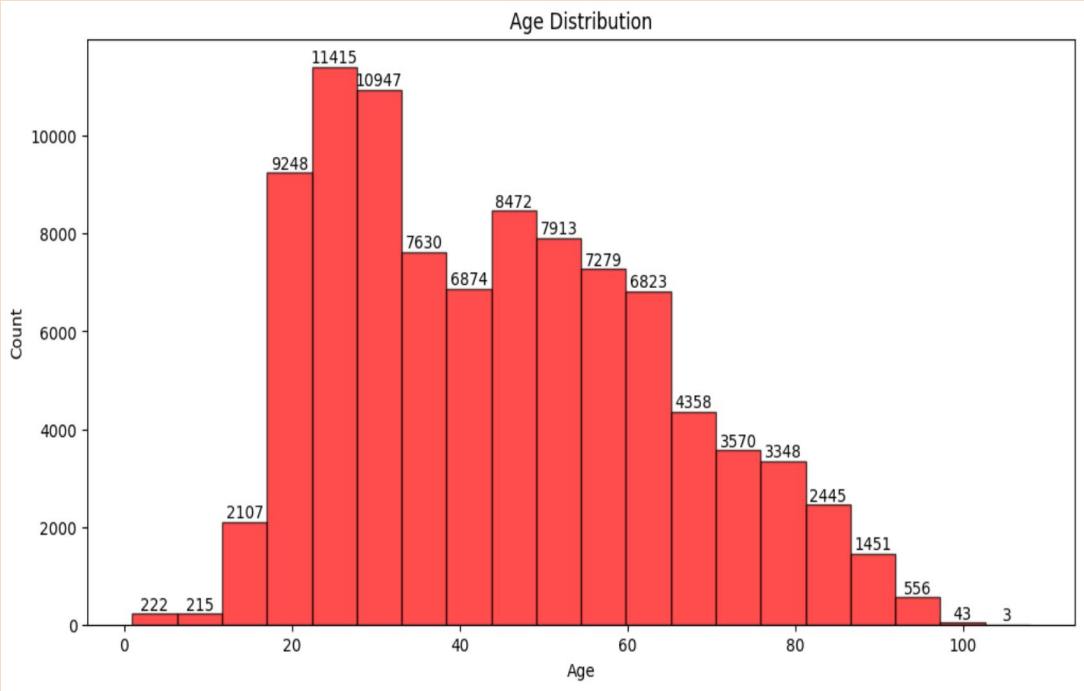


- Too many different categories
- **Other specified** and **Other unspecified** can be combined into "**Other Places**"
- **Trade/Service area**, **Farm**, **Industrial**, **School**, **Sports** and **Residential Institution** can be combined into "**Miscellaneous**"

- **57030** Home
- **10766** Street
- **22046** Other Places
- **5077** Miscellaneous

Data Visualisations

Visualisation 3: Distribution of feature - **Age**

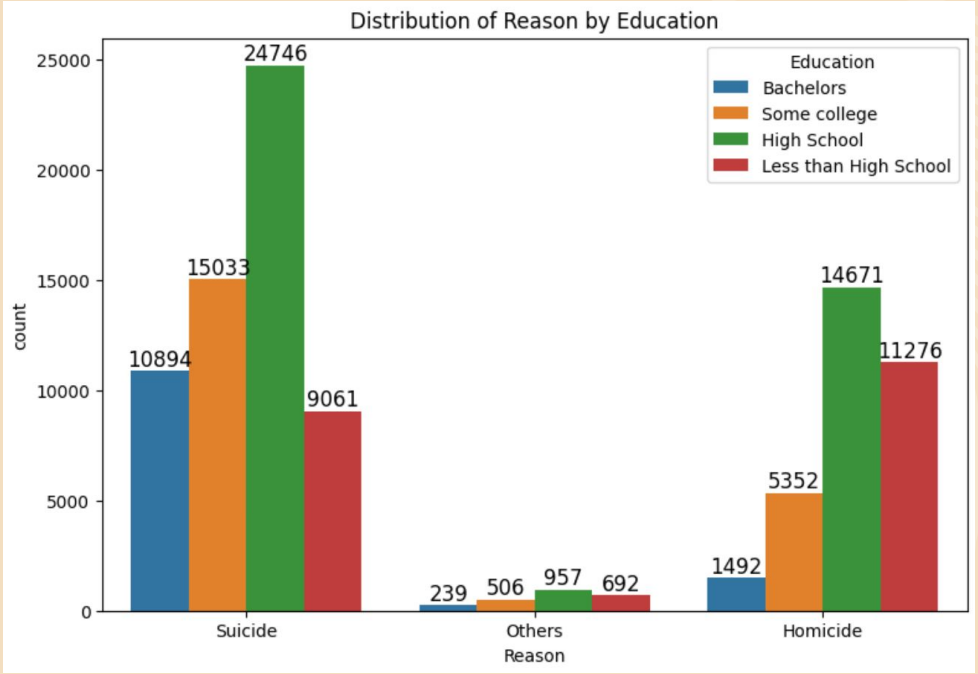


Key Observations

- Age ranges from **below 5 to over 105 years old**
- Most are in the **age range of 20 to 60**
- **No significant skew** in the age range (no obvious left or right skew)

Data Visualisations

Visualisation 4: Distribution of **Reason** by **Education**

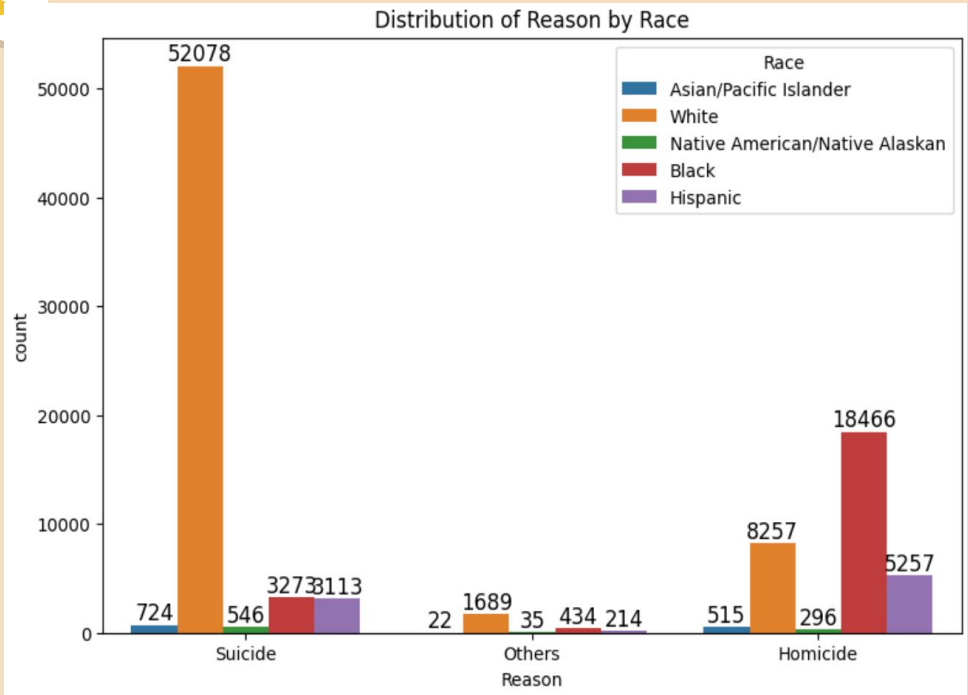


Key Observations

- **High School** contributes **the most to all three reason types**
- **Some College** contributes **second most to reason type of Suicide**
- **Less than High School** contributes **second most to reason types Others and Homicide**

Data Visualisations

Visualisation 5: Distribution of **Reason** by **Race**

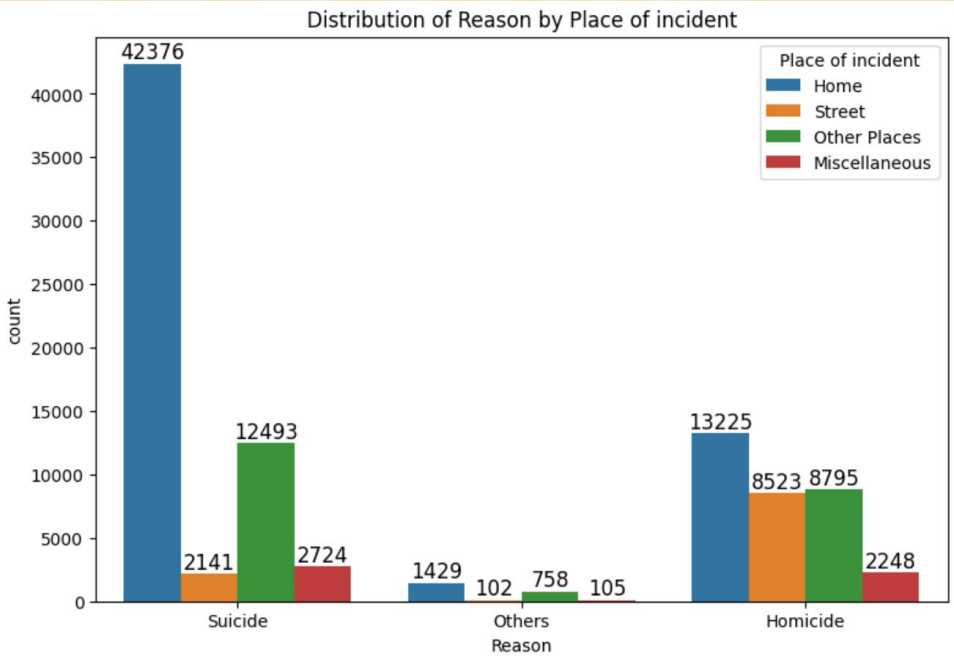


Key Observations

- **White** race contributes the most to the reason types of **Suicide** and **Others**
- **Black** race contributes the most to the reason type of **Homicide**
- **Asian/Pacific Islander** and **Native American/Alaskan** contribute **insignificantly** to all three reason types

Data Visualisations

Visualisation 6: Distribution of **Reason** by **Place of incident**

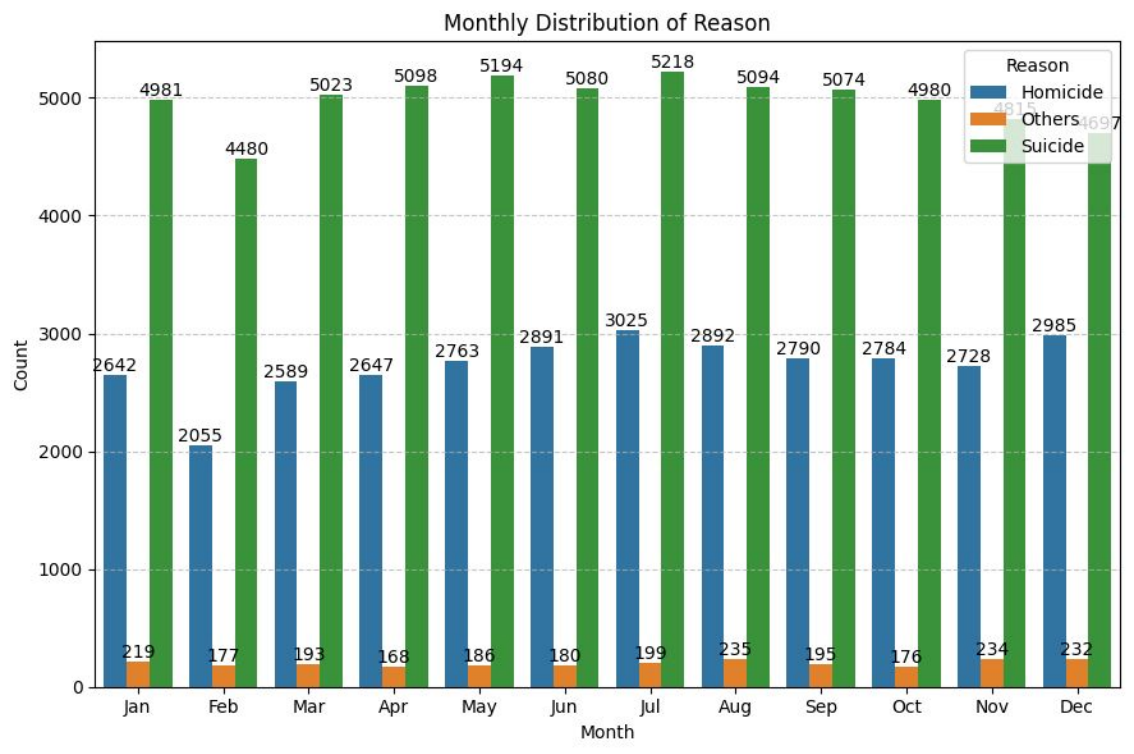


Key Observations

- **Home** contributes t**he most for all three reason types** (Suicide, Others, Homicide)
- **Other places** contributes **second most for all three reason types**
- Near zero cases for **Street** and **Miscellaneous** for reason type **Others**

Data Visualisations

Visualisation 7: Distribution of **Reason** by **Months**

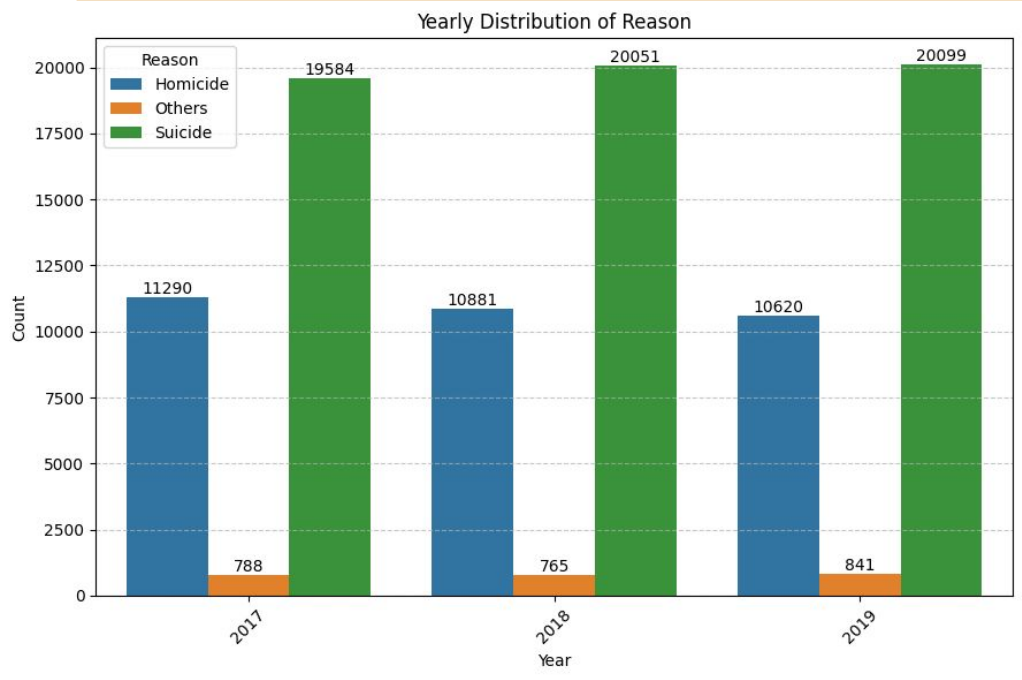


Key Observations

- Slight inverse U-shaped trend for the reason type **Suicide**
- Seasonal trend for the reason type **Homicide**
- No observable trend for the reason type **Others**

Data Visualisations

Visualisation 8: Distribution of **Reason** by **Year**



Key Observations

- Increasing trend for the reason type **Suicide**
- Decreasing trend for the reason type **Homicide**
- No observable trend for the reason type **Others**

Feature Engineering

Step 1: Encoding categorical features into numeric

Categorical Features Not Encoded:

- **Reason**
- **Education**
- **Sex**
- **Race**
- **Place of incident**

Categorical Features Already Encoded:

- Police Involvement

Other Features That Do Not Need Encoding:

- Date

Apply The Necessary Encodings

```
# Encode 'Reason' column
df['Reason Encoded'] = label_encoder_reason.fit_transform(df['Reason'])

# Encode 'Education' column
df['Education Encoded'] = label_encoder_education.fit_transform(df['Education'])

# Encode 'Sex' column
df['Sex Encoded'] = label_encoder_sex.fit_transform(df['Sex'])

# Encode 'Race' column
df['Race Encoded'] = label_encoder_race.fit_transform(df['Race'])

# Encode 'Place of incident' column
df['Place Encoded'] = label_encoder_place.fit_transform(df['Place of incident'])
```

Table Of Categories and Encoded Numbers

Encoded Number	Reason	Education	Sex	Race	Place
0	Homicide	Bachelors	Female	Asian/Pacific Islander	Home
1	Others	High School	Male	Black	Miscellaneous
2	Suicide	Less than High School		Hispanic	Other Places
3		Some College		Native American/Native Alaskan	Street
4				White	

Feature Engineering

Step 2: Extracting year, month and day of week from **Date** column

Date
2017-01-06
2017-01-19
2017-01-01
2017-02-06
2017-02-09

```
# Modifying the Year and Month Of Date column
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
```

```
# Modifying the Day of Date column
df['Day Of Week'] = pd.to_datetime(df['Date'], format='%Y-%m-%d').dt.dayofweek + 1
df_modified = df.drop(columns=['Date'])
```

Year	Month	Day Of Week
2017	1	5
2017	1	4
2017	1	7
2017	2	1
2017	2	4

Date column in datetime datatype

Extract year and month

Extract and modify day of week

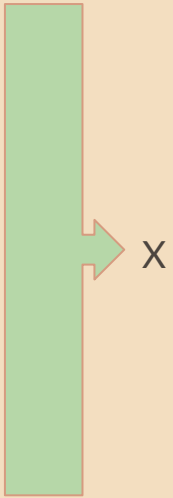
Three columns - **Year, Month and Day Of Week**, where **Day Of Week = 1** represents **Monday** (not Sunday)

Feature Engineering

Step 3: Extract relevant features and target

Features we have now:

1. Year
2. Month
3. Day Of Week
4. Education
5. Sex
6. Age
7. Race
8. Place of incident
9. Police Involvement



Target we have now:

1. Reason

Extract Relevant Features into X and y

```
# Subsetting the df_modified dataset into features (X) and target (y)
X = df_modified[['Year', 'Month', 'Day Of Week', 'Education Encoded',
                 'Sex Encoded', 'Age', 'Race Encoded', 'Place Encoded',
                 'Police involvement']]
y = df_modified['Reason Encoded']
```

Performing Train-Test-Split

```
# Performing the train-test-split of the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    random_state = 0)
```

03

Model 1 - Decision Tree

1st Model: Unoptimised Decision Tree Model

```
# Call the DecisionTreeClassifier from sklearn.trees
decision_tree_classifier = DecisionTreeClassifier()

# Train the Decision Tree model using the training dataset
decision_tree_classifier.fit(X_train, y_train)

# Predict the target (y_pred) of the test dataset from its features (X_test)
y_pred = decision_tree_classifier.predict(X_test)
```

Implementation

- Call **DecisionTreeClassifier** from **sklearn.trees**
- Fit the training dataset into the model
- Predict the target, y_pred, using the test dataset

Accuracy And Runtime Of Unoptimised Decision Tree Model

Accuracy: Using accuracy_score from the sklearn.metrics package

```
# Finding the accuracy of the Decision Tree model implemented
model_accuracy = accuracy_score(y_pred, y_test)
model_accuracy
```

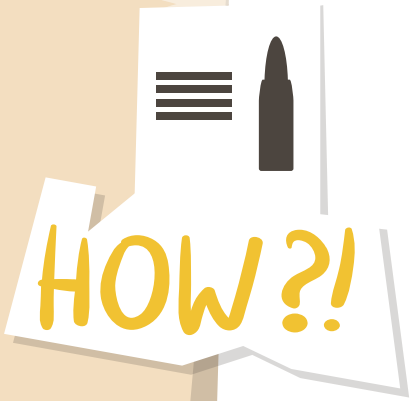
→ Obtained an accuracy score of **around 0.72**

Runtime: Using time.time from the time package

```
# Assign start time of run-time process
start_time_one = time.time()

# Copy-Paste the code from Step 1.5 - Unoptimised Decision Tree Classifier
decision_tree_classifier = DecisionTreeClassifier()
decision_tree_classifier.fit(X_train, y_train)
y_pred = decision_tree_classifier.predict(X_test)

# Assign end time of run-time process
end_time_one = time.time()

# Calculate the difference between start time and end time
run_time_one = end_time_one - start_time_one
```

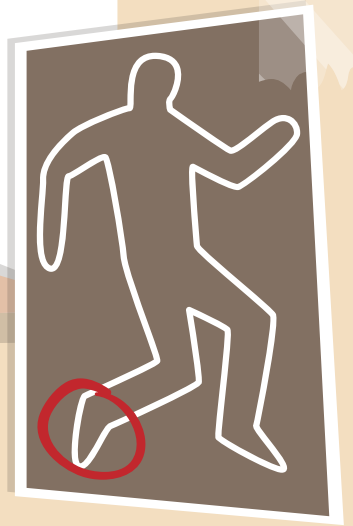
→ Obtained a runtime of around **0.2 seconds**

How To Improve The Accuracy Of The Decision Tree Model?



Optimization Techniques Used

1. Hyperparameter Tuning (GridSearchCV)
2. Hyperparameter Tuning (RandomizedSearchCV)
3. Bagging
4. Boosting



Hyperparameter Tuning - GridSearchCV

Grid Search Cross-Validation performs an exhaustive search over a manually specified subset of the hyperparameter space. It evaluates every possible combination of the hyperparameter values that is specified.

Performing GridSearchCV:

```
# Applying the Grid Search Cross Validation for the Decision Tree model
grid_search = GridSearchCV(estimator = decision_tree_classifier,
                           param_grid = param_grid_GSCV, cv = 5)

grid_search.fit(X_train, y_train)
```

Best Parameters

1. criterion : **Gini**
2. max_depth : **5**
3. min_samples_leaf : **4**
4. min_samples_split : **2**

Fitting & Testing:

```
# Similarly, perform fitting with the training data
optimised_decision_tree_GSCV.fit(X_train, y_train)

# Similarly, perform prediction on the testing data
y_pred_optimised_GSCV = optimised_decision_tree_GSCV.predict(X_test)
```

Accuracy And Runtime Of GridSearchCV Model

Accuracy :

```
# Similarly, perform accuracy score on the optimized Decision Tree model
model_accuracy_optimised_GSCV = accuracy_score(y_pred_optimised_GSCV, y_test)
```

Increased accuracy to **0.825**

Runtime:

```
# Assign start time of run-time process
start_time_two = time.time()

# Copy-Paste the code from Step 2.2 - Hyperparameter Tuning via GridSearchCV
grid_search = GridSearchCV(estimator = decision_tree_classifier,
                           param_grid = param_grid_GSCV, cv = 5)
grid_search.fit(X_train, y_train)

# Assign end time of run-time process
end_time_two = time.time()
```

```
# Assign start time of run-time process
start_time_three = time.time()

# Copy-Paste the code from Steps 3.1 and 3.2 - Optimised Decision Tree Classifie
optimised_decision_tree_GSCV = DecisionTreeClassifier(criterion = 'gini',
                                                      max_depth = 5,
                                                      min_samples_leaf = 4,
                                                      min_samples_split = 2)

optimised_decision_tree_GSCV.fit(X_train, y_train)
y_pred_optimised_GSCV = optimised_decision_tree_GSCV.predict(X_test)

# Assign end time of run-time process
end_time_three = time.time()
```

Runtime to obtain best parameters

```
Hyperparameter Tuning (GridSearchCV):

Run Time: 30.2 seconds
```

Runtime to fit model and predict outcomes

```
Fitting and Predicting (GridSearchCV):

Run Time: 0.1 seconds
```

Total runtime of GridSearchCV model: **30 seconds**

Hyperparameter Tuning - RandomizedSearchCV

Randomized Search Cross-Validation performs a random search over a specified number of parameter combinations. Instead of trying all possible combinations, it randomly samples a given number of hyperparameter combinations from a defined distribution or list.

Performing RandomizedSearchCV:

```
# Applying the Random Search Cross Validation for the Decision Tree model
randomized_search = RandomizedSearchCV(estimator = decision_tree_classifier,
                                       param_distributions = param_distributions,
                                       cv = 5, n_iter = 100,
                                       random_state = 42)

randomized_search.fit(X_train, y_train)
```

Best Parameters

1. criterion: **Gini**
2. max_depth: **5**
3. min_samples_leaf: **5**
4. min_samples_split: **9**

Fitting & Testing:

```
# Similarly, perform fitting with the training data
optimised_decision_tree_RSCV.fit(X_train, y_train)

# Similarly, perform prediction on the testing data
y_pred_optimised_RSCV = optimised_decision_tree_RSCV.predict(X_test)
```

Accuracy And Runtime Of RandomizedSearchCV Model

Accuracy :

```
# Similarly, perform accuracy score on the optimized Decision Tree model
model_accuracy_optimised_RSCV = accuracy_score(y_pred_optimised_RSCV, y_test)
```

Obtained an accuracy score of **0.825**

Runtime:

```
# Assign start time of run-time process
start_time_four = time.time()

# Copy-Paste the code from Step 2.3 - Hyperparameter Tuning (RandomizedSearchCV)
randomized_search = RandomizedSearchCV(estimator = decision_tree_classifier,
                                       param_distributions=param_distributions,
                                       cv = 5, n_iter = 100, random_state = 42)
randomized_search.fit(X_train, y_train)

# Assign end time of run-time process
end_time_four = time.time()
```

Runtime to obtain best parameters

```
Hyperparameter Tuning (RandomizedSearchCV):

Run Time: 21.1 seconds
```

```
# Assign start time of run-time process
start_time_five = time.time()

# Copy-Paste the code from Step 3.3 - Optimised Decision Tree Classifier
optimised_decision_tree_RSCV = DecisionTreeClassifier(criterion = 'gini',
                                                      max_depth = 5,
                                                      min_samples_leaf = 5,
                                                      min_samples_split = 9)
optimised_decision_tree_RSCV.fit(X_train, y_train)
y_pred_optimised_RSCV = optimised_decision_tree_RSCV.predict(X_test)

# Assign end time of run-time process
end_time_five = time.time()
```

Runtime to fit model and predict outcomes

```
Fitting and Predicting (RandomizedSearchCV):

Run Time: 0.1 seconds
```

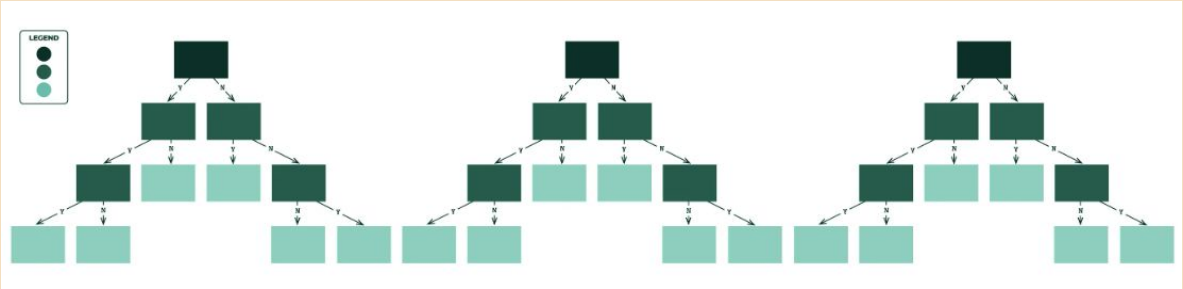
Total runtime of RandomizedSearchCV model: Around **20 seconds**

Evaluation Of The Three Models

Type Of Model	Accuracy Of Model	Runtime Of Model
Unoptimised	Around 0.72	Around 0.2 seconds
Optimised (GridSearchCV)	Around 0.825	Around 30 seconds
Optimised (RandomizedSearchCV)	Around 0.825	Around 20 seconds

- Accuracies of both **GridSearchCV** & **RandomizedSearchCV** are highly similar at **82.5%.**
- Both **GridSearchCV** & **RandomizedSearchCV have higher accuracies than the Unoptimised Decision Tree model**
- Compared to GridSearchCV, **RandomizedSearchCV** cuts down on runtime significantly
- Both the **GridSearchCV and the RandomizedSearchCV** models take longer than the **Unoptimised Decision Tree model**, which is expected as **time is required to obtain the best hyperparameters** for the model.

Bagging (Random Forest)



Each model is **trained individually**, and combined using an **averaging process.**
The primary focus of bagging is to achieve **less variance** than any model has individually.

The accuracy of the Random Forest Model might differ depending on two factors:

1) The **number of iterations** of Bagging (**n_iterations**)

2) The **number of data rows** selected for each iteration (n_**estimators**).

Bagging Process Implementation

Bagging Process (In Python):

```
n_iterations = 10
accuracies_bagging = []

# Perform bagging technique multiple times
for i in range(n_iterations):
    bagging_model = RandomForestClassifier(n_estimators = 50)
    bagging_model.fit(X_train, y_train)
    y_pred_rf_two = bagging_model.predict(X_test)
    model_accuracy_rf_two = accuracy_score(y_test, y_pred_rf_two)
    accuracies_bagging.append(model_accuracy_rf_two)
```

Step 1: Set a value for the number of iterations (n_iterations), as well as a list (accuracies_bagging) to store accuracy values.

Step 2: Create a Random Forest Model with an arbitrary number of estimators (n_estimators).

Step 3: Fit the model on the training dataset and predict the model on the test dataset.

Step 4: Obtain the accuracy score for each iteration and append the accuracy scores to the accuracies_bagging list.

Accuracy And Runtime Of Bagging Process

Accuracy:

```
# Determining the overall accuracy of the Random Forest model
print(f"\nOverall Accuracy Of Random Forest Model:
      {sum(accuracies_bagging) / len(accuracies_bagging):.3f}")
```

→ Obtained an accuracy of **around 0.79**

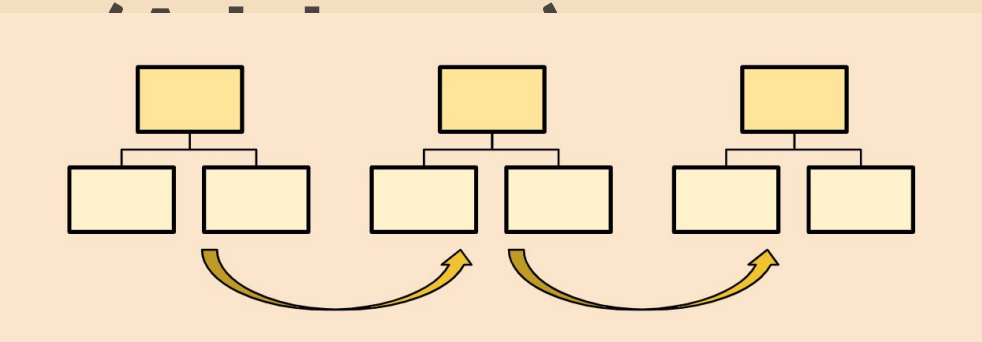
Accuracy of the **Random Forest model** is **higher** compared to the **Unoptimised Decision Tree model** but **lower** compared to **GridSearchCV and RandomizedSearchCV.**

Runtime:

```
# Start timing for the Random Forest model
start_time_rf = time.time()

# Copy-paste the Bagging Process from Step 6.1
for i in range(n_iterations):
    bagging_model = RandomForestClassifier(n_estimators = 50)
    bagging_model.fit(X_train, y_train)
    y_pred_rf_two = bagging_model.predict(X_test)

# End timing for the Random Forest model
end_time_rf = time.time()
```

→ Obtained a runtime of about **35 seconds**

Boosting



Boosting is an ensemble learning method that **combines a set of weak learners into a strong learner** to **minimise training errors**.
Boosting helps to **reduce the bias** of the model used.

The accuracy of the AdaBoost model might differ depending on two factors:

1) The **number of data rows** selected for each iteration (**n_estimators**)

2) **Maximum depth** of the individual decision trees used as weak learners (**max_depth**)

Boosting Process Implementation

Boosting Process (In Python):

```
n_iterations = 10
accuracies_boosting = []

# Performing boosting technique multiple times
for i in range(n_iterations):
    # Initialize DecisionTreeClassifier with max_depth=5
    base_tree = DecisionTreeClassifier(max_depth=5)

    # Initialize AdaBoostClassifier with the custom base estimator
    boosting_model = AdaBoostClassifier(estimator=base_tree,
                                        n_estimators=50,
                                        random_state=0,
                                        algorithm='SAMME')

    boosting_model.fit(X_train, y_train.values.ravel())
    y_pred_gb = boosting_model.predict(X_test)
    model_accuracy_gb = accuracy_score(y_test.values.ravel(), y_pred_gb)
    accuracies_boosting.append(model_accuracy_gb)
```

Step 1: Set a value for the number of iterations (n_iterations), as well as a list (accuracies_boosting) to store accuracy values.

Step 2: Create an AdaBoost Model with an arbitrary number of estimators (n_estimators).

Step 3: Fit the model on the training dataset and predict the model on the test dataset.

Step 4: Obtain the accuracy score for each iteration and append the accuracy scores to the accuracies_boosting list.

Accuracy And Runtime Of Boosting Process

Accuracy:

```
# Determining the overall accuracy of the Random Forest model
print(f"\nOverall Accuracy Of AdaBoost Model:
      {sum(accuracies_boosting) / len(accuracies_boosting):.3f}")
```

→ Obtained an accuracy of **around 0.825**

Accuracy of the **AdaBoost model** is **higher** than the **Random Forest Model, similar to the GridSearchCV and RandomizedSearchCV models.**

Runtime:

```
# Start timing for the Random Forest model
start_time_gb = time.time()

# Copy-paste the Boosting Process
for i in range(n_iterations):
    base_tree = DecisionTreeClassifier(max_depth=5)
    boosting_model = AdaBoostClassifier(estimator=base_tree,
                                        n_estimators=50,
                                        random_state=0,
                                        algorithm='SAMME')
    boosting_model.fit(X_train, y_train.values.ravel())
    y_pred_gb = boosting_model.predict(X_test)

# End timing for the Random Forest model
end_time_gb = time.time()
```

→ Obtained a runtime of about 37 seconds

Table Of The Five Models Implemented

Type Of Model	Accuracy Of Model	Runtime Of Model
Unoptimised	Around 0.72	Around 0.2 seconds
Optimised (GridSearchCV)	Around 0.825	Around 30 seconds
Optimised (RandomizedSearchCV)	Around 0.825	Around 20 seconds
Optimised (Bagging)	Around 0.79	Around 35 seconds
Optimised (Boosting)	Around 0.825	Around 40 seconds

Evaluation Of The Five Models Implemented

1) **Model Accuracy:**

★ **Boosting , GridSearchCV and RandomizedSearchCV models** have the **highest accuracies** of 82.5%
★ **Unoptimised Decision Tree model** has the **lowest accuracy** of around 72.5%
★ **Bagging** has **medium accuracy** of around 79%

2) **Model Run-time:**

★ **Unoptimised Decision Tree model** is the **fastest** - around 0.2 seconds
★ **Boosting model** is the **slowest** - around 40 seconds
★ **Bagging, GridSearchCV and RandomizedSearchCV** models have **medium efficiencies** - around 35 seconds, 30 seconds and 20 seconds respectively

However, accuracy and runtime of the models are not the only measures to determine the most suitable model to be used for the classification problem. We should turn to **other measures such as precision, recall, F1 score and micro/macro averaging** as well for a better gauge.

Performance Metrics

Multiclass Classification

Accuracy

Measures ratio of correctly predicted instances to the total number of instances

Precision

Measures accuracy of positive predictions- useful when we want to minimize false positives

Recall

Measures ability to identify all relevant instances- useful when we want to minimize false negatives

F1-Score

Harmonic mean of precision and recall - good choice when there is a trade-off between precision and recall

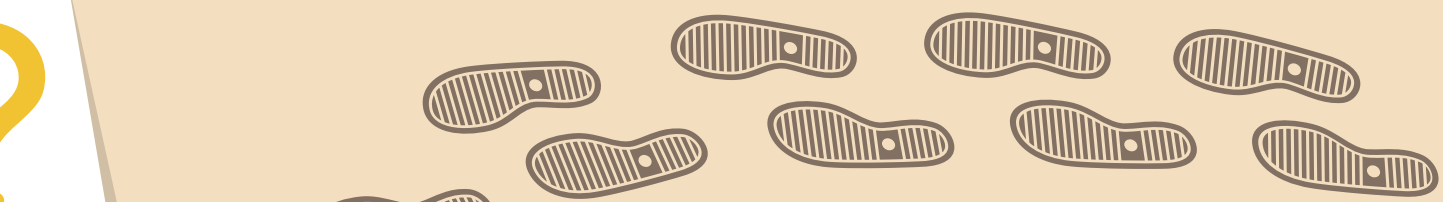
Averaging

Micro

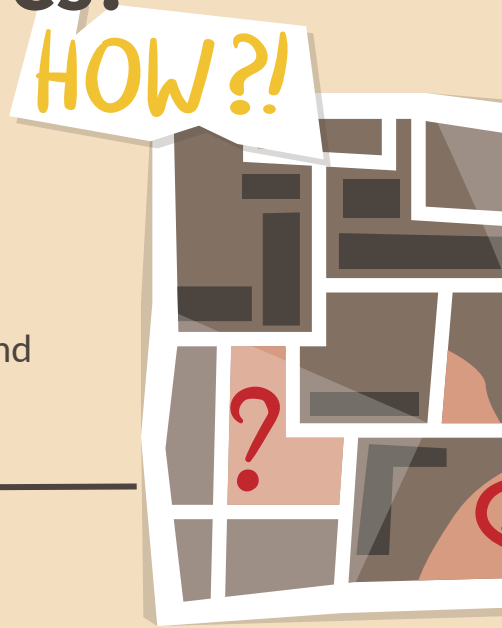
Micro-averaging aggregates contributions of all classes, sensitive to class imbalances.

Macro

Macro-averaging calculates metric independently for each class, treating all classes equally.



Implications Of False Negatives?



Suicide

- Failure to identify suicide cases have serious consequences, especially in a preventive or investigative context
- Leads to lack of necessary interventions, help and support for the family of the deceased

Homicide

- Failure to identify homicide cases would prevent a criminal investigation, leading to justice not being served
- Significant legal and ethical ramifications

Implications Of False Positives?

Suicide

- Misclassifying a non-suicide as a suicide might lead to unnecessary investigations or interventions
- In many cases, erring on the side of caution is preferable to prevent missing critical suicide cases

---

Homicide

- Misclassifying a death as a homicide when it is not could lead to unnecessary legal investigations, accusations, or resources spent
- Strain judicial resources, cause distress for involved families, or lead to wrongful accusations

False Positives VS False Negatives

False Positives (Less Costly)

False Negatives (More Costly)

- For both Suicide and Homicide cases, false negatives are more costly, as they lead to missed critical cases where urgent interventions or legal actions might be necessary
- False positives are generally less costly than false negatives though they can still lead to unnecessary interventions or resource usage

Therefore, we need to focus more on obtaining a **high value of recall** compared to a high value of precision.

Are Dataset Outcomes Balanced/Imbalanced?

```
# Converting y_train From A Series To A Dataframe
y_train = pd.DataFrame(y_train)

print("Overview Of y_train Dataset:")
print("\n")

# Find Number Of Rows In y_train That Has Homicide As Reason
y_train_homicide = y_train[y_train['Reason Encoded'] == 0]
print(f"Number Of Rows Of Homcide Class: {len(y_train_homicide)}")

# Find Number Of Rows In y_train That Has Others As Reason
y_train_others = y_train[y_train['Reason Encoded'] == 1]
print(f"Number Of Rows Of Others Class: {len(y_train_others)}")

# Find Number Of Rows In y_train That Has Suicide As Reason
y_train_suicide = y_train[y_train['Reason Encoded'] == 2]
print(f"Number Of Rows Of Suicide Class: {len(y_train_suicide)}")
```

```
Overview Of y_train Dataset:


Number Of Rows Of Homcide Class: 26262
Number Of Rows Of Others Class: 1913
Number Of Rows Of Suicide Class: 47760
```

From the training data, we observe:

- **26262 Homicide** cases
- **1913 Others** cases
- **47760 Suicide** cases

The dataset outcomes are **highly imbalanced**, where Suicide is the dominant outcome

Micro-Averaging is thus **a more suitable metric** to use compared to Macro-Averaging

Creating The Confusion Matrix

We visualised the **confusion matrix** of the various outcomes using a heatmap in Seaborn

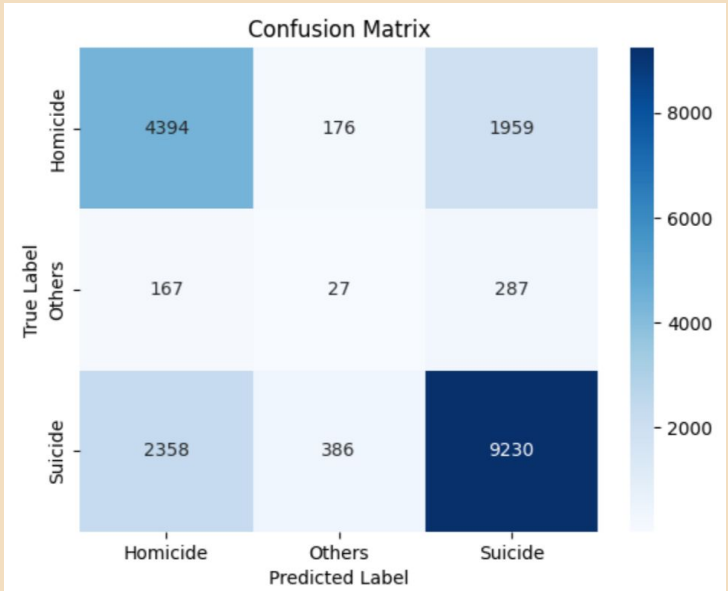
Creating The Confusion Matrix:

```
from sklearn.metrics import confusion_matrix

# Create the confusion matrix
cm = confusion_matrix(y_test, y_pred)
```

Visualising The Confusion Matrix:

```
# Visualize the confusion matrix using seaborn
sns.heatmap(cm, annot = True, fmt = "d", cmap = "Blues",
            xticklabels = ['Homicide', 'Others', 'Suicide'],
            yticklabels = ['Homicide', 'Others', 'Suicide'])

plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.title('Confusion Matrix')
plt.show()
```

Confusion Matrix Obtained:



Calculating Precision, Recall and F1-Score

We calculated the Precision, Recall and F1-Scores of the 5 models used

Unoptimised

```
precision_unoptimised = precision_score(y_test, y_pred, average = 'micro')
recall_unoptimised = recall_score(y_test, y_pred, average = 'micro')
f1_unoptimised = f1_score(y_test, y_pred, average = 'micro')
```

Optimised (GridSearchCV)

```
precision_GSCV = precision_score(y_test, y_pred_optimised_GSCV, average = 'micro')
recall_GSCV = recall_score(y_test, y_pred_optimised_GSCV, average = 'micro')
f1_GSCV = f1_score(y_test, y_pred_optimised_GSCV, average = 'micro')
```

Optimised (RandomizedSearchCV)

```
precision_RSCV = precision_score(y_test, y_pred_optimised_RSCV, average = 'micro')
recall_RSCV = recall_score(y_test, y_pred_optimised_RSCV, average = 'micro')
f1_RSCV = f1_score(y_test, y_pred_optimised_RSCV, average = 'micro')
```

Optimised (Bagging)

```
precision_bagging = precision_score(y_test, y_pred_rf_two, average = 'micro')
recall_bagging = recall_score(y_test, y_pred_rf_two, average = 'micro')
f1_bagging = f1_score(y_test, y_pred_rf_two, average = 'micro')
```

Optimised (Boosting)

```
precision_boosting = precision_score(y_test, y_pred_gb, average = 'micro')
recall_boosting = recall_score(y_test, y_pred_gb, average = 'micro')
f1_boosting = f1_score(y_test, y_pred_gb, average = 'micro')
```

Results Obtained For The 5 Models

Decision Tree Model	Accuracy	Precision	Recall	F1-Score
Unoptimised	0.719	0.719	0.719	0.719
Optimised (GridSearchCV)	0.825	0.825	0.825	0.825
Optimised (RandomizedSearchCV)	0.825	0.825	0.825	0.825
Optimised (Bagging)	0.794	0.792	0.792	0.792
Optimised (Boosting)	0.824	0.824	0.824	0.824

- **GridSearchCV and RandomizedSearchCV Models** have the **highest** accuracy, precision, recall and F1-score of 82.5**%.**
- **Unoptimised Decision Tree Model** has the **lowest** accuracy, precision, recall and F1-score at **71.9%**.

Suitable Sampling Methods For Dataset

Recall that the dataset outcomes are highly imbalanced

Sampling Techniques

Undersampling → Removal of date from majority class to fit size of minority class → 

- More than 95% of rows in Suicide removed
- More than 90% of rows in Homicide removed
- Substantial loss in data, compromising accuracy due to missing trends in data

Oversampling → Duplicating data from minority class to fit size of majority class → 

SMOTE → Interpolating between minority class instances to fit size of majority class → 

- No loss in data as no records are removed
- Duplicating or adding records through interpolation is possible to meet number of records of “Suicide” class

Implementing Oversampling And SMOTE

Oversampling

Using **RandomOverSampler** from **imblearn.over_sampling package**

```
# Initialize the RandomOverSampler
ros = RandomOverSampler(random_state = 0)

# Apply oversampling to X_train and y_train
X_resampled, y_resampled = ros.fit_resample(X_train, y_train)
```

SMOTE

Using **SMOTE** from **imblearn.over_sampling** package

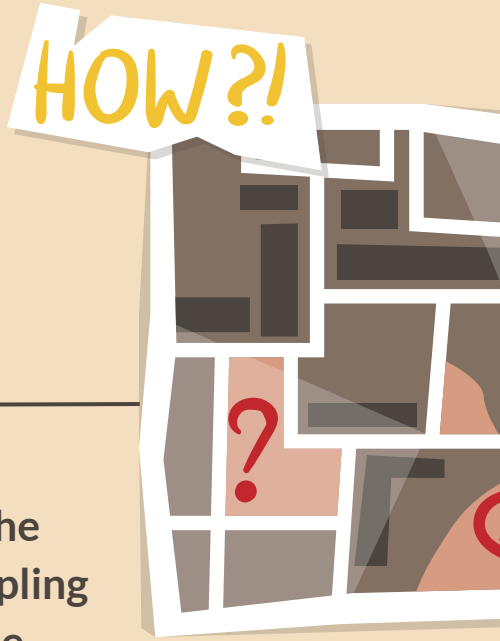
```
# Initialize the RandomOverSampler
ros_two = SMOTE(random_state = 0)

# Apply oversampling to X_train and y_train
X_SMOTE, y_SMOTE = ros_two.fit_resample(X_train, y_train)
```

```
Overview Of y_resampled Dataset:


Number Of Rows Of Homcide Class: 47760
Number Of Rows Of Others Class: 47760
Number Of Rows Of Suicide Class: 47760
```

Number of rows of Homicide, Others and Suicide outcomes are all exactly the same - **47760 rows**

Changes In Suitable Metrics After Resampling

Since the dataset outcomes are now completely balanced, we will use **macro-averaging instead of micro-averaging** to compute the Precision, Recall and F1-Score metrics



As for Precision, Recall and F1-Score, **Recall is still the most important** among the three as **applying resampling techniques has no effect on the implications of False Positives and False Negatives**

(ie. False Negatives are still more important than False Positives)

Re-Running Decision Tree - Oversampling

Unoptimised Decision Tree Model:

```
# Unoptimised Decision Tree Re-Implemented using resampled datasets
decision_tree_classifier.fit(X_resampled, y_resampled)
y_pred_oversampling = decision_tree_classifier.predict(X_test)
```

Bagging Process:

```
# Bagging Process Re-Implemented using resampled datasets
for i in range(n_iterations):
    bagging_oversampling = RandomForestClassifier(n_estimators = 50)
    bagging_oversampling.fit(X_resampled, y_resampled.values.ravel())
    y_pred_bagging_oversampling = bagging_oversampling.predict(X_test)
```

Boosting Process:

```
# Boosting Process Re-Implemented using resampled datasets
for i in range(n_iterations):
    base_tree = DecisionTreeClassifier(max_depth=5)
    boosting_oversampling = AdaBoostClassifier(estimator = base_tree,
                                               n_estimators = 50,
                                               random_state = 0,
                                               algorithm='SAMME')
    boosting_oversampling.fit(X_resampled, y_resampled.values.ravel())
    y_pred_boosting_oversampling = boosting_oversampling.predict(X_test)
```

GridSearchCV and RandomizedSearchCV:

Obtaining The Optimal Parameters:

```
grid_search.fit(X_resampled, y_resampled)
print("Best Parameters (Grid Search CV): ",grid_search.best_params_)

randomized_search.fit(X_resampled, y_resampled)
print("Best Parameters (Randomized Search CV): ",randomized_search.best_params_)
```

Fitting And Predicting Using Optimal Parameters:

```
# GridSearchCV Re-Implemented using resampled datasets
optimised_GSCV_oversampling = DecisionTreeClassifier(criterion = 'gini',
                                                     max_depth = 5,
                                                     min_samples_leaf = 2,
                                                     min_samples_split = 2)

optimised_GSCV_oversampling.fit(X_resampled, y_resampled)
y_pred_GSCV_oversampling = optimised_GSCV_oversampling.predict(X_test)

# RandomizedSearchCV Re-Implemented using resampled datasets
optimised_RSCV_oversampling = DecisionTreeClassifier(criterion = 'gini',
                                                     max_depth = 5,
                                                     min_samples_leaf = 5,
                                                     min_samples_split = 9)

optimised_RSCV_oversampling.fit(X_resampled, y_resampled)
y_pred_RSCV_oversampling = optimised_RSCV_oversampling.predict(X_test)
```

Re-Running Decision Tree - SMOTE

Unoptimised Decision Tree Model:

```
# Unoptimised Decision Tree Re-Implemented using SMOTE datasets
decision_tree_classifier.fit(X_SMOTE, y_SMOTE)
y_pred_SMOTE = decision_tree_classifier.predict(X_test)
```

Bagging Process:

```
# Bagging Process Re-Implemented using SMOTE datasets
for i in range(n_iterations):
    bagging_SMOTE = RandomForestClassifier(n_estimators=50)
    bagging_SMOTE.fit(X_SMOTE, y_SMOTE.values.ravel())
    y_pred_bagging_SMOTE = bagging_SMOTE.predict(X_test)
```

Boosting Process:

```
# Boosting Process Re-Implemented using SMOTE datasets
for i in range(n_iterations):
    base_tree = DecisionTreeClassifier(max_depth=5)
    boosting_SMOTE = AdaBoostClassifier(estimator = base_tree,
                                        n_estimators = 50,
                                        random_state = 0,
                                        algorithm='SAMME')
    boosting_SMOTE.fit(X_SMOTE, y_SMOTE.values.ravel())
    y_pred_boosting_SMOTE = boosting_SMOTE.predict(X_test)
```

GridSearchCV and RandomizedSearchCV:

Obtaining The Optimal Parameters:

```
grid_search.fit(X_SMOTE, y_SMOTE)
print("Best Parameters (Grid Search CV): ",grid_search.best_params_)

randomized_search.fit(X_SMOTE, y_SMOTE)
print("Best Parameters (Randomized Search CV): ",randomized_search.best_params_)
```

Fitting And Predicting Using Optimal Parameters:

```
# GridSearchCV Re-Implemented using SMOTE datasets
optimised_GSCV_SMOTE = DecisionTreeClassifier(criterion = 'gini', max_depth = 5,
                                              min_samples_leaf = 2,
                                              min_samples_split = 2)

optimised_GSCV_SMOTE.fit(X_SMOTE, y_SMOTE)
y_pred_GSCV_SMOTE = optimised_GSCV_SMOTE.predict(X_test)


# RandomizedSearchCV Re-Implemented using SMOTE datasets
optimised_RSCV_SMOTE = DecisionTreeClassifier(criterion = 'gini', max_depth = 5,
                                              min_samples_leaf = 5,
                                              min_samples_split = 9)

optimised_RSCV_SMOTE.fit(X_SMOTE, y_SMOTE)
y_pred_RSCV_SMOTE = optimised_RSCV_SMOTE.predict(X_test)
```

Metrics Results Obtained

No Sampling Techniques:

Decision Tree Model	Accuracy	Precision	Recall	F1-Score
Unoptimsed	0.719	0.719	0.719	0.719
Optimised - GridSearchCV	0.825	0.825	0.825	0.825
Optimised - RandomizedSearchCV	0.825	0.825	0.825	0.825
Optimised - Bagging	0.794	0.792	0.792	0.792
Optimised - Boosting	0.824	0.824	0.824	0.824

With Oversampling:

Decision Tree Model	Accuracy	Precision	Recall	F1-Score
Unoptimsed	0.711	0.489	0.485	0.487
Optimised - GridSearchCV	0.740	0.577	0.558	0.548
Optimised - RandomizedSearchCV	0.740	0.576	0.558	0.548
Optimised - Bagging	0.779	0.534	0.533	0.533
Optimised - Boosting	0.706	0.563	0.578	0.537

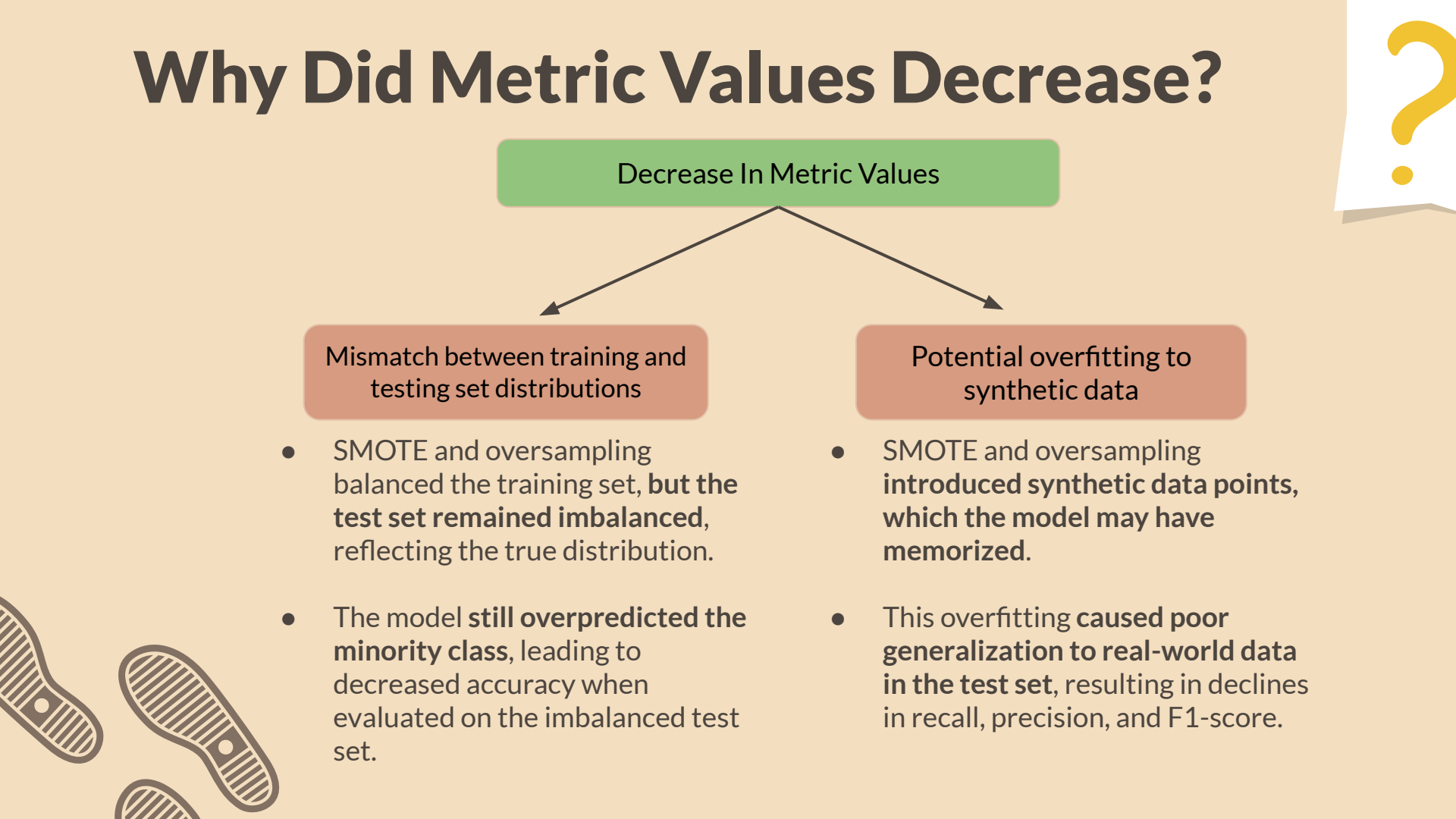
With SMOTE:

Decision Tree Model	Accuracy	Precision	Recall	F1-Score
Unoptimsed	0.691	0.502	0.491	0.491
Optimised - GridSearchCV	0.673	0.561	0.561	0.522
Optimised - RandomizedSearchCV	0.673	0.561	0.561	0.522
Optimised - Bagging	0.746	0.527	0.535	0.528
Optimised - Boosting	0.710	0.562	0.572	0.537

Key Observations And Comments:

- ★ With Sampling Techniques involved, the Precision, Recall and F1-Score **decrease to significantly lower values**
- ★ The **accuracy** of the Decision Tree models also **decrease** with sampling, **but to a lesser extent**
- ★ The Accuracy, Precision, Recall and F1-Score of **Oversampling is about the same as SMOTE**

Why Did Metric Values Decrease?



- SMOTE and oversampling balanced the training set, **but the test set remained imbalanced**, reflecting the true distribution.
- The model **still overpredicted the minority class**, leading to decreased accuracy when evaluated on the imbalanced test set.

- SMOTE and oversampling **introduced synthetic data points, which the model may have memorized**.
- This overfitting **caused poor generalization to real-world data in the test set**, resulting in declines in recall, precision, and F1-score.

Choosing The Best Decision Tree Model

1. We observe that **Boosting gives the highest recall** among all the five Decision Tree models

2. Although Boosting has **the highest runtime**, a **runtime of about 40 seconds is acceptable** given the small size of the dataset

3. Although **Oversampling and SMOTE have the same Recall value**, **SMOTE** is preferred:

    ★ **Reduces** likelihood of model **overfitting** by introducing diversity in the minority class

    ★ Help model **learn patterns that generalize better to unseen data**, leading to improved performance on test datasets

What we aim for:

- **Highest Recall**
- **Small runtime**

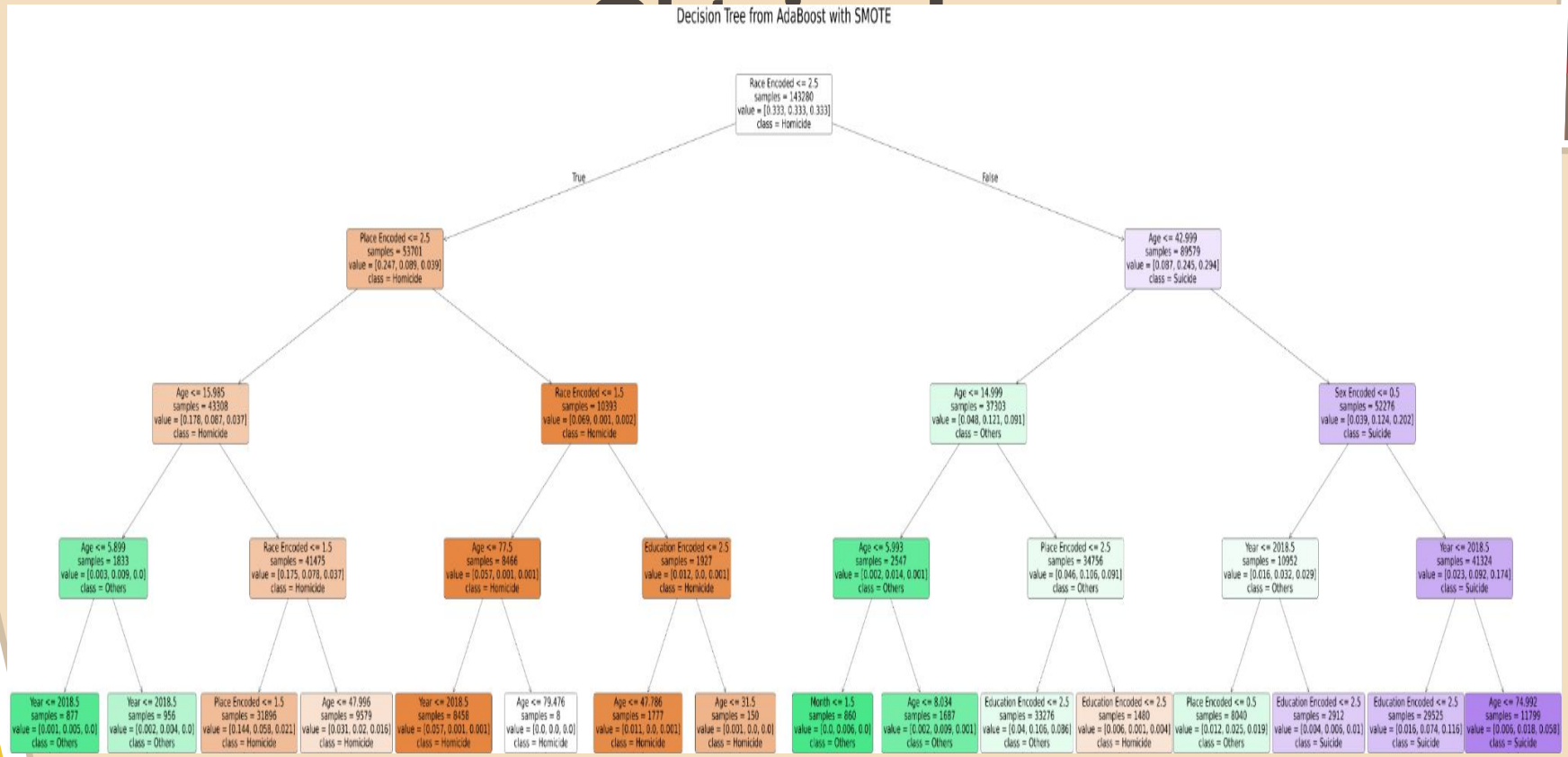
Conclusion: We will use **AdaBoost Classifier with SMOTE resampling technique**

Constructing The Decision Tree Model

```
tree_to_plot = boosting_SMOTE.estimators_[0]

plt.figure(figsize=(130, 40), dpi=300)

plot_tree(tree_to_plot, filled=True, rounded=True, fontsize=30,
          class_names=class_names, feature_names=feature_names,
          impurity=False, max_depth=4, proportion=False)

plt.title("Decision Tree from AdaBoost with SMOTE", fontsize=50)
plt.tight_layout()
plt.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)
plt.show()
```

- We take the **first iteration of the boosting process** to construct the tree
- We set a **relevant figure size** to plot the decision tree, setting dpi to 300
- Using the **plot_tree** function, we plot the Decision Tree, setting **max_depth to 4 to prevent any overlap among the nodes**

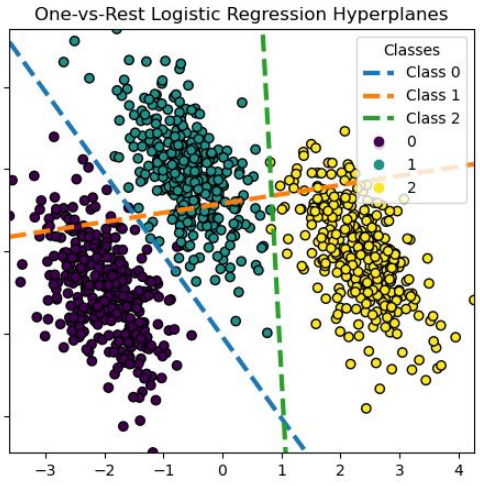
Final Decision Tree Graph



Model 2 - Logistic Regression (LR)

One-vs-Rest and Multinomial

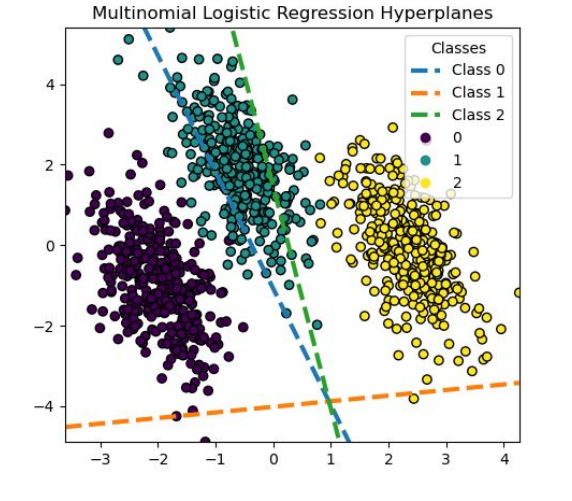
One-vs-Rest (OVR)



Each decision boundary is derived independently by considering one class against a group of all others.

E.g: For class 1, the hyperplane represents the decision boundary that best separates class 1 from the **combined** classes 0 and 2.

Multinomial



Decision boundaries are determined simultaneously, considering the relationships between classes together.

Optimised to estimate the conditional class probabilities.

Each decision boundary corresponds to where the probability of one class becomes higher than the others, based on the overall probability distribution.

Image Sources: https://scikit-learn.org/dev/auto_examples/linear_model/plot_logistic_multinomial.html

Implementing LR Model

1st Model: **Unoptimised One-vs-Rest (OVR) LR Model** (No Optimisation)

```
# Create Logistic Regression model without specifying the solver
lr_basic_model_ovr = LogisticRegression(max_iter=1000, multi_class='ovr')

# Fit the model directly with the training data
lr_basic_model_ovr.fit(X_train_scaled, y_train)

y_pred_lr_basic_ovr = lr_basic_model_ovr.predict(X_test_scaled)
```

Unoptimised Model Implementation

- Call LogisticRegression from sklearn.linear_model, with multi_class = ‘OVR’
- Fit the training dataset into the model
- Predict the target, y_pred_lr_basic,_ovr using the test dataset

Implementing LR Model

1st Model: **Unoptimised Multinomial LR Model** (No Optimisation)

```
# Create Logistic Regression model without specifying the solver
lr_basic_model = LogisticRegression(max_iter=1000, multi_class='multinomial')

# Fit the model directly with the training data
lr_basic_model.fit(X_train_scaled, y_train)

y_pred_lr_basic = lr_basic_model.predict(X_test_scaled)
```

Unoptimised Model Implementation

- Call LogisticRegression from sklearn.linear_model, with multi_class = ‘Multinomial’
- Fit the training dataset into the model
- Predict the target, y_pred_lr_basic, using the test dataset

Implementing LR Model

Finding the **accuracy** of the **Unoptimised LR models**

Accuracy: Using accuracy_score from the sklearn.metrics package

```
# Calculate accuracy
print("Accuracy score:",accuracy_score(y_test, y_pred_lr_basic_ovr))
```

OVR Model obtained an accuracy score of **around 0.82706**

```
# Calculate accuracy
print("Accuracy score:",accuracy_score(y_test, y_pred_lr_basic))
```

Multinomial Model obtained an accuracy score of **around 0.82712**

Hyperparameter Tuning - GridSearchCV

Parameters Used:

1. C (inverse of regularization)
2. Solver

```
# Define Param Grid
param_grid = {
    'C': [0.0001, 0.001, 0.01, 0.1, 1, 5, 10],  # Inverse of regularization strength
    'multi_class': ['ovr'],
    'solver': ['lbfgs', 'sag', 'saga'],  # Trying different solvers
}
```

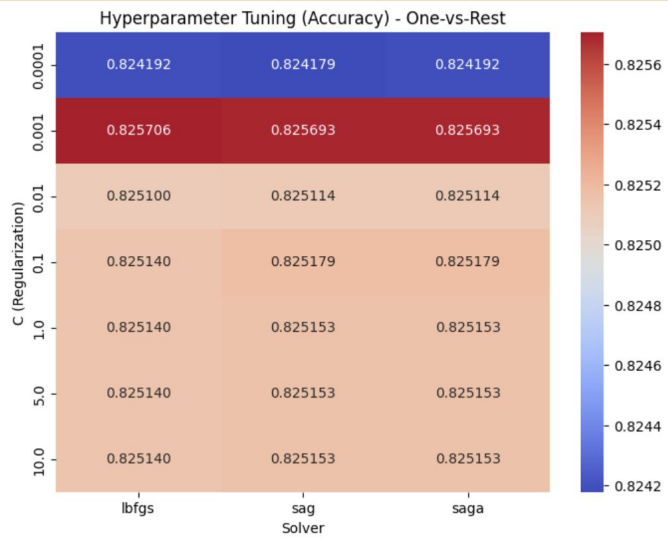
GridSearch Parameter Grid for OVR Model

```
# Define the hyperparameter distributions
param_grid = {
    'C': [0.0001, 0.001, 0.01, 0.1, 1, 5, 10], # Inverse of regularization strength
    'multi_class': ['multinomial'],
    'solver': ['lbfgs', 'sag', 'saga'], # Trying different solvers
}
```

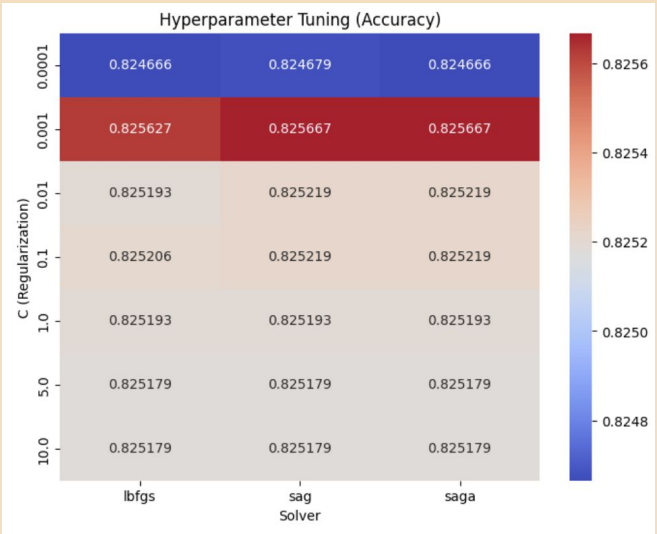
GridSearch Parameter Grid for Multinomial Model

Heat Maps - GridSearchCV

Heat Map for OVR Model:



Heat Map for Multinomial Model:



Red Cell = Higher Accuracy

Hyperparameter Tuning - GridSearchCV

Performing GridSearchCV:

```
# Set up GridSearchCV
grid_search = GridSearchCV(model, param_grid, cv=5, scoring='accuracy', n_jobs=-1)

# Fit the model with Grid Search
grid_search.fit(X_train_scaled, y_train)
```

Best Parameters for **OVR Model:** C: 0.001, solver: lbfgs

```
    # GridSearchCV
    grid_search_ovr = GridSearchCV(model_ovr, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
    # Fit the model with Grid Search
    grid_search_ovr.fit(X_train_scaled, y_train)
```

Best Parameters for **Multinomial Model:** C: 0.001, solver: sag

Re-implementation

Running the Models again with the **best parameters** from **GridSearchCV:**

OVR Model:

```
lr_model_ovr = grid_search_ovr.best_estimator_
y_pred_ovr = lr_model_ovr.predict(X_test_scaled)
```

Multinomial Model:

```
lr_model_mult1 = grid_search.best_estimator_
y_pred_lr_gs = lr_model_mult1.predict(X_test_scaled)
```

Fitting & Testing:

OVR Model:

```
# Calculate accuracy
print("Accuracy score:",accuracy_score(y_test, y_pred_ovr))
```

Multinomial Model:

```
# Calculate accuracy
print("Accuracy score:",accuracy_score(y_test, y_pred_lr_gs))
```

Analysis of Accuracies

Model:	Accuracy:
OVR Model (Baseline)	around 0.82706
OVR Model (GridSearchCV)	**around 0.82728**

Using the GridSearchCV gave us a slighter higher accuracy for the OVR Model.

Model:	Accuracy:
Multinomial Model (Baseline)	**around 0.82712**
Multinomial Model (GridSearchCV)	around 0.82691

However, using the GridSearchCV resulted in a lower accuracy for the Multinomial Model. This outcome could be attributed to the parameter grid we provided during the search.

Hyperparameter Tuning - RandomizedSearchCV

Parameters Used:

1. C (inverse of regularization)
2. Solver

```
# Define the refined parameter distribution for RandomizedSearchCV
param_distributions = {
    'C': uniform(0.0001, 1),  # Controlled C values up to 3 decimal places
    'multi_class': ['ovr'],        # One-vs-Rest strategy
    'solver': ['lbfgs', 'sag', 'saga'],  # Trying different solvers
}
```

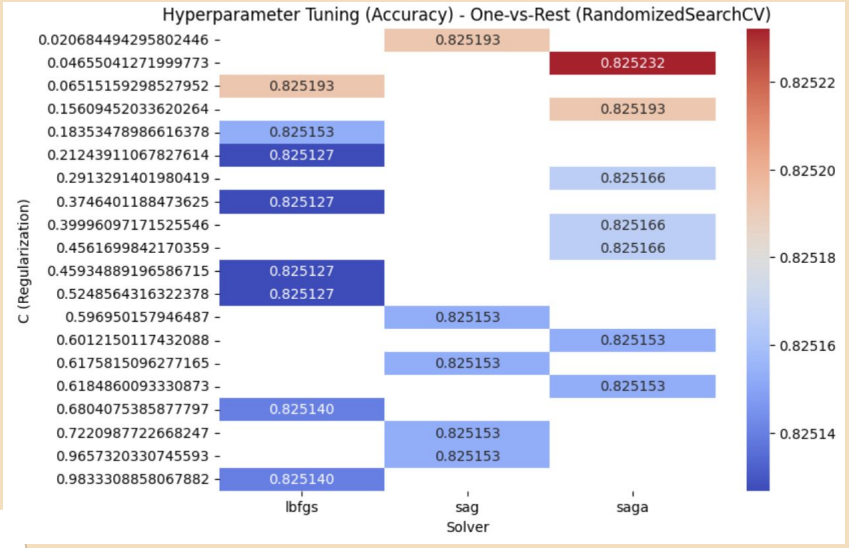
RandomizedSearch Parameter Grid for OVR Model

```
# Define the refined parameter distribution with predefined C values to reduce clutter
param_distributions = {
    'C': uniform(0.0001, 1),  # Controlled C values up to 3 decimal places
    'multi_class': ['multinomial'],               # One-vs-Rest strategy
    'solver': ['lbfgs', 'sag', 'saga'],    # Trying different solvers
}
```

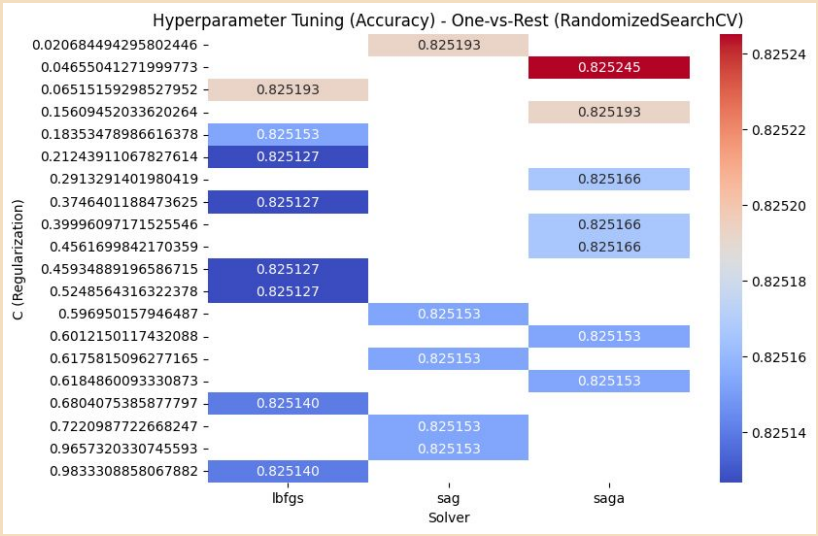
RandomizedSearch Parameter Grid for Multinomial Model

RandomizedSearchCV - Heat Maps

Heat Map for OVR Model:



Heat Map for Multinomial Model:



Red Cell = Higher Accuracy

Hyperparameter Tuning - RandomizedSearchCV

Performing RandomizedSearchCV:

```
# RandomizedSearchCV
random_search_ovr = RandomizedSearchCV(model_ovr, param_distributions,
                                       n_iter=20, cv=5, scoring='accuracy',
                                       n_jobs=-1, random_state=42)

# Fit the model with Randomized Search
random_search_ovr.fit(X_train_scaled, y_train)
```

Best Parameters for **OVR Model:** C: 0.04655, solver: saga

```
# Set up RandomizedSearchCV with fewer iterations for faster performance
random_search_mult2 = RandomizedSearchCV(model, param_distributions, n_iter=20, cv=5,
                                       scoring='accuracy', n_jobs=-1, random_state=42)

# Fit the model with Randomized Search
random_search_mult2.fit(X_train_scaled, y_train)
```

Best Parameters for **Multinomial Model:** C: 0.02068, solver: sag

Re-implementation

Running the Models again with the **best parameters** from **RandomizedSearchCV:**

OVR Model:

```
lr_model_ovr2 = random_search_ovr.best_estimator_
y_pred_ovr2 = lr_model_ovr2.predict(X_test_scaled)
```

Multinomial Model:

```
lr_model_mult2 = random_search_mult2.best_estimator_
y_pred_lr_rs = lr_model_mult2.predict(X_test_scaled)
```

Fitting & Testing:

OVR Model:

```
# Calculate accuracy
print("Accuracy score:",accuracy_score(y_test, y_pred_ovr2))
```

Multinomial Model:

```
# Calculate accuracy
print("Accuracy score:", accuracy_score(y_test, y_pred_lr_rs))
```

Analysis of Accuracies

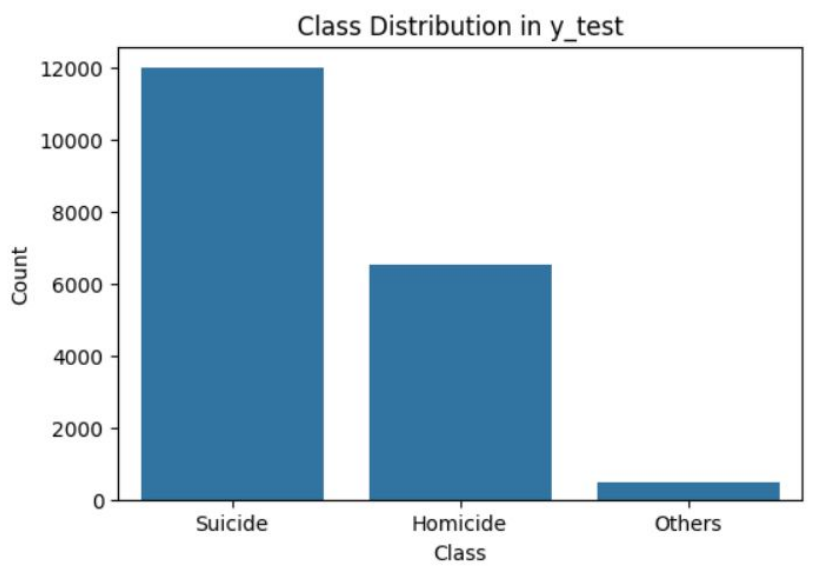
Model:	Accuracy:
OVR Model (Baseline)	around 0.82706
OVR Model (RandomizedSearchCV)	**around 0.82712**

Using the RandomizedSearchCV gave us a slightly higher accuracy for the OVR Model.

Model:	Accuracy:
Multinomial Model (Baseline)	**around 0.82712**
Multinomial Model (RandomizedSearchCV)	around 0.82696

However, using the RandomizedSearchCV resulted in a lower accuracy for the Multinomial Model. This outcome could be attributed to similar factors: Parameter Sampling, Random State and Baseline Robustness.

Data Sampling



By visualising the class distribution of the train set, we can see that "Others" is significantly underrepresented compared to the other two classes. When visualising the class distribution of the test set, we see that it actually follows the same **imbalanced class distribution**.

Applying Undersampling

Justification for Undersampling:

- Learning from past works
- Possible problems from oversampling (creating an artificially balanced train set)

- Using undersampling:
  - Will improve model's runtime
  - Will try to mitigate some data reduction issues

```
sampling_strategy = {
    0: 5000,
    2: 5000,
    1: 1913
}
# Create a RandomUnderSampler object
rus = RandomUnderSampler(random_state=0, sampling_strategy=sampling_strategy)
X_train_resampled, y_train_resampled = rus.fit_resample(X_train, y_train)
```

Analysis of Accuracies

Model:	Accuracy:
OVR Model (Baseline)	around 0.82706
OVR w/ undersampling (Baseline)	around 0.81063
OVR w/ undersampling (GridSearchCV)	around 0.81058
OVR w/ undersampling (RandomizedSearchCV)	around 0.81063

- The OVR model without data sampling performed better than the model with sampling but this is to be expected
  - Undersampling reduces the dataset size by removing instances of the majority class, potentially leading to the loss of important information

Analysis of Accuracies

Model:	Accuracy:
Multinomial Model (Baseline)	around 0.82712
Multinomial w/ undersampling (Baseline)	around 0.81142
Multinomial w/ undersampling (GridSearchCV)	**around 0.81184**
Multinomial w/ undersampling (RandomizedSearchCV)	around 0.81142

- Similarly, the Multinomial model without data sampling performed better than the model with sampling but this is to be expected likely due to the same reason
- However, if we compared amongst all the undersampled models, using GridSearchCV actually helped to improve the model's accuracy

Comparing Micro-Averages

→ Despite the lower accuracy, it is important for us to carry out data sampling as it can better ensure the model learns to identify patterns in minority classes, which may be more critical in real-world applications

Using the data that has been undersampled:

Model	Precision	Recall	F1 Score
OVR (GridSearchCV)	0.81058	0.81058	0.81058
OVR (RandomizedSearchCV)	0.81063	0.81063	0.81063
Multinomial (GridSearchCV)	**0.81184***	**0.81184***	**0.81184***
Multinomial (RandomizedSearchCV)	0.81142	0.81142	0.81142

- As we can see, the Multinomial Model (GridSearchCV) gave us the best Precision, Recall and F1 Score.

Comparing Time

→ Using **time.time** from the time package to measure how long each model takes.

Using the data that has been undersampled:

Model	Set-Up Time (s)	Run Time (s)	Prediction Time (s)
OVR (GridSearchCV)	0.008190	4.560332	0.001946
OVR (RandomizedSearchCV)	0.016420	5.386720	0.002269
Multinomial (GridSearchCV)	**0.005371***	3.708657	**0.001885***
Multinomial (RandomizedSearchCV)	0.006578	**2.432895***	0.001963

- In our analysis of the different model configurations, we observed that the time taken for set-up, run, and prediction across all the models is relatively similar, with only minor variations.

Choosing the Best Model

- Overall, the **Multinomial Model (GridSearch)** emerged as the most consistent option, given its strong performance across both Cross-Validation and Test Accuracy Scores.

Model	Best Cross-Validation Score	Test Accuracy Score
Multinomial (w/ GridSearch)	**0.6957099916741627***	**0.8118415507796038***

- The Multinomial Model (GridSearch) also achieved the highest precision, recall, and F1 scores, indicating superior performance in handling the imbalanced classes.

- Not only that, the Multinomial Model (GridSearch) had the quickest Set-Up time and Prediction Time, and the second fastest Run Time.

- Given these results, we conclude that the best model is the Multinomial Model (GridSearch), which was not only fast but had a high accuracy as well.

05

Questions Analysis

Questions For Consideration

1. Which age, race or gender groups are most affected by suicide incidents?

2. Is there a significant racial disparity in gun violence incidents?

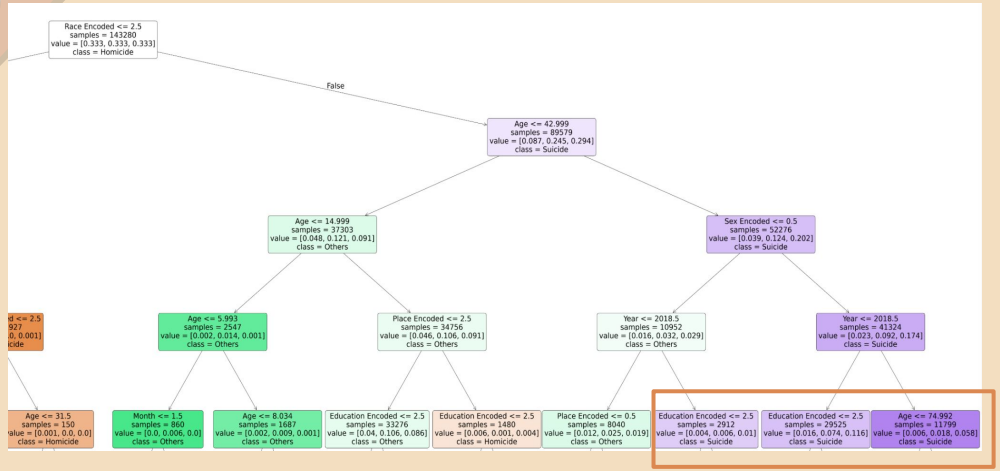
3. Do the locations where these incidents happen (at home or on the street) make a difference in the type of fatal event?

4. Are certain gun crimes more likely to occur in certain months or years?

Question 1

Which age, race or gender group is most affected by suicide incidents? (Decision Tree)

Right Branch Of Decision Tree Graph



- ★ Nodes with Suicide as the majority are **highlighted in purple**
- ★ The **rightmost 3 nodes** are in purple
- ★ The **rightmost node has the darkest shade of purple**, indicating **highest proportion of Suicide cases**
- ★ This Corresponds to:
  - Race > 2.5
  - Age > 75
  - Sex = 1
  - Year = 2019

Conclusion:

People who are **aged 75 years and above,** are **male,** and are **either Native American/Native Alaskan or White** are most affected by Suicide incidents, particularly incidents **that occurred in 2019.**

Question 2

Is there a significant racial disparity in gun violence incidents? (Logistic Regression) A: Yes!

	Feature	Coefficient	Odds Ratio
9	Race_Black	0.336461	1.40
10	Race_Hispanic	0.045148	1.05
11	Race_Native American/Native Alaskan	-0.047464	0.95
12	Race_White	-0.484108	0.62

- ★ Referring to the table, **Black** had the highest coefficient of 0.336 and odds ratio of 1.40.
- ★ While **White** had a negative coefficient of -0.484 and the smallest odds ratio of 0.62.
- ★ There is a significant racial disparity in gun violence incidents.

Question 3

Do the locations where these incidents happen make a difference in the type of fatal event? (Logistic Regression) A: Yes!

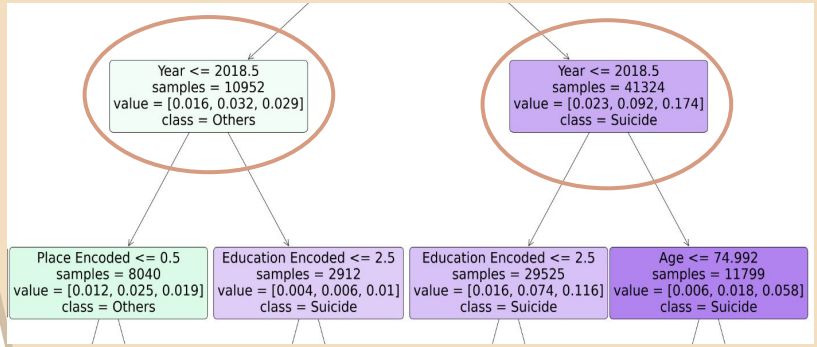
	Feature	Coefficient	Odds Ratio
13	Place of incident_Miscellaneous	0.143928	1.15
14	Place of incident_Other Places	0.085620	1.09
15	Place of incident_Street	0.398962	1.49

- ★ Observe that **Street**s had the **highest coefficient of 0.399** and **odds ratio of 1.49** among the places of incidence
  - This suggest that the odds of a gun incident occurring on the street is 49% higher than the other categories.

Question 4

Are certain gun crimes more likely to occur in certain months or years? (Decision Tree) A: Yes!

Part Of Decision Tree Model Involving Year



Conclusion:

Year 2017 and 2018: Majority Suicide or Others
Year 2019: Majority Suicide

- ★ **For the year 2019**, both branches of the tree (on the right side) lead to classifications of **Suicide**, indicating that in 2019, suicide cases are more consistently identified.
- ★ **For earlier years (2018 and 2017)**, there is a split in classification: one branch each leads to **Suicide** and **Others**.
- ★ This pattern hints at a possible **temporal trend**, where **more recent years, such as 2019, see a stronger association with suicide cases**, potentially reflecting an increase in gun-related suicides over time.

Thank You !