In the group part, we loaded, cleaned and preprocessed the data.I will create several models to predict the casualty serverity based on independent variables. Table of Contents

1.0 - Business Objective

Improving Casualty Severity Prediction and Response in the Health Sector

In the health sector, one of the primary objectives is to reduce fatalities and severe injuries from road accidents while improving the quality of care for casualties. Accurate and timely prediction of casualty severity can significantly enhance emergency response, allocate resources more effectively, and provide data-driven insights for preventative measures.

The current challenges in the health sector revolve around insufficient real-time data on casualty severity, which limits emergency response effectiveness, resource allocation, and subsequent medical interventions. Inadequate prediction models can lead to delays in providing appropriate treatment, with a high cost in terms of lives and hospital resources. There is an urgent need for advanced predictive models that integrate variables related to road accidents to predict casualty severity effectively.

The objective of this initiative is to build a robust predictive model for casualty severity, with the following goals:

- Predict Casualty Severity: Accurately predict the severity of casualties in road accidents based on various factors.
- Enhance Emergency Response: Provide emergency teams with valuable insights for optimal resource allocation and prioritization.
- Improve Health Outcomes: Tailor medical interventions to the severity of the injury, improving survival rates and recovery times.
- Data-Driven Policy Recommendations: Generate data-driven insights for policymakers to create better road safety policies.

In [1]:
```
import time
import numpy as np
import pandas as pd
#Library for Plotting
import seaborn as sns
sns.set(style="darkgrid")
import matplotlib.pyplot as plt
%matplotlib inline
# Suppressing warnings to keep the output clean and more readable
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
# We will monitor the time it takes to run the notebook
startnb = time.time()
```

2.Data Loading and Selection

2.1 Data Loading

In [3]:
```
trainset = pd.read_excel('trainset_inspection.xlsx')
testset = pd.read_excel('testset_inspection.xlsx')
```

In [4]:
```
trainset.head()
```

Out[4]:

	vehicle_type	towing_and_articulation	vehicle_manoeuvre	vehicle_location_restricted_lane	junction_location	skidding_and_overturning	hit_object_
0	Motorcycle 125cc and under	No tow/articulation	Going ahead other	On main c'way - not in restricted lane	Not at or within 20 metres of junction	Overturned	
1	Pedal cycle	unknown (self reported)	unknown (self reported)	unknown (self reported)	unknown (self reported)	unknown (self reported)	unkno
2	Car	No tow/articulation	unknown (self reported)	unknown (self reported)	unknown (self reported)	unknown (self reported)	unkno
3	Car	No tow/articulation	unknown (self reported)	unknown (self reported)	Not at or within 20 metres of junction	unknown (self reported)	unkno
4	Car	No tow/articulation	unknown (self reported)	unknown (self reported)	unknown (self reported)	unknown (self reported)	unkno

5 rows × 28 columns

In [5]:
```
trainset.info()
```
```
 10  vehicle_left_hand_drive          9257 non-null   object
 11  journey_purpose_of_driver        9257 non-null   object
 12  sex_of_driver                    9257 non-null   object
 13  age_band_of_driver               9257 non-null   object
 14  engine_capacity_cc               9257 non-null   float64
 15  propulsion_code                  9257 non-null   object
 16  age_of_vehicle                   9257 non-null   float64
 17  generic_make_model               9257 non-null   object
 18  driver_imd_decile                9257 non-null   object
 19  driver_home_area_type            9257 non-null   object
 20  casualty_class                   9257 non-null   object
 21  sex_of_casualty                  9257 non-null   object
 22  age_band_of_casualty             9257 non-null   object
 23  casualty_severity                9257 non-null   object
 24  pedestrian_location              9257 non-null   object
 25  pedestrian_movement              9257 non-null   object
 26  car_passenger                    9257 non-null   object
 27  casualty_home_area_type          9257 non-null   object
dtypes: float64(2), object(26)
memory usage: 2.0+ MB
```

In [6]:
```
trainset.shape
```
Out[6]:
```
(9257, 28)
```

2.2 Data Selection

In [7]:
```
from phik.report import plot_correlation_matrix
from scipy.stats import chi2_contingency
```

In [8]:
```python
# Define target variable and dataframe
target_variable = 'casualty_severity'
df = trainset

# Identify categorical columns (excluding target variable)
other_columns = df.select_dtypes(include=['object', 'category']).columns.tolist()
other_columns = [col for col in other_columns if col != target_variable]

# Compute Chi-Square test results
results = []

for column in other_columns:
    contingency_table = pd.crosstab(df[target_variable], df[column])
    chi2, p_value, dof, expected = chi2_contingency(contingency_table)
    results.append({
        'Column': column,
        'Chi-square Statistic': chi2,
        'P-value': f"{p_value:.2f}"
    })

results_df = pd.DataFrame(results)
sorted_results_df = results_df.sort_values('P-value').reset_index(drop=True)

# Compute Phi-K correlation matrix
phik_matrix = df.phik_matrix()

# Extract Phi-K values for the target variable
phik_target = phik_matrix[target_variable].drop(target_variable).reset_index()
phik_target.columns = ['Column', 'Phi-K Correlation']

# Merge Chi-Square results with Phi-K results
final_results = sorted_results_df.merge(phik_target, on='Column')
final_results
```

```
interval columns not set, guessing: ['engine_capacity_cc', 'age_of_vehicle']
```

Out[8]:

	Column	Chi-square Statistic	P-value	Phi-K Correlation
0	casualty_home_area_type	48.241579	0.00	0.160062
1	pedestrian_movement	1223.185911	0.00	0.321204
2	vehicle_location_restricted_lane	122.054637	0.00	0.127945
3	pedestrian_location	2385.717763	0.00	0.490737
4	age_band_of_casualty	3026.455370	0.00	0.569852
5	sex_of_casualty	154.502187	0.00	0.077430
6	casualty_class	535.575403	0.00	0.442660
7	car_passenger	94.151361	0.00	0.073284
8	junction_location	25.294465	0.12	0.033661
9	first_point_of_impact	14.882586	0.14	0.039558
10	vehicle_left_hand_drive	4.758345	0.31	0.021841
11	driver_home_area_type	2.988441	0.56	0.000000
12	journey_purpose_of_driver	6.473962	0.59	0.000000
13	towing_and_articulation	6.904867	0.73	0.000000
14	age_band_of_driver	13.128640	0.78	0.000000
15	sex_of_driver	1.593469	0.81	0.000000
16	skidding_and_overturning	6.117353	0.81	0.000000
17	hit_object_in_carriageway	11.093587	0.89	0.000000
18	vehicle_leaving_carriageway	9.231189	0.90	0.000000
19	driver_imd_decile	10.624333	0.91	0.000000
20	hit_object_off_carriageway	12.781867	0.94	0.000000
21	propulsion_code	3.985619	0.98	0.000000
22	vehicle_manoeuvre	17.724584	1.00	0.000000
23	generic_make_model	476.913199	1.00	0.000000
24	vehicle_type	14.304888	1.00	0.000000

Using correlation, the higher the Phi-K the more likely predictive power the variable has

In [9]:
```
# Select the most relevant 20 columns
trainset = trainset.loc[:,[ 'towing_and_articulation',
       'vehicle_location_restricted_lane', 'junction_location',
       'skidding_and_overturning', 'hit_object_in_carriageway',
       'vehicle_leaving_carriageway','first_point_of_impact', 'vehicle_left_hand_drive',
       'journey_purpose_of_driver', 'sex_of_driver', 'age_band_of_driver',
        'age_of_vehicle', 'driver_home_area_type',
       'casualty_class', 'sex_of_casualty', 'age_band_of_casualty',
       'casualty_severity', 'pedestrian_location', 'pedestrian_movement',
       'car_passenger', 'casualty_home_area_type']]
```

In [10]:
```
trainset.shape
```

Out[10]:
```
(9257, 21)
```

In [11]:
```
testset = testset.loc[:,[ 'towing_and_articulation',
       'vehicle_location_restricted_lane', 'junction_location',
       'skidding_and_overturning', 'hit_object_in_carriageway',
       'vehicle_leaving_carriageway','first_point_of_impact', 'vehicle_left_hand_drive',
       'journey_purpose_of_driver', 'sex_of_driver', 'age_band_of_driver',
        'age_of_vehicle', 'driver_home_area_type',
       'casualty_class', 'sex_of_casualty', 'age_band_of_casualty',
       'casualty_severity', 'pedestrian_location', 'pedestrian_movement',
       'car_passenger', 'casualty_home_area_type']]
```

2.3 Data Preparation

2.3.1 Feature Engineering

In [12]:
```
# Replace the value 'Fatal' with 'Serious' in the 'casualty_severity' column of the training dataset.
trainset['casualty_severity'] = trainset['casualty_severity'].replace('Fatal', 'Serious')

# Replace the value 'Fatal' with 'Serious' in the 'casualty_severity' column of the test dataset.
testset['casualty_severity'] = testset['casualty_severity'].replace('Fatal', 'Serious')
```

In [13]:
```
pd.DataFrame(trainset['casualty_severity'].value_counts())
```

Out[13]:

	casualty_severity
Slight	7630
Serious	1627

Note that slight is over 70% of the data, we will use SMOTE to deal with this imbalance

3. Model Building

In this part, I will create five models, train them on the training set and compare their results before picking the best ones to evaluate on the test set. I will use the following algorithms to handle the classification problem:

1. Decision Trees
2. Random Forest
3. AdaBoost
4. Support Vector Regression
5. Logistic Regression
6. KNN

This will show which model works best and could be interesting to see the outcome

In [14]:
```
# This selects the 'casualty_severity' column as y
ytrain = pd.DataFrame(trainset.loc[:, 'casualty_severity'])
# This excludes the 'casualty_severity' column for X
Xtrain = trainset.drop(columns=['casualty_severity'])
# This selects the 'casualty_severity' column as y
ytest = pd.DataFrame(testset.loc[:, 'casualty_severity'])
  # This excludes the 'casualty_severity' column for X
Xtest = testset.drop(columns=['casualty_severity'])
```

In [15]:
```
print(Xtrain.isnull().sum())
```

```
towing_and_articulation            0
vehicle_location_restricted_lane   0
junction_location                  0
skidding_and_overturning           0
hit_object_in_carriageway          0
vehicle_leaving_carriageway        0
first_point_of_impact              0
vehicle_left_hand_drive            0
journey_purpose_of_driver          0
sex_of_driver                      0
age_band_of_driver                 0
age_of_vehicle                     0
driver_home_area_type              0
casualty_class                     0
sex_of_casualty                    0
age_band_of_casualty               0
pedestrian_location                0
pedestrian_movement                0
car_passenger                      0
casualty_home_area_type            0
dtype: int64
```

Baseline

In [16]:
```
from sklearn.dummy import DummyClassifier
from sklearn.metrics import precision_recall_fscore_support

dummy_clf = DummyClassifier(strategy="most_frequent")
dummy_clf.fit(Xtrain, ytrain)
yhat = dummy_clf.predict(Xtrain)

p, r, f, s = precision_recall_fscore_support(ytrain, yhat, average="macro", zero_division=0.0)
print("Baseline:")
print(f"Precision: {p:.3f}")
print(f"Recall: {r:.3f}")
print(f"F score: {f:.3f}")
```

```
Baseline:
Precision: 0.412
Recall: 0.500
F score: 0.452
```

This model will serve as the baseline to beat.

In [17]:
```
# execution time
from timeit import default_timer as timer
from datetime import timedelta
```

In [18]:
```python
# create a folder where all trained models will be kept
import os

# Define the folder name
folder_name = "models"

# Check if the folder exists, and create it if it doesn't
if not os.path.exists(folder_name):
    os.makedirs(folder_name)
    print(f"Folder '{folder_name}' created successfully.")
else:
    print(f"Folder '{folder_name}' already exists.")
```

```
Folder 'models' already exists.
```

Decision Tree

In [20]:
```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
from time import time as timer
from datetime import timedelta

start = timer()

# Identify categorical and numerical columns
categorical_columns = Xtrain.select_dtypes(include=['object', 'category']).columns.tolist()
numerical_columns = Xtrain.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Use SimpleImputer to handle missing values in categorical columns
# Apply OneHotEncoder for categorical columns
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),  # Impute missing values
            ('encoder', OneHotEncoder(handle_unknown='ignore'))  # OneHotEncode categorical columns
        ]), categorical_columns),
        ('num', 'passthrough', numerical_columns)  # Leave numerical columns as they are
    ])

# Define the Decision Tree Classifier
dt = DecisionTreeClassifier(random_state=7)

# Define the hyperparameter grid
hp_grid = {
    'classifier__max_depth': [5, 10, 15, 20, 25, 30, 35, 40],
    'classifier__min_samples_split': [5, 10, 15, 20, 25, 30, 35],
}

# Create a pipeline with preprocessing, SMOTE, and the model
pipeline = ImbPipeline(steps=[
    ('preprocessor', preprocessor),
    ('smote', SMOTE(random_state=7)),  # Add SMOTE here since the data is imbalanced
```

```python
    ('classifier', dt)  # Add the classifier here
])

# Initialize GridSearchCV
grid_search = GridSearchCV(pipeline, hp_grid, cv=5,
                           scoring='f1_macro',
                           return_train_score=True, verbose=2)

# Fit the grid search
grid_search.fit(Xtrain, ytrain)

time_dt = timedelta(seconds=timer() - start)

# Print execution time
print("Execution time HH:MM:SS:", timedelta(seconds=timer() - start))
```

```
[CV] END classifier__max_depth=25, classifier__min_samples_split=35; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=5; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=5; total time=   0.6s
[CV] END classifier__max_depth=30, classifier__min_samples_split=5; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=5; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=5; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=10; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=10; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=10; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=10; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=10; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=15; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=15; total time=   0.5s

[CV] END classifier__max_depth=30, classifier__min_samples_split=15; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=15; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=15; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=20; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=20; total time=   0.5s
[CV] END classifier__max_depth=30, classifier__min_samples_split=20; total time=   0.5s
```

In [21]:
```
grid_search.best_estimator_
```

Out[21]:

```
Pipeline
  preprocessor: ColumnTransformer
    cat
      SimpleImputer
      SimpleImputer(strategy='most_frequent')
      OneHotEncoder
      OneHotEncoder(handle_unknown='ignore')
    num
      passthrough
      passthrough
  SMOTE
  SMOTE(random_state=7)
  DecisionTreeClassifier
  DecisionTreeClassifier(max_depth=5, min_samples_split=10, random_state=7)
```

In [22]:
```
cv_results = pd.DataFrame(grid_search.cv_results_)[['params', 'mean_train_score', 'mean_test_score']]
cv_results["diff, %"] = 100*(cv_results["mean_train_score"]-cv_results["mean_test_score"]
                                                    )/cv_results["mean_train_score"]

pd.set_option('display.max_colwidth', 100)
cv_results.sort_values('mean_test_score', ascending=False)
```

Out[22]:

	params	mean_train_score	mean_test_score	diff, %
1	{'classifier__max_depth': 5, 'classifier__min_samples_split': 10}	0.920444	0.919830	0.066688
6	{'classifier__max_depth': 5, 'classifier__min_samples_split': 35}	0.920444	0.919830	0.066688
2	{'classifier__max_depth': 5, 'classifier__min_samples_split': 15}	0.920444	0.919830	0.066688
3	{'classifier__max_depth': 5, 'classifier__min_samples_split': 20}	0.920444	0.919830	0.066688
4	{'classifier__max_depth': 5, 'classifier__min_samples_split': 25}	0.920444	0.919830	0.066688
5	{'classifier__max_depth': 5, 'classifier__min_samples_split': 30}	0.920444	0.919830	0.066688
0	{'classifier__max_depth': 5, 'classifier__min_samples_split': 5}	0.920547	0.919079	0.159561
13	{'classifier__max_depth': 10, 'classifier__min_samples_split': 35}	0.920894	0.905904	1.627817
12	{'classifier__max_depth': 10, 'classifier__min_samples_split': 30}	0.921670	0.905630	1.740345
9	{'classifier__max_depth': 10, 'classifier__min_samples_split': 15}	0.923438	0.904717	2.027295
11	{'classifier__max_depth': 10, 'classifier__min_samples_split': 25}	0.922198	0.904496	1.919599

The best model has: max depth as 5 and the minimum sample split of 10. We see that their is a small margin between the trainset and the validation set which means the model does a good job in prediction.

In [23]:
```
grid_search.best_score_
```

Out[23]:
```
0.9198296739509935
```

In [24]:
```python
from joblib import dump

# Save the best estimator from GridSearchCV to a file
dump(grid_search.best_estimator_, 'models/dt.joblib')
print("Model saved successfully.")
```

```
Model saved successfully.
```

Random Forest

In [25]:

```
start = timer()

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

# Identify categorical and numerical columns
categorical_columns = Xtrain.select_dtypes(include=['object', 'category']).columns.tolist()
numerical_columns = Xtrain.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Use SimpleImputer to handle missing values in categorical columns
# Apply OneHotEncoder for categorical columns
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),  # Impute missing values
            ('encoder', OneHotEncoder(handle_unknown='ignore'))  # OneHotEncode categorical columns
        ]), categorical_columns),
        ('num', 'passthrough', numerical_columns)  # Leave numerical columns as they are
    ])

# Define the Random Forest Classifier
rf = RandomForestClassifier(random_state=7)

# Define the hyperparameter grid for Random Forest
hp_grid = {
    'classifier__n_estimators': [50, 100, 200],  # Number of trees in the forest
    'classifier__max_depth': [5, 10, 15, 20, 25, 30, None],  # Maximum depth of the tree
    'classifier__min_samples_split': [2, 5, 10],  # Minimum number of samples required to split a node
    'classifier__min_samples_leaf': [1, 2, 4],  # Minimum number of samples required at each leaf node
}

# Create a pipeline with preprocessing and the model
pipeline = ImbPipeline(steps=[
    ('preprocessor', preprocessor),
    ('smote', SMOTE(random_state=7)),
    ('classifier', rf)
])
```

```python
# Initialize GridSearchCV
grid_search = GridSearchCV(pipeline, hp_grid, cv=5,
                           scoring='f1_macro',
                           return_train_score=True, verbose=2)

# Fit the grid search
grid_search.fit(Xtrain, ytrain)

time_rf = timedelta(seconds=timer() - start)

print("Execution time HH:MM:SS:", timedelta(seconds=timer() - start))
```

```
[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=100; total time=   3.8s
[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=100; total time=   3.6s
[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=100; total time=   3.5s
[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=100; total time=   3.6s

[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=200; total time=   7.2s
[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=200; total time=   7.2s
[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=200; total time=   7.2s
[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=200; total time=   7.0s
[CV] END classifier__max_depth=None, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__
n_estimators=200; total time=   7.2s
Execution time HH:MM:SS: 1:01:39.080936
```

In [26]:
```
#Get the best estimator
grid_search.best_estimator_
```

Out[26]:

```
Pipeline
preprocessor: ColumnTransformer
cat                     num
SimpleImputer           passthrough
OneHotEncoder
SMOTE
RandomForestClassifier
RandomForestClassifier(max_depth=30, min_samples_leaf=2, min_samples_split=5,
                       n_estimators=200, random_state=7)
```

In [27]:
```python
# Convert the cross-validation results from the grid search into a pandas DataFrame.
# We select three columns: 'params' (hyperparameter combinations), 'mean_train_score' (average training score),
# and 'mean_test_score' (average test score for each combination of hyperparameters).
cv_results = pd.DataFrame(grid_search.cv_results_)[['params', 'mean_train_score', 'mean_test_score']]

# Calculate the percentage difference between the training and test scores for each hyperparameter combination.
# The formula is: (mean_train_score - mean_test_score) / mean_train_score * 100
# This difference indicates how much overfitting or underfitting occurs. A higher percentage may indicate overfitting.
cv_results["diff, %"] = 100 * (cv_results["mean_train_score"] - cv_results["mean_test_score"]) / cv_results["mean_trai

# Set the maximum column width to 100 characters for better readability, particularly for the 'params' column that migh
pd.set_option('display.max_colwidth', 100)

# Sort the DataFrame by the mean test score in descending order. This ensures the best performing models (highest test
cv_results.sort_values('mean_test_score', ascending=False)
```

Out[27]:

	params	mean_train_score	mean_test_score	diff, %
149	{'classifier__max_depth': 30, 'classifier__min_samples_leaf': 2, 'classifier__min_samples_split'...	0.935019	0.922762	1.310873
35	{'classifier__max_depth': 10, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split'...	0.922900	0.922756	0.015585
32	{'classifier__max_depth': 10, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split'...	0.923307	0.922584	0.078275
29	{'classifier__max_depth': 10, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split'...	0.923719	0.922584	0.122880
6	{'classifier__max_depth': 5, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split':...	0.922413	0.922549	-0.014737
...	...	...	...	...
137	{'classifier__max_depth': 30, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split'...	0.992591	0.915609	7.755644
135	{'classifier__max_depth': 30, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split'...	0.991328	0.915182	7.681222
164	{'classifier__max_depth': None, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_spli...	0.993191	0.915174	7.855177
163	{'classifier__max_depth': None, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_spli...	0.993056	0.915151	7.844996
162	{'classifier__max_depth': None, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_spli...	0.993194	0.914563	7.916930

189 rows × 4 columns

Under Random Forest, the best model has:of max_depth=30, min_samples_leaf=2, n_estimators=200. We see that their is a small margin between the trainset and the validation set which means the model does a good job in prediction

In [28]:
```
dump(grid_search.best_estimator_, 'models/rf.joblib')
print("Model saved successfully.")
```

```
Model saved successfully.
```

ADA Boost

In [29]:
```python
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import GridSearchCV

start = timer()

# Identify categorical and numerical columns
categorical_columns = Xtrain.select_dtypes(include=['object', 'category']).columns.tolist()
numerical_columns = Xtrain.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Use SimpleImputer to handle missing values in categorical columns
# Apply OneHotEncoder for categorical columns
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),  # Impute missing values
            ('encoder', OneHotEncoder(handle_unknown='ignore'))  # OneHotEncode categorical columns
        ]), categorical_columns),
        ('num', 'passthrough', numerical_columns)  # Leave numerical columns as they are
    ])

# Define the AdaBoost Classifier
ada = AdaBoostClassifier(random_state=7)

# Define the hyperparameter grid for AdaBoost
hp_grid = {
    'classifier__n_estimators': [50, 100, 200],  # Number of weak learners
    'classifier__learning_rate': [0.01, 0.1, 1.0],  # Learning rate
}

# Create a pipeline with preprocessing and the model
pipeline = ImbPipeline(steps=[
    ('preprocessor', preprocessor),
    ('smote', SMOTE(random_state=7)),
    ('classifier', ada)
])

# Initialize GridSearchCV
```

```
grid_search = GridSearchCV(pipeline, hp_grid, cv=5,
                           scoring='f1_macro',
                           return_train_score=True, verbose=2)

# Fit the grid search
grid_search.fit(Xtrain, ytrain)

time_ada = timedelta(seconds=timer() - start)

print("Execution time HH:MM:SS:", timedelta(seconds=timer() - start))
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=50; total time=   1.7s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=50; total time=   1.7s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=50; total time=   1.6s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=50; total time=   1.6s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=50; total time=   1.6s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=100; total time=   3.0s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=100; total time=   3.0s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=100; total time=   3.3s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=100; total time=   3.2s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=100; total time=   3.2s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=200; total time=   6.1s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=200; total time=   5.8s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=200; total time=   5.6s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=200; total time=   6.0s
[CV] END classifier__learning_rate=0.01, classifier__n_estimators=200; total time=   6.2s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=50; total time=   1.7s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=50; total time=   1.7s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=50; total time=   1.7s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=50; total time=   1.8s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=50; total time=   1.8s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=100; total time=   3.1s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=100; total time=   3.0s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=100; total time=   3.0s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=100; total time=   3.1s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=100; total time=   3.1s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=200; total time=   6.0s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=200; total time=   5.8s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=200; total time=   5.7s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=200; total time=   5.5s
[CV] END classifier__learning_rate=0.1, classifier__n_estimators=200; total time=   6.0s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=50; total time=   1.6s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=50; total time=   1.6s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=50; total time=   1.6s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=50; total time=   1.6s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=50; total time=   1.6s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=100; total time=   2.9s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=100; total time=   3.1s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=100; total time=   2.9s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=100; total time=   2.9s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=100; total time=   2.9s
```

```
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=200; total time=   5.8s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=200; total time=   5.6s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=200; total time=   6.1s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=200; total time=   6.2s
[CV] END classifier__learning_rate=1.0, classifier__n_estimators=200; total time=   6.2s
Execution time HH:MM:SS: 0:03:02.494196
```

In [30]:

```
#Get the best parameter
grid_search.best_estimator_
```

Out[30]:

Pipeline

preprocessor: ColumnTransformer

cat

SimpleImputer

OneHotEncoder

num

passthrough

SMOTE

AdaBoostClassifier

AdaBoostClassifier(learning_rate=0.1, n_estimators=200, random_state=7)

In [31]:
```
# Convert the cross-validation results from the grid search into a pandas DataFrame.
# We select only the relevant columns: 'params' (hyperparameter settings), 'mean_train_score' (average score on the tra
# and 'mean_test_score' (average score on the test set for each combination of parameters).
cv_results = pd.DataFrame(grid_search.cv_results_)[['params', 'mean_train_score', 'mean_test_score']]

# Calculate the difference between the training and test scores as a percentage.
# The formula is: (mean_train_score - mean_test_score) / mean_train_score * 100
# This percentage difference helps to identify overfitting or underfitting.
cv_results["diff, %"] = 100 * (cv_results["mean_train_score"] - cv_results["mean_test_score"]) / cv_results["mean_trai

# Set the maximum column width to 100 for displaying the DataFrame, to prevent truncating long values in the 'params' c
pd.set_option('display.max_colwidth', 100)

# Sort the results by the mean test score in descending order, to view the best-performing hyperparameter combinations
cv_results.sort_values('mean_test_score', ascending=False)
```

Out[31]:

	params	mean_train_score	mean_test_score	diff, %
5	{'classifier__learning_rate': 0.1, 'classifier__n_estimators': 200}	0.915328	0.915804	-0.051998
6	{'classifier__learning_rate': 1.0, 'classifier__n_estimators': 50}	0.915067	0.915743	-0.073786
4	{'classifier__learning_rate': 0.1, 'classifier__n_estimators': 100}	0.915579	0.915621	-0.004600
7	{'classifier__learning_rate': 1.0, 'classifier__n_estimators': 100}	0.915787	0.914701	0.118505
8	{'classifier__learning_rate': 1.0, 'classifier__n_estimators': 200}	0.915507	0.911641	0.422259
3	{'classifier__learning_rate': 0.1, 'classifier__n_estimators': 50}	0.897676	0.898604	-0.103373
2	{'classifier__learning_rate': 0.01, 'classifier__n_estimators': 200}	0.842310	0.838816	0.414898
1	{'classifier__learning_rate': 0.01, 'classifier__n_estimators': 100}	0.784866	0.776564	1.057801
0	{'classifier__learning_rate': 0.01, 'classifier__n_estimators': 50}	0.761236	0.761080	0.020487

Under ADABoost, the best model has:learning_rate=1, n_estimators=200. We see that their is an even smaller margin between the trainset and the validation set which means the model does a good job in prediction better than the two models above.

In [32]:
```
dump(grid_search.best_estimator_, 'models/ada_boost.joblib')
print("Model saved successfully.")
```

```
Model saved successfully.
```

Linear SVM

In [33]:
```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV

start = timer()

# Identify categorical and numerical columns
categorical_columns = Xtrain.select_dtypes(include=['object', 'category']).columns.tolist()
numerical_columns = Xtrain.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Preprocessing for categorical and numerical columns
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),  # Impute missing values
            ('encoder', OneHotEncoder(handle_unknown='ignore'))  # OneHotEncode categorical columns
        ]), categorical_columns),
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),  # Impute missing values with mean
            ('scaler', StandardScaler())  # Scale numerical columns
        ]), numerical_columns)
    ])

# Define the Linear SVM Classifier
svm = LinearSVC(random_state=7, max_iter=10000)  # Increase max_iter for convergence

# Define the hyperparameter grid for Linear SVM
hp_grid = {
    'classifier__C': [0.01, 0.1, 1, 10, 100],  # Regularization parameter
    'classifier__penalty': ['l1', 'l2'],  # Regularization type
    'classifier__loss': ['hinge', 'squared_hinge']  # Loss function
}

# Create a pipeline with preprocessing and the model
pipeline = ImbPipeline(steps=[
    ('preprocessor', preprocessor),
    ('smote', SMOTE(random_state=7)),
    ('classifier', svm)
```

```
])

# Initialize GridSearchCV
grid_search = GridSearchCV(pipeline, hp_grid, cv=5,
                           scoring='f1_macro',
                           return_train_score=True, verbose=2)

# Fit the grid search
grid_search.fit(Xtrain, ytrain)

time_svm = timedelta(seconds=timer() - start)
print("Execution time HH:MM:SS:", timedelta(seconds=timer() - start))
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l2; total time=   0.3s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l2; total time=   0.5s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l2; total time=   0.4s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l2; total time=   0.5s
[CV] END classifier__C=0.01, classifier__loss=hinge, classifier__penalty=l2; total time=   0.5s
[CV] END classifier__C=0.01, classifier__loss=squared_hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=squared_hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=squared_hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=squared_hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=squared_hinge, classifier__penalty=l1; total time=   0.2s
[CV] END classifier__C=0.01, classifier__loss=squared_hinge, classifier__penalty=l2; total time=   0.3s
[CV] END classifier__C=0.01, classifier__loss=squared_hinge, classifier__penalty=l2; total time=   0.3s
[CV] END classifier__C=0.01, classifier__loss=squared_hinge, classifier__penalty=l2; total time=   0.2s
```

In [34]:

```
grid_search.best_estimator_
```

Out[34]:

Pipeline

preprocessor: ColumnTransformer

cat	**num**
SimpleImputer	SimpleImputer
OneHotEncoder	StandardScaler

SMOTE

LinearSVC

```
LinearSVC(C=0.1, loss='hinge', max_iter=10000, random_state=7)
```

In [35]:

```python
cv_results = pd.DataFrame(grid_search.cv_results_)[['params', 'mean_train_score', 'mean_test_score']]
cv_results["diff, %"] = 100*(cv_results["mean_train_score"]-cv_results["mean_test_score"]
                                                     )/cv_results["mean_train_score"]

pd.set_option('display.max_colwidth', 100)
cv_results.sort_values('mean_test_score', ascending=False)
```

Out[35]:

	params	mean_train_score	mean_test_score	diff, %
5	{'classifier__C': 0.1, 'classifier__loss': 'hinge', 'classifier__penalty': 'l2'}	0.914081	0.913800	0.030750
1	{'classifier__C': 0.01, 'classifier__loss': 'hinge', 'classifier__penalty': 'l2'}	0.913622	0.913656	-0.003711
3	{'classifier__C': 0.01, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l2'}	0.914112	0.913462	0.071137
7	{'classifier__C': 0.1, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l2'}	0.912897	0.911592	0.142941
13	{'classifier__C': 10, 'classifier__loss': 'hinge', 'classifier__penalty': 'l2'}	0.914888	0.911247	0.397888
17	{'classifier__C': 100, 'classifier__loss': 'hinge', 'classifier__penalty': 'l2'}	0.914825	0.910932	0.425561
9	{'classifier__C': 1, 'classifier__loss': 'hinge', 'classifier__penalty': 'l2'}	0.914752	0.910919	0.418952
11	{'classifier__C': 1, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l2'}	0.911864	0.909648	0.242984
19	{'classifier__C': 100, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l2'}	0.912591	0.909106	0.381898
15	{'classifier__C': 10, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l2'}	0.911752	0.908956	0.306624
0	{'classifier__C': 0.01, 'classifier__loss': 'hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
2	{'classifier__C': 0.01, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
4	{'classifier__C': 0.1, 'classifier__loss': 'hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
6	{'classifier__C': 0.1, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
8	{'classifier__C': 1, 'classifier__loss': 'hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
10	{'classifier__C': 1, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
12	{'classifier__C': 10, 'classifier__loss': 'hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
14	{'classifier__C': 10, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
16	{'classifier__C': 100, 'classifier__loss': 'hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN
18	{'classifier__C': 100, 'classifier__loss': 'squared_hinge', 'classifier__penalty': 'l1'}	NaN	NaN	NaN

Under SVM, the best model has:C=0.1 (smaller decision boundary between classes), max_iter=10000. We see that there is an even small margin between the trainset and the validation set which means the model does a good job in prediction.

In [36]:
```
dump(grid_search.best_estimator_, 'models/lin-svm.joblib')
print("Model saved successfully.")
```

```
Model saved successfully.
```

Logistic Regression

In [37]:

```python
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

start = timer()

# Identify categorical and numerical columns
categorical_columns = Xtrain.select_dtypes(include=['object', 'category']).columns.tolist()
numerical_columns = Xtrain.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Use SimpleImputer to handle missing values in categorical columns
# Apply OneHotEncoder for categorical columns
# Scale numerical columns using StandardScaler
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),  # Impute missing values
            ('encoder', OneHotEncoder(handle_unknown='ignore'))  # OneHotEncode categorical columns
        ]), categorical_columns),
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),  # Impute missing values in numerical columns
            ('scaler', StandardScaler())  # Scale numerical columns
        ]), numerical_columns)
    ])

# Define the Logistic Regression model
log_reg = LogisticRegression(random_state=7, max_iter=1000)  # Increase max_iter for convergence

# Define the hyperparameter grid for Logistic Regression
hp_grid = {
    'classifier__C': [0.01, 0.1, 1, 10, 100],  # Regularization strength
    'classifier__penalty': ['l2'],  # Regularization type (L2 is default for LogisticRegression)
    'classifier__solver': ['lbfgs', 'liblinear']  # Solvers compatible with L2 penalty
}

# Create a pipeline with preprocessing and the model
pipeline = ImbPipeline(steps=[
    ('preprocessor', preprocessor),
```

```
    ('smote', SMOTE(random_state=7)),
    ('classifier', log_reg)
])

# Initialize GridSearchCV
grid_search = GridSearchCV(pipeline, hp_grid, cv=5,
                           scoring='f1_macro',
                           return_train_score=True, verbose=2)

# Fit the grid search
grid_search.fit(Xtrain, ytrain)

time_log_reg = timedelta(seconds=timer() - start)

print("Execution time HH:MM:SS:", timedelta(seconds=timer() - start))
```

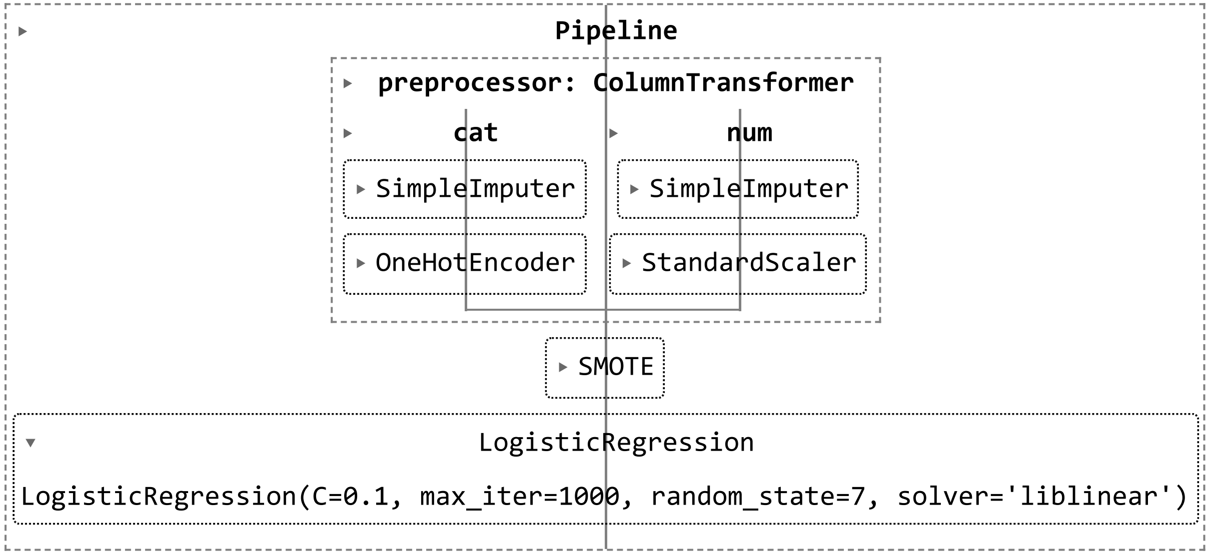
```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=0.01, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.2s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.4s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.4s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.4s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.4s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.4s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=0.1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.5s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.5s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.5s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.6s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=lbfgs; total time=   0.5s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.4s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.3s
[CV] END classifier__C=1, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.4s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.0s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.0s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.0s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.1s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.1s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.4s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.4s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.4s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.4s
[CV] END classifier__C=10, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.4s
```

```
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.4s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.1s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.4s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.4s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=lbfgs; total time=   1.1s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.5s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.5s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.5s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.5s
[CV] END classifier__C=100, classifier__penalty=l2, classifier__solver=liblinear; total time=   0.6s
Execution time HH:MM:SS: 0:00:36.991596
```

In [38]:

```
#Get the best parameters
grid_search.best_estimator_
```

Out[38]:



In [39]:
```
cv_results = pd.DataFrame(grid_search.cv_results_)[['params', 'mean_train_score', 'mean_test_score']]
cv_results["diff, %"] = 100*(cv_results["mean_train_score"]-cv_results["mean_test_score"]
                                                     )/cv_results["mean_train_score"]

pd.set_option('display.max_colwidth', 100)
cv_results.sort_values('mean_test_score', ascending=False)
```

Out[39]:

	params	mean_train_score	mean_test_score	diff, %
3	{'classifier__C': 0.1, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear'}	0.909470	0.906061	0.374861
2	{'classifier__C': 0.1, 'classifier__penalty': 'l2', 'classifier__solver': 'lbfgs'}	0.908997	0.905771	0.354832
4	{'classifier__C': 1, 'classifier__penalty': 'l2', 'classifier__solver': 'lbfgs'}	0.909665	0.905538	0.453738
5	{'classifier__C': 1, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear'}	0.909756	0.905538	0.463667
6	{'classifier__C': 10, 'classifier__penalty': 'l2', 'classifier__solver': 'lbfgs'}	0.908635	0.902707	0.652490
7	{'classifier__C': 10, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear'}	0.908555	0.902544	0.661635
9	{'classifier__C': 100, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear'}	0.908091	0.901585	0.716344
8	{'classifier__C': 100, 'classifier__penalty': 'l2', 'classifier__solver': 'lbfgs'}	0.908130	0.901169	0.766595
0	{'classifier__C': 0.01, 'classifier__penalty': 'l2', 'classifier__solver': 'lbfgs'}	0.875552	0.871376	0.477001
1	{'classifier__C': 0.01, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear'}	0.871165	0.865663	0.631520

Under Logistic Regression, the best model has:C=0.1 (the regularization parameter to limit over fitting and underfitting), max_iterations=1000,. We see that their is a small margin between the trainset and the validation set which means the model does a good job in prediction.

In [40]:
```
dump(grid_search.best_estimator_, 'models/log_reg.joblib')
print("Model saved successfully.")
```

```
Model saved successfully.
```

KNN

In [41]:
```python
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

start = timer()

# Identify categorical and numerical columns
categorical_columns = Xtrain.select_dtypes(include=['object', 'category']).columns.tolist()
numerical_columns = Xtrain.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Use SimpleImputer to handle missing values in categorical columns
# Apply OneHotEncoder for categorical columns
# Standardize numerical columns
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),  # Impute missing values
            ('encoder', OneHotEncoder(handle_unknown='ignore'))  # OneHotEncode categorical columns
        ]), categorical_columns),
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),  # Impute missing values with mean
            ('scaler', StandardScaler())  # Standardize numerical columns
        ]), numerical_columns)
    ])

# Define the K-Nearest Neighbors Classifier
knn = KNeighborsClassifier()

# Define the hyperparameter grid for KNN
hp_grid = {
    'classifier__n_neighbors': [3, 5, 7, 9, 11],  # Number of neighbors
    'classifier__weights': ['uniform', 'distance'],  # Weighting scheme
    'classifier__p': [1, 2]  # 1 for Manhattan distance, 2 for Euclidean distance
}

# Create a pipeline with preprocessing and the model
pipeline = ImbPipeline(steps=[
    ('preprocessor', preprocessor),
```

```
    ('smote', SMOTE(random_state=7)),
    ('classifier', knn)
])

# Initialize GridSearchCV
grid_search = GridSearchCV(pipeline, hp_grid, cv=5,
                           scoring='f1_macro',
                           return_train_score=True, verbose=2)

# Fit the grid search
grid_search.fit(Xtrain, ytrain)

time_knn = timedelta(seconds=timer() - start)

print("Execution time HH:MM:SS:", timedelta(seconds=timer() - start))
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=distance; total time=   0.9s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=distance; total time=   0.7s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=distance; total time=   0.7s
[CV] END classifier__n_neighbors=3, classifier__p=1, classifier__weights=distance; total time=   0.9s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=uniform; total time=   1.8s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=uniform; total time=   1.7s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=uniform; total time=   1.8s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=uniform; total time=   1.7s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=uniform; total time=   1.8s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=3, classifier__p=2, classifier__weights=distance; total time=   1.8s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=distance; total time=   0.7s
[CV] END classifier__n_neighbors=5, classifier__p=1, classifier__weights=distance; total time=   0.7s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=uniform; total time=   1.9s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=uniform; total time=   1.8s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=uniform; total time=   1.9s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=uniform; total time=   1.9s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=uniform; total time=   1.9s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=distance; total time=   1.8s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=distance; total time=   1.9s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=distance; total time=   2.0s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=distance; total time=   1.8s
[CV] END classifier__n_neighbors=5, classifier__p=2, classifier__weights=distance; total time=   1.7s
```

```
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=distance; total time=   0.7s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=distance; total time=   0.9s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=distance; total time=   0.9s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=7, classifier__p=1, classifier__weights=distance; total time=   0.9s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=uniform; total time=   2.0s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=uniform; total time=   1.9s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=uniform; total time=   1.9s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=uniform; total time=   1.8s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=uniform; total time=   1.9s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=distance; total time=   1.8s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=distance; total time=   1.8s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=distance; total time=   1.9s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=distance; total time=   1.8s
[CV] END classifier__n_neighbors=7, classifier__p=2, classifier__weights=distance; total time=   1.8s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=9, classifier__p=1, classifier__weights=distance; total time=   0.7s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=uniform; total time=   1.7s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=uniform; total time=   1.8s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=uniform; total time=   1.7s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=uniform; total time=   1.9s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=uniform; total time=   1.7s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=distance; total time=   1.8s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=distance; total time=   1.6s
[CV] END classifier__n_neighbors=9, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=uniform; total time=   0.9s
```

```
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=uniform; total time=   0.9s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=uniform; total time=   0.8s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=distance; total time=   0.7s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=distance; total time=   0.9s
[CV] END classifier__n_neighbors=11, classifier__p=1, classifier__weights=distance; total time=   0.8s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=uniform; total time=   1.7s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=uniform; total time=   1.8s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=uniform; total time=   1.7s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=uniform; total time=   2.0s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=uniform; total time=   1.7s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=distance; total time=   1.6s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=distance; total time=   1.7s
[CV] END classifier__n_neighbors=11, classifier__p=2, classifier__weights=distance; total time=   1.7s
Execution time HH:MM:SS: 0:09:10.268210
```

In [42]:

```
grid_search.best_estimator_
```

Out[42]:

```
Pipeline
preprocessor: ColumnTransformer
cat                     num
SimpleImputer           SimpleImputer
OneHotEncoder           StandardScaler
SMOTE
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=7, p=1, weights='distance')
```

In [43]:

```
# Convert the cross-validation results from the grid search into a DataFrame, selecting relevant columns
# The 'params' column contains the hyperparameters, 'mean_train_score' is the average score on the training set,
# and 'mean_test_score' is the average score on the test set for each combination of parameters
cv_results = pd.DataFrame(grid_search.cv_results_)[['params', 'mean_train_score', 'mean_test_score']]

# Calculate the difference between the training and test scores as a percentage.
# The formula for this is: (mean_train_score - mean_test_score) / mean_train_score * 100
cv_results["diff, %"] = 100 * (cv_results["mean_train_score"] - cv_results["mean_test_score"]) / cv_results["mean_trai

# Set the maximum column width for displaying in the DataFrame, making the columns easier to read
pd.set_option('display.max_colwidth', 100)

# Sort the results by the mean test score in descending order, so that the best-performing models appear first
cv_results.sort_values('mean_test_score', ascending=False)
```

Out[43]:

	params	mean_train_score	mean_test_score	diff, %
9	{'classifier__n_neighbors': 7, 'classifier__p': 1, 'classifier__weights': 'distance'}	0.993627	0.854847	13.966983
13	{'classifier__n_neighbors': 9, 'classifier__p': 1, 'classifier__weights': 'distance'}	0.993627	0.851241	14.329891
8	{'classifier__n_neighbors': 7, 'classifier__p': 1, 'classifier__weights': 'uniform'}	0.881850	0.850845	3.515902
5	{'classifier__n_neighbors': 5, 'classifier__p': 1, 'classifier__weights': 'distance'}	0.993624	0.849301	14.524939
17	{'classifier__n_neighbors': 11, 'classifier__p': 1, 'classifier__weights': 'distance'}	0.993627	0.848348	14.621112
4	{'classifier__n_neighbors': 5, 'classifier__p': 1, 'classifier__weights': 'uniform'}	0.893786	0.848169	5.103811
0	{'classifier__n_neighbors': 3, 'classifier__p': 1, 'classifier__weights': 'uniform'}	0.909704	0.843220	7.308264
12	{'classifier__n_neighbors': 9, 'classifier__p': 1, 'classifier__weights': 'uniform'}	0.874415	0.840712	3.854403
1	{'classifier__n_neighbors': 3, 'classifier__p': 1, 'classifier__weights': 'distance'}	0.993624	0.839038	15.557820
16	{'classifier__n_neighbors': 11, 'classifier__p': 1, 'classifier__weights': 'uniform'}	0.865243	0.835524	3.434767
19	{'classifier__n_neighbors': 11, 'classifier__p': 2, 'classifier__weights': 'distance'}	0.993358	0.795662	19.901766
11	{'classifier__n_neighbors': 7, 'classifier__p': 2, 'classifier__weights': 'distance'}	0.993538	0.793387	20.145255
15	{'classifier__n_neighbors': 9, 'classifier__p': 2, 'classifier__weights': 'distance'}	0.993494	0.791384	20.343359
7	{'classifier__n_neighbors': 5, 'classifier__p': 2, 'classifier__weights': 'distance'}	0.993624	0.789077	20.585962
3	{'classifier__n_neighbors': 3, 'classifier__p': 2, 'classifier__weights': 'distance'}	0.993624	0.788783	20.615550
2	{'classifier__n_neighbors': 3, 'classifier__p': 2, 'classifier__weights': 'uniform'}	0.897824	0.775968	13.572300
10	{'classifier__n_neighbors': 7, 'classifier__p': 2, 'classifier__weights': 'uniform'}	0.840355	0.771420	8.203127
6	{'classifier__n_neighbors': 5, 'classifier__p': 2, 'classifier__weights': 'uniform'}	0.861444	0.769577	10.664346
18	{'classifier__n_neighbors': 11, 'classifier__p': 2, 'classifier__weights': 'uniform'}	0.815991	0.768790	5.784524
14	{'classifier__n_neighbors': 9, 'classifier__p': 2, 'classifier__weights': 'uniform'}	0.824925	0.766054	7.136496

Under KNN, the best model has:nearest_neighbors=7, p=1(using Manhattan distance),. We see that their is a larger margin between the trainset and the validation set which means the model does a standard job in prediction.

In [44]:
```
#Add the model to the library
dump(grid_search.best_estimator_, 'models/KNN.joblib')
print("Model saved successfully.")
```

```
Model saved successfully.
```

4. Model Evaluation

4.1 Time taken

In [45]:
```
print("Elapsed time: Decision Tree", time_dt)
print("Elapsed time: Random Forest", time_rf)
print("Elapsed time: AdaBoost", time_ada)
print("Elapsed time: SVM ", time_svm)
print("Elapsed time: Logistic Regression", time_log_reg)
print("Elapsed time: KNN", time_knn)
```

```
Elapsed time: Decision Tree 0:03:03.755953
Elapsed time: Random Forest 1:01:39.080936
Elapsed time: AdaBoost 0:03:02.494196
Elapsed time: SVM  0:03:20.892258
Elapsed time: Logistic Regression 0:00:36.991596
Elapsed time: KNN 0:09:10.268210
```

The logistic regression is the fastest model to run and the Random Forest took the longest

4.2 Model Performance

In [46]:
```python
from joblib import load

best_dt= load("models/dt.joblib")
best_rf = load("models/rf.joblib")
best_ab = load("models/ada_boost.joblib")
best_svm = load("models/lin-svm.joblib")
best_log_reg = load("models/log_reg.joblib")
best_knn = load("models/KNN.joblib")
```

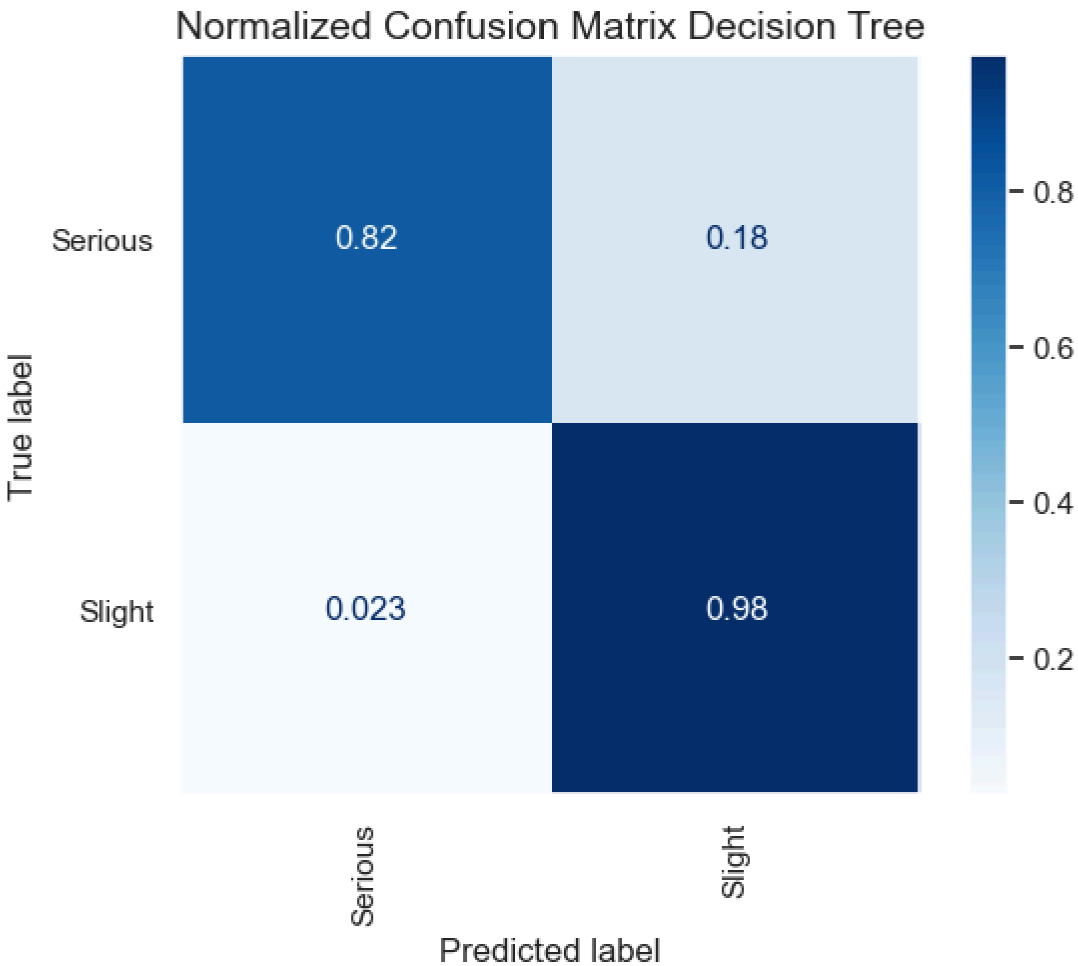
Decision Tree

In [47]:
```python
# Predict the model
yhat = best_dt.predict(Xtest)

# micro-averaged precision, recall and f-score
p, r, f, s = precision_recall_fscore_support(ytest, yhat, average="macro")
print("Decision Tree:")
print(f"Precision: {p}")
print(f"Recall: {r}")
print(f"F score: {f}")
```

```
Decision Tree:
Precision: 0.919865538928649
Recall: 0.8996146965735516
F score: 0.909329620752855
```

In [48]:

```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# generating the confusion matrix display
ConfusionMatrixDisplay.from_predictions(ytest, yhat, labels=best_dt.classes_,
                                        xticks_rotation="vertical", normalize="true",
                                        cmap=plt.cm.Blues)

# Accessing current axes and turning off grid lines
ax = plt.gca()  # Get current axes
ax.grid(False)  # Turn off grid
plt.title('Normalized Confusion Matrix Decision Tree', fontsize=14)
plt.show()  # Display the plot
```



Normalized Confusion Matrix Decision Tree
True label
Serious
Slight
0.82
0.18
0.023
0.98
Serious
Slight
Predicted label
0.8
0.6
0.4
0.2

In [49]:

```
from sklearn.model_selection import learning_curve

# Compute the learning curve
train_sizes, train_scores, val_scores = learning_curve(
    best_dt, Xtrain, ytrain, cv=5, scoring='accuracy',
    train_sizes=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

# Calculate the mean and standard deviation for plotting
train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)
train_std = train_scores.std(axis=1)
val_std = val_scores.std(axis=1)

# Plotting the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_mean, label='Training Accuracy', color='blue')
plt.plot(train_sizes, val_mean, label='Validation Accuracy', color='green')

# Adding shading for standard deviation
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='green')

plt.title('Learning Curve', fontsize=14)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```

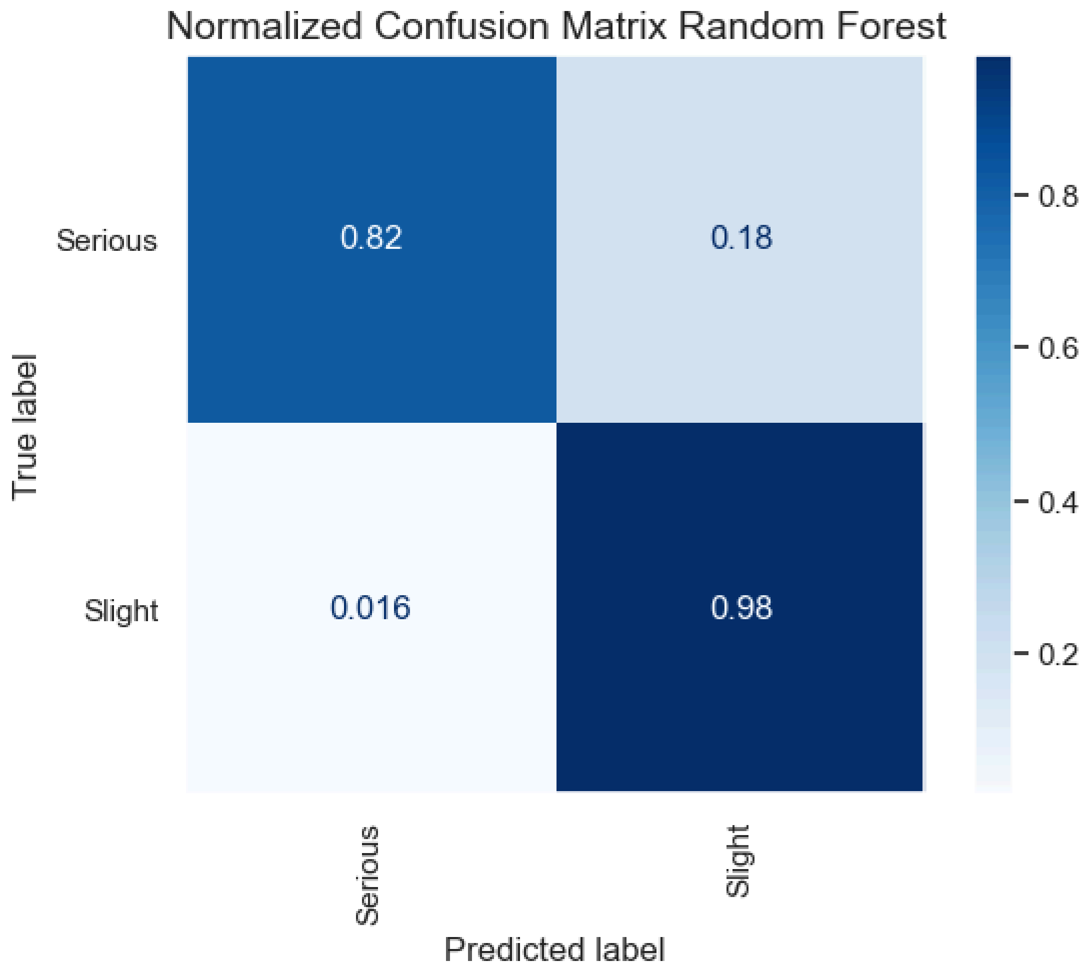
- At 1000 Training Set Size: The training accuracy might be high, but the validation accuracy is lower, indicating potential overfitting.
- At 7000 Training Set Size: Both training and validation accuracies might be closer, suggesting better generalization.

Random Forest

In [50]:
```python
#Predict the model
yhat = best_rf.predict(Xtest)

# micro-averaged precision, recall and f-score
p, r, f, s = precision_recall_fscore_support(ytest, yhat, average="macro")
print("Random Forest:")
print(f"Precision: {p}")
print(f"Recall: {r}")
print(f"F score: {f}")
```

```
Random Forest:
Precision: 0.937125078341839
Recall: 0.9031030686665749
F score: 0.9190042075736327
```

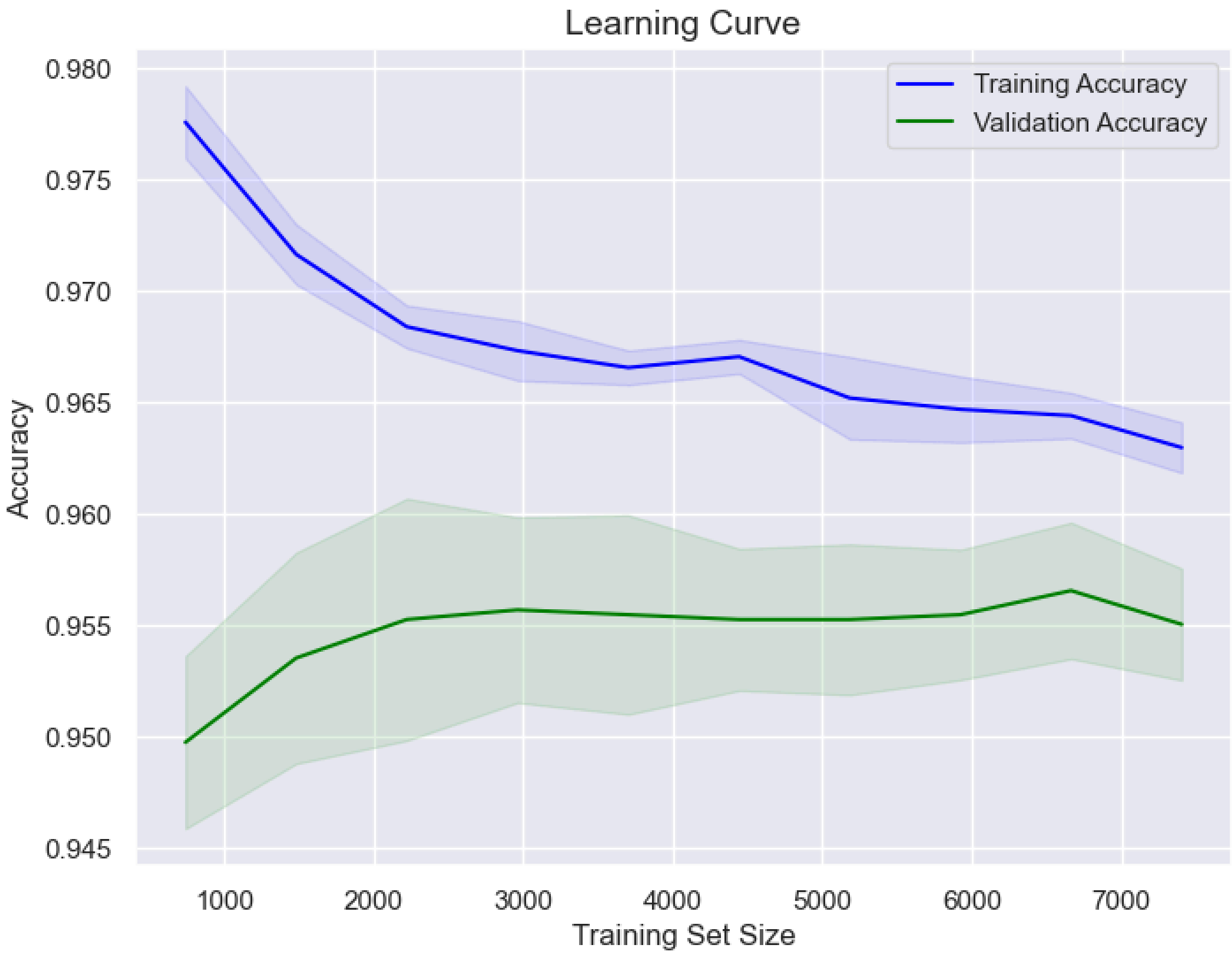
In [51]:
```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

#  for generating the confusion matrix display
ConfusionMatrixDisplay.from_predictions(ytest, yhat, labels=best_rf.classes_,
                                        xticks_rotation="vertical", normalize="true",
                                        cmap=plt.cm.Blues)

# Accessing current axes and turning off grid lines
ax = plt.gca()  # Get current axes
ax.grid(False)  # Turn off grid
# Adding a title to the confusion matrix plot
plt.title('Normalized Confusion Matrix Random Forest', fontsize=14)
plt.show()  # Display the plot
```



The model does well in predicting the slight and good in the serious occurrences

In [52]:

```python
from sklearn.model_selection import learning_curve

# Compute the learning curve
train_sizes, train_scores, val_scores = learning_curve(
    best_rf, Xtrain, ytrain, cv=5, scoring='accuracy',
    train_sizes=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

# Calculate the mean and standard deviation for plotting
train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)
train_std = train_scores.std(axis=1)
val_std = val_scores.std(axis=1)

# Plotting the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_mean, label='Training Accuracy', color='blue')
plt.plot(train_sizes, val_mean, label='Validation Accuracy', color='green')

# Adding shading for standard deviation
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='green')

plt.title('Learning Curve', fontsize=14)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



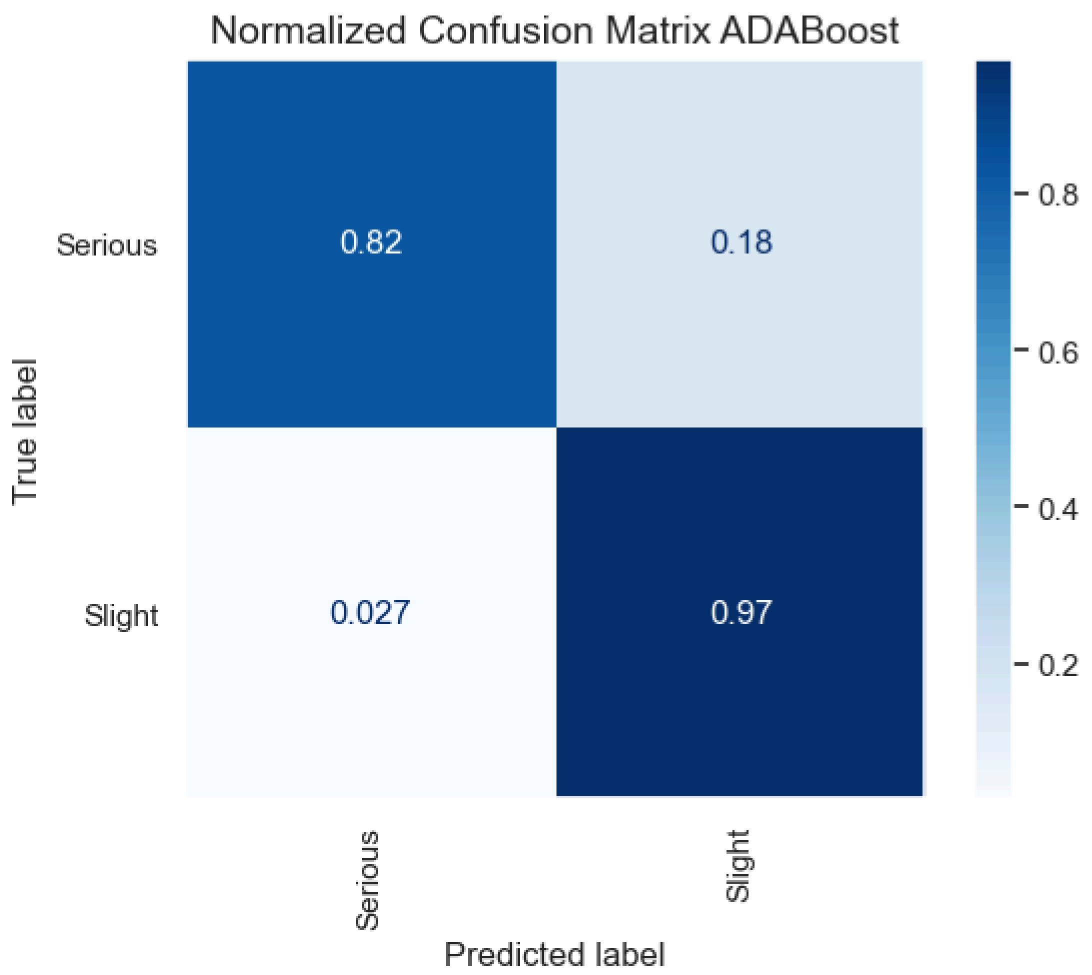
- At 1000 Training Set Size: The training accuracy is high, but the validation accuracy is lower, indicating potential overfitting.
- From 3000 Training Set Size: Both training and validation accuracies might be closer, suggesting better generalization.

ADABoost

In [53]:
```python
# Predict the model
yhat = best_ab.predict(Xtest)

# micro-averaged precision, recall and f-score
p, r, f, s = precision_recall_fscore_support(ytest, yhat, average="macro")
print("ADABoost:")
print(f"Precision: {p}")
print(f"Recall: {r}")
print(f"F score: {f}")
```

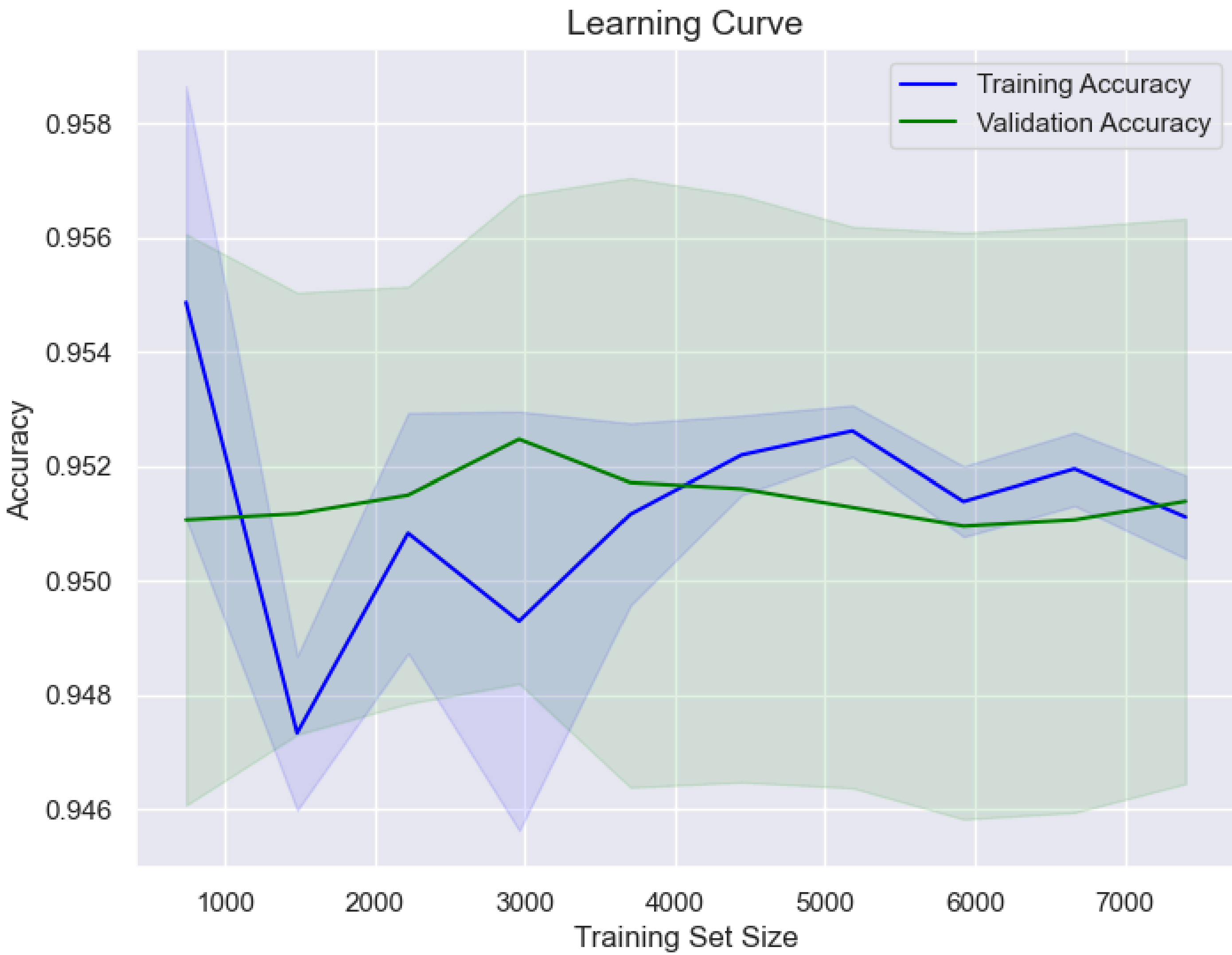
```
ADABoost:
Precision: 0.9117113076167287
Recall: 0.89787051052704
F score: 0.9045950489558934
```

In [54]:
```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# generating the confusion matrix display
ConfusionMatrixDisplay.from_predictions(ytest, yhat, labels=best_ab.classes_,
                                        xticks_rotation="vertical", normalize="true",
                                        cmap=plt.cm.Blues)

# Accessing current axes and turning off grid lines
ax = plt.gca()  # Get current axes
ax.grid(False)  # Turn off grid
# Adding a title to the confusion matrix plot
plt.title('Normalized Confusion Matrix ADABoost', fontsize=14)
plt.show()  # Display the plot
```



The model does well in predicting the slight and good in the serious occurrences

In [55]:
```
from sklearn.model_selection import learning_curve

# Compute the learning curve
train_sizes, train_scores, val_scores = learning_curve(
    best_ab, Xtrain, ytrain, cv=5, scoring='accuracy',
    train_sizes=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

# Calculate the mean and standard deviation for plotting
train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)
train_std = train_scores.std(axis=1)
val_std = val_scores.std(axis=1)

# Plotting the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_mean, label='Training Accuracy', color='blue')
plt.plot(train_sizes, val_mean, label='Validation Accuracy', color='green')

# Adding shading for standard deviation
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='green')

plt.title('Learning Curve', fontsize=14)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



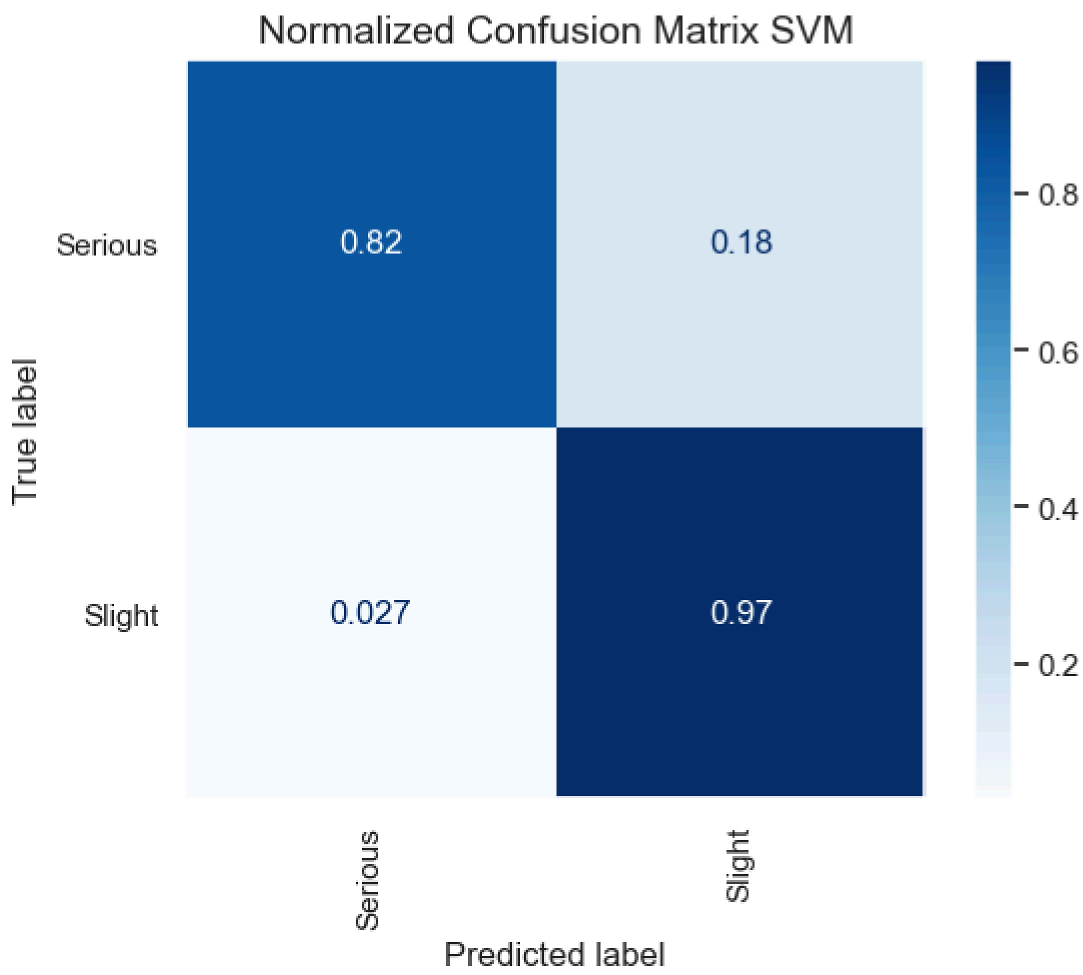
The slight decrease in accuracy as the training set size increases might indicate that the model's performance is stabilizing. This plateau suggests that adding more data may not significantly improve the model's performance.

Support Vector Machines

In [56]:
```python
#Pedict the model
yhat = best_svm.predict(Xtest)

# micro-averaged precision, recall and f-score
p, r, f, s = precision_recall_fscore_support(ytest, yhat, average="macro")
print("SVM:")
print(f"Precision: {p}")
print(f"Recall: {r}")
print(f"F score: {f}")
```

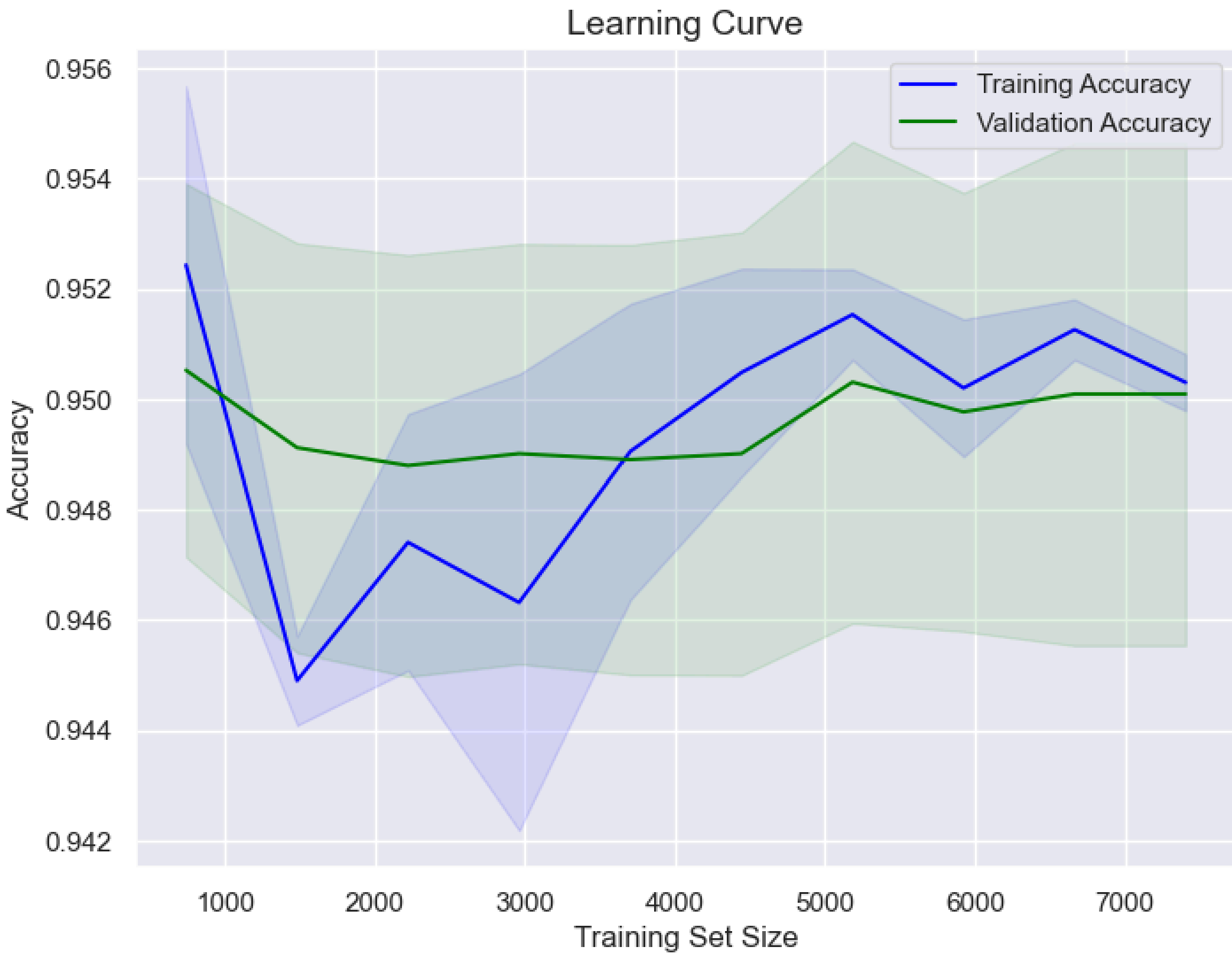
```
SVM:
Precision: 0.9117113076167287
Recall: 0.89787051052704
F score: 0.9045950489558934
```

In [57]:
```python
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# generating the confusion matrix display
ConfusionMatrixDisplay.from_predictions(ytest, yhat, labels=best_svm.classes_,
                                        xticks_rotation="vertical", normalize="true",
                                        cmap=plt.cm.Blues)

# Accessing current axes and turning off grid lines
ax = plt.gca()  # Get current axes
ax.grid(False)  # Turn off grid
# Adding a title to the confusion matrix plot
plt.title('Normalized Confusion Matrix SVM', fontsize=14)
plt.show()  # Display the plot
```



The model does well in predicting the slight and good in the serious occurrences

In [58]:
```python
from sklearn.model_selection import learning_curve

# Compute the learning curve
train_sizes, train_scores, val_scores = learning_curve(
    best_svm, Xtrain, ytrain, cv=5, scoring='accuracy',
    train_sizes=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

# Calculate the mean and standard deviation for plotting
train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)
train_std = train_scores.std(axis=1)
val_std = val_scores.std(axis=1)

# Plotting the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_mean, label='Training Accuracy', color='blue')
plt.plot(train_sizes, val_mean, label='Validation Accuracy', color='green')

# Adding shading for standard deviation
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='green')

plt.title('Learning Curve', fontsize=14)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



As the training set size increases, the training accuracy shows a slight decrease. This is a common trend because, with more data, the model finds it slightly harder to fit all the data points perfectly, which can be a sign of a more generalized model.

Logistic Regression

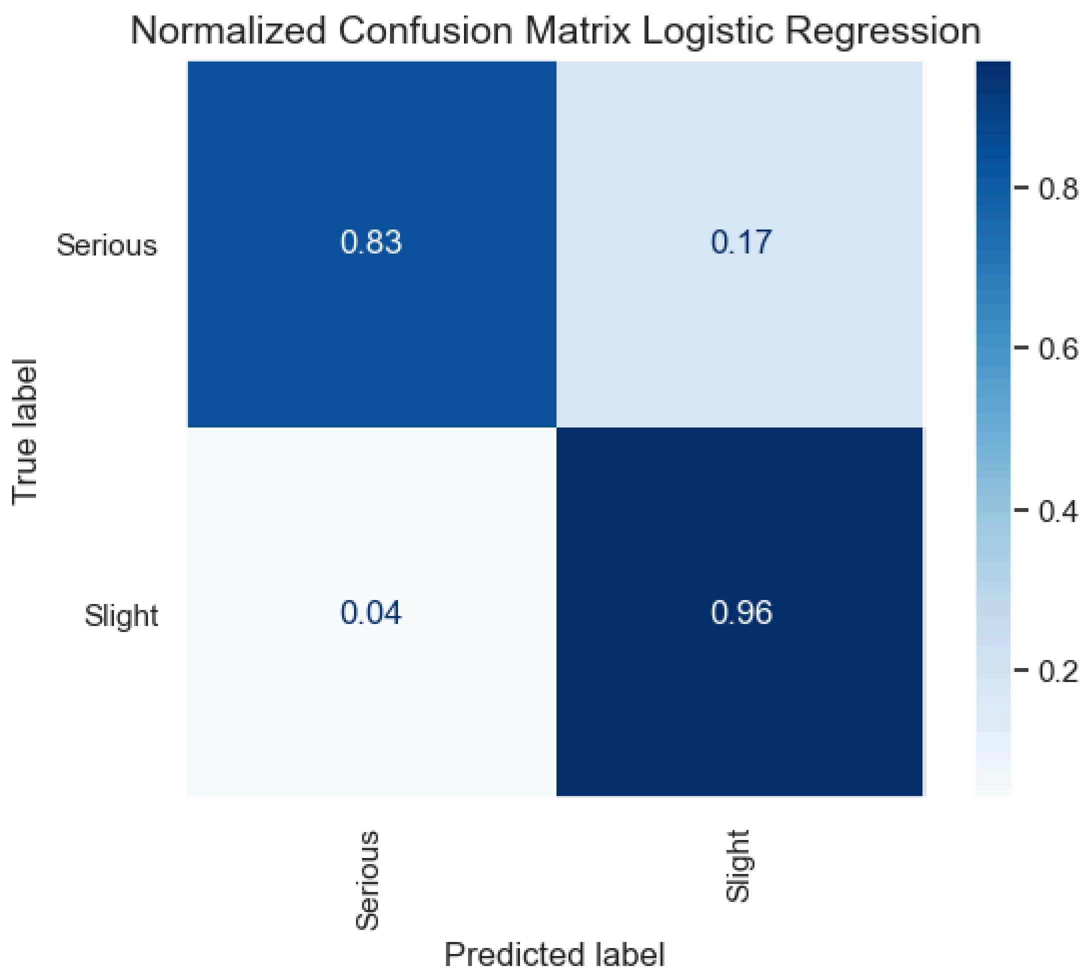
In [59]:

```python
#Predict the model
yhat = best_log_reg.predict(Xtest)

# micro-averaged precision, recall and f-score
p, r, f, s = precision_recall_fscore_support(ytest, yhat, average="macro")
print("Logistic Regression:")
print(f"Precision: {p}")
print(f"Recall: {r}")
print(f"F score: {f}")
```

```
Logistic Regression:
Precision: 0.886463700234192
Recall: 0.8973923214531443
F score: 0.8917973732056118
```

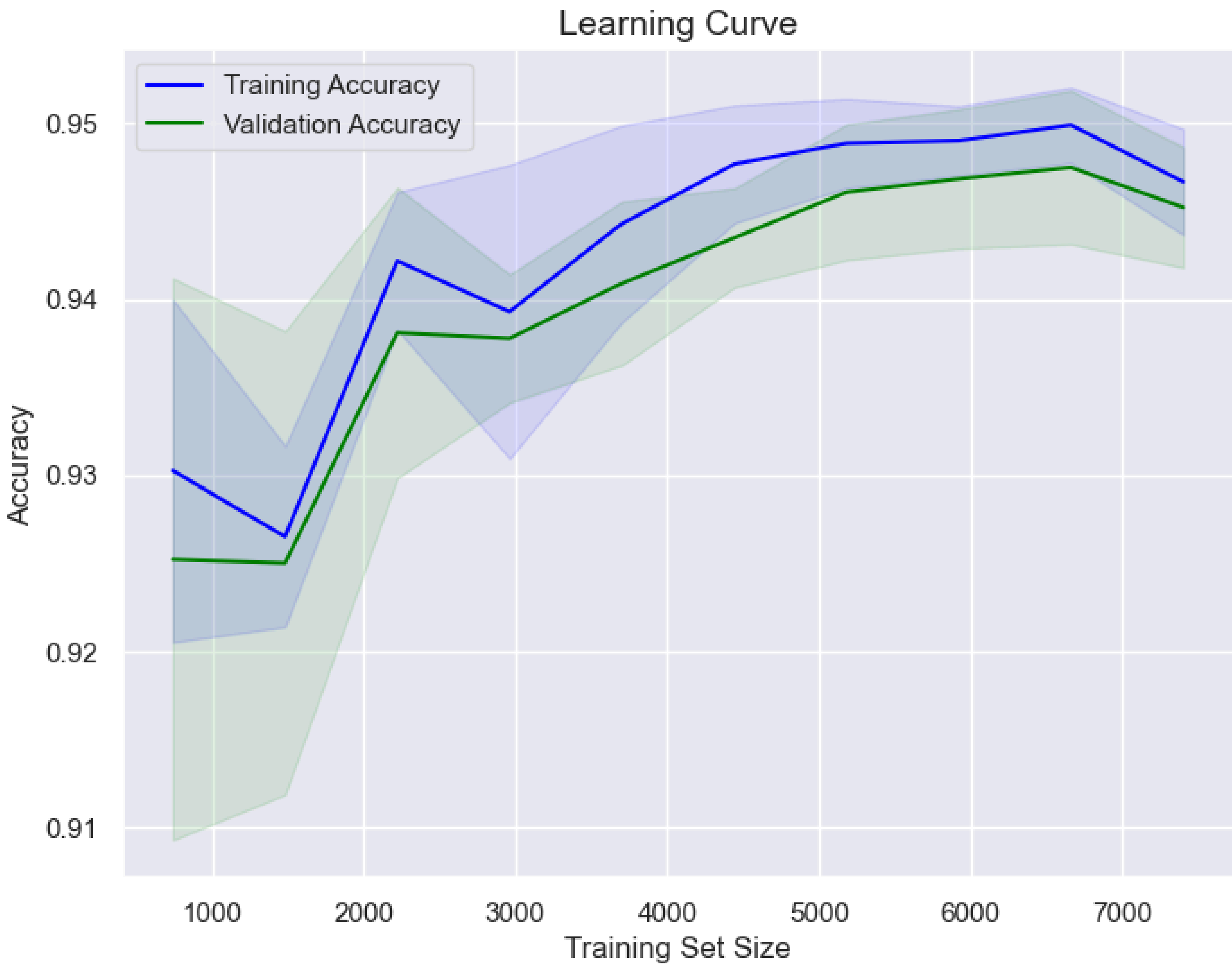
In [60]:

```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# generating the confusion matrix display
ConfusionMatrixDisplay.from_predictions(ytest, yhat, labels=best_log_reg.classes_,
                                        xticks_rotation="vertical", normalize="true",
                                        cmap=plt.cm.Blues)

# Accessing current axes and turning off grid lines
ax = plt.gca()  # Get current axes
ax.grid(False)  # Turn off grid
# Adding a title to the confusion matrix plot
plt.title('Normalized Confusion Matrix Logistic Regression', fontsize=14)
plt.show()  # Display the plot
```



The model does well in predicting the slight and good in the serious occurrences`

In [61]:
```python
from sklearn.model_selection import learning_curve

# Compute the learning curve
train_sizes, train_scores, val_scores = learning_curve(
    best_log_reg, Xtrain, ytrain, cv=5, scoring='accuracy',
    train_sizes=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

# Calculate the mean and standard deviation for plotting
train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)
train_std = train_scores.std(axis=1)
val_std = val_scores.std(axis=1)

# Plotting the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_mean, label='Training Accuracy', color='blue')
plt.plot(train_sizes, val_mean, label='Validation Accuracy', color='green')

# Adding shading for standard deviation
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='green')

plt.title('Learning Curve', fontsize=14)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



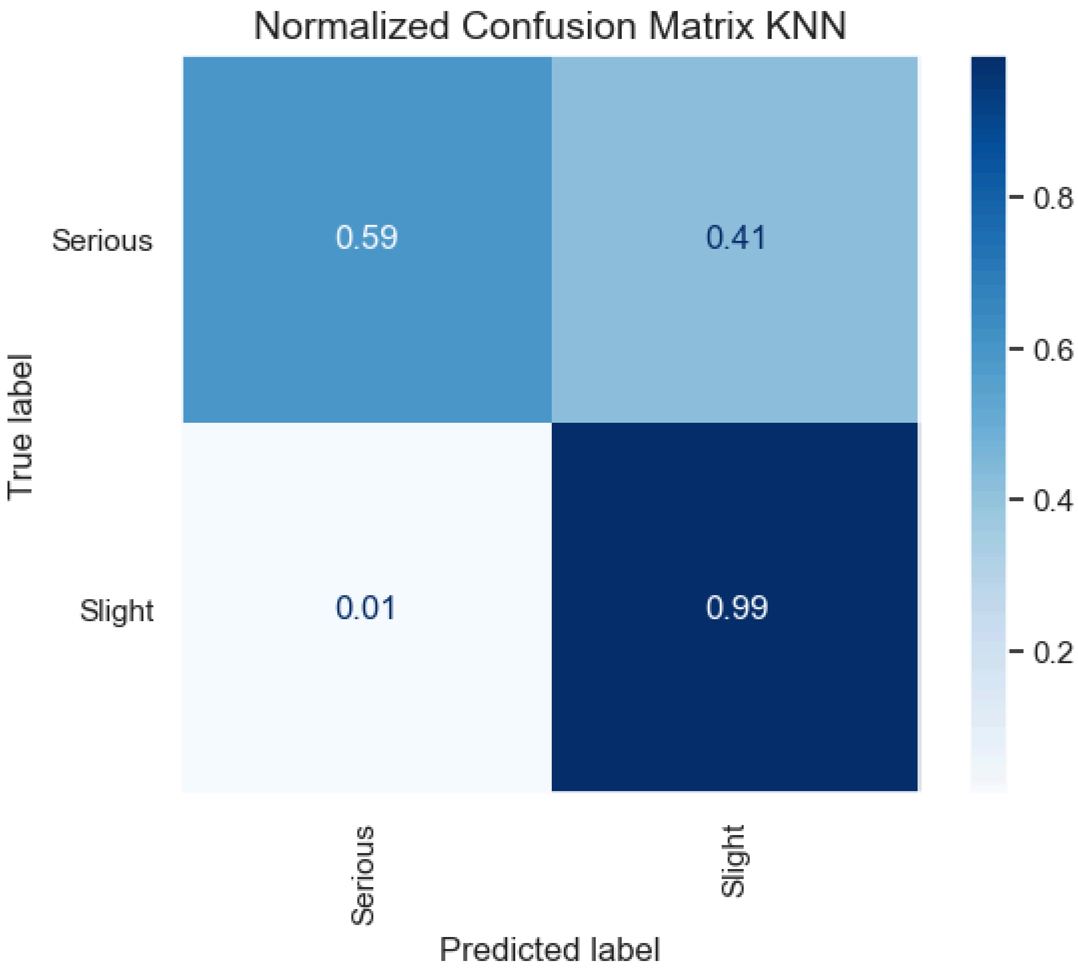
As the training set size increases, the training accuracy shows progressive increase then a fall, which can be a sign of a more generalized model.

KNN

In [62]:
```python
#Predict the model
yhat = best_knn.predict(Xtest)

# micro-averaged precision, recall and f-score
p, r, f, s = precision_recall_fscore_support(ytest, yhat, average="macro")
print("KNN:")
print(f"Precision: {p}")
print(f"Recall: {r}")
print(f"F score: {f}")
```

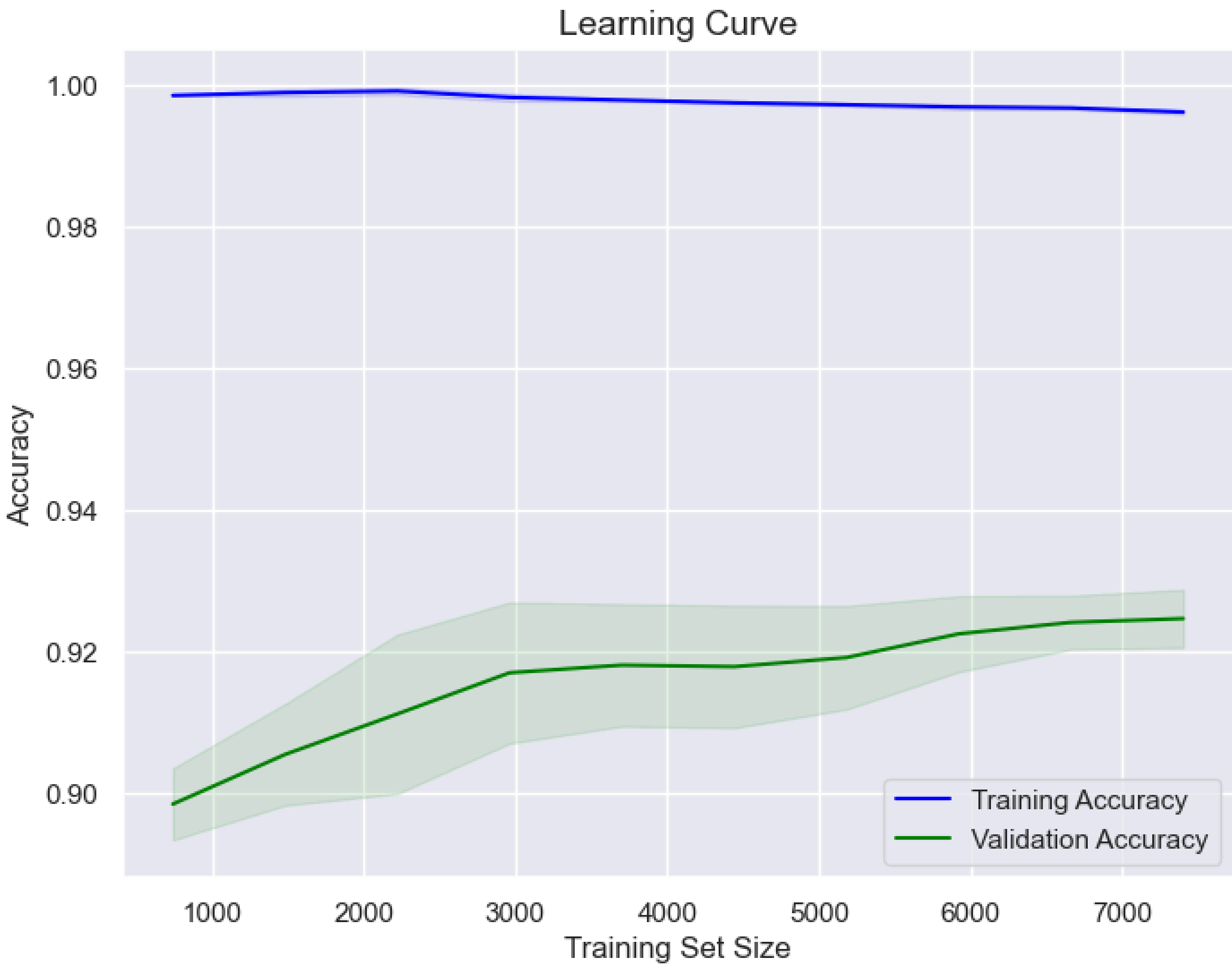
```
KNN:
Precision: 0.9212155963302753
Recall: 0.7906254300261456
F score: 0.8378021178562767
```

In [63]:
```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# generating the confusion matrix display
ConfusionMatrixDisplay.from_predictions(ytest, yhat, labels=best_knn.classes_,
                                        xticks_rotation="vertical", normalize="true",
                                        cmap=plt.cm.Blues)

# Accessing current axes and turning off grid lines
ax = plt.gca()  # Get current axes
ax.grid(False)  # Turn off grid
# Adding a title to the confusion matrix plot
plt.title('Normalized Confusion Matrix KNN', fontsize=14)
plt.show()  # Display the plot
```



The model does well in predicting the slight and not so well in the serious occurrences

In [64]:
```python
from sklearn.model_selection import learning_curve

# Compute the learning curve
train_sizes, train_scores, val_scores = learning_curve(
    best_knn, Xtrain, ytrain, cv=5, scoring='accuracy',
    train_sizes=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

# Calculate the mean and standard deviation for plotting
train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)
train_std = train_scores.std(axis=1)
val_std = val_scores.std(axis=1)

# Plotting the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_mean, label='Training Accuracy', color='blue')
plt.plot(train_sizes, val_mean, label='Validation Accuracy', color='green')

# Adding shading for standard deviation
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='green')

plt.title('Learning Curve', fontsize=14)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



There is overfitting as shown by the difference between the validation and train accuracy

Comparison of the Models

In [65]:
```
from sklearn.metrics import precision_score, recall_score, f1_score
import pandas as pd


# Create a list of models and their names
models = {
    "Decision Tree": best_dt,
    "Random Forest": best_rf,
    "AdaBoost": best_ab,
    "SVM": best_svm,
    "Logistic Regression": best_log_reg,
    "KNN": best_knn
}

# Initialize a dictionary to store the results
results = {
    "Model": [],
    "Precision": [],
    "Recall": [],
    "F1 Macro": []
}

# Evaluate each model
for model_name, model in models.items():
    # Make predictions
    y_pred = model.predict(Xtest)

    # Compute metrics
    precision = precision_score(ytest, y_pred, average='macro')
    recall = recall_score(ytest, y_pred, average='macro')
    f1 = f1_score(ytest, y_pred, average='macro')

    # Append results to the dictionary
    results["Model"].append(model_name)
    results["Precision"].append(precision)
    results["Recall"].append(recall)
    results["F1 Macro"].append(f1)

# Convert the results dictionary to a pandas DataFrame
results_df = pd.DataFrame(results)
```

```python
# Rank the models by F1 Macro score
results_df = results_df.sort_values(by="Recall", ascending=False)

# Display the results
results_df
```

Out[65]:

	Model	Precision	Recall	F1 Macro
1	Random Forest	0.937125	0.903103	0.919004
0	Decision Tree	0.919866	0.899615	0.909330
2	AdaBoost	0.911711	0.897871	0.904595
3	SVM	0.911711	0.897871	0.904595
4	Logistic Regression	0.886464	0.897392	0.891797
5	KNN	0.921216	0.790625	0.837802

Based on the results, Random Forest emerges as the top-performing model with the highest precision (0.9371) and a competitive F1 macro score (0.9190), making it the most reliable for minimizing false positives while maintaining a strong balance between precision and recall. Decision Tree follows closely, with the high recall (0.9198) and a comparable F1 score (0.9093), indicating its effectiveness in capturing true positives. While SVM and ADAboost also perform well, Logistic Regression and KNN lag behind, particularly in recall and F1 scores. Overall, Random Forest is the preferred model for tasks prioritizing precision and recall

Rebuilding the model using Bayes Search

Random Forest - Bayes

In [67]:
```python
from skopt import BayesSearchCV
from skopt.space import Integer, Categorical
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import RandomForestClassifier
from timeit import default_timer as timer
from datetime import timedelta

# Start the timer
start = timer()

# Identify categorical and numerical columns
categorical_columns = Xtrain.select_dtypes(include=['object', 'category']).columns.tolist()
numerical_columns = Xtrain.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Use SimpleImputer to handle missing values in categorical columns
# Apply OneHotEncoder for categorical columns
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),  # Impute missing values
            ('encoder', OneHotEncoder(handle_unknown='ignore'))  # OneHotEncode categorical columns
        ]), categorical_columns),
        ('num', 'passthrough', numerical_columns)  # Leave numerical columns as they are
    ])

# Define the Random Forest Classifier
rf = RandomForestClassifier(random_state=7)

# Define the hyperparameter search space for Bayesian optimization
hp_space = {
    'classifier__n_estimators': Integer(50, 200),  # Number of trees in the forest
    'classifier__max_depth': Integer(5, 30),  # Maximum depth of the tree
    'classifier__min_samples_split': Integer(2, 10),  # Minimum number of samples required to split a node
    'classifier__min_samples_leaf': Integer(1, 4),  # Minimum number of samples required at each leaf node
}

# Create a pipeline with preprocessing and the model
pipeline = ImbPipeline(steps=[
```

```python
    ('preprocessor', preprocessor),
    ('smote', SMOTE(random_state=7)),
    ('classifier', rf)
])

# Initialize BayesSearchCV
bayes_search = BayesSearchCV(pipeline, hp_space, cv=5,
                             scoring='f1_macro',
                             n_iter=50,  # Number of iterations for Bayesian optimization
                             random_state=7, verbose=2,return_train_score=True,)

# Fit the Bayesian search
bayes_search.fit(Xtrain, ytrain)

# Calculate execution time
time_rf_improved = timedelta(seconds=timer() - start)

print("Execution time HH:MM:SS:", time_rf_improved)
```

```
[CV] END classifier__max_depth=30, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_
estimators=65; total time=   2.4s
[CV] END classifier__max_depth=30, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_
estimators=65; total time=   2.6s
[CV] END classifier__max_depth=30, classifier__min_samples_leaf=4, classifier__min_samples_split=10, classifier__n_
estimators=65; total time=   2.6s
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[CV] END classifier__max_depth=19, classifier__min_samples_leaf=4, classifier__min_samples_split=7, classifier__n_e
stimators=133; total time=   4.3s
[CV] END classifier__max_depth=19, classifier__min_samples_leaf=4, classifier__min_samples_split=7, classifier__n_e
stimators=133; total time=   4.3s
[CV] END classifier__max_depth=19, classifier__min_samples_leaf=4, classifier__min_samples_split=7, classifier__n_e
stimators=133; total time=   4.3s
[CV] END classifier__max_depth=19, classifier__min_samples_leaf=4, classifier__min_samples_split=7, classifier__n_e
stimators=133; total time=   4.4s
[CV] END classifier__max_depth=19, classifier__min_samples_leaf=4, classifier__min_samples_split=7, classifier__n_e
stimators=133; total time=   4.2s
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[CV] END classifier__max_depth=14, classifier__min_samples_leaf=3, classifier__min_samples_split=10, classifier__n_
estimators=199; total time=   4.9s
```

In [68]:

```
bayes_search.best_estimator_
```

Out[68]:

Pipeline

preprocessor: ColumnTransformer

cat	**num**
SimpleImputer	passthrough
OneHotEncoder	

SMOTE

RandomForestClassifier

```
RandomForestClassifier(max_depth=9, min_samples_leaf=2, n_estimators=60,
                       random_state=7)
```

In [69]:
```python
cv_results = pd.DataFrame(bayes_search.cv_results_)[['params', 'mean_train_score', 'mean_test_score']]
cv_results["diff, %"] = 100*(cv_results["mean_train_score"]-cv_results["mean_test_score"]
                                                    )/cv_results["mean_train_score"]

pd.set_option('display.max_colwidth', 100)
cv_results.sort_values('mean_test_score', ascending=False)
```

18	{'classifier__max_depth': 5, 'classifier__min_samples_leaf': 4, 'classifier__min_samples_split':...	0.922105	0.922040	0.007050
32	{'classifier__max_depth': 10, 'classifier__min_samples_leaf': 2, 'classifier__min_samples_split'...	0.922697	0.922040	0.071191
49	{'classifier__max_depth': 5, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split':...	0.922403	0.921868	0.058005
42	{'classifier__max_depth': 9, 'classifier__min_samples_leaf': 3, 'classifier__min_samples_split':...	0.922464	0.921866	0.064907
27	{'classifier__max_depth': 13, 'classifier__min_samples_leaf': 2, 'classifier__min_samples_split'...	0.923482	0.921778	0.184452
3	{'classifier__max_depth': 13, 'classifier__min_samples_leaf': 3, 'classifier__min_samples_split'...	0.923203	0.921738	0.158657
41	{'classifier__max_depth': 23, 'classifier__min_samples_leaf': 2, 'classifier__min_samples_split'...	0.929815	0.921723	0.870261
23	{'classifier__max_depth': 30, 'classifier__min_samples_leaf': 3, 'classifier__min_samples_split'...	0.928374	0.921539	0.736287
19	{'classifier__max_depth': 23, 'classifier__min_samples_leaf': 3, 'classifier__min_samples_split'...	0.926789	0.921535	0.566979
20	{'classifier__max_depth': 14, 'classifier__min_samples_leaf': 4, 'classifier__min_samples_split'...	0.922718	0.921526	0.129147
8	{'classifier__max_depth': 27, 'classifier__min_samples_leaf': 3, 'classifier__min_samples_split'...	0.927312	0.921380	0.639742
40	{'classifier__max_depth': 14, 'classifier__min_samples_leaf': 4, 'classifier__min_samples_split'...	0.922776	0.921225	0.167982

In [70]:
```python
dump(bayes_search.best_estimator_, 'models/rf_bayes.joblib')
```

Out[70]: ['models/rf_bayes.joblib']

In [71]:
```python
best_rf_bayes = load("models/rf_bayes.joblib")
```

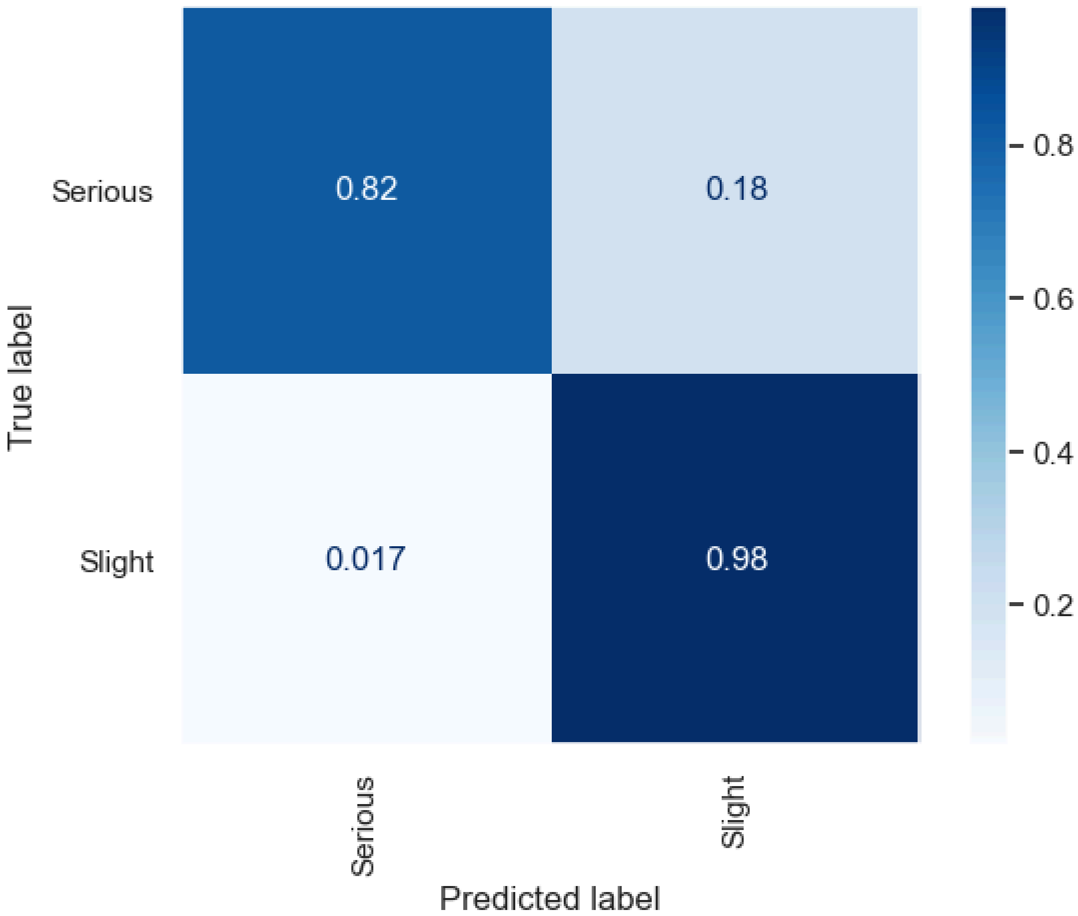
In [72]:
```python
yhat = best_rf_bayes.predict(Xtest)

# micro-averaged precision, recall and f-score
p, r, f, s = precision_recall_fscore_support(ytest, yhat, average="macro")
print("Random Forest - Bayes:")
print(f"Precision: {p}")
print(f"Recall: {r}")
print(f"F score: {f}")
```

```
Random Forest - Bayes:
Precision: 0.9341558441558442
Recall: 0.9025216733177377
F score: 0.9173722575637262
```

The results demonstrates strong performance with a precision of 94%, indicating its ability to minimize false positives, and a recall of 89%, reflecting its effectiveness in capturing true positives. The F1 score of 0.9176 further confirms its robust balance between precision and recall

In [73]:

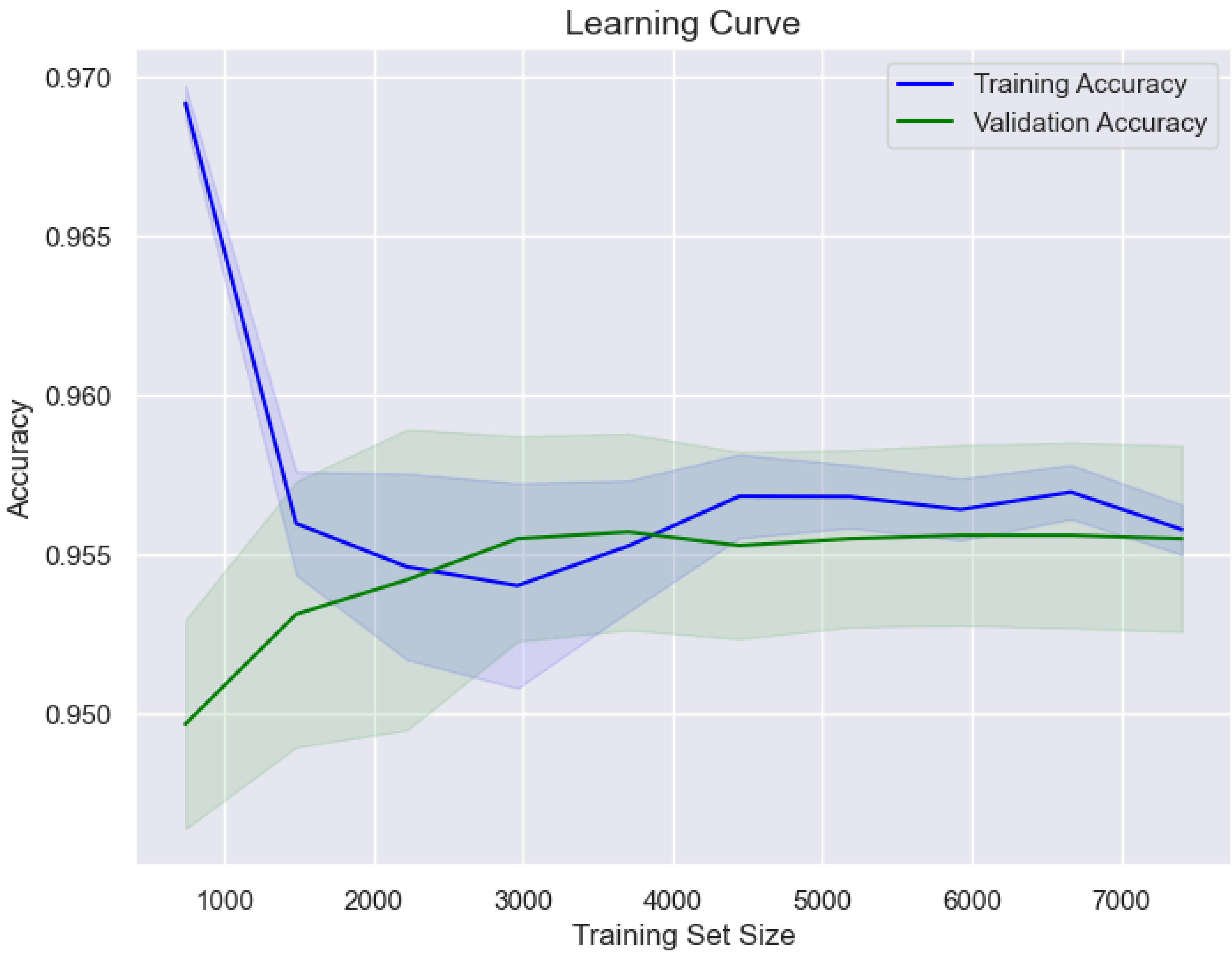
```python
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# for generating the confusion matrix display
ConfusionMatrixDisplay.from_predictions(ytest, yhat, labels=best_rf_bayes.classes_,
                                        xticks_rotation="vertical", normalize="true",
                                        cmap=plt.cm.Blues)

# Accessing current axes and turning off grid lines
ax = plt.gca()  # Get current axes
ax.grid(False)  # Turn off grid

plt.show()  # Display the plot
```



The model is very good at predicting slight casualties(99%) and has a high score in predicting serious casualties(80%)

In [74]:

```
from sklearn.model_selection import learning_curve

# Compute the learning curve
train_sizes, train_scores, val_scores = learning_curve(
    best_rf_bayes, Xtrain, ytrain, cv=5, scoring='accuracy',
    train_sizes=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

# Calculate the mean and standard deviation for plotting
train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)
train_std = train_scores.std(axis=1)
val_std = val_scores.std(axis=1)

# Plotting the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_mean, label='Training Accuracy', color='blue')
plt.plot(train_sizes, val_mean, label='Validation Accuracy', color='green')

# Adding shading for standard deviation
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='green')

plt.title('Learning Curve', fontsize=14)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



As the trainset size increases we see the validation accuracy increases and then seems to plateau indicating generalization of the model

Comparing the best model with the crossvalidation score, we see a very small difference between the two which suggests a good model

Compare the results

In [91]:

```
# Create a list of models and their names
models = {
    "Random Forest - Bayes": best_rf_bayes,
    "Random Forest - Grid Search": best_rf,
}

# Initialize a dictionary to store the results
results = {
    "Model": [],
    "Precision": [],
    "Recall": [],
    "F1 Macro": []
}

# Evaluate each model
for model_name, model in models.items():
    # Make predictions
    y_pred = model.predict(Xtest)

    # Compute metrics
    precision = precision_score(ytest, y_pred, average='macro')
    recall = recall_score(ytest, y_pred, average='macro')
    f1 = f1_score(ytest, y_pred, average='macro')

    # Append results to the dictionary
    results["Model"].append(model_name)
    results["Precision"].append(precision)
    results["Recall"].append(recall)
    results["F1 Macro"].append(f1)

# Convert the results dictionary to a pandas DataFrame
results_df = pd.DataFrame(results)

# Rank the models by F1 Macro score (or any other metric)
results_df = results_df.sort_values(by="F1 Macro", ascending=False)

# Display the results
results_df
```

Out[91]:

	Model	Precision	Recall	F1 Macro
1	Random Forest - Grid Search	0.937125	0.903103	0.919004
0	Random Forest - Bayes	0.934156	0.902522	0.917372

Random Forest - I observe that there is a slight improvement in the model in the precision and recall.

Feature Selection

In [86]:
```
# Load the stored model
best_rf_bayes = load("models/rf_bayes.joblib")

# Access the best estimator (in this case, best_rf_bayes is already the best estimator)
best_estimator = best_rf_bayes

# Check if the best estimator is a Pipeline
if hasattr(best_estimator, 'named_steps'):
    # Access the model step in the pipeline
    model = best_estimator.named_steps['classifier']
    # Access the preprocessor to get the transformed feature names
    preprocessor = best_estimator.named_steps['preprocessor']  # Replace 'preprocessor' with the name of the preprocess

    # Get the feature names after preprocessing
    try:
        # For ColumnTransformer
        feature_names = preprocessor.get_feature_names_out()
    except AttributeError:
        # For older versions of scikit-learn or non-ColumnTransformer preprocessors
        feature_names = Xtrain.columns  # Fallback to original feature names

    # Check if the model has feature_importances_
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
        importance_df = pd.DataFrame({'Feature': feature_names, 'Importance': importances})
        importance_df = importance_df.sort_values(by='Importance', ascending=False)
        print(importance_df)
    else:
        print("The model in the pipeline does not support feature importances.")
else:
    print("The best estimator is not a pipeline.")
```

```
41                     cat__hit_object_in_carriageway_Previous accident
100                cat__pedestrian_location_In carriageway, not crossing
27                              cat__skidding_and_overturning_Jackknifed
48   cat__vehicle_leaving_carriageway_Offside on to central reservation
0                      cat__towing_and_articulation_Articulated vehicle

     Importance
96     0.130234
93     0.080291
90     0.080281
105    0.077452
92     0.064478
..          ...
41     0.000000
100    0.000000
27     0.000000
48     0.000000
0      0.000000

[119 rows x 2 columns]
```

In [87]:
```
#Check the 20 most important features
importance_df.head(20)
```

Out[87]:

	Feature	Importance
96	cat__age_band_of_casualty_66 - 75	0.130234
93	cat__age_band_of_casualty_46 - 55	0.080291
90	cat__age_band_of_casualty_21 - 25	0.080281
105	cat__pedestrian_location_Unknown or other	0.077452
92	cat__age_band_of_casualty_36 - 45	0.064478
86	cat__sex_of_casualty_Male	0.053580
94	cat__age_band_of_casualty_56 - 65	0.049874
85	cat__sex_of_casualty_Female	0.046626
91	cat__age_band_of_casualty_26 - 35	0.035219
84	cat__casualty_class_Pedestrian	0.032238
102	cat__pedestrian_location_Not a Pedestrian	0.030215
107	cat__pedestrian_movement_Crossing from driver's offside	0.025636
106	cat__pedestrian_movement_Crossing from driver's nearside	0.023834
95	cat__age_band_of_casualty_6 - 10	0.023715
109	cat__pedestrian_movement_Not a Pedestrian	0.022416
110	cat__pedestrian_movement_Unknown or other	0.021787
82	cat__casualty_class_Driver or rider	0.020383
83	cat__casualty_class_Passenger	0.020318
103	cat__pedestrian_location_On footway or verge	0.015723
98	cat__pedestrian_location_Crossing on pedestrian crossing facility	0.014968

The most important feature was from the age band, pedestrian location, and sex. Under age band categorical variable particularly "66-75" was most important.

5. Conclusion

After an initial evaluation of six models, their performance was rigorously assessed, and the selection was narrowed down to Random Forest. The model underwent hyperparameter tuning using Bayesian optimization to enhance the predictive accuracy. The results of the evaluation are summarized as follows

Based on the results we pick Random Forest and get the best features from it.

Given the context of the problem—classifying casualties—recall is prioritized, as actual positives is critical. Consequently, Random Forest, optimized using Bayesian hyperparameter tuning, was selected as the best-performing model. Its higher precision aligns with the project's objectives, ensuring more reliable predictions and reducing the risk of misclassification. This decision underscores the importance of aligning model selection with the specific requirements and priorities of classifying casualty severity in accidents.

6. Possible Future Improvements and Business Scenarios for Model Implementation in real-world

Consider integrating external datasets to enrich the model, such as

- Traffic Patterns: Use data on traffic flow or congestion patterns in the area of the accident.
- Weather Data: Integrate weather data, such as precipitation, temperature, and visibility, which may have a significant effect on accident severity.
- Computational cost: The model run for over 2hrs and this was after the data set was chopped down from 1.8 million records to about 10,000 records meaning larger datasets would take more time.

Business Scenarios:

- Resource Allocation: The model can be used by emergency response teams to prioritize resources effectively. For instance, if the model predicts a high-severity casualty, the system could direct ambulances with specific equipment (like trauma care kits or life support systems) to that accident.
- Faster Response Times: Integration with emergency dispatch systems could reduce response time by optimizing routes for emergency responders based on predicted severity. Additionally, the system could predict the required level of medical attention based on severity, allowing hospitals to prepare in advance.

In [92]:
```python
# Finish Timer
notebook_duration = round((time.time() - startnb)/60, 5)
print(f'The completion of the notebook took {notebook_duration} minutes.')
```

```
The completion of the notebook took 145.96519 minutes.
```

In [95]:
```
%%javascript
var nb = IPython.notebook;
var kernel = IPython.notebook.kernel;
var command = "NOTEBOOK_FULL_PATH = '" + nb.notebook_path + "'";
kernel.execute(command);
```

In [96]:
```python
import io
from nbformat import read, NO_CONVERT

with io.open(NOTEBOOK_FULL_PATH.split("/")[-1], 'r', encoding='utf-8') as f:
    nb = read(f, NO_CONVERT)

word_count = 0
for cell in nb.cells:
    if cell.cell_type == "markdown":
        word_count += len(cell['source'].replace('#', '').lstrip().split(' '))
print(f"Word count: {word_count}")
```

```
Word count: 1511
```

References

Pekar, V. (2024). Big Data for Decision Making. Lecture examples and exercises. (Version 1.0.0). URL: https://github.com/vpekar/bd4dm (https://github.com/vpekar/bd4dm)