Group 31 Modeling Notebook

- Ana Margarida Valente, nr 20240936
- Eduardo Mendes, nr 20240850
- Julia Karpienia, nr 20240514
- Marta Boavida, nr 20240519
- Victoria Goon, nr 20240550

Table of Contents

0. Import Packages

In [1]:
```
## Import standard data processing libraries
import numpy as np
import pandas as pd
import seaborn as sns

# Importing encoder
from sklearn.preprocessing import LabelEncoder
```

```
## Import models
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier

## Import ensemble models
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import StackingClassifier
import xgboost as xgb

# Import Cross Validation methods
from sklearn.metrics import accuracy_score, precision_score, recall_score, f
from sklearn.model_selection import RandomizedSearchCV

# Import imbalanced data methods
from imblearn.over_sampling import SMOTE

# Settings
sns.set()
pd.set_option('display.max_rows', None)  # Show all rows
pd.set_option('display.max_colwidth', None) #Show all columns

## Supress warnings
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
# Code to import imbalaned-learn package for xgboost

# import sys
# !{sys.executable} -m pip install imbalanced-learn
```

1. Import Datasets

Importing the datasets that were created and saved from the preprocessing dataset.

In [3]:
```
# # Import the training, valaidation, and test datasets that were saved from
# train_data = pd.read_csv("train_encoded.csv", low_memory=False)
# validation_data = pd.read_csv("validation_encoded.csv", low_memory=False)
# test_data = pd.read_csv("test_encoded.csv")

# Import the training, valaidation, and test datasets that were saved from t
train_data = pd.read_csv("train_encoded_std_agg_out.csv", low_memory=False)
validation_data = pd.read_csv("validation_encoded_std_agg_out.csv", low_memc
test_data = pd.read_csv("test_encoded_std_agg_out.csv")
```

In [4]:
```
# Set the Claim Identifiers as the index
train_data = train_data.set_index("Claim Identifier")
```

```
validation_data = validation_data.set_index("Claim Identifier")
test_data = test_data.set_index("Claim Identifier")
```

In [5]:
```
# Seperate target variable from the features in both train and validation
X_train = train_data.drop('Claim Injury Type', axis = 1)
y_train = train_data['Claim Injury Type']

X_val = validation_data.drop('Claim Injury Type', axis = 1)
y_val = validation_data['Claim Injury Type']
```

1.1 Encode Target Variable

Label Encoder for target variable (training and validation):

(This needs to be done in both the proprocessing notebook as well as this modeling notebook to be able to interpret the results properly when a model is tested with the KaggleSubmission csv.)

In [6]:
```
#Initiate Label encoder
label_encoder = LabelEncoder()

#Fit the encoder on the training target variable
Y_train_encoded = label_encoder.fit_transform(y_train)

#Transform the training and validation target variable
Y_val_encoded = label_encoder.transform(y_val)

# create a copy of the unencoded target to use when assessing the data - mak
y_val_unencoded = y_val.copy()

#Convert the results back to DataFrames while overriding the previous variab
y_train = pd.DataFrame(Y_train_encoded, columns=['encoded_target'], index=pd
y_val = pd.DataFrame(Y_val_encoded, columns=['encoded_target'], index=pd.Ser
```

2. Sampling Techniques

Below is the WCB dataset class distribution:

1. CANCELLED: 8723
2. NON-COMP: 203607
3. MED ONLY: 48132
4. TEMPORARY: 100810
5. PPD SCH LOSS: 33570
6. PPD NSL: 2759
7. PTD: 49
8. DEATH: 328

Class 7 is significantly lower than the others which will cause the models to not fully train. The following cells is to help address this issue.

Undersampling

The below code takes the size of the minority class + half of the minority class and randomly creates subsamples of each of the majority classes along with all of the minoirty class and creates a new dataset from the amalgamation of those datapoints.

In [7]:
```
# # add the encoded variables back to the x set
# training_data_undersampled = pd.concat([X_train, y_train], axis=1)

# # Separate majority and minority classes
# majority_classes = {}
# for x in range(0,8):
#     if x != 6:
#         majority_classes[x] = training_data_undersampled[training_data_und

# minority_class = training_data_undersampled[training_data_undersampled["en

# size = int(len(minority_class) + (len(minority_class) * 0.5))

# print(size)

# # Perform undersampling
# undersampled_majority_0 = majority_classes[0].sample(n=size, random_state=
# undersampled_majority_1 = majority_classes[1].sample(n=size, random_state=
# undersampled_majority_2 = majority_classes[2].sample(n=size, random_state=
# undersampled_majority_3 = majority_classes[3].sample(n=size, random_state=
# undersampled_majority_4 = majority_classes[4].sample(n=size, random_state=
# undersampled_majority_5 = majority_classes[5].sample(n=size, random_state=
# undersampled_majority_7 = majority_classes[7].sample(n=size, random_state=
# # undersampled_majority.head()
# balanced_data = pd.concat([undersampled_majority_0, undersampled_majority_
#                            undersampled_majority_3, undersampled_majority_
#                            minority_class, undersampled_majority_7])

# # Separate features and target
# X_train = balanced_data.drop(columns='encoded_target')
# y_train = balanced_data['encoded_target']

# # Check class distribution after undersampling
# print("Class distribution after undersampling:", y_train.value_counts())
```

Oversampling

The below cell takes the minority class(es) and created synthetic data for it to match the remaining majority class(es). It uses SMOTE (Synthetic Minority Oversampling TEchnique).

https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html

In [8]:
```
# # Initialize SMOTE
# # sampling_strategy=auto : 'not majority' - specifies the class targeted b
# # k_neighbors default set to 5. Tried 3 and 10, but 5 had best f1 scores.
# smote = SMOTE(sampling_strategy='auto', random_state=42) #k_neighbors=3

# # Fit and resample the dataset
# X_train, y_train = smote.fit_resample(X_train, y_train)

# print("Class distribution after oversampling each at:", y_train.value_cour
```

3. Model

Type of Problem: Multiclassification Problem

Metric used:
As a classification problem, we observed the following metrics to determine the effectiveness of our model: - accuracy - precision - recall - macro f1 score
Each point is measured in a different and observing them all allows us to get an accurate view of our model's results.

In [9]:
```
# Functions to help display metrics for all models

# helper method for score_model - not to be used seperately
def print_scores(per_class):
    for x,y in zip(per_class, np.unique(y_val_unencoded)):
        if str(y) == "7. PTD": # add an extra tab for better alignment
            print("["+str(y)+"]:     \t\t" + str(round(x,2)))
        else:
            print("["+str(y)+"]:     \t" + str(round(x,2)))

# displays the scores for Precision, Recall, and F1
def score_model(y_actual, y_predicted, score_train, score_test):

    print("------------ F1 ------------")
    f1_per_class = f1_score(y_actual, y_predicted, average=None)
    print_scores(f1_per_class)#, y_actual)
    f1_per_weighted = f1_score(y_actual, y_predicted, average='macro')
    print("\nMacro f1: " + str(round(f1_per_weighted, 3)) + "\n")

    print("------ Individual Score Comparisons ------ ")
    print("Train Score: " + str(score_train))
    print("Test Score: " + str(score_test))
    diff = np.abs(score_train - score_test)
    print("Difference: " + str(diff))

    print("--------- Accuracy ---------\n")
    acc_score = accuracy_score(y_actual, y_predicted)
```

```
    print("Accuracy Score: " + str(acc_score) + "\n")

    print("--------- Precision ---------")
    precision_per_class = precision_score(y_actual, y_predicted, average=Nor
    print_scores(precision_per_class)#, y_actual)
    precision_weighted = precision_score(y_actual, y_predicted, average='mac
    print("\nMacro precision: " + str(round(precision_weighted, 3)) + "\n")

    print("---------- Recall ----------")
    recall_per_class = recall_score(y_actual, y_predicted, average=None)
    print_scores(recall_per_class)#, y_actual)
    recall_per_weighted = recall_score(y_actual, y_predicted, average='macro
    print("\nMacro recall: " + str(round(recall_per_weighted, 3)) + "\n")
```

Logistic Regression

Grid Search - Logistic Regression:

In [10]:
```
# param_grid = {'C': [0.1, 1, 10], 'solver': ['lbfgs'], 'class_weight': [Nor
# grid_search = GridSearchCV(LogisticRegression(multi_class='multinomial', r
# grid_search.fit(X_train_std_scaler_encoded, Y_train_encoded_df)

# print("Best Parameters:", grid_search.best_params_)
# print("Best Score:", grid_search.best_score_)
```

Model - Logistic Regression:

In [11]:
```
# Create the model
lr_model = LogisticRegression(multi_class='multinomial', solver='lbfgs', C=1

# Fit the model to the training set
lr_model.fit(X_train, y_train)

# Determine the scores for the model for both train and validation sets
score_train = lr_model.score(X_train, y_train)
score_test = lr_model.score(X_val, y_val)

# Use the model to predict on the validation set
lr_y_pred = lr_model.predict(X_val)

# Display the model metrics using the score_model function
score_model(y_val, lr_y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.53
[2. NON-COMP]:          0.89
[3. MED ONLY]:          0.15
[4. TEMPORARY]:         0.77
[5. PPD SCH LOSS]:      0.6
[6. PPD NSL]:           0.01
[7. PTD]:               0.0
[8. DEATH]:             0.35

Macro f1: 0.414

------ Individual Score Comparisons ------
Train Score: 0.7681128102558433
Test Score: 0.7681292390597417
Difference: 1.6428803898405064e-05
--------- Accuracy ---------

Accuracy Score: 0.7681292390597417

--------- Precision ---------
[1. CANCELLED]:         0.66
[2. NON-COMP]:          0.84
[3. MED ONLY]:          0.33
[4. TEMPORARY]:         0.73
[5. PPD SCH LOSS]:      0.63
[6. PPD NSL]:           0.11
[7. PTD]:               0.0
[8. DEATH]:             0.72

Macro precision: 0.501

---------- Recall ----------
[1. CANCELLED]:         0.44
[2. NON-COMP]:          0.96
[3. MED ONLY]:          0.1
[4. TEMPORARY]:         0.81
[5. PPD SCH LOSS]:      0.58
[6. PPD NSL]:           0.01
[7. PTD]:               0.0
[8. DEATH]:             0.23

Macro recall: 0.392
```

DECISION TREE

Gridsearch - decision tree:

In [12]:
```
# # Initialize the Decision Tree Classifier
# dt_classifier = DecisionTreeClassifier(random_state=42)

# # Define the parameter grid to search
```

```
# param_grid = {
#     'criterion': ['gini', 'entropy'],
#     'splitter': ['best', 'random'],
#     'max_depth': [None, 10, 20, 30],
#     'min_samples_split': [2, 5, 10],
#     'min_samples_leaf': [1, 2, 4],
#     'max_features': [None, 'sqrt', 'log2'],
#     'max_leaf_nodes': [None, 10, 20, 30],
#     'min_impurity_decrease': [0.0, 0.1, 0.2]
# }

# # Initialize GridSearchCV:
# grid_search = GridSearchCV(estimator=dt_classifier, param_grid=param_grid,

# # Fit GridSearchCV on the training data
# grid_search.fit(X_train, y_train)

# print("Best Parameters:", grid_search.best_params_)
# print("Best Score:", grid_search.best_score_)

# best_model = grid_search.best_estimator_

# #Best Parameters: {'criterion': 'gini', 'max_depth': 10, 'max_features': N
# #Best Score: 0.7769977245887005
```

Model - Decision Tree:

In [13]:

```
# Test History
# 0.366
# (oversampling) - 0.343 - 21s

# Initialize the Decision Tree Classifier
decision_tree = DecisionTreeClassifier(
    criterion='gini',
    max_depth=10,
    max_features=None,
    max_leaf_nodes=None,
    min_impurity_decrease= 0.0,
    min_samples_leaf= 1,
    min_samples_split=2,
    splitter='best',
    random_state=42
)

# Train the model
decision_tree.fit(X_train, y_train)

# Determine the scores for the model for both train and validation sets
score_train = decision_tree.score(X_train, y_train)
score_test = decision_tree.score(X_val, y_val)

# Make predictions
dt_y_pred = decision_tree.predict(X_val)
```

```
# Display the model metrics using the score_model function
score_model(y_val, dt_y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.56
[2. NON-COMP]:          0.9
[3. MED ONLY]:          0.13
[4. TEMPORARY]:         0.78
[5. PPD SCH LOSS]:      0.5
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.01

Macro f1: 0.361

------ Individual Score Comparisons ------
Train Score: 0.7804552010412635
Test Score: 0.7779487596395057
Difference: 0.0025064414017578196
--------- Accuracy ---------

Accuracy Score: 0.7779487596395057

--------- Precision ---------
[1. CANCELLED]:         0.7
[2. NON-COMP]:          0.85
[3. MED ONLY]:          0.55
[4. TEMPORARY]:         0.68
[5. PPD SCH LOSS]:      0.69
[6. PPD NSL]:           0.09
[7. PTD]:               0.0
[8. DEATH]:             0.17

Macro precision: 0.467

---------- Recall ----------
[1. CANCELLED]:         0.46
[2. NON-COMP]:          0.96
[3. MED ONLY]:          0.07
[4. TEMPORARY]:         0.92
[5. PPD SCH LOSS]:      0.39
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.01

Macro recall: 0.352
```

K Nearest Neighbors

Grid Search - KNN

In [14]:
```
# # Define the parameter grid for Randomized Search
# param_distributions = {
```

```
#     'n_neighbors' : [5,10],
#     'leaf_size': [30, 50],
#     'metric': ['euclidean', 'manhattan'],
# }

# # Initialize RandomizedSearchCV with KNN classifier
# random_search = RandomizedSearchCV(
#     estimator=KNeighborsClassifier(),
#     param_distributions=param_distributions,
#     n_iter=5,
#     cv=2,
#     scoring='f1_macro',
#     verbose=2,
#     random_state=42,
#     n_jobs=-1
# )

# # Fit the random search to the training data
# random_search.fit(X_train, y_train)

# print("Best Parameters:", random_search.best_params_)
# print("Best Score:", random_search.best_score_)

# # Best Parameters: {'n_neighbors': 5, 'metric': 'euclidean', 'leaf_size':
# # Best Score: 0.328445945439427
```

Model - KNN

- KNN will be commented out
- KNN is not appropriate for too large datasets. Too computational expensive due to memorization requirements.
- KNN takes too long to process due to our large dataset.

In [15]:
```
# Test History
# # 0.334
# # Initialize the KNN Classifier
# knn_model = KNeighborsClassifier(n_neighbors=5, leaf_size=30, metric='eucl

# # Train the model
# knn_model.fit(X_train, y_train)

# # Determine the scores for the model for both train and validation sets
# score_train = knn_model.score(X_train, y_train)
# score_test = knn_model.score(X_val, y_val)

# # Predict on the validation set
# y_pred = knn_model.predict(X_val)

# # Display the model metrics using the score_model function
# score_model(y_val, y_pred, score_train, score_test)
```

Model - Gaussian Naive Bayes:

- Will be commented out
- Assumes normality, independence, Homogeneity of Variance (Homoskedasticity):

In [16]:
```
# # create the model
# model = GaussianNB()

# # fit the model to the training set
# model.fit(X_train, y_train)

# # determine the scores for the model for both train and validation
# score_train = model.score(X_train, y_train)
# score_test = model.score(X_val, y_val)

# # use model to predict on validation set
# y_pred = model.predict(X_val)

# # display the model metrics
# score_model(y_val, y_pred, score_train, score_test)
```

Neural Network (MLPClassifier):

GridSearch - MLPClasssifer:

In [17]:
```
# # Define the parameter grid
# param_grid = {
#     'hidden_layer_sizes': [
#         (50),        # Larger single-layer model
#         (50, 30),      # Moderate two-layer model
#         (100, 50),     # Larger two-layer model
#         (128, 64, 32)  # Complex three-layer model
#     ],
#     'activation': ['relu', 'logistic'],
#     'solver': ['adam', 'sgd'],
#     'alpha': [0.0001, 0.001],
#     'learning_rate': ['adaptive', 'invscaling']
# }

# # Initialize the Neural Network model
# mlp = MLPClassifier(random_state=42)

# # Initialize Random Search for hyperparameter tuning
# random_search = RandomizedSearchCV(
#     estimator=mlp,
#     param_distributions=param_grid,  # Using param_distributions for rando
#     n_iter=10,  # Number of random combinations to try
#     cv=2,  # 3-fold cross-validation
#     scoring='f1_macro',  # Evaluation metric
#     verbose=2,  # Display progress logs
#     n_jobs=-1,  # Use all available processors for parallel computation
#     random_state=42  # For reproducibility
```

```
# )

# # Fit the randomized search to the training data
# random_search.fit(X_train, y_train)

#
# print("Best Parameters:", random_search.best_params_)
# print("Best Score:", random_search.best_score_)

# # Best Parameters: {'solver': 'adam', 'learning_rate': 'adaptive', 'hidden
# # Best Score: 0.4192846184298069
```

MODEL - MLPClassifier:

In [ ]:
```
# with agreement reached - 0.389 - 14m 57s
# (hidden_layer_sizes=(13,), max_iter=500, random_state=42) - 0.395 (no over
# (hidden_layer_sizes=(15,), max_iter=500, random_state=42) - 0.407 (no over
# (hidden_layer_sizes=(20,), max_iter=500, random_state=42) - 0.407 (no over
# (hidden_layer_sizes=(10,), max_iter=500, random_state=42) - 0.389 (no over
# (hidden_layer_sizes=(10,), max_iter=1000, random_state=42) - 0.389 (no ove

# # Initialize the Neural Network model
model = MLPClassifier(hidden_layer_sizes=(50, 30),
                      activation='logistic',
                      solver='adam',
                      alpha=0.0001,
                      learning_rate='adaptive',
                      max_iter=200,
                      random_state=42)

# Train the model
model.fit(X_train, y_train)

# Determine the scores for the model for both train and validation sets
score_train = model.score(X_train, y_train)  # Accuracy on training data
score_test = model.score(X_val, y_val)      # Accuracy on validation data

# Use the model to predict on the validation set
y_pred = model.predict(X_val)

# Display the model metrics using the score_model function
score_model(y_val, y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.58
[2. NON-COMP]:          0.9
[3. MED ONLY]:          0.18
[4. TEMPORARY]:         0.8
[5. PPD SCH LOSS]:      0.64
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.4

Macro f1: 0.439

------ Individual Score Comparisons ------
Train Score: 0.804612315253607
Test Score: 0.78951616649633
Difference: 0.015096148757276940
--------- Accuracy ---------

Accuracy Score: 0.78951616649633

--------- Precision ---------
[1. CANCELLED]:         0.68
[2. NON-COMP]:          0.86
[3. MED ONLY]:          0.5
[4. TEMPORARY]:         0.72
[5. PPD SCH LOSS]:      0.69
[6. PPD NSL]:           0.18
[7. PTD]:               0.0
[8. DEATH]:             0.45

Macro precision: 0.51

---------- Recall ----------
[1. CANCELLED]:         0.51
[2. NON-COMP]:          0.95
[3. MED ONLY]:          0.11
[4. TEMPORARY]:         0.9
[5. PPD SCH LOSS]:      0.6
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.36

Macro recall: 0.429
```

Ensemble Models

Random Forest

Fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

In [19]:
```
# 0.379
# (oversampling) - 0.433 (overfitting w/ 0.23 diff) - 6m 5s

# Initialize the Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model
rf_model.fit(X_train, y_train)

# Determine the scores for the model for both train and validation sets
score_train = rf_model.score(X_train, y_train)  # Accuracy on training data
score_test = rf_model.score(X_val, y_val)      # Accuracy on validation data

# Use the model to predict on the validation set
rf_y_pred = rf_model.predict(X_val)

# Display the model metrics using the score_model function
score_model(y_val, rf_y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.57
[2. NON-COMP]:          0.91
[3. MED ONLY]:          0.14
[4. TEMPORARY]:         0.8
[5. PPD SCH LOSS]:      0.62
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.12

Macro f1: 0.394

------ Individual Score Comparisons ------
Train Score: 0.9999773856846358
Test Score: 0.7917286072656322
Difference: 0.20824877841900358
--------- Accuracy ---------

Accuracy Score: 0.7917286072656322

--------- Precision ---------
[1. CANCELLED]:         0.74
[2. NON-COMP]:          0.86
[3. MED ONLY]:          0.57
[4. TEMPORARY]:         0.71
[5. PPD SCH LOSS]:      0.71
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.82

Macro precision: 0.551

---------- Recall ----------
[1. CANCELLED]:         0.46
[2. NON-COMP]:          0.96
[3. MED ONLY]:          0.08
[4. TEMPORARY]:         0.92
[5. PPD SCH LOSS]:      0.54
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.06

Macro recall: 0.379
```

XGBoost

Also using decision trees

In [20]:
```python
# 0.442
# (oversampling) 0.453 (overfit by 0.10 diff) 4m 18s
# (oversampling) 0.469 (overfit by 0.09 diff) 3m 28s - with agreement reache

# max_depth = 19, n_estimators = 150, lr = 0.6 -> overfitting
xgb_model = xgb.XGBClassifier(
    n_estimators=110,  # Number of trees
```

```
    learning_rate=0.2,  # Step size shrinkage
    max_depth=7,        # Maximum depth of a tree
    random_state=42,    # For reproducibility
    use_label_encoder=False,  # Avoid warning for encoding
    eval_metric='mlogloss'    # Evaluation metric for multi-class classifica
)

# Train the model
xgb_model.fit(X_train, y_train)

# Determine the scores for the model for both train and validation sets
score_train = xgb_model.score(X_train, y_train)  # Accuracy on training data
score_test = xgb_model.score(X_val, y_val)       # Accuracy on validation dat

# Use the model to predict on the validation set
xgb_y_pred = xgb_model.predict(X_val)

# Display the model metrics using the score_model function
score_model(y_val, xgb_y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.58
[2. NON-COMP]:          0.91
[3. MED ONLY]:          0.15
[4. TEMPORARY]:         0.81
[5. PPD SCH LOSS]:      0.65
[6. PPD NSL]:           0.01
[7. PTD]:               0.0
[8. DEATH]:             0.42

Macro f1: 0.442

------ Individual Score Comparisons ------
Train Score: 0.8123614873183945
Test Score: 0.8003751277524853
Difference: 0.011986359565909211
--------- Accuracy ---------

Accuracy Score: 0.8003751277524853

--------- Precision ---------
[1. CANCELLED]:         0.73
[2. NON-COMP]:          0.86
[3. MED ONLY]:          0.56
[4. TEMPORARY]:         0.74
[5. PPD SCH LOSS]:      0.68
[6. PPD NSL]:           0.31
[7. PTD]:               0.0
[8. DEATH]:             0.62

Macro precision: 0.562

---------- Recall ----------
[1. CANCELLED]:         0.48
[2. NON-COMP]:          0.97
[3. MED ONLY]:          0.08
[4. TEMPORARY]:         0.91
[5. PPD SCH LOSS]:      0.63
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.32

Macro recall: 0.424
```

Gradient Boosted Decision Trees

In [22]:
```
# 16 min = max_depth = 6 - .402 f1

gbdt_model = GradientBoostingClassifier(
    n_estimators=100,        # Number of boosting stages
    learning_rate=0.1,       # Shrinks contribution of each tree
    max_depth=6,             # Limits depth of each tree to prevent overfitti
    random_state=42          # For reproducibility
)
```

```
# Train the model
gbdt_model.fit(X_train, y_train)

# Determine the scores for the model for both train and validation sets
score_train = gbdt_model.score(X_train, y_train)  # Accuracy on training dat
score_test = gbdt_model.score(X_val, y_val)      # Accuracy on validation da

# Use the model to predict on the validation set
gbdt_y_pred = gbdt_model.predict(X_val)

# Display the model metrics using the score_model function
score_model(y_val, gbdt_y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.58
[2. NON-COMP]:          0.91
[3. MED ONLY]:          0.15
[4. TEMPORARY]:         0.81
[5. PPD SCH LOSS]:      0.65
[6. PPD NSL]:           0.02
[7. PTD]:               0.0
[8. DEATH]:             0.12

Macro f1: 0.404

------ Individual Score Comparisons ------
Train Score: 0.8054691465357382
Test Score: 0.796455449224194
Difference: 0.009013697311544222
--------- Accuracy ---------

Accuracy Score: 0.796455449224194

--------- Precision ---------
[1. CANCELLED]:         0.71
[2. NON-COMP]:          0.86
[3. MED ONLY]:          0.54
[4. TEMPORARY]:         0.73
[5. PPD SCH LOSS]:      0.68
[6. PPD NSL]:           0.1
[7. PTD]:               0.0
[8. DEATH]:             0.09

Macro precision: 0.464

---------- Recall ----------
[1. CANCELLED]:         0.49
[2. NON-COMP]:          0.96
[3. MED ONLY]:          0.09
[4. TEMPORARY]:         0.91
[5. PPD SCH LOSS]:      0.62
[6. PPD NSL]:           0.01
[7. PTD]:               0.0
[8. DEATH]:             0.2

Macro recall: 0.409
```

Bagging Decision Tree

In [ ]:
```python
# 0.362 - 0.002 diff - 28s
# Bagging Decision Tree
base_model = DecisionTreeClassifier(
    criterion='gini',
    max_depth=10,
    max_features=None,
    max_leaf_nodes=None,
    min_impurity_decrease= 0.0,
```

```
    min_samples_leaf= 1,
    min_samples_split=2,
    splitter='best',
    random_state=42
)
bagging_model_mlpc = BaggingClassifier(estimator=base_model, n_estimators=10
bagging_model_mlpc.fit(X_train, y_train)
bagging_y_pred = bagging_model_mlpc.predict(X_val)

score_train = bagging_model_mlpc.score(X_train, y_train)
score_test = bagging_model_mlpc.score(X_val, y_val)

score_model(y_val, bagging_y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.56
[2. NON-COMP]:          0.91
[3. MED ONLY]:          0.12
[4. TEMPORARY]:         0.79
[5. PPD SCH LOSS]:      0.51
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.01

Macro f1: 0.362

------ Individual Score Comparisons ------
Train Score: 0.7839503691158808
Test Score: 0.7815548638855337
Difference: 0.002395505230347039
--------- Accuracy ---------

Accuracy Score: 0.7815548638855337

--------- Precision ---------
[1. CANCELLED]:         0.71
[2. NON-COMP]:          0.86
[3. MED ONLY]:          0.59
[4. TEMPORARY]:         0.68
[5. PPD SCH LOSS]:      0.69
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.33

Macro precision: 0.483

---------- Recall ----------
[1. CANCELLED]:         0.46
[2. NON-COMP]:          0.96
[3. MED ONLY]:          0.06
[4. TEMPORARY]:         0.94
[5. PPD SCH LOSS]:      0.41
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.01

Macro recall: 0.355
```

Bagging Logistic Regression

In [25]:
```
# 13m - 0.372 - overfit by 0.1 diff
base_model_lr = LogisticRegression(multi_class='multinomial', solver='lbfgs'
bagging_model_lr = BaggingClassifier(estimator=base_model_lr, n_estimators=1
bagging_model_lr.fit(X_train, y_train)
bagging_y_pred = bagging_model_lr.predict(X_val)

score_train = bagging_model_lr.score(X_train, y_train)
score_test = bagging_model_lr.score(X_val, y_val)
```

```
score_model(y_val, bagging_y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.53
[2. NON-COMP]:          0.89
[3. MED ONLY]:          0.15
[4. TEMPORARY]:         0.77
[5. PPD SCH LOSS]:      0.61
[6. PPD NSL]:           0.01
[7. PTD]:               0.0
[8. DEATH]:             0.37

Macro f1: 0.417

------ Individual Score Comparisons ------
Train Score: 0.7678489765765947
Test Score: 0.7679085756759267
Difference: 5.9599099332063865e-05
--------- Accuracy ---------

Accuracy Score: 0.7679085756759267

--------- Precision ---------
[1. CANCELLED]:         0.66
[2. NON-COMP]:          0.84
[3. MED ONLY]:          0.32
[4. TEMPORARY]:         0.73
[5. PPD SCH LOSS]:      0.63
[6. PPD NSL]:           0.11
[7. PTD]:               0.0
[8. DEATH]:             0.69

Macro precision: 0.498

---------- Recall ----------
[1. CANCELLED]:         0.44
[2. NON-COMP]:          0.96
[3. MED ONLY]:          0.1
[4. TEMPORARY]:         0.81
[5. PPD SCH LOSS]:      0.59
[6. PPD NSL]:           0.01
[7. PTD]:               0.0
[8. DEATH]:             0.26

Macro recall: 0.395
```

Stacking

In [ ]:

```
# 0.440 - LR -> XGB -> MLP w/ a 0.015 difference in scores (3m 55s)
# 0.425 - LR -> MLP -> XGB w/ a 0.0099 difference in scores (37m)
# 0.410 - MLP -> XGB -> GBC w/ a 0.011 difference in scores (93m 35s)
# 0.452 - LR -> XGB -> MLP w/ 0,11 different (15m 16s)
# 0.429 0 LR -> DT -> MLPC w/ 0.001 difference (3m 55s)
```

```
base_models = [
    ('lr', LogisticRegression(multi_class='multinomial', solver='lbfgs', C=1
    ('dt', DecisionTreeClassifier(
    criterion='gini',
    max_depth=10,
    max_features=None,
    max_leaf_nodes=None,
    min_impurity_decrease= 0.0,
    min_samples_leaf= 1,
    min_samples_split=2,
    splitter='best',
    random_state=42
) )
]

nn = MLPClassifier(hidden_layer_sizes=(64, 32),  # Two hidden layers: 64 anc
                      activation='relu',           # ReLU activation functic
                      solver='adam',               # Adam optimizer
                      alpha=0.0001,                # Regularization term (L2
                      learning_rate_init=0.001,    # Initial learning rate
                      max_iter=200,                # Maximum number of itera
                      random_state=42)

stacked_model = StackingClassifier(estimators=base_models, final_estimator=r
stacked_model.fit(X_train, y_train)
y_pred = stacked_model.predict(X_val)

score_train = stacked_model.score(X_train, y_train)
score_test = stacked_model.score(X_val, y_val)

score_model(y_val, y_pred, score_train, score_test)
```

```
------------ F1 ------------
[1. CANCELLED]:         0.56
[2. NON-COMP]:          0.91
[3. MED ONLY]:          0.13
[4. TEMPORARY]:         0.8
[5. PPD SCH LOSS]:      0.64
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.4

Macro f1: 0.429

------ Individual Score Comparisons ------
Train Score: 0.7911392086999783
Test Score: 0.7893826070798104
Difference: 0.0017566016201678858
--------- Accuracy ---------

Accuracy Score: 0.7893826070798104

--------- Precision ---------
[1. CANCELLED]:         0.71
[2. NON-COMP]:          0.85
[3. MED ONLY]:          0.58
[4. TEMPORARY]:         0.72
[5. PPD SCH LOSS]:      0.65
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.51

Macro precision: 0.504

---------- Recall ----------
[1. CANCELLED]:         0.46
[2. NON-COMP]:          0.96
[3. MED ONLY]:          0.07
[4. TEMPORARY]:         0.89
[5. PPD SCH LOSS]:      0.63
[6. PPD NSL]:           0.0
[7. PTD]:               0.0
[8. DEATH]:             0.33

Macro recall: 0.418
```

Weighted Averaging

In [27]:
```
# # lr_y_pred_f1   = f1_score(y_val, lr_y_pred, average='macro')
# # dt_y_pred_f1   = f1_score(y_val, dt_y_pred, average='macro')
# # knn_y_pred_f1  = f1_score(y_val, knn_y_pred, average='macro')
# mplc_y_pred_f1 = f1_score(y_val, mplc_y_pred, average='macro')
# # rf_y_pred_f1   = f1_score(y_val, rf_y_pred, average='macro')
# xgb_y_pred_f1  = f1_score(y_val, xgb_y_pred, average='macro')
# gbdt_y_pred_f1 = f1_score(y_val, gbdt_y_pred, average='macro')

# # f1_score(y_actual, y_predicted, average='macro')
```

```
# # Assign weights based on F1 scores
# #weights = [lr_y_pred_f1, dt_y_pred_f1, knn_y_pred_f1, mplc_y_pred_f1, rf_
# weights = [mplc_y_pred_f1, xgb_y_pred_f1, gbdt_y_pred_f1]
# weights = np.array(weights) / np.sum(weights)  # Normalize weights

# # Make weighted predictions
# # lr_probs    = lr_model.predict_proba(X_val)[:, 1]
# # dt_probs    = decision_tree.predict_proba(X_val)[:, 1]
# # knn_probs   = knn_model.predict_proba(X_val)[:, 1]
# mplc_probs  = mlpc_model.predict_proba(X_val)[:, 1]
# # rf_probs    = rf_model.predict_proba(X_val)[:, 1]
# xgb_probs   = xgb_model.predict_proba(X_val)[:, 1]
# gbdt_probs  = gbdt_model.predict_proba(X_val)[:, 1]

# # Aggregate predictions using weights
# weighted_probs = (
#                     # weights[0] * lr_probs +
#                 #   weights[1] * dt_probs +
#                 #   weights[2] * knn_probs +
#                   weights[0] * mplc_probs +
#                 #   weights[4] * rf_probs +
#                   weights[1] * xgb_probs +
#                   weights[2] * gbdt_probs)

# # Final predictions (threshold = 0.5)
# final_predictions = (weighted_probs >= 0.2).astype(int)

# # Evaluate the ensemble
# final_f1 = f1_score(y_val, final_predictions, average='macro')
# print(f"Weighted Ensemble F1 Score: {final_f1:.2f}")
```

5. Kaggle Submission

In [28]:
```
# get the model prediction
# y_pred_test = bagging_model_lr.predict(test_data)
```

In [29]:
```
## decode the prediction labels back to their original values
# decoded_labels = label_encoder.inverse_transform(y_pred_test)
```

In [30]:
```
# # # combine the prediction values with their claim identifiers into a data
# kaggle_submission = pd.DataFrame({"Claim Identifier": test_data.index, "Cl
# kaggle_submission.head()
```

In [31]:
```
# # Compile the resulting dataframe into a csv file named "Kaggle_submission
# # this will be found in the directory the file is currently running from
# # if a file exists with the same name, it will overwrite it with the new c
# kaggle_submission.to_csv("Kaggle_Submission.csv", index=False)
```