Fraudulent Auto Insurance Claim Detection Model

Overview

Accoding to Verisk Analytics, auto insurance fraud is a $29 billion problem. This is a result of omitted or misrepresented underwriting information and criminally inflated claims, leading to inadequate insurance and lower rates. But, there is no such thing as a free lunch. As you can imagine, this means that Insurance Companies are getting scammed out of money, and their customer's wallets are collectively taking the hit. The goal of our model is to predict what auto insurance claims are likely to be overinflated.

The Fraudulent Auto Insurance Claim Detection Model developed in this project could be of great value to any insurance company seeking to probe for and detect fraudulent or inflated insurance claims.

Business Understanding

According to the FBI, the average(and most likely hard working, rule following) American family spends an extra $400 to 700$ on insurance premiums every year because of insurance fraud.

A major insurance company (think All-State, StateFarm, Geico, etc.) approached John and I a few weeks ago to help out their fraudulent claim division. Putting thier customers' needs first, they beleive they can save their company and their customers a substantial dollar amount if they had a better way to detect inflated and fraudlent insurance claims.

There must be something in the air in the "Windy City, becuase Chicago proper is one of our clients most fraudenlt territories in the United States. Before implementing nationally, they want to test a beta model in Illinois to guage efficacy. Utilizing the city of Chicago's transportation data portal, we were able to access information on every single documented car crash. Speficially, we used three sizable dataframes holding information about:

1)The crash itself

2)The people involved

3)The vehicles involved

Data Understanding and Preparation

All the data used was gathered from the city of Chicago's "Chicago Data Portal". In order to get the most relevant data, we isolated the data taken between January of 2017 and January of 2022. We used three dataframes: 1) "Traffic Crashes - Crashes" 2) "Traffic Crashes - People" 3) "Traffic Crashes - Cars"

Raw Data:

Traffic Crashes - Crashes: 617,346 rows × 49 features

Traffic Crashes - People: 777,348 rows × 11 features

Traffic Crashes - Cars: 1,266,486 rows × 72 features

Refined and merged data, before OneHotEncoding: 616067 rows × 41 columns

Our target variable comes from the "Traffic Crashes - Crashes dataset". It was originally called "DAMAGE", and contained information on the cost of damages to the car, which could be one of three categories: "Under 500 dollars"(12 percent), "500-1500 dollars"(28 percent), and "Over 1500 dollars(60 percent)".

In order to make our target binary and more balanced we combined the first two categories, making our new target: "Under 1500 dollars"(40 percent), "Over 1500 dollars"(60 percent).

In [ ]:

```
#import modules

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats as stats

from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV,\
cross_val_score, RandomizedSearchCV
from sklearn.metrics import accuracy_score, recall_score, precision_score
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import roc_auc_score, plot_roc_curve
from sklearn.metrics import log_loss
from sklearn.metrics import make_scorer
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone
from sklearn.dummy import DummyClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.impute import MissingIndicator, SimpleImputer

from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
```

Import, explore, and clean "Crash" Data

In [ ]:
```
#import Crash DataFrame
crash_df = pd.read_csv('data/Traffic_Crashes_-_Crashes.csv')
```

In [ ]:
```
crash_df
```

In [ ]:
```
crash_df.info()
```

In [ ]:
```
crash_df.describe()
```

In [ ]:
```
#Drop Irrelevant columns
crash_df.drop(['RD_NO', 'LANE_CNT','TRAFFIC_CONTROL_DEVICE','DEVICE_COND
```

In [ ]:
```
#crash_df.info()
```

In [ ]:
```
#Fill/Drop relevant nulls
crash_df["INTERSECTION_RELATED_I"].fillna("Unknown", inplace=True)
crash_df["NOT_RIGHT_OF_WAY_I"].fillna("Unknown", inplace=True)
crash_df["HIT_AND_RUN_I"].fillna("Unknown", inplace=True)
crash_df["MOST_SEVERE_INJURY"].fillna("Unknown", inplace=True)
crash_df.dropna(subset=["INJURIES_INCAPACITATING"], inplace=True)
```

In [ ]:
```
#create plot to show distribution of damage categories
sns.histplot(crash_df['DAMAGE'])

```

Import, explore, and clean "People" Data

In [ ]:
```
#import People DataFrame
people_df = pd.read_csv('data/Traffic_Crashes_-_People.csv')
```

In [ ]:
```
#people_df
```

In [ ]:
```
#people_df.info()
```

In [ ]:
```
#Drop irrelevant columns
people_df.drop(['RD_NO', 'CRASH_DATE', 'SEAT_NO','CITY','STATE','ZIPCODE
```

In [ ]:
```
#Remove nulls from relevant rows
people_df.dropna(subset=["VEHICLE_ID"], inplace=True)
people_df.dropna(subset=["SEX"], inplace=True)
people_df.dropna(subset=["SAFETY_EQUIPMENT"], inplace=True)
people_df.dropna(subset=["AIRBAG_DEPLOYED"], inplace=True)
people_df.dropna(subset=["DRIVER_ACTION"], inplace=True)
people_df.dropna(subset=["DRIVER_VISION"], inplace=True)
people_df.dropna(subset=["PHYSICAL_CONDITION"], inplace=True)
people_df.dropna(subset=["AGE"], inplace=True)
```

In [ ]:
```
people_df.info()
```

Import, explore, and clean "Car" Data

In [ ]:
```
car_df = pd.read_csv('data/Traffic_Crashes_-_Vehicles.csv')
```

In [ ]:
```
#car_df
```

In [ ]:
```
#car_df.info()
```

In [ ]:
```
#Create new Car DataFrame with only relevant columns
clean_car_df = car_df[['CRASH_RECORD_ID','UNIT_TYPE','MAKE','MODEL','VEHI
```

In [ ]:
```
#clean_car_df
```

In [ ]:
```
#clean_car_df.info()
```

In [ ]:
```
#drop nulls
clean_car_df.dropna(subset=["UNIT_TYPE"], inplace=True)
clean_car_df.dropna(subset=["MAKE"], inplace=True)
clean_car_df.dropna(subset=["MODEL"], inplace=True)
clean_car_df.dropna(subset=["VEHICLE_YEAR"], inplace=True)
clean_car_df.dropna(subset=["VEHICLE_DEFECT"], inplace=True)
clean_car_df.dropna(subset=["VEHICLE_USE"], inplace=True)
clean_car_df.dropna(subset=["MANEUVER"], inplace=True)
clean_car_df["TOWED_I"].fillna("Unknown", inplace=True)
clean_car_df["EXCEED_SPEED_LIMIT_I"].fillna("Unknown", inplace=True)
```

In [ ]:
```
clean_car_df.info()
```

Merge Crash, People, and Car Data

In [ ]:

```
#merge crash data and people data
crash_people_df = pd.merge(crash_df,people_df, how='left',left_on = 'CRAS

#remove duplicates
crash_people_df.drop_duplicates(subset = 'CRASH_RECORD_ID', inplace = Tru
```

In [ ]:

```
#rename '_merge' column to 'Check', necessary for second merge
crash_people_df.rename(columns = {'_merge':'Check'}, inplace = True)
```

In [ ]:

```
#merge crash and people, and car DataFrames together(CPC)
cpc_df = pd.merge(crash_people_df, clean_car_df, how='left',left_on = 'CI

#drop duplicates
cpc_df.drop_duplicates(subset = 'CRASH_RECORD_ID', inplace = True)
```

Explore and clean new DataFrame

In [ ]:

```
pd.set_option('display.max_columns', None)
```

In [ ]:

```
cpc_df
```

In [ ]:

```
cpc_df.info()
```

We predicted that the make of the car could be would be, to some extent, correlated with the cost of the repairs. You would image the repairs for fender-bender on a Rolls-Royce would be far more expensive than, say, a Toyota.

That being said, we also knew that we would have to OneHotEncode(OHE) every single make(which would've been several hundred new features), so we decided to just OHE the most popular 150 makes.

Further, the Car-Model could've been even more valuable, but without more time we didn't think we could create an efficient model adding that many more features. As you can imgaine, nearly every car model built under the sun was on that list.

In [ ]:

```
#create a new column with only the 150 most occuring "Makes", and an 'Ot
TOP_MAKES = cpc_df['MAKE'].value_counts()
threshold = 150
cpc_df['TOP_MAKES'] = np.where(cpc_df['MAKE'].isin(TOP_MAKES.index[TOP_M
```

In [ ]:

```
#create plot for damage density
damage_density = sns.histplot(crash_df['DAMAGE'], stat = 'density', colo
damage_density.set_xlabel("Repair Cost", fontsize = 15)
damage_density.set_ylabel("Percent of Crashes", fontsize = 15)
damage_density.set_title("Cost Of Repair For Car Crashes", fontsize = 20
```

Here we see an pretty imbalanced distribution within our target feature. In order to make these more even, we decided to combine the two lowest categories into one.

In [ ]:

```
#Use map function to create a binary target column
#This helps to create more balanced dataset
map = {"OVER $1,500":1,"$501 - $1,500": 0, "$500 OR LESS": 0}

cpc_df["Target"] = cpc_df["DAMAGE"].map(map)
```

In [ ]:

```
#check for balanced dataset
#check to see the number of "events" vs "non-events" or most frequent ou
cpc_df["Target"].value_counts(normalize=True)
```

Here, we see that an "event" (1)("over $1,500") occurs about 60% of the time.

In [ ]:

```
#cpc_df.info()
```

In [ ]:

```
#drop irrelevant columns
cpc_df.drop(['PERSON_ID','CRASH_RECORD_ID','DAMAGE','CRASH_DATE','PERSON_
```

In [ ]:

```
#drop nulls
cpc_df.dropna(subset=["SEX"], inplace=True)
cpc_df.dropna(subset=["VEHICLE_YEAR"], inplace=True)
```

In [ ]:

```
cpc_df.info()
```

In [ ]:

```
high_cost_df =  cpc_df[cpc_df['Target'] == 1]
low_cost_df = cpc_df[cpc_df['Target'] == 0]
```

In [ ]:

```
#visualize primary contributing causes
sns.histplot(high_cost_df['PRIM_CONTRIBUTORY_CAUSE'])

```

In [ ]:

```
high_cost_df['PRIM_CONTRIBUTORY_CAUSE'].value_counts()

```

In [ ]:
```
top_5_low = low_cost_df['PRIM_CONTRIBUTORY_CAUSE'].value_counts(normaliz

```

In [ ]:
```
top_5_high = high_cost_df['PRIM_CONTRIBUTORY_CAUSE'].value_counts(normal
top_5_high
```

In [ ]:
```
top_5_high.plot(kind = 'barh', title = "Top 5 Primary Cause for High Cos

```

Looking at the top 5 primary causes for high cost and low cost accidnets.

In [ ]:
```
ax = top_5_high.plot(kind = 'barh', title = "Top 5 Primary Cause for Hig
ax.set_xlabel("Percent of High Cost Accidents")
patches, labels = ax.get_legend_handles_labels()
```

In [ ]:
```
ax = top_5_low.plot(kind = 'barh', title = "Top 5 Primary Cause for Low C
ax.set_xlabel("Percent of Low Cost Accidents")
```

Modeling

Test Train Split

In [ ]:
```
#create a numeric feature dataframe
#perform test train split

numeric_df = cpc_df[['POSTED_SPEED_LIMIT','NUM_UNITS','INJURIES_INCAPACI
                     'CRASH_HOUR','CRASH_DAY_OF_WEEK','CRASH_MONTH','AGE
                     'VEHICLE_YEAR', 'Target']]
X = numeric_df.drop("Target", axis=1)
y = numeric_df["Target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=2(
```

1st Model - "Dummy Model" (Baseline)

This model will predict the most frequent class for every observation. In other words, our model will "guess" the target that occurs most often. This will be a good baseline to compare future models against.

In [ ]:
```
#instantiate dummy model
dummy_model = DummyClassifier(strategy="most_frequent")
```

In [ ]:
```
#fit model
dummy_model.fit(X_train, y_train)
```

In [ ]:
```
dummy_model.predict(y_train)[:50]
```

Here we see that guessing the most frequent event (1) every time, our model will be correct about 60% of the time(as this is the proportion of events(1) to nonevents(0).

In [ ]:
```
#create confusion matrix
plot_confusion_matrix(dummy_model, X_train, y_train)
```

Model Evaluation

Cross-validation will allow us to see how the model would do in generalizing to new data it's never seen.

In [ ]:
```
cv_results = cross_val_score(dummy_model, X_train, y_train, cv=5)
cv_results
```

As we predicted, our model was correct approximately 60% of the time.

To show the spread, we'll make a convenient class that can help us organize the model and the cross-validation:

In [ ]:

```python
class ModelWithCV():
    '''Structure to save the model and more easily see its crossvalidati

    def __init__(self, model, model_name, X, y, cv_now=True):
        self.model = model
        self.name = model_name
        self.X = X
        self.y = y
        # For CV results
        self.cv_results = None
        self.cv_mean = None
        self.cv_median = None
        self.cv_std = None
        #
        if cv_now:
            self.cross_validate()

    def cross_validate(self, X=None, y=None, kfolds=10):
        '''
        Perform cross-validation and return results.

        Args:
          X:
            Optional; Training data to perform CV on. Otherwise use X fr
          y:
            Optional; Training data to perform CV on. Otherwise use y fr
          kfolds:
            Optional; Number of folds for CV (default is 10)
        '''

        cv_X = X if X else self.X
        cv_y = y if y else self.y

        self.cv_results = cross_val_score(self.model, cv_X, cv_y, cv=kfo
        self.cv_mean = np.mean(self.cv_results)
        self.cv_median = np.median(self.cv_results)
        self.cv_std = np.std(self.cv_results)


    def print_cv_summary(self):
        cv_summary = (
        f'''CV Results for `{self.name}` model:
            {self.cv_mean:.5f} ± {self.cv_std:.5f} accuracy
        ''')
        print(cv_summary)


    def plot_cv(self, ax):
        '''
        Plot the cross-validation values using the array of results and 
        Axis for plotting.
        '''
        ax.set_title(f'CV Results for `{self.name}` Model')
        # Thinner violinplot with higher bw
        sns.violinplot(y=self.cv_results, ax=ax, bw=.4)
        sns.swarmplot(
```

```
                y=self.cv_results,
                color='orange',
                size=10,
                alpha= 0.8,
                ax=ax
        )

        return ax
```

In [ ]:

```
dummy_model_results = ModelWithCV(
                        model=dummy_model,
                        model_name='dummy',
                        X=X_train,
                        y=y_train
)
```

In [ ]:

```
fig, ax = plt.subplots()

ax = dummy_model_results.plot_cv(ax)
plt.tight_layout();

dummy_model_results.print_cv_summary()
```

In [ ]:

```
fig, ax = plt.subplots()

fig.suptitle("Dummy Model")

plot_confusion_matrix(dummy_model, X_train, y_train, ax=ax, cmap="plasma
```

In [ ]:

```
from sklearn.metrics import accuracy_score
```

In [ ]:

```
plot_roc_curve(dummy_model, X_train, y_train);
```

2nd Model - Logistic Regression

Next we will create a logistic regression model and compare its performance.

We're going to specifically avoid any regularization (the default) to see how the model does with little change. Set penalty paramter = 'none' = no regularization.

In [ ]:

```
#setting penalty = none means there is no regulaization, and thus we wil
simple_logreg_model = LogisticRegression(random_state=2021, penalty='non
```

In [ ]:

```
#fit model and then predict
simple_logreg_model.fit(X_train, y_train)
```

In [ ]:
```
simple_logreg_model.predict(X_train)[200000:200050]
```

Looking at 50 random samples, we see a mix of events and non-events this time.

2nd Model - Model Evaluation

In [ ]:
```
simple_logreg_results = ModelWithCV(
                        model=simple_logreg_model,
                        model_name='simple_logreg',
                        X=X_train,
                        y=y_train
)
```

In [ ]:
```
# Saving variable for convenience
model_results = simple_logreg_results

# Plot CV results
fig, ax = plt.subplots()
ax = model_results.plot_cv(ax)
plt.tight_layout();
# Print CV results
model_results.print_cv_summary()
```

We see that with no regularization and default parameters, the model performs nearly the same as our basline model.

In [ ]:
```
plot_confusion_matrix(simple_logreg_model, X_train, y_train)
```

In [ ]:
```
fig, ax = plt.subplots()

fig.suptitle("Logistic Regression with Numeric Features Only")

plot_confusion_matrix(simple_logreg_model, X_train, y_train, ax=ax, cmap
```

In [ ]:
```
plot_roc_curve(simple_logreg_model, X_train, y_train);
```

BUT, our ROC has improved. Our ROC curve now has an AUC of 0.56. This is better than our original model, but still not great. We hope by adding in more data preparation and feature engineering we can increase this more.

More Data Preparation

This time we performed a train-test split that contains all of the features.

In [ ]:

```python
X = cpc_df.drop("Target", axis=1)
y = cpc_df["Target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=2(
```

Handling Missing Values

In [ ]:

```python
indicator_demo = MissingIndicator()

indicator_demo.fit(X_train)

indicator_demo.features_
```

In [ ]:

```python
indicator_demo.transform(X_train)[:5, :]
```

In [ ]:

```python
# belowcreates a missing indicator column to help us see if something is
#missing a value for a partiucal
#column, --- NOT NECESSARY

#what is essential !! is an imputer!!
indicator = MissingIndicator(features="all")
indicator.fit(X_train)
```

In [ ]:

```python
def add_missing_indicator_columns(X, indicator):
    """
    Helper function for transforming features

    For every feature in X, create another feature indicating whether tha
    is missing. (This doubles the number of columns in X.)
    """

    # create a 2D array of True and False values indicating whether a giv
    # is missing for that row
    missing_array_bool = indicator.transform(X)

    # transform into 1 and 0 for modeling
    missing_array_int = missing_array_bool.astype(int)

    # helpful for readability but not needed for modeling
    missing_column_names = [col + "_missing" for col in X.columns]

    # convert to df so it we can concat with X
    missing_df = pd.DataFrame(missing_array_int, columns=missing_column_n

    return pd.concat([X, missing_df], axis=1)
```

In [ ]:

```python
X_train = add_missing_indicator_columns(X=X_train, indicator=indicator)
```

In [ ]:
```
X_train.head()
```

In [ ]:
```
#seperate into numeric and categ. features
numeric_feature_names = ['POSTED_SPEED_LIMIT','NUM_UNITS','INJURIES_INCAP
                          'CRASH_HOUR','CRASH_DAY_OF_WEEK','CRASH_MONTH
categorical_feature_names = [c for c in cpc_df.columns if cpc_df[c].dtype

X_train_numeric = X_train[numeric_feature_names]
X_train_categorical = X_train[categorical_feature_names]
```

In [ ]:
```
#imputing numeric columns using the mean for imputing, bc that is the def
numeric_imputer = SimpleImputer()
numeric_imputer.fit(X_train_numeric)
```

In [ ]:
```
categorical_imputer = SimpleImputer(strategy="most_frequent") #here, we
categorical_imputer.fit(X_train_categorical)
```

In [ ]:
```
def impute_missing_values(X, imputer):
    """
    Given a DataFrame and an imputer, use the imputer to fill in all
    missing values in the DataFrame
    """
    imputed_array = imputer.transform(X)
    imputed_df = pd.DataFrame(imputed_array, columns=X.columns, index=X.i
    return imputed_df
```

In [ ]:
```
X_train_numeric = impute_missing_values(X_train_numeric, numeric_imputer
X_train_categorical = impute_missing_values(X_train_categorical, categor
```

In [ ]:
```
X_train_imputed = pd.concat([X_train_numeric, X_train_categorical], axis
X_train_imputed.isna().sum()
```

In [ ]:
```
X_train = X_train.drop(numeric_feature_names + categorical_feature_names
X_train = pd.concat([X_train_imputed, X_train], axis=1)
```

In [ ]:
```
X_train.columns
```

In [ ]:
```
#confirmed there were no null values before OneHotEncoding
X_train.isna().sum()
```

One Hot Encode

In [ ]:
```
X = cpc_df.drop(columns='Target')
y = cpc_df["Target"]

X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.2, r
```

In [ ]:
```
categorical_feature_names = [c for c in cpc_df.columns if cpc_df[c].dtype
numerical_feature_names = ['POSTED_SPEED_LIMIT','NUM_UNITS','INJURIES_INC
                           'CRASH_HOUR','CRASH_DAY_OF_WEEK','CRASH_MONTH
```

In [ ]:
```

def encode_and_concat_feature_train(X_train, feature_name):
    """
    Helper function for transforming training data.  It takes in the full
    feature name, makes a one-hot encoder, and returns the encoder as wel
    with that feature transformed into multiple columns of 1s and 0s
    """
    # make a one-hot encoder and fit it to the training data
    ohe = OneHotEncoder(categories="auto", handle_unknown="ignore")
    single_feature_df = X_train[[feature_name]]
    ohe.fit(single_feature_df)

    # call helper function that actually encodes the feature and concats
    X_train = encode_and_concat_feature(X_train, feature_name, ohe)

    return ohe, X_train
```

In [ ]:
```

def encode_and_concat_feature(X, feature_name, ohe):
    """
    Helper function for transforming a feature into multiple columns of 1
    in both training and testing steps.  Takes in the full X dataframe, f
    and encoder, and returns the dataframe with that feature transformed
    columns of 1s and 0s
    """
    # create new one-hot encoded df based on the feature
    single_feature_df = X[[feature_name]]
    feature_array = ohe.transform(single_feature_df).toarray()
    ohe_df = pd.DataFrame(feature_array, columns=ohe.categories_[0], inde

    # drop the old feature from X and concat the new one-hot encoded df
    X = X.drop(feature_name, axis=1)
    X = pd.concat([X, ohe_df], axis=1)

    return X
```

In [ ]:
```
encoders = {}

for categorical_feature in categorical_feature_names:
    ohe, X_train = encode_and_concat_feature_train(X_train, categorical_f
    encoders[categorical_feature] = ohe
```

In [ ]:

```
encoders
```

In [ ]:

```
X_train.head()
```

In [ ]:

```
X_train.shape
```

Decision Tree - For Feature Importance

In [ ]:

```
#Instatiate Decision Tree
dt = DecisionTreeClassifier(max_depth=13, random_state=42)

dt.fit(X_train, y_train)

CV_results = cross_val_score(dt,X_train,y_train,cv=5)
CV_results
```

In [ ]:

```
plot_confusion_matrix(dt,X_train,y_train)

```

In [ ]:

```
#create dictionary of feature importance
list = {}
for fi, feature in zip(dt.feature_importances_,X_train):
    list.update({fi:feature})
```

In [ ]:

```
#Order by most important
import collections
od = collections.OrderedDict(sorted(list.items(),reverse=True))
od
```

In [ ]:

```
#visualize
n_features = dt.n_features_
plt.figure(figsize=(15, 70))
plt.barh(range(n_features), dt.feature_importances_);
plt.yticks(np.arange(n_features), X_train.columns.values, fontsize = 12)
plt.xlabel('Feature importance', fontsize = 20)
plt.ylabel('Features', fontsize = 20)
plt.title('FSM Feature Importance', fontsize = 20)
plt.tight_layout()

```

With more time, we would impute all of our "unknown" data and determine featuer importance again. Based on the results, we would remove the the unimportant features and focus on the most important ones.

"3rd Model"

In [ ]:
```
logreg_model = LogisticRegression(random_state=2021, penalty='none')
logreg_model.fit(X_train, y_train)
```

In [ ]:
```
#more iterations
logreg_model_more_iterations = LogisticRegression(
                                                random_state=2021,
                                                penalty='none',
                                                max_iter=100
)
logreg_model_more_iterations.fit(X_train, y_train)
```

In [ ]:
```
#higher tolerance (C-parameter is inverse of regularization strength)
#higher tolerance means that our models will stop training earlier (when
#true values are not as close as they could be).
logreg_model_higher_tolerance = LogisticRegression(
                                                random_state=2021,
                                                penalty='none',
                                                tol=25
)
logreg_model_higher_tolerance.fit(X_train, y_train)
```

3rd Model - Model Evaluations

In [ ]:
```
fix, axes = plt.subplots(nrows=1, ncols=2, figsize=(15, 6))

axes[0].set_title("More Iterations")
axes[1].set_title("Higher Tolerance")

plot_confusion_matrix(logreg_model_more_iterations, X_train, y_train,
                      ax=axes[0], cmap="plasma")
plot_confusion_matrix(logreg_model_higher_tolerance, X_train, y_train,
                      ax=axes[1], cmap="plasma");
```

In [ ]:

```
logreg_model_more_iterations_results = ModelWithCV(
                                        logreg_model_more_iterations,
                                        'more_iterations',
                                        X_train,
                                        y_train
)

logreg_model_higher_tolerance_results = ModelWithCV(
                                        logreg_model_higher_tolerance,
                                        'higher_tolerance',
                                        X_train,
                                        y_train
)

model_results = [
    logreg_model_more_iterations_results,
    logreg_model_higher_tolerance_results
]
```

In [ ]:

```
f,axes = plt.subplots(ncols=2, sharey=True, figsize=(12, 6))

for ax, result in zip(axes, model_results):
    ax = result.plot_cv(ax)
    result.print_cv_summary()
plt.tight_layout();
```

Here we see a slight improvement from our previous scores.

In [ ]:

```
fig, ax = plt.subplots()

plot_roc_curve(logreg_model_more_iterations, X_train, y_train,
               name='logreg_model_more_iterations', ax=ax)
plot_roc_curve(logreg_model_higher_tolerance, X_train, y_train,
               name='logreg_model_higher_tolerance', ax=ax);
```

Here, we see a major improvememnt! Could be result of overfitting.

4th Model - After Scaling

More Data Preparation - Scaling

In [ ]:

```
scaler = StandardScaler()

scaler.fit(X_train)
```

In [ ]:

```
def scale_values(X, scaler):
    """
    Given a DataFrame and a fitted scaler, use the scaler to scale all o
    """
    scaled_array = scaler.transform(X)
    scaled_df = pd.DataFrame(scaled_array, columns=X.columns, index=X.in
    return scaled_df
```

In [ ]:

```
X_train = scale_values(X_train, scaler)
```

In [ ]:

```
X_train.head()
```

Now that we have scaled data, lets see how well our logistic regression model fits without adjusting any hyperparameters.

In [ ]:

```
logreg_model = LogisticRegression(random_state=2021)
logreg_model.fit(X_train, y_train)
```

In [ ]:

```
fig, ax = plt.subplots()

fig.suptitle("Logistic Regression with All Features, Scaled")

plot_confusion_matrix(logreg_model, X_train, y_train, ax=ax, cmap="plasm
```

In [ ]:

```
all_features_results = ModelWithCV(
                            logreg_model,
                            'all_features',
                            X_train,
                            y_train
)
```

In [ ]:

```
# Saving variable for convenience
model_results = all_features_results

# Plot CV results
fig, ax = plt.subplots()
ax = model_results.plot_cv(ax)
plt.tight_layout();
# Print CV results
model_results.print_cv_summary()
```

We see that scaling improved our accuracy scores. We also see below that the AUC increased slightly.

In [ ]:

```
plot_roc_curve(logreg_model, X_train, y_train)
```

In [ ]:
```
# sorted(list(zip(X_train.columns, logreg_model.coef_[0])),
#        key=lambda x: abs(x[1]), reverse=True)[:50]
```

In [ ]:
```
#so now lets increase the regularization - the correct the overfitting
```

Hyperparameter Adjustment

Different Regularization Strengths

In [ ]:
```
all_features_results.print_cv_summary()
```

In [ ]:
```
model_results = [all_features_results]
C_values = [0.0001, 0.001, 0.01, 0.1, 1]

for c in C_values:
    logreg_model = LogisticRegression(random_state=2021, C=c)
    logreg_model.fit(X_train, y_train)
    # Save Results
    new_model_results = ModelWithCV(
                            logreg_model,
                            f'all_features_c{c:e}',
                            X_train,
                            y_train
    )
    model_results.append(new_model_results)
    new_model_results.print_cv_summary()
```

Here, we don't see any any significant improvement in accuracy with C-values.

In [ ]:
```
f,axes = plt.subplots(ncols=3, nrows=2, sharey='all', figsize=(18, 12))

for ax,result in zip(axes.ravel(),model_results):
    ax = result.plot_cv(ax)

plt.tight_layout();
```

In [ ]:
```
model_results = [all_features_results]
all_features_cross_val_score = all_features_results.cv_results
```

Different Solvers

In [ ]:
```
model_results = [all_features_results]
all_features_cross_val_score = all_features_results.cv_results
```

In [ ]:

```
ogreg_model = LogisticRegression(random_state=2021, solver="liblinear")
logreg_model.fit(X_train, y_train)
```

In [ ]:

```
# Save for later comparison
model_results.append(
    ModelWithCV(
        logreg_model,
        'solver:liblinear',
        X_train,
        y_train
    )
)

# Plot both all_features vs new model
f,axes = plt.subplots(ncols=2, sharey='all', figsize=(12, 6))

model_results[0].plot_cv(ax=axes[0])
model_results[-1].plot_cv(ax=axes[1])

plt.tight_layout();
```

In [ ]:

```
print("Old:", all_features_cross_val_score)
print("New:", model_results[-1].cv_results)
```

No major difference in the scores. Let's try adding some more regularization:

In [ ]:

```
logreg_model = LogisticRegression(random_state=2021, solver="liblinear",
logreg_model.fit(X_train, y_train)
```

In [ ]:

```
# Save for later comparison
model_results.append(
    ModelWithCV(
        logreg_model,
        'solver:liblinear_C:0.01',
        X_train,
        y_train
    )
)

# Plot both all_features vs new model
f,axes = plt.subplots(ncols=2, sharey='all', figsize=(12, 6))

model_results[0].plot_cv(ax=axes[0])
model_results[-1].plot_cv(ax=axes[1])

plt.tight_layout();
```

In [ ]:

```
print("Old:", all_features_cross_val_score)
print("New:", model_results[-1].cv_results)
```

Slightly better, if any. Lets try another different type of penalty.

In [ ]:
```
logreg_model = LogisticRegression(random_state=2021, solver="liblinear",
logreg_model.fit(X_train, y_train)
```

Type *Markdown* and LaTeX: α^2

In [ ]:
```
#Save for later comparison
# model_results.append(
#     ModelWithCV(
#         logreg_model,
#         'solver:liblinear_penalty:l1',
#         X_train,
#         y_train
#     )
# )

# # Plot both all_features vs new model
# f,axes = plt.subplots(ncols=2, sharey='all', figsize=(12, 6))

# model_results[0].plot_cv(ax=axes[0])
# model_results[-1].plot_cv(ax=axes[1])

# plt.tight_layout();
```

In [ ]:
```
print("Old:", all_features_cross_val_score)
print("New:", model_results[-1].cv_results)
```

This took too long to run.

In [ ]:
```
logreg_model = LogisticRegression(random_state=2021, solver="liblinear",
logreg_model.fit(X_train, y_train)
```

In [ ]:
```
# Save for later comparison
model_results.append(
    ModelWithCV(
        logreg_model,
        'solver:liblinear_penalty:l1_C:0.01',
        X_train,
        y_train
    )
)

# Plot both all_features vs new model
f,axes = plt.subplots(ncols=2, sharey='all', figsize=(12, 6))

model_results[0].plot_cv(ax=axes[0])
model_results[-1].plot_cv(ax=axes[1])

plt.tight_layout();
```

In [ ]:
```
print("Old:", all_features_cross_val_score)
print("New:", model_results[-1].cv_results)
```

In [ ]:
```
logreg_model = LogisticRegression(random_state=2021, solver="liblinear",
logreg_model.fit(X_train, y_train)

fig, ax = plt.subplots()

fig.suptitle("Logistic Regression with All Features (Scaled, Hyperparame

plot_confusion_matrix(logreg_model, X_train, y_train, ax=ax, cmap="plasm
```

Very Similar to our previous models scores.

As we said previously, our model could be overfitting. One way to address is this is to remove features, specifically, ones that have small modeling coefficients. We did this using SelectFromModel.

SelectFromModel

In [ ]:
```
selector = SelectFromModel(logreg_model)

selector.fit(X_train, y_train)
```

In [ ]:
```
#use a default threshold
thresh = selector.threshold_
thresh
```

In [ ]:
```
#Checking to see how many features will be eliminated
coefs = selector.estimator_.coef_
coefs
```

In [ ]:
```
coefs.shape
```

In [ ]:
```
coefs[coefs > thresh].shape
```

In [ ]:
```
selector.get_support()
```

In [ ]:
```
dict(zip(X_train.columns, selector.get_support()))
```

In [ ]:
```
def select_important_features(X, selector):
    """
    Given a DataFrame and a selector, use the selector to choose
    the most important columns
    """
    imps = dict(zip(X.columns, selector.get_support()))
    selected_array = selector.transform(X)
    selected_df = pd.DataFrame(selected_array,
                               columns=[col for col in X.columns if imps
                               index=X.index)
    return selected_df
```

In [ ]:
```
X_train_selected = select_important_features(X=X_train, selector=selecto
```

In [ ]:
```
X_train_selected.head()
```

In [ ]:
```
logreg_sel = LogisticRegression(random_state=2021, solver="liblinear", p

logreg_sel.fit(X_train_selected, y_train)
```

In [ ]:
```
# Save for later comparison
# select_results = ModelWithCV(
#                      logreg_sel,
#                      'logreg_sel',
#                      X_train_selected,
#                      y_train
# )

# Plot both all_features vs new model
#f,axes = plt.subplots(ncols=2, sharey='all', figsize=(12, 6))

# model_results[0].plot_cv(ax=axes[0])
# select_results.plot_cv(ax=axes[1])

#plt.tight_layout();
```

In [ ]:
```
# print("Old:", all_features_cross_val_score)
# print("New:", select_results.cv_results)
```

Unfortunately, our final two models were taking too long to run. My kernal kept stopping. So we were not able to get our final models or run a final model evaluation at this time.

With more time, there is a lot more I would have liked to do. For starters, there were alot of "unknown"s in our data. I think that running an imputer to impute data into those features could've been very helpful. As seen, the "Unknowns" were ranked among the most important features. From this, we could then run though a decision tree again to find the most important features, allowing us to eliminate the unimportant or overinflating ones, and assigning proper weight to the important ones. I beleive doing all of this would've given us better results on our test.

Final Model Evaluation

Now that we have a final model, run X_test through all of the preprocessing steps so we can evaluate the model's performance

In [ ]:

```
# X_test_no_transformations = X_test.copy()
```

In [ ]:

```
# add missing indicators
# X_test_mi = add_missing_indicator_columns(X_test_no_transformations, i
```

In [ ]:

```
# separate out values for imputation
# X_test_numeric = X_test_mi[numeric_feature_names]
# X_test_categorical = X_test_mi[categorical_feature_names]
```

In [ ]:

```
# separate out values for imputation
# impute missing values
# X_test_numeric = impute_missing_values(X_test_numeric, numeric_imputer,
# X_test_categorical = impute_missing_values(X_test_categorical, categor
# X_test_imputed = pd.concat([X_test_numeric, X_test_categorical], axis=1
# X_test_new = X_test_mi.drop(numeric_feature_names + categorical_feature
# X_test_final = pd.concat([X_test_imputed, X_test_new], axis=1)
```

In [ ]:

```
# one-hot encode categorical data
# for categorical_feature in categorical_feature_names:
#     X_test_final = encode_and_concat_feature(X_test_final,
#                                                  categorical_feature, encoders[c
```

In [ ]:

```
# # scale values
# X_test_scaled = scale_values(X_test_final, scaler)
```

In [ ]:

```
# select features
# X_test_selected = select_important_features(X_test_scaled, selector)
```

In [ ]:

```
# X_test_selected.head()
```

In [ ]:

```
# final_model = LogisticRegression(random_state=2021, solver="liblinear",
# final_model.fit(X_train_selected, y_train)

# final_model.score(X_test_selected, y_test)
```

Compare the past models

In [ ]:

```
# Create a way to categorize our different models
# model_candidates = [
#     {
#         'name':'dummy_model'
#         ,'model':dummy_model
#         ,'X_test':X_test
#         ,'y_test':y_test
#     },
#     {
#         'name':'simple_logreg_model'
#         ,'model':simple_logreg_model
#         ,'X_test':X_test_no_transformations[["SibSp", "Parch", "Fare"]]
#         ,'y_test':y_test
#     },
#     {
#         'name':'logreg_model_more_iterations'
#         ,'model':logreg_model_more_iterations
#         ,'X_test':X_test_final
#         ,'y_test':y_test
#     },
#     {
#         'name':'logreg_model_higher_tolerance'
#         ,'model':logreg_model_higher_tolerance
#         ,'X_test':X_test_final
#         ,'y_test':y_test
#     },
#     {
#         'name':'final_model'
#         ,'model':final_model
#         ,'X_test':X_test_selected
#         ,'y_test':y_test
#     }
# ]
```

In [ ]:

```
# final_scores_dict = {
#     "Model Name": [candidate.get('name') for candidate in model_candid
#     "Mean Accuracy": [
#         candidate.get('model').score(
#                                 candidate.get('X_test'),
#                                 candidate.get('y_test')
#         )
#         for candidate in model_candidates
#     ]

# }
# final_scores_df = pd.DataFrame(final_scores_dict).set_index('Model Name
# final_scores_df
```

In [ ]:

```
# nrows = 2
# ncols = math.ceil(len(model_candidates)/nrows)

# fig, axes = plt.subplots(
#                  nrows=nrows,
#                  ncols=ncols,
#                  figsize=(12, 6)
# )
# fig.suptitle("Confusion Matrix Comparison")

# # Turn off all the axes (in case nothing to plot); turn on while itera
# [ax.axis('off') for ax in axes.ravel()]


# for i,candidate in enumerate(model_candidates):
#     # Logic for making rows and columns for matrices
#     row = i // 3
#     col = i % 3
#     ax = axes[row][col]

#     ax.set_title(candidate.get('name'))
#     ax.set_axis_on()
#     cm_display = plot_confusion_matrix(
#                      candidate.get('model'),
#                      candidate.get('X_test'),
#                      candidate.get('y_test'),
#                      normalize='true',
#                      cmap='plasma',
#                      ax=ax,

#     )
#     cm_display.im_.set_clim(0, 1)

# plt.tight_layout()
```

In [ ]:

```
# fig, ax = plt.subplots()

# # Plot only the last models we created (so it's not too cluttered)
# for model_candidate in model_candidates[3:]:
#     plot_roc_curve(
#         model_candidate.get('model'),
#         model_candidate.get('X_test'),
#         model_candidate.get('y_test'),
#         name=model_candidate.get('name'),
#         ax=ax
#     )
```

In [ ]:

```
# fig, ax = plt.subplots()

# # Plot the final model against the other earlier models
# plot_roc_curve(
#     final_model,
#     X_test_selected,
#     y_test,
#     name='final_model',
#     ax=ax
# )

# for model_candidate in model_candidates[:3]:
#     plot_roc_curve(
#         model_candidate.get('model'),
#         model_candidate.get('X_test'),
#         model_candidate.get('y_test'),
#         name=model_candidate.get('name'),
#         ax=ax
#     )
```