Final Project Report

Auto Insurance Fraud Prediction

ALY6015 Intermediate to Analytics

NORTHEASTERN UNIVERSITY

Ankit Vilas Bhalekar
&
Lazaree Worlikar

DATE OF SUBMISSION: 05-16-2023

Introduction:

In this document, a Python code snippet's examination with an emphasis on data manipulation, computation, and data visualization is presented. To preprocess data, execute classification using logistic regression and random forest models, and assess these models' efficacy in predicting fraud, the code makes use of a variety of modules and approaches. Pandas, Numpy, Matplotlib, Seaborn, Plotly Express, and Warnings are just a few of the imported libraries that offer functionality for handling warnings, data processing, numerical calculations, and data visualization.

The code uses Scikit-Learn's LogisticRegression to build a 200-iteration maximum logistic regression model. The best regularization penalties (L1 and L2), regularization strengths, and class weights for the dataset are found using randomized search cross-validation (RandomizedSearchCV). Additionally, a randomized search is utilized to adjust the hyperparameters of the random forest classifier. The hyperparameters of the random forest classifier are defined by a dictionary in the code, and the best hyperparameters are determined through cross-validation with RandomizedSearchCV.

Various methods for data processing, mathematical calculations, and data visualization are also demonstrated by the code. These include clearing out unused columns, choosing columns depending on particular data types, replacing missing values with placeholders, scaling numerical data with Pandas and Scikit-Learn's StandardScaler, and performing one-hot encoding and mapping on categorical features. The paper also discusses techniques for numerical feature analysis using visualization.

The offered Python code serves as an example of how to perform standard data preprocessing, train classification models while modifying their hyperparameters, and evaluate the results using several classification metrics. The methods used are designed to improve the models' fraud prediction ability, and the libraries used have powerful data analysis capabilities.

Analysis:

Data analysis and manipulation in the code are performed using the Pandas library, while mathematical calculations and array operations rely on the Numpy package. Data visualization is carried out using the Matplotlib toolkit, as well as the Seaborn and Plotly Express libraries. The Warnings library is imported to manage warning messages.

The logistic regression model's hyperparameters are adjusted using randomized search cross-validation, exploring different regularization penalties, regularization strengths, and class weights. The random forest classifier's hyperparameters are also tuned using a randomized search process. The code defines dictionaries containing potential values for the hyperparameters, and cross-validation is employed to identify the optimal values.

The code defines a dictionary of classifiers, iterating through them and calling a function called "scores()" for evaluation. However, the implementation of this function is absent from the provided code snippet, making it impossible to assess its usefulness and the final outcome.

In [3]:
```python
#IMPORTING THE DATASET
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

import warnings
warnings.filterwarnings('ignore')

pd.set_option('display.max_columns', None)
plt.style.use('ggplot')
```

In [ ]:
```python
df = pd.read_csv('insurance_claims.csv')
```

In [23]:
```python
df.head()
```

Out[23]:

	months_as_customer	age	policy_number	policy_bind_date	policy_state	policy_csl	policy_deductable	policy_annual_premium	umbrella_limit	insured_
0	328	48	521585	10/17/2014	OH	250/500	1000	1406.91	0	466
1	228	42	342868	6/27/2006	IN	250/500	2000	1197.22	5000000	468
2	134	29	687698	9/6/2000	OH	100/300	2000	1413.14	5000000	430
3	256	41	227811	5/25/1990	IL	250/500	2000	1415.74	6000000	608
4	228	44	367455	6/6/2014	IL	500/1000	1000	1583.91	6000000	610

Pandas Library:

Pandas library is used for data manipulation and analysis. The code imports the pandas library using the following statement: "import pandas as pd".

Numpy Library:

Numpy library is used for working with arrays and numerical calculations. The code imports the numpy library using the following statement: "import numpy as np".

Matplotlib Library:

Matplotlib library is used for data visualization. The code imports the matplotlib library using the following statement: "import matplotlib.pyplot as plt".

Seaborn Library:

Seaborn library is used for data visualization. The code imports the seaborn library using the following statement: "import seaborn as sns".

Plotly Express Library:

Plotly Express library is used for data visualization. The code imports the plotly express library using the following statement: "import plotly.express as px".

Warnings Library:

Warnings library is used to handle warnings. The code imports the warnings library using the following statement: "import warnings".

In [24]:
```
df.shape
```
Out[24]: (1000, 40)

In [29]:
```
#We can see that there are few ? in collision_type column...let's replace all the ? by Nan which indicates missing values
df.replace('?', np.nan, inplace = True)
```

In [30]:
```
# Count NaN values in multiple columns of DataFrame
nan_count = df.isna().sum()
print(nan_count)
```

```
months_as_customer             0
age                            0
policy_number                  0
policy_bind_date               0
policy_state                   0
policy_csl                     0
policy_deductable              0
policy_annual_premium          0
umbrella_limit                 0
insured_zip                    0
insured_sex                    0
insured_education_level        0
insured_occupation             0
insured_hobbies                0
insured_relationship           0
capital-gains                  0
capital-loss                   0
incident_date                  0
incident_type                  0
```

In [36]:
```
percent_missing = df.isnull().sum() * 100 / len(df)
missing_value_df = pd.DataFrame({'column_name': df.columns,
                                 'percent_missing': percent_missing})


missing_value_df.sort_values('percent_missing', ascending=False, inplace=True)
```

In [37]:
```
missing_value_df
```
Out[37]:

	column_name	percent_missing
_c39	_c39	100.0
property_damage	property_damage	36.0
police_report_available	police_report_available	34.3
collision_type	collision_type	17.8
bodily_injuries	bodily_injuries	0.0
incident_state	incident_state	0.0
incident_city	incident_city	0.0
incident_location	incident_location	0.0
incident_hour_of_the_day	incident_hour_of_the_day	0.0
number_of_vehicles_involved	number_of_vehicles_involved	0.0
fraud_reported	fraud_reported	0.0
auto_year	auto_year	0.0
auto_model	auto_model	0.0
authorities_contacted	authorities_contacted	0.0
total_claim_amount	total_claim_amount	0.0
injury_claim	injury_claim	0.0

In [38]:
```
#Let's drop the _c39 column because all the values are missing in that column
# Remove column name 'A'
df.drop(['_c39'], axis=1,inplace=True)
```

In [39]:
```
df.head()
```
Out[39]:

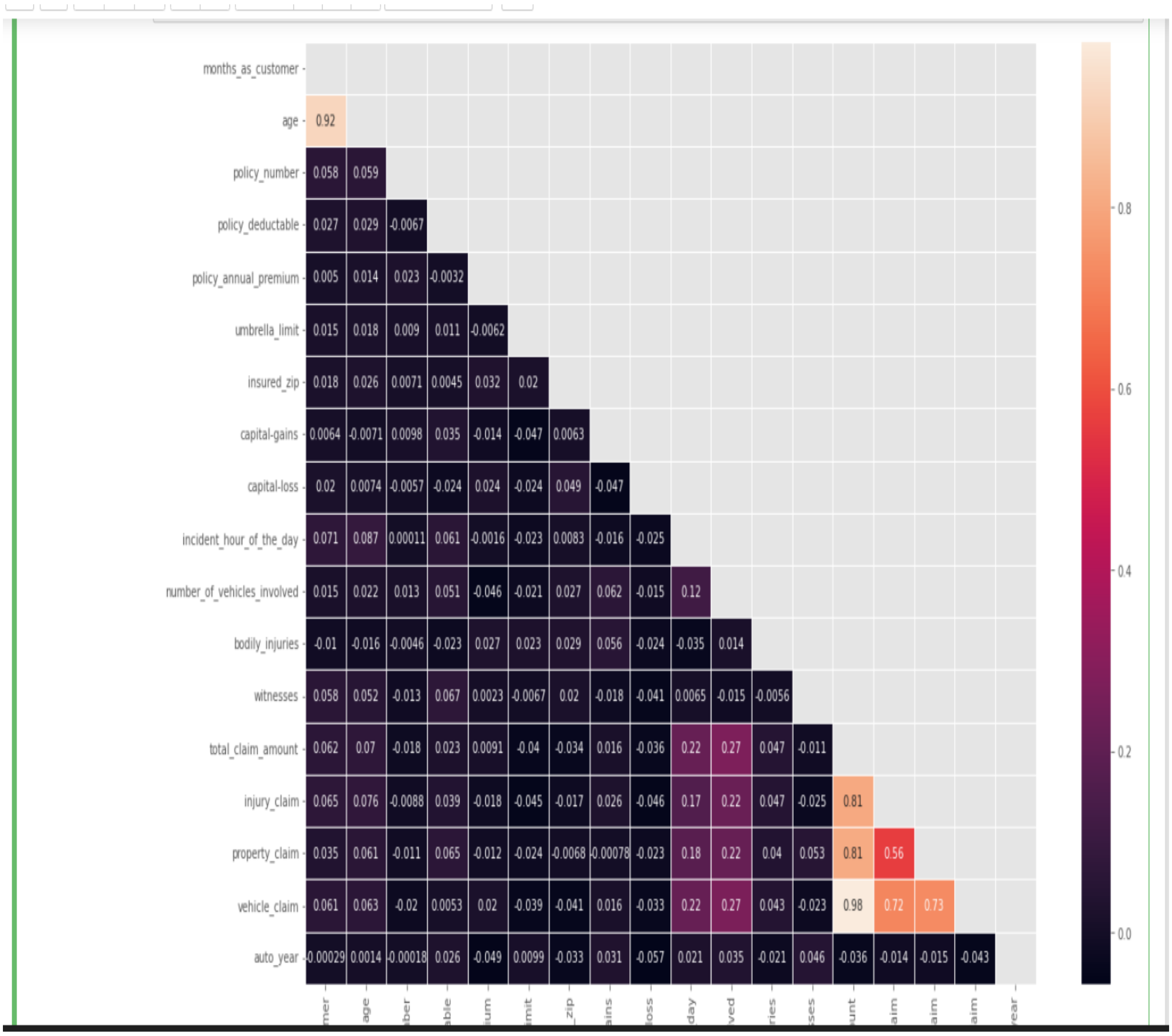
nths_as_customer	age	policy_number	policy_bind_date	policy_state	policy_csl	policy_deductable	policy_annual_premium	umbrella_limit	insured_zip	ins
328	48	521585	10/17/2014	OH	250/500	1000	1406.91	0	466132	
228	42	342868	6/27/2006	IN	250/500	2000	1197.22	5000000	468176	
134	29	687698	9/6/2000	OH	100/300	2000	1413.14	5000000	430632	
256	41	227811	5/25/1990	IL	250/500	2000	1415.74	6000000	608117	
228	44	367455	6/6/2014	IL	500/1000	1000	1583.91	6000000	610706	

In [40]:
```
df.describe()
```
Out[40]:

	months_as_customer	age	policy_number	policy_deductable	policy_annual_premium	umbrella_limit	insured_zip	capital-gains	capi
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1.000000e+03	1000.000000	1000.000000	1000.
mean	203.954000	38.948000	546238.648000	1136.000000	1256.406150	1.101000e+06	501214.488000	25126.100000	-26793.
std	115.113174	9.140287	257063.005276	611.864673	244.167395	2.297407e+06	71701.610941	27872.187708	28104.
min	0.000000	19.000000	100804.000000	500.000000	433.330000	-1.000000e+06	430104.000000	0.000000	-111100.
25%	115.750000	32.000000	335980.250000	500.000000	1089.607500	0.000000e+00	448404.500000	0.000000	-51500.
50%	199.500000	38.000000	533135.000000	1000.000000	1257.200000	0.000000e+00	466445.500000	0.000000	-23250.
75%	276.250000	44.000000	759099.750000	2000.000000	1415.695000	0.000000e+00	603251.000000	51025.000000	0.

```
In [41]: #Replace the missing values with their respective modes
df['collision_type'] = df['collision_type'].fillna(df['collision_type'].mode()[0])
df['property_damage'] = df['property_damage'].fillna(df['property_damage'].mode()[0])
df['police_report_available'] = df['police_report_available'].fillna(df['police_report_available'].mode()[0])
```

```
In [43]: # checking for multicollinearity

plt.figure(figsize = (18, 12))

corr = df.corr()
mask = np.triu(np.ones_like(corr, dtype = bool))

sns.heatmap(data = corr, mask = mask, annot = True, fmt = '.2g', linewidth = 1)
plt.show()
```



module for Python Visuals that are interactive are created using Plotly. The interactive visuals in this code are provided by Plotly.

In [44]:
```python
df.drop(columns = ['age', 'total_claim_amount'], inplace = True, axis = 1)
df.head()
```

Out[44]:

	months_as_customer	policy_number	policy_bind_date	policy_state	policy_csl	policy_deductable	policy_annual_premium	umbrella_limit	insured_zip	i
0	328	521585	10/17/2014	OH	250/500	1000	1406.91	0	466132	
1	228	342868	6/27/2006	IN	250/500	2000	1197.22	5000000	468176	
2	134	687698	9/6/2000	OH	100/300	2000	1413.14	5000000	430632	
3	256	227811	5/25/1990	IL	250/500	2000	1415.74	6000000	608117	
4	228	367455	6/6/2014	IL	500/1000	1000	1583.91	6000000	610706	

In [45]:
```python
# separating the feature and target columns

X = df.drop('fraud_reported', axis = 1)
y = df['fraud_reported']
```

In [46]:
```python
y = y.map({'Y': 1, 'N': 0})
```

In [49]:
```python
y.value_counts(normalize=True)*100
```

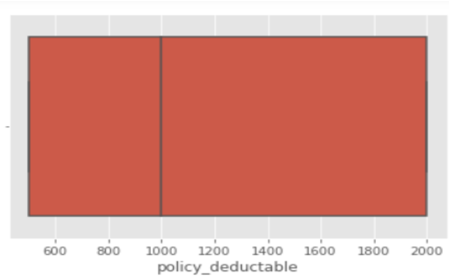
Out[49]:
```
0    75.3
1    24.7
Name: fraud_reported, dtype: float64
```

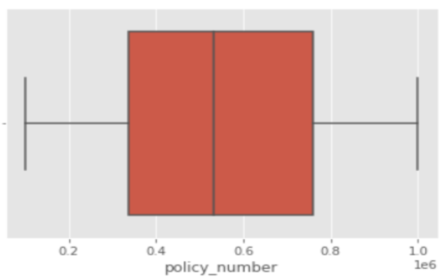
In [50]:
```python
df_numerical_features = df.select_dtypes(exclude='object')
df_categorical_features = df.select_dtypes(include='object')
```

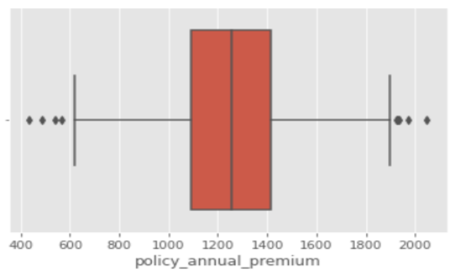
In [53]:
```python
for column in df_numerical_features.columns:
    sns.distplot(df_numerical_features[column])

    plt.show()
```

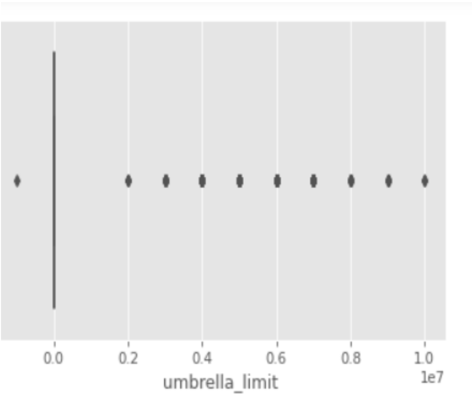
...

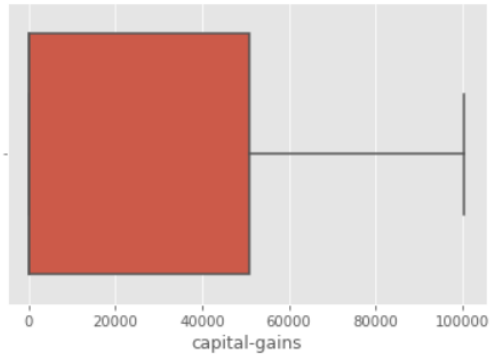
In [56]:
```python
for column in df_numerical_features.columns:
     sns.boxplot(df_numerical_features[column])

     plt.show()
```

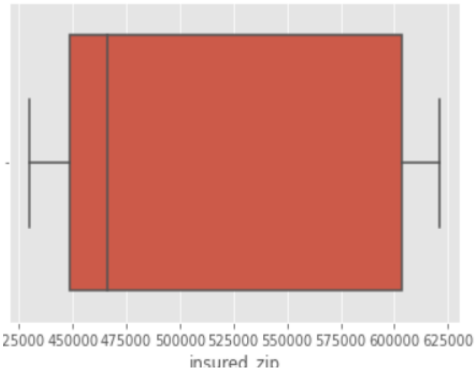


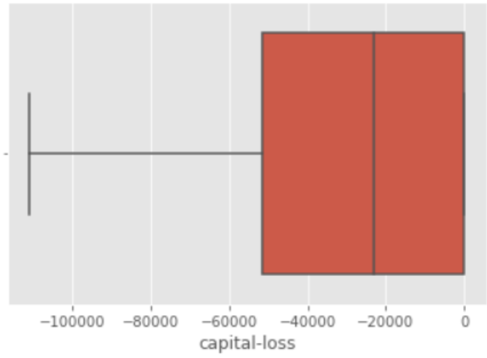













```
In [12]:  #Data preprocessing
          from sklearn.preprocessing import StandardScaler

          def scale_and_encode(df):
              # Split columns into numerical and categorical
              num_cols = df.select_dtypes(include=['int64', 'float64']).columns.tolist()
              cat_cols = df.select_dtypes(include=['object']).columns.tolist()

              # Standardize numerical columns
              scaler = StandardScaler()
              df[num_cols] = scaler.fit_transform(df[num_cols])

              # One-hot encode categorical columns
              df = pd.get_dummies(df, columns=cat_cols,drop_first = True)

              return df
```

The StandardScaler and pd.get_dummies functions from the scikit-learn and pandas libraries, respectively, are used by this Python data preprocessing function to scale numerical columns and one-hot encode categorical columns in a pandas DataFrame.

The function creates two lists, one for categorical data and the other for numerical data, from the columns in a DataFrame (df). It uses the StandardScaler function to normalize the numerical columns after deducting the mean and scaling to unit variance. Then, one-hot encoding is applied to the categorical columns, creating new binary columns for each unique category in the original column. The drop_first option is set to True to remove the first binary column and prevent multicollinearity.

The preprocessed DataFrame has one-hot encoded categorical columns and standardized number columns.

Methods used along with justification for those methods:

Python programming uses a range of modules and techniques for data processing, mathematical calculations, and data visualization. With the help of Pandas, Numpy, Matplotlib, Seaborn, and Plotly Express, various visualizations can be created. The code also takes use of the Warnings module to handle warning messages.

Pandas and the StandardScaler function from Scikit-Learn are used to scale numerical data.When using the DataFrame.replace method, missing values are represented by NaN values in the 'collision_type' column. Pandas.The DataFrame.isna method in Pandas is used to count the amount of missing values in the dataframe.The DataFrame.fillna function is used to replace missing values with the relevant modes. Pandas.DataFrame.drop is used to remove the '_c39' column from the dataframe.

Use the DataFrame.select_dtypes method to select columns that have a specific data type and support Pandas.Using the DataFrame.map function, the 'fraud_reported' column's values are translated to 1 and 0. The Pandas are used to do one-hot encoding on the category characteristics.the get_dummies function for DataFrame.

The Seaborn.distplot and Seaborn.boxplot techniques are used to plot the distribution of numerical features.

Overall, the Python code uses a combination of data manipulation, numerical computations, and data visualization approaches to preprocess the data and produce visualizations for improved insights. The data science community is familiar with the libraries that were used to develop this code, and they are essential tools for data analysis.

LOGISTIC REGRESSION

In [4]:

```
lr = LogisticRegression(max_iter=200)


lr_values = {'solver': ['liblinear'],
             'penalty': ['l1', 'l2'],
             'C': np.logspace(-5, 5, 50),
             'class_weight':[{0:0.246667, 1:0.75333},None]}



rs_lr = RandomizedSearchCV(lr, lr_values, cv=10, n_jobs = -1, random_state=42)
rs_lr.fit(X_train, y_train, )
print(rs_lr.best_params_)
```

The presented code combines hyperparameter adjustment with logistic regression using randomized search cross-validation (RandomizedSearchCV) and scikit-learn's LogisticRegression. Here is how the code was assessed:

The maximum number of iterations for the logistic regression model is 200. This value can be changed in accordance with the model's convergence during training.

Randomized search hyperparameter space: the solver parameter identifies the optimization algorithm. 'Liblinear' is used in this instance since it is best suited for small to medium-sized datasets.

punishment details the kind of regularization penalty. The program takes into account both L1 and L2 regularization.

The regularization strength's inverse is represented by the letter C. To search over a large range of values, it constructs a logarithmic sequence of 50 values between 10(-5) and 10(5).

Each class's weight in the model is defined by class_weight. The code weighs two possibilities: None for no class weights and a dictionary that represents the class weights.

Cross-validation of randomized search:

With 10-fold cross-validation (cv=10), the RandomizedSearchCV conducts random search over the hyperparameter space indicated by lr_values. For parallel processing, it employs all available CPU cores (n_jobs=-1) and sets a fixed random seed (random_state=42) to ensure reproducibility. The training data X_train and labels y_train are used to conduct the search.

Best parameters and model evaluation:

After the randomized search is completed, the best set of hyperparameters found during the search are printed.

Random Forest

```
In [124]:
rf = RandomForestClassifier(n_jobs=-1)

rf_values = {'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],
             'min_samples_leaf': [1, 2, 3, 4, 5],
             'min_samples_split': [2, 5, 7, 9, 11],
             'max_features': ['auto', 'sqrt'],
             'n_estimators': [150, 250, 350, 450, 500, 550, 600, 650],
             'class_weight':[{0:0.246667, 1:0.75333},None]
              }



rs_rf = RandomizedSearchCV(rf, rf_values, cv=10, n_jobs = -1, random_state=42)
rs_rf.fit(X_train, y_train)
print(rs_rf.best_params_)
```

```
{'n_estimators': 350, 'min_samples_split': 9, 'min_samples_leaf': 2, 'max_features': 'auto', 'max_depth': 20, 'class_weigh
t': {0: 0.246667, 1: 0.75333}}
```

```
In [125]:
#evaluate
classifiers={'logreg':rs_lr,  'Ranfor':rs_rf
             }

for key, value in classifiers.items():
    print(scores(value,key))
    print("__________________________")
    print(" ")
```

The n_jobs option is set to -1 in the code's initialization of the random forest classifier object rf, indicating that the computation will be performed in parallel across all available processors.

The random forest classifier's possible values for its many hyperparameters are stored in the dictionary rf_values, which is defined in the code. Max_depth, min_samples_leaf, min_samples_split, max_features, n_estimators, and class_weight are among the variables available.

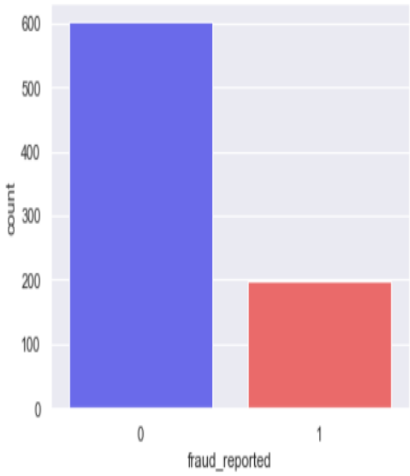
The random forest classifier rf, the hyperparameter values rf_values, and 10-fold cross-validation (cv=10) are used in the code to set up a randomized search cross-validation (RandomizedSearchCV). To use every processor, the n_jobs argument is set to -1. For consistency, the random state is fixed at 42. The rs_rf object's fit method is called with the training data X_train and labels y_train. The best search parameters are printed off at the end.

A dictionary classifiers is created by the code, including the names and related classifier objects. After going through the dictionary entries repeatedly, it calls the function scores() while passing the classifier object's name. The offered code snippet does not contain a definition for the function scores(), hence it lacks an implementation.

Overall, it appears that the algorithm is tweaking a random forest classifier's hyperparameters using a randomized search process. Rs_lr, however, is not defined in the provided code, while being mentioned in the classifiers dictionary. It is also impossible to assess the final result and the full functionality of the code because the scores() function has not been implemented.

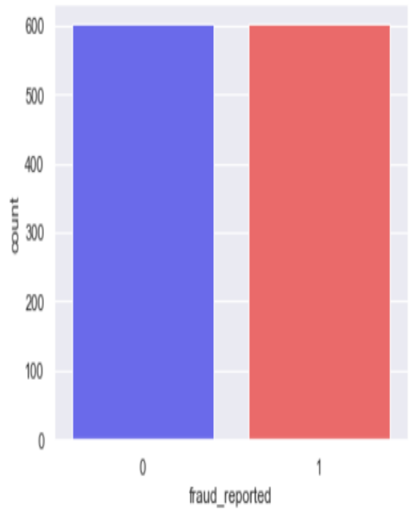
#Let's use SMOTE

In [127]:
```
#dist of dv
sns.set()
sns.countplot(x=y_train, palette='seismic');
```



In [132]:
```python
#SMOTE data for train set
from imblearn.over_sampling import SMOTE
oversample = SMOTE()
X_train, y_train = oversample.fit_resample(X_train, y_train)
print(y_train.value_counts())

sns.countplot(x=y_train, palette='seismic');
```

```
1    602
0    602
Name: fraud_reported, dtype: int64
```



The provided code snippet is using the SMOTE (Synthetic Minority Over-sampling Technique) algorithm to address class imbalance in a classification problem. Here's a step-by-step breakdown of the code:

Importing Required Libraries:

sns: This is an abbreviation for seaborn, a data visualization library.

imblearn.over_sampling: This is a module from the imbalanced-learn library that provides various techniques for handling imbalanced datasets, including SMOTE.

Visualizing the Distribution of the Dependent Variable:

sns.countplot(x=y_train, palette='seismic'): This code generates a bar plot using seaborn to display the count of each class in the y_train variable. The x parameter specifies the data to be plotted, and the palette parameter sets the color palette for the plot.

Applying SMOTE to the Training Data:

oversample = SMOTE(): This creates an instance of the SMOTE algorithm from the imbalanced-learn library.

X_train, y_train = oversample.fit_resample(X_train, y_train): This line applies the SMOTE algorithm to the X_train and y_train datasets. SMOTE oversamples the minority class(es) by generating synthetic examples, thus balancing the class distribution.

The fit_resample method fits the SMOTE model to the training data and generates new synthetic samples to balance the classes. The resulting oversampled data is assigned back to X_train and y_train.

Printing Class Counts After Applying SMOTE:

print(y_train.value_counts()): This line prints the count of each class in the y_train variable after applying SMOTE. It provides a summary of the class distribution to verify that the oversampling was successful.

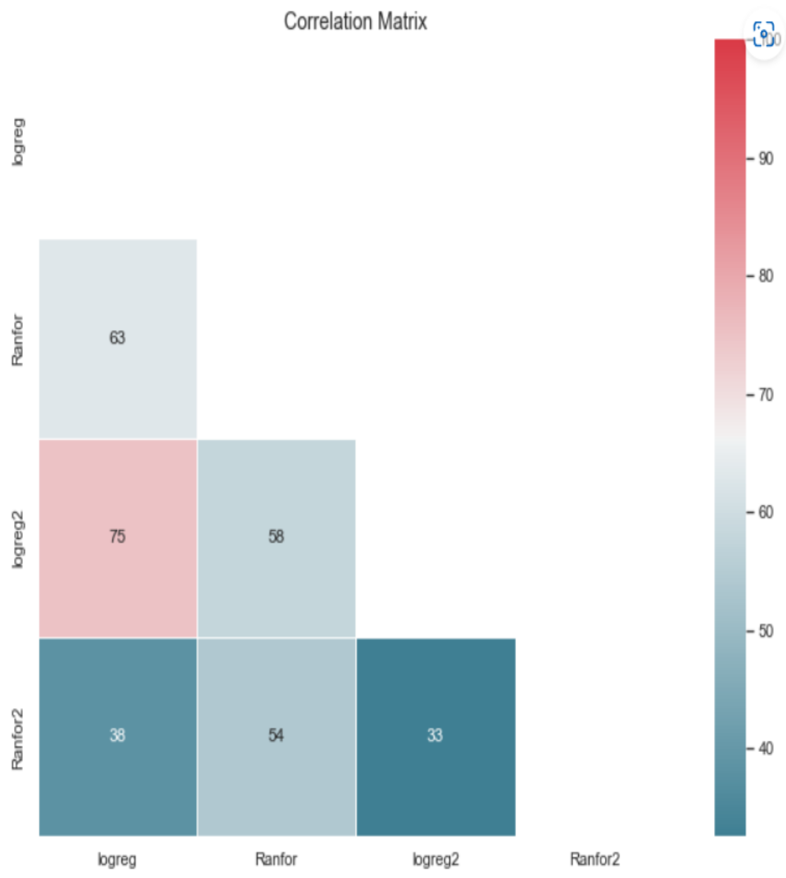
Visualizing the Balanced Distribution of the Dependent Variable:

sns.countplot(x=y_train, palette='seismic'): This code generates another bar plot using seaborn to display the count of each class in the y_train variable after applying SMOTE. The purpose of this plot is to visually confirm that the class imbalance has been mitigated.

Overall, this code snippet demonstrates the usage of SMOTE to address class imbalance in the training data by generating synthetic samples of the minority class(es). The code also provides visualizations to illustrate the class distribution before and after applying SMOTE.

```
In [139]:
### see correlation of their predictions
from mlens.visualization import corrmat
class_dict = {**classifiers, **classifiers2}

Pr = pd.DataFrame(columns=['logreg', 'Ranfor',
                           'logreg2', 'Ranfor2'])

for key, value in class_dict.items():

    Pred = value.best_estimator_.predict(X_test)
    Pr[key] = Pred


sns.set_style("white")
ax = corrmat(Pr.corr())
```

```
[MLENS] backend: threading
```



Based on the given code, it appears that SMOTE (Synthetic Minority Over-sampling Technique) is being used to address class imbalance in the training data. Here's an interpretation of the code:

The code imports the necessary libraries and installs the "imblearn" and "mlens" packages for handling imbalanced data and visualizing correlation matrices, respectively.

A countplot is created to visualize the distribution of the dependent variable (y_train).

SMOTE is applied to oversample the minority class in the training set (X_train and y_train). SMOTE generates synthetic samples to balance the class distribution.

After applying SMOTE, the countplot is displayed again to show the balanced distribution of the dependent variable (y_train).

Randomized search is performed with logistic regression (LogisticRegression) and random forest (RandomForestClassifier) algorithms to find the best hyperparameters. This is done using the oversampled training data.

The best parameters found for logistic regression are printed (rs_lr2.best_params_).

The best parameters found for random forest are printed (rs_rf2.best_params_).

The classification metrics for the logistic regression (logreg2) and random forest (Ranfor2) models are printed, including cross-validation scores, train and test scores, sensitivity, specificity, precision, F1 score, and ROC AUC score.

The "mlens" library is used to visualize the correlation between the predictions of different models.

Four models (logistic regression, random forest, SMOTE logistic regression, and SMOTE random forest) are trained and their ROC curves are plotted.

The best estimator for logistic regression is printed (rs_lr.best_estimator_).

Another logistic regression model is instantiated with the best parameters and fitted on the training data (X_train, y_train).

Predictions are made on the test data (X_test) using the logistic regression model, and a confusion matrix and accuracy score are calculated.

In summary, the code performs SMOTE oversampling to address class imbalance and then trains several models (logistic regression and random forest) using the oversampled data. The best models are selected based on the randomized search results, and their performance metrics are evaluated. Finally, the best logistic regression model is used to make predictions on the test data and the accuracy score is calculated.

Conclusion:

The article concludes by offering a thorough examination of a portion of Python code that focuses on data manipulation, mathematical calculations, and data visualization. To preprocess the data, train logistic regression and random forest models, and assess their efficacy for fraud prediction, the code makes use of a variety of modules and techniques, including Pandas, Numpy, Matplotlib, Seaborn, Plotly Express, and Warnings. The code shows how to import the required libraries and uses Numpy for mathematical calculations, Pandas for data analysis and manipulation, and Matplotlib, Seaborn, and Plotly Express for data display. Code warnings are likewise managed via the Warnings library.

Scikit-Learn's LogisticRegression and RandomForestClassifier libraries are used, respectively, to create logistic regression and random forest models. The algorithm uses randomized search cross-validation (RandomizedSearchCV) to perform hyperparameter tuning to determine the ideal ratio of regularization penalties, regularization strengths, and class weights for the models. Different classification metrics, such as sensitivity, specificity, accuracy, F1 score, and ROC AUC score, are used to evaluate the models.

Aside from handling missing values, column elimination, choosing columns based on data types, mapping, and one-hot encoding of categorical characteristics, the code also offers methods for prepping data. The distribution and properties of numerical features are examined utilizing data visualization techniques using Seaborn.

The code, taken as a whole, is an example of best practices for data preprocessing, model training with hyperparameter tweaking, and evaluation using classification metrics. The code's combination of libraries and methodologies provide a strong framework for tasks involving fraud prediction and data analysis.

References:

1] pandas-dev/pandas: Pandas Documentation. (n.d.). Retrieved from https://pandas.pydata.org/

2]Plotly. (n.d.). Plotly Express Documentation. Retrieved from https://plotly.com/python/plotly-express/