rf

November 24, 2024

[ ]:
```
from scipy.io import arff

# Data Processing
import pandas as pd
import numpy as np

# Visualisation
import matplotlib.pyplot as plt
import pandas as pd

# Replace 'your_file.arff' with the path to your ARFF file
arff_file_path = r"C:\Users\ronan\Fourth_Year\Data_Science\Data_Science\Random␣
↪Forest\kick.arff"



# Load the ARFF file
data, meta = arff.loadarff(arff_file_path)
df = pd.DataFrame(data)

# Save it as CSV
df.to_csv('converted_file.csv', index=False)
print("File successfully converted to CSV and saved as 'converted_file.csv'")
```

```
File successfully converted to CSV and saved as 'converted_file.csv'
```

[19]:
```
# Load the dataset
data = pd.read_csv('converted_file.csv')

# Display the first few rows of the dataset
print("First 5 rows of the dataset:")
print(data.head())

# Display information about the dataset
print("\nDataset Information:")
print(data.info())
```

```
First 5 rows of the dataset:
  IsBadBuy     PurchDate   Auction  VehYear  VehicleAge      Make  \
```

```
0     b'0'  1.260144e+09  b'ADESA'   2006.0         3.0  b'MAZDA'
1     b'0'  1.260144e+09  b'ADESA'   2004.0         5.0  b'DODGE'
2     b'0'  1.260144e+09  b'ADESA'   2005.0         4.0  b'DODGE'
3     b'0'  1.260144e+09  b'ADESA'   2004.0         5.0  b'DODGE'
4     b'0'  1.260144e+09  b'ADESA'   2005.0         4.0   b'FORD'

                  Model    Trim              SubModel      Color  …  \
0              b'MAZDA3'    b'i'         b'4D SEDAN I'     b'RED'  …
1  b'1500 RAM PICKUP 2WD'   b'ST'  b'QUAD CAB 4.7L SLT'   b'WHITE'  …
2          b'STRATUS V6'  b'SXT'   b'4D SEDAN SXT FFV'  b'MAROON'  …
3                b'NEON'  b'SXT'           b'4D SEDAN'  b'SILVER'  …
4               b'FOCUS'  b'ZX3'       b'2D COUPE ZX3'  b'SILVER'  …

  MMRCurrentRetailAveragePrice MMRCurrentRetailCleanPrice PRIMEUNIT  AUCGUART  \
0                      11597.0                    12409.0      b'?'      b'?'
1                      11374.0                    12791.0      b'?'      b'?'
2                       7146.0                     8702.0      b'?'      b'?'
3                       4375.0                     5518.0      b'?'      b'?'
4                       6739.0                     7911.0      b'?'      b'?'

      BYRNO    VNZIP1   VNST  VehBCost  IsOnlineSale  WarrantyCost
0  b'21973'  b'33619'  b'FL'    7100.0          b'0'        1113.0
1  b'19638'  b'33619'  b'FL'    7600.0          b'0'        1053.0
2  b'19638'  b'33619'  b'FL'    4900.0          b'0'        1389.0
3  b'19638'  b'33619'  b'FL'    4100.0          b'0'         630.0
4  b'19638'  b'33619'  b'FL'    4000.0          b'0'        1020.0

[5 rows x 33 columns]

Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 72983 entries, 0 to 72982
Data columns (total 33 columns):
 #   Column                             Non-Null Count  Dtype
---  ------                             --------------  -----
 0   IsBadBuy                           72983 non-null  object
 1   PurchDate                          72983 non-null  float64
 2   Auction                            72983 non-null  object
 3   VehYear                            72983 non-null  float64
 4   VehicleAge                         72983 non-null  float64
 5   Make                               72983 non-null  object
 6   Model                              72983 non-null  object
 7   Trim                               72983 non-null  object
 8   SubModel                           72983 non-null  object
 9   Color                              72983 non-null  object
 10  Transmission                       72983 non-null  object
 11  WheelTypeID                        72983 non-null  object
 12  WheelType                          72983 non-null  object
```

```
 13  VehOdo                             72983 non-null  float64
 14  Nationality                        72983 non-null  object
 15  Size                               72983 non-null  object
 16  TopThreeAmericanName               72983 non-null  object
 17  MMRAcquisitionAuctionAveragePrice  72965 non-null  float64
 18  MMRAcquisitionAuctionCleanPrice    72965 non-null  float64
 19  MMRAcquisitionRetailAveragePrice   72965 non-null  float64
 20  MMRAcquisitonRetailCleanPrice      72965 non-null  float64
 21  MMRCurrentAuctionAveragePrice      72668 non-null  float64
 22  MMRCurrentAuctionCleanPrice        72668 non-null  float64
 23  MMRCurrentRetailAveragePrice       72668 non-null  float64
 24  MMRCurrentRetailCleanPrice         72668 non-null  float64
 25  PRIMEUNIT                          72983 non-null  object
 26  AUCGUART                           72983 non-null  object
 27  BYRNO                              72983 non-null  object
 28  VNZIP1                             72983 non-null  object
 29  VNST                               72983 non-null  object
 30  VehBCost                           72915 non-null  float64
 31  IsOnlineSale                       72983 non-null  object
 32  WarrantyCost                       72983 non-null  float64
dtypes: float64(14), object(19)
memory usage: 18.4+ MB
None
```

[22]:
```python
# Define the columns to keep
columns_to_keep = ['IsBadBuy', 'Auction', 'Make', 'Model', 'Color',␣
 ↪'Transmission',
                   'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName']

# Retain only the necessary columns
data_cleaned = data[columns_to_keep]

# Verify the cleaned dataset
print("Cleaned Dataset Columns:")
print(data_cleaned.columns)
```

```
Cleaned Dataset Columns:
Index(['IsBadBuy', 'Auction', 'Make', 'Model', 'Color', 'Transmission',
       'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName'],
      dtype='object')
```

[23]:
```python
# Identify categorical columns for one-hot encoding
categorical_columns = ['Auction', 'Make', 'Model', 'Color', 'Transmission',␣
 ↪'WheelType', 'Nationality', 'Size', 'TopThreeAmericanName']

# Apply one-hot encoding
```

```
data_encoded = pd.get_dummies(data_cleaned, columns=categorical_columns,␣
↪drop_first=True)

# Display the dimensions (number of rows and columns) of the encoded dataset
print(f"Encoded Dataset Dimensions: {data_encoded.shape}")
```

```
Encoded Dataset Dimensions: (72983, 1139)
```

- **What the Code Does**:
  - Applies one-hot encoding to categorical columns, converting them into numerical binary columns.
  - Drops the first category for each categorical feature to avoid redundancy (dummy variable trap).
  - Prints the dimensions (rows and columns) of the encoded dataset.
- **Why It's Done**:
  - Machine learning models like Random Forest require numerical inputs, and one-hot encoding is a common method for handling categorical data.
  - Checking the dataset's dimensions ensures the encoding process worked as expected without overwhelming the output with data.

[38]:
```
# Define features (X) and target (y)
X = data_encoded.drop('IsBadBuy', axis=1)
y = data_encoded['IsBadBuy']

# Display the shapes of X and y to confirm the split
print(f"Features Shape: {X.shape}")
print(f"Target Shape: {y.shape}")
```

```
Features Shape: (72983, 1138)
Target Shape: (72983,)
```

- **What the Code Does**:
  - Separates the dataset into features (`X`) and the target variable (`y`).
  - Prints the shapes of `X` and `y` to verify the separation.
- **Why It's Done**:
  - Splitting the dataset into independent (features) and dependent (target) variables is necessary for training the machine learning model.
  - Ensures the target variable (`IsBadBuy`) is isolated, and all remaining columns are predictors.

[39]:
```
from sklearn.model_selection import train_test_split

# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.8,␣
↪random_state=42)
```

```
# Display the shapes of the resulting datasets
print(f"Training Features Shape: {X_train.shape}")
print(f"Training Target Shape: {y_train.shape}")
print(f"Testing Features Shape: {X_test.shape}")
print(f"Testing Target Shape: {y_test.shape}")
```

```
Training Features Shape: (14596, 1138)
Training Target Shape: (14596,)
Testing Features Shape: (58387, 1138)
Testing Target Shape: (58387,)
```

- **What the Code Does**:
  - Splits the dataset into training (20%) and testing (80%) sets.
  - Ensures reproducibility by setting a random seed (`random_state=42`).
  - Prints the shapes of training and test datasets for both features and targets.
- **Why It's Done**:
  - A split is necessary to evaluate the model's performance on unseen data.
  - A larger test set (80%) allows for a more robust assessment of the model's generalisation.
  - Ensures the training set (20%) is still large enough to train the model effectively.

[27]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix

# Initialize the Random Forest Classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model on the training data
rf.fit(X_train, y_train)

print("Model trained successfully!")
```

```
Model trained successfully!
```

[28]:
```
# Predict the target variable for the test set
y_pred = rf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Print a detailed classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Display the confusion matrix
```

```
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

```
Accuracy: 0.88

Classification Report:
              precision    recall  f1-score   support

        b'0'       0.90      0.97      0.93     51224
        b'1'       0.50      0.21      0.29      7163

    accuracy                           0.88     58387
   macro avg       0.70      0.59      0.61     58387
weighted avg       0.85      0.88      0.85     58387


Confusion Matrix:
[[49734  1490]
 [ 5690  1473]]
```

0.0.1 Analysis of the Results

1. Overall Accuracy

- **Accuracy: 0.88**
  - This is a high overall accuracy, indicating that the model correctly predicts whether a car is a "bad buy" in **88% of cases**.

2. Precision, Recall, and F1-Score

- For `b'0'` (Not a "bad buy"):
  - **Precision (0.90)**: Out of all predicted "not bad buys," 90% are correct.
  - **Recall (0.97)**: Out of all actual "not bad buys," 97% are correctly identified.
  - **F1-Score (0.93)**: Balances precision and recall well for this class.
- For `b'1'` ("bad buy"):
  - **Precision (0.50)**: Out of all predicted "bad buys," only 50% are correct.
  - **Recall (0.21)**: Out of all actual "bad buys," only 21% are correctly identified.
  - **F1-Score (0.29)**: Indicates poor performance in identifying "bad buys."

3. Confusion Matrix

- **True Negatives (`b'0'` correctly classified)**: 49,734
- **False Positives (`b'0'` misclassified as `b'1'`)**: 1,490
- **True Positives (`b'1'` correctly classified)**: 1,473
- **False Negatives (`b'1'` misclassified as `b'0'`)**: 5,690

The model excels at identifying "not bad buys" (`b'0'`) but struggles to detect actual "bad buys" (`b'1'`), likely due to class imbalance.

0.0.2 Interpretation

- **Imbalanced Data**: The results suggest a significant class imbalance. There are far more b'0' samples than b'1' samples, causing the model to favour the majority class.
- **High Weighted Average**: The **weighted average F1-score (0.85)** indicates good overall performance because the majority class dominates the data.

[ ]:
```
from imblearn.over_sampling import SMOTE
from collections import Counter

# Convert all boolean columns to integers (0 and 1)
X_train = X_train.astype(int)

# Apply SMOTE only on the training set
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

# Check class distribution before and after SMOTE
print(f"Class distribution before SMOTE: {Counter(y_train)}")
print(f"Class distribution after SMOTE: {Counter(y_train_resampled)}")
```

```
Class distribution before SMOTE: Counter({"b'0'": 12783, "b'1'": 1813})
Class distribution after SMOTE: Counter({"b'0'": 12783, "b'1'": 12783})
```

[35]:
```
# Train the Random Forest model on the resampled training data
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train_resampled, y_train_resampled)

print("Model trained successfully")
```

```
Model trained successfully
```

[36]:
```
# Make predictions on the test set
y_pred = rf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

print("\nClassification Report:")
print(classification_report(y_test, y_pred))

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

```
Accuracy: 0.80
```

```
Classification Report:
              precision    recall  f1-score   support

        b'0'       0.91      0.86      0.88     51224
        b'1'       0.26      0.35      0.30      7163

    accuracy                           0.80     58387
   macro avg       0.58      0.61      0.59     58387
weighted avg       0.83      0.80      0.81     58387


Confusion Matrix:
[[44080  7144]
 [ 4621  2542]]
```

0.0.3 Comparison to the Previous Result

1. **Previous Result**:
   - **Accuracy**: 0.88
   - The model heavily favoured the majority class (`b'0'`), achieving high accuracy because most of the test set was dominated by `b'0'` (imbalanced data).
   - **Issue**: It performed poorly for the minority class (`b'1'`), with low recall (only 21%) and a poor F1-score for `b'1'`.
2. **Current Result (After SMOTE)**:
   - **Accuracy**: 0.80
   - After balancing the training set with SMOTE, the model gives more attention to both classes (`b'0'` and `b'1'`).
   - This shift means the model is less biased towards the majority class and tries harder to identify the minority class (`b'1'`).
   - **Improvement**: The recall for `b'1'` improved significantly (from 21% to 35%), and the F1-score also increased from 0.29 to 0.30.

0.0.4 Why Accuracy is Lower

The accuracy dropped because:

1. **Balanced Training Set**:
   - Before SMOTE, the training data was imbalanced, and the model learned to prioritise predicting `b'0'` (the majority class).
   - After SMOTE, the model now treats `b'1'` as equally important, which sacrifices some accuracy on `b'0'`.
2. **Trade-Off Between Classes**:
   - To improve the performance for `b'1'`, the model has to “accept” more false positives for `b'0'`.

- This is evident in the **Confusion Matrix**:
  - **False Positives (b'0' misclassified as b'1')** increased from **1490** to **7144**.

3. **Test Set Imbalance**:

   - The test set is still heavily skewed towards `b'0'` (majority class). While SMOTE improved `b'1'` detection, the imbalance in the test set can artificially inflate accuracy when the model predicts mostly `b'0'`.

0.0.5 What This Means

- **Pros**:

  - The model is now better at identifying `b'1'` (minority class), which was the primary goal of applying SMOTE.
  - The recall for `b'1'` improved from 21% to 35%.

- **Cons**:

  - The overall accuracy is slightly lower because the model is now less biased towards the majority class (`b'0'`).

[40]:
```
# Train the Random Forest model with class weight adjustment
rf_weighted = RandomForestClassifier(n_estimators=100, class_weight='balanced',
 ↪random_state=42)
rf_weighted.fit(X_train, y_train)

# Test the model on the test set
y_pred_weighted = rf_weighted.predict(X_test)

# Evaluate the performance
accuracy_weighted = accuracy_score(y_test, y_pred_weighted)
print(f"Accuracy with Class Weights: {accuracy_weighted:.2f}")

print("\nClassification Report:")
print(classification_report(y_test, y_pred_weighted))

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred_weighted))
```

```
Accuracy with Class Weights: 0.81

Classification Report:
              precision    recall  f1-score   support

        b'0'       0.90      0.89      0.89     51224
        b'1'       0.25      0.27      0.26      7163

    accuracy                           0.81     58387
```

```
   macro avg       0.57      0.58      0.58     58387
weighted avg       0.82      0.81      0.81     58387


Confusion Matrix:
[[45368  5856]
 [ 5207  1956]]
```

0.0.6 Weighted Model Results

- **Accuracy**: **0.81** (slightly better than SMOTE but lower than the original model's 0.88).
- **Precision and Recall for `b'1'`**:
  - **Precision**: **0.25** (lower than SMOTE's **0.26** and original's **0.50**).
  - **Recall**: **0.27** (lower than SMOTE's **0.35** but higher than the original's **0.21**).
  - **F1-Score**: **0.26**, which is slightly worse than SMOTE but better than the original.

0.0.7 Comparison with Other Models

Metric	Original Model	SMOTE Model	Weighted Model
Accuracy	0.88	0.80	**0.81**
Precision (b'1')	0.50	0.26	**0.25**
Recall (b'1')	0.21	**0.35**	0.27
F1-Score (b'1')	0.29	**0.30**	0.26

Observations

1. **Original Model**:
   - Achieved the highest accuracy but heavily favoured the majority class (`b'0'`).
   - Performed poorly in detecting the minority class (`b'1'`) with the lowest recall (21%).
2. **SMOTE Model**:
   - Lower accuracy (0.80) but the best recall (35%) for `b'1'`.
   - Balanced the training data, improving minority class detection at the cost of majority class performance.
3. **Weighted Model**:
   - Accuracy is slightly higher than SMOTE (0.81), indicating better balance between the classes.
   - Recall (27%) for `b'1'` improved over the original model but dropped slightly compared to SMOTE.
   - Precision for `b'1'` (25%) is comparable to SMOTE but much lower than the original (50%).

0.0.8 Why the Accuracy is Lower

1. **Trade-Off Between Classes**:
   - The original model prioritised the majority class, inflating accuracy at the cost of minority class performance.
   - Weighted models redistribute focus, improving minority class performance but slightly reducing overall accuracy.
2. **Test Set Imbalance**:
   - The test set is still imbalanced, so improvements in detecting `b'1'` come at the expense of correctly identifying `b'0'`.
3. **Class Weights vs SMOTE**:
   - SMOTE artificially balances the training set, giving more opportunity to learn minority class patterns.
   - Weighted models adjust the importance of each class during training without changing the data distribution.

0.1 Data Sources

- OpenML Used Cars Dataset (ID 41162): This dataset contains details about used cars, including features such as make, model, age, mileage, and auction details, as well as a binary label (`IsBadBuy`) indicating whether the car was a bad purchase.

0.2 Pre-Processing

- **Dropped Irrelevant Columns**:
  - Removed columns like `PurchDate`, `SubModel`, `VNZIP1`, `BYRNO`, `AUCGUART`, and `PRIMEUNIT` as they provided minimal value for predicting `IsBadBuy`.
  - Further narrowed down the columns to only the most relevant categorical and numerical features:
    * `Auction`, `Make`, `Model`, `Color`, `Transmission`, `WheelType`, `Nationality`, `Size`, `TopThreeAmericanName`, and `VehOdo`.
- **Handled Missing Values**:
  - Dropped rows with missing or undefined values to ensure a clean dataset for training and testing the Random Forest model.
- **One-Hot Encoding**:
  - Converted categorical variables into numerical features using one-hot encoding for compatibility with the Random Forest algorithm. Each unique category became a binary column.
  - Example Resources:
    * GeeksforGeeks - One-Hot Encoding in Machine Learning
- **Data Splitting**:

- Divided the dataset into 20% training and 80% testing to evaluate model performance on a larger test set.

0.3 Data Understanding/Visualisation

- The target variable `IsBadBuy` was highly imbalanced, with significantly more `b'0'` (good buys) than `b'1'` (bad buys). This imbalance impacted model performance for minority class detection.
- Applied class balancing techniques:
  - Used **SMOTE** to oversample the minority class in the training data.
  - Experimented with **class weighting** in the Random Forest algorithm.

0.4 Algorithms

- **Random Forest Classifier**:
  - Implemented using Scikit-learn's `RandomForestClassifier`.
  - Trained the model on various versions of the dataset:
    * **Original Dataset**: High overall accuracy but poor minority class recall.
    * **SMOTE Resampled Dataset**: Improved minority class recall at the cost of overall accuracy.
    * **Class-Weighted Model**: Balanced performance between majority and minority classes, offering a middle ground.
  - Example Resources:
    * DataCamp - Random Forests Classifier in Python
- **Evaluation Metrics**:
  - Assessed using **accuracy**, **precision**, **recall**, **F1-score**, and **confusion matrices** to measure overall performance and focus on minority class detection.

0.5 Results

Metric	Original Model	SMOTE Model	Weighted Model
Accuracy	0.88	0.80	0.81
Precision (b'1')	0.50	0.26	0.25
Recall (b'1')	0.21	0.35	0.27
F1-Score (b'1')	0.29	0.30	0.26

- **Original Model**: Best overall accuracy but heavily biased towards the majority class (`b'0'`).
- **SMOTE Model**: Improved recall for `b'1'` at the cost of accuracy, suitable if minority class detection is the priority.
- **Weighted Model**: Balanced the trade-offs, achieving slightly better accuracy than SMOTE with reasonable recall.

0.6 Online Resources & Sources

- GeeksforGeeks - One-Hot Encoding in Machine Learning: Helped convert categorical data into numerical format.
- DataCamp - Random Forests Classifier in Python: Used as a reference for starting the Random Forest implementation.
- OpenML Dataset - Used Cars (ID 41162)

0.7 Tools & Technologies Used

- **Python Libraries**:
  - `Pandas` and `NumPy`: For data manipulation and cleaning.
  - `Scikit-Learn`: To implement Random Forest and evaluate model performance.
  - `Imbalanced-Learn`: For applying SMOTE to balance the dataset.
  - `Chat GPT`
- **Jupyter Notebook**:
  - Used for step-by-step data exploration, cleaning, and model building.

0.8 Challenges Faced

0.8.1 1. Addressing Class Imbalance

- The dataset was heavily skewed, with the majority of samples labelled as `b'0'` (good buy) and far fewer as `b'1'` (bad buy).
- Balancing techniques like SMOTE improved recall for `b'1'`, but at the cost of overall accuracy.
- Class weighting provided a middle ground but was less effective than SMOTE in improving minority class recall.

0.8.2 2. Handling High Dimensionality

- One-hot encoding of categorical features resulted in a significant increase in the number of columns (over 1,000 features).
- This increased dimensionality introduced computational overhead during model training and required more memory, especially when combined with SMOTE.

0.8.3 3. Balancing Trade-Offs

- Balancing the dataset using SMOTE improved minority class recall but decreased precision and overall test accuracy.
- Class weighting improved computational efficiency but failed to match SMOTE in minority class recall, making it a less optimal solution for certain goals.

0.8.4 4. Poor Generalisation to the Test Set

- Even after balancing, the model struggled to generalise well to the test set.
- The test set's imbalance led to a tendency to misclassify minority class samples, reflecting the challenge of applying balanced training to imbalanced testing.

0.8.5 5. Minority Class Performance

- Despite balancing efforts, the model consistently struggled to identify `b'1'` accurately.
- Precision and recall for `b'1'` remained low, reflecting the difficulty of predicting rare outcomes effectively in real-world imbalanced datasets.

0.8.6 6. Interpretability of Results

- The increase in feature space due to one-hot encoding made it harder to interpret which features contributed most to the predictions.
- Understanding the impact of categorical features like `Make`, `Model`, and `Transmission` required additional analysis beyond the initial model.