Classification Report: Undersampling with Forest Cover Type Dataset

Author: Syed Bokhari

Date of Submission: 25th November 2022

Introduction:

The dataset used for this project is originally a part of the UCI Machine Learning repository. An unaltered copy of the dataset from Kaggle was used for the project. The dataset can be found at: https://www.kaggle.com/datasets/uciml/forest-cover-type-dataset

The dataset includes data on 4 wilderness areas located in the Roosevelt National Forest of Northern Colorado. The areas covered by the dataset are those where human disturbances have largely been absent. Therefore, the tree growth in these areas is due to ecological processes and not human practices. Each observation captures data on a 30 x 30 meter cell for a forested area. The dataset captures information on 13 different features for 581,012 such observations. Two of the features have been one-hot encoded so that the dataset has a total of 55 columns. All columns are in integer format.

Elevation (int): The elevation of the area in meters.

Aspect (int): Aspect of the area in degrees azimuth (to determine sunlight patterns)

Slope (int): Slope of the area in degrees.

Horizontal_Distance_To_Hydrology (int): Horizontal distance to the nearest surface water features in meters.

Vertical_Distance_To_Hydrology (int): Horizontal distance to the nearest surface water features in meters.

Horizontal_Distance_To_Roadways (int): Horizontal distance to the nearest roadway in meters.

Hillshade_9am (int): An index ranging from 0 to 255 which captures hillshade at 9am for summer solstice.

Hillshade_noon (int): An index ranging from 0 to 255 which captures hillshade at noon for summer solstice.

Hillshade_3pm (int): An index ranging from 0 to 255 which captures hillshade at 3pm for summer solstice.

Horizontal_Distance_To_Fire_Points (int): Horizontal distance to the nearest wildfire points in meters.

Wilderness_Area (int): A variable designating which of the 4 areas covered in the study is the observed area a part of. This feature is already one-hot encoded into 4 columns in the dataset.

Soil_Type (int): A variable designating Soil type of area. This feature is already one-hot encoded into 40 columns in the dataset.

Cover_Type (int): Forest cover type designation of area ranging from 1 to 7 for different forest cover type.

Objectives:

This project aims to investigate the efficacy of using the undersampling approach with a dataset that has a large class imbalance and has a large number of observations as well. The project aims to use an undersampling technique to reduce the size of the dataset such that the remaining datapoints all have equal weights for the classes in the target column. These data points will then be used to train classifiers that will be evaluated on the data points from the dataset that were not used for training. The accuracy of the resulting predictions will be studied in detail to understand the results. While the project will primarily focus on predictions because of this but these predictions will be used to make an evaluation of undersampling as a technique so interpretation of results will be important also. The resulting observations will prove fruitful for any data researchers considering the use of undersampling techniques. The results of the analysis are elaborated upon in the Results section and the resulting insights in the Key Findings section. Some steps for future analysis are also presented in the Future Steps section.

Data Exploration:

The dataset was initially in csv format but was loaded into a Pandas dataframe. Pandas is a very useful library in Python for data preprocessing. It chiefly makes use of Dataframes, a built-in data structure, for performing data preprocessing and analysis in an efficient manner.

```
df = pd.read_csv('covtype.csv')
```

```
df.head()
```

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hillshade
0	2596	51	3	258	0	510	221	
1	2590	56	2	212	-6	390	220	
2	2804	139	9	268	65	3180	234	
3	2785	155	18	242	118	3090	238	
4	2595	45	2	153	-1	391	220	

5 rows × 55 columns

Once the dataset had been loaded into a Pandas dataframe, a basic exploratory data analysis (EDA) was conducted.

The dataset is quite large.

```
df.shape
```

```
(581012, 55)
```

The data types were all integer which is very convenient as models for machine learning require numeric inputs. Several features had been one-hot encoded in advance as well which greatly eased the task of subsequent data processing for machine learning.

```
typelist = df.dtypes
```

```
typelist.value_counts()
```

```
int64    55
dtype: int64
```

Some of the features however, were not in standardized form, with a varying range of values. This would require handling at later stages as some classification models to be used later took distance metrics between features for calculation purposes.

	count	mean	std	min	25%	50%	75%	max
Elevation	581012.0	2959.365301	279.984734	1859.0	2809.0	2996.0	3163.0	3858.0
Aspect	581012.0	155.656807	111.913721	0.0	58.0	127.0	260.0	360.0
Slope	581012.0	14.103704	7.488242	0.0	9.0	13.0	18.0	66.0
Horizontal_Distance_To_Hydrology	581012.0	269.428217	212.549356	0.0	108.0	218.0	384.0	1397.0
Vertical_Distance_To_Hydrology	581012.0	46.418855	58.295232	-173.0	7.0	30.0	69.0	601.0
Horizontal_Distance_To_Roadways	581012.0	2350.146611	1559.254870	0.0	1106.0	1997.0	3328.0	7117.0
Hillshade_9am	581012.0	212.146049	26.769889	0.0	198.0	218.0	231.0	254.0
Hillshade_Noon	581012.0	223.318716	19.768697	0.0	213.0	226.0	237.0	254.0
Hillshade_3pm	581012.0	142.528263	38.274529	0.0	119.0	143.0	168.0	254.0
Horizontal_Distance_To_Fire_Points	581012.0	1980.291226	1324.195210	0.0	1024.0	1710.0	2550.0	7173.0
Wilderness_Area1	581012.0	0.448865	0.497379	0.0	0.0	0.0	1.0	1.0
Wilderness_Area2	581012.0	0.051434	0.220882	0.0	0.0	0.0	0.0	1.0
Wilderness_Area3	581012.0	0.436074	0.495897	0.0	0.0	0.0	1.0	1.0
Wilderness_Area4	581012.0	0.063627	0.244087	0.0	0.0	0.0	0.0	1.0
Soil_Type1	581012.0	0.005217	0.072039	0.0	0.0	0.0	0.0	1.0
Soil_Type2	581012.0	0.012952	0.113066	0.0	0.0	0.0	0.0	1.0

Luckily, there were no missing values in the dataset. This was verified with positive results.

```
missing_vals = df.isna()
```

```
missing_vals.head()
```

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hillshade
0	False	False	False	False	False	False	False	
1	False	False	False	False	False	False	False	
2	False	False	False	False	False	False	False	
3	False	False	False	False	False	False	False	
4	False	False	False	False	False	False	False	

5 rows × 55 columns

```
missing_vals.sum().sum()
```

```
0
```

Finally, the class column Cover_Type was checked. The classes in the column turned out to be in label encoded form already. However, the column turned out to have an imbalanced distribution of class values.

```
df['Cover_Type'].value_counts()
```

```
2    283301
1    211840
3     35754
7     20510
6     17367
5      9493
4      2747
Name: Cover_Type, dtype: int64
```

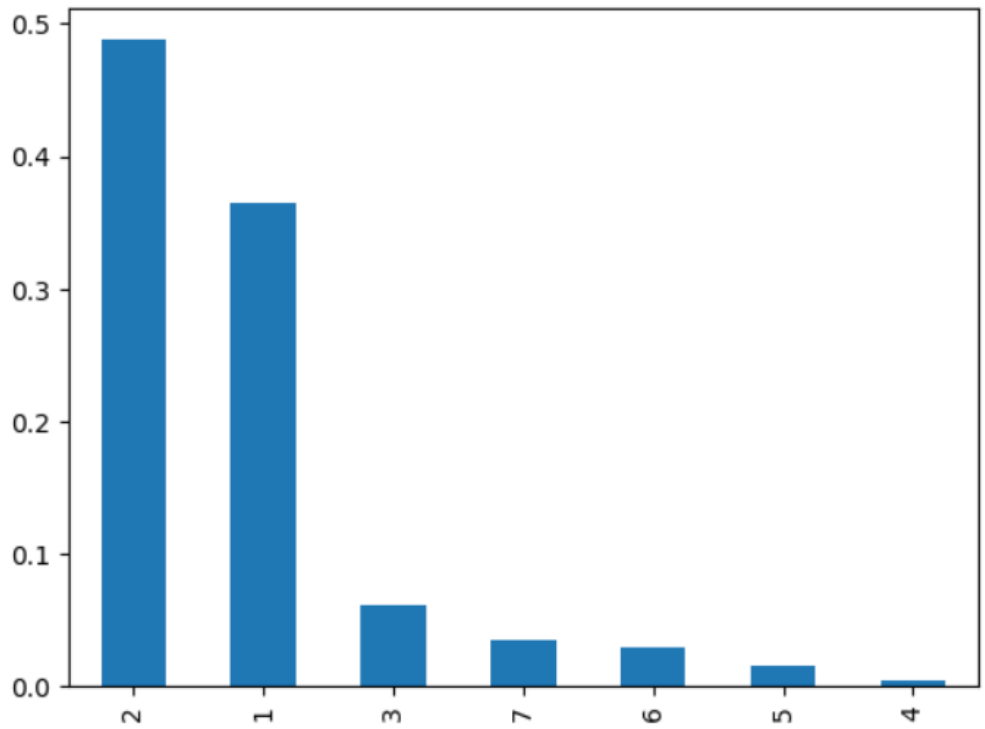
```
df['Cover_Type'].value_counts(normalize=True)
```

```
2    0.487599
1    0.364605
3    0.061537
7    0.035300
6    0.029891
5    0.016339
4    0.004728
Name: Cover_Type, dtype: float64
```

It is clear from the distribution of the values that the dataset suffers from highly imbalanced classes. Classes labelled 2 and 1 comprise about 85% of the total values in the dataset. Classes 3, 7 and 6 comprise only about 12.5 % of the dataset while classes 5 and 4 make up about 2% of the values only. This makes the dataset perfect for the objectives of the analysis. The class distribution was visualized for further clarity:

```
df['Cover_Type'].value_counts(normalize=True).plot(kind='bar')
```

```
<AxesSubplot:>
```



Plan:

The aim of the project was to make use of the class imbalance observed during Data Exploration. A RandomUnderSampler object would be used to downsample the dataset. However, given the class imbalance observed in the previous section, downsampling will reduce the dataset to a very small number of data rows. Given the size of the dataset, this would still be a large number of values. The question to answer was, can this downsampled dataset be used to train classifiers that can perform reasonably well on unseen data. For this purpose, the values of the dataset outputted by the RandomUnderSampler object would be used as training data and values of the dataset other than the training data will be used as the testing data. This will result in a fairly large testing set but a reasonable training set.

With the training and testing data ready, several different classifiers would be trained and then evaluated on the testing data. Afterwards their performance will be compared. Then, advanced techniques such as Bagging, Boosting and Stacking will be used to improve the outcomes obtained if possible. The final results will then be evaluated for drawing key insights if any.

Feature Engineering:

For the training and testing data, the approach outlined in the plan suffered from a complication. There was no means of distinguishing data rows from one another. Each row had differing values but no unique column to identify it. The RandomUnderSampler would output a portion of the dataset during downsampling as training data, but these rows needed to be removed from the original dataset to create the testing set. Without any unique identifier column, this seemed an impossible task.

As a solution to this problem, the index values were added as a column to the dataset. Pandas dataframes have an index for each value by default. These indexes can act as a unique column for identification purposes.

```
df.head()
```

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hillshade
0	2596	51	3	258	0	510	221	
1	2590	56	2	212	-6	390	220	
2	2804	139	9	268	65	3180	234	
3	2785	155	18	242	118	3090	238	
4	2595	45	2	153	-1	391	220	

5 rows × 55 columns

For this approach to work, we turn index column into a unique identifier column for each row.

```
df.reset_index(inplace=True)
```

```
df.tail()
```

	index	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9
581007	581007	2396	153	20	85	17	108	
581008	581008	2391	152	19	67	12	95	
581009	581009	2386	159	17	60	7	90	
581010	581010	2384	170	15	60	5	90	
581011	581011	2383	165	13	60	4	67	

5 rows × 56 columns

Train-Test Split:

Subsequently, this modified dataframe was used to create the features set.

```
X = df.iloc[:,:-1]
```

```
X.head()
```

	index	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	H
0	0	2596	51	3	258	0	510	221	
1	1	2590	56	2	212	-6	390	220	
2	2	2804	139	9	268	65	3180	234	
3	3	2785	155	18	242	118	3090	238	
4	4	2595	45	2	153	-1	391	220	

5 rows × 55 columns

It was also used to create the target set with the class distribution intact.

```
y = df.iloc[:, -1:]
```

```
y.head()
```

	Cover_Type
0	5
1	5
2	2
3	2
4	5

```
y.value_counts(normalize=True)
```

```
Cover_Type
2            0.487599
1            0.364605
3            0.061537
7            0.035300
6            0.029891
5            0.016339
4            0.004728
dtype: float64
```

The data was undersampled and the new training dataset was created. The target column now had equal number of values for each class.

```
under_sampler = RandomUnderSampler(random_state=123)
```

```
X_train, y_train = under_sampler.fit_resample(X, y)
```

```
X_train.shape
```

```
(19229, 55)
```

```
y_train.shape
```

```
(19229, 1)
```

```
y_train.value_counts(normalize=True)
```

```
Cover_Type
1            0.142857
2            0.142857
3            0.142857
4            0.142857
5            0.142857
6            0.142857
7            0.142857
dtype: float64
```

The testing datasets for features and target were subsequently created by removing training data's indexes from the original dataset.

```
X_test = df[~df.isin(X_train)].dropna()
```

```
X_test.head()
```

	index	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9a
19229	19229	2676.0	359.0	20.0	30.0	13.0	1736.0	187
19230	19230	2673.0	6.0	20.0	42.0	10.0	1708.0	191
19231	19231	2673.0	10.0	18.0	60.0	15.0	1679.0	198
19232	19232	2669.0	6.0	15.0	67.0	16.0	1651.0	200
19233	19233	2671.0	4.0	15.0	85.0	16.0	1622.0	200

5 rows × 56 columns

```
y_test = X_test.iloc[:,-1:]
```

```
y_test.shape
```

```
(561783, 1)
```

```
X_test = X_test.iloc[:,:-1]
```

```
X_test.shape
```

```
(561783, 55)
```

With the train and test sets created, the index column served no further purpose and was therefore dropped.

```
X_train = X_train.iloc[:,1:]
```

```
X_train.head()
```

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hillshade
0	3109	160	11	258	40	1384	231	
1	3157	275	27	85	11	1084	139	
2	3086	93	16	30	-2	2584	243	
3	3181	261	25	212	42	4656	149	
4	3054	103	4	690	50	5810	227	

5 rows × 54 columns

```
X_test = X_test.iloc[:,1:]
```

```
X_test.head()
```

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roadways	Hillshade_9am	Hills
19229	2676.0	359.0	20.0	30.0	13.0	1736.0	187.0	
19230	2673.0	6.0	20.0	42.0	10.0	1708.0	191.0	
19231	2673.0	10.0	18.0	60.0	15.0	1679.0	198.0	
19232	2669.0	6.0	15.0	67.0	16.0	1651.0	200.0	
19233	2671.0	4.0	15.0	85.0	16.0	1622.0	200.0	

5 rows × 54 columns

The data was now prepared for classification. In the end, the downsampling procedure reduced the original data to about 3.3% of the original dataset's size.

```
X_train.shape[0] + X_test.shape[0]
```

```
581012
```

```
df.shape[0]
```

```
581012
```

```
(X_train.shape[0] / df.shape[0])*100
```

```
3.309570198205889
```

Standardization:

The final step before proceeding with classification was to standardize the data. As was noted previously, some features in the dataset had widely varying values which could pose some complications for certain classifiers. Therefore, the features in the training data were standardized using a StandardScaler() object with the fit_transform method. Afterwards, using the calculated mean and standard deviation when standardizing the training dataset, the testing features data was also standardized. Using the transform method.

```
scaler = StandardScaler()
```

```
X_train_s = scaler.fit_transform(X_train)
```

```
X_test_s = scaler.transform(X_test)
```

Classification:

With the datasets ready, the next step was classification. Multiple classifiers were trained with the training dataset and subsequently evaluated on the testing dataset. For all classifiers, the target column was converted from a column vector to an array as that was required by the models.

All models were evaluated using the classification report available in Sci-kit Learn. This is a handy tool as it displays precision, recall and the f1 score for a model's predictions. It also shows the success of the model in predicting different classes which is convenient for multi-class predictions.

The precision score in classification report reflects the number of correctly identified labels over all the identified labels for a class.

Th accuracy score in classification report reflects the number of correctly identified labels from all present instances of a class in the data.

The F-1 score is a balanced metric that takes the harmonic mean of the precision and accuracy scores. This is the primary means used in the project of gauging an estimator's performance.

Logistic Regression:

The first classifier to be used was a Logistic Regression based classifier. For this step, a LogisticRegressionCV object was used. The maximum number of iterations were set to 1000 to avoid any convergence problems (the default is 100) and l2 penalty was selected. The solver was chosen to be the SAGA algorithm. Regularization was specified to be Cs=1 and 5 cross-folds were to be used for better accuracy.

```
lr = LogisticRegressionCV(Cs=1, max_iter=1000, penalty='l2', solver='saga')
```

```
lr.fit(X_train_s, y_train.values.ravel())
```

```
LogisticRegressionCV(Cs=1, solver='saga')
```

The resulting fitted object was evaluated on the training data.

```
lr_train_pred = lr.predict(X_train_s)
```

```
print(classification_report(y_train, lr_train_pred))
```

```
              precision    recall  f1-score   support

           1       0.57      0.51      0.54      2747
           2       0.49      0.53      0.51      2747
           3       0.55      0.42      0.47      2747
           4       0.70      0.83      0.76      2747
           5       0.61      0.61      0.61      2747
           6       0.54      0.61      0.58      2747
           7       0.86      0.83      0.84      2747

    accuracy                           0.62     19229
   macro avg       0.62      0.62      0.61     19229
weighted avg       0.62      0.62      0.61     19229
```

The classification report shows that Logistic Regression was only mildly successful in predicting the values in the training data. It had no particular prediction issues with any specific class.

Next, the Logistic Regression was evaluated on the testing data.

```
lr_test_pred = lr.predict(X_test_s)
```

```
print(classification_report(y_test, lr_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.64      0.51      0.56    209054
           2       0.69      0.51      0.59    277890
           3       0.63      0.42      0.50     33594
           4       0.04      0.81      0.08       587
           5       0.05      0.60      0.10      7101
           6       0.25      0.60      0.36     15207
           7       0.36      0.81      0.50     18350

    accuracy                           0.52    561783
   macro avg       0.38      0.61      0.38    561783
weighted avg       0.64      0.52      0.56    561783
```

The performance of the estimator has dropped considerably. Its success on different classes has also suffered. Results for classes 4 and 5 are abysmal. Note that these are the classes with the least representation in the dataset as is evident from the support column.

Decision Tree:

Next, a decision tree classifier was used on the dataset. No values for the estimator were specified. It used the default values for all parameters.

```
dt = DecisionTreeClassifier(random_state=4)
```

```
dt.fit(X_train_s, y_train)
```

```
DecisionTreeClassifier(random_state=4)
```

The decision tree classifier was evaluated on the training data.

```
dt_train_pred = dt.predict(X_train_s)
```

```
print(classification_report(y_train, dt_train_pred))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00      2747
           2       1.00      1.00      1.00      2747
           3       1.00      1.00      1.00      2747
           4       1.00      1.00      1.00      2747
           5       1.00      1.00      1.00      2747
           6       1.00      1.00      1.00      2747
           7       1.00      1.00      1.00      2747

    accuracy                           1.00     19229
   macro avg       1.00      1.00      1.00     19229
weighted avg       1.00      1.00      1.00     19229
```

The estimator predicted all values for the training data perfectly! This is a case of overfitting and is a problem that plagues decision trees.

Next, the decision tree was evaluated on the testing data. Note that performance of the decision tree may have been impacted by the overfitting.

```
dt_test_pred = dt.predict(X_test_s)
```

```
print(classification_report(y_test, dt_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.70      0.70      0.70    209054
           2       0.79      0.65      0.71    277890
           3       0.77      0.80      0.78     33594
           4       0.26      1.00      0.41       587
           5       0.21      0.93      0.35      7101
           6       0.50      0.84      0.62     15207
           7       0.52      0.95      0.67     18350

    accuracy                           0.69    561783
   macro avg       0.53      0.84      0.61    561783
weighted avg       0.73      0.69      0.70    561783
```

The estimator's success dropped but it still did a decent job of predicting values. The results for classes 4 and 5, though not as good as other classes are still better than the Logistic Regression model. Overall, a decent score.

K-Nearest Neighbors:

After decision tree, a K-Nearest Neighbors classifier was used. This classifier also had default values, but made use of features of 5 nearest neighbors for its estimation.

```
knn = KNeighborsClassifier(n_neighbors=5)
```

```
knn.fit(X_train_s, y_train.values.ravel())
```

```
KNeighborsClassifier()
```

The estimator was then evaluated on the training set.

```
knn_train_pred = knn.predict(X_train_s)
```

```
print(classification_report(y_train, knn_train_pred))
```

```
              precision    recall  f1-score   support

           1       0.82      0.81      0.82      2747
           2       0.83      0.71      0.77      2747
           3       0.87      0.82      0.84      2747
           4       0.91      0.98      0.94      2747
           5       0.88      0.97      0.92      2747
           6       0.86      0.88      0.87      2747
           7       0.94      0.97      0.96      2747

    accuracy                           0.88     19229
   macro avg       0.87      0.88      0.87     19229
weighted avg       0.87      0.88      0.87     19229
```

The estimator here performed decently. It did not do as well as the Decision Tree classifier but much better than Logistic Regression classifier. As the Decision Tree classifier overfit the training data, these results are decent.

Next, the testing data was used for evaluation.

```
knn_test_pred = knn.predict(X_test_s)
```

```
print(classification_report(y_test, knn_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.71      0.71      0.71    209054
           2       0.82      0.59      0.69    277890
           3       0.72      0.72      0.72     33594
           4       0.13      0.99      0.22       587
           5       0.13      0.95      0.23      7101
           6       0.42      0.81      0.55     15207
           7       0.50      0.95      0.65     18350

    accuracy                           0.66    561783
   macro avg       0.49      0.82      0.54    561783
weighted avg       0.74      0.66      0.69    561783
```

The performance of the estimator dropped considerably but is still reasonably good. Although its performance is not as good as Decision Tree, it does a decent job nevertheless.

Once more, the culprits are the under represented classes 4 and 5. Note the high recall but low accuracy scores. Clearly, the classifier can predict all instances of these classes but tends to mislabel other classes as 4 and 5 too many times.

Support Vector Machine:

The next classifier was a Support Vector Machine (SVM). The SVM was initialized with an 'rbf' or radial based function kernel.

```
svm = SVC(kernel='rbf')
```

```
svm.fit(X_train_s, y_train.values.ravel())
```

```
SVC()
```

The SVM was evaluated on the training data first.

```
svm_train_pred = svm.predict(X_train_s)
```

```
print(classification_report(y_train, svm_train_pred))
```

```
              precision    recall  f1-score   support

           1       0.70      0.73      0.71      2747
           2       0.67      0.58      0.62      2747
           3       0.71      0.62      0.66      2747
           4       0.81      0.95      0.88      2747
           5       0.79      0.84      0.81      2747
           6       0.69      0.69      0.69      2747
           7       0.91      0.89      0.90      2747

    accuracy                           0.76     19229
   macro avg       0.75      0.76      0.75     19229
weighted avg       0.75      0.76      0.75     19229
```

The SVM does a decent job of predicting classes on the training data. Clearly, it is not overfitting.

Next, the SVM was evaluated on the testing data.

```
svm_test_pred = svm.predict(X_test_s)
```

```
print(classification_report(y_test, svm_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.70      0.71      0.71    209054
           2       0.82      0.56      0.66    277890
           3       0.67      0.61      0.64     33594
           4       0.08      0.94      0.15       587
           5       0.10      0.84      0.18      7101
           6       0.35      0.67      0.46     15207
           7       0.48      0.88      0.62     18350

    accuracy                           0.64    561783
   macro avg       0.46      0.75      0.49    561783
weighted avg       0.73      0.64      0.67    561783
```

The performance has dropped somewhat but not by much. The SVM too has trouble predicting Classes 4 and 5. Otherwise it does a decent job.

At this stage, Logistic Regression is the worst model. All three estimators other than Logistic regression have outputted values close to each other and okay overall.

Performance Comparison:

A comparison of the performance of different estimators utilized until this stage based on f1-scores is undertaken. The results are tabulated:

	Logistic Regression	Decision Tree	K-Neighbors	Support Vector Machine
Micro_test	0.518246	0.693421	0.663929	0.637955
Macro_test	0.384057	0.605297	0.539205	0.489365
Weighted_test	0.558614	0.703074	0.685336	0.665271
Micro_train	0.617349	1.000000	0.875865	0.757190
Macro_train	0.614005	1.000000	0.873774	0.753734
Weighted_train	0.614005	1.000000	0.873774	0.753734

Three different approaches for scoring the results were undertaken for better understanding of the results.

The results indicate that Decision Tree is the best classifier of all the considered classifiers. But it is overfitting. To deal with this, ensemble methods can be used.

Bagging:

Bagging is an ensemble approach for classification which takes samples of data and fits them all to different instances of an estimator. As each instance evaluates a different data sample it produces slightly different results. The results for each data point from every classifier that evaluated it are averaged to assign it a class label.

The BaggingClassifier is initialized with the DecisionTree classifier used previously as the base estimator. Bootstrap parameter is set to True to enable sampling with replacement, meaning data values will be evaluated by multiple instances of the base classifier.

```
bag = BaggingClassifier(base_estimator=dt, random_state=0, bootstrap=True)
```

```
bag.fit(X_train_s, y_train.values.ravel())
```

```
BaggingClassifier(base_estimator=DecisionTreeClassifier(random_state=4),
                  random_state=0)
```

The estimator is evaluated on the training data.

```
bag_train_pred = bag.predict(X_train_s)
```

```
print(classification_report(y_train, bag_train_pred))
```

```
              precision    recall  f1-score   support

           1       0.98      1.00      0.99      2747
           2       1.00      0.98      0.99      2747
           3       0.99      1.00      1.00      2747
           4       1.00      1.00      1.00      2747
           5       1.00      1.00      1.00      2747
           6       1.00      0.99      0.99      2747
           7       1.00      1.00      1.00      2747

    accuracy                           0.99     19229
   macro avg       0.99      0.99      0.99     19229
weighted avg       0.99      0.99      0.99     19229
```

A slight improvement in overfitting is observed but nonetheless, the classifier still seems to be overfitting.

Next, the classifier is evaluated on the testing set.

```
bag_test_pred = bag.predict(X_test_s)
```

```
print(classification_report(y_test, bag_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.74      0.79      0.76    209054
           2       0.86      0.67      0.75    277890
           3       0.80      0.86      0.83     33594
           4       0.30      1.00      0.46       587
           5       0.25      0.96      0.40      7101
           6       0.56      0.88      0.68     15207
           7       0.61      0.96      0.75     18350

    accuracy                           0.75    561783
   macro avg       0.59      0.88      0.66    561783
weighted avg       0.78      0.75      0.76    561783
```

The estimator has produced better results. The bagging method seems to have been successful. These results are better than the Decision Tree classifier. There are still some problems with Classes 4 and 5 though but not as much as before.

Random Forest:

Random Forest is the next method to be tested. This is an advanced version of bagging where in addition to randomly choosing samples from the data. Feature selection too is randomized. One sample may have only some features from the original dataset. This produces more randomization and should produce better results.

The Random Forest classifier is initialized with default values. No changes are made to the base parameters. Random Forest uses decision trees for estimation.

```
rf = RandomForestClassifier()
```

```
rf.fit(X_train_s, y_train.values.ravel())
```

```
RandomForestClassifier()
```

The estimator is then evaluated on the training data.

```
rf_train_pred = rf.predict(X_train_s)
```

```
print(classification_report(y_train, rf_train_pred))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00      2747
           2       1.00      1.00      1.00      2747
           3       1.00      1.00      1.00      2747
           4       1.00      1.00      1.00      2747
           5       1.00      1.00      1.00      2747
           6       1.00      1.00      1.00      2747
           7       1.00      1.00      1.00      2747

    accuracy                           1.00     19229
   macro avg       1.00      1.00      1.00     19229
weighted avg       1.00      1.00      1.00     19229
```

The estimator predicts the training data perfectly indicating overfitting.

Next, the estimator's predictions for the test data are evaluated against actual values.

```
rf_test_pred = rf.predict(X_test_s)
```

```
print(classification_report(y_test, rf_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.78      0.80      0.79    209054
           2       0.88      0.72      0.79    277890
           3       0.81      0.85      0.83     33594
           4       0.25      1.00      0.40       587
           5       0.24      0.97      0.39      7101
           6       0.56      0.91      0.69     15207
           7       0.61      0.98      0.75     18350

    accuracy                           0.77    561783
   macro avg       0.59      0.89      0.66    561783
weighted avg       0.81      0.77      0.78    561783
```

There has been a slight improvement in results compared to Bagging. Oddly, while Random Forest also has trouble predicting Class 4 and 5, it seems to have done a worse job in this regard than the Bagging Classifier.

Extra Trees:

Extra Trees Classifier is an approach to estimation which tends to randomize the splits of trees in a random forest in addition to random selection of rows and features. The Extra Trees Classifier is initialized with default values:

```
et = ExtraTreesClassifier()
```

```
et.fit(X_train_s, y_train.values.ravel())
```

```
ExtraTreesClassifier()
```

The training data is evaluated by the classifier first.

```
et_train_pred = et.predict(X_train_s)
```

```
print(classification_report(y_train, et_train_pred))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00      2747
           2       1.00      1.00      1.00      2747
           3       1.00      1.00      1.00      2747
           4       1.00      1.00      1.00      2747
           5       1.00      1.00      1.00      2747
           6       1.00      1.00      1.00      2747
           7       1.00      1.00      1.00      2747

    accuracy                           1.00     19229
   macro avg       1.00      1.00      1.00     19229
weighted avg       1.00      1.00      1.00     19229
```

Once more, overfitting to the training data is observed.

Next, the classifier is evaluated on the testing data.

```
et_test_pred = et.predict(X_test_s)
```

```
print(classification_report(y_test, et_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.78      0.80      0.79    209054
           2       0.87      0.72      0.79    277890
           3       0.81      0.85      0.83     33594
           4       0.25      1.00      0.40       587
           5       0.24      0.97      0.39      7101
           6       0.56      0.91      0.70     15207
           7       0.62      0.98      0.76     18350

    accuracy                           0.77    561783
   macro avg       0.59      0.89      0.66    561783
weighted avg       0.81      0.77      0.78    561783
```

The results are nearly indistinguishable from the Random Forest approach with same problems for Classes 4 and 5.

Gradient Boosting:

Next, boosting methods are used, Boosting involves taking weak classifiers and chaining their output to achieve better results. First, a Gradient Boosting Classifier is used. The classifier is initialized with default values.

```
gbc = GradientBoostingClassifier()
```

```
gbc.fit(X_train_s, y_train.values.ravel())
```

```
GradientBoostingClassifier()
```

The classifier is evaluated first on the training dataset.

```
gbc_train_pred = gbc.predict(X_train_s)
```

```
print(classification_report(y_train, gbc_train_pred))
```

```
              precision    recall  f1-score   support

           1       0.77      0.76      0.76      2747
           2       0.79      0.63      0.70      2747
           3       0.80      0.77      0.78      2747
           4       0.92      0.97      0.95      2747
           5       0.83      0.92      0.87      2747
           6       0.78      0.82      0.80      2747
           7       0.91      0.96      0.93      2747

    accuracy                           0.83     19229
   macro avg       0.83      0.83      0.83     19229
weighted avg       0.83      0.83      0.83     19229
```

The results are reasonably good. It seems that this classifier did not overfit the data.

Next, the classifier is evaluated on the testing data.

```
gbc_test_pred = gbc.predict(X_test_s)
```

```
print(classification_report(y_test, gbc_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.71      0.71      0.71    209054
           2       0.83      0.57      0.68    277890
           3       0.75      0.73      0.74     33594
           4       0.18      0.96      0.30       587
           5       0.12      0.91      0.21      7101
           6       0.42      0.79      0.55     15207
           7       0.43      0.94      0.59     18350

    accuracy                           0.66    561783
   macro avg       0.49      0.80      0.54    561783
weighted avg       0.75      0.66      0.68    561783
```

These results are not promising. The classifier has not offered any improvements, suffering from issues for Classes 4 and 5.

Ada Boosting:

Ada Boosting is a boosting algorithm where the weak classifier can be chosen. In this case, it is initialized with the bagging classifier that has a decision tree as the base estimator.

```
abc = AdaBoostClassifier(bag)
```

```
abc.fit(X_train_s, y_train.values.ravel())
```

```
AdaBoostClassifier(base_estimator=BaggingClassifier(base_estimator=DecisionTreeClassifier(random_state=4),
                                                    random_state=0))
```

The classifier is evaluated on training data first.

```
abc_train_pred = abc.predict(X_train_s)
```

```
print(classification_report(y_train, abc_train_pred))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00      2747
           2       1.00      1.00      1.00      2747
           3       1.00      1.00      1.00      2747
           4       1.00      1.00      1.00      2747
           5       1.00      1.00      1.00      2747
           6       1.00      1.00      1.00      2747
           7       1.00      1.00      1.00      2747

    accuracy                           1.00     19229
   macro avg       1.00      1.00      1.00     19229
weighted avg       1.00      1.00      1.00     19229
```

The results show overfitting.

Next, the classifier is evaluated on testing data.

```
abc_test_pred = abc.predict(X_test_s)
```

```
print(classification_report(y_test, abc_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.76      0.79      0.78    209054
           2       0.87      0.73      0.79    277890
           3       0.82      0.88      0.85     33594
           4       0.31      1.00      0.48       587
           5       0.31      0.97      0.47      7101
           6       0.61      0.91      0.73     15207
           7       0.61      0.98      0.75     18350

    accuracy                           0.78    561783
   macro avg       0.62      0.89      0.69    561783
weighted avg       0.80      0.78      0.78    561783
```

The result shows a slight improvement with problems in evaluating classes 4 and 5 properly.

Voting Classifier:

The Voting Classifier takes a list of estimators and applies each of them to the data. It subsequently takes ‘votes; from each classifier on which label to assign a data point and

the label with the highest vote is assigned. In this case, many different estimators were tried but the best results were given by Extra Trees and Random Forest classifiers.

```
estimators = [('et', et), ('rf', rf)]
```

```
vc = VotingClassifier(estimators, voting='hard')
```

```
vc.fit(X_train_s, y_train.values.ravel())
```

```
VotingClassifier(estimators=[('et', ExtraTreesClassifier()),
                             ('rf', RandomForestClassifier())])
```

The classifiers were evaluated on training data first.

```
vc_train_pred = vc.predict(X_train_s)
```

```
print(classification_report(y_train, vc_train_pred))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00      2747
           2       1.00      1.00      1.00      2747
           3       1.00      1.00      1.00      2747
           4       1.00      1.00      1.00      2747
           5       1.00      1.00      1.00      2747
           6       1.00      1.00      1.00      2747
           7       1.00      1.00      1.00      2747

    accuracy                           1.00     19229
   macro avg       1.00      1.00      1.00     19229
weighted avg       1.00      1.00      1.00     19229
```

The results again show overfitting.

Next, the classifiers were evaluated on the testing data.

```
vc_test_pred = vc.predict(X_test_s)
```

```
print(classification_report(y_test, vc_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.76      0.83      0.79    209054
           2       0.89      0.70      0.79    277890
           3       0.81      0.88      0.84     33594
           4       0.27      1.00      0.43       587
           5       0.29      0.97      0.45      7101
           6       0.62      0.89      0.73     15207
           7       0.66      0.97      0.79     18350

    accuracy                           0.78    561783
   macro avg       0.61      0.89      0.69    561783
weighted avg       0.81      0.78      0.79    561783
```

The results are almost the same as the AdaBoost classifier with same issues in Class 4 and 5.

Stacking Classifier:

The final classifier to be used for this analysis is the Stacking Classifier. This is a meta-estimator which takes data from multiple estimators and feeds it to the final estimator. Once more, several combinations of estimators and final estimators were tried and the best one was chosen. In this case, Logistic Regression, K-Nearest Neighbors and Extra Trees Classifier were used as estimators and Random Forest Classifier was used as the final estimator.

```
estimators2 = [('lr', lr), ('knn', knn), ('et', et)]
```

```
sta2 = StackingClassifier(estimators=estimators2, final_estimator=RandomForestClassifier())
```

```
sta2.fit(X_train_s, y_train.values.ravel())
```

```
StackingClassifier(estimators=[('lr', LogisticRegression(max_iter=1000)),
                               ('knn', KNeighborsClassifier()),
                               ('et', ExtraTreesClassifier())],
                   final_estimator=RandomForestClassifier())
```

The meta-estimator was first evaluated on training data.

```
sta2_train_pred = sta2.predict(X_train_s)
```

```
print(classification_report(y_train, sta2_train_pred))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00      2747
           2       1.00      0.99      1.00      2747
           3       1.00      1.00      1.00      2747
           4       1.00      1.00      1.00      2747
           5       1.00      1.00      1.00      2747
           6       1.00      1.00      1.00      2747
           7       1.00      1.00      1.00      2747

    accuracy                           1.00     19229
   macro avg       1.00      1.00      1.00     19229
weighted avg       1.00      1.00      1.00     19229
```

Again, overfitting is evident from the results.

The meta-estimator was then evaluated on testing data.

```
sta2_test_pred = sta2.predict(X_test_s)
```

```
print(classification_report(y_test, sta2_test_pred))
```

```
              precision    recall  f1-score   support

           1       0.79      0.79      0.79    209054
           2       0.86      0.75      0.80    277890
           3       0.80      0.87      0.83     33594
           4       0.29      1.00      0.45       587
           5       0.32      0.96      0.48      7101
           6       0.62      0.91      0.74     15207
           7       0.66      0.97      0.78     18350

    accuracy                           0.79    561783
   macro avg       0.62      0.89      0.70    561783
weighted avg       0.81      0.79      0.79    561783
```

There is a slight improvement in the overall results. While Class 4 and 5 remain the bottlenecks, they are predicted with somewhat greater accuracy now.

Results:

The classification results indicated that when using lone estimators, the results for Decision Tree, Support Vector Machine and K Nearest Neighbors were decent with the undersampled dataset as the training set. Logistic Regression was not very good at prediction though.

With the addition of Bagging and Boosting techniques results improved but marginally in the end. The bottleneck for improvement of results remained the ability of the classifiers to predict Class Labels 4 and 5. The issue seems to have stemmed from the undersampling itself. As the results showed, the recall for Classes 4 and 5 was close to 1, but it was precision that remained the problem.

To put it differently, the classifier mistook many labels as belonging to Classes 4 and 5 which in fact belonged to other classes. This may be because undersampled data introduced bias into the classifiers by making the underrepresented classes in actual data appear more important.

The fact that results did not improve much after initial bagging techniques may also be because natural error in the dataset is present which puts a hard limit on the ability of classifiers to get correct results.

Key Findings:

The key findings from this analysis are as follows:

1. Downsampling a dataset can still produce a reasonably decent training dataset for classification.
2. Downsampling introduces a bias in favor of the underrepresented classes which impacts predictions for other classes. This impacts precision of the classifier for underrepresented classes negatively. This might not be problematic for certain applications.
3. Ensemble methods can improve prediction results for downsampled datasets but only to a certain extent.
4. Although Decision Tree methods tend to overfit training data, those methods are still quite successful for working with downsampled data, producing reasonably good results.
5. Downsampling improves the recall for underrepresented classes significantly even if the precision is low. The recall is close to perfect. This might be relevant for some applications.

Issues:

There were several issues with the analysis which need to be taken into account for future attempts to improve the results. These issues are:

1. The technique used for this analysis used a RandomUnderSampler object. This is not the only undersampling technique available. There are many other undersampling techniques such as Edited Nearest Neighbors, Tomek Links, Near Miss etc. These techniques in all likelihood would have produced different datasets.
2. The classifiers used in the analysis were initialized with their default values. There are many hyper-parameters for these classifiers that can be tuned to include various features such as regularization. It is very likely that tuning these hyperparameter may produce better results.
3. The analysis here only focused on the downsampled dataset. No comparison with any other version of the dataset was made.

Future Steps:

Some future steps for dealing with issues outlined above are listed below:

1. A more thorough analysis on downsampling will involve making use of multiple downsampling techniques and use the resulting datasets with different classifiers and classification techniques.
2. The classifiers to be used for this analysis can be tuned by making use of different hyperparameter combinations to possibly get improved results.
3. In addition to downsampling, results can be obtained from performing classification on the unsampled dataset and a comparison between results from classifying downsampled dataset can be made.
4. Another approach would be to make use of an upsampled dataset and see the results of classification for that dataset also.

The steps outlined above, if implemented separately, may yield interesting insights on their own. However, these steps can also be combined in any combination to produce results that could further enrich this analysis.