resample_v2.ipynb | ensemble_v2.ipynb

Code | Python 3 (ipykernel)

Credit Risk Resampling Techniques

Classification writeup

[1]:
```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
from pathlib import Path
from collections import Counter
# Load the data
file_path = Path('Data/lending_data.csv')
df = pd.read_csv(file_path)
df.head()
```

[1]:

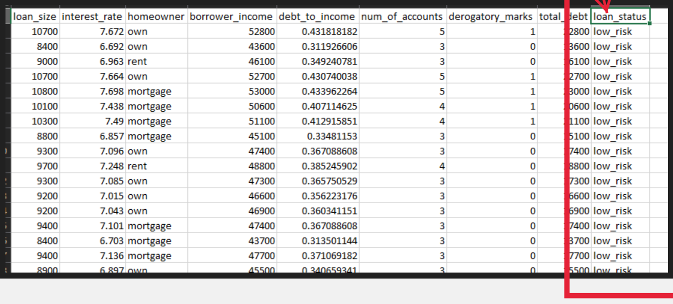
	loan_size	interest_rate	homeowner	borrower_income	debt_to_income	num_of_accounts	derogatory_marks	total_debt	loan_status
0	10700.0	7.672	own	52800	0.431818	5	1	22800	low_risk
1	8400.0	6.692	own	43600	0.311927	3	0	13600	low_risk
2	9000.0	6.963	rent	46100	0.349241	3	0	16100	low_risk
3	10700.0	7.664	own	52700	0.430740	5	1	22700	low_risk
4	10800.0	7.698	mortgage	53000	0.433962	5	1	23000	low_risk

Split the Data into Training and Testing

binary classification

[2]:
```
# Create our features
# Transform homeowner column
def changehomeowner(homeowner):
    if homeowner == "own":
        return 0
    else:
        return 1

def loan_status(loan_status):
    if loan_status == "low_risk":
        return 0
    else:
        return 1

df["homeowner"] = df["homeowner"].apply(changehomeowner)
df["loan_status"] = df["loan_status"].apply(loan_status)
x_cols = [i for i in df.columns if i not in ('loan_status')]
X = df[x_cols]

# Create our target
y = df['loan_status']
X.describe()
```

target variable



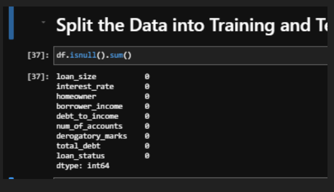
loan_size	interest_rate	homeowner	borrower_income	debt_to_income	num_of_accounts	derogatory_marks	total_debt	loan_status
10700	7.672	own	52800	0.431818182	5	1	2800	low_risk
8400	6.692	own	43600	0.311926606	3	0	3600	low_risk
9000	6.963	rent	46100	0.349240781	3	0	6100	low_risk
10700	7.664	own	52700	0.430740038	5	1	2700	low_risk
10800	7.698	mortgage	53000	0.433962264	5	1	3000	low_risk
10100	7.438	mortgage	50600	0.407114625	4	1	0600	low_risk
10300	7.49	mortgage	51100	0.412915851	4	1	1100	low_risk
8800	6.857	mortgage	45100	0.33481153	3	0	5100	low_risk
9300	7.096	own	47400	0.367088608	3	0	7400	low_risk
9700	7.248	rent	48800	0.385245902	4	0	8800	low_risk
9300	7.085	own	47300	0.365750529	3	0	7300	low_risk
9200	7.015	own	46600	0.356223176	3	0	6600	low_risk
9200	7.043	own	46900	0.360341151	3	0	6900	low_risk
9400	7.101	mortgage	47400	0.367088608	3	0	7400	low_risk
8400	6.703	mortgage	43700	0.313501144	3	0	3700	low_risk
9400	7.136	mortgage	47700	0.371069182	3	0	7700	low_risk
8900	6.897	own	45500	0.340659341	3	0	5500	low_risk

no missing data

[2]:

	loan_size	interest_rate	homeowner	borrower_income	debt_to_income	num_of_accounts	derogatory_marks	total_debt
count	77536.000000	77536.000000	77536.000000	77536.000000	77536.000000	77536.000000	77536.000000	77536.000000
mean	9805.562577	7.292333	0.601089	49221.949804	0.377318	3.826610	0.392308	19221.949804
std	2093.223153	0.889495	0.489678	8371.635077	0.081519	1.904426	0.582086	8371.635077
min	5000.000000	5.250000	0.000000	30000.000000	0.000000	0.000000	0.000000	0.000000
25%	8700.000000	6.825000	0.000000	44800.000000	0.330357	3.000000	0.000000	14800.000000
50%	9500.000000	7.172000	1.000000	48100.000000	0.376299	4.000000	0.000000	18100.000000
75%	10400.000000	7.528000	1.000000	51400.000000	0.416342	4.000000	1.000000	21400.000000
max	23800.000000	13.235000	1.000000	105200.000000	0.714829	16.000000	3.000000	75200.000000

added null check to block 1



[3]:
```
# Check the balance of our target values
y.value_counts()

print('Check: count total variables in X and y datasets:')
def population_check(X,y):
    count = "{:,}".format(len(X))
    if len(X) == len(y):
        print(f'X and y variable counts match without error at {count} datasets')
    else:
        print('ERROR, recheck X and y variable counts..')
print('')
population_check(X,y)
```

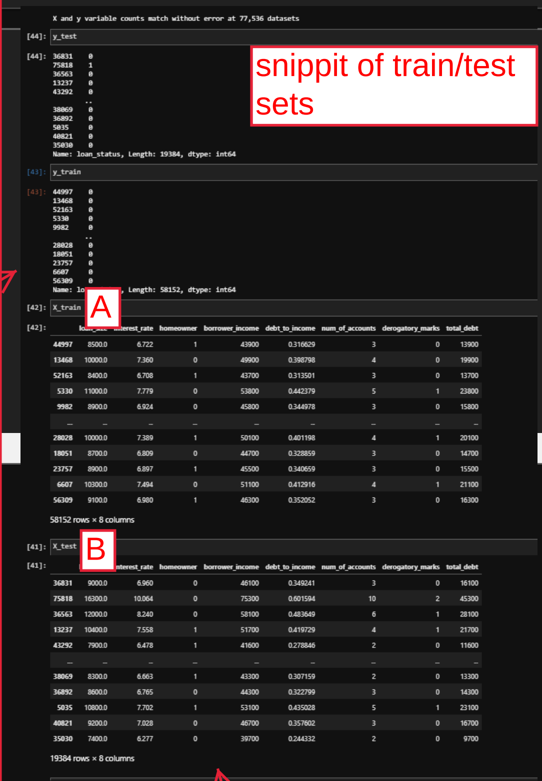
```
Check: count total variables in X and y datasets:

X and y variable counts match without error at 77,536 datasets
```

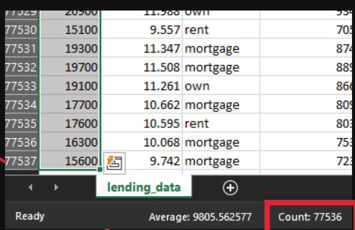
resample_v2.ipynb | ensemble_v2.ipynb | Code | Python 3 (ipykernel)

oversample with **standard scaler**

[4]:
```python
# Create X_train, X_test, y_train, y_test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    random_state=1,
                                                    stratify=y)

print(f'X_train has a shape of {X_train.shape}')
print('')
print(f'X_test has a shape of {X_test.shape}')
print('')
print(f'y_train has a shape of {y_train.shape}')
print('')
print(f'y_test has a shape of {y_test.shape}')
```

```
X_train has a shape of (58152, 8)

X_test has a shape of (19384, 8)

y_train has a shape of (58152,)

y_test has a shape of (19384,)
```





77,536

Data Pre-Processing

Scale the training and testing data using the `StandardScaler` from `sklearn`. Remember that when scaling the data, you only scale the features data (`X_train` and `X_testing`).

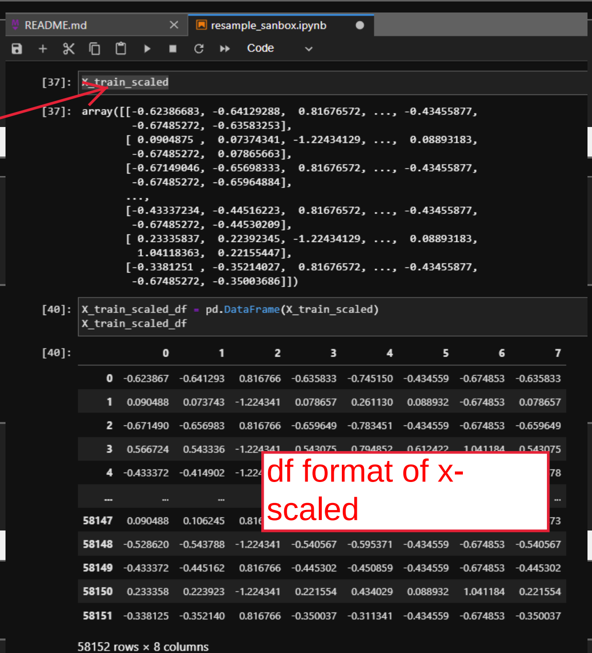
A B

only scale X variables

[5]:
```python
# Create the StandardScaler instance
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

[6]:
```python
# Fit the Standard Scaler with the training data
# When fitting scaling functions, only train on the training dataset
X_scaler = scaler.fit(X_train)
```

[7]:
```python
# Scale the training and testing data
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```



raw output = numpy array (list of lists)

df format of x-scaled

Simple Logistic Regression

[8]:
```python
from sklearn.linear_model import LogisticRegression
model_simple = LogisticRegression(solver='lbfgs', random_state=1)
model_simple.fit(X_train, y_train)
```

[8]: LogisticRegression(random_state=1)

[9]:
```python
# Calculated the balanced accuracy score
from sklearn.metrics import balanced_accuracy_score
y_pred = model_simple.predict(X_test)
balanced_accuracy_score(y_test, y_pred)
```

[9]: 0.9442676901753825

Precision:
87% of **predictions from x_test** (predicting high risk loans) were actually high risk. (100% precision for low risk '0')

[10]:
```python
# Display the confusion matrix
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_pred)
```

Recall:
87% of all high risk loans 1's **in total population (y_test & y_train)** were predicted correctly. 100% for low risk loans

[10]:
```
array([[18679,    80],
       [   67,   558]], dtype=int64)
```

STANDARD SCALER

Specificity:
89% of low risk loans were predicted as low risk. No high risk loans were predicted as low risk

[11]:
```python
# Print the imbalanced classification report
from imblearn.metrics import classification_report_imbalanced
print(classification_report_imbalanced(y_test, y_pred))
```

	pre	rec	spe	f1	geo	iba	sup
0	1.00	1.00	0.89	1.00	0.94	0.90	18759
1	0.87	0.89	1.00	0.88	0.94	0.88	625
avg / total	0.99	0.99	0.90	0.99	0.94	0.90	19384

resample_v2.ipynb × ensemble_v2.ipynb

Code Python 3 (ipykernel)

oversample with **over sampler**

Oversampling

In this section, you will compare two oversampling algorithms to determine which algorithm results in the best performance. You will oversample the data using the naive random oversampling algorithm and the SMOTE algorithm. For each algorithm, be sure to complete the folliowing steps:

1. View the count of the target classes using `Counter` from the collections library.
2. Use the resampled data to train a logistic regression model.
3. Calculate the balanced accuracy score from sklearn.metrics.
4. Print the confusion matrix from sklearn.metrics.
5. Generate a classication report using the `imbalanced_classification_report` from imbalanced-learn.

Note: Use a random state of 1 for each sampling algorithm to ensure consistency between tests

Naive Random Oversampling

[12]:
```
# Resample the training data with the RandomOversampler
from imblearn.over_sampling import RandomOverSampler
ros = RandomOverSampler(random_state=1)
X_resampled, y_resampled = ros.fit_resample(X_train,y_train)
# View the count of target classes with Counter
Counter(y_resampled)
```

[12]:
```
Counter({0: 56277, 1: 56277})
```

[13]:
```
# Train the Logistic Regression model using the resampled data
model_naive = LogisticRegression(solver='lbfgs', random_state=1)
model_naive.fit(X_resampled,y_resampled)
```

Precision:
87% of **predictions from x_test** (predicting high risk loans) were actually high risk. (100% precision for low risk'0')

[13]:
```
LogisticRegression(random_state=1)
```

[14]:
```
# Calculated the balanced accuracy score
balanced_accuracy_score(y_test,y_pred)
```

OVER SAMPLER

Recall:
89% of all high risk loans 1's **in total population (y_test & y_train)** were predicted correctly. 100% for low risk loans

[14]:
```
0.9442676901753825
```

[15]:
```
# Display the confusion matrix
print(confusion_matrix(y_test,y_pred))
```

```
[[18679    80]
 [   67   558]]
```

Specificity:
89% of low risk loans were predicted as low risk. No high risk loans were predicted as low risk

[16]:
```
# Print the imbalanced classification report
print(classification_report_imbalanced(y_test,y_pred))
```

	pre	rec	spe	f1	geo	iba	sup
0	1.00	1.00	0.89	1.00	0.94	0.90	18759
1	0.87	0.89	1.00	0.88	0.94	0.88	625
avg / total	0.99	0.99	0.90	0.99	0.94	0.90	19384

SMOTE Oversampling

total resample (oversample and undersample) with **SMOTE**

[17]:
```
# Resample the training data with SMOTE
from imblearn.over_sampling import SMOTE

X_resampled, y_resampled = SMOTE(random_state=1, sampling_strategy=1.0).fit_resample(
    X_train,y_train
)

# View the count of target classes with Counter
Counter(y_resampled)
```

[17]:
```
Counter({0: 56277, 1: 56277})
```

[18]:
```
# Train the Logistic Regression model using the resampled data
model_smote = LogisticRegression(solver='lbfgs', random_state=1)
model_smote.fit(X_resampled,y_resampled)
```

[18]:
```
LogisticRegression(random_state=1)
```

[19]:
```
# Calculated the balanced accuracy score
y_pred = model_smote.predict(X_test)
balanced_accuracy_score(y_test,y_pred)
```

[19]:
```
0.9959744975744975
```

accuracy is almost 100% **SMOTE**

[20]:
```
# Display the confusion matrix
confusion_matrix(y_test,y_pred)
```

[20]:
```
array([[18668,    91],
       [    2,   623]], dtype=int64)
```

```
         [    2,   623]], dtype=int64)
```

Precision:
87% of **predictions from x_test** (predicting high risk loans) were actually high risk. (100% precision for low risk)

```
[21]: # Print the imbalanced classification report
      print(classification_report_imbalanced(y_test,y_pred))
```

SMOTE

	pre	rec	spe	f1	geo	ib
0	1.00	1.00	1.00	1.00	1.00	0.9
1	0.87	1.00	1.00	0.93	1.00	0.9
avg / total	1.00	1.00	1.00	1.00	1.00	0.9

Recall:
100% of all high risk loans 1's **in total population (y_test & y_train)** were predicted correctly. 100% for low risk loans

Specificity:
100% of low risk loans were predicted as low risk. No high risk loans were predicted as low risk

Undersampling

resample (undersample) with **ClusterCentroids**

In this section, you will test an und algorithm results in the best performance compared to the oversampling algorithms above. You Cluster Centroids algorithm and complete the folliowing steps:

1. View the count of the target classes using `Counter` from the collections library.
2. Use the resampled data to train a logistic regression model.
3. Calculate the balanced accuracy score from sklearn.metrics.
4. Display the confusion matrix from sklearn.metrics.
5. Generate a classication report using the `imbalanced_classification_report` from imbalanced-learn.

Note: Use a random state of 1 for each sampling algorithm to ensure consistency between tests

```
[22]: # Resample the data using the ClusterCentroids resampler
      from imblearn.under_sampling import ClusterCentroids

      cc = ClusterCentroids(random_state=1)
      X_resampled, y_resampled = cc.fit_resample(X_train,y_train)

      # View the count of target classes with Counter
      Counter(y_resampled)
```

google definition:
Cluster sampling is a probability sampling technique where researchers divide the population into multiple groups (clusters) for research.

```
[22]: Counter({0: 1875, 1: 1875})
```

```
[23]: # Train the Logistic Regression model using the resampled data
      model_undersampled = LogisticRegression(solver='lbfgs', random_state=1)
      model_undersampled.fit(X_resampled,y_resampled)
```

```
[23]: LogisticRegression(random_state=1)
```

```
[24]: # Calculate the balanced accuracy score
      balanced_accuracy_score(y_test,y_pred)
```

Precision:
87% of **predictions from x_test** (predicting high risk loans) were actually high risk. (100% precision for low risk '0')

```
[24]: 0.9959744975744975
```

```
[25]: # Display the confusion matrix
      y_pred = model_undersampled.predict(X_test)
      confusion_matrix(y_test,y_pred)
```

Recall:
98% of all high risk loans 1's **in total population (y_test & y_train)** were predicted correctly. 100% for low risk loans

```
[25]: array([[18670,    89],
             [   15,   610]], dtype=int64)
```

ClusterCentroids

```
[26]: # Print the imbalanced classification report
      print(classification_report_imbalanced(y_test,y_pred))
```

Specificity:
98% of low risk loans were predicted as low risk. No high risk loans were predicted as low risk

	pre	rec	spe	f1	geo		
0	1.00	1.00	0.98	1.00	0.99		
1	0.87	0.98	1.00	0.92	0.99		
avg / total	1.00	0.99	0.98	0.99	0.99	0.97	19384

resample_v2.ipynb | ensemble_v2.ipynb

Code | Python 3 (ipykernel)

Combination (Over and Under) Sampling

In this section, you will test a combination over- and under-sampling algorithm to determine if the algorithm results in the best performance compared to the other sampling algorithms above. You will resample the data using the SMOTEENN algorithm and complete the folliowing steps:

1. View the count of the target classes using `Counter` from the collections library.
2. Use the resampled data to train a logistic regression model.
3. Calculate the balanced accuracy score from sklearn.metrics.
4. Display the confusion matrix from sklearn.metrics.
5. Generate a classication report using the `imbalanced_classification_report` from imbalanced-learn.

Note: Use a random state of 1 for each sampling algorithm to ensure consistency between tests

[27]:
```
# Resample the training data with SMOTEENN
from imblearn.combine import SMOTEENN

smote_enn = SMOTEENN(random_state=0)
X_resampled, y_resampled = smote_enn.fit_resample(X,y)

# View the count of target classes with Counter
Counter(y_resampled)
```

[27]:
```
Counter({0: 74592, 1: 73663})
```

[28]:
```
# Train the Logistic Regression model using the resampled data
model_combo = LogisticRegression(solver='lbfgs', random_state=1)
model_combo.fit(X_resampled,y_resampled)
y_pred = model_combo.predict(X_test)
```

Precision:
87% of **predictions from x_test** (predicting high risk loans) were actually high risk. (100% precision for low risk '0')

[29]:
```
# Calculate the balanced accuracy score
balanced_accuracy_score(y_test,y_pred)
```

[29]:
```
0.9959211898288822
```

[30]:
```
# Display the confusion matrix
confusion_matrix(y_test,y_pred)
```

Recall:
100% of all high risk loans 1's **in total population (y_test & y_train)** were predicted correctly. 100% for low risk loans

[30]:
```
array([[18666,    93],
       [    2,   623]], dtype=int64)
```

SMOTEENN

[31]:
```
# Print the imbalanced classification report
print(classification_report_imbalanced(y_test,y_pred))
```

```
                 pre       rec       spe        f1       geo
         0       1.00      1.00      1.00      1.00      1.00
         1       0.87      1.00      1.00      0.93      1.00

avg / total      1.00      1.00      1.00      1.00      1.00
```

Specificity:
100% of low risk loans were predicted as low risk. No high risk loans were predicted as low risk

Final Questions

1. Which model had the best balanced accuracy score?

   YOUR ANSWER HERE.

2. Which model had the best recall score?

   YOUR ANSWER HERE.

3. Which model had the best geometric mean score?

   YOUR ANSWER HERE.

[ ]:

resample_v2.ipynb | ensemble_v2.ipynb

Code | Python 3 (ipykernel)

Ensemble Learning

Initial Imports

[1]:
```python
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```python
import numpy as np
import pandas as pd
from pathlib import Path
from collections import Counter
```

[3]:
```python
from sklearn.metrics import balanced_accuracy_score
from sklearn.metrics import confusion_matrix
from imblearn.metrics import classification_report_imbalanced
```

Read the CSV and Perform Basic Data Cleaning

[4]:
```python
# Load the data
file_path = Path('Data/LoanStats_2019Q1.csv')
df = pd.read_csv(file_path)

# Preview the data
df.head()
```

[4]:

	loan_amnt	int_rate	installment	home_ownership	annual_inc	verification_status	issue_d	loan_status	pymnt_plan	dti	...	pct_tl_nvr_dlq	percent_bc_gt_75	pub_rec_bankruptcies	tax_liens	tot_hi_cred
0	10500.0	0.1719	375.35	RENT	66000.0	Source Verified	Mar-2019	low_risk	n	27.24	...	85.7	100.0	0.0	0.0	65
1	25000.0	0.2000	929.09	MORTGAGE	105000.0	Verified	Mar-2019	low_risk	n	20.23	...	91.2	50.0	1.0	0.0	271
2	20000.0	0.2000	529.88	MORTGAGE	56000.0	Verified	Mar-2019	low_risk	n	24.26	...	66.7	50.0	0.0	0.0	60
3	10000.0	0.1640	353.55	RENT	92000.0	Verified	Mar-2019	low_risk	n	31.44	...	100.0	50.0	1.0	0.0	99
4	22000.0	0.1474	520.39	MORTGAGE	52000.0	Not Verified	Mar-2019	low_risk	n	18.76	...	100.0	0.0	0.0	0.0	219

5 rows × 86 columns

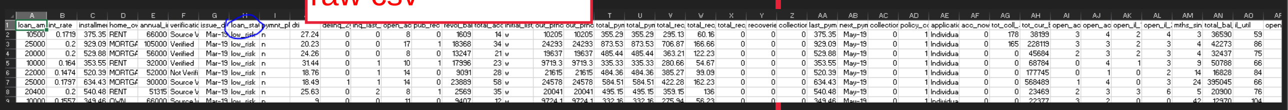
target variable (y)

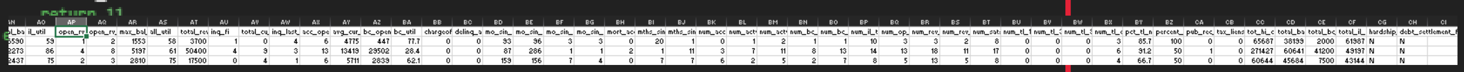
Split the Data into Training and Testing

[5]:
```python
#Clean data for model

#Convert issue date to numerical format
df['issue_month'] = df['issue_d'].str.split('-').str[0]

def rename_months(issue_month):
    if issue_month == "Jan":
        return 1
    elif issue_month == "Feb":
        return 2
    elif issue_month == "Mar":
        return 3
    elif issue_month == "Apr":
        return 4
    elif issue_month == "May":
        return 5
    elif issue_month == "Jun":
        return 6
    elif issue_month == "Jul":
        return 7
    elif issue_month == "Aug":
        return 8
    elif issue_month == "Sep":
        return 9
    elif issue_month == "Oct":
        return 10
    elif issue_month == "Nov":
        return 11
```

convert categorical data to numerical

raw csv





```python
df["issue_month"] = df["issue_month"].apply(rename_months)

#Alternative Method, use get_dummies
#df = pd.get_dummies(df, columns=['home_ownership','verification_status','initial_list_status','application_type'])

#Encode other categorical data
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

# Encoding home_ownership column
le.fit(df["home_ownership"])
df["home_ownership"] = le.transform(df["home_ownership"])

# Encoding initial_list_status column
le.fit(df["verification_status"])
df["verification_status"] = le.transform(df["verification_status"])

# Encoding initial_list_status column
le.fit(df["initial_list_status"])
df["initial_list_status"] = le.transform(df["initial_list_status"])

# Encoding application_type column
le.fit(df["application_type"])
df["application_type"] = le.transform(df["application_type"])

#Format target variable
def loan_status(loan_status):
    if loan_status == "low_risk":
        return 0
    else:
        return 1
df["loan_status"] = df["loan_status"].apply(loan_status)

#Drop columns in which values are consistent for entire dataset. There is no value in using these features.
df.drop(columns=['pymnt_plan','hardship_flag','debt_settlement_flag','issue_d','next_pymnt_d'], inplace=True)

# Create our features
x_cols = [i for i in df.columns if i not in ('loan_status')]
X = df[x_cols]

# Create our target
y = df['loan_status']
X.describe()
```

[5]:

	loan_amnt	int_rate	installment	home_ownership	annual_inc	verification_status	dti	delinq_2yrs	inq_last_6mths	open_acc	...	num_tl_op_past_12m	pct_tl_nvr_dlq	perce
count	68817.000000	68817.000000	68817.000000	68817.000000	6.881704e+04	68817.000000	68817.000000	68817.000000	68817.000000	68817.000000	...	68817.000000	68817.000000	6
mean	16677.594562	0.127718	480.652863	1.812779	8.821371e+04	0.669994	21.778153	0.217766	0.497697	12.587340	...	2.219423	95.057627	
std	10277.348590	0.048130	288.062432	0.941313	1.155800e+05	0.719105	20.199244	0.718367	0.758122	6.022869	...	1.897432	8.326426	
min	1000.000000	0.060000	30.890000	0.000000	4.000000e+01	0.000000	0.000000	0.000000	0.000000	2.000000	...	0.000000	20.000000	

resample_v2.ipynb | ensemble_v2.ipynb

Code | Python 3 (ipykernel)

[6]:
```python
# Check the balance of our target values
print('Check: count total variables in X and y datasets:')
def population_check(X,y):
    count = "{:,}".format(len(X))
    if len(X) == len(y):
        print(f'X and y variable counts match without error at {count} datasets')
    else:
        print('ERROR, recheck X and y variable counts..')
print('')
population_check(X,y)
```

BalancedRandomForestClassifier

```
Check: count total variables in X and y datasets:

X and y variable counts match without error at 68,817 datasets
```

[7]:
```python
# Split the X and y into X_train, X_test, y_train, y_test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    random_state=1,
                                                    stratify=y)
```

Data Pre-Processing

Scale the training and testing data using the `StandardScaler` from `sklearn`. Remember that when scaling the data, you only scale the features data (`X_train` and `X_testing`).

[8]:
```python
# Create the StandardScaler instance
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

[9]:
```python
# Fit the Standard Scaler with the training data
# When fitting scaling functions, only train on the training dataset
X_scaler = scaler.fit(X_train)
```

[10]:
```python
# Scale the training and testing data
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```

Ensemble Learners

In this section, you will compare two ensemble algorithms to determine which algorithm results in the best performance. You will train a Balanced Random Forest Classifier and an Easy Ensemble classifier . For each algorithm, be sure to complete the folliowing steps:

1. Train the model using the training data.
2. Calculate the balanced accuracy score from sklearn.metrics.
3. Display the confusion matrix from sklearn.metrics.
4. Generate a classication report using the `imbalanced_classification_report` from imbalanced-learn.
5. For the Balanced Random Forest Classifier only, print the feature importance sorted in descending order (most important feature to least important) along with the feature score

Note: Use a random state of 1 for each algorithm to ensure consistency between tests

Balanced Random Forest Classifier

[11]:
```python
# Resample the training data with the BalancedRandomForestClassifier
from imblearn.ensemble import BalancedRandomForestClassifier
clf_model = BalancedRandomForestClassifier(random_state=1)
clf_model = clf_model.fit(X_train_scaled,y_train)
y_predict = clf_model.predict(X_test_scaled)

results = pd.DataFrame({'Predictions': y_predict, 'Actual': y_test}).reset_index(drop=True)
results.head(3)
```

[11]:

	Predictions	Actual
0	0	0
1	0	0
2	0	0

[12]:
```python
# Calculated the balanced accuracy score
bas = balanced_accuracy_score(y_test,y_predict)
```

[13]:
```python
# Display the confusion matrix
# Calculating the confusion matrix
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
cm = confusion_matrix(y_test, y_predict)
cm_df = pd.DataFrame(
    cm, index=["Actual 0", "Actual 1"], columns=["Predicted 0", "Predicted 1"]
)

# Calculating the accuracy score
acc_score = accuracy_score(y_test, y_predict)

# Displaying results
print("Confusion Matrix")
display(cm_df)
print(f"Accuracy Score : {acc_score}")
print("Classification Report")
print(classification_report(y_test, y_predict))
```

Low_risk loans: 6.90% failure rate
High_risk loans: 27.59% failure rate

TP + TN / total sample count (17,205) = 88.06%

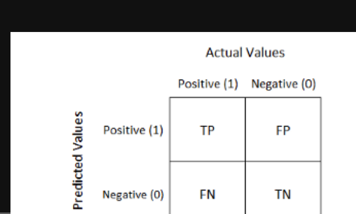
BalancedRandomForestClassifier

Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	15087	2031
Actual 1	24	63

```
Accuracy Score : 0.8805579773321709
Classification Report
              precision    recall  f1-score   support

           0       1.00      0.88      0.94     17118
           1       0.03      0.72      0.06        87

    accuracy                           0.88     17205
   macro avg       0.51      0.80      0.50     17205
weighted avg       0.99      0.88      0.93     17205
```

accuracy is 88%



[14]:
```python
# Print the imbalanced classification report
print(classification_report_imbalanced(y_test,y_predict))
```

```
                   pre       rec       spe        f1       geo       iba       sup

          0       1.00      0.88      0.72      0.94      0.80      0.65     17118
          1       0.03      0.72      0.88      0.06      0.80      0.63        87
```

resample_v2.ipynb | ensemble_v2.ipynb

Code

	Predicted 0	Predicted 1
Actual 0	15087	2031
Actual 1	24	63

```
Accuracy Score : 0.8805579773321709
Classification Report
              precision    recall  f1-score   support

           0       1.00      0.88      0.94     17118
           1       0.03      0.72      0.06        87

    accuracy                           0.88     17205
   macro avg       0.51      0.80      0.50     17205
weighted avg       0.99      0.88      0.93     17205
```

BalancedRandomForestClassifier

Precision:
87% of **predictions from x_test** (predicting high risk loans) were actually high risk. (100% precision for low risk)

[14]:
```python
# Print the imbalanced classification report
print(classification_report_imbalanced(y_test,y_predict))
```

```
                   pre       rec       spe        f1       geo       iba       sup

          0       1.00      0.88      0.72      0.94      0.80      0.65     17118
          1       0.03      0.72      0.88      0.06      0.80      0.63        87

avg / total       0.99      0.88      0.72      0.93      0.80      0.65     17205
```

Recall:
100% of all high risk loans 1's **in total population (y_test & y_train)** were predicted correctly. 100% for low risk loans

[15]:
```python
clf_model.feature_importances_
```

Specificity:
100% of low risk loans were predicted as low risk. No high risk loans were predicted as low risk

[15]:
```
array([0.01321271, 0.03065569, 0.01657395, 0.00293167, 0.01415232,
       0.00544397, 0.01934209, 0.00264085, 0.00631967, 0.00875553,
       0.00131662, 0.01523119, 0.00953949, 0.00128066, 0.01853967,
       0.01825105, 0.05289708, 0.05681406, 0.06200835, 0.05339051,
       0.00781971, 0.        , 0.        , 0.06585662, 0.00048097,
       0.        , 0.0015801 , 0.        , 0.00426901, 0.01783728,
       0.00556881, 0.00743031, 0.00408926, 0.00838655, 0.01246609,
       0.01317168, 0.01353289, 0.00403998, 0.00553176, 0.01785076,
       0.01540731, 0.01700061, 0.00994439, 0.00687164, 0.0081179 ,
       0.00742666, 0.01491399, 0.01214287, 0.01733149, 0.00013023,
       0.        , 0.01346859, 0.01744199, 0.01061462, 0.00940193,
       0.00798753, 0.0118198 , 0.01711228, 0.00375916, 0.00861738,
       0.00861675, 0.00699718, 0.00854916, 0.0089459 , 0.01015187,
       0.01139955, 0.00776938, 0.00988477, 0.        , 0.        ,
       0.00049359, 0.00741724, 0.00975752, 0.00778601, 0.00140686,
       0.        , 0.01304245, 0.01410466, 0.01768427, 0.01307536,
       0.03419819])
```

[16]:
```python
# List the features sorted in descending order by feature importance
feature_importance = clf_model.feature_importances_
feature_importance_sorted = sorted(zip(clf_model.feature_importances_, X.columns), reverse=True)
feature_importance_sorted[:10]
```

[16]:
```
[(0.06585662369230211, 'last_pymnt_amnt'),
 (0.06200834608249454, 'total_rec_prncp'),
 (0.05681405742930091 5, 'total_pymnt_inv'),
 (0.05339050684419236, 'total_rec_int'),
 (0.05289707638395531, 'total_pymnt'),
 (0.03419819064481047, 'issue_month'),
 (0.030655688608152216, 'int_rate'),
 (0.019342092192370475, 'dti'),
 (0.018539669026893753, 'out_prncp'),
 (0.01825105490320566, 'out_prncp_inv')]
```

Easy Ensemble Classifier

EasyEnsembleClassifier

[17]:
```python
# Train the Classifier
from imblearn.ensemble import EasyEnsembleClassifier
eec_model = EasyEnsembleClassifier(random_state=1)
eec_model = eec_model.fit(X_train_scaled,y_train)
y_predict = eec_model.predict(X_test_scaled)

results = pd.DataFrame({'Predictions': y_predict, 'Actual': y_test}).reset_index(drop=True)
results.head(3)
```

[17]:

	Predictions	Actual
0	0	0
1	0	0
2	0	0

[18]:
```python
# Calculated the balanced accuracy score
bas = balanced_accuracy_score(y_test,y_predict)
```

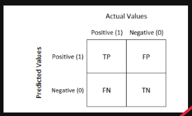
[19]:
```python
# Display the confusion matrix
# Calculating the confusion matrix
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
cm = confusion_matrix(y_test, y_predict)
cm_df = pd.DataFrame(
    cm, index=["Actual 0", "Actual 1"], columns=["Predicted 0", "Predicted 1"]
)

# Calculating the accuracy score
acc_score = accuracy_score(y_test, y_predict)

# Displaying results
print("Confusion Matrix")
display(cm_df)
print(f"Accuracy Score : {acc_score}")
print("Classification Report")
print(classification_report(y_test, y_predict))
```

Low_risk loans: 14.76% failure rate
High_risk loans: 17.24% failure rate

TP + TN / total sample count (17,205) = 85.22%

Confusion Matrix

	Predicted 0	Predicted 1
Actual 0	14591	2527
Actual 1	15	72



Recall:
100% of all high risk loans 1's **in total population (y_test & y_train)** were predicted correctly. 100% for low risk loans

```
Accuracy Score : 0.8522522522522522
Classification Report
              precision    recall  f1-score   support

           0       1.00      0.85      0.92     17118
           1       0.03      0.83      0.05        87

    accuracy                           0.85     17205
   macro avg       0.51      0.84      0.49     17205
weighted avg       0.99      0.85      0.92     17205
```

Precision:
87% of **predictions from x_test** (predicting high risk loans) were actually high risk. (100% precision for low risk)

[20]:
```python
# Print the imbalanced classification report
print(classification_report_imbalanced(y_test,y_predict))
```

Specificity:
100% of low risk loans were predicted as low risk. No high risk loans were predicted as low risk

```
                   pre       rec       spe        f1       geo       iba       sup

          0       1.00      0.85      0.83      0.92      0.84      0.71     17118
          1       0.03      0.83      0.85      0.05      0.84      0.70        87

avg / total       0.99      0.85      0.83      0.92      0.84      0.71     17205
```

[ ]:

Final Questions

README.md × resample_v2.ipynb × ensemble_v2.ipynb ×

Unit 11 - Risky Business

Credit Risk

Background

Mortgages, student and auto loans, and debt consolidation are just a few examples of credit and loans that people seek online. Peer-to-peer lending services such as Loans Canada and Mogo let investors loan people money without using a bank. However, because investors always want to mitigate risk, a client has asked that you help them predict credit risk with machine learning techniques.

In this assignment you will build and evaluate several machine learning models to predict credit risk using data you'd typically see from peer-to-peer lending services. Credit risk is an inherently imbalanced classification problem (the number of good loans is much larger than the number of at-risk loans), so you will need to employ different techniques for training and evaluating models with imbalanced classes. You will use the imbalanced-learn and Scikit-learn libraries to build and evaluate models using the two following techniques:

1. Resampling
2. Ensemble Learning

Files

Resampling Starter Notebook

Ensemble Starter Notebook

Lending Club Loans Data

Instructions

Resampling

Use the imbalanced learn library to resample the LendingClub data and build and evaluate logistic regression classifiers using the resampled data.

To begin:

1. Read the CSV into a DataFrame.
2. Split the data into Training and Testing sets.
3. Scale the training and testing data using the `StandardScaler` from `sklearn.preprocessing`.
4. Use the provided code to run a Simple Logistic Regression:
   - Fit the `logistic regression classifier`.
   - Calculate the `balanced accuracy score`.
   - Display the `confusion matrix`.
   - Print the `imbalanced classification report`.

Next you will:

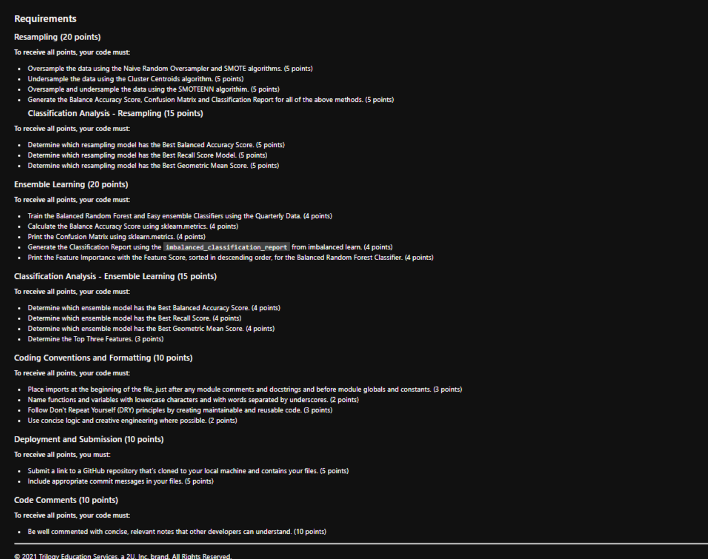
1. Oversample the data using the `Naive Random Oversampler` and `SMOTE` algorithms.
2. Undersample the data using the `Cluster Centroids` algorithm.
3. Over- and undersample using a combination `SMOTEENN` algorithm.

For each of the above, you will need to:

1. Train a `logistic regression classifier` from `sklearn.linear_model` using the resampled data.
2. Calculate the `balanced accuracy score` from `sklearn.metrics`.
3. Display the `confusion matrix` from `sklearn.metrics`.
4. Print the `imbalanced classification report` from `imblearn.metrics`.

Use the above to answer the following questions:

- Which model had the best balanced accuracy score?
- Which model had the best recall score?
- Which model had the best geometric mean score?



Requirements

Resampling (20 points)

To receive all points, your code must:

- Oversample the data using the Naive Random Oversampler and SMOTE algorithms. (5 points)
- Undersample the data using the Cluster Centroids algorithm. (5 points)
- Oversample and undersample the data using the SMOTEENN algorithim. (5 points)
- Generate the Balance Accuracy Score, Confusion Matrix and Classification Report for all of the above methods. (5 points)

Classification Analysis - Resampling (15 points)

To receive all points, your code must:

- Determine which resampling model has the Best Balanced Accuracy Score. (5 points)
- Determine which resampling model has the Best Recall Score Model. (5 points)
- Determine which resampling model has the Best Geometric Mean Score. (5 points)

Ensemble Learning (20 points)

To receive all points, your code must:

- Train the Balanced Random Forest and Easy ensemble Classifiers using the Quarterly Data. (4 points)
- Calculate the Balance Accuracy Score using sklearn.metrics. (4 points)
- Print the Confusion Matrix using sklearn.metrics. (4 points)
- Generate the Classification Report using the `imbalanced_classification_report` from imbalanced learn. (4 points)
- Print the Feature Importance with the Feature Score, sorted in descending order, for the Balanced Random Forest Classifier. (4 points)

Classification Analysis - Ensemble Learning (15 points)

To receive all points, your code must:

- Determine which ensemble model has the Best Balanced Accuracy Score. (4 points)
- Determine which ensemble model has the Best Recall Score. (4 points)
- Determine which ensemble model has the Best Geometric Mean Score. (4 points)
- Determine the Top Three Features. (3 points)

Coding Conventions and Formatting (10 points)

To receive all points, your code must:

- Place imports at the beginning of the file, just after any module comments and docstrings and before module globals and constants. (3 points)
- Name functions and variables with lowercase characters and with words separated by underscores. (2 points)
- Follow Don't Repeat Yourself (DRY) principles by creating maintainable and reusable code. (3 points)
- Use concise logic and creative engineering where possible. (2 points)

Deployment and Submission (10 points)

To receive all points, you must:

- Submit a link to a GitHub repository that's cloned to your local machine and contains your files. (5 points)
- Include appropriate commit messages in your files. (5 points)

Code Comments (10 points)

To receive all points, your code must:

- Be well commented with concise, relevant notes that other developers can understand. (10 points)

Ensemble Learning

In this section, you will train and compare two different ensemble classifiers to predict loan risk and evaluate each model. You will use the Balanced Random Forest Classifier and the Easy Ensemble Classifier. Refer to the documentation for each of these to read about the models and see examples of the code.

To begin:

1. Read the data into a DataFrame using the provided starter code.
2. Split the data into training and testing sets.
3. Scale the training and testing data using the `StandardScaler` from `sklearn.preprocessing`.

Then, complete the following steps for each model:

1. Train the model using the quarterly data from LendingClub provided in the `Resource` folder.
2. Calculate the balanced accuracy score from `sklearn.metrics`.
3. Display the confusion matrix from `sklearn.metrics`.
4. Generate a classification report using the `imbalanced_classification_report` from imbalanced learn.
5. For the balanced random forest classifier only, print the feature importance sorted in descending order (most important feature to least important) along with the feature score.

Use the above to answer the following questions:

- Which model had the best balanced accuracy score?
- Which model had the best recall score?
- Which model had the best geometric mean score?
- What are the top three features?

Hints and Considerations

Use the quarterly data from the LendingClub data provided in the `Resources` folder. Keep the file in the zipped format and use the starter code to read the file.

Refer to the imbalanced-learn and scikit-learn official documentation for help with training the models. Remember that these models all use the model->fit->predict API.

For the ensemble learners, use 100 estimators for both models.

Submission

- Create Jupyter notebooks for the homework and host the notebooks on GitHub.
- Include a markdown that summarizes your homework and include this report in your GitHub repository.
- Submit the link to your GitHub project to Bootcamp Spot.

Requirements

Resampling (20 points)

To receive all points, your code must:

- Oversample the data using the Naive Random Oversampler and SMOTE algorithms. (5 points)
- Undersample the data using the Cluster Centroids algorithm. (5 points)
- Oversample and undersample the data using the SMOTEENN algorithim. (5 points)