Class Imbalance Problems ¶

Introduction

You've learned about precision, recall, accuracy, f1 score, ROC curves, and AUC as metrics for evaluating the performance of classifiers. With this, you've seen how measuring the performance of classification algorithms is substantially different from that of regression. For example, we briefly discussed a scenario where only 2 in 1000 cases were labeled 'positive'. In such drastically cases, even a naive classifier that simply always predicts a 'negative' label would be 99.8% accurate. Moreover, such scenarios are relatively common in areas such as medical conditions or credit card fraud. This is known as the 'class imbalance' problem. As such, there has been a lot of work and research regarding class imbalance problems and methods for tuning classification algorithms to better fit these scenarios.

Objectives

You will be able to:

- Describe why class imbalance can lead to problems in machine learning
- List the different methods of fixing class imbalance issues

Class weight

One initial option for dealing with class imbalance problems is to weight the two classes. By default the class weights for logistic regression in scikit-learn is `None` , meaning that both classes will be given equal importance in tuning the model. Alternatively, you can pass `'balanced'` in order to assign weights that are inversely proportional to that class's frequency. The final option is to explicitly pass weights to each class using a dictionary of the form `{'class_label': weight}` .

First, here's the documentation for this parameter of `LogisticRegression()` :

```
class_weight : dict or 'balanced', default: None
    Weights associated with classes in the form {class_label: weight}.
    If not given, all classes are supposed to have weight one.

    The "balanced" mode uses the values of y to automatically adjust
    weights inversely proportional to class frequencies in the input data
    as n_samples / (n_classes * np.bincount(y)).

    Note that these weights will be multiplied with sample_weight (passed
    through the fit method) if sample_weight is specified.

    .. versionadded:: 0.17
       *class_weight='balanced'*
```

To investigate this, we'll load a dataset on **Mobile App Downloads**. The data represents information regarding users of a website and whether or not they downloaded the app to their phone. As you can guess, most visitors don't download the app making it an imbalanced dataset as we had been discussing.

First, load in the dataset and relevant functions:

In [1]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, roc_curve, auc
import pandas as pd
import warnings
warnings.filterwarnings("ignore")

# Load the data
df = pd.read_csv('mobile_app_attribution.csv')

# Data preview
df.head()
```

Out[1]:

	ip	app	device	os	channel	click_time	attributed_time	is_attributed
0	87540	12	1	13	497	2017-11-07 09:30:38	NaN	0
1	105560	25	1	17	259	2017-11-07 13:40:27	NaN	0
2	101424	12	1	19	212	2017-11-07 18:05:24	NaN	0
3	94584	13	1	13	477	2017-11-07 04:58:08	NaN	0
4	68413	12	1	1	178	2017-11-09 09:00:09	NaN	0

Let's look at the level of class imbalance in the dataset:

In [2]:
```
print('Raw counts: \n')
print(df['is_attributed'].value_counts())
print('-----------------------------------')
print('Normalized counts: \n')
print(df['is_attributed'].value_counts(normalize=True))
```

```
Raw counts: 

0    99773
1      227
Name: is_attributed, dtype: int64
-----------------------------------
Normalized counts: 

0    0.99773
1    0.00227
Name: is_attributed, dtype: float64
```

As you can see, over 99% of the data is the negative case.

Then we'll define `X` and `y` :

In [3]:
```
# Define appropriate X and y
y = df['is_attributed']
X = df[['ip', 'app', 'device', 'os', 'channel']]
X = pd.get_dummies(X)

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

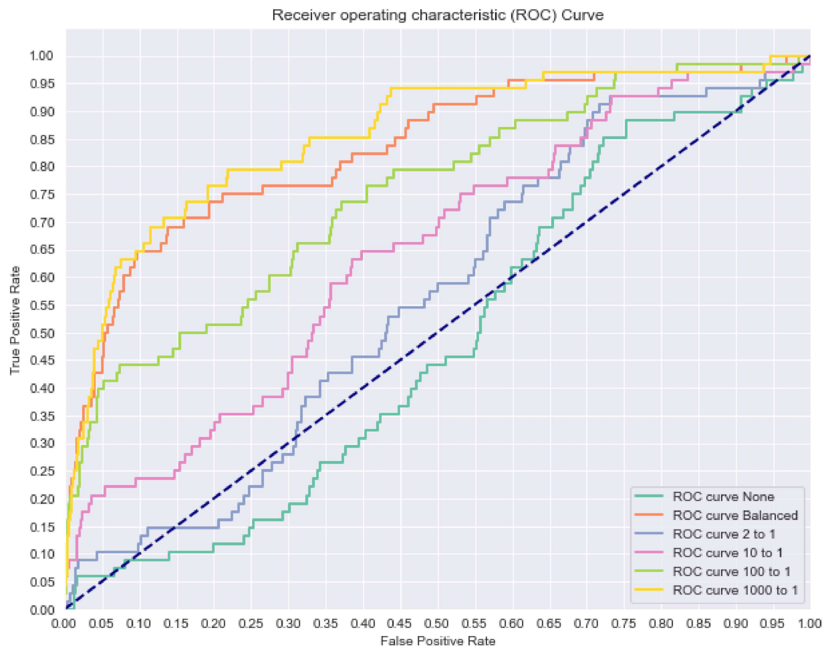
With that, let's compare a few models with varying class weights.

Compare models with varying class weights

In [4]:
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('darkgrid')
%matplotlib inline
```

In [5]:
```
# Now let's compare a few different regularization performances on the dataset:
weights = [None, 'balanced', {1:2, 0:1}, {1:10, 0:1}, {1:100, 0:1}, {1:1000, 0:1}]
names = ['None', 'Balanced', '2 to 1', '10 to 1', '100 to 1', '1000 to 1']
colors = sns.color_palette('Set2')

plt.figure(figsize=(10,8))

for n, weight in enumerate(weights):
    # Fit a model
    logreg = LogisticRegression(fit_intercept=False, C=1e20, class_weight=weight, solver='lbfgs')
    model_log = logreg.fit(X_train, y_train)
    print(model_log)

    # Predict
    y_hat_test = logreg.predict(X_test)

    y_score = logreg.fit(X_train, y_train).decision_function(X_test)

    fpr, tpr, thresholds = roc_curve(y_test, y_score)

    print('AUC for {}: {}'.format(names[n], auc(fpr, tpr)))
    print('-------------------------------------------------------------------------------------')
    lw = 2
    plt.plot(fpr, tpr, color=colors[n],
             lw=lw, label='ROC curve {}'.format(names[n]))

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])

plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
LogisticRegression(C=1e+20, fit_intercept=False)
AUC for None: 0.4811906031464407
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, class_weight='balanced', fit_intercept=False)
AUC for Balanced: 0.8320950632779984
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, class_weight={0: 1, 1: 2}, fit_intercept=False)
AUC for 2 to 1: 0.5602680467341757
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, class_weight={0: 1, 1: 10}, fit_intercept=False)
AUC for 10 to 1: 0.6415562093600474
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, class_weight={0: 1, 1: 100}, fit_intercept=False)
AUC for 100 to 1: 0.7484994479100802
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, class_weight={0: 1, 1: 1000}, fit_intercept=False)
AUC for 1000 to 1: 0.8551925944451261
-------------------------------------------------------------------------------------
```



As you can see, class weight can have a significant impact! In this case, typically the heavier we weight the positive case, the better our classifier appears to be performing.

Oversampling and undersampling

Another technique that can be used is oversampling or undersampling. This can help address class imbalance problems when one category is far more prevalent than the other. This is a common case that occurs in medicine, image classification or fraud detection. In many of these scenarios, class imbalance can cause difficulties for the learning algorithm. After all, simply predicting the majority class could yield 99%+ accuracy if the rare class occurs <1% of the time. Due to this, sampling techniques such as **oversampling the minority class** or **undersampling the majority class** can help by producing a synthetic dataset that the learning algorithm is trained on. With this, it is important to still maintain a test set from the original dataset in order to accurately judge the accuracy of the algorithm overall.

Undersampling can only be used when you have a truly massive dataset and can afford to lose data points. However, even with very large datasets, you are losing potentially useful data. Oversampling can run into the issue of overfitting to certain characteristics of certain data points because there will be exact replicas of data points.

While these initial modifications will improve the performance of classification algorithms on imbalanced datasets, a more advanced technique known as SMOTE has produced even better results in practice.

SMOTE

SMOTE stands for **Synthetic Minority Oversampling**. Here, rather then simply oversampling the minority class with replacement (which simply adds duplicate cases to the dataset), the algorithm generates new sample data by creating 'synthetic' examples that are combinations of the closest minority class cases. You can read more about SMOTE here (https://jair.org/index.php/jair/article/view/10302/24590).

SMOTE is generally a powerful way to deal with class imbalances, but it runs into computational issues when you have a large number of features due to the curse of dimensionality.

Implementing this technique is very easy using the `imblearn` package:

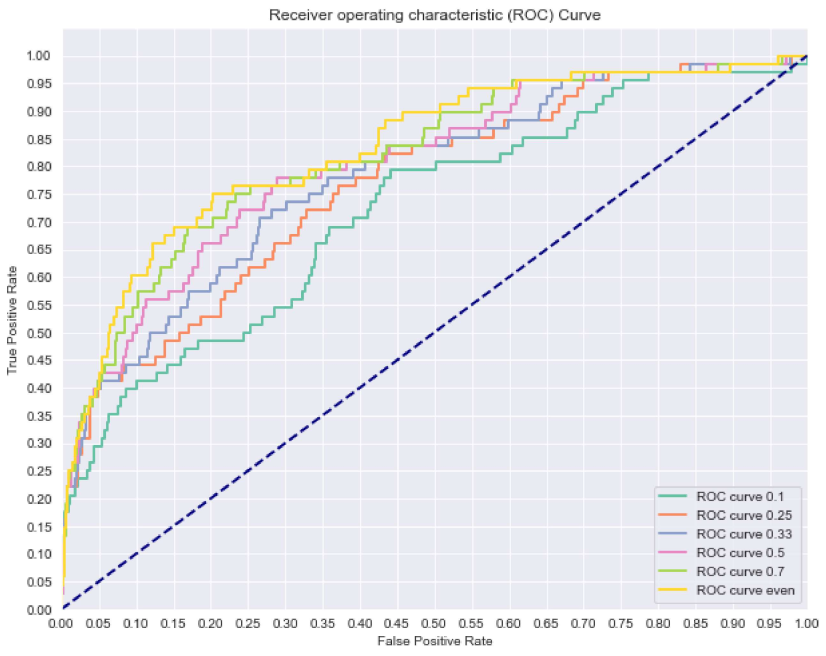
```
In [6]: from imblearn.over_sampling import SMOTE
```

In [7]:
```
# Previous original class distribution
print('Original class distribution: \n')
print(y.value_counts())
smote = SMOTE()
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
# Preview synthetic sample class distribution
print('-----------------------------------------')
print('Synthetic sample class distribution: \n')
print(pd.Series(y_train_resampled).value_counts())
```

```
Original class distribution: 

0    99773
1      227
Name: is_attributed, dtype: int64
-----------------------------------------
Synthetic sample class distribution: 

1    74841
0    74841
Name: is_attributed, dtype: int64
```

In [8]:
```
# Now let's compare a few different ratios of minority class to majority class
ratios = [0.1, 0.25, 0.33, 0.5, 0.7, 1]
names = ['0.1', '0.25', '0.33','0.5','0.7','even']
colors = sns.color_palette('Set2')

plt.figure(figsize=(10, 8))

for n, ratio in enumerate(ratios):
    # Fit a model
    smote = SMOTE(sampling_strategy=ratio)
    X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
    logreg = LogisticRegression(fit_intercept=False, C=1e20, solver ='lbfgs')
    model_log = logreg.fit(X_train_resampled, y_train_resampled)
    print(model_log)

    # Predict
    y_hat_test = logreg.predict(X_test)

    y_score = logreg.decision_function(X_test)

    fpr, tpr, thresholds = roc_curve(y_test, y_score)

    print('AUC for {}: {}'.format(names[n], auc(fpr, tpr)))
    print('-------------------------------------------------------------------------------------')
    lw = 2
    plt.plot(fpr, tpr, color=colors[n],
             lw=lw, label='ROC curve {}'.format(names[n]))

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])

plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
LogisticRegression(C=1e+20, fit_intercept=False)
AUC for 0.1: 0.7202349213389831
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, fit_intercept=False)
AUC for 0.25: 0.7657793905304783
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, fit_intercept=False)
AUC for 0.33: 0.7825508913656911
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, fit_intercept=False)
AUC for 0.5: 0.8040859372788101
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, fit_intercept=False)
AUC for 0.7: 0.8187511206953502
-------------------------------------------------------------------------------------
LogisticRegression(C=1e+20, fit_intercept=False)
AUC for even: 0.8324300922037353
-------------------------------------------------------------------------------------
```



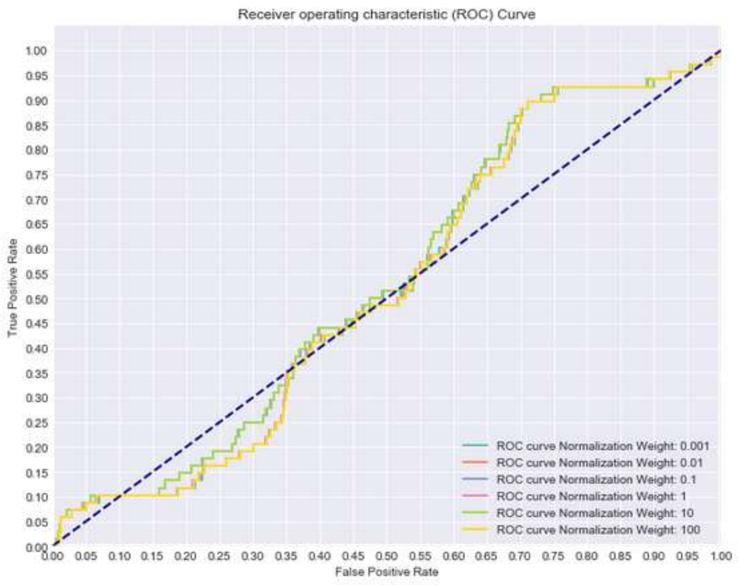
Hopefully this should make sense; after synthetically resampling our data, we no longer need to lean on penalized class weights in order to improve our model tuning. Since *SMOTE* recreated our dataset to have a balanced number of positive and negative cases, aggressive weighting schemas such as 10:1, 100:1 or 1000:1 drastically impact our model performance; the data is effectively no longer class imbalanced, so creating the class weights effectively reintroduces the original problem. Overall, our SMOTE unweighted model appears to be the current top performer. In practice, it is up to you the modeler, to make this and other choices when comparing models. For example, you may also wish to tune other parameters in your model such as how to perform regularization.

As a review, recall that regularization terms are penalties to a more straightforward error expression between our model and its outputs. The two most common regularizations are the l1 lasso and l2 ridge penalties. These add additional complexity to the loss function. In scikit-learn, these can be specified when initializing your regression object as in:

```
logreg = LogisticRegression(penalty='l1')
```

The default is to use an 'l2' penalty, so unless you specified otherwise, that's what you've been using.

In addition to simply specifying how to regularize the model, you can also specify the amount of regularization. This is controlled through the `C` parameter. For example, here is the ROC curve of various regularization values with no corrections for class imbalance:



As you can see, all of these models perform poorly regardless of the amount of regularization.

Summary

In this lesson, we investigated various tuning parameters for our model, as well as dealing with class imbalance as a whole. In the upcoming lab, you'll have a chance to try and adjust these parameters yourself in order to optimize a model for predicting credit fraud.