Machine Learning on Imbalanced Data

Applications in Python

Trevor H. Drees

Contents

Introduction

Imbalanced data can prove to be quite challenging to work with when using conventional machine learning algorithms, as many of these algorithms operate under the assumption that all classes in the data are approximately balanced. However, in many cases, the data we are most interested in classifying are often a minority of observations in the entire set; for example, a few spam emails out of hundreds of legitimate messages, or a few fraudulent credit card transactions out of thousands of legitimate purchases. Because traditional measurements of model performance often focus on metrics such as classification accuracy, they often fail to adequately capture the minority class, and we may observe instances when a model on a dataset with 990 observations in class A and 10 observations in class B can achieve 99% accuracy by completely ignoring class A and classifying everything as B. Also, the costs of misclassification might vary between the classes; for example, while fraudulent credit card transactions are extremely rare compared to legitimate purchases, a fraudulent transaction that is not detected may cost a bank or credit card company significantly more than a legitimate transaction that is falsely flagged as fraudulent. Many algorithms by default will assume equal costs of misclassification, which may not necessarily be true for certain scenarios.

Here, we use synthetic data to explore some of the tools available in Python for classifying imbalanced data, heavily relying on `sklearn` for most of model and scoring functions but also utilising some of the various sampling functions from `imbalanced-learn`. Some of the topics we explore include scoring metrics such as area under the precision-recall curve (AUPRC), cost-sensitive learning using class weights, and various techniques to rebalance data such as SMOTE and ADASYN. We also explore imbalanced classification in datasets with binary classes as well as in datasets with three or more different classes.

Two-class Classification

Data Generation

First, we set up a two-class classification problem. We randomly generate a total of 10000 samples, with the majority class making up 99% of the data and the minority class making up the other 1%. We can do this using the `make_classification` function from `sklearn`.

```
data_x, data_y = make_classification(n_samples = 10000, n_classes = 2, n_features = 2,
                                     n_informative = 2, n_redundant = 0, n_repeated = 0,
                                     weights = [0.99, 0.01], n_clusters_per_class = 1,
                                     flip_y = 0.006, random_state = 1, class_sep = 2)
```

As can be seen in Figure 1, we allow for a bit of overlap between the two classes so that we can explore the trade-offs involved in misclassifying points in the overlap zone; this becomes especially interesting once we assign different costs to the correct classification of each class.

Discriminant Analysis

First, we fit linear discriminant analysis (LDA) and quadratic discriminant analysis (QDA) models to our data. We start with a single validation set approach, splitting the data into a training set (50%) and test set (50%), done using the code below.

```
train_x, test_x, train_y, test_y = train_test_split(data_x, data_y, test_size = 0.5,
                                                    random_state = 2, stratify = data_y)
```

We can then fit the model to the training set before evaluating it on the test set. To fit the LDA and QDA models, we can use the `LinearDiscriminantAnalysis` and `QuadraticDiscriminantAnalysis` functions

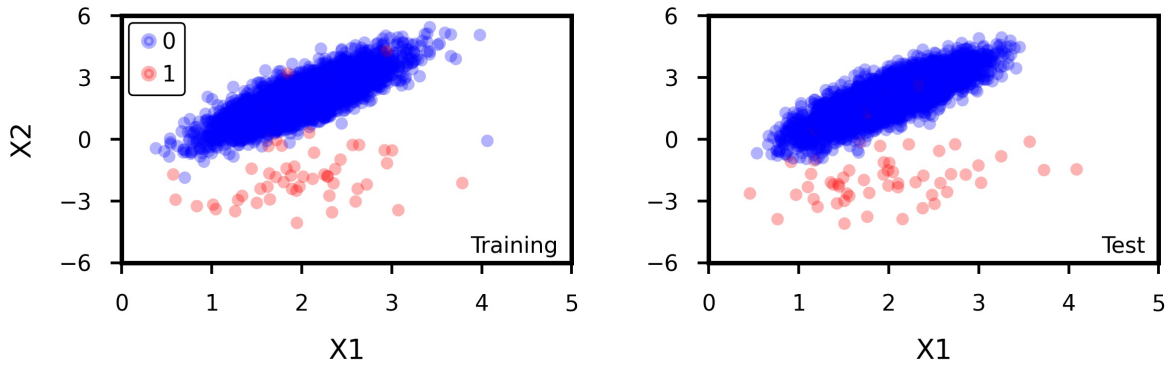

Figure 1: A 50/50 split of the data into training (left) and test (right) sets. The majority class outnumbers the minority class by a ratio of 99:1.

from `sklearn`, as is shown in the code below. For each model, we first create a model object using each its respective function and then use the `fit` method to fit the models to the training data.

```
clfLDA = LinearDiscriminantAnalysis()
clfLDA.fit(train_x, train_y)
clfQDA = QuadraticDiscriminantAnalysis()
clfQDA.fit(train_x, train_y)
```

For now, we assume that the cost of misclassifying a point from the majority class is the same as misclassifying a point from the minority class, so we do not place weights on either class. As such, we can compare performance on the training data using a micro-averaged F-score, and find that the LDA model outperforms the QDA model at 0.9962 versus 0.9958. We can also compare performance using the area under the precision-recall curve (AUPRC) on the training data and find that LDA again outperforms QDA at 0.8429 versus 0.8207. As can be seen in Figure 2, these differences are likely due to the fact that the QDA model misclassifies several majority class data points near the decision boundary, while the LDA model does so to a lesser extent. At this point, it is clear that the linear model performs better than the quadratic model, so we choose the former over the latter and then evaluate its performance on the test data; upon doing so, we get a micro-averaged F-score of 0.9976 and an AUPRC of 0.8994, indicating that the linear model actually performs better on the test data than it does on the training data.

It is important to note that the way the data is partitioned can affect model results, especially when using a single validation set; cross-validation provides an alternative approach that is more robust to randomness in how the training and test data are split, often allowing for more accurate model fitting and selection. We can write a function that performs cross-validation on a specified model and then returns the desired metrics, as has been done below.

```
def model_cv(obj, n_rep, metList, data_x = data_x, data_y = data_y):
    cv = RepeatedStratifiedKFold(n_splits = 2, n_repeats = n_rep, random_state = 32463)
    output = cross_validate(obj, data_x, data_y, cv = cv, scoring = metList, n_jobs = -1)
    df = DataFrame(columns = ["metric", "value"])
    for i in metList:
        newdat = DataFrame({"metric":[i], "value":[mean(output["test_" + i])]})
        df = df.append(newdat, ignore_index = True)
    print(df)
```

Here, we supply the function with a model, the number of replicates, and a list of metrics that we would like to calculate. The function then performs a 50/50 split on the data, fits the model to the training data, then

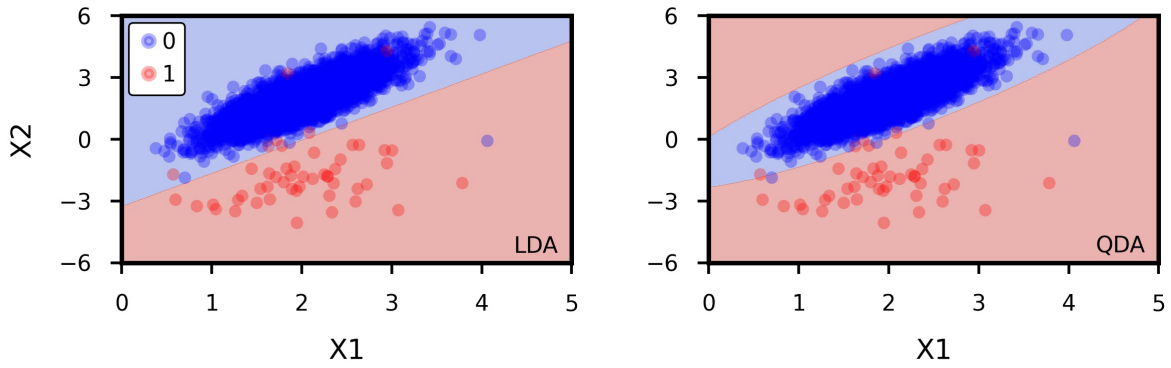

Figure 2: Decision boundaries generated by fitting LDA (left) and QDA (right) models to the training data.

evaluates the supplied metrics on the test data. This is then done for the specified number of replicates, and then each metric is averaged over the results of all replicates.

We can use this function for LDA and QDA, getting the mean micro-averaged and weighted F-scores over 1000 replicates of 2-fold cross validation.

```
model_cv(obj = clfLDA, n_rep = 1000, metList = ["f1_micro", "f1_weighted"])
model_cv(obj = clfQDA, n_rep = 1000, metList = ["f1_micro", "f1_weighted"])
```

We again find that LDA outperforms QDA, with a mean micro-averaged F-score of 0.9969 versus 0.9965.

Support Vector Machines: Cost-Insensitive

We can also use the `SVC` function to fit support vector machines to our data, using the resulting decision boundary to classify the observations. Similar to what we did with LDA and QDA, we first create model objects using SVC and then use the `fit` method to fit the models to the training data.

```
clfLin = svm.SVC(gamma = "auto", kernel = "linear")
clfLin.fit(train_x, train_y)
clf2dg = svm.SVC(gamma = "auto", kernel = "poly", degree = 2)
clf2dg.fit(train_x, train_y)
clf3dg = svm.SVC(gamma = "auto", kernel = "poly", degree = 3)
clf3dg.fit(train_x, train_y)
clfRbf = svm.SVC(gamma = "auto", kernel = "rbf")
clfRbf.fit(train_x, train_y)
```

Here, we fit four different kernels to the same training set we used earlier: a linear kernel, a 2nd-degree polynomial kernel, a 3nd-degree polynomial kernel, and a radial kernel (with default $c = 1$). The results of doing so, which can be seen in the left-hand panels of Figure 3, show that much like LDA and QDA, the models can correctly classify almost all of the observations in majority class while only missing a few observations in the minority class.

After using the `model_cv` function we defined earlier to perform 1000 instances of 2-fold cross-validation, we find that the 2nd-degree polynomial model performs the best with a mean micro-averaged F-score of 0.99678, and the 3rd-degree polynomial model performs the worst with a mean micro-averaged F-score of 0.99661. Note that we still assume that the cost of misclassifying a point from the majority class is the same as misclassifying a point from the minority class, so the use of micro-averaged F-score as a scoring metric is still appropriate.

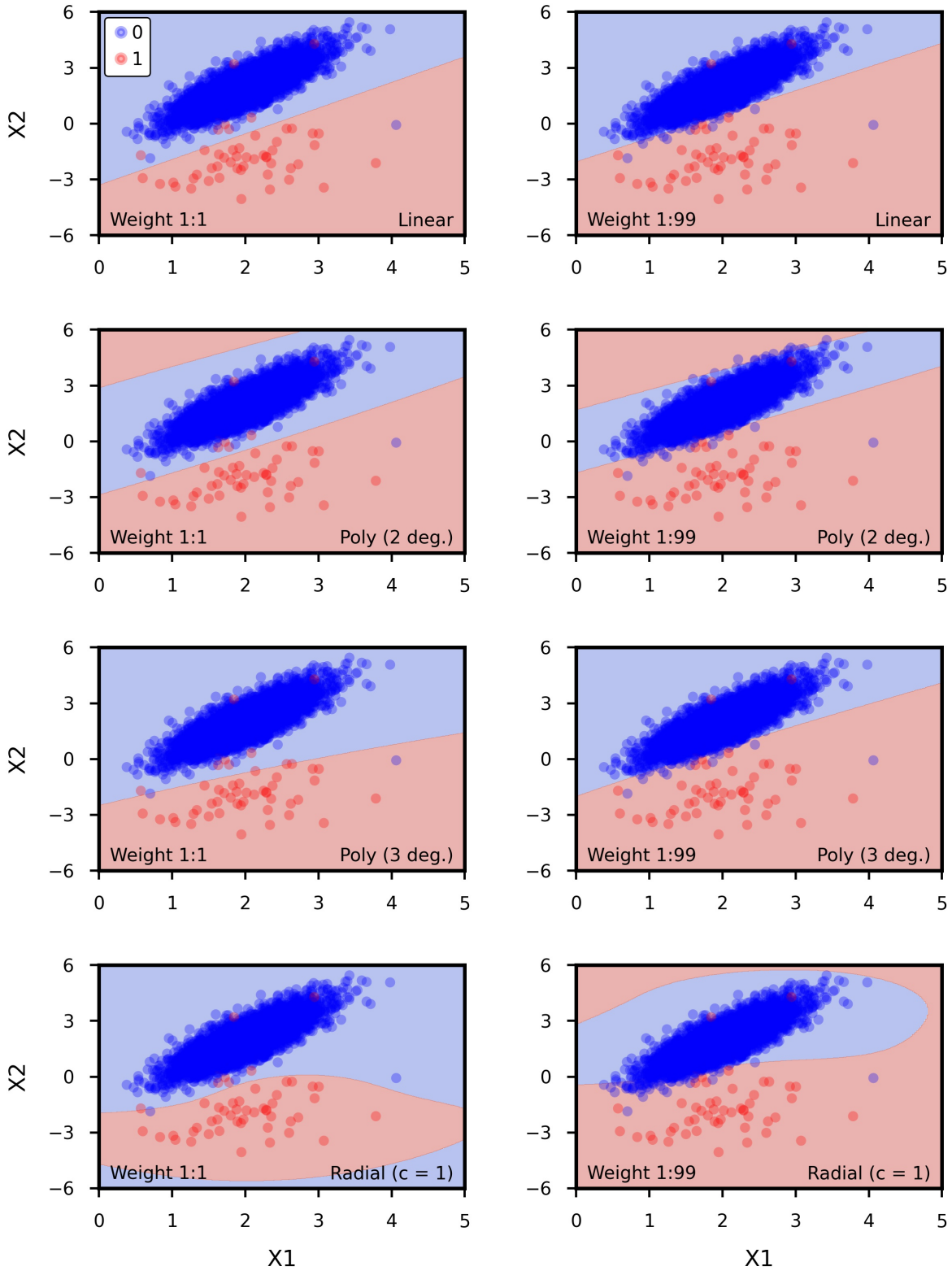

Figure 3: Decision boundaries generated by various SVM kernels to the training data, with no weight on the minority class (left column) and a 1:99 weight on the minority class (right column).