Exercise6-1

October 21, 2021

0.1 Exercise 6: Unbalanced Datasets and Discriminant Analysis

[1]:
```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, precision_score, recall_score,␣
↪roc_curve, accuracy_score
from sklearn.dummy import DummyClassifier
```

0.1.1 Part 1)

A) The following code generates synthetic data with one feature from two classes using the make_classification function. Split the data into train and test, then estimate the parameters needed to implement LDA classification. Predict the test data using the LDA discriminant score (you need to write your own implementation for evaluating the discriminant score). Feel free to double-check your answer by comparing with the Scikit-learn function.

[2]:
```
X, y = make_classification(n_samples=10000, n_features=1, \
                           n_informative =1, n_redundant =0, \
                           n_classes=2, n_clusters_per_class=1, \
                           n_repeated=0, weights=[0.5], flip_y=0, \
                           random_state=0)

X_train, X_test, Y_train, Y_test = train_test_split(X, y, random_state=0)

# split up classes to calculate mean, variance, and priors
c0 = []
c1 = []

for i, j in zip(X_train, Y_train):
    val = i[0]
    if j == 0:
        c0.append(val)
    else:
        c1.append(val)
```

```
c0 = np.asarray(c0)
c1 = np.asarray(c1)
```

[3]:
```
# calculate mean, variance, and priors
u = [np.mean(c0), np.mean(c1)]                                          # means of classes
s = (np.var(c0) + np.var(c1)) / 2                                       # weighted average of all variances
pi = [len(c0) / (len(c0) + len(c1)), len(c1) / (len(c0) + len(c1))]     # prior class probabilities

# calculate discriminant score for each class
preds = []

for sample in X_test:
    sample = sample[0]
    scores = []

    for k in range(0, 2):
        score = sample * u[k]/s - u[k]**2/s**2 + np.log(pi[k])
        scores.append(score)

    if scores[0] > scores[1]:
        preds.append(0)
    else:
        preds.append(1)

acc = np.count_nonzero(preds==Y_test) / len(preds)
print('Accuracy is %.2f.' % acc)
```

```
Accuracy is 0.94.
```

[4]:
```
# double check with scikit-learn LDA function
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis()
lda.fit(X_train, Y_train)
lda_preds = lda.predict(X_test)
lda_acc = np.count_nonzero(lda_preds==Y_test) / len(lda_preds)
print('Accuracy is %.2f.' % lda_acc)
```

```
Accuracy is 0.94.
```

B) With the same data generated above, Estimate the parameters needed to implement QDA classification. Predict the test data using the QDA discriminant score (you need to write your own implementation for evaluating the discriminant score). Find the accuracy. Feel free to double-check your answer by comparing with the Scikit-learn function.

[5]:
```
s = [np.var(c0), np.var(c1)]    # each class has its own variance now

# calculate discriminant score for each class
preds = []

for sample in X_test:
    sample = sample[0]
    scores = []

    for k in range(0, 2):
        score = -0.5*sample*1/s[k]*sample + sample*1/s[k]*u[k] - 0.5*u[k]*1/
↪s[k]*u[k] -0.5*np.log(s[k]) + np.log(pi[k])
        scores.append(score)

    if scores[0] > scores[1]:
        preds.append(0)
    else:
        preds.append(1)

acc = np.count_nonzero(preds==Y_test) / len(preds)
print('Accuracy is %.2f.' % acc)
```

```
Accuracy is 0.98.
```

[6]:
```
# double check with scikit-learn QDA function
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

qda = QuadraticDiscriminantAnalysis()
qda.fit(X_train, Y_train)
qda_preds = qda.predict(X_test)
qda_acc = np.count_nonzero(qda_preds==Y_test) / len(qda_preds)
print('Accuracy is %.2f.' % qda_acc)
```

```
Accuracy is 0.98.
```

0.1.2 Part 2)

Use the make_classification function (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_c to create an imbalanced dataset for two classes with two features. Example input argument can be: (n_samples=10000, n_features=2, n_redundant=0, n_clusters_per_class=1, weights=[0.9], flip_y=0, random_state=0). The weights argument define how imbalaced the data in the classes are.

A) After creating the dataset, split it into train and test. Train a Logistic regression classifier (use solver='lbfgs' and keep the rest with default values) and find its accuracy, precision and recall.

B) Apply a threshold of 0.5 (default) on the probability values obtained via Logisitic regression to make the prediction. Then, find the accuracy, precision and recall. You should obtain the

same results as in previous part. (You can use predict_proba method to find the probabilities.)

C) Change the threshold to a lower value (e.g. 0.4), and find the accuracy, precision, and recall. Evaluate the precision and recall without using sklearn.metrics. Feel free to double check your results with values obtained from sklearn.metrics library.

D) Train a dummy classifier that predict the most frequent class and find its accuracy. (you can use: https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html) Select dummy classifier with "most frequent" strategy as follows: - dummy_majority=DummyClassifier(strategy='most_frequent')

E) Comment on your results

[17]:
```
X, y = make_classification(n_samples=10000, n_features=2, \
                           n_redundant =0, \
                           n_clusters_per_class=1, \
                           weights=[0.9], flip_y=0, \
                           random_state=0)

X_train, X_test, Y_train, Y_test = train_test_split(X, y, random_state=0)

LogRegModel = LogisticRegression(solver='lbfgs')
LogRegModel.fit(X_train, Y_train)

Y_pred = LogRegModel.predict(X_test)

# Only do for positive class (label=1)
print('Accuracy: %.2f' % LogRegModel.score(X_test, Y_test))
print(f'Precision: %.2f' % precision_score(Y_test, Y_pred))
print(f'Recall: %.2f' % recall_score(Y_test, Y_pred))
```

```
Accuracy: 0.97
Precision: 0.94
Recall: 0.71
```

[19]:
```
probs = LogRegModel.predict_proba(X_test)

thresholds = [0.5, 0.4, 0.3]

for threshold in thresholds:
    print(f'THRESHOLD = {threshold}.')

    tp = 0
    tn = 0
    fn = 0
    fp = 0
```

```
    for prob, val in zip(probs, Y_test):
        if prob[0] > (1 - threshold): # prediction is 0
            if val == 0:
                tn += 1
            else:                 # real value is 1
                fn += 1
        else:                     # prediction is 1
            if val == 0:
                fp += 1
            else:
                tp += 1

    acc = (tp + tn) / (tp + tn + fp + fn)
    prec = tp / (tp + fp)
    rec = tp / (tp + fn)

    print('Accuracy: %.2f' % acc)
    print('Precision: %.2f' % prec)
    print('Recall: %.2f' % rec)

    print('\n')
```

```
THRESHOLD = 0.5.
Accuracy: 0.97
Precision: 0.94
Recall: 0.71


THRESHOLD = 0.4.
Accuracy: 0.97
Precision: 0.91
Recall: 0.75


THRESHOLD = 0.3.
Accuracy: 0.97
Precision: 0.88
Recall: 0.77
```

[9]:
```
dummy = DummyClassifier(strategy='most_frequent')
dummy.fit(X_train, Y_train)

print('Accuracy: %.2f' % dummy.score(X_test, Y_test))
```

```
Accuracy: 0.91
```

(AP) Since the dummy classifier has an accuracy around 0.91, just relying on the accuracy for the logistic regression classifier is not sufficient for this unbalanced dataset. The precision and recall metrics for each class are more useful for evaluating its performance.

0.1.3 Part 3)

In this part, we will use the handwritten digits data set of Scikit-learn (load_digits). Run the code below. Read the description of the data set and check a sample image.

[10]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits


DigitsData=load_digits()
print(DigitsData.keys())
print(DigitsData.DESCR) #read description of the dataset
print(DigitsData.data[1])

#plot one of the images in the data
plt.gray()
plt.matshow(DigitsData.images[1])
plt.show()
```

```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images',
'DESCR'])
.. _digits_dataset:

Optical recognition of handwritten digits dataset
--------------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 5620
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digit
s

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
```

normalized bitmaps of handwritten digits from a preprinted form. From a total of 43 people, 30 contributed to the training set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block. This generates an input matrix of 8x8 where each element is an integer in the range 0..16. This reduces dimensionality and gives invariance to small distortions.

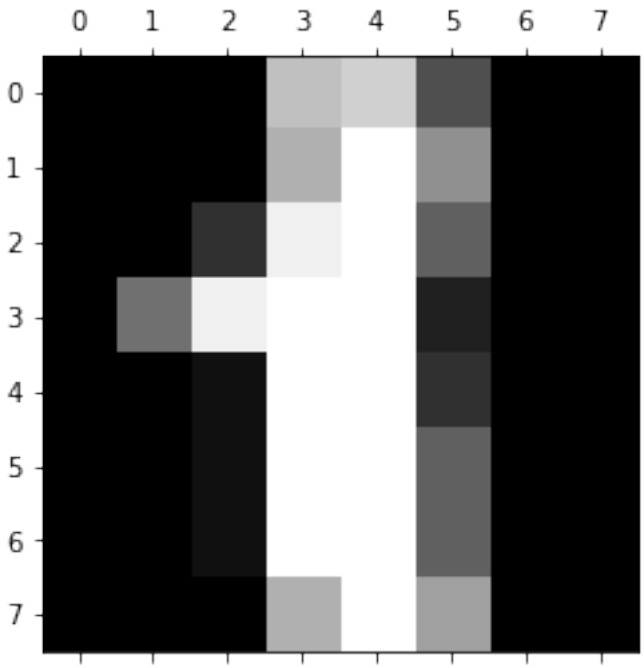
For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G. T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C. L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469, 1994.

.. topic:: References

  - C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their Applications to Handwritten Digit Recognition, MSc Thesis, Institute of Graduate Studies in Science and Engineering, Bogazici University.
  - E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
  - Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin. Linear dimensionalityreduction using relevance weighted LDA. School of Electrical and Electronic Engineering Nanyang Technological University. 2005.
  - Claudio Gentile. A New Approximate Maximal Margin Classification Algorithm. NIPS. 2000.


```
[ 0.  0.  0. 12. 13.  5.  0.  0.  0.  0.  0. 11. 16.  9.  0.  0.  0.  0.
  3. 15. 16.  6.  0.  0.  0.  7. 15. 16. 16.  2.  0.  0.  0.  0.  1. 16.
 16.  3.  0.  0.  0.  0.  1. 16. 16.  6.  0.  0.  0.  0.  1. 16. 16.  6.
  0.  0.  0.  0.  0. 11. 16. 10.  0.  0.]
```

<Figure size 432x288 with 0 Axes>



A) Our objective is to build classfiers that identify digit 9. For this purpose, answer the following:

Define the target value to be equal to 1 (or True) only for digit 9, and 0 (or False) otherwise. You can define: y= (DigitsData.target == 9)

- Find how many times y is equal to 9 and how many times it is not equal to 9

what do you observe? Is the dataset for this classification problem balanced or not?

[11]:
```
y = np.array((DigitsData.target == 9),dtype=int)

t = np.count_nonzero(y==True)
f = np.count_nonzero(y==False)

print(f'Number of times y is equal to 9: {t}.')
print(f'Number of times y is not equal to 9: {f}.')
```

```
Number of times y is equal to 9: 180.
Number of times y is not equal to 9: 1617.
```

(AP) The dataset is not balanced.

B) Find the accuracy of a dummy classifier (imported below) that always selects the majority class. Use the DigitsData.data as features and y (defined above) as the response.

- Use train_test_split with random_state= 0 for splitting the data
- Select dummy classifier with "most frequent" strategy as follows:
  - dummy_majority=DummyClassifier(strategy='most_frequent')

[12]:
```
X = DigitsData.data
X_train, X_test, Y_train, Y_test = train_test_split(X, y, random_state=0)

dummy = DummyClassifier(strategy='most_frequent')

dummy.fit(X_train, Y_train)

print('Accuracy: %.2f' % dummy.score(X_test, Y_test))
```

```
Accuracy: 0.90
```

C) Instead of a dummy classifier, use an LDA classifer (with default threshold) to solve the above classification problem. Find accuracy, confusion matrix, precision, and recall

- Note: you may get a warning that features are correlated (collinear). However, we can still get the performance metrics as usual

[13]:
```
lda = LinearDiscriminantAnalysis()

lda.fit(X_train, Y_train)
lda_preds = lda.predict(X_test)

print('Accuracy is %.2f.' % accuracy_score(Y_test, lda_preds))
print(confusion_matrix(Y_test, lda_preds))
print(f'Precision is %.2f.' % precision_score(Y_test, lda_preds))
print(f'Recall is %.2f.' % recall_score(Y_test, lda_preds))
```

```
Accuracy is 0.96.
[[394   9]
 [  7  40]]
Precision is 0.82.
Recall is 0.85.
```

D) Use QDA classifer to perform the classification. Find accuracy, confusion matrix, precision, and recall.

- Note: expect to see a warning that features are correlated (collinear). You can still get performance metrics.

[14]:
```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

qda = QuadraticDiscriminantAnalysis()

qda.fit(X_train, Y_train)
qda_preds = qda.predict(X_test)

print('Accuracy is %.2f.' % accuracy_score(Y_test, qda_preds))
print(confusion_matrix(Y_test, qda_preds))
print(f'Precision is %.2f.' % precision_score(Y_test, qda_preds))
print(f'Recall is %.2f.' % recall_score(Y_test, qda_preds))
```

```
Accuracy is 0.76.
[[299 104]
 [  2  45]]
Precision is 0.30.
Recall is 0.96.
```

```
C:\Users\Avery Peiffer\anaconda3_new\lib\site-
packages\sklearn\discriminant_analysis.py:715: UserWarning: Variables are
collinear
  warnings.warn("Variables are collinear")
```

E) From the prediction of the QDA, plot the ROC curve

[15]:
```
from sklearn.metrics import plot_roc_curve

plot_roc_curve(qda, X_test, Y_test)
```

[15]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1e2b4f2ab20>
```