Lecture 18: Classification Models (Code)

Goals for today:

- Error metric implementations
- sklearn implementations of models from last lecture
- Important hyperparameters
- Generic behaviors on toy datasets
- Application to a multilabel equipment classification problem.

Error Metrics – Definitions from Last Time

- **Recall** - fraction labeled correctly over all that should have been labeled

- **Precision** – fraction labeled correctly out of all that were labeled

Error Metrics – Example by Hand

- Consider the following binary class case (0 = in spec; 1 = out of spec):

Precision:

Error Metrics – Example by Sklearn

Let's redo this example using scikit-learn error metrics:

```
import numpy as np
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score

y_true = np.array([0,0,0,0,1,1,1,1])
y_pred = np.array([0,0,0,0,1,1,1,0])

print("P: {}".format(precision_score(y_true,y_pred,average=None)))
print("R: {}".format(recall_score(y_true,y_pred,average=None)))
print("A: {}".format(accuracy_score(y_true,y_pred)))
print("f1: {}".format(f1_score(y_true,y_pred,average=None)))
```

```
P: [0.8 1. ]
R: [1.   0.75]
A: 0.875
f1: [0.88888889 0.85714286]
```

Note: the default behavior for scikit-learn is to calculate P, R, and F1 with respect to the positive label (1) for binary classifications. We have used the `average=None` option to get the values for each class.

Alternatively, scikit learn supports various averaging options. For example, `average = 'macro'`, calculates the average across the individual classes, and `average = 'binary'`, reports the statistic for the positive label for binary problems.

Error Metrics – Example by Sklearn

Let's compare the distinct averaging behaviors:

```
y_true = np.array([0,0,0,0,1,1,1,1])
y_pred = np.array([0,0,0,0,1,1,1,0])

print("P (None) : {}".format(precision_score(y_true,y_pred,average=None)))
print("P (macro): {}".format(precision_score(y_true,y_pred,average='macro')))
print("P (binary): {}".format(precision_score(y_true,y_pred,average='binary')))

print("\nR (None) : {}".format(recall_score(y_true,y_pred,average=None)))
print("R (macro): {}".format(recall_score(y_true,y_pred,average='macro')))
print("R (binary): {}".format(recall_score(y_true,y_pred,average='binary')))
```

```
P (None) : [0.8 1. ]
P (macro): 0.9
P (binary): 1.0

R (None) : [1.   0.75]
R (macro): 0.875
R (binary): 0.75
```

Note: `average=binary` would be the default behavior if we hadn't supplied any `average` option for this particular example.

Error Metrics – Example by Sklearn

Scikit-learn generally allows non-integer classes:

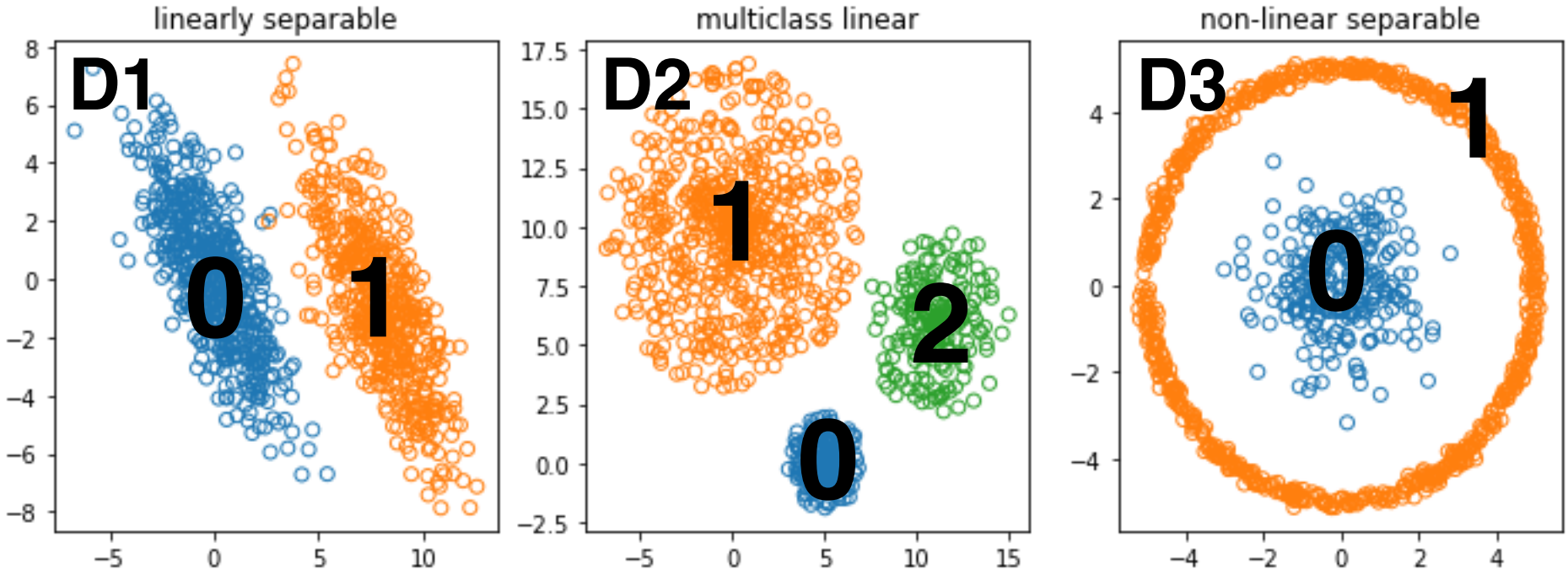
```
y_true = np.array(["in-spec","in-spec","in-spec","in-spec","out-of-spec","out-of-
spec","out-of-spec","out-of-spec"])
y_pred = np.array(["in-spec","in-spec","in-spec","in-spec","out-of-spec","out-of-
spec","out-of-spec","in-spec"])

print("P: {}".format(precision_score(y_true,y_pred,average=None)))
print("R: {}".format(recall_score(y_true,y_pred,average=None)))
print("A: {}".format(accuracy_score(y_true,y_pred)))
print("f1: {}".format(f1_score(y_true,y_pred,average=None)))
```

```
P: [0.8 1. ]
R: [1.   0.75]
A: 0.875
f1: [0.88888889 0.85714286]
```

This example is completely equivalent to the previous case, except that we've taken `0="in-spec"` and `1="out-of-spec"`.

However, the support for string-based classes isn't complete and you should generally convert your classes to integers from `[0,n-1]` for multi-class problems. At the end you can always convert back.

Three Artificial Datasets for Illustrating General Behaviors



• These examples span the two most important distinctions among classification problems: **linear/non-linear**, **binary/multi-class**, and **balanced/unbalanced**

• The specific balances for each dataset are **D1=(0.5,0.5), D2=(0.2,0.6,0.2),** and **D3=(0.75,0.25)** for labels 0, 1, and 2 respectively.

Predicting the Most Abundant Label

If you recall, a useful reference for the performance of **regression** models, is comparison with **predicting the mean**. The corresponding baseline for **classification** models is a model that always predicts the most prevalent category in the dataset.

```
from collections import Counter

# Show distribution of labels
data1_counts = Counter(data1_y).most_common()
data2_counts = Counter(data2_y).most_common()
data3_counts = Counter(data3_y).most_common()
print("data1 labels: {}".format(data1_counts))
print("data2 labels: {}".format(data2_counts))
print("data3 labels: {}".format(data3_counts))

# Calculate the majority label in each dataset
data1_maj = data1_counts[0][0]
data2_maj = data2_counts[0][0]
data3_maj = data3_counts[0][0]
print("\nmajority (data1): {}".format(data1_maj))
print("majority (data2): {}".format(data2_maj))
print("majority (data3): {}".format(data3_maj))

# Check accuracy from predicting the majority label
data1_pred = np.array([data1_maj]*len(data1_y))
data2_pred = np.array([data2_maj]*len(data2_y))
data3_pred = np.array([data3_maj]*len(data3_y))
print("\nA (data1): {}".format(accuracy_score(data1_y,data1_pred)))
print("A (data2): {}".format(accuracy_score(data2_y,data2_pred)))
print("A (data3): {}".format(accuracy_score(data3_y,data3_pred)))
```

```
data1 labels: [(0, 500), (1, 500)]
data2 labels: [(1, 600), (0, 200), (2, 200)]
data3 labels: [(1, 750), (0, 250)]

majority (data1): 0
majority (data2): 1
majority (data3): 1

A (data1): 0.5
A (data2): 0.6
A (data3): 0.75
```

We have used the `Counter` class from the `collections` library to do the counting for us.

Why is the baseline accuracy different for the three datasets?

Finding the "Balanced" Accuracy

A common adjustment to the accuracy metric is to average the accuracies of the individual labels.

The scikit-learn metric that does this is `balanced_accuracy_score:`

```
from sklearn.metrics import balanced_accuracy_score

print("A (data1): {}".format(balanced_accuracy_score(data1_y,data1_pred)))
print("A (data2): {}".format(balanced_accuracy_score(data2_y,data2_pred)))
print("A (data3): {}".format(balanced_accuracy_score(data3_y,data3_pred)))
```

```
A (data1): 0.5
A (data2): 0.3333333333333333
A (data3): 0.5
```

When we compare the balanced accuracy for the major label models they go as 1/N, which is what you might have intuitively expected for the standard accuracy metric.

Models – Logistic Regression

The relevant scikit-learn class for logistic regression is `sklearn.linear_model.LogisticRegression`.

Among the most important hyperparameters are :

`penalty`: this controls the regularization to use (L2 by default)

`tol`: this controls the stopping threshold for numerical optimization

`max_iter`: secondary stopping criteria for optimization

`C`: This is the inverse of the regularization strength (1 by default)

Recall that logistic regression doesn't have a closed form solution, so some of the parameters have to do with the optimization.

Logistic regression is commonly combined with **L1/L2/elastic** net type regularizations. By default scikit-learn used L2 with a weight of 1.

Models – Logistic Regression

The relevant scikit-learn class for logistic regression is `sklearn.linear_model.LogisticRegression`.

Among the most important hyperparameters are :

`penalty`: this controls the regularization to use (L2 by default)

`tol`: this controls the stopping threshold for numerical optimization

`max_iter`: secondary stopping criteria for optimization

`C`: This is the inverse of the regularization strength (1 by default)

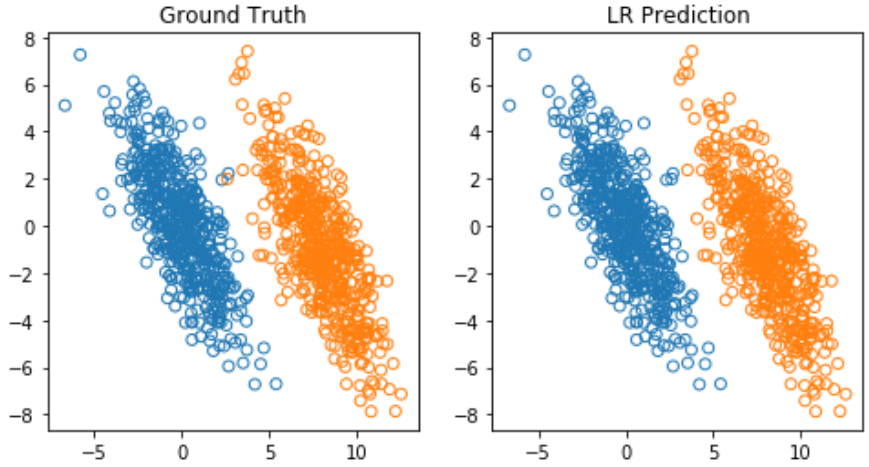
Recall that logistic regression doesn't have a closed form solution, so some of the parameters have to do with the optimization.

Logistic regression is commonly combined with **L1/L2/elastic** net type regularizations. By default scikit-learn used L2 with a weight of 1.

Models – Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

datasets = {"D1":(data1_x,data1_y), "D2":(data2_x,data2_y), "D3":(data3_x,data3_y)}

for i in datasets.keys():

    # Initialize LR model and fit to standardized data
    model = LogisticRegression()
    scaler = StandardScaler().fit(datasets[i][0]) # object that standardizes data
    model.fit(scaler.transform(datasets[i][0]),datasets[i][1])
    y_pred = model.predict(scaler.transform(datasets[i][0]))

    …print error statistics and make plots…
```

```
D1 Summary Statistics:
P: 0.999001996007984
R: 0.999
A: 0.999
f1: 0.9989999989999989
```

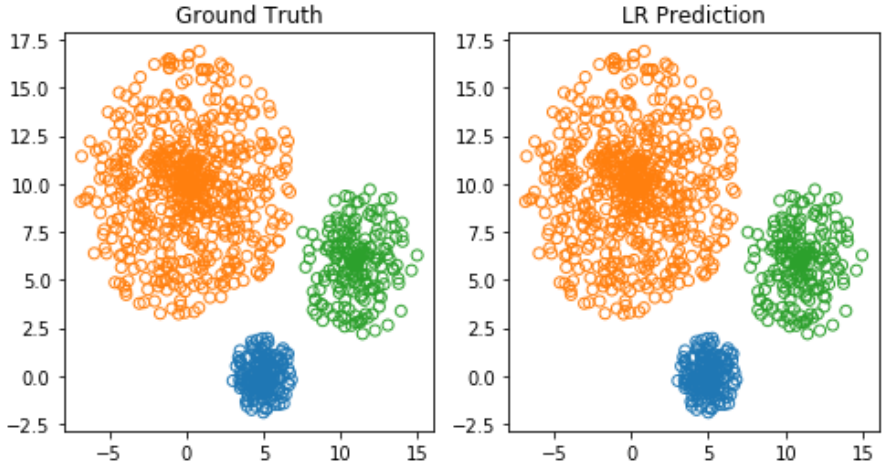


Note: we have used the StandardScalar() class from scikit learn to standardize the data.

Models – Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

datasets = {"D1":(data1_x,data1_y), "D2":(data2_x,data2_y), "D3":(data3_x,data3_y)}

for i in datasets.keys():

    # Initialize LR model and fit to standardized data
    model = LogisticRegression()
    scaler = StandardScaler().fit(datasets[i][0]) # object that standardizes data
    model.fit(scaler.transform(datasets[i][0]),datasets[i][1])
    y_pred = model.predict(scaler.transform(datasets[i][0]))

    …print error statistics and make plots…
```

```
D2 Summary Statistics:
P: 1.0
R: 1.0
A: 1.0
f1: 1.0
```

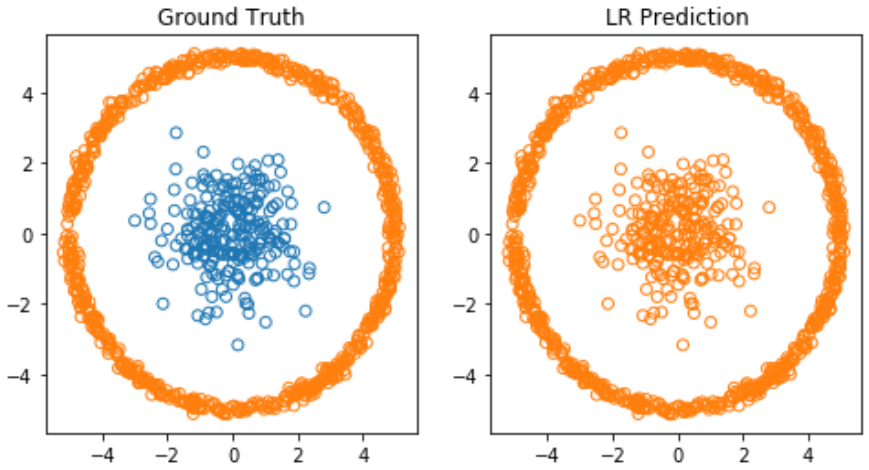


Note: we have used the StandardScalar() class from scikit learn to standardize the data.

Models – Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

datasets = {"D1":(data1_x,data1_y), "D2":(data2_x,data2_y), "D3":(data3_x,data3_y)}

for i in datasets.keys():

    # Initialize LR model and fit to standardized data
    model = LogisticRegression()
    scaler = StandardScaler().fit(datasets[i][0]) # object that standardizes data
    model.fit(scaler.transform(datasets[i][0]),datasets[i][1])
    y_pred = model.predict(scaler.transform(datasets[i][0]))

    …print error statistics and make plots…
```

```
D3 Summary Statistics:
P: 0.375
R: 0.5
A: 0.5
f1: 0.42857142857142855
```



Note: we have used the StandardScalar() class from scikit learn to standardize the data.

See notebook for the rest of the implementations