Notes for ISL Chapter 4: Classifiers

Justin Burruss

2025-08-06

Background

The Python code and notes below are for the ISL study group. This is to try out some of the concepts from Chapter 4 on the `Default` data set. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

Loading the data set

The Pandas library ("Panda" as in "**Pan**el **da**ta") makes it easy to load the CSV and offers some functionality similar to `data.frame` or `data.table` in R. First we load our data from the CSV.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from sklearn.metrics import roc_curve

# read CSV
c = pd.read_csv(r"E:\docs\Classes\ISL\Default.csv")
```

Logistic regression

We start off 4.3.1 with a linear regression model $p(X) = \beta_0 + \beta_1 X$ that we'll adapt for a simple logistic regression.

Simple Logistic regression

In 4.3.1 the text suggests coding dependent variable `Default=Yes` as `1` and `No` and `0` and choose as our independent variable `balance`. Let's do so.

```python
X = c['balance'].values
Y = c['default'].map({"Yes":1.0, "No":0.0}).values
```

The book recommends $p(X) = Pr(Y = 1|X)$ as a convenience function so that we can use $p(X) = \beta_0 + \beta_1 X$ with a logistic wrapper to ensure probabilities stay between 0 and 1 and total to 1. If we start with the logistic function

$$f(x) = \frac{e^x}{1 + e^x}$$

then plug in $\beta_0 + \beta_1 X$ we get exactly what the book provides in equation 4.2.

$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$

We could also rewrite $\frac{e^x}{1+e^x}$ as follows.

$$\begin{aligned} f(x) &= \frac{e^x}{1 + e^x} && \text{given equation} \\ &= \frac{e^x}{1 + e^x}\frac{e^{-x}}{e^{-x}} && \text{mutiplies by 1} \\ &= \frac{1}{(1 + e^x)(e^{-x})} && e^x e^{-x} = 1 \\ &= \frac{1}{(e^{-x} + e^x e^{-x})} && \text{distribute } e^{-x} \\ &= \frac{1}{(e^{-x} + 1)} && e^x e^{-x} = 1 \end{aligned}$$

Plug in $\beta_0 + \beta_1 X$ (and reorder the terms in the denominator to be easier to read) and we get the form below.

$$p(X) = \frac{1}{1 + e^{-\beta_0 - \beta_1 X}}$$

This is easy enough to code in Python. If the value in `exp()` gets too large it will throw an overflow and the function will return zero; we can use `np.clip` to avoid that.

```python
def p(X, Beta_0, Beta_1):
    return(1 / (1 + np.exp(np.clip(-Beta_0 - Beta_1 * X, -709, 709))))
```

Our dependent variable only takes on values of `0` or `1`, and the book provides a likelihood formula appropriate for this kind of Bernoulli response.

$$\ell(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} [1 - p(x_{i'})]$$

Note that if we code this up as-is we're just asking for underflow. So let's rewrite using logarithms: "the log of the product is the sum of the logs".

$$\begin{aligned} log(\ell(\beta_0, \beta_1)) &= log(\prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} [1 - p(x_{i'})]) && \text{take log of both sides} \\ &= log(\prod_{i:y_i=1} p(x_i)) + log(\prod_{i':y_{i'}=0} [1 - p(x_{i'})]) && log(ab) = log(a) + log(b) \\ &= log(\prod_{i:y_i=1} p(x_i)) + \sum_{i':y_{i'}=0} [log([1 - p(x_{i'})])] && \text{product rule of logarithms} \\ &= \sum_{i:y_i=1} [log(p(x_i))] + \sum_{i':y_{i'}=0} [log([1 - p(x_{i'})])] && \text{product rule of logarithms} \end{aligned}$$

Above formula for log likelihood would work just fine, but in Python it may be more convenient to use the fact that our dependent variable y_i is always 1 or 0, with $1 - y_i = 1$ when y_i is 0, and express log likelihood as follows.

$$\sum_{i=1}^{n} [\, y_i \, log(\, p(x_i)\,) + (1 - y_i)\, log(\, 1 - p(x_i)\,)\,]$$

Let's code that up in Python. We add in `np.clip` to avoid overflows (it works without the guardrails but leads to `RuntimeWarning:` messages in the RMarkdown output).

```
def ll(X, Y, Beta_0, Beta_1, Epsilon=1e-15):
    Y_hat = np.clip(p(X, Beta_0, Beta_1), Epsilon, 1 - Epsilon)
    return(np.sum(Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat)))
```

We should test this out against known results from the book. In table 4.1 we're given coefficients of -10.6513 and 0.0055, so a solver ought to end up with similar coefficients, and our log likelihood ought to be about the same, keeping in mind we're shown just four digits after the decimal. We'll use `minimize` from `scipy.optimize` to help validate. If we're minimizing we need to wrap our `ll` function in a negative log likelihood function `nll` to flip the sign.

```
def nll(params):
    return(-ll(X, Y, params[0], params[1]))

result = minimize(nll, [0.0, 0.0])

print("Expected coeffs of ['-10.6513', '0.0055'], found",
      ["{:.4f}".format(c) for c in result.x])
```

```
## Expected coeffs of ['-10.6513', '0.0055'], found ['-10.6510', '0.0055']
```

Looks like it ties.

Multiple Logistic regression

The book introduces multiple logistic regression by adding `income` and `student` into the mix. If we continue with the book notation we'll be typing all those `Beta_0`, `Beta_1`, `Beta_2`, `Beta_3` variables over and over again. We will rewrite our `p` and `ll` functions to work with matrices instead. Let's rewrite then first retest against what we tested above (the table 4.1 coefficients of -10.6513 and 0.0055) before running as a multiple logistic regression.

Our `Y` is unchanged from before. For the `X`, NumPy makes building the matrix easy with `column_stack` and `np.ones`. We'll use a vector `w` ("w" as in "weights") for our coefficients, and rather than hard-coding a length of 2 we take care to use the number of columns of `X` to determine the number of elements of `w`.

```
X = np.column_stack((np.ones(len(c['balance'].values)),
                     c['balance'].values))
w = np.zeros(X.shape[1])
```

We added a column of ones to our X for use with the intercept term (what was β_0). In Python we can use `.dot()`, `@`, or `.inner()` to sum the products of `w` and `X` in the way we want. Once again we use `np.clip` to avoid overflows.

```
def p(X, w):
    z = np.clip(X.dot(w), -709, 709)
    return(1 / (1 + np.exp(-z)))

def ll(X, Y, w, Epsilon=1e-15):
    Y_hat = np.clip(p(X, w), Epsilon, 1 - Epsilon)
    return(np.sum(Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat)))

def nll(w):
    return(-ll(X, Y, w))
```

Notice how we can have simply `p(X, w)` and have it work whether for 2 parameters or 20.

Next we run it and make sure it ties.

```
result = minimize(nll, w)

print("Expected coeffs of ['-10.6513', '0.0055'], found",
      ["{:.4f}".format(c) for c in result.x])
```

```
## Expected coeffs of ['-10.6513', '0.0055'], found ['-10.6510', '0.0055']
```

Looks like it ties. Now let's apply it to the multiple logistic regression shown in table 4.3 of the text. There we have `balance` as before, `income` in thousands of dollars, and `student=Yes` is coded as `1`, `No` as `0`. Let's see if we fit to the same coefficients as in the book.

```
X = np.column_stack((np.ones(len(c['balance'].values)),
                     c['balance'].values,
                     c['income'].values/1000.0, # i.e., thousands of dollars
                     c['student'].map({"Yes":1.0, "No":0.0}).values))

w = np.zeros(X.shape[1])

result = minimize(nll, w)

print("Coefficients:\n",
      "expected ['-10.8690', '0.0057', '0.0030', '-0.6468']\n",
      "   found",
      ["{:.4f}".format(c) for c in result.x])
```

```
## Coefficients:
##  expected ['-10.8690', '0.0057', '0.0030', '-0.6468']
##     found ['-10.8687', '0.0057', '0.0030', '-0.6467']
```

Results tie nicely.

We can further simplify. Let z be our stand in for $\beta_0 + \beta_1 x_i$ so that we can represent $\frac{1}{1+e^{-\beta_0-\beta_1 x_i}}$ as just $\frac{1}{1+e^{-z}}$. Then we do a little algebra to trim our log likelihood function `ll` down a bit.

$$
\begin{aligned}
\ell\ell &= \sum_{i=1}^{n}[\, y_i\, log(\frac{1}{1+e^{-z}}) + (1-y_i)\, log(\, 1 - \frac{1}{1+e^{-z}}\,)\,] \\
&= \sum_{i=1}^{n}[\, y_i\, log(\frac{1}{1+e^{-z}}) + (1-y_i)\, log(\, (1 - \frac{1}{1+e^{-z}})(\frac{1+e^{-z}}{1+e^{-z}})\,)\,] \\
&= \sum_{i=1}^{n}[\, y_i\, log(\frac{1}{1+e^{-z}}) + (1-y_i)\, log(\frac{1-1+e^{-z}}{1+e^{-z}})\,] \\
&= \sum_{i=1}^{n}[\, y_i\, log(\frac{1}{1+e^{-z}}) + (1-y_i)\, log(\frac{e^{-z}}{1+e^{-z}})\,] \\
&= \sum_{i=1}^{n}[\, y_i\, log(\frac{1}{1+e^{-z}}) + (1)\, log(\frac{e^{-z}}{1+e^{-z}}) - (y_i)\, log(\frac{e^{-z}}{1+e^{-z}})\,] \\
&= \sum_{i=1}^{n}[\, log(\frac{e^{-z}}{1+e^{-z}}) + y_i\, log(\frac{1}{1+e^{-z}}) - y_i\, log(\frac{e^{-z}}{1+e^{-z}})\,] && \text{rearrange } y_i \text{ terms} \\
&= \sum_{i=1}^{n}[\, log(\frac{e^{-z}}{1+e^{-z}}) + y_i(log(\frac{1}{1+e^{-z}}) - log(\frac{e^{-z}}{1+e^{-z}}))\,] && \text{factor out } y_i \\
&= \sum_{i=1}^{n}[\, log((\frac{e^{-z}}{1+e^{-z}})(\frac{e^{z}}{e^{z}})) + y_i(log(\frac{1}{1+e^{-z}}) - log(\frac{e^{-z}}{1+e^{-z}}))\,] \\
&= \sum_{i=1}^{n}[\, log(\frac{1}{1+e^{z}}) + y_i(log(\frac{1}{1+e^{-z}}) - log(\frac{e^{-z}}{1+e^{-z}}))\,] \\
&= \sum_{i=1}^{n}[\, log(\frac{1}{1+e^{z}}) + y_i(log(\,(\frac{1}{1+e^{-z}})/(\frac{e^{-z}}{1+e^{-z}})\,)\,)\,] && log(a)-log(b) = log(\tfrac{a}{b}) \\
&= \sum_{i=1}^{n}[\, log(\frac{1}{1+e^{z}}) + y_i(log(\,(\frac{1}{1+e^{-z}})(\frac{1+e^{-z}}{e^{-z}})\,)\,)\,] \\
&= \sum_{i=1}^{n}[\, log(\frac{1}{1+e^{z}}) + y_i(log(\frac{1}{e^{-z}})\,)\,] \\
&= \sum_{i=1}^{n}[\, log(1) - log(1+e^{z}) + y_i(log(\frac{1}{e^{-z}})\,)\,] && log(\tfrac{a}{b}) = log(a)-log(b) \\
&= \sum_{i=1}^{n}[\, log(1) - log(1+e^{z}) + y_i(log(1) - log(e^{-z})\,)\,] && log(\tfrac{a}{b}) = log(a)-log(b) \\
&= \sum_{i=1}^{n}[\, 0 - log(1+e^{z}) + y_i(0 - log(e^{-z})\,)\,] && log(1) = 0 \\
&= \sum_{i=1}^{n}[\, 0 - log(1+e^{z}) + y_i(0 - (-z)\,)\,] \\
&= \sum_{i=1}^{n}[\, 0 - log(1+e^{z}) + y_i z\,] \\
&= \sum_{i=1}^{n}[\, y_i z - log(1+e^{z})\,] && \text{reorder terms}
\end{aligned}
$$

So if we have weights w for $\beta_0, \beta_1, ...,$ we sub in $x_i w$ for z above and find that our log likelihood can be expressed as follows.

$$\ell\ell(X, Y, w) = \sum_{i=1}^{n} [\, y_i x_i w - log(1 + e^{x_i w}) \,] \qquad \text{weights w}$$

Note that x_i and w above are vectors, and $x_i w$ represents their dot product, often expressed as $\mathbf{w} \cdot \mathbf{x_i}$, $\vec{w} \cdot \vec{x_i}$, or $\mathbf{w}^\intercal \mathbf{x_i}$. In Python we do not write up a loop from 1 to n over all n of the x_i but instead handle it all in one go with `X.dot(w)`.

Coding up the above formula in Python, our `ll` code is just the following (we'll test it out once more).

```
def ll(X, Y, w):
    z = np.clip(X.dot(w), -709, 709)
    ll = np.sum( Y * z - np.log(1 + np.exp(z)) )
    return ll

X = np.column_stack((np.ones(len(c['balance'].values)),
                     c['balance'].values,
                     c['income'].values/1000.0,
                     c['student'].map({"Yes":1.0, "No":0.0}).values))

w = np.zeros(X.shape[1])

result = minimize(nll, w)

print("Coefficients:\n",
      "expected ['-10.8690', '0.0057', '0.0030', '-0.6468']\n",
      "   found",
      ["{:.4f}".format(c) for c in result.x])
```

```
## Coefficients:
##  expected ['-10.8690', '0.0057', '0.0030', '-0.6468']
##     found ['-10.8687', '0.0057', '0.0030', '-0.6467']
```

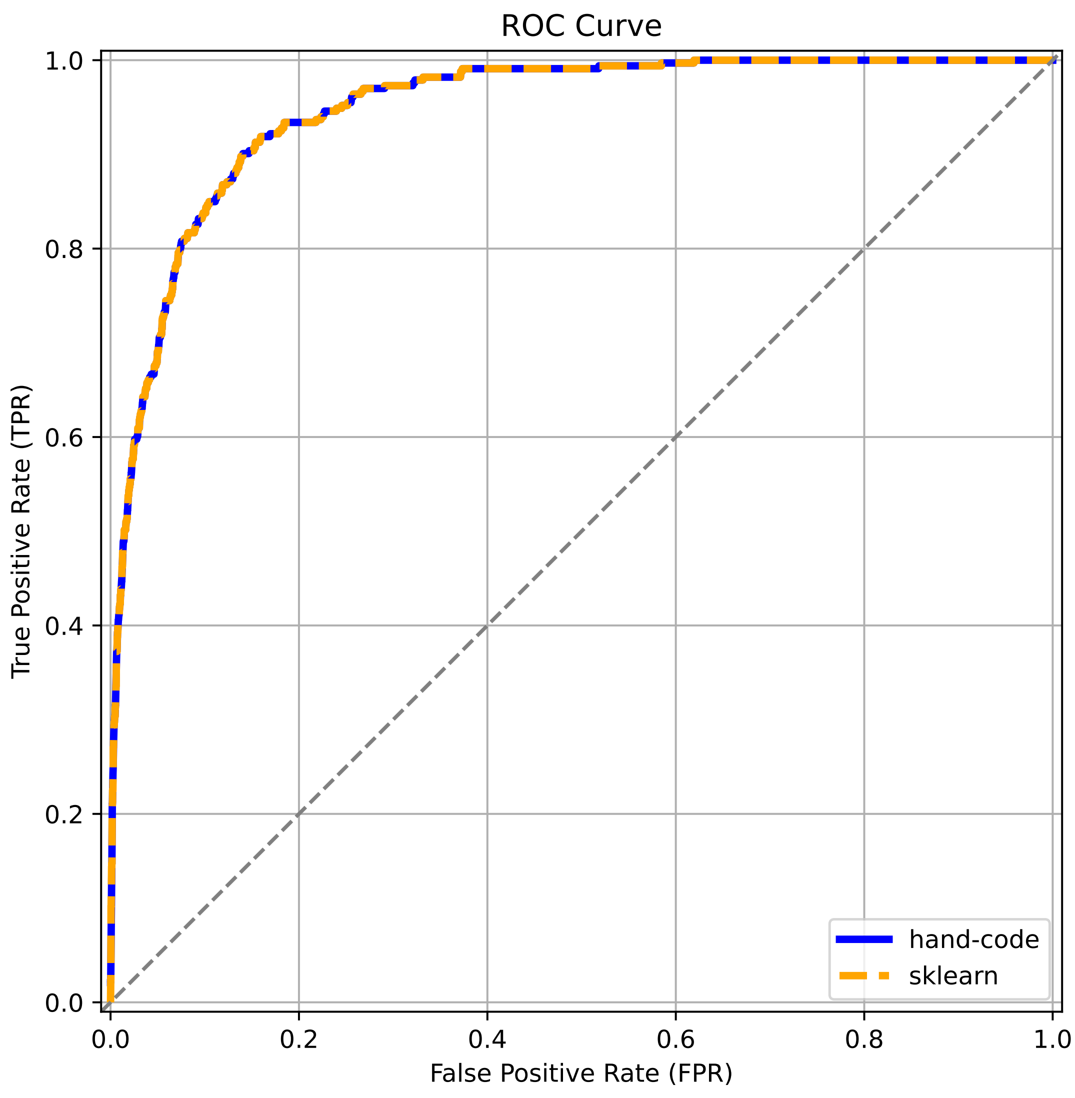
The results tie.

ROC

The book shows us an ROC curve in figure 4.8. Let's hand code our own without using `sklearn.metrics`. The book has nothing about the train/test split used and gave us no random seeds, so instead of training a model and scoring we'll just take all 10,000 rows to demonstrate ROC. We'll tie against `roc_curve` from `sklearn.metrics`.

```
P = sum(c['default']=='Yes')
N = c['default'].count() - P

Thresholds = c.groupby('balance') \
    .agg(GECnt=('balance', 'count'),
         TP=('default', lambda x: (x == 'Yes').sum())) \
    .sort_values(by='balance', ascending=False) \
    .cumsum() \
    .reset_index() \
    .assign(FP=lambda x: x['GECnt'] - x['TP'],
```

```python
            FN=lambda x: P - x['TP'],
            TN=lambda x: N - (x['GECnt'] - x['TP'])) \
    .assign(FPR=lambda x: x['FP'] / N,
            TPR=lambda x: x['TP'] / P)

ROC = Thresholds.groupby('FPR') \
    .agg(TPR=('TPR', 'max')) \
    .reset_index()

# check against sklearn result  (a one-liner)
FPR_b, TPR_b, Th_b = roc_curve(c['default'].map({"Yes":1.0, "No":0.0}),
                               c['balance'])

# blue line for hand-code, orange dashes for sklearn
plt.figure(figsize=(7, 7))
plt.plot(ROC['FPR'], ROC['TPR'], linewidth=3, label='hand-code', color='blue')
plt.plot(FPR_b, TPR_b, linewidth=3, linestyle='--', label='sklearn',
         color='orange')
plt.title('ROC Curve')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend(loc='lower right')
plt.xlim([-0.01, 1.01]);
plt.ylim([-0.01, 1.01]);
plt.grid()
plt.axline((0, 0), slope=1, color='gray', linestyle='--')
plt.gca().set_aspect('equal', adjustable='box') # square plot area
plt.show()
```



The lines match.

Pandas makes it easy to look up thresholds given a target FPR, TPR, etc.

```
# example: 20% FPR
print(Thresholds.merge(ROC.iloc[(ROC['FPR'] - 0.2).abs().argsort()[:1]]))
```

```
##        balance  GECnt   TP    FP  FN    TN       FPR       TPR
## 0  1207.694726   2244  311  1933  22  7734  0.199959  0.933934
```

```
# example confusion matrix
ex=Thresholds.merge(ROC.iloc[(ROC['FPR'] - 0.2).abs().argsort()[:1]])
print(f"Confusion Matrix:\n"
      f"                      Predicted=Yes            Predicted=No\n"
      f"Actual=Yes {ex['TP'].iloc[0]:>20} TP {ex['FN'].iloc[0]:>20} TN\n"
      f"Actual=No  {ex['FP'].iloc[0]:>20} FP {ex['TN'].iloc[0]:>20} FN")
```

```
## Confusion Matrix:
##                        Predicted=Yes            Predicted=No
## Actual=Yes                    311 TP                  22 TN
## Actual=No                    1933 FP                7734 FN
```