Notes for ISL Chapter 5: Resampling Methods

Justin Burruss

2025-08-23

Background

The Python code and notes below are for the ISL study group. This is to try out some of the concepts from Chapter 5 on the `Bikeshare` 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.

```
import pandas as pd
import numpy as np
from scipy.optimize import minimize
from scipy.stats import poisson
from scipy.stats import norm
import statsmodels.api as sm
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt

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

Setting up a Poisson regression

Chapter 5 is all about resampling methods, but before we get into that we have to have something to test against.

For the `Bikeshare` data the text (back in Chapter 4) suggests a Poisson regression and provides the following likelihood formula where $\lambda(x_i) = e^{\beta_0+\beta_1 x_{i1}+\ldots+\beta_p x_{ip}}$.

$$\ell(\beta_0, \beta_1, ..., \beta_p) = \prod_{i=1}^{n} \frac{e^{-\lambda(x_i)}\lambda(x_i)^{y_i}}{y_i!}$$

That looks like an underflow waiting to happen, so let's rewrite using logs to find the log likelihood.

$$\begin{aligned}
\ell\ell &= log(\prod_{i=1}^{n} \frac{e^{-\lambda(x_i)}\lambda(x_i)^{y_i}}{y_i!}) \\
&= \sum_{i=1}^{n} log(\frac{e^{-\lambda(x_i)}\lambda(x_i)^{y_i}}{y_i!}) && \text{product rule of logarithms} \\
&= \sum_{i=1}^{n}[\, log(e^{-\lambda(x_i)}\lambda(x_i)^{y_i}) - log(y_i!)\,] && log(\tfrac{a}{b}) = log(a) - log(b) \\
&= \sum_{i=1}^{n}[\, log(e^{-\lambda(x_i)}) + log(\lambda(x_i)^{y_i}) - log(y_i!)\,] && log(ab) = log(a) + log(b) \\
&= \sum_{i=1}^{n}[\, -\lambda(x_i) + log(\lambda(x_i)^{y_i}) - log(y_i!)\,] && log(e^a) = a \\
&= \sum_{i=1}^{n}[\, -\lambda(x_i) + y_i log(\lambda(x_i)) - log(y_i!)\,] && log(a^b) = b\, log(a) \\
&= \sum_{i=1}^{n}[\, y_i log(\lambda(x_i)) - \lambda(x_i) - log(y_i!)\,] && \text{rearrange} \\
&= \sum_{i=1}^{n}[\, y_i log(\lambda(x_i))\,] - \sum_{i=1}^{n}[\, \lambda(x_i)] - \sum_{i=1}^{n}[\, log(y_i!)\,] && \text{summation is linear}
\end{aligned}$$

This gives us the log likelihood. Keep in mind though that we want to find $\beta_0, \beta_1, ..., \beta_p$ to maximize log likelihood. Notice how the third term $\sum_{i=1}^{n} log(y_i!)$ will be just some constant that can be ignored when we're minimizing. Let's drop that third term.

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

Now we sub back in $\lambda(x_i) = e^{\beta_0+\beta_1 x_{i1}+...+\beta_p x_{ip}}$ and observe that can simplify further by the property $log(e^a) = a$.

$$\begin{aligned}
\sum_{i=1}^{n}[\, y_i log(\lambda(x_i))\,] - \sum_{i=1}^{n}[\, \lambda(x_i)\,] &= \sum_{i=1}^{n}[\, y_i log(e^{\beta_0+\beta_1 x_{i1}+...+\beta_p x_{ip}})\,] - \sum_{i=1}^{n}[\, e^{\beta_0+\beta_1 x_{i1}+...+\beta_p x_{ip}}\,] \\
&= \sum_{i=1}^{n}[\, y_i(\beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip})\,] - \sum_{i=1}^{n}[\, e^{\beta_0+\beta_1 x_{i1}+...+\beta_p x_{ip}}\,]
\end{aligned}$$

If we continue with the book notation we'll be typing all those $\beta_0, \beta_1, ...$ variables over and over again, so let's use weights w for $\beta_0, \beta_1, ...$. We just need to remember to add a column of ones to our X for the β_0 term, then we can simply write $x_i w$. Often this inner product is expressed as $\mathbf{w}^\intercal \mathbf{x_i}$ to treat these vectors as $n \times 1$ matrices, or to write as $\mathbf{w} \cdot \mathbf{x_i}$ or $\vec{w} \cdot \vec{x_i}$ to emphasize that this is a dot product between vectors of reals. Substituting in $x_i w$ we get the following.

$$\sum_{i=1}^{n}[\, y_i(\beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip})\,] - \sum_{i=1}^{n}[\, e^{\beta_0+\beta_1 x_{i1}+...+\beta_p x_{ip}}\,] = \sum_{i=1}^{n} y_i x_i w - \sum_{i=1}^{n} e^{x_i w}$$

Above is no longer log likelihood, but if we maximize it by changing w we effectively maximize log likelihood.

Since it's not exactly log likelihood we'll call it `llprime`. 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. To multiply $x_i w$ we could use `X.dot(w)` or `X @ w`, either one works; we'll try `@` this time. Notice that in Python we will not need to loop over all the x_i and y_i but instead handle it all in one go.

```
# log likelihood with a constant term dropped
def llprime(w, X, Y):
    z = np.clip(X @ w, -709, 709)
    return(np.sum(Y * z) - np.sum(np.exp(z)))
```

In Python we'll have all our y_i in a 1-dimensional array `Y`. Likewise all our x_i plus the 1 for β_0 go into a 2-dimensional array `X`. For our `X` let's choose the same variables as the book (see page 171). The Pandas library built-in `get_dummies` makes it easy to transform hour of the day, month of the year, and the weather situation into dummy variables, we just need to remember to use `drop_first=True` to avoid the dummy variable trap. The observed counts form our target `Y`. We will name our weights `w`.

```
# target variable is a count, hence Poisson model
Y = c['cnt'].values

# use same variables as page 171, one row per element of Y
X = np.column_stack((np.ones(Y.shape[0]),
                     c['workingday'].values,
                     c['temp'].values,
                     pd.get_dummies(c['weathersit'], drop_first=True),
                     pd.get_dummies(c['mnth'], drop_first=True),
                     pd.get_dummies(c['hr'], drop_first=True)))

# weights, one per column of X
w = np.zeros(X.shape[1])
```

The calls to `np.ones` and `np.zeros` used `np.shape`, so there's no need to hard code number of columns or number of rows. We can print the shapes to see how many columns and rows we have and confirm that everything is compatible.

```
np.shape(X)
```

```
## (17379, 40)
```

```
np.shape(Y)
```

```
## (17379,)
```

```
np.shape(w)
```

```
## (40,)
```

Let's validate our `llprime` against `GLM` from `statsmodels`. If we get approximately the same coefficients we probably have a correct hand-coded function. We'll write a negative log likelihood wrapper to permit the use of `minimize` from `scipy.optimize` to validate.

```python
# negative llprime so we can minimize
def nllprime(w, X, Y):
    return(-llprime(w, X, Y))

# fit the model
result = minimize(nllprime, w, args=(X, Y))
w = result.x

# fit a statsmodels Poisson for validation
sm_pois = sm.GLM(Y, X, family=sm.families.Poisson()).fit()

coeffs = pd.DataFrame({
    'hand-code': w,
    'statsmodels': sm_pois.params,
    'diff': result.x - sm_pois.params
    })

pd.set_option('display.float_format', '{:.5f}'.format)

# if these are close we probably coded llprime correctly
print("Coefficients\n", coeffs)
```

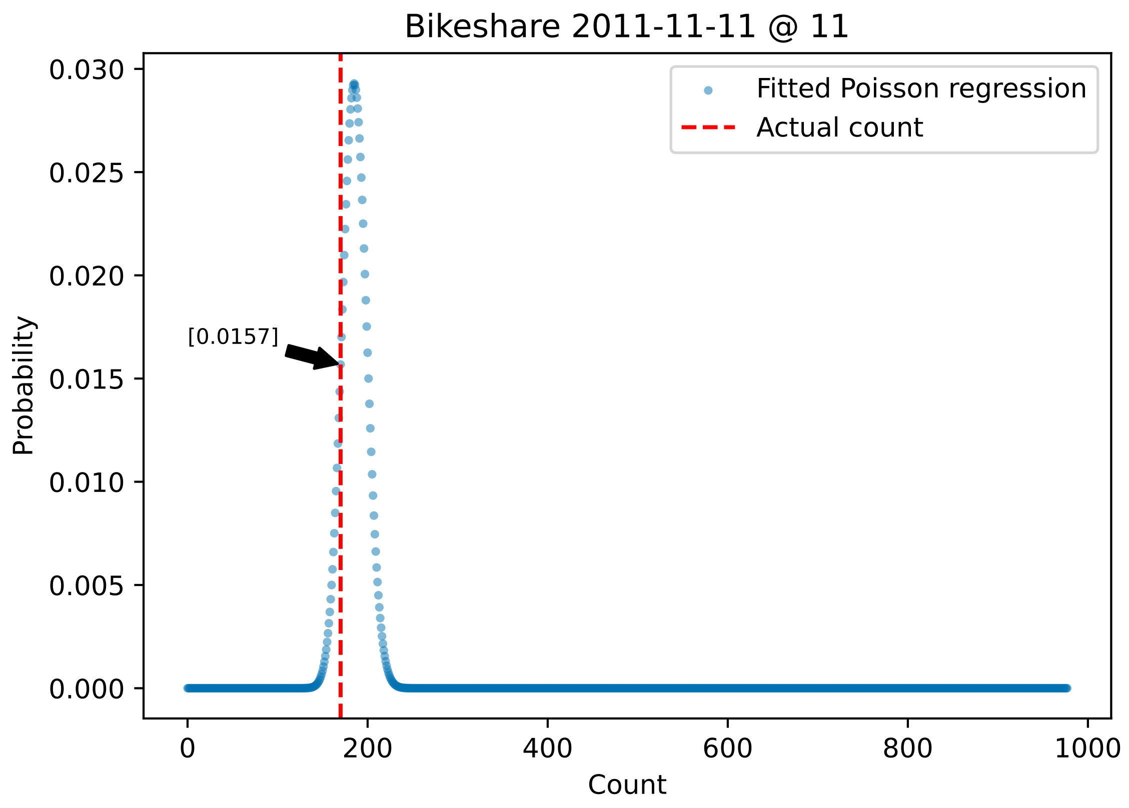
```
## Coefficients
##      hand-code  statsmodels     diff
## 0     3.04787      3.04787 -0.00000
## 1     0.04516      0.04516  0.00000
## 2     1.25624      1.25624  0.00000
## 3    -0.08601     -0.08601 -0.00000
## 4    -0.57656     -0.57656 -0.00000
## 5    -0.50143     -0.50134 -0.00008
## 6     0.10103      0.10103  0.00000
## 7     0.30162      0.30162 -0.00000
## 8     0.39583      0.39583 -0.00000
## 9     0.40946      0.40946 -0.00000
## 10    0.34783      0.34783 -0.00000
## 11    0.22055      0.22055 -0.00000
## 12    0.32016      0.32016 -0.00000
## 13    0.48267      0.48267 -0.00000
## 14    0.57090      0.57090 -0.00000
## 15    0.47278      0.47278  0.00000
## 16    0.33538      0.33538  0.00000
## 17   -0.46707     -0.46707  0.00000
## 18   -0.83837     -0.83837  0.00000
## 19   -1.50471     -1.50471  0.00000
## 20   -2.10849     -2.10850  0.00000
## 21   -0.95725     -0.95726  0.00000
## 22    0.39594      0.39594  0.00000
## 23    1.41996      1.41996  0.00000
## 24    1.91684      1.91684  0.00000
## 25    1.39397      1.39397  0.00000
## 26    1.12569      1.12569  0.00000
## 27    1.27538      1.27538  0.00000
## 28    1.45353      1.45352  0.00000
## 29    1.43450      1.43450  0.00000
## 30    1.37032      1.37032  0.00000
## 31    1.40928      1.40928  0.00000
## 32    1.63011      1.63011  0.00000
## 33    2.04018      2.04018  0.00000
## 34    1.97418      1.97418  0.00000
## 35    1.67886      1.67886  0.00000
## 36    1.38275      1.38275  0.00000
## 37    1.12506      1.12506  0.00000
## 38    0.86751      0.86751  0.00000
## 39    0.48697      0.48697  0.00000
```

Looks like they tie.

We'll need a way to measure performance. Let's first look at the fitted Poisson regression for a specific hour then decide how we want to measure things. Here it arguably looks a little cleaner to use `.dot` instead of `@`.

```python
# let's check a particular hour: 2011-11-11 @ 11.
estimated = np.exp(X[c[(c['dteday'] == '2011-11-11') &
                       (c['hr'] == 11)].index.tolist()].dot(w))
actual = Y[c[(c['dteday'] == '2011-11-11') &
                       (c['hr'] == 11)].index.tolist()]

# plot actual in a red line, compare to PMF curve
plt.figure(1)
plt.scatter(range(np.max(Y)+1),
            poisson.pmf(range(np.max(Y)+1), estimated),
            s=5,
            alpha=0.5,
            color='#0072B2',
            label="Fitted Poisson regression")
plt.axvline(x=actual, color='red', linestyle='--', label='Actual count')
plt.annotate(f'{poisson.pmf(actual, estimated).round(4)}',
             xy=(actual, poisson.pmf(actual, estimated)),
             xytext=(0, poisson.pmf(actual, estimated)+0.001),
             arrowprops=dict(facecolor='black',
                             headwidth=8,
                             headlength=8,
                             linewidth=1,
                             shrink=0.05),
             fontsize=8)
plt.legend(loc='upper right')
plt.title('Bikeshare 2011-11-11 @ 11')
plt.xlabel('Count')
plt.ylabel('Probability')
plt.show()
```



The `poisson.pmf` above is the probability mass function (PMF), so a higher value of `poisson.pmf(actual, estimated)` indicates a better fit, i.e., we're less surprised by the observation.

So how shall we score the fit? A reasonable method would be to find the arithmetic mean `poisson.pmf(Y, np.exp(X @ result.x))` across all `Y`, treating each `Y` and estimate pair equally regardless of count. Alternatively we might have selected MSE using actual and estimated count, in which case lower is better. We might also have done a weighed mean of the PMF result (i.e., weighted by count).

We ought to also see our baseline: if we simply used the mean `Y` as our λ for all observations, what would be our score?

```
print("Unfitted score =", poisson.pmf(Y, np.mean(Y)).mean().round(6))
```

```
## Unfitted score = 0.001981
```

If our fitting is any good we ought to easily beat that score.

With that settled we can move on to the subject of chapter 5.

Cross-validation

Hand-coding a cross-validation requires some way to randomly assign observations to folds. Both Python and NumPy have random modules to implement pseudo-random number generation. We'll go with `default_rng` from NumPy, taking care to set a seed for reproducibility.

```python
rng = np.random.default_rng(2025)
```

A straightforward way to carry out k-fold would be to shuffle, split into folds, then for each fold use that fold to test against the model fit using the other folds. Let's put that into a function. We'll have the function return the scores in a DataFrame.

```python
def cvk(X, Y, k=5):
    """Return a DataFrame of train & test scores from k-fold"""
    train_scores = []
    test_scores = []
    indices = np.arange(len(Y))
    rng.shuffle(indices)
    folds = np.array_split(indices, k)
    w = np.zeros(X.shape[1])

    for i in range(k):

        # for each test fold, the other folds serve as training data
        test_fold = folds[i]
        train_fold = np.concatenate([folds[j] for j in range(k) if j != i])

        # train the model
        result = minimize(nllprime, w, args=(X[train_fold], Y[train_fold]))
        w = result.x # may be a better starting place for next estimate

        # append training scores to the list
        train_scores.append(
            np.mean(poisson.pmf(Y[train_fold],
                                np.exp(X[train_fold] @ w))))

        # append test scores to the list
        test_scores.append(
            np.mean(poisson.pmf(Y[test_fold],
                                np.exp(X[test_fold] @ w))))

    return(pd.DataFrame({
        'TrainScore': train_scores,
        'TestScore': test_scores
        }, index=["Fold " + str(i) for i in range(1, k + 1)]))
```

The `index=["Fold " + str(i) for i in range(1, k + 1)]` and `[folds[j] for j in range(k) if j != i]` parts above use list comprehension. For example, the `["Fold " + str(i) for i in range(1, k + 1)` is like doing `paste("Fold", 1:5)` in R.

We need to pick a k; we'll try 5-fold.

```python
CV_5 = cvk(X, Y, 5)
```

Our last step is to take the arithmetic mean. Let's put this final result in the same DataFrame and print.

```python
CV_5 = pd.concat([CV_5,
                 pd.DataFrame({
                     'TrainScore': [np.mean(CV_5['TrainScore'])],
                     'TestScore': [np.mean(CV_5['TestScore'])]
                     }, index=['Hand-code Result'])])

CV_5['Diff'] = CV_5['TestScore'] - CV_5['TrainScore']

print(CV_5)
```

```
##                     TrainScore  TestScore     Diff
## Fold 1                 0.01055    0.01053 -0.00002
## Fold 2                 0.01064    0.01037 -0.00027
## Fold 3                 0.01048    0.01068  0.00019
## Fold 4                 0.01047    0.01120  0.00073
## Fold 5                 0.01063    0.00986 -0.00077
## Hand-code Result       0.01056    0.01053 -0.00003
```

Really not much to it: randomly split observations into folds, for each fold use that fold to test against the model fit using the other folds, then take the arithmetic mean.

Let's validate against the Sklearn implementation of cross-fold validation. We should get similar results even if the 5 folds are different.

```
kf = KFold(n_splits=5, shuffle=True, random_state=2025)
train_scores_sklearn = []
test_scores_sklearn = []

for tr, te in kf.split(X):
    # train the model
    result = minimize(nllprime, w, args=(X[tr], Y[tr]))
    w = result.x

    # append scores to the lists
    train_scores_sklearn.append(
        np.mean(poisson.pmf(Y[tr], np.exp(X[tr] @ w))))
    test_scores_sklearn.append(
        np.mean(poisson.pmf(Y[te], np.exp(X[te] @ w))))

CV_Comp = pd.concat([CV_5[-1:],
                    pd.DataFrame({
                        'TrainScore': [np.mean(train_scores_sklearn)],
                        'TestScore': [np.mean(test_scores_sklearn)],
                        'Diff': [np.mean(test_scores_sklearn) -
                                 np.mean(train_scores_sklearn)]},
                        index=['Sklearn Result'])])
print(CV_Comp)
```

```
##                     TrainScore  TestScore     Diff
## Hand-code Result       0.01056    0.01053 -0.00003
## Sklearn Result         0.01056    0.01053 -0.00003
```

Results tie, a good indicator that the hand-coded k-fold cross validation was implemented correctly.