# PredictandEvaluate

August 8, 2023

## 1 Implementing the Prediction Step and Evaluating the Model

Matrix operations are core components of machine learning math. Most of these operations are supported by existing Python machine learning packages. However, matrix operations are so fundamentally important that it is wise to be comfortable using them.

In this exercise you will practice implementing the dot-product and taking the inverse of a matrix. We will use this knowledge to build a function that implements the Logistic Regression prediction step, and to build a function that implements log loss. We will use our log loss function to evaluate the predictions. We will then compare our log loss function with scikit-learn's implementation.

### 1.0.1 Import Packages

Before you get started, import a few packages. Run the code cell below.

[1]:
```
import pandas as pd
import numpy as np
from sklearn.metrics import log_loss
```

### 1.1 Step 1. Matrix Multiplication and Inverse

We are going to start by generating a $100x3$ matrix. We'll use a multivariate random number generator to create synthetic data.

Run the code cell below.

[2]:
```
np.random.seed(1234)

mean_vector = [0, 0, 0]
cov_matrix = [[1, 0.25, 0.25],
              [0.25, 1, 0.25],
              [0.25, 0.25, 1]]

X = np.random.multivariate_normal(mean_vector, cov_matrix, size=100)
X
```

[2]:
```
array([[ 0.67972181, -0.11057272, -1.56921409],
       [ 0.84839721,  0.34868749, -0.53384987],
       [-0.59672179, -0.22358085, -1.00315975],
       [ 2.2872295 ,  0.53086168,  1.93936204],
```

```
[-0.91033033,  0.68177297, -1.79374851],
[ 0.20292096, -0.35199584,  0.14458115],
[-1.07749251,  0.0847287 , -1.80983594],
[ 0.85518078,  0.14972425,  0.38661609],
[-0.45438503, -0.88352872, -1.45830796],
[ 2.03363657,  1.02256075,  0.7982995 ],
[ 1.02206544, -0.29569695,  0.11757808],
[-0.82592181, -1.2253408 , -0.16750748],
[ 0.50696852, -0.18809168, -0.58343297],
[-2.09120215, -1.53705804, -1.44373299],
[ 0.14965934,  1.15749511, -1.38382307],
[ 0.64731806,  0.71168883,  0.54415006],
[ 0.06052467, -0.96332441, -0.69967532],
[ 0.95098217,  1.91655187,  0.19949229],
[ 0.63799613,  0.35106187,  0.17393986],
[ 1.11805732, -0.86780395, -0.17491887],
[ 0.90665598,  0.62308295,  0.53692842],
[-1.54994615,  0.36304834,  1.62919048],
[-1.28602941, -0.81104977, -2.21048159],
[-0.17158716, -0.18069665, -1.14265439],
[-1.15292526, -0.49106605,  0.15009765],
[-2.23481244, -1.15121843, -0.87324723],
[-0.11850199, -0.63526091, -0.58686795],
[ 2.62771289,  1.65681508,  3.27483241],
[ 0.42627168, -0.12426703, -0.65102452],
[ 0.28783938, -1.35443735, -1.02273253],
[-0.78325714,  0.55187029,  0.06201572],
[ 0.3555129 , -0.05762874,  0.94249048],
[-0.51210283,  0.2982219 ,  0.94524009],
[ 0.76255456, -0.0166713 ,  0.33998097],
[-0.01364974, -0.70202207, -0.35260764],
[-1.29506549, -0.32033045, -1.27698514],
[-0.24685016, -0.39091677, -1.95994756],
[ 1.73939721,  1.22082513,  0.66875288],
[ 0.62549879,  0.01511327, -0.20800513],
[ 1.56506999,  1.11744909,  1.17509983],
[-0.30720391,  0.79537047,  0.03883742],
[-0.03008528,  0.59519965, -1.49055651],
[ 0.82973839,  0.27664002, -0.08957865],
[-0.37293027, -0.68002782, -0.38663284],
[-0.12480044, -1.96278175,  0.35631488],
[ 2.04514769,  1.27477102, -1.27914738],
[ 1.79094895,  0.20393534,  1.68634658],
[-0.33271883,  0.13897985,  0.99943933],
[-0.33651545, -0.75885139, -1.40102934],
[ 0.97692498, -1.45780846, -1.6068947 ],
[-0.68600941, -0.67104707,  0.3045784 ],
```

```
[-0.52901346, -0.506836  ,  1.15611923],
[ 0.87462101,  2.01935661,  1.61090121],
[-0.76228112, -0.30204149,  0.35538174],
[-0.38145895,  1.09141191, -0.03069795],
[-0.02792445, -0.61077656,  0.1593134 ],
[-0.72484573, -1.2529412 , -0.04272476],
[-0.48682084,  1.40191181,  0.25291421],
[ 1.40290595, -0.71303878, -0.98618014],
[ 0.91974264,  0.27353526, -1.45265903],
[ 1.77584183,  2.21541515,  0.56503161],
[-1.01456706, -0.25624917,  1.30211018],
[ 0.55226376,  1.35435311,  0.62947197],
[ 0.58895541,  1.08957839,  1.34603358],
[ 1.82254509,  1.20024462,  0.10216211],
[ 0.93388088,  0.08946274, -0.10788886],
[-0.42987726,  0.55300225, -0.73486467],
[ 0.03432136,  0.54429982,  0.73658277],
[-1.11882715, -0.07768989, -1.04706175],
[ 0.50634972, -1.49950944,  0.84068337],
[ 0.37537648, -0.39728377, -1.0665043 ],
[-0.77242909, -2.74344857, -0.95540987],
[-0.21843816, -0.27174045, -1.09908192],
[ 0.87367389, -0.99819149, -1.33701242],
[-0.13070218, -0.91627654,  0.17468327],
[ 1.40821088,  1.73813261,  1.26307094],
[-0.71079365, -0.99064016, -0.68799135],
[-0.73500424,  0.14658495, -0.04582142],
[ 1.29310473,  0.5240239 ,  0.8295429 ],
[-1.48164037, -2.43521504, -1.94614326],
[ 0.80528049, -0.41661997,  0.1963439 ],
[ 1.55955943,  1.18809562, -0.51681548],
[-1.45875507, -0.76392859, -0.28459842],
[-1.41941354, -0.74461319,  1.53806761],
[ 0.99761764,  0.91599241,  1.07126702],
[ 1.28899592,  1.21515799,  0.18778942],
[ 1.43786161,  0.29850627, -0.66713181],
[ 0.46744415, -0.64332047, -0.25995924],
[ 0.04533037, -0.9991212 ,  0.31936643],
[-1.06490661, -1.17863587, -1.77355906],
[-0.24200202,  0.50032074, -0.30829196],
[ 1.20644786,  0.05464834,  0.27190181],
[ 0.23353241, -0.48081434,  0.19742117],
[-0.06989253,  0.62761609,  0.02689883],
[ 1.73989882,  0.72930328,  2.61667185],
[ 1.46542246,  0.34400256, -1.20603521],
[ 0.52379811,  0.17826703,  0.04996139],
[ 0.1657656 , -0.53683383, -1.37430191],
```

```
[-0.0582795 , -0.82720198,  0.81785385],
[ 0.03161627,  1.32150676, -0.5809413 ]])
```

We specified the covariance of this data in the variable `cov_matrix` above. Let's quickly check to see what the covariance of the resulting matrix looks like. We generally don't expect the resulting covariance matrix to be identical to what we specified because we took a small sample.

In the code cell below, use the NumPy `np.cov()` function to compute the covariance matrix of `X`. Look at the online documentation for information about this function.

Use the parameter `rowvar=False` when calling the `np.cov()` function. This makes sure we treat the columns as the features and ensures that we return a 3x3 matrix.

Assign the result to variable `cov_numpy`. Print the resulting matrix so you can visualize it.

### 1.1.1 Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

[3]:
```
# YOUR CODE HERE
cov_numpy=np.cov(X,rowvar=False)
print(cov_numpy)
```

```
[[1.02353553 0.50390451 0.39438664]
 [0.50390451 0.91435157 0.40221637]
 [0.39438664 0.40221637 1.11511219]]
```

### 1.1.2 Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

[4]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testCovNumpy

try:
    p, err = testCovNumpy(cov_numpy, X)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

Now we are going to compute the same covariance matrix manually. This is quite straightforward when the steps are outlined for you!

First we will create a centered data matrix. In the code cell below, do the following:

1. Using the NumPy `mean()` method, compute the mean of each column in array `X`. Remember to use the parameter `axis=0` to make sure we are computing the mean of each column. Assign the result to variable `X_means`.

2. We then need to subtract the mean from each column of NumPy array `X`. Create a new data matrix called `X_centered`, which is just the original NumPy array `X` with `X_means` subtracted from it.

### 1.1.3 Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

[5]:
```
# YOUR CODE HERE
X_means=np.mean(X,axis=0)
X_centered=X-X_means
```

### 1.1.4 Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

[6]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testCenteredMatrix

try:
    p, err = testCenteredMatrix(X, X_means, X_centered)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

Now that we have a centered data matrix, we will next do a matrix operation to compute the covariance matrix. The mathematical formula for this is:

$$\frac{X_c^T \cdot X_c}{N-1}$$

The matrix $X_c$ is the centered matrix, as was done above. The $T$ superscript means we are taking the transpose of the matrix. Remember, to do matrix multiplication on two matrices, the inner dimension of the two matrices need to align. Since our matrix here is 100x3, and we want a 3x3 covariance matrix, we need the left matrix to have $3x100$ dimension and the right matrix to have $100x3$.

In the code cell below, implement this covariance matrix formula using the `X_centered` matrix you created above. Remember to use the NumPy `T` attribute to get a transpose and the NumPy `dot()` method to compute the dot product. Assign the result to variable `cov_manual`.

For more information on the NumPy `T` attribute, consult the online documentation. For more information on the NumPy `dot()` method, consult the online documentation.

### 1.1.5 Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

[7]:
```
# YOUR CODE HERE
cov_manual=(np.dot(np.transpose(X_centered),X_centered))/(len(X_centered)-1)
```

### 1.1.6 Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

[8]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testCovManual

try:
    p, err = testCovManual(cov_manual, X, X_means, X_centered)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

## 2 Last, let's see if our computation of the covariance matrix matches the NumPy version. We are going to do one preprocessing step though. We may not expect the lower decimals to equal each other, so we want to round each covariance matrix to a desired tolerance and then check the equality of those rounded matrices.

In the code cell below:

1. Create a new matrix named `cov_numpy_round` using the NumPy `np.round_()` function. Use the `cov_numpy` matrix for the first argument and use the `tolerance` variable for the second argument (the decimals input).
2. Create a new matrix `cov_manual_round` using NumPy `np.round_()` function. Use the `cov_manual` matrix for the first argument and the `tolerance` variable for the second argument (the decimals input).
3. Now test for equality of the two rounded covariance matrices. In a single line do the following:
    1. Write a boolean expression to test equality between `cov_numpy_round` and `cov_manual_round`.
    2. Apply the `sum()` method on that expression to sum up the entries.
    3. Include the expression above in another boolean expression to see if the sum equals 9. This should return either `True` or `False`.
    4. Assign the result to variable `result`.

### 2.0.1 Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

[9]:
```
tolerance=10

# YOUR CODE HERE
cov_numpy_round=np.round_(cov_numpy,decimals=tolerance)
cov_manual_round=np.round_(cov_manual,decimals=tolerance)

result=np.sum(cov_numpy_round==cov_manual_round)==9
```

### 2.0.2 Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

[10]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testRound

try:
    p, err = testRound(cov_numpy_round, cov_manual_round, result, X)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

Now we will take the inverse of the covariance matrix. The inverse of the covariance matrix is used to compute a distance metric called Mahalanobis distance. This distance function is outside the scope of this exercise, but we call it out to indicate that computing an inverse covariance matrix has utility in certain ML applications. For now, our emphasis is just on the inverse computation itself.

In the cell below, do the following:

1. Use the NumPy `np.linalg.inv()` function to compute the inverse of `cov_manual`. Assign the result to variable `cov_inv`.
2. Use the NumPy `dot()` method to get the dot product of `cov_manual` and `cov_inv` to see if this returns a 3x3 identity matrix. Assign the result to variable `cov_dot` and print the variable.

For more information about the `np.linalg.inv()` function, consult the online documentation.

### 2.0.3 Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

[13]:
```
# YOUR CODE HERE
cov_inv=np.linalg.inv(cov_manual)
cov_dot=np.dot(cov_manual,cov_inv)
```

#### 2.0.4 Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

[14]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testConInv

try:
    p, err = testConInv(cov_manual, X, cov_inv, cov_dot)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

### 2.1 Step 2: Implement the Prediction Step: Predicting Probabilities

We want to implement the prediction step of a logistic regression model and then evaluate the model using our own implementation of the log loss function.

Remember the log-loss is defined by the following formula:

$L_{\text{LL}} = -\frac{1}{N}\sum_{i=1}^{N}(y_i * \log(P_i) + (1-y_i) * \log(1-P_i))$

Here, $N$ is the total numer of training examples, $y_i$ is the label for a given example and $P_i$ is the Logistic Regression model's predicted probability, which is determined by the following inverse logit function:

$P_i = P(y=1|X_i) = \frac{1}{1+e^{-(X_i \cdot W + \alpha)}}$

In this last function, $X_i$ is the feature vector for a given example, $W$ is the logistic regression weight vector, and $\alpha$ is the intercept.

Let's start by writing a function that will implement the prediction step. This involves computing probabilities. The function definition has been provided. You will complete the function by implementing the inverse logit. You will do it with a slight difference though. Instead of computing the probability at an instance level, you will do it for all examples. The only difference needed is to replace an instance level feature vector $X_i$ with the full data matrix $X$.

Complete the function in the code cell below to implement the inverse logit. This can be done in two steps: 1. Compute the linear portion $X \cdot W + \alpha$. Assign the result to variable `xw`. *Hint*: Use the `dot()` method. 2. Compute the probabilities $\frac{1}{1+e^{-xw}}$. Assign the result to variable `p`.

### 2.1.1 Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

[15]:
```
def compute_lr_prob(X, W, alpha): # Do not remove this line of code
    '''
    X = Nxk data matrix (array)
    W = kx1 weight vector (array)
    alpha = scalar intercept (float)
    '''

    # YOUR CODE HERE
    xw=np.dot(X,W)+alpha
    p=1/(1+np.exp(-xw))

    return p # Do not remove this line of code
```

[16]:
```
W = [1, -1, 2]
alpha = -1
p = (compute_lr_prob(X, W, alpha))
p
```

[16]:
```
array([0.03395633, 0.17250312, 0.03294358, 0.99038807, 0.00206697,
       0.46109855, 0.0030734 , 0.61743818, 0.02967079, 0.83308832,
       0.63481245, 0.28179079, 0.18666831, 0.01163988, 0.00836348,
       0.50598205, 0.20173244, 0.17270375, 0.40970476, 0.65385403,
       0.58841814, 0.58549837, 0.00274304, 0.03639678, 0.20397001,
       0.02124648, 0.16016822, 0.99852933, 0.1478568 , 0.19731066,
       0.09875848, 0.78551885, 0.52002814, 0.61282148, 0.26564281,
       0.01067936, 0.00836061, 0.70184055, 0.30882362, 0.85788343,
       0.11661332, 0.00988896, 0.3484047 , 0.18752574, 0.82500197,
       0.05798056, 0.98128811, 0.62882515, 0.03293522, 0.14441974,
       0.3999183 , 0.78415751, 0.7459385 , 0.32093546, 0.0734906 ,
       0.47538963, 0.3641599 , 0.08448566, 0.29808878, 0.03700088,
       0.42323432, 0.69970689, 0.36745618, 0.76699926, 0.45676442,
       0.40821254, 0.03069115, 0.49079781, 0.01574657, 0.93626865,
       0.08624674, 0.28094069, 0.04129423, 0.1415884 , 0.53368413,
       0.76785169, 0.10947293, 0.12204216, 0.80661549, 0.01910132,
       0.64898673, 0.1594713 , 0.09414669, 0.80239562, 0.77279467,
       0.36572911, 0.23238225, 0.39911502, 0.66444906, 0.01173484,
       0.08636042, 0.66721221, 0.52727019, 0.16196074, 0.99474831,
       0.09190025, 0.36481031, 0.04539029, 0.8029176 , 0.03071603])
```

### 2.1.2 Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

[22]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testProb

try:
    p, err = testProb(compute_lr_prob, X)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

## 2.2 Step 3: Implement Log-Loss

Now we will write a function to compute log loss. Again, we will provide the function definition and your role is to complete the function.

Implement the following equation $L_{\text{LL}} = -\frac{1}{N}\sum_{i=1}^{N}(y_i * \log(P_i) + (1 - y_i) * \log(1 - P_i))$. Store this as a variable called `log_loss`. You will use: 1. NumPy function `np.log()` to return the logarithm of variable `p`. 2. NumPy array method `sum()` to compute the sum of the elements contained inside the brackets in the equation above: `y_i*log(P_i)+(1-y_i)*log(1-P_i)`.

### 2.2.1 Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

[27]:
```
def compute_log_loss(y, p): # Do not remove this line of code
    '''
    y = Nx1 vector of labels (array)
    p = Nx1 vector of probabilities (array)
    '''

    n = len(y) # Do not remove this line of code

    # YOUR CODE HERE

    log_loss=-np.mean(y*np.log(p)+(1-y)*np.log(1-p))

    return log_loss # Do not remove this line of code
```

### 2.2.2 Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

[28]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
```

```
from jn import testLogLoss

try:
    p, err = testLogLoss(compute_log_loss, compute_lr_prob, X)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

### 2.3 Step 4: Use Log Loss to Evaluate the Model's Predictions

Now we will test our `compute_log_loss()` function.

First we need to generate some labels `y` to compare against our predictions. We are working off of synthetic feature data so we also need to generate synthetic labels.

Run the code cell below to generate labels.

[29]:
```
W = [1, -1, 2]
alpha = -1
y = (compute_lr_prob(X, W, alpha) > np.random.rand(X.shape[0])).astype(int)
y
```

[29]:
```
array([0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
       0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0,
       0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1,
       0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1,
       1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1])
```

Now let's evaluate our predictions using our log loss implementation. The code cell below:

1. Creates an array of probabilities called `p` using the `compute_lr_prob()` function, and using the `X`, `W` and `alpha` variables already created.
2. Computes the log loss using the `compute_log_loss()` function and using the `p` array just created and the array `y`of ground-truth labels created above. It stores the resulting log loss in a variable called `log_loss_manual`.

Run the cell and inspect the results.

[30]:
```
p = compute_lr_prob(X, W, alpha)
log_loss_manual = compute_log_loss(y, p)
log_loss_manual
```

[30]:
```
0.47841929853176457
```

### 2.4 Step 5: Compare Our Log Loss Implementation with Scikit-Learn's

Next, let's compare our `compute_log_loss()` function with scikit-learn's `log_loss()` function. The code cell below accomplishes the following:

The code cell below: 1. Computes the log loss using scikit-learn and the same `p` and `y` arrays as our log loss function above . It stores the resulting log loss in a variable called `log_loss_sklearn`. 2. Compares the resulting log loss with our own implementation.

Run the cell and inspect the results.

[31]:
```
log_loss_sklearn = log_loss(y, p)
print(log_loss_sklearn)
print(log_loss_manual == log_loss_sklearn)
```

```
0.47841929853176457
True
```

[ ]:
```

```