CS452, Spring 2018, Problem Set # 1 Solutions
Ashesi University

February 21, 2018

1. Recall that, for two random variables x and y, the covariance is defined as $cov(x, y) = E_{x,y}[xy] - E[x]E[y]$. Using the definition of expectation, and the fact that $p(x, y) = p(x)p(y)$ when x and y are independent, we obtain

$$\begin{aligned} E_{x,y}[xy] &= \sum_x \sum_y p(x,y)xy && (1) \\ &= \sum_x p(x)x \sum_y p(y)y && (2) \\ &= E[x]E[y] && (3) \end{aligned}$$

   and hence $cov(x, y) = 0$. The case where x and y are continuous variables is analogous, with sums replaced by integrals.

2. Let us denote apples, oranges and limes by a, o and l respectively. The marginal probability of selecting an apple is given by

$$\begin{aligned} p(a) &= p(a|r)p(r) + p(a|b)p(b) + p(a|g)p(g) && (4) \\ &= 3/10 \times 0.2 + 1/3 \times 0.3 + 3/10 \times 0.5 = 0.31 && (5) \end{aligned}$$

   where the conditional probabilities are obtained from the proportions of apples in each box. To find the probability that the box was green, given that the fruit we selected was an orange, we can use Bayes theorem

$$p(g|o) = \frac{p(o|g)p(g)}{p(o)} \quad . \tag{6}$$

   The denominator in Eq. 6 is given by

$$\begin{aligned} p(o) &= p(o|r)p(r) + p(o|b)p(b) + p(o|g)p(g) && (7) \\ &= 4/10 \times 0.2 + 2/3 \times 0.3 + 3/10 \times 0.5 = 0.43 && (8) \end{aligned}$$

   from which we obtain

$$p(g|o) = (3/10 \times 0.5)/0.43 = 0.3488. \tag{9}$$

3. (a) The likelihood is given by

$$\mathcal{L}(\mathcal{D};\mu) = \prod_{i=1}^{m} P(c^{(i)}|\mu) = \prod_{i=1}^{m} \mu^{c^{(i)}}(1-\mu)^{1-c^{(i)}} = \mu^H(1-\mu)^{(m-H)} \tag{10}$$

(b) The log-likelihood is

$$l(\mathcal{D};\mu) = \log \mathcal{L}(\mathcal{D};\mu) = H\log\mu + (m-H)\log(1-\mu) \tag{11}$$

We can derive the maximum of $l(\mathcal{D};\mu)$ by setting $\frac{dl(\mathcal{D};\mu)}{d\mu}$ to zero and solving for μ:

$$\frac{dl(\mathcal{D};\mu)}{d\mu} = \frac{H}{\mu} - \frac{(m-H)}{1-\mu} = 0 \tag{12}$$

From the above follows that $\mu = H/m$.

(c)

```
import numpy as np
def q3_likelihood(mu, m, H):
    # Returns the likelihood for different values of mu, given the scalar parameters m and H.
    #
    # INPUT
    # mu: N-dimensional numpy.ndarray vector of type 'float' containing N different values for mu
    # m: int
    # H: int
    #
    # OUTPUT
    # lik: N-dimensional numpy.ndarray vector of type 'float' containing the likelihood values associated with the entries of mu

    lik = ((1 - mu)**(m - H)) * (mu**H)

    return lik
```

(d)

```
import numpy as np
def q3_prior(mu, a, Z):
    # Returns the prior for multiple values of mu, given the parameters a and Z.
    #
    # INPUT
    # mu: N-dimensional numpy.ndarray vector of type 'float' containing N different values for mu
    # a: int
    # Z: float
    #
    # OUTPUT
    # prob: N-dimensional numpy.ndarray vector of type 'float' containing the prior probabilities associated with the entries of mu


    prob = ((1-mu)**(a-1)) * (mu**(a-1)) / Z

    return prob
```

(e) We want to maximize the log-posterior $\log \mathcal{P}(\mu|\mathcal{D})$:

$$\begin{aligned} \log \mathcal{P}(\mu|\mathcal{D}) &= \log \mathcal{L}(\mathcal{D};\mu) + \log p(\mu; a) + \text{const} && (13) \\ &= (H+a-1)\log\mu + (m-H+a-1)\log(1-\mu) + \text{const} && (14) \end{aligned}$$

By setting $\frac{d\log \mathcal{P}(\mu|\mathcal{D})}{d\mu} = 0$, we derive $\mu = (H+a-1)/(m+2a-2)$.

(f) We can view the prior as providing $2(a-1)$ additional observations: $a-1$ heads and $a-1$ tails. A large value of a will bias the estimate of μ toward 0.5.

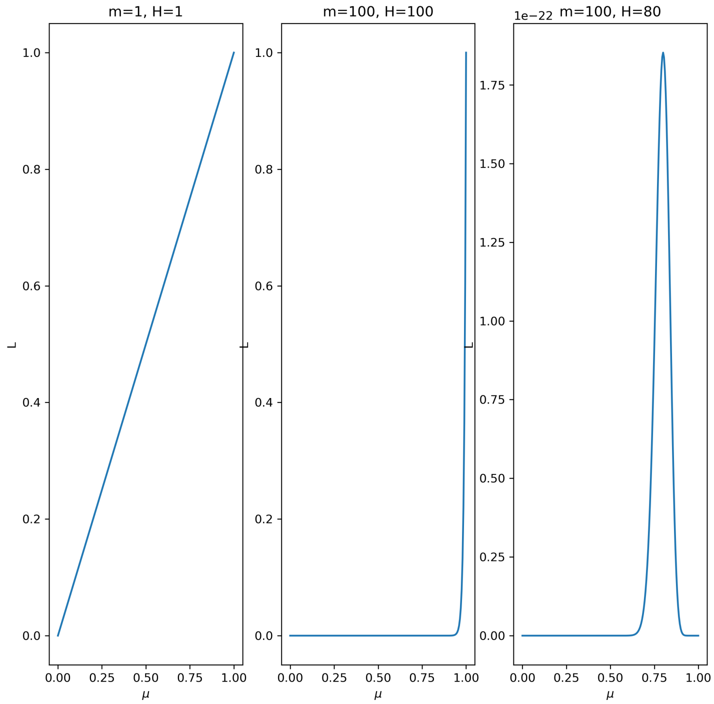

Figure 1: The result of running `q3_c.py` using the function `q3_likelihood.py` given above.

(g)

```
import numpy as np
def q3_posterior(mu, m, H, a, Z):
    # Returns the posterior for multiple values of mu, given the parameters m, H, a, and Z.
    #
    # INPUT
    # mu: N-dimensional numpy.ndarray vector of type 'float' containing N different values for mu
    # m: scalar
    # H: scalar
    # a: scalar
    # Z: scalar
    #
    # OUTPUT
    # prob: N-dimensional numpy.ndarray vector of type 'float' containing he posterior values associated with the entries of mu

    prob = ((1-mu)**(m - H)) * (mu**H) * ((1-mu)**(a-1)) * (mu**(a-1)) / Z

    return prob
```

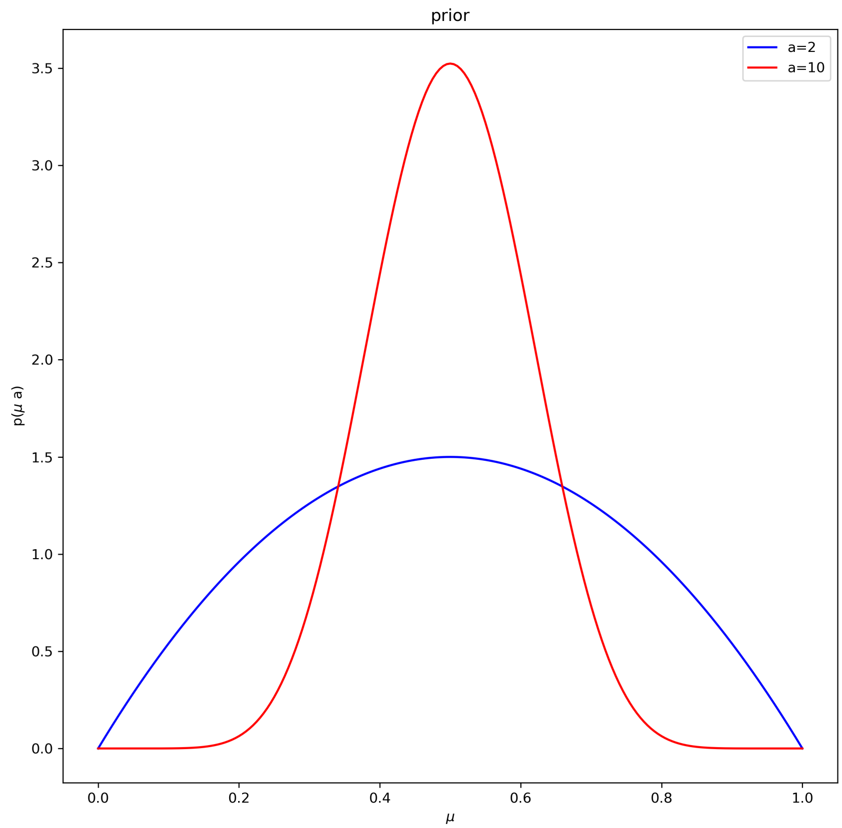

Figure 2: The figure generated by `q3_d.py` using the function `q3_prior.py` given above.

4. (a)

```
import numpy as np
def q4_features(X, mode):
# Given the data matrix X (where each row X[i,:] is an example), the function
# computes the feature matrix B, where row B[i,:] represents the feature vector
# associated to example X[i,:]. The features should be either linear or quadratic
# functions of the inputs, depending on the value of the input argument 'mode'.
# Please make sure to implement the features according to the *exact* order
# specified in the text of the homework assignment.
#
# INPUT:
# X: a numpy.ndarray matrix of size [m x d] and type 'float' where each row
# is a d-dimensional input example
# mode: specifies the type of features;
# it is a 'str' that can be either 'linear' or 'quadratic'.
#
# OUTPUT:
# B: a numpy.ndarray matrix of size [m x n] and type 'float', with each row
# containing the feature vector of an example

    m = X.shape[0]
    d = X.shape[1]

    if mode == 'linear':
        B = np.hstack((np.ones((m, 1)), X))
```

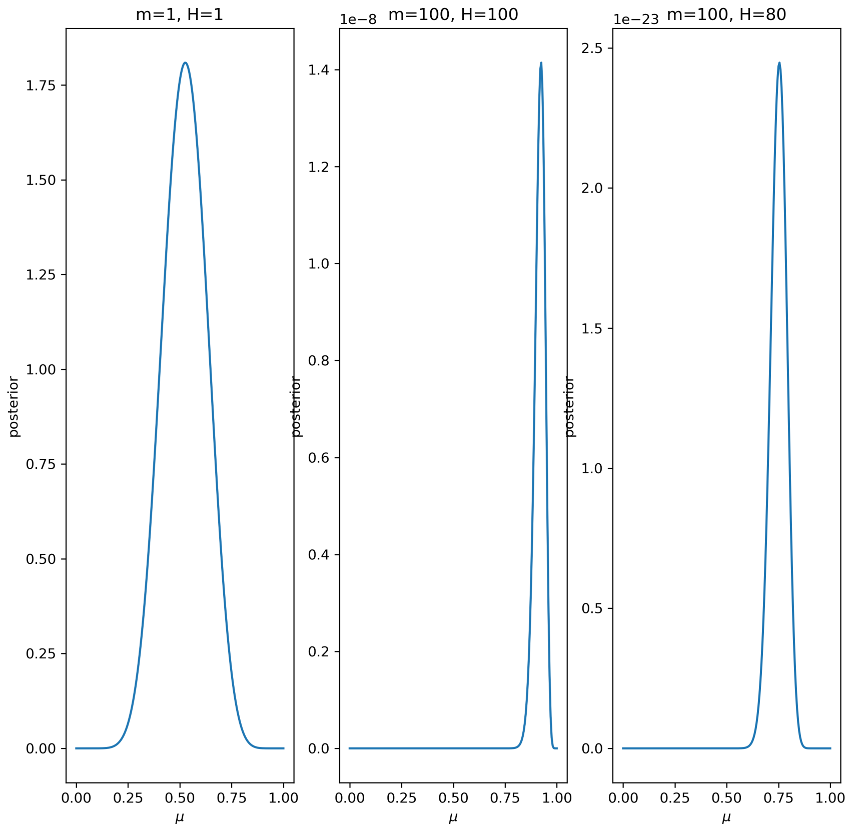

Figure 3: The posterior distributions plotted by `q3_g.py` using the function `q3_posterior.py` given above.

```
    elif mode == 'quadratic':
        n = int(d+1+(d*(d+1))/2)
        B = np.zeros((m, n))

        B[:,:d+1] = np.hstack((np.ones((m, 1)), X))
        count = d+1;
        for i in range(d):
            for j in range(i, d):
                B[:,count] = X[:,i] * X[:,j]
                count = count + 1
    else:
        print('Error, only linear and quadratic forms are supported');
        return []

    return B

import numpy as np
def q4_mse(pred_Y, correct_Y):
# This function calculates the Mean Squared Error given two sets of output
# values, one set corresponding to the correct values, the other set
# representing the output values predicted by a regression model
# INPUT:
# pred_Y: a numpy.ndarray vector of type 'float' containing m predicted values
# correct_Y: a numpy.ndarray vector of type 'float' containing m correct values
```

```
#
# OUTPUT:
# err: 'float' representing the Mean Squared Error
#

        residual = correct_Y - pred_Y
        err = np.dot(np.transpose(residual), residual) / residual.size

        return err


import numpy as np
from q4_features import q4_features
def q4_train(X, Y, lambdaval, mode):

# Trains the regularized least squares regression model using the closed form
# solution given the training data X, Y.
#
# INPUT:
# X: a numpy.ndarray matrix of size [m x d] and type 'float' where each row
# is a d-dimensional input example
# Y: a numpy.ndarray vector of size [m x 1] and type 'float', where the
# i-th element is the correct output value for the i-th input example.
# lambda: 'float' regularization hyperparameter
# mode: specifies the type of features;
# it is a 'str' that can be either 'linear' or 'quadratic'.
#
# OUTPUT:
# theta: a numpy.ndarray vector of size [n x 1] and type 'float'
# containing the learned model parameters.
#

    # computes the B matrix
    B = q4_features(X, mode);

    # computes closed-form solution of \theta, without regularizing the bias
    n = B.shape[1]
    lambda_eye = lambdaval*np.eye(n)
    lambda_eye[0,0] = 0;

    theta = np.linalg.solve(np.dot(np.transpose(B), B) + lambda_eye, np.dot(np.transpose(B), Y))

    return theta


import numpy as np
from q4_features import q4_features
def q4_predict(theta, X, mode):
# Predicts the output values of the input examples X, given the learned parameter vector theta.
#
# INPUT
# theta: a numpy.ndarray vector of size [n x 1] and type 'float'
# containing the learned model parameters.
# X: a numpy.ndarray matrix of size [m x d] and type 'float' where each row
# is a d-dimensional input example
# mode: specifies the type of features;
# it is a 'str' that can be either 'linear' or 'quadratic'.
#
# OUTPUT
# pred_Y: a numpy.ndarray vector of size [m x 1] and type 'float' containing
# the m predicted values
#

    B = q4_features(X, mode)
    pred_Y = B.dot(theta)

    return pred_Y
```

(b)

```
import numpy as np
import math
from q4_train import q4_train
from q4_predict import q4_predict
from q4_mse import q4_mse
def q4_cross_validation_error(X, Y, lambdavec, mode, N):
# Calculates the cross-validation errors for different values of lambdavec, given the
# training set X, Y.
#
# ** Implementation notes **
# - As discussed in class, you should first randomly permute the examples, before starting the
# cross-validation stage. Here we did it for you: we created Xr and Yr which are obtained from
# X and Y by permuting examples. You should use Xr and Yr in your code (not X and Y)
# - Do not change/initialize/reset the Python pseudo-number generator.
#
# INPUT
```

```
# X: a numpy.ndarray matrix of size [m x d] and type 'float' where each row
# is a d−dimensional input example
# Y: a numpy.ndarray vector of size [m x 1] and type 'float', where the
# i−th element is the correct output value for the i−th input example.
# lambdavec: a numpy.ndarray vector of size [k x 1] and type 'float'
# containing the set of regularization hyperparameter values
# mode: specifies the type of features;
# it is a 'str' that can be either 'linear' or 'quadratic'.
# N: 'int' representing the number of folds for the cross−validation stage
#
# OUTPUT
# error: a numpy.ndarray vector of size [k x 1] and type 'float'
# containing the cross−validation error (i.e., the average of the mean
# squared errors over the N validation sets) for each value in lambdavec.
#


# ******** DO NOT TOUCH THE FOLLOWING 5 LINES *******************
    np.random.seed(0)
    m = X.shape[0]
    idxperm = np.random.permutation(m)−1
    Xr = X[idxperm,:]
    Yr = Y[idxperm]
# ***************************************************************

    # make sure to use Xr and Yr in your code, NOT X and Y (read Implementiation notes in the header)
    error = np.zeros(lambdavec.size)
    count = 0
    for lambdaval in lambdavec:
        # N−fold cross validation
        err = 0
        for k in range(N):
            # cut out a part from the training set as the testing set for cross
            # validation
            st = math.floor(m / N * k)
            en = math.floor(m/ N * (k + 1))
            Xtest = Xr[st:en, :]
            Ytest = Yr[st:en]
            Xtrain = np.vstack((Xr[0:st,:], Xr[en:,:]))
            Ytrain = np.concatenate((Yr[0:st], Yr[en:]))
            # Traininghe
            theta = q4_train(Xtrain, Ytrain, lambdaval, mode)
            # Testing
            pred_values = q4_predict(theta, Xtest, mode)
            # Calculate error
            err = err + q4_mse(pred_values, Ytest)

        # we average the error over the number of chunks
        error[count] = err / N
        count = count + 1

    return error
```

(c) Looking at the cross-validation errors in Figure 4 it is clear that both models underfit the data for large values of λ. The model based on linear features exhibits underfitting for $\lambda > 10$, while the nonlinear regression function underfits clearly only when $\lambda > 10^5$.

(d) The linear model does not show any signs of overfitting. On the other hand the nonlinear model exhibits overfitting for $\lambda < 10$.

(e) The nonlinear model incurs in overfitting problems when the regularization term is given small weight. This happens because the non-linear features provide the model with a large number of degrees of freedom. As a result the model may capture and represent even patterns due to noise or specific idiosyncrasies of the training set. Such problems can be prevented by using a large regularization term which forces the function to be smooth.

The linear model does not overfit: it has very few parameters and thus the function can only capture the main pattern of the data.

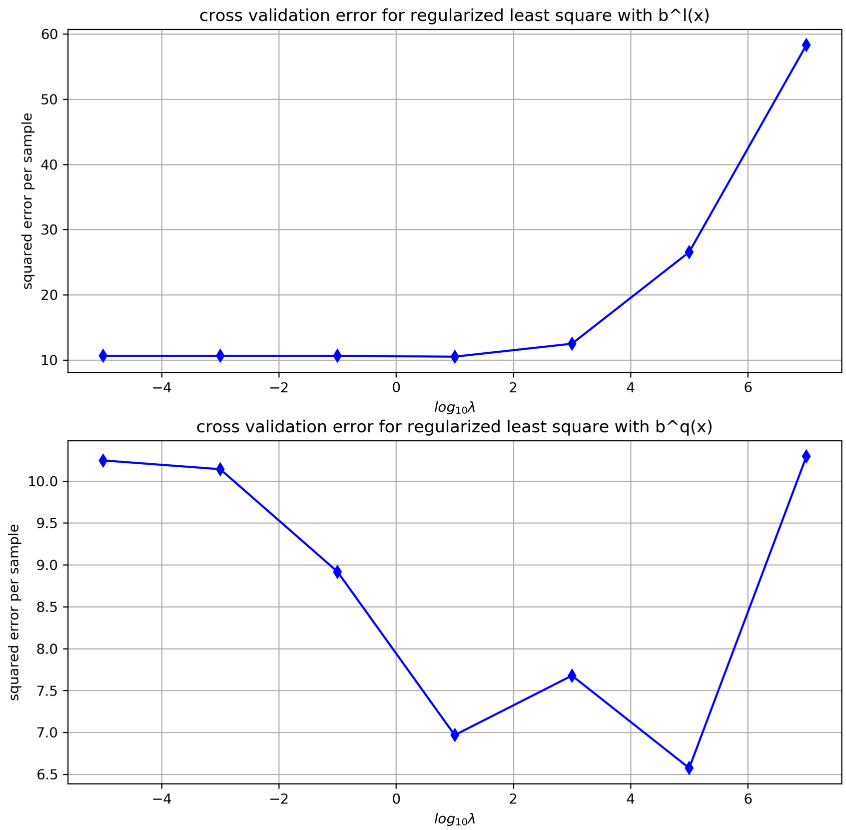

Figure 4: The figure plotted by `q4b.py` using the function `q4_cross_validation_error.py` given above.

(f)
```
import numpy as np
from q4_train import q4_train
from q4_predict import q4_predict
from q4_mse import q4_mse
def q4_test_error(X, Y, Xtest, Ytest, lambdavec, mode):
# Given training and test set, it trains the model and calculates the test error.
#
# INPUT
# X: a numpy.ndarray matrix of size [m x d] and type 'float' where each row
# is a d−dimensional input training example
# Y: a numpy.ndarray vector of size [m x 1] and type 'float', where the
# i−th element is the correct output value for the i−th input training example.
# Xtest: a numpy.ndarray vector of size [M x d] and type 'float', where
# each row is a d−dimensional test example
# Ytest: a numpy.ndarray vector of size [M x 1] and type 'float',
# containing the output values of the test examples
# lambdavec: a numpy.ndarray vector of size [k x 1] and type 'float'
# containing the set of regularization hyperparameter values
# mode: specifies the type of features;
# it is a 'str' that can be either 'linear' or 'quadratic'.
#
# OUTPUT
# error: a numpy.ndarray vector of size [k x 1] and type 'float'
```

```
# containing the test errors, one for each value in lambdavec.
#

    count = 0
    error = np.zeros(lambdavec.size)
    for lambdaval in lambdavec:
        theta = q4_train(X, Y, lambdaval, mode)
        pred_values = q4_predict(theta, Xtest, mode)
        error[count] = q4_mse(pred_values, Ytest)
        count = count + 1

    return error
```

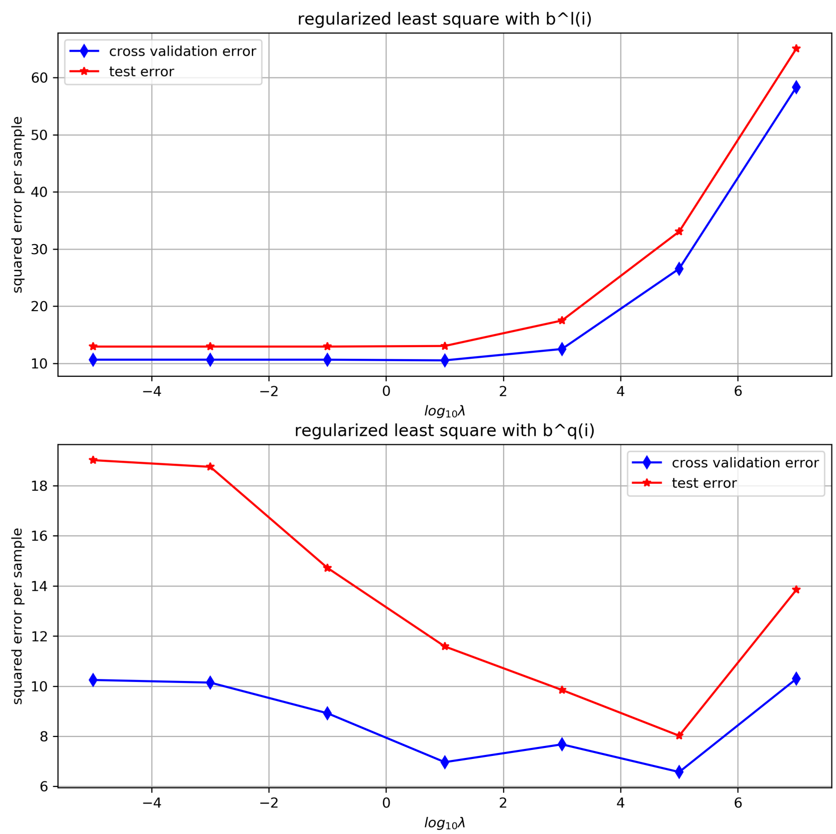

Figure 5: The figure generated by `q4f.py` using the function `q4_test_error.py` given above.

(g) The curves of the cross-validation scores obtained by varying the regularization parameter λ match quite closely the shapes of the test error curves. However, the cross validation scores tend to underestimate the generalization error. This occurs when the training set and the test set have slightly different distributions.