CSCI2291 Homework 5

Jack Moffatt

March 3, 2022

Problem 1

(a) We will use the following code as our solution.

```
import sklearn.datasets as skd
import numpy as np

diabetes = skd.load_diabetes()
data = diabetes['data']
cov_matrix = np.cov(data, rowvar = False)
print(f"{'Covariance Matrix Shape:' :<45}{cov_matrix.shape}")
```

We see that printing this result, we are given that

```
>>>  Covariance Matrix Shape:                    (10, 10)
```

We understand that this answer makes sense as there are 10 non-target attributes of this data set. The 10 columns each represent an attribute while the 10 rows also each represent an attribute. The cell indexed at row i, column j, represents the corelation coefficient between the attribute represented by the i^{th} row and the attribute represented by the j^{th} column.

So we see this 10 x 10 matrix has a diagonal of 1's as, every attribute has a perfect correlation to itself.

(b) We will use the following code as solution:

```
bmi_idx, hdl_idx = 0, 0
for i, feat in enumerate(diabetes.feature_names):
    if feat == 'bmi':
        bmi_idx = i
    if feat == 's3':
        hdl_idx = i
print(f"{'Correlation of bmi to hdl:' :<45}{cov_matrix[bmi_idx][hdl_idx] :.5f}")
```

When looking at the documentation, it can be found that the hdl attribute is labeled as **s3** as a feature label. So accessing the datasets list of feature names, which is an ordered list correlating to the order of the order of the columns in the dataset, we search for the indexes of the **bmi** feature and the **s3** (hdl) feature.

Then, as explained in the latter half of the solution in part **a**, we index into the covariance matrix using the **bmi** index as the row index, and the **s3** index as the column index. (*Switching these indices would yield the same result.*)

We see that we find the correlation coefficient to be

```
>>>  Correlation of bmi to hdl:                -0.00083
```

We conclude, from this result, that there seems to be little to no correlation between the **bmi** attributes and the **s3** attributes.

(c) Despite concluding that there is a very small corelation between the **bmi** and **hdl** attributes, since we found there corelation to be negative, we can still logically hypothesize that patients with a higher **bmi** would have a lower hdl reading in relation to those patients with a lower **bmi** reading.

To test this hypothesis, we will use the following code:

```
bmi_mean = np.mean(data[:,2])
hdl_above_bmi_mean = data[:,6][data[:,2] > bmi_mean]
hdl_below_bmi_mean = data[:,6][data[:,2] < bmi_mean]
median_above = np.median(hdl_above_bmi_mean)
median_below = np.median(hdl_below_bmi_mean)
print(f"{'Median of hdl for bmi values above mean:' :<45}{median_above :.5f}")
print(f"{'Median of hdl for bmi values below mean:' :<45}{median_below :.5f}")
```

First, we calculate the mean of our **bmi** column. Then, using boolean indexing, we index the **hdl** column by first taking the indices where the **bmi** is greater than the mean, and then repeat with the indices where the **bmi** is less than the mean.

Finally, we take the median of each column and print them to compare. We see that our results yield:

```
>>> Median of hdl for bmi values above mean:     -0.02131
>>> Median of hdl for bmi values below mean:     0.00814
```

So we see that our hypothesis is correct. As a patients **bmi** increases, there **hdl** tends to be slightly lower. We would need to do much more testing to determine if there is any true relationship between these attributes, but on the surface level their does appear to be at least a small correlation.

(d) One way that we could further investigate this relationship is by examining the effect size between the two attributes. We will measure the effect size as calculated by Cohen's d value. As we are given in the problem set

$$d = \frac{\overline{a} - \overline{b}}{s}$$

where

$$\begin{aligned} s &= \sqrt{\frac{\sum_{i=1}^{|a|}(a_i - \overline{a})^2 + \sum_{i=1}^{|b|}(b_i - \overline{b})^2}{|a| - 1 + |b| - 1}} \\ &= \sqrt{Var(a) + Var(b)} \end{aligned}$$

So we use the following code as our solution:

```
1    s = math.sqrt((np.var(hdl_above_bmi_mean) + np.var(hdl_below_bmi_mean)))
2    d = round(abs(((np.mean(hdl_above_bmi_mean)) - (np.mean(hdl_below_bmi_mean))) / s), 3)
3    print(f"{'Cohen\'s d Value:' :<45}{d}")
```

We see that Choen's d value is

```
1    >>>  Cohen's d Value:                          0.472
```

From this we can interpret that there is a moderate effect between the two attributes. A d value of .8 and above is considered to be significant while the range of .4 to .8 is considered to e moderate.

Problem 2

(a) We will use the following code as a solution.

```
from sklearn.impute import SimpleImputer as Simp
from sklearn.linear_model import LinearRegression as Lnr

cholesterol = skd.fetch_openml(name = 'cholesterol', version = 1, as_frame=True)
imp = Simp(missing_values=np.nan, strategy='mean')
imp.fit(cholesterol.data)
data = imp.transform(cholesterol.data)

lin_reg = Lnr().fit(data, cholesterol.target)
score = round(lin_reg.score(data, cholesterol.target), 2)
print(f"{'R^2 Goodness of Fit Score:' :<45}{score}")

predict_target = lin_reg.predict(data).astype(int)
real_target = cholesterol.target.astype(int)
num_correct = np.count_nonzero(predict_target == real_target)
frac_correct = round(num_correct / (predict_target.shape[0]), 2)
print(f"{'Fraction of Correct Predictions:' :<45}{frac_correct}")
```

We start by importing the SimpleImputer class and the LinearRegression class. Then, using `sklearn.datasets` we fetch the cholesterol dataset from openml, loading the data as a `DataFrame`.

Next, we clean the data using the simple imputer. Now the data is ready to be fit to a Linear Regression. Using the entire dataset and the entire target column, we fit the regression and check the score. We see that are regression has an r^2 goodness of fit score of

```
>>> R^2 Goodness of Fit Score:                0.13
```

Then, we want to compute the number of times that our regression correctly predicts the target value. We use the predict method to get a prediction of our target values and convert it to a column of type integer. Then we compare it to the real target values column in integer form, using NumPy to count the number of non-zero columns, which will count every instance that a boolean value is `True`. Then we divide this number by the number of rows in our target column. We print the result:

```
>>> Fraction of Correct Predictions:           0.01
```

So we see that our model correctly predicted the target value 1% of the time.

(b) We will use the following code as a solution:

```
import random

train_scores, test_scores = [], []
for i in range(5000):
    indx = list(range(len(data)))
    random.shuffle(indx)
    train_indx, test_indx = indx[:202], indx[202:]
    train_set, test_set = data[train_indx], data[test_indx]
    train_target, test_target = cholesterol.target[train_indx], cholesterol.target[test_indx]
    lin_reg = Lnr().fit(train_set, train_target)
    train_scores.append(lin_reg.score(train_set, train_target))
    test_scores.append(lin_reg.score(test_set, test_target))

print(f"{'Median R^2 Score on Training Set:' :<45}{round(np.median(train_scores), 2)}")
print(f"{'Median R^2 Score on Testing Set:' :<45}{round(np.median(test_scores), 2)}")
```

We begin by importing the random package. Then we initialize two empty lists to hold the scores of the scores on the training set and the testing set respectively. Then we enter a for loop that will iterate 5000 times. This for loop is intending to help serve against the random package given us a biased split of indices.

In each iteration we initialize a list of `indx` to hold each row index of our data set. Then, we shuffle this list to randomize its order. Then we split the list into two groups. The first 202 indices for the training set and the latter 101 indices for the testing set.

Then we split our data set into a training set and testing set and we repeat the same steps for our target column.

We initialize a Linear Regression and fit it to our training set, then we append the fit score on both the training set and the testing set and append those values to their respective lists. Upon completion of the for loop, we print the median value of both of these lists to see how our linear regression performs both on our training set and on our testing set. We see that

```
>>>  Median R^2 Score on Training Set:           0.14
>>>  Median R^2 Score on Testing Set:            0.01
```

So we see that our regression has a similar score on the training set as on the whole set in part **a**, but the score on the testing set is incredibly poor (1%).

Problem 3

Now we will compute our own linear regression following the steps outlined in class. First, let's define these steps mathematically. If we let X be our dataset and y be our target column, then let $\tilde{X}$ be our dataset with an extra column of ones appended to it. Let our column of coefficients with an intercept row be denoted $\tilde{c}$. We see that

$$\begin{aligned} \tilde{X} * \tilde{c} &= y \\ \tilde{X}^T * \tilde{X} * \tilde{c} &= \tilde{X}^T * y \\ (\tilde{X}^T * \tilde{X})^{-1} \cdot (\tilde{X}^T * \tilde{X}) * \tilde{c} &= (\tilde{X}^T * \tilde{X})^{-1} \cdot (\tilde{X}^T * y) \\ \tilde{c} &= (\tilde{X}^T * \tilde{X})^{-1} * (\tilde{X}^T * y) \end{aligned}$$

where $*$ represents matrix multiplication. So, we need to compute the matrix product $(\tilde{X}^T * \tilde{X})^{-1}$ and $(\tilde{X}^T * y)$, then dot these two products together. This will yield a column of length 14 which will serve as the coefficient matrix.

We use the following code, (using some variables previously established in Question **2**):

```
from numpy.linalg import pinv

data_mod = np.append(data, np.ones((303,1)), axis = 1)      #\tilde{X}
target = cholesterol.target                                 #y

left_term = pinv(np.dot(data_mod.T, data_mod))
right_term = (np.dot(data_mod.T, target))
c_self = np.dot(left_term, right_term)                      #\tilde{c}
print("Calculated c-column:")
print(np.around(c_self, decimals = 2))

lin_reg = Lnr(fit_intercept=True).fit(data, target)
c_np = lin_reg.coef_
print("sklearn.LinearRegression c-column:")
print(np.around(c_np, decimals = 2))

diff = np.around((c[:13] - c_column), decimals = 20)
print("Difference Vector Between sklearn.LinearRegression and Custom Regression:")
print(diff)
print(f"{'Sum of Difference Vector Terms:' :<45}{sum(diff)}")
```

We start by importing `pinv` to use for computing matrix inverses. Then we append a column of ones to our data matrix and initialize a local variable to

store the target column. Then we calculate $(\tilde{X}^T * \tilde{X})^{-1}$ in line 6.[1] Then, we calculate the matrix product $(\tilde{X}^T * y)$ in line 7.

Now we have 14 x 14 matrix as our left term and a 14 x 1 vector as our right term. So we see that we can perform a compatible multiplication operation to return a 14 x 1 column vector as our coefficient vector. We then print our computed coefficient column:

```
1    >>> Calculated c-column:
2        [  1.   -24.16   1.7    0.12  -3.8    7.43   0.29   8.38   0.67  -5.05  2.79   1.6    0.77  132.31]
```

Then we initialize and fit a linear regression as in Question **2** and print its coefficient column:

```
1    >>> sklearn.LinearRegression c-column:
2        [  1.   -24.16   1.7    0.12  -3.8    7.43   0.29   8.38   0.67  -5.05  2.79   1.6    0.77]
```

Initially, we can see that the `sklearn` coefficient column only has 13 terms. This because this regression model did not compute an intercept term. Now to compare the two vectors we will need to subtract the first 13 terms of our computed coefficient vector from the coefficient vector computed by **sklearn**. We print our vector and see that

```
1    >>> Difference Vector Between sklearn.LinearRegression and Custom Regression:
2        [-0.  0.  0.  0. -0. -0. -0. -0.  0. -0.  0. -0. -0.]
```

the difference vector is 0 up to 20 decimals of precision. This tells us that our custom linear regression is essentially equivalent to the regression computed by `sklearn`. We can also see that the sume of all the terms in our difference vector is:

```
1    >>> Sum of Difference Vector Terms:             5.6922244500000015e-12
```

[1] We use `np.dot()` to compute the matrix product because on 2D arrays `np.dot()` acts as standard matrix multiplication.

Problem 4

(a) We will use the following code as a solution:

```
from sklearn.preprocessing import PolynomialFeatures as Poly

poly = Poly(degree = 2)
quad_data = poly.fit_transform(data)
print(f"{'Original Data Shape:' :<45}{data.shape}")
print(f"{'Polynomial Data Shape:' :<45}{quad_data.shape}")
```

We import the `PolynomialFeatures` class from `sklearn.preprocessing`. Next, we initialize a Polynomial Features transformation of degree 2. Then we fit our data to the transformation. Comparing the original dataset size to the size of the polynomial dataset we see that:

```
>>> Original Data Shape:                    (303, 13)
>>> Polynomial Data Shape:                  (303, 105)
```

We can explain the relationship by looking at the quadratic expansion of 13 variables and a constant term. We can see that

$$
\begin{aligned}
&(a_1 + a_2 + \cdots + a_{13} + c)^2 \\
&(a_1 + a_2 + \cdots + a_{13} + c)(a_1 + a_2 + \cdots + a_{13} + c) \\
&a_1^2 + a_1a_2 + a_1a_3 + \cdots a_1a_{13} + ca_1 + a_2a_1 + a_2^2 + \cdots a_{13}^2 + c^2 \\
&a_1^2 + \cdots a_{13}^2 + 2a_1a_2 + \cdots 2a_{12}a_{13} + ca_1 + \cdots + ca_{13} + c^2
\end{aligned}
$$

We can now try to verify the number of terms in this polynomial. We see there is an a_i^2 term for each $i \in 1, 2, \ldots 13$. Next, we see that we have a $2a_ia_j$ term for each $i, j \in 1, 2, \ldots 13$ such that $i \neq j$. So, by combinatorics, we have 13 choices for our a_i term and 12 choices for our a_j term.

Also, since $2a_ia_j = 2a_ja_i$, we can see that we should not count repetitions.[2] So the number of $2a_ia_j$ terms is $\frac{13 \cdot 12}{2} = 78$.

Then we see that we also have 13 terms of the form ca_i for $i \in 1, 2, \ldots 13$. Additionally, we have a c^2 term. Adding up the numbers of all of these

[2] In other words, order doesn't matter.

terms we see that we have the sum our 13 a^2 terms, 78 $2a_i a_j$ terms, 13 ca_i terms and c^2 term is:

$$\begin{array}{r} 13 \\ 78 \\ 13 \\ +\ 1 \\ \hline 105 \end{array}$$

So we conclude that `sklearn` builds a polynomial feature extraction transformation by adding a constant column to the data set and then since we set our degree to 2, it creates new columns by performing a quadratic expansion of the 13 terms and the constant term.

This is how our quadratic dataset results in having 105 columns.

(b) We will use the following code for the solution:

```
lin_reg = Lnr().fit(quad_data, cholesterol.target)
score = lin_reg.score(quad_data, cholesterol.target)
print(f"{'Linear Regression on Polynomial Data Score:' :<45}{score}")

train_scores = []
test_scores = []
for i in range(5000):
    indx = list(range(len(quad_data)))
    random.shuffle(indx)
    indx_train, indx_test = indx[:202], indx[202:]
    quad_train, quad_test = quad_data[indx_train], quad_data[indx_test]
    target_train, target_test = target[indx_train], target[indx_test]

    lin_reg_2 = Lnr().fit(quad_train, target_train)
    train_scores.append(lin_reg_2.score(quad_train, target_train))
    test_scores.append(lin_reg_2.score(quad_test, target_test))
print(f"{'Median R^2 Score on Training Set:' :<45}{round(np.median(train_scores), 3)}")
print(f"{'Median R^2 Score on Testing Set:' :<45}{round(np.median(test_scores), 3)}")
```

We begin by initializing a Linear Regression and fitting it to our entire quadratic dataset. We then print the r^2 score and see that

```
>>> Linear Regression on Polynomial Data Score: 0.37427746208493384
```

This is a much better fit than we originally received on the original dataset in Question **2**. We will now try to train this dataset on a training portion of the data and then test it on unseen data points. Following the same steps outlined in Question **2**, we initialize a for loop to iterate 5000 times and for each iteration split our data and target values in a $2:1$ ratio.

Then we fit a linear regression to each training set and append it scores on the training set to a list and do the same for its score on the test set. Upon completion of the for loop we compute and print the median r^2 score for both the training and test sets:

```
>>> Median R^2 Score on Training Set:          0.274
>>> Median R^2 Score on Testing Set:           -2.351
```

We see that the training score is significantly lower on the smaller "training" sets than on the entire dataset. This is likely due to having less data points to average out outlier data points. However, what is more significant is the score on the testing sets.

This suggests that the regression fits the data significantly worse than just a simple horizontal line. This is likely due to the fact that using polynomial feature extraction likely overfit to our training set.