Brendan Sherman

Prof. Alvarez

CSCI 2291

March 3, 2022

Homework 5 Master Writeup

1. a) Compute the covariance matrix of the set of non-target attributes of the data set. What is the shape (size) of the resulting matrix?

Explanation:

The code below produces the attached output by computing the covariance matrix for the set of non-target attributes within the Diabetes dataset, using *np.cov()* on the non-target data, given by *d.data*. I then use the *shape* property of the ndarray to print the covariance matrix shape, which is given as (10, 10). This indicates 10 rows and 10 columns, which is to expected as there are 10 non-target attributes, and the covariance matrix should be square with dimension (# of attributes). The parameter *rowvar=False* specifies that columns, not rows, should be treated as attributes.

Code:

```
from sklearn.datasets import load_diabetes

import numpy as np

#Computes covariance matrix of non-target attributes

def q1a(d):

      return np.cov(d.data, rowvar=False)

def main():

      d = load_diabetes() #Store diabetes dataset (contains target and non
      target matrices)

      print("Shape of covariance matrix (for non-target attributes): " +
      str(q1a(d).shape))
```

Output:

```
(base) C:\Users\Brend\Documents\Spring 2022\Data Science\homeworks\hwc
Shape of covariance matrix (for non-target attributes): (10, 10)
```

b) Compute the correlation of the bmi and hdl attributes, directly from the elements of the covariance matrix. Explain your procedure.

Explanation:

The code below uses the matrix generated in part a to extract the correlation between the bmi and hdl attributes. Using the definition of a covariance matrix from class, I recalled that every point (i, j) within a covariance matrix stores the covariance between attributes (columns) **i** and **j.** Thus, because in the original dataset the bmi attribute was stored in column 2 and the hdl in column 6, the value at point (2, 6) within the covariance matrix should then contain the covariance between the two attributes. I therefore use NumPy indexing to access this value, printing it to the output.

Code:

```
#Uses matrix to extract correlation between bmi (col 2) and hdl (col 6) attributes

def q1b(d):

	cov_matrix = q1a(d)

	bmi_hdl_corr = cov_matrix[2, 6]

	return bmi_hdl_corr

def main():

	d = load_diabetes() #Store diabetes dataset (contains target and non target matrices

	print("Correlation between bmi and hdl attributes: " + str(round(q1b(d), 6)))
```

Output:

```
dCorrelation between bmi and hdl attributes: -0.000832
```

c) Based on the result of the preceding part, would you expect patients in this data set with higher bmi to have higher hdl than those with lower bmi, or lower hdl, on average? Explain. Check whether the data are consistent with your assessment, by computing the median hdl among patients whose bmi is larger than the mean and the median hdl among patients whose bmi is lower than the mean. Do the results agree with your expectations from the preceding part?

Explanation: Because the covariance (-.000832) calculated in part b is slightly negative, an interpretation would be expecting patients in this data set with a higher bmi to have, on average, **a lower hdl** (and vice versa). As covariance between two attributes is the mean of the product of differences between both attributes and their mean, negative covariance indicates that a patient's bmi is generally above the mean when their hdl is below the mean, and vice versa. To test this, the code below computes median hdl among patients with bmi larger than and lower than the mean bmi in the dataset. I accomplish this by splitting the rows in the dataset using Boolean indexing, then separately calculating the median of the hdl column for both groups, printing both to the output. The results **do** agree with the expectations above; in this dataset patients with bmi greater than the mean have a lower (-.021) median hdl than those with bmi less than the mean (.08), exemplifying the implication of negative covariance.

Code:

```
#Verifying correlation from part b

def q1c(d):

      bmi_above_mean = d.data[d.data[:,2] > np.mean(d.data[:,2])] #All rows with bmi above mean

      bmi_below_mean = d.data[d.data[:,2] < np.mean(d.data[:,2])] #All rows with bmi below mean

      med_hdl_above = np.median(bmi_above_mean[:,6])

      med_hdl_below = np.median(bmi_below_mean[:,6])

      return [med_hdl_above, med_hdl_below]

def main():
```

```
    d = load_diabetes() #Store diabetes dataset

    test_medians = q1c(d)

    print("Median hdl among patients with bmi greater than the mean: " +
str(round(test_medians[0], 3)))

    print("Median hdl among patients with bmi less than the mean: " +
str(round(test_medians[1], 3)))
```

Output:

```
Median hdl among patients with bmi greater than the mean: -0.021
Median hdl among patients with bmi less than the mean: 0.008
```

d) In order to gauge the difference between the means of the hdl values for the above-mean bmi and below-mean bmi groups from the preceding part, compute the associated effect size as measured by Cohen's d.

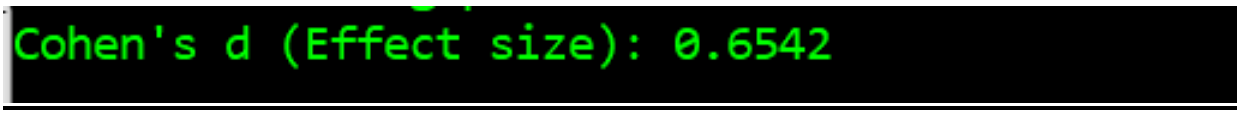
Explanation: The code below computes the effect size (via Cohen's d) to gauge difference between mean hdl values for those with above-mean and below-mean bmi. The output value, .6542, indicates that the groups differ by .6542 (pooled) standard deviations, which Cohen would label a "medium" effect size. To accomplish this, I used numpy's variance, size, and mean functions. I first use the variance and size of both groups to calculate the pooled standard deviation, which is the denominator of Cohen's d. I then divide the difference between means of both groups by this pooled standard deviation, yielding the correct value for d which is then printed to output. I also take the absolute value of this value, as the sign of Cohen's d is meaningless (simply reflecting order of means in the numerator).

Code:

```
#Compute effect size, as measured by Cohen's d

def q1d(d):

    a = d.data[d.data[:,2] > np.mean(d.data[:,2])] #All rows with bmi above mean

    b = d.data[d.data[:,2] < np.mean(d.data[:,2])] #All rows with bmi below mean

    a_size = a[:,6].size
```

```
    b_size = b[:,6].size

    a_var = np.var(a[:,6], ddof=1)

    b_var = np.var(b[:,6], ddof=1)

    #Uses above values to calculate pooled standard deviation

    pooled_sd = ((a_var * (a_size - 1) + b_var * (b_size - 1)) / (a_size + b_size - 2)) ** (1/2)

    #Final value for Cohen's d (magnitude)

    return abs(((np.mean(a[:,6]) - np.mean(b[:,6])) / pooled_sd))
def  main():

    d = load_diabetes()

    print("Cohen's d (Effect size): " + str(round(q1d(d), 4)))
```

Output:



2. In this task, you will work with the cholesterol data set (version 1) from openml.org, using sklearn.linear_model.LinearRegression to build linear regression models.

a) Use sklearn.impute.SimpleImputerfor this purpose. Next, fit a LinearRegression model to the data set. Report, to two digits of precision, the r squared goodness of fit metric for the model after fitting, as well as the fraction of instances for which the model's predictions match the corresponding target values exactly (again to two digits). Compute the latter value directly in numpy. Discuss the results, comparing the overall assessments implied by the two metrics. Do the two metrics paint a similar picture about the predictive quality of the model? Explain any disagreements between the two.

Explanation:

The code below imports the cholesterol dataset, imputes it to address missing values, and finally fits a linear regression model to the data and then reports two metrics of the model's predictive quality. The impute function handles all missing values, using the *fit* and *transform* functions included in the imputer class to replace all NaN values with the mean of the attribute they fall under. I then use the LinearRegression class and it's *fit* function to fit a

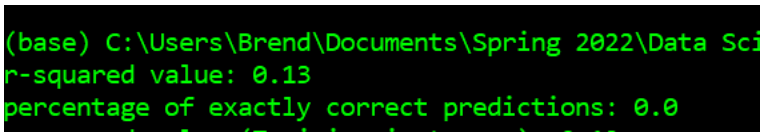
regression model to the dataset, then using *score()* on this model to get the r squared "goodness of fit" variable. Finally, I use Boolean indexing on the model's predictions, accessed using *predict(),* to calculate the fraction of these predictions exactly equal to the target value within the dataset. The results of these calculations, as shown in the output, yield an r-squared value of 0.13 and a proportion of 0. The r-squared value of 0.13 is low, indicating that the model explains about 13% of the target value's variance around its mean, which reflects poorly on predictive quality. This poor performance is also reflected in the proportion of correct predictions, which evaluates 0, meaning the model did not exactly predict the target value for any of the patients within the dataset. While both metrics seem to present the model's performance as poor, they seemingly disagree in that the model should explain 13% of the variance of the target attribute, yet predicts 0% of the target values correctly. However, as the target attribute (cholesterol) is *continuous*, it is unrealistic to expect exactly correct predictions, but rather predicted values "close" to the target. Therefore, while there are 0 correct predictions made by the model (suggesting it doesn't work at all), the r squared of .13 suggests that there is *some* (albeit poor) predictive quality within the model.

Code:

```
#Instantiate imputer to replace missing values within dataset
def impute(ds):
    imp = SimpleImputer(missing_values = np.nan, strategy="mean")
    imp = imp.fit(ds.data)
    ds.data = imp.transform(ds.data)


#Trains and evaluates linear regression model
def q2a(ds):
    #Handle missing values
    impute(ds)
    #Fit a linear regression model to the data, using actual target values
    model = LinearRegression(fit_intercept=True).fit(ds.data, ds.target)
    #Evaluate model performance by "goodness of fit"
    r_sq = model.score(ds.data, ds.target)
```

```
    #Evaluate model performance by % of exactly correct predictions
    predictions = model.predict(ds.data)
    prop_correct = ((predictions[predictions[:] == ds.target[:]].size)) /
ds.target.size
    return [r_sq, prop_correct, model.coef_, model.intercept_]
def main():
    ds = fetch_openml(name="cholesterol", version=1)
    scores = q2a(ds)
    print("r-squared value: " + str(round(scores[0], 2)))
    print("percentage of exactly correct predictions: " +
str(round(scores[1], 2)))
```

Output:



b) Split the list of row indices of all of the instances of the data set randomly into two non-overlapping parts, with sizes in a two-to-one ratio (the first twice the size of the other). Report two r2 goodness of fit values for this model: first, the r2 value for the training instances; second, the r2 value for the test instances. Discuss the results. Are the r2 values the same? If not, which of the two represents a better fit? Is the result surprising?

Explanation:

The code below produces the attached output by splitting the dataset into training and test portions, training a linear regression model on the training portion, then evaluating this model using both the training and test portions (calculating r squared for both). To split the dataset, I use random.shuffle() on all row indices within the set, then using indexing to split the shuffled indices into a 2:1 ratio. I then fit a regression model, using the procedure mentioned above, to the training portion, finally evaluating the model on both the training and test portions. As shown in the output, the model yielded an r-squared value of .12 when tested on the training data, and .07 when tested on the test data. To this extent, the value of .12 represents a better fit, meaning that the model "fits" (or better explains the variance of the target for) the training data. This is to be expected, as the model was trained *using* the training data, and as such should capture its trends more accurately than the "out of sample"

test data, which the model was not fitted to. Because the test portion is out of sample, and thus can be seen as representing the larger population, these results reflect poorly on the model's out of sample performance

Code:

```
def q2b(ds):
    impute(ds)
    #Randomly split indices into two groups (2:1 ratio)
    indices = list(range(0, ds.data.shape[0]))
    random.shuffle(indices)
    training_indices = indices[0:202]
    test_indices = indices[202:]

    #Use above indices to split datset into training, target sections
    training_data = ds.data[training_indices, :]
    test_data = ds.data[test_indices, :]
    training_targets = ds.target[training_indices]
    test_targets = ds.target[test_indices]

    #Use training data to train regression model
    model = LinearRegression().fit(training_data, training_targets)

    #Evaluate model against training, test data
    return[model.score(training_data, training_targets),
model.score(test_data, test_targets)]
def main():
    ds = fetch_openml(name="cholesterol", version=1)
```

```
    r_sqs = q2b(ds)

    print("r-squared value (Training instances): " + str(round(r_sqs[0],
2)))

    print("r-squared value (Test instances): " + str(round(r_sqs[1], 2)))
```

Output:

```
r-squared value (Training instances): 0.12
r-squared value (Test instances): 0.07
```

3. Compute the linear regression fit to the cholesterol data set from the first part of the preceding problem, but doing all of the matrix computations yourself, in numpy. Specify the resulting vector of all of the coefficient values, including the bias coefficient, c0, and those associated with the non-target attribute Explain each step carefully in your writeup, using mathematical notation, and include your code, as well. Report the full coefficient vector obtained. Report, also, the full coefficient vector found via the LinearRegression class, as in the first part of the preceding problem, and the difference between the two vectors.

Explanation:

The code below accomplishes the task of fitting a linear model to the cholesterol dataset, doing all matrix computations manually in numpy rather than using the predefined LinearRegression class. To do so, I first insert a column of "1s" into column index 0 of the non-target data, accounting for the intercept coefficient. I then create an empty array, the same length as the number of attributes + 1, to store each coefficient. I then use the normal equations to calculate the value of vector c that minimizes MSE (and thus has the best predictive quality) as follows:

$(X * X^T) * C = X^T y$ **(when derivative of MSE wrt c = 0)**

$C = (X * X^T)^{-1} * X * y$

Where X represents the non-target data (with extra column), X^T is the transpose of X, C is the coefficient vector, and y is the target data (from the dataset). The code below essentially implements these matrix operations in python, using np.transpose(), np.pinv(), and np.dot() to represent transposing, finding the pseudo inverse of, and multiplying matrices (respectively). I use the pseudoinverse function to account for the case when ($X *$

X^T) is not invertible. Finally, I print the resulting coefficient vector, then comparing it with the coefficients found (non-manually) in the previous question. The element-wise differences between the two, all very close to 0, indicate that my method was successful; the coefficients (and intercept) calculated were all roughly equal to those generated by SKL.

Code:

```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.linear_model import LinearRegression
from q2_regression import q2a, impute

#Q3: Compute linear regression fit from q2, manually doing matrix calculations
def q3(ds):
    #Instantiate imputer to replace missing values within dataset
    impute(ds)
    #Preprocessing: add column of 1s to dataset (for constant term)
    attributes = np.empty((ds.data.shape[0], ds.data.shape[1]+1))
    attributes[:, 0] = 1
    attributes[:, 1:] = ds.data
    #Determining coefficient vector (using normal equations)
    c = np.empty((ds.data.shape[1]+1, 1))
    x_t = np.transpose(attributes)
    c = np.linalg.pinv(x_t.dot(attributes)) #psuedoinverse
    c = c.dot(x_t).dot(ds.target) #normal equations, minimize coefs wrt MSE

    #Accessing correlation, intercept values from problem 2
    prev = q2a(ds)
    c_prev = prev[2]
    c_prev = np.insert(c_prev, 0, prev[3])
    return [c, c_prev, c-c_prev]
def main():
    ds = fetch_openml(name="cholesterol", version=1)
```

```
result = q3(ds)
print("Coefficient Vector (manually computed): ")
print(result[0])
print("Coefficient Vector (using SKL): " )
print(result[1])
print("Difference between vectors (element-wise): ")
print(result[2])
```

Output:

```
Coefficient Vector (manually computed):
[ 1.32311322e+02  1.00101156e+00 -2.41631881e+01  1.69948302e+00
  1.18493635e-01 -3.79889397e+00  7.42987029e+00  2.87603228e-01
  8.38127579e+00  6.74288843e-01 -5.05061181e+00  2.78855408e+00
  1.59863510e+00  7.74364275e-01]
Coefficient Vector (using SKL):
[ 1.32311322e+02  1.00101156e+00 -2.41631881e+01  1.69948302e+00
  1.18493635e-01 -3.79889397e+00  7.42987029e+00  2.87603228e-01
  8.38127579e+00  6.74288843e-01 -5.05061181e+00  2.78855408e+00
  1.59863510e+00  7.74364275e-01]
Difference between vectors (element-wise):
[-8.25906454e-10  3.65862896e-12  2.47588616e-11  2.40398812e-11
  1.17081345e-12 -1.86024529e-11 -3.29425376e-12  2.17226237e-12
  9.98134908e-12 -6.24045260e-12  3.32445182e-11 -3.13260529e-12
  1.69109171e-12 -3.75899312e-12]
```

4. Generate a new version of the cholesterol data set (same as in the preceding tasks) that uses an expanded set of attributes through a quadratic feature transformation as discussed in class

a) Apply PolynomialFeatures to obtain the new version of the data set using quadratic features. Report the sizes of the original non-target data matrix and of the transformed non-target data matrix. Explain in detail how the new attributes relate to the original ones. In particular, express the number of non-target attributes of the new data set in algebraic notation, in terms of the number of non-target attributes of the original data set

Explanation:

The code below produces the attached output by using feature extraction to obtain a new version of the dataset (using quadratic features). To do this, I use the PolynomialFeatures class within SKL, specifying a degree of two to indicate quadratic transformation. I then call *fit_transform()* on the non-target data matrix, generating a transformed non-target matrix. I then print the shape of the original matrix (303, 13) and of the transformed matrix (303, 105). These results indicate that the dataset was expanded from 13 columns

(attributes) into 105, with the number of rows (examples) staying the same. These 105 transformed attributes, as per quadratic feature extraction, should contain a column of ones, a column for each original attribute, a column for the square of each original attribute, and a column for each "interaction term" or unique pairing of attributes (multiplied together). We can thus express the number of expected attributes algebraically, using the fact that $n(n-1)/2$ is the number of pairs that can be formed from a set of n distinct items.

$$d_t = 1 + 2(d) + d(d-1)/2$$

(general equation, where d is the original number of non-target attributes and d_t is the number of non-target attributes within the transformed data matrix.)

Thus, because we know the dataset has 13 attributes, we can use this equation to verify the size of the transformed non-target data matrix found above.

$d_t = 1 + 2(13) + \frac{13(13-1)}{2} = 105$, which is equal to the number of attributes within the transformed non-target data matrix.

Code:

```
import numpy as np

import random

from sklearn.datasets import fetch_openml

from sklearn.linear_model import LinearRegression

from sklearn.preprocessing import PolynomialFeatures

from q2_regression import impute


def q4a(ds):

      impute(ds) #Handle missing data values

      poly = PolynomialFeatures(degree=2)

      transformed = poly.fit_transform(ds.data)
```

```
        return (transformed, ds.data.shape, transformed.shape)

    def main():
        ds = fetch_openml(name="cholesterol", version=1)
        results = q4a(ds)
        print("Size of original non-target data matrix: " +
str(results[1]))
        print("Size of transformed non-target data matrix: " +
str(results[2]))
```

Output:

```
(base) C:\Users\Brend\Documents\Spring 2022\Data Science
Size of original non-target data matrix: (303, 13)
Size of transformed non-target data matrix: (303, 105)
```

b) Train a LinearRegression model on the new, transformed data set from the preceding part. Report the resulting r squared fit score. Next, generate a 2-to-1 train-to-test split. Train a linear regression model on the training portion, and report the r squared fit score of that trained model on the test portion. How do the r2 values of the models trained on the transformed data compare with those of the corresponding models that were trained on the original data set (in prior tasks)? Discuss.

Explanation:

The code below produces the attached output by training a linear regression model on the transformed dataset from part A and calculating the r-squared goodness of fit for that model. It also splits the transformed data into training and test portions, as done previously, fitting the model to the training portion and then testing it against the test portion via r squared. The code for this part was very similar to the previous parts, performing the same operations (splitting, imputing, fitting a model, and calculating r squared) on the transformed dataset. The first r-squared fit score, for the model generated using the entire dataset, is substantially higher than those seen before (0.37). This suggests that the model trained using quadratic feature extraction has a higher predictive

quality than those trained on the original dataset, accounting for more of the variance in the target value. However, when the transformed dataset was split into training and test portions, the model performed poorly, yielding an average r-squared value of -3.81 on the test data (over 100 trials, each in which the portions were redefined and the model retrained), which reflects very poorly on the out of sample predictive performance of the model. Therefore, using quadratic features to transform the dataset may have increased the in-sample predictive performance of the regression model, but the out of sample performance (as suggested by r squared for the test data) remained very poor.

Code:

```
import numpy as np
import random
from sklearn.datasets import fetch_openml
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from q2_regression import impute
def q4b(ds, ds_t):
	impute(ds)
	model = LinearRegression(fit_intercept=True)
	model.fit(ds_t, ds.target) #using entire dataset
	r_sq = model.score(ds_t, ds.target)

	#split dataset into training, test portions
	indices = list(range(0, ds.data.shape[0]))
	r_sqs = np.empty((0,))
	for i in range(100):
		random.shuffle(indices)
		training_indices = indices[0:202]
```

```
                test_indices = indices[202:]


                training_data = ds_t[training_indices, :]
                test_data = ds_t[test_indices, :]
                training_targets = ds.target[training_indices]
                test_targets = ds.target[test_indices]


                training_model = LinearRegression().fit(training_data,
      training_targets)
                score = training_model.score(test_data, test_targets)
                r_sqs = np.append(r_sqs, score)


          return (r_sq, r_sqs)
      def main():
          ds = fetch_openml(name="cholesterol", version=1)
          results = q4b(ds, results[0])
          print("r-squared (for transformed dataset): " +
str(round(results[0], 2)))
          print("mean r-squared value on test portion (across 100 trials):
" + str(round(np.mean(results[1]), 2)))
```

Output:

```
r-squared (for transformed dataset): 0.37
mean r-squared value on test portion (across 100 trials): -3.81
```