q1

March 2, 2021

[1]:
```
from sklearn.datasets import load_diabetes
from numpy import cov as covM
from numpy import mean
from numpy import median
from numpy import std
from math import sqrt
```

[2]:
```
diabetes_data = load_diabetes()
# diabetes_data
```

[3]:
```
diabetes_features = diabetes_data.data
diabetes_target = diabetes_data.target
```

Part A: Computer CovM of non-target attributes of the data set

[4]:
```
cov_matrix = covM(diabetes_features, rowvar=False)
# cov_matrix
```

[5]:
```
cov_matrix.shape
```

[5]:
```
(10, 10)
```

The covariance matrix has dimensions 10x10.

Part B: Compute he correlation of the age and bp attributes (directly from the elements of the covariance matrix

$$Corr(Age, BP) = \frac{Cov(Age, BP)}{\sqrt{Cov(Age, Age) \cdot Cov(BP, BP)}}$$

We can directly use the covariance matrix to get these values.

[6]:
```
cov_age_bp = cov_matrix[0, 3]
cov_age_age = cov_matrix[0, 0]
cov_bp_bp = cov_matrix[3, 3]
```

[7]:
```
corr_age_bp = cov_age_bp / sqrt(cov_age_age * cov_bp_bp)
corr_age_bp
```

[7]: 0.33542671054424283

So, the correlation of age and blood pressure is approximately .3354. We found this by using the formula for correlation, which is essentially a standardized version of the covariance. Because of this, we can index the appropriate positions of the matrix to get the corresponding covariances and compute the correlation using these values.

Part C: Evaluate previous results

Based on the previous result, I would expect that older patients in the dataset have higher blood pressure. Since blood pressure and age are positively correlated (in the dataset), this means that as age increases, blood pressure increases as well (this is how you interperet correlation). We could think of the data as being globally "related" by a line with a positive slope of about .33, suggesting the trend just described.

Part D: Check whether the data is consistent with what we found above

[8]:
```
# compute median blood pressure among patients whose age is larger than the␣
↪median
mean_age = mean(diabetes_features[:, 0])
mean_age
```

[8]: -3.6396225400041895e-16

[9]:
```
older_patients = diabetes_features[diabetes_features[:, 0] > mean_age]
younger_patients = diabetes_features[diabetes_features[:, 0] < mean_age]
```

[10]:
```
median_older_patients = median(older_patients[:, 3])
median_older_patients
```

[10]: 0.0115437429137471

[11]:
```
median_younger_patients = median(younger_patients[:, 3])
median_younger_patients
```

[11]: -0.0228849640236156

The median blood pressure for younger patients is lower than that of older patients. This is evidence in favor of our findings from above. We compute the difference between medians as a number of standard deviations of the blood pressure attribute below:

[12]:
```
std_bp = std(diabetes_features[:, 3])
median_difference = abs(median_older_patients - median_younger_patients)
median_diff_in_std = median_difference / std_bp
median_diff_in_std
```

[12]: 0.7238221126281117

The difference between the median blood pressures, in terms of standard deviations, is .7238.

q2

March 2, 2021

[1]:
```
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_diabetes
from random import shuffle
from numpy import rint
```

Part A: Fit a Linear Regression model to the diabetes dataset

[2]:
```
diabetes_data = load_diabetes()
diabetes_features = diabetes_data.data
diabetes_targets = diabetes_data.target
```

[3]:
```
regression_obj = LinearRegression().fit(diabetes_features, diabetes_targets)
```

[4]:
```
predictions = regression_obj.predict(diabetes_features)
```

[5]:
```
r_squared = regression_obj.score(diabetes_features, diabetes_targets)
r_squared
```

[5]:
```
0.5177494254132934
```

[6]:
```
num_exact_predictions = sum(diabetes_targets[predictions == diabetes_targets])
proportion_exact = num_exact_predictions / len(predictions)
proportion_exact
```

[6]:
```
0.0
```

0 of our predictions were exactly correct, whereas the r^2 value was .51. This implies that while the regression line does provide exact predictions, the model itself is middle-of-the-pack in terms of fitting the sample data (which we hope also reflects performance on the population). This is expected, because we should not expect our regression line to perfectly fit many of the points exactly, if any. The point of the regression model is to best fit the linear relationship between the data, so to expect one line to go through many of the data points while also modelling the rest of the data is unrealistic. Thus, we cannot say that the model is ineffective because it does not exactly predict any of the target values perfectly. It may get very close to exact points, but by the nature of how the model is fit, we cannot expect it to perfectly predict every point.

Part B: Randomly split row indices of instances of the data set

[7]:
```
num_data_samples = len(diabetes_features)
num_data_samples
```

[7]:
```
442
```

[8]:
```
# get a list of indices and randomly shuffle them to generate test/training sets
data_indices = list(range(0, num_data_samples))
shuffle(data_indices)
```

[9]:
```
training_indices = data_indices[0:300]
testing_indices = data_indices[300:]
```

[10]:
```
training_set = diabetes_features[training_indices]
testing_set = diabetes_features[testing_indices]
```

[11]:
```
regression = LinearRegression().fit(training_set,␣
 ↪diabetes_targets[training_indices])
```

[12]:
```
r_squared_train = regression.score(training_set,␣
 ↪diabetes_targets[training_indices])
r_squared_train
```

[12]:
```
0.5258839489557672
```

[13]:
```
r_squared_test = regression.score(testing_set,␣
 ↪diabetes_targets[testing_indices])
r_squared_test
```

[13]:
```
0.47848698142628354
```

The r^2 value for the training set is .52, as opposed to the smaller r^2 value of .477 for the testing set. The values are not the same, but the small difference suggests that the model forms slightly worse on the population data (testing set) than on the training data. The training set, then, represents a better fit, which should not come as surprising, since the model is fit based off of the training set. Since we fit the model with data from the training set, we can expect that the goodness of fit will be at least a little higher for the training set. Because we randomly selected indices, however, we are not seeing a massive difference in goodness of fit between training and testing set, since the population data is more or less accurately reflected in the sample data due to the random nature of how we selected indices.

q3

March 2, 2021

[1]:
```
from numpy.linalg import pinv as pseudo_inverse
from numpy import mean
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression
```

Q3: Compute the Linear Regression Fit to the Diabetes Set using Matrix Computations by Hand

Primary Steps:

1. Mean-Shift X, y such that each column of X has mean of 0 and y has mean of 0
2. Assume $c_0 = 0$ and solve X.dot(c) $\approx$ y
3. Remember the mean information in that $c_0 = mean(y)$

[2]:
```
diabetes_data = load_diabetes()
diabetes_features = diabetes_data.data
diabetes_targets = diabetes_data.target
```

We would stary by meanshifting the data such that each column of our data set has a mean of 0 and our column of target values has a mean of 0. However, the diabetes data is already standardized.

The column vector of c values that minimize the mean squared error for our data is given by the following equation:

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

It also suffices to use the More-Penrose Pseudo-Inverse (in case $(X^T * X)^{-1}$ does not exist) in order to minimize the MSE:

$$C = (X^T * X)^{\dagger} * X * y$$

We use the variant with the pseduo inverse to compute C below.

[3]:
```
diabetes_features_transposed = diabetes_features.transpose()
pseudo_inverse = pseudo_inverse(diabetes_features_transposed.
↪dot(diabetes_features))
```

[4]:
```
C = pseudo_inverse.dot(diabetes_features_transposed).dot(diabetes_targets)
C
```

[4]:
```
array([ -10.01219782, -239.81908937,  519.83978679,  324.39042769,
        -792.18416163,  476.74583782,  101.04457032,  177.06417623,
         751.27932109,   67.62538639])
```

[5]:
```
# remember that the bias coefficient is equal to mean(y)
c_0 = mean(diabetes_targets)
```

[6]:
```
# print out the resulting coefficients
print("Reporting manually-obtained regression coefficients: \n")
print("c_0: {}".format(c_0))

for i, c in enumerate(C):

    print("c_{}: {}".format(i+1, c))
```

```
Reporting manually-obtained regression coefficients:

c_0: 152.13348416289594
c_1: -10.012197817492385
c_2: -239.81908936567024
c_3: 519.8397867900575
c_4: 324.3904276894312
c_5: -792.184161627983
c_6: 476.74583782339937
c_7: 101.04457032117944
c_8: 177.0641762322956
c_9: 751.279321087222
c_10: 67.62538639102785
```

[7]:
```
# comparing the values computed above with the coefficient values found via the␣
 ↪LinearRegression class
regression_model = LinearRegression().fit(diabetes_features, diabetes_targets)
regression_coefficients = regression_model.coef_
bias_coefficient = regression_model.intercept_

print("Reporting regression coefficients (obtained from sklearn): \n")

print("c_0: {}".format(bias_coefficient))

for i, c in enumerate(regression_coefficients):

      print("c_{}: {}".format(i+1, c))
```

```
Reporting regression coefficients (obtained from sklearn):
```

```
c_0: 152.1334841628965
c_1: -10.012197817470962
c_2: -239.81908936565566
c_3: 519.8397867901349
c_4: 324.39042768937657
c_5: -792.1841616283053
c_6: 476.7458378236622
c_7: 101.04457032134493
c_8: 177.06417623225025
c_9: 751.2793210873947
c_10: 67.62538639104369
```

As shown in the results above, the coefficients calculated in the different methods come out to be the same value. This makes sense because under the hood, scikit learn is performing these calculations (maybe in a slightly different way) to get the same values. As a library, scikitlearn provides a high level implementation of the process we computed by hand.