**VIVIAN JOSEPH**

(20BCE0777)

Machine Learning

**Lab DA-5**

Dimensionality Reduction and Unbalanced Data Processing

**PCA:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import sklearn

from sklearn.metrics import r2\_score

from sklearn.model\_selection import train\_test\_split

from sklearn.decomposition import PCA

import seaborn as sn

from sklearn.preprocessing import StandardScaler

**Load the breast cancer dataset:**

from sklearn.datasets import load\_breast\_cancer

cancer = load\_breast\_cancer()

X = cancer.data

X=pd.DataFrame(X)

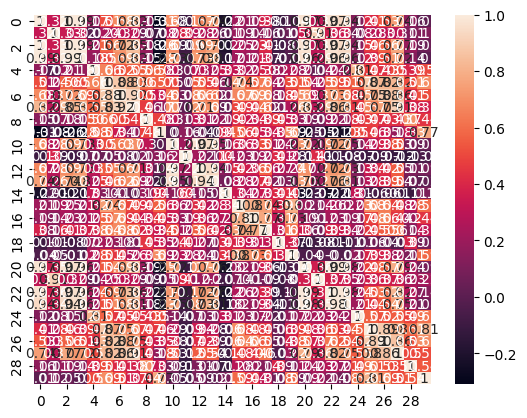
y = cancer.target

**Visualizing the correlations between features**

correlation\_matrix = X.corr().round(2)

# annot = True to print the values inside the square

sn.heatmap(data=correlation\_matrix, annot=True)



**Total** **30 input features (Clearly, too many features)**

**Training and testing without PCA:**

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split( X, y, test\_size=0.30, random\_state=30)

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

lm = LinearRegression()

lm.fit(X\_train, Y\_train)

# model evaluation for training set

y\_train\_predict = lm.predict(X\_train)

rmse = (np.sqrt(mean\_squared\_error(Y\_train, y\_train\_predict)))

r2 = r2\_score(Y\_train, y\_train\_predict)

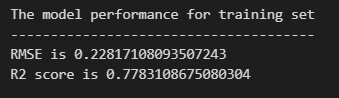
print("The model performance for training set")

print("--------------------------------------")

print('RMSE is {}'.format(rmse))

print('R2 score is {}'.format(r2))

print("\n")



# model evaluation for testing set

Y\_pred = lm.predict(X\_test)

rmse = (np.sqrt(mean\_squared\_error(Y\_test, Y\_pred)))

r2 = r2\_score(Y\_test, Y\_pred)

print("The model performance for testing set")

print("--------------------------------------")

print('RMSE is {}'.format(rmse))

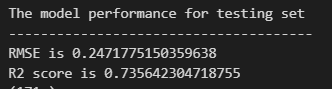
print('R2 score is {}'.format(r2))

# model evaluation for testing set

print(Y\_pred.shape)

Y\_pred

pd.DataFrame(Y\_pred,Y\_test)





**Test R2 Score with 30 features**

**Applying PCA:**

scaler = StandardScaler()

X\_std = scaler.fit\_transform(X)

# Initialize PCA model

pca = PCA(n\_components=4)



# Fit and transform data

X\_pca = pca.fit\_transform(X\_std)

# Plot the transformed data

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=y+1)

plt.xlabel('PC1')

plt.ylabel('PC2')

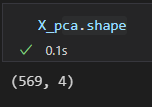
plt.title('Iris dataset after PCA')

plt.show()

# Print explained variance ratio

print("Explained variance ratio:", pca.explained\_variance\_ratio\_)

**final shape:**

****

**Training and testing after PCA:**

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split( X\_pca, y, test\_size=0.30, random\_state=30)

lm = LinearRegression()

lm.fit(X\_train, Y\_train)

# model evaluation for training set

y\_train\_predict = lm.predict(X\_train)

rmse = (np.sqrt(mean\_squared\_error(Y\_train, y\_train\_predict)))

r2 = r2\_score(Y\_train, y\_train\_predict)

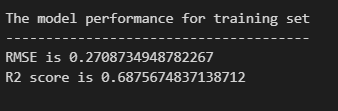
print("The model performance for training set")

print("--------------------------------------")

print('RMSE is {}'.format(rmse))

print('R2 score is {}'.format(r2))

print("\n")

****

Y\_pred = lm.predict(X\_test)

rmse = (np.sqrt(mean\_squared\_error(Y\_test, Y\_pred)))

r2 = r2\_score(Y\_test, Y\_pred)

print("The model performance for testing set")

print("--------------------------------------")

print('RMSE is {}'.format(rmse))

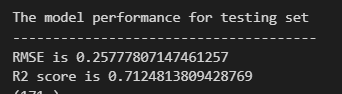
print('R2 score is {}'.format(r2))

# model evaluation for testing set

print(Y\_pred.shape)

Y\_pred

pd.DataFrame(Y\_pred,Y\_test)

****



**Test R2 Score with only 4 features**

|  |  |  |
| --- | --- | --- |
| **Number of input Features** | **R2 Score** | **RMSE** |
| **30** input features | 0.735 | 0.247 |
| only **6** input features | 0.712 | 0.258 |

**Clearly, the r2 score is affected barely even after drastically reducing the number of features from 30 to 6.**

**Under-Sampling and Over-Sampling:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import sklearn

from sklearn.metrics import r2\_score

import seaborn as sn

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

**Creating an imbalanced dataset:**

from sklearn.datasets import load\_breast\_cancer

import numpy as np

breast\_cancer = load\_breast\_cancer()

X = breast\_cancer.data

y = breast\_cancer.target

# Generate an imbalanced version of the dataset with a 90/10 split

np.random.seed(42)

mask = np.random.rand(len(y)) < 0.9 # 90% of samples in one class

y[mask] = 0 # Assign the first class to the majority of samples

y[~mask] = 1 # Assign the second class to the minority of samples

print("Original class distribution:", np.bincount(breast\_cancer.target))

print("Imbalanced class distribution:", np.bincount(y))

target data is highly imbalanced

****

(504 samples with label 0, 65 samples with label 1)

**Over sample the data:**

# Get the indices of the majority and minority classes

majority\_class = np.where(y == 0)[0]

minority\_class = np.where(y == 1)[0]

# Randomly select a subset of samples from the minority class

random\_indices = np.random.choice(minority\_class, size=len(majority\_class), replace=True)

# Combine the majority class samples with the randomly duplicated minority class samples

oversampled\_indices = np.concatenate([majority\_class, random\_indices])

X\_oversampled = X[oversampled\_indices]

y\_oversampled = y[oversampled\_indices]

# Check the class distribution

print(np.unique(y\_oversampled, return\_counts=True))

Oversampled Data:

****

(504 samples with label 0, 504 samples with label 1)

**Under sample the data:**

num\_class0 = np.sum(y == 0)

num\_class1 = np.sum(y == 1)

# Randomly select a subset of samples from the minority class

random\_indices = np.random.choice(minority\_class, size=len(majority\_class), replace=True)

# Combine the majority class samples with the randomly duplicated minority class samples

oversampled\_indices = np.concatenate([majority\_class, random\_indices])

X\_oversampled = X[oversampled\_indices]

y\_oversampled = y[oversampled\_indices]

# Check the class distribution

print(np.unique(y\_oversampled, return\_counts=True))

****

(65 samples with label 0, 65 samples with label 1)

**Training and testing for all strategies:**

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split( X, y, test\_size=0.30, random\_state=30)

lm = LinearRegression()

lm.fit(X\_train, Y\_train)

# model evaluation for training set

y\_train\_predict = lm.predict(X\_train)

rmse = (np.sqrt(mean\_squared\_error(Y\_train, y\_train\_predict)))

r2 = r2\_score(Y\_train, y\_train\_predict)

print("The model performance for training set")

print("--------------------------------------")

print('RMSE is {}'.format(rmse))

print('R2 score is {}'.format(r2))

print("\n")

Y\_pred = lm.predict(X\_test)

rmse = (np.sqrt(mean\_squared\_error(Y\_test, Y\_pred)))

r2 = r2\_score(Y\_test, Y\_pred)

print("The model performance for testing set")

print("--------------------------------------")

print('RMSE is {}'.format(rmse))

print('R2 score is {}'.format(r2))

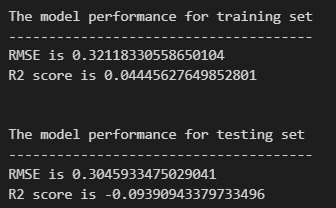
# model evaluation for testing set

print(Y\_pred.shape)

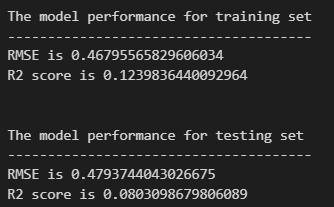
Y\_pred

pd.DataFrame(Y\_pred,Y\_test)

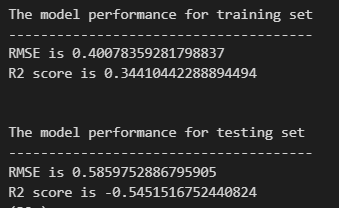
**For Unbalanced Data:**

****

**For Over Sampled Data:**

****

**For Under Sampled Data:**

****

**Clearly, Over Sampling helps improve on the results but under sampling is not very useful in this case**