Problem 1

```
import numpy as np
from sklearn.preprocessing import StandardScaler as StandardScaler
from sklearn.decomposition import PCA as PCA
import pandas as pd

X_array = [0, 1, 2, 2, 3, 3, 4]
Y_array = [1, 1, 1, 3, 2, 3, 5]
raw_data = np.column_stack((X_array, Y_array))
```

(a) PCA after standardizing data

```
scaler = StandardScaler()
_ = scaler.fit(raw_data)
standardized_data = scaler.transform(raw_data)
```

```
standardized_pca = PCA(n_components=2)
_ = standardized_pca.fit(standardized_data)
print("First Two Components of Standardized PCA")
print(standardized_pca.components_)
```

```
First Two Components of Standardized PCA
[[ 0.70710678  0.70710678]
 [ 0.70710678 -0.70710678]]
```

```
standardized_pca_transformed_data = standardized_pca.fit_transform(standardized_data)
standardized_pca_dataframe = pd.DataFrame(data = standardized_pca_transformed_data
             , columns = ['PC 1', 'PC 2'])
standardized_pca_dataframe["Standardized X"] = standardized_data[:, 0]
standardized_pca_dataframe["Standardized Y"] = standardized_data[:, 1]
print(standardized_pca_dataframe)
```

```
       PC 1      PC 2  Standardized X  Standardized Y
0 -1.873053 -0.560268       -1.720618       -0.928279
1 -1.305278  0.007507       -0.917663       -0.928279
2 -0.737503  0.575282       -0.114708       -0.928279
3  0.283552 -0.445773       -0.114708        0.515711
4  0.340799  0.632529        0.688247       -0.206284
5  0.851327  0.122002        0.688247        0.515711
6  2.440157 -0.331278        1.491202        1.959700
```

```
print(np.matmul(standardized_data, standardized_pca.components_).round(2))
```

```
[[-1.87 -0.56]
 [-1.31  0.01]
 [-0.74  0.58]
 [ 0.28 -0.45]
 [ 0.34  0.63]
 [ 0.85  0.12]
 [ 2.44 -0.33]]
```

(b) PCA without standardizing data

```
raw_data_pca = PCA(n_components=2)
_ = raw_data_pca.fit(raw_data)
print("First Two Components of Raw PCA")
print(raw_data_pca.components_)
```

```
First Two Components of Raw PCA
[[ 0.65908697  0.75206673]
 [ 0.75206673 -0.65908697]]
```

```
raw_pca_transformed_data = raw_data_pca.fit_transform(raw_data)
raw_pca_dataframe = pd.DataFrame(data = raw_pca_transformed_data
             , columns = ['PC 1', 'PC 2'])
raw_pca_dataframe["Raw X"] = raw_data[:, 0]
raw_pca_dataframe["Raw Y"] = raw_data[:, 1]
print(raw_pca_dataframe)
```

```
       PC 1      PC 2  Raw X  Raw Y
0 -2.379272 -0.764174      0      1
1 -1.720185 -0.012107      1      1
2 -1.061098  0.739959      2      1
3  0.443035 -0.578215      2      3
4  0.350055  0.832939      3      2
5  1.102122  0.173852      3      3
6  3.265343 -0.392255      4      5
```

```
print(np.matmul(raw_data, raw_data_pca.components_).round(2))
```

```
[[ 0.75 -0.66]
 [ 1.41  0.09]
 [ 2.07  0.85]
 [ 3.57 -0.47]
 [ 3.48  0.94]
 [ 4.23  0.28]
 [ 6.4  -0.29]]
```

Since manually multiplying the Principal Components with the Raw Dataset does not yield the same result as the sklearn PCA model, we know that PCA is not scale invariant.

Therefore, it is best to standardize the data before the procedure.

Problem 2

```
import numpy as np

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

from matplotlib import pyplot as plt
```

```
raw_poly_data = np.loadtxt("poly_data.csv", delimiter=" ")
print(str("Shape of Data: {}").format(raw_poly_data.shape))
```

```
Shape of Data: (200, 2)
```

```
number_of_splits = 5
polynomial_degree_array = np.arange(1, 41, 1)
mse_error_array = []
```

```
for poly_degree in polynomial_degree_array:
    polynomial_features = PolynomialFeatures(degree=poly_degree)
    transformed_poly_data = polynomial_features.fit_transform(raw_poly_data[:, 0].reshape(-1, 1))
    linear_regression_obj = LinearRegression()

    _ = linear_regression_obj.fit(transformed_poly_data, raw_poly_data[:, 1])

    if (poly_degree % 5 == 0):
        print(str("Polynomial Degree: {:2d}; Training Accuracy: {:3.1f}%").format(poly_degree,
                                                                                   linear_regression_c
                                                                                       transformed_pol
                                                                                       raw_poly_data[:

    cross_validation_obj = KFold(n_splits=number_of_splits)
    computed_cross_val_scores = -1 * cross_val_score(linear_regression_obj, transformed_poly_data, ra
                                                     scoring='neg_mean_squared_error', cv=cross_valic

    mse_error_array.append(computed_cross_val_scores)
```

```
Polynomial Degree:  5; Training Accuracy: 94.7%
Polynomial Degree: 10; Training Accuracy: 94.8%
Polynomial Degree: 15; Training Accuracy: 94.9%
Polynomial Degree: 20; Training Accuracy: 95.0%
Polynomial Degree: 25; Training Accuracy: 95.1%
Polynomial Degree: 30; Training Accuracy: 76.3%
Polynomial Degree: 35; Training Accuracy: 60.2%
Polynomial Degree: 40; Training Accuracy: 54.9%
```

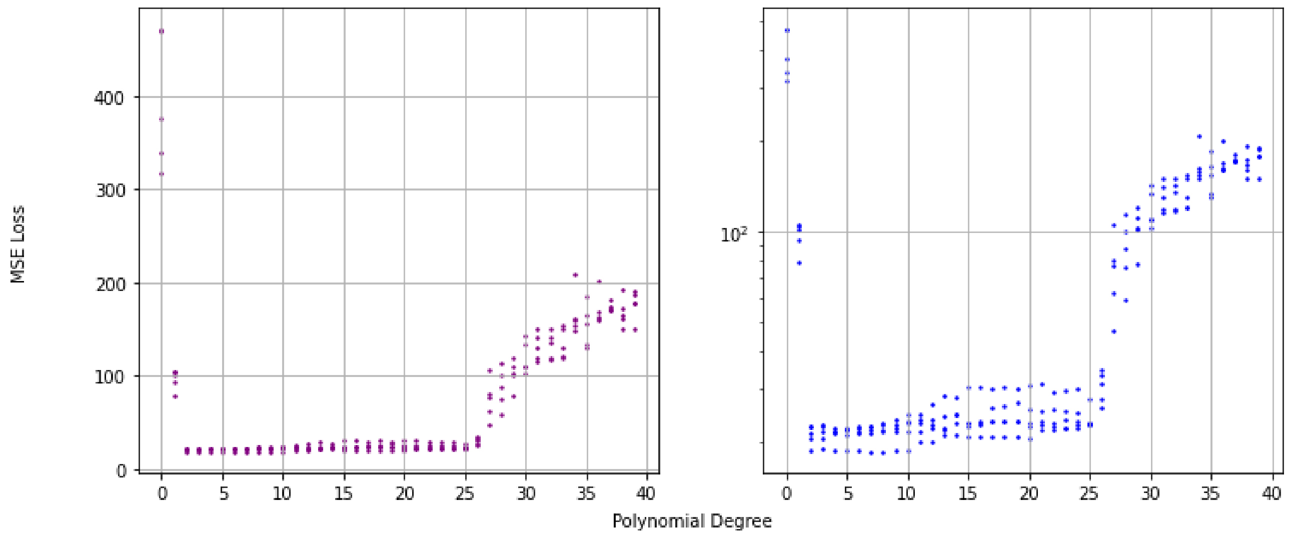
```
plt.rcParams['figure.figsize'] = (12, 5)
fig, axs = plt.subplots(1, 2, sharex="all")

mse_error_array = np.array(mse_error_array)
for poly_degree in range(polynomial_degree_array[-1]):
    for split in range(number_of_splits):
        axs[0].scatter(poly_degree, mse_error_array[poly_degree][split], c="Purple", s=2)
        axs[0].set_yscale('linear')

for poly_degree in range(polynomial_degree_array[-1]):
    for split in range(number_of_splits):
        axs[1].scatter(poly_degree, mse_error_array[poly_degree][split], c="Blue", s=2)
        axs[1].set_yscale('log')

axs[0].grid()
axs[1].grid()
fig.text(0.5, 0.04, 'Polynomial Degree', ha='center')
fig.text(0.04, 0.5, 'MSE Loss', va='center', rotation='vertical')

plt.show()
```



Which Polynomial Degree fits the data the best?

Based on median MSE error computed above for each polynomial degree.

```
median_mse_error_array = np.median(mse_error_array, axis=1)
min_mse = min(median_mse_error_array)
lowest_error_polynomial_degree = median_mse_error_array.tolist().index(min_mse) + 1
print(str("Best Fitting Polynomial Degree is: {}").format(lowest_error_polynomial_degree))
```

```
Best Fitting Polynomial Degree is: 3
```

Regression using Degree 3 Polynomial

```
polynomial_features = PolynomialFeatures(degree=3)
transformed_poly_data = polynomial_features.fit_transform(raw_poly_data[:, 0].reshape(-1, 1))
linear_regression_obj = LinearRegression()
_ = linear_regression_obj.fit(transformed_poly_data, raw_poly_data[:, 1])

print(str("Polynomial Degree: {:2d}; Training Accuracy: {:3.1f}%").format(3,
                                                                          linear_regression_obj.score
                                                                              transformed_poly_data,
                                                                              raw_poly_data[:, 1]) *
```

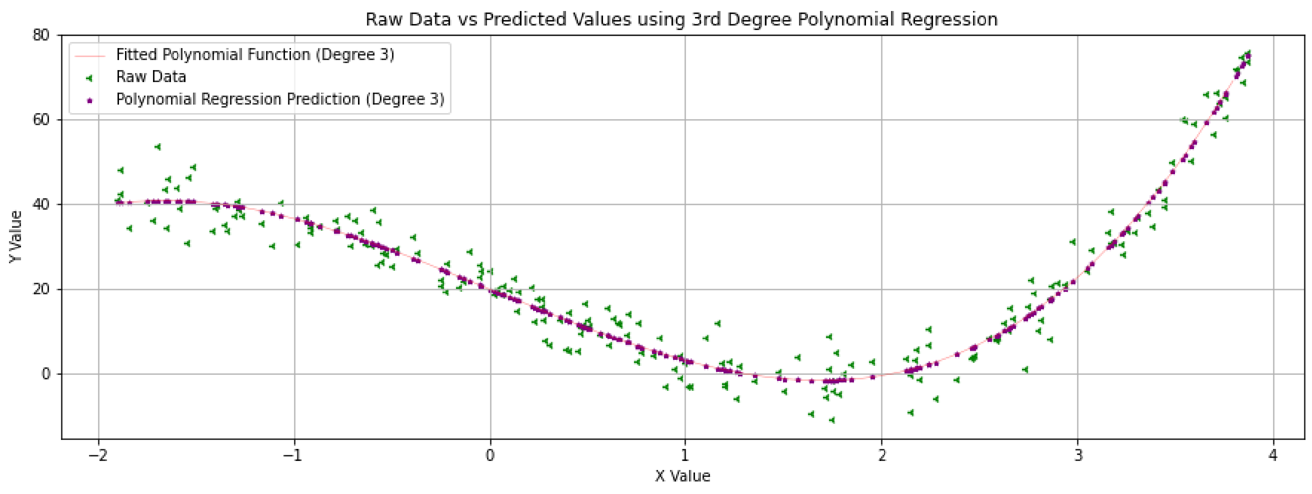
```
Polynomial Degree:  3; Training Accuracy: 94.7%
```

```
beta_array = linear_regression_obj.coef_
X_array = np.arange(raw_poly_data[:, 0].min(), raw_poly_data[:, 0].max(), 0.1)
evaluated_polynomial_function = linear_regression_obj.intercept_ + beta_array[0] + X_array * beta_arr
    1] + X_array ** 2 * beta_array[2] + X_array ** 3 * beta_array[3]

print(r'Fitted Polynomial Function: {:.1f} + (X * {:.1f}) + (X^2 * {:.1f}) + (X^3 * {:.1f})'.format(
    linear_regression_obj.intercept_, beta_array[1], beta_array[2], beta_array[3]))
```

```
Fitted Polynomial Function: 19.8 + (X * -19.0) + (X^2 * -0.1) + (X^3 * 2.2)
```

```
plt.rcParams['figure.figsize'] = (15, 5)
plt.scatter(raw_poly_data[:, 0], raw_poly_data[:, 1], c="Green", s=26, marker="3", label="Raw Data")
plt.scatter(raw_poly_data[:, 0], linear_regression_obj.predict(transformed_poly_data), color="Purple"
            label="Polynomial Regression Prediction (Degree 3)")
plt.plot(X_array, evaluated_polynomial_function, color="Red", linewidth=0.5, alpha=0.5,
         label="Fitted Polynomial Function (Degree 3)")
plt.title("Raw Data vs Predicted Values using 3rd Degree Polynomial Regression")
plt.ylabel("Y Value")
plt.xlabel("X Value")
plt.legend()
plt.grid()
plt.show()
```



Problem 3

```python
import numpy as np
from matplotlib import pyplot as plt
import scipy.stats as stats
```

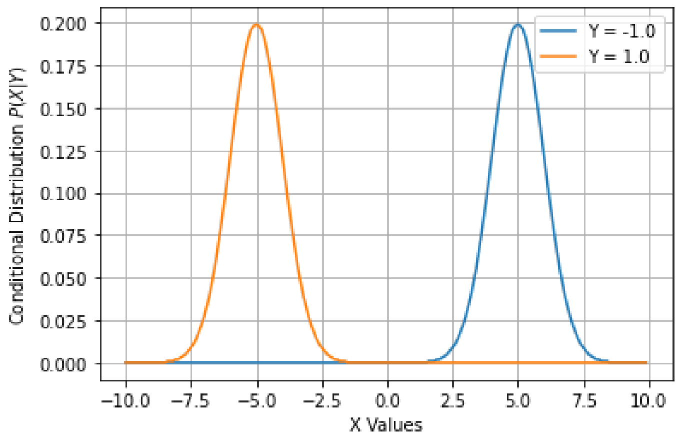
```python
def evaluate_conditional_distribution(x, y):
    return (1.0 / (2 * np.sqrt(2 * np.pi)) * np.exp(-(x + 5 * y) ** 2 / 2.0))


vectorized_conditional = np.vectorize(evaluate_conditional_distribution)
```

```python
X_array = np.arange(-10.0, 10.0, 0.1)
Y_array = [-1.0, 1.0]
```

```python
for y in Y_array:
    evaluated_distribution = vectorized_conditional(X_array, y).round(3)
    plt.plot(X_array, evaluated_distribution, label="Y = " + str(y))

plt.grid()
plt.legend()
plt.xlabel("X Values")
plt.ylabel(r"Conditional Distribution $P(X | Y)$")
plt.show()
```



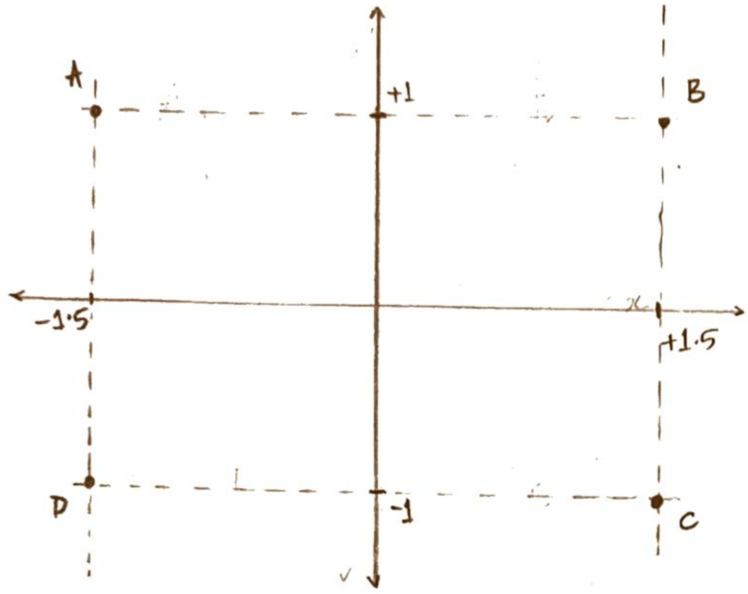
```python
print("Classification Error of Bayes Optimal Classifier described in (c): {:.3e}".format(
    1 - stats.norm.cdf(5, loc=0, scale=1)))
```

```
Classification Error of Bayes Optimal Classifier described in (c): 2.867e-07
```

Problem 4

Given :
$$\min_{c_1, \dots c_k} \sum_{i=1}^{n} \|x_i - c(x_i)\|_2^2$$

To Prove: k-means algorithm does not always find the optimal solution to the above objective



Step 0: centroid arrangements

① A & D ② B & C

Step 1: compute objective

$= 2 \cdot \|1-0\|_2^2 = 2$

Step 2: Optimal objective $= 2+2=4$ = OPT

Now consider a bad initialization:

Step 0: centroid arrangements

① A & B ② C & D

Step 1: compute objective

$= 2 \cdot \|1.5\|_2^2 = 4.5$

In this case, the next step of the algorithm does not change any point assignments because A & B are closer to $(0,1)$ than $(0,-1)$

Then, when we recalculate the cluster centroids, that doesnt change either because $(0,1)$ is the center of A & B, same for C & D. Hence, the algorithm converges

Now, we wish to generalize this example to

p dimensions

n data points

k clusters

4 types of points :

$(-t-0.5, t)$ & 0 for all other dimensions

$(t+0.5, t)$

$(-t-0.5, -t)$

$(t+0.5, -t)$

$$\text{Points} = \left\{ \begin{matrix} (-t-0.5, t) \\ (t+0.5, t) \\ (-t-0.5, -t) \\ (t+0.5, -t) \end{matrix} : t \in [1, T] \right\}$$

This could work for any $t \geqslant 0$; $4T$ points ; $2T$ clusters

The optimal objective function cost =

All the optimal clusters are the midpoints of the pair of points that have the same y-value & are symmetrically reflected across the y-axis; i.e.

- $(-t-0.5, t) <> (-t-0.5, -t)$; centroid location is $(-t, -0.5, 0)$
- $(t+0.5, -t) <> (t+0.5, t)$; centroid location is $(t+0.5, 0)$

Thus, optimal value of objective would be :

$$\sum_{t=1}^{T} 2 \cdot t^2 = 2 \sum_{t=1}^{T} t^2$$

In the non-optimal case, centroids would be located:

- $(-t-0.5, t) <> (t+0.5, t)$; center is $(0, t)$
- $(-t-0.5, -t) <> (t+0.5, -t)$; center is $(0, -t)$

The value of the objective function in this case

$$\sum_{t=1}^{T} 2 \cdot (t+0.5)^2 = 2\sum_{t=1}^{T} t^2 + 2\sum_{t=1}^{T} (0.25+t)$$

$$= 2\sum_{t=1}^{T} t^2 + \sum_{t=1}^{T} (t+0.5)$$

$$> 2\sum_{t=1}^{T} t^2 \quad \because \quad t > 0 \ \& \ T \geq 1$$

Thus, since we have proved suboptimal objective is greater in this setup, we have proved k-means is not ALWAYS optimal.

Problem 5

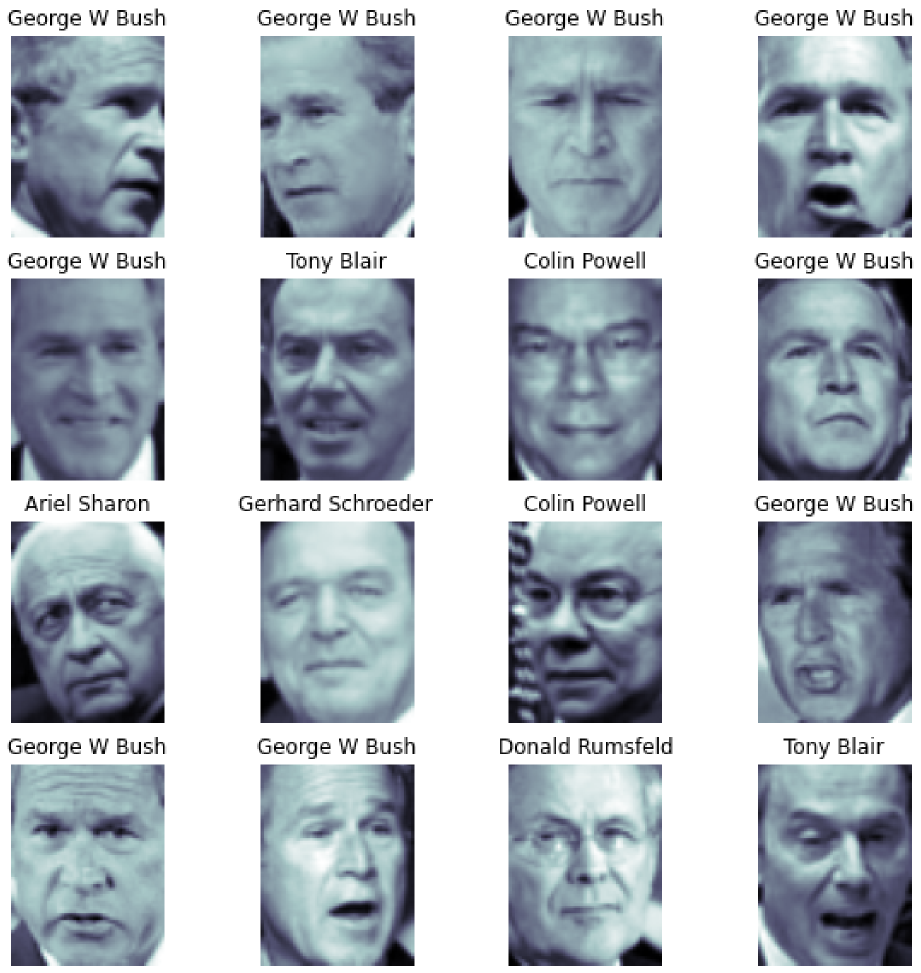
```python
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA
import numpy as np
from matplotlib import pyplot as plt
```

```python
faces_dataset = fetch_lfw_people(min_faces_per_person = 60)
print("Shape of Dataset: {}".format(faces_dataset.data.shape))

index = np.random.randint(0, faces_dataset.data.shape[0], size=16)
plt.rcParams['figure.figsize']=(10, 10)

i = 0
fig, axs = plt.subplots(4, 4)
for r in range(4):
    for c in range(4):
        axs[r, c].imshow(faces_dataset.images[index[i]], cmap=plt.cm.bone)
        axs[r, c].set_title(str(faces_dataset.target_names[faces_dataset.target[index[i]]]))
        axs[r, c].axis('off')
        i+=1
```

```
Shape of Dataset: (1348, 2914)
```



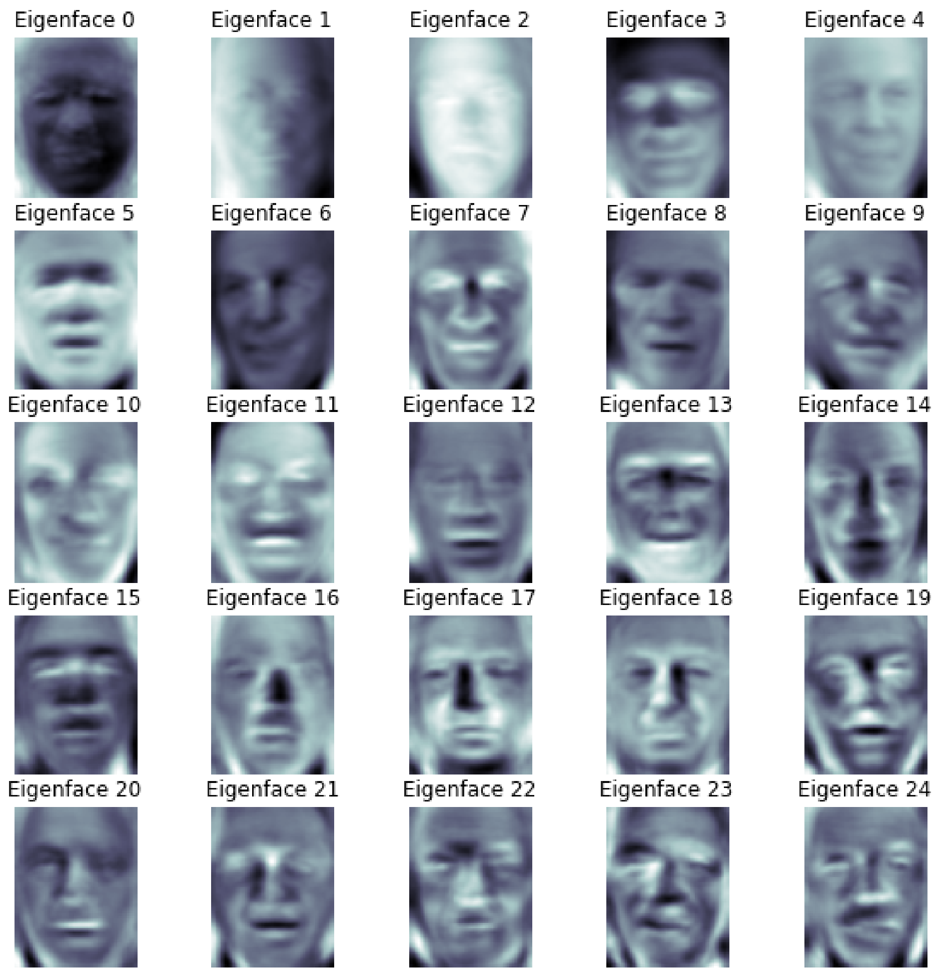
PCA on the dataset to find the first 150 components, using randomized PCA from `sklearn`

```
faces_pca = PCA(n_components=150, whiten=True, svd_solver='randomized')
_ = faces_pca.fit(faces_dataset.data)
```

Eigenfaces associated with first 25 principal components

```
pca_eigenfaces = faces_pca.components_
plt.rcParams['figure.figsize']=(10, 10)

i = 0
fig, axs = plt.subplots(5, 5)
for r in range(5):
    for c in range(5):
        axs[r, c].imshow(pca_eigenfaces[i].reshape(faces_dataset.images.shape[1:]), cmap=plt.cm.bone)
        axs[r, c].set_title(str("Eigenface {}").format(i))
        axs[r, c].axis('off')
        i+=1
```



Reconstructing a few random faces using the first 150 Principal Components

```python
plt.rcParams['figure.figsize']=(5.5, 10)
index = np.random.randint(0, faces_dataset.data.shape[0], size=5)

i = 0
fig, axs = plt.subplots(5, 2)
for r in range(5):
    c = 0
    axs[r, c].imshow(faces_dataset.images[index[i]], cmap=plt.cm.bone)
    axs[r, c].set_title(str(faces_dataset.target_names[faces_dataset.target[index[i]]]))
    axs[r, c].axis('off')

    pca_transformed_face = faces_pca.transform(faces_dataset.images[index[i]].reshape(1, -1))
    axs[r, c+1].imshow(faces_pca.inverse_transform(pca_transformed_face).reshape(faces_dataset.images
    axs[r, c+1].set_title(str("Recreated: " + faces_dataset.target_names[faces_dataset.target[index[i
    axs[r, c+1].axis('off')
    i+=1
```