[EE488] 2020 FALL Project1 Report

20180381 Heewon Yang

Part 1. Data Preparation

1-1. Download and decompress the MNIST dataset (4, 9 classes)
1-2. Download and define a function('cifar10_subset') for preparing the CIFAR10 dataset

Part 2. Logistic Regression

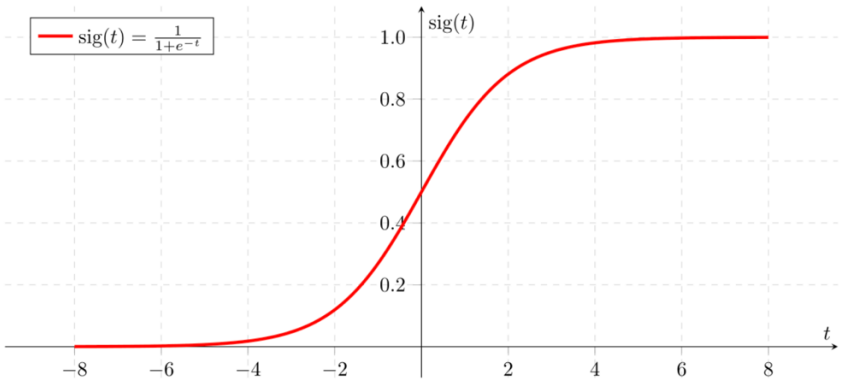
Logistic regression with a neural network mindset simply means that we will be doing a forward and backward propagation mode to code the algorithm as is usually the case with neural network algorithms. Logistic regression takes an input, passes it through a function called sigmoid function then returns an output of probability between 0 and 1. This sigmoid function is responsible for classifying the input.

2-1. Implement logistic regression using Numpy

For logistic regression, the forward propagation is used to calculate the loss function and the output, y, by predicting each data point in x, while the backward propagation is used to calculate the gradient descent. The new weights can be calculated by adding learning rate multiplied by gradient to the old weights. This algorithm can be used to classify images as opposed to the ML form of logistic regression and that is what makes it stand out.

1) Implement sigmoid function

$sigmoid(z) = \frac{1}{1+e^{-z}}$ $where\ z = w^T x$




```
def sigmoid(z):
    ######################## Blank ########################
    exp = np.exp(-z)
    sig = 1./(1.+exp)
    ######################################################
    return sig
```

2) Implement forward propagation

x is an input data (sample_size, feature_size), y is a target (sample_size, 1) and w (feature_size, 1) is an array of model weights. The prediction for each data point $h(w^T x) = \frac{1}{1+e^{-w^T x}}$, so I could find that prediction function h is same as sigmoid function. Also, loss is negative log-likelihood cost for logistic regression, $J(w) = \frac{1}{n}\sum_{i=1}^{n}[-y_i \log\{h(w^T x)\} - (1-y_i)\log\{1-h(w^T x)\}]$ (n is the sample_size). Before using np.log method, it is recommended to use as follows, np.log(x + eps) where eps is small positive number used to avoid taking of almost zero value.

Since X_train's size is (11791, 784) which is (sample_size, feature_size) from the MNIST dataset, to handle the dimension, I transpose x and multiply it with w.T where w's size is (784,) which is

(feature_size, 1). And I calculated predict by sigmoid function and loss by the above equation. Then, predict size is (sample_size, 1) and loss is a scalar value.

```
def forward(x, y, w, eps=1e-8):
    ######################## Blank ########################
    n = x.shape[0]
    linear = np.matmul(w.T, x.T)
    predict = sigmoid(np.array(linear))
    loss = -1/n * (np.matmul(y, np.log(predict+eps)) + np.matmul((1-y), np.log(1-predict+eps)))
    ########################################################
    return predict, loss
```

3) Implement backward propagation

For backward propagation, I calculated the gradient of the loss with respect to weights. Therefore, the result of backward propagation would have same size as w. To make grad_w same size as w, I used method matmul for the transpose of (predict – y) and x ((sample_size, feature_size) * (sample_size, 1) = (feature_size, 1)), then the size of w is (feature_size, 1).

$$\begin{aligned}\frac{\partial J(w)}{\partial w} &= -\frac{1}{n}\sum_{i=1}^{n}\left[y_i\left(1-h\left(w^T x_i\right)\right)-(1-y_i)\,h\left(w^T x_i\right)\right]x_i \\ &= \frac{1}{n}\sum_{i=1}^{n}\left[h\left(w^T x_i\right)-y_i\right]x_i \\ &= \frac{1}{n}(\hat{Y}-Y)X\end{aligned}$$

```
def backward(x, y, predict):
    ######################## Blank ########################
    n = x.shape[0]
    grad_w = 1./n * (np.matmul((predict - y).T, x))
    ########################################################
    return grad_w
```

4) Add bias unit

To implement the bias unit, insert ones column vector into dataset x and inset b into w, like below figure. Then after adding bias unit, the size of x_bar would be (sample_size, feature_size +1) and the size of w_bar would be (feature_size +1, 1).

$$\bar{x} := \begin{bmatrix} x_1^T & 1 \\ x_2^T & 1 \\ \vdots & \\ x_n^T & 1 \end{bmatrix} \in \mathbf{R}^{n\times(d+1)}, \bar{w} := \begin{bmatrix} w \\ b \end{bmatrix} \in \mathbf{R}^{d+1}$$

```
def bias_unit(x, w, b):
    ######################## Blank ########################
    n = x.shape[0]
    a = np.array([[1]*n])
    x_bar = np.concatenate((x, a.T), axis=1)
    w_bar = np.concatenate((w, np.array([[b]])), axis=0)
    ########################################################
    return x_bar, w_bar
```

5) Model initialization

To initialize the model w and b with arbitrary values, I set the size of w as (feature_size, 1) and b as zero. And filled w with gaussian distribution using np.random.normal method.

```
def initialize_params(X_train, verbose=False):

    ######################## Blank ########################
    w = np.random.normal(size=(X_train.shape[1] , 1))
    b = 0
    ########################################################

    X_train_bar, w_bar = bias_unit(X_train, w, b) # add bias unit
    if verbose:
        print('Before adding the bias unit')
        print('shape of X_train:, ', X_train.shape)
        print('w: ', w.__repr__())
        print('b: ', b.__repr__(), end='\n\n')
        print('After adding the bias unit')
        print('shape of X_train_bar:, ', X_train_bar.shape)
        print('w_bar: ', w_bar.__repr__())

    return X_train_bar, w_bar
```

6) Implement accuracy function for computing the accuracy

When predict $h(w^T x)$ and y are given, I calculated accuracy as below equation.

acc: $\frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{[\hat{y}_i = y_i]} \times 100(\%)$ where $\hat{y}_i = \mathbf{1}_{[h(w^T x_i) \geq 0.5]}$. And $\mathbf{1}_A$ is defined as:

$$\mathbf{1}_A := \begin{cases} 1 & \text{if } A \text{ is true} \\ 0 & \text{if } A \text{ is false} \end{cases}$$

```
def accuracy(predict, y):
    ######################## Blank ########################
    row = y.shape[0]
    pred_label = np.zeros((1, row))
    for i in range(row):
      if (predict[i] >= 0.5):
        pred_label[0][i] = 1
    acc = 0
    for i in range(row):
      if (y[i] == pred_label[0][i]):
        acc += 1
    result = (acc/row)*100
    ########################################################
    return result
```

2-2. Apply to the MNIST dataset

To implement training code with all the early defined methods, compute prediction and loss by forward propagation. And compute train accuracy, and get gradient using backward propagation. Then, update the weight using gradient and learning rate.

```
######################## Blank ########################
predict, loss = forward(X_train_bar, Y_train, w_bar)
train_acc = accuracy(predict, Y_train)
grad_w = backward(X_train_bar, Y_train, predict)
w_bar -= learning_rate * grad_w
########################################################
```

```
100%|■■■■■■■■■■| 5000/5000 [07:09<00:00, 11.64it/s, accuracy=97.30, loss=0.0774]
train accuracy: 97.32
test accuracy: 96.38
```

Logistic regression accuracy: 97.30%

Train accuracy: 97.32%

Test accuracy: 96.38%

2-3. Apply to the CIFAR10 dataset

To implement training code with all the early defined methods, compute prediction and loss by forward propagation. And compute train accuracy, and get gradient using backward propagation. Then, update the weight using gradient and learning rate.

```
####################### Blank #######################
predict, loss = forward(X_train_bar, Y_train, w_bar)
train_acc = accuracy(predict, Y_train)
grad_w = backward(X_train_bar, Y_train, predict)
w_bar -= learning_rate * grad_w
#####################################################
```

```
100%|■■■■■■■■■■| 5000/5000 [11:11<00:00,  7.44it/s, accuracy=74.10, loss=1.6516]
train accuracy: 74.22
test accuracy: 73.80
```

Logistic regression accuracy: 74.10%
Train accuracy: 74.22%
Test accuracy: 73.80%

2-4. Implement logistic regression with L2 regularization

Regularization is a technique to solve the problem of overfitting in a machine learning algorithm by penalizing the cost function. It does so by using an additional penalty term in the cost function.

By L2 regularization, cost function becomes $J(w) = \frac{1}{n}\sum_{i=1}^{n}[-y_i \log\{h(w^T x)\} - (1-y_i)\log\{1 - h(w^T x)\}] + \frac{\lambda}{2n}\|w\|_2^2$. λ is called the regularization parameter. It controls the trade-off between two goals: fitting the training data well vs keeping the params small to avoid overfitting. Hence, the gradient of J(w) becomes like below.

$$\frac{\partial J(w)}{\partial w} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i\left(1 - h\left(w^T x_i\right)\right) - (1-y_i)\,h\left(w^T x_i\right)\right]x_i + \frac{\lambda}{n}w$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left[h\left(w^T x_i\right) - y_i\right]x_i + \frac{\lambda}{n}w$$

$$= \frac{1}{n}(\hat{Y} - Y)X + \frac{\lambda}{n}w$$

The regularization term will heavily penalize large w_i. The effect will be less on smaller w_i's. As such, the growth of w is controlled. The h(x) we obtain with these controlled params w will be more generalizable. Note that λ is a hyper-parameter value. So, we need to find it using cross-validation. Larger value λ will make w_i shrink closer to 0, which might lead to underfitting and zero λ will have no regularization effect. Therefore, we have to take proper care of bias vs variance trade-off when choosing λ.

1) Implement forward propagation with regularization term

Prediction process is same as no-regularization-version. I modified forward function with λ value.

```
from numpy.linalg import norm

def forward_with_regularization(x, y, w, lambda_, eps=1e-8):
    ####################### Blank #######################
    linear = np.matmul(w.T, x.T)
    predict = sigmoid(linear).T
    n = x.shape[0]
    loss = ((-1/n)*(np.matmul(y, np.log(predict+eps)) + np.matmul((1-y), np.log(1-predict+eps))))
           + (lambda_/(2*n))*(norm(w)**2)
    #####################################################
    return predict, loss
```

2) Implement backward propagation with regularization term

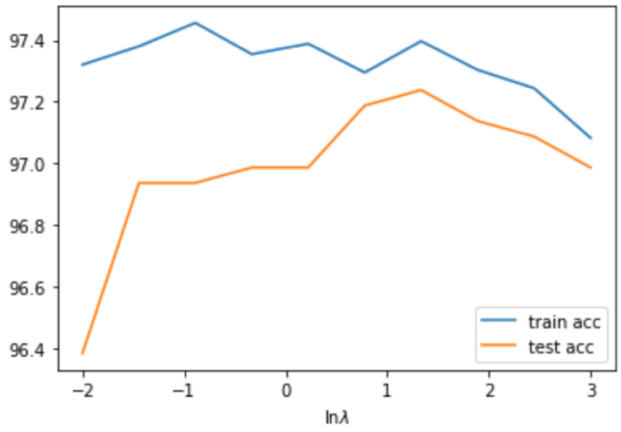
I modified backward function, adding (λ / n)*w to grad_w.

```
def backward_with_regularization(x, y, w, predict, lambda_):
    ######################## Blank ########################
    n = x.shape[0]
    grad_w = 1/n * ((np.matmul((predict.T - y), x)) + (lambda_* w))
    ###############################################################
    return grad_w
```

3) Training regularized logistic regression with different λ (Apply to the MNIST dataset)
To implement training code with all the early defined methods, compute prediction and loss by forward propagation for certain λ value. And compute train accuracy, and get gradient using backward propagation. Then, update the weight using gradient and learning rate.

```
######################## Blank ########################
predict, loss = forward_with_regularization(X_train_bar, Y_train, w_bar, lambda_)
train_acc = accuracy(predict, Y_train)
grad_w = backward_with_regularization(X_train_bar, Y_train, w_bar, predict, lambda_)
w_bar -= learning_rate * grad_w.T
###############################################################
```

Lambda value	e^{-2}	$e^{-1.45}$	$e^{-0.85}$	$e^{-0.3}$	$e^{0.25}$	$e^{0.8}$	$e^{1.35}$	$e^{1.9}$	$e^{2.45}$	e^{3}
Logistic acc	97.29	97.37	97.43	97.35	97.38	97.30	97.40	97.29	97.24	97.08
Train acc	97.32	97.38	97.46	97.35	97.39	97.29	97.40	97.30	97.24	97.08
Test acc	96.38	96.94	96.94	96.99	96.99	97.19	97.24	97.14	97.09	96.99



Part 3. Hyperparameter Optimization
Whenever a machine learning algorithm is implemented on a specific dataset, the performance is judged based on how well it generalizes, i.e how it reacts to new, never-before-seen data. In case the performance of the learning algorithm is not satisfactory or there is room for improvement, certain parameters in the algorithm need to be tuned. These parameters are known as 'hyperparameters' and the process of varying these hyperparameters to better the learning algorithm's performance is known as 'hyperparameter tuning'. These hyperparameters are not learnt directly through the training of algorithms. These values are fixed before the training of the data begins. They deal with parameters such as how quickly the model should be able to learn, how complicated the model is, and so on.
To optimize hyperparameters, I used GridSearchCV in sklearn.model_selection library. This approach is considered to be the traditional way of performing hyperparameter optimization. It searches the specific subset of the hyperparameters continuously until a condition is met or the end is reached. Scikit-learn module has the GridSearchCV that can be used to implement this approach.

3-1. Sample training data (set the train, test sample size to be 1500, 500 respectively for CIFAR10 dataset)

3-2. Optimize hyperparameters for logistic regression classifier

With the given grids to be searched, run GridSearchCV module with ‘sgd’ as the estimator, which is defined at the third line of this code. SGDClassifier is an estimator implementing regularized linear models with stochastic gradient descent (SGD) learning: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). I set the loss to ‘log’ for logistic regression. Since alpha is the constant that multiplies the regularization term, I set alpha values from given regularization parameter array. Also, eta0 is the initial learning rate for the ‘constant’, ‘invscaling’ or ‘adaptive’ schedules. So, I set the learning_rate to ‘constant’ and set eta0 values from given learning rate array.

```
###################### Blank ######################
param_grid = dict(alpha=[1e-3, 1e-4, 1e-5], learning_rate=["constant"], eta0=[1e-1, 1e-2, 1e-3])
from sklearn.model_selection import GridSearchCV
sgd = SGDClassifier(loss = "log")
grid = GridSearchCV(estimator=sgd, param_grid=param_grid, cv=5, n_jobs=-1, verbose=10)
grid_result = grid.fit(X_train, Y_train)
pred = grid_result.predict(X_test)
acc = accuracy(pred, Y_test)
eta0 = grid_result.best_params_['eta0']
alpha = grid_result.best_params_['alpha']
#########################################################
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:    1.3s
[Parallel(n_jobs=-1)]: Done   4 tasks      | elapsed:    2.8s
[Parallel(n_jobs=-1)]: Done   9 tasks      | elapsed:    8.5s
[Parallel(n_jobs=-1)]: Done  14 tasks      | elapsed:   15.9s
[Parallel(n_jobs=-1)]: Done  21 tasks      | elapsed:   24.6s
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed:   33.4s
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed:   45.6s
[Parallel(n_jobs=-1)]: Done  45 out of  45 | elapsed:   55.7s finished
eta0 is 0.01, alpha is 0.001, Acc = 25.933333333333337%
```

By GridSearchCV module, the best parameters for SGDClassifier for the MNIST dataset are eta0 as 1e-2 and alpha as 1e-3. With these parameters, I got 25.93% accuracy.

3-3. Optimize hyperparameters for SVM classifier

With the given grids to be searched, run GridSearchCV module with ‘svc’ as the estimator, which is defined at the third line of this code. SVC is support vector classifier which find a hyperplane in an N-dimensional space that distinctly classifies the data points by maximizing the margin distances. I set C and kernel with given grids, where C is regularization parameter which affects the strength of regularization and kernel specifies the kernel type to be used in the algorithm.

```
###################### Blank ######################
param_grid = {'C': [0.1, 1, 10, 100], 'kernel': ['linear', 'rbf', 'poly']}
from sklearn.model_selection import GridSearchCV
svc = SVC()
grid = GridSearchCV(estimator=svc, param_grid=param_grid, scoring='accuracy', cv=5, n_jobs=-1, verbose=10)
grid_result = grid.fit(X_train,Y_train)
pred = grid_result.predict(X_test)
acc = accuracy(pred, Y_test)
#print('Best Score: ', grid_result.best_score_)
#print('Best Params: ', grid_result.best_params_)
C = grid_result.best_params_['C']
kernel = grid_result.best_params_['kernel']
#########################################################
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:  1.9min
[Parallel(n_jobs=-1)]: Done   4 tasks      | elapsed:  4.0min
[Parallel(n_jobs=-1)]: Done   9 tasks      | elapsed:  7.9min
[Parallel(n_jobs=-1)]: Done  14 tasks      | elapsed: 10.2min
[Parallel(n_jobs=-1)]: Done  21 tasks      | elapsed: 15.7min
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed: 19.7min
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed: 26.5min
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed: 32.0min
[Parallel(n_jobs=-1)]: Done  57 tasks      | elapsed: 39.1min
[Parallel(n_jobs=-1)]: Done  60 out of  60 | elapsed: 40.6min finished
kernel is rbf, C is 10, Acc = 25.733333333333334%
```

By GridSearchCV module, the best parameters for SVC for the MNIST dataset are kernel as 'rbf' and C as 10. With these parameters, I got 25.7% accuracy.

Part 4. PCA

PCA, principal component analysis is the analysis of data to identify patterns and finding patterns to reduce the dimensions of the dataset with minimal loss of information. It is often useful to measure data in terms of its principal components rather than on a normal x-y axis.

Principal components are the underlying structure in the data. They are the directions where there is the most variance, the directions where the data is most spread out. PCA finds a new set of dimensions (or a set of basis of views) such that all the dimensions are orthogonal (and hence linearly independent) and ranked according to the variance of data along them. It means more important principle axis occurs first. (more important = more variance/more spread out data)

How does PCA work:

1. Calculate the covariance matrix X of data points.
2. Calculate eigen vectors and corresponding eigen values.
3. Sort the eigen vectors according to their eigen values in decreasing order.
4. Choose first k eigen vectors and that will be the new k dimensions.
5. Transform the original n dimensional data points into k dimensions.

4-1. Load Iris dataset

The iris dataset contains measurements for 150 iris flowers from three different species.

The three classes in the Iris dataset are Iris-setosa (n=50), Iris-versicolor (n=50), and Iris-virginica (n=50). And the four features of in Iris dataset are sepal length in cm, sepal width in cm, petal length in cm, and petal width in cm. So, the shape of iris_x is (150, 4) and the shape of iris_y is (150, 1).

4-2. Using Numpy to implement PCA (with Iris dataset)

1) Standardize the dataset

Whether to standardize the data prior to a PCA on the covariance matrix depends on the measurement scales of the original features. Since PCA yields a feature subspace that maximizes the variance along the axes, it makes sense to standardize the data, especially, if it was measured on different scales. Although, all features in the Iris dataset were measured in centimeters, I transformed the data onto unit scale (mean=0 and variance=1), which is a requirement for the optimal performance of many machine learning algorithms.

First, I computed mean and std across axis = 0 (feature-wise). Then, standardized the dataset using the equation:

$$z = \frac{(x - u)}{s}$$

(s is standard deviation of the training samples and u is the sample mean)

```
#step1: Standardization
####################### Blank ######################
mean = np.mean(iris_x, axis=0)
A = iris_x - (mean * np.ones((150, 1)))
std = np.std(iris_x, axis=0)
z = A / (std * np.ones((150, 1)))
####################################################
```

```
The mean of z:[-1.69031455e-15 -1.84297022e-15 -1.69864123e-15 -1.40924309e-15]
```

2) Find the covariance matrix

The classic approach to PCA is to perform the eigen-decomposition on the covariance matrix Σ, which is a dim×dim (dim is feature_size) matrix where each element represents the covariance between two features. The covariance between two features is calculated as follows:

$$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^{n} \left(x_{ij} - \bar{x}_j\right) \left(x_{ik} - \bar{x}_k\right)$$

We can summarize the calculation of the covariance matrix via the following matrix equation:

$$\Sigma = \frac{1}{n-1} \left((\mathbf{X} - \bar{\mathbf{x}})^T \ (\mathbf{X} - \bar{\mathbf{x}}) \right)$$

($\bar{X}$ is the mean vector)

By using np.cov method, we can calculate the covariance matrix more easily.

$$\text{cov}(\mathbf{X}) = \frac{1}{n-1} \mathbf{X}^T \mathbf{X}$$

```
#step2: Find the covariance matrix
####################### Blank ######################
#co = np.matmul(z.T, z) / 149
#print(co)
covar_matrix = np.cov(z.T)
####################################################
```

```
The shape of covariance matrix: (4, 4)
```

3) Find the eigenvalues and eigenvectors of the covariance matrix

The eigenvectors and eigenvalues of a covariance (or correlation) matrix represent the “core” of a PCA: The eigenvectors (principal components) determine the directions of the new feature space, and the eigenvalues determine their magnitude. In other words, the eigenvalues explain the variance of the data along the new feature axes. We can perform an eigen-decomposition on the covariance matrix using np.linalg.eig method.

```
#step3 : Find the eigenvalues and eigenvectors of the covariance matrix
####################### Blank ######################
eig_vals, eig_vecs = np.linalg.eig(covar_matrix)
####################################################
```

4) Get eigenvectors of 2 biggest eigenvalues and form the projection matrix

The typical goal of a PCA is to reduce the dimensionality of the original feature space by projecting it onto a smaller subspace, where the eigenvectors will form the axes. In order to decide which eigenvector(s) can dropped without losing too much information for the construction of lower-dimensional subspace, we need to inspect the corresponding eigenvalues: The eigenvectors with the lowest eigenvalues bear the least information about the distribution of the data; those are the ones can be dropped. In order to do so, the common approach is to rank the eigenvalues from highest to lowest in order choose the top k eigenvectors. In this project, we choose the top 2 eigenvalues and corresponding eigenvectors to express our dataset in 2 dimensions.

```
#step4 : Get first 2 eigenvectors of 2 biggest eigenvalues and form the projection matrix
######################### Blank ########################
# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]
# Sort the (eigenvalue, eigenvector) tuples from high to low
eig_pairs.sort(key=lambda x: x[0], reverse=True)
# Visually confirm that the list is correctly sorted by decreasing eigenvalues
print('Eigenvalues in descending order:')
for i in eig_pairs:
    print(i[0])
print("All Eigen Values along with Eigen Vectors")
print(eig_pairs)
w_matrix = np.hstack((eig_pairs[0][1].reshape(4,1),
                      eig_pairs[1][1].reshape(4,1)))
projected_X = np.dot(iris_x, w_matrix)
########################################################
```

```
Eigenvalues in descending order:
2.938085050199995
0.9201649041624864
0.1477418210449475
0.020853862176462696
All Eigen Values along with Eigen Vectors
[(2.938085050199995, array([ 0.52106591, -0.26934744,  0.5804131,
0.56485654])), (0.9201649041624864, array([-0.37741762, -0.92329566,
-0.02449161, -0.06694199])), (0.1477418210449475, array([-0.71956635,
0.24438178,  0.14212637,  0.63427274])), (0.020853862176462696, array([
0.26128628, -0.12350962, -0.80144925,  0.52359713]))]
The shape of projected data: (150, 2)
```

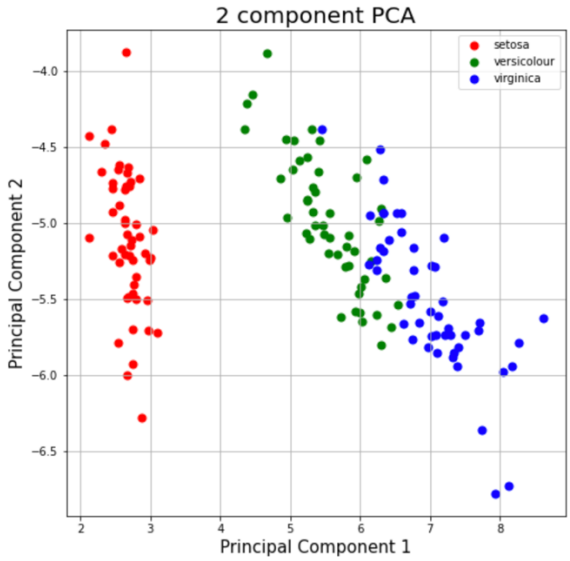
5) Get variance ratio

The fraction of variance explained by a principal component is the ratio between the variance of that principal component and the total variance.

```
#Step5 : get variance ratio
variance_ratio = eig_vals[:2]/eig_vals.sum()
```

```
[0.72962445 0.22850762]
```

6) Visualization



4-3. Using sklearn PCA package (with Iris dataset)

1) Standardization using sklearn.preprocessing.StandardScaler module

```
from sklearn.preprocessing import StandardScaler
#Step1. Standardization using StandardScaler
z = StandardScaler().fit_transform(iris_x)
print("The mean of z:", z.mean(axis=0))
```

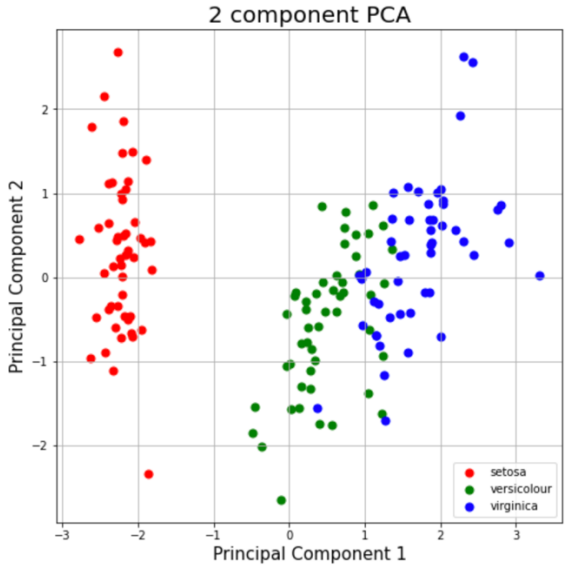
```
The mean of z: [-1.69031455e-15 -1.84297022e-15 -1.69864123e-15     -
1.40924309e-15]
```

The mean of z is exactly same as the Numpy implemented PCA.

2) PCA projection to 2 components using sklearn.decomposition.PCA module

```
from sklearn.decomposition import PCA
#Step2. PCA projection to 2 components
pca = PCA(n_components = 2)
principalComponents = pca.fit_transform(z)
```

3) Visualization



4) Eigenvalue and variance ratio of covariance matrix

```
#Step3 : Get eigenvlaue and variance ratio of covariance matrix

print('eigen_value :', pca.explained_variance_)
print('explained variance ratio:', pca.explained_variance_ratio_)
```

```
eigen_value : [2.93808505 0.9201649 ]
explained variance ratio: [0.72962445 0.22850762]
```

The 2 biggest eigenvalue and corresponding variance ratio are also exactly same as the Numpy implemented PCA.

Together, the first two principal components contain 95.81% of the information. The first principal component contains 72.96% of the variance and the second principal component contains 22.85% of the variance. The third and fourth principal component contained the rest of the variance of the dataset. Therefore, the three classes appear to be well separated.

Numpy implemented PCA and sklearn PCA behave quite same for the Iris dataset.