Project 1

20170616 정희진

Part 2. Logistic Regression

Step 1-1. Sigmoid Function

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

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

위 코드는 아래에 있는 식을 그대로 나타낸 것이다. 따라서 함수는 0에서 1사이 값을 반환할 것이다. Sigmoid function은 forward function 안에서 사용될 수 있다.

Step 1-2. Forward Function

```
def forward(x, y, w, eps=1e-8):
    ######################### Blank ########################
    predict = sigmoid(np.matmul(w,x.T))
    n = y.size
    loss = np.sum(- y * np.log(predict + eps) - (1 - y) * np.log(1 - predict + eps)) / n
    ########################################################
    return predict, loss
```

predict: prediction for each data point x_i; $h(w^T x_i) = \frac{1}{1+e^{-w^T x_i}}$

loss: $J(w) = \frac{1}{n}\sum_{i=1}^{n}[-y_i \log\{h(w^T x_i)\} - (1-y_i)\log\{1-h(w^T x_i)\}]$

위 코드는 아래에 있는 식을 그대로 나타낸 것이다. w와 x를 곱하기 위해 np.matmul()를 사용하였다. 식에 나타난 n은 y벡터의 길이와 같으므로 y.size로 지정하였다. 또한 log함수로 나타내기 위해 np.log()를 사용하였고, 각 element마다 계산된 값을 모두 더하기 위해 np.sum()을 사용하였다. 여기서 predict값은 backward function과 accuracy function의 parameter로 사용될 수 있으며, loss은 data를 apply할 때 w가 새로 바뀔 때마다 어떻게 변화하는지 관찰할 수 있다. Loss값을 구하는 식을 살펴보면, y가 0일 때 predict값이 0에 가까울수록 loss가 작아지고 1에 가까울수록 커진다. 반대로 y사 1일 때 predict값이 1에 가까울수록 loss가 작아지고 0에 가까울수록 커진다.

Step 1-3. Backward Function

```
def backward(x, y, predict):
    ######################### Blank ########################
    grad_w = np.matmul(predict - y, x) / y.size
    ########################################################
    return grad_w
```

$$\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i\left(1-h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)\right)-(1-y_i)\,h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)\right]\boldsymbol{x}_i$$
$$= \frac{1}{n}\sum_{i=1}^{n}\left[h\left(\boldsymbol{w}^T\boldsymbol{x}_i\right)-y_i\right]\boldsymbol{x}_i$$
$$= \frac{1}{n}(\hat{Y}-Y)X$$

위 코드는 아래에 있는 식을 그대로 나타낸 것이다. 마찬가지로 위에서 정의한 predict – y와 x를 곱하기 위해 np.matmul()을 사용하였고, n은 y.size로 나타낼 수 있다. 함수의 return값인 grad_w는 data를 apply할 때 w를 새로 바꾸는 데에 사용이 된다.

Step 1-4. Bias Unit Function

```
def bias_unit(x, w, b):
    ######################### Blank ########################
    w_bar = np.append(w, b)
    ones_vec = np.ones(x.shape[0]).reshape(1,-1)
    x_bar = np.append(x, ones_vec.T,axis = 1)
    ########################################################
    return x_bar, w_bar
```

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

위 코드는 아래에 있는 행렬과 벡터를 그대로 나타낸 것이다. 기존의 w에서 b값을 추가하기 위해 np.append()를 사용하였다. 그림의 x_bar 변수에서 여러 개의 1로 이루어진 벡터를 만들기 위해 np.ones()를 사용하였고, x 행렬에서 쉽게 append하기 위해 reshape()를 사용해 벡터의 형태를 바꿔주었다. 이 함수는 밑의 initialize parameters function에서 사용될 것이다.

Step 1-5. Initialize Parameters Function

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

    ######################### Blank ########################
    w = np.random.normal(size = X_train.shape[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
```

w의 초기값을 설정하기 위해서 np.random.normal()을 사용하였다. 벡터의 길이는 X_train변수의 column의 길이와 같을 것이므로 위와 같이 설정해주었다. b는 0으로 초기값을 설정해주었다.

Step 1-6. Accuracy Function

```
def accuracy(predict, y):
    ######################### Blank ########################
    acc = 0
    n = y.size
    y_bar = []
    for i in range(n):
        if predict[i] >= 0.5:
            y_bar = np.append(y_bar, 1)
        else:
            y_bar = np.append(y_bar, 0)
    for j in range(n):
        if y_bar[j] == y[j]:
            acc = acc + 1
    acc = acc / n * 100
    ########################################################
    return acc
```

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:

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

위 코드는 아래에 있는 식을 그대로 나타낸 것이다. 밑의 식에서 나타난 n은 y 변수의 길이로 나타내었다. 함수 안에 있는 y_bar변수는 밑의 식의 $\hat{y}_i$와 같다. 처음에는 빈 리스트로 만들고 predict의 값에 따라 1이나 0 값을 리스트에 넣어 총 n개의 element가 있는 리스트를 만들었다. 함수의 리턴값인 acc는 y_bar변수와 y변수의 각각의 element를 비교하면서 더하고 마지막에 n으로 나누고 100을 곱하여 단위를 %로 나타내었다.

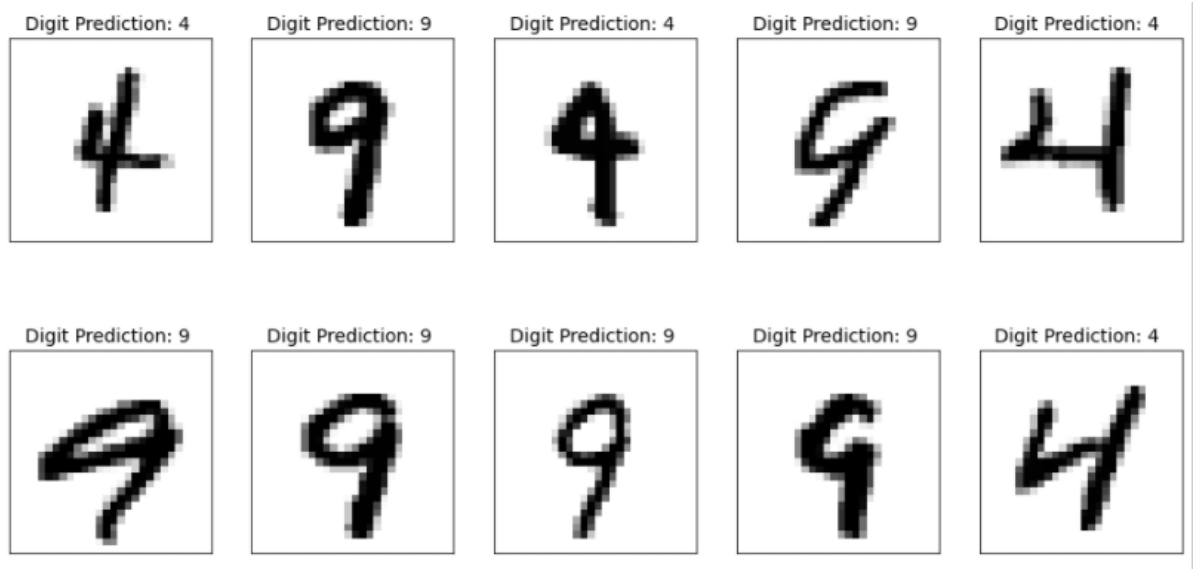
Step 2. Apply to the MNIST Dataset

빈 곳의 코드는 다음과 같이 썼다.

```
######################### 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 = w_bar - learning_rate * grad_w
#########################################################
```

결과는 다음과 같다.

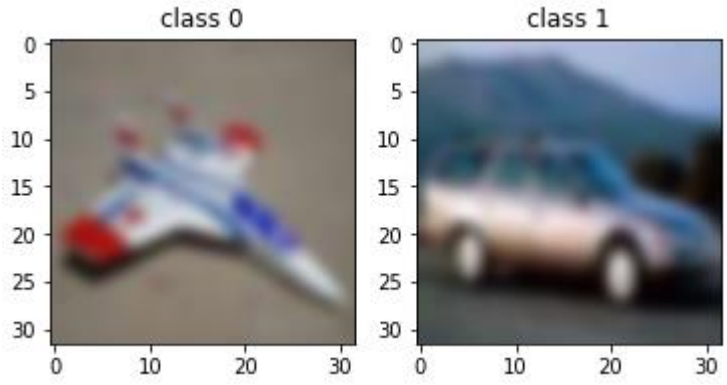
```
train accuracy: 97.32
test accuracy: 96.38
```



Loop가 한번 돌 때마다 진행도가 한 줄씩 나타나 결과를 나타내는데 어려움을 겪어 코드에서 다음을 주석 처리하였다.

```
#    if i % 100 == 0:
#        it.set_postfix(accuracy='{:.2f}'.format(train_acc),
#                       loss='{:.4f}'.format(loss))
```

Step 3. Apply to the CIFAR10 dataset



빈 곳의 코드는 다음과 같이 썼다.

```
######################## 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 = w_bar - learning_rate * grad_w
#######################################################
```

결과는 다음과 같다.

```
train accuracy: 74.19
test accuracy: 73.80
```

Loop가 한번 돌 때마다 진행도가 한 줄씩 나타나 결과를 나타내는데 어려움을 겪어 코드에서 다음을 주석 처리하였다.

```
#    if i % 100 == 0:
#        it.set_postfix(accuracy='{:.2f}'.format(train_acc),
#                       loss='{:.4f}'.format(loss))
```

Step 4-1. Forward with Regularization Function

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

predict: Prediction for each data point x_i : $h(w^T x_i) = \dfrac{1}{1+e^{-w^T x_i}}$

loss: $J(w) = \frac{1}{n}\sum_{i=1}^{n}[-y_i \log\{h(w^T x_i)\} - (1-y_i)\log\{1-h(w^T x_i)\}] + \frac{\lambda}{2n}\|w\|_2^2$

Step 1-2에서 추가된 부분은 loss를 계산한 식에서 lambda 변수가 있는 항이다. 이때 $\|w\|^2$을 계산하기 위해서 w의 각각의 element들을 제곱하고 np.sum()을 이용해 모두 더하였다. 다음으로 lambda 변수를 곱하고 2n을 나누어 주었다.

Step 4-2. Backward with Regularization Function

```
def backward_with_regularization(x, y, w, predict, lambda_):
    ######################## Blank ########################
    grad_w = np.matmul(predict - y, x) / y.size + lambda_ * w / y.size
    ########################################################
    return grad_w
```

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

Step 1-2에서 추가된 부분은 grad_w를 계산한 식에서 lambda 변수가 있는 항이다. w에 lambda를 곱하고 y.size(=n)을 나누어 주었다.

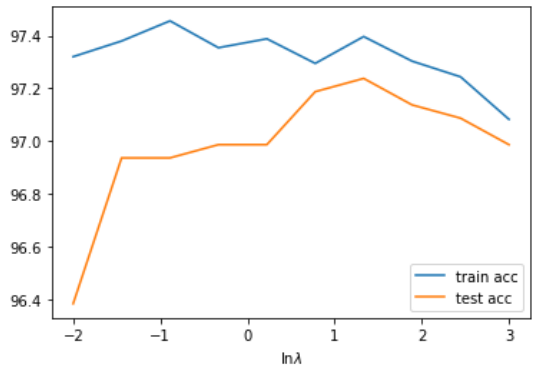
Step 4-3. Apply to the MNIST Dataset

빈 곳의 코드는 다음과 같이 썼다.

```
######################## 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 = w_bar - learning_rate * grad_w
#######################################################
```

결과는 다음과 같다.

```
train accuracy: 97.32
test accuracy: 96.38
train accuracy: 97.38
test accuracy: 96.94
train accuracy: 97.46
test accuracy: 96.94
train accuracy: 97.35
test accuracy: 96.99
train accuracy: 97.39
test accuracy: 96.99
train accuracy: 97.29
test accuracy: 97.19
train accuracy: 97.40
test accuracy: 97.24
train accuracy: 97.30
test accuracy: 97.14
train accuracy: 97.24
test accuracy: 97.09
train accuracy: 97.08
test accuracy: 96.99
```



Loop가 한번 돌 때마다 진행도가 한 줄씩 나타나 결과를 나타내는데 어려움을 겪어 코드에서 다음을 바꾸어 주었다.

```
#it = tqdm(range(num_of_iteration))
it = range(num_of_iteration)
```

```
#     if i % 100 == 0:
#         it.set_postfix(accuracy='{:.2f}'.format(train_acc),
#                        loss='{:.4f}'.format(loss))
```

Part 3. Hyperparameter Optimization

Step 2. Logistic Regression Classifier

빈 곳의 코드는 다음과 같이 썼다.

```
from sklearn.linear_model import SGDClassifier
np.random.seed(0)

######################## Blank ########################
def SGDaccuracy(predict, y):
    acc = 0
    n = y.size
    for i in range(n):
        if predict[i] == y[i]:
            acc = acc + 1
    acc = acc / n * 100
    return acc

cl1 = SGDClassifier(loss = 'log',alpha = 1e-3,eta0 = 1e-1)
cl2 = SGDClassifier(loss = 'log',alpha = 1e-3,eta0 = 1e-1)
cl3 = SGDClassifier(loss = 'log',alpha = 1e-3,eta0 = 1e-1)
cl4 = SGDClassifier(loss = 'log',alpha = 1e-4,eta0 = 1e-2)
cl5 = SGDClassifier(loss = 'log',alpha = 1e-4,eta0 = 1e-2)
cl6 = SGDClassifier(loss = 'log',alpha = 1e-4,eta0 = 1e-2)
cl7 = SGDClassifier(loss = 'log',alpha = 1e-5,eta0 = 1e-3)
cl8 = SGDClassifier(loss = 'log',alpha = 1e-5,eta0 = 1e-3)
cl9 = SGDClassifier(loss = 'log',alpha = 1e-5,eta0 = 1e-3)

cl1.fit(X_train,Y_train)
cl2.fit(X_train,Y_train)
cl3.fit(X_train,Y_train)
cl4.fit(X_train,Y_train)
cl5.fit(X_train,Y_train)
cl6.fit(X_train,Y_train)
cl7.fit(X_train,Y_train)
cl8.fit(X_train,Y_train)
cl9.fit(X_train,Y_train)

Y_pred1 = cl1.predict(X_test)
Y_pred2 = cl2.predict(X_test)
Y_pred3 = cl3.predict(X_test)
Y_pred4 = cl4.predict(X_test)
Y_pred5 = cl5.predict(X_test)
Y_pred6 = cl6.predict(X_test)
Y_pred7 = cl7.predict(X_test)
Y_pred8 = cl8.predict(X_test)
Y_pred9 = cl9.predict(X_test)
```

```
print('acc1: {:.2f}, acc2: {:.2f}, acc3: {:.2f}, acc4: {:.2f}, acc5: {:.2f}, acc6: {:.2f}, acc7: {:.2f}, acc8: {:.2f}, acc9: {:.2f}'.format(acc1,acc2,acc3,acc4,acc5,acc6,acc7,acc8,acc9))
eta0 = 1e-3
alpha = 1e-5
acc = acc7
################################################
print(f'eta0 is {eta0}, alpha is {alpha}, Acc = {acc}%')
```

결과는 다음과 같다.

```
acc1: 52.60, acc2: 59.20, acc3: 53.80, acc4: 61.13, acc5: 60.47, acc6: 56.60, acc7: 62.47, acc8: 58.60, acc9: 62.40
eta0 is 0.001, alpha is 1e-05, Acc = 62.46666666666667%
```

Step 3. SVM Classifier

빈 곳의 코드는 다음과 같이 썼다.

```
from sklearn.svm import SVC
np.random.seed(0)

####################### Blank ######################
def SVCaccuracy(predict, y):
    acc = 0
    n = y.size
    for i in range(n):
        if predict[i] == y[i]:
            acc = acc + 1
    acc = acc / n * 100
    return acc

svc1 = SVC(kernel= 'linear', C = 0.1)
svc2 = SVC(kernel= 'linear', C = 1)
svc3 = SVC(kernel= 'linear', C = 10)
svc4 = SVC(kernel= 'linear', C = 100)
svc5 = SVC(kernel= 'rbf', C = 0.1)
svc6 = SVC(kernel= 'rbf', C = 1)
svc7 = SVC(kernel= 'rbf', C = 10)
svc8 = SVC(kernel= 'rbf', C = 100)
svc9 = SVC(kernel= 'poly', C = 0.1)
svc10 = SVC(kernel= 'poly', C = 1)
svc11 = SVC(kernel= 'poly', C = 10)
svc12 = SVC(kernel= 'poly', C = 100)

svc1.fit(X_train,Y_train)
svc2.fit(X_train,Y_train)
svc3.fit(X_train,Y_train)
svc4.fit(X_train,Y_train)
svc5.fit(X_train,Y_train)
svc6.fit(X_train,Y_train)
svc7.fit(X_train,Y_train)
svc8.fit(X_train,Y_train)
svc9.fit(X_train,Y_train)
svc10.fit(X_train,Y_train)
svc11.fit(X_train,Y_train)
svc12.fit(X_train,Y_train)
```

```
Y_pred1 = svc1.predict(X_test)
Y_pred2 = svc2.predict(X_test)
Y_pred3 = svc3.predict(X_test)
Y_pred4 = svc4.predict(X_test)
Y_pred5 = svc5.predict(X_test)
Y_pred6 = svc6.predict(X_test)
Y_pred7 = svc7.predict(X_test)
Y_pred8 = svc8.predict(X_test)
Y_pred9 = svc9.predict(X_test)
Y_pred10 = svc10.predict(X_test)
Y_pred11 = svc11.predict(X_test)
Y_pred12 = svc12.predict(X_test)


acc1 = SVCaccuracy(Y_pred1, Y_test)
acc2 = SVCaccuracy(Y_pred2, Y_test)
acc3 = SVCaccuracy(Y_pred3, Y_test)
acc4 = SVCaccuracy(Y_pred4, Y_test)
acc5 = SVCaccuracy(Y_pred5, Y_test)
acc6 = SVCaccuracy(Y_pred6, Y_test)
acc7 = SVCaccuracy(Y_pred7, Y_test)
acc8 = SVCaccuracy(Y_pred8, Y_test)
acc9 = SVCaccuracy(Y_pred9, Y_test)
acc10 = SVCaccuracy(Y_pred10, Y_test)
acc11 = SVCaccuracy(Y_pred11, Y_test)
acc12 = SVCaccuracy(Y_pred12, Y_test)

print('acc1: {:.2f}, acc2: {:.2f}, acc3: {:.2f}, acc4: {:.2f}, acc5: {:.2f}, acc6: {:.2f}, acc7: {:.2f}, acc8: {:.2f}, acc9: {:.2f}, acc10: {:.2f}, acc11: {:.2f}, acc12: {:.2f}'.format(acc1,acc2,acc3,acc4,acc5,acc6,acc7,acc8,acc9,acc10,acc11,acc12))
kernel = 'rbf'
C = 10
acc = acc7
#################################################
print(f'kernel is {kernel}, C is {C}, Acc = {acc}%')
```

결과는 다음과 같다.

```
acc1: 55.47, acc2: 55.47, acc3: 55.47, acc4: 55.47, acc5: 68.20, acc6: 74.60, acc7: 75.27, acc8: 73.93, acc9: 73.40, acc10: 71.27, acc11: 69.13, acc12: 68.40
kernel is rbf, C is 10, Acc = 75.26666666666667%
```

Part 4. PCA

Step 2-1. Standardization

```
#step1: Standardization
######################## Blank ########################
mean = np.mean(iris_x, axis = 0)
std = np.std(iris_x, axis = 0)
z = (iris_x - mean) / std
#######################################################
print("The mean of z:", z.mean(0))
```

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

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

위 코드는 아래에 있는 식을 그대로 나타낸 것이다. 평균을 구하기 위하여 np.mean()을 사용하였고, 표준편차를 구하기 위해 np.std()을 사용하였다. 표준화된 z를 구하기 위해 원래의 값(iris_x)에 평균을 빼고 표준편차로 나누어 주었다.

Step 2-2. Covariance Matrix

```
#step2: Find the covariance matrix
######################## Blank ########################
covar_matrix = np.cov(z.T)
#######################################################
print("The shape of covariance matrix: ",covar_matrix.shape)
```

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

Covariance matrix를 구하기 위해 np.cov()을 사용하였다. 위에서 구한 z행렬을 transpose 시키고 parameter값으로 넣었다.

Step 2-3. Eigenvalues, and Eigenvectors

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

Eigenvalue와 eigenvector을 구하기 위해 np.linalg.eig()을 사용하였다. Eig_vals은 shaperk (4,) eig_vecs는 (4, 4)였는데, eigenvector은 각 column별로 각 vector을 나타냈다.

Step 2-4. Eigenvectors of 2 Biggest Eigenvectors

```
#step4 : Get first 2 eigenvectors of 2 biggest eigenvalues and from the projection matrix
######################### Blank ########################
eig_vecs2 = eig_vecs[:,(0,1)]
projected_X = np.matmul(iris_x,eig_vecs2)
########################################################
print("The shape of projected data: ", projected_X.shape)
```

```
The shape of projected data:  (150, 2)
```

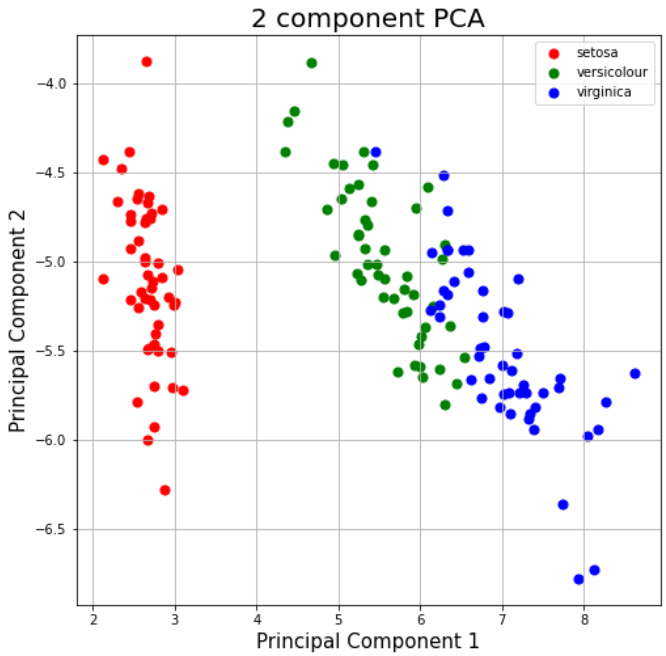
가장 큰 eigenvalue와 두번째로 큰 eigenvalue가 각각 첫번째, 두번째 위치에 있었기 때문에 eigenvector를 구할 때도 첫번째 column과 두번째 column에 있는 vector을 나타내었다. 이 두 벡터로 이루어진 좌표평면과 이에 맞는 데이터를 나타내기 위해 iris_x를 이 두 벡터에 project시켰다. 이 과정에서 np.matmul()을 사용하였다.

Step 2-5. Variance Ratio

결과는 다음과 같다.

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

```
[0.72962445 0.22850762]
```

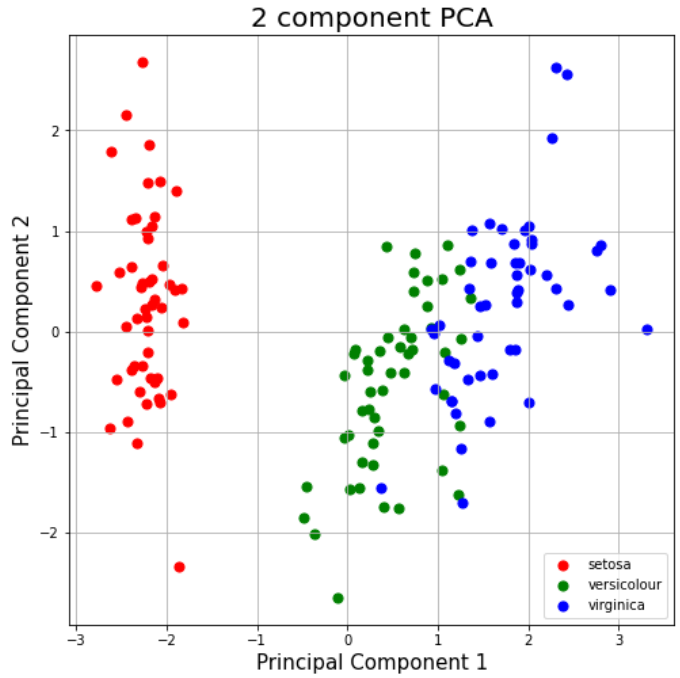


Step 3. Using sklearn PCA Package

결과는 다음과 같다.

```
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]
```




```
#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]
```

구한 2개의 principal component를 통해 데이터를 얼마나 잘 나타냈는지 수치적으로 확인하는 방법은 variance ratio를 보는 것이다. 앞 예제와 비교를 했을 때 두 결과 모두 variance ratio가 [0.72962445 0.22850762]인 것을 생각하면 두 예제 모두 전체 데이터 분산의 약 95.81%을 보존한다는 것을 알 수 있다.