

# PCA1

September 5, 2018

## 1 Principal Component Analysis

### 1.0.1 1- Description

- If the dataset includes a large number of variables, it can be difficult to visualise it
- It is useful to truncate the number of variables into a small number of dimensions in order to be able to visualise the most critical dimensions
- This means that we are looking for a representation of the data in a lower dimensional space
- The task is to find representative dimensions for the dataset derived from a linear combination of the original variables and order them from the most important to the least important
- This is what Principal Component Analysis (PCA) aims to accomplish

### 1.0.2 2- Iris Dataset

- The Iris dataset consists of 3 different kinds of irises (Setosa, Versicolor, and Virginica)
- The variables are petal and sepal length and width
- The classes correspond to 0 - Setosa, 1 - Versicolor, 2 - Virginica

First we import the iris dataset

```
In [1]: # Import a dummy dataset (iris)
        from sklearn import datasets

        # Load the dataset
        iris = datasets.load_iris()
```

We then import any modules we may use

```
In [2]: # Import modules
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        %matplotlib inline
```

We now create the data frames we will need

```
In [3]: # Create the features and class data frames
iris_features = pd.DataFrame(iris.data, columns=iris.feature_names)
iris_class = pd.DataFrame(iris.target, columns=['Class'])

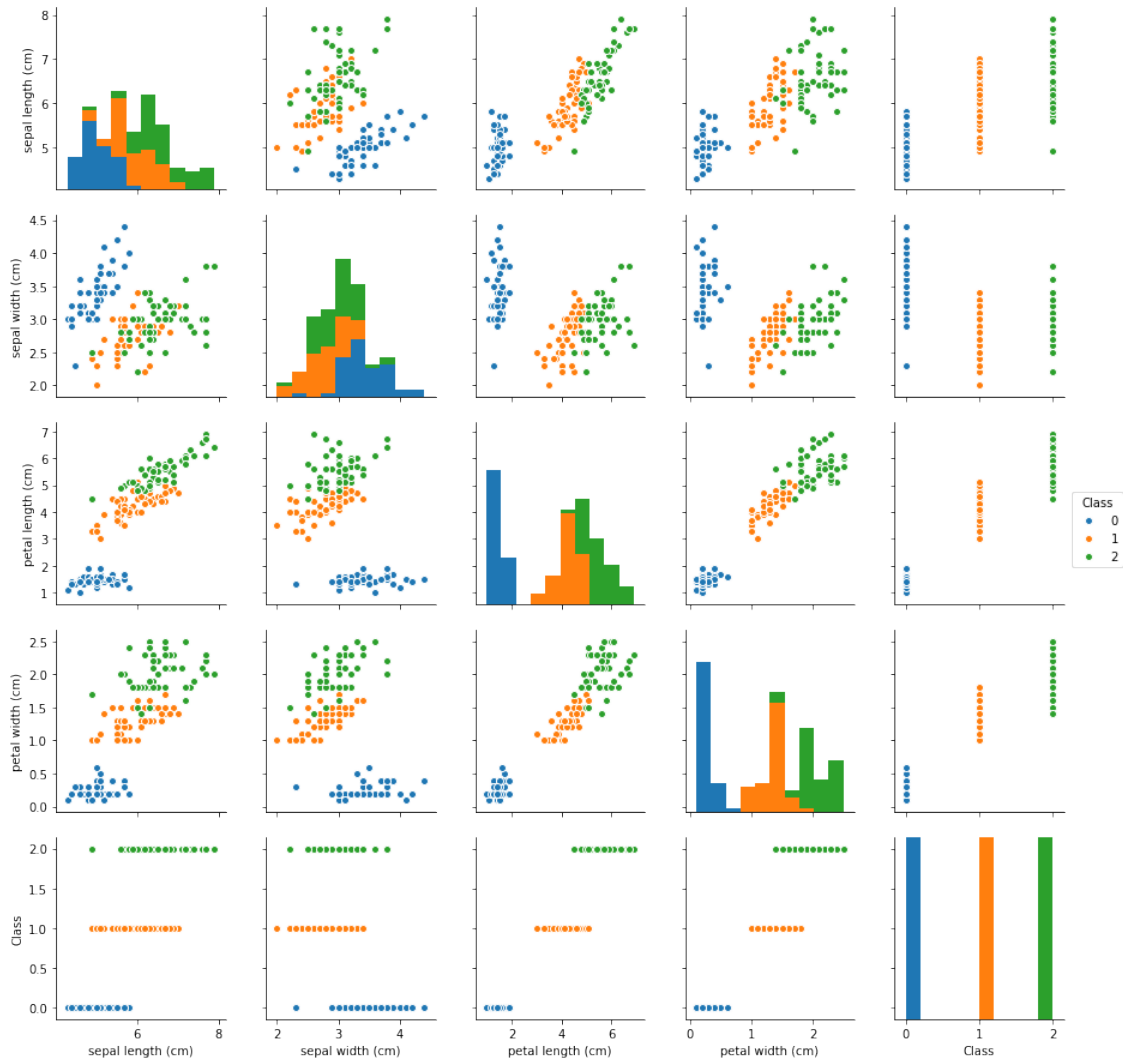
# Create a complete data frame
iris_data = pd.concat([iris_features, iris_class], axis=1)
```

It is useful to understand the data we are dealing with. In the above, `iris_features` is a dataframe consisting only of the explanatory variables - sepal length (cm), sepal width (cm), petal length (cm) and petal width (cm) - whereas the `iris_class` dataframe consists of only the response variable - Class.

A pairplot is useful in quickly investigating any relationships between the variables. We can see from the pairplot below that petal width and petal length are correlated and either are an indicator to the class of the iris. However this is not true for the sepal width and sepal length. It can be seen that a large sepal width and a small sepal length may be indicative of class 2 - Virginica. We therefore expect that there exists a principal component with these two variables having opposite signs.

```
In [4]: # Create a pairplot of each variable to identify if there is the class is easily disti
sns.pairplot(data=iris_data, hue='Class')
```

```
Out[4]: <seaborn.axisgrid.PairGrid at 0x27e7e793eb8>
```



### 1.0.3 3- Principal Component Analysis (PCA)

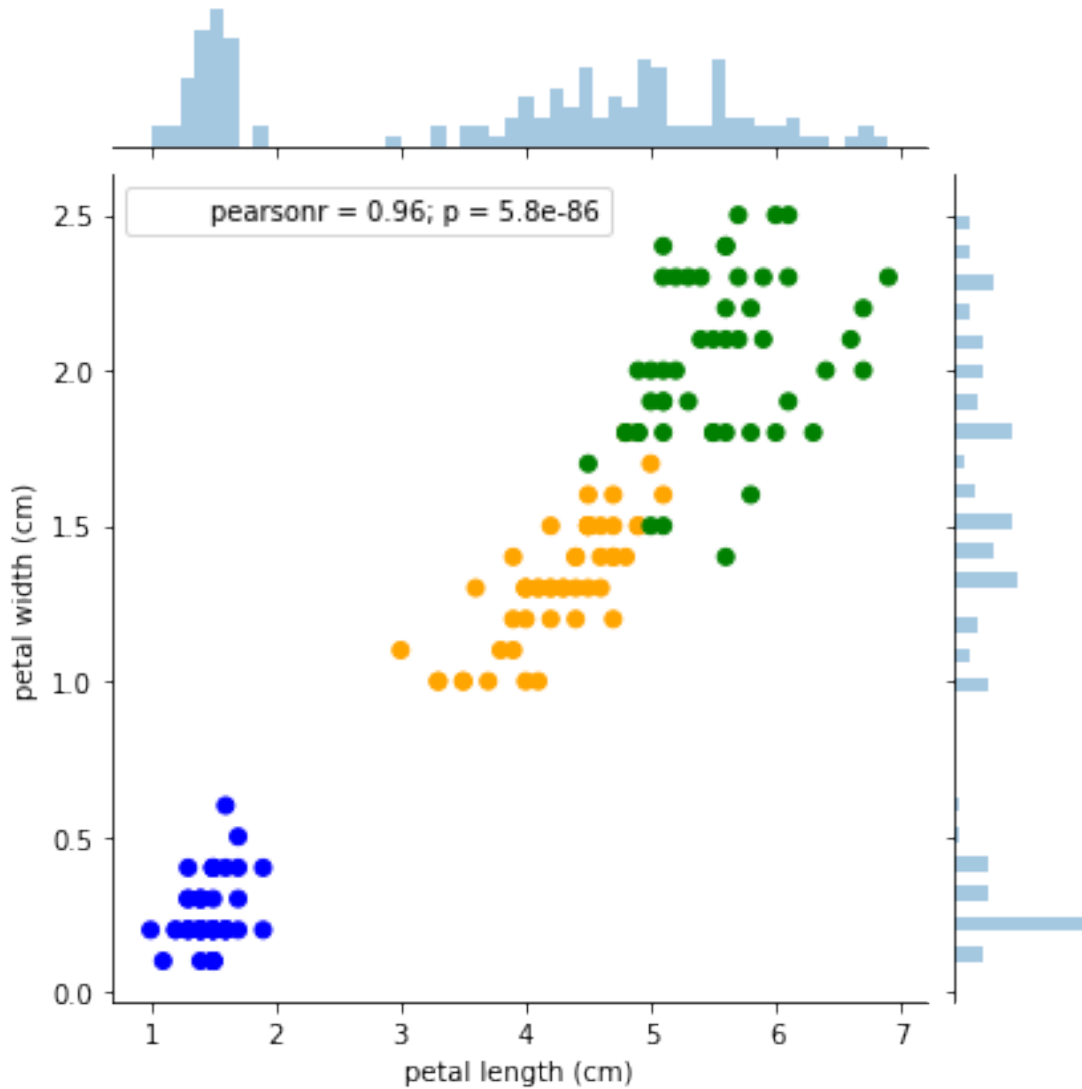
The idea behind PCA is the determination of orthonormal dimensions such that each dimension captures an axis with approximately maximum variance. For example in a simplified plot below of petal length, petal width and class, the maximum variance is obtained approximately along the petal width = petal length line, since the projections of the points onto this line has the largest variance.

```
In [5]: def colorify(x):
        if x==0:
            return 'blue'
        elif x==1:
            return 'orange'
        else:
            return 'green'
```

```
# Create a color array to attach a color to each point depending on its label
color = iris_data['Class'].apply(colorify)
```

```
In [6]: sns.jointplot(x=iris_data['petal length (cm)'],y=iris_data['petal width (cm)'],joint_k
```

```
Out[6]: <seaborn.axisgrid.JointGrid at 0x27e0342a4e0>
```



Let there be  $p$  variables in our dataset with  $\mathbf{X} = (\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_p)^T$  a column vector of random variables representing the variables of the dataset. A principal component,  $P_1$ , then has the form

$$P_1 = \alpha_{11}\mathbf{X}_1 + \alpha_{12}\mathbf{X}_2 + \dots + \alpha_{1p}\mathbf{X}_p = \sum_{i=1}^p \alpha_{1i}\mathbf{X}_i = (\alpha_{11}, \alpha_{12}, \dots, \alpha_{1p})\mathbf{X} = \boldsymbol{\alpha}_1^T \mathbf{X}$$

Then the *first principal component* is the linear combination of the variables which results in the greatest variance. The variance of  $P_1$  can be calculated as

$$\text{Var}[P_1] = E[\mathbf{P}_1^2] - E[\mathbf{P}_1]^2 = \boldsymbol{\alpha}_1^T \boldsymbol{\Sigma} \boldsymbol{\alpha}_1$$

where  $\boldsymbol{\Sigma}$  is the  $p \times p$  covariance matrix of  $\mathbf{X}$ . This result is shown in Appendix A.1.

Since  $P_1$ , the *first principal component*, is the principal component with the greatest variance, we will need to maximise  $\text{Var}[P_1]$  in terms of  $\alpha_i$ . i.e.

$$\max_{\alpha_i} P_1(\alpha_i, \mathbf{X})$$

The initial point here is that  $P_1$  is just a linear combination of the variables, so nothing stops the unbounded variables,  $\alpha_i$  from going to infinity. We solve this issue by adding the constraint

$$\sum_i \alpha_{1i}^2 = \boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_1 = 1$$

Given this bound, the coefficients will have to compete with each other to maximise  $P_1$ . Note: A constraint of the form  $\sum_i \alpha_{1i} = 1$  is not sufficient as it doesn't solve the issue of unboundedness.

To calculate  $\text{Var}(P_1)$ , as it is currently written, we will need to know the covariance matrix  $\boldsymbol{\Sigma}$ . However, often we are dealing with samples and do not know the covariance matrix. An unbiased, Maximum Likelihood Estimator (MLE) for the covariance matrix of the data  $\mathbf{X}$  is

$$\mathbf{S} = \frac{1}{n-1} \sum_{i=1}^n (\mathbf{x}_i - \hat{\boldsymbol{\mu}})(\mathbf{x}_i - \hat{\boldsymbol{\mu}})^T$$

where  $\mathbf{X}$  is a  $p \times n$  data matrix,  $\mathbf{x}_i$  is the  $p \times 1$  column vector for observation  $i$ , and  $\hat{\boldsymbol{\mu}} = \frac{1}{n} \sum_{i=1}^n \mathbf{x}_i$  is the  $p \times 1$  unbiased MLE for the mean  $\boldsymbol{\mu}$  where  $\mu_i$  is the mean value of the  $i$ th parameter/variable.

Sometimes  $\mathbf{S}$  is expressed as

$$\mathbf{S} = \frac{1}{n-1} \mathbf{X}^T \mathbf{X}$$

where  $\mathbf{X}$  is already mean centered.

We therefore need to maximise

$$\boldsymbol{\alpha}_1^T \mathbf{S} \boldsymbol{\alpha}_1$$

with the constraint

$$\boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_1 = 1$$

We can do this via *Lagrange Multipliers*. Let  $\mathcal{L}(\boldsymbol{\alpha}_1, \lambda_1)$  be the Lagrange function defined by

$$\mathcal{L}(\boldsymbol{\alpha}_1, \lambda_1) = f(\boldsymbol{\alpha}_1) - \lambda_1 g(\boldsymbol{\alpha}_1)$$

where  $f(\boldsymbol{\alpha}_1)$  is the function to be maximised/minimised and  $g(\boldsymbol{\alpha}_1)$  is a constraint. We need to solve

$\nabla_{\boldsymbol{\alpha}_1, \lambda_1} \mathcal{L}(\boldsymbol{\alpha}_1, \lambda_1) = 0$  which in turn implies solving  $\nabla_{\boldsymbol{\alpha}_1} f(\boldsymbol{\alpha}_1) = \lambda_1 \nabla_{\boldsymbol{\alpha}_1} g(\boldsymbol{\alpha}_1)$  and  $g(\boldsymbol{\alpha}_1) = 0$ .

In our particular case we have

$$\mathcal{L}(\boldsymbol{\alpha}_1, \lambda_1) = \boldsymbol{\alpha}_1^T \mathbf{S} \boldsymbol{\alpha}_1 - \lambda_1 (\boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_1 - 1)$$

Then

$$\nabla \mathcal{L}(\alpha_1, \lambda_1) = 2S\alpha_1 - 2\lambda_1\alpha_1 = 0$$

Resulting in the eigenvalue equation

$$S\alpha_1 = \lambda_1\alpha_1 \implies \lambda_1 \text{ is a solution to } \det(S - \lambda I) = 0$$

This has at most  $p$  roots. Since we're looking for the  $P_1$  which is the solution resulting in the largest variance, we choose the largest of these roots since  $\lambda_1$  is related to  $\text{Var}(P_1)$ . This can be seen by multiplying the eigenvalue equation above on both sides by  $\alpha_1$  to obtain  $\text{Var}(P_1)$

$$\alpha_1^T S \alpha_1 = \lambda_1$$

We first calculate  $S$  for our dataset.

```
In [7]: def VectorScalarSubtract(u,v):
        '''
        Subtracts scalar v from vector u.
        '''
        return len(u)

def VectorSubtract(u,v):
    '''
    Subtracts vector v from vector u.
    Vectors must be of the same size.
    '''
    tempArray = u.loc[:]

    for i in range(len(u)):
        tempArray[i] = u[i] - v[i]

    return tempArray

def MatrixScalarMult(X,s):
    rows = len(X[0])
    cols = len(X)
    temp = X

    for i in range(rows):
        for j in range(cols):
            temp[i][j] = X[i][j]*s

    return temp

def DotProduct(u,v):
    total = 0
    for i in range(len(u)):
        total += u[i]*v[i]

    return total
```

```

def MultMatrix(X,Y):
    p = len(X)
    obs = len(X[0])
    temparray = []
    Z = []

    for i in range(p):
        u = X[i]
        temparray = []
        for j in range(p):
            v = Y.transpose()[j]
            temparray.append(DotProduct(u,v))

        Z.append(temparray)

    return Z

def CenterMatrix(X):
    obs = len(X.iloc[:,0])
    params = len(X.columns)
    tempArray = []

    mu = []
    for i in range(params):
        v = sum(X.iloc[:,i])/obs
        mu.append(v)

    for i in range(obs):
        a = X.iloc[i,:]
        tempArray.append(VectorSubtract(a,mu))

    return tempArray

def CalcCovarMatrix(X):
    obs = len(X.iloc[:,0])

    M = np.array(CenterMatrix(X))
    MT = M.transpose()

    Z = MultMatrix(MT,M)

    Z = MatrixScalarMult(Z,1/(obs-1))

    return Z

def Determinant(X):
    det = 0

```

```

if len(X) == 2:
    return (X.iloc[0,0] * X.iloc[1,1]- X.iloc[0,1] * X.iloc[1,0])
else:
    # Go along the top row
    for i in range(len(X.iloc[0,:])):
        #print(GetSubMatrix(X,0,i))
        #print(Determinant(GetSubMatrix(X,0,i)))

        det += (-1)**i * X.iloc[0,i] * Determinant(GetSubMatrix(X,0,i))
    return det

def MatrixSubtract(X,Y,dp):
    F = pd.DataFrame.copy(X)
    for i in range(len(C)):
        for j in range(len(C[0])):
            F[i][j] = round(Z[i][j] - C[i][j],dp)

    return F

def GetSubMatrix(X,i,j):
    T = X.drop(X.columns[j],axis=1)
    T = T.drop(T.index[i],axis=0)

    return T

def power_iteration(A, num_simulations):
    # Ideally choose a random vector
    # To decrease the chance that our vector
    # Is orthogonal to the eigenvector
    b_k = np.random.rand(A.shape[1])
    l = 0

    for _ in range(num_simulations):
        # calculate the matrix-by-vector product Ab
        b_k1 = np.dot(A, b_k)

        # calculate the norm
        b_k1_norm = np.linalg.norm(b_k1)

        # re normalize the vector
        b_k = b_k1 / b_k1_norm

    l = (np.dot(b_k.transpose(),np.dot(A,b_k)))/(np.dot(b_k.transpose(),b_k))

    return b_k,l

```

```

In [38]: # The Iris dataset
X = iris_features

```



```
# Calculate the covariance matrix for our dataset
Z = CalcCovarMatrix(X)
Z = pd.DataFrame(Z)

Z
```

```
Out [38]:
```

	0	1	2	3
0	0.685694	-0.039268	1.273682	0.516904
1	-0.039268	0.188004	-0.321713	-0.117981
2	1.273682	-0.321713	3.113179	1.296387
3	0.516904	-0.117981	1.296387	0.582414

The calculation of the covariance matrix can also be done using numpy with the `np.cov()` method as follows (we see that we have the same result)

```
In [39]: C = np.matrix(X).transpose()
C = pd.DataFrame(np.cov(C))
C
```

```
Out [39]:
```

	0	1	2	3
0	0.685694	-0.039268	1.273682	0.516904
1	-0.039268	0.188004	-0.321713	-0.117981
2	1.273682	-0.321713	3.113179	1.296387
3	0.516904	-0.117981	1.296387	0.582414

The eigenvector of the covariance matrix corresponding to the eigenvalue with the greatest magnitude can be iteratively calculated using the Power Iteration method (for now taken from wikipedia)

```
In [40]: a_1, l_1 = power_iteration(Z, 20)
```

The eigenvalue is

```
In [41]: print(l_1)
```

```
4.224840768320114
```

And the corresponding eigenvector (for the *first principal component*) is

```
In [43]: a_1
```

```
Out [43]: array([ 0.36158968, -0.08226889,  0.85657211,  0.35884393])
```

The eigenvalue,  $a_1$ , and eigenvector,  $v$  are by definition a solution to the equation

$$S a_1 = l_1 a_1$$

We can check that this is indeed true

In [44]: `np.subtract(np.dot(Z,a_1),np.dot(l_1,a_1))`

Out [44]: `array([0., 0., 0., 0.])`

We now need to find the *second principal component*

$$P_2 = \alpha_2^T X$$

However, the difference with what we did before is that the eigenvector for  $P_2$  has to be orthogonal to that for  $P_1$ . i.e.

$$\alpha_1^T \alpha_2 = 0$$

This constraint along with the constraint analogous to the previous case

$$\alpha_2^T \alpha_2 = 1$$

allows the discovery of an eigenvector that is different from  $\alpha_1$

This results in the lagrange equation (maximising this corresponds to minimising  $\alpha_1^T \alpha_2$ )

$$\mathcal{L}(\alpha_2, \lambda_2, r) = f(\alpha_2) - \lambda_2 g(\alpha_2) - r \alpha_1^T \alpha_2$$

Resulting in

$$\nabla \mathcal{L} = 2S\alpha_2 - 2\lambda_2 \alpha_2 - r\alpha_1$$

Setting this to 0

$$S\alpha_2 - \lambda_2 \alpha_2 = \frac{1}{2} r \alpha_1$$

multiplying both sides by  $\alpha_1^T$

$$\alpha_1^T S\alpha_2 - \lambda_2 \alpha_1^T \alpha_2 = \frac{1}{2} r \alpha_1^T \alpha_1$$

$$\implies \alpha_1^T S\alpha_2 = \frac{1}{2} r$$

where  $\alpha_1^T \alpha_2 = 0$  and  $\alpha_1^T \alpha_1 = 1$

Remembering that

$$S\alpha_1 = \lambda_1 \alpha_1$$

and multiplying both sides by  $\alpha_2^T$

$$\alpha_2^T S\alpha_1 = \lambda_1 \alpha_2^T \alpha_1 = 0$$

This means that  $r = 0$ . Finally,

$$S\alpha_2 = \lambda_2 \alpha_2 \implies \alpha_2^T S\alpha_2 = \lambda_2$$

meaning that  $\lambda_2$  is related to the variance corresponding to the *second principal component* and must be the second largest in magnitude.

Here we can use the `numpy.linalg` module to calculate the eigenvalues for our covariance matrix.

```
In [45]: from numpy import linalg as LA
```

```
In [46]: L,E = LA.eig(Z)
         E = E.transpose()
         print('Eigenvectors')
         print(E)

         print('Eigenvalues')
         print(L)
```

Eigenvectors

```
[[ 0.36158968 -0.08226889  0.85657211  0.35884393]
 [-0.65653988 -0.72971237  0.1757674   0.07470647]
 [-0.58099728  0.59641809  0.07252408  0.54906091]
 [ 0.31725455 -0.32409435 -0.47971899  0.75112056]]
```

Eigenvalues

```
[4.22484077  0.24224357  0.07852391  0.02368303]
```

In order to check that this is indeed the eigenvalue/eigenvector decomposition of  $Z$ , we test the following equation

$$EZ = \Lambda E^T$$

where  $E = (\alpha_1, \alpha_2, \dots, \alpha_p)^T$  is the matrix of eigenvectors and  $\Lambda$  is a diagonal matrix where  $\Lambda_{ii}$  is the  $i$ th eigenvalue.

```
In [47]: # The diagonal matrix
         LI = np.identity(4)

         for i in range(4):
             LI[i] = np.dot(LI[i],L[i])

         LI
```

```
Out[47]: array([[4.22484077, 0.          , 0.          , 0.          ],
                [0.          , 0.24224357, 0.          , 0.          ],
                [0.          , 0.          , 0.07852391, 0.          ],
                [0.          , 0.          , 0.          , 0.02368303]])
```

We can see below that we have approximate equality

```
In [48]: print(np.subtract(np.dot(E,Z),np.dot(LI,E)))

[[-2.22044605e-16  8.32667268e-16  2.66453526e-15  6.66133815e-16]
 [-4.71844785e-16  1.66533454e-16 -8.04911693e-16 -5.23886490e-16]
 [-2.84494650e-16  1.59594560e-16 -4.47558657e-16 -9.71445147e-17]
 [ 4.33680869e-18 -2.51534904e-17 -1.21430643e-16 -7.28583860e-17]]
```

The principal components are then as follows:

$$P_1 = \alpha_{11}X_1 + \alpha_{12}X_2 + \alpha_{13}X_3 + \alpha_{14}X_4 + \alpha_{15}X_5$$

$$P_2 = \alpha_{21}X_1 + \alpha_{22}X_2 + \alpha_{23}X_3 + \alpha_{24}X_4 + \alpha_{25}X_5$$

$$P_3 = \alpha_{31}X_1 + \alpha_{32}X_2 + \alpha_{33}X_3 + \alpha_{34}X_4 + \alpha_{35}X_5$$

$$P_4 = \alpha_{41}X_1 + \alpha_{42}X_2 + \alpha_{43}X_3 + \alpha_{44}X_4 + \alpha_{45}X_5$$

$$P_5 = \alpha_{51}X_1 + \alpha_{52}X_2 + \alpha_{53}X_3 + \alpha_{54}X_4 + \alpha_{55}X_5$$

This can be written concisely in matrix form as

$$K_{ij} = \sum_{l=1}^5 X_{il} V_{lj}$$

where  $V = \alpha^T = (\alpha_1, \alpha_2, \dots, \alpha_5)$  and  $X$  is the  $n \times p$  data matrix with  $n$  observations in  $p = 5$  variables.

$K$  is called the *scores matrix* where  $K_{ij}$  is the value of the  $j$ th principal component of the  $i$ th observation and  $V$  is the *loadings matrix*.  $K$  is the projections of the data onto the principal components. This means we can plot the columns of the scores matrix against each other to see how the principal components distinguish the data.

```
In [49]: V = E.transpose()
         K = np.dot(X,V)
```

```
In [50]: # This is the first principal component for all 150 observations
         K[:,0]
```

```
Out[50]: array([-2.68420713, -2.71539062, -2.88981954, -2.7464372 , -2.72859298,
                -2.27989736, -2.82089068, -2.62648199, -2.88795857, -2.67384469,
                -2.50652679, -2.61314272, -2.78743398, -3.22520045, -2.64354322,
                -2.38386932, -2.6225262 , -2.64832273, -2.19907796, -2.58734619,
                -2.3105317 , -2.54323491, -3.21585769, -2.30312854, -2.35617109,
                -2.50791723, -2.469056 , -2.56239095, -2.63982127, -2.63284791,
                -2.58846205, -2.41007734, -2.64763667, -2.59715948, -2.67384469,
                -2.86699985, -2.62522846, -2.67384469, -2.98184266, -2.59032303,
                -2.77013891, -2.85221108, -2.99829644, -2.4055141 , -2.20883295,
                -2.71566519, -2.53757337, -2.8403213 , -2.54268576, -2.70391231,
                 1.28479459,  0.93241075,  1.46406132,  0.18096721,  1.08713449,
                 0.64043675,  1.09522371, -0.75146714,  1.04329778, -0.01019007,
                -0.5110862 ,  0.51109806,  0.26233576,  0.98404455, -0.174864 ,
                 0.92757294,  0.65959279,  0.23454059,  0.94236171,  0.0432464 ,
                 1.11624072,  0.35678657,  1.29646885,  0.92050265,  0.71400821,
                 0.89964086,  1.33104142,  1.55739627,  0.81245555, -0.30733476,
                -0.07034289, -0.19188449,  0.13499495,  1.37873698,  0.58727485,
                 0.8072055 ,  1.22042897,  0.81286779,  0.24519516,  0.16451343,
                 0.46303099,  0.89016045,  0.22887905, -0.70708128,  0.35553304,
                 0.33112695,  0.37523823,  0.64169028, -0.90846333,  0.29780791,
                 2.53172698,  1.41407223,  2.61648461,  1.97081495,  2.34975798,
```

```

3.39687992, 0.51938325, 2.9320051 , 2.31967279, 2.91813423,
1.66193495, 1.80234045, 2.16537886, 1.34459422, 1.5852673 ,
1.90474358, 1.94924878, 3.48876538, 3.79468686, 1.29832982,
2.42816726, 1.19809737, 3.49926548, 1.38766825, 2.27585365,
2.61419383, 1.25762518, 1.29066965, 2.12285398, 2.3875644 ,
2.84096093, 3.2323429 , 2.15873837, 1.4431026 , 1.77964011,
3.07652162, 2.14498686, 1.90486293, 1.16885347, 2.10765373,
2.31430339, 1.92245088, 1.41407223, 2.56332271, 2.41939122,
1.94401705, 1.52566363, 1.76404594, 1.90162908, 1.38966613])

```

By plotting a pair plot of the first and second principal components against each other, we can see that these principal components adequately segregate the data.

```

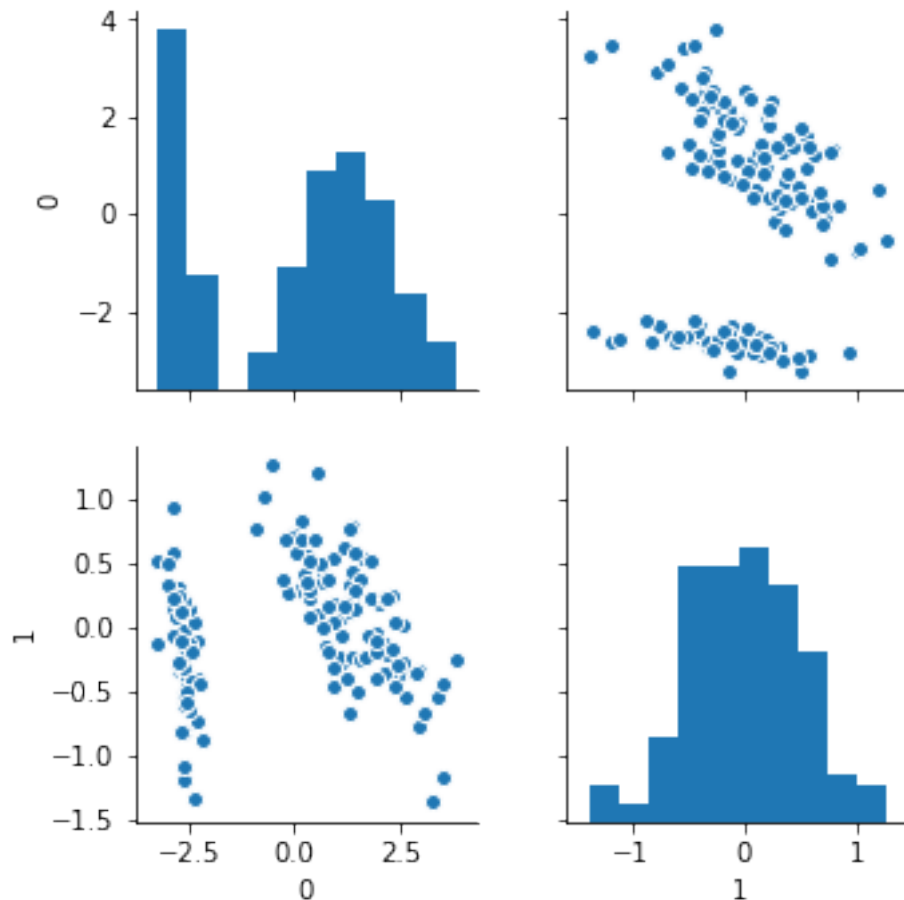
In [51]: # Create a pairplot of each variable to identify if there is the class is easily dist
sns.pairplot(data=pd.DataFrame(K[:,0:2]))

```

```

Out[51]: <seaborn.axisgrid.PairGrid at 0x27e06c76828>

```

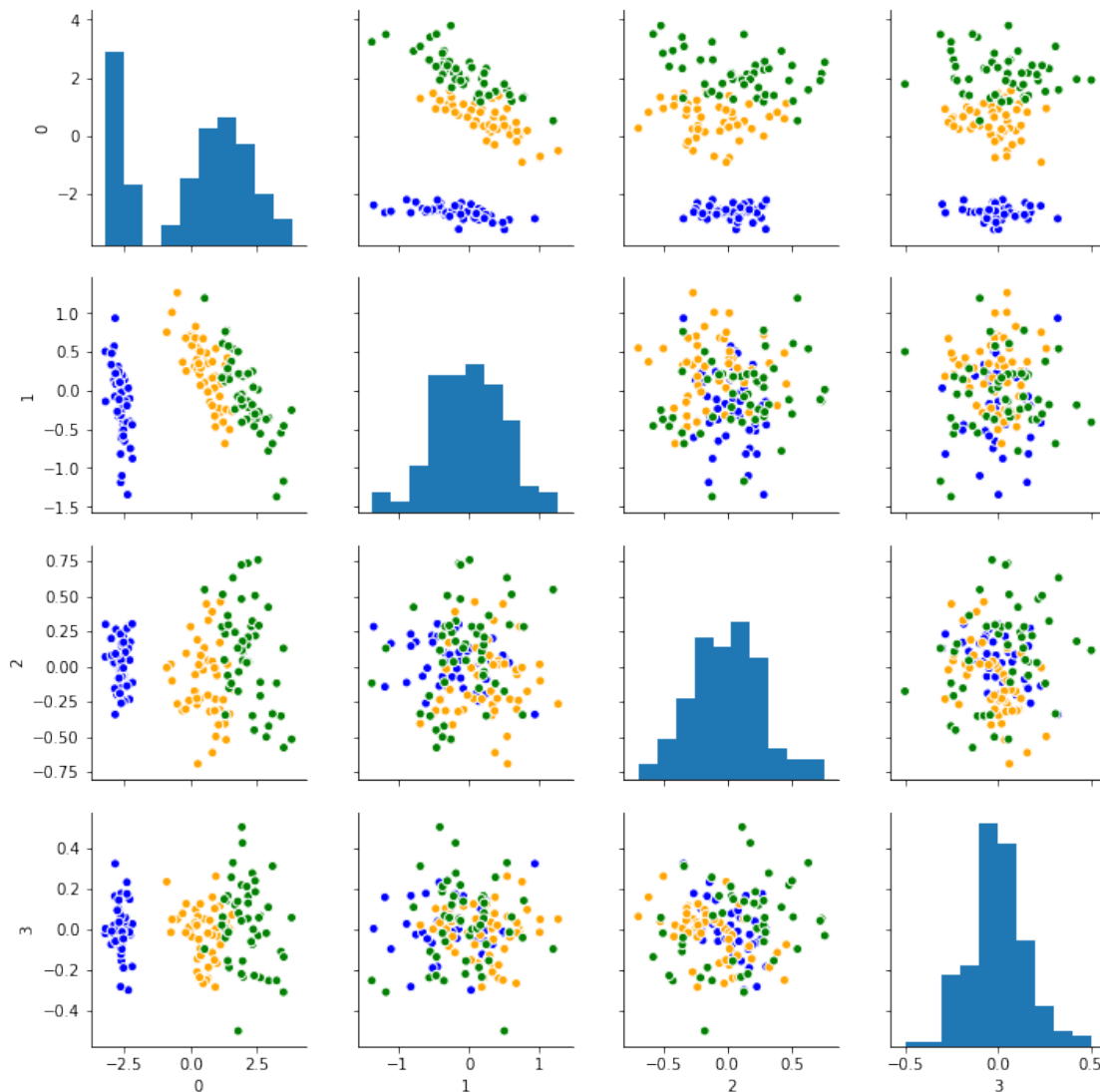


By producing and pairs plot we can find the components showing the most discrimination between the classes. Here we find that the first principal component (principal component 0 in the

plot below) produces the most discrimination followed by the third principal component (principal component 2 in the plot below).

```
In [52]: # Create a pairplot of each principal component to identify if the class is easily di.
sns.pairplot(data=pd.DataFrame(K[:, :]), plot_kws=dict({'color' : color}))
```

```
Out [52]: <seaborn.axisgrid.PairGrid at 0x27e06a1c4a8>
```



```
In [53]: pd.DataFrame(V, columns=['PC1', 'PC2', 'PC3', 'PC4'], index=iris_data.columns[:-1])
```

```
Out [53]:
```

	PC1	PC2	PC3	PC4
sepal length (cm)	0.361590	-0.656540	-0.580997	0.317255
sepal width (cm)	-0.082269	-0.729712	0.596418	-0.324094
petal length (cm)	0.856572	0.175767	0.072524	-0.479719
petal width (cm)	0.358844	0.074706	0.549061	0.751121

The principal components derived using the covariance matrix may give rise to domination of a feature over others. An approach to negate the effects of differing magnitudes amongst features is to use the correlation matrix instead. The relationship between the covariance matrix and the correlation matrix of  $X$  is

$$\text{cor}(X_i, X_j) = \frac{\text{cov}(X_i, X_j)}{\sqrt{\text{cov}(X_i, X_i)\text{cov}(X_j, X_j)}}$$

where  $\text{cov}(X_i, X_i) = \text{var}(X_i)$

```
In [82]: C = np.matrix(X).transpose()
C = pd.DataFrame(np.corrcoef(C))
C

L,E = LA.eig(C)
E = E.transpose()
print('Eigenvectors')
print(E)

print('Eigenvalues')
print(L)

# The diagonal matrix
LI = np.identity(4)

for i in range(4):
    LI[i] = np.dot(LI[i],L[i])

print(LI)

V = E.transpose()
K = np.dot(X,V)

# Create a pairplot of each principal component to identify if the class is easily di
sns.pairplot(data=pd.DataFrame(K[:,:]),plot_kws=dict({'color' : color}))

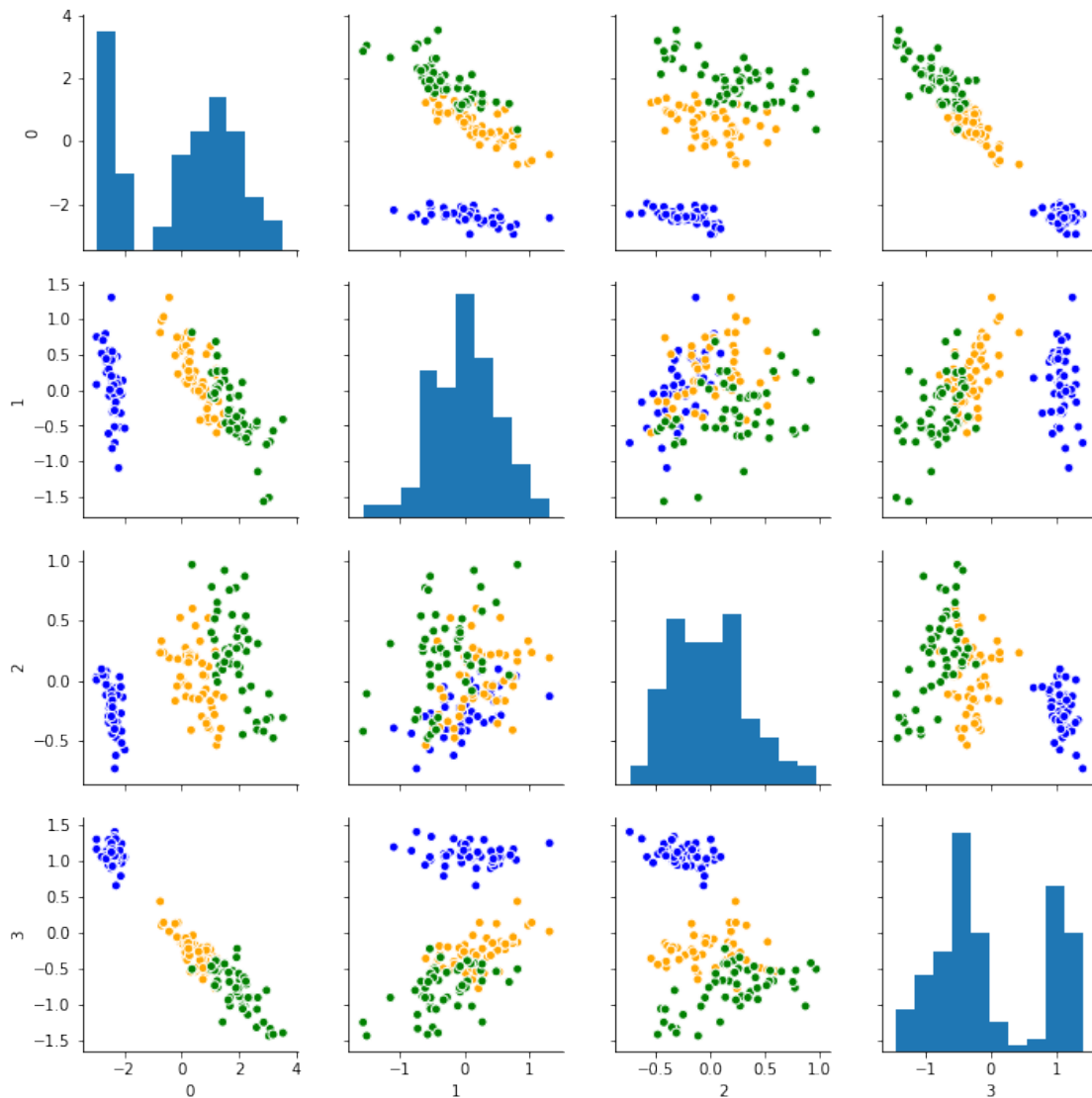
pd.DataFrame(V,columns=['PC1','PC2','PC3','PC4'],index=iris_data.columns[:-1])

Eigenvectors
[[ 0.52237162 -0.26335492  0.58125401  0.56561105]
 [-0.37231836 -0.92555649 -0.02109478 -0.06541577]
 [-0.72101681  0.24203288  0.14089226  0.6338014 ]
 [ 0.26199559 -0.12413481 -0.80115427  0.52354627]]
Eigenvalues
[2.91081808 0.92122093 0.14735328 0.02060771]
[[2.91081808 0.          0.          0.          ]
 [0.          0.92122093 0.          0.          ]
 [0.          0.          0.14735328 0.          ]
```

```
[0.      0.      0.      0.02060771]]
```

Out[82]:

	PC1	PC2	PC3	PC4
sepal length (cm)	0.522372	-0.372318	-0.721017	0.261996
sepal width (cm)	-0.263355	-0.925556	0.242033	-0.124135
petal length (cm)	0.581254	-0.021095	0.140892	-0.801154
petal width (cm)	0.565611	-0.065416	0.633801	0.523546



From the pair plot above, we see that principal components 1 and 4, discriminate between the classes most. Looking at the principal component dataframe above, shows that these two components have a significant contribution from petal length and petal width. This is not surprising since in our initial pair plot, these two features showed the most discrimination.



```
In [71]: from sklearn.decomposition import PCA
pca = PCA(n_components=4)
pca.fit(X)
```

```
Out[71]: PCA(copy=True, iterated_power='auto', n_components=4, random_state=None,
          svd_solver='auto', tol=0.0, whiten=False)
```

```
In [72]: pca.components_.transpose()
```

```
Out[72]: array([[ 0.36158968,  0.65653988, -0.58099728,  0.31725455],
                [-0.08226889,  0.72971237,  0.59641809, -0.32409435],
                [ 0.85657211, -0.1757674 ,  0.07252408, -0.47971899],
                [ 0.35884393, -0.07470647,  0.54906091,  0.75112056]])
```

```
In [73]: pd.DataFrame(pca.components_.transpose(), columns=['PC1', 'PC2', 'PC3', 'PC4'], index=iris)
```

```
Out[73]:
```

	PC1	PC2	PC3	PC4
sepal length (cm)	0.361590	0.656540	-0.580997	0.317255
sepal width (cm)	-0.082269	0.729712	0.596418	-0.324094
petal length (cm)	0.856572	-0.175767	0.072524	-0.479719
petal width (cm)	0.358844	-0.074706	0.549061	0.751121

The principal components obtained using `sklearn.decomposition.PCA` matches our results. Note that if  $\alpha$  is a vector satisfying  $X\alpha = \lambda\alpha$  for some  $\lambda$ , then the following is also true:

$$X(-\alpha) = \lambda(-\alpha)$$

Hence, if  $\alpha$  is an eigenvector, so is  $-\alpha$ . Additionally, if  $\alpha$  and  $\beta$  are two orthonormal vectors, then so are  $-\alpha$  and  $\beta$ . This explains why some of the principal vectors obtained via `sklearn.decomposition.PCA` differs from ours only by sign.

Below, we see that after fitting the PCA method, it is possible to retrieve the covariance matrix which matches ours.

```
In [81]: print(pca.get_covariance())
Z
```

```
[[ 0.68569351 -0.03926846  1.27368233  0.5169038 ]
 [-0.03926846  0.18800403 -0.32171275 -0.11798121]
 [ 1.27368233 -0.32171275  3.11317942  1.29638747]
 [ 0.5169038  -0.11798121  1.29638747  0.58241432]]
```

```
Out[81]:
```

	0	1	2	3
0	0.685694	-0.039268	1.273682	0.516904
1	-0.039268	0.188004	-0.321713	-0.117981
2	1.273682	-0.321713	3.113179	1.296387
3	0.516904	-0.117981	1.296387	0.582414

#### 1.0.4 A. Appendix

**A1. The Variance of a Linear Combination of Random Variables** Let  $\mathbf{X} = (X_1, X_2, \dots, X_p)^T$  be a column vector of random variables,  $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_p)^T$  be a column vector of constants (i.e.  $\in \mathbb{R}^p$ ) and  $P = \alpha_1 X_1 + \alpha_2 X_2 + \dots + \alpha_p X_p = \sum_{i=1}^p \alpha_i X_i$ . Then

$$E[P] = E\left[\sum_{i=1}^p \alpha_i X_i\right] = \sum_{i=1}^p \alpha_i E[X_i]$$

$$(E[P])^2 = \sum_{i=1}^p \alpha_i E[X_i] \sum_{i=1}^p \alpha_i E[X_i] = \sum_{i=1}^p \alpha_i^2 E[X_i]^2 + \sum_{i \neq j, i, j \in \{1, 2, \dots, p\}} \alpha_i \alpha_j E[X_i] E[X_j]$$

$$E[P^2] = E\left[\left(\sum_{i=1}^p \alpha_i X_i\right)^2\right] = E\left[\sum_{i=1}^p \alpha_i^2 X_i^2 + \sum_{i \neq j} \alpha_i \alpha_j X_i X_j\right] = \sum_{i=1}^p \alpha_i^2 E[X_i^2] + \sum_{i \neq j} \alpha_i \alpha_j E[X_i X_j]$$

Putting these together

$$Var[P] = E[P^2] - E[P]^2 = \sum_{i=1}^p \alpha_i^2 (E[X_i^2] - E[X_i]^2) + \sum_{i \neq j} \alpha_i \alpha_j (E[X_i X_j] - E[X_i] E[X_j]) = \sum_{i=1}^p \alpha_i^2 Var[X_i] + \sum_{i \neq j} \alpha_i \alpha_j Cov[X_i, X_j]$$

where  $\Sigma_{ij} = Cov(X_i, X_j)$  is the  $p \times p$  covariance matrix of  $\mathbf{X}$ .