PCA1

August 26, 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 or 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 0x229bf443b70>
```

1.0.3 3- Principal Component Analysis (PCA)

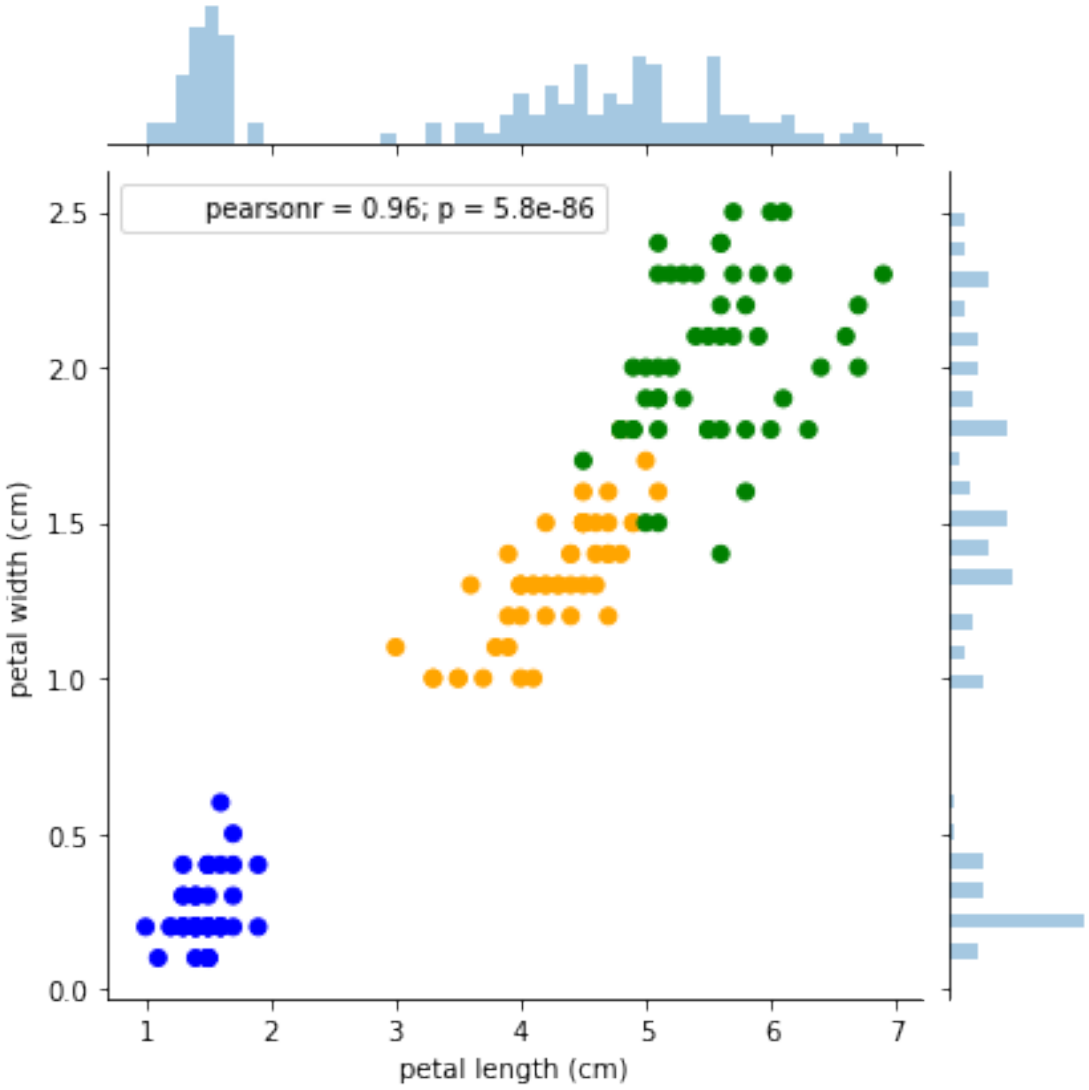
The idea behind PCA is the determination of orthonormal dimensions such that each dimension captures an axis with approximately maximum variance. For example in a simplified plot below of petal length, petal width and class, the maximum variance is obtained approximately along the petal width = petal 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_ku
```

```
Out[6]: <seaborn.axisgrid.JointGrid at 0x229c40a60f0>
```



Let there be p variables in our dataset with $\mathbf{X} = (\mathbf{X_1}, \mathbf{X_2}, ..., \mathbf{X_p})^\mathbf{T}$ a column vector of random variables representing the variables of the dataset. A principal component, P_1, then has the form

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

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

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

where $\sum$ 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 $Var[P_1]$ in terms of α_i. i.e.

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

The initial point here is that P_1 is just a linear combination of the variables, so nothing stops the unbounded variables, α_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 $Var(P_1)$, as it is currently written, we will need to know the covariance matrix $\sum$. 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 $\boldsymbol{X}$ is

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

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

Sometimes $\boldsymbol{S}$ is expressed as

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

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

We therefore need to maximise

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

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

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

Then

$$\nabla \mathcal{L}(\boldsymbol{\alpha}_1, \lambda_1) = 2\boldsymbol{S}\boldsymbol{\alpha}_1 - 2\lambda_1\boldsymbol{\alpha}_1 = 0$$

Resulting in the eigenvalue equation

$$\boldsymbol{S}\alpha_1 = \lambda_1\boldsymbol{\alpha}_1 \implies \lambda_1 \text{is a solution to } \det(\boldsymbol{S} - \lambda\boldsymbol{I}) = 0$$

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 λ_1 is related to $Var(P_1)$. This can be seen by multiplying the eigenvalue equation above on both sides by $\boldsymbol{\alpha}_1$ to obtain $Var(P_1)$

$$\boldsymbol{\alpha}_1^T\boldsymbol{S}\boldsymbol{\alpha}_1 = \lambda_1$$

We first calculate $\boldsymbol{S}$ 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 [8]: # The Iris dataset
    X = iris_data
```

```
In [8]: # Calculate the covariance matrix for our dataset
        Z = CalcCovarMatrix(X)
        Z = pd.DataFrame(Z)

        Z
```

Out[8]:

	0	1	2	3	4
0	0.685694	-0.039268	1.273682	0.516904	0.530872
1	-0.039268	0.188004	-0.321713	-0.117981	-0.148993
2	1.273682	-0.321713	3.113179	1.296387	1.371812
3	0.516904	-0.117981	1.296387	0.582414	0.597987
4	0.530872	-0.148993	1.371812	0.597987	0.671141

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

Out[9]:

	0	1	2	3	4
0	0.685694	-0.039268	1.273682	0.516904	0.530872
1	-0.039268	0.188004	-0.321713	-0.117981	-0.148993
2	1.273682	-0.321713	3.113179	1.296387	1.371812
3	0.516904	-0.117981	1.296387	0.582414	0.597987
4	0.530872	-0.148993	1.371812	0.597987	0.671141

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

The eigenvalue is

```
In [11]: print(l_1)
```

```
4.835034736076369
```

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

```
In [12]: a_1
```

```
Out[12]: array([ 0.33418606, -0.07826103,  0.80046289,  0.33708273,  0.35753221])
```

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

$$\boldsymbol{S}\boldsymbol{a}_1 = l_1\boldsymbol{a}_1$$

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([-4.44089210e-16,  5.55111512e-17, -4.44089210e-16, -4.44089210e-16,
                 0.00000000e+00])
```

We now need to find the *second principal component*

$$P_2 = \boldsymbol{\alpha}_2^T \boldsymbol{X}$$

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.

$$\boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_2 = 0$$

This constraint along with the constraint analoguous to the previous case

$$\boldsymbol{\alpha}_2^T \boldsymbol{\alpha}_2 = 1$$

allows the discovery of an eigenvector that is different from $\boldsymbol{\alpha}_1$
This results in the lagrange equation (maximising this corresponds to minimising $\boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_2$)

$$\mathcal{L}(\boldsymbol{\alpha}_2, \lambda_2, r) = f(\boldsymbol{\alpha}_2) - \lambda_2 g(\boldsymbol{\alpha}_2) - r\boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_2$$

Resulting in

$$\nabla \mathcal{L} = 2S\alpha_2 - 2\lambda_2 \boldsymbol{\alpha}_2 - r\boldsymbol{\alpha}_1$$

Setting this to 0

$$\boldsymbol{S\alpha}_2 - \lambda_2 \boldsymbol{\alpha}_2 = \frac{1}{2} r \boldsymbol{\alpha}_1$$

multiplying both sides by $\boldsymbol{\alpha}_1^T$

$$\boldsymbol{\alpha}_1^T \boldsymbol{S\alpha}_2 - \lambda_2 \boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_2 = \frac{1}{2} r \boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_1$$

$$\implies \boldsymbol{\alpha}_1^T \boldsymbol{S\alpha}_2 = \frac{1}{2} r$$

where $\boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_2 = 0$ and $\boldsymbol{\alpha}_1^T \boldsymbol{\alpha}_1 = 1$
Remembering that

$$\boldsymbol{S\alpha}_1 = \lambda_1 \boldsymbol{\alpha}_1$$

and multiplying both sides by $\boldsymbol{\alpha}_2^T$

$$\boldsymbol{\alpha}_2^T \boldsymbol{S\alpha}_1 = \lambda_1 \boldsymbol{\alpha}_2^T \boldsymbol{\alpha}_1 = 0$$

This means that $r = 0$. Finally,

$$\boldsymbol{S\alpha}_2 = \lambda_2 \boldsymbol{\alpha}_2 \implies \boldsymbol{\alpha}_2^T \boldsymbol{S\alpha}_2 = \lambda_2$$

meaning that λ_2 is related to the variance corresponding to the *second principal component* and must be the second largets in magnitude.

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

```
In [14]: from numpy import linalg as LA
```

```
In [64]: L,E = LA.eig(Z)
         E = E.transpose()
         print('Eigenvectors')
         print(E)

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

```
Eigenvectors
[[-0.33418606  0.07826103 -0.80046289 -0.33708273 -0.35753221]
 [ 0.68858026  0.68410307 -0.09876295 -0.06819839 -0.2084588 ]
 [ 0.44123572 -0.61107009  0.14879435 -0.4238525  -0.47970166]
 [ 0.43143341 -0.30096245 -0.48845415 -0.07214371  0.69245552]
 [-0.18262847  0.24878332  0.29795609 -0.83478653  0.34511928]]
Eigenvalues
[4.83503474 0.25208634 0.09589794 0.03668754 0.02072566]
```

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

$$\boldsymbol{EZ} = \boldsymbol{\Lambda E}^T$$

where $\boldsymbol{E} = (\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, ..., \boldsymbol{\alpha}_p)^T$ is the matrix of eigenvectors and $\boldsymbol{\Lambda}$ is a diagonal matrix where Λ_{ii} is the ith eigenvalue.

```
In [56]: # The diagonal matrix
         LI = np.identity(5)

         for i in range(5):
             LI[i] = np.dot(LI[i],L[i])

         LI
```

```
Out[56]: array([[4.83503474, 0.        , 0.        , 0.        , 0.        ],
                [0.        , 0.25208634, 0.        , 0.        , 0.        ],
                [0.        , 0.        , 0.09589794, 0.        , 0.        ],
                [0.        , 0.        , 0.        , 0.03668754, 0.        ],
                [0.        , 0.        , 0.        , 0.        , 0.02072566]])
```

We can see below that we have approximate equality

```
In [67]: print(np.subtract(np.dot(E,Z),np.dot(LI,E)))
```

```
[[-2.22044605e-16 -6.10622664e-16 -8.88178420e-16 -2.22044605e-16
  -1.11022302e-15]
 [ 5.82867088e-16 -2.22044605e-16 -5.55111512e-17  1.45716772e-16
  -1.24900090e-16]
```

```
[ 1.38777878e-17 -1.04083409e-16  2.61943245e-16  0.00000000e+00
  1.52655666e-16]
[ 1.70002901e-16 -1.19695920e-16  5.06539255e-16  1.06251813e-16
  1.28369537e-16]
[ 9.19403442e-17  3.64291930e-17  4.07660017e-17  4.85722573e-17
  4.16333634e-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

$$\boldsymbol{K}_{ij} = \sum_{l=1}^{5} X_{il} V_{lj}$$

where $\boldsymbol{V} = \boldsymbol{\alpha}^T = (\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, ..., \boldsymbol{\alpha}_5)$ and $\boldsymbol{X}$ is the $n \times p$ data matrix with n observations in $p = 5$ variables.

$\boldsymbol{K}$ is called the *scores matrix* where K_{ij} is the value of the jth principal component of the ith observation and $\boldsymbol{V}$ is the *loadings matrix*. $\boldsymbol{K}$ 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 [72]: V = E.transpose()
         K = np.dot(X,V)
```

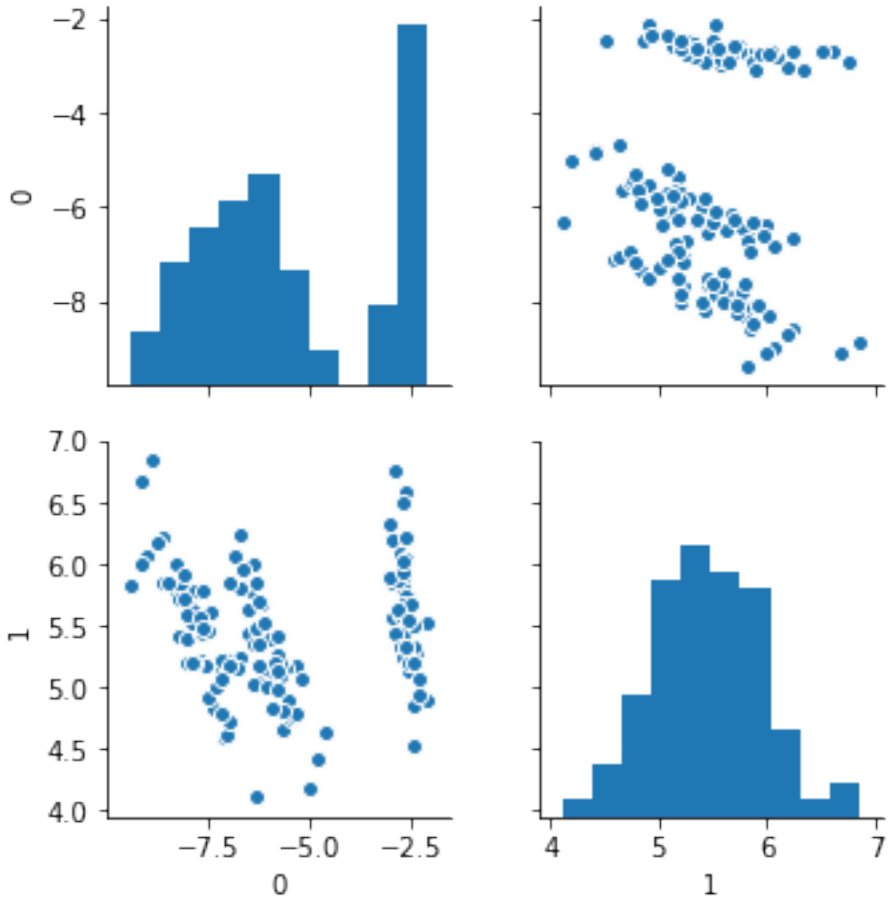
```
In [85]: # This is the first principal component for all 150 observations
         K[:,0]
```

```
Out[85]: array([-2.61849988, -2.59079318, -2.42825747, -2.56275755, -2.57725517,
                -2.99500669, -2.49294122, -2.67295366, -2.43152626, -2.62930509,
                -2.78314978, -2.68616274, -2.5236663 , -2.11643441, -2.65320702,
                -2.89603942, -2.67482154, -2.65220815, -3.06938034, -2.70877613,
                -2.96672066, -2.75031051, -2.12339559, -2.97541577, -2.92630161,
                -2.78430436, -2.8204165 , -2.73196477, -2.65974459, -2.66839634,
                -2.70964105, -2.87404463, -2.65129988, -2.69739158, -2.62930509,
                -2.448467  , -2.67212801, -2.62930509, -2.34365386, -2.70637227,
                -2.53874325, -2.46556346, -2.32800166, -2.88000694, -3.06266956,
                -2.59108285, -2.75511415, -2.47488516, -2.74973117, -2.60073348,
                -6.68049072, -6.35359478, -6.84869907, -5.65561427, -6.49836409,
                -6.08355241, -6.50615093, -4.78582769, -6.45654005, -5.47771602,
                -5.01064329, -5.9620151 , -5.72940858, -6.40320158, -5.3218911 ,
```

```
-6.34792214, -6.10189814, -5.70348714, -6.3650186 , -5.52591783,
-6.52776544, -5.81699539, -6.69514406, -6.34361114, -6.14956397,
-6.32232964, -6.72500421, -6.9371508 , -6.24339867, -5.19761691,
-5.42027904, -5.30652448, -5.61081111, -6.77303688, -6.03506093,
-6.23797643, -6.62176928, -6.24314827, -5.71429644, -5.63996206,
-5.91861284, -6.31532919, -5.6986835 , -4.8270724 , -5.81782104,
-5.79405306, -5.83558744, -6.08272676, -4.6384082 , -5.76336725,
-8.20765936, -7.16485671, -8.28360713, -7.6828207 , -8.03675748,
-9.01102418, -6.33204718, -8.57733078, -8.00789211, -8.56499479,
-7.3933647 , -7.52546092, -7.86316616, -7.10075229, -7.32557197,
-7.62116349, -7.66178552, -9.09558852, -9.38330261, -7.05594505,
-8.10844168, -6.96380909, -9.10643301, -7.13814888, -7.96636182,
-8.280295  , -7.01685788, -7.04783336, -7.82519023, -8.06843808,
-8.49219119, -8.85487032, -7.8588985 , -7.18929054, -7.50462871,
-8.71162789, -7.84593982, -7.62054081, -6.93436847, -7.80871237,
-8.00309256, -7.63599005, -7.16485671, -8.23511565, -8.10119491,
-7.65702523, -7.26755565, -7.4890632 , -7.61872037, -7.14108873])
```

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.

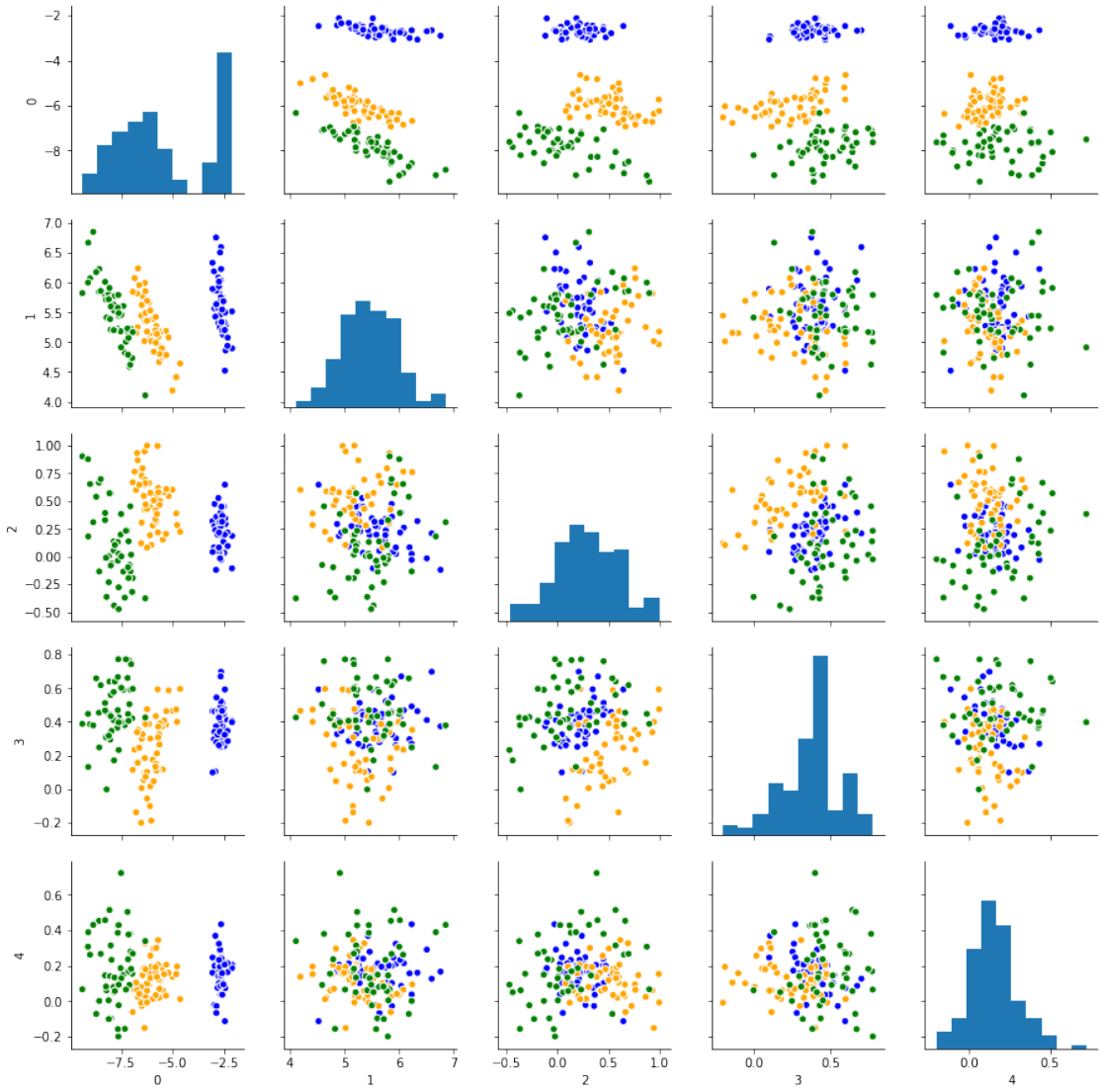
```
In [109]: # Create a pairplot of each variable to identify if there is the class is easily dis
          sns.pairplot(data=pd.DataFrame(K[:,0:2]))

Out[109]: <seaborn.axisgrid.PairGrid at 0x229cf9f5c88>
```



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

```
Out[110]: <seaborn.axisgrid.PairGrid at 0x229cfeabd30>
```



```
In [19]: from sklearn.decomposition import PCA
         pca = PCA(n_components=5)
         pca.fit(Z)
```

```
Out[19]: PCA(copy=True, iterated_power='auto', n_components=5, random_state=None,
           svd_solver='auto', tol=0.0, whiten=False)
```

```
In [111]: pca.components_.transpose()
```

```
Out[111]: array([[ 0.30772997,  0.79830468,  0.47271805,  0.07190551,  0.1984418 ],
                 [-0.11986436,  0.42781019, -0.73444157, -0.3748819 ,  0.35024293],
                 [ 0.80759875,  0.0083804 , -0.32507889, -0.06712611, -0.48737237],
                 [ 0.33259464, -0.20655172, -0.22155939,  0.65710285,  0.60485078],
                 [ 0.35791042, -0.37007328,  0.28699891, -0.64653158,  0.48432886]])
```

1.0.4 A. Appendix

A1. The Variance of a Linear Combination of Random Variables Let $\boldsymbol{X} = (\boldsymbol{X}_1, \boldsymbol{X}_2, ..., \boldsymbol{X}_p)^T$ be a column vector of random variables, $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, ..., \alpha_p)T$ be a column vector of constants (i.e. $\in R^p$) and $P = \alpha_1\boldsymbol{X}_1 + \alpha_2\boldsymbol{X}_2 + ... + \alpha_p\boldsymbol{X}_p = \sum_{i=1}^{p} \alpha_i\boldsymbol{X}_i$. Then

$$\mathrm{E}[P] = \mathrm{E}[\sum_{i=1}^{p} \alpha_i\boldsymbol{X}_i] = \sum_{i=1}^{p} \alpha_i\mathrm{E}[\boldsymbol{X}_i]$$

$$(\mathrm{E}[P])^2 = \sum_{i=1}^{p} \alpha_i\mathrm{E}[\boldsymbol{X}_i] \sum_{i=1}^{p} \alpha_i\mathrm{E}[\boldsymbol{X}_i] = \sum_{i=1}^{p} \alpha_i^2\mathrm{E}[\boldsymbol{X}_i]^2 + \sum_{i\neq j, i,j\in\{1,2,..,p\}}^{p} \alpha_i\alpha_j\mathrm{E}[\boldsymbol{X}_i]\mathrm{E}[\boldsymbol{X}_j]$$

$$\mathrm{E}[P^2] = \mathrm{E}[(\sum_{i=1}^{p} \alpha_i\boldsymbol{X}_i)^2] = \mathrm{E}[\sum_{i=1}^{p} \alpha_i^2\boldsymbol{X}_i^2 + \sum_{i\neq j}^{p} \alpha_i\alpha_j\boldsymbol{X}_i\boldsymbol{X}_j] = \sum_{i=1}^{p} \alpha_i^2\mathrm{E}[\boldsymbol{X}_i^2] + \sum_{i\neq j}^{p} \alpha_i\alpha_j\mathrm{E}[\boldsymbol{X}_i\boldsymbol{X}_j]$$

Putting these together

$$Var[P] = \mathrm{E}[P^2] - \mathrm{E}[P]^2 = \sum_{i=1}^{p} \alpha_i^2(\mathrm{E}[\boldsymbol{X}_i^2] - \mathrm{E}[\boldsymbol{X}_i]^2) + \sum_{i\neq j}^{p} \alpha_i\alpha_j(\mathrm{E}[\boldsymbol{X}_i\boldsymbol{X}_j] - \mathrm{E}[\boldsymbol{X}_i]\mathrm{E}[\boldsymbol{X}_j]) = \sum_{i=1}^{p} \alpha_i^2 Var[\boldsymbol{X}_i] + \sum_{i\neq j}^{p} \alpha_i\alpha_j Cov[\boldsymbol{X}_i,$$

where $\sum_{ij} = Cov(\boldsymbol{X}_i, \boldsymbol{X}_j)$ is the $p \times p$ covariance matrix of $\boldsymbol{X}$.