Análisis de las Componentes Principales ACP

Visualización IRIS PLOTLY

In [1]:
```
import numpy as np
import pandas as pd
import plotly as py
import plotly.graph_objs as go
from plotly.offline import init_notebook_mode, iplot
init_notebook_mode(connected=True)
import matplotlib.pyplot as plt

import os
```

In [2]:
```
data = pd.read_csv("../../Data-Sets/datasets/iris/iris.csv")
```

In [3]:
```
data.head()
```

Out[3]:

	sepal.length	sepal.width	petal.length	petal.width	variety
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
2	4.7	3.2	1.3	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
4	5.0	3.6	1.4	0.2	Setosa

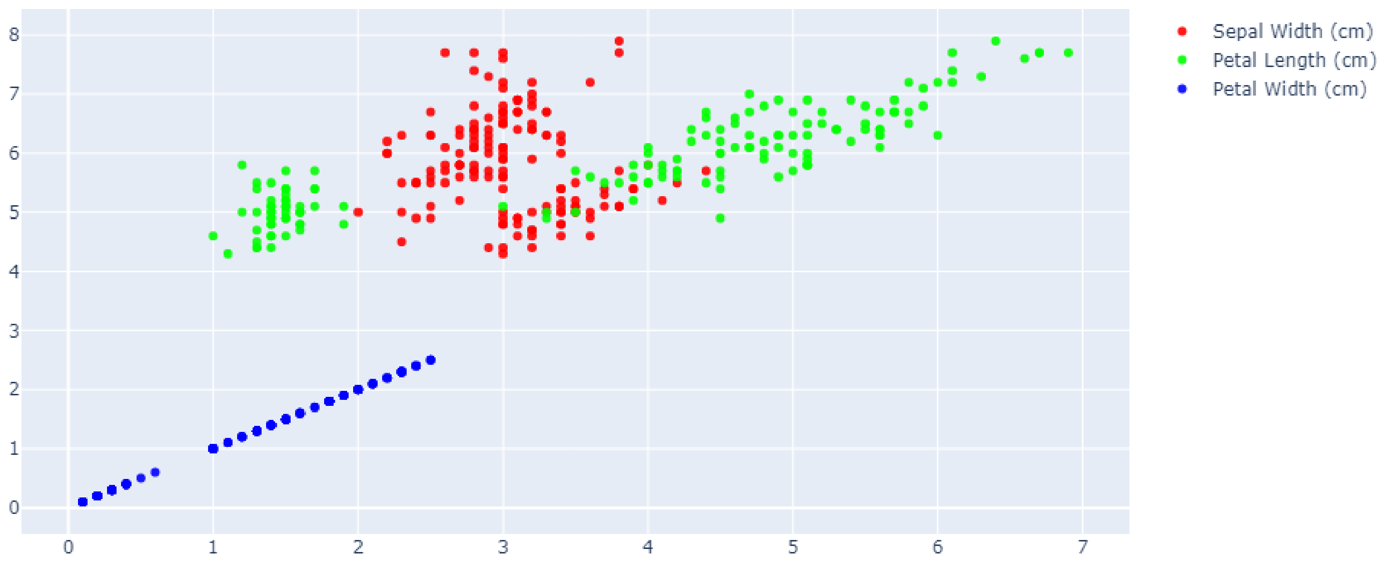
In [4]:
```
data.columns=["Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width", "Species"]
```

In [5]:
```
data_sorted_bySW = data.sort_values('Sepal_Width')
data_sorted_byPL = data.sort_values('Petal_Length')
data_sorted_byPW = data.sort_values('Petal_Width')
```

Scatter Plot

Scatter plot is a good way to visualize the correlations among features. I will be examining the correlation of SepalLengthCm with other features. So Sepal Length will be our y-axis, others will be laying on the x-axis. And I sorted and kept them in distinct dataframes to see correlations clearly.

In [6]:
```
df = data.iloc[:100, :]

bySW = go.Scatter(
                    x = data_sorted_bySW.Sepal_Width,
                    y = data_sorted_bySW.Sepal_Length,
                    mode = "markers",
                    name = "Sepal Width (cm)",
                    marker = dict(color = 'rgba(255, 0, 0, 0.9)'),
                    text = data_sorted_bySW.Species
)

byPL = go.Scatter(
                    x = data_sorted_byPL.Petal_Length,
                    y = data_sorted_byPL.Sepal_Length,
                    mode = "markers",
                    name = "Petal Length (cm)",
                    marker = dict(color = 'rgba(0, 255, 0, 0.9)'),
                    text = data_sorted_byPL.Species
)

byPW = go.Scatter(
                    x = data_sorted_byPW.Petal_Width,
                    y = data_sorted_byPW.Petal_Width,
                    mode = "markers",
                    name = "Petal Width (cm)",
                    marker = dict(color = 'rgba(0, 0, 255, 0.9)'),
                    text = data_sorted_byPW.Species
)

layout = dict(title = 'Change of Sepal Length by Other Properties',
              xaxis= dict(title= 'centimeters',ticklen= 5,zeroline= False)
             )
u = [bySW, byPL, byPW]
fig = dict(data = u)
iplot(fig)
```

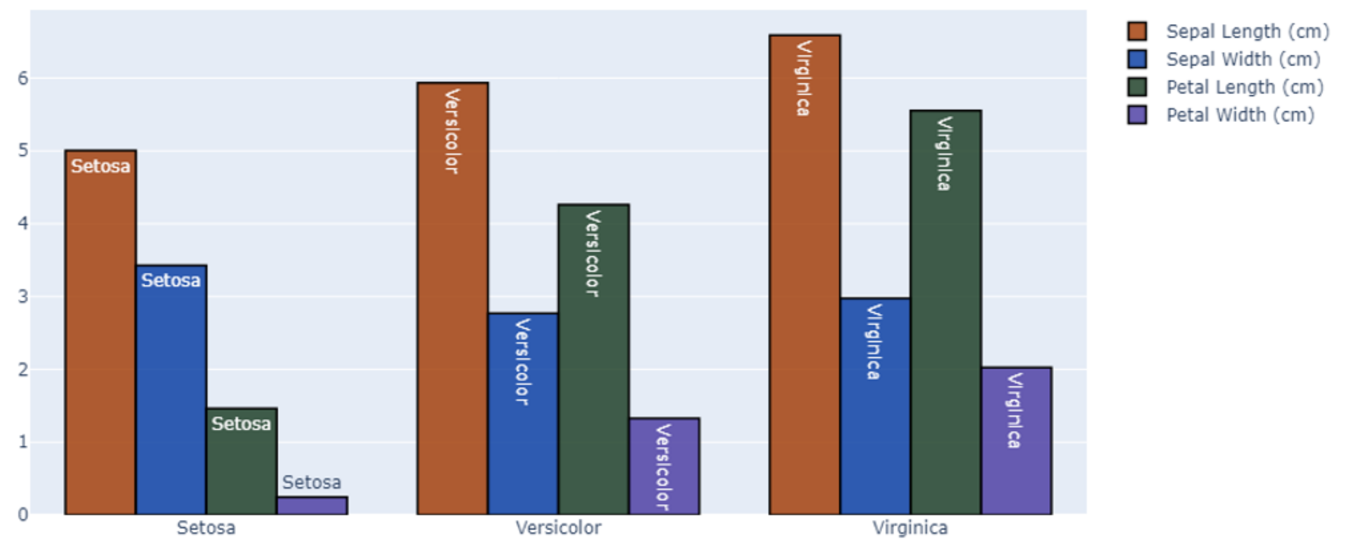


- Seems like Petal Width and Sepal Length has a very strong correlation.
- We can say there is a correlation between Petal Length and Sepal Length, but not like the one above.
- There is no correlation between the Sepal Length and Sepal Width.

Bar Plot

Let's visualize each species' average lengths, so we will be able to see how features change as genre of the flower changes.

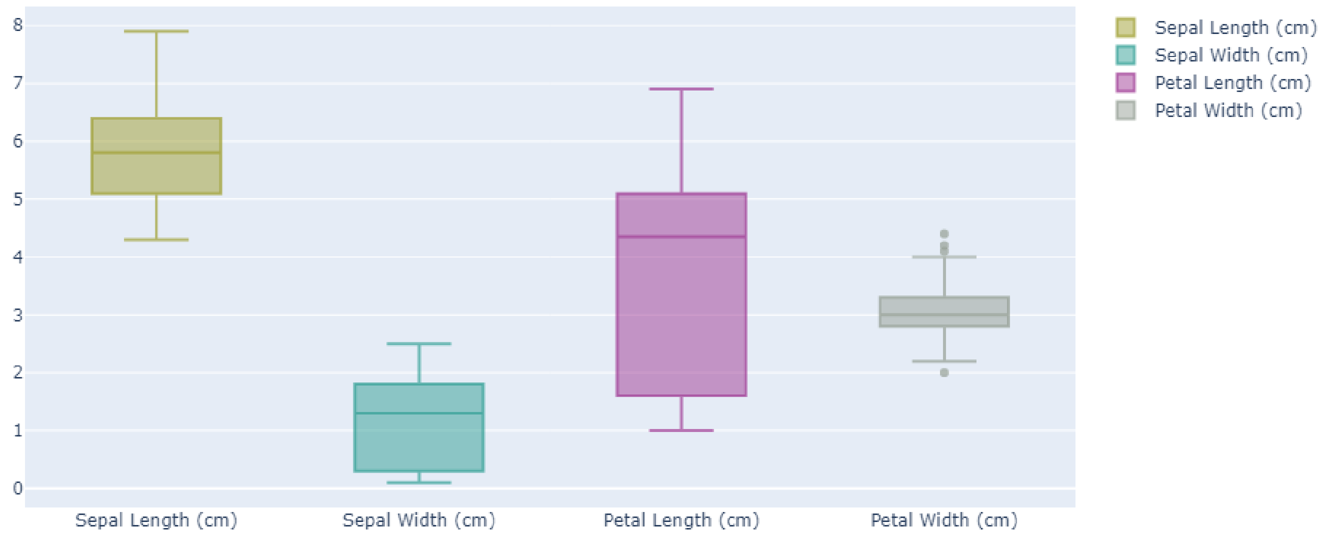
```
In [7]:
data1 = data.groupby(data.Species).mean()
data1['Species'] = data1.index

t1 = go.Bar(
            x = data1.Species,
            y = data1.Sepal_Length,
            name = "Sepal Length (cm)",
            marker = dict(color = 'rgba(160, 55, 0, 0.8)', line = dict(color = 'rgb(0,0,0)',
            text = data1.Species
)

t2 = go.Bar(
            x = data1.Species,
            y = data1.Sepal_Width,
            name = "Sepal Width (cm)",
            marker = dict(color = 'rgba(0, 55, 160, 0.8)', line = dict(color = 'rgb(0,0,0)',
            text = data1.Species
)

t3 = go.Bar(
            x = data1.Species,
            y = data1.Petal_Length,
            name = "Petal Length (cm)",
            marker = dict(color = 'rgba(20, 55, 30, 0.8)', line = dict(color = 'rgb(0,0,0)',
            text = data1.Species
)

t4 = go.Bar(
            x = data1.Species,
            y = data1.Petal_Width,
            name = "Petal Width (cm)",
            marker = dict(color = 'rgba(70, 55, 160, 0.8)', line = dict(color = 'rgb(0,0,0)'
            text = data1.Species
)

b = [t1,t2,t3,t4]
layout_bar = go.Layout(barmode = "group")
fig_bar = go.Figure(data = b, layout = layout_bar)
iplot(fig_bar)
```



Boxplot

Boxplot is always the best choice, if we want to get some statistical information from the data.

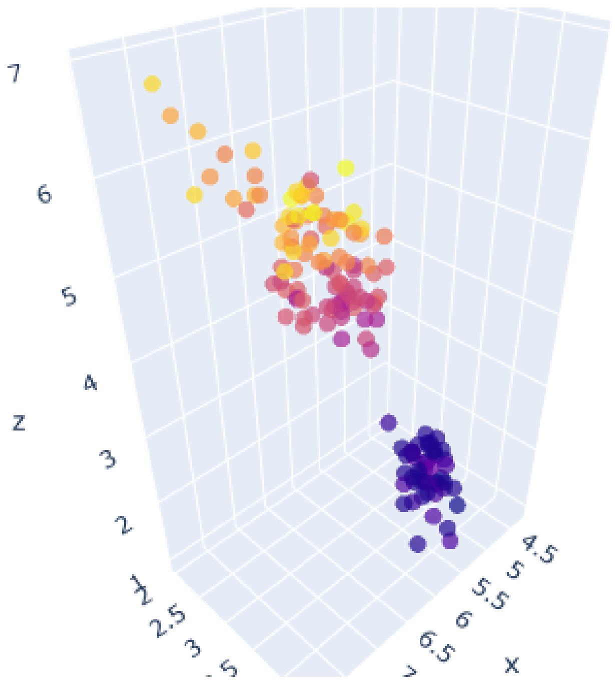
In [8]:
```
t1_box = go.Box(
                name = 'Sepal Length (cm)',
                y = data.Sepal_Length,
                marker = dict(color = 'rgba(160,160,50,0.7)')
)

t2_box = go.Box(
                name = 'Sepal Width (cm)',
                y = data.Petal_Width,
                marker = dict(color = 'rgba(50,160,150,0.7)')
)

t3_box = go.Box(
                name = 'Petal Length (cm)',
                y = data.Petal_Length,
                marker = dict(color = 'rgba(160,60,150,0.7)')
)

t4_box = go.Box(
                name = 'Petal Width (cm)',
                y = data.Sepal_Width,
                marker = dict(color = 'rgba(150,160,150,0.7)')
)

fig_box = [t1_box, t2_box, t3_box, t4_box]

iplot(fig_box)
```



3D Scatter

Let's jump into third dimension, and decide what each variable correspond to: x : SepalLengthCm y : SepalWidthCm z : PetalLengthCm color : PetalWidthCm

In [9]:
```
trace_3d = go.Scatter3d(
                        x = data.Sepal_Length,
                        y = data.Sepal_Width,
                        z = data.Petal_Length,
                        mode = 'markers',
                        opacity = 0.7,
                        #name = data.Species,
                        marker = dict(
                                    size = 5,
                                    color = data.Petal_Width
                        )
)

list_3d = [trace_3d]

fig_3d = go.Figure(data = list_3d)
iplot(fig_3d)
```



Since trying to use name parameter with data.Species giving ValueError, I decided to eliminate the color part of 3D graph, and plot them as different traces.

- Iris-setosa : pink
- Iris-versicolor : lime
- Iris-virginica : blue

PROCEDIMIENTO

- Estandarizar los datos (para cada una de las m observaciones)
- Obtener los vectores y valores propios a partir de la matriz de covarianzas o de correlaciones o incluso la técnica de singular vector descomposition.
- Ordenar los valores propios en orden descendente y quedarnos con los *p* que se corresponden a los *p* mayores y así disminuír el número de variables del dataset (p<m)
- Construir la matriz de proyección W a partir de los p vectores propios.
- Transformar el dataset original X a través de W para así obtener datos en el subespacio dimensional de dimensión *p*, que será Y.

In [10]:
```
df = pd.read_csv("../../Data-Sets/datasets/iris/iris.csv")
```

In [11]:
```
X = df.iloc[:,0:4].values
y = df.iloc[:,4].values
```

In [12]:
```
from sklearn.preprocessing import StandardScaler
```

In [13]:
```
X_std = StandardScaler().fit_transform(X) #Centrado en cero
```

1-Calculamos la descomposición de valores y vectores propios

a) Utilizando la Matriz de Covarianzas

$\sigma_{jk} = \frac{1}{n-1}^{m} \sum_{i=1}^{m} (x_{ij} - \overline{x_j})(x_{ik} - \overline{x_k})$

Reduciendo nos queda:

$\sigma_{jk} = \frac{1}{n-1}((X - \overline{x})^T (X - \overline{x}))$ **Matriz de covarianzas**

Vector m dimensional para calcular la matriz de covarianzas.

$\overline{x} = \sum_{i=1}^{n} x_i \in \mathbb{R}^m$

In [14]:
```
mean_vect = np.mean(X_std, axis=0)
mean_vect
```

Out[14]:
```
array([-4.73695157e-16, -7.81597009e-16, -4.26325641e-16, -4.73695157e-16])
```

In [15]:
```
cov_matrix = (X_std - mean_vect).T.dot(X_std - mean_vect)/(X_std.shape[0]-1)
print("La matriz de covarianzas es \n%s" %cov_matrix)
```

```
La matriz de covarianzas es
[[ 1.00671141 -0.11835884  0.87760447  0.82343066]
 [-0.11835884  1.00671141 -0.43131554 -0.36858315]
 [ 0.87760447 -0.43131554  1.00671141  0.96932762]
 [ 0.82343066 -0.36858315  0.96932762  1.00671141]]
```

In [16]:
```
np.cov(X_std.T)
```

Out[16]:
```
array([[ 1.00671141, -0.11835884,  0.87760447,  0.82343066],
       [-0.11835884,  1.00671141, -0.43131554, -0.36858315],
       [ 0.87760447, -0.43131554,  1.00671141,  0.96932762],
       [ 0.82343066, -0.36858315,  0.96932762,  1.00671141]])
```

In [17]:
```
eig_vals, eig_vectors = np.linalg.eig(cov_matrix) #Calculamos los valores y vectores propios
print("Valores propios \n%s" %eig_vals)
print("Vectores propios \n%s" %eig_vectors)
```

```
Valores propios
[2.93808505 0.9201649  0.14774182 0.02085386]
Vectores propios
[[ 0.52106591 -0.37741762 -0.71956635  0.26128628]
 [-0.26934744 -0.92329566  0.24438178 -0.12350962]
 [ 0.5804131  -0.02449161  0.14212637 -0.80144925]
 [ 0.56485654 -0.06694199  0.63427274  0.52359713]]
```

b) Utilizando la Matriz de correlaciones

Se utiliza a veces en el campo de las finanzas, más típicamente. Si el dato ya se ha estandarizado entonces da lo mismo que la matriz de covarianzas, la matriz de correlaciones es una normalización de la matriz de varianzas.

In [18]:
```
corr_matrix = np.corrcoef(X_std.T)
corr_matrix
```

Out[18]:
```
array([[ 1.        , -0.11756978,  0.87175378,  0.81794113],
       [-0.11756978,  1.        , -0.4284401 , -0.36612593],
       [ 0.87175378, -0.4284401 ,  1.        ,  0.96286543],
       [ 0.81794113, -0.36612593,  0.96286543,  1.        ]])
```

In [19]:
```
eig_vals_corr, eig_vectors_corr = np.linalg.eig(corr_matrix)
print("Valores propios \n%s" %eig_vals_corr)
print("Vectores propios \n%s" %eig_vectors_corr)
```

```
Valores propios
[2.91849782 0.91403047 0.14675688 0.02071484]
Vectores propios
[[ 0.52106591 -0.37741762 -0.71956635  0.26128628]
 [-0.26934744 -0.92329566  0.24438178 -0.12350962]
 [ 0.5804131  -0.02449161  0.14212637 -0.80144925]
 [ 0.56485654 -0.06694199  0.63427274  0.52359713]]
```

La matriz de correlaciones tiene siempre en su diagonal **unos** mientras que en la matriz de covarianzas puede ser un valor cercano o nó a uno. Porque la covarianza de una variable en sí misma es la varianza.

c) Singular Value Decomposition

Este método mejora muchísimo la eficacia computacional.

In [20]:
```
u, s, v = np.linalg.svd(X_std.T)
```

In [21]:
```
s #Son otros valores propios que no se parecen en nada.
```

Out[21]:
```
array([20.92306556, 11.7091661 ,  4.69185798,  1.76273239])
```

```
In [22]: v
```

```
Out[22]: array([[ 1.08239531e-01,  9.94577561e-02,  1.12996303e-01, ...,
                 -7.27030413e-02, -6.56112167e-02, -4.59137323e-02],
                [-4.09957970e-02,  5.75731483e-02,  2.92000319e-02, ...,
                 -2.29793601e-02, -8.63643414e-02,  2.07800179e-03],
                [ 2.72186462e-02,  5.00034005e-02, -9.42089147e-03, ...,
                 -3.84023516e-02, -1.98939364e-01, -1.12588405e-01],
                ...,
                [ 5.43380310e-02,  5.12936114e-03,  2.75184277e-02, ...,
                  9.89532683e-01, -1.41206665e-02, -8.30595907e-04],
                [ 1.96438400e-03,  8.48544595e-02,  1.78604309e-01, ...,
                 -1.25488246e-02,  9.52049996e-01, -2.19201906e-02],
                [ 2.46978090e-03,  5.83496936e-03,  1.49419118e-01, ...,
                 -7.17729676e-04, -2.32048811e-02,  9.77300244e-01]])
```

Esta técnica es la que utlilizaría python o R u otros.

2-Las componentes principales

```
In [23]: for ev in eig_vectors:
             print ("La lonigitud del VP es: %s" %np.linalg.norm(ev))
```

```
La lonigitud del VP es: 0.9999999999999997
La lonigitud del VP es: 0.9999999999999999
La lonigitud del VP es: 1.0
La lonigitud del VP es: 1.0000000000000002
```

```
In [24]: eigen_pairs =  [(np.abs(eig_vals[i]), eig_vectors[:,i]) for i in range(len(eig_vals))]
         eigen_pairs
```

```
Out[24]: [(2.9380850501999918,
           array([ 0.52106591, -0.26934744,  0.5804131 ,  0.56485654])),
          (0.9201649041624875,
           array([-0.37741762, -0.92329566, -0.02449161, -0.06694199])),
          (0.14774182104494815,
           array([-0.71956635,  0.24438178,  0.14212637,  0.63427274])),
          (0.020853862176462876,
           array([ 0.26128628, -0.12350962, -0.80144925,  0.52359713]))]
```

Ordenamos los vectores propios con valor propio de mayor a menor

```
In [25]: eigen_pairs.sort()
         eigen_pairs.reverse()
         eigen_pairs
```

```
Out[25]: [(2.9380850501999918,
           array([ 0.52106591, -0.26934744,  0.5804131 ,  0.56485654])),
          (0.9201649041624875,
           array([-0.37741762, -0.92329566, -0.02449161, -0.06694199])),
          (0.14774182104494815,
           array([-0.71956635,  0.24438178,  0.14212637,  0.63427274])),
          (0.020853862176462876,
           array([ 0.26128628, -0.12350962, -0.80144925,  0.52359713]))]
```

```
In [26]: print("Valores propios en orden descendente")
         for ep in eigen_pairs:
             print(ep[0])
```

```
Valores propios en orden descendente
2.9380850501999918
0.9201649041624875
```

```
0.14774182104494815
0.020853862176462876
```

In [27]:
```
total_sum = sum(eig_vals) #suma todas las varianzas
var_exp = [(i/total_sum)*100 for i in sorted(eig_vals, reverse=True)]
```

In [28]:
```
var_exp
```

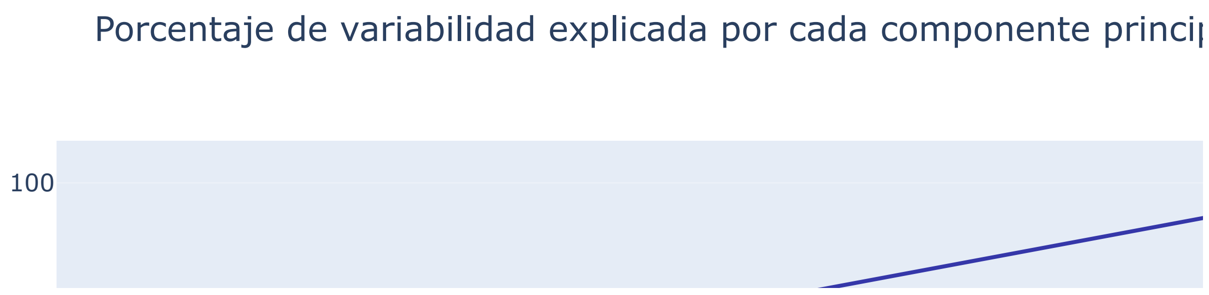
Out[28]:
```
[72.96244541329983, 22.85076178670179, 3.668921889282881, 0.5178709107154951]
```

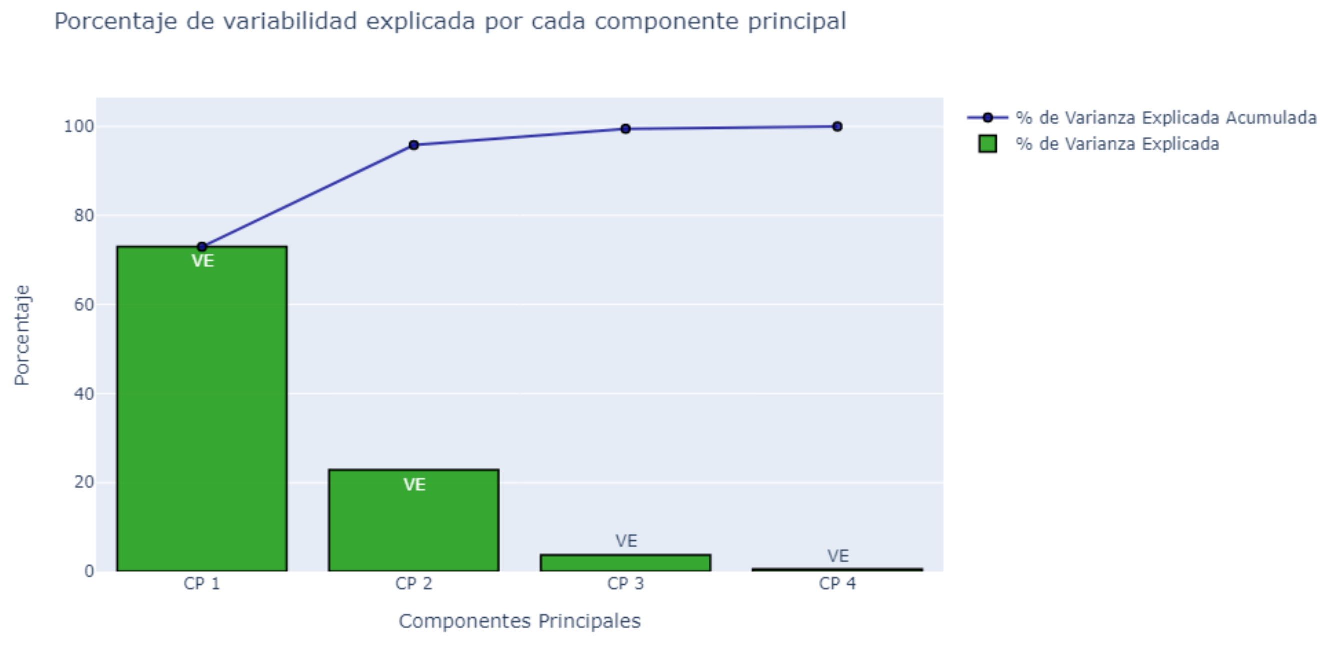
Este array me dice aquí que sólamente quedándome con una dimensión explico casi el 76% de la variabilidad, el segundo un 20% de la variabilidad, el tercero un 3.25 de la variabilidad y la última dimensión no explica casi nada.

In [29]:
```
cum_var_exp = np.cumsum(var_exp)
cum_var_exp # Varianza explicada acumulada, esta va sumando acumulativamente hasta el 100%
```

Out[29]:
```
array([ 72.96244541,  95.8132072 ,  99.48212909, 100.        ])
```

In [30]:
```
t1 = go.Scatter(
            x = ["CP %s" %i for i in range(1,5)],
            y = cum_var_exp,
            name = "% de Varianza Explicada Acumulada",
            marker = dict(color = 'rgba(10, 10, 150, 0.8)', line = dict(color = 'rgb(0,0,0)',
            text = "CP",
            showlegend=True
)
t2 = go.Bar(
            x = ["CP %s" %i for i in range(1,5)],
            y = var_exp,
            name = "% de Varianza Explicada",
            marker = dict(color = 'rgba(10, 150, 0, 0.8)', line = dict(color = 'rgb(0,0,0)',
            text = "VE",
            showlegend=True
)
b = [t1, t2]

layout_bar = go.Layout(barmode = "group")

layout = dict(barmode = "group", title='Porcentaje de variabilidad explicada por cada compone
              xaxis= dict(title= 'Componentes Principales', ticklen= 5, zeroline= False),
              yaxis= dict(title= 'Porcentaje ', ticklen= 5, zeroline= False)
             )

fig_bar = go.Figure(data = b, layout = layout)

iplot(fig_bar)
```





Por lo que vemos vamos a reducir de un espacio de dimensión 4 a uno de dimensión 2, que son los dos primeros vectores propios que aportan más información.

In [31]:
```
print(eigen_pairs)
```

```
[(2.9380850501999918, array([ 0.52106591, -0.26934744,  0.5804131 ,  0.56485654])), (0.920164
9041624875, array([-0.37741762, -0.92329566, -0.02449161, -0.06694199])), (0.1477418210449481
5, array([-0.71956635,  0.24438178,  0.14212637,  0.63427274])), (0.020853862176462876, array
([ 0.26128628, -0.12350962, -0.80144925,  0.52359713]))]
```

In [32]:
```
HOLA=eigen_pairs[0][1].reshape(4,1)
HOLA
```

Out[32]:
```
array([[ 0.52106591],
       [-0.26934744],
       [ 0.5804131 ],
       [ 0.56485654]])
```

In [33]:
```
W = np.hstack((eigen_pairs[0][1].reshape(4,1), #hstack acumula los vectores horizontalmente y
              eigen_pairs[1][1].reshape(4,1))) #ya están en vertical, crea la matriz en hori
W
```

Out[33]:
```
array([[ 0.52106591, -0.37741762],
       [-0.26934744, -0.92329566],
       [ 0.5804131 , -0.02449161],
       [ 0.56485654, -0.06694199]])
```

3- Proyectando las variables en el nuevo subespacio vectorial

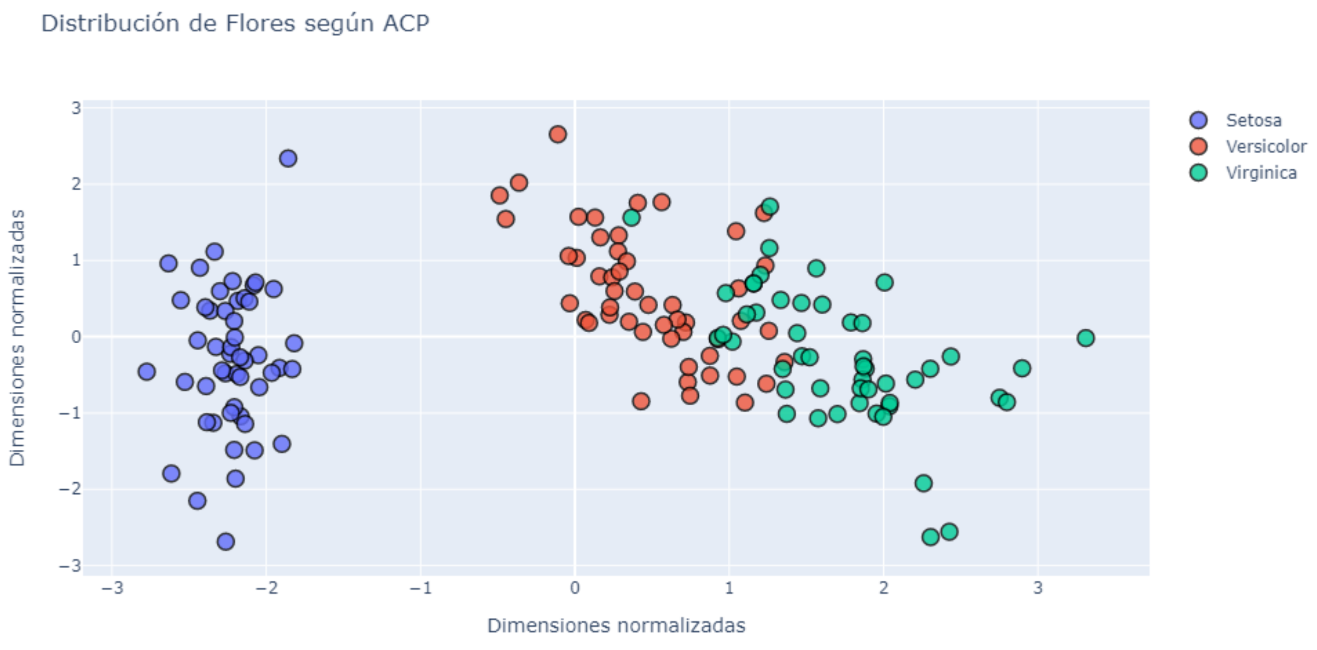
- $Y = X \cdot W$
- $X \in M(\mathbb{R})_{150,4}$
- $W \in M(\mathbb{R})_{4,2}$
- $Y \in M(\mathbb{R})_{150,2}$

El número de filas de la primer matriz queda igual (X) y el número de columnas de la segunda matriz (W) queda igual. Por eso las dimensiones de Y=150x2

In [34]:
```
Y = X_std.dot(W)
```

In [35]:
```python
results = []

for name in ('Setosa', 'Versicolor', 'Virginica'):
    result = go.Scatter(
                x=Y[y==name, 0],
                y=Y[y==name, 1],
                mode="markers",
                name=name,
                marker = dict(size=12, line = dict(color = 'rgb(0,0,0)', width = 1.5), op
                )
    results.append(result)

layout = dict(showlegend=True, title='Distribución de Flores según ACP',
              xaxis= dict(title= 'Dimensiones normalizadas', ticklen= 5, zeroline= True),
              yaxis= dict(title= 'Dimensiones normalizadas', ticklen= 5, zeroline= True)
             )

fig = go.Figure(data = results, layout = layout)

iplot(fig)
```

Distribución de Flores según ACP



Con este método, hemos reducido a dos dimensiones el dataset y a as vez conservamos el 95% de la varianza de los datos, lo cual es todo un éxito.

In [ ]: