EECS 491 Assignment 4

Yue Shu
Spring 2019
Prof. Lewicki

Exercise 1. Multivariate Gaussians

1.1 Consider the 2D normal distribution

$$p(x, y) \sim \mathcal{N}(\mu, \mathbf{\Sigma})$$

Define three separate 2D covariance matrices $\mathbf{\Sigma}$ for each of the following cases: x and y are uncorrelated; x and y are correlated; and x and y are anti-correlated.

The covariance matrix Σ could be defined as below in general:

$$\Sigma = \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{bmatrix}$$

x and y are uncorrelated

Since x and y are uncorrelated, we may conclude that $\Sigma_{xy} = \Sigma_{yx} = 0$, and thus the convariance matrix should be defined as:

$$\Sigma = \begin{bmatrix} var(x) & 0 \\ 0 & var(y) \end{bmatrix}$$

x and y are correlated

Since x and y are positively correlated, we may conclude that $\Sigma_{xy} > 0$, $\Sigma_{yx} > 0$, and thus the convariance matrix should be defined as:

$$\Sigma = \begin{bmatrix} var(x) & \Sigma_{xy} \\ \Sigma_{yx} & var(y) \end{bmatrix}, \Sigma_{xy} > 0, \Sigma_{yx} > 0$$

x and y are anti-correlated

Since x and y are negatively correlated, we may conclude that $\Sigma_{xy} < 0$, $\Sigma_{yx} < 0$, and thus the convariance matrix should be defined as:

$$\Sigma = \begin{bmatrix} var(x) & \Sigma_{xy} \\ \Sigma_{yx} & var(y) \end{bmatrix}, \Sigma_{xy} < 0, \Sigma_{yx} < 0$$

1.2 Compute the principal axes for each of these distributions, i.e. the eigenvectors of the covariance matrices.

The expression of eigenvector and eigenvalue is

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

x and y are uncorrelated

$$\begin{bmatrix} var(x) & 0 \\ 0 & var(y) \end{bmatrix} \mathbf{v} = \lambda\mathbf{v}$$

$$\begin{bmatrix} var(x) & 0 \\ 0 & var(y) \end{bmatrix} \mathbf{v} = \lambda\mathbf{v}$$

$$\begin{bmatrix} var(x) & 0 \\ 0 & var(y) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \lambda v_1 \\ \lambda v_2 \end{bmatrix}$$

$$\begin{bmatrix} var(x)v_1 \\ var(y)v_2 \end{bmatrix} = \begin{bmatrix} \lambda v_1 \\ \lambda v_2 \end{bmatrix}$$

Therefore, if $var(x) \neq var(y)$, we may conclude that the eigenvector does not exist.

However, if $var(x) = var(y)$, which means $\lambda = var(x) = var(y)$, then any vector would suffice.

x and y are correlated/anticorrelated

The steps for positively correlated and negatively correlated x and y should be the same, since the only difference is the sign of the covariance, which is only represented in the actual calculations.

$$\begin{bmatrix} var(x) & \Sigma_{xy} \\ \Sigma_{yx} & var(y) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \lambda v_1 \\ \lambda v_2 \end{bmatrix}$$
$$\begin{bmatrix} var(x)v_1 + \Sigma_{xy}v_2 \\ \Sigma_{yx}v_1 + var(y)v_2 \end{bmatrix} = \begin{bmatrix} \lambda v_1 \\ \lambda v_2 \end{bmatrix}$$
$$\Sigma_{xy}v_2 = (\lambda - var(x)v_1)v_1$$
$$\Sigma_{yx}v_1 = (\lambda - var(y)v_2)v_2$$
$$\Sigma_{xy}v_2 = \lambda v_1 - var(x)v_1^2$$
$$v_2 = \frac{\lambda v_1 - var(x)v_1^2}{\Sigma_{xy}}$$
$$v_1 = \frac{\lambda v_2 - var(y)v_2^2}{\Sigma_{yx}}$$
$$v_1 = \frac{\lambda \frac{\lambda v_1 - var(x)v_1^2}{\Sigma_{xy}} - var(y)(\frac{\lambda v_1 - var(x)v_1^2}{\Sigma_{xy}})^2}{\Sigma_{yx}}$$

After v_1 is solved, we just go back and solve for v_2. I will put a stop here since the process should be pretty trivial and in reality once we've gain the actual covariance matrix it should be pretty straight forward.

The steps of solving for the eigenvalue λ should also be quite simple:

$$\Sigma - \lambda I = \begin{bmatrix} var(x) & \Sigma_{xy} \\ \Sigma_{yx} & var(y) \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} = \begin{bmatrix} var(x) - \lambda & \Sigma_{xy} \\ \Sigma_{yx} & var(y) - \lambda \end{bmatrix}$$

Then we use the determinant to solve for the eigenvalue:

$$var(x)var(y) + \lambda^2 - \lambda var(x) - \lambda var(y) = \Sigma_{xy}\Sigma_{yx}$$
$$\lambda^2 - (var(x) + var(y))\lambda - \Sigma_{xy}\Sigma_{yx} + var(x)var(y) = 0$$
$$\lambda_1 = \frac{var(x) + var(y) + \sqrt{(var(x) - var(y))^2 - 4\Sigma_{xy}\Sigma_{yx}}}{2}$$
$$\lambda_2 = \frac{var(x) + var(y) - \sqrt{(var(x) - var(y))^2 - 4\Sigma_{xy}\Sigma_{yx}}}{2}$$

All we should do next is to plug in the two eigenvalues, check which one is valid, and then solve for the corresponding eigenvector. Once again, I will pause my solution here since solving for the eigenvectors without any actual value is quite trivial, and as long as the steps I listed above are strictly followed, the calculation should be quite simple.

Exercise 2. Linear Gaussian Models (20 pts)

Consider two multi-dimensional Gaussian random vector variables

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mu_{\mathbf{x}}, \mathbf{\Sigma}_x)$$
$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mu_{\mathbf{z}}, \mathbf{\Sigma}_z)$$

Now consider a third variable that is the sum of the first two:

$$\mathbf{y} = \mathbf{x} + \mathbf{z}$$

2.1 What is the expression for the distribution $p(\mathbf{y})$?

First of all, assuming joint normality of $(\mathbf{x}, \mathbf{z})$, then we should have, according to the linear properties of multivariate normal random vectors,

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z} = (\mathbf{A} \;\; \mathbf{B}) \begin{pmatrix} \mathbf{x} \\ \mathbf{z} \end{pmatrix}$$

where both $\mathbf{A}$ and $\mathbf{B}$ are identity matrix $\mathbf{I}$.

And thus

$$p(\mathbf{y}) = p(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{z}) = \mathcal{N}(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} | (\mathbf{A} \;\; \mathbf{B}) \begin{pmatrix} \mu_{\mathbf{x}} \\ \mu_{\mathbf{z}} \end{pmatrix}, (\mathbf{A} \;\; \mathbf{B}) \mathbf{\Sigma}_{\mathbf{x},\mathbf{z}} \begin{pmatrix} \mathbf{A}^{\mathbf{T}} \\ \mathbf{B}^{\mathbf{T}} \end{pmatrix}$$

Then we shall plug in $\mathbf{A} = \mathbf{B} = \mathbf{I}$, and the final expression of $p(\mathbf{y})$ would be:

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{x} + \mathbf{z} | \mu_{\mathbf{x}} + \mu_{\mathbf{z}}, \mathbf{\Sigma}_{\mathbf{xx}} + \mathbf{\Sigma}_{\mathbf{xz}}^{\mathbf{T}} + \mathbf{\Sigma}_{\mathbf{xz}} + \mathbf{\Sigma}_{\mathbf{zz}})$$

2.2 What is the expression for the condidtional distribution $p(\mathbf{y}|\mathbf{x})$?

Through the slids, we know that the expresion of $p(y|x)$ would just be

$$\mathcal{N}(\mathbf{y} | \mathbf{\Sigma}_{\mathbf{xy}} \mathbf{\Lambda}_{\mathbf{yy}} (\mathbf{x} - \mu_{\mathbf{x}}) + \mu_{\mathbf{y}}, \mathbf{\Sigma}_{\mathbf{yy}} - \mathbf{\Sigma}_{\mathbf{yx}} \mathbf{\Lambda}_{\mathbf{xx}} \mathbf{\Sigma}_{\mathbf{xy}})$$

Where

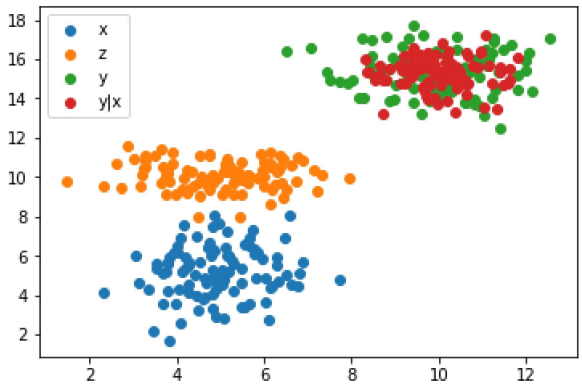
$$\begin{aligned} \Sigma_{xx} &= (x_i - \mu_x).T(x_i - \mu_x)/N \\ \Sigma_{xy} &= (x_i - \mu_x).T(y_i - \mu_y)/N \\ \Sigma_{yx} &= (y_i - \mu_y).T(x_i - \mu_x)/N \\ \Sigma_{yy} &= (y_i - \mu_y).T(y_i - \mu_y)/N \end{aligned}$$

2.3 Write code that simulates this data and illustrate the results.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

## init x and z

mux = [5, 5]
sigx = [[1, 0], [0, 1.5]]
x = np.random.multivariate_normal(mux, sigx, 100)

muz = [5, 10]
sigz = [[1.5, 0], [0, 0.5]]
z = np.random.multivariate_normal(muz, sigz, 100)
```

In [2]:
```
## compute y

meanx = np.mean(x, axis = 0)
meanz = np.mean(z, axis = 0)

sigxx = (x - meanx).T.dot(x - meanx) / x.size
sigxz = (x - meanx).T.dot(z - meanz) / x.size
sigzx = (z - meanz).T.dot(x - meanx) / x.size
sigzz = (z - meanz).T.dot(z - meanz) / x.size

muy = np.add(mux, muz)
sigy = np.add(np.add(sigxx, sigxz), np.add(sigzx, sigzz))
y = np.random.multivariate_normal(muy, sigy, 100)

## compute y given x

meany = np.mean(y, axis = 0)

sigxx = (x - meanx).T.dot(x - meanx) / x.size
sigxy = (x - meanx).T.dot(y - meany) / x.size
sigyx = (y - meany).T.dot(x - meanx) / x.size
sigyy = (y - meany).T.dot(y - meany) / x.size

muy_x = meany + sigxy.dot(np.linalg.inv(sigyy)).dot(np.array([4, 6]) - meanx)
sigy_x = sigyy - sigyx.dot(np.linalg.inv(sigxx)).dot(sigxy)
y_x = np.random.multivariate_normal(muy_x, sigy_x, 100)

plt.scatter(x[:, 0], x[:, 1], label = 'x')
plt.scatter(z[:, 0], z[:, 1], label = 'z')
plt.scatter(y[:, 0], y[:, 1], label = 'y')
plt.scatter(y_x[:, 0], y_x[:, 1], label = 'y|x')

plt.legend()
```

Out[2]: <matplotlib.legend.Legend at 0x18e0c036a90>



Exercise 3. Dimensionality Reduction and PCA

In this quesiton you will use principal component analysis to reduce the dimensionality of your data and analyze the results.

3.1 Find a set of high dimensional data. It should be continuous and have at least 6 dimensions, e.g. stats for sports teams, small sound segments or images patches also work. Note that if the dimensionality of the data is too large, you might run into computational efficiency problems using standard methods. Describe the data and illustrate it, if appropriate.

So for this problem I will be using the breast cancer dataset retrieved from the UCI machine learning database (https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29 (https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29)).
Credits: A tutorial on the `plotly` official cite (https://plot.ly/ipython-notebooks/principal-component-analysis/ (https://plot.ly/ipython-notebooks/principal-component-analysis/)) is followed in this problem as I found it very helpful to apply different packages.

Let's first import the data and briefly take a look at it:

In [3]:
```
import pandas as pd

cancer = pd.read_csv(
    filepath_or_buffer = 'http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data',
    header = None,
    sep = ',')

cancer.columns = ['id', 'diagnosis', 'radius_mean', 'texture_mean', 'perimeter_mean',
                  'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
                  'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean', 'radius_se',
                  'texture_se', 'perimeter_se', 'area_se', 'smoothness_se', 'compactness_se', 'concavity_se',
                  'concave points_se', 'symmetry_se', 'fractal_dimension_se', 'radius_worst', 'texture_worst',
                  'perimeter_worst', 'area_worst', 'smoothness_worst', 'compactness_worst', 'concavity_worst',
                  'concave points_worst', 'symmetry_worst', 'fractal_dimension_worst']

## drops the empty line if exist
cancer.dropna(how = "all", inplace = True)
cancer.tail()
```

Out[3]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
564	926424	M	21.56	22.39	142.00	1479.0	0.11100	0.11590
565	926682	M	20.13	28.25	131.20	1261.0	0.09780	0.10340
566	926954	M	16.60	28.08	108.30	858.1	0.08455	0.10230
567	927241	M	20.60	29.33	140.10	1265.0	0.11780	0.27700
568	92751	B	7.76	24.54	47.92	181.0	0.05263	0.04362

5 rows × 32 columns

As we can see above, there are 32 columns in the data, where the first column is the `id` number, the second column is the label or class `diagnosis` representing the diagnosis of the breast tissues, where `B` = benign, `M` = malignant. The rest of the columns are different continuously valued attributes as described by the column headers above. A more detailed description on the attributes of the dataset can also be found from the source link I previously provided.

Notice that for the sake of simplicity and efficiency, we will only look at six attributes for our problem 3, which is `radius_worst`, `texture_worst`, `perimeter_worst`, `area_worst`, `smoothness_worst`, and `compactness_worst`.

In [4]:
```
## split the dataset into data X and class labels y
## for simplicity, we only include six attributes in total
X = cancer.iloc[:,22:28].values
y = cancer.iloc[:,1].values
```

3.2 Compute the principal components of the data. Plot a few of the largest eigenvectors and interpret them in terms of how there are modeling the structure of the data.

To compute the eigenvectors of the data, we need to first have the covariance matrix. Let's start with standardizing the data as below.

In [5]:
```
from sklearn.preprocessing import StandardScaler
X_std = StandardScaler().fit_transform(X)
```

Then we shall have the covariance matrix along with the eigenvalues and vectors as below:

In [6]:
```
mean_vec = np.mean(X_std, axis=0)
cov_mat = (X_std - mean_vec).T.dot((X_std - mean_vec)) / (X_std.shape[0]-1)
cov_mat = np.cov(X_std.T)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)

print('eigenvalues: \n', eig_vals)
print('eigenvectors: \n', eig_vecs)
```

```
eigenvalues:
 [3.62371526 1.21294634 0.76985699 0.38028693 0.00413964 0.01961823]
eigenvectors:
 [[-4.97942824e-01 -2.72489891e-01 -8.40792994e-02  8.57684507e-02
   7.20948420e-01  3.78959132e-01]
 [-2.73102692e-01  2.28015536e-01  9.29096477e-01  1.00836736e-01
  -5.95488854e-03 -2.51370999e-04]
 [-5.04104514e-01 -2.36650175e-01 -9.49259445e-02  4.93865145e-03
  -6.88397894e-01  4.54915759e-01]
 [-4.90377681e-01 -2.91513263e-01 -8.94458489e-02  1.49878619e-01
  -5.63234946e-02 -8.00570578e-01]
 [-2.28045622e-01  7.07536477e-01 -3.05247522e-01  5.94380987e-01
  -1.12294382e-02  2.82209008e-02]
 [-3.61761561e-01  4.81837130e-01 -1.39724240e-01 -7.78908837e-01
   5.48435787e-02 -8.79311954e-02]]
```

The largest eigenvalues are already quite obvious, which are the first three with the corresponding largest eigenvectors. This typically means that the three eigenvectors bear the most information about the dataset, and thus contribute the most to the modeling of the data structure. Ultimately, they should be chosen as the principal components while we are trying to reduce the dimension of the data.

3.3 Plot, in decreasing order, the cumulative percentage of variance each eigenvector accounts for as a function of the eigenvector number. These values should be in decreasing order of the eigenvalues. Interpret these results.

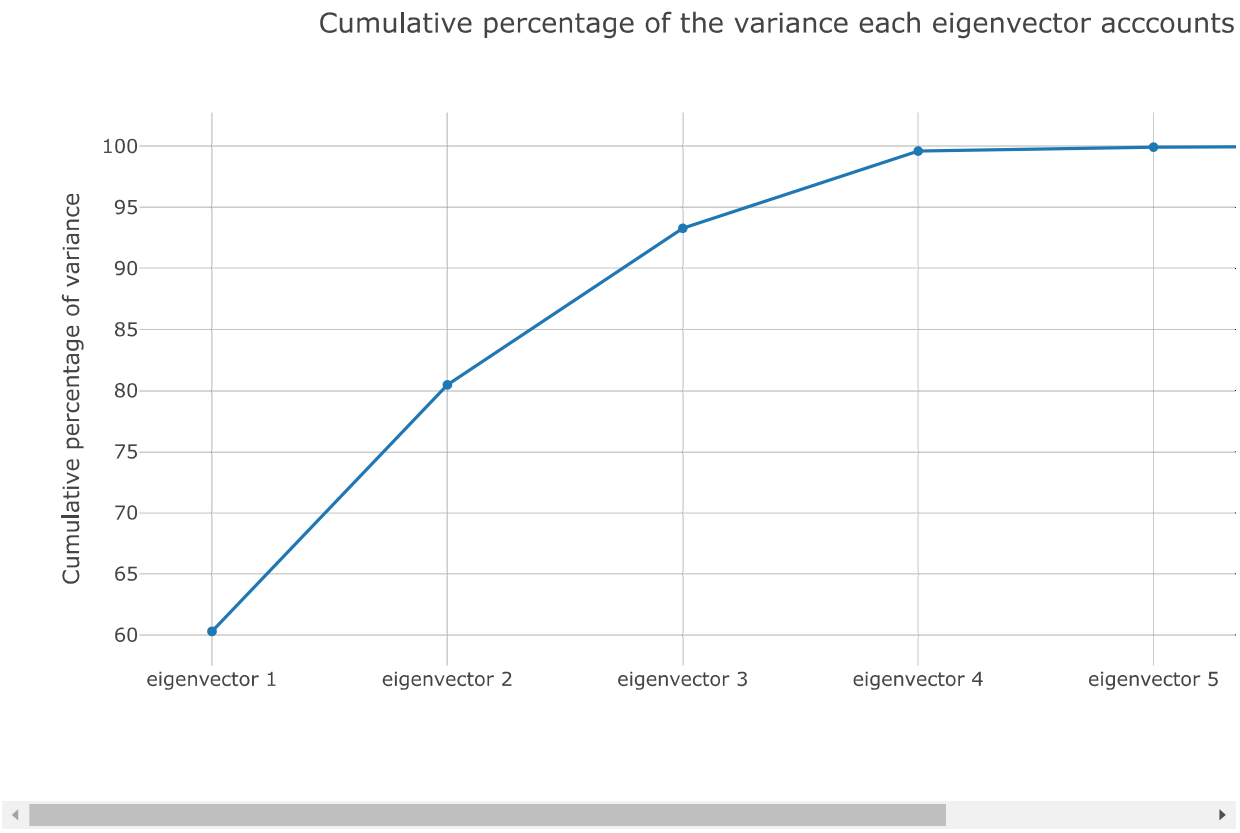
We shall first have the eigenvalues and eigenvectors we just computed listed as pairs in the decreasing order so as to compute the cumulative percentage of variance in decreasing order.

In [7]:
```
## have all the eigenvalues and eigenvectors listed in pairs
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

## Sort the pairs in decreasing order
eig_pairs.sort()
eig_pairs.reverse()

for p in eig_pairs:
    print(p[0])
```

```
3.623715261672426
1.21294633922263
0.7698569853011382
0.38028692695674704
0.019618229300002357
0.004139637828750932
```

And then we shall plot the cumulative percentage of the variance each eigenvector acccounts for as a function of the eigenvector number.

In [8]:
```
import plotly
import plotly.plotly as py
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
init_notebook_mode(connected=True)


summ = sum(eig_vals)
var_exp = [(i / summ) * 100 for i in sorted(eig_vals, reverse = True)]
cum_var_exp = np.cumsum(var_exp)

trace = dict(type = 'scatter',
            x = ['eigenvector %s' %i for i in range(1,7)],
            y = cum_var_exp)

data = [trace]

layout = dict(title = 'Cumulative percentage of the variance each eigenvector acccounts for',
              yaxis = dict(title = 'Cumulative percentage of variance'))

cancercum = dict(data = data, layout = layout)
plotly.offline.iplot(cancercum)
```



Apparently, the first two principal components capture 80.5% of the variance, which means they contain the most information of the dataset. However, it's quite a delimma whether to drop the third principal component or not since it also individually captures almost 13% of the variance. Later on, since we will be performing the reduction of dimensionality to 2-D, it makes better sense to pick the first two principal components instead of three.

3.4 Plot the original data projected into the space of the two principal eigenvectors (i.e. the eigenvectors with the largest two eigenvalues). Be sure to either plot relative to the mean, or subtract the mean when you do this. Interpret your results. What insights can you draw? Interpret the dimensions of the two largest principal components. Which dimensions of the data are correlated? Or anti-correlated?

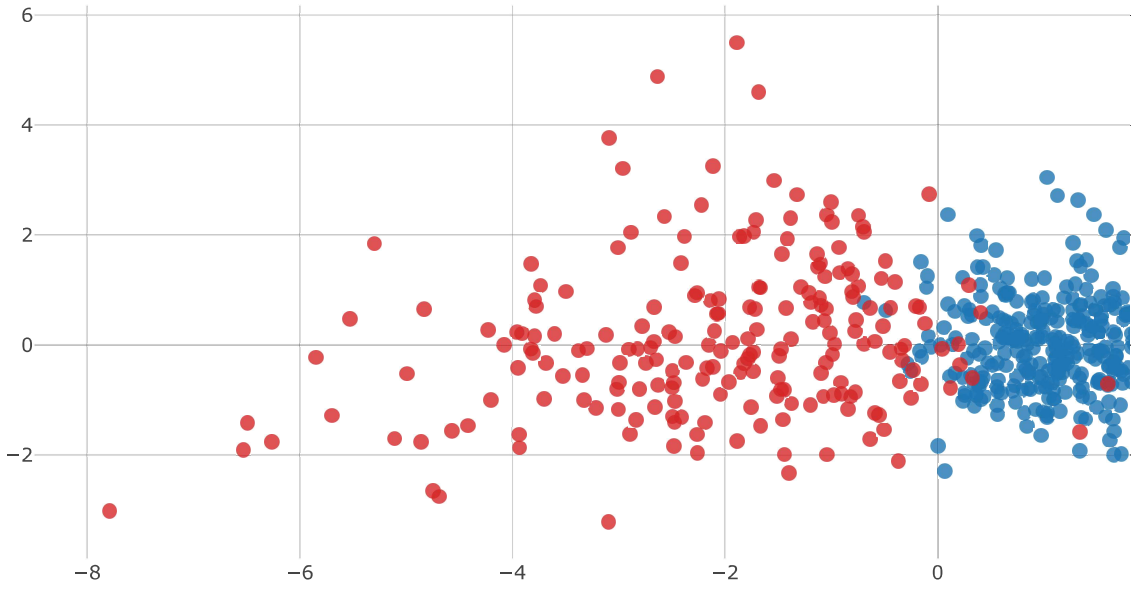
Let's first reduce our attribute space from 6-dimensional to 2-dimensional subspace since we are only picking two principal eigenvectors:

In [9]:
```
points = np.hstack((eig_pairs[0][1].reshape(6,1),
                    eig_pairs[1][1].reshape(6,1)))

points
```

Out[9]:
```
array([[-0.49794282, -0.27248989],
       [-0.27310269,  0.22801554],
       [-0.50410451, -0.23665018],
       [-0.49037768, -0.29151326],
       [-0.22804562,  0.70753648],
       [-0.36176156,  0.48183713]])
```

Then we may perform the transform:

In [10]:
```
Y = X_std.dot(points)

data = []

## muted blue for benign, brick red for malignant
colors = {'B': '#1f77b4', 'M': '#d62728'}

for name, col in zip(('B', 'M'), colors.values()):
    trace = dict(type='scatter',
                 x = Y[y == name, 0],
                 y = Y[y == name, 1],
                 mode = 'markers',
                 name = name,
                 marker = dict(color = col, size = 9, opacity = 0.8))
    data.append(trace)

layout = dict(showlegend = True,
              scene = dict(xaxis = dict(title = 'PC1'),
                           yaxis = dict(title = 'PC2')))

fig = dict(data = data, layout = layout)
plotly.offline.iplot(fig)
```



The reduced subspace separate the two diagnosis pretty well, so it should be sufficient to drop the rest of the pcs in fact.

However, the scattered plot above isn't an ideal one to present the correlational relationship between the data and the dimensions... It roughly seems that neither of the two classes are correlated with the two largest principal components...

Please refer to my exploration part for a better example that demonstrates the correlational relationships.

Exercise 4. Gaussian Mixture Models

4.1 Use the EM equations for multivariate Gaussian mixture model to write a program that implements the Gaussian Mixture Model to estimates from an ensemble of data the means, covariance matrices, and class probabilities. Choose reasonable values for your initial values and a reasonable stopping criterion. Explain your code and the steps of the algorithm. Do not assume a diagonal or isotropic covariance matrices.

Let's first derive the equations for our EM algorithm by completing the expectation step and maximization step respectively.

Expectation step

By Bayesian, we know that

$$p(c_k|x^{(n)},\theta_{1:K}) = \frac{p(x^{(n)}|c_k,\theta_{1:K})p(c_k)}{p(x^{(n)})} = \frac{p(x^{(n)}|c_k,\theta_{1:K})p(c_k)}{\Sigma_k p(x^{(n)}|c_k,\theta_{1:K})p(c_k)}$$

Then, by applying the Gaussian distribution, we are given:

$$p(c_k|x^{(n)},\theta_{1:K}) = \frac{\mathcal{N}(x^{(n)}|c_k,\theta_{1:K})p(c_k)}{\Sigma_k \mathcal{N}(x^{(n)}|c_k,\theta_{1:K})p(c_k)}$$

Maximization step

From the slides, we have the equation:

$$p(c_k|x^{(n)},\theta_{1:K}) = \frac{\mathcal{N}(x^{(n)}|c_k,\theta_{1:K})p(c_k)}{\sum_k \mathcal{N}(x^{(n)}|c_k,\theta_{1:K})p(c_k)}$$

Then to iteratively maximize and update the mean μ_k, covariance Σ_k, and class prior p_k, we use the following equations as our update function during each iteration:

$$\mu_k = \frac{\sum_{n=1}^{N} p(k|x^{(n)})x^{(n)}}{\sum_{i=1}^{N} p(k|x^{(n)})}$$

$$p_k = \frac{p(k|x^{(n)})}{\sum_{n=1}^{N} p(k|x^{(n)})}$$

$$\mu_k' = \sum_n p(k|x^{(n)})(x^{(n)} - \mu_k)^2$$

Then we shall go ahead and implement our algorithm as below:

In [11]:
```
## compute the posterior of each data point
def expectation(data, gmm):
    numerator = np.zeros((len(gmm), data.shape[0]))
    denominator = np.zeros((len(gmm), data.shape[0]))
    for k in range(len(gmm)):
        numerator[k] = gmm[k]["prior"] * multivariate_normal.pdf(data, gmm[k]["mean"], gmm[k]["cova
riance"])
        for j in range(len(gmm)):
            denominator[k] += gmm[j]["prior"] * multivariate_normal.pdf(data, gmm[j]["mean"], gmm[j
]["covariance"])
    return np.divide(numerator, denominator)

## compute new mean and class prior for each class
def maximization(posterior, data, gmm):
    N = np.zeros(len(gmm))
    for k in range(N.shape[0]):
        N[k] = np.sum(posterior[k])

    ## new prior
    prior = np.zeros(len(gmm))
    for k in range(prior.shape[0]):
        prior[k] = np.divide(N[k], N.sum())
        gmm[k]["prior"] = prior[k]

    ## new mean
    mu = np.zeros((len(gmm), len(gmm[0]["mean"])))
    for k in range(mu.shape[0]):
        for n in range(data.shape[0]):
            mu[k] += posterior[k, n] * data[n]
        gmm[k]["mean"] = 1 / N[k] * mu[k]

    return gmm

## perform the EM algorithm
def EM (data, gmm, n):
    for j in range(n):
        posterior = expectation(data, gmm)
        gmm = maximization(posterior, data, gmm)
    return gmm
```

4.2 Write code to plot the 3-sigma contours of each Gaussian overlayed on the data (try to find a library function to plot ellipses). Illustrate with an example.

Credits: a lot of help received from *Xiangda Tian* and *Yihe Guo* on making the plot for the gmm.

Let's first define the function to make the plot as below:

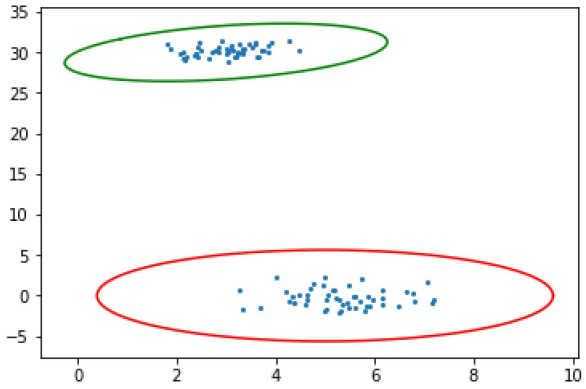
In [12]:
```
## this is the part that needs credits from Xiangda Tian for his help

import csv, copy
from scipy.stats import chi2, multivariate_normal

%matplotlib inline

## plot the gaussian model
def __gmm__(mu, sigma, pltopt = 'k'):
    if sigma.any():
        ## setup eigenvector and eigenvalues
        eigenValue, eigenVector = np.linalg.eig(sigma)
        ## setup eclipse parameter
        c = chi2.ppf(0.9, 2)
        t = np.linspace(0, 2 * np.pi, 100)
        u = [np.cos(t), np.sin(t)]
        w = c * eigenVector.dot(np.diag(np.sqrt(eigenValue)).dot(u))
        z = w.T + mu

    else:
        z = mu

    plt.plot(z[:,0], z[:,1], pltopt)

def colorPicker(index):
    colors = 'rgbcmyk'
    return colors[np.remainder(index, len(colors))]

def gmmplot(data, gmm):
    # plot data points
    plt.scatter(data[:, 0], data[:, 1], s = 4)
    # plot gmm
    for index, model in enumerate(gmm):
        __gmm__(model['mean'], model['covariance'], colorPicker(index))
```

Then we may have our own example to demonstrate the gmm and dataset. Of course, currently since the gaussian dataset is also generated by the same parameters as the gmm, the plot should overlap upon init.

In [13]:
```
## the gaussian parameters
mu = np.asarray([[5, 0], [3, 30]]).astype('float')

sigma = np.asarray([[[1, 0], [0, 1.5]],
                    [[0.5, 0.2], [0.2, 0.6]]]).astype('float')
```

In [14]:
```
## take the same parameters to init the gaussian distribution
mu1 = np.random.multivariate_normal(mu[0], sigma[0], 50)
mu2 = np.random.multivariate_normal(mu[1], sigma[1], 50)
```

In [15]:
```
## gmm
gmm = [{'mean': mu[m], 'covariance': sigma[m], 'prior': 1.0 / 2} for m in range(2)]
## the data points
data = np.concatenate((mu1, mu2), axis = 0)
## make the plot and compare the results
gmmplot(data, gmm)
```



As we can see above, the location and shape of the eclipse confirms with the mean parameter as well as covariance parameter of our gaussian mixture model.

We may go ahead and apply our algorithm in the next section to see how the plot would change.

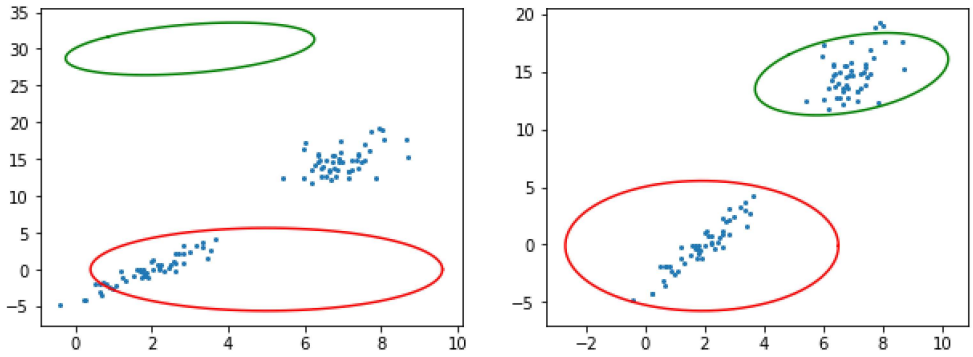
4.3 Define a two-model Gaussian mixture test case, synthesize the data, and verify that your algorithm infers the (approximately) correct values based on training data sampled from the model and plotting the results.

As we have already defined our 2-model Gaussian mixture test case `gmm` in the previous section, we just perform our EM algorithm on a new testing dataset and make the plot as below:

In [16]:
```
## initialize the testing data
mu_test = np.asarray([[2, 0], [7, 15]]).astype('float')

sigma_test = np.asarray([[[1, 2], [2, 4.5]],
                         [[0.5, 2], [0.2, 3]]]).astype('float')

mut1 = np.random.multivariate_normal(mu_test[0], sigma_test[0], 50)
mut2 = np.random.multivariate_normal(mu_test[1], sigma_test[1], 50)

data = np.concatenate((mut1, mut2), axis = 0)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:8: RuntimeWarning:

covariance is not symmetric positive-semidefinite.
```

In [17]:
```
plt.figure(figsize=(16, 8))

## we synthesize the gmm as the same one we used above
gmm0 = copy.deepcopy(gmm)

## plot the init stage of our model
plt.subplot(231)
gmmplot(data, gmm0)

# perform the EM algorithm on the model
gmm0 = EM(data, gmm0, 5)

## plot the final stage
plt.subplot(232)
gmmplot(data, gmm0)
```



Apparently the performance of our algorithm is pretty reasonable!

4.4 Apply your model to the Old Faithful dataset (supplied with the assignment files). Run the algorithm for the cases $K = 1$, $K = 2$, and $K = 3$. For each case, plot the progression of the solutions at the beginning, middle, and final steps in the learning. For each your plots (you should have 9 total), you should also print out the corresponding values of the mean, covariance, and class probabilities.

To apply our model to the Old Faithful dataset, let's first import the dataset as below:

In [18]:
```
with open('faithful.txt') as csvfile:
    ## read and deliminate the txt file
    file = csv.reader(csvfile, delimiter = ' ')
    olddata = []
    for row in file:
        olddata.append(np.array(row).astype(np.float))

    ## put the data in an array
    olddata = np.asarray(olddata)
```

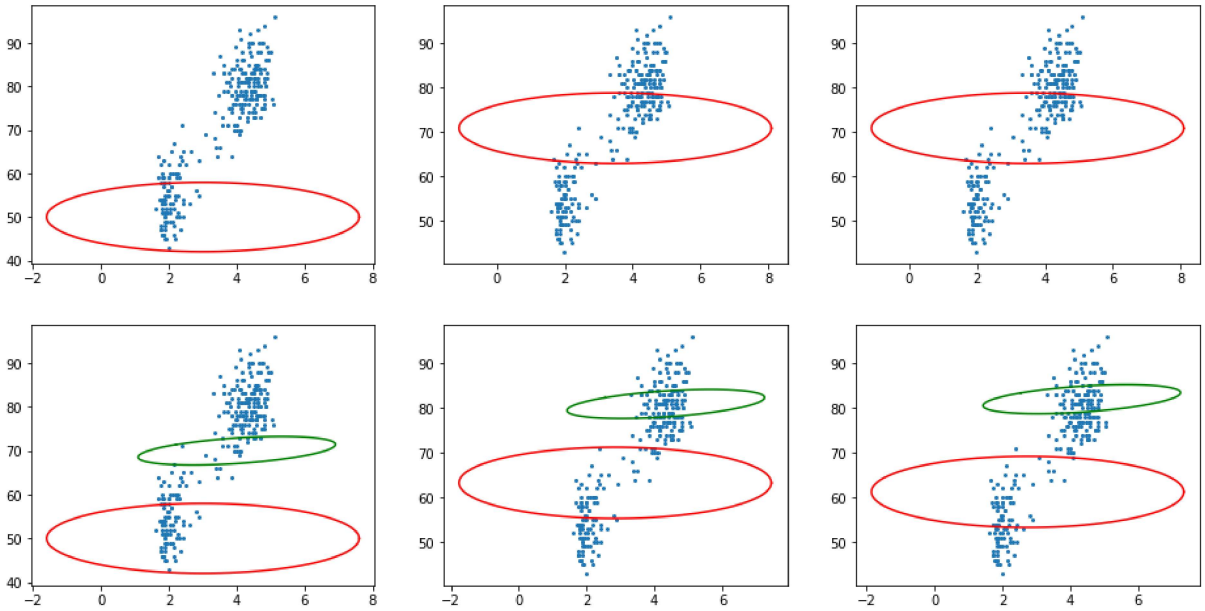
And then we may go ahead and initialize the model so as to make the plots:

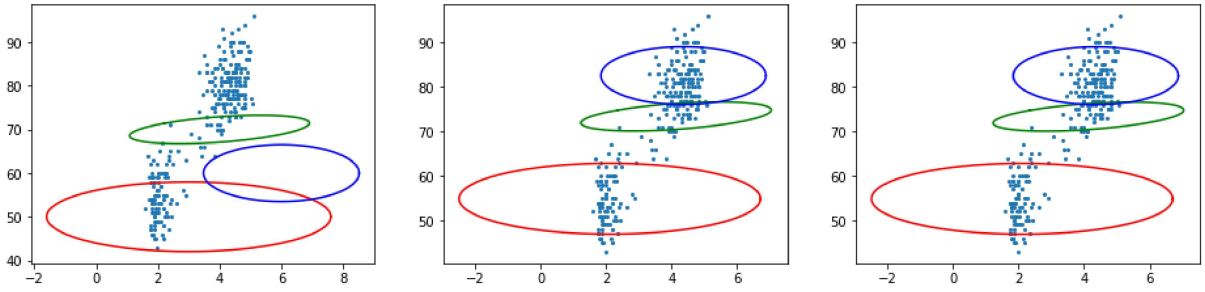
In [19]:
```
## initialize the gmm parameters
mu = np.asarray([[3, 50], [4, 70], [6, 60]]).astype('float')
sigma = np.asarray([[[1, 0], [0, 3]],
                    [[0.4, 0.2], [0.2, 0.5]],
                    [[0.3, 0], [0, 2]]]).astype('float')

## initialize the gmms
## k = 1
gmm1 = [{'mean': mu[m], 'covariance': sigma[m], 'prior': 1.0 / 1} for m in range(1)]
## k = 2
gmm2 = [{'mean': mu[m], 'covariance': sigma[m], 'prior': 1.0 / 2} for m in range(2)]
## k = 3
gmm3 = [{'mean': mu[m], 'covariance': sigma[m], 'prior': 1.0 / 3} for m in range(3)]

gmms = [gmm1, gmm2, gmm3]
```

In [20]:
```
## helper function to plot three different gmms, each with 3 plots
def __plot__ (gmms, data):
    ## counter for printing
    k = 1

    for gmmm in gmms:
        ## init new plot each time
        plt.figure(figsize=(16, 8))
        ## index for subplot
        i = 1
        ## synthesize the gmm data to avoid errors
        gmm = copy.deepcopy(gmmm)

        ## plot and print the init stage of our model
        plt.subplot(230 + i)
        gmmplot(data, gmm)
        print("Plot ", k, ":", gmm)
        i += 1
        k += 1

        # perform the EM algorithm on the model
        for j in range(5):
            posterior = expectation(data, gmm)
            gmm = maximization(posterior, data, gmm)
            ## plot and print in the middle of the training
            if (j == 3):
                plt.subplot(230 + i)
                gmmplot(data, gmm)
                print("Plot ", k,":", gmm)
                i += 1
                k += 1

        ## plot and print the final stage
        plt.subplot(230 + i)
        gmmplot(data, gmm)
        print("Plot ", k, ":", gmm)
        i += 1
        k += 1
```

In [21]:
```
__plot__(gmms, olddata)
```

```
Plot  1 : [{'mean': array([ 3., 50.]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 1.0}]
Plot  2 : [{'mean': array([ 3.48778309, 70.89705882]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 1.0}]
Plot  3 : [{'mean': array([ 3.48778309, 70.89705882]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 1.0}]
Plot  4 : [{'mean': array([ 3., 50.]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 0.5}, {'mean': array([ 4., 70.]), 'covariance': array([[0.4, 0.2],
       [0.2, 0.5]]), 'prior': 0.5}]
Plot  5 : [{'mean': array([ 2.84620396, 63.27896441]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 0.5691671475561435}, {'mean': array([ 4.33536413, 80.9612149 ]), 'cova
riance': array([[0.4, 0.2],
       [0.2, 0.5]]), 'prior': 0.4308328524438565}]
Plot  6 : [{'mean': array([ 2.74661774, 61.24770526]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 0.5358836647722091}, {'mean': array([ 4.34355647, 82.03851315]), 'cova
riance': array([[0.4, 0.2],
       [0.2, 0.5]]), 'prior': 0.464116335227791}]
Plot  7 : [{'mean': array([ 3., 50.]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 0.3333333333333333}, {'mean': array([ 4., 70.]), 'covariance': array
([[0.4, 0.2],
       [0.2, 0.5]]), 'prior': 0.3333333333333333}, {'mean': array([ 6., 60.]), 'covariance': array
([[0.3, 0. ],
       [0. , 2. ]]), 'prior': 0.3333333333333333}]
Plot  8 : [{'mean': array([ 2.1061183 , 54.87609775]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 0.3711794984124237}, {'mean': array([ 4.13523516, 73.40086521]), 'cova
riance': array([[0.4, 0.2],
       [0.2, 0.5]]), 'prior': 0.15316618228437479}, {'mean': array([ 4.35748586, 82.5928505 ]), 'c
ovariance': array([[0.3, 0. ],
       [0. , 2. ]]), 'prior': 0.47565431930320157}]
Plot  9 : [{'mean': array([ 2.10516048, 54.86585277]), 'covariance': array([[1., 0.],
       [0., 3.]]), 'prior': 0.370889972214986}, {'mean': array([ 4.13363578, 73.36097197]), 'covar
iance': array([[0.4, 0.2],
       [0.2, 0.5]]), 'prior': 0.1517544267401682}, {'mean': array([ 4.35671571, 82.56949721]), 'co
variance': array([[0.3, 0. ],
       [0. , 2. ]]), 'prior': 0.4773556010448458}]
```





The result looks pretty good!

Exploration

Introduction

So in Exercise 3, we have performed PCA on the dataset to reducte data dimensions, while in Exercise 4, we have applied our EM algorithm on a few Gaussian mixture models to train and cluster the data points. In this part, I'd like to combine the two exercises a little bit to see how the two methods can contribute to real data classification in practice.

My exploration exercise can be broken down into a few steps as below:

1. Choose and inspect the dataset.
2. Perform PCA on the dataset to find appropriate principal components so as the "flatten" the dimension of our datasets as a preparation for step 3.
3. Apply the EM algorithm we came up with in Exercise 4 on the flatten dataset and see how it goes.
4. Try to come up with different sets of mean and covariances to fit the dataset. See how different parameters can affect the final training effects of our EM algorithm. See if there is any alternative.

Data inspection

Before searching for an appropriate dataset, let's first define what an ideal dataset would be for this experiment.

First of all, of course, we need a high dimensional dataset but the number of attributes should not be more then ten, otherwise it would be incovenient for data extraction and the PCA practice.
Secondly, the target of the dataset should be classification since I'd still like to compare the training results with the actual results to compare the clustered result with the actual classfication.
Lastly, we want to have more than two classes so that we'd be able to try more combinations of gmm parameters and see how these parameters would influence the final result.

After playing with different datasets online, I figured out that the `iris dataset` can almost fulfill all of the criterion above except that the dimensionality of the iris is only four. I spent a little more time on the UCI machine learning database trying to find a better dataset but failed. So let's just focus on the iris, back to the good old days.

As in the Exercise 3, we shall start with importing the dataset. Again, this time I will still be retrieving the dataset from the UCI machine learning database (https://archive.ics.uci.edu/ml/datasets/Iris (https://archive.ics.uci.edu/ml/datasets/Iris)).

In [22]:
```
import pandas as pd

iris = pd.read_csv(
    filepath_or_buffer = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
,
    header = None,
    sep = ',')

iris.columns = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class']
## drops the empty line if exist
iris.dropna(how = "all", inplace = True)
iris.tail()
```

Out[22]:

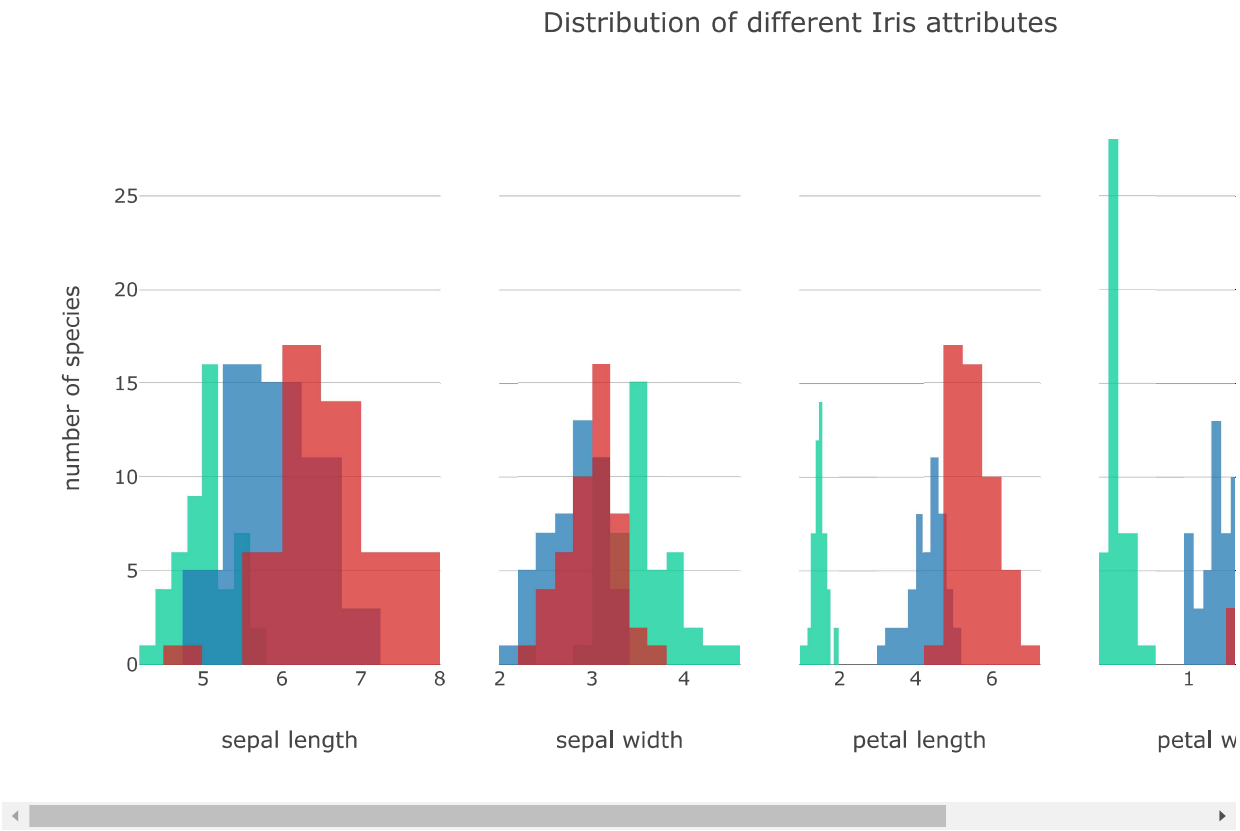
	sepal_length	sepal_width	petal_length	petal_width	class
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

As all of us should be quite familiar with the iris dataset, it may be quite redundant to repeat the description on the iris dataset. But in case some of you might have not seen the dataset before, the iris dataset is majorly composed of four attributes and one class. All of the attributes are continuous values representing some feature of the iris flower, and the classes contain three species of the flower: `virginica`, `setosa`, and `versicolor`.

Now we shall go ahead and have a deeper inspection on the distribution of iris attributes through the scope of histograms to see what kind of distribution does the iris dataset bear:

In [23]:
```
## split the dataset into data X and class labels y
X = iris.iloc[:, 0:4].values
y = iris.iloc[:, 4].values
```

In [24]:
```
import plotly
import plotly.plotly as py

data = []

## muted blue for versicolor, brick red for virginica, caribbean grean for setosa
colors = {'Iris-setosa': '#00cc96',
          'Iris-versicolor': '#1f77b4',
          'Iris-virginica': '#d62728'}

legend = {0:False, 1:False, 2:False, 3:True}

## data for plotting the histograms
for col in range(4):
    for key in colors:
        trace = dict(type = 'histogram',
                     x = list(X[y == key, col]),
                     opacity = 0.75,
                     xaxis = 'x%s' %(col + 1),
                     marker = dict(color = colors[key]),
                     name = key,
                     showlegend = legend[col])
        data.append(trace)

## arrange the layout
layout = dict(
    barmode = 'overlay',
    xaxis = dict(domain = [0, 0.25], title = 'sepal length'),
    xaxis2 = dict(domain = [0.3, 0.5], title = 'sepal width'),
    xaxis3 = dict(domain = [0.55, 0.75], title = 'petal length'),
    xaxis4 = dict(domain = [0.8, 1], title = 'petal width'),
    yaxis = dict(title = 'number of species'),
    title = 'Distribution of different Iris attributes')

fig = dict(data = data, layout = layout)
plotly.offline.iplot(fig, filename = 'iris data visualization')
```



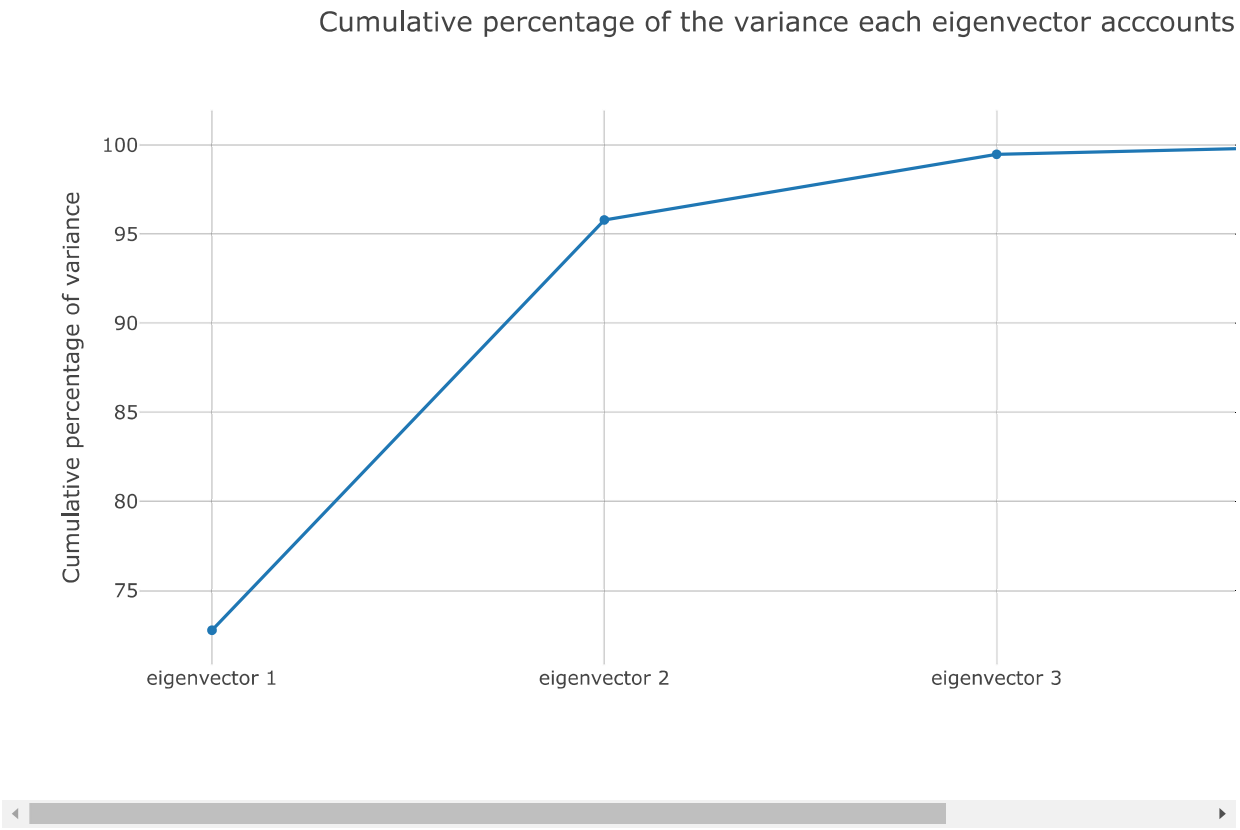
We can see that the last two attributes of the dataset make a great job in separating the three classes, and we might doubt wheter its still necessary, given such a good separation, to still apply PCA on the dataset. We might compare the different outcome after the clustering in the following part!

PCA dimension reduction

Now that we have our dataset setup, we may go ahead and perform the PCA as in Exercise 3.

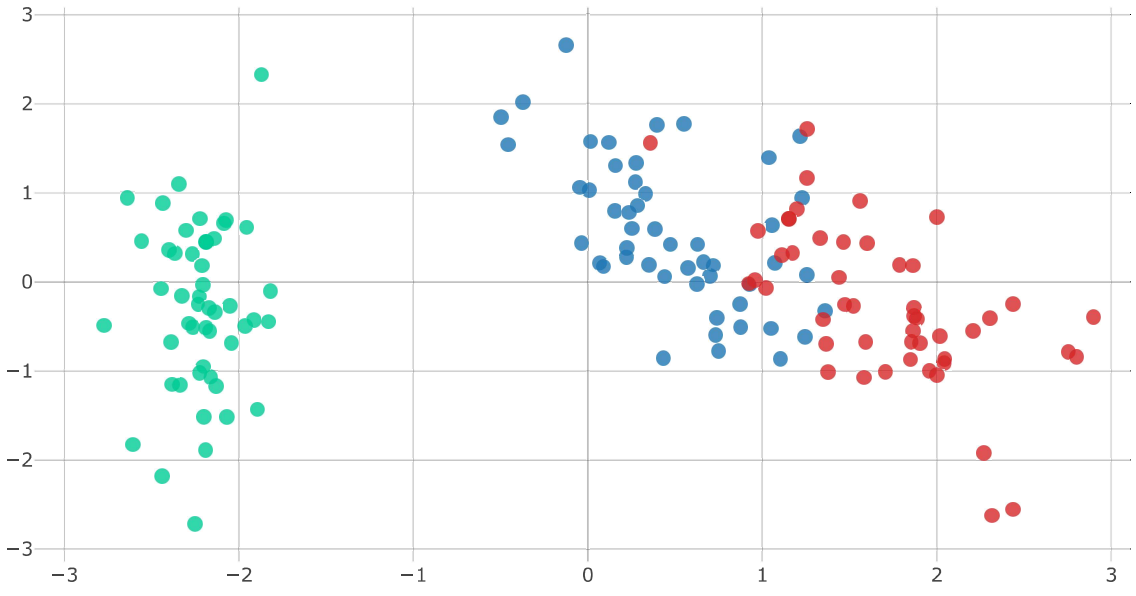
In [25]:
```
## setup eigenvalues and vectors

from sklearn.preprocessing import StandardScaler
X_std = StandardScaler().fit_transform(X)

import numpy as np
mean_vec = np.mean(X_std, axis=0)
cov_mat = (X_std - mean_vec).T.dot((X_std - mean_vec)) / (X_std.shape[0]-1)

cov_mat = np.cov(X_std.T)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)

eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

## Sort the pairs in decreasing order
eig_pairs.sort()
eig_pairs.reverse()
```

In [26]:
```
summ = sum(eig_vals)
var_exp = [(i / summ) * 100 for i in sorted(eig_vals, reverse = True)]
cum_var_exp = np.cumsum(var_exp)

trace = dict(type = 'scatter',
             x = ['eigenvector %s' %i for i in range(1,5)],
             y = cum_var_exp)

data = [trace]

layout = dict(title = 'Cumulative percentage of the variance each eigenvector acccounts for',
              yaxis = dict(title = 'Cumulative percentage of variance'))

fig = dict(data = data, layout = layout)
plotly.offline.iplot(fig, filename = 'cumulative percentage of variance')
```



From the plot above, it should be quite obvious that pertaining the first two principal components should be enough to keep necessary information of our dataset. So let's do it!

In [27]:
```
points = np.hstack((eig_pairs[0][1].reshape(4,1),
                    eig_pairs[1][1].reshape(4,1)))

Y = X_std.dot(points)

data = []

for name, col in zip(('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'), colors.values()):
    trace = dict(type='scatter',
                 x = Y[y == name, 0],
                 y = Y[y == name, 1],
                 mode = 'markers',
                 name = name,
                 marker = dict(color = col, size = 9, opacity = 0.8))
    data.append(trace)

layout = dict(showlegend = True,
              scene = dict(xaxis = dict(title = 'PC1'),
                           yaxis = dict(title = 'PC2')))

fig = dict(data = data, layout = layout)
plotly.offline.iplot(fig, filename = 'iris projection')
```



This time, unlike the result we got in Exercise 3, the scattered plot clearly demonstrate the anti-correlated relationship between the versicolor as well as virginica class and the principal components. The setosa class is uncorrelated with the principal components in this case.

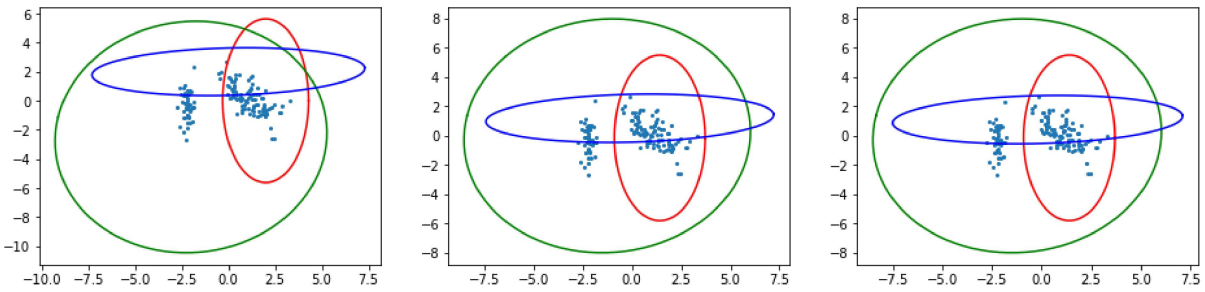
Apply EM algorithm on the GMM

As usual, let's first find a random gaussian parameter for our gmm and see how it goes:

In [28]:
```
## the gmm parameters
mu1 = np.asarray([[2, 0], [-2, -2.5], [0, 2]]).astype('float')

sigma1 = np.asarray([[[0.25, 0], [0, 1.5]],
                     [[2.5, 0.1], [0.1, 3]],
                     [[2.5, 0.5], [0.1, 0.15]]]).astype('float')
```

In [29]:
```
gmm_1 = [{'mean': mu1[m], 'covariance': sigma1[m], 'prior': 1.0 / 3} for m in range(3)]

__plot__([gmm_1], Y)
```

```
Plot  1 : [{'mean': array([2., 0.]), 'covariance': array([[0.25, 0.  ],
       [0.  , 1.5 ]]), 'prior': 0.3333333333333333}, {'mean': array([-2. , -2.5]), 'covariance': a
rray([[2.5, 0.1],
       [0.1, 3. ]]), 'prior': 0.3333333333333333}, {'mean': array([0., 2.]), 'covariance': array
([[2.5 , 0.5 ],
       [0.1 , 0.15]]), 'prior': 0.3333333333333333}]
Plot  2 : [{'mean': array([ 1.41181663, -0.16662627]), 'covariance': array([[0.25, 0.  ],
       [0.  , 1.5 ]]), 'prior': 0.4391942047000901}, {'mean': array([-1.25397558, -0.02556818]),
'covariance': array([[2.5, 0.1],
       [0.1, 3. ]]), 'prior': 0.4880335213369917}, {'mean': array([-0.11102529,  1.17708865]), 'co
variance': array([[2.5 , 0.5 ],
       [0.1 , 0.15]]), 'prior': 0.07277227396291813}]
Plot  3 : [{'mean': array([ 1.40305611, -0.15631825]), 'covariance': array([[0.25, 0.  ],
       [0.  , 1.5 ]]), 'prior': 0.4385319931626477}, {'mean': array([-1.22744921, -0.01972589]),
'covariance': array([[2.5, 0.1],
       [0.1, 3. ]]), 'prior': 0.48967703651218614}, {'mean': array([-0.19823243,  1.08941096]), 'c
ovariance': array([[2.5 , 0.5 ],
       [0.1 , 0.15]]), 'prior': 0.07179097032516607}]
```



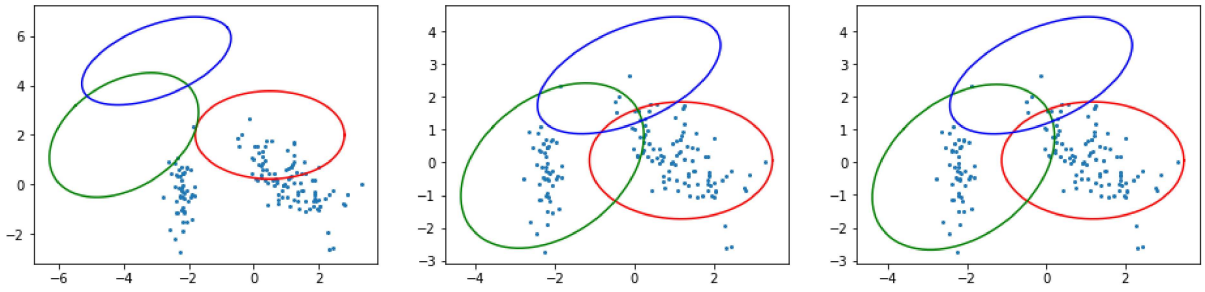
As shown in the plot above, the result doesn't seem quite ideal if our goal is to cluster the points according to the original classification. So we shall make a few experiments in the following section to see if we can improve this situation.

Experiment different parameters

In this case, since our ultimate goal is to "classify" the data points, that criterion should be reflected on our gaussian parameters. Let's start with looking at the prettier one and see what's the effect:

In [30]:
```
## the gmm parameters
mu2 = np.asarray([[0.5, 2], [-4, 2], [-3, 5]]).astype('float')

sigma2 = np.asarray([[[0.25, 0], [0, 0.15]],
                     [[0.25, 0.1], [0.1, 0.3]],
                     [[0.25, 0.1], [0.1, 0.15]]]).astype('float')

gmm_2 = [{'mean': mu2[m], 'covariance': sigma2[m], 'prior': 1.0 / 3} for m in range(3)]

__plot__([gmm_2], Y)
```

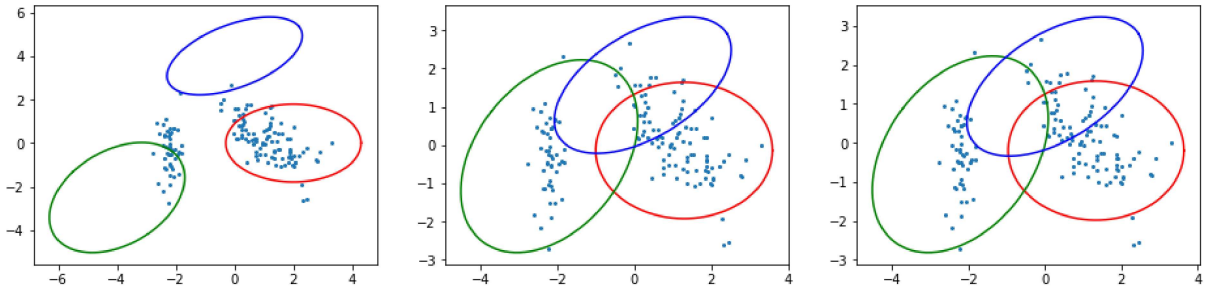
```
Plot  1 : [{'mean': array([0.5, 2. ]), 'covariance': array([[0.25, 0.  ],
       [0.  , 0.15]]), 'prior': 0.3333333333333333}, {'mean': array([-4.,  2.]), 'covariance': arr
ay([[0.25, 0.1 ],
       [0.1 , 0.3 ]]), 'prior': 0.3333333333333333}, {'mean': array([-3.,  5.]), 'covariance': arr
ay([[0.25, 0.1 ],
       [0.1 , 0.15]]), 'prior': 0.3333333333333333}]
Plot  2 : [{'mean': array([1.17874298, 0.0587724 ]), 'covariance': array([[0.25, 0.  ],
       [0.  , 0.15]]), 'prior': 0.6358073163781832}, {'mean': array([-2.05785213, -0.10260983]),
'covariance': array([[0.25, 0.1 ],
       [0.1 , 0.3 ]]), 'prior': 0.3641920214840286}, {'mean': array([-0.12826651,  2.65666691]),
'covariance': array([[0.25, 0.1 ],
       [0.1 , 0.15]]), 'prior': 6.621377882084657e-07}]
Plot  3 : [{'mean': array([1.17942789, 0.05788382]), 'covariance': array([[0.25, 0.  ],
       [0.  , 0.15]]), 'prior': 0.6354148205855668}, {'mean': array([-2.08851997, -0.1479174 ]),
'covariance': array([[0.25, 0.1 ],
       [0.1 , 0.3 ]]), 'prior': 0.35845478691132343}, {'mean': array([-0.12821078,  2.64933461]),
'covariance': array([[0.25, 0.1 ],
       [0.1 , 0.15]]), 'prior': 0.006130392503109711}]
```



This time the result looks a lot much better comparing to the previous one. But can it get even closer to the actual classification given a set of parameters close enough to the actual mean? We shall see in the next experiment:

In [31]:
```
mu3 = np.asarray([[2, 0], [-4, -2.5], [0, 4]]).astype('float')

sigma3 = np.asarray([[[0.25, 0], [0, 0.15]],
                     [[0.25, 0.1], [0.1, 0.3]],
                     [[0.25, 0.1], [0.1, 0.15]]]).astype('float')

gmm_3 = [{'mean': mu3[m], 'covariance': sigma3[m], 'prior': 1.0 / 3} for m in range(3)]

__plot__([gmm_3], Y)
```

```
Plot  1 : [{'mean': array([2., 0.]), 'covariance': array([[0.25, 0.  ],
       [0.  , 0.15]]), 'prior': 0.3333333333333333}, {'mean': array([-4. , -2.5]), 'covariance': a
rray([[0.25, 0.1 ],
       [0.1 , 0.3 ]]), 'prior': 0.3333333333333333}, {'mean': array([0., 4.]), 'covariance': array
([[0.25, 0.1 ],
       [0.1 , 0.15]]), 'prior': 0.3333333333333333}]
Plot  2 : [{'mean': array([ 1.29238697, -0.14457678]), 'covariance': array([[0.25, 0.  ],
       [0.  , 0.15]]), 'prior': 0.5530611656814954}, {'mean': array([-2.22028524, -0.29469846]),
'covariance': array([[0.25, 0.1 ],
       [0.1 , 0.3 ]]), 'prior': 0.33302738012123534}, {'mean': array([0.21636746, 1.56351665]), 'c
ovariance': array([[0.25, 0.1 ],
       [0.1 , 0.15]]), 'prior': 0.11391145419726939}]
Plot  3 : [{'mean': array([ 1.33518354, -0.19715845]), 'covariance': array([[0.25, 0.  ],
       [0.  , 0.15]]), 'prior': 0.5272018699082599}, {'mean': array([-2.21990895, -0.2920658 ]),
'covariance': array([[0.25, 0.1 ],
       [0.1 , 0.3 ]]), 'prior': 0.33337574967272227}, {'mean': array([0.25930236, 1.44388555]), 'c
ovariance': array([[0.25, 0.1 ],
       [0.1 , 0.15]]), 'prior': 0.1394223804190177}]
```



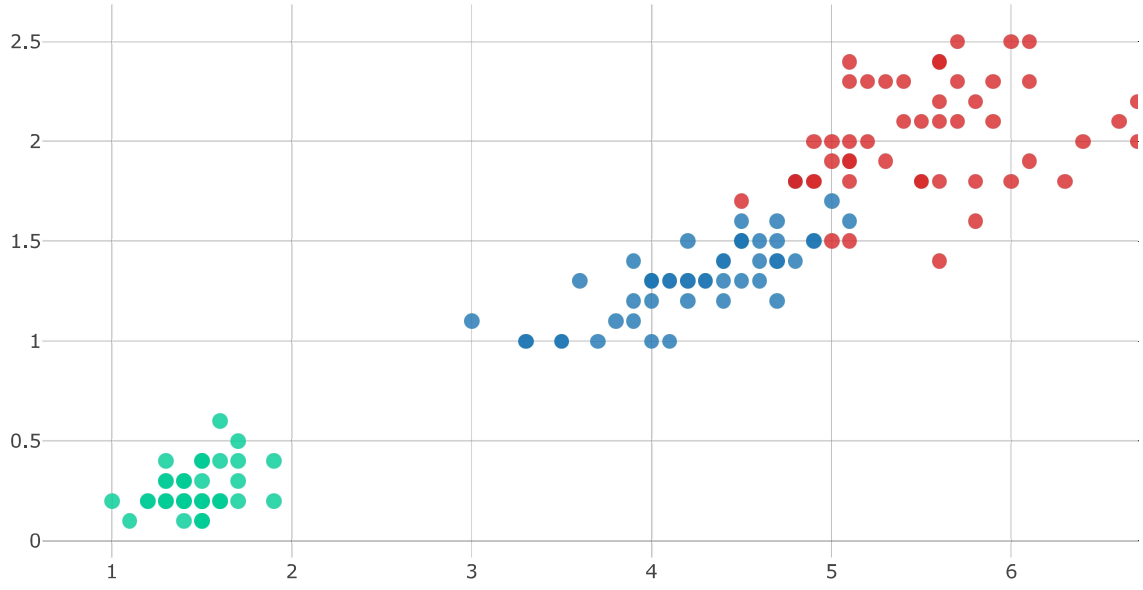
If we compare the above clustering with the actual classification, we shall see that the result we have is actually quite close to the actual classification, and the plot also reflects the mix between `versicolor` and `virginica` at the appopriate location.

Therefore, we can basically confirm the importance of finding a good init parameter for the clustering. After all, it should always be better to have the mean set as close to the true center point as possible at first, so as to improve future performance.

However, as the goal of clustering is to cluster unclassified data, it seems to be an "egg first or chicken first" problem to find a good parameter.

So, as we have mentioned in the previous part, the iris dataset has a very good separation between the classes with some given dimensions according to the attributes. Now let's take a look at whether dataset flattened by PCA shares any advantages over such cases.

In [32]:
```
## choose the two attributes as dimensions with best separation between the data
Y2 = X[:,2:4]

data = []

for name, col in zip(('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'), colors.values()):
    trace = dict(type='scatter',
                 x = Y2[y == name, 0],
                 y = Y2[y == name, 1],
                 mode = 'markers',
                 name = name,
                 marker = dict(color = col, size = 9, opacity = 0.8))
    data.append(trace)

layout = dict(showlegend = True,
              scene = dict(xaxis = dict(title = 'PC1'),
                           yaxis = dict(title = 'PC2')))

fig = dict(data = data, layout = layout)
plotly.offline.iplot(fig, filename = 'iris projection')
```

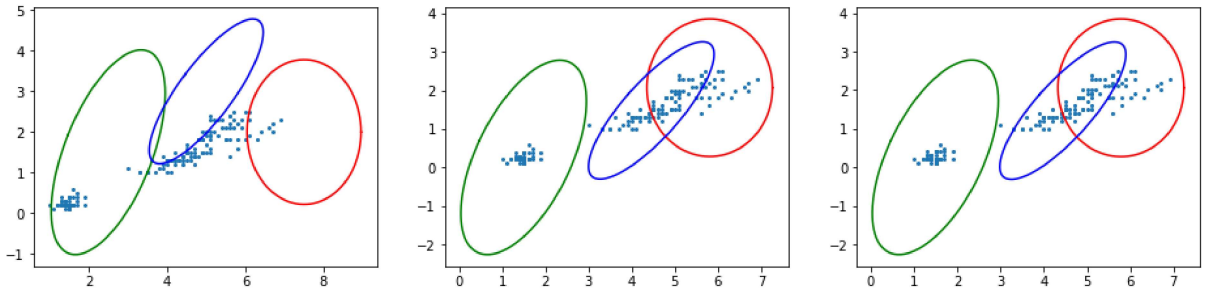


Above is our original data, if we compare that with the projection on principal components we obtained in the previous sections, they are basically in the same distribution, except that both `versicolor` and `virginica` are correlated to the dimensions here, and so does the `setosa` class.

Now let's apply the same covariance parameter as last time, with the mean centered at roughly the same relative location, and see how EM algorithm can improve the clustering on the gmm this time.

In [33]:
```
mu4 = np.asarray([[7.5, 2], [2.5, 1.5], [5, 3]]).astype('float')

sigma4 = np.asarray([[[0.1, 0], [0, 0.15]],
                     [[0.1, 0.1], [0.1, 0.3]],
                     [[0.1, 0.1], [0.1, 0.15]]]).astype('float')

gmm_4 = [{'mean': mu4[m], 'covariance': sigma4[m], 'prior': 1.0 / 3} for m in range(3)]

__plot__([gmm_4], Y2)
```

```
Plot  1 : [{'mean': array([7.5, 2. ]), 'covariance': array([[0.1 , 0.  ],
       [0.  , 0.15]]), 'prior': 0.3333333333333333}, {'mean': array([2.5, 1.5]), 'covariance': arr
ay([[0.1, 0.1],
       [0.1, 0.3]]), 'prior': 0.3333333333333333}, {'mean': array([5., 3.]), 'covariance': array
([[0.1 , 0.1 ],
       [0.1 , 0.15]]), 'prior': 0.3333333333333333}]
Plot  2 : [{'mean': array([5.81109748, 2.06620068]), 'covariance': array([[0.1 , 0.  ],
       [0.  , 0.15]]), 'prior': 0.2284935341251451}, {'mean': array([1.49412945, 0.26078873]), 'co
variance': array([[0.1, 0.1],
       [0.1, 0.3]]), 'prior': 0.3400021178134262}, {'mean': array([4.45618056, 1.47828047]), 'cova
riance': array([[0.1 , 0.1 ],
       [0.1 , 0.15]]), 'prior': 0.4315043480614287}]
Plot  3 : [{'mean': array([5.78508224, 2.06050601]), 'covariance': array([[0.1 , 0.  ],
       [0.  , 0.15]]), 'prior': 0.23763473942035088}, {'mean': array([1.49412402, 0.26078629]), 'c
ovariance': array([[0.1, 0.1],
       [0.1, 0.3]]), 'prior': 0.34000103967713685}, {'mean': array([4.44148993, 1.468759  ]), 'cov
ariance': array([[0.1 , 0.1 ],
       [0.1 , 0.15]]), 'prior': 0.42236422090251224}]
```



The result kind of looks the same, which confirms that the PCA just gives us the same effect as a good separation of data points. Still, given different combinations and "shapes" of the gaussian parameters, the results can differ a lot.

Therefore, in conclusion, it is quite important to find a possible algorithm to approach a likely init mean and covariance before starting the clustering, since without a good gaussian parameter, the final clustered resuts could vary frankly.

end of exploration

In [ ]:
```
```