Problem 1.a. The Lagrangian function corresponding to the constrained optimization is given by:

$$\wedge(p(x),\lambda_1,\lambda_2,\lambda_3) = -\int p(x) logp(x)dx + \lambda_1(\int p(x)dx - 1) + \lambda_2(\int xp(x)dx) + \lambda_3(\int x^2 p(x)dx - \sigma^2)$$

the partial derivatives is:

$$\frac{\partial}{\partial p(x)dx} \wedge (p(x),\lambda_1,\lambda_2,\lambda_3) = -\log p(x) - 1 + \lambda_1 + \lambda_2 x + \lambda_3 x^2$$

$$\log p(x) = -1 + \lambda_1 + \lambda_2 x + \lambda_3 x^2 \tag{1}$$

$\Rightarrow$

$$p(x) = \exp^{(-1+\lambda_1+\lambda_2 x+\lambda_3 x^2)}$$

Now we replace p(x) by it's value: $p(x) = exp(s(x))$
then:

$\wedge(s(x),\lambda_1,\lambda_2,\lambda_3) = -\int \exp(s(x))s(x)dx + \lambda_1(\int \exp(s(x))dx - 1) + \lambda_2(\int x\exp(s(x))dx) + \lambda_3(\int x^2 \exp(s(x))dx - \sigma^2)$
hence:

$$\frac{\partial}{\partial s(x)dx} \wedge(s(x),\lambda_1,\lambda_2,\lambda_3) = -\exp(s(x)) - \exp(s(x))s(x) + \lambda_1\exp(s(x)) + \lambda_2 x\exp(s(x)) + \lambda_3 x^2\exp(s(x))$$

$$\frac{\partial}{\partial s(x)dx} \wedge (s(x),\lambda_1,\lambda_2,\lambda_3) = 0 = \exp(s(x))(-1 + s(x) + \lambda_1 + \lambda_2 x + \lambda_3 x^2)$$

$\Rightarrow$

$$s(x) = -1 + \lambda_1 + \lambda_2 x + \lambda_3 x^2$$

$$\frac{\partial\wedge}{\partial\lambda_1} = 0 = \int \exp(s(x))dx - 1$$

$$\frac{\partial\wedge}{\partial\lambda_2} = 0 = \int x\exp(s(x))dx$$

$$\frac{\partial\wedge}{\partial\lambda_3} = 0 = \int x^2\exp(s(x))dx - \sigma^2$$

Problem 1.b.

$$p(x) = \exp^{(-1+\lambda_1+\lambda_2 x+\lambda_3 x^2)}$$

$$p(x) = \exp^{(-1+\lambda_1+\lambda_2 x)} . \exp^{(\lambda_3 x^2)}$$

Now let take:

$$\lambda_3 = -\frac{1}{2\lambda} \quad where \quad Var(x) = \lambda = \sigma^2$$

$$p(x) = k.exp^{\frac{-x^2}{2\lambda}}$$

where: $k = \exp^{(-1+\lambda_1+\lambda_2 x)}$
Since k is constant we take $k = \frac{1}{\sqrt{2\pi\lambda}}$ Where: $Var(x) = \lambda = \sigma^2$ so: $p(x) = \frac{1}{\sqrt{2\pi\lambda}}.exp^{\frac{-x^2}{2\lambda}}$
hence:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}.exp^{\frac{-x^2}{2\sigma^2}}$$

So the distribution that maximize the entropy is gaussian.

Problem 2.a. Sketch the joint probability distribution $p(x, y)$, along with the projections $z(\theta)$.

Solution:

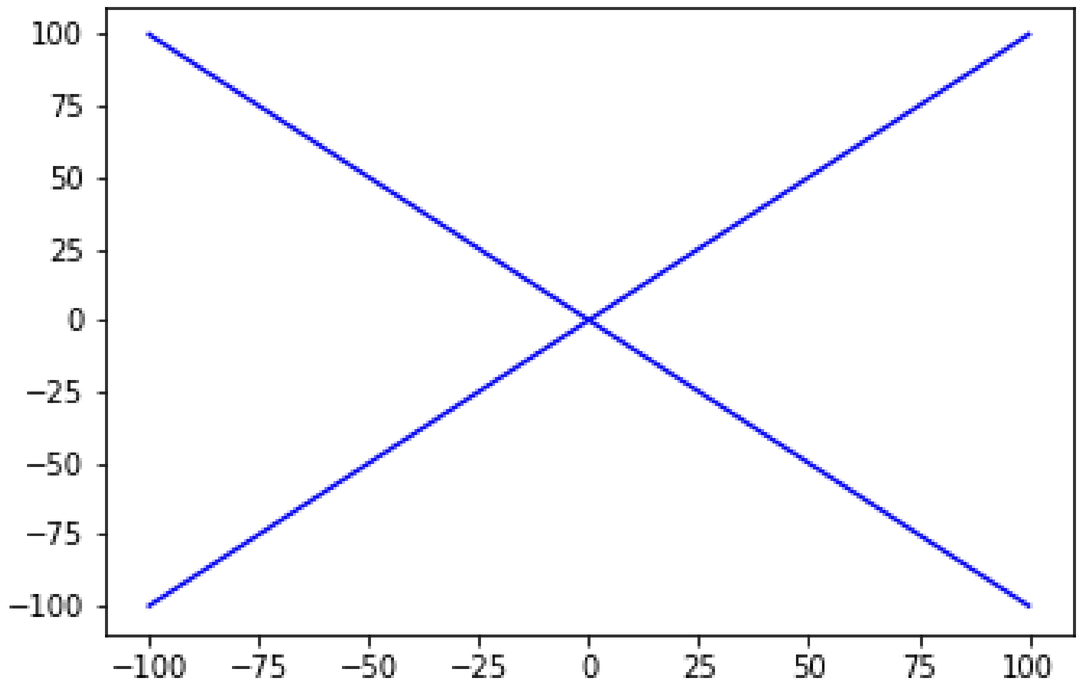


Figure 1: Blue lines correspond to value of infinity, other values are 0.

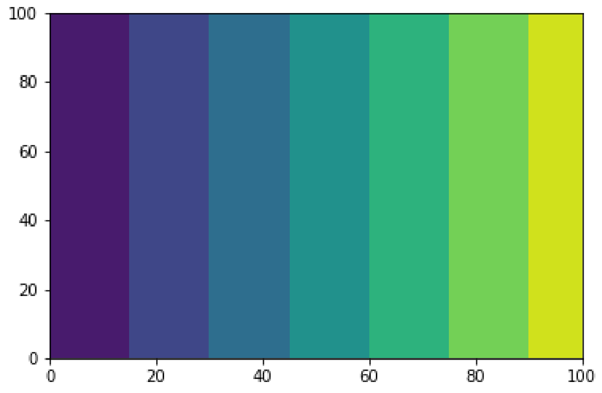


Figure 2: Sketch of projection $z(0)$

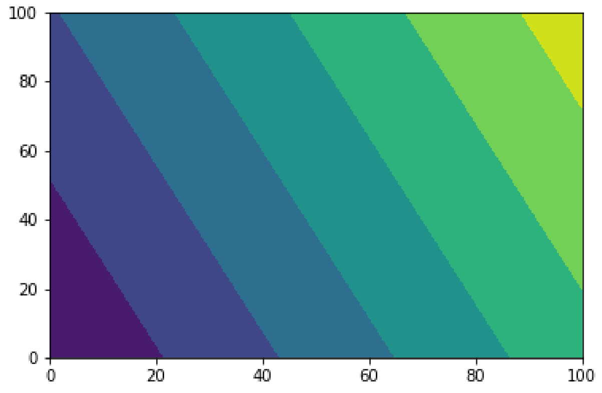


Figure 3: Sketch of projection $z(\frac{\pi}{8})$

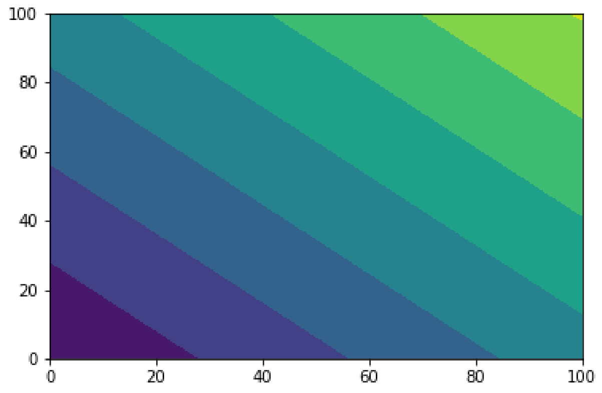


Figure 4: Sketch of projection $z(\frac{\pi}{4})$

$\square$

Problem 2.b. Find the principal components of $p(x, y)$.

Solution: We want to maximize the variance of $z(\theta)$. First note that $\mathbf{E}[X] = 0$ and $\mathbf{E}[X^2] = 1$ because of the way X was defined. We compute the other expected values that will be needed.

$$\begin{aligned}\mathbf{E}[XY] &= \iint xy \cdot p(x,y)dxdy \\ &= \int xp(x)dx \int_{-\infty}^{\infty} \frac{1}{2} y \cdot (\delta(y-x) + \delta(y+x))dy \\ &= 0\,,\end{aligned}$$

because the integral over y is equal to zero. Similarly we get that $\mathbf{E}[Y] = 0$. Next we calculate the variance of Y. Since its mean is zero this corresponds to calculating $\mathbf{E}[Y^2]$.

$$\begin{aligned}\mathbf{E}[Y^2] &= \int p(x)dx \int_{-\infty}^{\infty} \frac{1}{2} y^2 \cdot (\delta(y-x) + \delta(y+x))dy \\ &= \int p(x)dx \cdot x^2\,,\end{aligned}$$

which is equivalent to the variance of X, thus $\mathbf{E}[Y^2] = 1$. Now we are ready to calculate the variance of $z(\theta)$.

$$Var(z(\theta)) = \mathbf{E}[z(\theta)^2] + \mathbf{E}[z(\theta)]^2 = \mathbf{E}[z(\theta)^2]$$

, because of

$$\begin{aligned}\mathbf{E}[z(\theta)] &= \mathbf{E}[Xcos(\theta) + Ysin(\theta)] \\ &= cos(\theta)\mathbf{E}[X] + sin(\theta)\mathbf{E}[Y] \\ &= 0\,.\end{aligned}$$

Thus the variance becomes

$$\begin{aligned}Var(z(\theta)) &= \mathbf{E}[z(\theta)^2] \\ &= \mathbf{E}[x^2cos^2(\theta) + y^2sin^2(\theta) + 2sin(\theta)cos(\theta)xy] \\ &= \mathbf{E}[x^2cos^2(\theta)] + \mathbf{E}[y^2sin^2(\theta)] + \mathbf{E}[2sin(\theta)cos(\theta)xy] \\ &= cos^2(\theta)\mathbf{E}[x^2] + sin^2(\theta)\mathbf{E}[y^2] + sin(2\theta)\mathbf{E}[xy] \\ &= 1 \;\forall\theta \in [0, 2\pi)\,.\end{aligned}$$

Thus the variance of the projected data is equally large in all directions. □

Problem 2.c.

Solution:

$$kurt[z(\theta) = \frac{\mathbf{E}[(z(\theta) - \mathbf{E}[z(\theta)])^4]}{Var(z(\theta)^2)} - 3 = \mathbf{E}[z(\theta)^4] - 3 =$$

□

sheet03

May 8, 2019

1 Independent Component Analysis

In this exercise, you will implement the FastICA algorithm, and apply it to model independent components of a distribution of image patches. The description of the fastICA method is given in the paper *"A. Hyvärinen and E. Oja. 2000. Independent component analysis: algorithms and applications"* linked from ISIS, and we frequently refer to sections and equations in that paper.

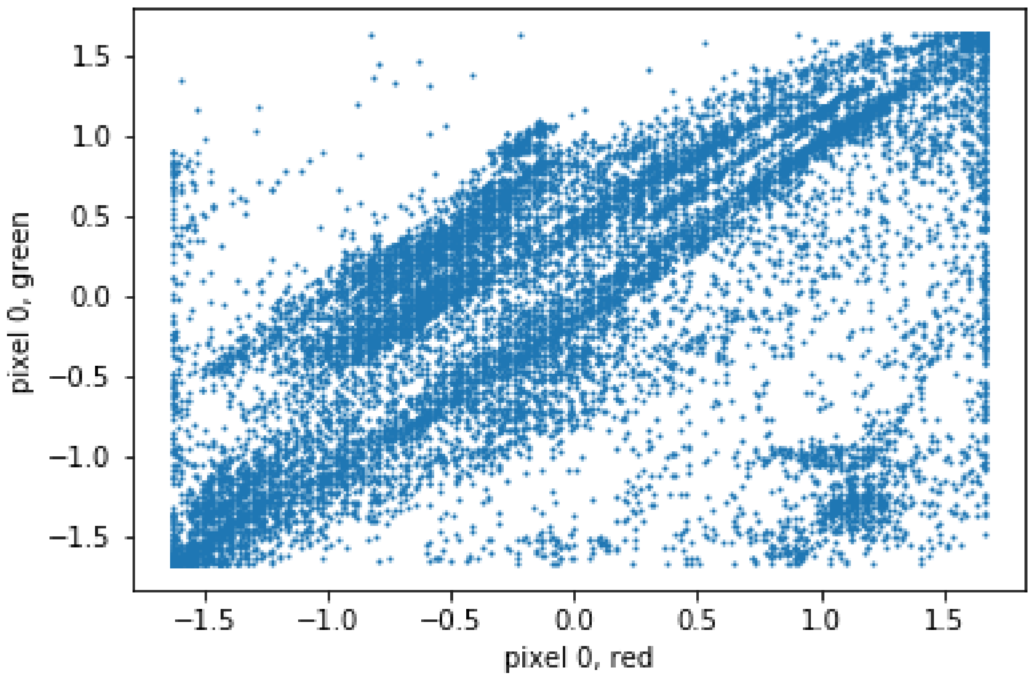
Three methods are provided for your convenience:

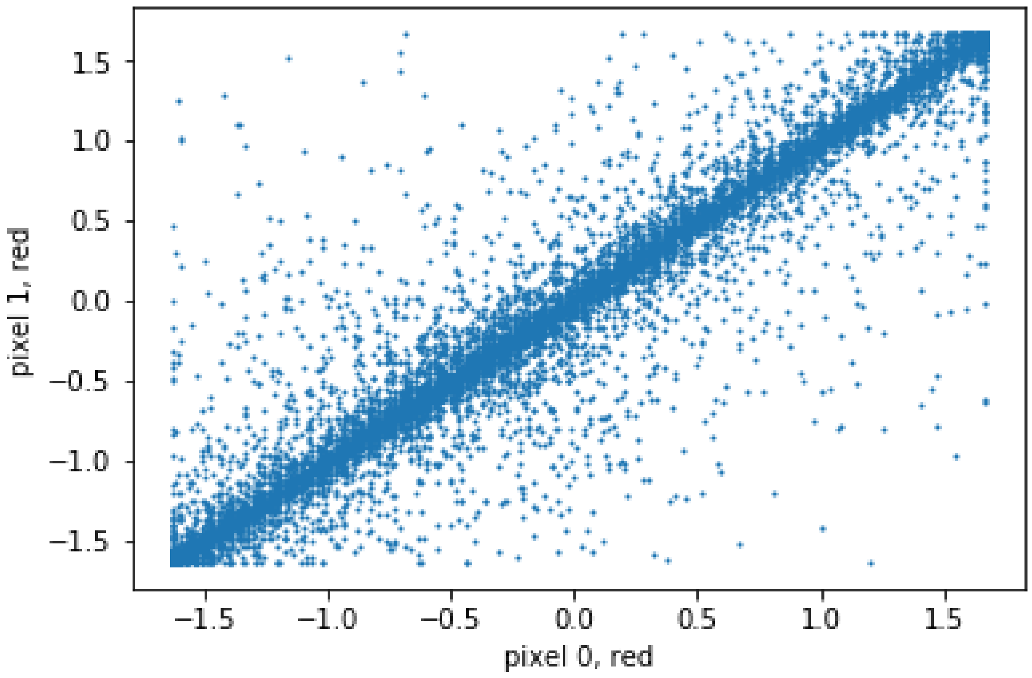
- **`utils.load()`** extracts a dataset of image patches from an collection of images (contained in the folder `images/` that can be extracted from the `images.zip` file). The method returns a list of RGB image patches of size 12×12, presented as a matrix of size $\#patches \times 432$. (Note that $12 \cdot 12 \cdot 3 = 432$).
- **`utils.scatterplot(...)`** produces a scatter plot from a two-dimensional data set. Each point in the scatter plot represents one image patch.
- **`utils.render(...)`** takes a matrix of size $\#patches \times 432$ as input and renders these patches in the IPython notebook.

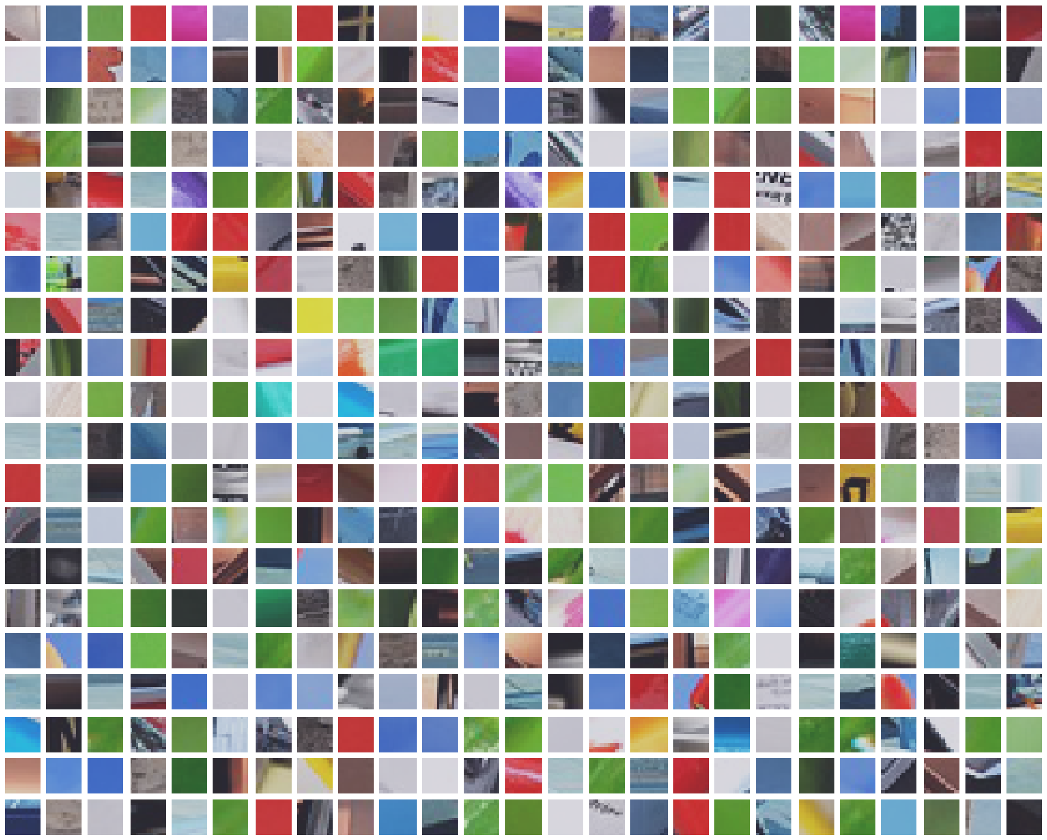
1.1 Demo code

A demo code that makes use of these three methods is given below. The code performs basic analysis such as loading the data, plotting correlations between neighboring pixels, or different color channels of the same pixel, and rendering some image patches.

```
In [1]: import utils
        %matplotlib inline

        # Load the dataset of image patches
        X = utils.load()

        # Plot the red vs. green channel of the first pixel
        utils.scatterplot(X[:,0],X[:,1],xlabel='pixel 0, red',ylabel='pixel 0, green')

        # Plot the red channel of the first and second pixel
        utils.scatterplot(X[:,0],X[:,3],xlabel='pixel 0, red',ylabel='pixel 1, red')

        # Visualize 500 image patches from the image
        utils.render(X[:500])
```


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
pixel 0, green
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
pixel 0, red


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
pixel 1, red
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
pixel 0, red



1.2 Whitening (10 P)

Independent component analysis applies whitening to the data as a preprocessing step. The whitened data matrix $\tilde{X}$ is obtained by linear projection of X, such data such that $\mathrm{E}[\tilde{x}\tilde{x}^\top] = I$, where $\tilde{x}$ is a row of the whitened matrix $\tilde{X}$. See Section 5.2 of the paper for a complete description of the whitening procedure.

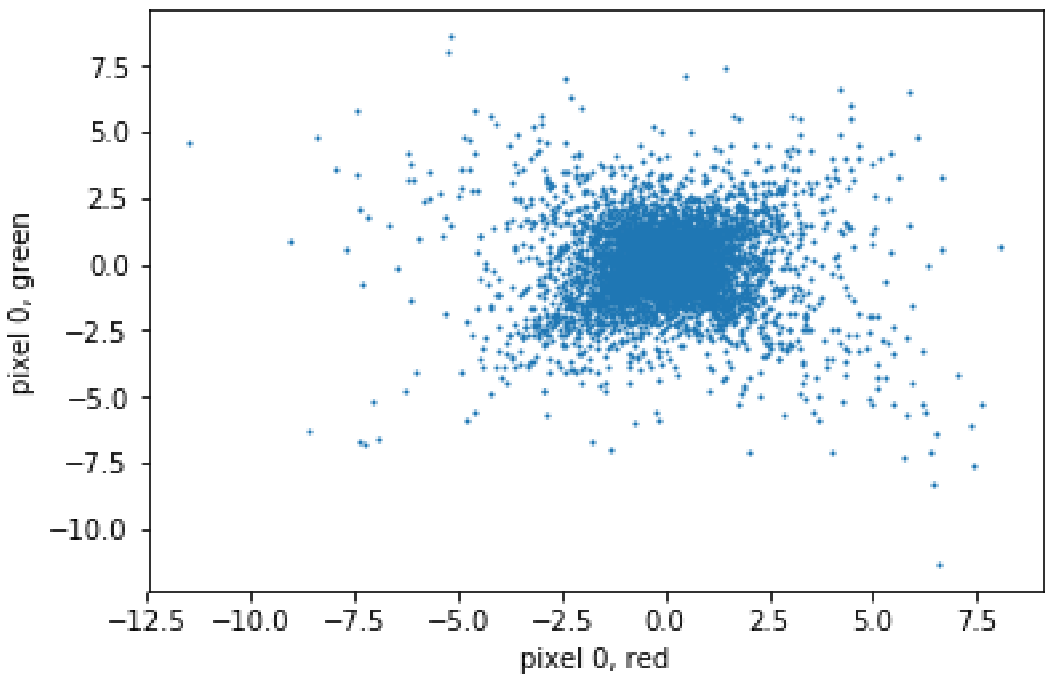
Tasks:

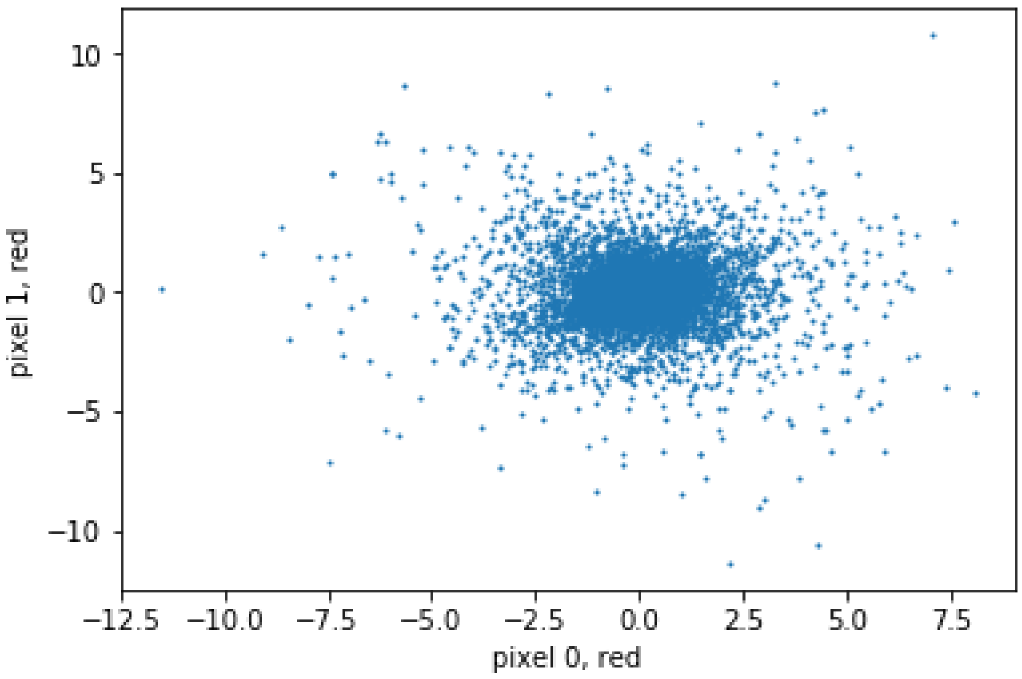
- **Implement a function that returns a whitened version of the data given as input.**
- **Add to this function a test that makes sure that $\mathrm{E}[\tilde{x}\tilde{x}^\top] \approx I$ (up to numerical accuracy).**
- **Reproduce the scatter plots of the demo code, but this time, using the whitened data.**
- **Render 500 whitened image patches.**

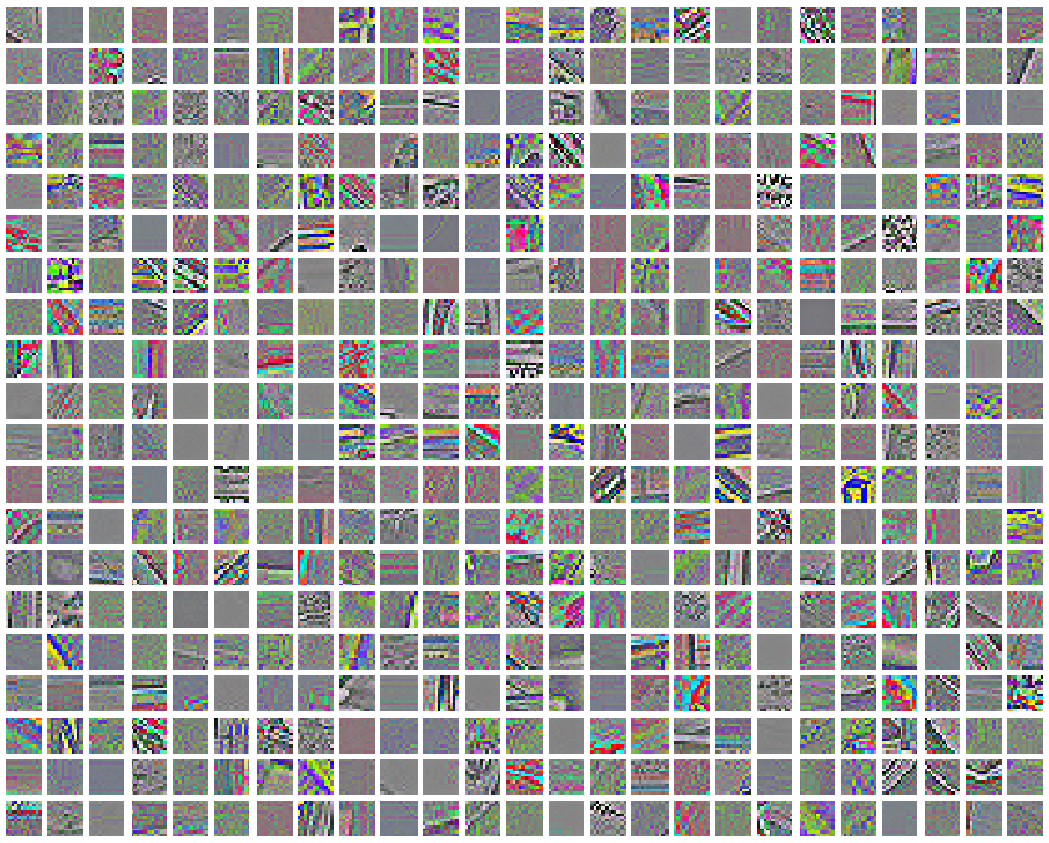
```
In [3]: import numpy as np

        def whitening(X):
            covm = np.cov(X,rowvar=False)
            d,E = np.linalg.eig(covm)
            epsilon = 1e-5
```

```
    D = np.diag(1./np.sqrt(d+epsilon))
    W = np.dot(E,np.dot(D,E.T))
    X_white = np.dot(X,W)
    covm_w = np.cov(X_white, rowvar=False)
    test = np.abs((covm_w-np.identity(len(covm_w))).sum())
    if test < 1e-4:
        print("cov matrix is close to identity matrix. Difference sums to " + str(test
        return X_white
    else:
        print("covariance matrix of whitened data not close enough to identity. Differe

X_white = whitening(X)


# Plot the red vs. green channel of the first pixel of the whitened data
utils.scatterplot(X_white[:,0],X_white[:,1],xlabel='pixel 0, red',ylabel='pixel 0, gree

# Plot the red channel of the first and second pixel of the whitened data
utils.scatterplot(X_white[:,0],X_white[:,3],xlabel='pixel 0, red',ylabel='pixel 1, red

# Visualize 500 whitened image patches
utils.render(X_white[:500])
```

```
cov matrix is close to identity matrix. Difference sums to 1.95543663274e-05
```




pixel 1, red
pixel 0, red
10
5
0
−5
−10
−12.5
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5



1.3 Implementing FastICA (20 P)

We now would like to learn 100 independent components of the distribution of whitened image patches. For this, we follow the procedure described in the Chapter 6 of the paper. Implementation details specific to this exercise are given below:

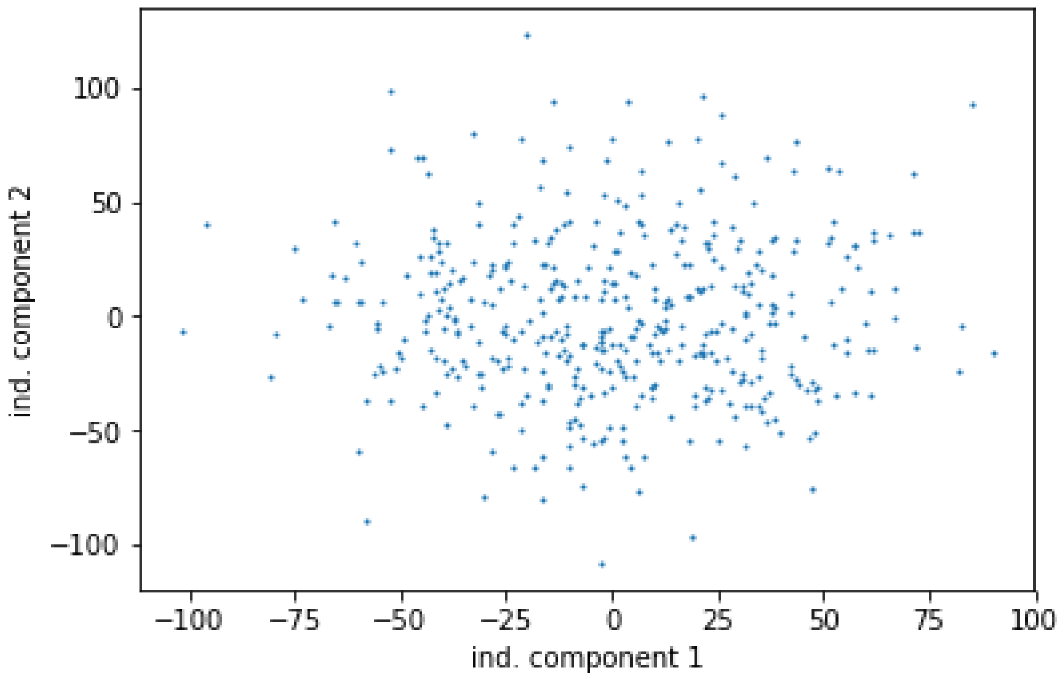
- **Nonquadratic function G**: In this exercise, we will make use of the nonquadratic function $G(x) = \frac{1}{a} \log \cosh(ax)$, proposed in Section 4.3.2 of the paper, with $a = 1.5$. This function admits as a derivative the function $g(x) = \tanh(ax)$, and as a double derivative the function $g'(x) = a \cdot (1 - \tanh^2(ax))$.
- **Number of iterations**: The FastICA procedure will be run for 64 iterations. Note that the training procedure can take a relatively long time (up to 5 minutes depending on the system). Therefore, during the developement phase, it is advised to run the algorithm for a fraction of the total number of iterations.
- **Objective function**: The objective function that is maximized by the ICA training algorithm is given in Equation 25 of the paper. Note that since we learn 100 independent components, the objective function is in fact the *sum* of the objective functions of each independent components.
- **Finding multiple independent components**: Conceptually, finding multiple independent components as described in the paper is equivalent to running multiple instances of FastICA (one per independent component), under the constraint that the components learned by these instances are decorrelated. In order to keep the learning procedure computationally affordable, the code must be parallelized, in particular, make use of numpy matrix multiplications instead of loops whenever it is possible.
- **Weight decorrelation**: To decorrelate outputs, we use the inverse square root method given in Equation 45.

Tasks:

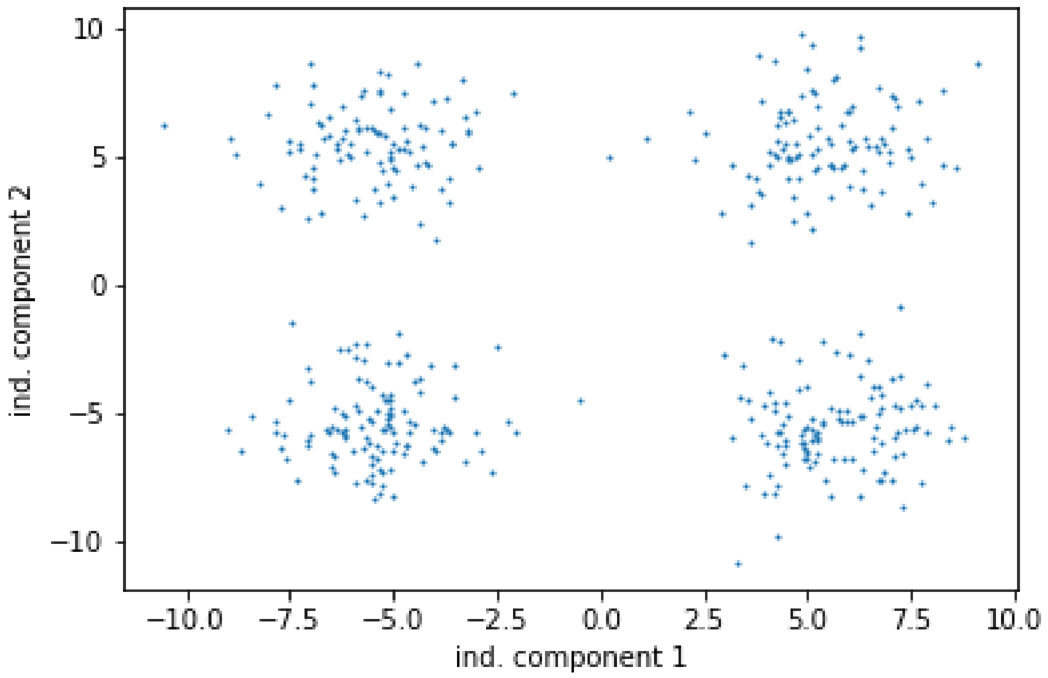
- **Implement the FastICA method described in the paper, and run it for 64 iterations.**
- **Print the value of the objective function at each iteration.**
- **Create a scatter plot of the projection of the whitened data on two distinct independent components after 0, 1, 3, 7, 15, 31, 63 iterations.**
- **Visualize the learned independent components using the function `render(...)`.**

```
In [8]: # Number of independent components
        c = 100
        # Constant a
        a = 1.5
        # No. of iterations
        N = 64
```

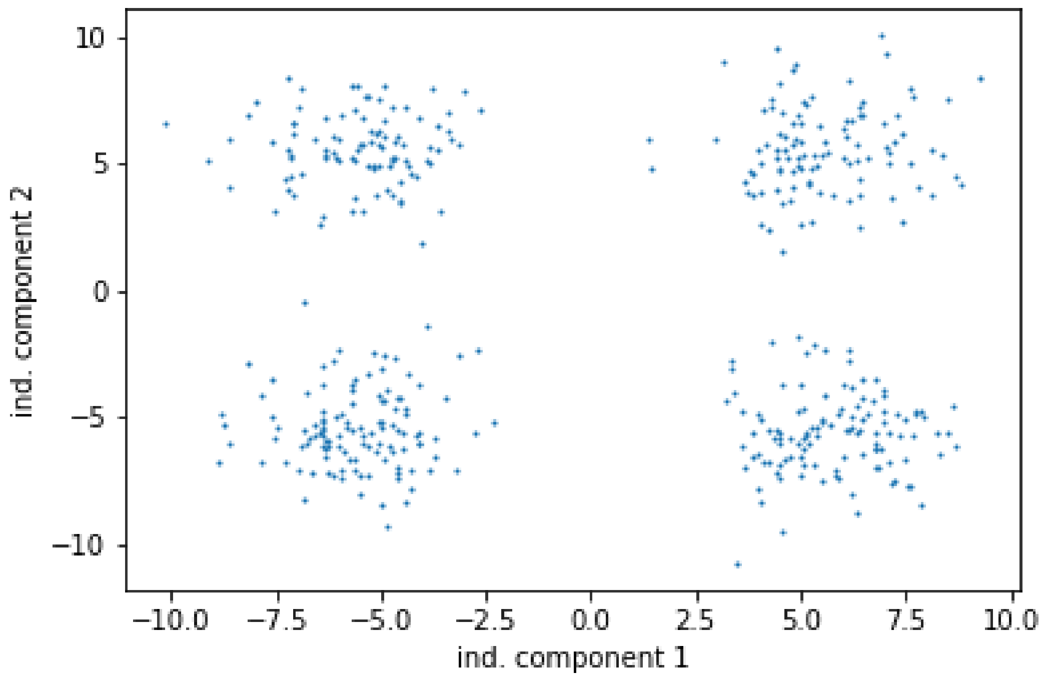
```python
def fastICA(X,c,a,iterations):
    #Initialize random matrix W
    W = np.random.rand(len(X),c)

    for j in range(iterations):
        #Scatterplot of two independent components
        if j in [0,1,3,7,15,31,63]:
            projection = np.matmul(W.T,X)
            #print(projection.shape)
            utils.scatterplot(projection[0],projection[1],xlabel='ind. component 1',yla

        #Precalculate g and g'
        wtx = np.matmul(W.T,X)
        g = np.tanh(a*wtx)
        dg = a*(1-(np.tanh(a*wtx))**2)

        #Calculate the objective
        G = ((1/a)*np.log(np.cosh(a*wtx))).mean(axis=1).sum()
        G_normal = ((1/a)*np.log(np.cosh(a*np.random.normal(size=wtx.shape)))).mean(axi
        objective = (G-G_normal)**2
        print("Iteration: " + str(j) + ", Objective: " + str(objective))

        #Update W
        for i in range(c):
            w = W[:,i,np.newaxis]
            w = (np.dot(X,g[i,np.newaxis].T)-(np.sum(dg[i])*w))/432
            w = w / np.linalg.norm(w)
            W[:,i] = w.flatten()

        #Decorrelate
        D,F = np.linalg.eig(np.matmul(W.T,W))
        epsilon = 1e-5
        D = np.diag(1./np.sqrt(D+epsilon))
        DC = np.matmul(F,np.matmul(D,F.T))
        W = np.matmul(DC,W.T).T
    return np.matmul(W.T,X)

S = fastICA(X_white,c,a,N)

utils.render(S)
```



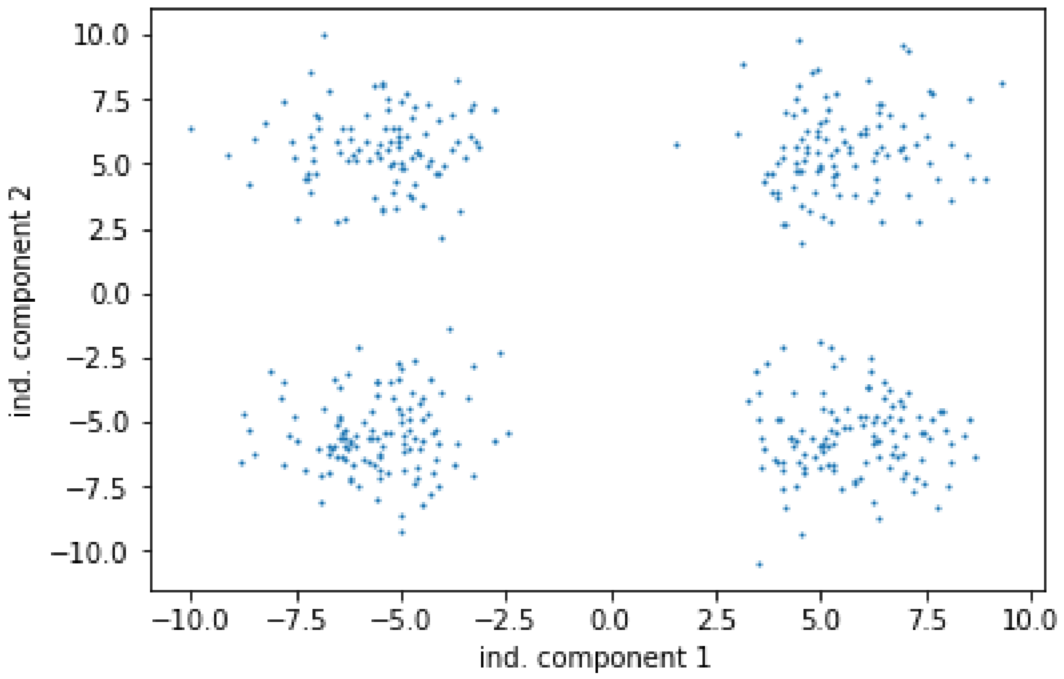
```
Iteration: 0, Objective: 7007551.60472
```



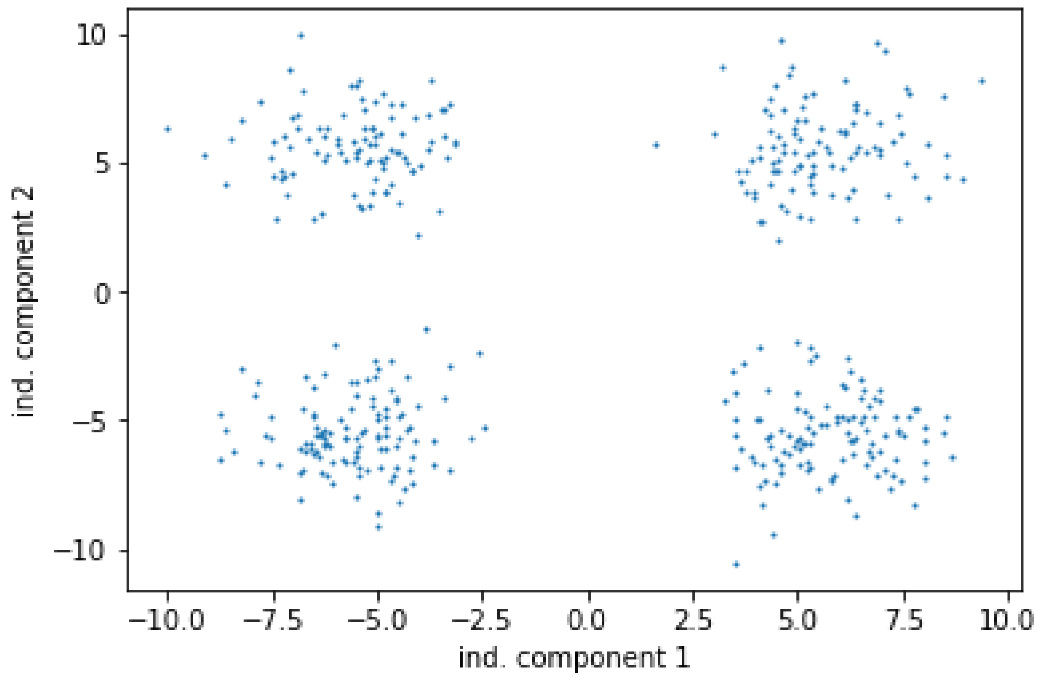
```
Iteration: 1, Objective: 206408.561582
Iteration: 2, Objective: 217528.524521
```



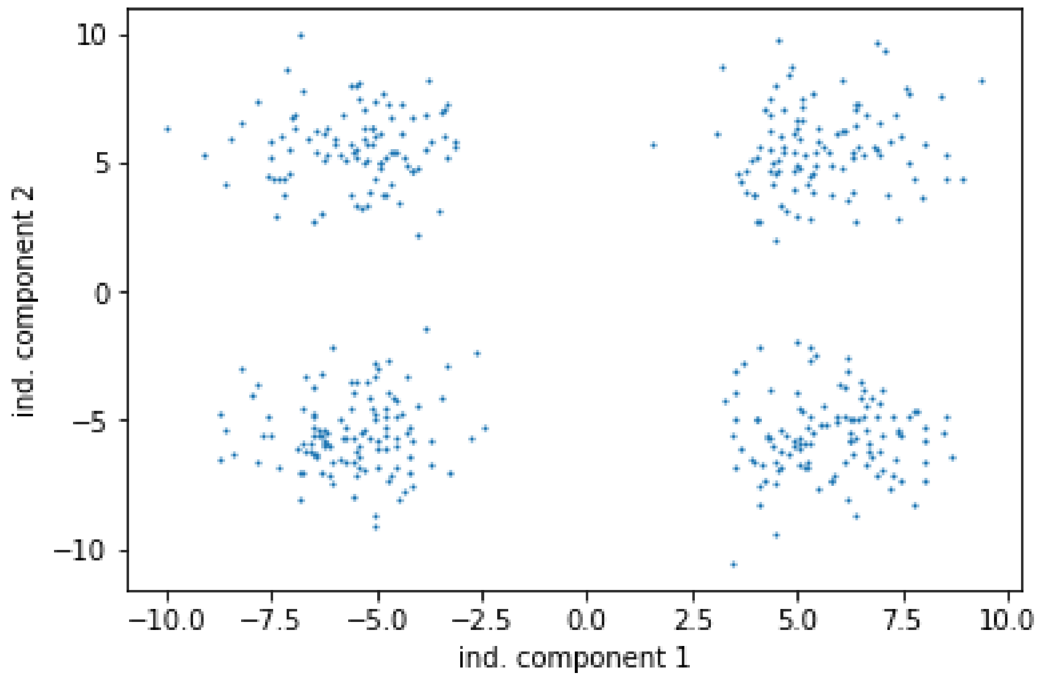
```
Iteration: 3, Objective: 218685.117412
Iteration: 4, Objective: 218816.294219
Iteration: 5, Objective: 219349.698761
Iteration: 6, Objective: 218887.633168
```



```
Iteration: 7, Objective: 219265.668693
Iteration: 8, Objective: 219268.881381
Iteration: 9, Objective: 219599.750376
Iteration: 10, Objective: 219484.667965
Iteration: 11, Objective: 219706.045862
Iteration: 12, Objective: 219051.052036
Iteration: 13, Objective: 218994.809248
Iteration: 14, Objective: 218939.358187
```

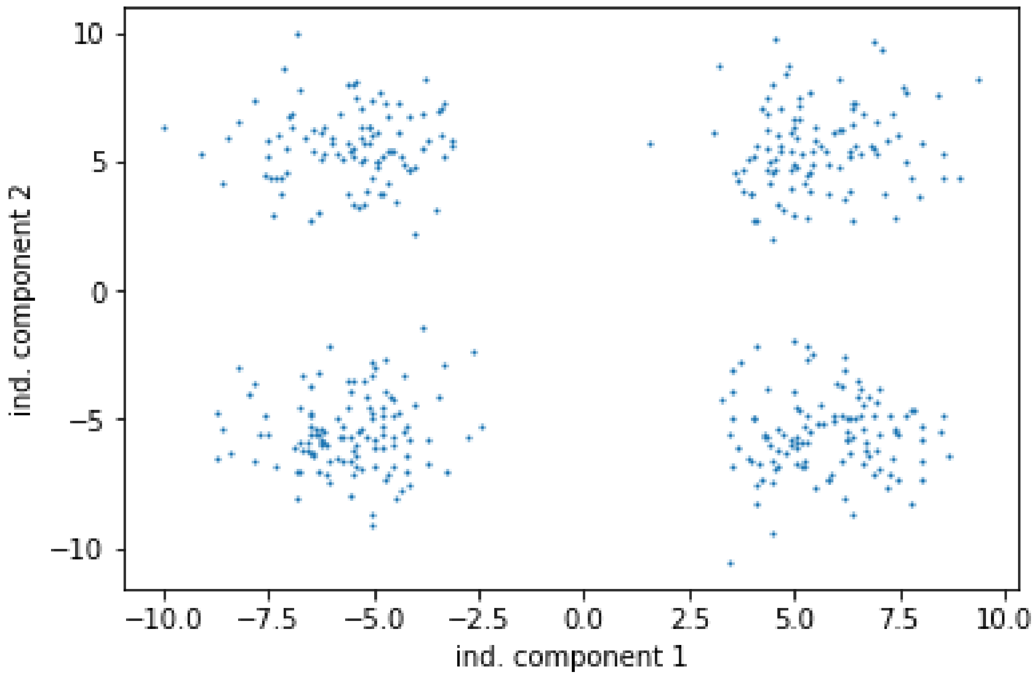


```
Iteration: 15, Objective: 219682.517032
Iteration: 16, Objective: 219225.424814
Iteration: 17, Objective: 219711.565911
Iteration: 18, Objective: 219523.108782
Iteration: 19, Objective: 219216.315244
Iteration: 20, Objective: 219411.062627
Iteration: 21, Objective: 219153.443206
Iteration: 22, Objective: 219296.610463
Iteration: 23, Objective: 219318.794666
Iteration: 24, Objective: 219611.047765
Iteration: 25, Objective: 219204.929444
Iteration: 26, Objective: 219433.186276
Iteration: 27, Objective: 219639.878783
Iteration: 28, Objective: 219027.472692
Iteration: 29, Objective: 219435.400504
Iteration: 30, Objective: 219059.753717
```



```
Iteration: 31, Objective: 219533.238124
Iteration: 32, Objective: 219172.614746
Iteration: 33, Objective: 219489.233444
Iteration: 34, Objective: 219304.514292
Iteration: 35, Objective: 219344.049501
Iteration: 36, Objective: 219437.61912
Iteration: 37, Objective: 219395.175608
Iteration: 38, Objective: 219490.230116
Iteration: 39, Objective: 219113.285342
Iteration: 40, Objective: 219226.55471
Iteration: 41, Objective: 219202.571109
Iteration: 42, Objective: 219620.421575
Iteration: 43, Objective: 219566.580404
Iteration: 44, Objective: 219194.563171
Iteration: 45, Objective: 219148.922264
Iteration: 46, Objective: 219119.054194
Iteration: 47, Objective: 219229.804164
Iteration: 48, Objective: 219361.25065
Iteration: 49, Objective: 219093.958148
Iteration: 50, Objective: 219425.533939
Iteration: 51, Objective: 219648.587575
Iteration: 52, Objective: 219352.528637
Iteration: 53, Objective: 219629.197896
Iteration: 54, Objective: 219391.089593
```

```
Iteration: 55, Objective: 219309.939456
Iteration: 56, Objective: 219607.273145
Iteration: 57, Objective: 219262.244992
Iteration: 58, Objective: 219423.476256
Iteration: 59, Objective: 219278.182073
Iteration: 60, Objective: 219604.968423
Iteration: 61, Objective: 218936.597619
Iteration: 62, Objective: 219145.591887
```



```
Iteration: 63, Objective: 219022.622571
```