Project 4 Group 2 final presentation

Table of contents

Import of Data and libraries

Libraries

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import scanpy as sc
from matplotlib import colors
from datetime import datetime
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans, MiniBatchKMeans
from sklearn.ensemble import IsolationForest
from mpl_toolkits.mplot3d import Axes3D
```

Data

In [2]:

```
data = sc.read_10x_mtx('./data/filtered_gene_bc_matrices/hg19/', var_names='gene_symbol
s', cache=True)
```

Data adjustments

Filtering

In [3]:

```
sc.pp.filter_genes(data, min_cells=1)

filtered_data = np.array(data._X.todense())
```

All genes which are not expressed at least once are removed from the dataset. Conversion of data into a numpy array for easier handling.

Removal of outliers and PCA

In [4]:

```
def remove_outliers():
    global pca_data
    X_train = pca_data
    clf = IsolationForest(behaviour="new", contamination=.07, max_samples=0.25)
    clf.fit(X_train)
    y_pred_train = clf.predict(X_train)
    pca_data = X_train[np.where(y_pred_train == 1, True, False)]

def pca(d, rmo=False):
    global dim, pca_data, pbmcs, genes

    dim = d
    pca = PCA(n_components=dim)
    pca_data = pca.fit_transform(filtered_data)
    if rmo == True:
        remove_outliers()
    print("Sum of explained variances: ""%.2f" % (sum(pca.explained_variance_ratio_)) +
"\n")

    pbmcs = pca_data.shape[0]
    genes = pca_data.shape[1]
```

PCA to reduce dimensionality of our data to d dimension. The variables *pbmcs* and *genes* describe the dimensionalty of our data. Removal of outliers by Isolation forest is an optional part of the function (parameter *rmo*).

Basic k-means

Initial centroid generation

In [5]:

```
def random_start_centroids(starttype):
    global centroids_array, genes, pbmcs, genes

    centroids_array = np.empty([0, genes])

    if starttype == "randcell":
        centroids_numbers = np.random.randint(pbmcs, size=k)
        i = 0
        while i < k:
            random_cell = centroids_numbers[i]
            centroids_array = np.append(centroids_array, [pca_data[random_cell, :]], ax
is=0)
            i += 1

    elif starttype == "randnum":
        centroids_array = (np.amax(pca_data) - np.amin(pca_data)) * np.random.random_sa
mple((k, genes)) + np.amin(
            pca_data)

    elif starttype == "k++":
        kppcentroids()
```

This function creats an empty array of centroids (*centroids_array*) which can be extended by three different methods. **"randcell"** selects k random samples from the given data. **"randnum"** creates k new centroids by creating random numbers inbetween the maximum and minimum of the data values. **"k++"** performs the k++ centroid function.

Assignment of datapoints to closest centroids

In [6]:

```
def dist(cell_point, cluster_number):

    return np.linalg.norm(pca_data[cell_point, :] - centroids_array[cluster_number - 1,
:])


def assign_centroids(data_array):
    global nearest_centroid
    i = 0
    nearest_centroid = np.zeros([data_array.shape[0], 1])

    # Loop over all datapoints
    while i < data_array.shape[0]:
        sml_distance = -1

        # Loop over every centroid
        j = 1
        while j <= k:

            if sml_distance == -1 or dist(i, j) < sml_distance:
                sml_distance = dist(i, j)
                nearest_centroid[i, 0] = j
            j += 1
        i += 1
```

dist function: returns linear distance between a datapoint i and centroid j.

assign_centroids function: creates *nearest_centroid* array in which the (number of the) closest centroid for each data point is saved.

In [7]:

```
def empty_check():
    i = 0
    while i < k:
        if list(nearest_centroid).count(i + 1) == 0:
            print("Empty cluster! Correcting centroids.")
            random_start_centroids("randnum")
            assign_centroids(pca_data)
            empty_check()
        i += 1
```

empty_check: Additional function for **randnum starttype**, to ensure that no empty clusters were generated (which would result in fewer clusters than wanted):

Generation of new centroids

In [8]:

```
def new_centroids():
    global centroids_array, centroids_oldarray, nearest_centroid_squeeze
    centroids_oldarray = centroids_array # create copy of old array for threshold funci
on
    nearest_centroid_squeeze = np.squeeze(nearest_centroid.astype(int))
    centroids_array = np.empty([0, genes])

    i = 1
    while i <= k:
        calc_means = np.mean(pca_data[nearest_centroid_squeeze == i], axis = 0)
        centroids_array = np.append(centroids_array, np.expand_dims(calc_means, axis =
0), axis = 0)
        i += 1
```

Centroids of last generation are saved as *centroids_oldarray* to calculate distance to the subsequent generation for the threshold function. Creation of second centroid generation by calculation of the mean of associated data points.

Threshholding

In [9]:

```
def improv():
    distances = []
    i = 0
    while i < k:
        d = np.linalg.norm(centroids_array[i, :] - centroids_oldarray[i,:])
        distances.append(d)
        i += 1
    c_str = np.array2string(np.array(distances), precision=2)
    print("Distances of clusters as compared to last generation: \n" + str(c_str))
```

Calculation of distance inbetween each cluster after n iterations.

Runtime function

In [10]:

```
def runtime_start():
    global t1
    t1 = datetime.now().time()


def runtime_end():
    t2 = datetime.now().time()
    fmt = '%H:%M:%S.%f'
    elapsed = str(datetime.strptime(str(t2), fmt) - datetime.strptime(str(t1), fmt))
    return str("\truntime: " + elapsed)
```

To evaluate efficiency.

Complete basic k-means

In [11]:

```
def kmeans(start, k1, n_iterations, t):
    global k
    k = k1
    i = 0
    runtime_start()

    random_start_centroids(start)
    assign_centroids(pca_data)

    if start == "randnum":
        empty_check()

    if t == None:

        while i < n_iterations:
            new_centroids()
            assign_centroids(pca_data)
            i += 1
        improv()

    else:
        count = 0
        d = t

        while d >= t:
            new_centroids()
            assign_centroids(pca_data)
            d = np.linalg.norm(centroids_oldarray-centroids_array)
            count+=1
        print("%s iterations were performed" %count)

    print("\nKMEANS:")
    print("\ngroup 4_2 algorithm:")
    print(runtime_end())
    print("\twss: " + str(wss('self')))
```

Function performing a basic k-means with **k(k1)** clusters until change of clusters is below the threshold t. If no threshold is chosen n iterations are performed.

Mini Batch

In [12]:

```
def minibatch(k1, n_iterations, b):
    global k, pca_data, nearest_centroid_squeeze, pca_data, bg, n_iterationsg, centroid
s_array, cnnew
    k = k1
    bg = b
    n_iterationsg = n_iterations
    runtime_start()
    v = np.zeros((k, 1))
    j = 1
    random_start_centroids("randcell")
    cnnew = centroids_array
    while (j <= n_iterations):

        # Reduce data to batch
        pca_batch = pca_data[np.random.randint(pca_data.shape[0], size=b), :]

        # Start centroids
        assign_centroids(pca_batch)
        i = 0

        # Centroid movement along gradient
        while (i < b):
            c = cnnew[int(nearest_centroid[i, 0])-1, :]
            v[int((nearest_centroid[i, 0]-1)), 0] =  int(v[int((nearest_centroid[i, 0]-
1)), 0]) + 1
            n = 1/v[int((nearest_centroid[i, 0]-1)), 0]
            cnnew[int(nearest_centroid[i, 0])-1, :] = c * (1-n) + pca_data[i, :] * n
            i+=1
        j+=1

    centroids_array = cnnew
    assign_centroids(pca_data)
    nearest_centroid_squeeze = np.squeeze(nearest_centroid.astype(int))
    print("\nMINI-BATCH:")
    print("\ngroup 4_2 algorithm:")
    print(runtime_end())
    print("\twss: " + str(wss('self')))
```

Minibatch algorithm was implemented as described in "Web-Scale K-Means Clustering" by D. Sculley.

K++

In [13]:

```
def kppcentroids():
    global centroids_array, dist_array, prob_array
    first_centroid = np.random.randint(pbmcs, size=1)
    i = 0
    centroids_array = np.append(centroids_array, pca_data[first_centroid, :], axis=0)
    dist_array = np.empty ([0,pbmcs])
    prob_array = np.empty ([0,pbmcs])
    j = 0

    while i < k - 1:
        z = centroids_array.shape[0] + 1
        while j < pbmcs:
            sml_distance = -1
            l = 1
            while l < z:
                if sml_distance == -1 or dist(j, l) < sml_distance:
                    sml_distance = dist(j, l)
                l += 1
            dist_array = np.append(dist_array,sml_distance **2)
            j += 1
        prob_array = dist_array / np.sum(dist_array)
        s = np.random.choice(pbmcs,p = prob_array )
        centroids_array = np.append(centroids_array, np.expand_dims(pca_data[s, :], axi
s=0), axis=0)
        i += 1
```

Creation of empty centroids array similar to randcell method. First centroid is chosen randomly. For every consecutive centroid an array of the squared distances between each data point to its closest centroids is created (*dist_array*). According probability array is created with identical dimension (*prob_array*). Centroid array is appended by selecting a random datapoint with according probabilty until k centroids are generated. Afterwards clustering is performed as in basic k-means algorithm.

Clustering Function and plotting

Sklearn k-means

In [14]:

```
def sklearn_kmeans_function(var, k, start):
    global y_sklearnkmeans, sklearn_kmeans, pca_data
    runtime_start()
    if start == "randcell" or start == "randnum":
        if var == "kmeans":
            sklearn_kmeans = KMeans(init='random', n_clusters=k).fit(pca_data)
        if var == "mini":
            sklearn_kmeans = MiniBatchKMeans(n_clusters=k, init = 'random', max_iter=n_
iterationsg, batch_size=bg).fit(pca_data)
    if start == "k++":
        if var == "kmeans":
            sklearn_kmeans = KMeans(n_clusters=k).fit(pca_data)
        if var == "mini":
            sklearn_kmeans = MiniBatchKMeans(n_clusters=k, max_iter=n_iterationsg, batc
h_size=bg).fit(pca_data)
    y_sklearnkmeans = sklearn_kmeans.predict(pca_data)
    print("\nsklearn kmeans:")
    print(runtime_end())
    print("\twss: " + str(wss('sklearn')))
```

Plotting

In [15]:

```
def plots(add = ""):
    global fig1, fig2
    # 2D plots:
    additional = ""
    if add == "mini":
        additional = " (mini-batch)"
    # Kmeans
    fig1 = plt.figure(1, figsize=[10, 5], dpi=200)
    plt1, plt2 = fig1.subplots(1, 2)
    plt1.scatter(pca_data[:, 0], pca_data[:, 1], c=nearest_centroid_squeeze, s=0.5, cma
p='viridis')
    # plt1.plot(centroids_array[:, 0], centroids_array[:, 1], markersize=5, marker="s",
linestyle='None', c='k')
    plt1.set_title('kmeans' + additional)

    # Sklearnkmeans
    plt2.scatter(pca_data[:, 0], pca_data[:, 1], c=y_sklearnkmeans, s=0.5, cmap='viridi
s')
    # plt2.plot(sklearn_kmeans.cluster_centers_[:, 0], sklearn_kmeans.cluster_centers_
[:, 1], markersize=5, marker="s", linestyle='None', c='k')
    plt2.set_title('sklearn kmeans' + additional)

    # 3D plots
    if dim >= 3:
        fig2 = plt.figure(2, figsize=[15,10], dpi=200)

        # Kmeans
        plt21 = fig2.add_subplot(221, projection = '3d')
        plt21.scatter(pca_data[:, 1], pca_data[:, 2], pca_data[:, 0], s=2, c = nearest_
centroid_squeeze, cmap='viridis')
        # plt21.plot(centroids_array[:, 0], centroids_array[:, 1], centroids_array[:,
 2], markersize=5, marker="s", linestyle='None', c='k')
        plt21.set_title('3d kmeans' + additional)

        # Sklearnkmeans
        plt22 = fig2.add_subplot(222, projection = '3d')
        plt22.scatter(pca_data[:, 1], pca_data[:, 2], pca_data[:, 0], s=2, c = y_sklear
nkmeans, cmap='viridis')
        # plt22.plot(sklearn_kmeans.cluster_centers_[:, 0], sklearn_kmeans.cluster_cent
ers_[:, 1], sklearn_kmeans.cluster_centers_[:, 2], markersize=5, marker="s", linestyle
='None', c='k')
        plt22.set_title('3D kmeans by sklearn' + additional)
```

Quality control

In [16]:

```
def wss(where):
        i = 0
        wsssum = 0
        while (i < len(pca_data)):
            if where == "self":
                assigned_centroid = int(nearest_centroid[i,0])
                centr_val = centroids_array[assigned_centroid-1]
            if where == "sklearn":
                assigned_centroid = int(y_sklearnkmeans[i])
                centr_val = sklearn_kmeans.cluster_centers_[assigned_centroid]
            point_val = pca_data[i]
            i+=1
            sqdist = np.linalg.norm(centr_val - point_val)**2
            wsssum += np.trunc(sqdist)
        return(wsssum)
```

Calculation of within-cluster sum of squares.

Final Clustering Function

In [17]:

```
def cluster(pcas = 5, rmo=True, variant = 'kmeans', start='randcell', k = 3, max_iterat
ions = 10, threshold = 0.00001, batch_size = 2000):
    pca(pcas, rmo)

    if variant == "kmeans":
        kmeans(start, k, max_iterations, threshold)
        sklearn_kmeans_function("kmeans", k, start)
        plots()

    if variant == "mini":
        minibatch(k, max_iterations, batch_size)
        sklearn_kmeans_function("mini", k, start)
        plots("mini")
```

The depicted cluster function allows us to perform pca and kmeans clustering while giving us the ability to select following parameters:

1. Amount of principal components
2. Removal of outliers (Y/N)
3. Performance of k-means or mini batch k-means algorithm
4. Method for creating the initial centroids (random cells, random numbers, k++)
5. k (number of clusters)
6. Maximum iterations (not utilizied if a threshold is defined)
7. Minimum threshhold for change of cluster centers
8. Batch size for mini batch k-means

By combining these features we are able to utilize the same function for all three subprojects.

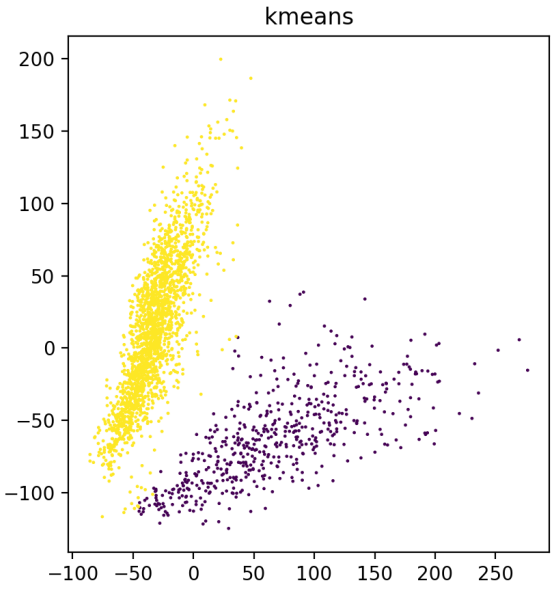
Results

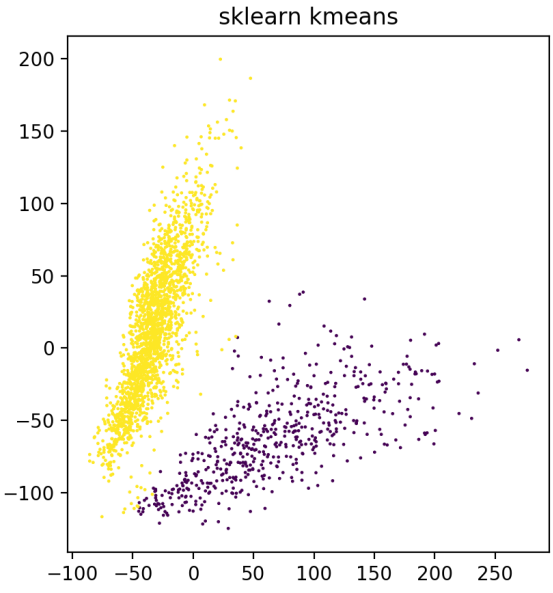
Basic k-means

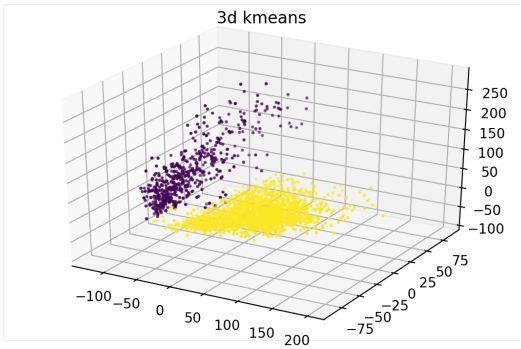
In [18]:

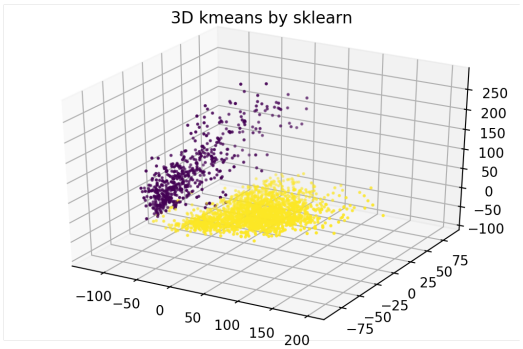
```
cluster(variant = 'kmeans', start = "randcell", k=2)
```

```
Sum of explained variances: 0.60

28 iterations were performed

KMEANS:

group 4_2 algorithm:
        runtime: 0:00:01.198798
        wss: 11523549.0

sklearn kmeans:
        runtime: 0:00:00.068815
        wss: 11523550.0
```







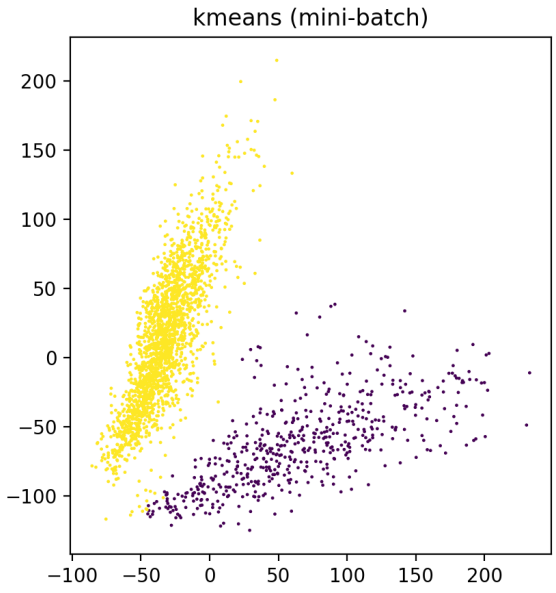


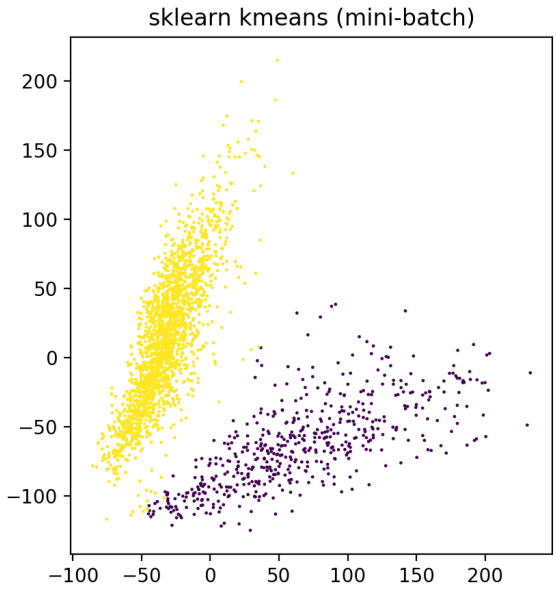
Mini batch k-means

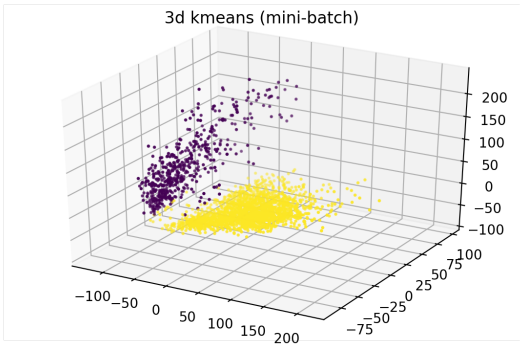
In [19]:

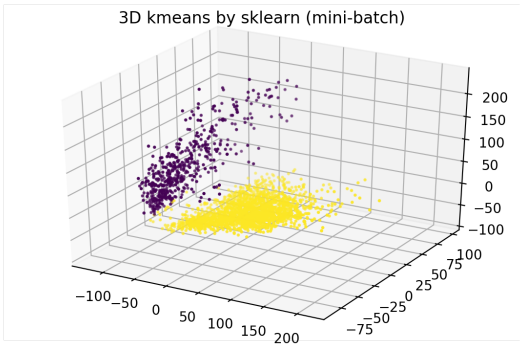
```
cluster(variant = 'mini', start = "randcell", k=2)
```

```
Sum of explained variances: 0.60


MINI-BATCH:

group 4_2 algorithm:
        runtime: 0:00:00.621338
        wss: 11485026.0

sklearn kmeans:
        runtime: 0:00:00.033911
        wss: 11482297.0
```







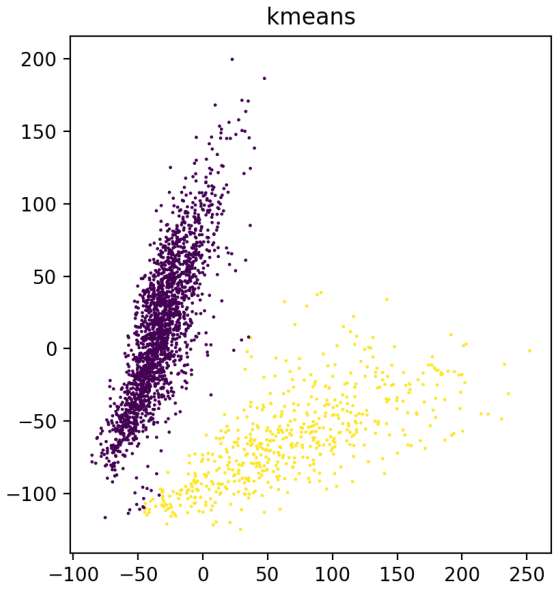


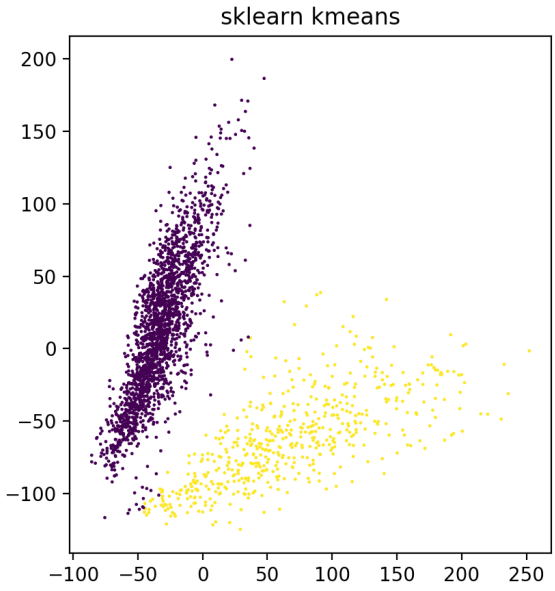
K++

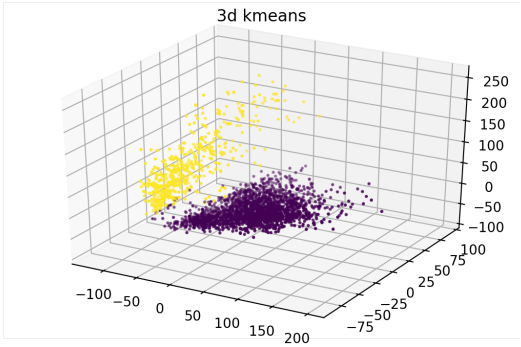
In [20]:

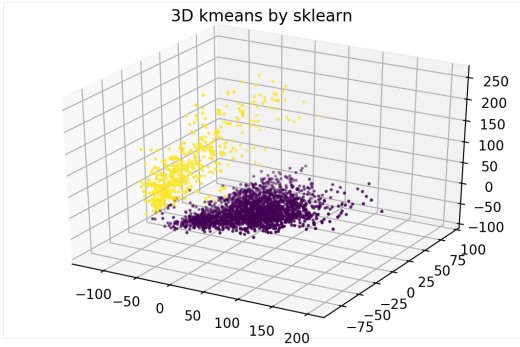
```
cluster(variant = 'kmeans', start = "k++", k=2)
```

```
Sum of explained variances: 0.60

7 iterations were performed

KMEANS:

group 4_2 algorithm:
        runtime: 0:00:00.337100
        wss: 11474363.0

sklearn kmeans:
        runtime: 0:00:00.066815
        wss: 11474362.0
```









In [ ]: