Differentially-Private Kmeans clustering
KFUPM
College of Computer Science and Engineering
Computer Engineering Department
COE 449: Privacy Enhancing Technologies

Fall 2019 (191)

Student: **Faris Hijazi s201578750**

In [14]:
```
import numpy as np
import pandas as pd
import sklearn
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import utils
import dp_kmeans

%matplotlib inline
```

In [15]:
```
epsilon = 0.1 # privacy budget E
k = 5
```

Importing the data

In [16]:
```
data_dir = './data/amzn-anon-access-samples-history-2.0.csv'
#data_dir = './data/amzn-anon-access-samples-2.0.csv'
data_dir = './data/ipums.csv'

data_raw = pd.read_csv(data_dir)

print('shape:', data_raw.shape)
print('count NaNs:\n', data_raw.isna().sum())
data_raw.head()
```

```
shape: (20000, 8)
count NaNs:
 Age             0
Gender          0
Marital         0
Race status     0
Birth place     0
Language        0
Occupation      0
Income (K)      0
dtype: int64
```

Out[16]:

	Age	Gender	Marital	Race status	Birth place	Language	Occupation	Income (K)
0	33	1	6	2	1	1	10	144
1	40	2	4	1	1	1	6	830
2	21	2	6	1	1	1	3	992
3	39	1	4	1	1	1	6	673
4	55	2	4	1	1	1	10	470

In [17]:
```
data_raw.describe()
```

Out[17]:

	Age	Gender	Marital	Race status	Birth place	Language	O
count	20000.000000	20000.000000	20000.000000	20000.000000	20000.000000	20000.000000	200
mean	41.154150	1.492000	2.663050	1.614050	41.978350	3.358600	
std	14.277047	0.499948	2.124227	1.603066	102.422734	9.861373	
min	16.000000	1.000000	1.000000	1.000000	1.000000	1.000000	
25%	30.000000	1.000000	1.000000	1.000000	1.000000	1.000000	
50%	41.000000	1.000000	1.000000	1.000000	6.000000	1.000000	
75%	51.000000	2.000000	5.000000	1.000000	36.000000	1.000000	
max	94.000000	2.000000	6.000000	9.000000	950.000000	96.000000	

In [18]:
```
# trimming the data
data_raw = data_raw.loc[:2000,:]
data_raw.shape
```

Out[18]:
```
(2001, 8)
```

In [19]:
```
# normalizing the values
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler as StdSc

pca = PCA().fit(data_raw)

# data = pd.DataFrame(StandardScaler().fit_transform(data_raw), columns=data_raw.columns)
data = pd.DataFrame(StdSc().fit_transform(pca.transform(data_raw)), columns=data_raw.columns)
```

In [20]:
```
data.head()
```

Out[20]:

	Age	Gender	Marital	Race status	Birth place	Language	Occupation	Income (K)
0	-1.301742	-0.343343	-0.717460	-0.015729	0.277231	1.855550	0.643561	-1.320834
1	1.347888	-0.190368	-0.050845	0.020103	0.322336	0.569122	-0.415865	1.170075
2	1.973914	-0.155304	-1.355969	-0.013604	1.483425	0.117719	-0.571240	1.151090
3	0.741491	-0.225390	-0.159329	0.007534	0.513705	0.360386	-0.415024	-0.955258
4	-0.042869	-0.269702	0.917052	0.043954	-0.388916	1.884025	-0.313299	1.016793

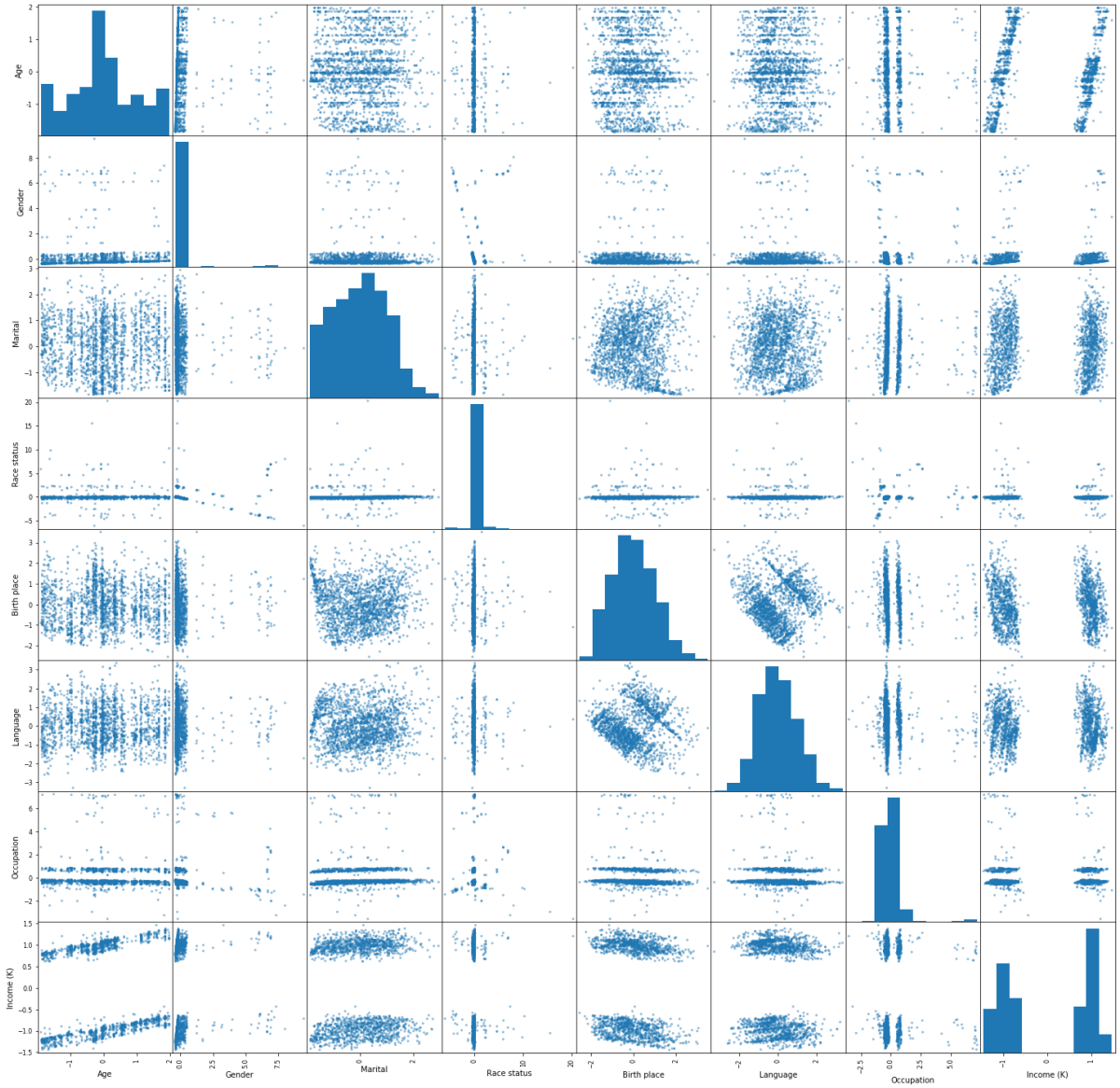
In [21]:
```
data.describe()
```

Out[21]:

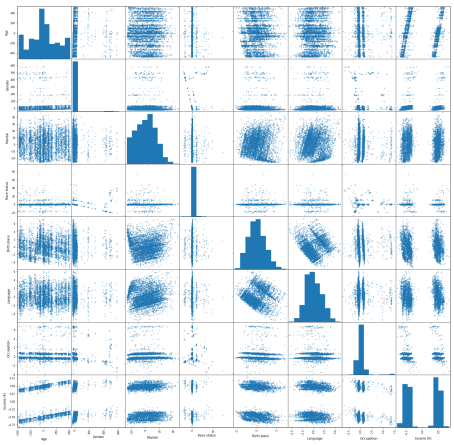
	Age	Gender	Marital	Race status	Birth place	Language
count	2.001000e+03	2.001000e+03	2.001000e+03	2.001000e+03	2.001000e+03	2.001000e+03
mean	2.729784e-17	1.446730e-17	-2.466238e-17	-2.258868e-17	3.606422e-18	-5.090603e-17
std	1.000250e+00	1.000250e+00	1.000250e+00	1.000250e+00	1.000250e+00	1.000250e+00
min	-1.885114e+00	-3.750642e-01	-1.871365e+00	-6.070064e+00	-2.544869e+00	-3.303130e+00
25%	-6.491585e-01	-2.814998e-01	-8.271756e-01	-9.237801e-02	-7.254969e-01	-7.258353e-01
50%	-3.469361e-02	-2.364295e-01	4.289488e-02	-1.810219e-02	-5.205488e-02	-8.741306e-02
75%	5.939825e-01	-1.096292e-01	7.537648e-01	1.952133e-02	7.067538e-01	6.663929e-01
max	1.973982e+00	9.485418e+00	2.947852e+00	2.025469e+01	3.511840e+00	3.351501e+00

Choosing a subset of features ['Age', 'Gender', 'Marital', 'Race status', 'Birth place', 'Language', 'Occupation', 'Income (K)']

In [22]:
```
pd.plotting.scatter_matrix(data, figsize=(25, 25));
```

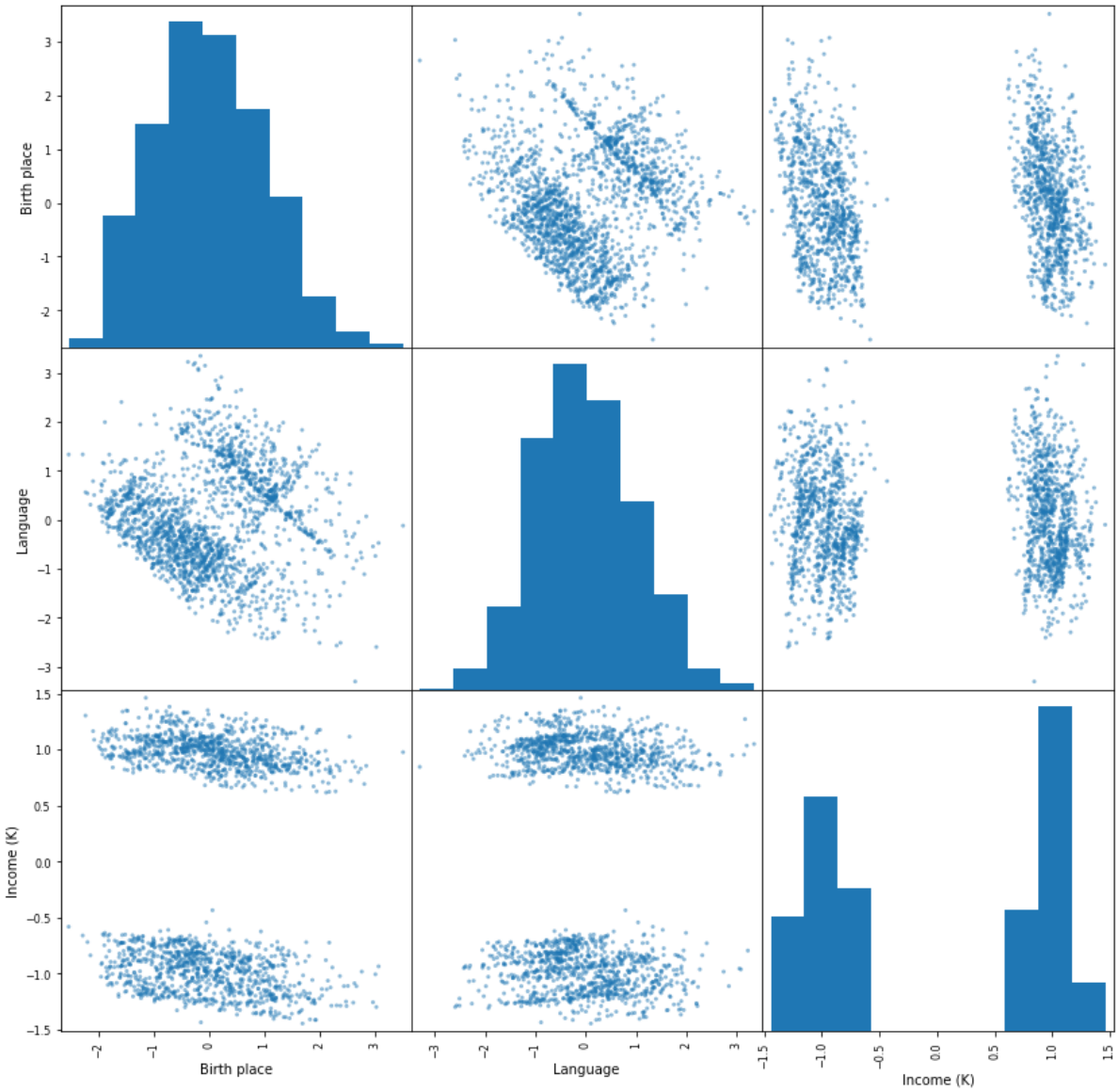


Raw Data **PCA**



```
In [23]: df = data.loc[:, ['Birth place', 'Language', 'Income (K)']]
```

In [24]:
```
pd.plotting.scatter_matrix(df, figsize=(14, 14));
```



Test on normal (vanilla) kmeans

In [25]:
```
# normal kmeans clustering
from sklearn.cluster import KMeans
from ipywidgets import interact, interactive, fixed, interact_manual, IntSlide
r


def regular_kmeans(k=k):
    clf = KMeans(n_clusters=k, random_state=0).fit(df.values)
    centroids, labels = clf.cluster_centers_, clf.labels_

    print('regular kmeans')
    utils.kmeans_matrix(df.values, k, centroids, labels, chunk_size=3, columns
=df.columns)

    return centroids, labels

interact_manual(regular_kmeans, k=IntSlider(min=2, max=20, step=1, value=k))
```

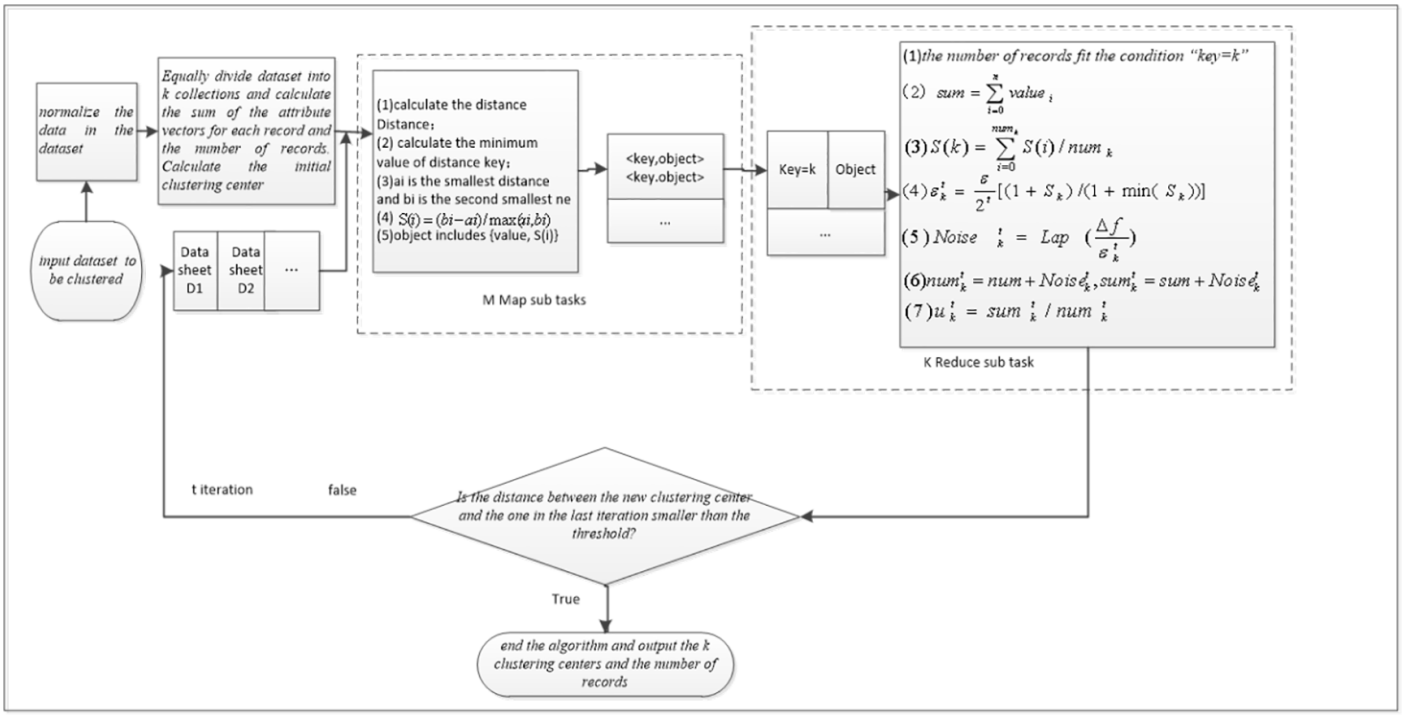
Out[25]:
```
<function __main__.regular_kmeans(k=5)>
```

Implementing Differentially Private Kmeans

This is done following the bellow diagram:

(See article (https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0206832))

image source (https://doi.org/10.1371/journal.pone.0206832.g003)



- A Convergent Differentially Private k-Means Clustering Algorithm (https://link.springer.com/chapter/10.1007/978-3-030-16148-4_47)

Algorithm

1. Divide equally to `k` clusters
2. For each cluster:
   - find `sum` of points, and the `num` (number of records)
   - find `centroids` $centroid = \frac{sum}{num}$
3. For each cluster: find S_k Add laplase noise to `sum` and `num` : $noise = Lap(\frac{\delta f}{\sum_k t})$
4. Find the new cluster centroids
5. Check condition

Basic definition. contour coefficients is a way of evaluating clustering results. The combination of cohesion and resolution can be used to evaluate the effects of different algorithms or clustering results of different operation modes based on the same original data. As for the same sample point i, the contour coefficient calculation formula is as follows:

(4)

$$S(i) = \frac{bi - ai}{\max(ai, bi)}$$

In the formula:

- `ai` represents the average similarity between sample i and other samples in the same cluster. The smaller `ai` is, more sample i should be clustered.
- `bi` represents the minimum value of the average distance from i to all samples from other clusters. That is to say, `bi = min{bi1,bi2,…,bik}` . The contour coefficient is in [–1,1].

The larger S(i) is, the closer the cluster where the point i locates is. So the average contour coefficient for each cluster is calculated as follows:

(5)

$$S(k) = \sum_{i=1}^{num_k} S(i)/num_k$$

In the formula, numk stands for the number of samples in cluster No. k. The larger the S(k) value, the better the clustering effect and vice versa.

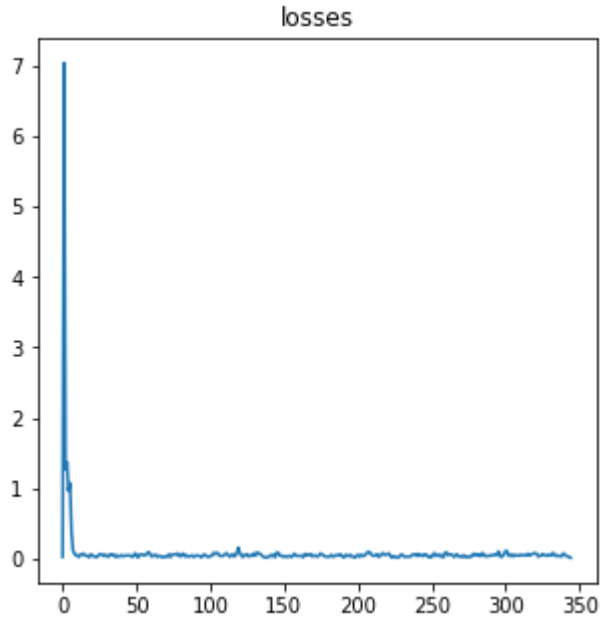
Running the algorithm

In [26]:
```
n_clusters=3
X = np.arange(9)
np.array([k for k in range(n_clusters)] * int(np.ceil(len(X) / n_clusters)), d
type=int)[:len(X)]
```

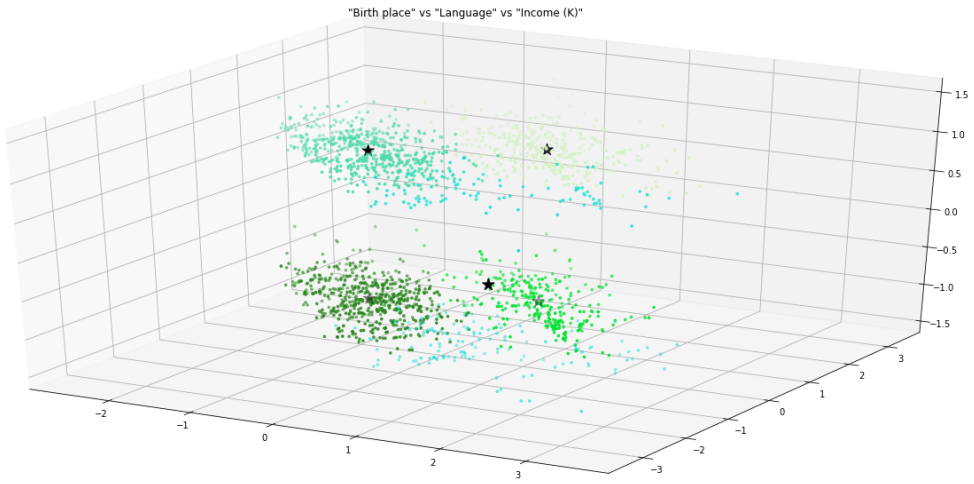
Out[26]:
```
array([0, 1, 2, 0, 1, 2, 0, 1, 2])
```

In [33]:
```
from importlib import reload; reload(utils); reload(dp_kmeans);

# dp kmeans
n_clusters = k
eps = 0.5


## generating data
from sklearn.datasets import make_blobs
X_blobs, y_blobs = make_blobs(n_samples=1000, centers=n_clusters, n_features=2
, random_state=12)
print('X_blobs.shape', X_blobs.shape)


print('DP Kmeans eps=', eps)
centroids_dp, labels_dp = dp_kmeans.kmeans(df, eps=eps, n_clusters=n_clusters,
plot=True,
                                           seed=None, verbose=False, plot_ever
y=None)
```

```
X_blobs.shape (1000, 2)
DP Kmeans eps= 0.5
Stop condition reached: current error < STOP_THRESHOLD: 0.01, stopping iterat
ions
```



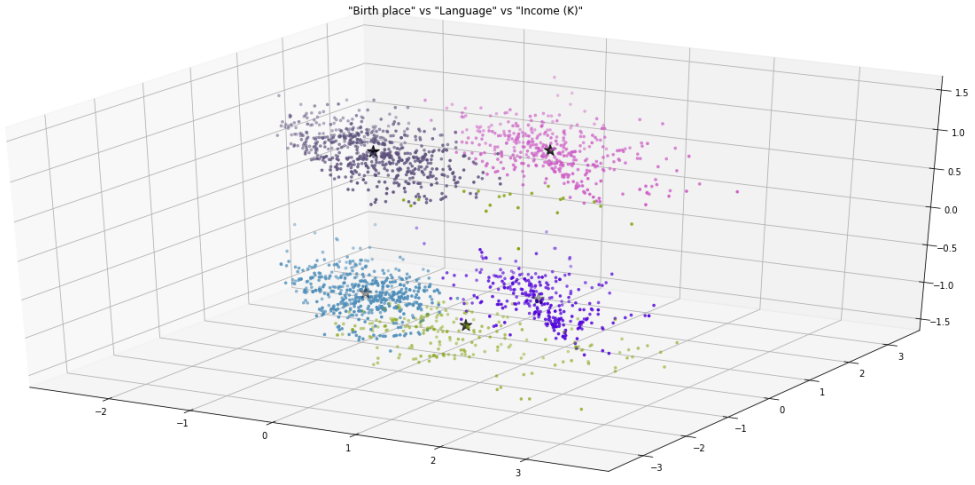

```
plotting kmeans clusters matrix...
plotting kmeans_matrix...
```



In [34]:
```
# regular kmeans
clf = KMeans(n_clusters=n_clusters, random_state=0).fit(df.values)
centroids_reg, labels_reg = clf.cluster_centers_, clf.labels_
```

In [35]:
```
print('Regular Kmeans')
utils.kmeans_matrix(df, n_clusters, centroids_reg, labels_reg, chunk_size=3);
```

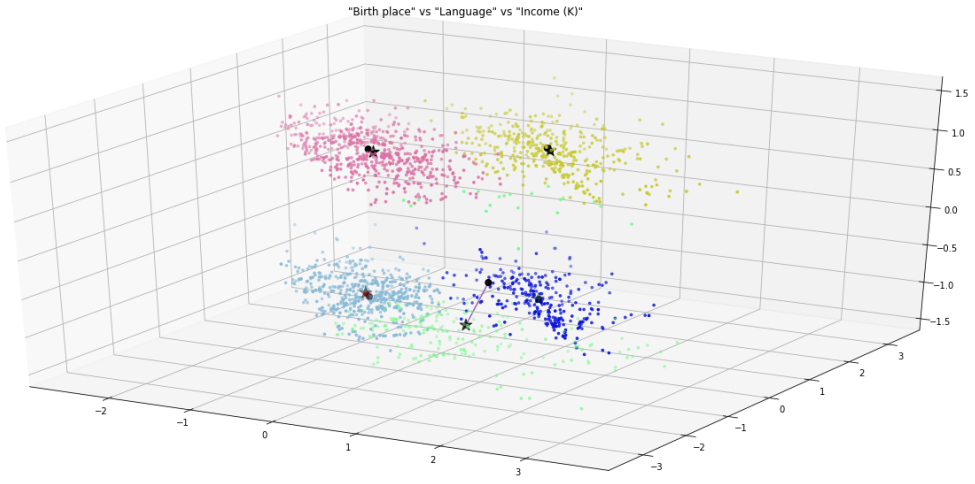
```
Regular Kmeans
plotting kmeans_matrix...
```



Comparing results

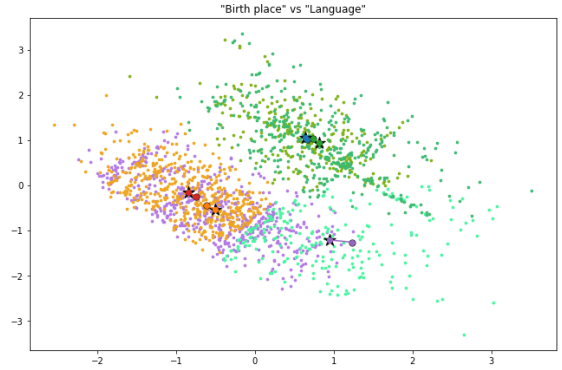
In [36]:
```
utils.kmeans_matrix_compare(df, n_clusters, centroids_reg, labels_reg, centroi
ds_dp, labels_dp, chunk_size=3);
```

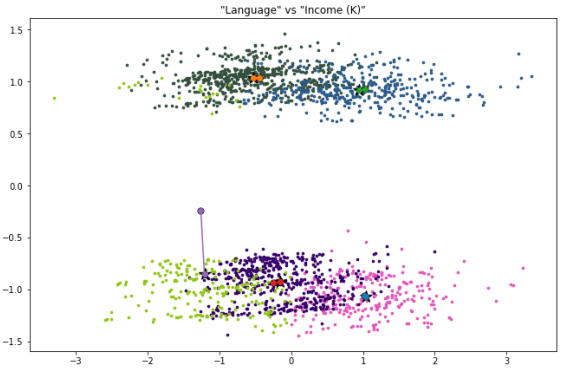
```
plotting comparison kmeans_matrix...
```



In [37]:
```
utils.kmeans_matrix_compare(df, n_clusters, centroids_reg, labels_reg, centroi
ds_dp, labels_dp, chunk_size=2);
```

```
plotting comparison kmeans_matrix...
```

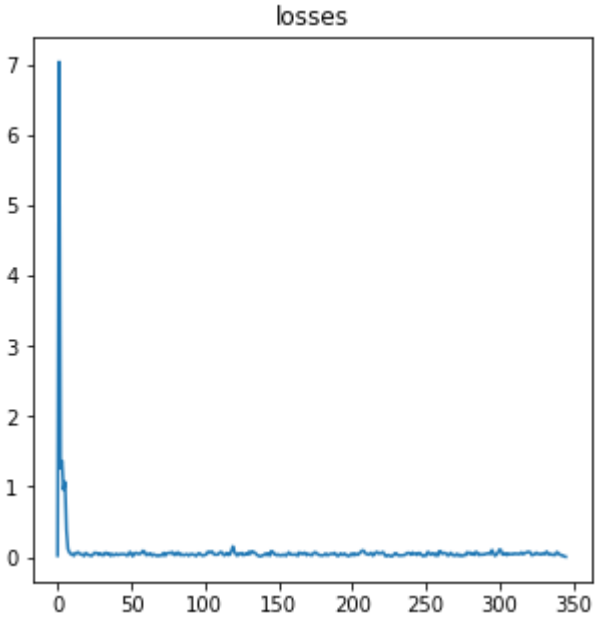




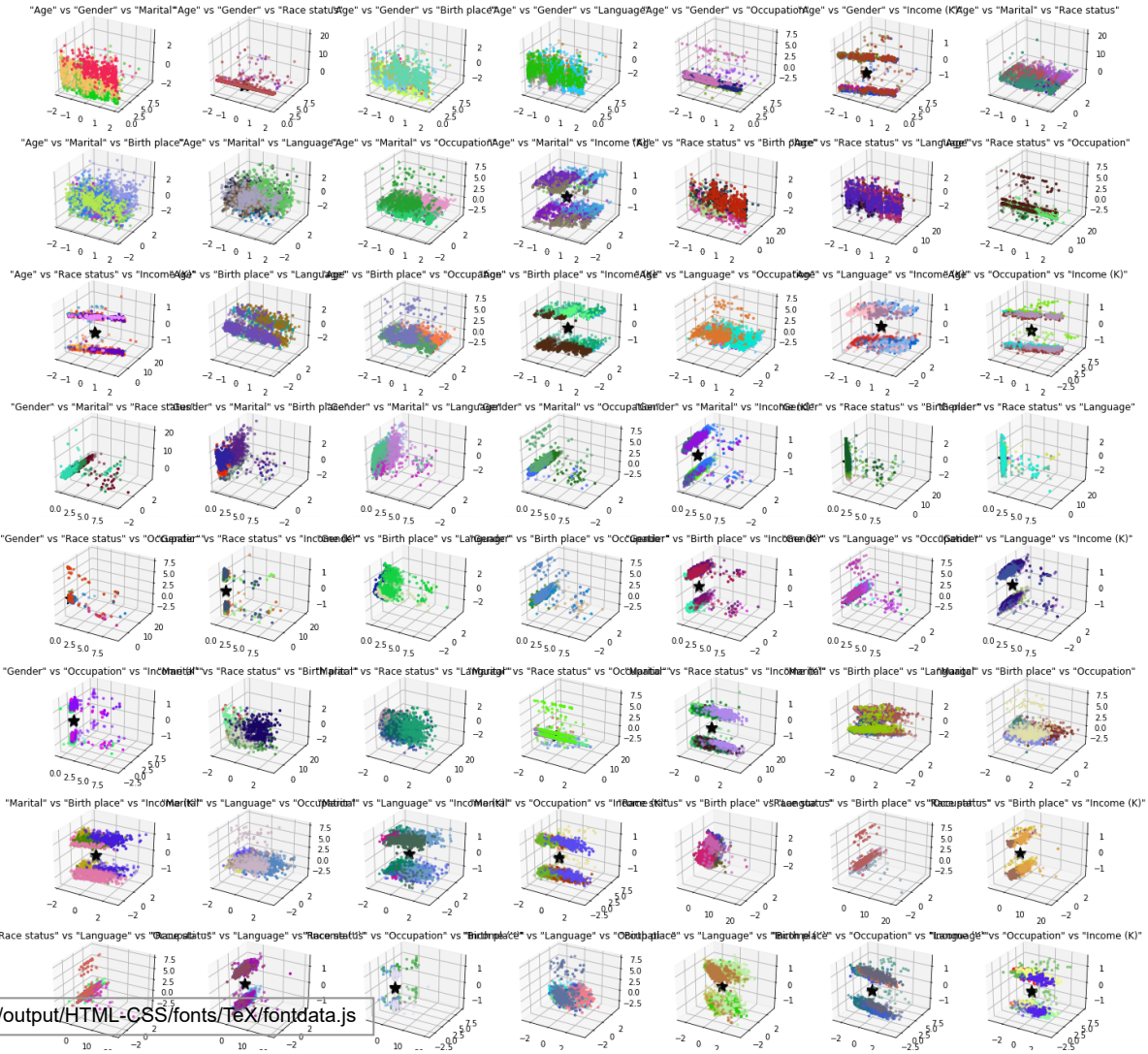
running on all features

In [38]:
```
centroids_dp, labels_dp = dp_kmeans.kmeans(data, eps=0.1, n_clusters=n_clusters, MAX_LOOPS=100, seed=0)

print('centroids:', centroids_dp.shape)
```

```
Stop condition reached: current error < STOP_THRESHOLD: 0.01, stopping iterat
ions
```



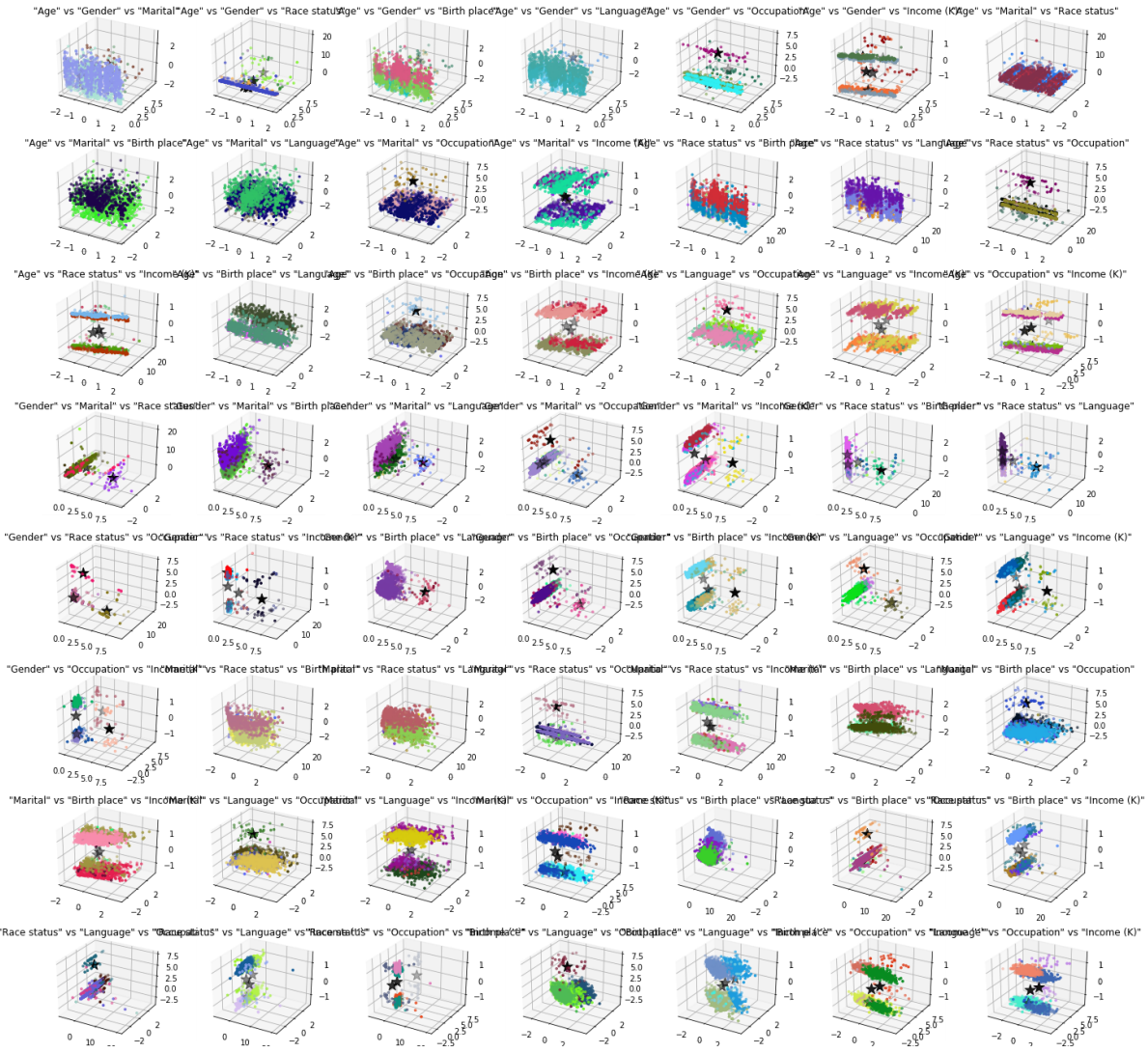

```
plotting kmeans clusters matrix...
plotting kmeans_matrix...
```



```
centroids: (5, 8)
```

In [39]:
```
clf = KMeans(n_clusters=n_clusters, random_state=0).fit(data)
centroids_reg, labels_reg = clf.cluster_centers_, clf.labels_

utils.kmeans_matrix(data, n_clusters, centroids_reg, labels_reg, chunk_size=3
);
```

```
plotting kmeans_matrix...
```

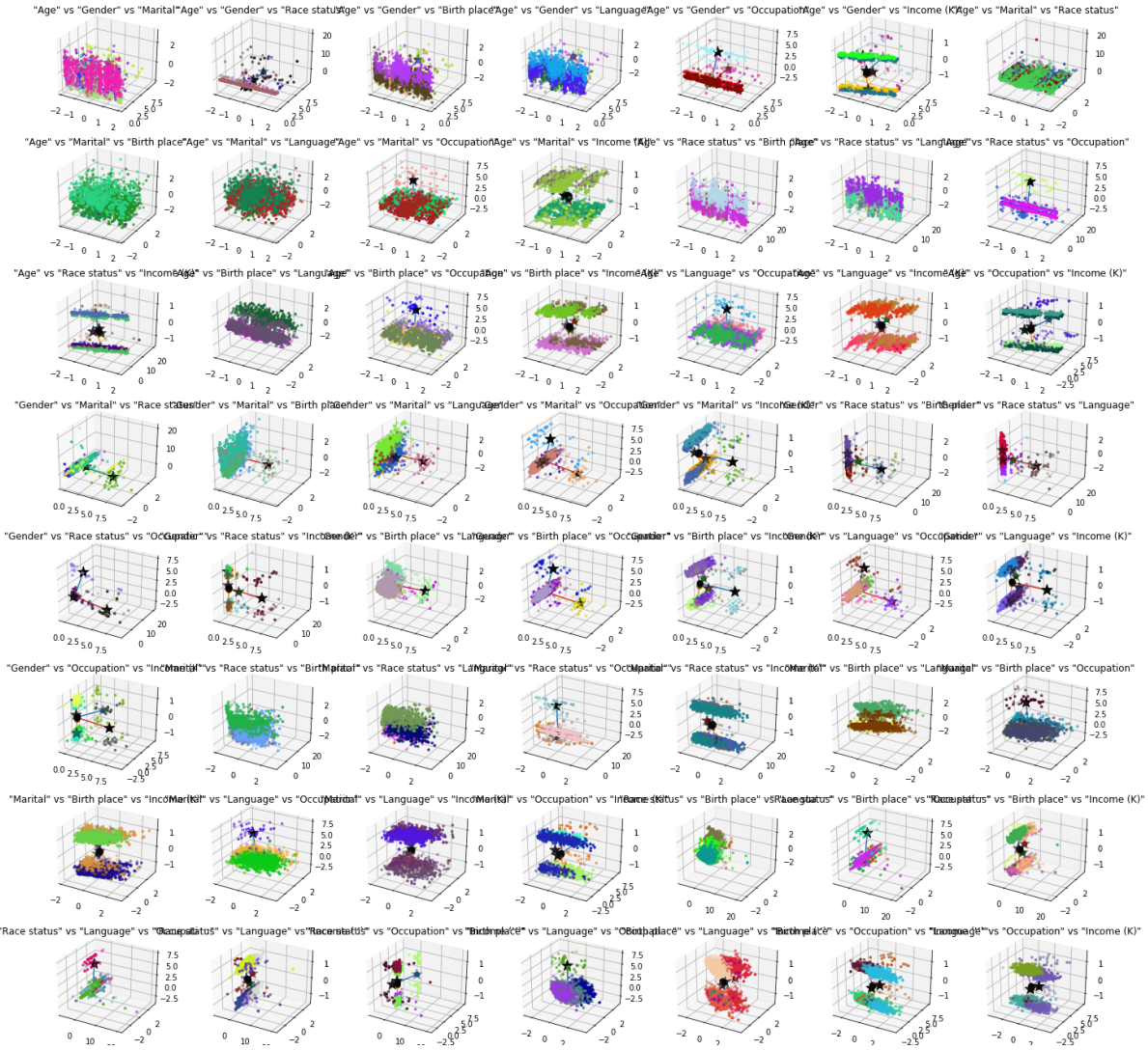


Comparison

We compare the results between the normal kmeans and the DP kmeans, the bellow plot shows which centroids moved where. This technique guesses the corresponding centroids by pairing the closest ones.

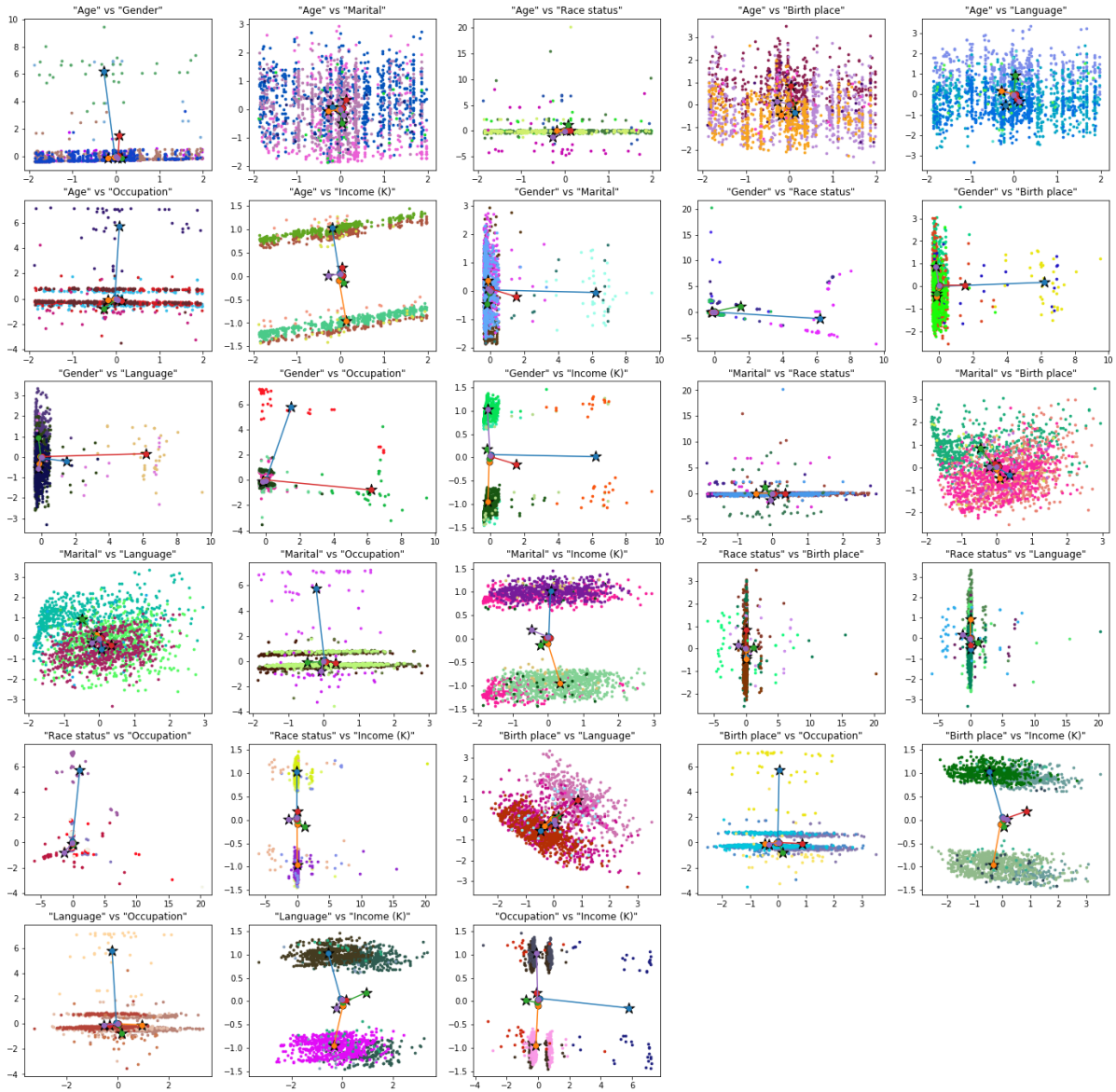
In [40]:
```
# showing the changes in centroids
utils.kmeans_matrix_compare(data, n_clusters, centroids_reg, labels_reg, centr
oids_dp, labels_dp, chunk_size=3);
```

```
plotting comparison kmeans_matrix...
```



In [41]:
```
utils.kmeans_matrix_compare(data, n_clusters, centroids_reg, labels_reg, centr
oids_dp, labels_dp);
```

```
plotting comparison kmeans_matrix...
```



Measuring performance

Clustering is an unsupervised problem, and we can't measure the performance if we don't have the ground truth values.

Davies Bouldin score

However some metrics can still be used even if the true labels are unkown, such as the [Davies Bouldin score (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies_bouldin_score.html#sklearn.metrics.davies_bouldin_sc](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies_bouldin_score.html#sklearn.metrics.davies_bouldin_sc)

(Lower is better)

In [42]:
```
from sklearn import metrics
```

In [43]:
```
print('regular kmeans davies_bouldin_score:', metrics.davies_bouldin_score(dat
a.values, labels_reg))

print('dp kmeans davies_bouldin_score:', metrics.davies_bouldin_score(data.val
ues, labels_dp))
```

```
regular kmeans davies_bouldin_score: 1.4575597023312743
dp kmeans davies_bouldin_score: 2.4019529367597015
```

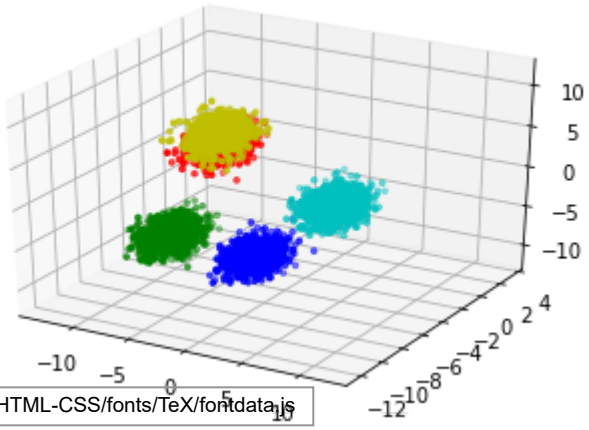
Generating Blobs

Another way to get by this problem, is to generate data with known labels, just for measuring performance. Notice that we need to choose the same number of centers that we will be passing to the kmeans (in this case 5)

I made sure to choose a seed where the clusters aren't intersecting too much, `random_state=12` seemed fine.

In [54]:
```
from sklearn.datasets import make_blobs
n_features = 3
n_clusters = 5

X_blobs, y_blobs = make_blobs(n_samples=5000, centers=n_clusters, n_features=n
_features, random_state=9)
print(X_blobs.shape, y_blobs.shape)
labels_true = y_blobs


## plotting
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

colors = ['r', 'g', 'b', 'y', 'c', 'm']
for k_ in range(n_clusters):
    points = np.array([X_blobs[j] for j in range(len(X_blobs)) if y_blobs[j]==
k_])
    ax.scatter(*points.T, s=7, c=colors[k_])
```

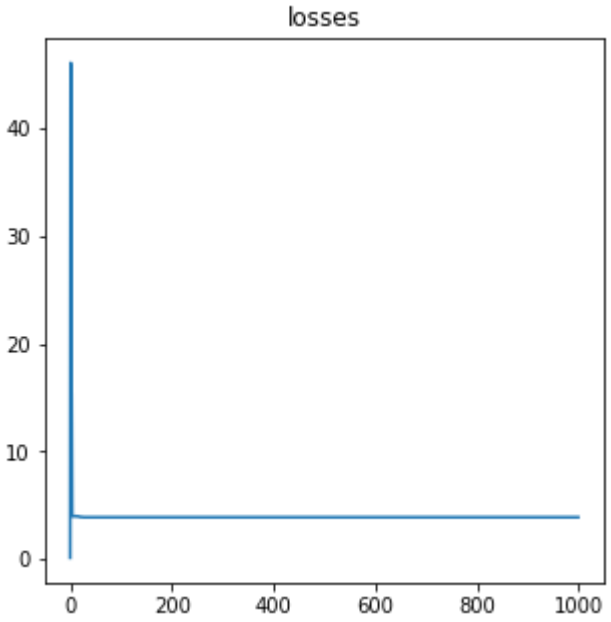
```
(5000, 3) (5000,)
```



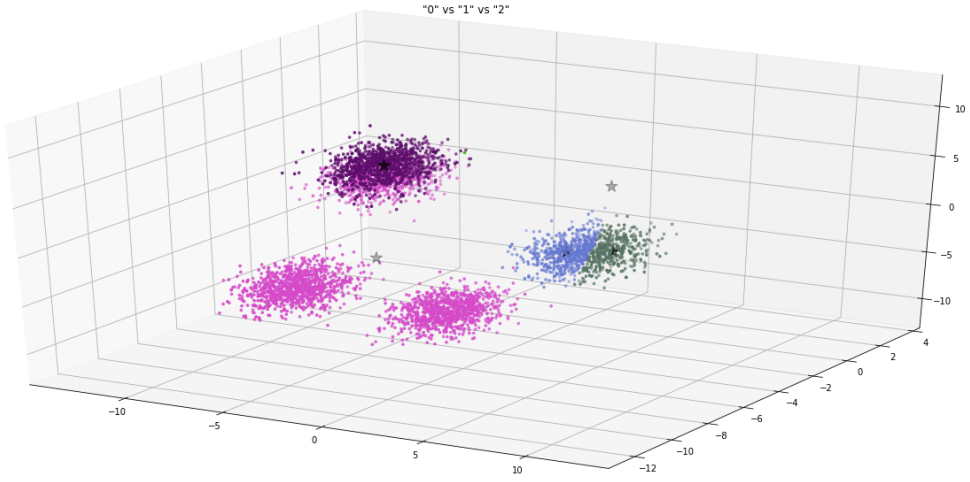
In [55]:
```
clf = KMeans(n_clusters=n_clusters, random_state=0).fit(X_blobs)
centroids_reg2, labels_reg2 = clf.cluster_centers_, clf.labels_
#executed in 40ms
```

In [56]:
```
from importlib import reload; reload(utils); reload(dp_kmeans);
centroids_dp2, labels_dp2 = dp_kmeans.kmeans(X_blobs, eps=0.5, n_clusters=n_cl
usters, STOP_THRESHOLD=0.001, MAX_LOOPS=1000, seed=42)
```

```
Stop condition reached: MAX_LOOPS exceeded: 1000, stopping iterations
```




```
plotting kmeans clusters matrix...
plotting kmeans_matrix...
```



In [57]:
```
print('adjusted rand scores')
print('regular kmeans:\t', metrics.adjusted_rand_score(labels_true, labels_reg
2)*100,'%')
print('dp kmeans:\t', metrics.adjusted_rand_score(labels_true, labels_dp2)*100
,'%')
```

```
adjusted rand scores
regular kmeans:  100.0 %
dp kmeans:       42.58404344219056 %
```

In [58]:
```
print('v_measure_score')
print('regular kmeans:\t', metrics.v_measure_score(labels_true, labels_reg2)*1
00,'%')
print('dp kmeans:\t', metrics.v_measure_score(labels_true, labels_dp2)*100,'%'
)
```

```
v_measure_score
regular kmeans:  100.0 %
dp kmeans:       70.46043245158621 %
```