Customer Segmentation using Clustering

This mini-project is based on this blog post by yhat. Please feel free to refer to the post for additional information, and solutions.

In [1]:

```
%matplotlib inline
import pandas as pd
import sklearn
from sklearn import cluster
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Setup Seaborn
sns.set_style("whitegrid")
sns.set_context("poster")
```

Data

The dataset contains information on marketing newsletters/e-mail campaigns (e-mail offers sent to customers) and transaction level data from customers. The transactional data shows which offer customers responded to, and what the customer ended up buying. The data is presented as an Excel workbook containing two worksheets. Each worksheet contains a different dataset.

In [2]:

```
df_offers = pd.read_excel("C:/WineKMC.xlsx", sheetname=0)
df_offers.columns = ["offer_id", "campaign", "varietal", "min_qty", "discou
nt", "origin", "past_peak"]
df_offers.head()
```

Out[2]:

	offer_id	campaign	varietal	min_qty	discount	origin	past_peak
0	1	January	Malbec	72	56	France	False
1	2	January	Pinot Noir	72	17	France	False
2	3	February	Espumante	144	32	Oregon	True
3	4	February	Champagne	72	48	France	True
4	5	February	Cabernet Sauvignon	144	44	New Zealand	True

We see that the first dataset contains information about each offer such as the month it is in effect and several attributes about the wine that the offer refers to: the variety, minimum quantity, discount, country of origin and whether or not it is past peak. The second dataset in the second worksheet contains transactional data -- which offer each customer responded to.

In [3]:

```
df_transactions = pd.read_excel("C:/WineKMC.xlsx", sheetname=1)
df_transactions.columns = ["customer_name", "offer_id"]
df_transactions['n'] = 1
df_transactions.head()
```

Out[3]:

	customer_name	offer_id	n
0	Smith	2	1
1	Smith	24	1
2	Johnson	17	1
3	Johnson	24	1
4	Johnson	26	1

Data wrangling

We're trying to learn more about how our customers behave, so we can use their behavior (whether or not they purchased something based on an offer) as a way to group similar minded customers together. We can then study those groups to look for patterns and trends which can help us formulate future offers.

The first thing we need is a way to compare customers. To do this, we're going to create a matrix that contains each customer and a 0/1 indicator for whether or not they responded to a given offer.

Checkup Exercise Set I

Exercise: Create a data frame where each row has the following columns (Use the pandas [`merge`](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.merge.html) and [`pivot_table`](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.pivot_table.html) functions for this purpose):

- customer_name
- One column for each offer, with a 1 if the customer responded to the offer

Make sure you also deal with any weird values such as `NaN`. Read the documentation to develop your solution.

In [4]:

```
#your turn

# join the offers and transactions table
df = pd.merge(df_offers, df_transactions)
df.head()
```

```
# create a "pivot table" which will give us the number of times each custom
er responded to a given offer
cust_offer= df.pivot_table(index=['customer_name'], columns=['offer_id'], v
alues='n')

# filling NaN with 0s
cust_offer = cust_offer.fillna(0.0)

cust_offer.head(10)

# cust_offer.shape
```

Out[4]:

offer_id	1	2	3	4	5	6	7	8	9	10	...	23	24	25	26	27	28	29	30
customer_name																			
Adams	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0
Allen	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
Anderson	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
Bailey	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
Baker	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Barnes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Bell	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0
Bennett	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
Brooks	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Brown	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0

10 rows × 32 columns

K-Means Clustering

Recall that in K-Means Clustering we want to *maximize* the distance between centroids and *minimize* the distance between data points and the respective centroid for the cluster they are in. True evaluation for unsupervised learning would require labeled data; however, we can use a variety of intuitive metrics to try to pick the number of clusters K. We will introduce two methods: the Elbow method, the Silhouette method and the gap statistic.

Choosing K: The Elbow Sum-of-Squares Method

The first method looks at the sum-of-squares error in each cluster against K. We compute the distance from each data point to the center of the cluster (centroid) to which the data point was assigned.
$$SS = \sum_k \sum_{x_i \in C_k} \sum_{x_j \in C_k} \left(x_i - x_j \right)^2 = \sum_k \sum_{x_i \in C_k} \left(x_i - \mu_k \right)^2$$

where x_i is a point, C_k represents cluster k and μ_k is the centroid for cluster k. We can plot SS vs. K and choose the *elbow point* in the plot as the best value for K. The elbow point

is the point at which the plot starts descending much more slowly.

Checkup Exercise Set II

Exercise:

- What values of SS do you believe represent better clusterings? Why?
- Create a numpy matrix `x_cols` with only the columns representing the offers (i.e. the 0/1 colums)
- Write code that applies the [`KMeans`](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html) clustering method from scikit-learn to this matrix.
- Construct a plot showing SS for each K and pick K using this plot. For simplicity, test $2 \le K \le 10$.
- Make a bar chart showing the number of points in each cluster for k-means under the best K.
- What challenges did you experience using the Elbow method to pick K?

In [5]:

```
#Create a numpy matrix x_cols with only the columns representing the
offers (i.e. the 0/1 colums)
import numpy as np
x_cols = np.matrix(cust_offer)
x_cols
```

Out[5]:

```
matrix([[ 0.,  0.,  0., ...,  1.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        ...,
        [ 1.,  0.,  0., ...,  0.,  1.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  1.,  1.]])
```

In [6]:

```
# Write code that applies the KMeans clustering method from scikit-learn to
this matrix.
from sklearn import cluster
SS_scores = pd.DataFrame(columns=['K', 'SS_score'])
SS_scores['K'] = range(2, 11)
SS_scores
K = range(2,11)
for k in K:
    kmeans = sklearn.cluster.KMeans(n_clusters = k)
    kmeans.fit(x_cols)
    SS_scores['SS_score'][SS_scores.K == k] = kmeans.inertia_
SS_scores
```

```
C:\Users\anands\AppData\Local\Continuum\Anaconda3\lib\site-
packages\ipykernel\__main__.py:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

```
See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/
stable/indexing.html#indexing-view-versus-copy
```

Out[6]:

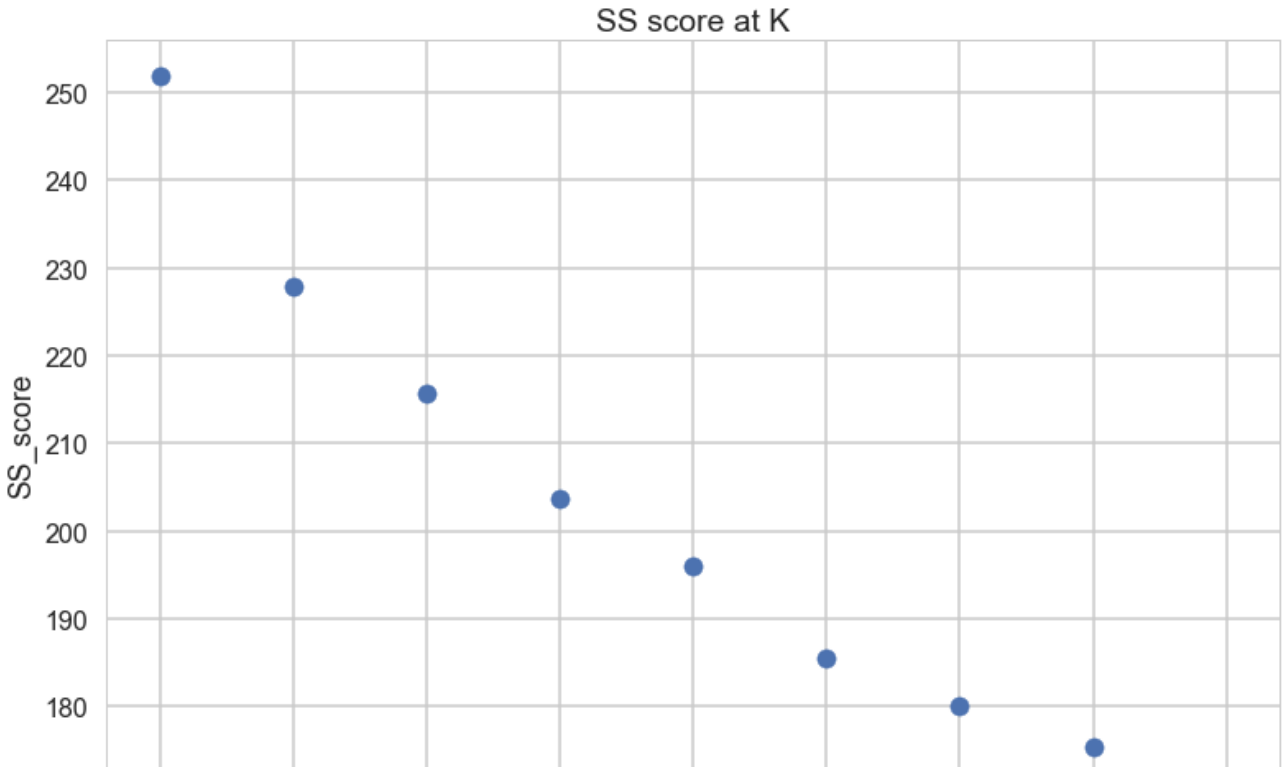
	K	SS_score
0	2	251.862
1	3	227.793
2	4	215.715
3	5	203.768
4	6	195.937
5	7	185.583
6	8	179.998
7	9	175.404
8	10	170.369

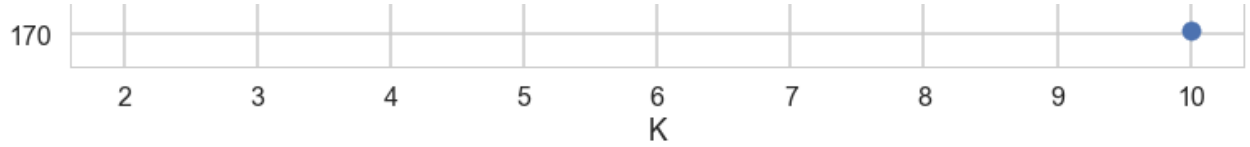
In [7]:

```
# Construct a plot showing  SSSS  for each  KK  and pick  KK  using this pl
ot. For simplicity, test  2≤K≤102≤K≤10
# import pylab as pl
plt.plot()
plt.scatter(data=SS_scores, x='K', y='SS_score')
# plt.plot(x,y,'go-')
plt.xlabel("K")
plt.ylabel("SS_score")
plt.title("SS score at K")
# plt.show()
```

Out[7]:

```
<matplotlib.text.Text at 0xaf42240>
```





In [9]:

```
# testing at n_clusters = 3 as we see the line steeping at 3
from sklearn import cluster
import numpy
kmeans = sklearn.cluster.KMeans(n_clusters = 3)
cust_offer['kmeans'] = kmeans.fit_predict(x_cols)
cust_offer.kmeans.value_counts()
```

Out[9]:

```
2    48
0    36
1    16
Name: kmeans, dtype: int64
```

In [10]:

```
# more number of data points centered around cluster 1 above, it will be in
teresting to see the groupings at n = 4, 5 and 6
#n_clusters = 4
from sklearn import cluster
kmeans = sklearn.cluster.KMeans(n_clusters=4)
cust_offer['kmeans'] = kmeans.fit_predict(x_cols)
cust_offer.kmeans.value_counts()
```

Out[10]:

```
1    35
3    31
2    18
0    16
Name: kmeans, dtype: int64
```

In [11]:

```
# n_clusters = 5
kmeans = sklearn.cluster.KMeans(n_clusters=5)
cust_offer['kmeans'] = kmeans.fit_predict(x_cols)
cust_offer.kmeans.value_counts()
```

Out[11]:

```
2    33
1    21
0    17
4    15
3    14
Name: kmeans, dtype: int64
```

In [12]:

```
# n_clusters = 6
kmeans =sklearn.cluster.KMeans(n_clusters=6)
cust_offer['kmeans'] = kmeans.fit_predict(x_cols)
cust_offer.kmeans.value_counts()
```

Out[12]:

Out[12]:

```
4    27
1    26
5    16
0    14
2    13
3     4
Name: kmeans, dtype: int64
```

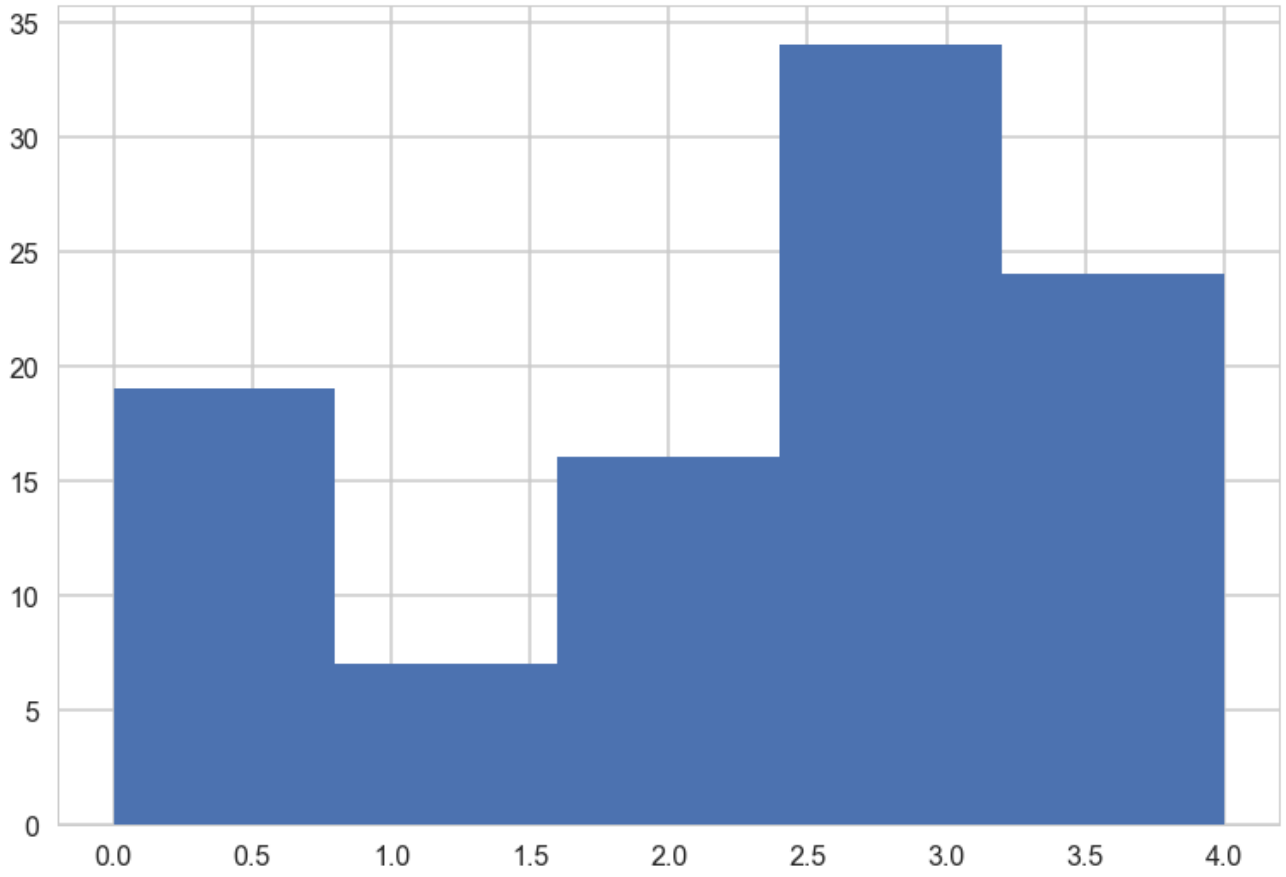
We notice a continuous steep in the curve from one pt to another . I am going to choose n_cluster = 5 as the data points look more distributed as compared to n = 3,4 and 6

In [15]:

```
# Make a bar chart showing the number of points in each cluster for k-means
under the best  K .

#Bar graph of number of points in each cluster under K=5
cluster_hist = sklearn.cluster.KMeans(n_clusters = 5)
cluster_hist.fit(x_cols)
plt.hist(x=cluster_hist.labels_, bins=5)
```

Out[15]:

```
(array([ 19.,   7.,  16.,  34.,  24.]),
 array([ 0. ,  0.8,  1.6,  2.4,  3.2,  4. ]),
 <a list of 5 Patch objects>)
```



Choosing K: The Silhouette Method

There exists another method that measures how well each datapoint x_i "fits" its assigned cluster *and also* how poorly it fits into other clusters. This is a different way of looking at the same objective.

Denote a_{x_i} as the *average* distance from x_i to all other points within its own cluster k. The lower the value, the better. On the other hand b_{x_i} is the minimum average distance from x_i to points in a different cluster, minimized over clusters. That is, compute separately for each cluster the average distance from x_i to the points within that cluster, and then take the minimum. The silhouette $s(x_i)$ is defined as
$$s(x_i) = \frac{b_{x_i} - a_{x_i}}{\max{\left(a_{x_i}, b_{x_i}\right)}}$$

The silhouette score is computed on *every datapoint in every cluster*. The silhouette score ranges from -1 (a poor clustering) to +1 (a very dense clustering) with 0 denoting the situation where clusters overlap. Some criteria for the silhouette coefficient is provided in the table below.

Range	Interpretation
0.71 - 1.0	A strong structure has been found.
0.51 - 0.7	A reasonable structure has been found.
0.26 - 0.5	The structure is weak and could be artificial.
< 0.25	No substantial structure has been found.

</pre>
Source: http://www.stat.berkeley.edu/~spector/s133/Clus.html

Fortunately, scikit-learn provides a function to compute this for us (phew!) called `sklearn.metrics.silhouette_score`. Take a look at this article on picking K in scikit-learn, as it will help you in the next exercise set.

Checkup Exercise Set III

Exercise: Using the documentation for the `silhouette_score` function above, construct a series of silhouette plots like the ones in the article linked above.

Exercise: Compute the average silhouette score for each K and plot it. What K does the plot suggest we should choose? Does it differ from what we found using the Elbow method?

In [16]:

```
# Your turn.
# choosing SS with the Silhouette_score Method
sil_scores = pd.DataFrame(columns=['K', 'sil_score'])
sil_scores['K'] = range(2, 11)
for K in range(2, 11):
    kmeans = sklearn.cluster.KMeans(n_clusters = K, random_state=3)
```

```
    kmeans = sklearn.cluster.KMeans(n_clusters = K, random_state=3)
    kmeans.fit_predict(x_cols)
    sil_scores['sil_score'][sil_scores.K == K] =
sklearn.metrics.silhouette_score(x_cols, kmeans.labels_, random_state=3)
sil_scores
```

```
C:\Users\anands\AppData\Local\Continuum\Anaconda3\lib\site-
packages\ipykernel\__main__.py:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/
stable/indexing.html#indexing-view-versus-copy
```

Out[16]:

	K	sil_score
0	2	0.0917487
1	3	0.107273
2	4	0.133192
3	5	0.130497
4	6	0.134193
5	7	0.136471
6	8	0.115994
7	9	0.124564
8	10	0.127608

In [17]:

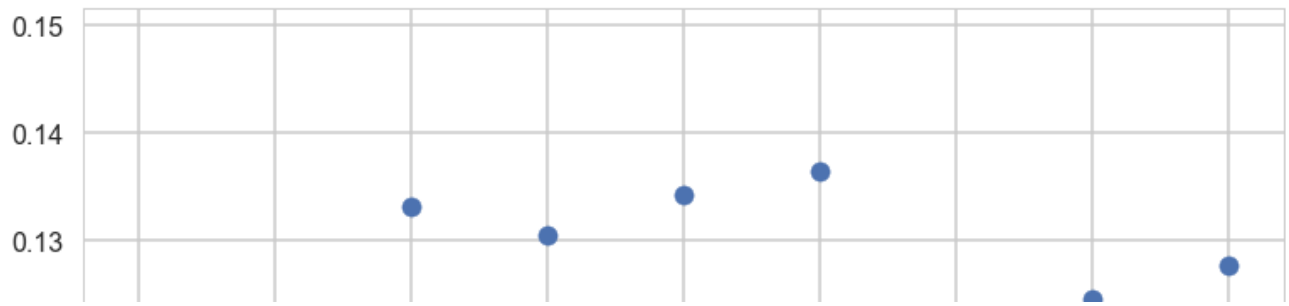
```
# from sklearn.cluster import KMeans
# from sklearn import metrics
# for n_clusters in range(2, 11):
#     km = KMeans(n_clusters=n_clusters, init='k-means++', max_iter=100, n_
init=1)
#     pred = km.fit_predict(x_cols)
#     silhouette_avg = metrics.silhouette_score(x_cols, pred,
sample_size=100)

#     print (n_clusters, silhouette_avg)
```

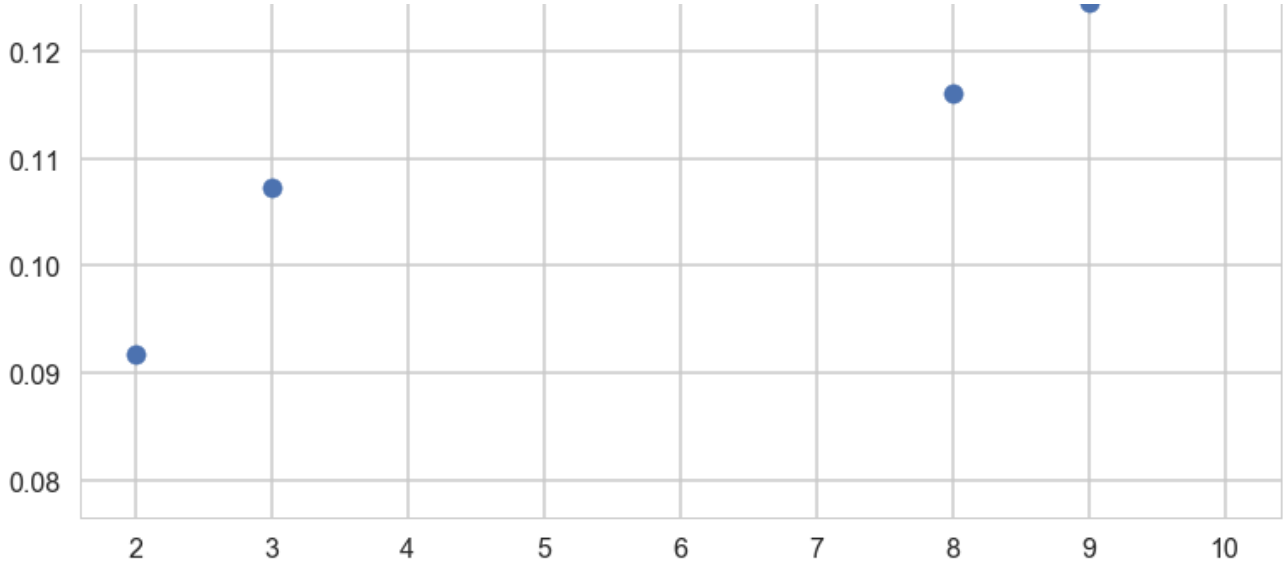
In [18]:

```
plt.scatter(data=sil_scores, x='K', y='sil_score')
```

Out[18]:

```
<matplotlib.collections.PathCollection at 0xe1e80f0>
```



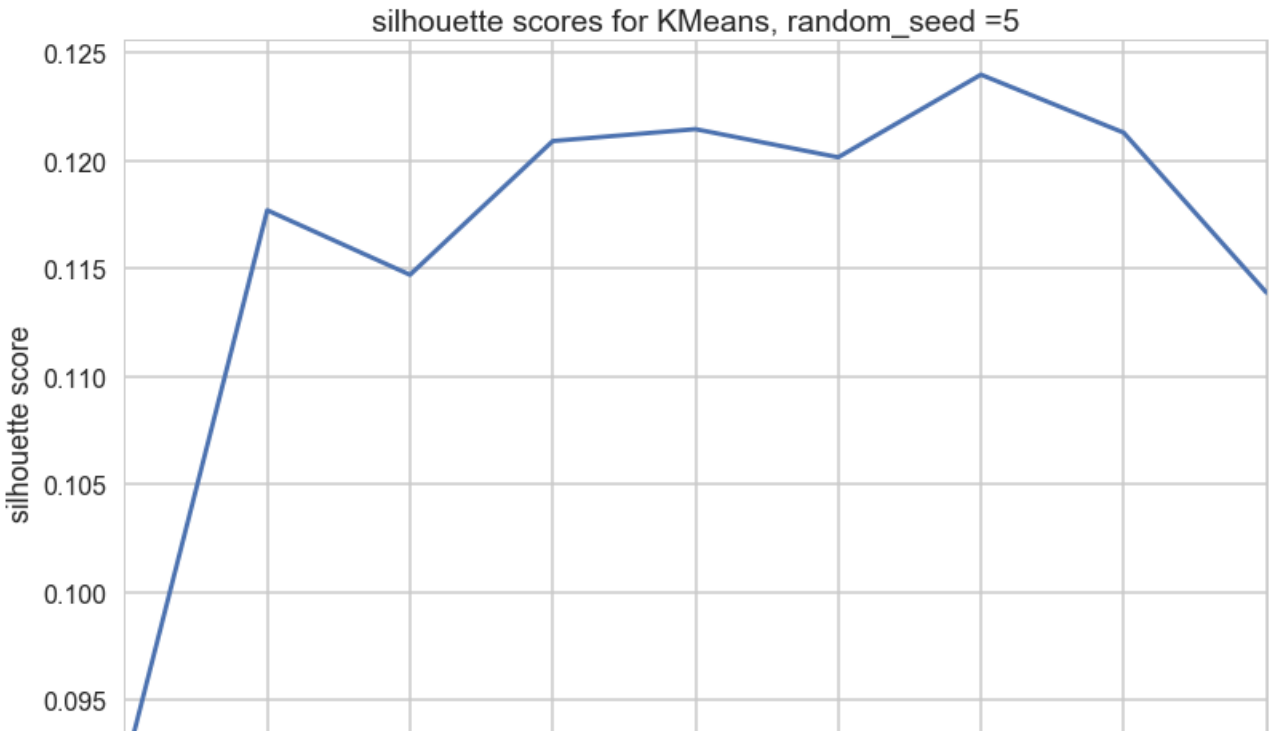


In [19]:

```
# sil_scores at random_state = 5
from sklearn.cluster import KMeans
scores = []
from sklearn.metrics import silhouette_score
for K in range(2,11):
    kmeans = KMeans(n_clusters=K,random_state=5)
    pred = kmeans.fit_predict(x_cols)
    scores.append(silhouette_score(x_cols, pred))
scores = pd.Series(data=scores, index=range(2,11))

scores

scores.plot()
plt.title('silhouette scores for KMeans, random_seed =5')
plt.xlabel('number of clusters')
plt.ylabel('silhouette score')
```

Out[19]:

```
<matplotlib.text.Text at 0xee1ccf8>
```



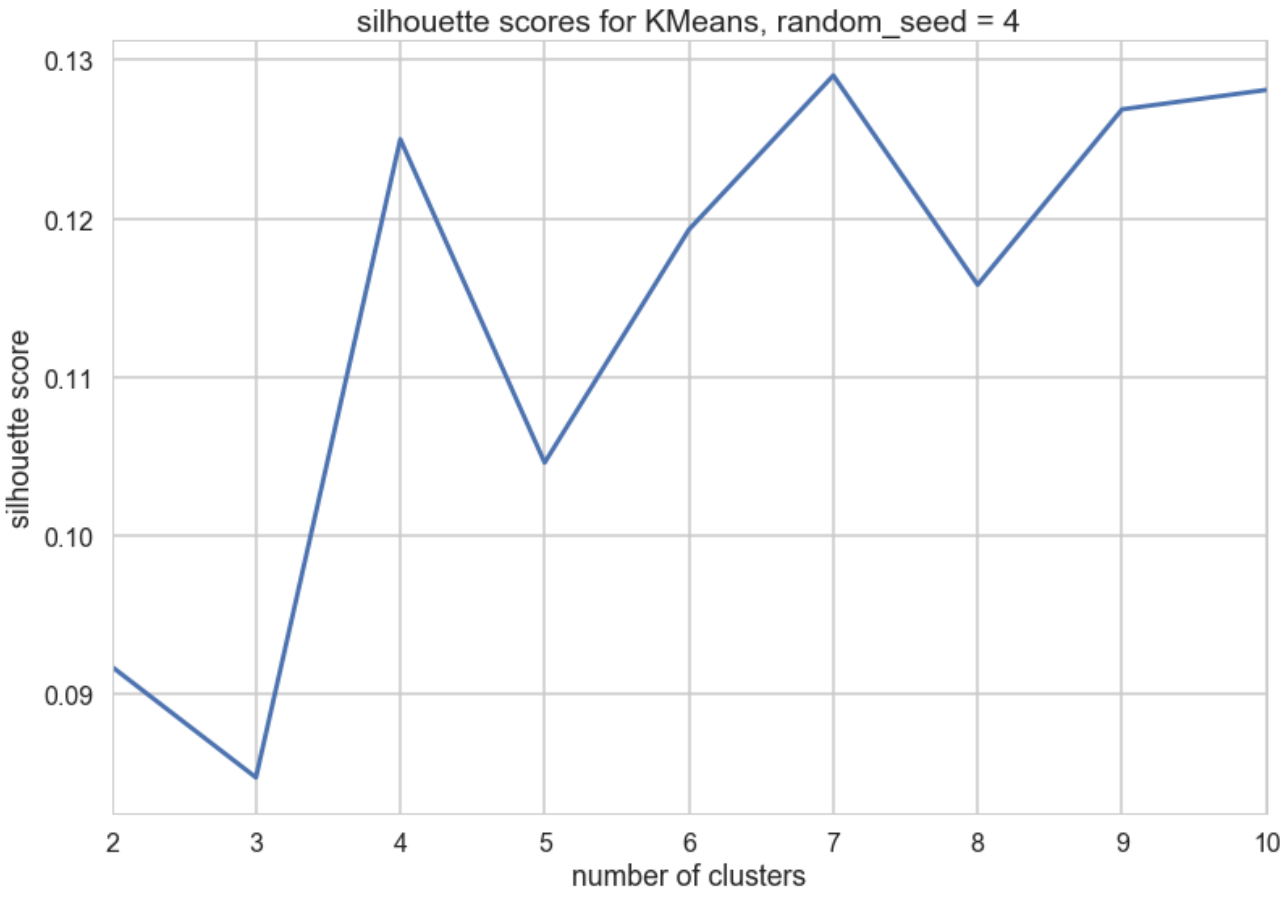
The plot doesnot give definitive insights for the best K like the elbow method. I generated the score with random_state = 5. I will generate individual plots below for random _state = 4 and 6

In [20]:

```
# sil_scores at random_state = 4
from sklearn.cluster import KMeans
scores = []
from sklearn.metrics import silhouette_score
for K in range(2,11):
    kmeans = KMeans(n_clusters=K,random_state=4)
    pred = kmeans.fit_predict(x_cols)
    scores.append(silhouette_score(x_cols, pred))
scores = pd.Series(data=scores, index=range(2,11))


scores.plot()
plt.title('silhouette scores for KMeans, random_seed = 4')
plt.xlabel('number of clusters')
plt.ylabel('silhouette score')
```

Out[20]:

```
<matplotlib.text.Text at 0xee4e2e8>
```

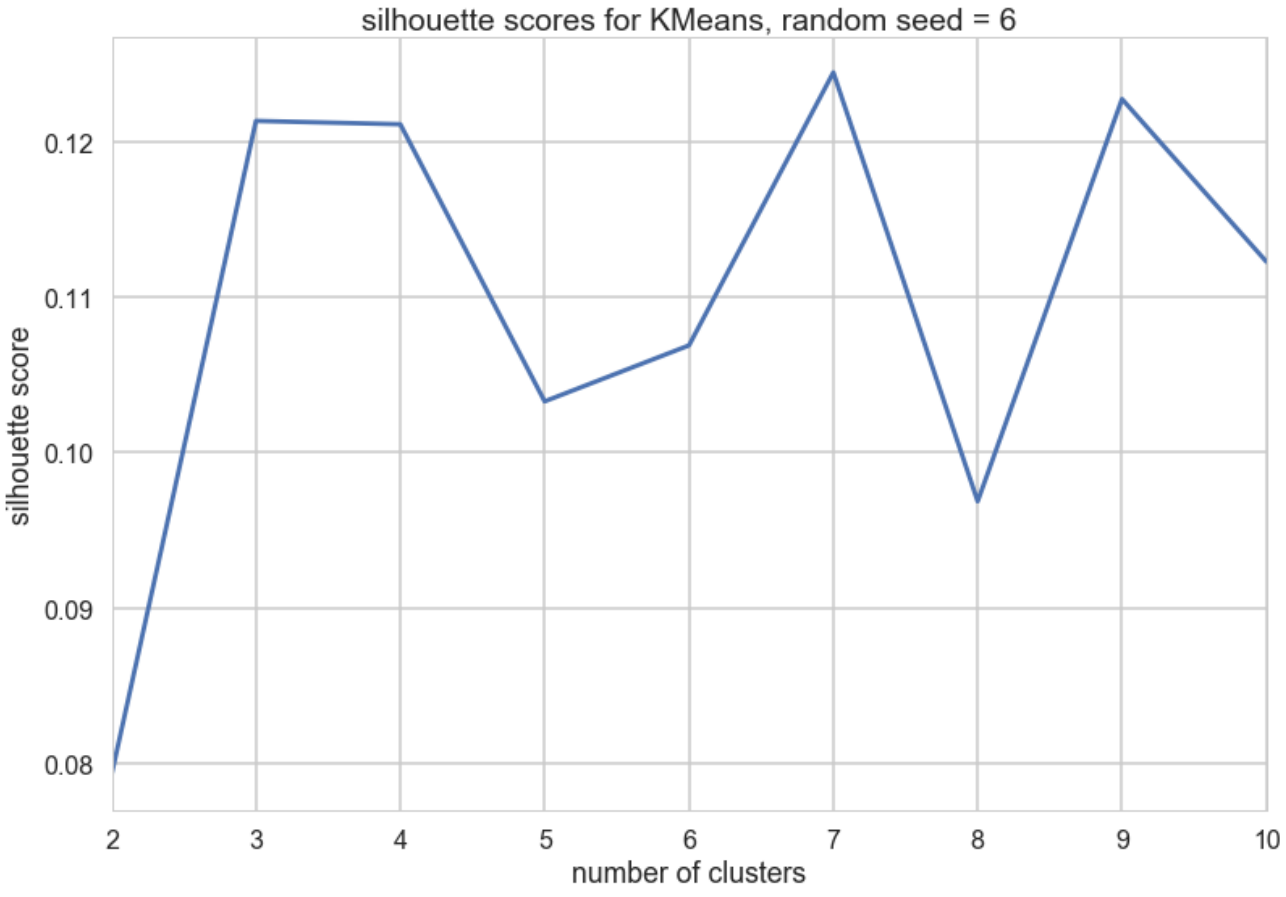


In [21]:

```
# sil_scores at random_state = 6
scores = []
from sklearn.metrics import silhouette_score
for K in range(2,11):
    kmeans = KMeans(n_clusters=K,random_state=6)
    pred = kmeans.fit_predict(x_cols)
    scores.append(silhouette_score(x_cols, pred))
scores = pd.Series(data=scores, index=range(2,11))


scores.plot()
plt.title('silhouette scores for KMeans, random seed = 6')
plt.xlabel('number of clusters')
plt.ylabel('silhouette score')

scores
```

Out[21]:

```
2     0.079267
3     0.121331
4     0.121113
5     0.103279
6     0.106881
7     0.124449
8     0.096849
9     0.122737
10    0.112272
dtype: float64
```



None of the above silhouette plots generated at different random_seed provide any striking differences in the scores to pick the best K.

Choosing K: The Gap Statistic

There is one last method worth covering for picking K, the so-called Gap statistic. The computation for the gap statistic builds on the sum-of-squares established in the Elbow method discussion, and compares it to the sum-of-squares of a "null distribution," that is, a random set of points with no clustering. The estimate for the optimal number of clusters K is the value for which $\log{SS}$ falls the farthest below that of the reference distribution:
$$G_k = E_n^*\{\log SS_k\} - \log SS_k$$

In other words a good clustering yields a much larger difference between the reference distribution and the clustered data. The reference distribution is a Monte Carlo (randomization) procedure that constructs B random distributions of points within the bounding box (limits) of the original data and then applies K-means to this synthetic distribution of data points.. $E_n^*\{\log SS_k\}$ is just the average SS_k over all B replicates. We then compute the standard deviation σ_{SS} of the values of SS_k computed from the B replicates of the reference distribution and compute
$$s_k = \sqrt{1+1/B}\sigma_{SS}$$

Finally, we choose $K=k$ such that $G_k \geq G_{k+1} - s_{k+1}$.

Aside: Choosing K when we Have Labels

Unsupervised learning expects that we do not have the labels. In some situations, we may wish to cluster data that is labeled. Computing the optimal number of clusters is much easier if we have access to labels. There are several methods available. We will not go into the math or details since it is rare to have access to the labels, but we provide the names and references of these measures.

- Adjusted Rand Index
- Mutual Information
- V-Measure
- Fowlkes–Mallows index

See this article for more information about these metrics.

Visualizing Clusters using PCA

How do we visualize clusters? If we only had two features, we could likely plot the data as is. But we have 100 data points each containing 32 features (dimensions). Principal Component Analysis (PCA) will help us reduce the dimensionality of our data from 32 to something lower. For a visualization on the coordinate plane, we will use 2 dimensions. In this exercise, we're going to use it to transform our multi-dimensional dataset into a 2 dimensional dataset.

This is only one use of PCA for dimension reduction. We can also use PCA when we want to perform regression but we have a set of highly correlated variables. PCA untangles these correlations into a smaller number of features/predictors all of which are orthogonal (not correlated). PCA is also used to reduce a large set of variables into a much smaller one.

Checkup Exercise Set IV

Exercise: Use PCA to plot your clusters:

- Use scikit-learn's [`PCA`](http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html) function to reduce the dimensionality of your clustering data to 2 components
- Create a data frame with the following fields:
  - customer name
  - cluster id the customer belongs to
  - the two PCA components (label them `x` and `y`)
- Plot a scatterplot of the `x` vs `y` columns
- Color-code points differently based on cluster ID
- How do the clusters look?
- Based on what you see, what seems to be the best value for K? Moreover, which method of choosing K seems to have produced the optimal result visually?

Exercise: Now look at both the original raw data about the offers and transactions and look at the fitted clusters. Tell a story about the clusters in context of the original data. For example, do the clusters correspond to wine variants or something else interesting?

In [22]:

```
kmeans = sklearn.cluster.KMeans(n_clusters = 3)
kmeans.fit(x_cols)
```

Out[22]:

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

In [23]:

```
PCA_2 = sklearn.decomposition.PCA(n_components=2)
pred = PCA_2.fit(x_cols)
```

In [24]:

```
cust_offer_pca = pd.DataFrame(PCA_2.transform(x_cols))
cust_offer_pca.index = cust_offer.index
cust_offer_pca.head(10)

#Create labels
cust_offer_labels = pd.DataFrame(kmeans.labels_)
cust_offer_labels.index = cust_offer.index
cust_offer_labels.head(10)

# Create merged data frame with x, y and labels
df_pca_with_labels = cust_offer_pca.merge(cust_offer_labels, left_index=True, right_index=True)
df_pca_with_labels.columns = ['x', 'y', 'label']
df_pca_with_labels.head(10)
```

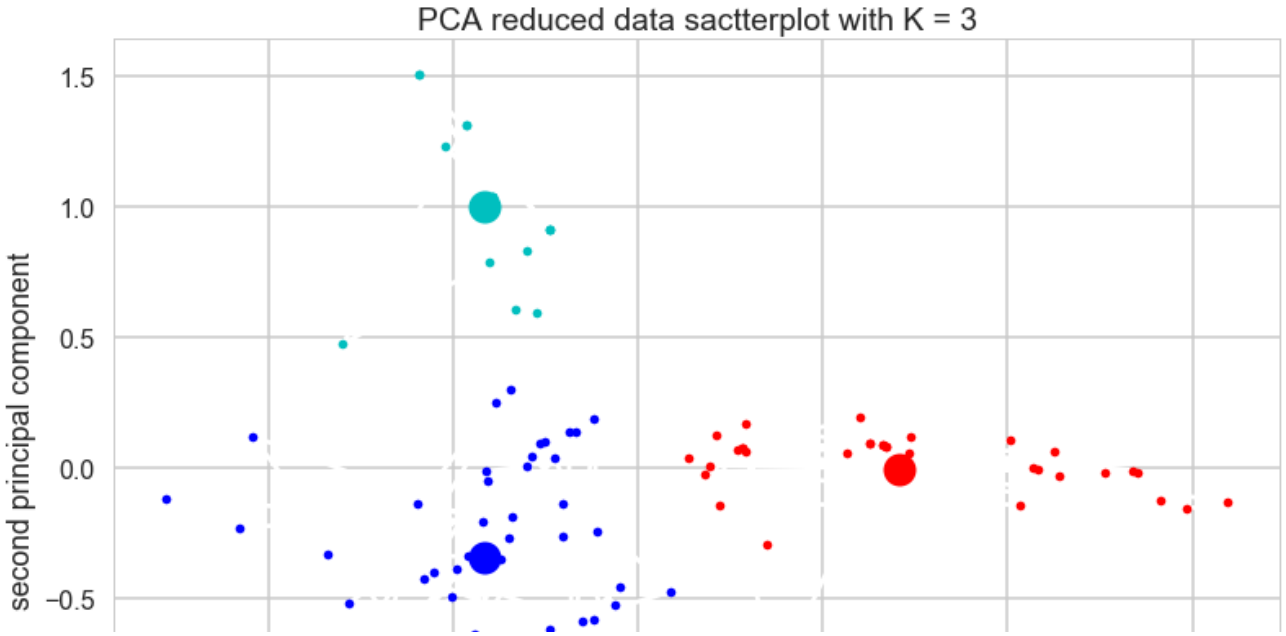
Out[24]:

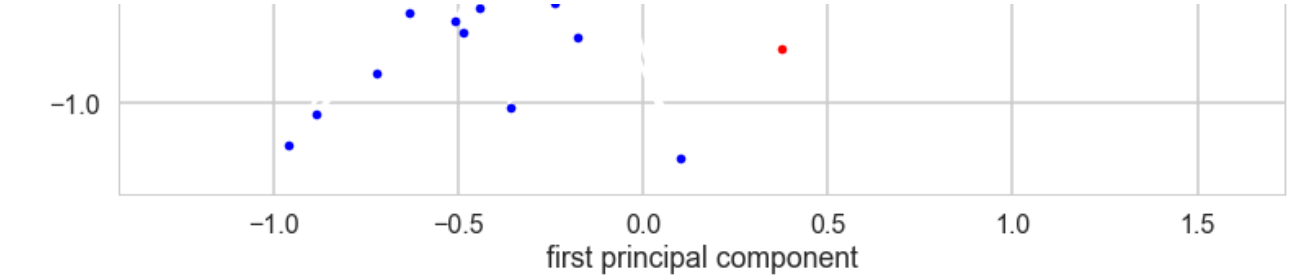
	x	y	label
customer_name			
Adams	1.007580	0.108215	1

customer_name	x	y	label
Allen	-0.287539	0.044715	0
Anderson	-0.392032	1.038391	2
Bailey	0.699477	-0.022542	1
Baker	0.088183	-0.471695	0
Barnes	-0.485282	-0.725503	0
Bell	-0.591941	1.506500	2
Bennett	0.661732	0.090308	1
Brooks	-0.118943	-0.577499	0
Brown	1.079827	-0.007488	1

In [25]:

```
#your turn
# K= 3
from sklearn.decomposition import PCA
mypca = PCA(n_components=2)
reduced = mypca.fit_transform(x_cols)

kmeans = KMeans(n_clusters=3)
pred = kmeans.fit_predict(reduced)
colors = ['b' , 'c' , 'r']

for k, col in zip(range(5) , colors):
    members = (pred== k)
    plt.plot(reduced[members,0], reduced[members,1], 'w' , markerfacecolor=
col, marker='.')
    plt.plot(kmeans.cluster_centers_[k,0], kmeans.cluster_centers_[k,1], 'o'
,
             markerfacecolor = col, markeredgecolor='k',markersize=20)

plt.title('PCA reduced data sactterplot with K = 3')
plt.xlabel('first principal component')
plt.ylabel('second principal component')
plt.show()
```



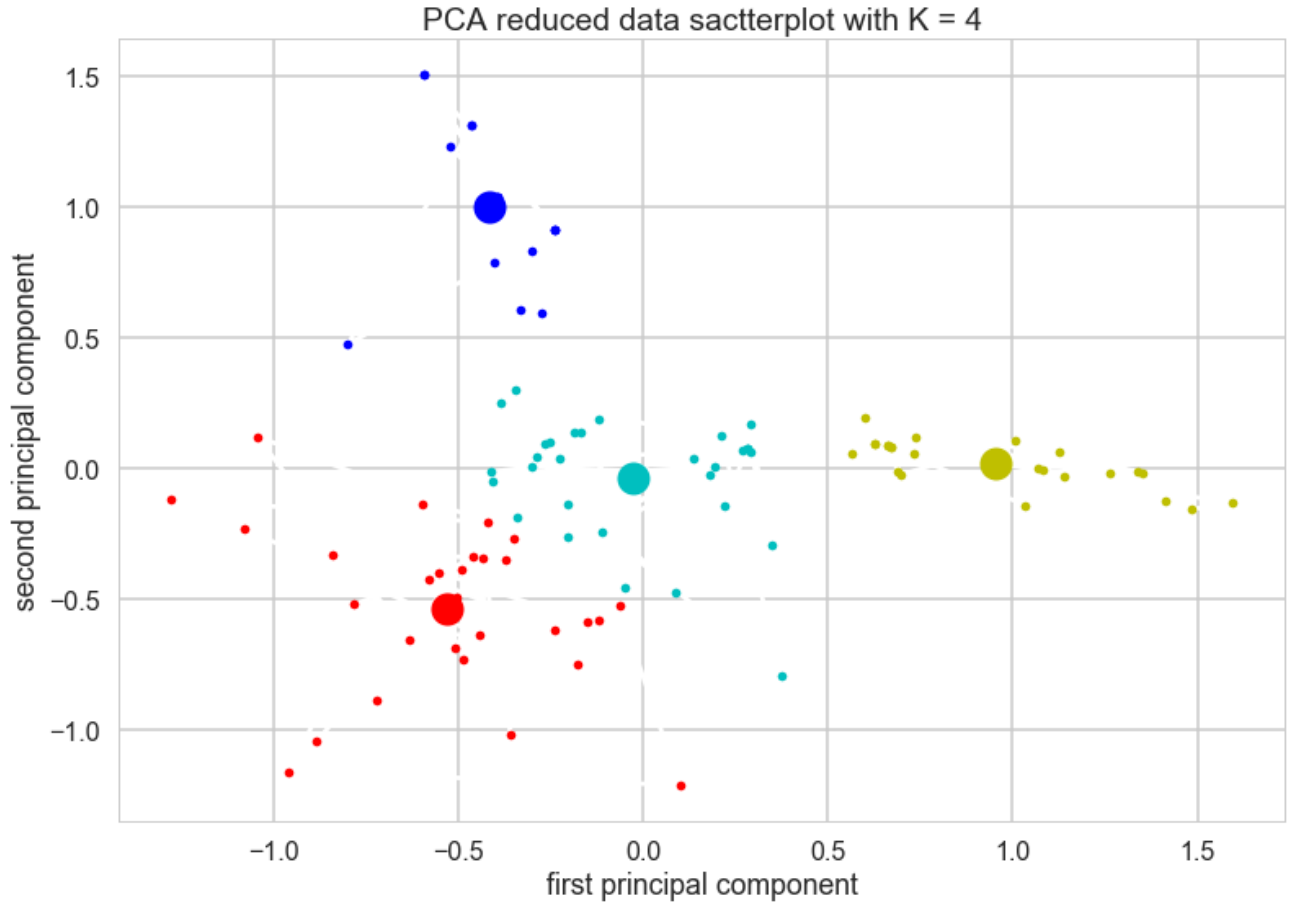


In [26]:

```
# sns.lmplot(data=df_pca_with_labels, x='x', y='y', hue='label', size=10, f
it_reg=False)
```

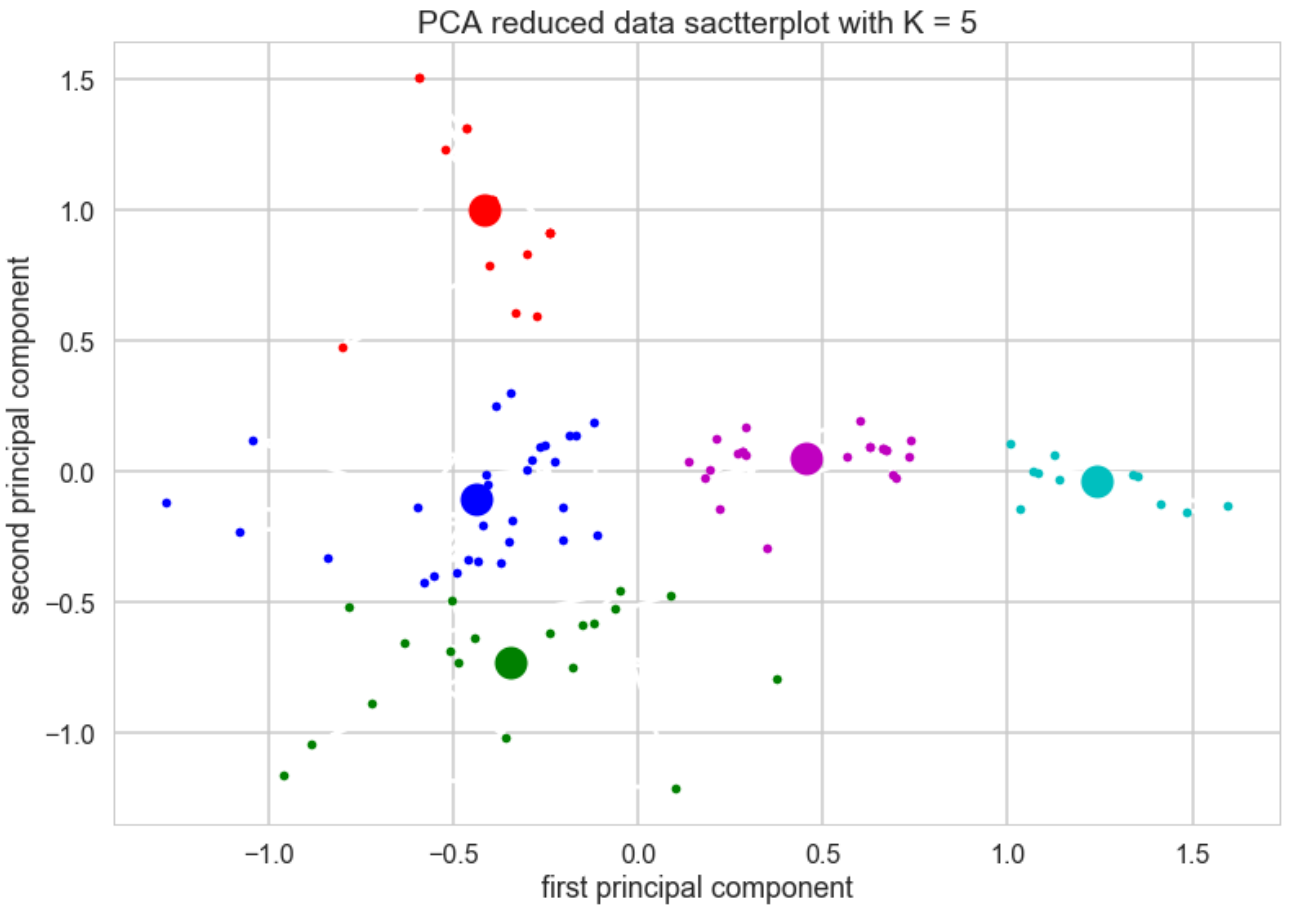
In [27]:

```
kmeans = KMeans(n_clusters=4)
pred = kmeans.fit_predict(cust_offer_pca)
colors = ['b' , 'c' , 'r','y']

for k, col in zip(range(4) , colors):
    members = (pred == k)
    plt.plot(reduced[members,0], reduced[members,1], 'w' , markerfacecolor=
col, marker='.')
    plt.plot(kmeans.cluster_centers_[k,0], kmeans.cluster_centers_[k,1], 'o'
,
             markerfacecolor = col, markeredgecolor='k',markersize=20)

plt.title('PCA reduced data sactterplot with K = 4')
plt.xlabel('first principal component')
plt.ylabel('second principal component')
plt.show()
```



In [28]:

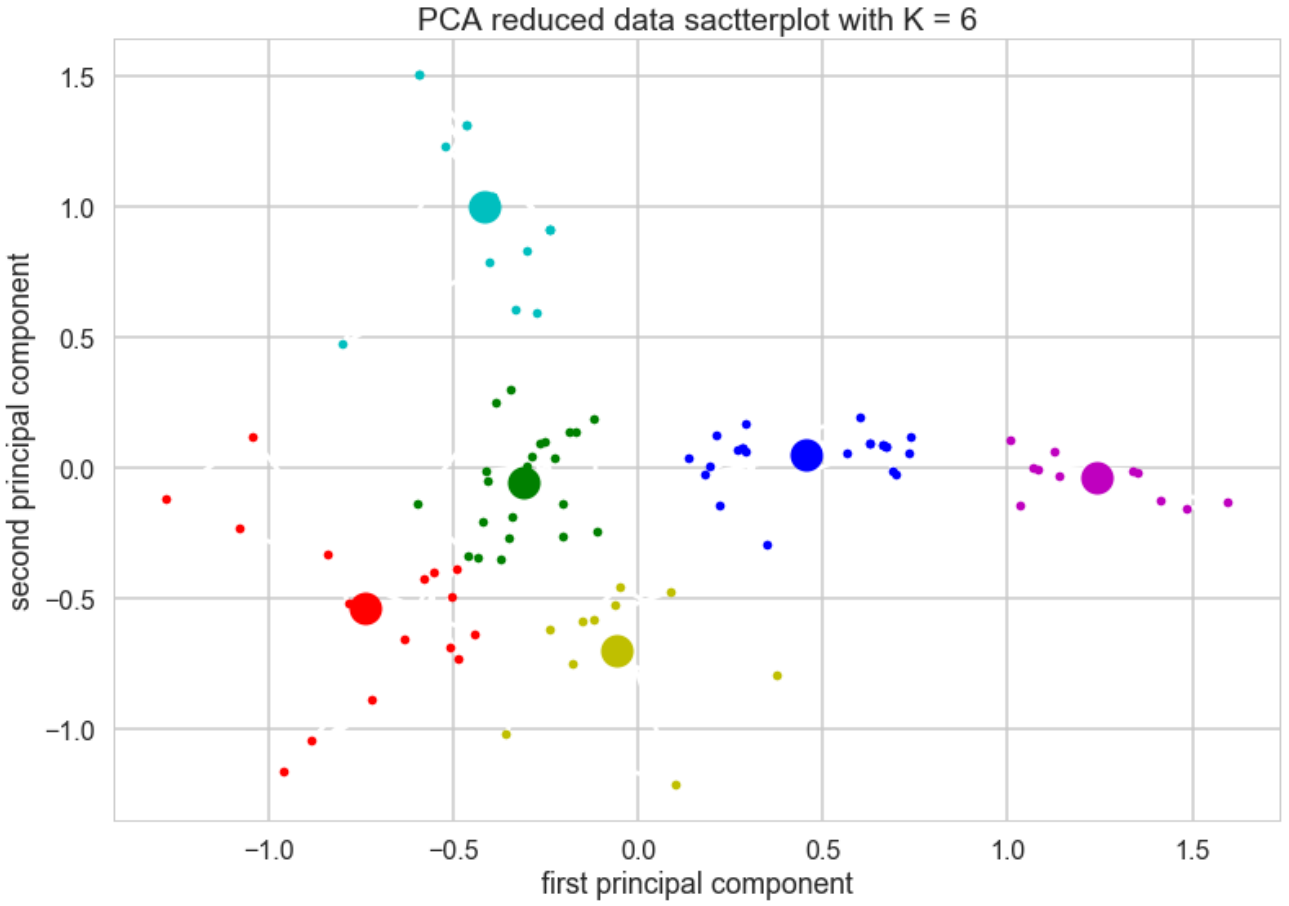
```
#your turn
# K= 5
from sklearn.decomposition import PCA
mypca = PCA(n_components=2)
reduced = mypca.fit_transform(x_cols)

kmeans = KMeans(n_clusters=5)
pred = kmeans.fit_predict(reduced)
colors = ['b' , 'c' , 'r', 'm','g','y']

for k, col in zip(range(5) , colors):
    members = (pred== k)
    plt.plot(reduced[members,0], reduced[members,1], 'w' , markerfacecolor=
col, marker='.')
    plt.plot(kmeans.cluster_centers_[k,0], kmeans.cluster_centers_[k,1], 'o'
,
             markerfacecolor = col, markeredgecolor='k',markersize=20)

plt.title('PCA reduced data sactterplot with K = 5')
plt.xlabel('first principal component')
plt.ylabel('second principal component')
plt.show()
```



In [29]:

```
#your turn
#K = 6
from sklearn.decomposition import PCA
mypca = PCA(n_components=2)
reduced = mypca.fit_transform(x_cols)
```

```
reduced = mypca.fit_transform(x_cols)

kmeans = KMeans(n_clusters=6)
pred = kmeans.fit_predict(reduced)
colors = ['b' , 'c' , 'r', 'm','g','y','p']

for k, col in zip(range(6) , colors):
    members = (pred == k)
    plt.plot(reduced[members,0], reduced[members,1], 'w' , markerfacecolor=
col, marker='.')
    plt.plot(kmeans.cluster_centers_[k,0], kmeans.cluster_centers_[k,1], 'o'
,
             markerfacecolor = col, markeredgecolor='k',markersize=20)

plt.title('PCA reduced data sactterplot with K = 6')
plt.xlabel('first principal component')
plt.ylabel('second principal component')
plt.show()
```



The graph for n_cluster = 6 looks more segmented as compared to other n values.

What we've done is we've taken those columns of 0/1 indicator variables, and we've transformed them into a 2-D dataset. We took one column and arbitrarily called it `x` and then called the other `y`. Now we can throw each point into a scatterplot. We color coded each point based on it's cluster so it's easier to see them.

Exercise Set V

As we saw earlier, PCA has a lot of other uses. Since we wanted to visualize our data in 2

dimensions, restricted the number of dimensions to 2 in PCA. But what is the true optimal number of dimensions?

Exercise: Using a new PCA object shown in the next cell, plot the `explained_variance_` field and look for the elbow point, the point where the curve's rate of descent seems to slow sharply. This value is one possible value for the optimal number of dimensions. What is it?

In [31]:

```
#your turn
# Initialize a new PCA model with a default number of components.
import sklearn.decomposition
pca = sklearn.decomposition.PCA()
pca.fit(x_cols)

# Do the rest on your own :)
pca.explained_variance_
```

Out[31]:

```
array([ 0.40555241,  0.30446016,  0.20026967,  0.1653668 ,  0.14865096,
        0.14200293,  0.13680698,  0.12070372,  0.1151981 ,  0.10696228,
        0.09838435,  0.09401002,  0.08603449,  0.07184171,  0.06543861,
        0.06183019,  0.05578045,  0.05274121,  0.04681513,  0.04349972,
        0.03861419,  0.03589526,  0.03421157,  0.0320274 ,  0.02911226,
        0.02592039,  0.02285085,  0.02121206,  0.01862586,  0.01635995,
        0.01411925,  0.00770111])
```

In [32]:

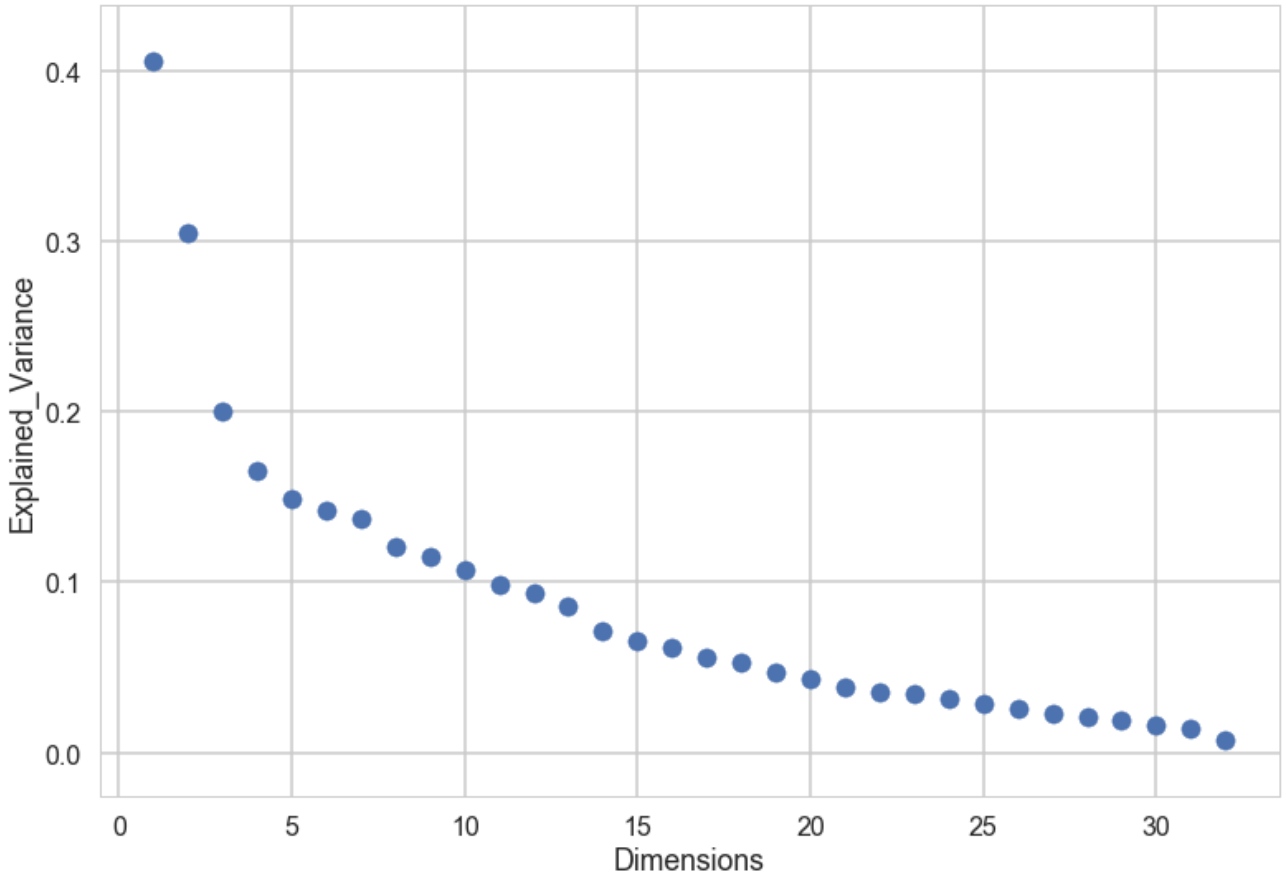
```
length_pca_expvar = len(pca.explained_variance_)
expvar_df = pd.DataFrame({ "Dimension": range(1, length_pca_expvar + 1),
                           "Explained_Variance": pca.explained_variance_

                          })
expvar_df.head()
```

Out[32]:

	Dimension	Explained_Variance
0	1	0.405552
1	2	0.304460
2	3	0.200270
3	4	0.165367
4	5	0.148651

In [33]:

```
import pylab as pl
plt.scatter(data=expvar_df, x='Dimension', y='Explained_Variance')
# pl.plot(x,y,'go-')
plt.xlabel('Dimensions')
plt.ylabel('Explained_Variance')
plt.show()
```



The curve's rate of descent seems to slow sharply from 3 to 4. Hence, we can assume optimal number of dimensions should be 3.

I tried varying the number of dimensions and found that K = 3 worked reasonably well.

Other Clustering Algorithms

k-means is only one of a ton of clustering algorithms. Below is a brief description of several clustering algorithms, and the table provides references to the other clustering algorithms in scikit-learn.

- **Affinity Propagation** does not require the number of clusters K to be known in advance! AP uses a "message passing" paradigm to cluster points based on their similarity.
- **Spectral Clustering** uses the eigenvalues of a similarity matrix to reduce the dimensionality of the data before clustering in a lower dimensional space. This is tangentially similar to what we did to visualize k-means clusters using PCA. The number of clusters must be known a priori.
- **Ward's Method** applies to hierarchical clustering. Hierarchical clustering algorithms take a set of data and successively divide the observations into more and more clusters at each layer of the hierarchy. Ward's method is used to determine when two clusters in the hierarchy should be combined into one. It is basically an extension of hierarchical clustering. Hierarchical clustering is *divisive*, that is, all observations are part of the same cluster at first, and at each successive iteration, the clusters are made smaller and smaller. With hierarchical clustering, a hierarchy is constructed, and there is not really the concept of "number of clusters." The number of clusters simply determines how low or how high in the hierarchy we reference and can be determined empirically or by looking at the dendogram.
- **Agglomerative Clustering** is similar to hierarchical clustering but but is not divisive, it is *agglomerative*. That is, every observation is placed into its own cluster and at each iteration or level or the hierarchy, observations are merged into fewer and fewer clusters until

or level of the hierarchy, observations are merged into fewer and fewer clusters until convergence. Similar to hierarchical clustering, the constructed hierarchy contains all possible numbers of clusters and it is up to the analyst to pick the number by reviewing statistics or the dendogram.

- **DBSCAN** is based on point density rather than distance. It groups together points with many nearby neighbors. DBSCAN is one of the most cited algorithms in the literature. It does not require knowing the number of clusters a priori, but does require specifying the neighborhood size.

Clustering Algorithms in Scikit-learn

</colgroup>

</tr> </thead>

</tr>

</tr>

</tr>

</tr>

</tr>

</tr>

</tr>

</tr>

</tr>

</tr> </tbody> </table> Source: http://scikit-learn.org/stable/modules/clustering.html

Exercise Set VI

Exercise: Try clustering using the following algorithms.

1. Affinity propagation
2. Spectral clustering
3. Agglomerative clustering
4. DBSCAN

How do their results compare? Which performs the best? Tell a story why you think it performs the best.

In [34]:

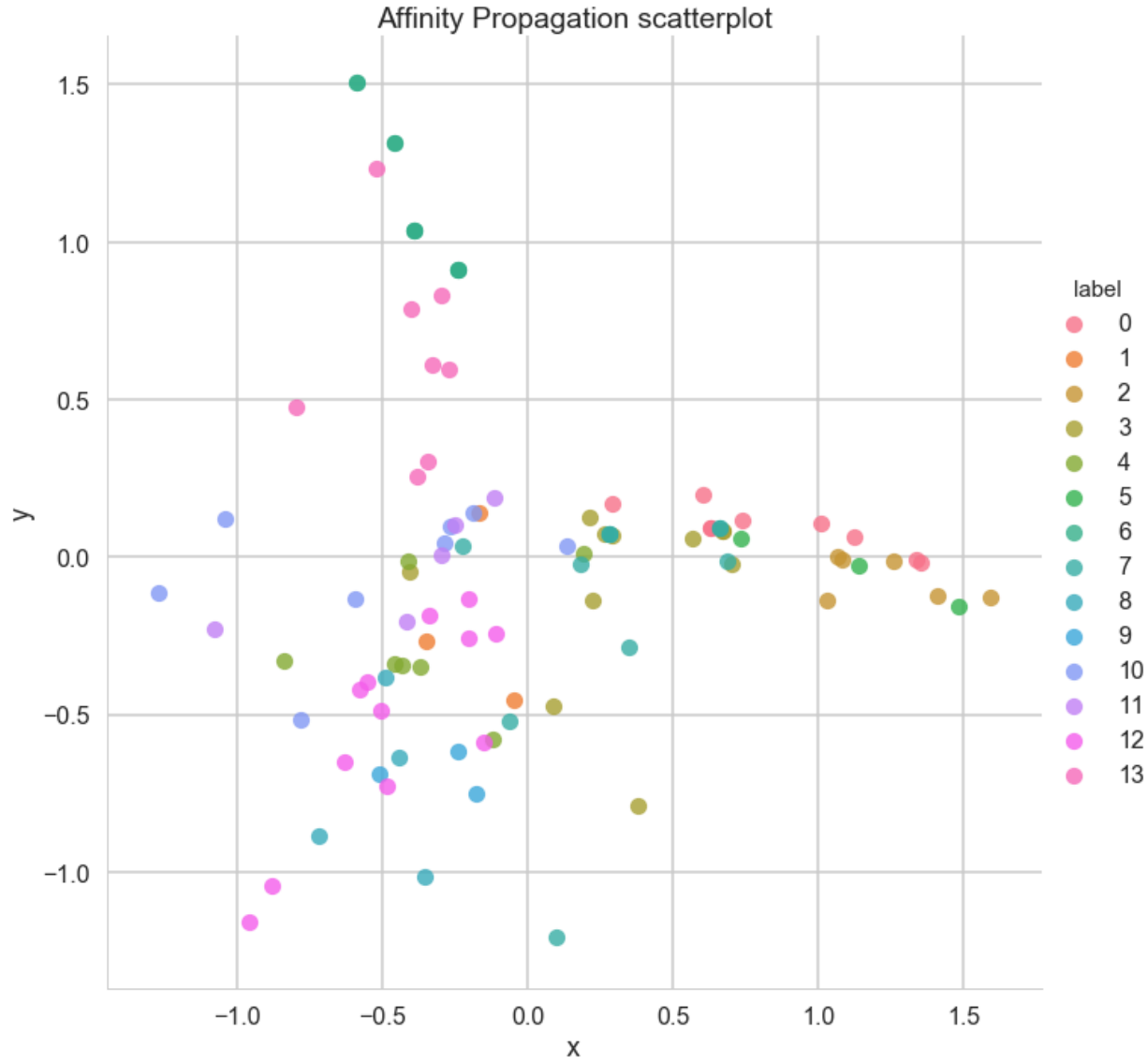
Out[34]:

```
AffinityPropagation(affinity='euclidean', convergence_iter=15,
copy=True,
          damping=0.5, max_iter=200, preference=None, verbose=False)
```

In [35]:

Out[35]:

```
<matplotlib.text.Text at 0xda9c860>
```



In [36]:

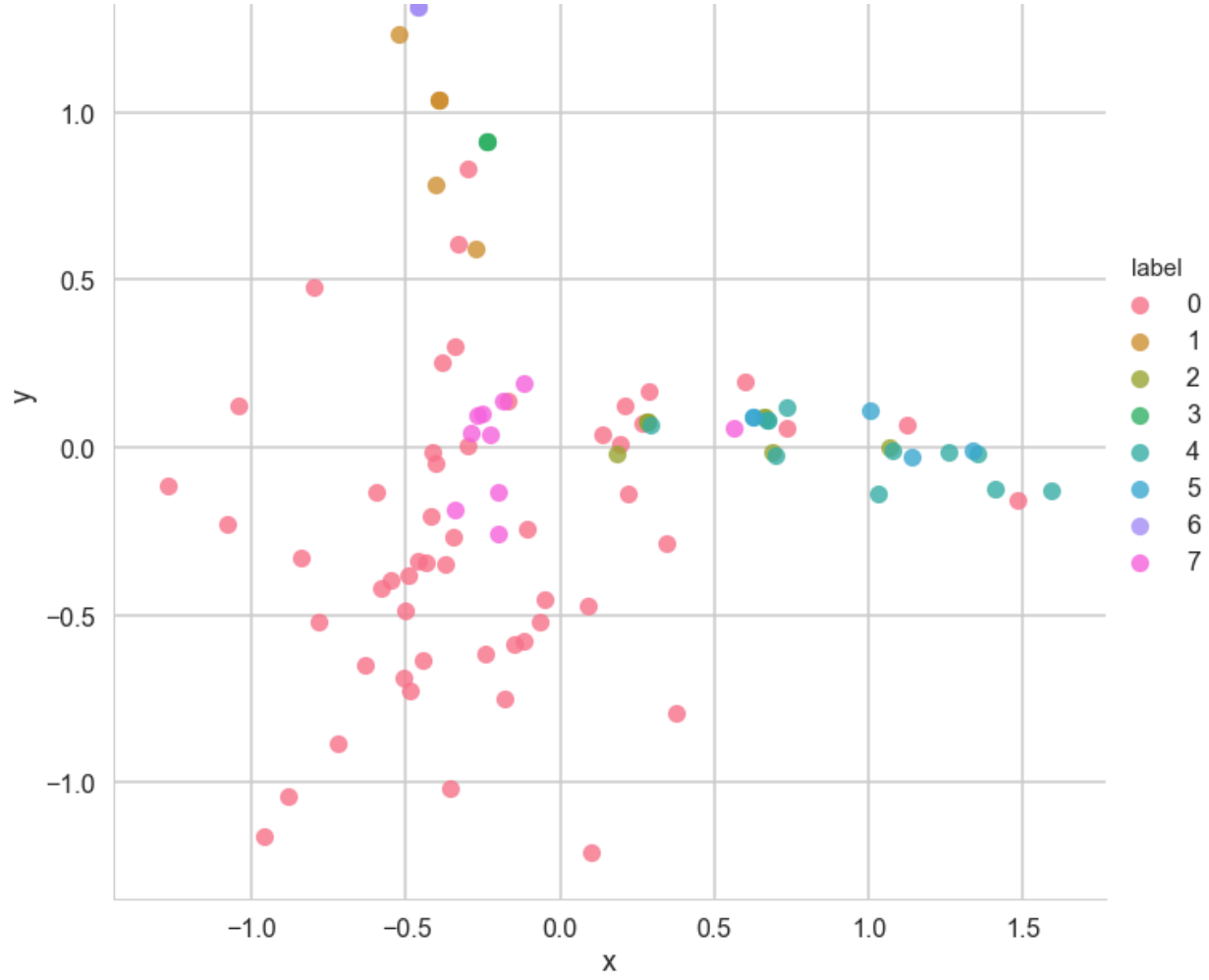
Out[36]:

```
SpectralClustering(affinity='rbf', assign_labels='kmeans', coef0=1, degr
ee=3,
          eigen_solver=None, eigen_tol=0.0, gamma=1.0, kernel_params=None
,
          n_clusters=8, n_init=10, n_jobs=1, n_neighbors=10,
          random_state=None)
```

In [37]:

Out[37]:

```
<matplotlib.text.Text at 0xf6f1668>
```



In [38]:

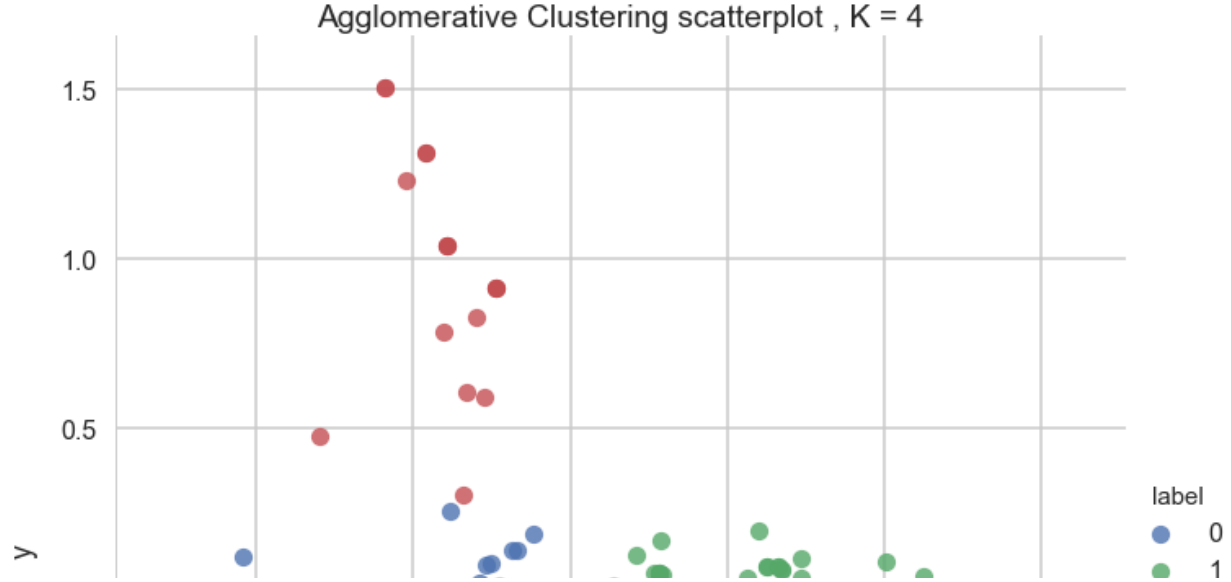
Out[38]:

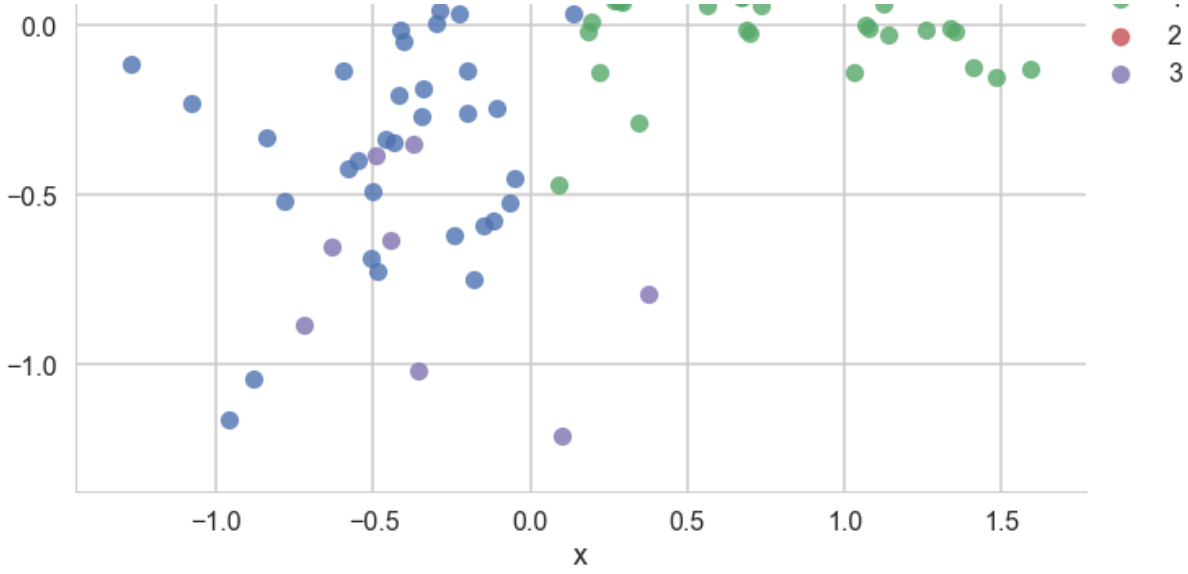
```
AgglomerativeClustering(affinity='euclidean', compute_full_tree='auto',
            connectivity=None, linkage='ward',
            memory=Memory(cachedir=None), n_clusters=4,
            pooling_func=<function mean at 0x00000000055AB950>)
```

In [39]:

Out[39]:

```
<matplotlib.text.Text at 0xf0d5e10>
```





In [40]:

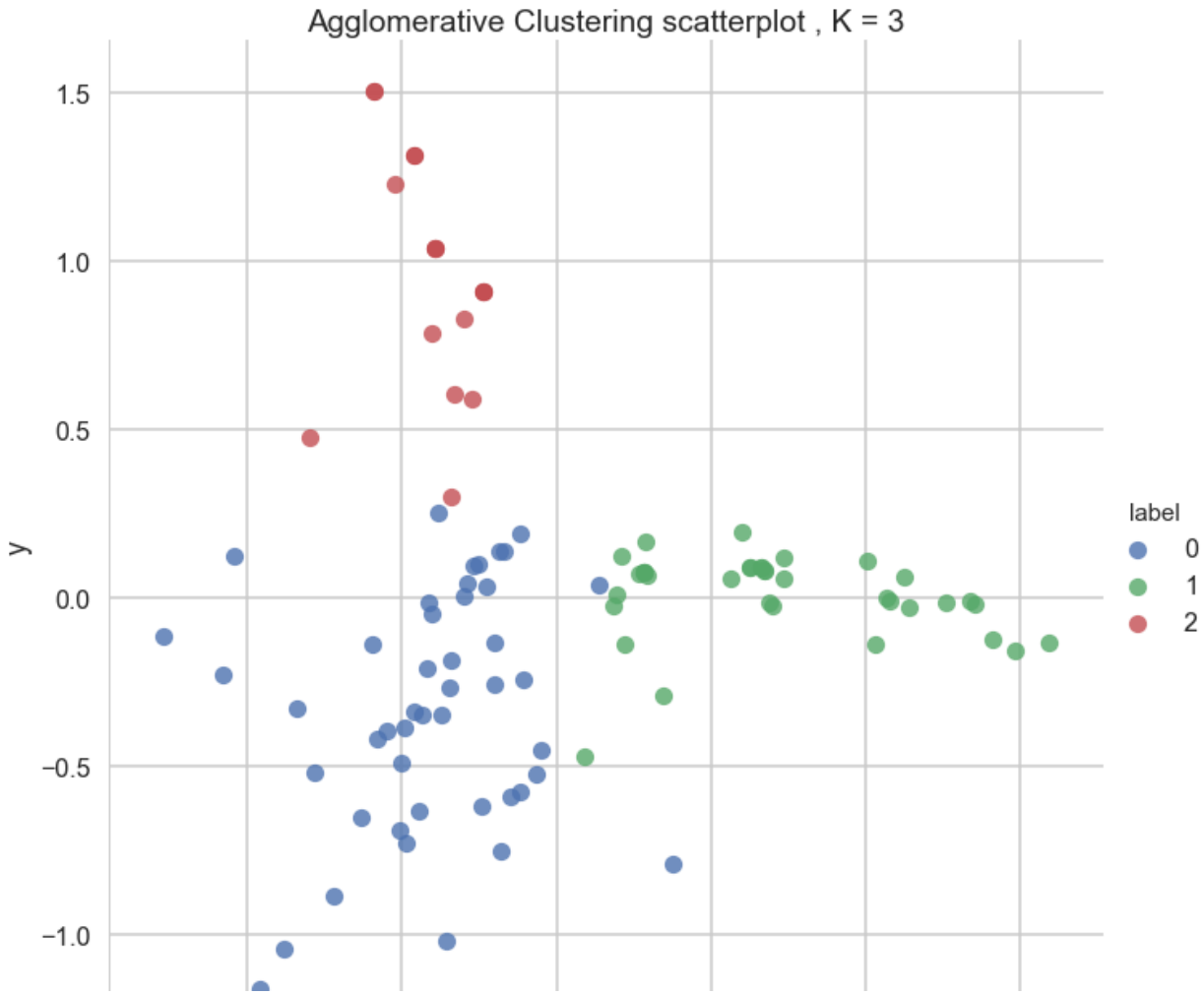
Out[40]:

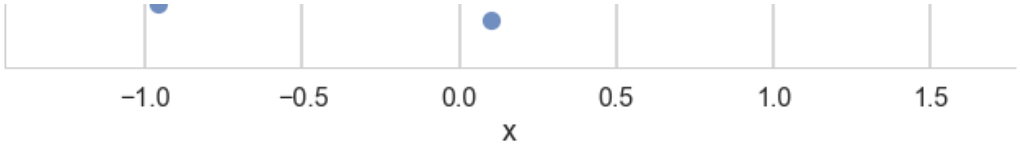
```
AgglomerativeClustering(affinity='euclidean', compute_full_tree='auto',
            connectivity=None, linkage='ward',
            memory=Memory(cachedir=None), n_clusters=3,
            pooling_func=<function mean at 0x00000000055AB950>)
```

In [41]:

Out[41]:

```
<matplotlib.text.Text at 0xda469b0>
```





In [42]:

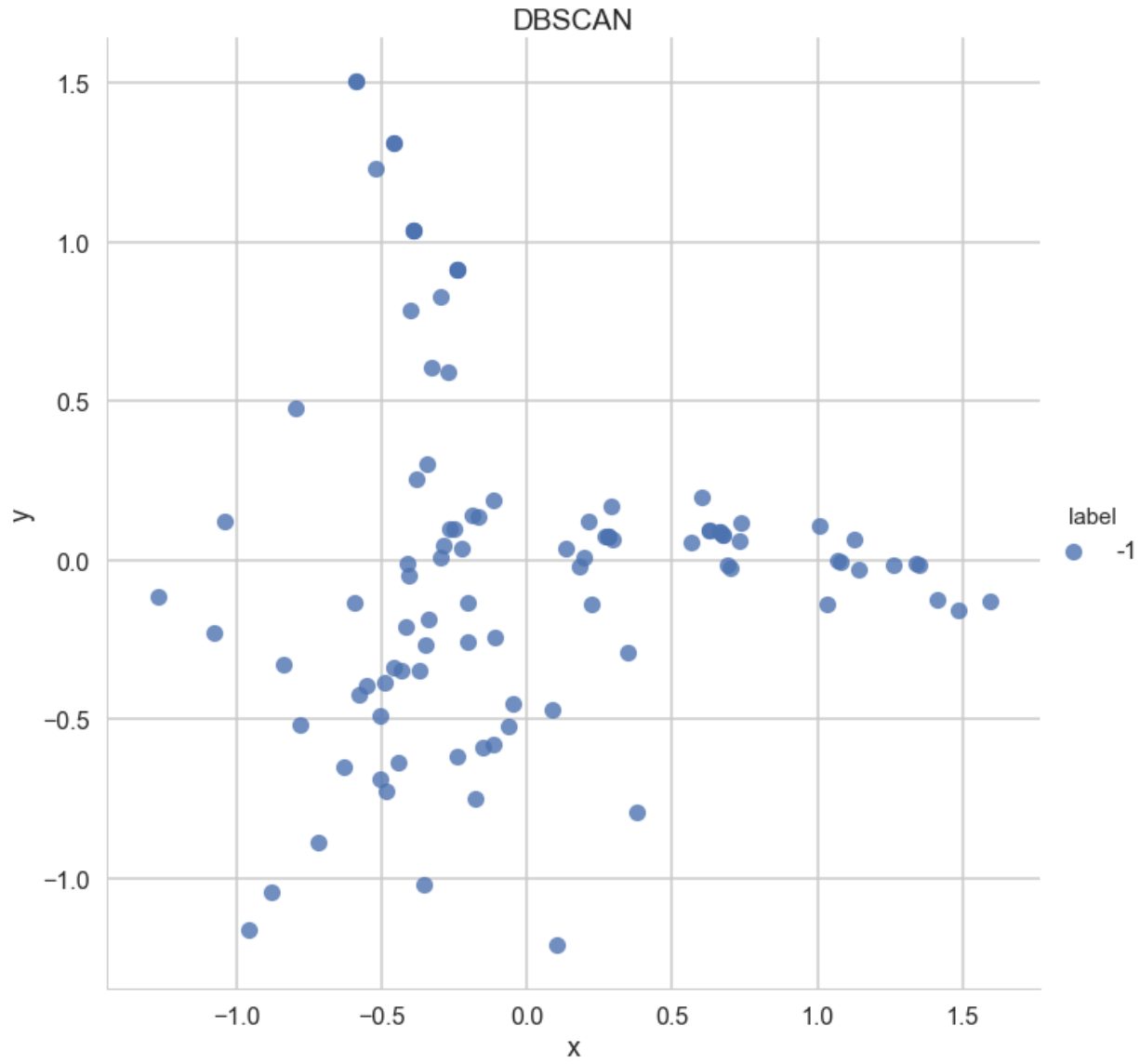
Out[42]:

```
DBSCAN(algorithm='auto', eps=0.5, leaf_size=30, metric='euclidean',
    min_samples=5, n_jobs=1, p=None)
```

In [43]:

Out[43]:

```
<matplotlib.text.Text at 0xf7885f8>
```



Based on the results of different algorithms we ran above, K = 3 comes out to be the optimal number of clusters for this dataset. Agglomomerative clustering gives the best visualization at k = 3. Spectral clustering did OK once I restricted the number of clusters to 3. Spectral clustering and Affinity propagation could not set up distinct clusters with the datapoints overlapping at K = 3 and K = 4. DBSCAN also did very badly, grouping all datapoints to the same cluster.

Overall KMeans and agglomerative clustering appear to be the best algorithms for this dataset based on above analysis

based on above analysis.

Method name	Parameters	Scalability	Use Case	Geometry (metric used)
K-Means	number of clusters	Very largen_samples, medium n_clusters with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium n_samples, small n_clusters	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters, linkage type, distance	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large n_samples, medium n_clusters	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large n_clusters and n_samples	Large dataset, outlier removal, data reduction.	Euclidean distance between points