

Elements Of Data Science - F2021

Week 11: Clustering and Recommendation Systems

11/29/2021

TODOs

- Readings:
 - PDSH: Chap 3.11 Working with Time Series
 - PDSH: Chap 5.06 Example: Predicting Bicycle Traffic
 - Recommended: DSFS: Chap 9: Getting Data
 - Optional: Python for Data Analysis: Chap 11: Time Series
 - Optional: PML: Chap 9: Embedding a Machine Learning Model into a Web Application
- HW4: due Saturday Dec 11th 11:59pm
- Quiz 11: due Sunday Dec 5th, 11:59pm ET

Today

- Clustering
- Recommendation Systems
- Start Time-Series Data?

Questions?

Environment Setup

Environment Setup

```
In [1]: import numpy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from mlxtend.plotting import plot_decision_regions

import warnings
warnings.filterwarnings('ignore')

sns.set_style('darkgrid')
%matplotlib inline
```

Clustering

- Can we group our data based on the features alone?
- **Unsupervised:** There is no label/target y
- Use similarity to group X into k clusters
- Many methods:
 - k-Means
 - **Heirarchical Agglomerative Clustering**
 - Spectral Clustering
 - DBScan
 - ...

Why do Clustering?

- Exploratory data analysis
- Group media: images, music, news articles,...
- Group people: social network
- Science applications: gene families, psychological groups,...
- Image segmentation: group pixels, regions, ...
- ...

Clustering: K-Means

- Not to be confused with k-NN!
- Idea:
 - Finds k points in space as cluster centers (means)
 - Assigns datapoints to their closest cluster mean
- Need to specify the number of clusters k up front
- sklearn uses euclidean distance to judge similarity

k -Means: How it works

FIRST: choose initial k points (means)

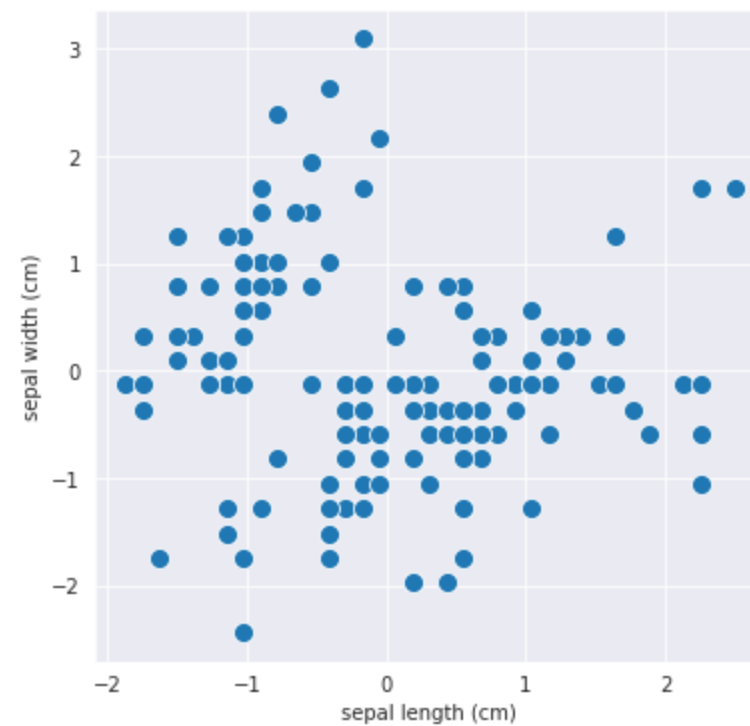
A: fix means \rightarrow assign all datapoints to their closest mean

B: fix cluster assignments \rightarrow recalculate means

RETURN TO A and Repeat until convergence!

Load Example Data

```
In [2]: from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
iris = load_iris()
X_iris = StandardScaler().fit_transform(iris.data[:, :2])
X_iris = pd.DataFrame(X_iris, columns=iris.feature_names[:2])
fig, ax = plt.subplots(1, 1, figsize=(6, 6))
sns.scatterplot(x='sepal length (cm)', y='sepal width (cm)', data=X_iris, s=100);
```



KMeans in sklearn

KMeans in sklearn

```
In [3]: from sklearn.cluster import KMeans  
  
km = KMeans(n_clusters=2, init='random', random_state=0) # default init=k-means++  
  
c = km.fit_predict(X_iris)
```

KMeans in sklearn

```
In [3]: from sklearn.cluster import KMeans

km = KMeans(n_clusters=2, init='random', random_state=0) # default init=k-means++

c = km.fit_predict(X_iris)
```

```
In [4]: # cluster assignments
tmp = X_iris.copy()
tmp['cluster_assignments'] = c
tmp.sample(5, random_state=0)
```

Out[4]:

	sepal length (cm)	sepal width (cm)	cluster_assignments
114	-0.052506	-0.592373	1
62	0.189830	-1.973554	1
33	-0.416010	2.630382	0
107	1.765012	-0.362176	1
7	-1.021849	0.788808	0

KMeans in sklearn

```
In [3]: from sklearn.cluster import KMeans

km = KMeans(n_clusters=2, init='random', random_state=0) # default init=k-means++

c = km.fit_predict(X_iris)
```

```
In [4]: # cluster assignments
tmp = X_iris.copy()
tmp['cluster_assignments'] = c
tmp.sample(5, random_state=0)
```

Out[4]:

	sepal length (cm)	sepal width (cm)	cluster_assignments
114	-0.052506	-0.592373	1
62	0.189830	-1.973554	1
33	-0.416010	2.630382	0
107	1.765012	-0.362176	1
7	-1.021849	0.788808	0

```
In [5]: # cluster centers
km.cluster_centers_
```

```
Out[5]: array([[ -0.97822861,  0.90390597],
               [ 0.4891143 , -0.45195298]])
```

Plotting clusters and centers

Plotting clusters and centers

```
In [6]: # plot data colored by cluster assignment
def plot_clusters(X, c=None, km=None, title=None, ax=None, marker_size=100):
    if not ax:
        fig, ax = plt.subplots(1, 1, figsize=(6, 6))
    if km:
        c = km.fit_predict(X)
    for i in range(np.max(c)+1):
        X_cluster = X[c == i]
        sns.scatterplot(x=X_cluster.iloc[:, 0], y=X_cluster.iloc[:, 1], s=marker_size, ax=ax);
    if km:
        for m in km.cluster_centers_:
            ax.plot(m[0], m[1], marker='x', c='k', ms=20, mew=5)
    if title:
        ax.set_title(title)

plot_clusters(X_iris, km=km, title="KMeans Assignments")
```



K-Means: How good are the clusters?

- One way: **Within Cluster Sum of Squared Distances**
- How close is every point to its assigned cluster center?

$$SSD = \sum_{k=1}^K \sum_{x_i \in C_k} ||x_i - \mu_k||_2^2$$

$$\text{where } ||x - \mu||_2 = \sqrt{\sum_{j=1}^d (x_j - \mu_j)^2}$$

- If this is high, items in cluster are far from their means.
- If this is low, items in cluster are close to their means.

K-Means: How good are the clusters?

- One way: **Within Cluster Sum of Squared Distances**
- How close is every point to it's assigned cluster center?

$$SSD = \sum_{k=1}^K \sum_{x_i \in C_k} ||x_i - \mu_k||_2^2$$

$$\text{where } ||x - \mu||_2 = \sqrt{\sum_{j=1}^d (x_j - \mu_j)^2}$$

- If this is high, items in cluster are far from their means.
- If this is low, items in cluster are close to their means.

```
In [8]: # SSD stored in KMeans as `.inertia_`  
        round(km.inertia_,2)
```

```
Out[8]: 166.95
```

KMeans in Action

KMeans in Action

```
In [9]: import ipywidgets as widgets  
kmeans_video = widgets.Video.from_file('images/kmeans.mp4', width=750, autoplay=False, controls=True)  
kmeans_video
```



KMeans in Action

```
In [9]: import ipywidgets as widgets  
kmeans_video = widgets.Video.from_file('images/kmeans.mp4', width=750, autoplay=False, controls=True)  
kmeans_video
```



From <https://dashee87.github.io/data%20science/general/Clustering-with-Scikit-with-GIFs/>

Things you need to define for KMeans

- number of clusters k or `n_clusters`
- initial locations of means
 - random
 - k-means++ (pick starting points far apart from each other)

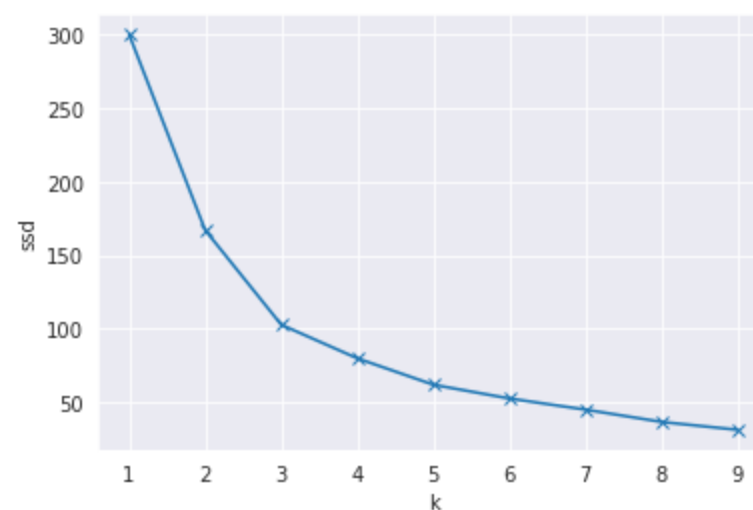
How to choose k or `n_clusters`?

- One way: use "elbow" in sum of squared distances (SSD) or `KMeans.inertia_`
- "elbow" is where SSD ceases to drop rapidly

How to choose k or `n_clusters`?

- One way: use "elbow" in sum of squared distances (SSD) or `KMeans.inertia_`
- "elbow" is where SSD ceases to drop rapidly

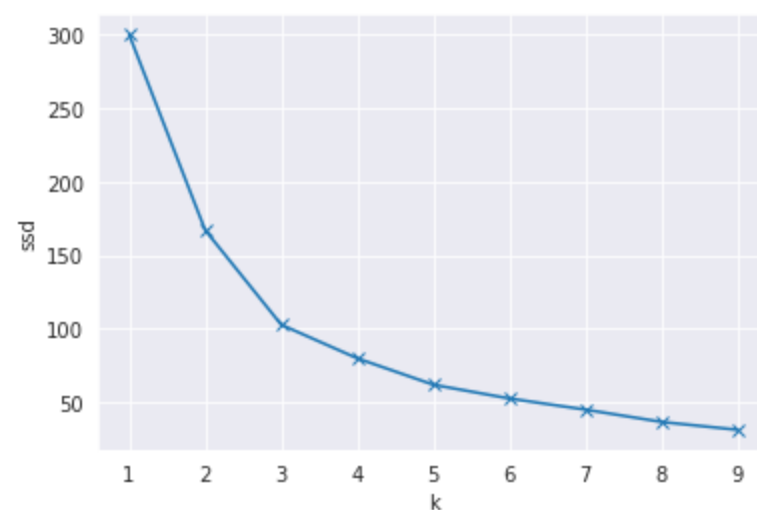
```
In [10]: ssd = []  
         for i in range(1,10):  
             ssd.append(KMeans(n_clusters=i).fit(X_iris).inertia_)  
         fig,ax=plt.subplots(1,1,figsize=(6,4))  
         ax.plot(range(1,10),ssd,marker='x');  
         ax.set_xlabel('k');ax.set_ylabel('ssd');
```



How to choose k or `n_clusters`?

- One way: use "elbow" in sum of squared distances (SSD) or `KMeans.inertia_`
- "elbow" is where SSD ceases to drop rapidly

```
In [10]: ssd = []
for i in range(1,10):
    ssd.append(KMeans(n_clusters=i).fit(X_iris).inertia_)
fig,ax=plt.subplots(1,1,figsize=(6,4))
ax.plot(range(1,10),ssd,marker='x');
ax.set_xlabel('k');ax.set_ylabel('ssd');
```

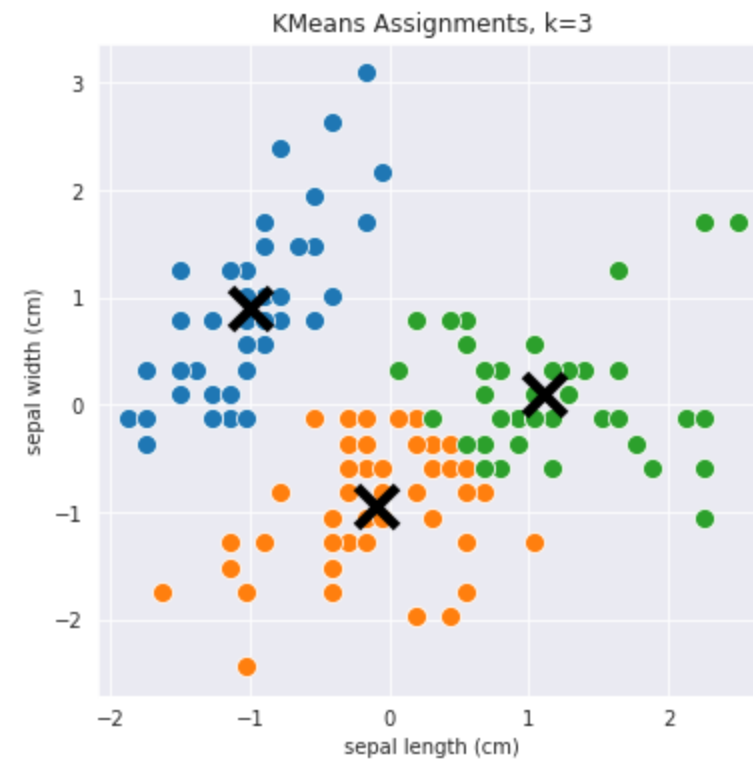


- Question: What value k will minimize SSD?

Refitting with $k=3$

Refitting with k=3

```
In [11]: plot_clusters(X_iris, km=KMeans(n_clusters=3, random_state=0), title="KMeans Assignments, k=3")
```



KMeans: Another Example

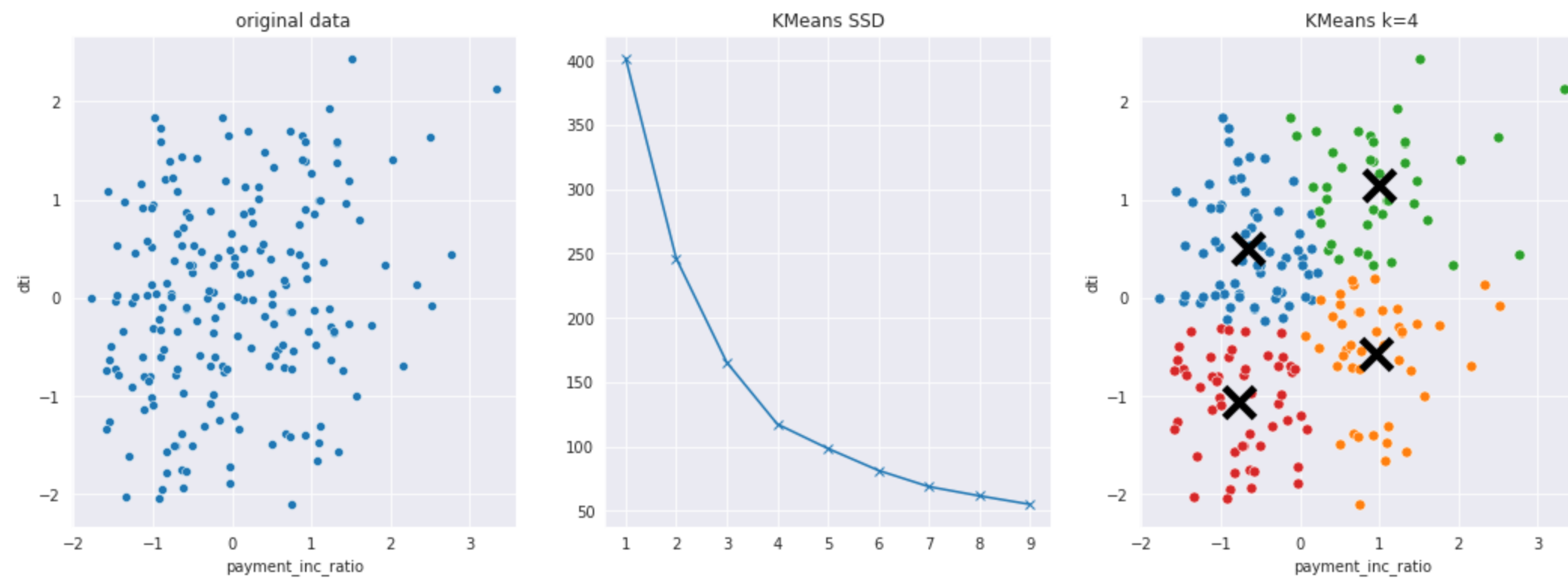
KMeans: Another Example

```
In [13]: # loading and plotting the data
data = pd.read_csv('../data/loan200.csv')[['payment_inc_ratio', 'dti']]
from sklearn.preprocessing import StandardScaler
X_loan = pd.DataFrame(StandardScaler().fit_transform(data), columns=data.columns)

fig, ax = plt.subplots(1, 3, figsize=(18, 6))
sns.scatterplot(x=X_loan.iloc[:, 0], y=X_loan.iloc[:, 1], ax=ax[0]);
ax[0].set_title('original data');

ssd = [KMeans(n_clusters=i).fit(X_loan).inertia_ for i in range(1, 10)]
ax[1].plot(range(1, 10), ssd, marker='x');
ax[1].set_title('KMeans SSD');

plot_clusters(X_loan, km=KMeans(n_clusters=4, random_state=0), title='KMeans k=4', marker_size=50, ax=ax[2])
```



KMeans: Synthetic Example

KMeans: Synthetic Example

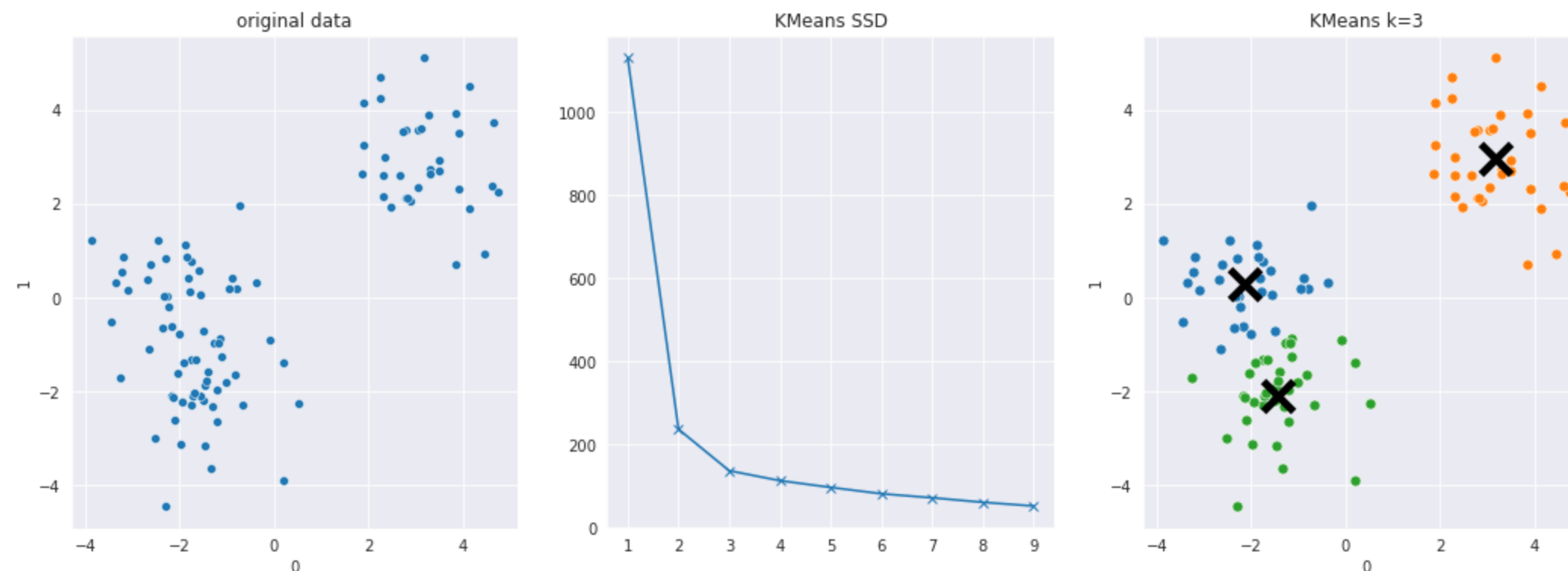
```
In [14]: from sklearn.datasets import make_blobs
X_blobs,y_blobs = make_blobs(centers=[[3,3],(-2,0),(-2,-2)],random_state=1)
X_blobs = pd.DataFrame(X_blobs)

fig,ax = plt.subplots(1,3,figsize=(18,6))

sns.scatterplot(x=X_blobs.iloc[:,0],y=X_blobs.iloc[:,1],ax=ax[0]);
ax[0].set_title('original data');

ssd = [KMeans(n_clusters=i).fit(X_blobs).inertia_ for i in range(1,10)]
ax[1].plot(range(1,10),ssd,marker='x');
ax[1].set_title('KMeans SSD')

plot_clusters(X_blobs,km=KMeans(n_clusters=3, random_state=0),title='KMeans k=3',marker_size=50,ax=ax[2])
```



Hierarchical Agglomerative Clustering (HAC)

- group clusters together from the bottom up
- don't have to specify number of clusters up front
- generates binary tree over data

HAC: How it works

FIRST: every point is it's own cluster

A: Find pair of clusters that are "closest"

B: Merge into single cluster

GOTO A and Repeat till there is a single cluster

HAC in Action

HAC in Action

```
In [15]: import ipywidgets as widgets  
         hac_video = widgets.Video.from_file('images/hac.mp4', width=750, autoplay=False, controls=True)  
         hac_video
```



HAC in Action

```
In [15]: import ipywidgets as widgets  
         hac_video = widgets.Video.from_file('images/hac.mp4', width=750, autoplay=False, controls=True)  
         hac_video
```



From <https://dashee87.github.io/data%20science/general/Clustering-with-Scikit-with-GIFs/>

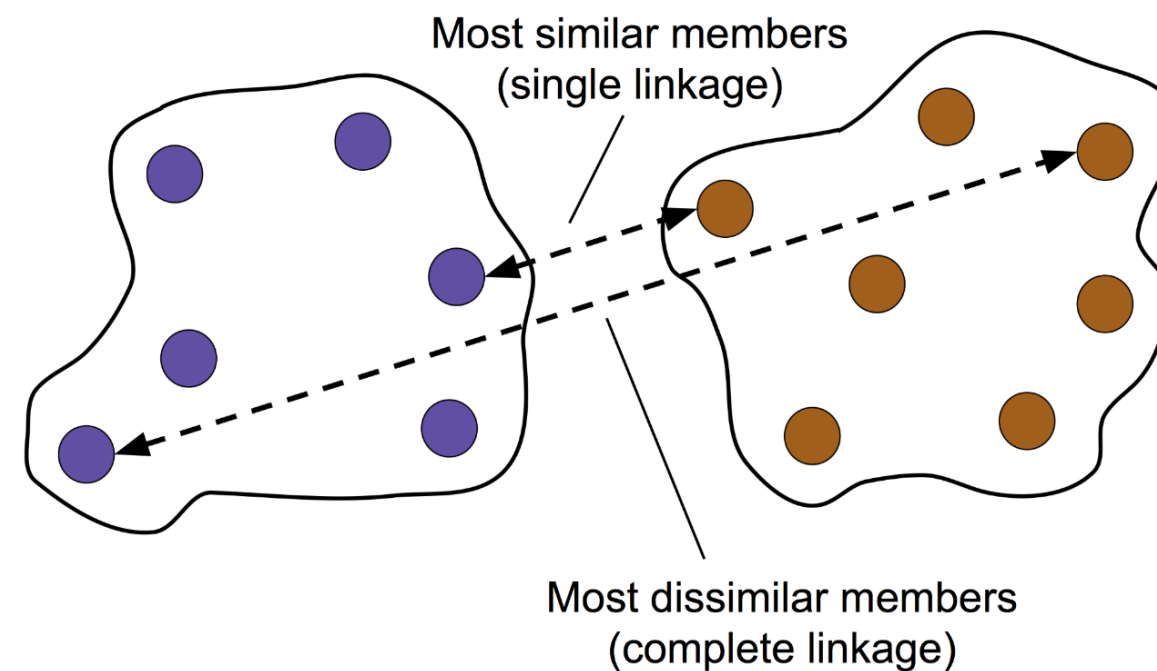
What is "close"?

- Need to define what we mean by "closeness" by choosing
 - distance metric (how to measure distance)
 - linkage criteria (how to compare clusters)

Need to define: Distance Metric

- **Euclidean**: $\sqrt{\sum_{i=1}^n (a_i - b_i)^2}$
 - easy to use analytically, sensitive to outliers
- **Manhattan**: $\sum_{i=1}^n |a_i - b_i|$
 - more difficult to use analytically, robust to outliers
- **Cosine**: $1 - \frac{\sum a_i b_i}{\|a_i\|_2 \|b_i\|_2}$
 - angle between vectors while ignoring their scale
- many more (see <https://numerics.mathdotnet.com/Distance.html>)

Need to define: Linkage



single : shortest distance from item of one cluster to item of the other

complete : greatest distance from item of one cluster to item of the other

average : average distance of items in one cluster to items in the other

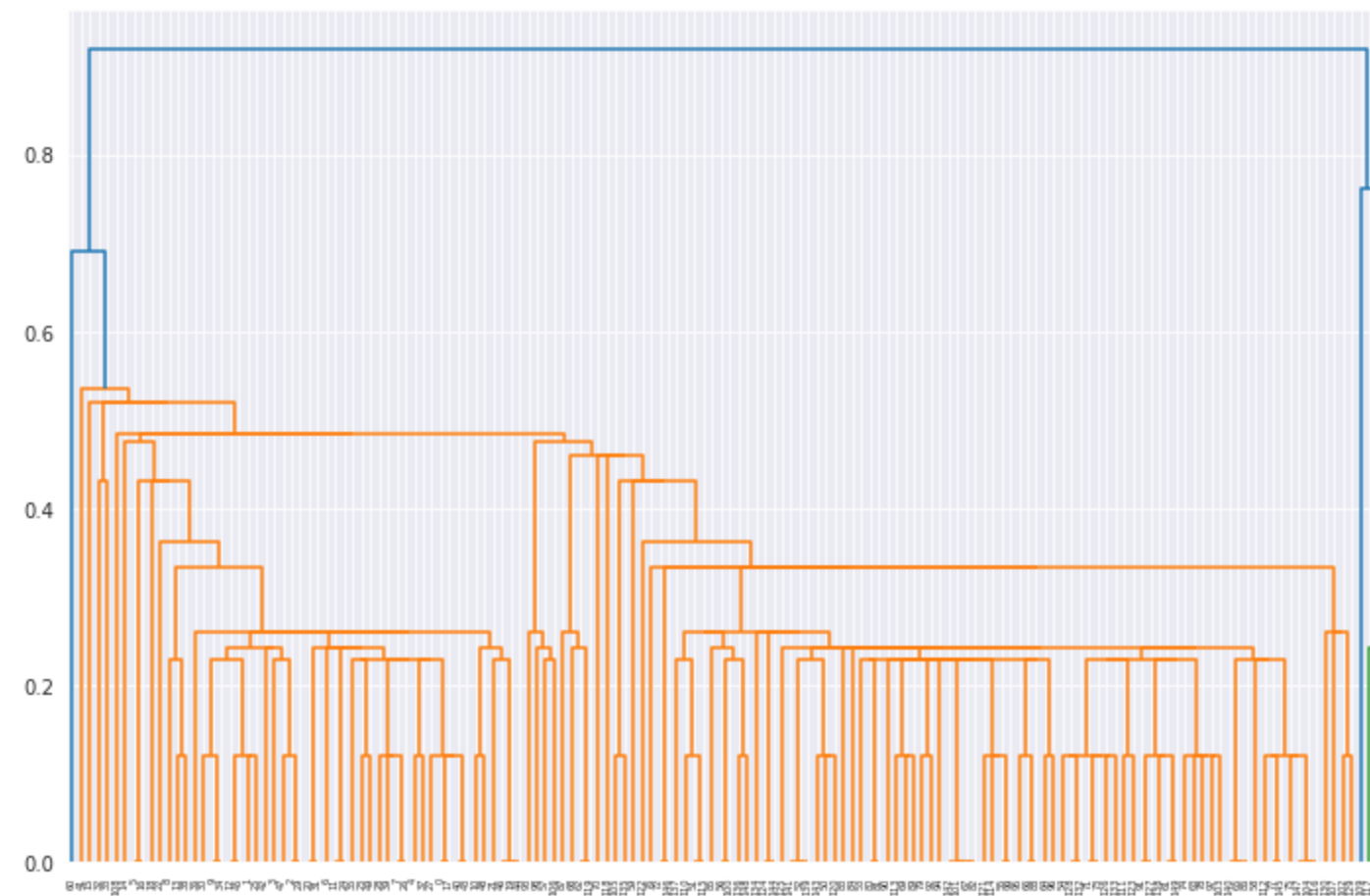
ward : minimize variance of clusters being merged (only euclidean metric)

HAC and Dendrograms: Single Linkage

HAC and Dendrograms: Single Linkage

```
In [16]: # nice helper function for creating a dendrogram
from scipy.cluster import hierarchy

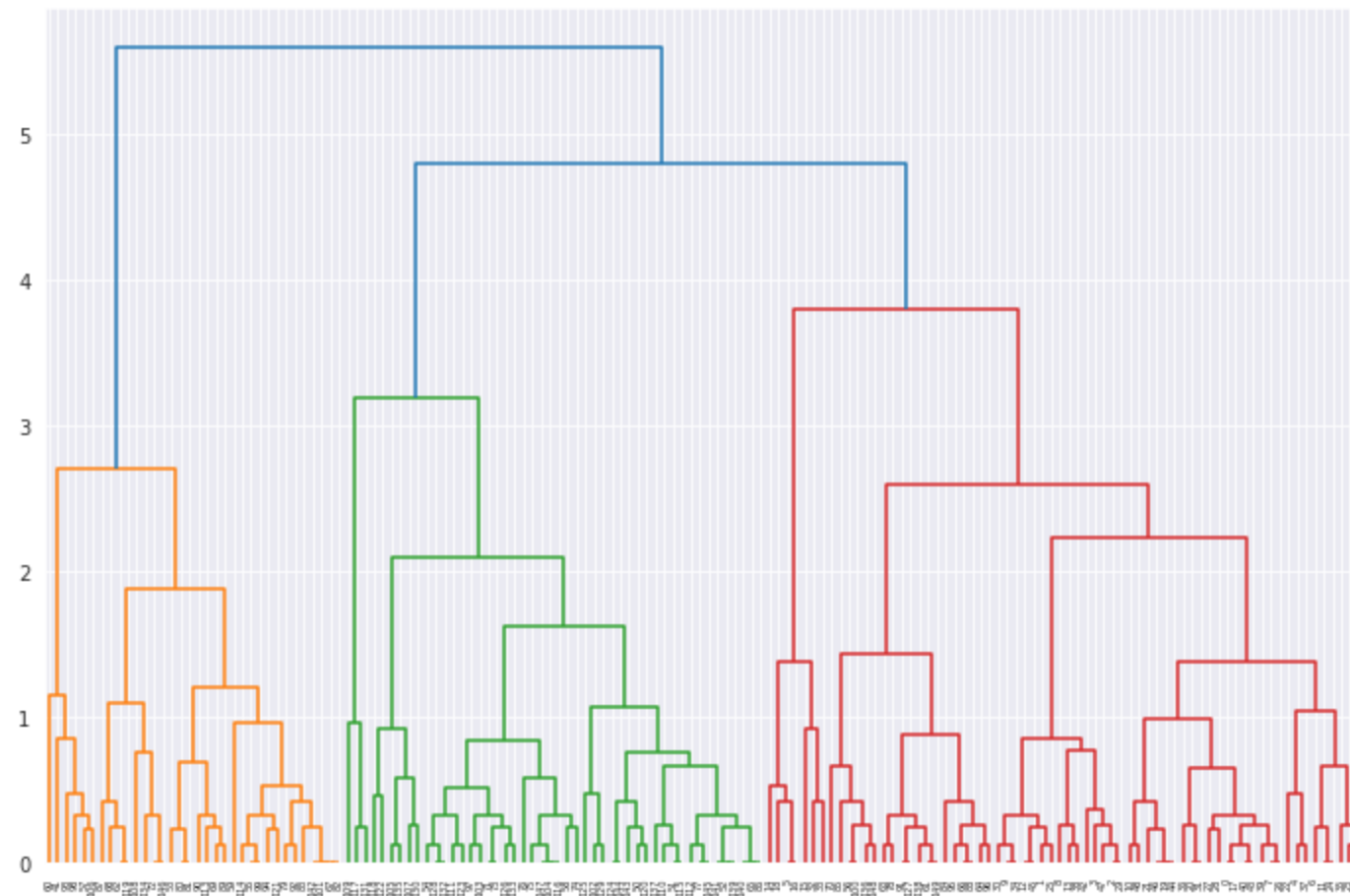
Z = hierarchy.linkage(X_iris, 'single')
fig = plt.figure(figsize=(12,8)); hierarchy.dendrogram(Z);
```



HAC and Dendrograms: Complete Linkage

HAC and Dendrograms: Complete Linkage

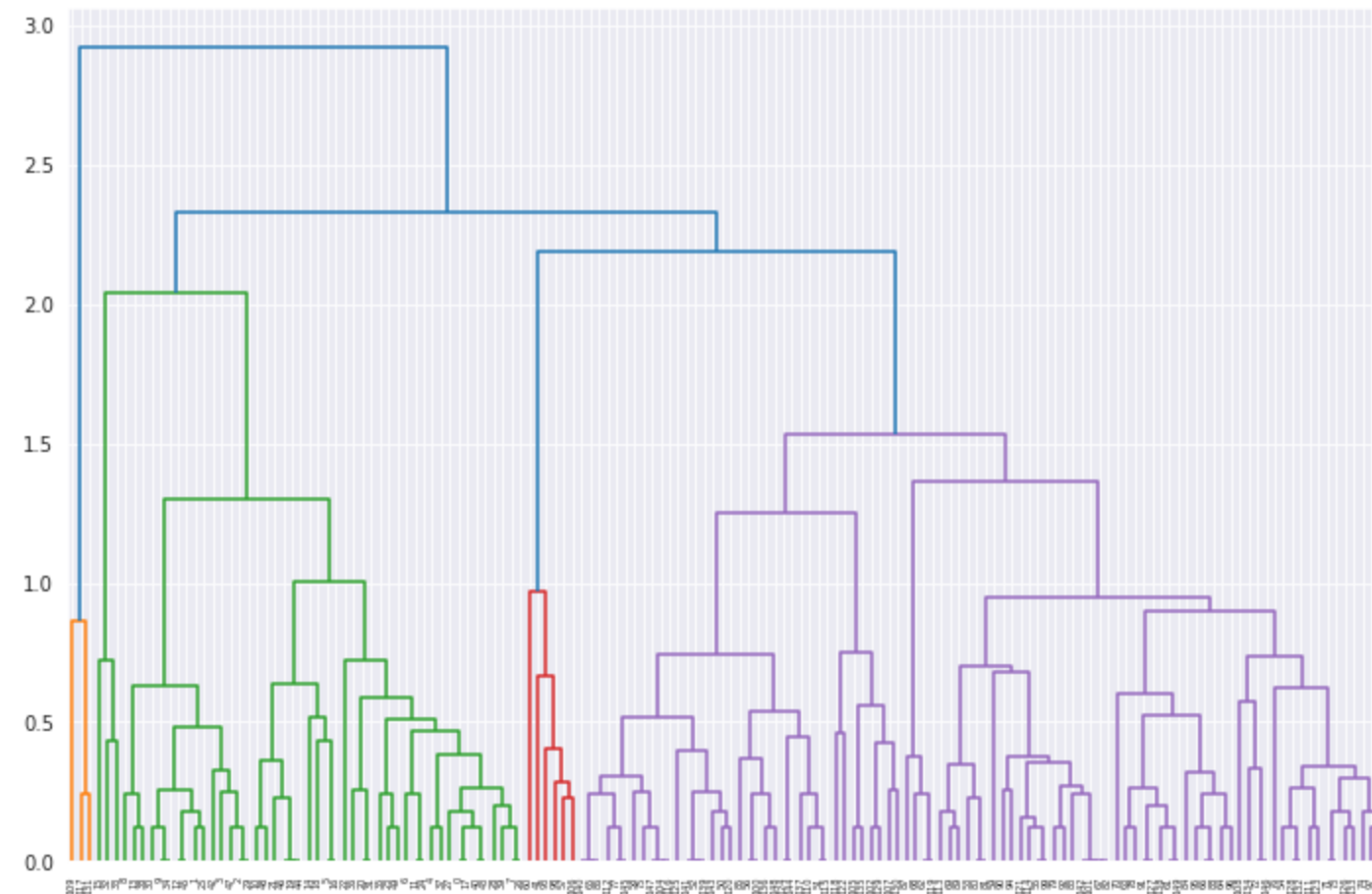
```
In [17]: Z = hierarchy.linkage(X_iris, 'complete')  
fig = plt.figure(figsize=(12,8)); hierarchy.dendrogram(Z);
```



HAC and Dendrograms: Average Linkage

HAC and Dendrograms: Average Linkage

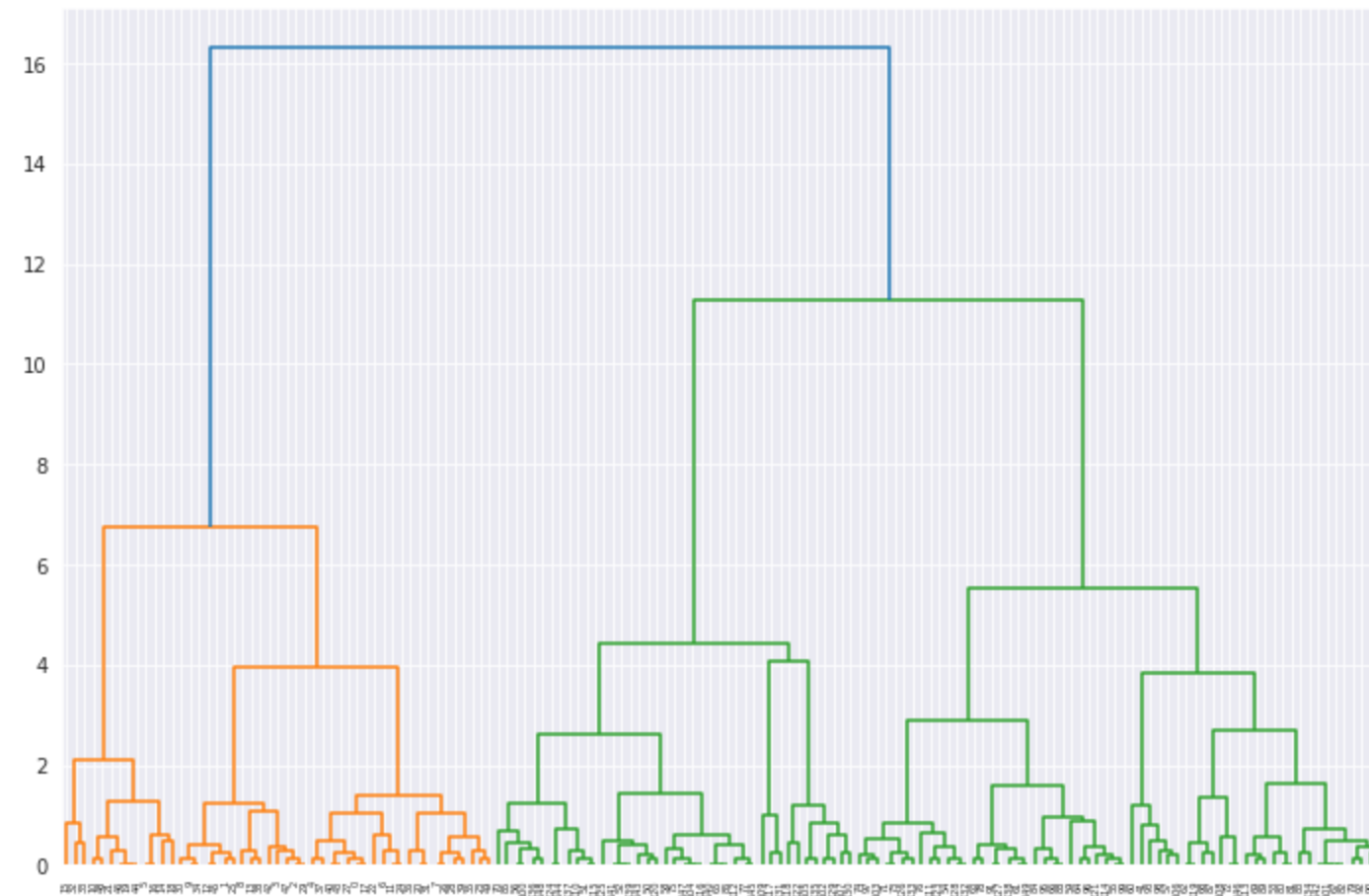
```
In [18]: Z = hierarchy.linkage(X_iris, 'average')  
fig = plt.figure(figsize=(12,8)); hierarchy.dendrogram(Z);
```



HAC and Dendrograms: Ward Linkage

HAC and Dendrograms: Ward Linkage

```
In [19]: Z = hierarchy.linkage(X_iris, 'ward')  
fig = plt.figure(figsize=(12,8)); hierarchy.dendrogram(Z);
```



HAC in sklearn

HAC in sklearn

```
In [20]: from sklearn.cluster import AgglomerativeClustering

         hac = AgglomerativeClustering(linkage='single',
                                       affinity='euclidean',
                                       n_clusters=4)
         c_single = hac.fit_predict(X_iris)

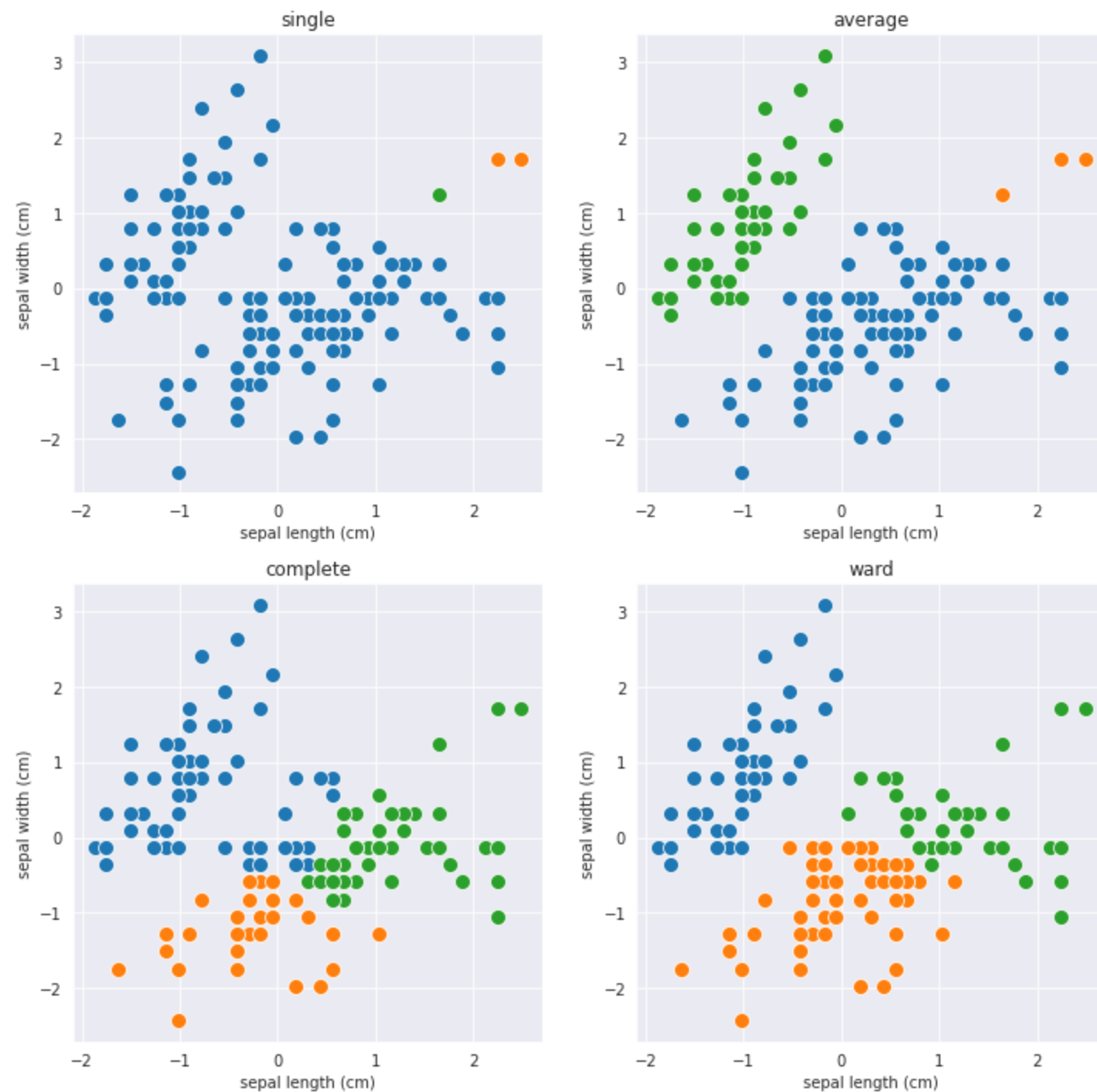
         # generate models and assignments for all linkages
         models, assignments = [], []
         linkages = ['single', 'average', 'complete', 'ward']
         for linkage in linkages:
             models.append(AgglomerativeClustering(linkage=linkage, affinity='euclidean', n_clusters=3))
             assignments.append(models[-1].fit_predict(X_iris))

         # plot on the next slide
```

HAC in sklearn

HAC in sklearn

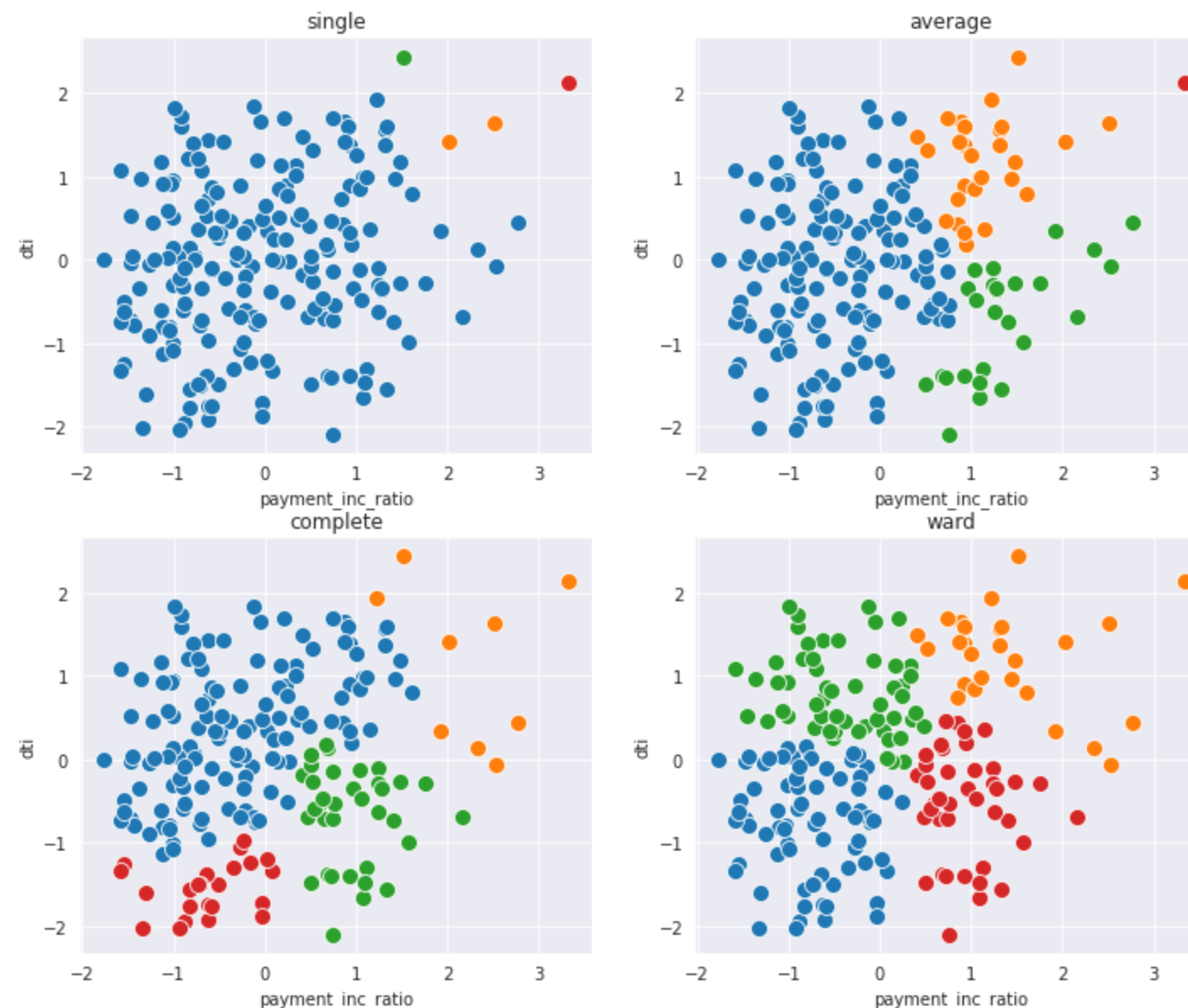
```
In [21]: fig, ax = plt.subplots(2, 2, figsize=(12, 12))
         axs = ax.flatten()
         for i in range(len(linkage)):
             plot_clusters(X_iris, assignments[i], title=linkages[i], ax=axs[i])
```



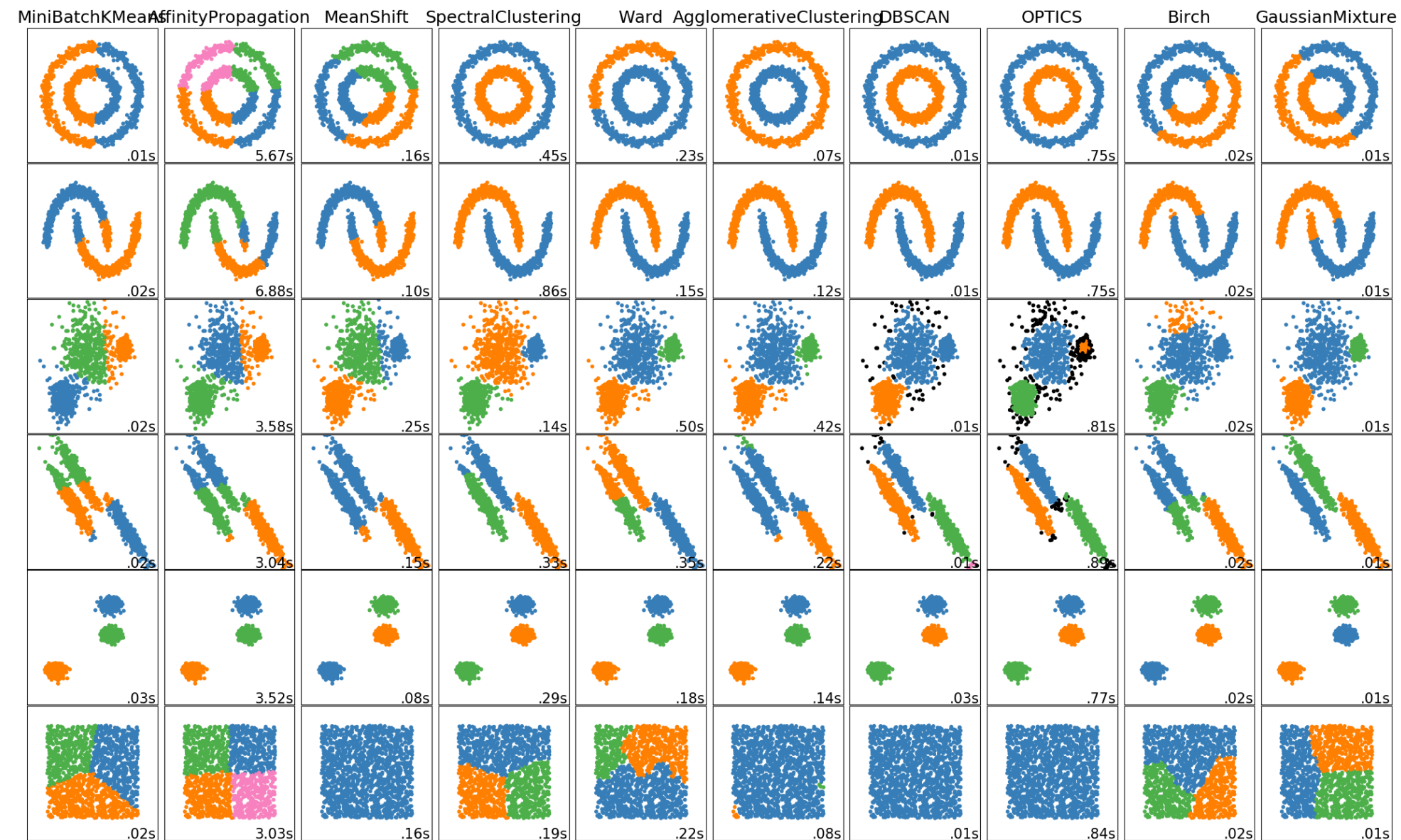
HAC: Another Example

HAC: Another Example

```
In [22]: models,assignments,linkages = [],[],['single','average','complete','ward']
for linkage in linkages:
    models.append(AgglomerativeClustering(linkage=linkage,affinity='euclidean',n_clusters=4))
    assignments.append(models[-1].fit_predict(X_loan))
fig,ax = plt.subplots(2,2,figsize=(12,10))
axs = ax.flatten()
for i in range(len(linkage)):
    plot_clusters(X_loan,assignments[i],title=linkages[i],ax=axs[i])
```



Clustering: Many Other Methods



From <https://scikit-learn.org/stable/modules/clustering.html>

How to evaluate clustering?

- Within Cluster Sum of Squared Distances (SSD)
- If we have labels
 - How "pure" are the clusters? Homogeneity
 - Mutual Information
- Silhouette plots (see PML)
- many others ([see sklearn](#))

Clustering Review

- k-Means
- Heirarchical Agglomerative Clustering
 - linkages
 - distance metrics
- Evaluating

Questions re Clustering?

Recommendation Engines

- Given a user and a set of items to recommend (or rank):
 - Recommend things **similar to the things I've liked**
 - Content-Based Filtering
 - Recommend things **that people with similar tastes have liked**
 - Collaborative Filtering
 - Hybrid/Ensemble

Example: Housing Data

Example: Housing Data

```
In [23]: df_house = pd.read_csv('../data/house_sales_subset.csv')
df_house = df_house.iloc[:10].loc[:, ['SqFtTotLiving', 'SqFtLot', 'AdjSalePrice']]
X_house_scaled = StandardScaler().fit_transform(df_house)
df_house_scaled = pd.DataFrame(X_house_scaled, columns=['SqFtTotLiving_scaled', 'SqFtLot_scaled', 'AdjSalePrice_scaled'])
df_house_scaled.head()
```

Out[23]:

	SqFtTotLiving_scaled	SqFtLot_scaled	AdjSalePrice_scaled
0	0.399969	-0.466145	-0.699629
1	2.030444	0.647921	2.479556
2	-0.006455	1.255424	1.190602
3	1.356259	-0.544149	-0.120423
4	-0.412878	-0.543943	-0.714964

Content-Based Filtering

- Find **other things** similar to **the things I've liked**
- Assume: If I like product A, and product B is like product A, I'll like product B
- Use similarity of items
- Matrix: items x items
- Values: Similarity of items

Calculate Distances

- to maximize similarity \rightarrow minimize distance

Calculate Distances

- to maximize similarity → minimize distance

```
In [24]: # using euclidean distance
from sklearn.metrics.pairwise import euclidean_distances

# calculate all pairwise distances between houses
dists = euclidean_distances(X_house_scaled)

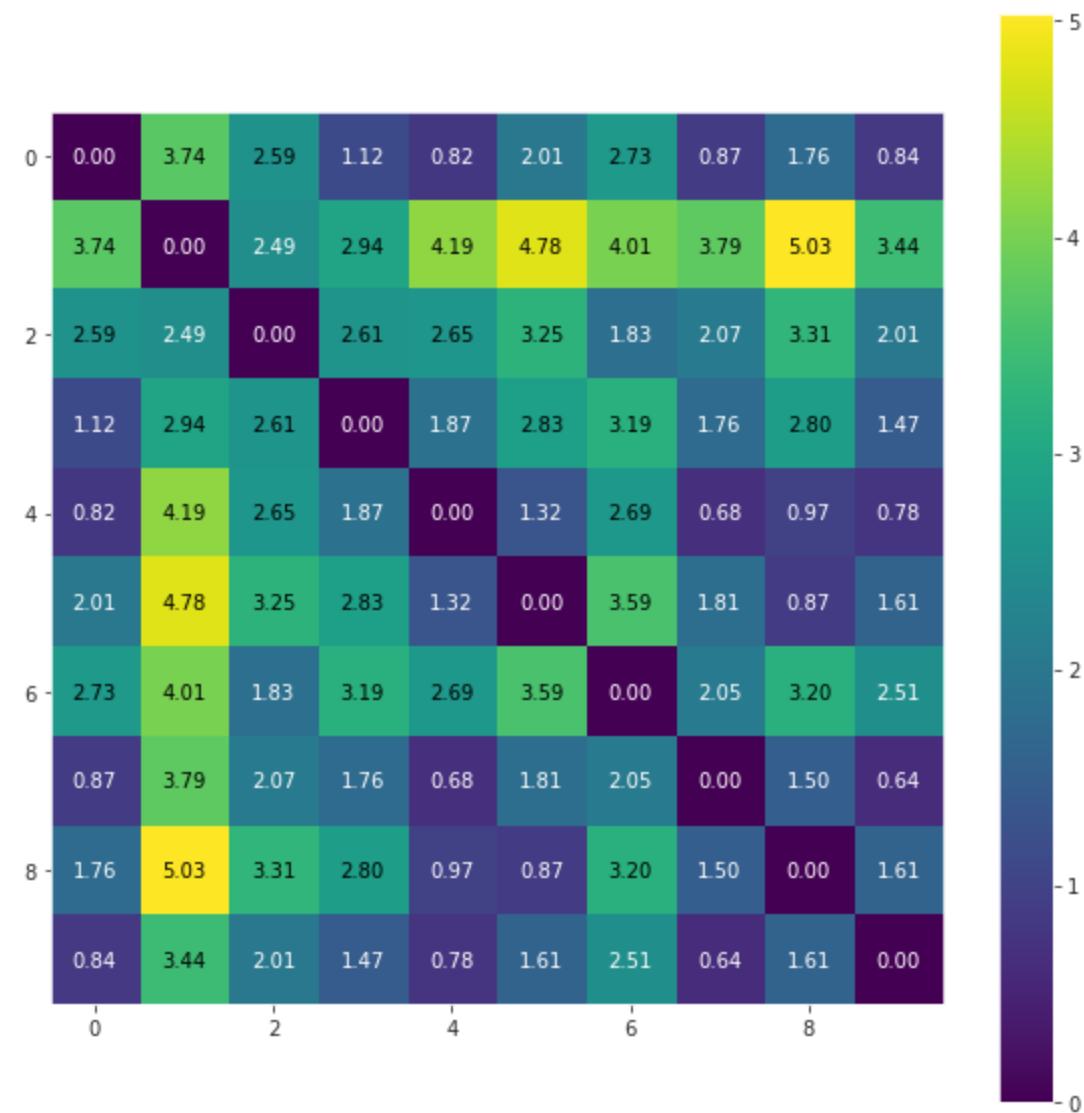
np.round(dists,2)
```

```
Out[24]: array([[0.   , 3.74, 2.59, 1.12, 0.82, 2.01, 2.73, 0.87, 1.76, 0.84],
 [3.74, 0.   , 2.49, 2.94, 4.19, 4.78, 4.01, 3.79, 5.03, 3.44],
 [2.59, 2.49, 0.   , 2.61, 2.65, 3.25, 1.83, 2.07, 3.31, 2.01],
 [1.12, 2.94, 2.61, 0.   , 1.87, 2.83, 3.19, 1.76, 2.8 , 1.47],
 [0.82, 4.19, 2.65, 1.87, 0.   , 1.32, 2.69, 0.68, 0.97, 0.78],
 [2.01, 4.78, 3.25, 2.83, 1.32, 0.   , 3.59, 1.81, 0.87, 1.61],
 [2.73, 4.01, 1.83, 3.19, 2.69, 3.59, 0.   , 2.05, 3.2 , 2.51],
 [0.87, 3.79, 2.07, 1.76, 0.68, 1.81, 2.05, 0.   , 1.5 , 0.64],
 [1.76, 5.03, 3.31, 2.8 , 0.97, 0.87, 3.2 , 1.5 , 0.   , 1.61],
 [0.84, 3.44, 2.01, 1.47, 0.78, 1.61, 2.51, 0.64, 1.61, 0.   ]])
```


Visualizing Distances With a Heatmap

Visualizing Distances With a Heatmap

```
In [25]: from mlxtend.plotting import heatmap
heatmap(np.round(dists, 2), figsize=(10, 10));
```



Query For Similarity

- Imagine I like house 5
- What houses are similar to house 5?

Query For Similarity

- Imagine I like house 5
- What houses are similar to house 5?

```
In [26]: query_idx = 5  
df_house.iloc[query_idx]
```

```
Out[26]: SqFtTotLiving    930.0  
SqFtLot              1012.0  
AdjSalePrice         411781.0  
Name: 5, dtype: float64
```

Query For Similarity

- Imagine I like house 5
- What houses are similar to house 5?

```
In [26]: query_idx = 5  
df_house.iloc[query_idx]
```

```
Out[26]: SqFtTotLiving    930.0  
         SqFtLot        1012.0  
         AdjSalePrice   411781.0  
         Name: 5, dtype: float64
```

```
In [27]: # Distances to house 5  
[f'{x:0.1f}' for x in dists[query_idx]]
```

```
Out[27]: ['2.0', '4.8', '3.3', '2.8', '1.3', '0.0', '3.6', '1.8', '0.9', '1.6']
```

Query For Similarity Cont.

Query For Similarity Cont.

```
In [28]: # find indexes of best scores (for distances, want ascending)
         best_idxasc = np.argsort(dists[query_idx])
         best_idxasc
```

```
Out[28]: array([5, 8, 4, 9, 7, 0, 3, 2, 6, 1])
```

Query For Similarity Cont.

```
In [28]: # find indexes of best scores (for distances, want ascending)
best_idx asc = np.argsort(dists[query_idx])
best_idx asc
```























```
Out[28]: array([5, 8, 4, 9, 7, 0, 3, 2, 6, 1])
```

```
In [75]: # the top 10 recommendations with their distances
list(zip(['house ' + str(x) for x in best_idx asc],
        np.round(dists[query_idx][best_idx asc], 2)
        )
    )
```

```
Out[75]: [('house 5', 0.0),
          ('house 8', 0.87),
          ('house 4', 1.32),
          ('house 9', 1.61),
          ('house 7', 1.81),
          ('house 0', 2.01),
          ('house 3', 2.83),
          ('house 2', 3.25),
          ('house 6', 3.59),
          ('house 1', 4.78)]
```


(User Based) Collaborative Filtering

- Recommend things that people with similar tastes have liked
- Assume: If both you and I like Movie A, and you like Movie B, I'll like movie B
- Use similarity of user preferences
- Matrix: Users x Items
- Values: Rankings

					
A					
B					
C					
D					
E					

Example: User Interests

Can we recommend topics based on a users existing interests?

Example: User Interests

Can we recommend topics based on a users existing interests?

```
In [30]: # from Data Science from Scratch by Joel Grus
#https://github.com/joelgrus/data-science-from-scratch.git

users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

Example: User Interests

Can we recommend topics based on a users existing interests?

```
In [30]: # from Data Science from Scratch by Joel Grus
#https://github.com/joelgrus/data-science-from-scratch.git

users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

```
In [31]: # interests of user0
sorted(users_interests[0])
```

```
Out[31]: ['Big Data', 'Cassandra', 'HBase', 'Hadoop', 'Java', 'Spark', 'Storm']
```

All Unique Interests

All Unique Interests

```
In [32]: # get a sorted list of unique interests (here using set)
unique_interests = sorted({interest
                           for user_interests in users_interests
                           for interest in user_interests})

# the first 5 unique interests
unique_interests
```

```
Out[32]: ['Big Data',
          'C++',
          'Cassandra',
          'HBase',
          'Hadoop',
          'Haskell',
          'Java',
          'Mahout',
          'MapReduce',
          'MongoDB',
          'MySQL',
          'NoSQL',
          'Postgres',
          'Python',
          'R',
          'Spark',
          'Storm',
          'artificial intelligence',
          'databases',
          'decision trees',
          'deep learning',
          'libsvm',
          'machine learning',
          'mathematics',
          'neural networks',
          'numpy',
          'pandas',
          'probability']
```

Transform User Interest Matrix

Transform User Interest Matrix

```
In [33]: # Transform between lists of strings and fixed length lists of ints
from sklearn.preprocessing import MultiLabelBinarizer

mlb = MultiLabelBinarizer(classes=unique_interests)

# a matrix of "user" rows and "interest" columns
user_interest_matrix = mlb.fit_transform(users_interests)

# The interests for user0
user_interest_matrix[0]
```

```
Out[33]: array([1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```


Transform User Interest Matrix

```
In [33]: # Transform between lists of strings and fixed length lists of ints
from sklearn.preprocessing import MultiLabelBinarizer

mlb = MultiLabelBinarizer(classes=unique_interests)

# a matrix of "user" rows and "interest" columns
user_interest_matrix = mlb.fit_transform(users_interests)

# The interests for user0
user_interest_matrix[0]
```

```
Out[33]: array([1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
In [34]: # transforming back from interest matrix to list of interests
mlb.inverse_transform(user_interest_matrix)[0]
```

```
Out[34]: ('Big Data', 'Cassandra', 'HBase', 'Hadoop', 'Java', 'Spark', 'Storm')
```

Calculate Similarity

Calculate Similarity

```
In [35]: from sklearn.metrics.pairwise import cosine_similarity

# using similarity, higher values are better
user_similarities = cosine_similarity(user_interest_matrix)

# what are the similarites for user0 to other users?
user_similarities[0]
```

Out[35]: array([1. , 0.3380617 , 0. , 0. , 0. ,
0.15430335, 0. , 0. , 0.18898224, 0.56694671,
0. , 0. , 0. , 0.16903085, 0.])

Calculate Similarity

```
In [35]: from sklearn.metrics.pairwise import cosine_similarity

# using similarity, higher values are better
user_similarities = cosine_similarity(user_interest_matrix)

# what are the similarites for user0 to other users?
user_similarities[0]
```

```
Out[35]: array([1.          , 0.3380617 , 0.          , 0.          , 0.          ,
                0.15430335, 0.          , 0.          , 0.18898224, 0.56694671,
                0.          , 0.          , 0.          , 0.16903085, 0.          ])
```

```
In [36]: # what users does user0 share interests with?
np.where(user_similarities[0])[0]
```

```
Out[36]: array([ 0,  1,  5,  8,  9, 13])
```

Find Similar Users

Find Similar Users

```
In [37]: # return a sorted list of users based on similarity  
# skip query user and similarity == 0  
def most_similar_users_to(query_idx):  
    users_scores = [(idx, np.round(sim, 4))  
                    for idx, sim in enumerate(user_similarities[query_idx])  
                    if idx != query_idx and sim > 0]  
    return sorted(users_scores, key=lambda x: x[1])  
  
most_similar_users_to(0)
```

```
Out[37]: [(5, 0.1543), (13, 0.169), (8, 0.189), (1, 0.3381), (9, 0.5669)]
```

Recommend Based On User Similarity

- Want to return items sorted by the similarity of other users

Recommend Based On User Similarity

- Want to return items sorted by the similarity of other users

```
In [38]: from collections import defaultdict

def user_based_suggestions(user_idx):
    suggestions = defaultdict(float)

    # iterate over interests of similar users
    for other_idx, sim in most_similar_users_to(user_idx):
        for interest in users_interests[other_idx]:
            suggestions[interest] += sim

    # sort suggestions based on weight
    suggestions = sorted(suggestions.items(),
                        key=lambda x:x[1],
                        reverse=True)

    # return only new interests
    return [(suggestion, weight)
            for suggestion, weight in suggestions
            if suggestion not in users_interests[user_idx]]
```


Recommend Based On User Similarity

Recommend Based On User Similarity

```
In [39]: # reminder: original interests  
users_interests[0]
```

```
Out[39]: ['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
```

Recommend Based On User Similarity

```
In [39]: # reminder: original interests
users_interests[0]
```

```
Out[39]: ['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
```

```
In [40]: # top 5 new recommended interests
user_based_suggestions(0)[:5]
```

```
Out[40]: [('MapReduce', 0.5669),
          ('Postgres', 0.5071),
          ('MongoDB', 0.5071),
          ('NoSQL', 0.3381),
          ('neural networks', 0.189)]
```

Issues with Collab. Filtering

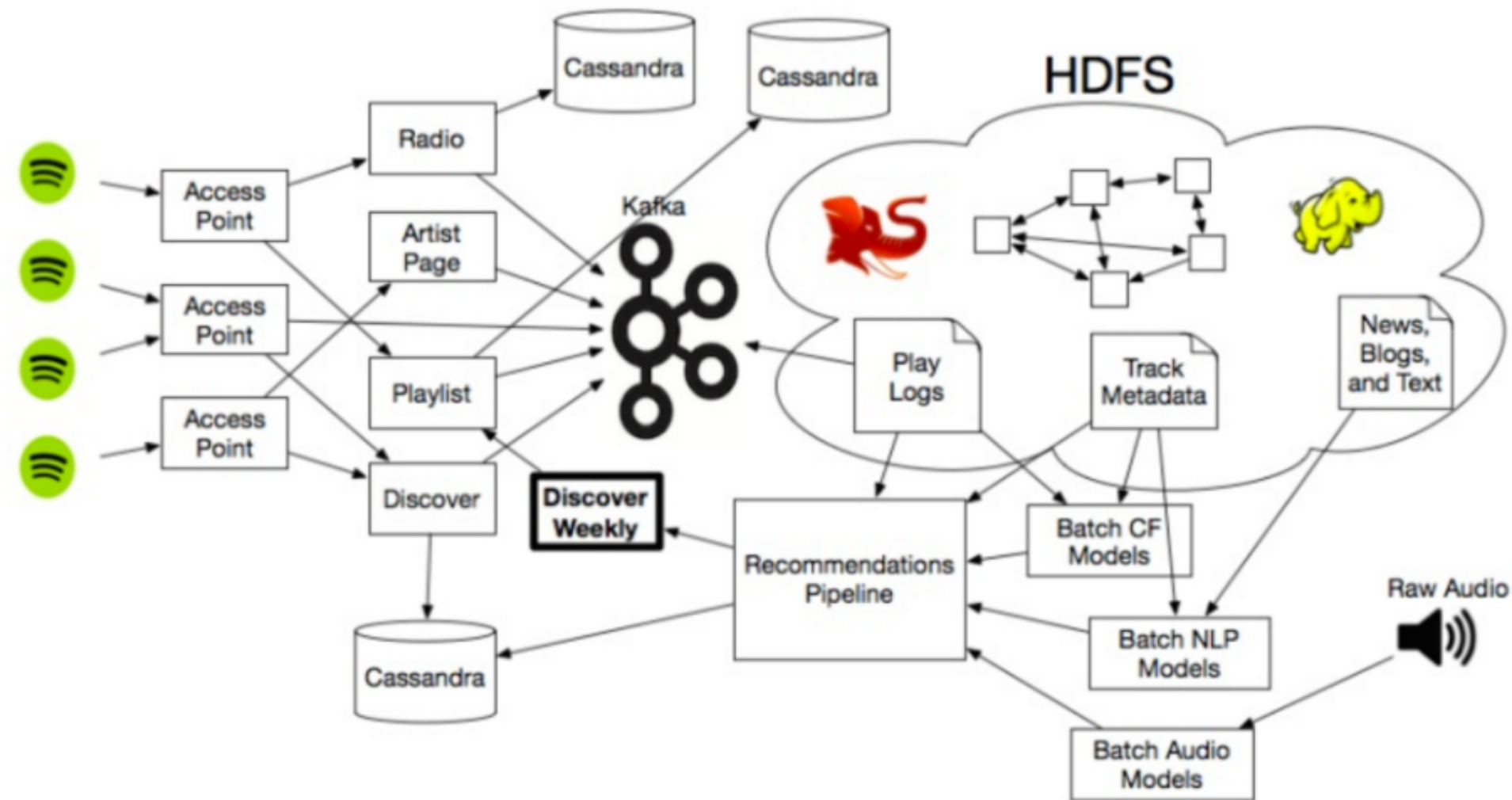
- **the cold start problem** : What if it's your first time?
- **sparsity** : How to recommend movies no one's seen?

Evaluating Rec. Systems

- **Precision@N**: Out of top N, how many were true?
- **Recall@N**: Out of all true, how many were in top N
- Surprise/Novelty?
- Diversity?

Spotify's Recommendation Engine

How Does Spotify Know You So Well?



Recommendation Engines Review

- Content-Based
- User-Based Collaborative Filtering
- Issues
- Evaluating

Questions re Recommendation Engines?

Time Series

- Data ordered in time
- Applications
 - Financial
 - Economic
 - Scientific
 - etc.

Time Series Differences

- **Non-i.i.d.** : not independent and identically distributed
- not independent
 - Ex: Stock price
- not-identically distributed
 - Ex: Seasonality
- Order matters

Representing Time in Python

- `datetime` library
- Pandas `Timestamp`

datetime.date

datetime.date

```
In [41]: from datetime import date  
  
         friday = date(2020,12,4) # year,month,day  
         friday
```

```
Out[41]: datetime.date(2020, 12, 4)
```

datetime.date

```
In [41]: from datetime import date  
  
         friday = date(2020,12,4) # year,month,day  
         friday
```

```
Out[41]: datetime.date(2020, 12, 4)
```

```
In [42]: today = date.today()  
         today
```

```
Out[42]: datetime.date(2021, 11, 29)
```

datetime.date

```
In [41]: from datetime import date

        friday = date(2020,12,4) # year,month,day
        friday
```

```
Out[41]: datetime.date(2020, 12, 4)
```

```
In [42]: today = date.today()
        today
```

```
Out[42]: datetime.date(2021, 11, 29)
```

```
In [43]: today.year
```

```
Out[43]: 2021
```

`datetime.time`

datetime.time

```
In [44]: from datetime import time  
  
noon = time(12,0,0) # hour,minute,second,microsecond  
noon
```

```
Out[44]: datetime.time(12, 0)
```

datetime.time

```
In [44]: from datetime import time

noon = time(12,0,0) # hour,minute,second,microsecond
noon
```

```
Out[44]: datetime.time(12, 0)
```

```
In [45]: noon.hour
```

```
Out[45]: 12
```

`datetime.datetime`

datetime.datetime

```
In [46]: from datetime import datetime

         # year, month, day, hour, minute, second, microsecond
         monday_afternoon = datetime(2020, 11, 30, 19, 10)
         monday_afternoon
```

```
Out[46]: datetime.datetime(2020, 11, 30, 19, 10)
```

datetime.datetime

```
In [46]: from datetime import datetime

# year, month, day, hour, minute, second, microsecond
monday_afternoon = datetime(2020, 11, 30, 19, 10)
monday_afternoon
```

```
Out[46]: datetime.datetime(2020, 11, 30, 19, 10)
```

```
In [47]: now = datetime.now()
now
```

```
Out[47]: datetime.datetime(2021, 11, 29, 14, 54, 19, 41336)
```

`datetime.timedelta`

datetime.timedelta

```
In [48]: diff = datetime(2020,11,30,1) - datetime(2020,11,29,0)
diff
```

```
Out[48]: datetime.timedelta(days=1, seconds=3600)
```

datetime.timedelta

```
In [48]: diff = datetime(2020,11,30,1) - datetime(2020,11,29,0)
diff
```

```
Out[48]: datetime.timedelta(days=1, seconds=3600)
```

```
In [49]: diff.total_seconds()
```

```
Out[49]: 90000.0
```


datetime.timedelta

```
In [48]: diff = datetime(2020,11,30,1) - datetime(2020,11,29,0)
diff
```

```
Out[48]: datetime.timedelta(days=1, seconds=3600)
```

```
In [49]: diff.total_seconds()
```

```
Out[49]: 90000.0
```

```
In [50]: from datetime import timedelta

#days,seconds,microseconds,milliseconds,minutes,hours,weeks
one_day = timedelta(1)

date(2020,11,30) + 2*one_day
```

```
Out[50]: datetime.date(2020, 12, 2)
```

Printing Datetimes: `strftime()`

Printing Datetimes: `strftime()`

```
In [51]: print(now)
```

```
2021-11-29 14:54:19.041336
```

Printing Datetimes: `strftime()`

In [51]: `print(now)`

2021-11-29 14:54:19.041336

In [52]: `now.strftime('%a %h %d, %Y %I:%M %p')`

Out[52]: 'Mon Nov 29, 2021 02:54 PM'

Printing Datetimes: `strftime()`

```
In [51]: print(now)
```

```
2021-11-29 14:54:19.041336
```

```
In [52]: now.strftime('%a %h %d, %Y %I:%M %p')
```

```
Out[52]: 'Mon Nov 29, 2021 02:54 PM'
```

```
%Y 4-digit year  
%y 2-digit year  
%m 2-digit month  
%d 2-digit day  
%H Hour (24-hour)  
%M 2-digit minute  
%S 2-digit second
```

Printing Datetimes: `strftime()`

```
In [51]: print(now)
```

```
2021-11-29 14:54:19.041336
```

```
In [52]: now.strftime('%a %h %d, %Y %I:%M %p')
```

```
Out[52]: 'Mon Nov 29, 2021 02:54 PM'
```

```
%Y 4-digit year  
%y 2-digit year  
%m 2-digit month  
%d 2-digit day  
%H Hour (24-hour)  
%M 2-digit minute  
%S 2-digit second
```

See strftime.org

Parsing Datetimes: `pandas.to_datetime()`

- `dateutil.parser` available
- pandas has parser built in: `pd.to_datetime()`

Parsing Datetimes: `pandas.to_datetime()`

- `dateutil.parser` available
- pandas has parser built in: `pd.to_datetime()`

```
In [53]: pd.to_datetime('11/22/2019 2:36pm')
```

```
Out[53]: Timestamp('2019-11-22 14:36:00')
```


Parsing Datetimes: `pandas.to_datetime()`

- `dateutil.parser` available
- pandas has parser built in: `pd.to_datetime()`

```
In [53]: pd.to_datetime('11/22/2019 2:36pm')
```

```
Out[53]: Timestamp('2019-11-22 14:36:00')
```

```
In [54]: dt_index = pd.to_datetime([datetime(2020, 11, 26),  
                                     '27th of November, 2020',  
                                     '2020-Nov-28',  
                                     '11-29-2030',  
                                     '20201130',  
                                     None  
                                     ])  
  
dt_index
```

```
Out[54]: DatetimeIndex(['2020-11-26', '2020-11-27', '2020-11-28', '2030-11-29',  
                        '2020-11-30', 'NaT'],  
                        dtype='datetime64[ns]', freq=None)
```

pandas.Timestamp

- like datetime.datetime
- can include **timezone** and **frequency** info
- can handle a missing time: NaT
- can be used anywhere datetime can be used
- an array of Timestamps can be used as an index

pandas.Timestamp

- like datetime.datetime
- can include **timezone** and **frequency** info
- can handle a missing time: NaT
- can be used anywhere datetime can be used
- an array of Timestamps can be used as an index

```
In [55]: dt_index[0]
```

```
Out[55]: Timestamp('2020-11-26 00:00:00')
```

DateIndex Indexing/Selecting/Slicing

DateIndex Indexing/Selecting/Slicing

```
In [56]: s = pd.Series([101,102,103],  
                        index=pd.to_datetime(['20191201', '20200101', '20200201']))  
s
```

```
Out[56]: 2019-12-01    101  
         2020-01-01    102  
         2020-02-01    103  
         dtype: int64
```

DateIndex Indexing/Selecting/Slicing

```
In [56]: s = pd.Series([101,102,103],  
                      index=pd.to_datetime(['20191201', '20200101', '20200201']))  
s
```

```
Out[56]: 2019-12-01    101  
         2020-01-01    102  
         2020-02-01    103  
         dtype: int64
```

```
In [57]: # can index normally using iloc  
s.iloc[0:2]
```

```
Out[57]: 2019-12-01    101  
         2020-01-01    102  
         dtype: int64
```

DateIndex Indexing/Selecting/Slicing Cont.

DateIndex Indexing/Selecting/Slicing Cont.

```
In [58]: # only rows from the year 2020  
s.loc['2020']
```

```
Out[58]: 2020-01-01    102  
         2020-02-01    103  
         dtype: int64
```


DateIndex Indexing/Selecting/Slicing Cont.

```
In [58]: # only rows from the year 2020  
s.loc['2020']
```

```
Out[58]: 2020-01-01    102  
         2020-02-01    103  
         dtype: int64
```

```
In [59]: # only rows from January 2020  
s.loc['2020-01']
```

```
Out[59]: 2020-01-01    102  
         dtype: int64
```

DateIndex Indexing/Selecting/Slicing Cont.

```
In [58]: # only rows from the year 2020  
s.loc['2020']
```

```
Out[58]: 2020-01-01    102  
         2020-02-01    103  
         dtype: int64
```

```
In [59]: # only rows from January 2020  
s.loc['2020-01']
```

```
Out[59]: 2020-01-01    102  
         dtype: int64
```

```
In [60]: # only rows between Jan 1st 2019 and Jan 1st 2020, inclusive  
s.loc['01/01/2019':'01/01/2020']
```

```
Out[60]: 2019-12-01    101  
         2020-01-01    102  
         dtype: int64
```

DateIndex Indexing/Selecting/Slicing Cont.

```
In [58]: # only rows from the year 2020  
s.loc['2020']
```

```
Out[58]: 2020-01-01    102  
         2020-02-01    103  
         dtype: int64
```

```
In [59]: # only rows from January 2020  
s.loc['2020-01']
```

```
Out[59]: 2020-01-01    102  
         dtype: int64
```

```
In [60]: # only rows between Jan 1st 2019 and Jan 1st 2020, inclusive  
s.loc['01/01/2019':'01/01/2020']
```

```
Out[60]: 2019-12-01    101  
         2020-01-01    102  
         dtype: int64
```

```
In [61]: # can use the indexing shortcut  
s['2020']
```

```
Out[61]: 2020-01-01    102  
         2020-02-01    103  
         dtype: int64
```

Datetimes in DataFrames

Datetimes in DataFrames

```
In [62]: df = pd.DataFrame([['12/1/2020', 101, 'A'],  
                             ['1/1/2021', 102, 'B']], columns=['col1', 'col2', 'col3'])  
df.col1 = pd.to_datetime(df.col1)  
df.set_index('col1', drop=True, inplace=True)  
df
```

Out[62]:

	col2	col3
col1		
2020-12-01	101	A
2021-01-01	102	B

Datetimes in DataFrames

```
In [62]: df = pd.DataFrame([['12/1/2020', 101, 'A'],  
                           ['1/1/2021', 102, 'B']], columns=['col1', 'col2', 'col3'])  
df.col1 = pd.to_datetime(df.col1)  
df.set_index('col1', drop=True, inplace=True)  
df
```

Out[62]:

	col2	col3
col1		
2020-12-01	101	A
2021-01-01	102	B

```
In [63]: # only return rows from 2020  
df.loc['2020']
```

Out[63]:

	col2	col3
col1		
2020-12-01	101	A

Timestamp Index: Setting Frequency

Timestamp Index: Setting Frequency

```
In [64]: s = pd.Series([101, 103], index=pd.to_datetime(['20201201', '20201203']))  
s
```

```
Out[64]: 2020-12-01    101  
         2020-12-03    103  
         dtype: int64
```


Timestamp Index: Setting Frequency

```
In [64]: s = pd.Series([101,103],index=pd.to_datetime(['20201201','20201203']))  
s
```

```
Out[64]: 2020-12-01    101  
         2020-12-03    103  
         dtype: int64
```

```
In [65]: # Use resample() and asfreq() to set frequency  
s.resample('D').asfreq()
```

```
Out[65]: 2020-12-01    101.0  
         2020-12-02      NaN  
         2020-12-03    103.0  
         Freq: D, dtype: float64
```

Timestamp Index: Setting Frequency

```
In [64]: s = pd.Series([101,103],index=pd.to_datetime(['20201201','20201203']))  
s
```

```
Out[64]: 2020-12-01    101  
         2020-12-03    103  
         dtype: int64
```

```
In [65]: # Use resample() and asfreq() to set frequency  
s.resample('D').asfreq()
```

```
Out[65]: 2020-12-01    101.0  
         2020-12-02      NaN  
         2020-12-03    103.0  
         Freq: D, dtype: float64
```

```
In [66]: pd.to_datetime(['20191201','20191203'])
```

```
Out[66]: DatetimeIndex(['2019-12-01', '2019-12-03'], dtype='datetime64[ns]', freq=None)
```

Timestamp Index: Setting Frequency

```
In [64]: s = pd.Series([101,103],index=pd.to_datetime(['20201201','20201203']))  
s
```

```
Out[64]: 2020-12-01    101  
         2020-12-03    103  
         dtype: int64
```

```
In [65]: # Use resample() and asfreq() to set frequency  
s.resample('D').asfreq()
```

```
Out[65]: 2020-12-01    101.0  
         2020-12-02      NaN  
         2020-12-03    103.0  
         Freq: D, dtype: float64
```

```
In [66]: pd.to_datetime(['20191201','20191203'])
```

```
Out[66]: DatetimeIndex(['2019-12-01', '2019-12-03'], dtype='datetime64[ns]', freq=None)
```

```
In [67]: # Use date_range with freq to get a range of dates of a certain frequency  
pd.date_range(start='20191201',end='20191203',freq='D')
```

```
Out[67]: DatetimeIndex(['2019-12-01', '2019-12-02', '2019-12-03'], dtype='datetime64[ns]', freq='D')
```

Sample of Available Frequencies

B	business day frequency
D	calendar day frequency
W	weekly frequency
M	month end frequency
BM	business month end frequency
...	
Q	quarter end frequency
BQ	business quarter end frequency
...	
Y	year end frequency
BY	business year end frequency
...	
BH	business hour frequency
H	hourly frequency
T,min	minutely frequency
S	secondly frequency
L,ms	milliseconds
U,us	microseconds
N	nanoseconds

Timezones

- Handled by `pytz` library

Timezones

- Handled by `pytz` library

```
In [68]: import pytz

[x for x in pytz.common_timezones if x.startswith('U')]
```

```
Out[68]: ['US/Alaska',
          'US/Arizona',
          'US/Central',
          'US/Eastern',
          'US/Hawaii',
          'US/Mountain',
          'US/Pacific',
          'UTC']
```

Timezones

- Handled by `pytz` library

```
In [68]: import pytz

[x for x in pytz.common_timezones if x.startswith('U')]

Out[68]: ['US/Alaska',
          'US/Arizona',
          'US/Central',
          'US/Eastern',
          'US/Hawaii',
          'US/Mountain',
          'US/Pacific',
          'UTC']
```

UTC: coordinated universal time (EST is 5 hours behind, -5:00)

Timezones Cont.

Timezones Cont.

```
In [69]: ts = pd.date_range('11/2/2019 9:30am', periods=2, freq='D')  
ts
```

```
Out[69]: DatetimeIndex(['2019-11-02 09:30:00', '2019-11-03 09:30:00'], dtype='datetime64[ns]', freq='D')
```

Timezones Cont.

```
In [69]: ts = pd.date_range('11/2/2019 9:30am', periods=2, freq='D')
         ts
```

```
Out[69]: DatetimeIndex(['2019-11-02 09:30:00', '2019-11-03 09:30:00'], dtype='datetime64[ns]', freq='D')
```

```
In [70]: # Set timezone using .localize()
         ts_utc = ts.tz_localize('UTC')
         ts_utc
```

```
Out[70]: DatetimeIndex(['2019-11-02 09:30:00+00:00', '2019-11-03 09:30:00+00:00'], dtype='datetime64[ns, UTC]', freq='D')
```

Timezones Cont.

```
In [69]: ts = pd.date_range('11/2/2019 9:30am', periods=2, freq='D')
ts
```

```
Out[69]: DatetimeIndex(['2019-11-02 09:30:00', '2019-11-03 09:30:00'], dtype='datetime64[ns]', freq='D')
```

```
In [70]: # Set timezone using .localize()
ts_utc = ts.tz_localize('UTC')
ts_utc
```

```
Out[70]: DatetimeIndex(['2019-11-02 09:30:00+00:00', '2019-11-03 09:30:00+00:00'], dtype='datetime64[ns, UTC]', freq='D')
```

```
In [71]: # Change timezones using .tz_convert()
ts_utc.tz_convert('US/Eastern')
```

```
Out[71]: DatetimeIndex(['2019-11-02 05:30:00-04:00', '2019-11-03 04:30:00-05:00'], dtype='datetime64[ns, US/Eastern]', freq='D')
```

Timeseries in Python so far:

- `datetime .date .time .datetime .timedelta`
- format with `.strftime()`
- parse time with `pd.to_datetime()`
- `pandas Timestamp Timedelta DatetimeIndex`
- Indexing with `DatetimeIndex`
- Frequencies
- Timezones

Additional pandas functionality we won't discuss:

- `Period` and `PeriodIndex`
- `Panels`

Next: Operations on Time Series data

Questions re Datetimes in Python?