Elements Of Data Science - F2020

Week 11: Clustering and Recommendation Systems

11/30/2020

TODOs

- Readings:
  - PDSH: Chap 3.11 Working with Time Series
  - PDSH: Chap 5.06 Example: Predicting Bicycle Traffic
  - Recommended: DSFS: Chap 9: Getting Data
  - Recommended: DSFS: Chap 23: Databases and SQL
  - Optional: Python for Data Analysis: Chap 11: Time Series
  - Optional: PML: Chap 9: Embedding a Machine Learning Model into a Web Application

- HW3, Due Friday Dec 4th 11:59pm

- Answer and submit Quiz 11, **Sunday Dec 6th, 11:59pm ET**

Today

- Aggregating with groupby
- Clustering
- Recommendation Systems
- Time-Series Data?

Questions?

Aggregations Over Groups: `groupby`

In [2]:
```
# Example data with a categorical feature
df = pd.read_csv('../data/yellowcab_demo.csv',parse_dates=['pickup_datetime','dropoff_datetime'])
df.head(2)
```

Out[2]:

	pickup_datetime	dropoff_datetime	trip_distance	fare_amount	tip_amount	payment_type
0	2017-01-05 14:49:04	2017-01-05 14:53:53	0.89	5.5	1.26	Credit card
1	2017-01-15 01:07:22	2017-01-15 01:26:47	2.70	14.0	0.00	Cash

In [3]:
```
# We looked at taking aggregations over all rows or subsets of rows
df.trip_distance.mean()
```

Out[3]:
```
2.8800100000000004
```

In [4]:
```
# This dataset contains a categorical feature
df.payment_type.value_counts()
```

Out[4]:
```
Credit card    663
Cash           335
No charge        2
Name: payment_type, dtype: int64
```

Aggregations Over Groups: `groupby`

In [5]:
```python
# How can we take a mean per category?
df.groupby('payment_type').mean()
```

Out[5]:

	trip_distance	fare_amount	tip_amount
payment_type			
Cash	2.732209	11.856716	0.000000
Credit card	2.961870	12.761086	2.683322
No charge	0.500000	5.000000	0.000000

In [6]:
```python
# Specifying a single column for the aggregation
df.groupby('payment_type')['trip_distance'].mean()
```

Out[6]:
```
payment_type
Cash           2.732209
Credit card    2.961870
No charge      0.500000
Name: trip_distance, dtype: float64
```

Aggregations Over Groups: `groupby`

In [7]:

```
# Specifying multiple grouping columns, aggregation columns, and aggregations
df['tip_given'] = df.tip_amount > 0
df.groupby(['tip_given','payment_type'])[['trip_distance','fare_amount']].agg(['mean','std'])
```

Out[7]:

		trip_distance		fare_amount	
		mean	std	mean	std
tip_given	payment_type				
False	Cash	2.732209	4.123076	11.856716	11.634738
	Credit card	2.601519	3.285537	12.341772	11.325189
	No charge	0.500000	0.707107	5.000000	3.535534
True	Credit card	3.010616	3.455906	12.817808	10.241810

Questions?

Clustering

- Can we group our data based on the features alone?
- **Unsupervised:** There is no label/target
- Use similarity to group X into k clusters

- Many methods:
    - **k-Means**
    - **Heirarchical Agglomerative Clustering**
    - Spectral Clustering
    - DBScan
    - ...

Why do Clustering?

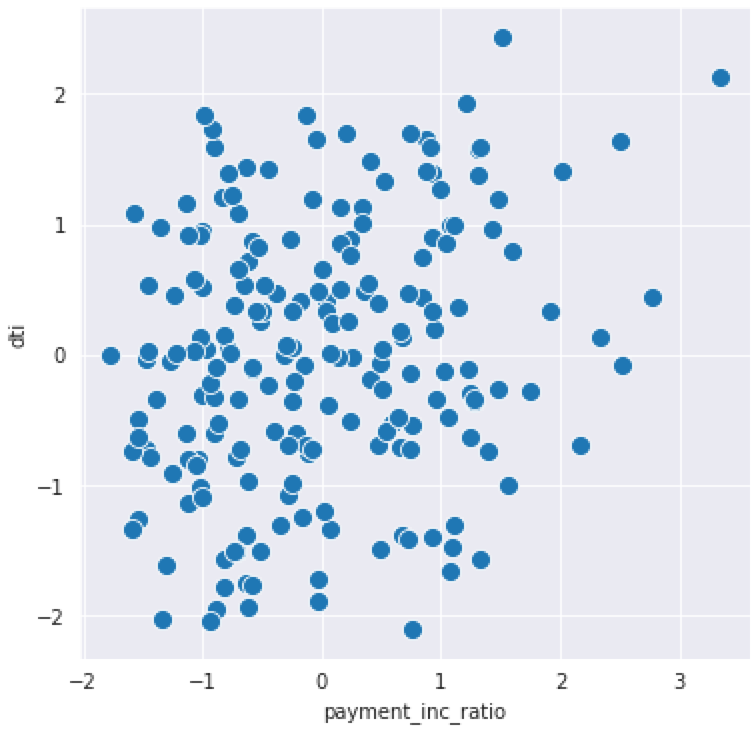
- Exploratory data analysis
- Group media: images, music, news articles,...
- Group people: social network
- Science applications: gene families, psychological groups,...
- Image segmentation: group pixels, regions, ...
- ...

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

Load Example Data

In [8]:
```
# loading and plotting the data
data = pd.read_csv('../data/loan200.csv')[['payment_inc_ratio','dti']]
from sklearn.preprocessing import StandardScaler
X = pd.DataFrame(StandardScaler().fit_transform(data),columns=data.columns)
fig,ax = plt.subplots(1,1,figsize=(6,6))
sns.scatterplot(x='payment_inc_ratio',y='dti',data=X,s=100);
```



KMeans in sklearn

In [9]:
```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=4, init='random') # default init=k-means++

c = km.fit_predict(X)
```

In [10]:
```
# cluster assignments of first 10 datapoints
c[:10]
```

Out[10]:
```
array([1, 2, 0, 3, 0, 0, 2, 3, 1, 1], dtype=int32)
```

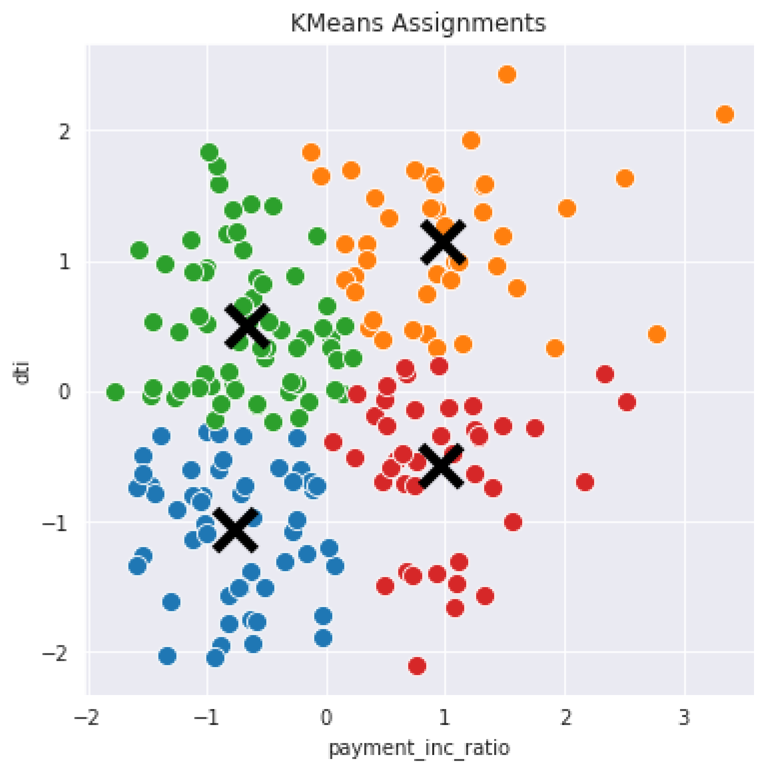
In [11]:
```
# cluster centers
km.cluster_centers_
```

Out[11]:
```
array([[-0.7733316 , -1.06665358],
       [ 0.98465708,  1.14651055],
       [-0.66442402,  0.50100968],
       [ 0.9623438 , -0.57562969]])
```

Plotting clusters and centers

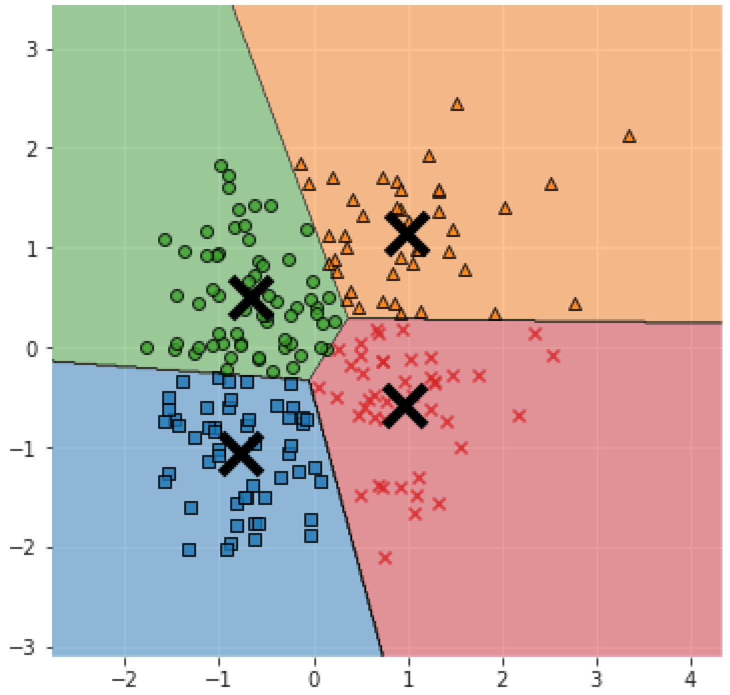
In [12]:

```
# plot data colored by cluster assignment
def plot_clusters(X,c,km=None,title=None,ax=None):
    if not ax:
        fig,ax = plt.subplots(1,1,figsize=(6,6))
    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=100,ax=ax);
    # plot cluster centers
    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,c,km,title="KMeans Assignments")
```



Plotting clusters and centers

In [13]:
```
fig,ax = plt.subplots(1,1,figsize=(6,6))
plot_decision_regions(X.values,km.predict(X),km,ax=ax,legend=None);
for m in km.cluster_centers_:
            ax.plot(m[0],m[1], marker='x',c='k', ms=20, mew=5);
```



K-Means: How good are the clusters?

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

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

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.

- animated visualization on next slide using Voronoi diagram

KMeans in Action

In [14]:
```python
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/

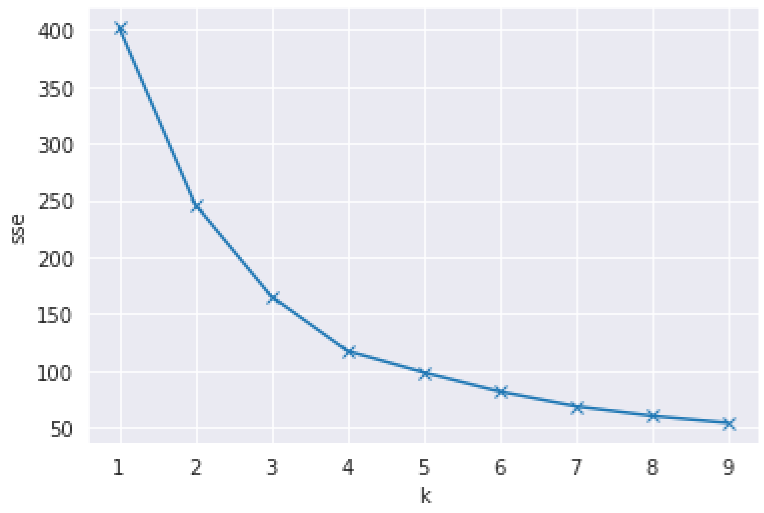
Things you need to define for KMeans

- 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, n_clusters?

- One way: use elbow in sum of squared errors (SSE)
- Stored in KMeans as `.inertia_`

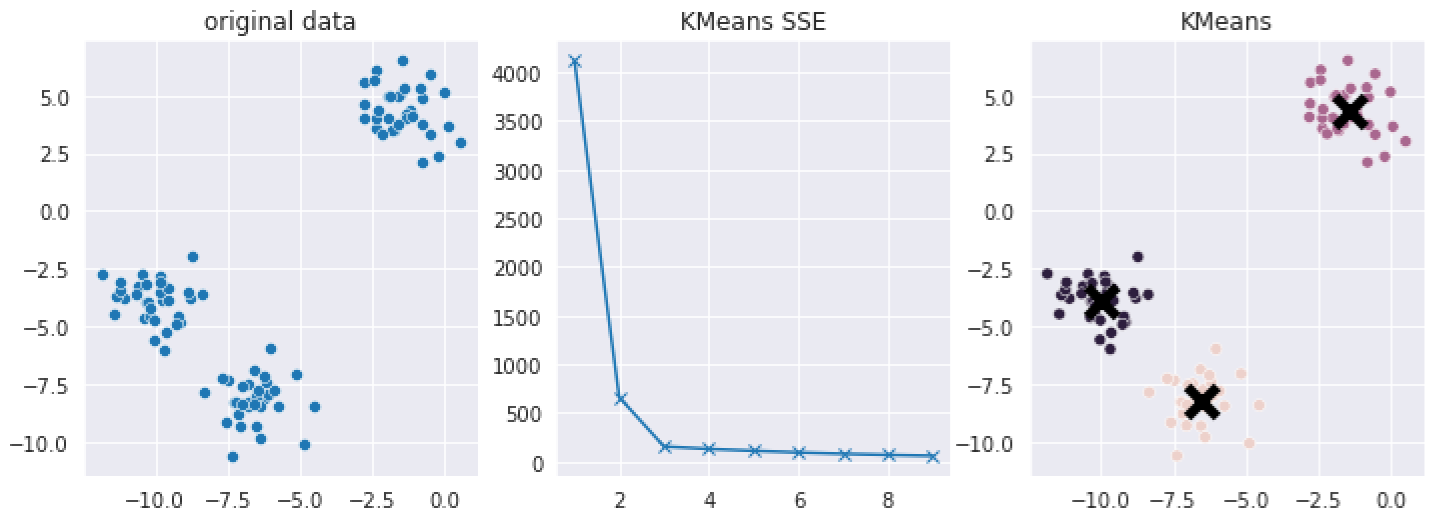
In [15]:
```python
sse = []
for i in range(1,10):
    sse.append(KMeans(n_clusters=i).fit(X).inertia_)

fig,ax=plt.subplots(1,1,figsize=(6,4))
ax.plot(range(1,10),sse,marker='x');
ax.set_xlabel('k');
ax.set_ylabel('sse');
```



- Question: What value k will minimize SSE?

KMeans: Another Example

In [16]:
```
from sklearn.datasets import make_blobs
X_blobs,y_blobs = make_blobs(random_state=1)

fig,ax = plt.subplots(1,3,figsize=(12,4))
sns.scatterplot(x=X_blobs[:,0],y=X_blobs[:,1],ax=ax[0]);
ax[0].set_title('original data');

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

km_blobs = KMeans(n_clusters=3, random_state=1)
c_blobs = km_blobs.fit_predict(X_blobs)
sns.scatterplot(x=X_blobs[:,0],y=X_blobs[:,1],hue=c_blobs,ax=ax[2],legend=False);
ax[2].set_title('KMeans');
for c in km_blobs.cluster_centers_:
    ax[2].plot(c[0],c[1],marker='x',c='k', ms=15, mew=5)
```



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

In [17]:
```python
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/

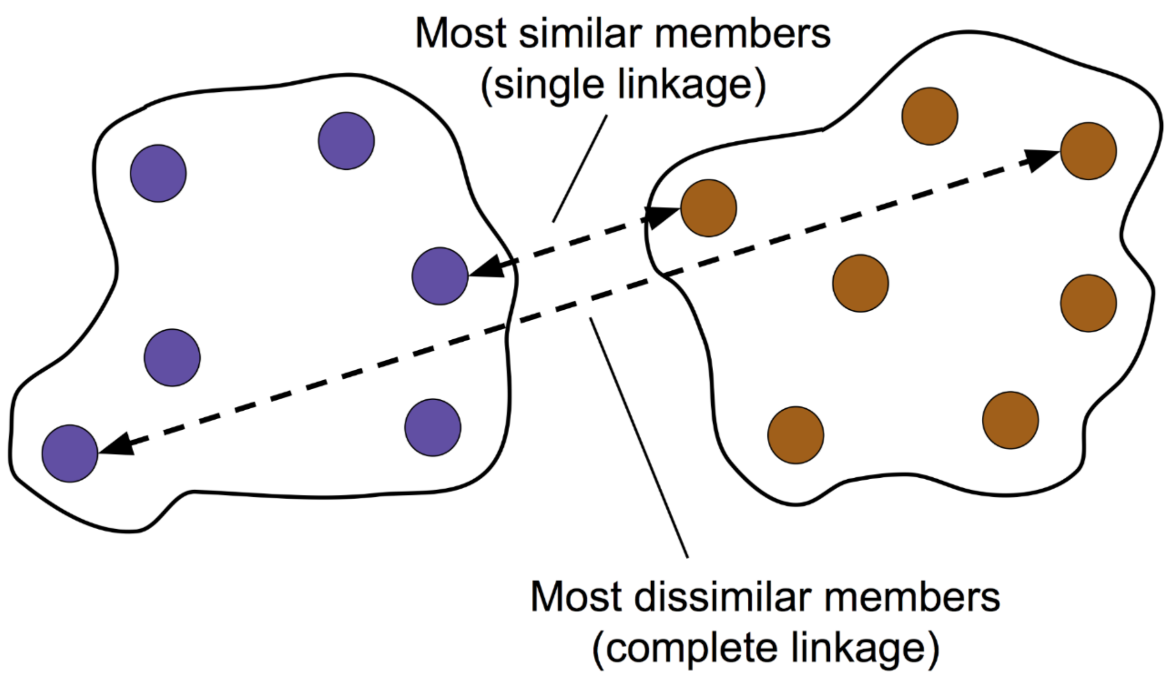
What is ”close”?

- 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 analyitically, sensitive to outliers
- **Manhattan** : $\sum_{i=1}^{n} \mid a_i - b_i \mid$
  - 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

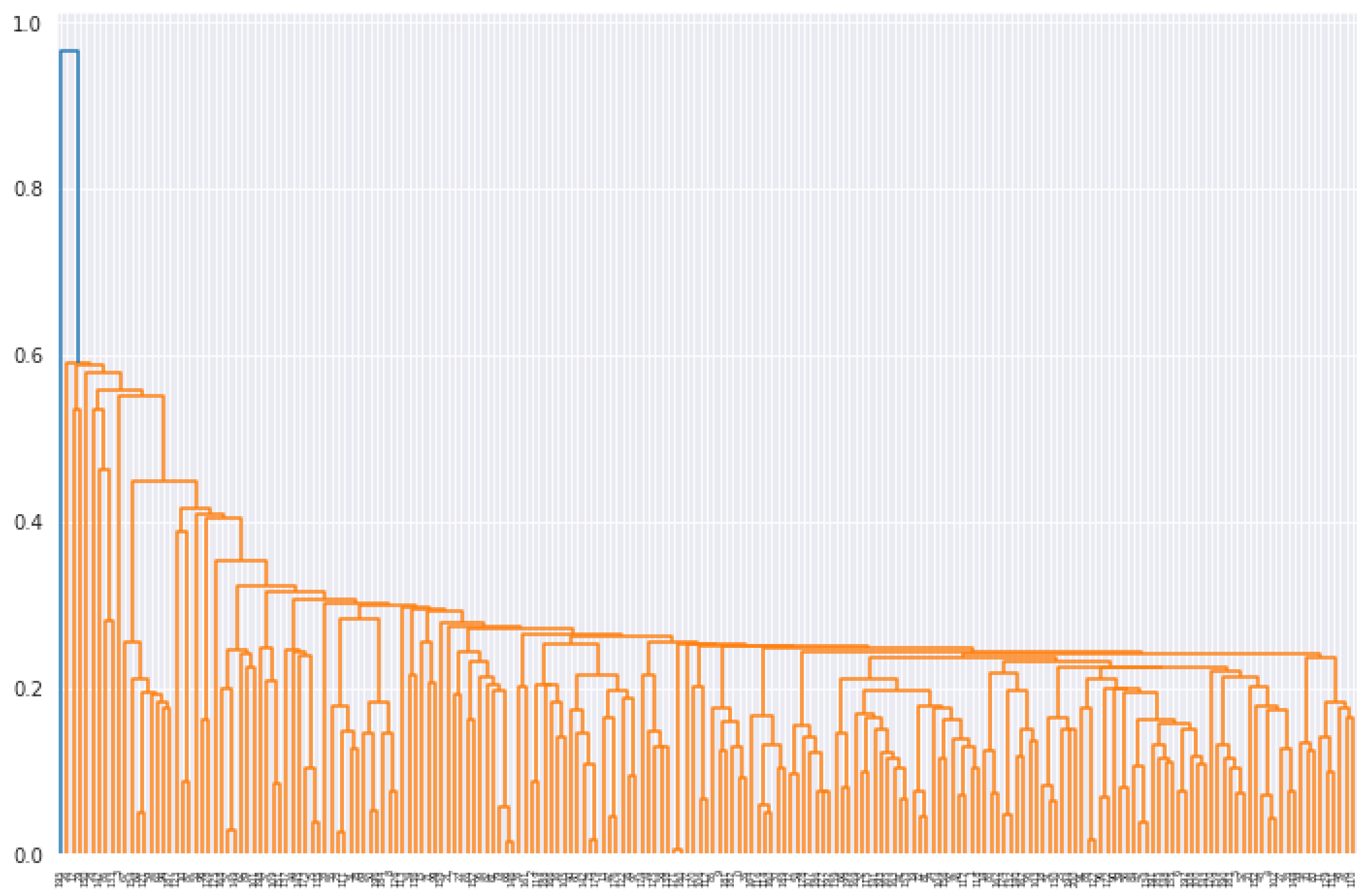
complete : greatest distance from item of one cluster to item of the other

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

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

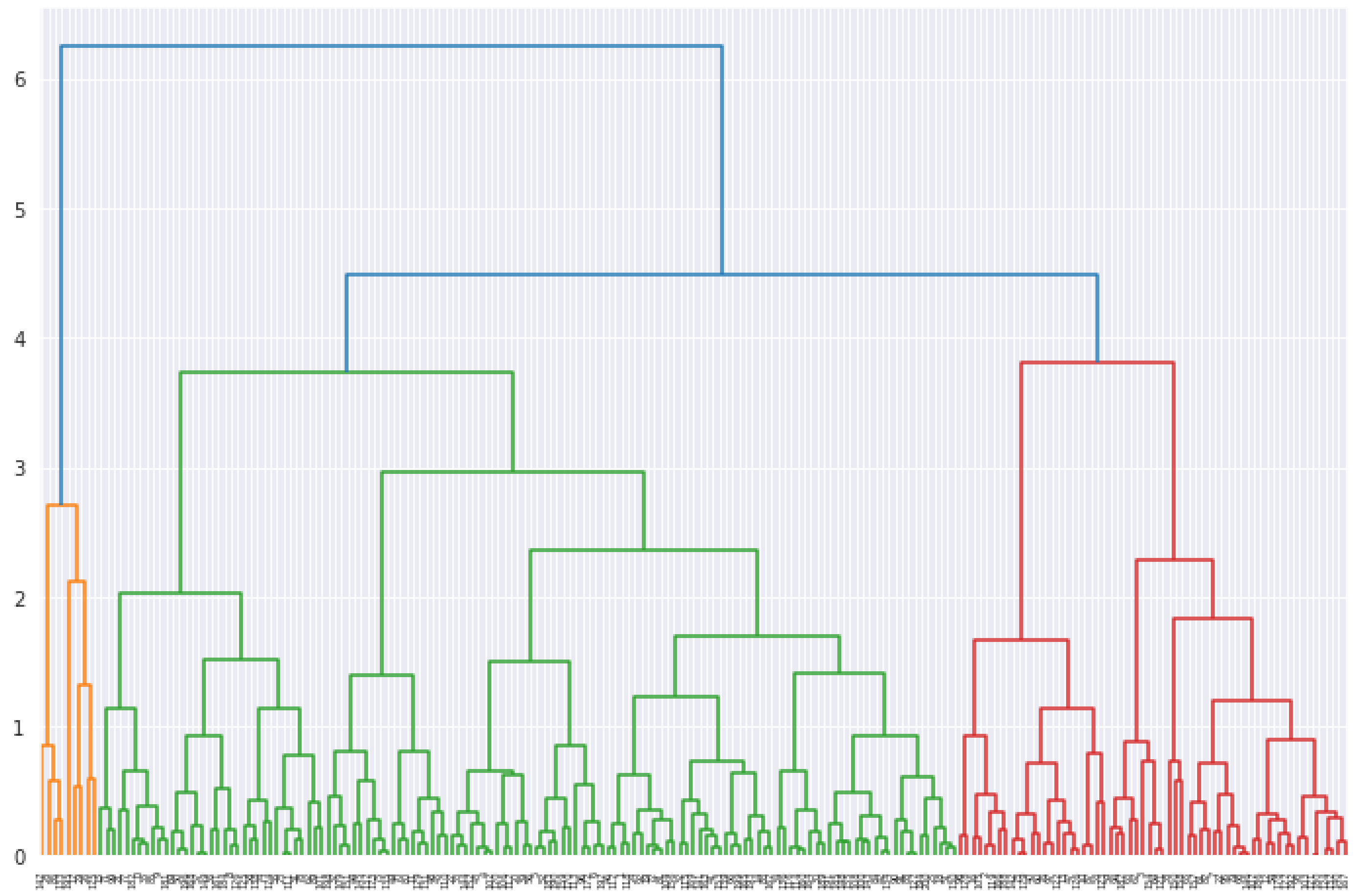
HAC and Dendrograms: Single Linkage

In [18]:
```
# nice helper function for creating a dendrogram
from scipy.cluster import hierarchy

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



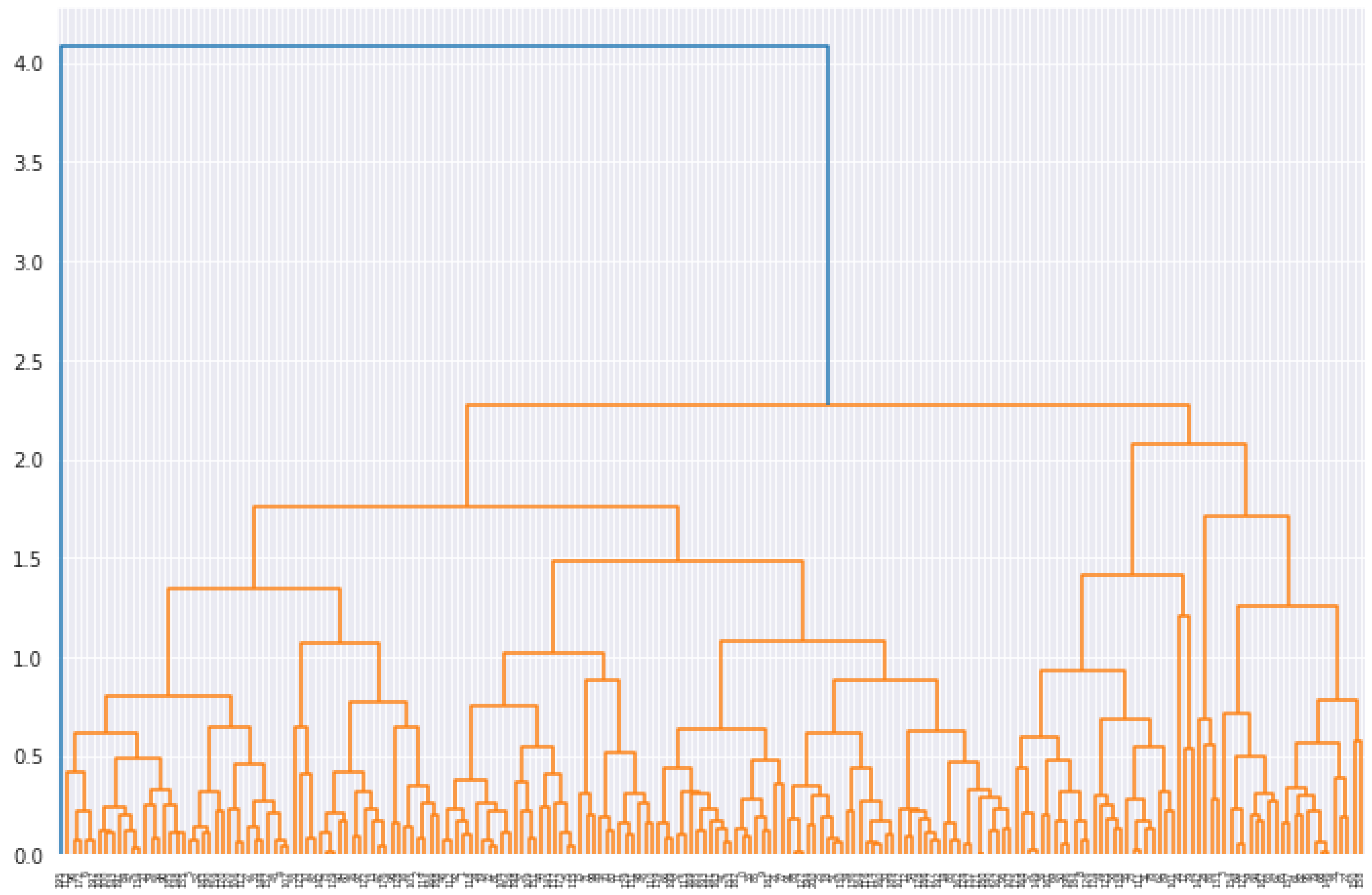
HAC and Dendrograms: Complete Linkage

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



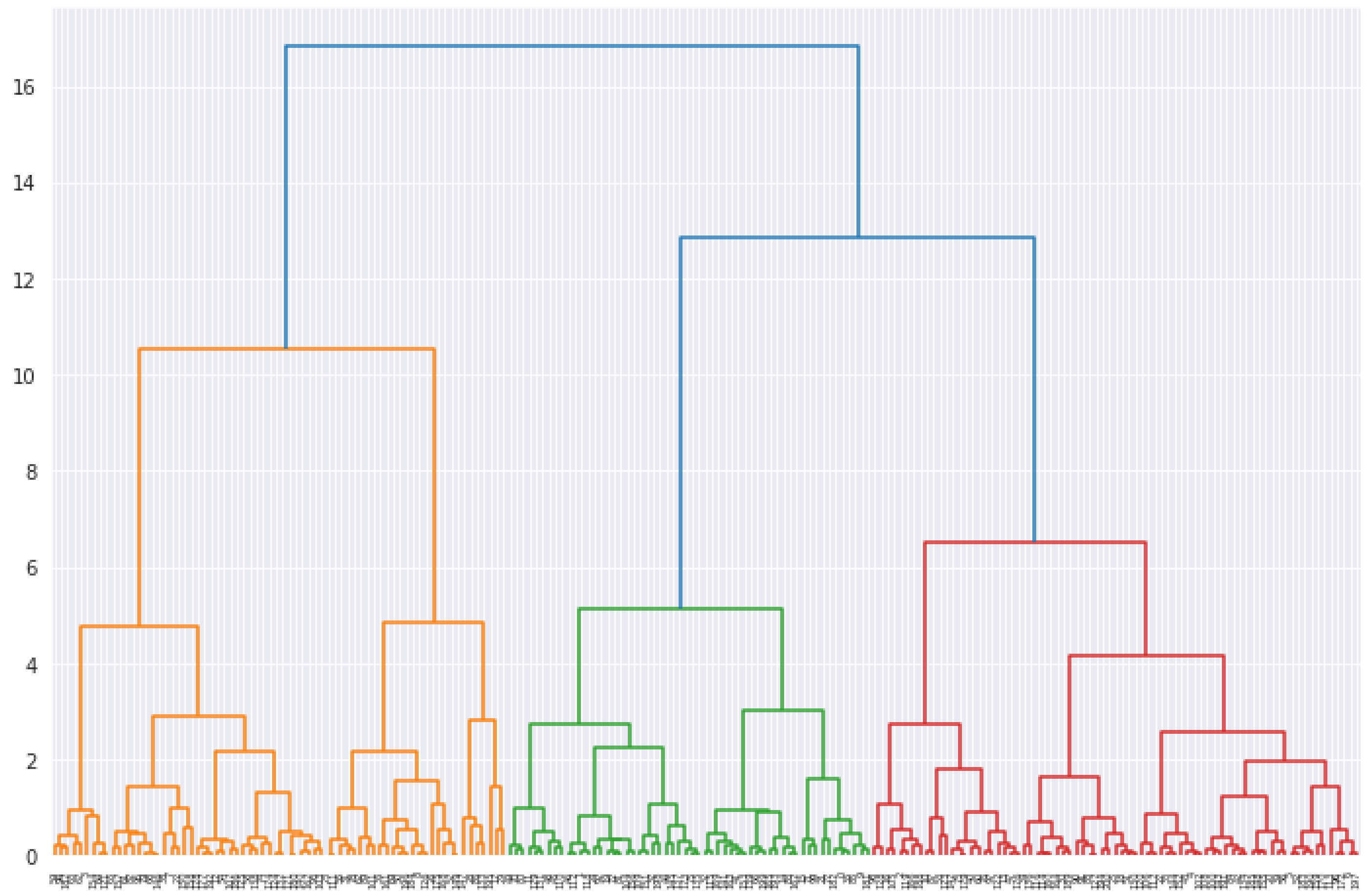
HAC and Dendrograms: Average Linkage

In [20]:
```
Z = hierarchy.linkage(X,'average')
fig = plt.figure(figsize=(12,8)); hierarchy.dendrogram(Z);
```



HAC and Dendrograms: Ward Linkage

In [21]:
```
Z = hierarchy.linkage(X,'ward')
fig = plt.figure(figsize=(12,8)); hierarchy.dendrogram(Z);
```

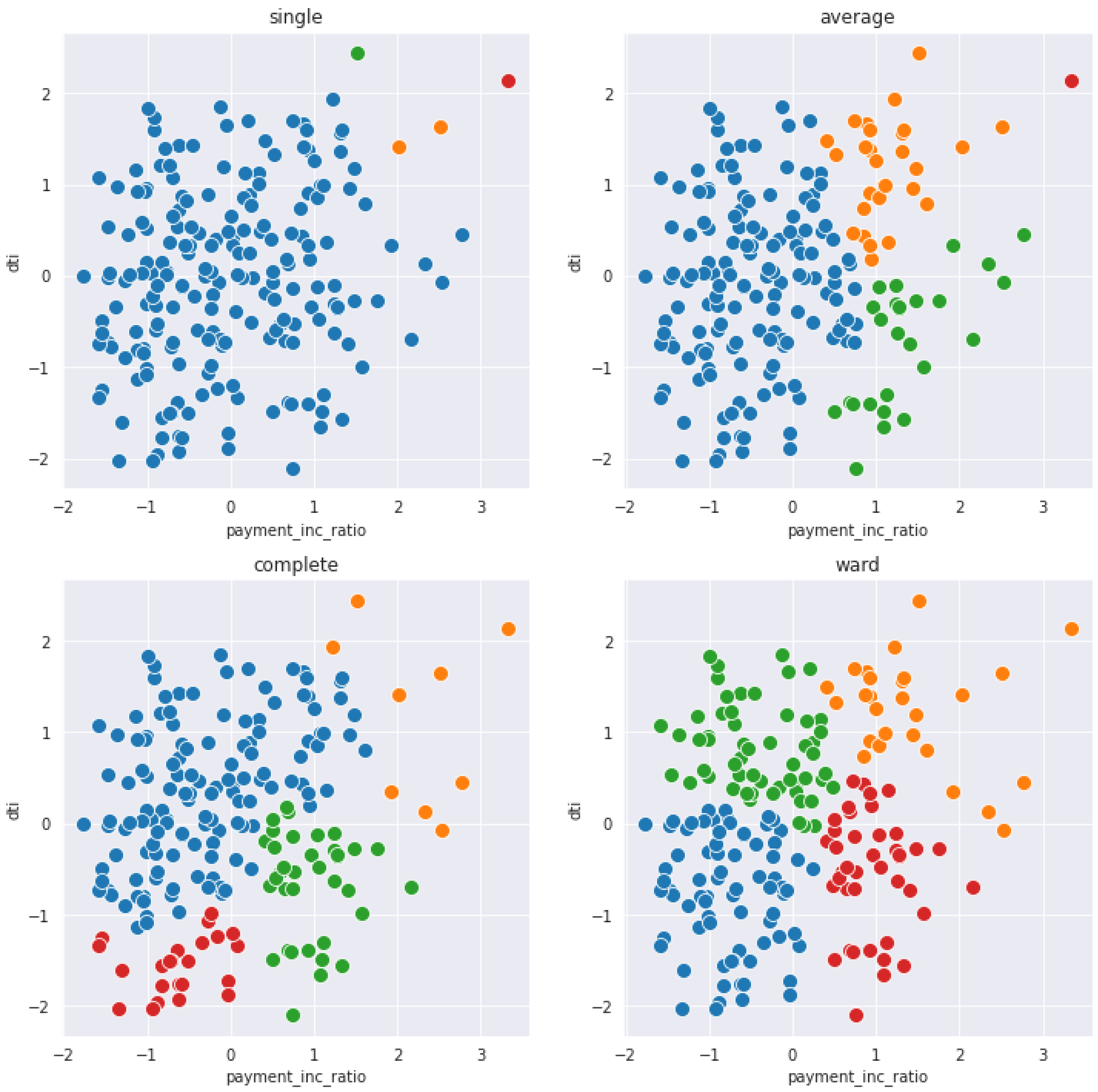


HAC in sklearn

In [22]:
```python
from sklearn.cluster import AgglomerativeClustering

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

# 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=4))
    assignments.append(models[-1].fit_predict(X))

# plot on the next slide
```

HAC in sklearn

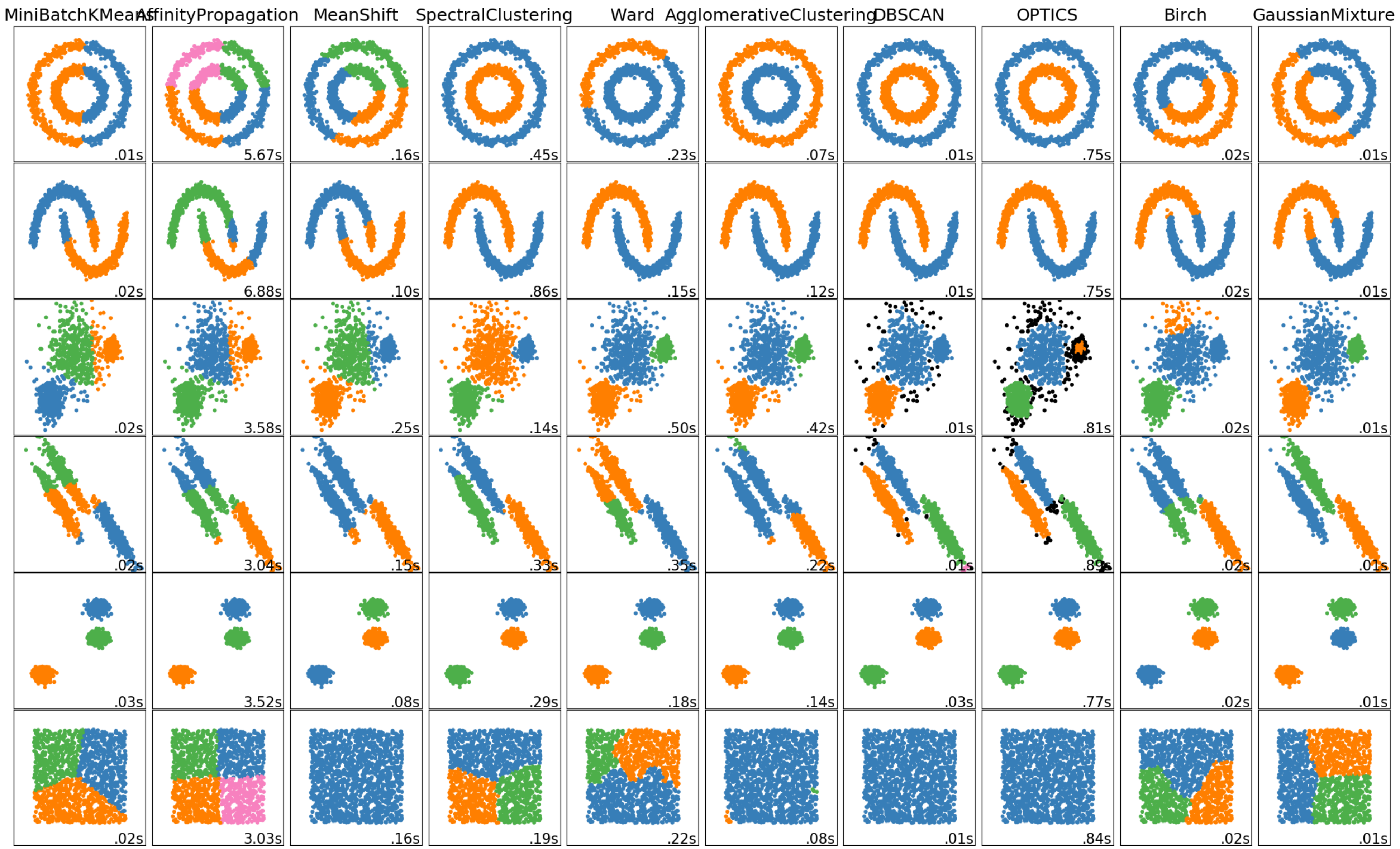
In [23]:
```
fig,ax = plt.subplots(2,2,figsize=(12,12))
axs = [ax[0][0],ax[0][1],ax[1][0],ax[1][1]]
for i in range(len(linkage)):
    plot_clusters(X,assignments[i],title=linkages[i],ax=axs[i])
```



HAC: Another Example

In [24]:
```
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_blobs))

fig,ax = plt.subplots(2,2,figsize=(12,8))
axs = [ax[0][0],ax[0][1],ax[1][0],ax[1][1]]
for i in range(len(linkage)):
    plot_clusters(pd.DataFrame(X_blobs),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?

- Inertia in k-means (weighted sse)
- 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?

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

In [25]:
```
df_house = pd.read_csv('../data/house_sales_subset.csv')
df_house = df_house.iloc[:10].loc[:,['SqFtTotLiving','SqFtLot','AdjSalePrice']]
X_house_norm = StandardScaler().fit_transform(df_house)
df_house_norm = pd.DataFrame(X_house_norm,columns=['SqFtTotLiving_norm','SqFtLot_norm','AdjSalePrice_norm'])
df_house_norm.head()
```

Out[25]:

	SqFtTotLiving_norm	SqFtLot_norm	AdjSalePrice_norm
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

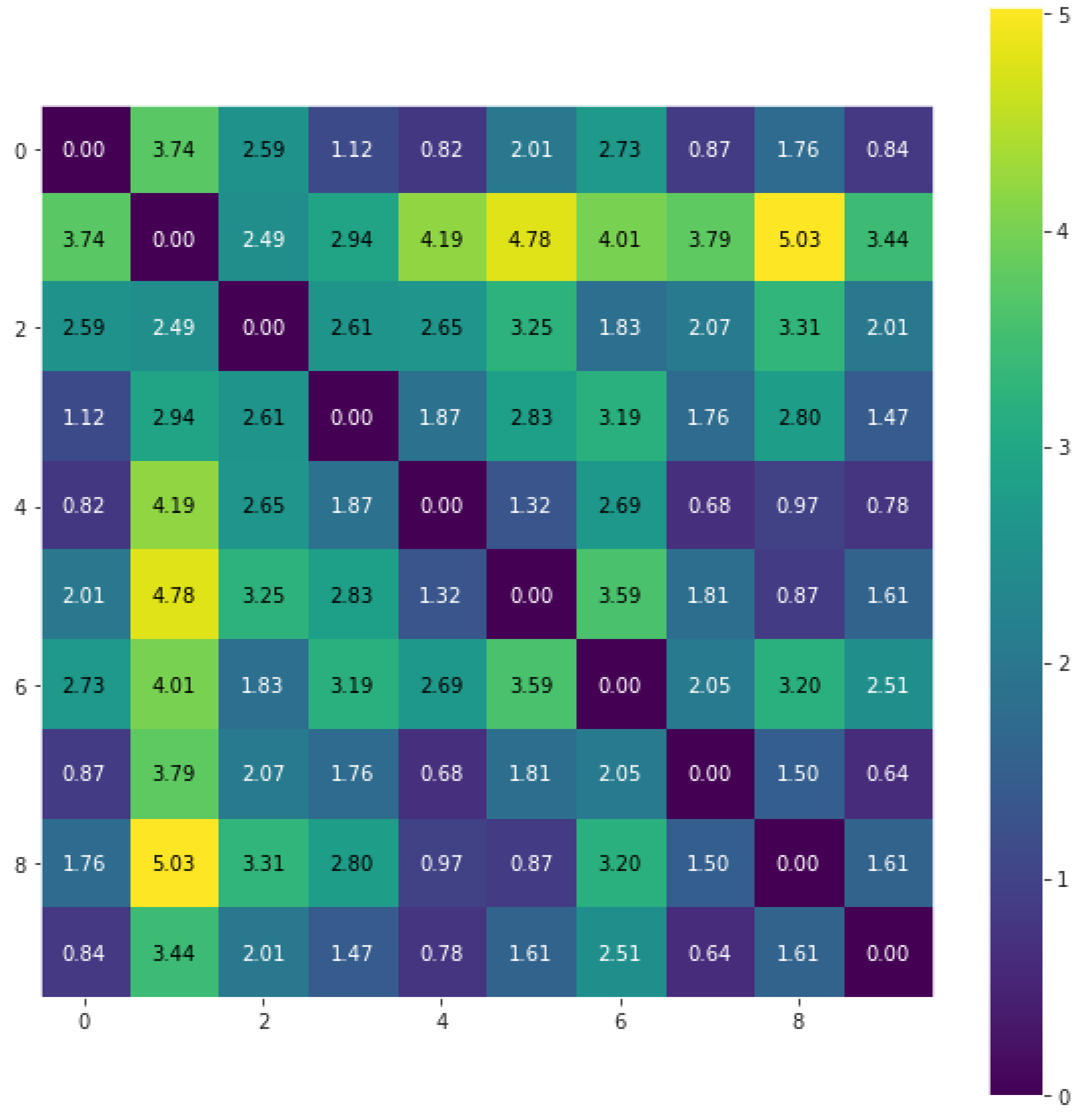
- to maximize similarity → minimize distance

In [26]:
```python
# using euclidean distance
from sklearn.metrics.pairwise import euclidean_distances

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

np.round(dists,2)
```

Out[26]:
```
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

In [27]:
```
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?

In [28]:
```
query_idx = 5
df_house.iloc[query_idx]
```

Out[28]:
```
SqFtTotLiving       930.0
SqFtLot            1012.0
AdjSalePrice     411781.0
Name: 5, dtype: float64
```

In [29]:
```
# Distances to house 5
[f'{x:0.1f}' for x in dists[query_idx]]
```

Out[29]:
```
['2.0', '4.8', '3.3', '2.8', '1.3', '0.0', '3.6', '1.8', '0.9', '1.6']
```

Query For Similarity Cont.

In [30]:
```
# find indexes of best scores (for distances, want ascending)
best_idxs_asc = np.argsort(dists[query_idx])
best_idxs_asc
```

Out[30]:
```
array([5, 8, 4, 9, 7, 0, 3, 2, 6, 1])
```

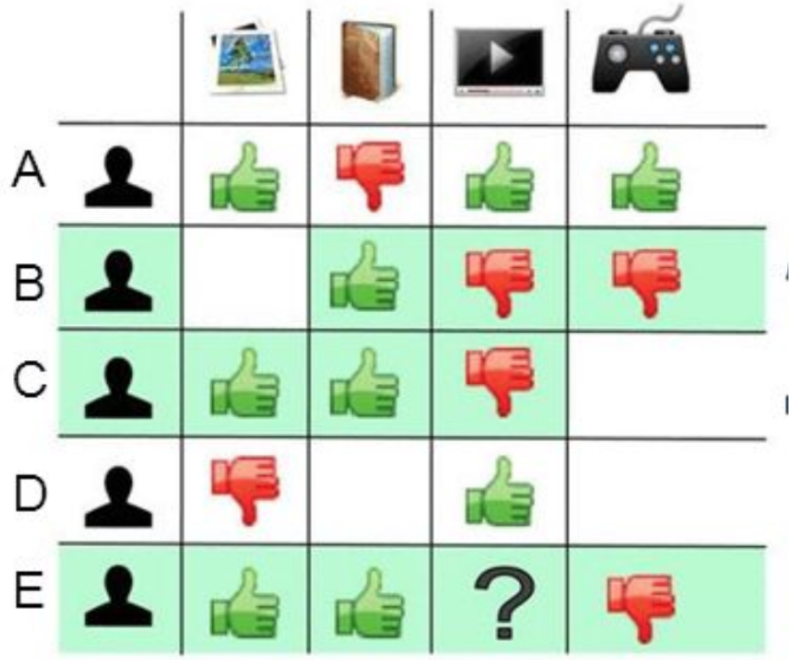
In [31]:
```
# the top 10 recommendations with distance
list(zip(best_idxs_asc,np.round(dists[query_idx][best_idxs_asc],2)))
```

Out[31]:
```
[(5, 0.0),
 (8, 0.87),
 (4, 1.32),
 (9, 1.61),
 (7, 1.81),
 (0, 2.01),
 (3, 2.83),
 (2, 3.25),
 (6, 3.59),
 (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



Example: User Interests

Can we recommend topics based on a users existing interests?

In [32]:
```python
# 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 [33]:
```python
# interests of user0
sorted(users_interests[0])
```

Out[33]:
```
['Big Data', 'Cassandra', 'HBase', 'Hadoop', 'Java', 'Spark', 'Storm']
```

All Unique Interests

```
In [34]: # 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[:5]

Out[34]: ['Big Data', 'C++', 'Cassandra', 'HBase', 'Hadoop']
```

Transform User Interest Matrix

In [35]:

```
# 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[35]:

```
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, 0])
```

In [36]:

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

Out[36]:

```
('Big Data', 'Cassandra', 'HBase', 'Hadoop', 'Java', 'Spark', 'Storm')
```

Calculate Similarity

In [37]:
```
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[37]:
```
array([1.        , 0.3380617 , 0.        , 0.        , 0.        ,
       0.15430335, 0.        , 0.        , 0.18898224, 0.56694671,
       0.        , 0.        , 0.        , 0.16903085, 0.        ])
```

In [38]:
```
# what users does user0 share interests with?
np.where(user_similarities[0])[0]
```

Out[38]:
```
array([ 0,  1,  5,  8,  9, 13])
```

Find Similar Users

In [39]:
```
# 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[39]:
```
[(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

In [40]:
```
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

In [41]:
```
# reminder: original interests
users_interests[0]
```

Out[41]:
```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
```

In [42]:
```
# top 5 new recommended interests
user_based_suggestions(0)[:5]
```

Out[42]:
```
[('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?

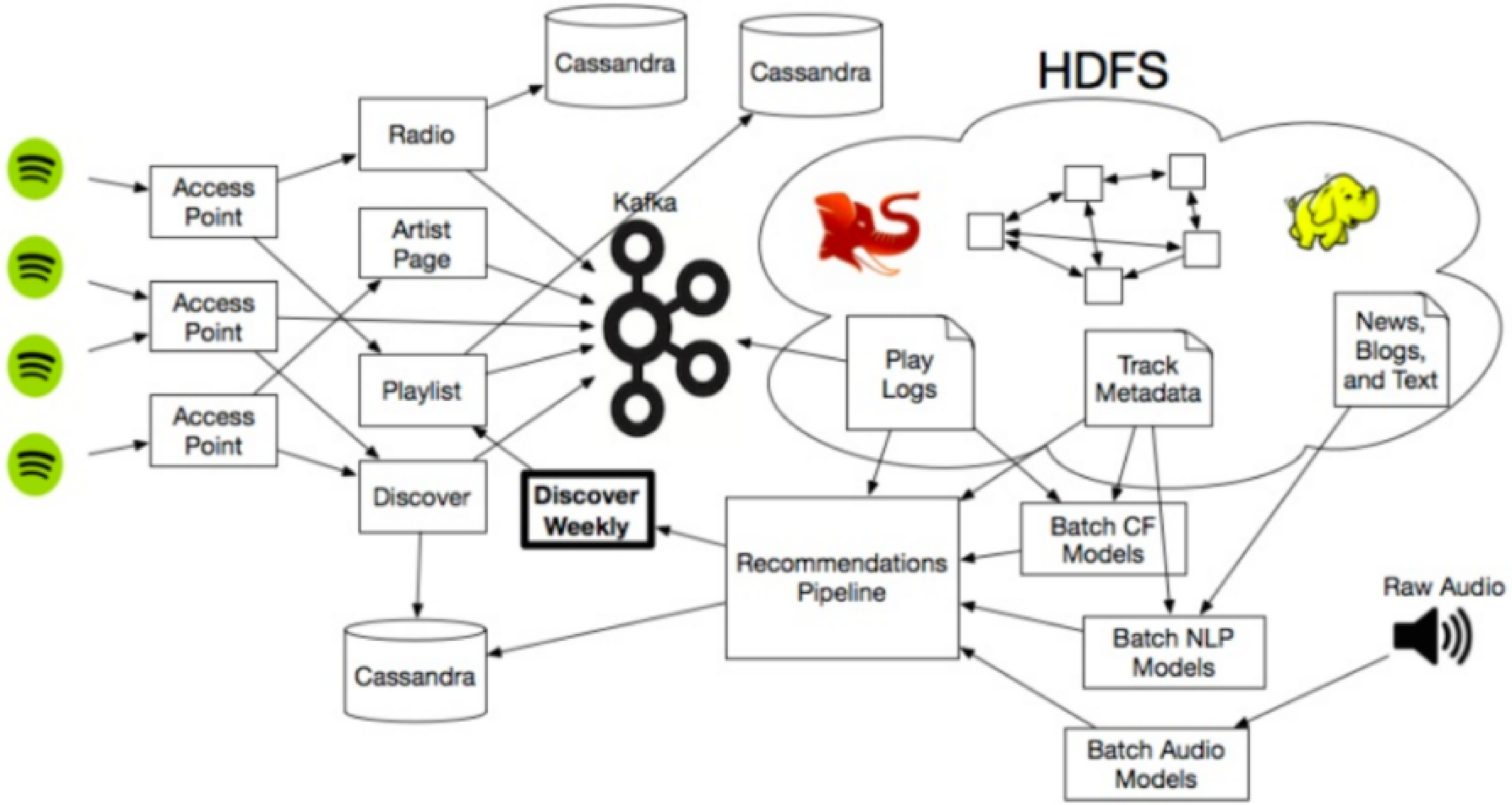
- **sparcity** : How to recommend movies no one's seen?

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?

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

In [43]:
```
from datetime import date

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

Out[43]:
```
datetime.date(2020, 12, 4)
```

In [44]:
```
today = date.today()
today
```

Out[44]:
```
datetime.date(2020, 11, 30)
```

In [45]:
```
today.year
```

Out[45]:
```
2020
```

datetime.time

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

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

Out[46]:
```
datetime.time(12, 0)
```

In [47]:
```
noon.hour
```

Out[47]:
```
12
```

datetime.datetime

In [48]:
```python
from datetime import datetime

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

Out[48]:
```
datetime.datetime(2020, 11, 30, 19, 10)
```

In [49]:
```python
now = datetime.now()
now
```

Out[49]:
```
datetime.datetime(2020, 11, 30, 17, 53, 36, 601476)
```

`datetime.timedelta`

In [50]:
```
diff = datetime(2020,11,30,1) - datetime(2020,11,29,0)
diff
```

Out[50]:
```
datetime.timedelta(days=1, seconds=3600)
```

In [51]:
```
diff.total_seconds()
```

Out[51]:
```
90000.0
```

In [52]:
```
from datetime import timedelta

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

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

Out[52]:
```
datetime.date(2020, 12, 2)
```

Printing Datetimes: `strftime()`

In [53]:
```
print(now)
```

```
2020-11-30 17:53:36.601476
```

In [54]:
```
now.strftime('%a %h %d, %Y %I:%M %p')
```

Out[54]:
```
'Mon Nov 30, 2020 05:53 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()`

In [55]:
```
pd.to_datetime('11/22/2019 2:36pm')
```
Out[55]:
```
Timestamp('2019-11-22 14:36:00')
```

In [56]:
```
dt_index = pd.to_datetime([datetime(2020, 11, 26),
                           '27th of November, 2020',
                           '2020-Nov-28',
                           '11-29-2030',
                           '20201130',
                           None
                          ])
dt_index
```
Out[56]:
```
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

In [57]:
```
dt_index[0]
```

Out[57]:
```
Timestamp('2020-11-26 00:00:00')
```

DateIndex Indexing/Selecting/Slicing

In [58]:
```
s = pd.Series([101,102,103],
              index=pd.to_datetime(['20191201','20200101','20200201']))
s
```

Out[58]:
```
2019-12-01    101
2020-01-01    102
2020-02-01    103
dtype: int64
```

In [59]:
```
# can index normally using iloc
s.iloc[0:2]
```

Out[59]:
```
2019-12-01    101
2020-01-01    102
dtype: int64
```

DateIndex Indexing/Selecting/Slicing Cont.

In [60]:
```
# only rows from the year 2020
s.loc['2020']
```

Out[60]:
```
2020-01-01    102
2020-02-01    103
dtype: int64
```

In [61]:
```
# only rows from January 2020
s.loc['2020-01']
```

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

In [62]:
```
# only rows between Jan 1st 2019 and Jan 1st 2020, inclusive
s.loc['01/01/2019':'01/01/2020']
```

Out[62]:
```
2019-12-01    101
2020-01-01    102
dtype: int64
```

In [63]:
```
# can use the indexing shortcut
s['2020']
```

Out[63]:
```
2020-01-01    102
2020-02-01    103
dtype: int64
```

Datetimes in DataFrames

In [64]:

```
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[64]:

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

In [65]:

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

Out[65]:

	col2	col3
col1		
2020-12-01	101	A

Timestamp Index: Setting Frequency

In [66]:
```
s = pd.Series([101,103],index=pd.to_datetime(['20201201','20201203']))
s
```

Out[66]:
```
2020-12-01    101
2020-12-03    103
dtype: int64
```

In [67]:
```
# Use resample() and asfreq() to set frequency
s.resample('D').asfreq()
```

Out[67]:
```
2020-12-01    101.0
2020-12-02      NaN
2020-12-03    103.0
Freq: D, dtype: float64
```

In [68]:
```
pd.to_datetime(['20191201','20191203'])
```

Out[68]:
```
DatetimeIndex(['2019-12-01', '2019-12-03'], dtype='datetime64[ns]', freq=None)
```

In [69]:
```
# 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[69]:
```
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
SM   semi-month end frequency (15th and end of month)
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

```
In [70]: import pytz

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

Out[70]: ['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.

```
In [71]: ts = pd.date_range('11/2/2019 9:30am',periods=2,freq='D')
         ts
```

```
Out[71]: DatetimeIndex(['2019-11-02 09:30:00', '2019-11-03 09:30:00'], dtype='datetime64[ns]', freq='D')
```

```
In [72]: # Set timezone using .localize()
         ts_utc = ts.tz_localize('UTC')
         ts_utc
```

```
Out[72]: DatetimeIndex(['2019-11-02 09:30:00+00:00', '2019-11-03 09:30:00+00:00'], dtype='datetime64[ns, UTC]', freq='D')
```

```
In [73]: # Change timezones using .tz_convert()
         ts_utc.tz_convert('US/Eastern')
```

```
Out[73]: 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

Shifting

- Moving data backward or forward in time (lagging/leading)
- Ex: calculate percent change

In [74]:
```
ts = pd.Series([1,2,8],
               index=pd.date_range('1/1/2019',periods=3,freq='M'))
ts
```

Out[74]:
```
2019-01-31    1
2019-02-28    2
2019-03-31    8
Freq: M, dtype: int64
```

In [75]:
```
ts.shift(1) # old value
```

Out[75]:
```
2019-01-31    NaN
2019-02-28    1.0
2019-03-31    2.0
Freq: M, dtype: float64
```

Shifting

- **percent change :**
  - (new_value - old_value) / old_value
  - (new_value / old_value) - 1

In [76]:
```
#  multiply by 100 to turn into a percent
((ts / ts.shift(1)) - 1) * 100
```

Out[76]:
```
2019-01-31      NaN
2019-02-28    100.0
2019-03-31    300.0
Freq: M, dtype: float64
```

Example Dataset: Twitter Stock

In [77]:
```
# first run: conda install pandas-datareader
#from pandas_datareader import data
#df_twtr = data.DataReader('TWTR', start='2015', end='2021', data_source='yahoo')
df_twtr = pd.read_csv('../data/twtr_2015-2020.csv',parse_dates=['Date'],index_col='Date')
df_twtr.head(3)
```

Out[77]:

	High	Low	Open	Close	Volume	Adj Close
Date						
2015-01-02	36.740002	35.540001	36.230000	36.560001	12062500	36.560001
2015-01-05	37.110001	35.639999	36.259998	36.380001	15062700	36.380001
2015-01-06	39.450001	36.040001	36.270000	38.759998	33050800	38.759998

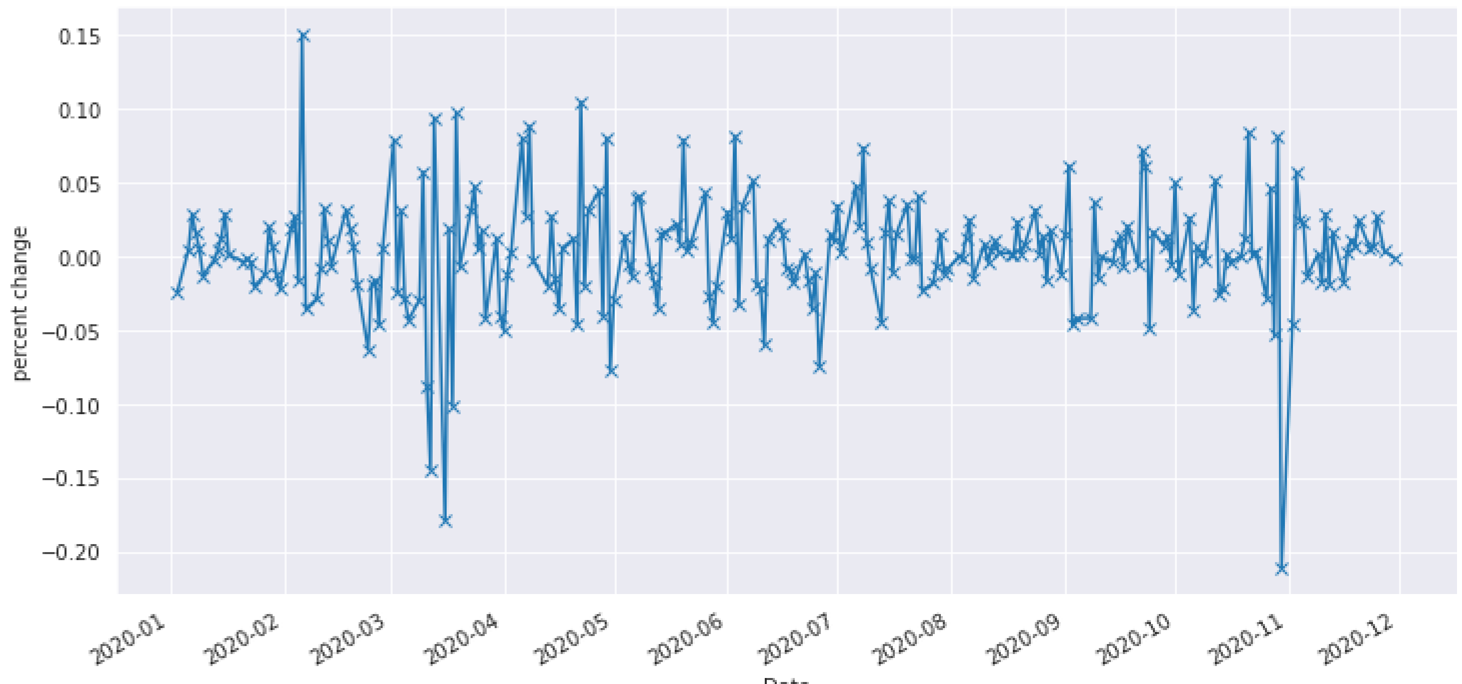
In [78]:
```
fig,ax = plt.subplots(1,1,figsize=(12,6))
df_twtr[['Close']].plot(ax=ax);
ax.set_ylabel('stock price');
```

Example Dataset: Twitter Stock

- Calculate Percent Change of Close

```
In [79]: (df_twtr.Close / df_twtr.Close.shift(1) - 1).tail(5)

Out[79]: Date
         2020-11-23    0.005819
         2020-11-24    0.006453
         2020-11-25    0.026531
         2020-11-27    0.003446
         2020-11-30   -0.001717
         Name: Close, dtype: float64
```

```
In [80]: # plot percent change of close in 2020
         fig,ax = plt.subplots(1,1,figsize=(12,6))
         close_2020 = df_twtr.loc['2020','Close']
         ((close_2020 / close_2020.shift(1)) - 1 ).plot(marker='x',ax=ax);
         ax.set_ylabel('percent change');
```



Resampling

- Convert from one frequency to another
- Downsampling
  - from higher to lower (day to month)
  - need to aggregate
- Upsampling
  - from lower to higher (month to day)
  - need to fill missing
- Can also be used to set frequency from None

Resampling: Initialize Frequency

In [81]:
```
df_twtr.index
```

Out[81]:
```
DatetimeIndex(['2015-01-02', '2015-01-05', '2015-01-06', '2015-01-07',
               '2015-01-08', '2015-01-09', '2015-01-12', '2015-01-13',
               '2015-01-14', '2015-01-15',
               ...
               '2020-11-16', '2020-11-17', '2020-11-18', '2020-11-19',
               '2020-11-20', '2020-11-23', '2020-11-24', '2020-11-25',
               '2020-11-27', '2020-11-30'],
              dtype='datetime64[ns]', name='Date', length=1489, freq=None)
```

In [82]:
```
df_twtr_B = df_twtr.resample('B').asfreq() # set frequncy to business day
df_twtr_B.index
```

Out[82]:
```
DatetimeIndex(['2015-01-02', '2015-01-05', '2015-01-06', '2015-01-07',
               '2015-01-08', '2015-01-09', '2015-01-12', '2015-01-13',
               '2015-01-14', '2015-01-15',
               ...
               '2020-11-17', '2020-11-18', '2020-11-19', '2020-11-20',
               '2020-11-23', '2020-11-24', '2020-11-25', '2020-11-26',
               '2020-11-27', '2020-11-30'],
              dtype='datetime64[ns]', name='Date', length=1542, freq='B')
```

Resampling: Downsampling

- Go from shorter to longer
- Need to aggregate (like groupby)
- Example: Downsampling from business day to business quarter

In [83]:
```
df_twtr_BQ = df_twtr.resample('BQ')
df_twtr_BQ
```

Out[83]:
```
<pandas.core.resample.DatetimeIndexResampler object at 0x7fef902311f0>
```

In [84]:
```
str(df_twtr_BQ)
```

Out[84]:
```
'DatetimeIndexResampler [freq=<BusinessQuarterEnd: startingMonth=12>, axis=0, closed=right, label=right, convention=start, orig
in=start_day]'
```

In [85]:
```
df_twtr_BQ.mean().head(3)
```

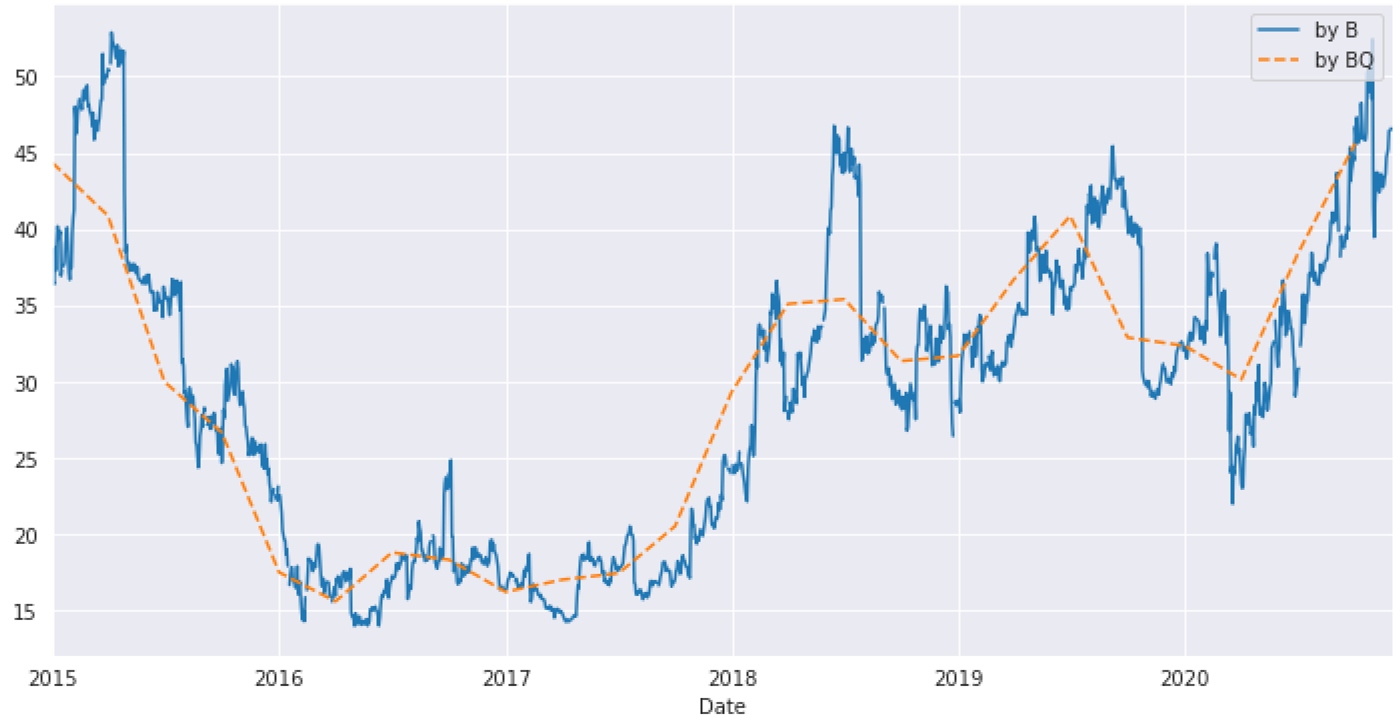
Out[85]:

	High	Low	Open	Close	Volume	Adj Close
Date						
2015-03-31	45.080328	43.552459	44.228688	44.335574	2.084619e+07	44.335574
2015-06-30	41.634921	40.385079	41.173492	40.874603	2.232030e+07	40.874603
2015-09-30	30.638281	29.420625	30.047812	30.000625	2.031210e+07	30.000625

Resampling: Downsampling

In [86]:

```
fig,ax = plt.subplots(1,1,figsize=(12,6))
df_twtr_B.Close.plot(style='-', label='by B',ax=ax)
df_twtr_BQ.Close.mean().plot(style='--',label='by BQ',ax=ax)
plt.legend(loc='upper right');
```



Resampling: Upsampling

- Go from longer to shorter
- Need to decide how to handle missing values
- Example: Upsample from business day to hour

In [87]:
```
df_twtr_B.Close.resample('H').asfreq().head(10)
```

Out[87]:
```
Date
2015-01-02 00:00:00    36.560001
2015-01-02 01:00:00          NaN
2015-01-02 02:00:00          NaN
2015-01-02 03:00:00          NaN
2015-01-02 04:00:00          NaN
2015-01-02 05:00:00          NaN
2015-01-02 06:00:00          NaN
2015-01-02 07:00:00          NaN
2015-01-02 08:00:00          NaN
2015-01-02 09:00:00          NaN
Freq: H, Name: Close, dtype: float64
```

Resampling: Upsampling

- `ffill()` : Forward Fill

```
In [88]: df_twtr_B.Close.resample('H').ffill().head(3)
```

```
Out[88]: Date
2015-01-02 00:00:00    36.560001
2015-01-02 01:00:00    36.560001
2015-01-02 02:00:00    36.560001
Freq: H, Name: Close, dtype: float64
```

- `bfill()` : Backward Fill

```
In [89]: df_twtr_B.Close.resample('H').bfill().head(3)
```

```
Out[89]: Date
2015-01-02 00:00:00    36.560001
2015-01-02 01:00:00    36.380001
2015-01-02 02:00:00    36.380001
Freq: H, Name: Close, dtype: float64
```

Moving Windows

- Apply function on a fixed window moving accross time
- Method of smoothing out the data
- **center** : place values at center of window

In [90]:
```
df_twtr_B.Close['2020-11-02':'2020-11-06']
```

Out[90]:
```
Date
2020-11-02    39.470001
2020-11-03    41.730000
2020-11-04    42.759998
2020-11-05    43.709999
2020-11-06    43.119999
Freq: B, Name: Close, dtype: float64
```

In [91]:
```
rolling = df_twtr_B.Close.rolling(5, center=True)
rolling
```

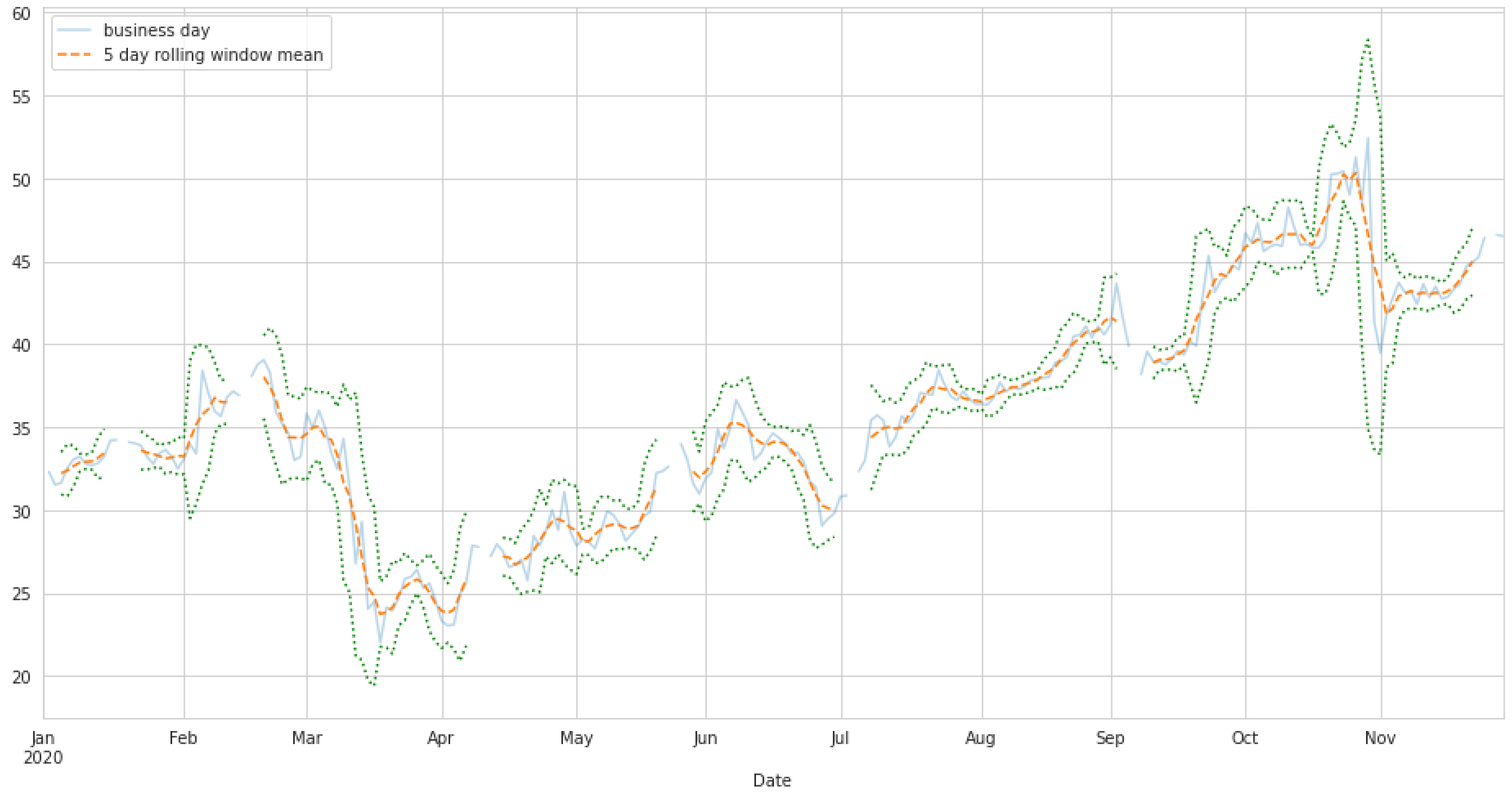
Out[91]:
```
Rolling [window=5,center=True,axis=0]
```

In [92]:
```
rolling.mean()['2020-11-02':'2020-11-06']
```

Out[92]:
```
Date
2020-11-02    43.550000
2020-11-03    41.806000
2020-11-04    42.157999
2020-11-05    42.901999
2020-11-06    43.037999
Freq: B, Name: Close, dtype: float64
```

Moving Windows

```
In [93]: sns.set_style("whitegrid")
         fig,ax = plt.subplots(1,1,figsize=(16,8));
         df_twtr_B['2020'].Close.plot(style='-',alpha=0.3,label='business day');
         rolling.mean()['2020'].plot(style='--',label='5 day rolling window mean');
         (rolling.mean()['2020'] + 2*rolling.std()['2020']).plot(style=':',c='g',label='_nolegend_');
         (rolling.mean()['2020'] - 2*rolling.std()['2020']).plot(style=':',c='g',label='_nolegend_');
         ax.legend();
```



Timeseries Operations Review

- Shifting
- Resampling
  - Downsampling
  - Upsampling
- Moving/Rolling Windows

Questions?