# Exercise: Clustering of artists

===

In this exercise, we will use a real world music dataset from Last.fm (http://last.fm) to experience with Unsupervised Clustering methods.

Note: The play data (and user/artist matrix) comes from the Last.fm 1K Users dataset (http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-1K.html), while the tags come from the Last.fm Music Tags dataset (http://musicmachinery.com/2010/11/10/lastfm-artisttags2007/). You won't have to interact with these datasets directly, because we've already preprocessed them for you.

## Files

Data files for this assignment can be found at: `/volumes/data/lastfm/`

The folder includes the following files:

- **artists-tags.txt**, User-defined tags for top artists
- **userart-mat-training.csv**, Training data containing a matrix mapping artist-id to users who have played songs by the artists
- **userart-mat-test.csv**, Test data containing a matrix mapping artist-id to users who have played songs by the artists
- **train_model_data.csv**, Aggregate statsitstics and features about songs we'll use to train regression models.
- **validation_model_data.csv**, Similar statistics computed on a hold-out set of users and songs that we'll use to validate our regression models.

We will explain the datasets and how they need to used later.

# Part 0: Preliminaries

## Exercise 0

Read in the file **artists-tags.txt** and store the contents in a Pandas DataFrame. The file format for this file is `artist-id|artist-name|tag|count`. The fields mean the following:

1. artist-id : a unique id for an artist (Formatted as a MusicBrainz Identifier (https://musicbrainz.org/doc/MusicBrainz_Identifier))
2. artist-name: name of the artist
3. tag: user-defined tag for the artist
4. count: number of times the tag was applied

Similarly, read in the file **userart-mat-training.csv** . The file format for this file is `artist-id, user1, user2, .... user1000`. i.e. There are 846 such columns in this file and each column has a value 1 if the particular user played a song from this artist.

In [1]:
```
import pandas as pd

DATA_PATH = "/volumes/data/lastfm"

def parse_artists_tags(filename):
    df = pd.read_csv(filename, sep="|", names=["ArtistID", "ArtistName", "Tag",
"Count"])
    return df

def parse_user_artists_matrix(filename):
    df = pd.read_csv(filename)
    return df

artists_tags = parse_artists_tags(DATA_PATH + "/artists-tags.txt")
user_art_mat = parse_user_artists_matrix(DATA_PATH + "/userart-mat-training.csv"
)

print "Number of tags %d" % len(artists_tags) # Change this line. Should be 9528
03
print "Number of artists %d" % len(user_art_mat) # Change this line. Should be 1
7119
```

```
Number of tags 952810
Number of artists 17119
```

# Part 1: Finding genres by clustering

The first task we will look at is how to discover artist genres by only looking at data from plays on Last.fm. One of the ways to do this is to use clustering. To evaluate how well our clustering algorithm performs we will use the user-generated tags and compare those to our clustering results.

## 1.1 Data pre-processing

Last.fm allows users to associate tags with every artist (See the top tags (http://www.last.fm/charts/toptags) for a live example). However as there are a number of tags associated with every artists, in the first step we will pre-process the data and get the most popular tag for an artist.

#### Exercise 1

**a**. For every artist in **artists_tags** calculate the most frequently used tag.

First, we have look at the dataframe:

In [2]:
```
artists_tags.head()
```

Out[2]:

| | ArtistID | ArtistName | Tag | Count |
|---|---|---|---|---|
| **0** | 000077f7-26b1-4710-80cc-f6beddbdd157 | Ryan Adams and The Cardinals | I love you baby can I have some more | 1 |
| **1** | 000077f7-26b1-4710-80cc-f6beddbdd157 | Ryan Adams and The Cardinals | alt country | 2 |
| **2** | 000077f7-26b1-4710-80cc-f6beddbdd157 | Ryan Adams and The Cardinals | whoa | 1 |
| **3** | 00034ede-a1f1-4219-be39-02f36853373e | O Rappa | Artist | 1 |
| **4** | 00034ede-a1f1-4219-be39-02f36853373e | O Rappa | Black | 1 |

Try out the 'groupby' method to organize data along AristID...

In [3]:
```
artists_tags.groupby('ArtistID')
```

Out[3]:
```
<pandas.core.groupby.DataFrameGroupBy object at 0x7fdd93e6dd10>
```

Let's Apply a function for the rows belonging to the same artists. We select the row with maximal count value for each ArtistID.

In [4]:
```
artists_tags.groupby('ArtistID').apply(lambda t: t[t.Count == t.Count.max()])
```

Out[4]:

| | | ArtistID | ArtistName |
|---|---|---|---|
| **ArtistID** | | | |
| **000077f7-26b1-4710-80cc-f6beddbdd157** | **1** | 000077f7-26b1-4710-80cc-f6beddbdd157 | Ryan Adam and The Cardinals |
| **00034ede-a1f1-4219-be39-02f36853373e** | **54** | 00034ede-a1f1-4219-be39-02f36853373e | O Rappa |
| **00050add-f633-4901-8d93-6f88c640c0da** | **59** | 00050add-f633-4901-8d93-6f88c640c0da | 9 Inch Dix |
| **000b1990-4dd8-4835-abcd-bb6038c13ac7** | **125** | 000b1990-4dd8-4835-abcd-bb6038c13ac7 | Hayden |
| **000ba849-700e-452e-8858-0db591587e4a** | **163** | 000ba849-700e-452e-8858-0db591587e4a | The Mutton Birds |
| **000d90ec-d64c-48a1-b775-e726fd240e9f** | **207** | 000d90ec-d64c-48a1-b775-e726fd240e9f | Get Cape. Wear Cape. Fly |
| | **239** | 000d90ec-d64c-48a1-b775-e726fd240e9f | Get Cape. Wear Cape. Fly |
| **000fc734-b7e1-4a01-92d1-f544261b43f5** | **363** | 000fc734-b7e1-4a01-92d1-f544261b43f5 | Cocteau Tw |
| **0019749d-ee29-4a5f-ab17-6bfa11deb969** | **450** | 0019749d-ee29-4a5f-ab17-6bfa11deb969 | DJ Food |
| **001aca82-d3bf-4a02-afd3-297740d12c14** | **510** | 001aca82-d3bf-4a02-afd3-297740d12c14 | David Dondero |
| **001ce2d7-c045-4343-b703-a4fc7dcee0a6** | **553** | 001ce2d7-c045-4343-b703-a4fc7dcee0a6 | Gravy Train |
| | **580** | 001ce2d7-c045-4343-b703-a4fc7dcee0a6 | Gravy Train |
| **002c6137-d274-4c06-bb70-6ba61c0e9faa** | **620** | 002c6137-d274-4c06-bb70-6ba61c0e9faa | Readymade FC |
| **002d040e-8a6c-4fae-bd3a-e3ffc322a2c6** | **661** | 002d040e-8a6c-4fae-bd3a-e3ffc322a2c6 | The Jeevas |
| **002e9f6e-13af-4347-83c5-f5ace70e0ec4** | **741** | 002e9f6e-13af-4347-83c5-f5ace70e0ec4 | Lulu |
| **00330ad2-6d94-4957-b3b0-7ca1ea6f6fd6** | **801** | 00330ad2-6d94-4957-b3b0-7ca1ea6f6fd6 | Blacklisted |
| **00370693-7679-46c1-8ddd-63e1d082c459** | **881** | 00370693-7679-46c1-8ddd-63e1d082c459 | Barbara Morgensterr |
| **0038bcbd-5f12-4a05-9f77-324652334345** | **939** | 0038bcbd-5f12-4a05-9f77-324652334345 | Ultrabeat |
| **0039c7ae-e1a7-4a7d-9b49-0cbc716821a6** | **1026** | 0039c7ae-e1a7-4a7d-9b49-0cbc716821a6 | Death Cab f Cutie |
| **003ae819-7141-4587-ae28-01aec96f4848** | **1106** | 003ae819-7141-4587-ae28-01aec96f4848 | Doris |
| **003b2747-b74a-46c1-a51e-aeaffe88256c** | **1130** | 003b2747-b74a-46c1-a51e-aeaffe88256c | Erdmöbel |
| **0043f16b-cc9f-4de4-8268-** | [illegible] | 0043f16b-cc9f-4de4-8268- | [illegible] |

We do not need all the columns

In [5]:
```
artists_tags.groupby('ArtistID')\
.apply(lambda t: t[t.Count == t.Count.max()][['ArtistName', 'Tag']].head(1))\
.reset_index()[['ArtistID', 'ArtistName', 'Tag']]
```

Out[5]:

| | ArtistID | ArtistName | Tag |
|---|---|---|---|
| **0** | 000077f7-26b1-4710-80cc-f6beddbdd157 | Ryan Adams and The Cardinals | alt country |
| **1** | 00034ede-a1f1-4219-be39-02f36853373e | O Rappa | rock |
| **2** | 00050add-f633-4901-8d93-6f88c640c0da | 9 Inch Dix | rap |
| **3** | 000b1990-4dd8-4835-abcd-bb6038c13ac7 | Hayden | indie |
| **4** | 000ba849-700e-452e-8858-0db591587e4a | The Mutton Birds | New Zealand |
| **5** | 000d90ec-d64c-48a1-b775-e726fd240e9f | Get Cape. Wear Cape. Fly | acoustic |
| **6** | 000fc734-b7e1-4a01-92d1-f544261b43f5 | Cocteau Twins | shoegaze |
| **7** | 0019749d-ee29-4a5f-ab17-6bfa11deb969 | DJ Food | ninja tune |
| **8** | 001aca82-d3bf-4a02-afd3-297740d12c14 | David Dondero | seen live |
| **9** | 001ce2d7-c045-4343-b703-a4fc7dcee0a6 | Gravy Train!!!! | dance |
| **10** | 002c6137-d274-4c06-bb70-6ba61c0e9faa | Readymade FC | electronica |
| **11** | 002d040e-8a6c-4fae-bd3a-e3ffc322a2c6 | The Jeevas | rock |
| **12** | 002e9f6e-13af-4347-83c5-f5ace70e0ec4 | Lulu | pop |
| **13** | 00330ad2-6d94-4957-b3b0-7ca1ea6f6fd6 | Blacklisted | hardcore |
| **14** | 00370693-7679-46c1-8ddd-63e1d082c459 | Barbara Morgenstern | electronic |
| **15** | 0038bcbd-5f12-4a05-9f77-324652334345 | Ultrabeat | dance |
| **16** | 0039c7ae-e1a7-4a7d-9b49-0cbc716821a6 | Death Cab for Cutie | indie |
| **17** | 003ae819-7141-4587-ae28-01aec96f4848 | Doris | swedish |
| **18** | 003b2747-b74a-46c1-a51e-aeaffe88256c | Erdmöbel | deutsch |
| **19** | 0043f16b-cc9f-4de4-8268-dbddf103d6d4 | Jaguar | NWOBHM |
| **20** | 0045feba-d3a2-4a75-bf78-f7f74038be56 | Enh?rjarna | viking rock |
| **21** | 004913ca-cc81-44ca-b1b5-6f82149cf475 | Lee Aaron | Canadian |
| **22** | 0049c4d7-6459-4b54-a865-fd19969f427f | Inside Out | hardcore |
| **23** | 0049e899-0bda-405a-a6e3-4734a5cb00f5 | Yae | japanese |
| **24** | 004b130c-ac97-4d3b-b6e5-a60178f036cf | Daniel May | hearts of space |
| **25** | 004e5eed-e267-46ea-b504-54526f1f377d | The Gathering | Gothic Metal |
| **26** | 0053dbd9-bfbc-4e38-9f08-66a27d914c38 | Bad Company | classic rock |
| **27** | 00565b31-14a3-4913-bd22-385eb40dd13c | King Diamond | heavy metal |
| **28** | 0059d7c5-2cf4-4555-9ae4-7d0a12681213 | Angizia | avantgarde |
| **29** | 006307bc-af41-4da1-aa11-b35a9c6f0316 | Curtis Fuller | jazz |
| **...** | ... | ... | ... |
| **20877** | ffb0c195-4aeb-4d33-8936-8aaf78e171ad | The Sins of Thy Beloved | Gothic Metal |
| **20878** | ffb18e19-64a4-4a65-b4ce-979e00c3c69d | The Album Leaf | post-rock |
| **20879** | ffb2b23d-1f88-4a7e-96ac-1b05805eb659 | Sewing With Nancie | chaos |
| **20880** | ffb307e6-123f-4186-ba03-491c17cd7992 | God Macabre | death metal |
| **20881** | ffb390b8-8df4-4b72-97d1-7b2fc008a452 | Cobra Starship | seen live |
| **20882** | ffb45501-b01c-443a-8cc4-6d80c9c13545 | Pearl Django | Gypsy |
| **20883** | ffb565eb-c5a4-49d1-866d-2d164627d956 | Sport | seen live |
| **20884** | ffb89363-2a3c-4b70-bf85-18782dba5b11 | Collide | industrial |

In [6]:
```
# TODO Implement this. You can change the function arguments if necessary
# Return a data structure that contains (artist id, artist name, top tag) for ev
ery artist
def calculate_top_tag(all_tags):
    return all_tags.groupby('ArtistID')\
    .apply(lambda t: t[t.Count == t.Count.max()][['ArtistName', 'Tag']].head())\
    .reset_index()[['ArtistID', 'ArtistName', 'Tag']]

top_tags = calculate_top_tag(artists_tags)

# Print the top tag for Nirvana
# Artist ID for Nirvana is 5b11f4ce-a62d-471e-81fc-a69a8278c7da
# Should be 'Grunge'
print "Top tag for Nirvana is %s" %\
top_tags[ top_tags.ArtistID == '5b11f4ce-a62d-471e-81fc-a69a8278c7da']['Tag'].it
em() # Complete this line
```

```
Top tag for Nirvana is Grunge
```

In [7]:
```
# An alternative way, storing the tag and artist name in a dictionary:

# top_tags = { r[1]['ArtistID'] : (r[1]['Tag'], r[1]['ArtistName']) for r in art
ists_tags.groupby('ArtistID')\
# .apply(lambda t: t[t.Count==t.Count.max()][['Tag','ArtistName']]).reset_index(
).iterrows()}
```

**b**. To do clustering we will be using `numpy` matrices. Create a matrix from **user_art_mat** with every row in the matrix representing a single artist. The matrix will have 846 columns, one for whether each user listened to the artist.

In [8]:
```
def create_user_matrix(input_data):
    return input_data[input_data.columns[1:]].values

user_np_matrix = create_user_matrix(user_art_mat)

print user_np_matrix.shape # Should be (17119, 846)
```

```
(17119, 846)
```

## 1.2 K-Means clustering

Having pre-processed the data we can now perform clustering on the dataset. In this assignment we will be using the python library scikit-learn (http://scikit-learn.org/stable/index.html) for our machine learning algorithms. scikit-learn provides an extensive library of machine learning algorithms that can be used for analysis. Here is a nice flow chart (http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html) that shows various algorithms implemented and when to use any of them. In this part of the assignment we will look at K-Means clustering

> **Note on terminology**: "samples" and "features" are two words you will come across frequently when you look at machine learning papers or documentation. "samples" refer to data points that are used as inputs to the machine learning algorithm. For example in our dataset each artist is a "sample". "features" refers to some representation we have for every sample. For example the list of 1s and 0s we have for each artist are "features". Similarly the bag-of-words approach from the previous homework produced "features" for each document.

#### K-Means algorithm

Clustering is the process of automatically grouping data points that are similar to each other. In the K-Means algorithm (http://en.wikipedia.org/wiki/K-means_clustering) we start with `K` initially chosen cluster centers (or centroids). We then compute the distance of every point from the centroids and assign each point to the centroid. Next we update the centroids by averaging all the points in the cluster. Finally, we repeat the algorithm until the cluster centers are stable.

## Running K-Means

#### K-Means interface

Take a minute to look at the scikit-learn interface for calling KMeans (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html). The constructor of the KMeans class returns a `estimator` on which you can call fit (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans.fit) to perform clustering.

#### K-Means parameters

From the above description we can see that there are a few parameters which control the K-Means algorithm. We will look at one parameter specifically, the number of clusters used in the algorithm. The number of clusters needs to be chosen based on domain knowledge of the data. As we do not know how many genres exist we will try different values and compare the results.

#### Timing your code

We will also measure the performance of clustering algorithms in this section. You can time the code in a cell using the **%%time** IPython magic (http://nbviewer.ipython.org/github/ipython/ipython/blob/1.x/examples/notebooks/Cell%20Magics.ipynb) as the first line in the cell.

> **Note**: By default, the scikit-learn KMeans implementation runs the algorithm 10 times with different center initializations. For this assignment you can run it just once by passing the `n_init` argument as 1.

#### Exercise 2

**a**. Run K-means using *5* cluster centers on the `user_np_matrix`.

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

# Run K-means using 5 cluster centers on user_np_matrix
kmeans_5 = KMeans(n_clusters=5, n_init=1)

kmeans_5.fit(user_np_matrix)
```

```
CPU times: user 26.3 s, sys: 36 ms, total: 26.3 s
Wall time: 26.3 s
```

**b**. Run K-means using *25* and *50* cluster centers on the `user_np_matrix`. Also measure the time taken for both cases.

In [10]:
```
%%time
kmeans_25 = KMeans(n_clusters=25, n_init=1)
kmeans_25.fit(user_np_matrix)
```

```
CPU times: user 1min 32s, sys: 60 ms, total: 1min 32s
Wall time: 1min 32s
```

In [11]:
```
%%time
kmeans_50 = KMeans(n_clusters=50, n_init=1)
kmeans_50.fit(user_np_matrix)
```

```
CPU times: user 3min 49s, sys: 92 ms, total: 3min 49s
Wall time: 3min 49s
```

## 1.3 Evaluating K-Means

In addition to the performance comparisons we also wish to compare how good our clusters are. To do this we are first going to look at internal evaluation metrics. For internal evaluation we only use the input data and the clusters created and try to measure the quality of clusters created. We are going to use two metrics for this:

#### Inertia

Inertia is a metric that is used to estimate how close the data points in a cluster are. This is calculated as the sum of squared distance for each point to it's closest centroid, i.e., its assigned cluster center. The intution behind inertia is that clusters with lower inertia are better as it means closely related points form a cluster.Inertia is calculated by scikit-learn by default.

#### Exercise 3

**a**. Print inertia for all the kmeans model computed above.

In [12]:
```
print "Inertia for KMeans with 5 clusters = %lf " % kmeans_5.inertia_
print "Inertia for KMeans with 25 clusters =  %lf " % kmeans_25.inertia_
print "Inertia for KMeans with 50 clusters = %lf " % kmeans_50.inertia_
```

```
Inertia for KMeans with 5 clusters = 349841.084301
Inertia for KMeans with 25 clusters =  321305.104358
Inertia for KMeans with 50 clusters = 308313.661346
```

**b**. Does KMeans run with 25 clusters have lower or greater inertia than the ones with 5 clusters ? Which algorithm is better and why ?

TODO: Answer question

**Silhouette Score:**

The silhouette score measures how close various clusters created are. A higher silhouette score is better as it means that we dont have too many overlapping clusters. The silhouette score can be computed using [sklearn.metrics.silhouette_score (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html#sklearn.metrics.silhouette_score)](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html#sklearn.metrics.silhouette_score) from scikit learn.

**c.** Calculate the Silhouette Score using 500 sample points for all the kmeans models.

In [13]:
```
from sklearn.metrics import silhouette_score

# NOTE: Use 500 sample points to calculate the silhouette score
def get_silhouette_score(data, model):
    return silhouette_score(data, labels=model.predict(data), sample_size=500)

print "Silhouette Score for KMeans with 5 clusters = %lf" % get_silhouette_score
(user_np_matrix, kmeans_5)
print "Silhouette Score for KMeans with 25 clusters = %lf " % get_silhouette_sco
re(user_np_matrix, kmeans_25)
print "Silhouette Score for KMeans with 50 clusters = %lf " % get_silhouette_sco
re(user_np_matrix, kmeans_50)
```

```
/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)

Silhouette Score for KMeans with 5 clusters = 0.228003

/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)

Silhouette Score for KMeans with 25 clusters = 0.086785

/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)

Silhouette Score for KMeans with 50 clusters = -0.007088
```

## 1.4 External Evaluation

While internal evaluation is useful, a better method for measuring clustering quality is to do external evaluation. This might not be possible always as we may not have ground truth data available. In our application we will use `top_tags` from before as our ground truth data for external evaluation. We will first compute purity and accuracy and finally we will predict tags for our **test** dataset.

**Exercise 4**

**a**. As a first step we will need to **join** the `artist_tags` data with the set of labels generated by K-Means model. That is, for every artist we will now have the top tag, cluster id and artist name in a data structure.

In [14]:
```
# Return a data structure that contains artist_id, artist_name, top tag, cluster
_label for every artist
def join_tags_labels(artists_data, user_data, kmeans_model):
    res = artists_data[['ArtistID', 'ArtistName', 'Tag']].merge(user_data, on='A
rtistID')
    clusters = kmeans_model.predict(res[ [x for x in res.columns if x not in ['A
rtistID', 'ArtistName', 'Tag']]].values)
    res['ClusterLabel'] = pd.Series(clusters)
    return res[['ArtistID', 'ArtistName', 'Tag', 'ClusterLabel']]

# Run the function for all the models
kmeans_5_joined = join_tags_labels(top_tags, user_art_mat, kmeans_5)
kmeans_25_joined = join_tags_labels(top_tags,user_art_mat, kmeans_25)
kmeans_50_joined = join_tags_labels(top_tags,user_art_mat, kmeans_50)
```

```
/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)
/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)
/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)
```

In [15]:
```
kmeans_5_joined.head()
```

Out[15]:

| | ArtistID | ArtistName | Tag | ClusterLabel |
|---|---|---|---|---|
| **0** | 000b1990-4dd8-4835-abcd-bb6038c13ac7 | Hayden | indie | 4 |
| **1** | 000ba849-700e-452e-8858-0db591587e4a | The Mutton Birds | New Zealand | 2 |
| **2** | 000d90ec-d64c-48a1-b775-e726fd240e9f | Get Cape. Wear Cape. Fly | acoustic | 4 |
| **3** | 000d90ec-d64c-48a1-b775-e726fd240e9f | Get Cape. Wear Cape. Fly | indie | 4 |
| **4** | 000fc734-b7e1-4a01-92d1-f544261b43f5 | Cocteau Twins | shoegaze | 1 |

**b**. Next we need to generate a genre for every cluster id we have (the cluster ids are from 0 to N-1). You can do this by **grouping** the data from the previous exercise on cluster id.

One thing you might notice is that we typically get a bunch of different tags associated with every cluster. How do we pick one genre or tag from this ? To cover various tags that are part of the cluster, we will pick the **top 5** tags in each cluster and save the list of top-5 tags as the genre for the cluster.

In [16]:
```
# Return a data structure that contains cluster_id, list of top 5 tags for every
cluster
def assign_cluster_tags(joined_data):
    return joined_data[['ClusterLabel', 'Tag']].groupby('ClusterLabel').agg(lamb
da x: ';'.join(x.value_counts().index[0:5])).reset_index()


kmeans_5_genres = assign_cluster_tags(kmeans_5_joined)
kmeans_25_genres = assign_cluster_tags(kmeans_25_joined)
kmeans_50_genres = assign_cluster_tags(kmeans_50_joined)
```

In [17]:
```
kmeans_50_genres.head()
```

Out[17]:

| | ClusterLabel | Tag |
|---|---|---|
| **0** | 0 | metal;death metal;Gothic Metal;Power metal;Pro... |
| **1** | 1 | jazz;reggae;french;rock;Progressive rock |
| **2** | 2 | 80s;new wave;pop;rock;female vocalists |
| **3** | 3 | electronic;rock;industrial |
| **4** | 4 | indie;electronic;Hip-Hop;seen live;rock |

#### Purity and Accuracy

Two commonly used metrics used for evaluating clustering using external labels are purity and accuracy. **Purity** measures the frequency of data belonging to the same cluster sharing the same class label i.e. if we have a number of items in a cluster how many of those items have the same label ? Meanwhile, **accuracy** measures the frequency of data from the same class appearing in a single cluster i.e. of all the items which have a particular label what fraction appear in the same cluster ?

**d**. Compute the purity for each of our K-Means models. To do this find the top tags of all artists that belong to a cluster. Check what fraction of these tags are covered by the top 5 tags of the cluster. Average this value across all clusters. **HINT**: We used similar ideas to get the top 5 tags in a cluster.

In [18]:
```
def f(x):
    good = x.value_counts().index[0:5]
    cnt = 0
    for i in good:
        cnt += x.value_counts()[i]
    return 1.0*cnt/len(x)

def get_cluster_purity(joined_data):
    return joined_data[['ClusterLabel', 'Tag']].groupby('ClusterLabel').agg(lamb
da x: f(x)).reset_index()['Tag'].mean()



print "Purity for KMeans with 5 centers %lf " % get_cluster_purity(kmeans_5_join
ed)
print "Purity for KMeans with 25 centers %lf " % get_cluster_purity(kmeans_25_jo
ined)
print "Purity for KMeans with 50 centers %lf " % get_cluster_purity(kmeans_50_jo
ined)
```

```
Purity for KMeans with 5 centers 0.389189
Purity for KMeans with 25 centers 0.653850
Purity for KMeans with 50 centers 0.757769
```

**e**. To compute the accuracy first get all the unique tags from *top_tags*. Then for each tag, compute how many artists are found in the largest cluster. We denote these as correct cluster assignments. For example, lets take a tag 'rock'. If there are 100 artists with tag 'rock' and say 90 of them are in one cluster while 10 of them are in another. Then we have 90 correct cluster assignments

Add the number of correct cluster assignments for all tags and divide this by the total size of the training data to get the accuracy for a model.

First check how Pandas groupby and aggregate functions work:

In [19]:
```
u_tags = top_tags['Tag'].unique()
#for t in u_tags:
t=u_tags[1]

print "Counting artists with top tag '%s' in the different clusters:" % t
print kmeans_5_joined[kmeans_5_joined['Tag']==t].groupby('ClusterLabel')['Artist
ID'].count()

print "The number of element in the max. cluster aka the correct cluster assignm
ent for '%s':" % t,
print kmeans_5_joined[kmeans_5_joined['Tag']==t].groupby('ClusterLabel')['Artist
ID'].count().max()

print "The total number of training points:", len(kmeans_5_joined)
```

```
Counting artists with top tag 'rock' in the different clusters:
ClusterLabel
0    105
1     22
2    250
3     42
4     85
Name: ArtistID, dtype: int64
The number of element in the max. cluster aka the correct cluster assignment f
or 'rock': 250
The total number of training points: 9969
```

In [20]:
```
t='hearts of space'
kmeans_5_joined[kmeans_5_joined['Tag']==t].groupby('ClusterLabel')['ArtistID']
```

Out[20]:
```
<pandas.core.groupby.SeriesGroupBy object at 0x7fdd66bb3950>
```

In [21]:
```
import numpy as np
def get_accuracy(joined_data):
    u_tags = joined_data['Tag'].unique()
    correct = 0
    for t in u_tags:
        correct += joined_data[joined_data['Tag']==t].groupby('ClusterLabel')['A
rtistID'].count().max()
    return 1.0 * correct / len(joined_data)

print "Accuracy of KMeans with 5 centers %lf " % get_accuracy(kmeans_5_joined)
print "Accuracy of KMeans with 25 centers %lf " % get_accuracy(kmeans_25_joined)
print "Accuracy of KMeans with 50 centers %lf " % get_accuracy(kmeans_50_joined)
```

```
Accuracy of KMeans with 5 centers 0.641890
Accuracy of KMeans with 25 centers 0.489317
Accuracy of KMeans with 50 centers 0.455211
```

**f.** What do the numbers tell you about the models? Do you have a favorite?

TODO: Your answer here.

## 1.5 Evaluating Test Data

Finally we can treat the clustering model as a multi-class classifier and make predictions on external test data. To do this we load the test data file **userart-mat-test.csv** and for every artist in the file we use the K-Means model to predict a cluster. We mark our prediction as successful if the artist's top tag belongs to one of the five tags for the cluster.

**Exercise 5**

**a** Load the testdata file and create a NumPy matrix named user_np_matrix_test.

In [22]:
```
user_art_mat_test = parse_user_artists_matrix(DATA_PATH + "/userart-mat-test.csv
")
# NOTE: the astype(float) converts integer to floats here
user_np_matrix_test = create_user_matrix(user_art_mat_test).astype(float)

user_np_matrix_test.shape # Should be (1902, 846)
```

Out[22]: (1902, 846)

**b.** For each artist in the test set, call **[predict (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans.predict)](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans.predict)** to get the predicted cluster. Join the predicted labels with test artist ids. Return 'artist_id', 'predicted_label' for every artist in the test dataset.

In [23]:
```
# For every artist return a list of labels
def predict_cluster(test_data, test_np_matrix, kmeans_model):
    res = test_data[['ArtistID', 'ArtistName', 'Tag']].merge(test_np_matrix, on=
'ArtistID')
    clusters = kmeans_model.predict(res[ [x for x in res.columns if x not in ['A
rtistID', 'ArtistName', 'Tag']]].values)
    res['ClusterLabel'] = pd.Series(clusters)
    return res[['ArtistID', 'ClusterLabel']]


# Call the function for every model from before
kmeans_5_predicted = predict_cluster(artists_tags, user_art_mat_test, kmeans_5)
kmeans_25_predicted = predict_cluster(artists_tags, user_art_mat_test, kmeans_25
)
kmeans_50_predicted = predict_cluster(artists_tags, user_art_mat_test, kmeans_50
)
```

```
/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)
/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)
/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)
```

**c**. Get the tags for the predicted genre and the tag for the artist from `top_tags`. Output the percentage of artists for whom the top tag is one of the five that describe its cluster. This is the *recall* of our model.

> NOTE: Since the tag data is not from the same source as user plays, there are artists in the test set for whom we do not have top tags. You should exclude these artists while making predictions and while computing the recall.

In [24]:
```
kmeans_5_genres
```

Out[24]:

| | ClusterLabel | Tag |
|---|---|---|
| **0** | 0 | rock;pop;classic rock;punk;emo |
| **1** | 1 | indie;electronic;Hip-Hop;rock;seen live |
| **2** | 2 | seen live;rock;indie;hardcore;punk |
| **3** | 3 | indie;rock;electronic;classic rock;pop |
| **4** | 4 | electronic;indie;jazz;Hip-Hop;seen live |

In [25]:
```
## Test how it works
kmeans_5_predicted.merge(top_tags, on='ArtistID').merge(kmeans_5_genres, on='Clu
sterLabel')\
.apply(lambda x: x['Tag_x'] in x['Tag_y'].split(';'), axis=1).sum()
```

Out[25]: 18851

In [26]:
```
# Calculate recall for our predictions
def verify_predictions(predicted_artist_labels, cluster_genres, top_tag_data):
    return 1.0 * predicted_artist_labels.merge(top_tag_data, on='ArtistID').merg
e(cluster_genres, on='ClusterLabel')\
    .apply(lambda x: x['Tag_x'] in x['Tag_y'].split(';'), axis=1).sum()/ len(pre
dicted_artist_labels)
```

**d**. Print the recall for each KMeans model. We define recall as num_correct_predictions / num_artists_in_test_data

In [27]:
```
# Use verify_predictions for every model
print "Recall of KMeans with 5 centers %lf " % verify_predictions(kmeans_5_predi
cted, kmeans_5_genres, top_tags )
print "Recall of KMeans with 25 centers %lf " % verify_predictions(kmeans_25_pre
dicted, kmeans_25_genres, top_tags )
print "Recall of KMeans with 50 centers %lf " % verify_predictions(kmeans_50_pre
dicted, kmeans_50_genres, top_tags )
```

```
Recall of KMeans with 5 centers 0.266216
Recall of KMeans with 25 centers 0.400616
Recall of KMeans with 50 centers 0.480137
```

## 1.5 Visualizing Clusters using PCA

Another way to evaluate clustering is to visualize the output of clustering. However the data we are working with is in 846 dimensions !, so it is hard to visualize or plot this. Thus the first step for visualization is to reduce the dimensionality of the data. To do this we can use Prinicipal Component Analysis (PCA) (http://en.wikipedia.org/wiki/Principal_component_analysis). PCA reduces the dimension of data and keeps only the most significant components of it. This is a commonly used technique to visualize data from high dimensional spaces.

> **NOTE**: We use RandomizedPCA (http://scikit-learn.org/stable/modules/decomposition.html#approximate-pca), an approximate version of the algorithm as this has lower memory requirements. The approximate version is good enough when we are reducing to a few dimensions (2 in this case). We also sample the input data before PCA to further reduce memory requirements.

**Exercise 6**

**a**. Calcluate the RandomizedPCA of the sampled training data set `sampled_data` and reduce it to 2 components. Use the fit_transform (http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.RandomizedPCA.html#sklearn.decomposition.RandomizedPCA.fit_transform) method to do this.

In [28]:
```
from sklearn.decomposition import RandomizedPCA
import numpy as np

input_data = user_np_matrix

sample_percent = 0.50
rows_to_sample = int(np.ceil(sample_percent * user_np_matrix.shape[0]))
sampled_data = input_data[np.random.choice(input_data.shape[0], rows_to_sample,
replace=False),:]

# Return the data reduced to 2 principal components
def get_reduced_data(input_data):
    rpca = RandomizedPCA(n_components=2)
    return rpca.fit_transform(input_data)

user_np_2d = get_reduced_data(sampled_data)
```

**b**. Fit the reduced data with the KMeans model with 5 cluster centers. Plot the cluster centers and all the points. Make sure to color points in every cluster differently to see a visual separation. You may find `scatter` (http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.scatter) and `plot` (http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot) functions from matplotlib to be useful.

In [29]:
```
# TODO: Write code to fit and plot reduced_data.
import matplotlib.pyplot as plt
%matplotlib inline

sampled_pred = kmeans_5.predict(sampled_data)
plt.figure(figsize=(16,16))

plt.scatter( x=user_np_2d[:,0], y=user_np_2d[:,1], c=sampled_pred, s=20, alpha=0
.3)
```

```
/usr/local/lib/python2.7/dist-packages/sklearn/utils/validation.py:420: DataCo
nversionWarning: Data with input dtype int64 was converted to float64.
  warnings.warn(msg, DataConversionWarning)
```

Out[29]: <matplotlib.collections.PathCollection at 0x7fdd72d31e10>