# Lab 2 Solutions

November 18, 2020

## 1 Introduction

This notebook provides a solution to Lab 2 of NANO281 - Data Science in Materials Science.

[1]:
```
import itertools
from collections import Counter

import pandas as pd
import numpy as np
from pymatgen import Element, Composition

import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,␣
↪QuadraticDiscriminantAnalysis
from sklearn.linear_model import LogisticRegression

rcparams = {'legend.fontsize': 20,
            'figure.figsize': (12, 8),
            'axes.labelsize': 24,
            'axes.titlesize': 28,
            'xtick.labelsize':20,
            'ytick.labelsize': 20}
mpl.rcParams.update(rcparams)

%matplotlib inline
%config InlineBackend.figure_format ='retina'
```

## 2 Q1 Exploratory data analysis

[2]:
```
orig_data = pd.read_csv("data.csv", na_filter=False)
```

[3]:
```
orig_data.head()
```

[3]:
```
      task_id formula  formation_energy_per_atom  e_above_hull  band_gap  \
0  mp-1007923   CrNi3                  -0.016354      0.000802       0.0
1  mp-1008754   TbHg2                  -0.069428      0.329909       0.0
2  mp-1016886   MgVO3                  -2.241438      0.545977       0.0
3  mp-1018902  PrSbPt                  -1.111238      0.000000       0.0
4  mp-1020595  Rb8PO3                  -0.772860      0.491666       0.0

   has_bandstructure
0              False
1               True
2               True
3               True
4               True
```

### 2.1 1. How many elements are there in this data set?

[4]:
```
elements = []
for i in orig_data['formula']:
    elements.append(Composition(i).elements)

orig_data["elements"] = elements

unique = set(itertools.chain(*elements))

print("There are %d elements." % len(unique))
```

```
There are 89 elements.
```

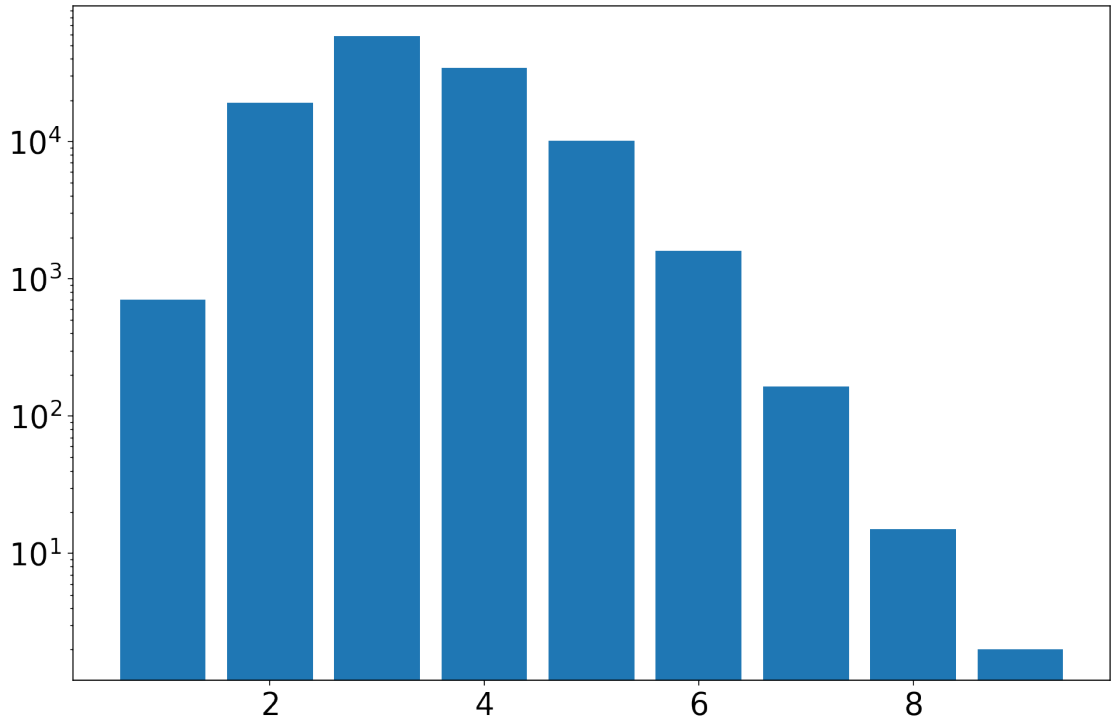
### 2.2 2. What is the maximum number of elements in a single structure?

[5]:
```
nelements = [len(l) for l in elements]
max_els = max(nelements)
print("Maximum number of elements in a structure is %d" % max_els)
```

```
Maximum number of elements in a structure is 9
```

### 2.3 3. Plot a histogram of the number of materials having 1, 2, 3, ... max_els.

[6]:
```
import collections

el_count = collections.Counter(nelements)
fig, ax = plt.subplots(figsize=(12, 8))
plt.bar(range(1, max_els+1), [el_count[i] for i in range(1, max_els+1)])

# Given the very different number of materials in each category, a log scale is␣
 ↪better.
plt.yscale('log')
```




### 2.4 4. Count the number of materials where each element is present. Sort this count and answer the following questions. What are the 10 most common elements in this data set? and what are the 10 least common elements in this data set?

[7]:
```
el_count = Counter(itertools.chain(*elements))
counts = sorted(el_count.items(), key=lambda k: k[1])
print('Top 10 most common elements are ')
print(counts[-10:][::-1])
print('Top 10 least common elements are ')
print(counts[:10])
```

```
Top 10 most common elements are
[(Element O, 60428), (Element Li, 18580), (Element P, 13321), (Element Mn,
11233), (Element S, 10516), (Element Fe, 10109), (Element N, 9723), (Element F,
9606), (Element Si, 8940), (Element Mg, 8711)]
Top 10 least common elements are
[(Element Ne, 1), (Element Ar, 2), (Element He, 8), (Element Kr, 15), (Element
Xe, 147), (Element Pa, 253), (Element Ac, 297), (Element Np, 340), (Element Pu,
388), (Element Pm, 515)]
```

# 3 Q2 - Data cleaning and feature computations

## 3.1 1. Filter out materials that contain noble gas elements and save it in the variable data. How many materials are left? This number is stored in variable n.

[8]:
```
elements_temp = [Element.from_Z(i) for i in range(1, 95)]
noble_gases = [i for i in elements_temp if i.is_noble_gas]

def is_contain_noble_gas(elements):
    return any([i in noble_gases for i in elements])

not_contain_noble_gas = [not is_contain_noble_gas(i) for i in␣
 ↪orig_data['elements']]

data = orig_data.copy()
data = data[not_contain_noble_gas]
print("The remaining number of materials is %d" % data.shape[0])
```

```
The remaining number of materials is 124342
```

## 3.2 2. Load the element property data file element_properties.csv in variable element_data using pandas by setting index_col=0 in pandas.read_csv function. How many NaN are there in each column?

[9]:
```
element_data = pd.read_csv('element_properties.csv', index_col=0)
```

[10]:
```
print('The numbers of NaN in each column are')
element_data.isna().sum()
```

```
The numbers of NaN in each column are
```

[10]:
```
AtomicRadius              7
AtomicVolume              2
AtomicWeight              0
BulkModulus              26
BoilingT                  2
Column                    0
CovalentRadius            0
Density                   2
ElectronAffinity          9
Electronegativity         4
FirstIonizationEnergy     1
HeatCapacityMass         10
Row                       0
phi                      22
SecondIonizationEnergy   12
ShearModulus             34
```

```
dtype: int64
```

### 3.3 3. Compute the mean values for each column. What are the means for each column? For each column, fill the NaN with the mean value of that column. This is a common data imputation technique.

[11]:
```
mean_columns = element_data.mean(skipna=True)
```

[12]:
```
print('The means for each column are')
mean_columns
```

```
The means for each column are
```

[12]:
```
AtomicRadius                 1.500682
AtomicVolume              3426.442121
AtomicWeight               116.153896
BulkModulus                 90.794203
BoilingT                  2549.858065
Column                       8.315789
CovalentRadius             151.810526
Density                   7489.235725
ElectronAffinity            76.162209
Electronegativity            1.747033
FirstIonizationEnergy        8.094711
HeatCapacityMass             0.635447
Row                          4.831579
phi                          4.034247
SecondIonizationEnergy      18.947504
ShearModulus                47.362295
dtype: float64
```

[13]:
```
element_data = element_data.fillna(mean_columns)
```

### 3.4 4. Compute the composition-averaged AtomicRadius for all materials and store the results in variable `atomic_radius`. For example, averaged `AtomicRadius` for `Li2O` can be computed as `(2 * 1.45 + 0.6) / 3`, where `1.45` is the `AtomicRadius` for `Li` and `0.6` is the `AtomicRadius` for `O`.

[14]:
```
data['composition'] = [Composition(i).to_data_dict['unit_cell_composition']
                       for i in data['formula']]
```

[15]:
```
def composition_to_dict(c):
    if isinstance(c, dict):
        unit_cell_composition = c
    else:
        if isinstance(c, str):
```

```
            c = Composition(c)
        unit_cell_composition = c.to_data_dict['unit_cell_composition']
    return unit_cell_composition

def compute_average_from_composition(c, prop):
    unit_cell_composition = composition_to_dict(c)
    res = 0
    total = 0
    for i, j in unit_cell_composition.items():
        res += element_data.loc[i, prop] * j
        total += j
    return res / total

def get_maxmin_properties(c, prop, mode='max'):
    if mode == 'max':
        func = np.max
    elif mode == "min":
        func = np.min

    unit_cell_composition = composition_to_dict(c)
    res = func([element_data.loc[i, prop] for i in unit_cell_composition])
    return res
```

[16]:
```
atomic_radius = [compute_average_from_composition(i, 'AtomicRadius') for i in␣
↪data['composition']]
```

### 3.5 5. Compute the composition-averaged properties for all properties in element_data and for all materials. Store the results in average_properties. average_properties should have a dimension of (n, 16) where n is the number of materials and 16 is the number of properties.

[17]:
```
properties = element_data.columns
average_properties = []
for prop in properties:
    average_properties.append([compute_average_from_composition(i, prop) for i␣
↪in data['composition']])
```

[18]:
```
average_properties = np.array(average_properties).T
```

[19]:
```
average_properties.shape
```

[19]:
```
(124342, 16)
```

### 3.6 6. Similar to the previous computations of average properties, compute the maximum properties and minimum properties for all properties and all materials, and store them in variables `max_properties` and `min_properties` respectively. Both variables should have dimension `(n, 16)`.

[20]:
```
max_properties = []
min_properties = []
for prop in properties:
    max_properties.append([get_maxmin_properties(i, prop, mode='max') for i in␣
 ↪data['composition']])
    min_properties.append([get_maxmin_properties(i, prop, mode='min') for i in␣
 ↪data['composition']])

max_properties = np.array(max_properties).T
min_properties = np.array(min_properties).T
```

[21]:
```
max_properties.shape, min_properties.shape
```

[21]:
```
((124342, 16), (124342, 16))
```

### 3.7 7. Concatenate average_properties, max_properties and min_properties, and store the result in variable design_matrix with dimension (n, 48).

[22]:
```
design_matrix = np.concatenate([average_properties,
                                max_properties, min_properties], axis=1)
design_matrix.shape
```

[22]:
```
(124342, 48)
```

## 4 Q3 - Regression and classification modeling

[23]:
```
targets = data[['band_gap', 'formation_energy_per_atom','e_above_hull']]
```

### 4.1 1. Split the data (design_matrix as X, and targets as y) into train and test (ratio 90%:10%), and store them in train_X, train_y for train and test_X and test_y for test. Store the normalized design matrices to norm_train_X, norm_test_X. To make sure the data is reproducible, set the `random_state=42` in `sklearn.model_selection.train_test_split`.

[24]:
```
train_x, test_x, train_y, test_y = train_test_split(design_matrix, targets,␣
 ↪test_size=0.1,
                                                    random_state=42)
```

### 4.2 2. Compute the mean and standard deviation of columns in `train_X`. Both of them should be length 48 vectors. Use them to normalize `train_X` and `test_X`, so that each column has a mean of 0 and standard deviation of 1. Store the normalized design matrices to `norm_train_X`, `norm_test_X`.

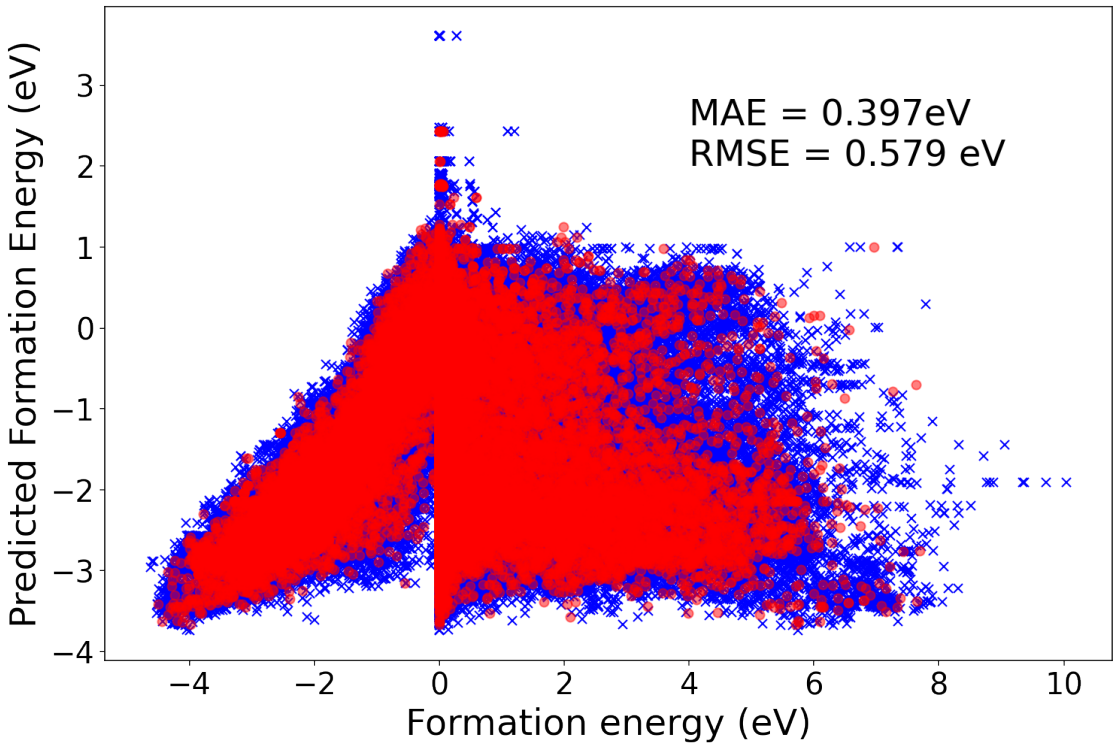
[25]:
```
train_x_mean = np.mean(train_x, axis=0)
train_x_std = np.std(train_x, axis=0)
train_x_mean.shape, train_x_std.shape
```

[25]:
```
((48,), (48,))
```

[26]:
```
norm_train_x = (train_x - train_x_mean) / train_x_std
norm_test_x = (test_x - train_x_mean) / train_x_std
```

### 4.3 3. Train a linear model to predict formation_energy_per_atom. What are the mean absolute error (MAE) and root mean squared error (RMSE) on the test data?

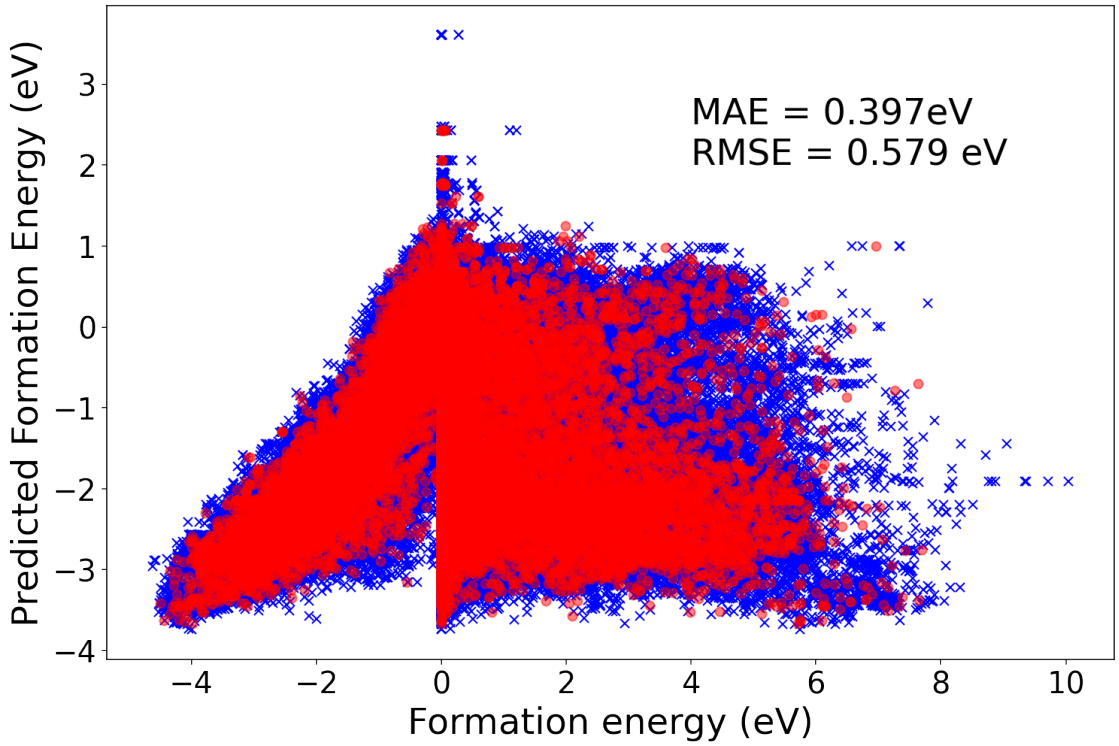
[27]:
```
lr = LinearRegression()
lr.fit(norm_train_x, train_y[['formation_energy_per_atom']].values)
pred_test_y = lr.predict(norm_test_x)
pred_train_y = lr.predict(norm_train_x)

mae = mean_absolute_error(pred_test_y, test_y[['formation_energy_per_atom']].
 ↪values)
rmse = mean_squared_error(pred_test_y, test_y[['formation_energy_per_atom']].
 ↪values, squared=False)

fig, ax = plt.subplots(figsize=(12, 8))
plt.plot(train_y, pred_train_y, 'bx')
plt.plot(test_y, pred_test_y, 'ro', alpha=0.5)
plt.xlabel("Formation energy (eV)")
plt.ylabel("Predicted Formation Energy (eV)")
plt.annotate(f"MAE = {mae:.3f}eV\nRMSE = {rmse:.3f} eV", (4, 2), fontsize=24);
```




### 4.4 4. Train a Ridge regression model and a LASSO regression model using alpha=0.1, what are the test MAE and RMSE?

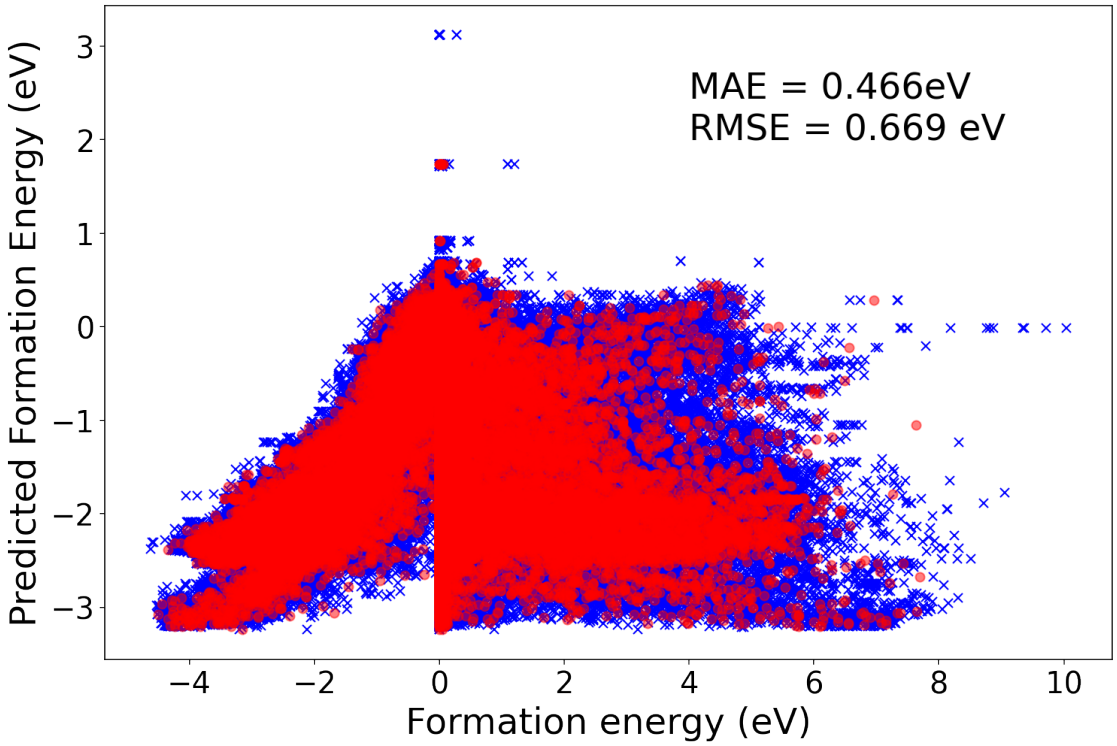
[28]:
```
ridge = Ridge(alpha=0.1)
ridge.fit(norm_train_x, train_y[['formation_energy_per_atom']].values)

# The models were wrongly used. Corrections done by Ji on Nov 17 2020
# pred_test_y = lr.predict(norm_test_x)
# pred_train_y = lr.predict(norm_train_x)

pred_test_y = ridge.predict(norm_test_x)
pred_train_y = ridge.predict(norm_train_x)

mae = mean_absolute_error(pred_test_y, test_y[['formation_energy_per_atom']].
 ↪values)
rmse = mean_squared_error(pred_test_y, test_y[['formation_energy_per_atom']].
 ↪values, squared=False)

fig, ax = plt.subplots(figsize=(12, 8))
plt.plot(train_y, pred_train_y, 'bx')
plt.plot(test_y, pred_test_y, 'ro', alpha=0.5)
plt.xlabel("Formation energy (eV)")
```

```
plt.ylabel("Predicted Formation Energy (eV)")
plt.annotate(f"MAE = {mae:.3f}eV\nRMSE = {rmse:.3f} eV", (4, 2), fontsize=24);
```




[29]:
```
lasso = Lasso(alpha=0.1)
lasso.fit(norm_train_x, train_y[['formation_energy_per_atom']].values)

# The models were wrongly used. Corrections done by Ji on Nov 17 2020
# pred_test_y = lr.predict(norm_test_x)
# pred_train_y = lr.predict(norm_train_x)

pred_test_y = lasso.predict(norm_test_x)
pred_train_y = lasso.predict(norm_train_x)

mae = mean_absolute_error(pred_test_y, test_y[['formation_energy_per_atom']].
 ↪values)
rmse = mean_squared_error(pred_test_y, test_y[['formation_energy_per_atom']].
 ↪values, squared=False)

fig, ax = plt.subplots(figsize=(12, 8))
plt.plot(train_y, pred_train_y, 'bx')
plt.plot(test_y, pred_test_y, 'ro', alpha=0.5)
plt.xlabel("Formation energy (eV)")
plt.ylabel("Predicted Formation Energy (eV)")
```

```
plt.annotate(f"MAE = {mae:.3f}eV\nRMSE = {rmse:.3f} eV", (4, 2), fontsize=24);
```




### 4.5 5. Let's define band_gap < 0.001 as metallic and band_gap >= 0.001 as nonmetallic. Construct linear discriminant analysis, quadratic discriminant analysis, and logistic regression models on train data and predict the accuracy of the models on test data.

[30]:
```
train_bg_label = train_y['band_gap'] < 0.001
test_bg_label = test_y['band_gap'] < 0.001
```

[31]:
```
lda = LinearDiscriminantAnalysis(solver='svd')
qda = QuadraticDiscriminantAnalysis()
logistic = LogisticRegression(penalty='none', max_iter=1000)

lda.fit(norm_train_x, train_bg_label)
qda.fit(norm_train_x, train_bg_label)
logistic.fit(norm_train_x, train_bg_label)
```

[31]:
```
LogisticRegression(max_iter=1000, penalty='none')
```

[32]:
```
lda_accuracy = lda.score(norm_test_x, test_bg_label)
qda_accuracy = qda.score(norm_test_x, test_bg_label)
```

```
lg_accuracy = logistic.score(norm_test_x, test_bg_label)
```

[33]:
```
print('LDA, QDA and logistic regression accuracies are %.3f, %.3f, %.3f,␣
↪respectively '% (
    lda_accuracy, qda_accuracy, lg_accuracy))
```

```
LDA, QDA and logistic regression accuracies are 0.787, 0.785, 0.792,
respectively
```

### 4.6 6. What are the problems of using only the compositions to predict material properties?

The models cannot distinguish polymorphs. This leads to problems where the X's are the same, but the y's are very different in the dataset (X, y).

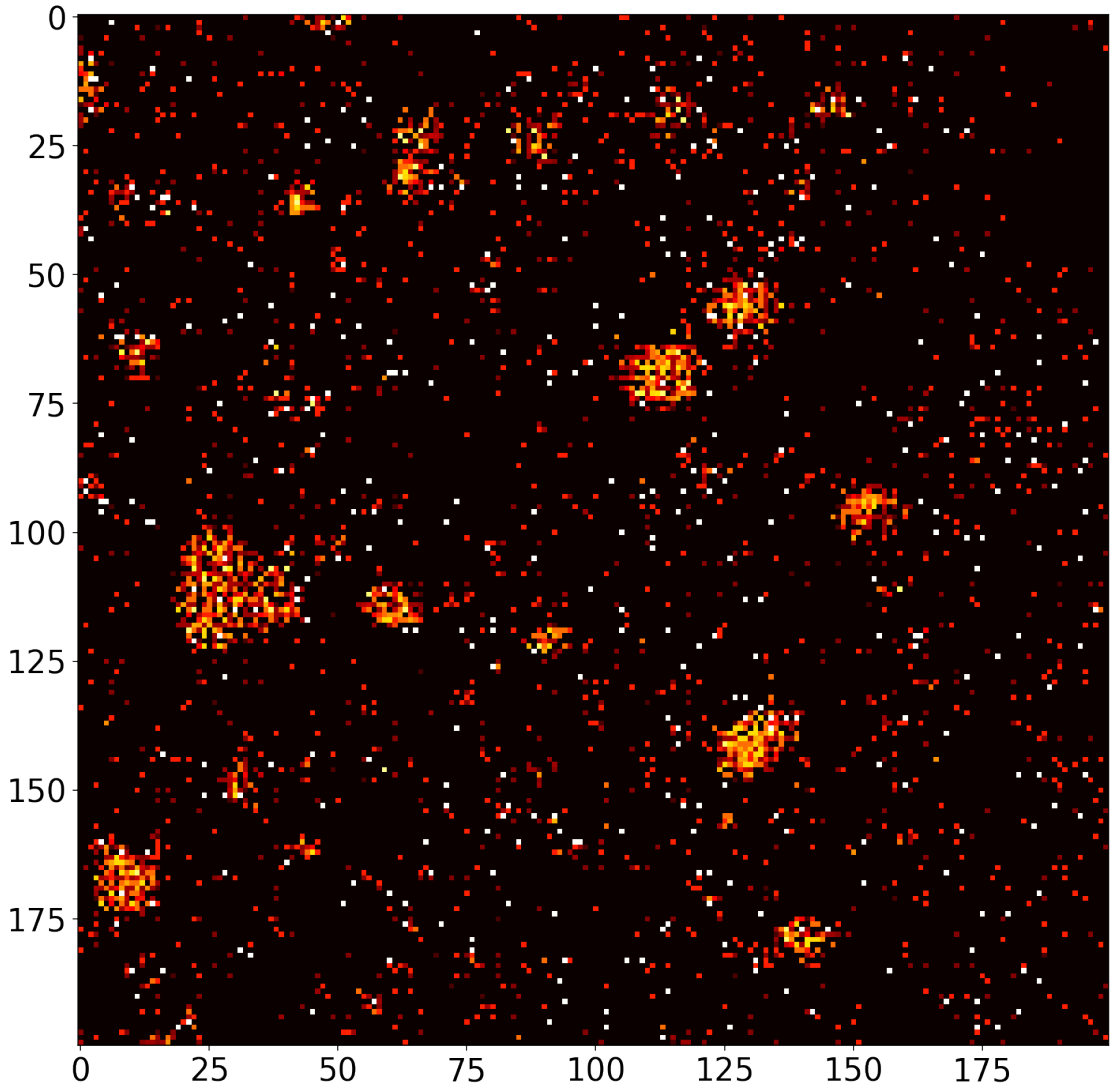
For example, graphite carbon and diamond both have the chemical formula C, but their band gaps are vastly different, i.e., graphite is a conductor (band gap of 0) and diamond is an insulator (band gap of 5.5 eV).

## 5 Q4 - Clustering

### 5.1 1. Read in the image as a numpy array using matplotlib. Show the image in your Jupyter notebook. What are the dimensions of the array?

[36]:
```
from matplotlib import image
import matplotlib.pyplot as plt
import numpy as np
# load image as pixel array
data = image.imread('catalyst.png')
print("The data has shape %d x %d." % (data.shape))
f, ax = plt.subplots(figsize=(12, 12))
plt.imshow(data, cmap=plt.cm.hot);
```

```
The data has shape 200 x 200.
```

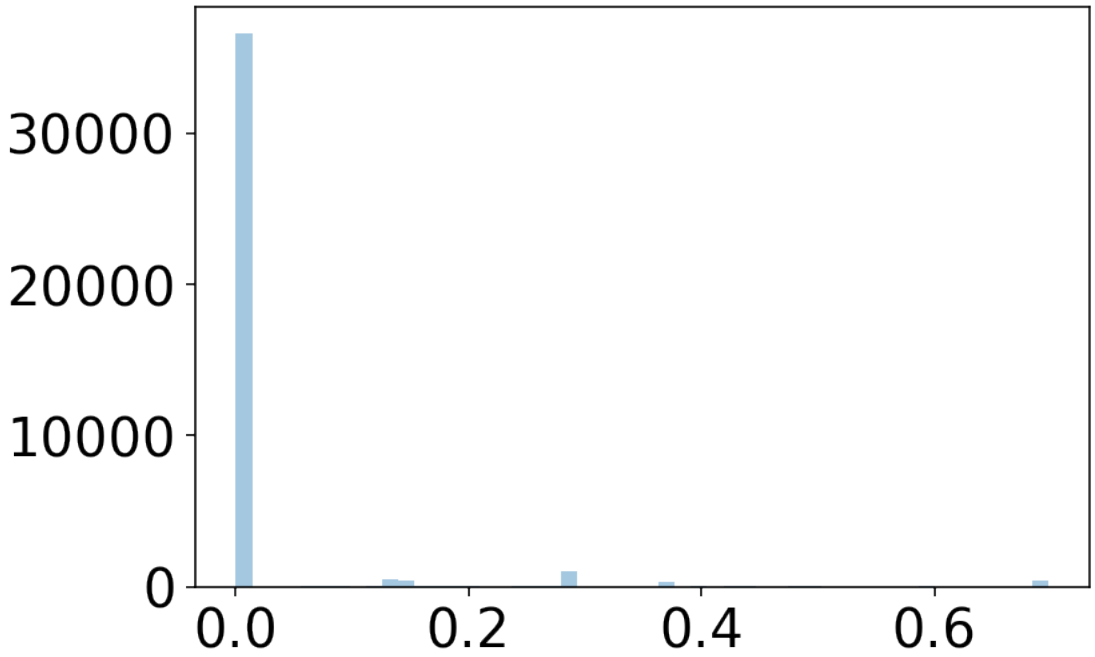



### 5.2 2. Plot the distribution of the values in the numpy array representing the image. Note that the values in the numpy array are between zero and 1.

[37]:
```
import seaborn as sns
sns.distplot(data.ravel(), kde=False)
```

```
/Users/miracle_qi/miniconda3/lib/python3.8/site-
packages/seaborn/distributions.py:2551: FutureWarning: `distplot` is a
deprecated function and will be removed in a future version. Please adapt your
code to use either `displot` (a figure-level function with similar flexibility)
or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

[37]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb0f8db64c0>

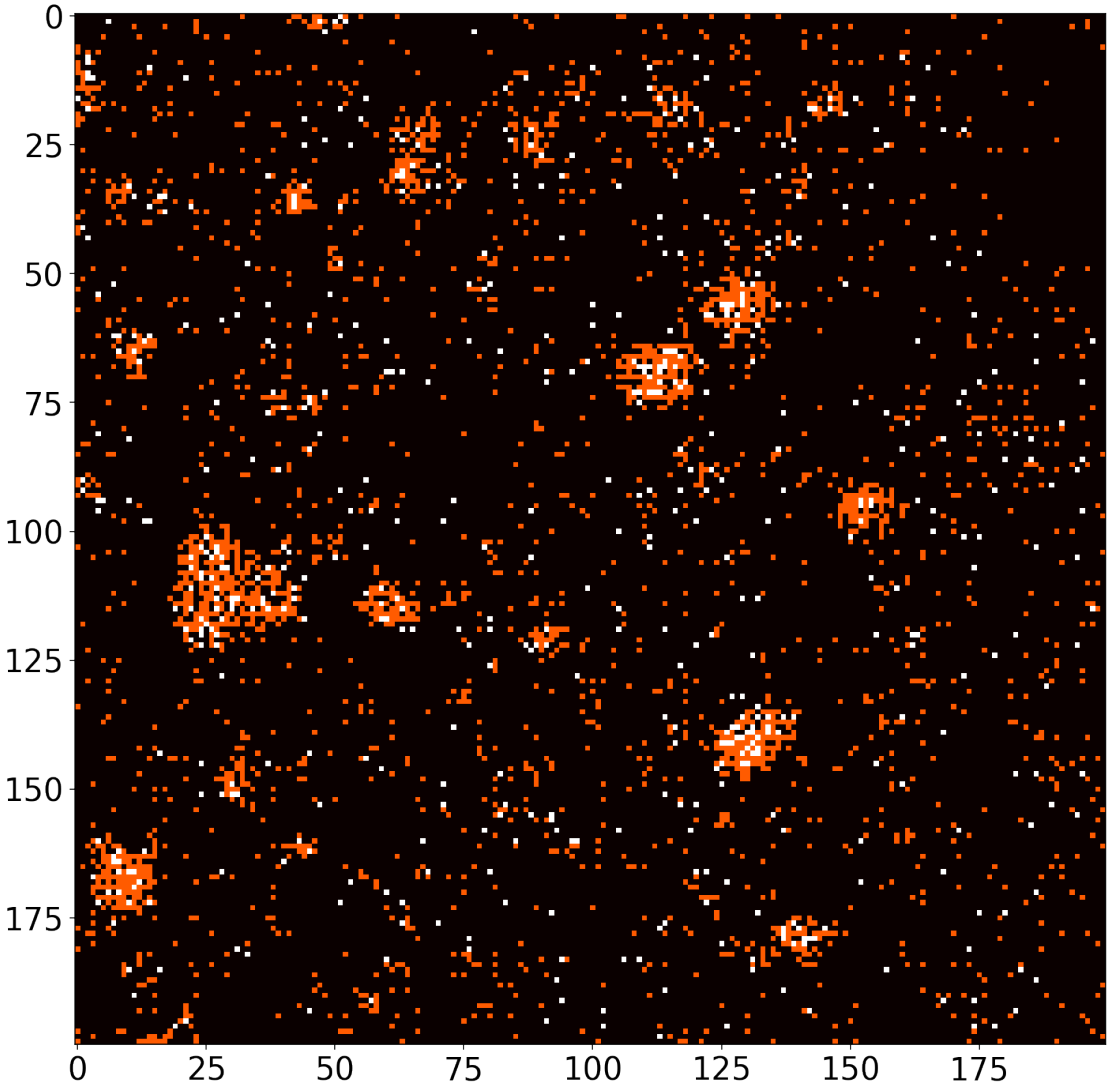



**5.3 3. Measured images has a variety of levels. Sometimes, we want to label each pixel at pre-specified levels, e.g., 0 representing the background, and fixed values representing certain features. This is known as vector quantization. Here, we will quantize the image using K-means. We know for a fact that there are two elements (Pd and Ru) in the system. Using K-means, quantization the image such that there are three levels representing the background, and one level for each element. Ensure that 0 corresponds to the background (this should be the cluster with the largest number of data points) and non-zero levels correspond to the elements. Plot the quantized image.**

[38]:
```
from sklearn.cluster import KMeans

values = data.ravel()
values = values.reshape(-1, 1)
kmeans = KMeans(3).fit(values)
labels = kmeans.labels_
c = Counter(labels)
ranked = sorted(c.keys(), key=lambda k: -c[k])
mapping = {k: i for k, i in zip(ranked, range(3))}
mapped_labels = [mapping[l] for l in labels]
quantized = np.array(mapped_labels) / 3
```

```
quantized = quantized.reshape(data.shape)
f, ax = plt.subplots(figsize=(12, 12))
plt.imshow(quantized, cmap=plt.cm.hot)
```

[38]: <matplotlib.image.AxesImage at 0x7fb0f8fdcd60>



### 5.4 4. For the purposes of this last exercise, we will not attempt to distinguish between different elements. Any value within the numpy array that is > 0 is considered a catalyst particle. Use K-means clustering to distinguish identify clusters of metal particles (you will need to figure out what a good value of K is). Plot your clustered image, ensuring that each cluster has a different color. Comment on how you chose your value of K.

[39]:
```
coordinates = np.array(np.nonzero(data)).T
```

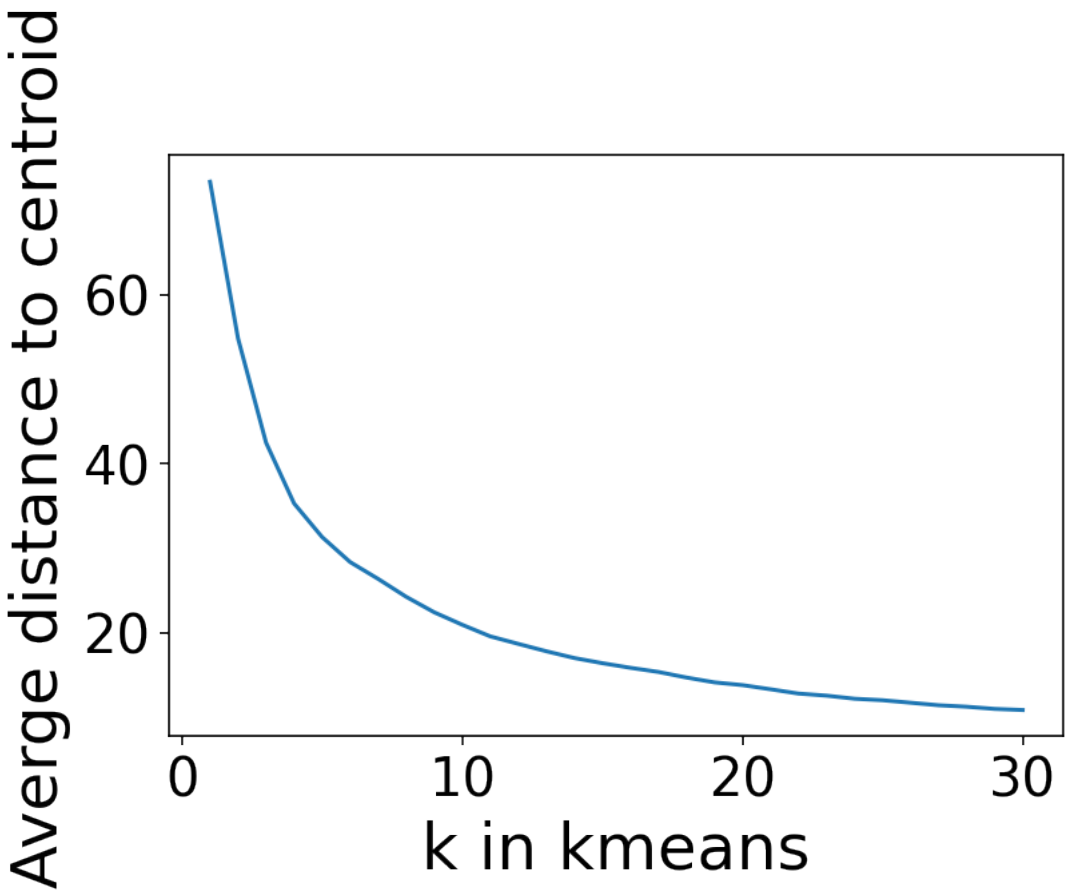
[40]:
```
from scipy.spatial.distance import cdist
import random
def cluster(k):
    kmeans = KMeans(k).fit(coordinates)
    new_image = np.zeros(data.shape)
    shuffled_labels = list(range(k))
    random.shuffle(shuffled_labels)
    mapping = dict(zip(range(k), shuffled_labels))
    for ind, label in zip(coordinates, kmeans.labels_):
        new_image[ind[0], ind[1]] = mapping[label] / k
    return new_image

def compute_avg_dist_to_centroid(k):
    kmeans = KMeans(k).fit(coordinates)
    return sum(np.min(cdist(coordinates, kmeans.cluster_centers_,
                            'euclidean'), axis=1)) / coordinates.shape[0]
```

[41]:
```
dists = []
n_clusters = list(range(1, 31))
for i in n_clusters:
    if i % 10 == 0:
        print(f"Cluster number {i}")
    dists.append(compute_avg_dist_to_centroid(i))
```

```
Cluster number 10
Cluster number 20
Cluster number 30
```

[42]:
```
plt.plot(n_clusters, dists)
plt.xlabel('k in kmeans')
plt.ylabel('Averge distance to centroid')
```

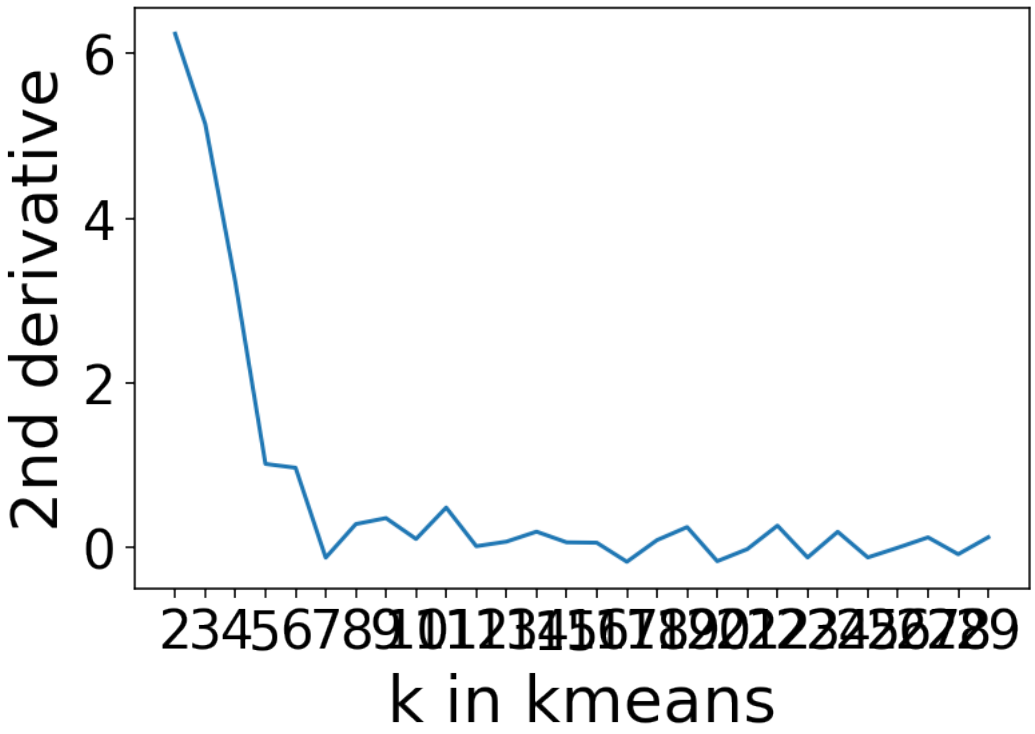
[42]:
```
Text(0, 0.5, 'Averge distance to centroid')
```




Ideally, we should choose the corner of the elbow. Here let's do some simple estimate regarding where it is.

[43]:
```
left = np.array(dists[:-2])
mid = np.array(dists[1:-1])
right = np.array(dists[2:])
second_der = left + right -2 * mid
plt.plot(n_clusters[1:-1], second_der)
_ = plt.xticks(n_clusters[1:-1])
plt.xlabel('k in kmeans')
plt.ylabel('2nd derivative')
```

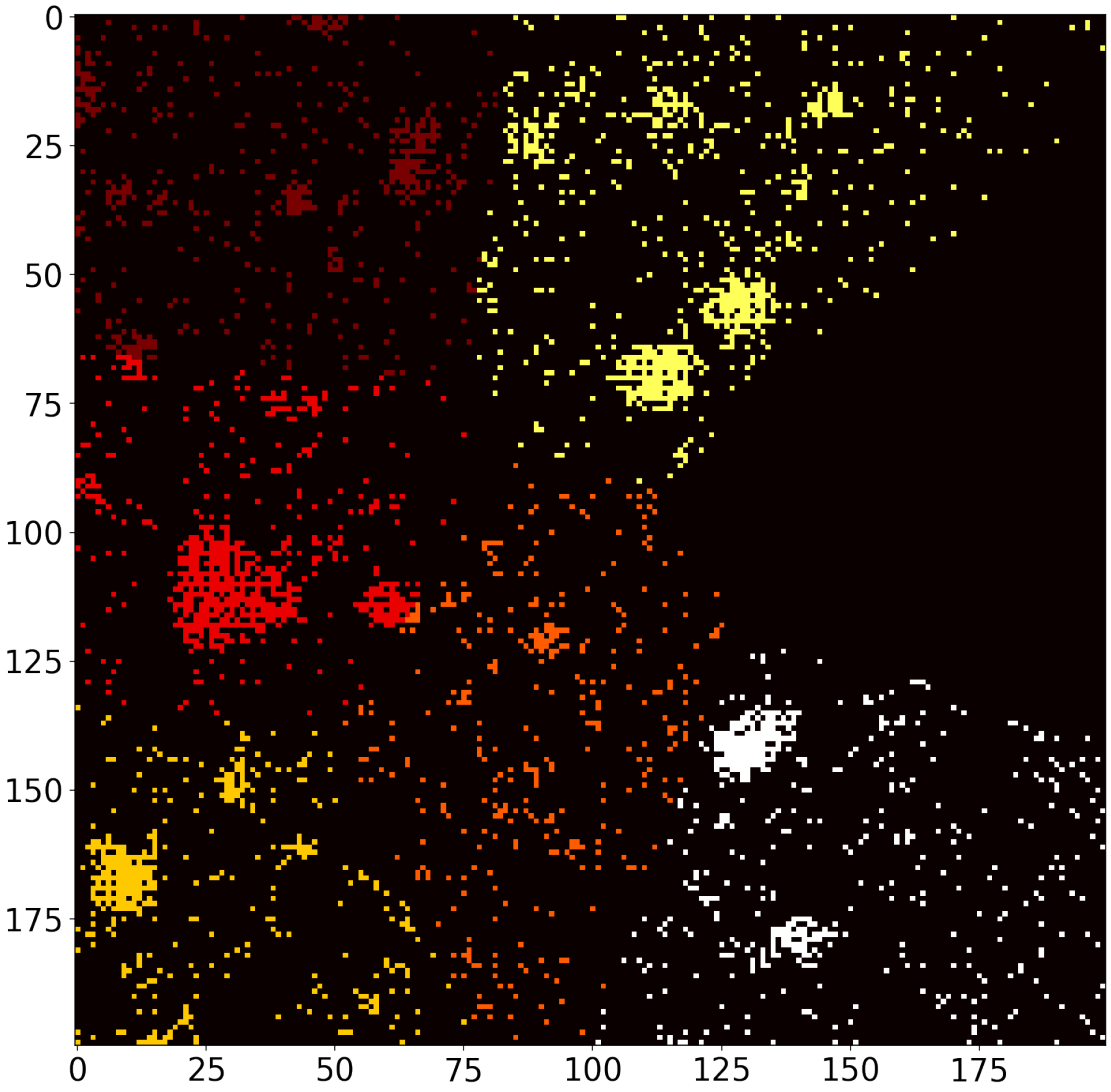
[43]:
```
Text(0, 0.5, '2nd derivative')
```




Here we can choose k = 7

[44]:
```
f, ax = plt.subplots(figsize=(12, 12))
plt.imshow(cluster(7), cmap=plt.cm.hot)
```

[44]: <matplotlib.image.AxesImage at 0x7fb0f8c78700>

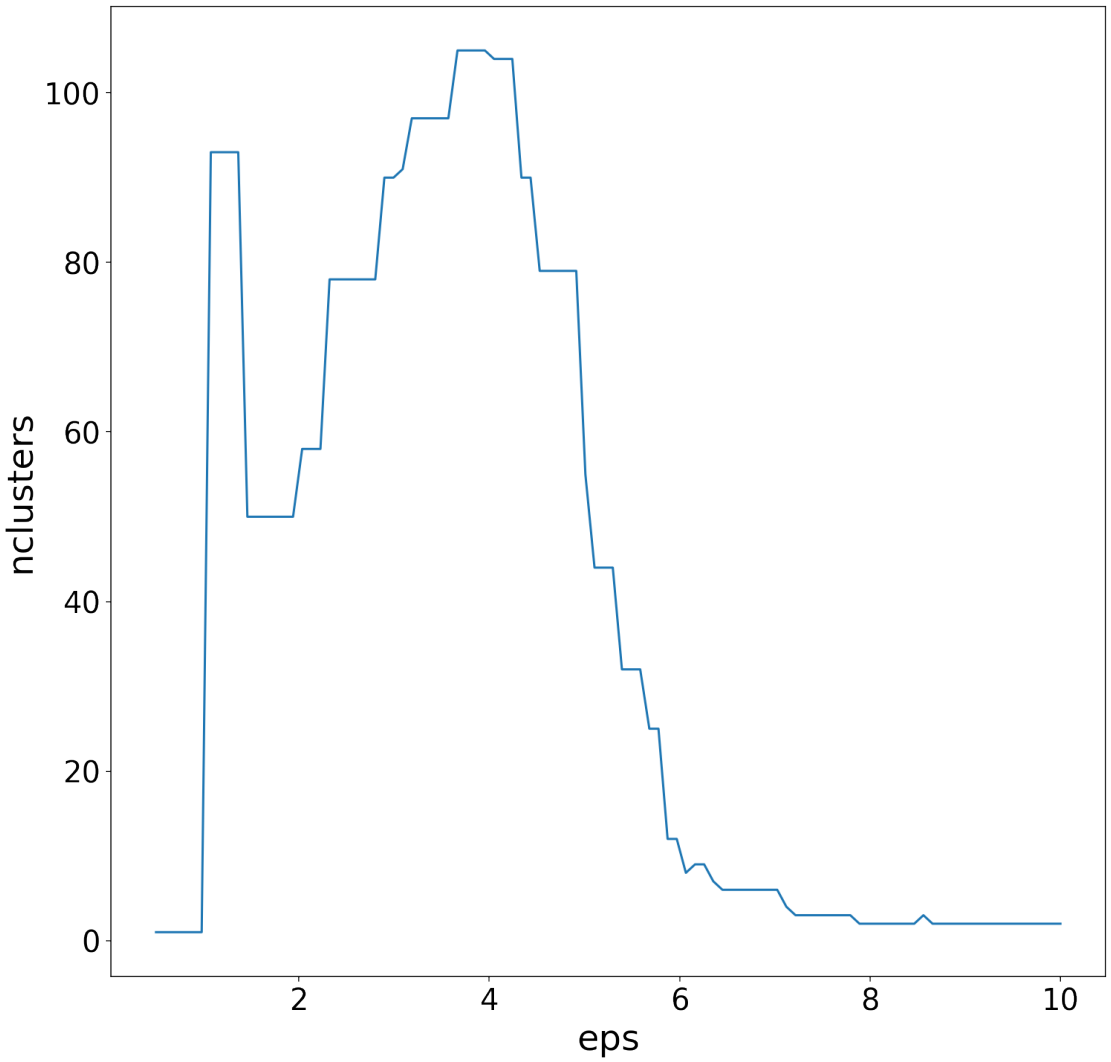



**5.5** **5. Finally, we will use a density-based clustering method called DBSCAN. Similar to part 4, any value in the numpy array that is $> 0$ is considered a catalyst particle. Use DBSCAN clustering to distinguish identify clusters of metal particles (you will need to figure out what a good value of eps is). Plot your clustered image, ensuring that each cluster has a different color. Comment on how you chose your value of eps.**

[45]:
```
from sklearn.cluster import DBSCAN

nclusters = []
for eps in np.linspace(0.5, 10, 100):
    dbscan = DBSCAN(eps=eps).fit(coordinates)
    nclusters.append(len(set(dbscan.labels_)))
```

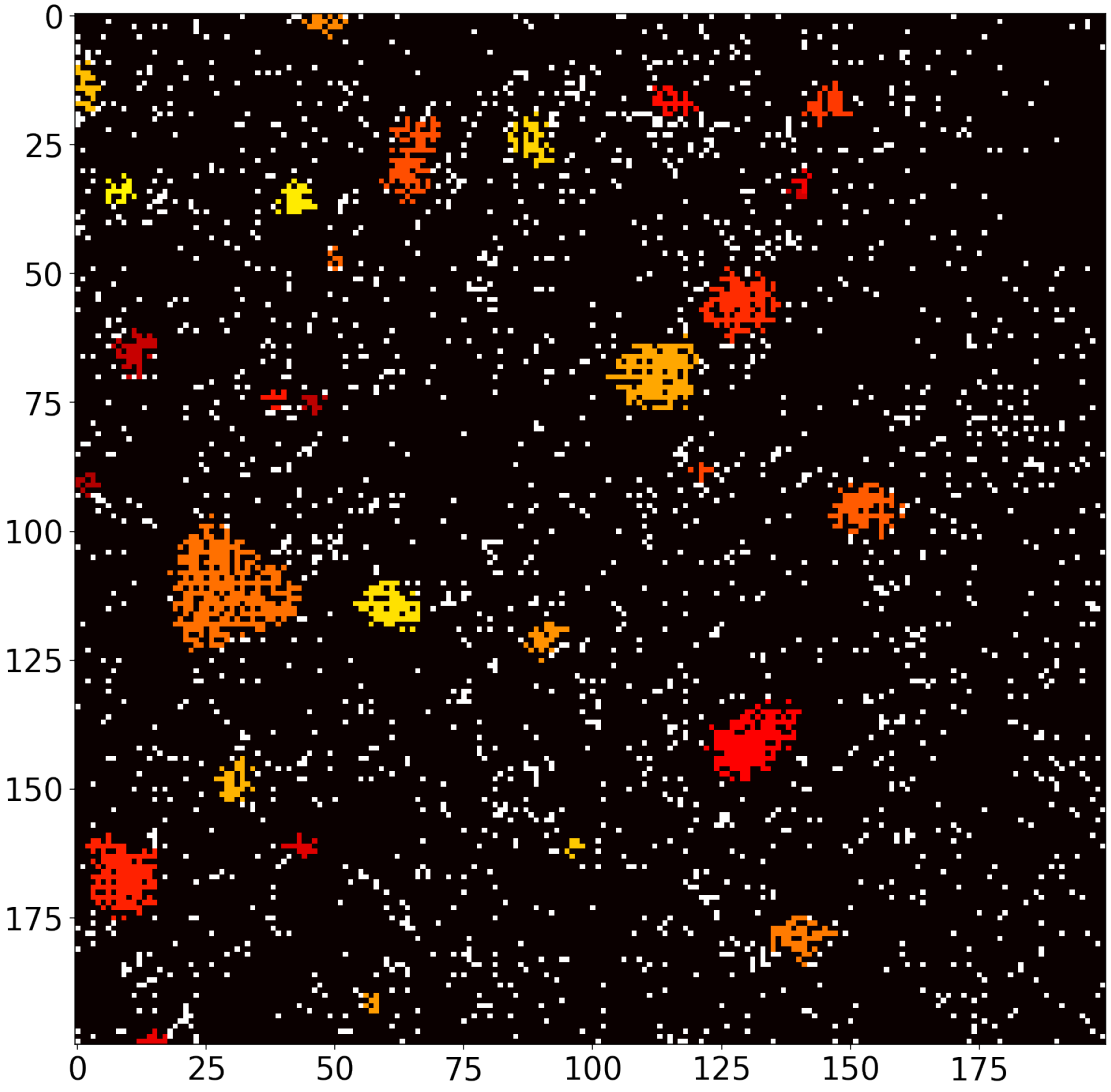
```
f, ax = plt.subplots(figsize=(12, 12))
plt.plot(np.linspace(0.5, 10, 100), nclusters)
plt.xlabel('eps')
plt.ylabel('nclusters')
```

[45]: Text(0, 0.5, 'nclusters')




[46]:
```
new_image = np.zeros(data.shape)
dbscan = DBSCAN(eps=2, min_samples=7).fit(coordinates)
k = max(dbscan.labels_) + 1
shuffled_labels = list(range(k))
random.shuffle(shuffled_labels)
mapping = dict(zip(range(k), shuffled_labels))
for ind, label in zip(coordinates, dbscan.labels_):
```

```
    if label == -1:
        new_image[ind[0], ind[1]] = 1
    else:
        new_image[ind[0], ind[1]] = 0.25 + 0.5 * mapping[label] / k
f, ax = plt.subplots(figsize=(12, 12))
plt.imshow(new_image, cmap=plt.cm.hot);
```




When eps is too small, there are very few clusters. The same applies to the case when eps is too large. Therefore, a good eps value should be somewhere in between. From the nclusters vs eps plot, we can see that when eps ~= 2, the number of clusters reaches a local minimal, which strikes a nice balance between identifying clusters and overfitting random points to clusters (more clusters than necessary).

### 5.6 6. Discuss on the differences between the K-means and DBSCAN results, and which method is more appropriate for the purpose we are using it for.

The k-means method can only find regular shaped clusters since it depends on the geometric distances. It essentially partitions the space into voronoi cells with a given K value. The results of it is not very robust and heavily depend on the choice of K.

The DBSCAN, however, can identify continous shapes as long as the points are densely connected. It is more robust in identifying clusters with general shapes.

In our case, we believe the DBSCAN is more appropriate.

[ ]:

[ ]: