Unsupervised Learning (on Tileset7) - July 2017

Created: 24 July 2018
Last update: 24 july 2018

Try a number of unsupervised learning techniques on a simple data set

The data used here has been pre-labelled in one of my prior notebooks. The labeleing here functions as check, but should not be used for the learning itself (as the goal is to achieve the clustering in unsupervised fashion).

---

1. Imports

In [1]:
```
# this will remove warnings messages
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

%matplotlib inline

# import
from sklearn import cluster
from sklearn.preprocessing import LabelEncoder

from scipy.cluster.hierarchy import linkage, dendrogram

import imgutils
```

In [2]:
```
# Re-run this cell if you altered imgutils
import importlib
importlib.reload(imgutils)
```

Out[2]:
```
<module 'imgutils' from 'C:\\JADS\\SW\\Grad Proj\\realxtals1\\sources\\imgutils.py'>
```

---

2. Import Crystal Image Data & Statistics

The data was labeled and exported to csv in the notebook realxtals1_dataeng1.ipynb

About the data:

The CSV contains the image files, slice information (sub-images) and associated statistics, which are the features for which a classifier needs to be found.

The goal is to find the clustering in feature-space and use those to categorize the images. For this particular dataset, a single statistics could be used to label into three classes:

A = subimage contains no crystal,
B = part of subimage contains crystal,
C = (most of) subimage contains crystal

But the labels have been added here for analyses, eventually the data will be unlabelled.

Import data:

In [224]:
```
df = pd.read_csv('../data/Crystals_Apr_12/Tileset7-2.csv', sep=';')
df.head(3)
```

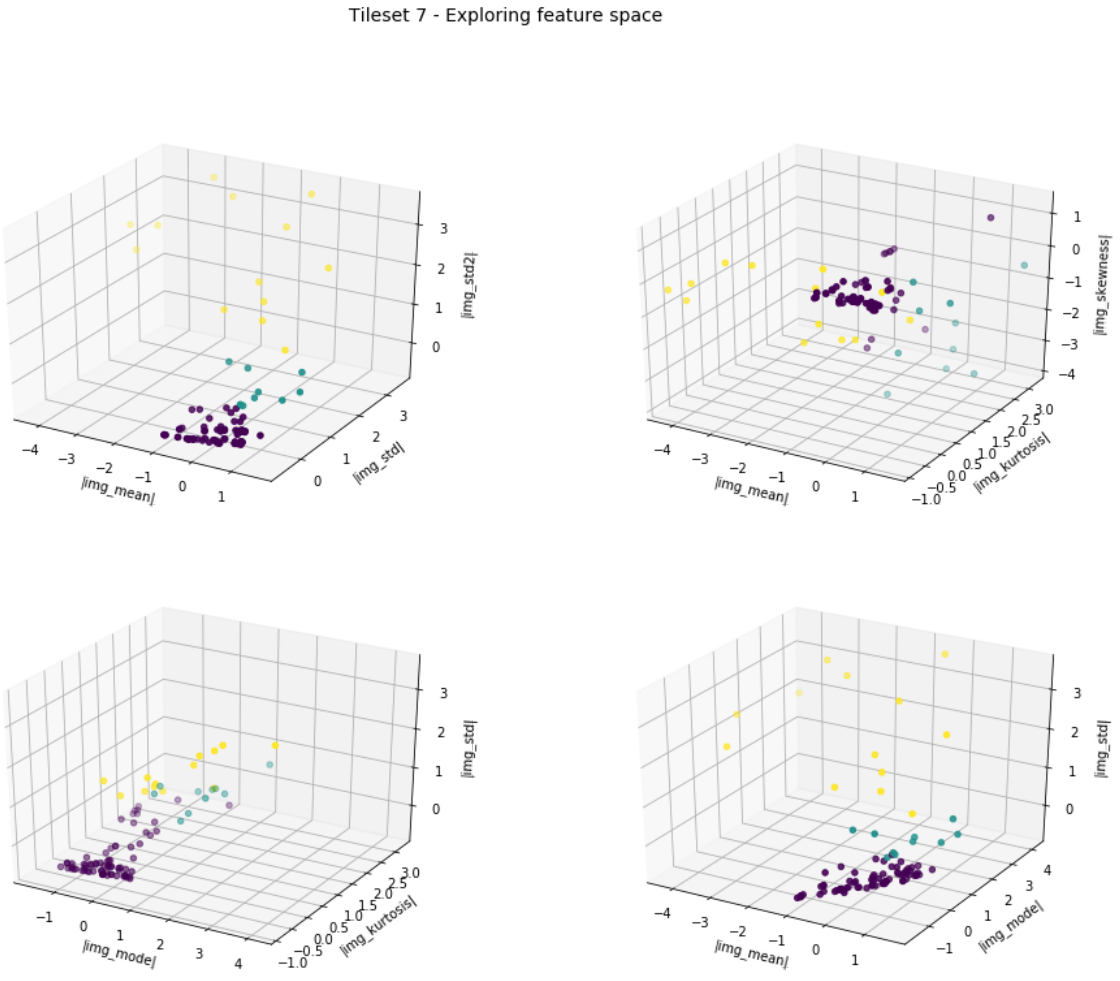
Out[224]:

	Unnamed: 0	filename	s_y	s_x	n_y	n_x	alias	img_mean	img_std	img_kurtosis	img_skewness	im
0	0	..\data\Crystals_Apr_12\Tileset7\Tile_001-001-...	0	0	4	4	img0_0-0	8955.557637	489.754848	4.163737	0.107415	896
1	1	..\data\Crystals_Apr_12\Tileset7\Tile_001-001-...	0	1	4	4	img0_0-1	8883.137305	501.739963	6.528225	-0.146746	892
2	2	..\data\Crystals_Apr_12\Tileset7\Tile_001-001-...	0	2	4	4	img0_0-2	8786.996070	327.512136	1.323241	-0.110828	875

3. Quick visual inspection of the 'feature space'

(Some of this is a repeat from the feature selection notebook)

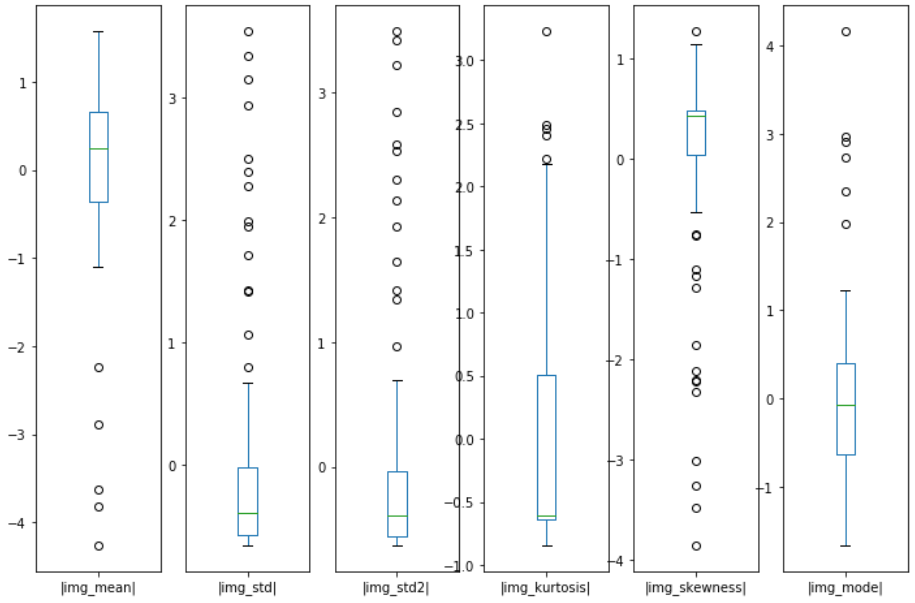
In [4]:
```
# plot it in 3 dimensions, choosing some stat combinations
fig0 = plt.figure(figsize=(16, 12))
plt.suptitle("Tileset 7 - Exploring feature space",fontsize=14)

# trick to convert category labels into color codes
color = pd.DataFrame(df['class'].astype('category'))['class'].cat.codes

# define alias for later reference
org_labels = color

def scatter_3d(ax, df, feat1, feat2, feat3, colors):
    ax.scatter(df[feat1], df[feat2], df[feat3], c=colors)
    ax.set_xlabel(feat1)
    ax.set_ylabel(feat2)
    ax.set_zlabel(feat3)


ax = fig0.add_subplot(221, projection='3d')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', color)

ax = fig0.add_subplot(222, projection='3d')
scatter_3d(ax, df, '|img_mean|', '|img_kurtosis|', '|img_skewness|', color)

ax = fig0.add_subplot(223, projection='3d')
scatter_3d(ax, df, '|img_mode|', '|img_kurtosis|', '|img_std|', color)

ax = fig0.add_subplot(224, projection='3d')
scatter_3d(ax, df, '|img_mean|', '|img_mode|', '|img_std|', color)

plt.show()
```



Let's also make some box plots of the individual features

(practising new skills learned from datacamp)

In [5]:
```
df_subset = df[['|img_mean|','|img_std|', '|img_std2|', '|img_kurtosis|', '|img_skewness|','|img_mode|']]
df_subset.plot(kind='box', subplots=True,figsize=(12, 8))
plt.show()
```



All have many outliers, which could be related to the 'separation' of classes. I need **interactive box plots** that show the images!

(add to TODO list)

---

4. Let's try k-means

First create numbers for classed for better plotting

In [6]:
```
le = LabelEncoder()
df["|class|"] = le.fit_transform(df["class"])
```

First vectorize the data:

In [7]:
```
# convert into X Y vectors:
feature_cols = ['|img_std|', '|img_std2|', '|img_mean|','|img_skewness|', '|img_kurtosis|','|img_mode|']
X = df.loc[:,feature_cols]
```

In [8]:
```
number_of_clusters = 3
```

In [9]:
```
k_means = cluster.KMeans(algorithm='auto', n_clusters=3, n_init=10, init='k-means++')
k_means.fit(X)
```

Out[9]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

In [10]:
```
print(k_means.labels_)
print(k_means.cluster_centers_)
```

```
[0 0 0 0 0 0 2 1 0 0 1 1 0 0 2 1 0 0 0 0 1 2 2 2 1 1 1 2 1 2 0 2 0 0 0 0 0
 2 2 2 2 0 0 0 0 0 0 2 0 0 0 1 0 0 2 2 0 0 2 0 0 2 0 2 1 2 0 0 2 0 0 0 0 0
 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2]
[[-0.46132293 -0.45669419  0.26804287  0.43058697 -0.44714166 -0.25772032]
 [ 2.47924659  2.50485228 -1.75383921  0.03923733 -0.59305226  1.88987724]
 [ 0.08143784  0.05537995  0.10934228 -1.25266317  1.57697724 -0.20692134]]
```

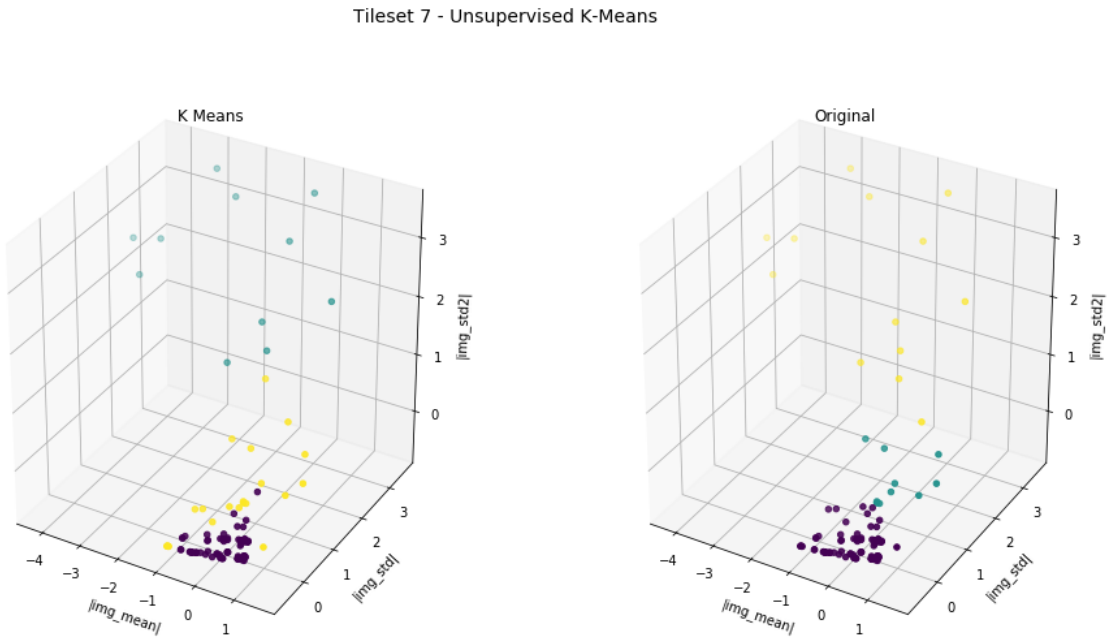
eh... this is multidimensional space. Lets' see if there is a way to visualize this in some way

In [223]:
```
X.head(3)
```

Out[223]:

	|img_std|	|img_std2|	|img_mean|	|img_skewness|	|img_kurtosis|	|img_mode|
0	-0.100998	-0.122335	0.615795	0.471479	0.082333	0.204925
1	-0.079451	-0.095243	0.289248	0.090672	0.504401	0.006384
2	-0.392677	-0.385181	-0.144258	0.144487	-0.424704	-0.766354

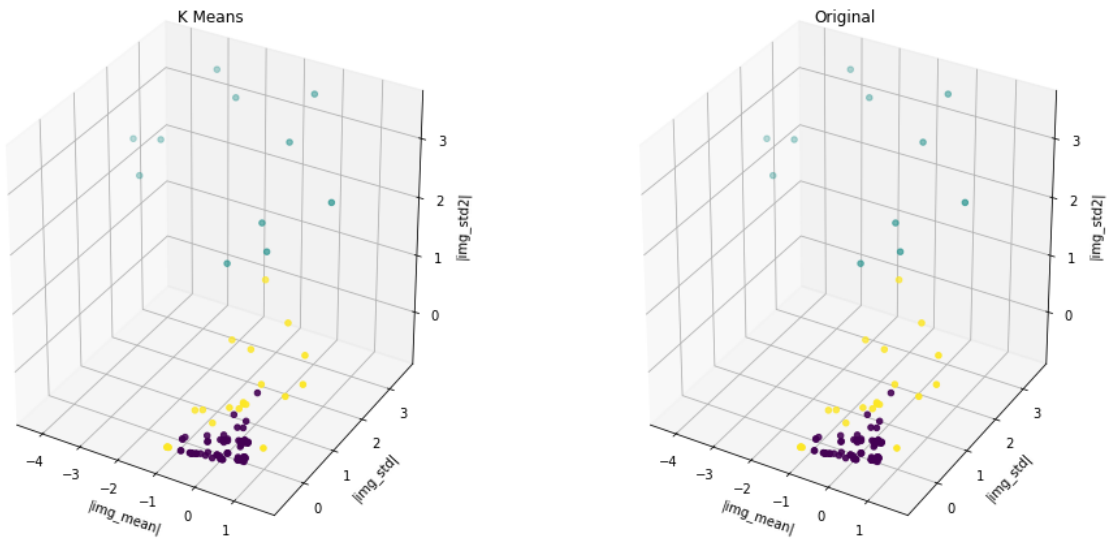
In [12]:
```
# plotting first three dimensions (i.e. std, std2 and mean) the k-means and the original labels
fig0 = plt.figure(figsize=(16, 8))
plt.suptitle("Tileset 7 - Unsupervised K-Means ",fontsize=14)

ax = fig0.add_subplot(121, projection='3d', title='K Means')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', k_means.labels_)

ax = fig0.add_subplot(122, projection='3d', title='Original')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', color)
```



Not perfect (some mistakes), but not that bad! And that for a first attempt.

Is there a difference between the assigned label and using predict on the trainng data? Let's check it

In [13]:
```
# combine into one dataframe:
df2 = pd.concat([df, pd.Series(k_means.labels_)], axis=1)
df2 = df2.rename(columns = { 0 : 'k_means'})
df2['k_means_predict'] = k_means.predict(X)

# plotting first three dimensions (i.e. std, std2 and mean) the k-means and the original labels
fig0 = plt.figure(figsize=(16, 8))
plt.suptitle("Tileset 7 - Unsupervised K-Means - Labels vs Predict",fontsize=14)

ax = fig0.add_subplot(121, projection='3d', title='K Means')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', colors=df2['k_means'])

ax = fig0.add_subplot(122, projection='3d', title='Original')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', colors=df2['k_means_predict'])
```



Yes, it's the same (pfew!)

---

5. Try DBScan

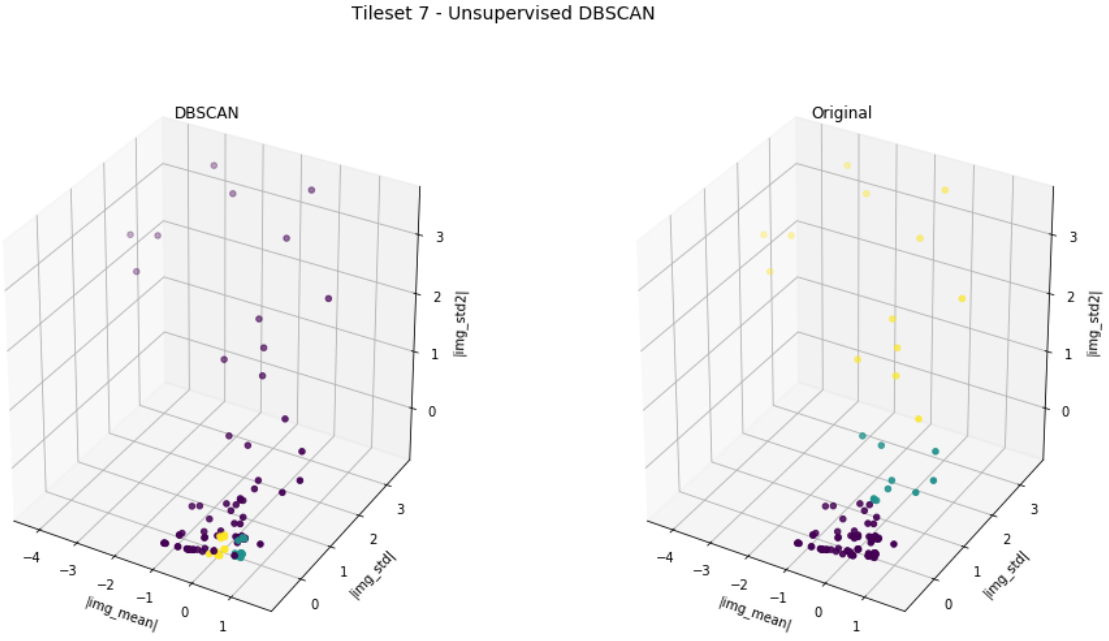
In [14]:
```
dbscan = cluster.DBSCAN(eps=0.3, min_samples=10).fit(X)
dbscan.fit(X)
```

Out[14]:
```
DBSCAN(algorithm='auto', eps=0.3, leaf_size=30, metric='euclidean',
    metric_params=None, min_samples=10, n_jobs=1, p=None)
```

In [15]:
```
print(dbscan.labels_)
```

```
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1  0  1 -1  1 -1 -1 -1
 -1  1 -1 -1  0  0 -1 -1  0 -1 -1 -1  1 -1 -1 -1 -1 -1  1 -1 -1  0  1 -1
  0 -1  1 -1  1 -1 -1 -1  0  1 -1 -1  0  0  1 -1  0 -1  1 -1  1 -1 -1 -1]
```

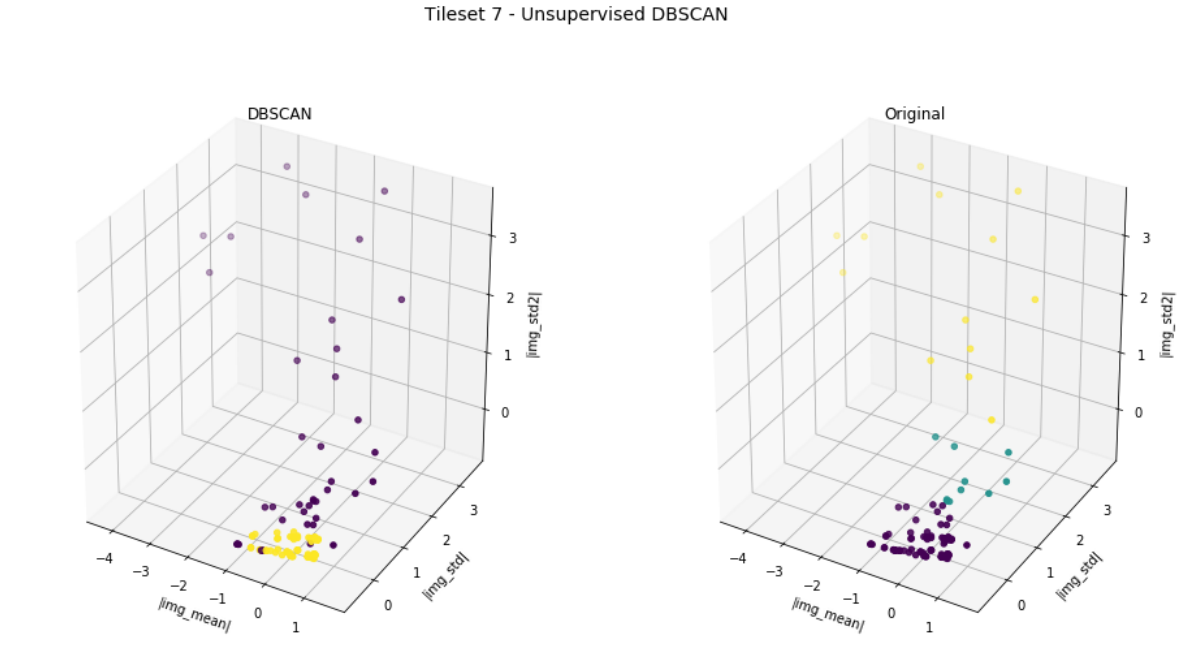
In [16]:
```
# plotting first three dimensions (i.e. std, std2 and mean) the k-means and the original labels
fig0 = plt.figure(figsize=(16, 8))
plt.suptitle("Tileset 7 - Unsupervised DBSCAN ",fontsize=14)

ax = fig0.add_subplot(121, projection='3d', title='DBSCAN')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', dbscan.labels_)

ax = fig0.add_subplot(122, projection='3d', title='Original')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', color)
```



that looks like crap. Let's try with other parameters

In [17]:
```
dbscan = cluster.DBSCAN(eps=0.5, metric='euclidean', min_samples=8)
dbscan.fit(X)

print("Number of clusters: " + str(len(np.unique(dbscan.labels_))))

fig0 = plt.figure(figsize=(16, 8))
plt.suptitle("Tileset 7 - Unsupervised DBSCAN ",fontsize=14)

ax = fig0.add_subplot(121, projection='3d', title='DBSCAN')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', dbscan.labels_)

ax = fig0.add_subplot(122, projection='3d', title='Original')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', color)
```

```
Number of clusters: 2
```



Too many hyper parameters that have a lot of impact on the outcome (which is **poor** in almost anycase)

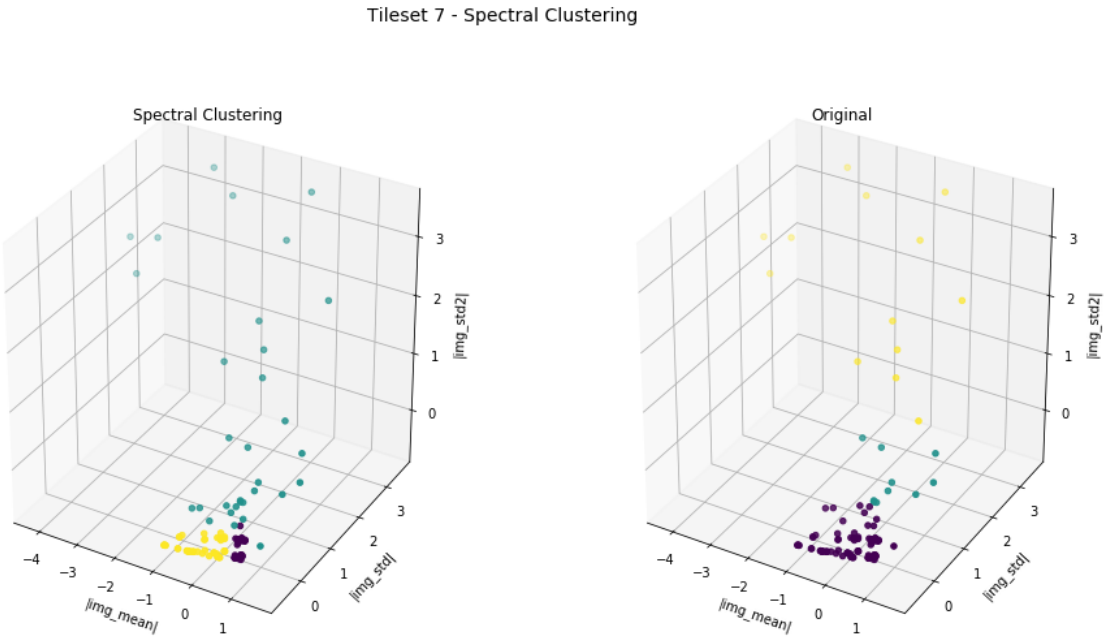
---

6. Spectral Clustering

In [18]:
```
spectral = cluster.SpectralClustering(n_clusters=number_of_clusters,eigen_solver='arpack',affinity="nearest_neighbors")
spectral.fit(X)
print(spectral.labels_)
```

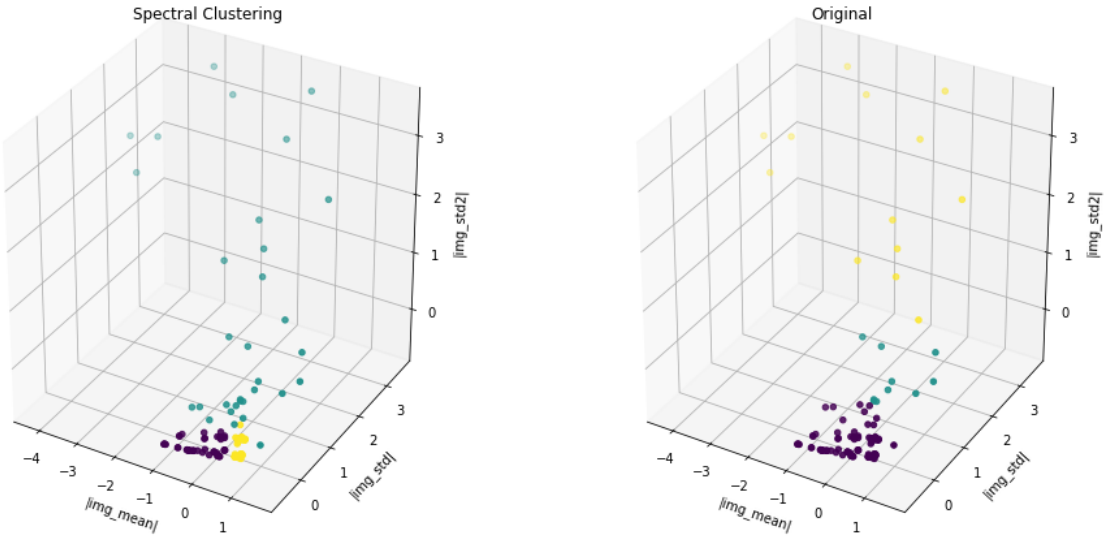
```
[1 1 2 2 0 0 1 1 0 0 1 1 2 2 1 1 0 2 2 2 1 1 1 1 1 1 1 1 1 2 2 0 2 2 2 1
 1 1 1 1 0 2 2 2 2 2 2 0 2 0 1 0 0 1 1 0 1 1 2 2 1 2 2 1 1 2 2 1 0 2 2 0 0
 2 2 2 2 2 2 0 2 2 2 0 0 2 2 0 0 2 2 2 2 2 2]
```

In [19]:
```
# plotting first three dimensions (i.e. std, std2 and mean) the k-means and the original labels
fig0 = plt.figure(figsize=(16, 8))
plt.suptitle("Tileset 7 - Spectral Clustering ",fontsize=14)

ax = fig0.add_subplot(121, projection='3d', title='Spectral Clustering')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', spectral.labels_)

ax = fig0.add_subplot(122, projection='3d', title='Original')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', color)
```



hmm, what can we say... Let's try other parametrization

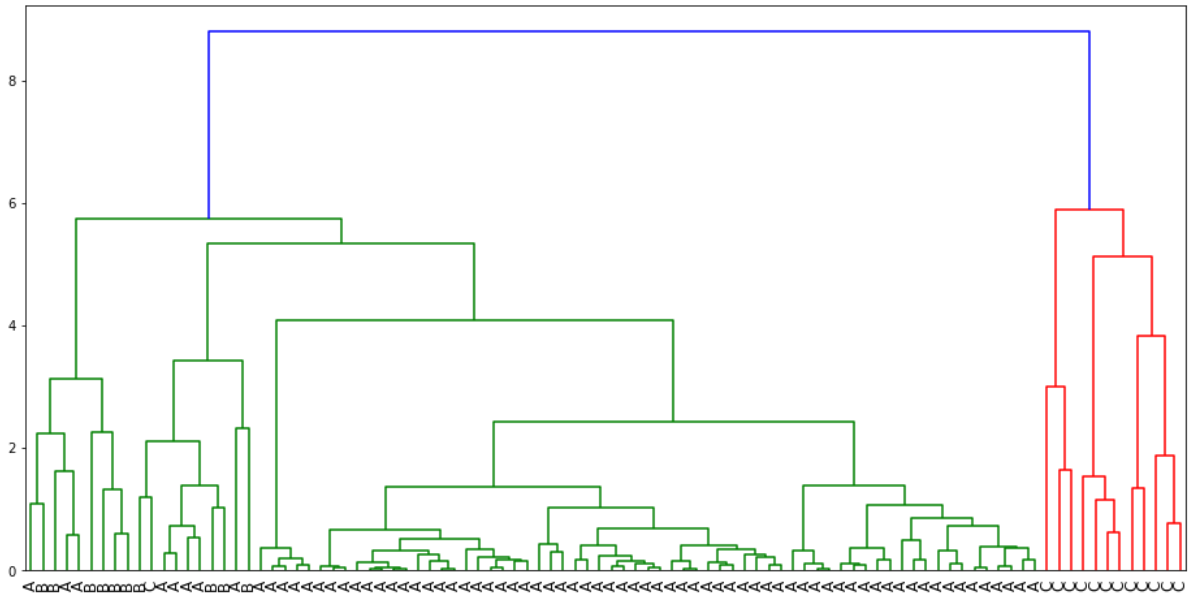
In [20]:
```
spectral = cluster.SpectralClustering(n_clusters=number_of_clusters,eigen_solver='arpack',affinity="nearest_neighbors",
                                      n_init=10)
spectral.fit(X)
fig0 = plt.figure(figsize=(16, 8))
plt.suptitle("Tileset 7 - Spectral Clustering ",fontsize=14)

ax = fig0.add_subplot(121, projection='3d', title='Spectral Clustering')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', spectral.labels_)

ax = fig0.add_subplot(122, projection='3d', title='Original')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', color)
```



7. Try Hierarchical Clustering

(see e.g. https://towardsdatascience.com/unsupervised-learning-with-python-173c51dc7f03 (https://towardsdatascience.com/unsupervised-learning-with-python-173c51dc7f03))

In [21]:
```
hierarchical = linkage(X.values, method='complete')
labels = df['class'].tolist()

# Plot a so calles 'dendrogram'
fig = plt.figure(figsize=(16, 8))
dendrogram(hierarchical,
           labels=labels,
           leaf_rotation=90,
           leaf_font_size=12,
           )

plt.show()
```



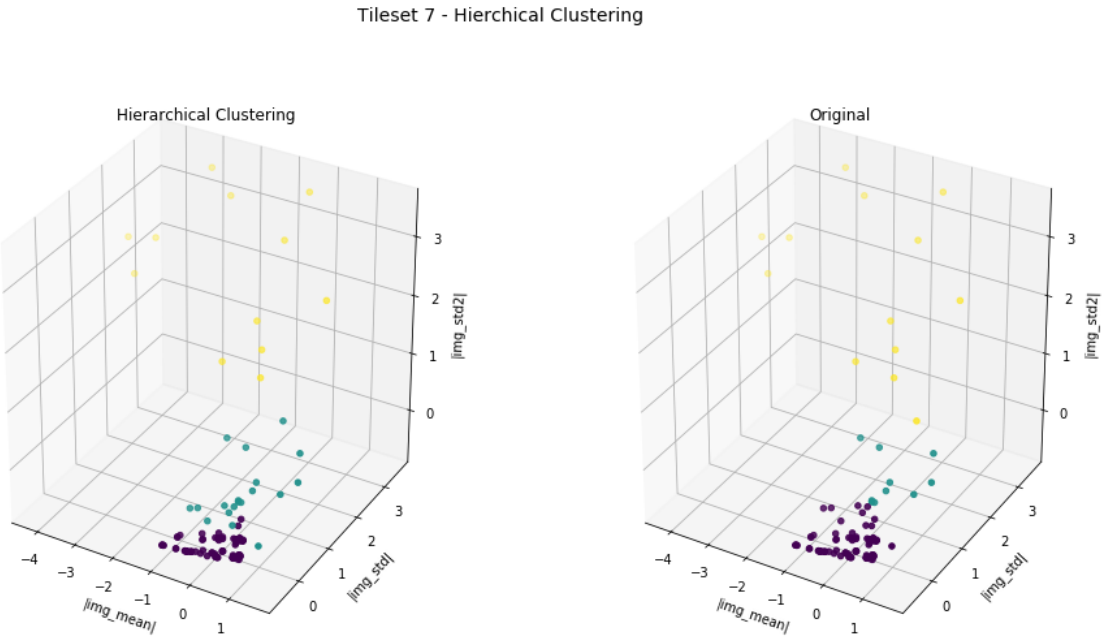
C is clearly separted (= with crystal), A mosty (no crystal), but B (partial) is more fuzzy. Lets plot it

Unclear however how to git this into a more suitable form to compare it with the original lables. Let's try the hierarchical clustering from sklearn

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

# Affinity = {“euclidean”, “l1”, “l2”, “manhattan”, “cosine”}
# Linkage = {“ward”, “complete”, “average”}

Hclustering = AgglomerativeClustering(n_clusters=3, affinity='cosine', linkage='complete')
Hclustering.fit(X)
```

Out[22]:
```
AgglomerativeClustering(affinity='cosine', compute_full_tree='auto',
            connectivity=None, linkage='complete', memory=None,
            n_clusters=3,
            pooling_func=<function mean at 0x0000000005C0B8C8>)
```

In [23]:
```
# combine into one dataframe:
df2 = pd.concat([df, pd.Series(Hclustering.labels_)], axis=1)
df2 = df2.rename(columns = { 0 : 'hc'})

fig0 = plt.figure(figsize=(16, 8))
plt.suptitle("Tileset 7 - Hierchical Clustering ",fontsize=14)

ax = fig0.add_subplot(121, projection='3d', title='Hierarchical Clustering')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', Hclustering.labels_)

ax = fig0.add_subplot(122, projection='3d', title='Original')
scatter_3d(ax, df, '|img_mean|', '|img_std|', '|img_std2|', org_labels)
```



not bad (I tried a few combinations; 'complete' or 'average' with the 'cosine' metric gave best results)

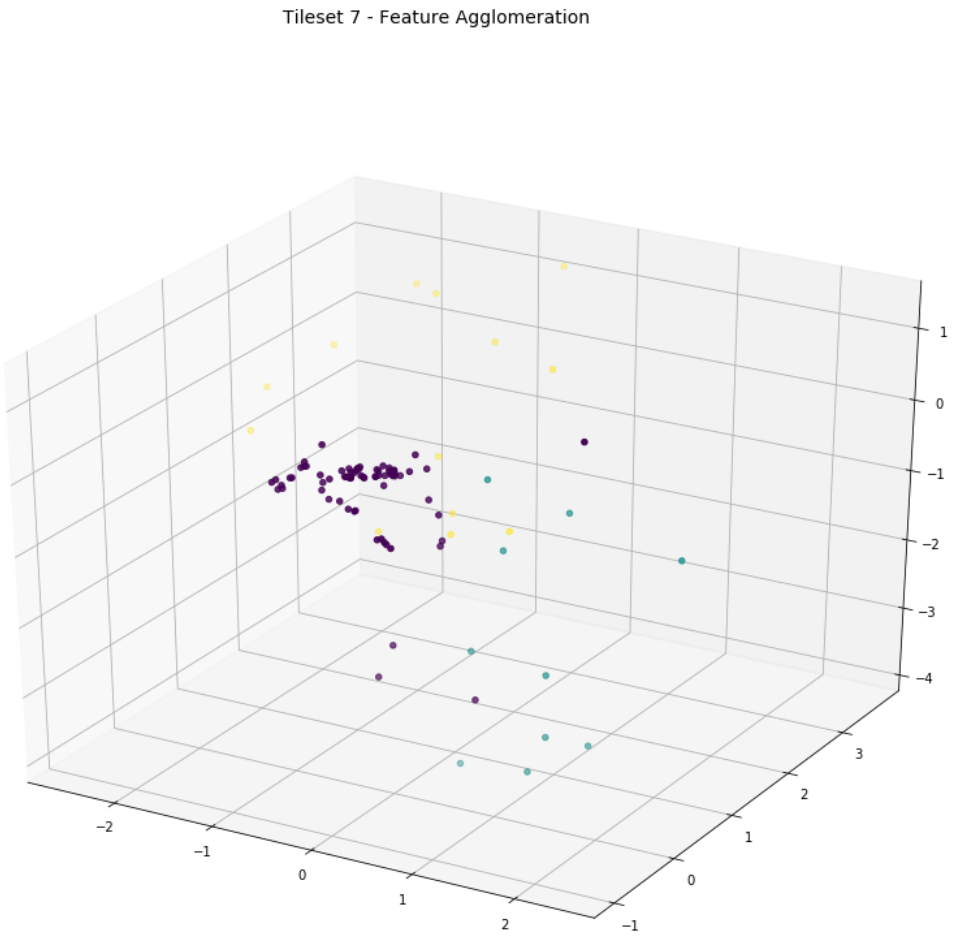
another one to try is 'feature aggoleration', which is combining hierarchical clustering with dimensionality reduction

In [24]:
```
from sklearn.cluster import FeatureAgglomeration
agglo=FeatureAgglomeration(n_clusters=3).fit_transform(X)
aggloX=agglo[:,0]
aggloY=agglo[:,1]
print(aggloX.shape, aggloY.shape)
```

```
(96,) (96,)
```

In [25]:
```
# plot it
fig0 = plt.figure(figsize=(16, 12))
ax = fig0.add_subplot(111, projection='3d')
plt.suptitle("Tileset 7 - Feature Agglomeration",fontsize=14)
ax.scatter(agglo[:,0], agglo[:,1], agglo[:,2], c=org_labels)
```

Out[25]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0xd6857b8>
```



Actually, I do not understand this very well.

---

8. Comparing results

Need a way to compare the results more quantively. Hard part is that the labels are different. As a first step, maybe just list the mean and count of the clusters

In [26]:
```
df2 = pd.concat([df, pd.Series(k_means.labels_)], axis=1)
df2 = df2.rename(columns = { 0 : 'k_means'})
df2.head(1)
print(df2.groupby("class")[['|img_mean|', '|img_std|', '|img_std2|']].count())
print(df2.groupby("k_means")[['|img_mean|', '|img_std|', '|img_std2|']].count())
print(df2.groupby("class")[['|img_mean|', '|img_std|', '|img_std2|']].mean())
print(df2.groupby("k_means")[['|img_mean|', '|img_std|', '|img_std2|']].mean())
```

```
       |img_mean|  |img_std|  |img_std2|
class
A              73         73          73
B              10         10          10
C              13         13          13
         |img_mean|  |img_std|  |img_std2|
k_means
0                63         63          63
1                11         11          11
2                22         22          22
       |img_mean|  |img_std|  |img_std2|
class
A        0.164528  -0.458595   -0.452851
B        0.662461   0.370912    0.318567
C       -1.433476   2.289869    2.297881
         |img_mean|  |img_std|  |img_std2|
k_means
0          0.268043  -0.461323   -0.456694
1         -1.753839   2.479247    2.504852
2          0.109342   0.081438    0.055380
```

- category A maps on cluster 1 of k-means
- category B maps on cluster 0 of k-means (I think)
- category C maps on cluster 2 of k-means (I think)

Probably visualizing in the form of the heatmap is clearer to see it's effect. (as I cannot think of a way now how to properly quantify this)

From SkLearn docs:

http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation (http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation)

2.3.9. Clustering performance evaluation

Evaluating the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall of a supervised classification algorithm. In particular any evaluation metric should not take the absolute values of the cluster labels into account but rather if this clustering define separations of the data similar to some ground truth set of classes or satisfying some assumption such that members belong to the same class are more similar that members of different classes according to some similarity metric.

Many ways of scoring exist. Most point to the Adjusted Rand Index (ARI) as a good one. Let's try it

In [27]:
```
from sklearn import metrics

k_means_pred = k_means.labels_
spectral_pred = spectral.labels_
dbscan_pred = dbscan.labels_
hierarch_pred = Hclustering.labels_

print('Adjusted Rand Index scoring:')
print("k-means: %f" % metrics.adjusted_rand_score(org_labels, k_means_pred))
print("spectral: %f" % metrics.adjusted_rand_score(org_labels, spectral_pred))
print("dbscan: %f" % metrics.adjusted_rand_score(org_labels, dbscan_pred))
print("hierarchical: %f" % metrics.adjusted_rand_score(org_labels, hierarch_pred))
```

```
Adjusted Rand Index scoring:
k-means: 0.600606
spectral: 0.257207
dbscan: 0.276250
hierarchical: 0.722755
```

According to the ARI score, **hierarchical clustering worked best** on this dataset.

Also try some of the other scoring methods:

In [28]:
```
scorefunc =  metrics.adjusted_mutual_info_score
print('Mutual Information Scoring:')

print("k-means: %f" % scorefunc(org_labels, k_means_pred))
print("spectral: %f" % scorefunc(org_labels, spectral_pred))
print("dbscan: %f" % scorefunc(org_labels, dbscan_pred))
print("hierarchical: %f" % scorefunc(org_labels, hierarch_pred))
```

```
Mutual Information Scoring:
k-means: 0.505779
spectral: 0.320783
dbscan: 0.327220
hierarchical: 0.634900
```

In [29]:
```
scorefunc =  metrics.homogeneity_score
print('Homogeneity Scoring:')

print("k-means: %f" % scorefunc(org_labels, k_means_pred))
print("spectral: %f" % scorefunc(org_labels, spectral_pred))
print("dbscan: %f" % scorefunc(org_labels, dbscan_pred))
print("hierarchical: %f" % scorefunc(org_labels, hierarch_pred))
```

```
Homogeneity Scoring:
k-means: 0.626843
spectral: 0.493346
dbscan: 0.337546
hierarchical: 0.762652
```

In [30]:
```
scorefunc =  metrics.completeness_score
print('Completeness Scoring:')

print("k-means: %f" % scorefunc(org_labels, k_means_pred))
print("spectral: %f" % scorefunc(org_labels, spectral_pred))
print("dbscan: %f" % scorefunc(org_labels, dbscan_pred))
print("hierarchical: %f" % scorefunc(org_labels, hierarch_pred))
```

```
Completeness Scoring:
k-means: 0.519494
spectral: 0.335585
dbscan: 0.350754
hierarchical: 0.645305
```

All these scoring metrics agree: hierarchical gave best results, followed by k-means

---

9. Visualize unsupervised result as heatmap

The numbers are still hard to interpret how good/bad it is. We need to visually check the result in the context of the actual image.

In [186]:
```
# Add the unsupervised clustering results to the dataframe
df3 = df
df3 = pd.concat([df3, pd.Series(k_means_pred).rename('k_means')], axis=1)
df3 = pd.concat([df3, pd.Series(dbscan_pred).rename('dbscan')], axis=1)
df3 = pd.concat([df3, pd.Series(spectral_pred).rename('spectral')], axis=1)
df3 = pd.concat([df3, pd.Series(hierarch_pred).rename('hierarch')], axis=1)

df3.head(1)
```

Out[186]:

	Unnamed: 0	filename	s_y	s_x	n_y	n_x	alias	img_mean	img_std	img_kurtosis	...	|img_kurtosis|	|i
0	0	..\data\Crystals_Apr_12\Tileset7\Tile_001-001-...	0	0	4	4	img0_0-0	8955.557637	489.754848	4.163737	...	0.082333	0

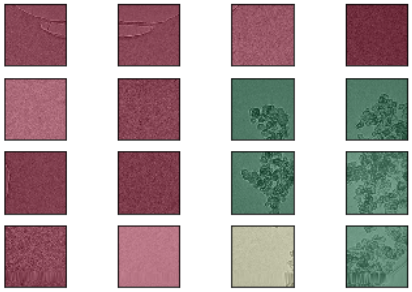
1 rows × 25 columns

In [228]:
```
%matplotlib inline

# see https://matplotlib.org/examples/color/colormaps_reference.html
colmap = 'RdYlGn'
opac = 0.4
figsize=(6,4)

"""Show heatmaps of all images using the specified column as heats"""
def show_heatmaps_allimgs(df_imgstats, heatcolname):
    print('Heats from: ' + heatcolname)
    imgnames = df_imgstats['filename'].unique()
    for imgname in imgnames:
        subimgs, heats = imgutils.getimgslices_fromdf(df_imgstats, imgname, heatcolname)
        #rescale the heats to [0-1] range:
        heats = (heats - np.min(heats)) / (np.max(heats)-np.min(heats))
        print(imgname + ': ' + heatcolname)
        imgutils.showheatmap(subimgs, heats, heatdepend_opacity = False, opacity=opac, cmapname=colmap, title='image: ' + img
name, figsize=figsize)
        print(heats)

"""Show multiple heatmaps of one image using different heats (as specified in heatcolnames"""
def show_heatmap_multistats(df_imgstats, imgname, heatcolnames):
    print('Image: ' + imgname)
    for colname in heatcolnames:
        subimgs, heats = imgutils.getimgslices_fromdf(df_imgstats, imgname, colname)
        heats = (heats - np.min(heats)) / (np.max(heats)-np.min(heats))
        imgutils.showheatmap(subimgs, heats, heatdepend_opacity = False, opacity=opac,  cmapname=colmap, title='Heats from: '
 + colname, figsize=figsize)
        print(heats)
```

In [229]:
```
# show all the techniques on first image
imgnames = df3['filename'].unique()
show_heatmap_multistats(df3, imgnames[0], ['|class|', 'hierarch', 'k_means', 'dbscan', 'spectral'])
```

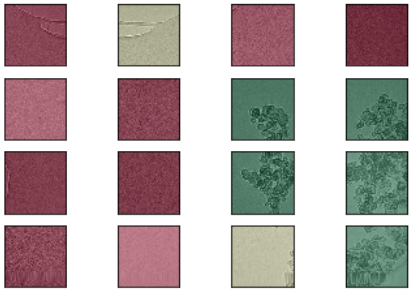
```
Image: ..\data\Crystals_Apr_12\Tileset7\Tile_001-001-000_0-000.tif
```

Heats from: |class|



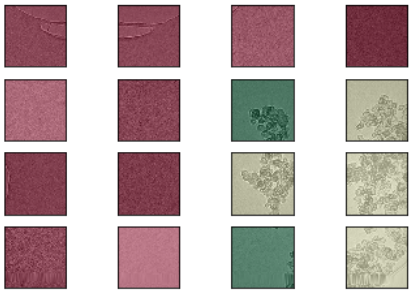
```
[[0.  0.  0.  0. ]
 [0.  0.  1.  1. ]
 [0.  0.  1.  1. ]
 [0.  0.  0.5 1. ]]
```

Heats from: hierarch



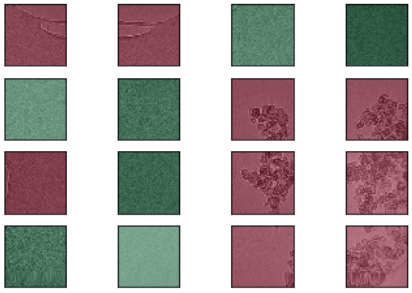
```
[[0.  0.5 0.  0. ]
 [0.  0.  1.  1. ]
 [0.  0.  1.  1. ]
 [0.  0.  0.5 1. ]]
```

Heats from: k_means

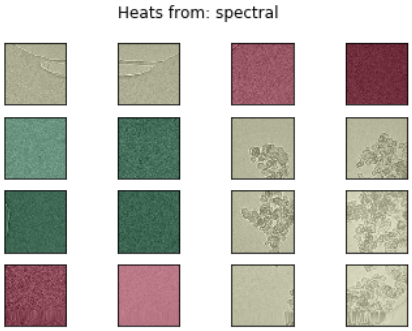


```
[[0.  0.  0.  0. ]
 [0.  0.  1.  0.5]
 [0.  0.  0.5 0.5]
 [0.  0.  1.  0.5]]
```

Heats from: dbscan



```
[[0. 0. 1. 1.]
 [1. 1. 0. 0.]
 [0. 1. 0. 0.]
 [1. 1. 0. 0.]]
```

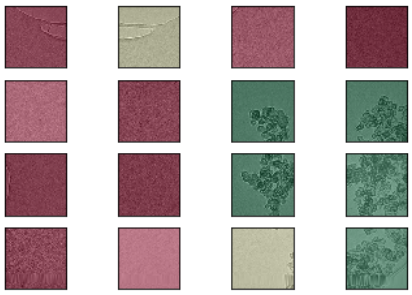



```
[[0.5 0.5 0.  0. ]
 [1.  1.  0.5 0.5]
 [1.  1.  0.5 0.5]
 [0.  0.  0.5 0.5]]
```

In [230]:
```
# show only hierarchical clustering on all images
show_heatmaps_allimgs(df3, 'hierarch')
```

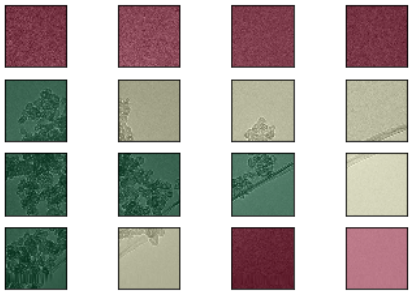
```
Heats from: hierarch
..\data\Crystals_Apr_12\Tileset7\Tile_001-001-000_0-000.tif: hierarch
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_001-001-000_0-000.tif



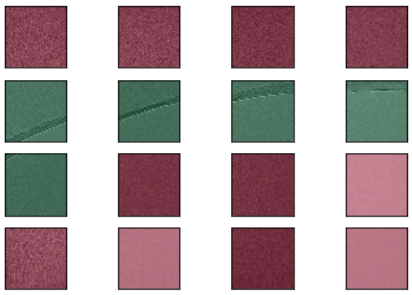
```
[[0.  0.5 0.  0. ]
 [0.  0.  1.  1. ]
 [0.  0.  1.  1. ]
 [0.  0.  0.5 1. ]]
..\data\Crystals_Apr_12\Tileset7\Tile_001-002-000_0-000.tif: hierarch
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_001-002-000_0-000.tif



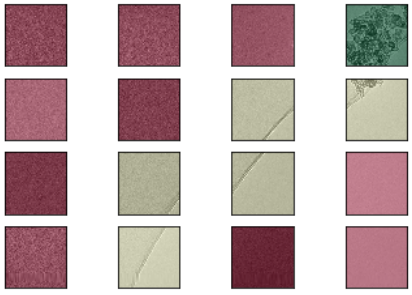
```
[[0.  0.  0.  0. ]
 [1.  0.5 0.5 0.5]
 [1.  1.  1.  0.5]
 [1.  0.5 0.  0. ]]
..\data\Crystals_Apr_12\Tileset7\Tile_001-003-000_0-000.tif: hierarch
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_001-003-000_0-000.tif



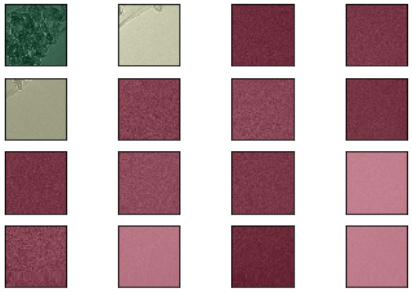
```
[[0. 0. 0. 0.]
 [1. 1. 1. 1.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]]
..\data\Crystals_Apr_12\Tileset7\Tile_002-001-000_0-000.tif: hierarch
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_002-001-000_0-000.tif



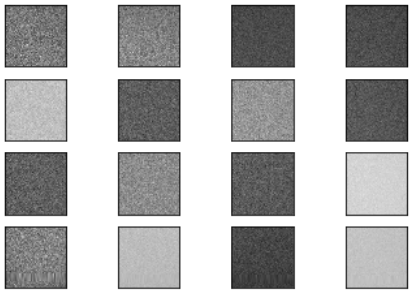
```
[[0.  0.  0.  1. ]
 [0.  0.  0.5 0.5]
 [0.  0.5 0.5 0. ]
 [0.  0.5 0.  0. ]]
..\data\Crystals_Apr_12\Tileset7\Tile_002-002-000_0-000.tif: hierarch
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_002-002-000_0-000.tif



```
[[1.  0.5 0.  0. ]
 [0.5 0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  0. ]]
..\data\Crystals_Apr_12\Tileset7\Tile_002-003-000_0-000.tif: hierarch
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_002-003-000_0-000.tif

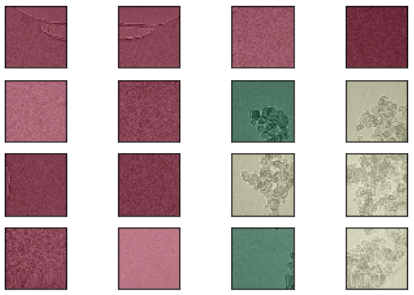


```
[[nan nan nan nan]
 [nan nan nan nan]
 [nan nan nan nan]
 [nan nan nan nan]]
```

In [231]:
```
# show only k-means clustering on all images
show_heatmaps_allimgs(df3, 'k_means')
```

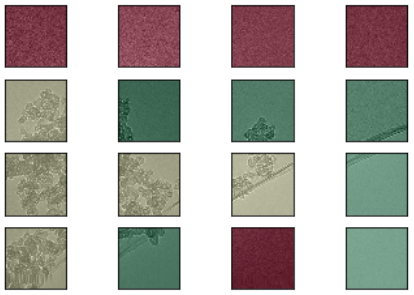
```
Heats from: k_means
..\data\Crystals_Apr_12\Tileset7\Tile_001-001-000_0-000.tif: k_means
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_001-001-000_0-000.tif



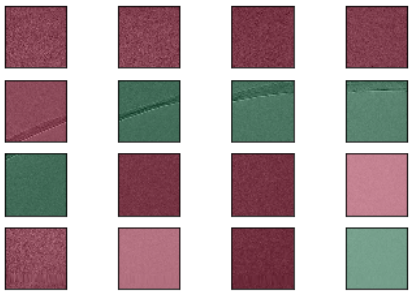
```
[[0.  0.  0.  0. ]
 [0.  0.  1.  0.5]
 [0.  0.  0.5 0.5]
 [0.  0.  1.  0.5]]
..\data\Crystals_Apr_12\Tileset7\Tile_001-002-000_0-000.tif: k_means
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_001-002-000_0-000.tif



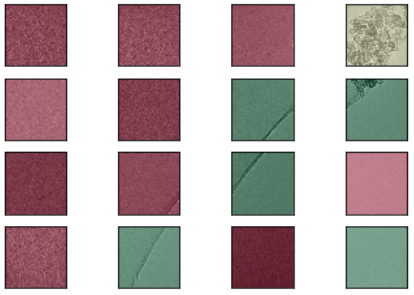
```
[[0.  0.  0.  0. ]
 [0.5 1.  1.  1. ]
 [0.5 0.5 0.5 1. ]
 [0.5 1.  0.  1. ]]
..\data\Crystals_Apr_12\Tileset7\Tile_001-003-000_0-000.tif: k_means
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_001-003-000_0-000.tif



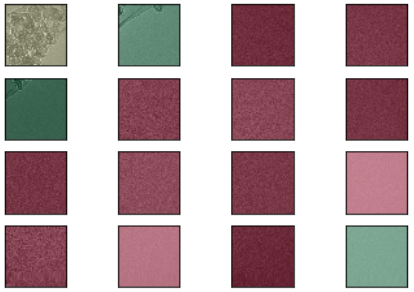
```
[[0. 0. 0. 0.]
 [0. 1. 1. 1.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]]
..\data\Crystals_Apr_12\Tileset7\Tile_002-001-000_0-000.tif: k_means
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_002-001-000_0-000.tif



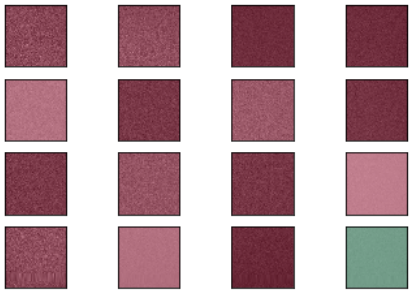
```
[[0.  0.  0.  0.5]
 [0.  0.  1.  1. ]
 [0.  0.  1.  0. ]
 [0.  1.  0.  1. ]]
..\data\Crystals_Apr_12\Tileset7\Tile_002-002-000_0-000.tif: k_means
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_002-002-000_0-000.tif



```
[[0.5 1.  0.  0. ]
 [1.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.  0.  0.  1. ]]
..\data\Crystals_Apr_12\Tileset7\Tile_002-003-000_0-000.tif: k_means
```

image: ..\data\Crystals_Apr_12\Tileset7\Tile_002-003-000_0-000.tif



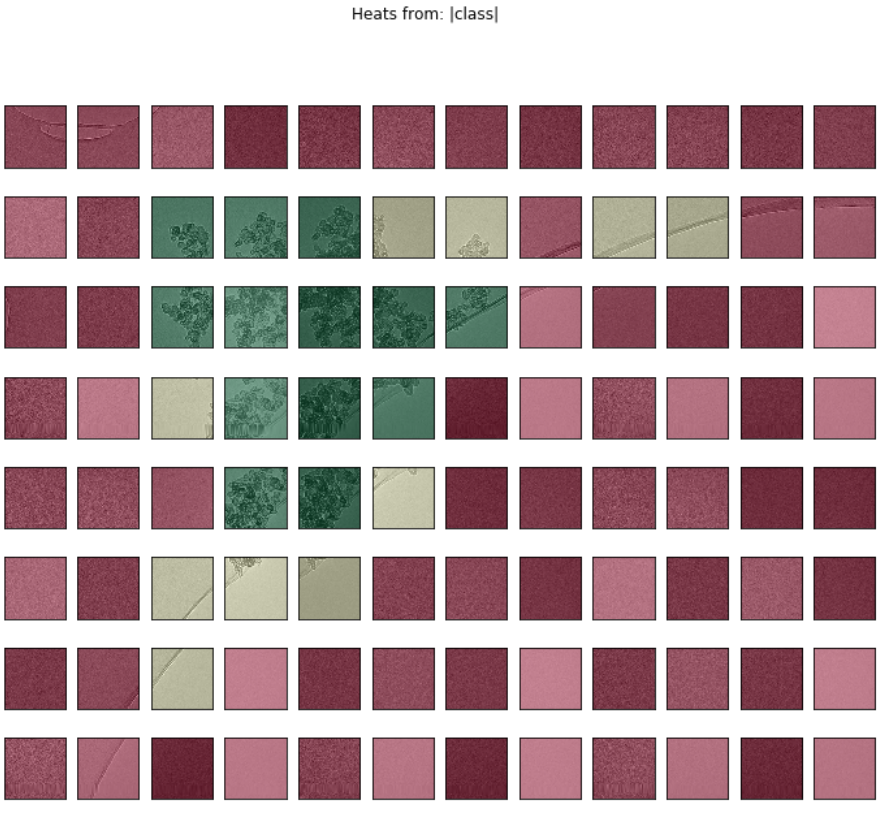
```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 1.]]
```

With the visualization, it looked not that well.

Need to manually count the number of positives, and false positives / false negatives to get a metric for this first unsupervised attempt. And see if I can combine multiple images in one view (makes the manual counting easier)

In [191]:
```
"""Show heatmaps of all images as one large image"""
def show_large_heatmap(df_imgstats, heatcolname, imgnames, n_rows, n_cols, show_extra_info=False):

    assert len(imgnames) == n_rows * n_cols

    # use first image to get the number of subimages per image
    df_img1 = df_imgstats.loc[df['filename'] == imgnames[0]]
    n_y = df_img1.iloc[0]['n_y']
    n_x = df_img1.iloc[0]['n_x']

    # grab all subimgs and heats into one large 2d array
    i = 0
    allsubimgs = np.empty((n_rows*n_y, n_cols*n_x), dtype=object)
    allheats = np.empty((n_rows*n_y, n_cols*n_x), dtype=float)
    for row in range(0,n_rows):
        for col in range(0,n_cols):
            imgname = imgnames[i]
            subimgs, heats = imgutils.getimgslices_fromdf(df_imgstats, imgname, heatcolname)
            for sub_row in range(0,n_y):
                for sub_col in range(0,n_x):
                    all_row = row * n_y + sub_row
                    all_col = col * n_x + sub_col
                    allsubimgs[all_row, all_col] = subimgs[sub_row, sub_col]
                    allheats[all_row, all_col] = heats[sub_row, sub_col]

            #print(heats.shape)
            #print(heats)
            #print(subimgs.shape)
            #print(subimgs)

            i = i + 1

    #rescale all heats to normalized range
    allheats = (allheats - np.min(allheats)) / (np.max(allheats)-np.min(allheats))
    tittxt = 'Heats from: ' + heatcolname
    imgutils.showheatmap(allsubimgs, allheats, heatdepend_opacity = False, opacity=opac, cmapname=colmap, title= tittxt, figs
ize=(12,10))

    # show info if requested
    if show_extra_info:
        print(allheats)
        i=0;
        for row in range(0,n_rows):
            for col in range(0,n_cols):
                print("image %d at (%d , %d): %s" % (i, row, col,imgnames[i]))

    return (allsubimgs, allheats)
```

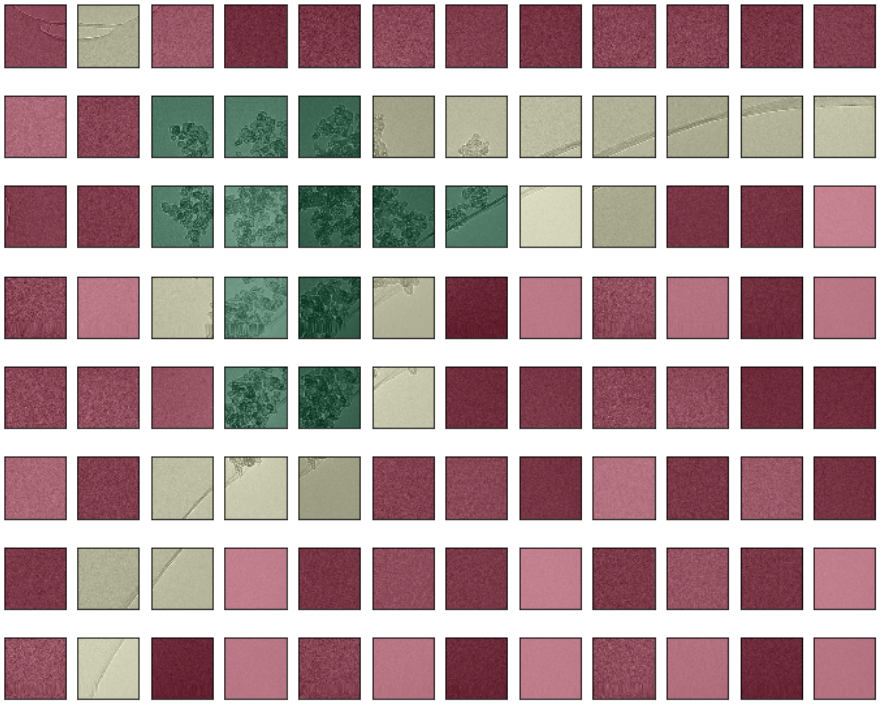
In [192]:
```
hm_classes = show_large_heatmap(df3, '|class|', imgnames[0:6], n_rows=2, n_cols=3)
```



Interesting: the (semi) **manual labelling has some mistakes**. So for better analyses I need to fix the labels

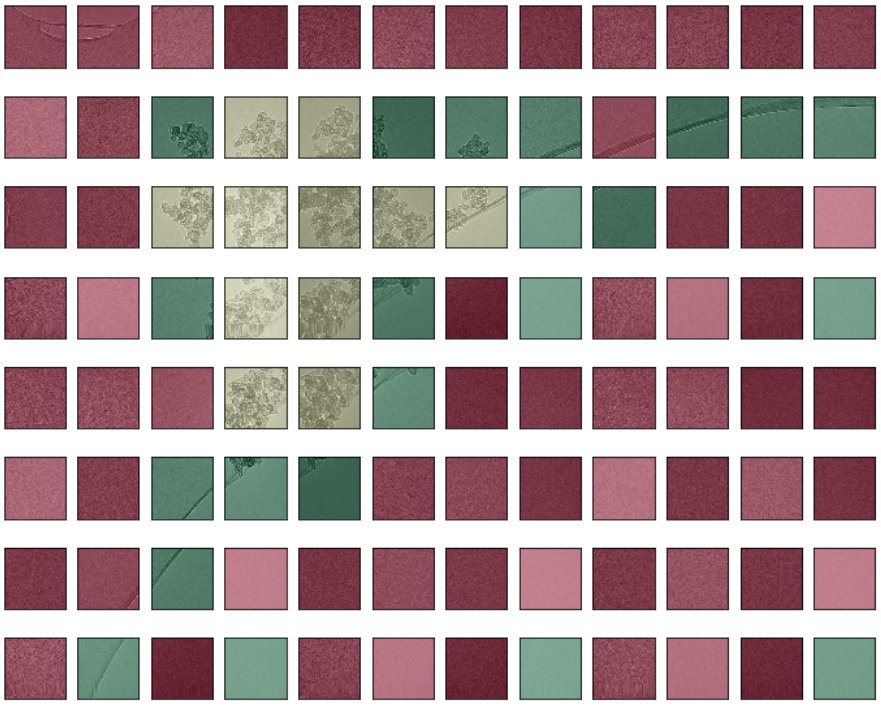
In [193]:
```
hm_hierarch = show_large_heatmap(df3, 'hierarch', imgnames[0:6], n_rows=2, n_cols=3)
```

Heats from: hierarch



In [194]:
```
hm_kmeans = show_large_heatmap(df3, 'k_means', imgnames[0:6], n_rows=2, n_cols=3)
```

Heats from: k_means



Remark : The color coding can be misleading. The only thing that matters is that same types have same images (the cluster nr is used as the 'heat', but there is no ordering in this cluster numbering)

It's better than I thought initially, it was the visualization!

(the individual heatmap used different color scales for each image, giving the wrong impression)

TO DO: Add this large heatmap to imgutils

---

10. Get a baseline score (manual counting)

So the real assessment can only be done manually by visual inspection, but counting the good ones and false positives / negatives is easy in large heatmaps.

- It found all the 'full ones' (class C)
- Tthe unsupervised learning did a good job on 'class B' if you agree that partial and the other texture are same class.
- It missed only one in this class (19 of 20)
- Interesting to check what it will do if we make it a two class problem or four class problem

There are many ways how to express the performance, see https://en.wikipedia.org/wiki/Confusion_matrix (https://en.wikipedia.org/wiki/Confusion_matrix). The confusion matrix is informative, but for metrics I can use some from (the block on the right). For the crystal case, we do not care too much about missing a few, so the True Positive Rate ('sensitivity) for the 'full ones' is a suitable indicator. The False Discovery Rate is alos intersting, as false positives are undesirable because they result in performing the next image acquisition experiment at places where there is not much to see.

So, we use these:

- TPR = True Positives / Real Positives
- FDR = False Positives / (True Positives + False Positives)

And then we want to see them for the 'Category C' i.e. with (almost) full crystal, and for partial

Let's create some helper function for the counting and these scores

In [195]:
```
def count_imgs_per_class(df_imgstats, classcolumn):
    return df_imgstats[classcolumn].value_counts()

def print_scores(methodname, class_count_tuples):
    print("")
    print("{:<20}|{:^12}|{:^12}|".format(methodname.upper(), "True Pos", "False Pos"))
    print("-"*(20+12+12+3))

    def print_score_line(class_name, TPR, FDR):
        print("{:<20}|{:^12.2%}|{:^12.2%}| ".format(class_name, TPR, FDR ))

    for (class_name, n_true_pos, n_false_pos, n_real_pos) in class_count_tuples:
        TPR = n_true_pos/n_real_pos
        FDR = n_false_pos/(n_true_pos + n_false_pos)
        print_score_line(class_name, TPR, FDR)

    print("-"*(20+12+12+3))
```

In [196]:
```
print_scores('Manual (using STD)', [('Full Crystal', 11, 2, 11),  ('Partial Crystal', 6, 4, 8) ])
print_scores('Hierarchical', [('Full Crystal', 11, 1, 11), ('Partial Crystal', 7, 12, 8) ])
print_scores('K-means', [('Full Crystal', 10, 1, 11), ('Partial Crystal', 7, 13, 8) ])
```

```
MANUAL (USING STD)  |  True Pos  | False Pos  |
-------------------------------------------------
Full Crystal        |  100.00%   |   15.38%   |
Partial Crystal     |   75.00%   |   40.00%   |
-------------------------------------------------

HIERARCHICAL        |  True Pos  | False Pos  |
-------------------------------------------------
Full Crystal        |  100.00%   |   8.33%    |
Partial Crystal     |   87.50%   |   63.16%   |
-------------------------------------------------

K-MEANS             |  True Pos  | False Pos  |
-------------------------------------------------
Full Crystal        |   90.91%   |   9.09%    |
Partial Crystal     |   87.50%   |   65.00%   |
-------------------------------------------------
```

REMARKS:

- some of the false positives in category 'Partial' are 'Full Ones' and some false positives in 'Full' are partial ones. So this score is a bit to strict, but accounting for this would require more complex scoring (or a full confusion matrix, where you still need to remark that some confusion is not so critical)
- running the algorithms gives some variation, so these scores may deviate a bit (it is manually counted)

11. Try-out: what will PCA + hierarchical give?

In [197]:
```
from sklearn import decomposition
```

In [198]:
```
fieldnames = ['pca_1','pca_2','pca_3', 'pca_4', 'pca_5']

n_comp = 3;

pca = decomposition.TruncatedSVD(n_components=n_comp)
X_fit = pca.fit_transform(X)
```

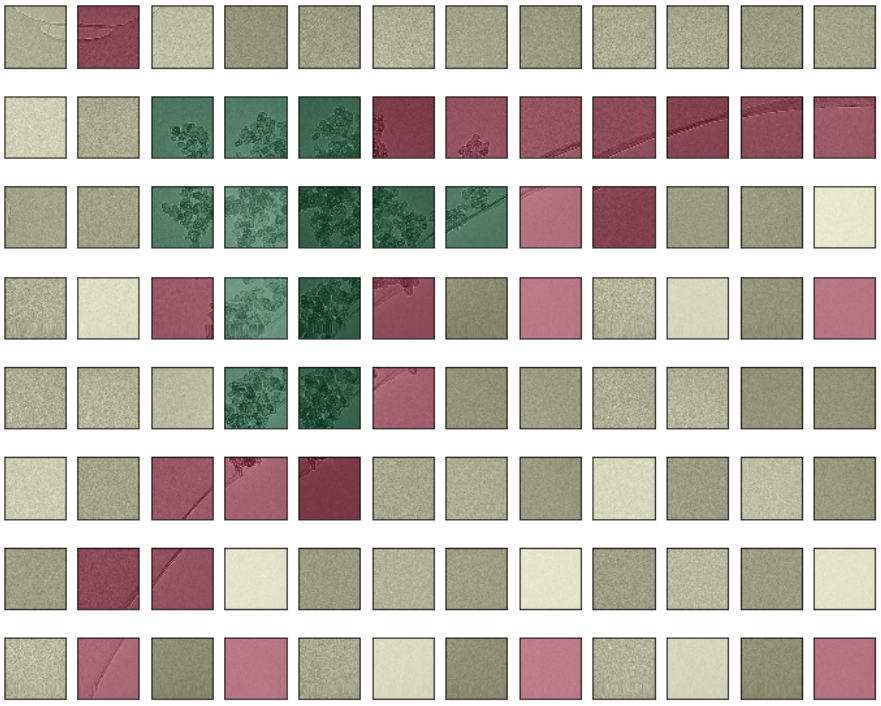
In [199]:
```
# convert into X Y vectors:
df_pca = pd.DataFrame(X_fit[:,0:n_comp], columns=fieldnames[:n_comp])
X_pca = df_pca.loc[:,fieldnames[:n_comp]]
```

In [200]:
```
Hclustering_pca = AgglomerativeClustering(n_clusters=3, affinity='cosine', linkage='complete')
Hclustering_pca.fit(X_pca)
hierarch_pca_pred = Hclustering_pca.labels_
```

In [201]:
```
df3 = pd.concat([df3, pd.Series(hierarch_pca_pred).rename('hierarch_pca')], axis=1)
```

In [202]:
```
hm_hierarch_pca = show_large_heatmap(df3, 'hierarch_pca', imgnames[0:6], n_rows=2, n_cols=3)
```

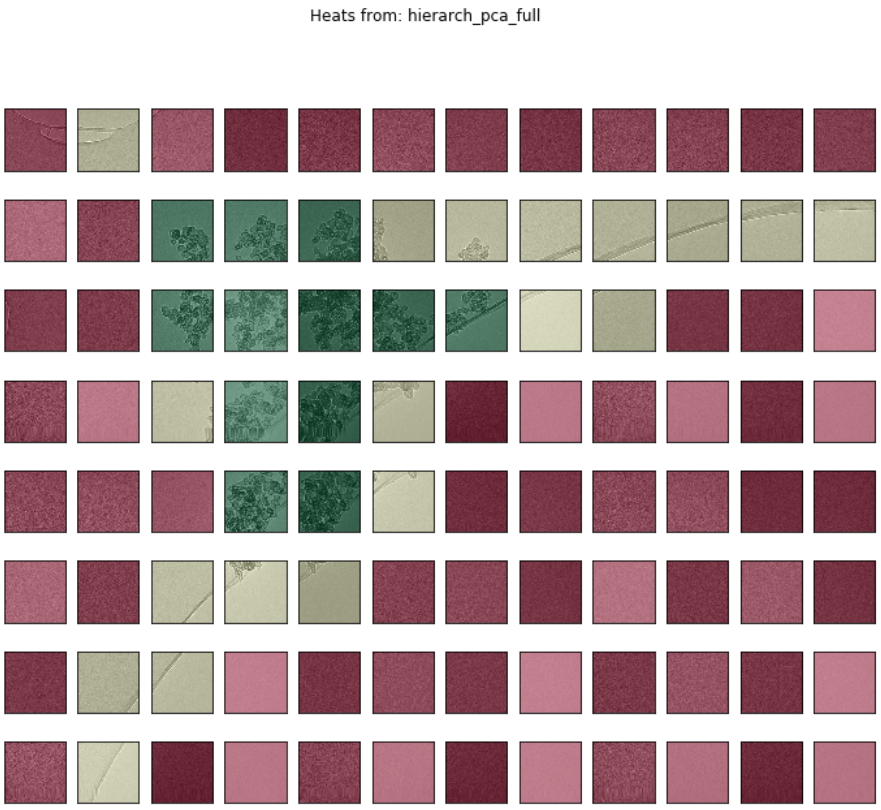
Heats from: hierarch_pca



Not bad. Also see what happens with 5 components

In [203]:
```
n_comp = 5;
pca = decomposition.TruncatedSVD(n_components=n_comp)
X_fit = pca.fit_transform(X)
df_pca = pd.DataFrame(X_fit[:,0:n_comp], columns=fieldnames[:n_comp])
X_pca = df_pca.loc[:,fieldnames[:n_comp]]
Hclustering_pca = AgglomerativeClustering(n_clusters=3, affinity='cosine', linkage='complete')
Hclustering_pca.fit(X_pca)
hierarch_pca_pred = Hclustering_pca.labels_

df3 = pd.concat([df3, pd.Series(hierarch_pca_pred).rename('hierarch_pca_full')], axis=1)
```

In [204]:
```
hm_hierarch_pca_full = show_large_heatmap(df3, 'hierarch_pca_full', imgnames[0:6], n_rows=2, n_cols=3)
```



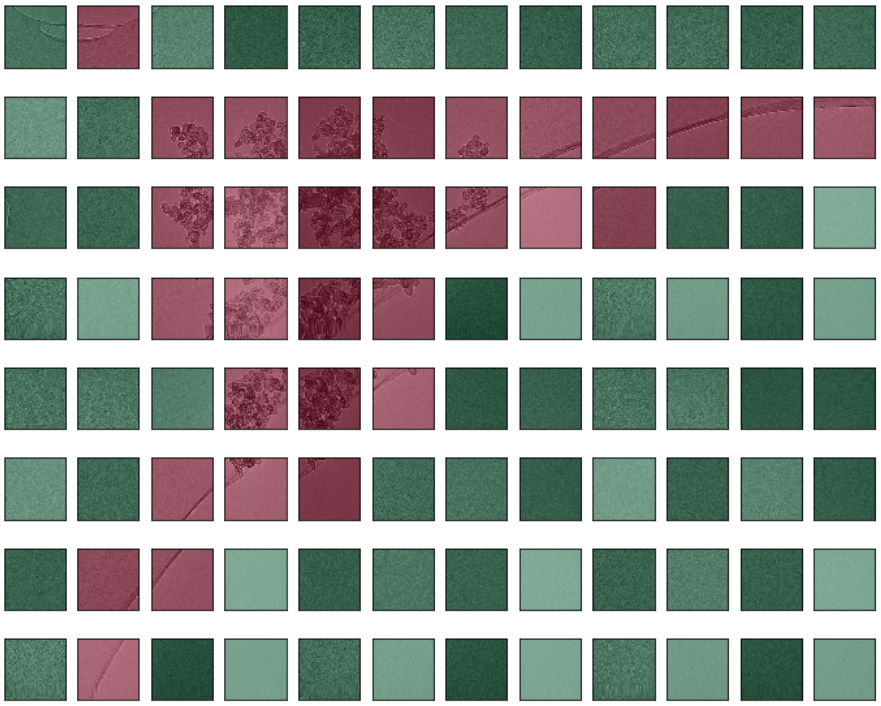
Result is almost identical to the hierarchical clustering without PCA (and whether to use 3 or 5 components did not matter that much.

Interesting of coarse with this PCA approach is that it will do feature selection for us

Now let's also try how it looks when going for 2 or 4 clusters

In [211]:
```
n_comp = 5;
pca = decomposition.TruncatedSVD(n_components=n_comp)
X_fit = pca.fit_transform(X)
df_pca = pd.DataFrame(X_fit[:,0:n_comp], columns=fieldnames[:n_comp])
X_pca = df_pca.loc[:,fieldnames[:n_comp]]
Hclustering_pca = AgglomerativeClustering(n_clusters=2, affinity='cosine', linkage='complete')
Hclustering_pca.fit(X_pca)
hierarch_pca_pred = Hclustering_pca.labels_

df3 = pd.concat([df3, pd.Series(hierarch_pca_pred).rename('hierarch_pca_2cats')], axis=1)
hm_hierarch_pca_2cats = show_large_heatmap(df3, 'hierarch_pca_2cats', imgnames[0:6], n_rows=2, n_cols=3)
```

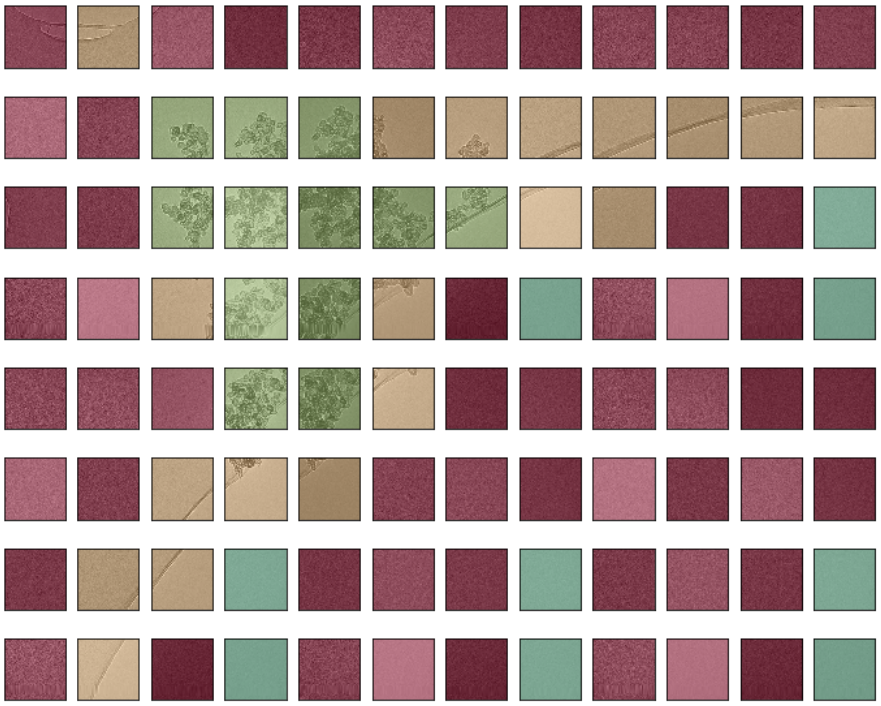
Heats from: hierarch_pca_2cats



Determining 2 classes perfectly finds the subimages without any features

In [212]:
```
n_comp = 5;
pca = decomposition.TruncatedSVD(n_components=n_comp)
X_fit = pca.fit_transform(X)
df_pca = pd.DataFrame(X_fit[:,0:n_comp], columns=fieldnames[:n_comp])
X_pca = df_pca.loc[:,fieldnames[:n_comp]]
Hclustering_pca = AgglomerativeClustering(n_clusters=4, affinity='cosine', linkage='complete')
Hclustering_pca.fit(X_pca)
hierarch_pca_pred = Hclustering_pca.labels_

df3 = pd.concat([df3, pd.Series(hierarch_pca_pred).rename('hierarch_pca_4cats')], axis=1)
hm_hierarch_pca_4cats = show_large_heatmap(df3, 'hierarch_pca_4cats', imgnames[0:6], n_rows=2, n_cols=3)
```

Heats from: hierarch_pca_4cats

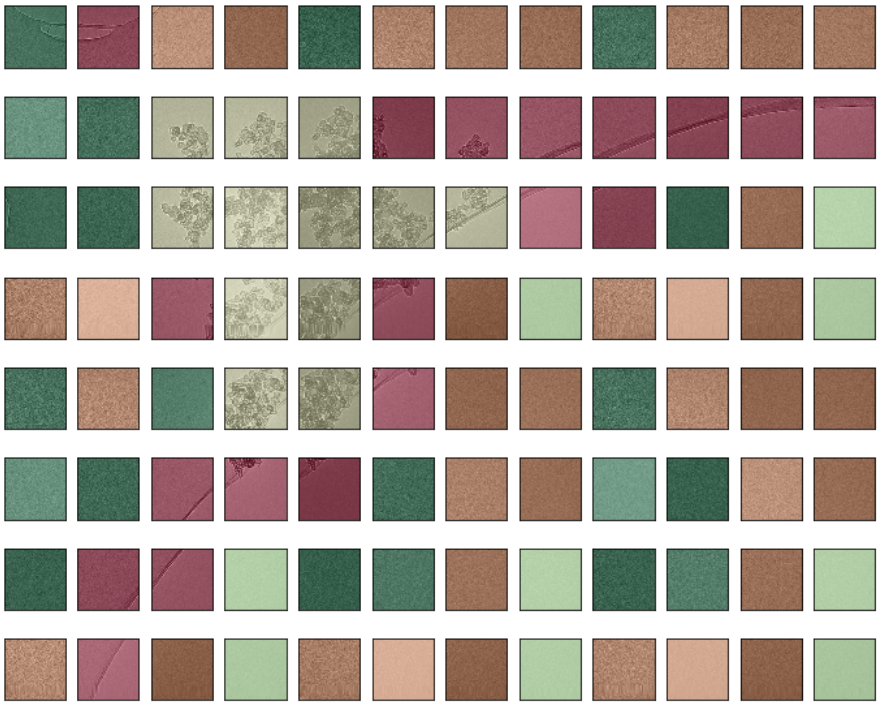


With 4 classes, results are not improved as it separates the empty ones

(and not the partial from the other texture; so let's also try 5 clusters)

In [214]:
```
n_comp = 5;
pca = decomposition.TruncatedSVD(n_components=n_comp)
X_fit = pca.fit_transform(X)
df_pca = pd.DataFrame(X_fit[:,0:n_comp], columns=fieldnames[:n_comp])
X_pca = df_pca.loc[:,fieldnames[:n_comp]]
Hclustering_pca = AgglomerativeClustering(n_clusters=5, affinity='cosine', linkage='complete')
Hclustering_pca.fit(X_pca)
hierarch_pca_pred = Hclustering_pca.labels_

df3 = pd.concat([df3, pd.Series(hierarch_pca_pred).rename('hierarch_pca_5cats')], axis=1)
hm_hierarch_pca_5cats = show_large_heatmap(df3, 'hierarch_pca_5cats', imgnames[0:6], n_rows=2, n_cols=3)
```

Heats from: hierarch_pca_5cats

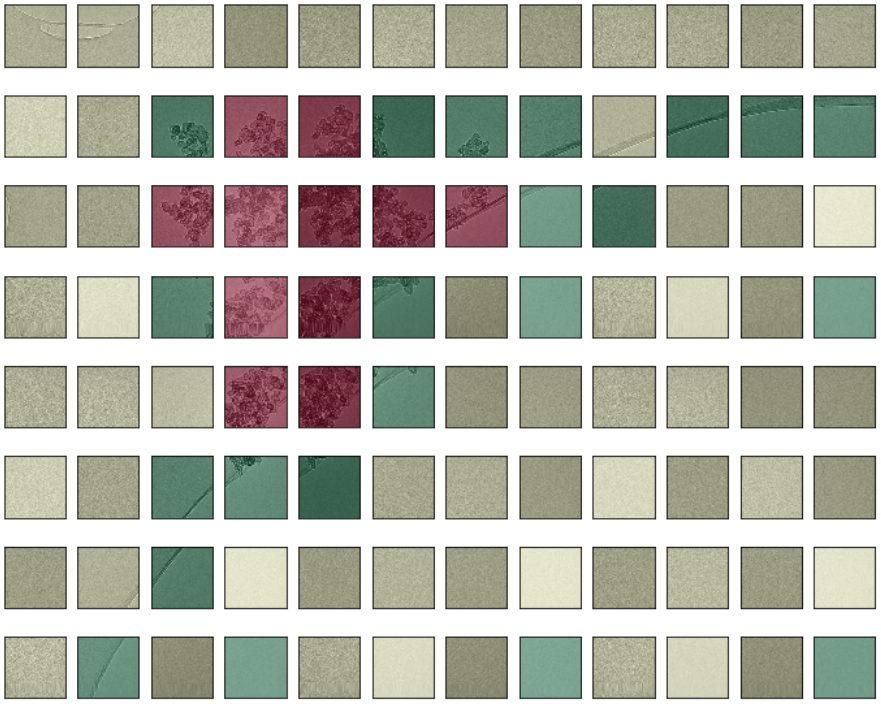


No, it also does not get better with 5. **So 3 it is for this data set!**

Finally, let's also see how PCA plus k-means works out

In [220]:
```
#df3.drop(columns=['k_means_pca'], inplace=True)

n_comp = 3;
pca = decomposition.TruncatedSVD(n_components=n_comp)
X_fit = pca.fit_transform(X)
df_pca = pd.DataFrame(X_fit[:,0:n_comp], columns=fieldnames[:n_comp])
X_pca = df_pca.loc[:,fieldnames[:n_comp]]

k_means_pca = cluster.KMeans(algorithm='auto', n_clusters=3, n_init=10, init='k-means++')
k_means_pca.fit(X)

k_means_pca_pred = k_means_pca.labels_

df3 = pd.concat([df3, pd.Series(k_means_pca_pred).rename('k_means_pca')], axis=1)
hm_kmeans_pca = show_large_heatmap(df3, 'k_means_pca', imgnames[0:6], n_rows=2, n_cols=3)
```

Heats from: k_means_pca



Hierarchical performs better in combination with k-means (there are quite some false pos. in the category B (partial)

---

12. Conclusions

- On this data set, the **unsupervised learning** concept **worked quite well**, both with hierarchical or k-means clustering
- Using PCA with **hierarchical clustering** gave similar results as the original features, with the advantage that it can reduced the number of required features
- Best **assessment** of the quality at the moment is **by visualization and human inspection**
- Scoring is difficult, but I have a **metric** now (though it involves manual counting)

13. Next Steps:

- move some functions into imgutils
- combine the full pipeline into one script (with debugging options) so I can start trying this out on other data sets and play with parameters like 'number of sub images'
- start thinking about hyper-parameter optimization

Michael Janus, 27 July 2018