03 - Exploratory Data Analysis of Point Cloud Features

May 5, 2021

1 Exploratory Data Analysis of Point Cloud Features

In this notebook, an exploratory data analysis for the features that are calculated from the 3D point cloud (see previous exercise notebook) is performed.

The learning objectives of this notebook are:

- Explore the features of the 3D point cloud and calculated features
- Hand-on experience with plots
- Get a brief introduction by example to matplotlib

Please note that there are many different ways to generate one and the same plot using matplotlib. The ones shown in this notebook uses the object-oriented interface and might not be the shortest way to achieve the result. There are often shortcuts that can be used.

For a more in-depth coverage of matplotlib, we refer to the literature in ISIS. This notebook is not intended to be a comprehensive and complete documentation of matplotlib.

2 Point properties

We start exploring the original 3D point cloud that was used for feature extraction. Especially the point properties recorded by the laser scanner sensor are of interest.

Load the original 3D point cloud file to explore the stored information therein.

[1]:
```
from pathlib import Path
import os

data_dir = str(Path.home()) + r'/coursematerial/GIS/ISPRS'

filepath = os.path.join(data_dir, r'Vaihingen3DTraining.las')

print(filepath)
```

```
/home/jovyan/coursematerial/GIS/ISPRS/Vaihingen3DTraining.las
```

[2]:
```
import laspy
import numpy as np
```

```
file = laspy.file.File(filepath, mode='r')
```

2.1 Classification

The 3D point cloud contains a classification of the individual points according to the following classes:

Power line, Low vegetation, Impervious surfaces, Car, Fence/Hedge, Roof, Facade, Shrub, Tree

We assign the NumPy array of the LAS file to the variable classes.

[3]:
```
classes = file.classification
```

And define a list of class names.

[4]:
```
class_names = ['Powerline', 'Low vegetation', 'Impervious surfaces', 'Car',␣
↪'Fence/Hedge', 'Roof', 'Facade', 'Shrub', 'Tree']
```

With the function **unique()**, we can get the unique values of the array as well as the number of times this unique value is present.

[5]:
```
u, c = np.unique(classes, return_counts=True)

print(u)
print(c)
```

```
[0 1 2 3 4 5 6 7 8]
[   546 180850 193723   4614  12070 152045  27250  47605 135173]
```

For a nicer output, we use pandas DataFrames, which we construct from the classes array and the list of class names as the index. (For a more detailed explanation of pandas, please refer to the respective notebook. At this point, it is not really necessary to understand pandas well.)

[6]:
```
import pandas as pd

df = pd.DataFrame(c, columns=['labels'],
                  index=class_names)

df
```

[6]:

	labels
Powerline	546
Low vegetation	180850
Impervious surfaces	193723
Car	4614
Fence/Hedge	12070
Roof	152045
Facade	27250
Shrub	47605

```
Tree                   135173
```

As we can see from the output, there are only few points of the classes power line, car, fences, facades, and maybe shrub.

For categorical data, a bar plots can be used to show the classes.

After using the magic command (%matplotlib inline) and importing the library, a figure of a certain size is constructed with the **figure()** function. A figure is top-level container for all the plot elements. And you can set figure properties that are valid for the whole figure and not just for parts thereof like the size of the figure or the resolution in DPI (dots per inch). With the method **suptitle()**, we give the figure a title.

Using the figure object, an axes is added to the figure with the method **add_axes()**, providing the size of a rectangle that the axes should take within the figure. An axes is again a container that holds a specific type of plot. The rectangle parameter is a quadruple with the elements left, bottom, width, and height. Left and bottom are the position of the axes within the figure, and width and height the size. The values of all four dimensions are given relative to the size of the figure with values between 0.0 and 1.0. By adding several axes objects to a figure, you can layout them with the rectangle parameter. But often it is sufficient to use some convenient methods for constructing a figure with several axes. (Try changing the parameter values for the add_axes() method to, e.g., 0.5 or 0.3 and see what happens.)

The axes object is then used to construct a bar plot with the method **bar()** taking the x-coordinates values as first parameter, the heights as second parameter, and then tick labels with the named **tick_label** parameter. For our plot, the unique values (u) are the x-coordinates, the count values (c) the y-coordinates, and the class names the tick labels.

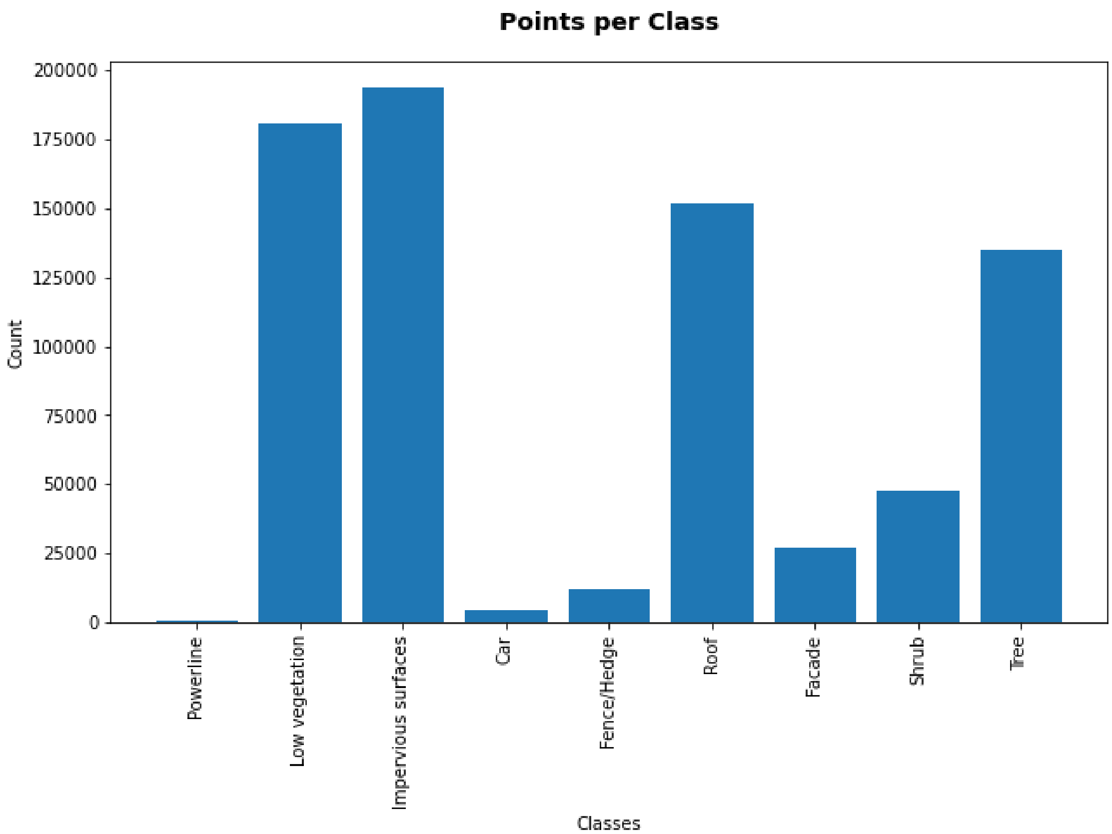
If we do not change the rotation of the tick labels, then the class names are written on top of each other as there is not enough horizontal space under the plot. We therefore change the rotation of all tick labels on the x-axis to a vertical rotation. We therefore loop through all tick labels of the x-axis with the method **xticklabels()**, and set the rotation of each tick label to vertical with the method **set_rotation()**. Instead of the string 'vertical', you can also provide a rotation angle in degrees.

Finally, the labels of the x-axis and the y-axis are set with the methods **set_xlabel()** and **set_ylabel()**.

[7]:
```
%matplotlib inline

import matplotlib.pyplot as plt

# construct a figure of a certain size
fig = plt.figure(figsize=(8, 5))

# set title of figure
fig.suptitle('Points per Class', fontsize=14, fontweight='semibold')

# add an axes that takes almost all the space of the figure,
# but leaves some space at the top for the title
```

```
ax = fig.add_axes((0, 0, 1, 0.90))

# draw a bar plot with u as the difference x values,
# c as the height, and class names for the labels
ax.bar(u, c, tick_label=class_names)

# rotate all labels to vertical (rotate by 90 degree)
for tick in ax.get_xticklabels():
    tick.set_rotation('vertical')

# set labels of axes
ax.set_xlabel('Classes')
ax.set_ylabel('Count')
```

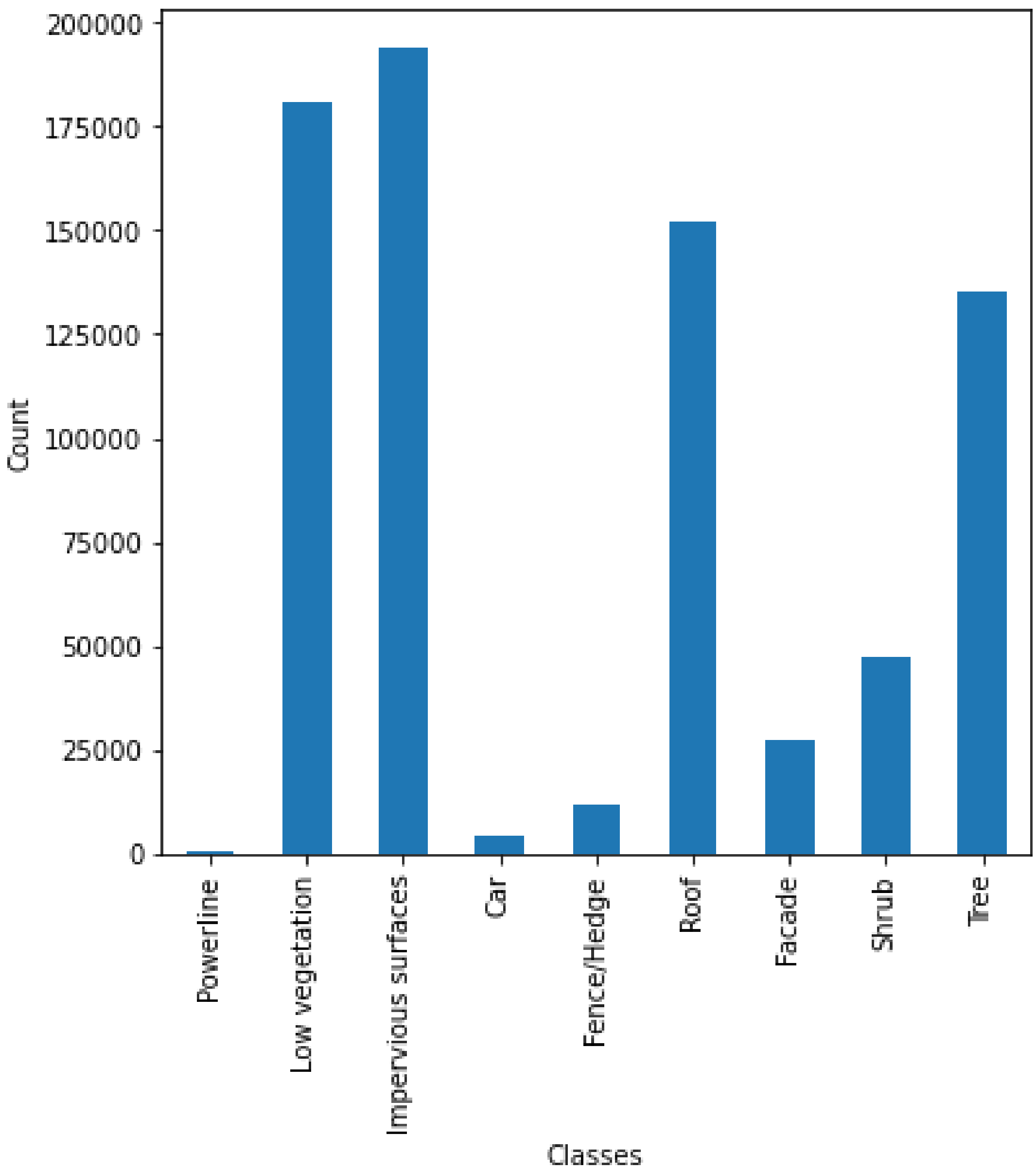
[7]: Text(0, 0.5, 'Count')



The pandas DataFrame class also provides a lot of convenient plotting methods. Pandas uses internally the matplotlib, so that the **bar()** method returns the axes object, which can be used to change some properties of the plot.

[8]:
```python
ax = df.plot.bar(figsize=(6,6), legend=False)
ax.set_xlabel('Classes')
ax.set_ylabel('Count')
```

[8]: Text(0, 0.5, 'Count')



From the plot, the absolute number of points per class, but also the proportions between the individual classes, become readily apparent. From a machine learning perspective, the underrepresented classes will probably be difficult to learn. But for low vegetation, impervious surfaces, roof, and tree, there should be enough points.

The LAS file also contains information about **intensity**, **number of returns** (**num_returns**), and **return number** (**return_num**), which we explore in the following.

Intensity is the returned intensity of the laser beam. The laser emits a laser signal with a certain intensity. This laser signal hits an object and is partially absorbed and partially reflected, where the ratio depends on the material of the surface. The portion of the laser beam that is reflected and reaches the laser scanner again is recorded as the intensity value.

Some objects reflect the laser signal only partially, so some portion is reflected and another portion is going through the object. This typically happens when the laser hits higher vegetation like trees, where the signal is then reflected at different height levels at the branches until it reaches the ground. Then we have several returns. The **number of returns** (**num_returns**) is the number of returns the laser scanner recorded for this measured point.

If there are several returns per laser beam, then these returns are converted into different point coordinates. This is because the laser beam traveled further towards the ground than the reflected signal and reached an object at different 3D coordinates. So, there are typically as many points in the point cloud from the same laser beam as there are returns. And the **return number** (**return_num**) gives information, which return it was.

First, we take a look at the data types.

[9]:
```
print('Intensity:', file.intensity.dtype)
print('Number of returns:', file.num_returns.dtype)
print('Return number:', file.return_num.dtype)
```

```
Intensity: uint16
Number of returns: uint8
Return number: uint8
```

Since the data types are integer values, we can also take a look at how many unique values there are.

[10]:
```
print('Intensity:', np.unique(file.intensity))
print('Number of returns:', np.unique(file.num_returns))
print('Return number:', np.unique(file.return_num))
```

```
Intensity: [  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
```

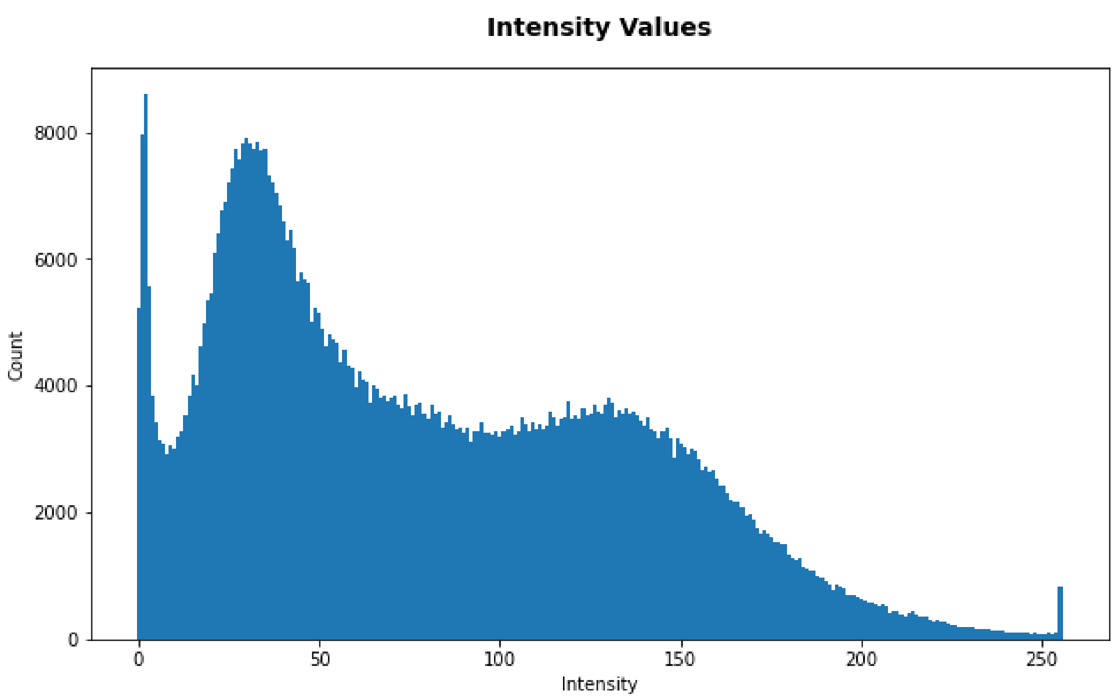
 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251
 252 253 254 255]
Number of returns: [1 2 3 4]
Return number: [1 2 3 4]

2.2 Intensity

Because of the large number of different values for intensity, we use a bar plot to show how many times these values occur. (We could also use a histogram plot, but then would need to define the bins ourselves.) The parameter **width** determines the occupied bar width, and a value of 1.0 means that the bars are one after another without any gaps. With this many bars, not all gaps would be visible and it would make the visualization look unattractive.

[11]:
```
u, c = np.unique(file.intensity, return_counts=True)

fig = plt.figure(figsize=(8, 5))
fig.suptitle('Intensity Values', fontsize=14, fontweight='semibold')

ax = fig.add_axes((0, 0, 1, 0.90))
ax.bar(u, c, width=1.0)
ax.set_xlabel('Intensity')
ax.set_ylabel('Count')
```

[11]:
```
Text(0, 0.5, 'Count')
```

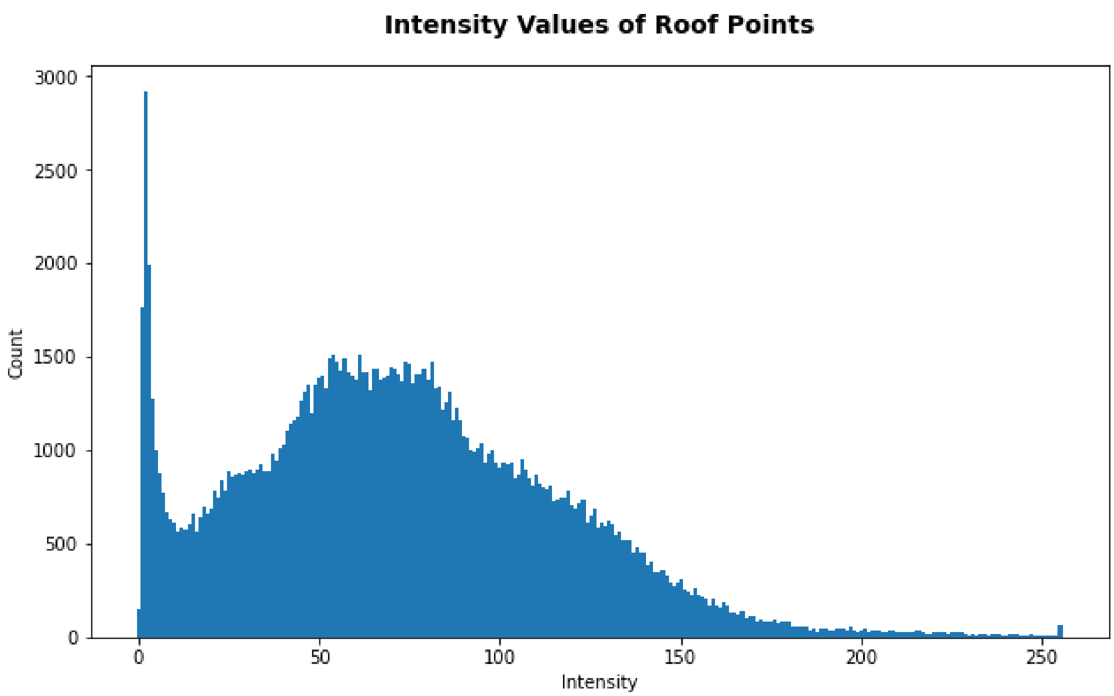


Let us show the intensity only for one class. With the NumPy function **where()**, we can get an array of indices, where a given condition is true. By subsequently applying the function **take()** with these indices, the values can be taken at the appropriate indexes.

What follows are the intensities for the class roof.

[12]:
```
roof_intensities = np.take(file.intensity, np.where(classes == 5))

u, c = np.unique(roof_intensities, return_counts=True)

fig = plt.figure(figsize=(8, 5))
fig.suptitle('Intensity Values of Roof Points', fontsize=14,␣
 ↪fontweight='semibold')

ax = fig.add_axes((0, 0, 1, 0.90))
ax.bar(u, c, width=1.0)
ax.set_xlabel('Intensity')
ax.set_ylabel('Count')
```

[12]:
```
Text(0, 0.5, 'Count')
```

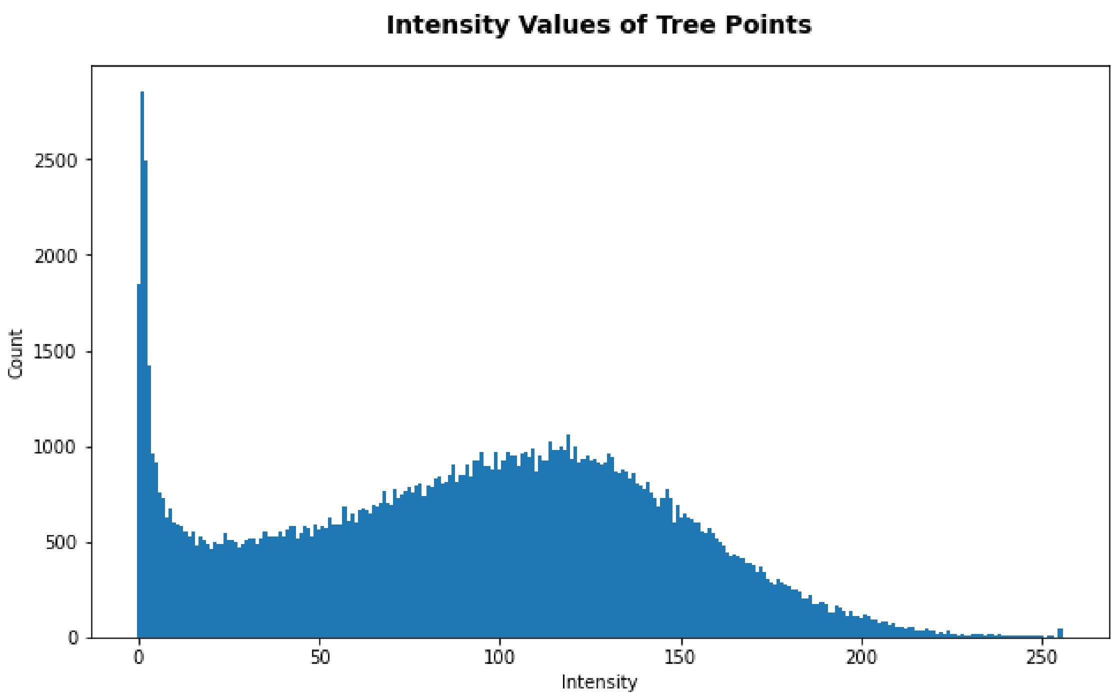


And here for the class trees.

[13]:
```
tree_intensities = np.take(file.intensity, np.where(classes == 8))

u, c = np.unique(tree_intensities, return_counts=True)
```

```
fig = plt.figure(figsize=(8, 5))
fig.suptitle('Intensity Values of Tree Points', fontsize=14,␣
 ↪fontweight='semibold')

ax = fig.add_axes((0, 0, 1, 0.90))
ax.bar(u, c, width=1.0)
ax.set_xlabel('Intensity')
ax.set_ylabel('Count')
```

[13]: Text(0, 0.5, 'Count')



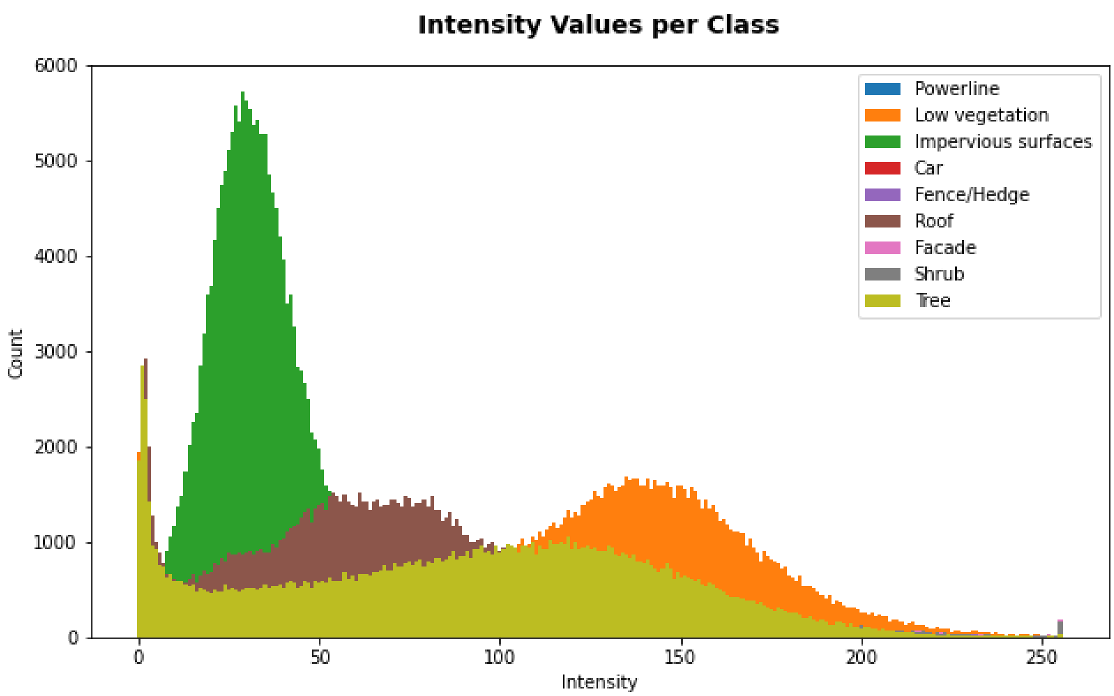
We can already see that the roof points have generally a lower intensity as the tree points.

Next, we plot several bar plots on top of each other, one per class, so that we can better see which object categories emit a low or high intensity return of the laser pulse. For this purpose, we loop over all class names, extract all points of a certain class, construct a bar plot on the axes, and give the plot a label (with the **label** parameter) that is used create a legend for the axes with the method **legend()**. We let matplotlib find the best location for the legend.

[14]:
```
fig = plt.figure(figsize=(8, 5))
fig.suptitle('Intensity Values per Class', fontsize=14, fontweight='semibold')



ax = fig.add_axes((0, 0, 1, 0.90))
```

```
ax.set_xlabel('Intensity')
ax.set_ylabel('Count')

for i,n in enumerate(class_names):
    u, c = np.unique(np.take(file.intensity, np.where(classes == i)),␣
↪return_counts=True)
    ax.bar(u, c, width=1.0, label=n)

ax.legend(loc='best')
```

[14]: <matplotlib.legend.Legend at 0x7f443b9c1b90>



Although the underrepresented classes are not really visible, a general trend can be seen for the 4 classes that are well represented. Impervious surfaces mostly emit low intensity, roof surfaces higher intensity, trees seem to be more balanced with a peak between roof surfaces and low vegetation, and low vegetation has mostly higher intensity.

If we wanted to assign points to categories (perform a classification by hand) based only on their intensity, then we could try to find values for intensity that separates the individual class ranges. For example, a point with intensity <= 50 is more likely to belong to the impervious surface class than to the low vegetation class (or the other 2 mentioned classes). A machine learning algorithm would find these separating values in such a way that the correct class is predicted for most points. If we then have a point without a class, we could check in which range this point falls according to its intensity, and give it the respective class label. However, the task of classification is rarely based on one feature only, but more often on several features.

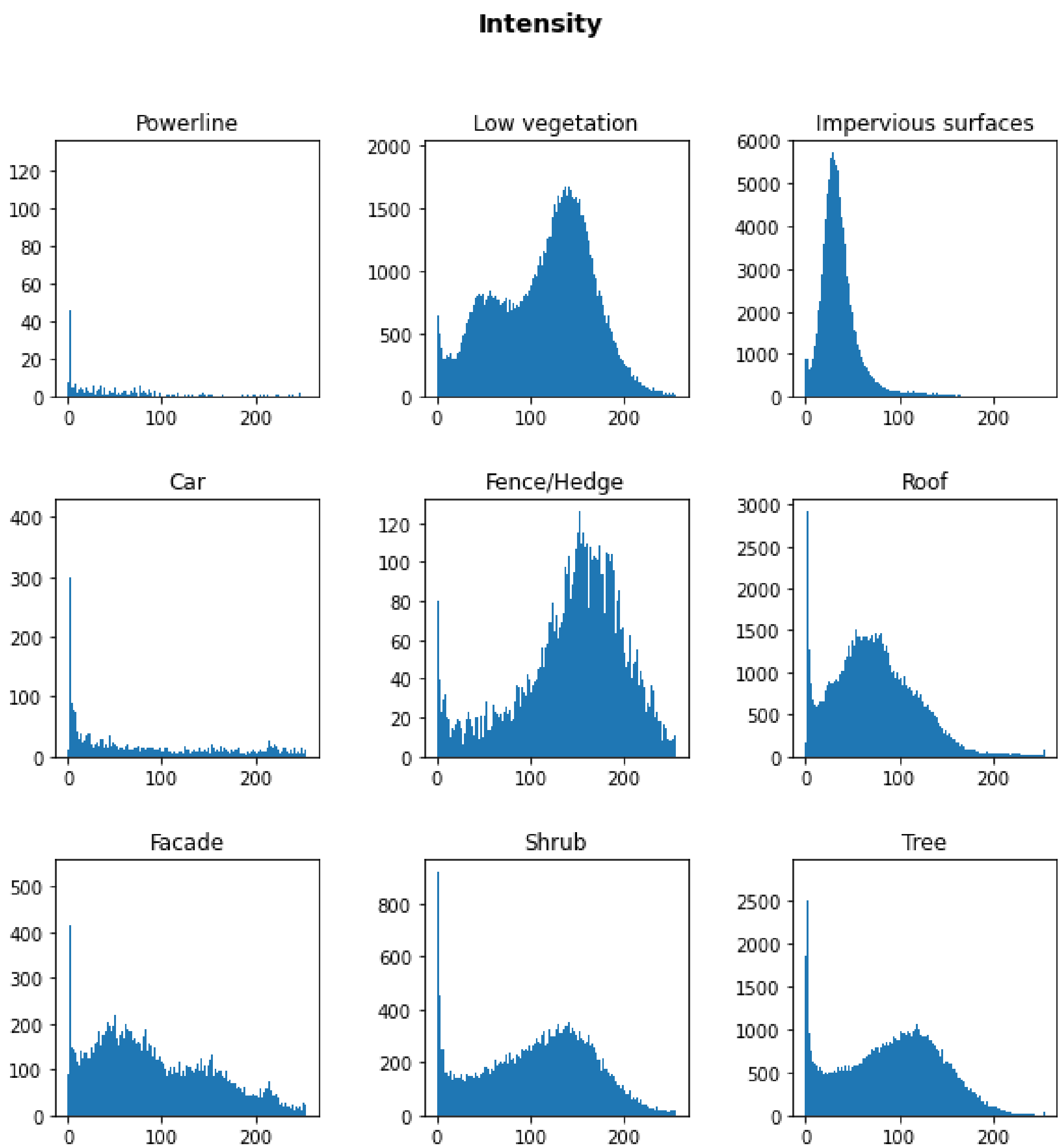
The next example shows subplots, in which all the classes are given in their own plot. The function **subplots()** constructs a figure and a number of axis according to the given number of rows (first parameter) and number of columns (second parameter). We can also define that the properties (scale, ticks, etc.) of the axes objects can be shared for the x-axis (**sharex**) and y-axis (**sharey**) in each column or row, respectively. Try also the value **all**, so that the scales are the same for all plots.

Because the plots would be rather close together, we adjust the space between subplots with the **subplots_adjust()** method. The two parameters defined are given as a fraction of the average axis width or height reserved for the space between subplots. (Try to comment out this line to see how it would otherwise look like.)

The axes object (returned by subplots) is a 2D array of the number of axes constructed, which need to be indexed accordingly with 2 index values. Here, we calculate the row index of the array by dividing the class index (which is the variable i in the loop) by 3 (, because we have 3 plots per row). In order to have an integer number and not a floating point number as the result of the division, we use the floor division operator // instead of the true division (/). (A floating point number as an index value would be considered an error.) For the column index (the second index), we use the modulo operator **%**, again by 3 for the same reason as before (number of columns per row is 3).

The title for each axes can be set with the **set()** method, where we have to specify which properties of the bar plot we would like to change.

[15]:
```
#fig, ax = plt.subplots(3, 3, sharex='all', sharey='all', figsize=(10, 10))
fig, ax = plt.subplots(3, 3, sharex='none', sharey='none', figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
fig.suptitle('Intensity', fontsize=14, fontweight='semibold')

for i,n in enumerate(class_names):
    u, c = np.unique(np.take(file.intensity, np.where(classes == i)),␣
 ↪return_counts=True)
    ax[i//3, i%3].bar(u, c, width=1.0)
    ax[i//3, i%3].set(title=class_names[i])
```



What can be learned from the bar plots?

- For all classes, there are quite a number of points without (or zero) intensity values.
- All vegetation related classes like tree, shrub, hedge, and low vegetation have a peak somewhere between 120 to 170. In comparison, all classes that are constructed from building material (stone, concrete, asphalt) like impervious, facade, and roof surfaces, show lower intensity. This means they absorb more light then the vegetation classes.
- Besides the main peak, low vegetation shows another, lower peak at around 50, which is close to the main peak of impervious surfaces. So, it could well be that low vegetation also contains some points from impervious surfaces. Or that the reflections are a mixture of vegetation and ground. This is quite understandable, because signals resulting from these two classes

are difficult to separate. And it seems that rather the vegetation class is "impure" and the impervious surface class is not.

2.3 Number of returns

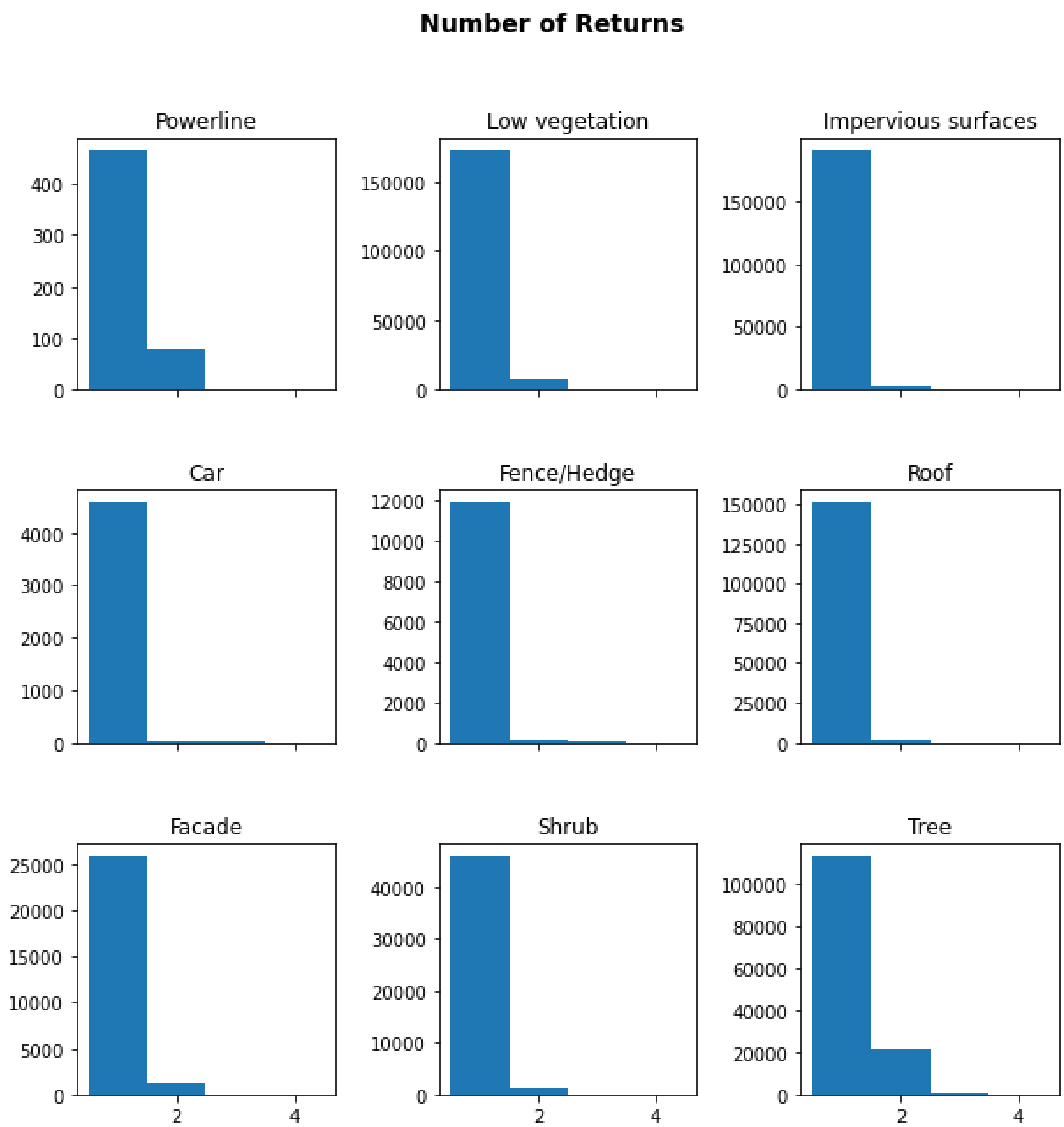
Let us move on to the number of returns. First, we count the occurrences of the 4 values.

[16]:
```
u, c = np.unique(file.num_returns, return_counts=True)
print(u, c)
```

```
[1 2 3 4] [716755  36216    902      3]
```

Most laser pulses resulted in only 1 return. Let us check which classes are responsible for it. Maybe some underrepresented class is the only cause.

[17]:
```
fig, ax = plt.subplots(3, 3, sharex='all', sharey='none', figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
fig.suptitle('Number of Returns', fontsize=14, fontweight='bold')

for i,n in enumerate(class_names):
    u, c = np.unique(np.take(file.num_returns, np.where(classes == i)),␣
 ↪return_counts=True)
    ax[i//3, i%3].bar(u, c, width=1.0)
    ax[i//3, i%3].set(title=class_names[i])
```



Although mostly power lines, trees, and maybe facades resulted in several returns, the higher number of returns are not dominant for these classes. The feature is therefore very likely not a good one to separate any classes.

2.4 Return number

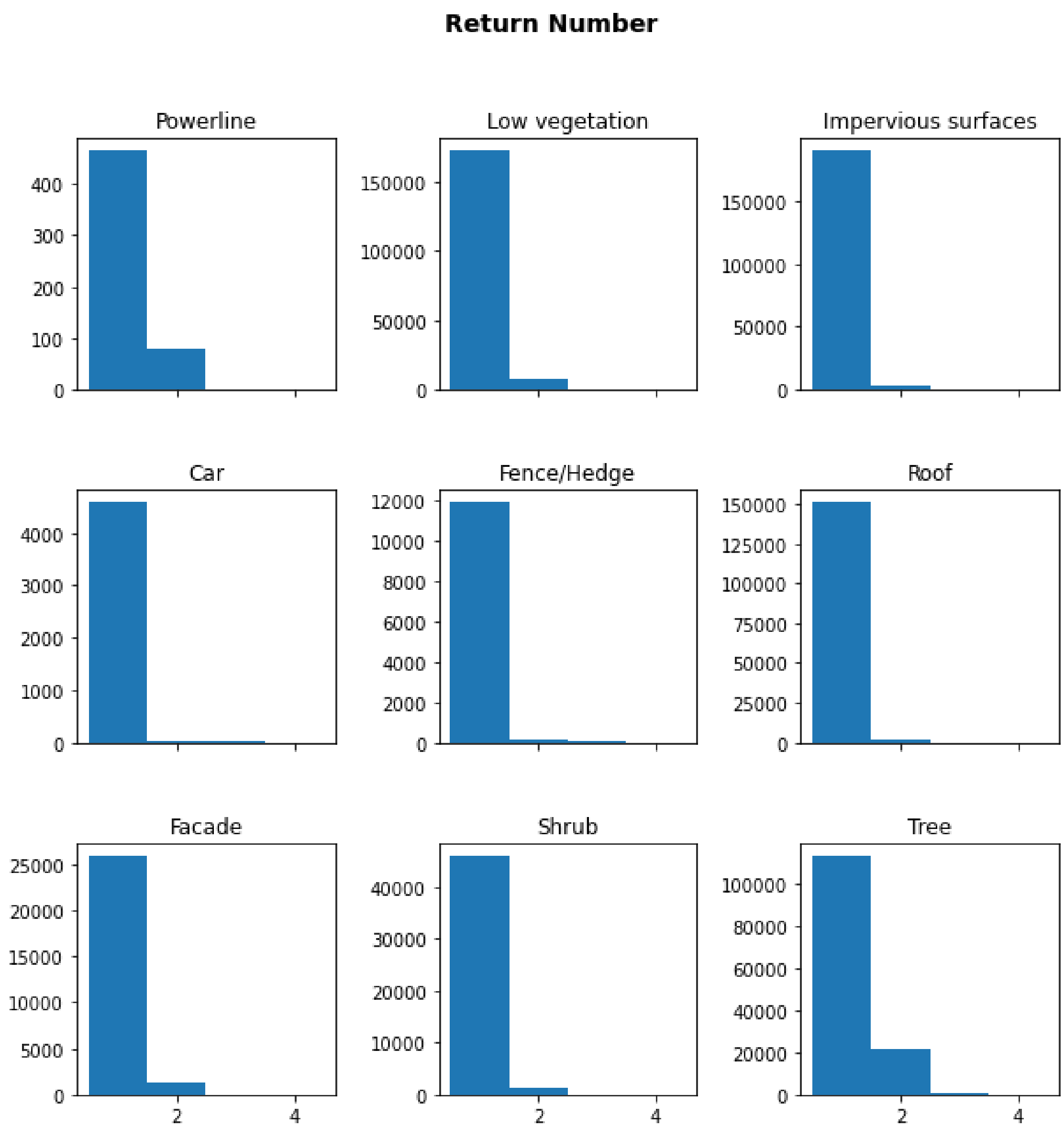
As a last attribute of the LAS file, we count the occurrences of the return number.

[18]:
```
u, c = np.unique(file.return_num, return_counts=True)
print(u, c)
```

```
[1 2 3 4] [735339  18147     389       1]
```

And have the same plot as above.

```
[19]: fig, ax = plt.subplots(3, 3, sharex='all', sharey='none', figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
fig.suptitle('Return Number', fontsize=14, fontweight='semibold')

for i,n in enumerate(class_names):
    u, c = np.unique(np.take(file.num_returns, np.where(classes == i)),
 ↪return_counts=True)
    ax[i//3, i%3].bar(u, c, width=1.0)
    ax[i//3, i%3].set(title=class_names[i])
```



From the point properties of the point cloud, only the intensity shows a relevant spectrum of values that can also be allocated to the individual classes. Although with intensity alone, it will only be possible to differentiate between the broad categories of vegetation and construction materials.

We therefore continue with the extracted features.

3 Features

Load the extracted features from the exercise notebook '02 - Feature Calculations with NumPy (Part 2)'.

The following code cell uses the reference features from the '02 - Check Your Features" notebook. If you want to use your own calculated features, then copy them into the same folder as this notebook and uncomment the respective line.

[20]:
```
data_dir = str(Path.home()) + r'/coursematerial/GIS/ISPRS/PointsWithFeatures'

filepath = os.path.join(data_dir, r'FeaturesReference.npy')

features = np.load(filepath)

#features = np.load('Features.npy')

print(features.shape)
```

```
(753876, 23)
```

Construct a pandas DataFrame, so that we can select the columns with a string index for convenience. Otherwise we would need to remember which column index a certain features has. When selecting a column of a DataFrame, we get an object of type Series, which we can convert with the method **to_numpy()** to a 1D NumPy array with n elements to use in matplotlib. However, matplotlib also accepts pandas DataFrames as data.

[21]:
```
feature_names = ['linearity',
                 'planarity',
                 'scattering',
                 'omnivariance',
                 'anisotropy',
                 'eigenentropy',
                 'sum_eigenvalues',
                 'change_of_curvature',
                 'radius_knn3d',
                 'density_3d',
                 'verticality',
                 'absolute_height',
                 'delta_z_knn3d',
```

```
                 'std_z',
                 'sum_of_eigenvalues_2d',
                 'ratio_of_eigenvalues_2d',
                 'radius_knn_2d',
                 'density_2d',
                 'eigenvalue1',
                 'eigenvalue2',
                 'eigenvalue3',
                 'eigenvalue2D1',
                 'eigenvalue2D2'
                ]

df = pd.DataFrame(features, columns=feature_names)

df
```

[21]:

```
        linearity  planarity  scattering  omnivariance  anisotropy  \
0        0.514636   0.484771    0.000593      0.044426    0.999407
1        0.286950   0.712251    0.000799      0.048378    0.999201
2        0.337194   0.662073    0.000733      0.047258    0.999267
3        0.670130   0.327899    0.001971      0.065043    0.998029
4        0.591237   0.406962    0.001801      0.064009    0.998199
…             …          …           …            …          …
753871   0.738930   0.257483    0.003587      0.077360    0.996413
753872   0.466459   0.532517    0.001024      0.053272    0.998976
753873   0.730894   0.266702    0.002404      0.068023    0.997596
753874   0.718838   0.277748    0.003415      0.076797    0.996585
753875   0.781174   0.215357    0.003469      0.074632    0.996531

        eigenentropy  sum_eigenvalues  change_of_curvature  radius_knn3d  \
0           0.635132         0.514617             0.000399      1.423552
1           0.682779         0.483546             0.000466      1.291007
2           0.675992         0.485428             0.000441      1.257975
3           0.570471         0.566638             0.001480      1.278241
4           0.611306         0.947597             0.001277      1.804384
…                …                …                    …             …
753871      0.528006         0.093828             0.002836      0.511273
753872      0.651261         0.097857             0.000667      0.512640
753873      0.529421         0.097016             0.001891      0.487750
753874      0.543241         0.093028             0.002658      0.521632
753875      0.488836         0.111733             0.002838      0.520769

        density_3d  …     std_z  sum_of_eigenvalues_2d  \
0         1.655089  …  0.035433               0.513361
1         2.218989  …  0.035964               0.482252
2         2.398423  …  0.035526               0.484166
3         2.286142  …  0.057638               0.563316
```

```
4         0.812746  …  0.070328               0.942651
…              …   …       …                  …
753871   35.725886  …  0.022804               0.093308
753872   35.440786  …  0.016182               0.097595
753873   41.148094  …  0.018638               0.096668
753874   33.639426  …  0.019628               0.092643
753875   33.807018  …  0.023458               0.111183

        ratio_of_eigenvalues_2d  radius_knn_2d  density_2d  eigenvalue1  \
0                      0.483495       1.423517    3.141629     0.672967
1                      0.711442       1.290039    3.825380     0.583482
2                      0.663891       1.257816    4.023891     0.601128
3                      0.331784       1.273499    3.925390     0.750840
4                      0.410825       1.799694    1.965543     0.708937
…                           …            …           …          …
753871                 0.260700       0.510392   24.438379     0.790728
753872                 0.531483       0.511077   24.372886     0.651650
753873                 0.268709       0.487647   26.771227     0.786466
753874                 0.281240       0.520096   23.534927     0.778466
753875                 0.218782       0.519230   23.613493     0.818133

        eigenvalue2  eigenvalue3  eigenvalue2D1  eigenvalue2D2
0          0.326634     0.000399       0.674084       0.325916
1          0.416052     0.000466       0.584302       0.415698
2          0.398431     0.000441       0.601001       0.398999
3          0.247680     0.001480       0.750873       0.249127
4          0.289787     0.001277       0.708805       0.291195
…               …           …             …             …
753871     0.206435     0.002836       0.793210       0.206790
753872     0.347682     0.000667       0.652962       0.347038
753873     0.211643     0.001891       0.788203       0.211797
753874     0.218875     0.002658       0.780494       0.219506
753875     0.179029     0.002838       0.820491       0.179509

[753876 rows x 23 columns]
```

Add the classification of the points to the features DataFrame, so that everything is together.

[22]:
```
df['class'] = classes

df
```

[22]:
```
        linearity  planarity  scattering  omnivariance  anisotropy  \
0        0.514636   0.484771    0.000593      0.044426    0.999407
1        0.286950   0.712251    0.000799      0.048378    0.999201
2        0.337194   0.662073    0.000733      0.047258    0.999267
3        0.670130   0.327899    0.001971      0.065043    0.998029
```

```
4         0.591237   0.406962    0.001801      0.064009    0.998199
…            …          …           …            …            …
753871    0.738930   0.257483    0.003587      0.077360    0.996413
753872    0.466459   0.532517    0.001024      0.053272    0.998976
753873    0.730894   0.266702    0.002404      0.068023    0.997596
753874    0.718838   0.277748    0.003415      0.076797    0.996585
753875    0.781174   0.215357    0.003469      0.074632    0.996531

        eigenentropy  sum_eigenvalues  change_of_curvature  radius_knn3d  \
0           0.635132         0.514617             0.000399      1.423552
1           0.682779         0.483546             0.000466      1.291007
2           0.675992         0.485428             0.000441      1.257975
3           0.570471         0.566638             0.001480      1.278241
4           0.611306         0.947597             0.001277      1.804384
…              …                …                   …             …
753871      0.528006         0.093828             0.002836      0.511273
753872      0.651261         0.097857             0.000667      0.512640
753873      0.529421         0.097016             0.001891      0.487750
753874      0.543241         0.093028             0.002658      0.521632
753875      0.488836         0.111733             0.002838      0.520769

        density_3d  …  sum_of_eigenvalues_2d  ratio_of_eigenvalues_2d  \
0         1.655089  …               0.513361                 0.483495
1         2.218989  …               0.482252                 0.711442
2         2.398423  …               0.484166                 0.663891
3         2.286142  …               0.563316                 0.331784
4         0.812746  …               0.942651                 0.410825
…            …   …                     …                       …
753871   35.725886  …               0.093308                 0.260700
753872   35.440786  …               0.097595                 0.531483
753873   41.148094  …               0.096668                 0.268709
753874   33.639426  …               0.092643                 0.281240
753875   33.807018  …               0.111183                 0.218782

        radius_knn_2d  density_2d  eigenvalue1  eigenvalue2  eigenvalue3  \
0            1.423517    3.141629     0.672967     0.326634     0.000399
1            1.290039    3.825380     0.583482     0.416052     0.000466
2            1.257816    4.023891     0.601128     0.398431     0.000441
3            1.273499    3.925390     0.750840     0.247680     0.001480
4            1.799694    1.965543     0.708937     0.289787     0.001277
…               …           …            …            …            …
753871       0.510392   24.438379     0.790728     0.206435     0.002836
753872       0.511077   24.372886     0.651650     0.347682     0.000667
753873       0.487647   26.771227     0.786466     0.211643     0.001891
753874       0.520096   23.534927     0.778466     0.218875     0.002658
753875       0.519230   23.613493     0.818133     0.179029     0.002838
```

```
        eigenvalue2D1  eigenvalue2D2  class
0            0.674084       0.325916      1
1            0.584302       0.415698      1
2            0.601001       0.398999      1
3            0.750873       0.249127      1
4            0.708805       0.291195      1
…                 …              …     …
753871       0.793210       0.206790      2
753872       0.652962       0.347038      2
753873       0.788203       0.211797      2
753874       0.780494       0.219506      2
753875       0.820491       0.179509      2

[753876 rows x 24 columns]
```

The method **describe()** of Pandas shows some basic statistics that can be useful for a first look at the data.

(See how the mean of classes makes no real sense, but the 50% percentile (or median) is more meaningful.)

[23]:
```
df.describe()
```

[23]:
```
           linearity      planarity     scattering   omnivariance  \
count  753876.000000  753876.000000  753876.000000  753876.000000
mean        0.557764       0.387218       0.055018       0.124257
std         0.254856       0.251118       0.093513       0.088348
min         0.000276       0.000062       0.000008       0.003198
25%         0.349201       0.169167       0.001215       0.051112
50%         0.555117       0.363175       0.005859       0.079920
75%         0.774457       0.590015       0.074130       0.204756
max         0.999620       0.999046       0.869420       0.332718

          anisotropy   eigenentropy  sum_eigenvalues  change_of_curvature  \
count  753876.000000  753876.000000    753876.000000        753876.000000
mean        0.944982       0.637717         0.530745             0.032339
std         0.093513       0.216737         0.644467             0.049539
min         0.130580       0.004297         0.000411             0.000007
25%         0.925870       0.537542         0.179740             0.000844
50%         0.994141       0.673145         0.362666             0.004596
75%         0.998785       0.768934         0.676156             0.049049
max         0.999992       1.096792        34.571103             0.307947

         radius_knn3d     density_3d  …  sum_of_eigenvalues_2d  \
count  753876.000000  753876.000000  …          753876.000000
mean        0.972458     118.045849  …               0.434099
std         0.487351    1200.733030  …               0.441575
min         0.030000       0.002944  …               0.000048
```

```
25%          0.671118        2.419261  ...               0.157143
50%          0.890730        6.756209  ...               0.335010
75%          1.254352       15.795880  ...               0.515722
max         11.749357  176838.830873  ...              27.902223

       ratio_of_eigenvalues_2d  radius_knn_2d     density_2d    eigenvalue1  \
count            753876.000000  753876.000000  753876.000000  753876.000000
mean                  0.411878       0.935126      44.341698       0.694408
std                   0.262245       0.458810     467.636812       0.139602
min                   0.000014       0.014142       0.067502       0.351351
25%                   0.177836       0.650000       4.369688       0.582220
50%                   0.406867       0.870919       8.393141       0.664603
75%                   0.628703       1.207021      15.067924       0.797438
max                   0.999513       9.711364   31830.989300       0.999530

         eigenvalue2    eigenvalue3  eigenvalue2D1  eigenvalue2D2  \
count  753876.000000  753876.000000  753876.000000  753876.000000
mean        0.273253       0.032339       0.733804       0.266196
std         0.127630       0.049539       0.139479       0.139479
min         0.000380       0.000007       0.500122       0.000014
25%         0.178383       0.000844       0.613985       0.150985
50%         0.291794       0.004596       0.710799       0.289201
75%         0.378779       0.049049       0.849015       0.386015
max         0.499762       0.307947       0.999986       0.499878

               class
count  753876.000000
mean        3.937998
std         2.642773
min         0.000000
25%         2.000000
50%         3.000000
75%         6.000000
max         8.000000

[8 rows x 24 columns]
```

3.1 Height

As a first feature, we take a look at the elevation (absolute height, z-coordinate) stored with the points. We can again use the **describe()** method of pandas, which is now applied to a column of type Series.

[24]:
```python
df['absolute_height'].describe()
```

[24]:
```
count    753876.000000
mean        268.098009
std           5.963596
min         250.600000
25%         264.300000
50%         267.830000
75%         271.840000
max         289.910000
Name: absolute_height, dtype: float64
```

Assuming that there might be outliers to both sides, let us define maybe more interesting percentiles with the **describe()** method. In the following example, we want to additionally output the 1 and 99 percentile to see how much differences the lowest and highest 1% of the points make.

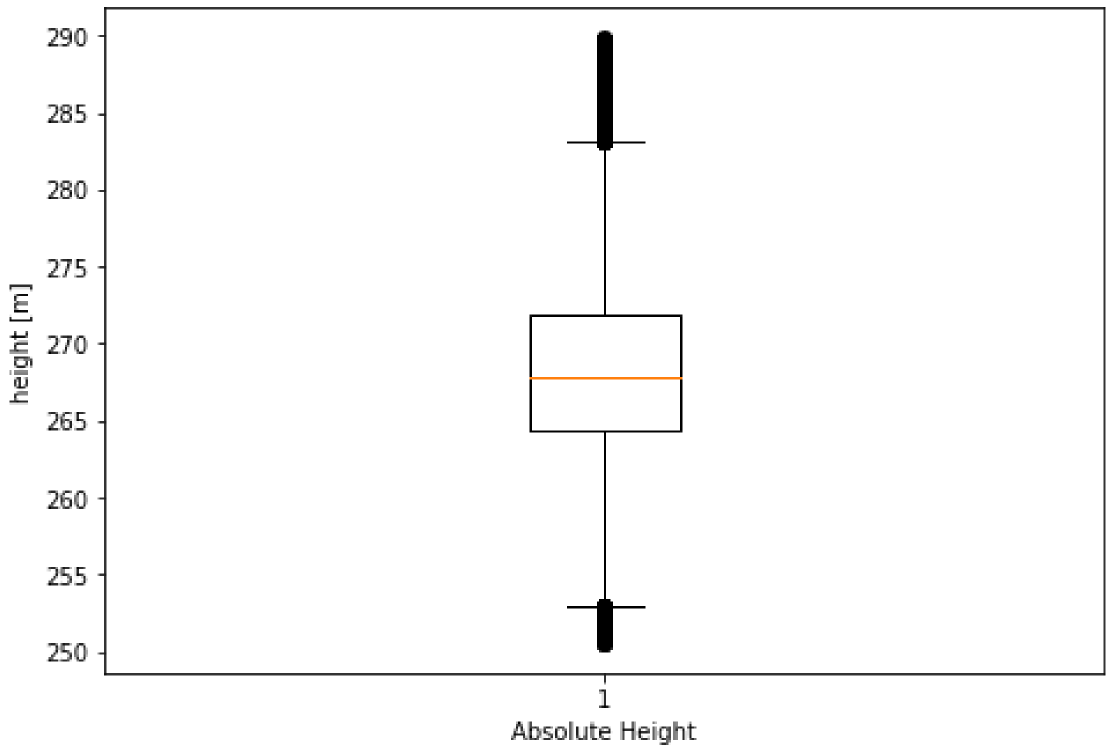
[25]:
```
df['absolute_height'].describe(percentiles=[0.01, 0.25, 0.5, 0.75, 0.99])
```

[25]:
```
count    753876.000000
mean        268.098009
std           5.963596
min         250.600000
1%          254.240000
25%         264.300000
50%         267.830000
75%         271.840000
99%         282.250000
max         289.910000
Name: absolute_height, dtype: float64
```

We can either convert the pandas DataFrame into a NumPy array with the method **to_numpy()** to use with matplotlib.

[26]:
```
heights = df['absolute_height'].to_numpy()

fig = plt.figure()

ax = fig.add_axes([0, 0, 1, 1])
ax.boxplot(heights)
ax.set_xlabel('Absolute Height')
ax.set_ylabel('height [m]')
```

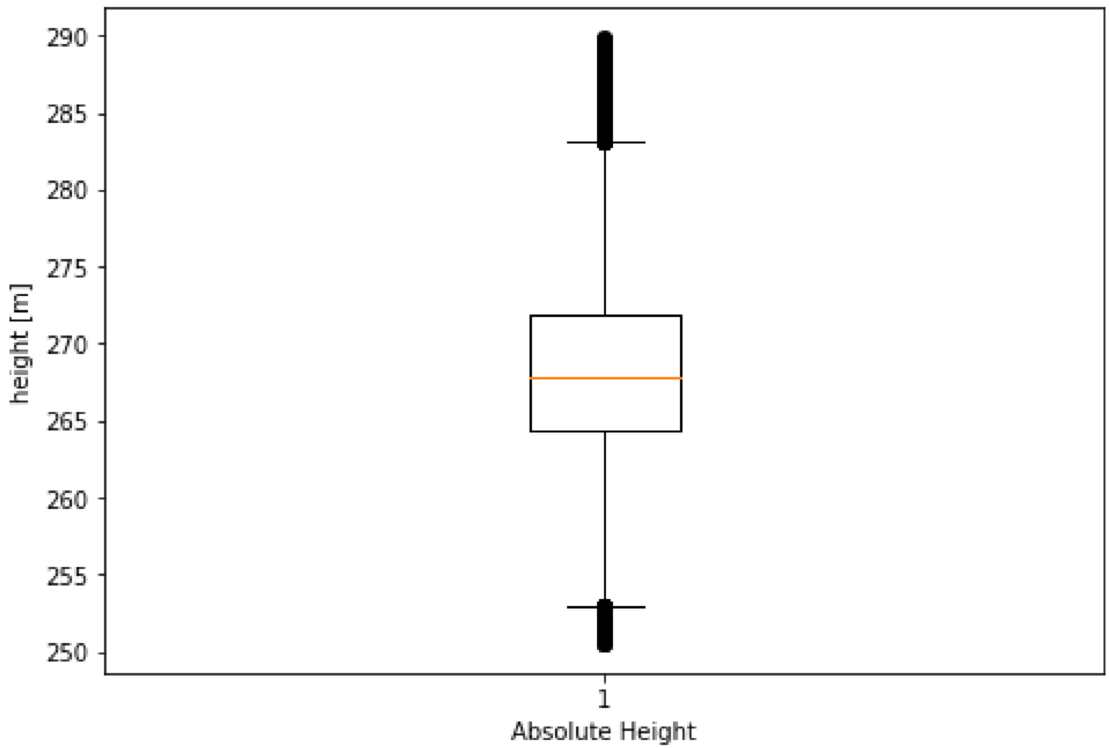
[26]:
```
Text(0, 0.5, 'height [m]')
```



Or use directly a column of the pandas DataFrame.

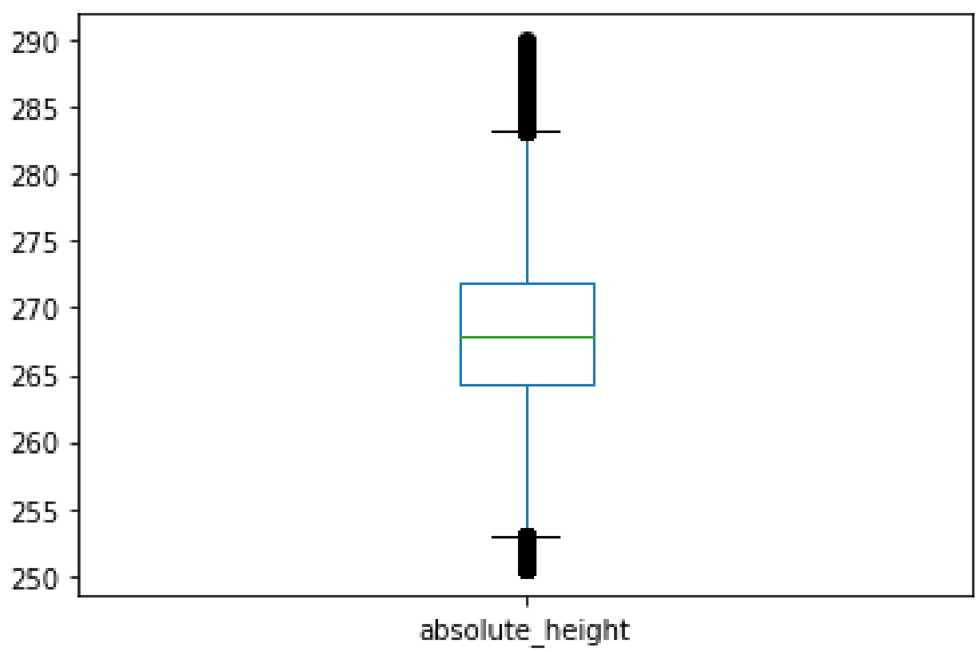
[27]:
```
fig = plt.figure()

ax = fig.add_axes([0, 0, 1, 1])
ax.boxplot(df['absolute_height'])
ax.set_xlabel('Absolute Height')
ax.set_ylabel('height [m]')
```

[27]: Text(0, 0.5, 'height [m]')



Pandas also offers convenient methods that can be directly called on a column of a DataFrame (which is an object of class Series).

[28]:
```
ax = df['absolute_height'].plot.box()
```



Next, we plot a combination of a box plot and a histogram of heights.

The bin widths are provided with the **bins** parameter as a NumPy array. Here, an array is constructed with the function **arange()** that contains values between start (first parameter), end (second parameter), and with a step size (third parameter) of 0.5. The start and end of the range is the floor of the minimum value and the ceiling of the maximum value in the elevation (height) column.

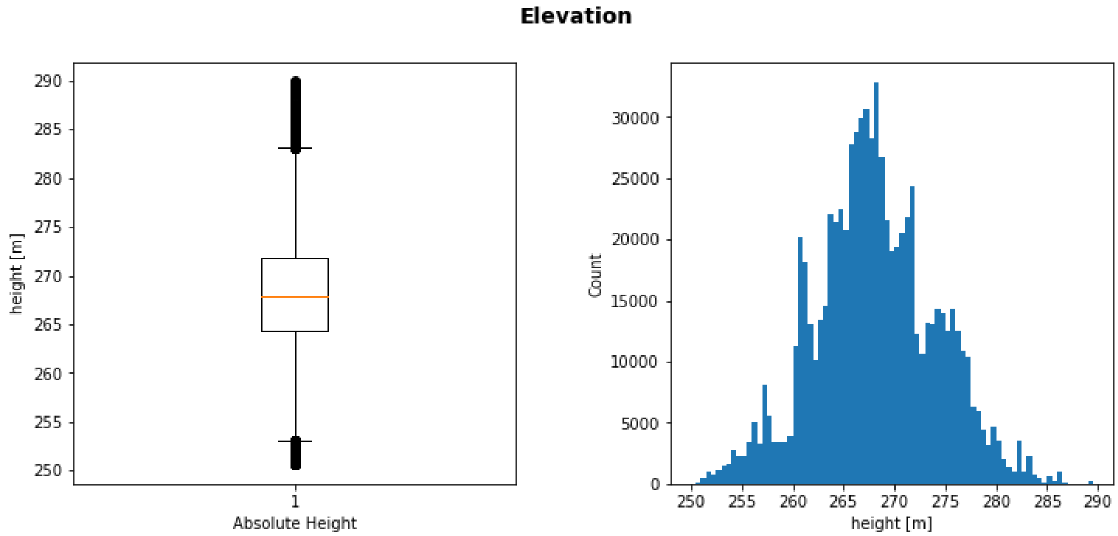
[29]:
```
np.arange(np.floor(np.min(heights)), np.ceil(np.max(heights)), 0.5)
```

[29]:
```
array([250. , 250.5, 251. , 251.5, 252. , 252.5, 253. , 253.5, 254. ,
       254.5, 255. , 255.5, 256. , 256.5, 257. , 257.5, 258. , 258.5,
       259. , 259.5, 260. , 260.5, 261. , 261.5, 262. , 262.5, 263. ,
       263.5, 264. , 264.5, 265. , 265.5, 266. , 266.5, 267. , 267.5,
       268. , 268.5, 269. , 269.5, 270. , 270.5, 271. , 271.5, 272. ,
       272.5, 273. , 273.5, 274. , 274.5, 275. , 275.5, 276. , 276.5,
       277. , 277.5, 278. , 278.5, 279. , 279.5, 280. , 280.5, 281. ,
       281.5, 282. , 282.5, 283. , 283.5, 284. , 284.5, 285. , 285.5,
       286. , 286.5, 287. , 287.5, 288. , 288.5, 289. , 289.5])
```

These bins are then provided with the parameter **bins** of the **hist()** method to construct a histogram.

[30]:
```
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
fig.subplots_adjust(wspace=0.35)
fig.suptitle('Elevation', fontsize=14, fontweight='semibold')

ax[0].boxplot(heights)
ax[0].set_xlabel('Absolute Height')
ax[0].set_ylabel('height [m]')

ax[1].hist(heights, bins=np.arange(np.floor(np.min(heights)), np.ceil(np.
 ↪max(heights)), 0.5))
ax[1].set_xlabel('height [m]')
ax[1].set_ylabel('Count')
```

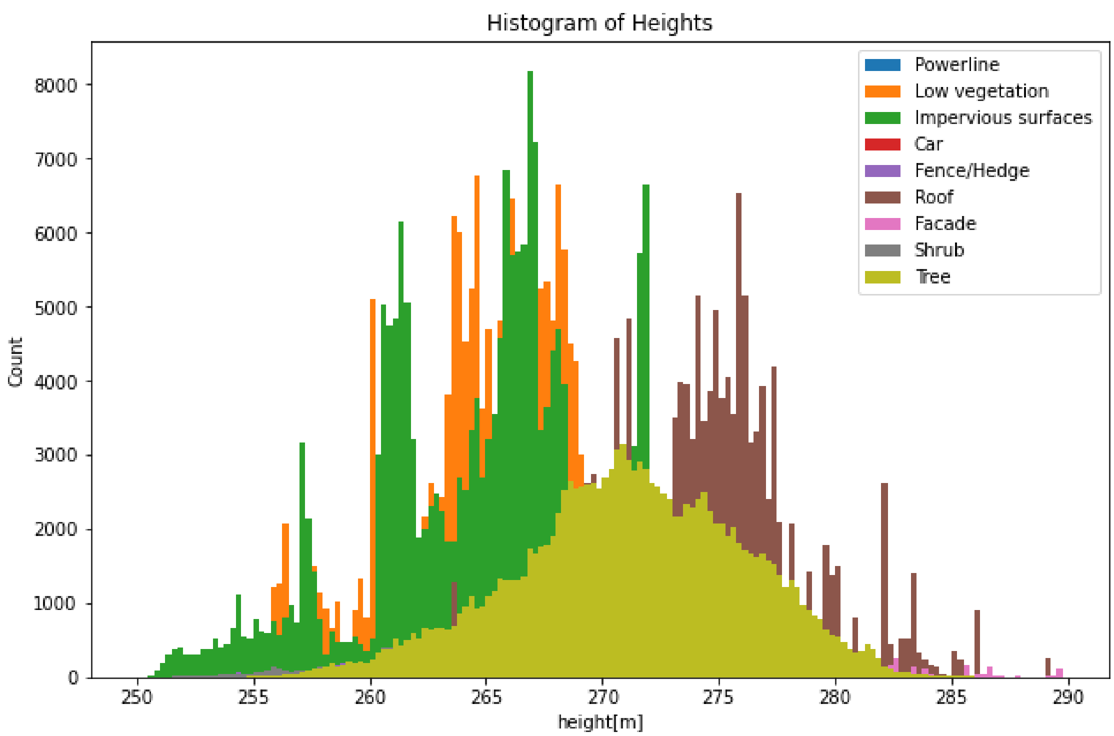
[30]:
```
Text(0, 0.5, 'Count')
```



And finally, a histogram of heights per classes as shown above for the intensity values.

[31]:
```
fig = plt.figure(figsize=(8, 5))
ax = fig.add_axes((0, 0, 1, 1))

bins = np.arange(np.floor(np.min(heights)), np.ceil(np.max(heights)), 0.25)

for i,n in enumerate(class_names):
    ax.hist(heights[np.where(classes == i)], bins=bins)

ax.legend(labels=class_names)
ax.set_title('Histogram of Heights')
ax.set_xlabel('height[m]')
ax.set_ylabel('Count')
```

[31]:
```
Text(0, 0.5, 'Count')
```



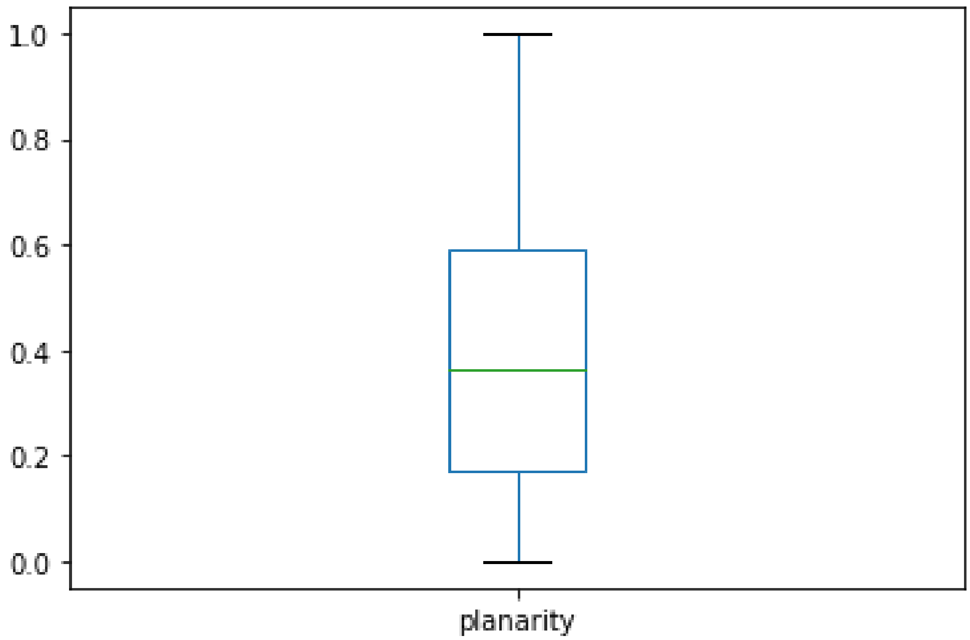
Take a look at these heights and their classes and think if they make sense or not. It might also help to take a look at the point cloud in a 3D visualization, e.g., using CloudCompare.

3.2 Planarity

Next, we turn to the feature of planarity. One could assume that impervious ground and roof surfaces have a high planarity and that vegetation has a rather low planarity.

We start with a simple boxplot of planarity values, and for simplicity use the pandas method **box()**.

[32]:
```
ax = df['planarity'].plot.box()
```



Pandas also has a quick way to get the percentiles or quantiles with the **percentile()** or **quantile()** method, respectively.

[33]:
```
df['planarity'].quantile([0.0, 0.25, 0.5, 0.75, 1.0])
```

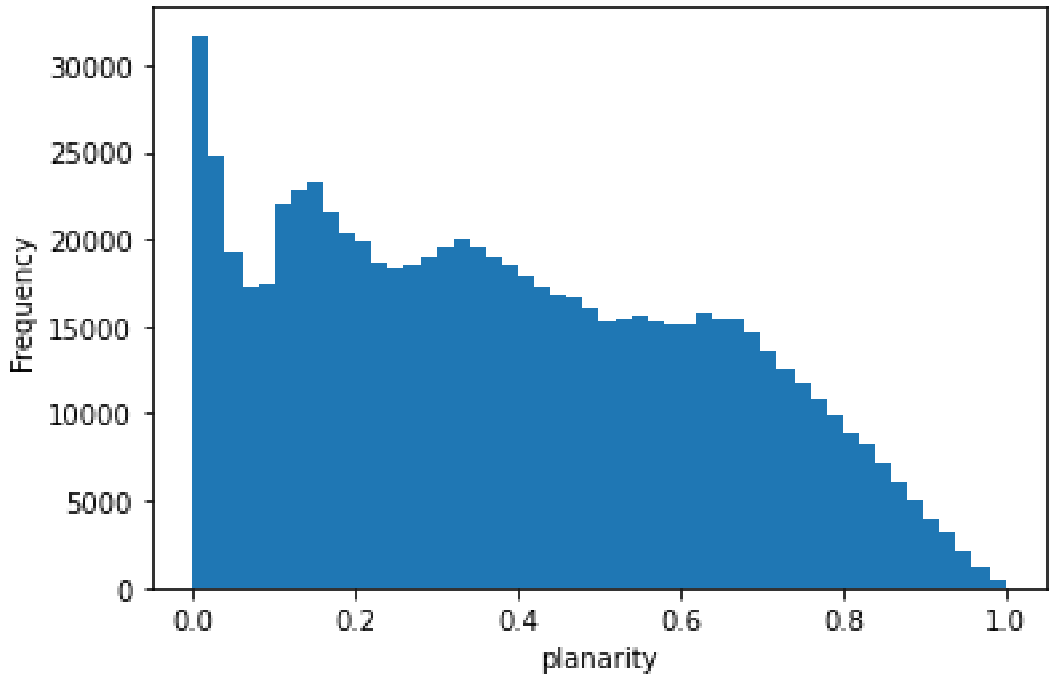
[33]:
```
0.00    0.000062
0.25    0.169167
0.50    0.363175
0.75    0.590015
1.00    0.999046
Name: planarity, dtype: float64
```

We can now see where most of the values are located. The median is not quite at 0.5, but also not that far off. There are hardly any perfect planarity values.

Let us also plot a histogram of planarity values. Note that the pandas method returns an Axes object, which we can use to set some properties of the plot.

[34]:
```
ax = df['planarity'].plot.hist(bins=50)
ax.set_xlabel('planarity')
```

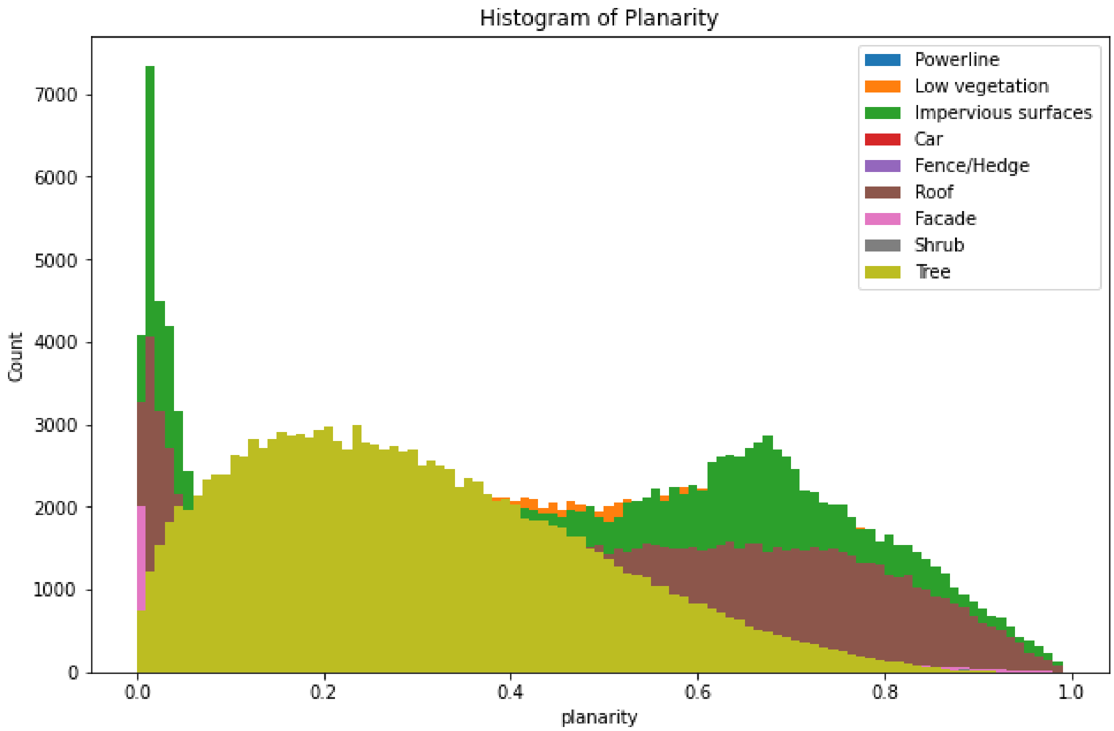
[34]:
```
Text(0.5, 0, 'planarity')
```



Next, a histogram plot per class is generated, kind of like we did with the bar plots.

[35]:
```
planarity = df['planarity'].to_numpy()

fig = plt.figure(figsize=(8, 5))
ax = fig.add_axes((0, 0, 1, 1))

bins = np.arange(0.0, 1.0, 0.01)

for i,n in enumerate(class_names):
    ax.hist(planarity[np.where(classes == i)], bins=bins)

ax.legend(labels=class_names)
ax.set_title('Histogram of Planarity')
ax.set_xlabel('planarity')
ax.set_ylabel('Count')
```

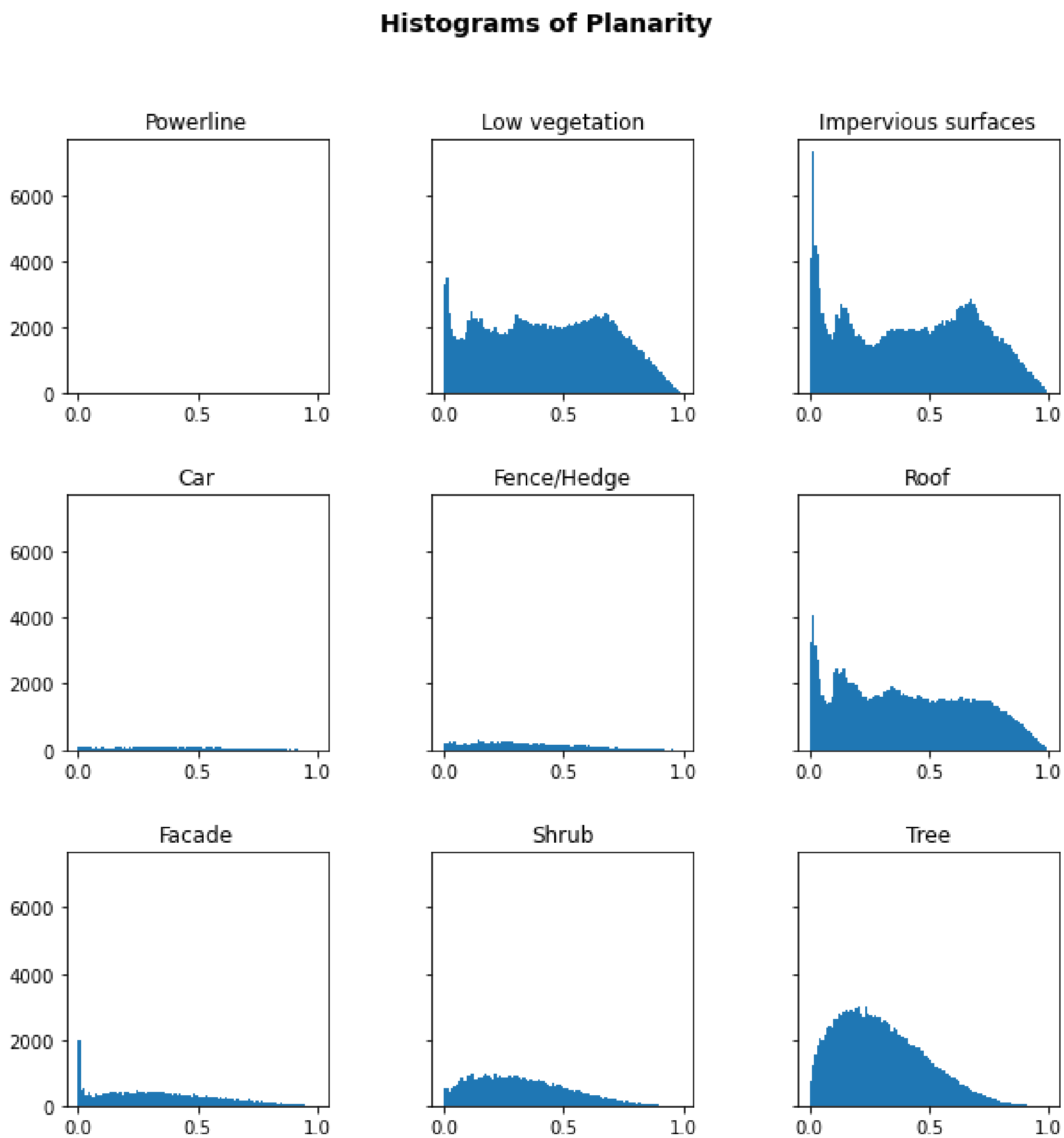
[35]: Text(0, 0.5, 'Count')



Unfortunately, the tree class occludes almost everything. But it can be seen that the tree class has mostly low planarity.

So let us better plot all the classes on their own.

[36]:
```
fig, ax = plt.subplots(3, 3, sharex='none', sharey='all', figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
fig.suptitle('Histograms of Planarity', fontsize=14, fontweight='bold')

for i,n in enumerate(class_names):
    ax[i//3, i%3].hist(planarity[np.where(classes == i)], bins=np.arange(0.0, 1.
↪0, 0.01))
    ax[i//3, i%3].set(title=class_names[i])
```



As a conclusion, the tree, shrub, fence/hedge classes clearly have a tendency to lower planarity. And although impervious surfaces and roofs have more points with high planarity values, they are nevertheless rather homogeneously distributed. Perfectly planar points are very seldom, which might have to do with the low density of the point cloud itself, and the comparably large neighborhood of 20 points from which the feature is calculated. All 20 points would need to be located on a plane for a very high planarity value.

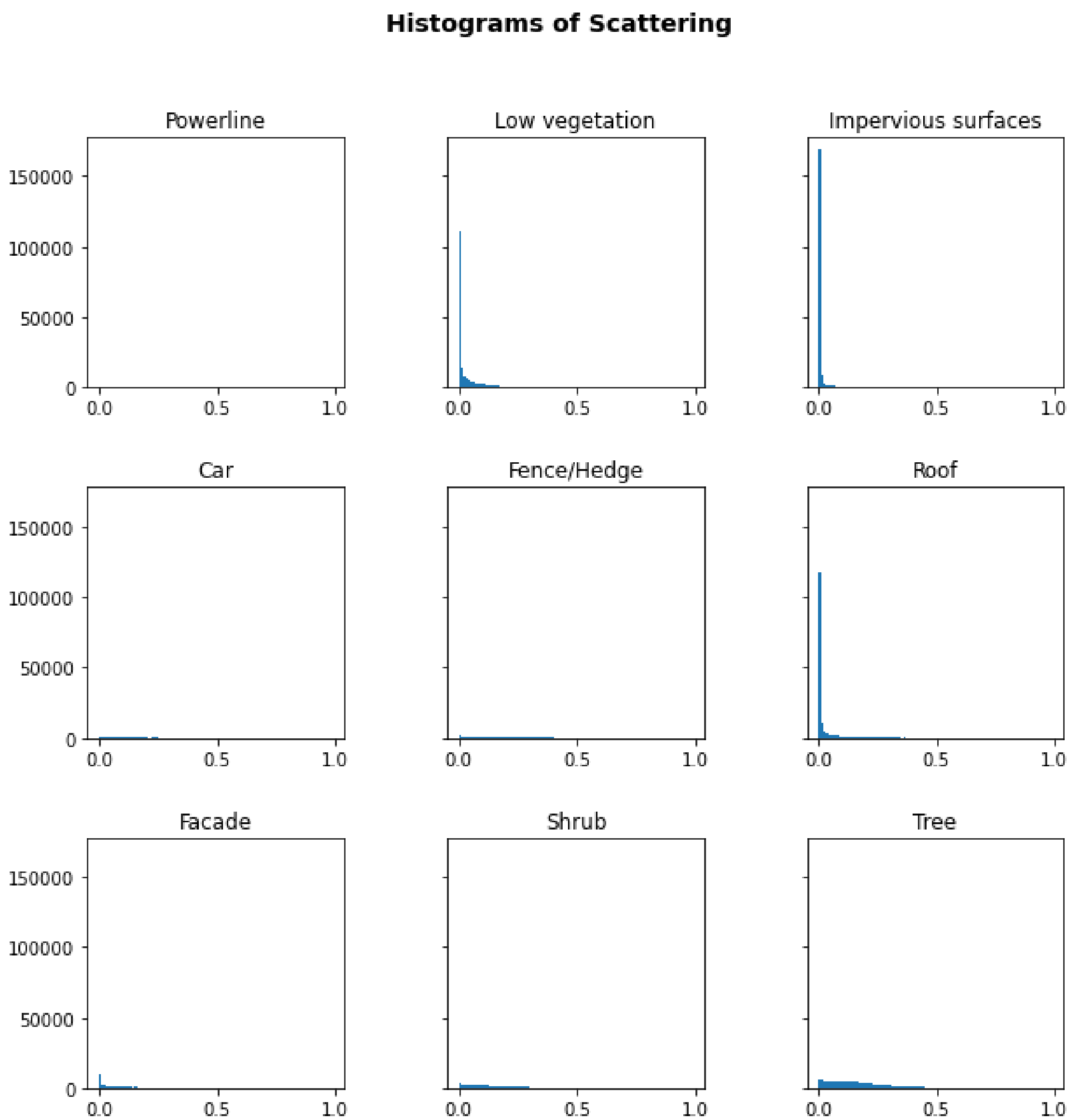
The point cloud has a point density of approximately 4 points per square meter. And a group of 20 points therefore take over 4 square meters. There are not that many points with that many neighbor points that make flat areas.

By the way, to better see the histograms of classes like power line, car, and fence/hedge, you can set the **sharey** parameter to none in the above figure.

4 Scattering

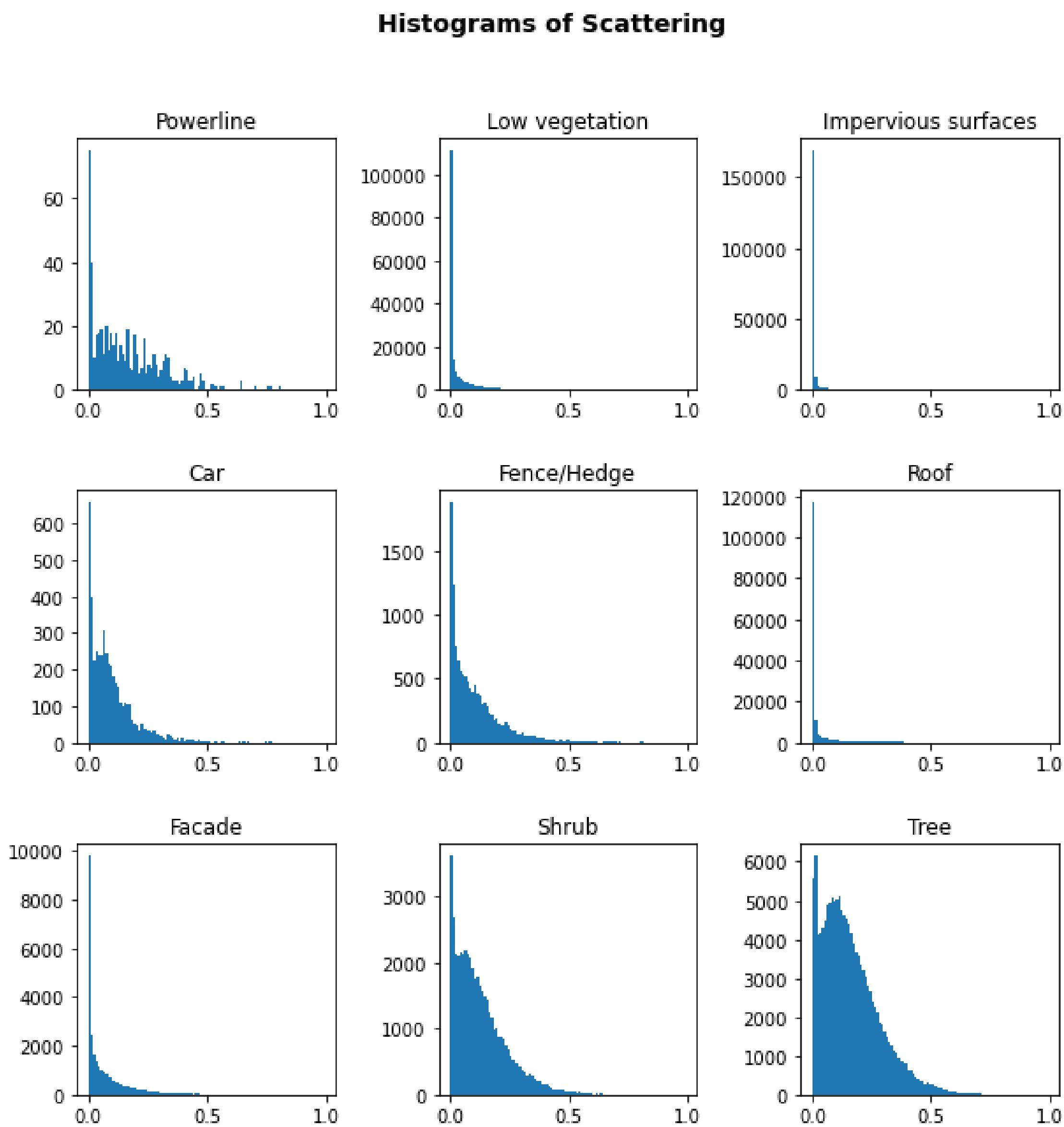
For the feature, scattering, we would expect a reversed image. So, let us start with the histograms per classes.

[37]:
```
scattering = df['scattering'].to_numpy()

fig, ax = plt.subplots(3, 3, sharex='none', sharey='all', figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
fig.suptitle('Histograms of Scattering', fontsize=14, fontweight='bold')

for i,n in enumerate(class_names):
    ax[i//3, i%3].hist(scattering[np.where(classes == i)], bins=np.arange(0.0,␣
↪1.0, 0.01))
    ax[i//3, i%3].set(title=class_names[i])
```



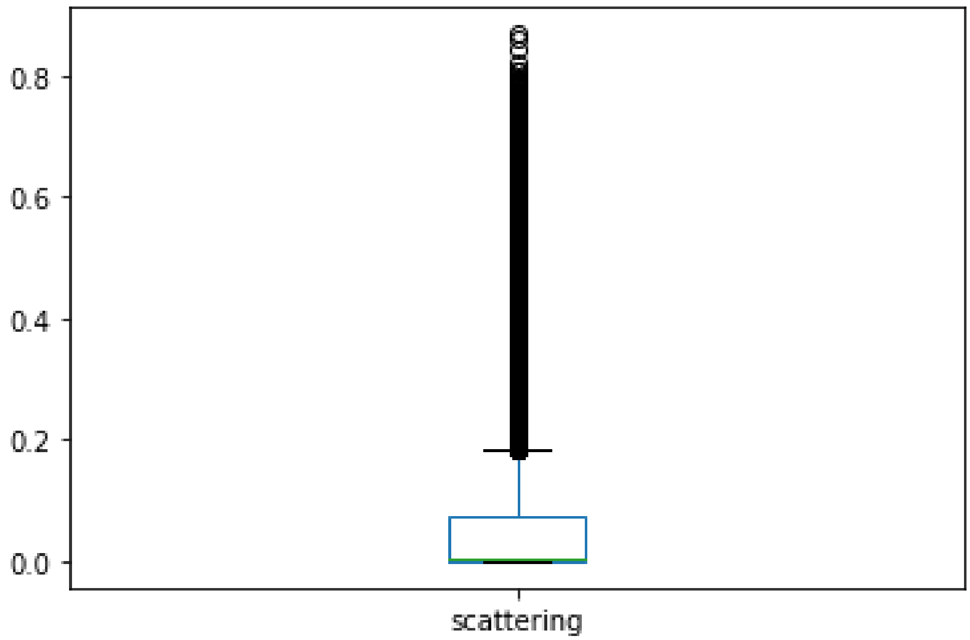
That is an unexpected outcome. It can be improved by turning off to option to share the y-scale.

[38]:
```
fig, ax = plt.subplots(3, 3, sharex='none', sharey='none', figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
fig.suptitle('Histograms of Scattering', fontsize=14, fontweight='bold')

for i,n in enumerate(class_names):
    ax[i//3, i%3].hist(scattering[np.where(classes == i)], bins=np.arange(0.0,␣
↪1.0, 0.01))
    ax[i//3, i%3].set(title=class_names[i])
```



To investigate this further, let us take a look at a box plot.

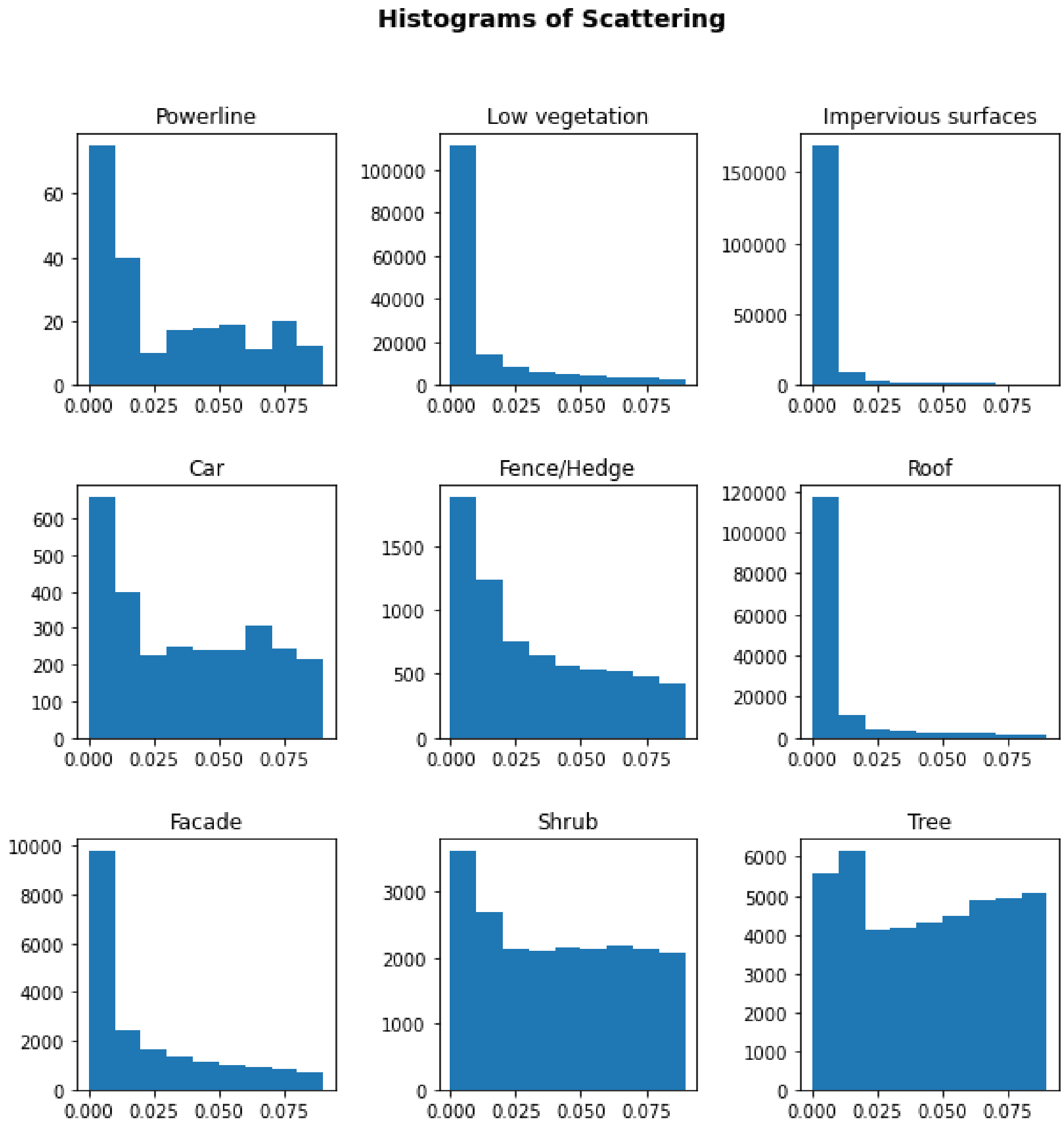
[39]:
```
ax = df['scattering'].plot.box()
```



It seems that most points have a low scattering value. But at least from the second histogram plot, we can see that tree, shrub, and fence/hedge do actually have some noticeable scattering that is higher than 0.025. We can concentrate on the values with low scattering (<0.01) and have another plot.

Notice how the two conditions are combined with the **logical_and()** function of NumPy. Each condition ("classes == i" and "scattering < 0.1") gives an array of Boolean values, that have the value true where the condition is true. These two arrays of Boolean values are then combined to a single Boolean array, which can ten be used in where() to find the indices where both conditions hold.

[40]:
```
fig, ax = plt.subplots(3, 3, sharex='none', sharey='none', figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
fig.suptitle('Histograms of Scattering', fontsize=14, fontweight='bold')

for i,n in enumerate(class_names):
    ax[i//3, i%3].hist(scattering[np.where(np.logical_and(classes == i,␣
 ↪scattering < 0.1))], bins=np.arange(0.0, 0.1, 0.01))
    ax[i//3, i%3].set(title=class_names[i])
```

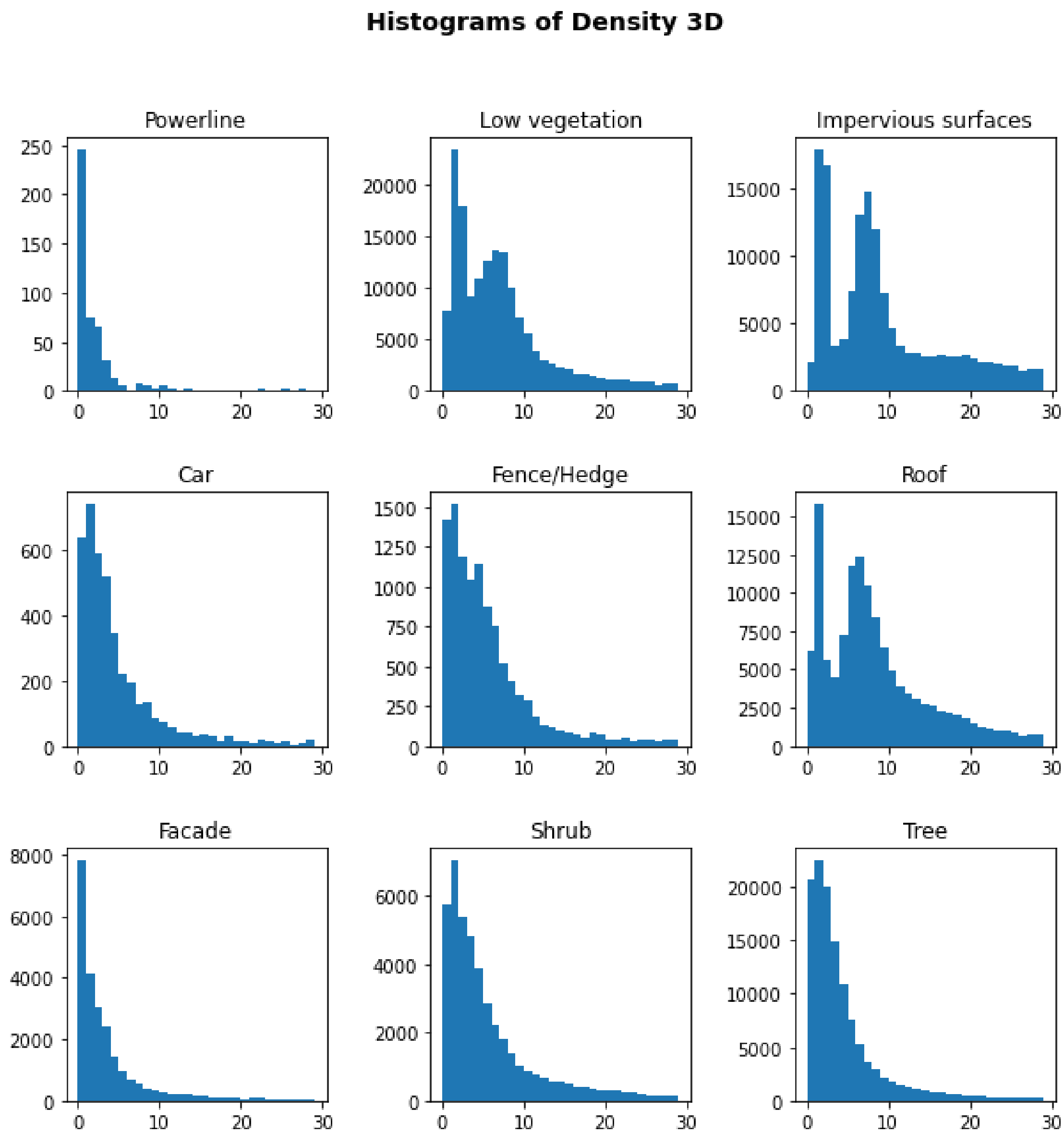


From this we notice that impervious surfaces and roof have a very low scattering as expected, because they should mostly consists of flat surfaces. (Although a more pronounced value range would have been nicer.) Because only few beams of an aerial laser scanner hits building facades, these points are also rather scattered instead of planar. This is to be expected.

4.1 Density 3D

As a last feature, we take a look at the 3d density of the k nearest neighbor points. We limit the range of bins to be between 0 and 30 as most values are in this range.

[41]:
```
density3d = df['density_3d'].to_numpy()

fig, ax = plt.subplots(3, 3, sharex='none', sharey='none', figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
fig.suptitle('Histograms of Density 3D', fontsize=14, fontweight='bold')

for i,n in enumerate(class_names):
    ax[i//3, i%3].hist(density3d[np.where(classes == i)], bins=np.arange(0.0,␣
↪30.0, 1.0))
    ax[i//3, i%3].set(title=class_names[i])
```



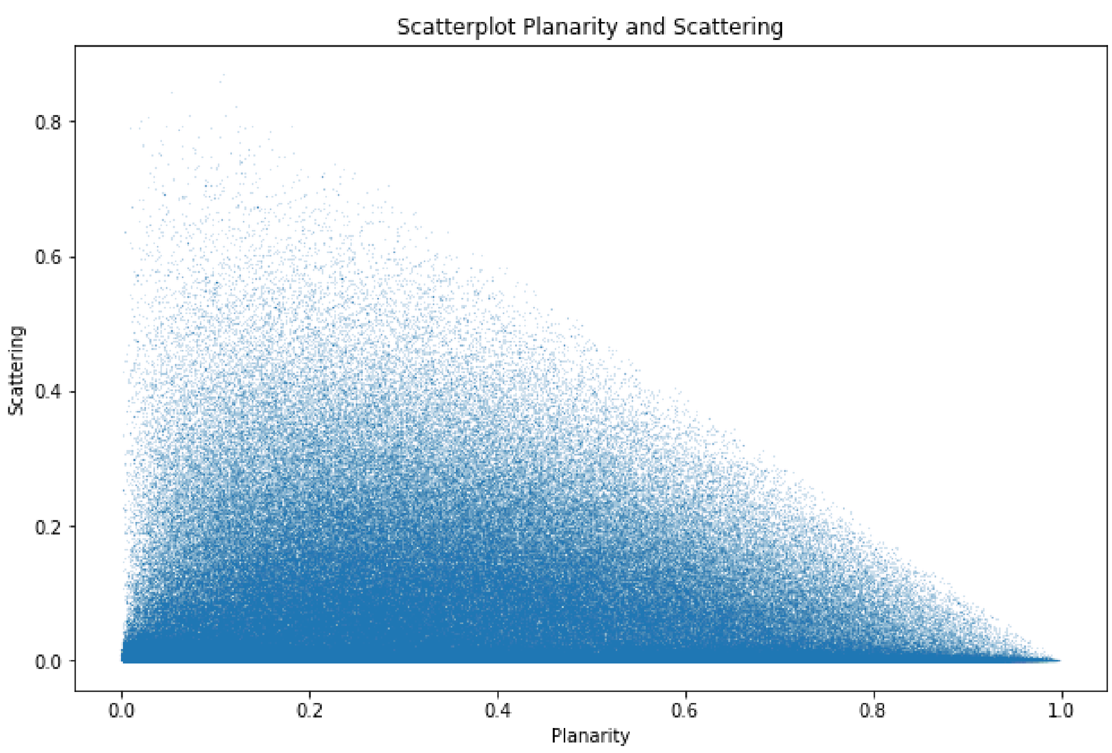
It is interesting to note that vegetation classes seem to have more values close to 0 and impervious surfaces and roof have higher values.

5 Scatterplots

Before we come to an end, let us also do a few scatterplots in the end. A scatterplot is constructed with the **scatter()** method of an Axes object and takes two NumPy arrays as input. The **s** parameter gives the marker size. As there are many points, a very small marker size is used.

[42]:
```
planarity = df['planarity'].to_numpy()
scattering = df['scattering'].to_numpy()

fig = plt.figure(figsize=(8, 5))
ax = fig.add_axes((0, 0, 1, 1))

ax.scatter(planarity, scattering, s=0.005)

ax.set_title('Scatterplot Planarity and Scattering')
ax.set_xlabel('Planarity')
ax.set_ylabel('Scattering')
```

[42]:
```
Text(0, 0.5, 'Scattering')
```



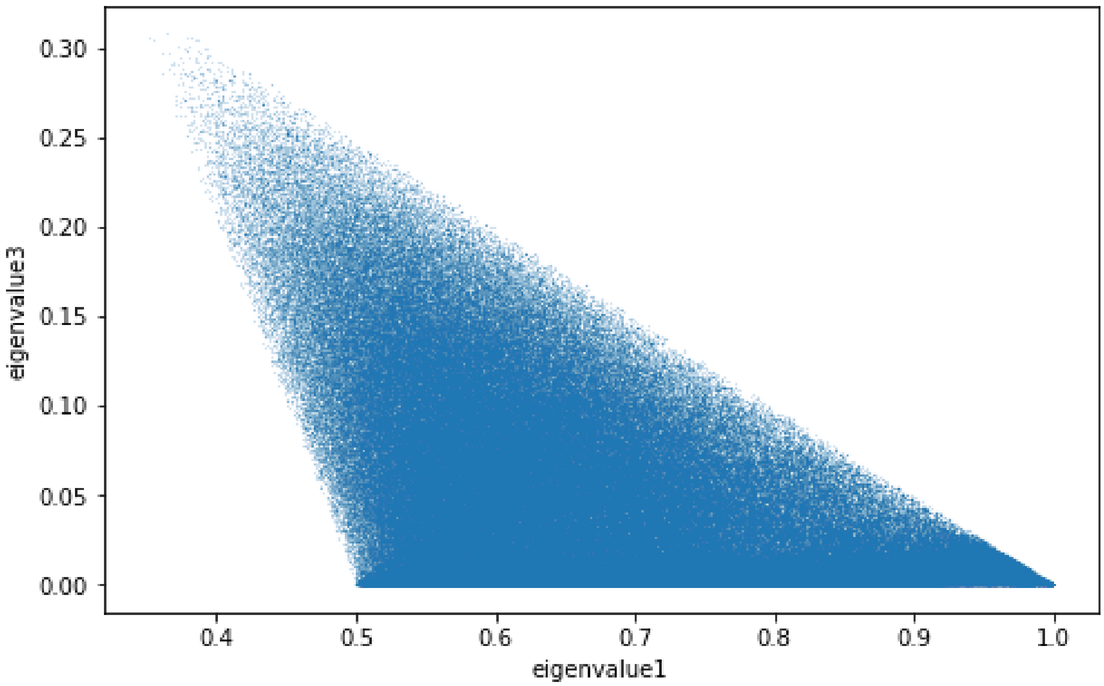
Because most scattering values are rather low, most points gather at the bottom of the plot.

Let us try the eigenvalues 1 and 3.

Pandas has a nice interface where we only need to provide the two column names of the features to plot.

[43]:
```
df.plot.scatter(x='eigenvalue1', y='eigenvalue3', figsize=(8, 5), s=0.005)
```

[43]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4438216650>



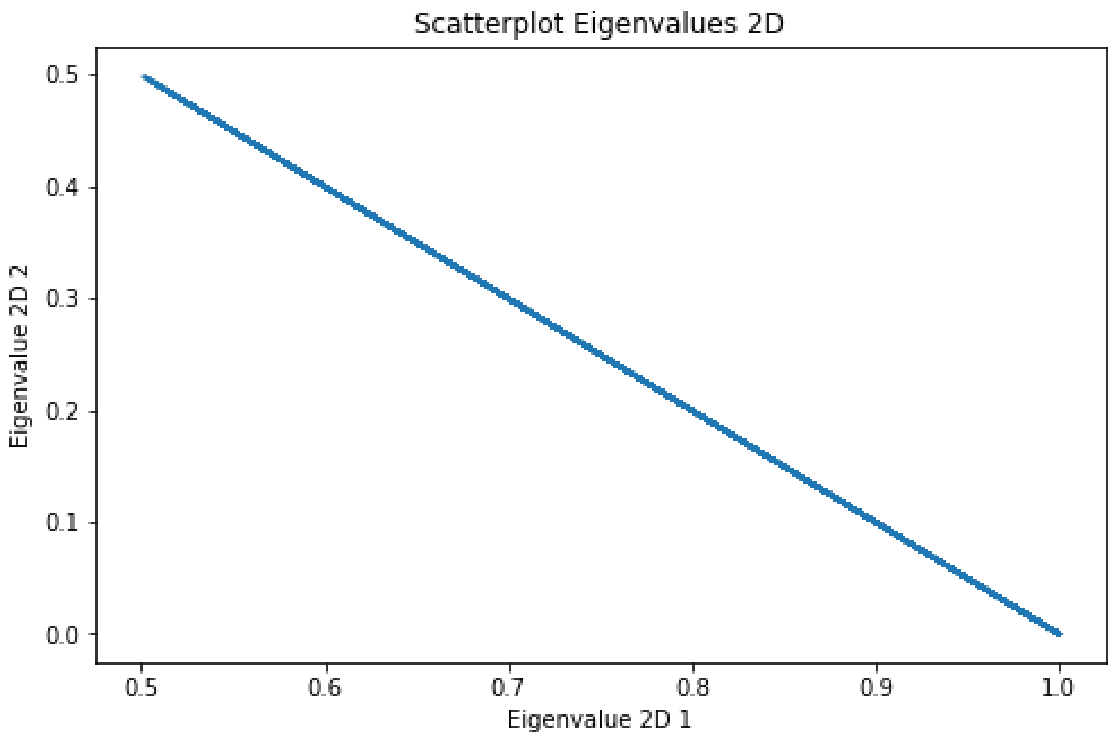
As the eigenvalues are dependent on each other, we see some negative correlation, but there is also some differences.

Even more prominent is the correlation of the 2D eigenvalues as they are even defined accordingly.

Since the **scatter()** method of pandas returns an Axes object, we can change the plot properties accordingly.

[44]:
```
ax = df.plot.scatter(x='eigenvalue2D1', y='eigenvalue2D2', figsize=(8, 5), s=0.
 ↪005)

ax.set_title('Scatterplot Eigenvalues 2D')
ax.set_xlabel('Eigenvalue 2D 1')
ax.set_ylabel('Eigenvalue 2D 2')
```

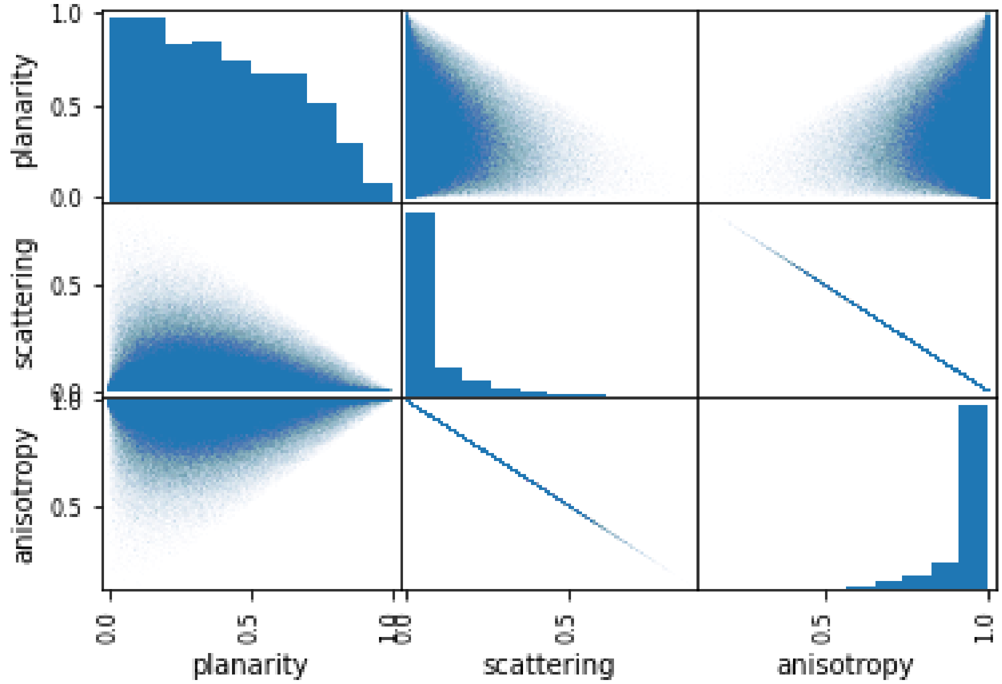
[44]: Text(0, 0.5, 'Eigenvalue 2D 2')



Pandas provides also a **scatter_matrix()** function, although we need to provide a DataFrame as first argument.

[45]:
```
cutout = df[['planarity', 'scattering', 'anisotropy']]
pd.plotting.scatter_matrix(cutout, s=0.1)
```

[45]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f4438969ad0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4437341c90>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4438a3ddd0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f4437857210>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f442e9f79d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f442e3c4d50>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f442dbd94d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f442df55a90>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f442df55ad0>]],
      dtype=object)
```



Note that the diagonals are not scatterplots, as they would be scatterplots of the feature with themselves, but rather histograms of the respective features are shown. (As an alternative, kernel density estimations can also be displayed in the diagonal.)

6 Final words

Feel free to explore further features on your own in the same or similar manner from what is shown above.

You can also try a different k in the k nearest neighbor query when calculating the features and see if the features look different.

There are many more ways to customize plots in matplotlib. We leave it to you to further explore them.