PyMinimax

Release 0.1.2

beginnerSC

Aug 16, 2023

TABLE OF CONTENTS

1 Quick Start **3**
- 1.1 Installation . . . 3
- 1.2 Usage . . . 3
- 1.3 Getting Prototypes . . . 5
- 1.4 See Also . . . 6

2 Examples **7**
- 2.1 Random Points in 2D . . . 7
- 2.2 Hand-Written Digits . . . 11

3 Performance **17**

4 API Reference **19**

5 Reference **21**

Index **23**

Package Status

PyMinimax is a Python implementation of Bien and Tibshirani's paper "Hierarchical Clustering With Prototypes via Minimax Linkage"[1]. This is an agglomerative hierarchical clustering algorithm available in the protoclust R package[2] but not currently in SciPy nor scikit-learn. It has a great advantage over classical hierarchical clustering methods that in each cluster one prototype is selected from the original data. Thus the results are better interpretable.

PyMinimax has a SciPy-compatible API. Users who are familiar with the scipy.cluster.hierarchy module will find it easy to use.

[1] J. Bien and R. Tibshirani. Hierarchical clustering with prototypes via minimax linkage. Journal of the American Statistical Association, 106(495), 2011, 1075-1084.

[2] J. Bien and R. Tibshirani. Package 'protoclust', 2015.

CHAPTER

ONE

QUICK START

This page is generated by a Jupyter notebook which can be opened and run in Binder or Google Colab by clicking on the above badges. **To run it in Google Colab, you need to install PyMinimax in Colab first.**

1.1 Installation

The recommended way to install PyMinimax is using pip:

```
[ ]: pip install pyminimax
```

Or if you have installed PyMinimax before, please update to the latest version:

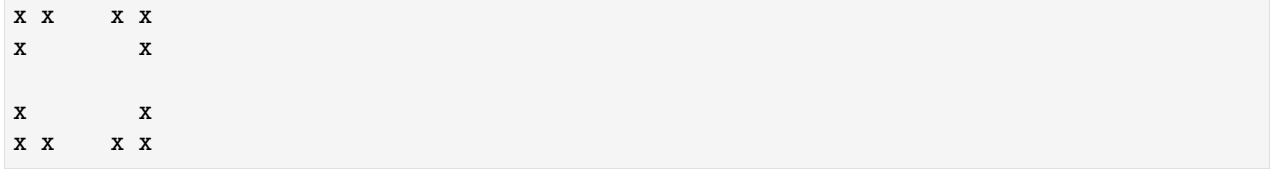
```
[ ]: pip install --upgrade pyminimax
```

PyMinimax runs on any platform with Python 3 and SciPy installed.

1.2 Usage

The most important function in PyMinimax is `pyminimax.minimax`, and by default its usage is the same as the hierarchical clustering methods in SciPy, say `scipy.cluster.hierarchy.complete`. Here we demonstrate with an example from the SciPy documentation. First consider a dataset of $n = 12$ points:

```
[1]: X = [[0, 0], [0, 1], [1, 0],
          [0, 4], [0, 3], [1, 4],
          [4, 0], [3, 0], [4, 1],
          [4, 4], [3, 4], [4, 3]]
```



The minimax function takes a flattened distance matrix of the data as an argument, which can be computed by `scipy.spatial.distance.pdist`. By default, the return value of minimax has the same format as that of `scipy.cluster.hierarchy.linkage`. This is an $(n-1)$ by 4 matrix keeping the clustering result, called the *linkage matrix*. A detailed explanation of its format can be found in the SciPy documentation.

[2]:
```
from pyminimax import minimax
from scipy.spatial.distance import pdist

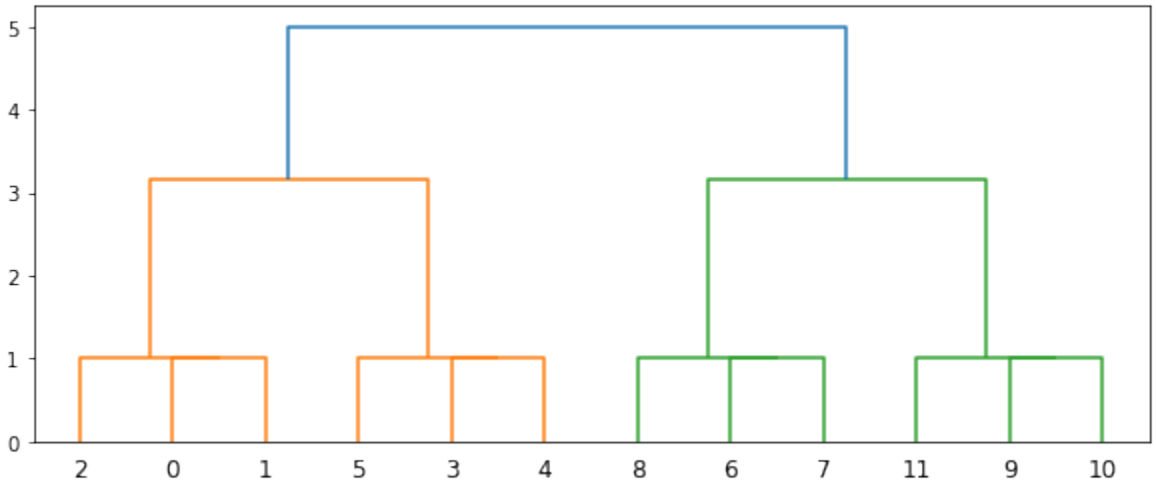
Z = minimax(pdist(X))
Z
```

[2]:
```
array([[ 0.        ,  1.        ,  1.        ,  2.        ],
       [ 2.        , 12.        ,  1.        ,  3.        ],
       [ 3.        ,  4.        ,  1.        ,  2.        ],
       [ 6.        ,  7.        ,  1.        ,  2.        ],
       [ 5.        , 14.        ,  1.        ,  3.        ],
       [ 8.        , 15.        ,  1.        ,  3.        ],
       [ 9.        , 10.        ,  1.        ,  2.        ],
       [11.        , 18.        ,  1.        ,  3.        ],
       [13.        , 16.        ,  3.16227766,  6.        ],
       [17.        , 19.        ,  3.16227766,  6.        ],
       [20.        , 21.        ,  5.        , 12.        ]])
```

Given the linkage matrix, one can then utilize the methods in SciPy to present the clustering result in a more readable manner. Below are examples applying `dendrogram` and `fcluster`.

[3]:
```
from scipy.cluster.hierarchy import dendrogram
from matplotlib import pyplot as plt

fig = plt.figure(figsize=(10, 4))
dendrogram(Z)
plt.show()
```



[4]:
```
from scipy.cluster.hierarchy import fcluster

fcluster(Z, t=1.8, criterion='distance')
```

[4]:
```
array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4], dtype=int32)
```

The above result says that, cutting the dendrogram at 1.8 threshold, the data has 4 clusters, with the first 3 points being in the first cluster, and the following 3 in the second cluster, and so on.

1.3 Getting Prototypes

A unique advantage of minimax linkage hierarchical clustering is that in each cluster one prototype is selected from the original data. Thus the resulting clusters are better interpretable. Starting 0.1.2, PyMinimax can also compute those prototypes. Below we demonstrate how this can be done.

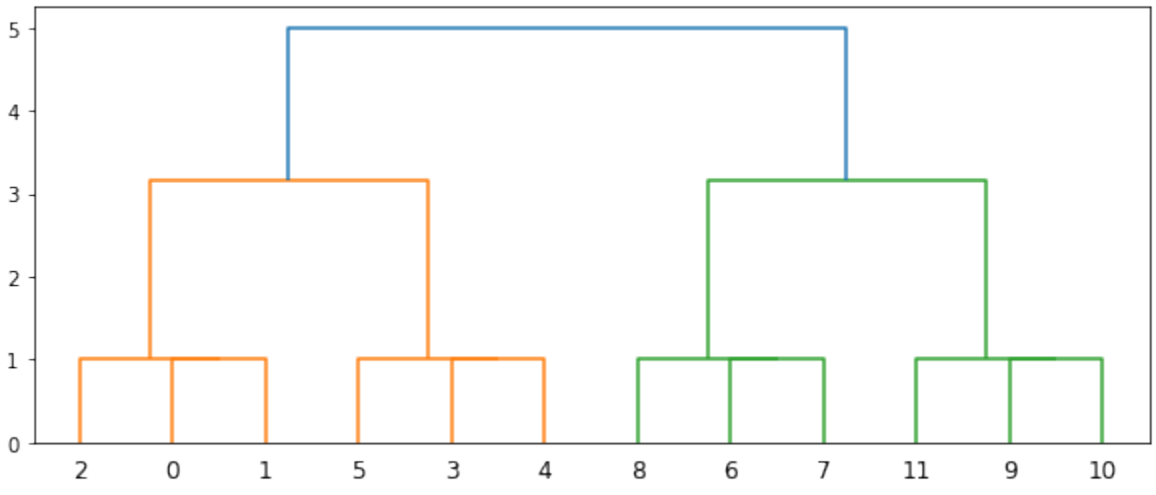
To obtain the prototypes, first we need an *extended linkage matrix* that contains prototypes information. This is an $(n-1)$ by 5 matrix, where the first 4 columns are in the same format as the standard linkage matrix and the 5th column keeps the indices of the prototypes of each merged cluster. The extended linkage matrix can be computed by `pyminimax.minimax` with `return_prototype=True`.

[5]:
```
Z_ext = minimax(pdist(X), return_prototype=True)
Z_ext
```

[5]:
```
array([[ 0.        ,  1.        ,  1.        ,  2.        ,  0.        ],
       [ 2.        , 12.        ,  1.        ,  3.        ,  0.        ],
       [ 3.        ,  4.        ,  1.        ,  2.        ,  3.        ],
       [ 6.        ,  7.        ,  1.        ,  2.        ,  6.        ],
       [ 5.        , 14.        ,  1.        ,  3.        ,  3.        ],
       [ 8.        , 15.        ,  1.        ,  3.        ,  6.        ],
       [ 9.        , 10.        ,  1.        ,  2.        ,  9.        ],
       [11.        , 18.        ,  1.        ,  3.        ,  9.        ],
       [13.        , 16.        ,  3.16227766,  6.        ,  1.        ],
       [17.        , 19.        ,  3.16227766,  6.        ,  8.        ],
       [20.        , 21.        ,  5.        , 12.        ,  1.        ]])
```

The 5-column extended linkage matrix is no longer SciPy-compatible. Any SciPy function that takes a linkage matrix will only work if we slice the extended linkage matrix to drop the 5th column, hence the `Z_ext[:, :4]` below.

[6]:
```
fig = plt.figure(figsize=(10, 4))
dendrogram(Z_ext[:, :4])
plt.show()
```



In PyMinimax, the prototypes are computed by `pyminimax.fcluster_prototype`. Its usage is exactly the same as `scipy.cluster.hierarchy.fcluster`, except

1. it takes the 5-column extended linkage matrix instead of a standard 4-column one, and
2. it returns information regarding clusters *and* prototypes.

[7]:
```
from pyminimax import fcluster_prototype

fcluster_prototype(Z_ext, t=1.8, criterion='distance')
```

[7]:
```
array([[1, 0],
       [1, 0],
       [1, 0],
       [2, 3],
       [2, 3],
       [2, 3],
       [3, 6],
       [3, 6],
       [3, 6],
       [4, 9],
       [4, 9],
       [4, 9]], dtype=int32)
```

The above result says that, cutting the dendrogram at 1.8 threshold, the data has 4 clusters. The first 3 points are in the first cluster and the prototype of this cluster is the 0th data point, the following 3 points are in the second cluster and the prototype of this cluster is the 3rd data point, and so on.

1.4 See Also

- scipy.cluster.hierarchy.complete
- scipy.cluster.hierarchy.dendrogram
- scipy.cluster.hierarchy.fcluster
- scipy.cluster.hierarchy.linkage
- scipy.spatial.distance.pdist

CHAPTER

TWO

EXAMPLES

It is recommended that you go through the quick start guide before reading this page.

This page is generated by a Jupyter notebook which can be opened and run in Binder or Google Colab by clicking on the above badges. **To run it in Google Colab, first you need to install PyMinimax and a newer version of scikit-learn in Colab:**

[ ]:
```
!pip install pyminimax scikit-learn==0.23
```

2.1 Random Points in 2D

In this example we perform minimax linkage clustering on a toy dataset of 20 random points in 2D:

[1]:
```
import numpy as np
from pandas import DataFrame

np.random.seed(0)
X = np.random.rand(20, 2)

DataFrame(X, columns=['x', 'y'])
```

[1]:

	x	y
0	0.548814	0.715189
1	0.602763	0.544883
2	0.423655	0.645894
3	0.437587	0.891773
4	0.963663	0.383442
5	0.791725	0.528895
6	0.568045	0.925597
7	0.071036	0.087129
8	0.020218	0.832620
9	0.778157	0.870012
10	0.978618	0.799159
11	0.461479	0.780529
12	0.118274	0.639921
13	0.143353	0.944669
14	0.521848	0.414662
15	0.264556	0.774234
16	0.456150	0.568434

(continues on next page)

(continued from previous page)

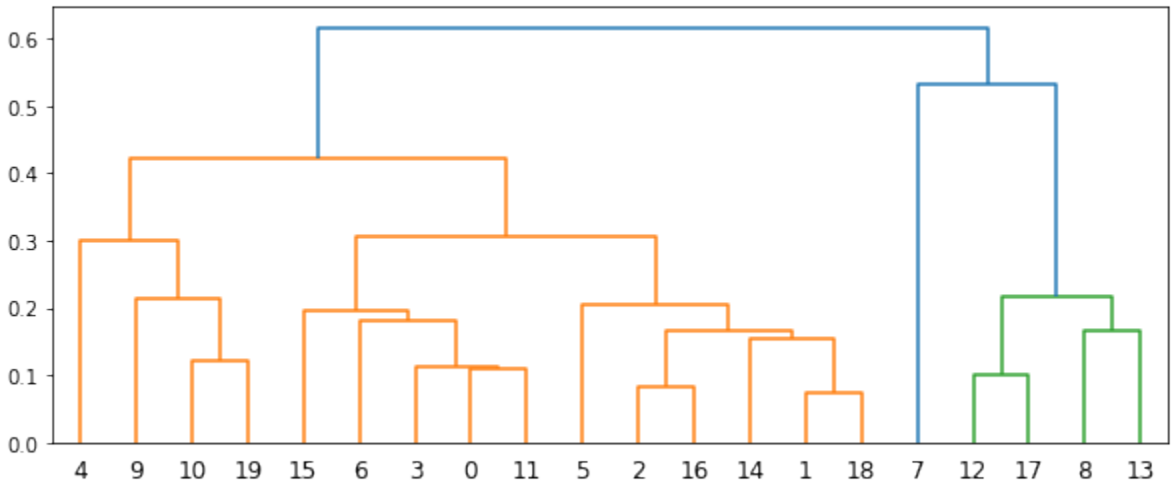
```
17  0.018790  0.617635
18  0.612096  0.616934
19  0.943748  0.681820
```

Below is the dendrogram.

[2]:
```
import matplotlib.pyplot as plt
from pyminimax import minimax
from scipy.spatial.distance import pdist
from scipy.cluster.hierarchy import dendrogram

Z = minimax(pdist(X), return_prototype=True)

plt.figure(figsize=(10, 4))
dendrogram(Z[:, :4])
plt.show()
```



A unique advantage of minimax linkage hierarchical clustering is that every cluster has a prototype selected from the original data. This is a representative data point of the cluster.

The threshold used to cut the dendrogram is also interpretable. Suppose the dendrogram is cut at threshold t, splitting the data into clusters $G_1, G_2, \ldots$ with corresponding prototypes $p_1, p_2, \ldots$. Then, for any i, all data points in G_i must be in the circle centered at p_i with radius t. That is, the distance from the prototype of a cluster to any data point in the same cluster must be less than or equal to t.

Here we draw the clusters and the circles for various thresholds. The data points at the center of the circles are the prototypes.

[3]:
```
import seaborn as sns
from pandas import DataFrame
from pyminimax import fcluster_prototype

cuts = [0.1, 0.25, 0.3, 0.35, 0.6, 0.7]

fig, axs = plt.subplots(3, 2, figsize=(10, 15))

for ax, cut in zip(axs.ravel(), cuts):
```

(continues on next page)

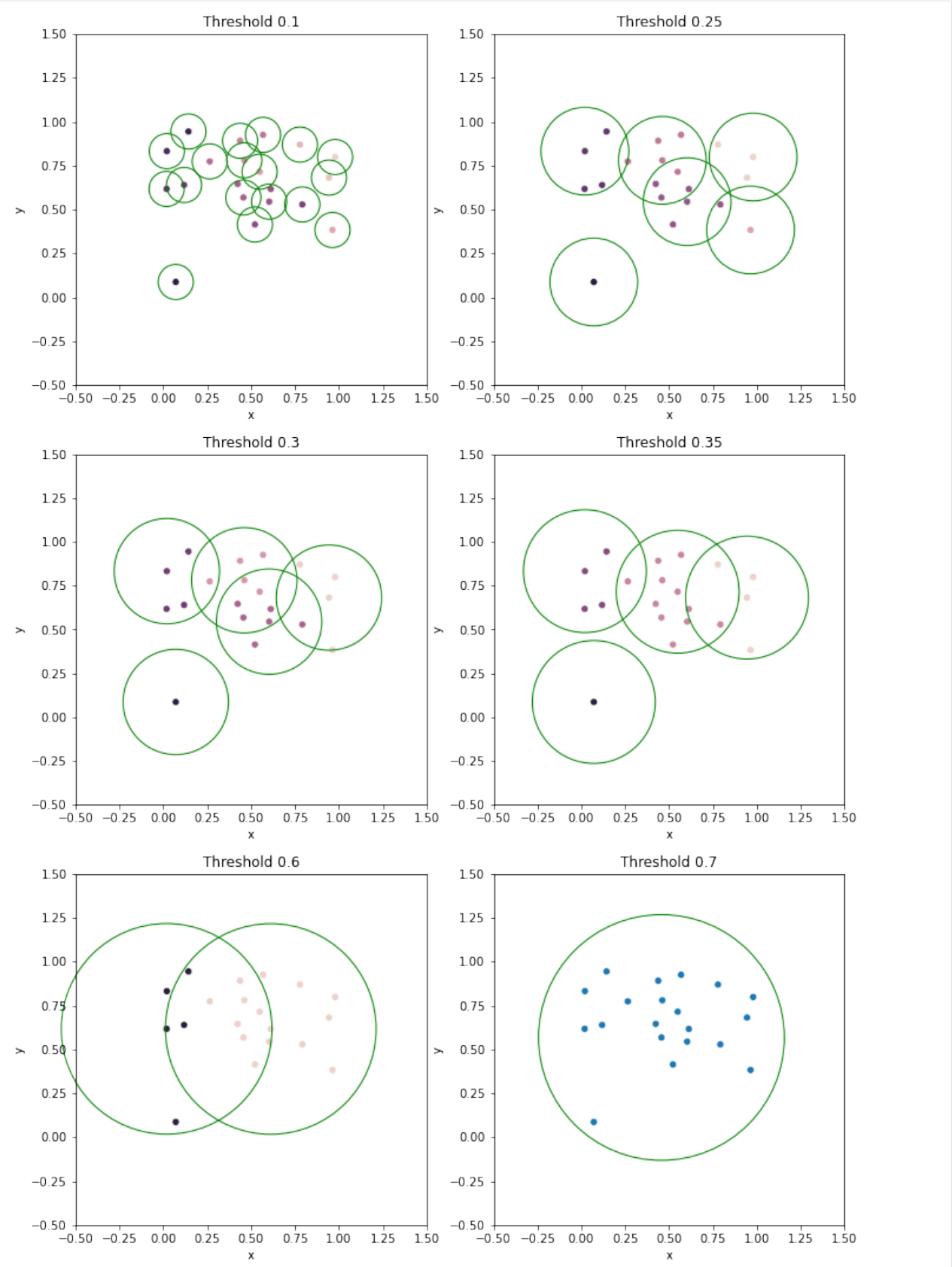
(continued from previous page)

```
    clust_proto = fcluster_prototype(Z, t=cut, criterion='distance')

    df = DataFrame(np.concatenate([X, clust_proto], axis=1), columns=['x', 'y', 'clust',
↪'proto'])
    sns.scatterplot(data=df, x='x', y='y', hue='clust', legend=None, ax=ax)

    ax.set(xlim=(-0.5, 1.5), ylim=(-0.5, 1.5), aspect=1, title=f'Threshold {cut}')
    protos = np.unique(df['proto'].map(int).values)
    for proto in protos:
        circle = plt.Circle(X[proto], cut, edgecolor='g', facecolor='none', clip_
↪on=False)
        ax.add_patch(circle)

fig.tight_layout()
plt.show()
```


Threshold 0.1
Threshold 0.25
Threshold 0.3
Threshold 0.35
Threshold 0.6
Threshold 0.7
x
y

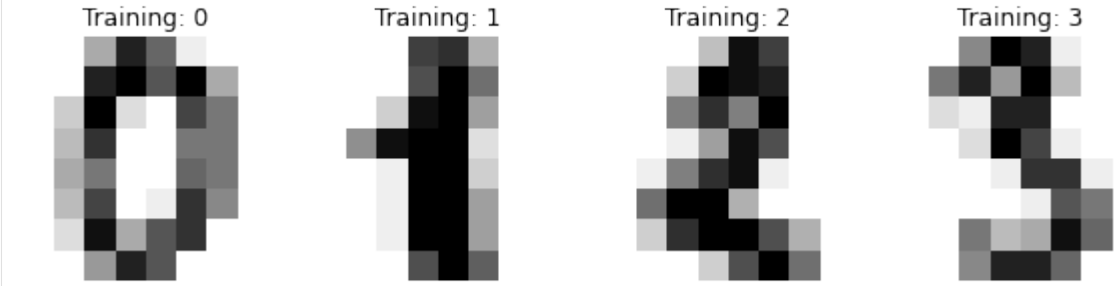
2.2 Hand-Written Digits

In this example we perform minimax linkage clustering on images of hand-written digits 1, 4 and 7. The data we use is a subset of the scikit-learn hand-written digit images data. The below code from its documentation prints the first few images in this dataset.

[4]:
```
import matplotlib.pyplot as plt
from sklearn import datasets

digits = datasets.load_digits()

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image, label in zip(axes, digits.images, digits.target):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Training: %i' % label)
```



First we load the data in a pandas DataFrame, and filter out images that are not 1, 4 or 7. The resulting DataFrame `digits147` has 542 rows, each having 65 values. The first 64 are a flattened 8×8 matrix representing the image, and the last value in the target column indicates this image is a 1, 4 or 7.

[5]:
```
digits = datasets.load_digits(as_frame=True)['frame']
digits147 = digits[digits['target'].isin([1, 4, 7])].reset_index(drop=True)
digits147
```

[5]:
```
     pixel_0_0  pixel_0_1  pixel_0_2  pixel_0_3  pixel_0_4  pixel_0_5  \
0          0.0        0.0        0.0       12.0       13.0        5.0
1          0.0        0.0        0.0        1.0       11.0        0.0
2          0.0        0.0        7.0        8.0       13.0       16.0
3          0.0        0.0        0.0        0.0       14.0       13.0
4          0.0        0.0        0.0        8.0       15.0        1.0
..         ...        ...        ...        ...        ...        ...
537        0.0        0.0        0.0        1.0       13.0        8.0
538        0.0        0.0        3.0       10.0       16.0       16.0
539        0.0        1.0       10.0       16.0       15.0        2.0
540        0.0        0.0        0.0        1.0       12.0        6.0
541        0.0        0.0        0.0        3.0       15.0        4.0

     pixel_0_6  pixel_0_7  pixel_1_0  pixel_1_1  ...  pixel_6_7  pixel_7_0  \
0          0.0        0.0        0.0        0.0  ...        0.0        0.0
1          0.0        0.0        0.0        0.0  ...        0.0        0.0
2         15.0        1.0        0.0        0.0  ...        0.0        0.0
3          1.0        0.0        0.0        0.0  ...        0.0        0.0
```

(continues on next page)

(continued from previous page)

```
4          0.0        0.0        0.0        0.0  ...        0.0        0.0
..         ...        ...        ...        ...  ...        ...        ...
537        0.0        0.0        0.0        0.0  ...        0.0        0.0
538        4.0        0.0        0.0        0.0  ...        0.0        0.0
539        0.0        0.0        0.0        1.0  ...        0.0        0.0
540        0.0        0.0        0.0        0.0  ...        0.0        0.0
541        0.0        0.0        0.0        0.0  ...        0.0        0.0

     pixel_7_1  pixel_7_2  pixel_7_3  pixel_7_4  pixel_7_5  pixel_7_6  \
0          0.0        0.0       11.0       16.0       10.0        0.0
1          0.0        0.0        2.0       16.0        4.0        0.0
2          0.0       13.0        5.0        0.0        0.0        0.0
3          0.0        0.0        1.0       13.0       16.0        1.0
4          0.0        0.0       10.0       15.0        4.0        0.0
..         ...        ...        ...        ...        ...        ...
537        0.0        0.0        0.0       15.0        7.0        0.0
538        0.0        3.0       12.0        0.0        0.0        0.0
539        0.0       10.0       15.0        2.0        0.0        0.0
540        0.0        0.0        0.0       14.0        9.0        0.0
541        0.0        0.0        1.0       16.0        4.0        0.0

     pixel_7_7  target
0          0.0       1
1          0.0       4
2          0.0       7
3          0.0       1
4          0.0       4
..         ...     ...
537        0.0       4
538        0.0       7
539        0.0       7
540        0.0       4
541        0.0       4

[542 rows x 65 columns]
```

For example, the first 64 values of the first row is the below matrix flattened. This is a matrix of grayscale values representing an image of 1.

[6]:
```
digits147.iloc[0].values[:-1].reshape(8, 8)
```

[6]:
```
array([[ 0.,  0.,  0., 12., 13.,  5.,  0.,  0.],
       [ 0.,  0.,  0., 11., 16.,  9.,  0.,  0.],
       [ 0.,  0.,  3., 15., 16.,  6.,  0.,  0.],
       [ 0.,  7., 15., 16., 16.,  2.,  0.,  0.],
       [ 0.,  0.,  1., 16., 16.,  3.,  0.,  0.],
       [ 0.,  0.,  1., 16., 16.,  6.,  0.,  0.],
       [ 0.,  0.,  1., 16., 16.,  6.,  0.,  0.],
       [ 0.,  0.,  0., 11., 16., 10.,  0.,  0.]])
```

We drop the target column from the data, compute the extended linkage matrix and draw the dendrogram.

[7]:
```
from pyminimax import minimax
```

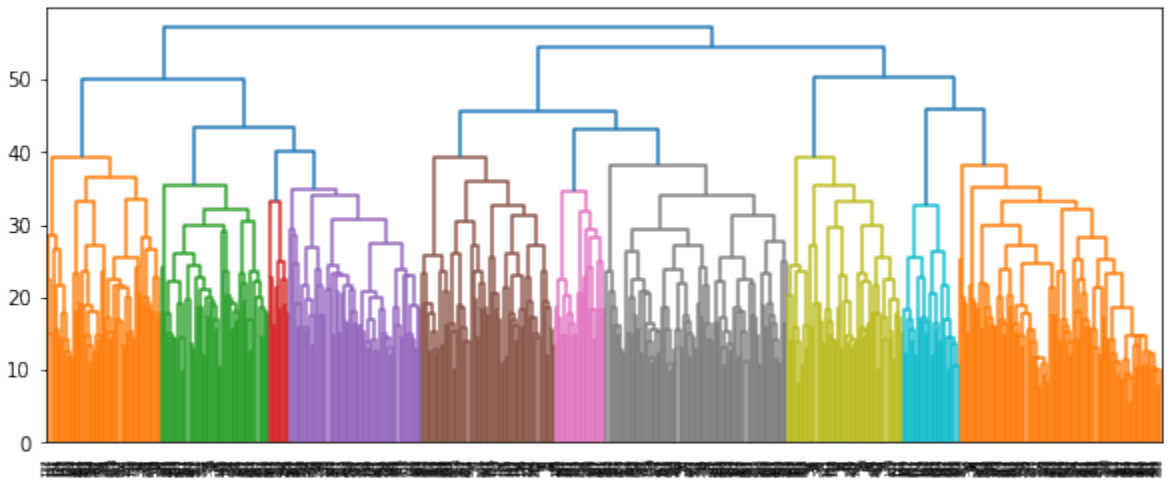
(continues on next page)

(continued from previous page)

```
from scipy.spatial.distance import pdist
from scipy.cluster.hierarchy import dendrogram

X = digits147.drop('target', axis=1).values
Z = minimax(pdist(X), return_prototype=True)

plt.figure(figsize=(10, 4))
dendrogram(Z[:, :4])
plt.show()
```



The 3rd column of the extended linkage matrix is the distance between the two clusters to be merged in each row. The 3rd last merge has distance 50.3388, indicating that if the dendrogram is cut at a threshold slightly above 50.3388, there will be 3 clusters.

The format of the extended linkage matrix is detailed in the quick start guide and the Scipy documentation.

[8]:
```
from pandas import DataFrame

DataFrame(Z[-3:, :], columns=['x', 'y', 'distance', 'n_pts', 'prototype'])
```

[8]:

	x	y	distance	n_pts	prototype
0	1072.0	1078.0	50.338852	182.0	135.0
1	1077.0	1080.0	54.488531	360.0	137.0
2	1079.0	1081.0	57.122675	542.0	22.0

The cluster and prototypes is computed by `pyminimax.fcluster_prototype` and put together with the target column. The result is sorted by target for better visualization. As expected, most of the images of 1 are in the same cluster (cluster #3), and most of the images of 7 are in a different cluster (cluster #1).

[9]:
```
from pyminimax import fcluster_prototype

clust, proto = fcluster_prototype(Z, t=52, criterion='distance').T

res = digits147.assign(clust=clust, proto=proto)
res = res[['target', 'clust', 'proto']].sort_values(by='target')
res
```

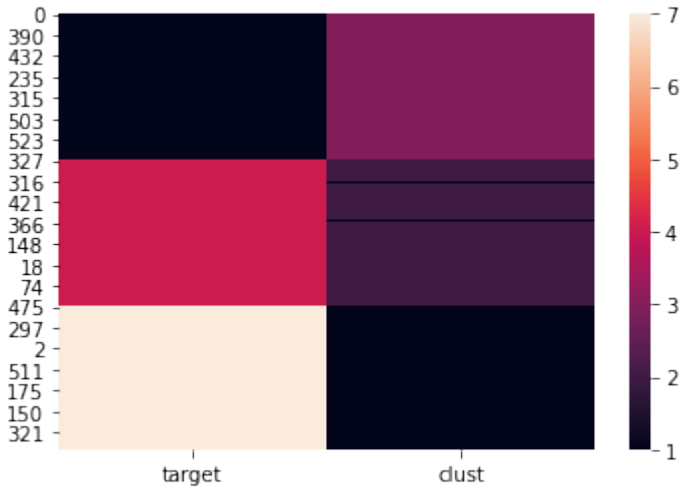
[9]:
```
     target  clust  proto
0         1      3    135
154       1      3    135
157       1      3    135
160       1      3    135
379       1      3    135
..      ...    ...    ...
414       7      1    341
126       7      1    341
314       7      1    341
138       7      1    341
136       7      1    341

[542 rows x 3 columns]
```

An even better visualization is the below heat map of the target column and the cluster column. It is clear that all images of 1 are in cluster #3, all images of 7 are in cluster #1, and most of images of 4 are in cluster #2. There are only a few images of 4 wrongly put into cluster #1. That is, minimax linkage clustering finds those images closer to 7.

[10]:
```
import seaborn as sns

sns.heatmap(res[['target', 'clust']]);
```



The prototypes are the 135th, 341st, and 464th row of the original DataFrame `digits147`.

[11]:
```
import numpy as np

protos = np.unique(res['proto'])
protos
```

[11]:
```
array([135, 341, 464], dtype=int32)
```

We print out the images of the prototypes. These are the representative images of 1, 4 and 7.

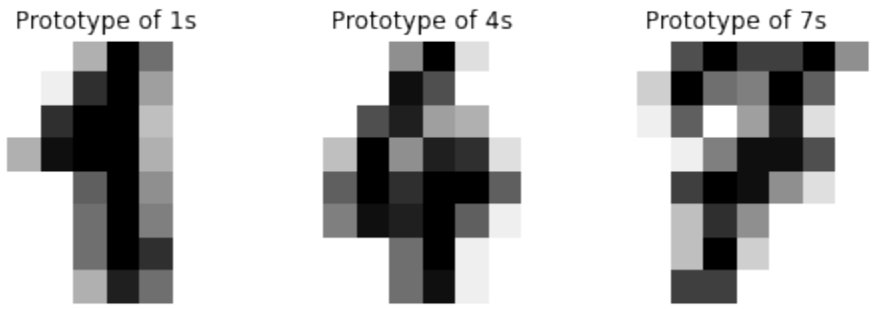
[12]:
```
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(8, 3))

for ax, proto, label in zip(axs, protos[[0, 2, 1]], [1, 4, 7]):
```

(continues on next page)

(continued from previous page)

```
    ax.set_axis_off()
    image = digits147.drop('target', axis=1).iloc[proto].values.reshape(8, 8)
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title(f'Prototype of {label}s')
```



There are 3 images of 4 considered closer to 7. Their indices are 482, 488 and 501, given which we can print out the images for inspection.

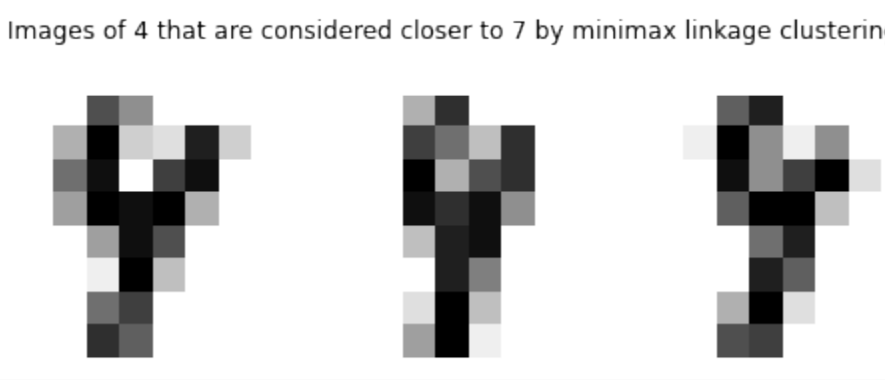
[13]:
```
res[(res['target']==4) & (res['clust']==1)]
```

[13]:
```
     target  clust  proto
501       4      1    341
488       4      1    341
482       4      1    341
```

Arguably they are indeed closer to 7's prototype than 4's.

[14]:
```
fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(8, 3))
plt.suptitle("Images of 4 that are considered closer to 7 by minimax linkage clustering")

for ax, idx in zip(axs, [501, 488, 482]):
    ax.set_axis_off()
    image = digits147.drop('target', axis=1).iloc[idx].values.reshape(8, 8)
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
```



CHAPTER

THREE

PERFORMANCE

This page is generated by a Jupyter notebook which can be opened and run in Binder or Google Colab by clicking on the above badges. **To run it in Google Colab, you need to install PyMinimax in Colab first:**

[ ]:
```
pip install pyminimax
```

The most computationally intensive function in PyMinimax is `pyminimax.minimax`. Although implemented as an efficient nearest-neighbor chain algorithm, PyMinimax is a pure Python implementation at least for now, so it is relatively slow. Below is a plot of the running time on Binder against size of the dataset.

When $n = 3000$, `pyminimax.minimax` takes a little more than 10 minutes on Binder to compute the linkage matrix. For now PyMinimax is only suitable for small to medium datasets.

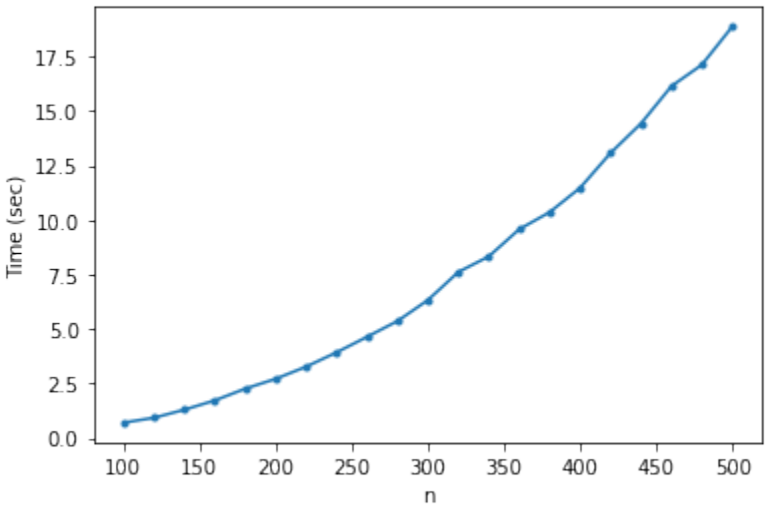
[1]:
```
import time
import numpy as np
import matplotlib.pyplot as plt
from pandas import DataFrame
from pyminimax import minimax
from scipy.spatial.distance import pdist

np.random.seed(0)
n_pts = np.arange(100, 501, 20)
elapsed_time = []

def test_run(n):
    X = np.random.rand(n, 2)
    Z = minimax(pdist(X))

for n in n_pts:
    start = time.process_time()
    test_run(n)
    end = time.process_time()
    elapsed_time.append(end - start)

df = DataFrame({'n': n_pts, 'time': elapsed_time}).set_index('n')
ax = df.plot(style='.-', legend=None);
ax.set(ylabel='Time (sec)')
plt.show()
```


Time (sec)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
100
150
200
250
300
350
400
450
500
n

CHAPTER

FOUR

API REFERENCE

pyminimax.**minimax**(*dists*, *return_prototype=False*)

Perform minimax-linkage clustering using nearest-neighbor chain algorithm.

Parameters

- **dists** (*ndarray*) – The upper triangular of the distance matrix. The result of `scipy.spatial.distance.pdist` is returned in this form.
- **return_prototype** (*bool, default False*) – whether to return prototypes. When this is False, the returned linkage matrix Z has 4 columns, structured the same as the return value of the `scipy.cluster.hierarchy.linkage` function. When this is True, the returned linkage matrix has a 5th column which contains the indices of the prototypes corresponding to each merge.

Returns **Z** – A linkage matrix containing the hierarchical clustering. The first 4 columns has the same structure as the return value of the `scipy.cluster.hierarchy.linkage` function. See the documentation for more information on its structure. Depending on the value of `return_prototype` there is an optional 5th columns.

Return type ndarray

pyminimax.**fcluster_prototype**(*Z*, *t*, *criterion='inconsistent'*, *depth=2*, *R=None*, *monocrit=None*)

Form flat clusters from the hierarchical clustering defined by the given linkage matrix, and the

Parameters

- **Z** (*ndarray*) – The hierarchical clustering encoded with the matrix returned by the *minimax* function.
- **t** (*scalar*) –

  For criteria 'inconsistent', 'distance' or 'monocrit', this is the threshold to apply when forming flat clusters.

  For 'maxclust' or 'maxclust_monocrit' criteria, this would be max number of clusters requested.
- **criterion** (*str, optional*) – The criterion to use in forming flat clusters. This can be any of the following values:

  inconsistent : If a cluster node and all its descendants have an inconsistent value less than or equal to *t*, then all its leaf descendants belong to the same flat cluster. When no non-singleton cluster meets this criterion, every node is assigned to its own cluster. (Default)

  distance : Forms flat clusters so that the original observations in each flat cluster have no greater a cophenetic distance than *t*.

maxclust : Finds a minimum threshold `r` so that the cophenetic distance between any two original observations in the same flat cluster is no more than `r` and no more than *t* flat clusters are formed.

monocrit : Forms a flat cluster from a cluster node c with index i when `monocrit[j] <= t`. For example, to threshold on the maximum mean distance as computed in the inconsistency matrix R with a threshold of 0.8 do:

```
MR = maxRstat(Z[:, :4], R, 3)
fcluster_prototype(Z, t=0.8, criterion='monocrit', monocrit=MR)
```

maxclust_monocrit : Forms a flat cluster from a non-singleton cluster node `c` when `monocrit[i] <= r` for all cluster indices `i` below and including `c`. `r` is minimized such that no more than `t` flat clusters are formed. monocrit must be monotonic. For example, to minimize the threshold t on maximum inconsistency values so that no more than 3 flat clusters are formed, do:

```
MI = maxinconsts(Z[:, :4], R)
fcluster_prototype(Z, t=3, criterion='maxclust_monocrit',␣
↪monocrit=MI)
```

- **depth** (*int, optional*) – The maximum depth to perform the inconsistency calculation. It has no meaning for the other criteria. Default is 2.
- **R** (*ndarray, optional*) – The inconsistency matrix to use for the 'inconsistent' criterion. This matrix is computed if not provided.
- **monocrit** (*ndarray, optional*) – An array of length n-1. *monocrit[i]* is the statistics upon which non-singleton i is thresholded. The monocrit vector must be monotonic, i.e., given a node c with index i, for all node indices j corresponding to nodes below c, `monocrit[i] >= monocrit[j]`.

Returns **fcluster_prototype** – An array of shape `(n, 2)`. `T[i]` is the flat cluster number to which original observation `i` belongs, and the index of the prototype of this cluster.

Return type ndarray

CHAPTER

FIVE

REFERENCE

INDEX

F

fcluster_prototype() (*in module pyminimax*), 19

M

minimax() (*in module pyminimax*), 19