# Final Assignment: Analytical Design

## Task 1

### a) Retrieve and clean data

# import pandas library  
import pandas as pd  
  
# download dataset  
dataset = pd.read\_csv(  
 "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data",  
 names =["sepal\_length","sepal\_width","petal\_length","petal\_width", "species"]  
 )  
  
'''  
# remove attributes we aren't using in this exercise  
dataset.drop("species", inplace=True, axis=1)  
'''  
# remove any rows with missing data  
dataset.dropna(inplace=True)  
  
#show first 10 rows using head() method  
dataset.head(10)

sepal\_length

sepal\_width

petal\_length

petal\_width

species

0

5.1

3.5

1.4

0.2

Iris-setosa

1

4.9

3.0

1.4

0.2

Iris-setosa

2

4.7

3.2

1.3

0.2

Iris-setosa

3

4.6

3.1

1.5

0.2

Iris-setosa

4

5.0

3.6

1.4

0.2

Iris-setosa

5

5.4

3.9

1.7

0.4

Iris-setosa

6

4.6

3.4

1.4

0.3

Iris-setosa

7

5.0

3.4

1.5

0.2

Iris-setosa

8

4.4

2.9

1.4

0.2

Iris-setosa

9

4.9

3.1

1.5

0.1

Iris-setosa

### b)Extract statistics

#### i)Statistics of the whole dataset

dataset.describe()

sepal\_length

sepal\_width

petal\_length

petal\_width

count

150.000000

150.000000

150.000000

150.000000

mean

5.843333

3.054000

3.758667

1.198667

std

0.828066

0.433594

1.764420

0.763161

min

4.300000

2.000000

1.000000

0.100000

25%

5.100000

2.800000

1.600000

0.300000

50%

5.800000

3.000000

4.350000

1.300000

75%

6.400000

3.300000

5.100000

1.800000

max

7.900000

4.400000

6.900000

2.500000

#### ii) Statistics for each species

# extract statistics for setosa  
setosa = dataset[dataset["species"] == "Iris-setosa"]  
setosa.describe()

sepal\_length

sepal\_width

petal\_length

petal\_width

count

50.00000

50.000000

50.000000

50.00000

mean

5.00600

3.418000

1.464000

0.24400

std

0.35249

0.381024

0.173511

0.10721

min

4.30000

2.300000

1.000000

0.10000

25%

4.80000

3.125000

1.400000

0.20000

50%

5.00000

3.400000

1.500000

0.20000

75%

5.20000

3.675000

1.575000

0.30000

max

5.80000

4.400000

1.900000

0.60000

# extract statistics for versicolor  
versicolor = dataset[dataset["species"] == "Iris-versicolor"]  
versicolor.describe()

sepal\_length

sepal\_width

petal\_length

petal\_width

count

50.000000

50.000000

50.000000

50.000000

mean

5.936000

2.770000

4.260000

1.326000

std

0.516171

0.313798

0.469911

0.197753

min

4.900000

2.000000

3.000000

1.000000

25%

5.600000

2.525000

4.000000

1.200000

50%

5.900000

2.800000

4.350000

1.300000

75%

6.300000

3.000000

4.600000

1.500000

max

7.000000

3.400000

5.100000

1.800000

# extract statistics for virginica  
virginica = dataset[dataset["species"] == "Iris-virginica"]  
virginica.describe()

sepal\_length

sepal\_width

petal\_length

petal\_width

count

50.00000

50.000000

50.000000

50.00000

mean

6.58800

2.974000

5.552000

2.02600

std

0.63588

0.322497

0.551895

0.27465

min

4.90000

2.200000

4.500000

1.40000

25%

6.22500

2.800000

5.100000

1.80000

50%

6.50000

3.000000

5.550000

2.00000

75%

6.90000

3.175000

5.875000

2.30000

max

7.90000

3.800000

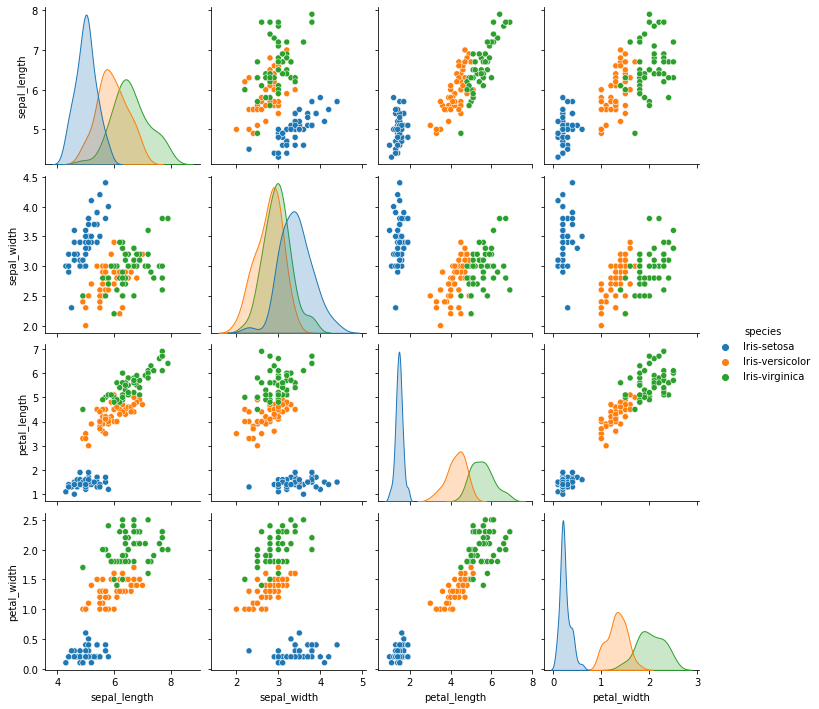
6.900000

2.50000

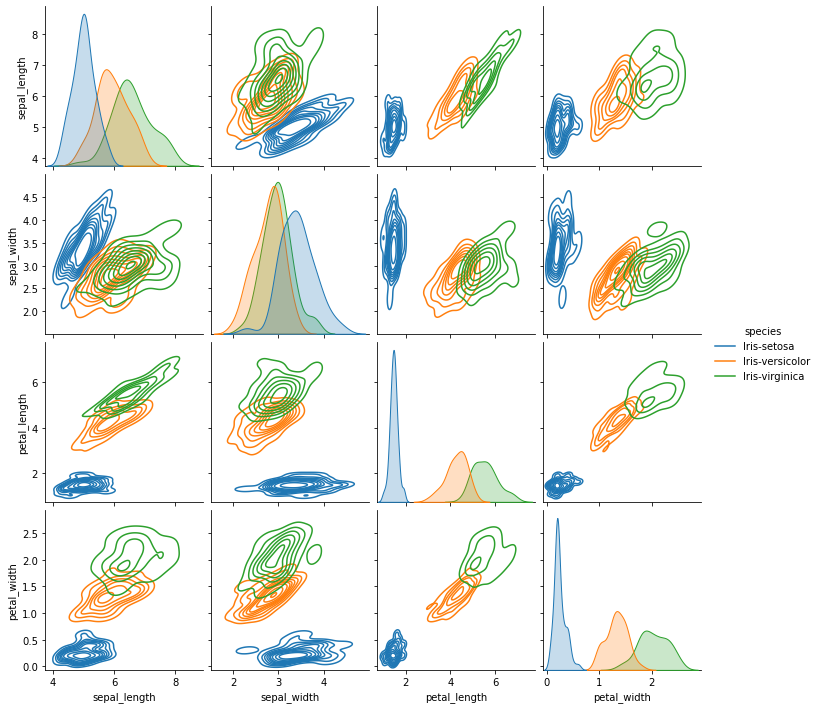
### c)Visualisations

#### i)Pairplots

# import seaborn visualisation library  
import seaborn as sns  
import matplotlib.pyplot as plt  
  
# generate scattergraph pairplot from data  
plt.show(  
 sns.pairplot(dataset, hue="species", kind="scatter")  
)  
  
# generate Kernel Density pairplot from data  
plt.show(  
 sns.pairplot(dataset, hue="species", kind="kde")  
)



png

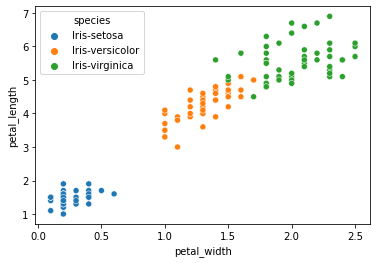


png

These pairplots suggest two things: \* Recorded values for sepal width overlap significantly between all three species. As such, sepal width may be considered a less significant attribute in terms of identification. \* There is a strong, nearly linear relationship between petal length and petal width. This linearity suggests that this dataset might be a good candidate for dimensionality reduction, e.g. either removing one of these attributes, or else by using a process like Principal Component Analysis

#### ii)A scatterplot of petal length x. petal width

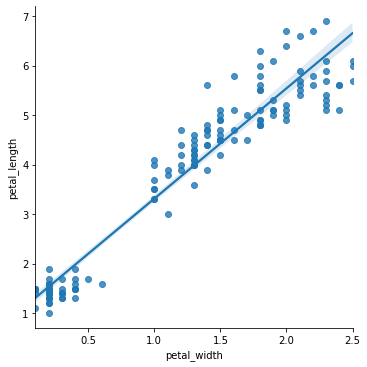
plt.show(  
 sns.scatterplot(data=dataset, x="petal\_width", y="petal\_length", hue="species")  
)



png

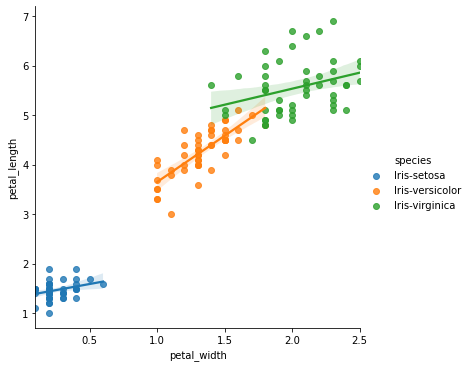
#### iii) Linear regression of petal length x petal width

plt.show(  
 sns.lmplot(data=dataset, x="petal\_width", y="petal\_length", )  
)



png

plt.show(  
 sns.lmplot(data=dataset, x="petal\_width", y="petal\_length", hue="species" )  
)

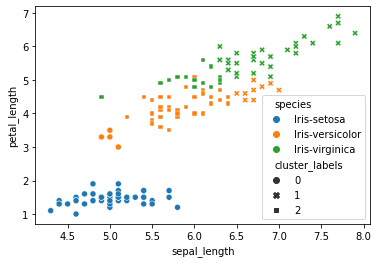


png

### d) clustering

#### i) K-means of petal length vs. sepal length

from sklearn.cluster import KMeans  
# from sklearn.metrics import silhouette\_score  
from sklearn.preprocessing import StandardScaler  
  
# create subset of data  
petals\_and\_sepal\_length = dataset.filter(items=["petal\_length","sepal\_length"])  
  
# standardize values  
scaler = StandardScaler()  
petals\_and\_sepal\_length = scaler.fit\_transform(petals\_and\_sepal\_length)  
  
# initialise kmeans class  
kmeans = KMeans(  
 init="random",  
 n\_clusters=3,  
 n\_init=50,  
 max\_iter=300,  
 random\_state=None)  
  
# run clustering algorithm  
kmeans.fit(petals\_and\_sepal\_length)  
cluster\_labels = kmeans.labels\_  
  
# add k means labels back to original dataframe  
dataset["cluster\_labels"] = cluster\_labels  
  
# visualise cluster\_labels against recorded species  
plt.show(  
 sns.scatterplot(data=dataset, x="sepal\_length", y="petal\_length", hue="species", style ="cluster\_labels")  
)

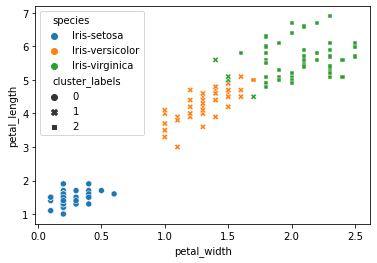


png

* this figure shows the limitations of using a k-means algorithm with these two attributes.
* whilst it is easy to see that each species broadly aligns with one of the identified clusters (e.g, iris setosa with cluster\_label 0) the model is far from perfect.
* The 27 misidentified samples equate to an error rate of 18%.

#### K-means clustering of petal data

# create subset of data  
petals = dataset.filter(items=["petal\_length","petal\_width"])  
  
# standardize values  
petals = scaler.fit\_transform(petals)  
  
# run clustering algorithm  
kmeans.fit(petals)  
cluster\_labels = kmeans.labels\_  
  
# add k means labels back to original dataframe  
dataset["cluster\_labels"] = cluster\_labels  
  
# visualise cluster\_labels against recorded species  
plt.show(  
 sns.scatterplot(data=dataset, x="petal\_width", y="petal\_length", hue="species", style ="cluster\_labels")  
)



png

* the algorithm is much more succesful on these two attributes. The error rate is only 6 in 150, ie 96% sucess rate.

### e) Conclusion

This exploratory anaysis of the dataset has suggested that: \* a strong correlation exists between petal size and petal width, though the strength of this correlation is significantly weaker at the species level. \* sepal width is one of the weaker attributes by which to identify these three iris species. \* K-means clustering (using petal length and width as its axes) can create clusters that largely align with species categorisation.

## Task 2

### a) A flowchart of the steps for creating a dendogram based on single linkage clustering.

Many tutorials for single-linkage clustering involve a redrawing the distance matrix of all clusters after every clustering operation. However, as single linkage is based on single links (ie, the distance between single plots in differing clusters), no new information is generated by this redrawing. If record of each point’s clustering is stored against it then the original distance matrix already contains all the information needed to determine the next cluster. For clarity of code, I have skipped the redundant step of redrawing the distance matrix in each epoch. Names of the functions I will use to accomplish these steps are on the left of the diagram.

single%20link%20clustering%20algorithm.svg

single%20link%20clustering%20algorithm.svg

### b) Subsetting the data

# filter out irrelevant attributes  
sepals = dataset.filter(items=["sepal\_length","sepal\_width"])  
  
# reducing set to first 6 entries  
sepals = sepals.iloc[:6]

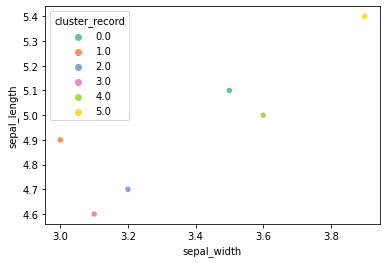
### c) Code for carrying out single linkage clustering.

import math  
  
class Clustering:  
  
 def euclidean\_distance(self, ax, ay, bx, by):  
 ed = math.sqrt((abs(ax-bx) + abs(ay-by)))  
 return ed  
  
 def get\_coord(self, point):  
   
 x = self.points.at[point, "sepal\_width"]  
 y = self.points.at[point, "sepal\_length"]  
  
 return x,y  
  
 def get\_distance\_matrix(self):  
   
 # create empty 2d matrix  
 rows, cols = (6, 6)  
 dm = [[0 for i in range(cols)] for j in range(rows)]  
  
 # fill in values   
 for i in range(rows):  
 point\_a = self.get\_coord(i)  
 for j in range(cols):  
  
 #this if/else is to stop doubling of information between identical i:j and j:i  
 if i <= j:  
 dm[i][j] = None  
 continue  
 else:  
 point\_b = self.get\_coord(j)  
 dm[i][j] = self.euclidean\_distance(\*point\_a, \*point\_b)  
   
 # remove blank fields  
 for i in dm:  
 while None in i:  
 i.remove(None)  
  
 return dm  
   
 def create\_cluster\_records(self):  
 for i in range(len(self.points)):  
 self.points.at[i, "cluster\_record"] = i  
   
 def \_\_init\_\_(self, dataframe):  
 self.points = dataframe  
 self.distance\_matrix = self.get\_distance\_matrix()  
 self.create\_cluster\_records()  
   
   
 def next\_cluster(self):  
   
 #abbreviated here for more legible code  
 dm = self.distance\_matrix  
   
 # this tuple, once filled with appropriate values, will be passed back to caller  
 next\_cluster = {  
 "distance" : 99999, # an arbitrary high number   
 "i" : None,  
 "j" : None,  
 }  
   
 # iterate over distance matrix  
 for i in range(len(dm)):  
 for j in range(len(dm[i])):  
   
 # 'is this the shortest distance we've seen?  
 if dm[i][j] > next\_cluster["distance"]:  
 continue  
   
 # 'are these two points in different clusters? '  
 elif self.points.at[i, "cluster\_record"] == self.points.at[j, "cluster\_record"]:  
 continue  
   
 # record of best candidate for clustering based on this pass of dm so far  
 else:  
 next\_cluster = {  
 "distance" : dm[i][j],  
 "i" : i,  
 "j" : j,  
 }  
 return next\_cluster  
   
 def join\_clusters(self, next\_cluster):  
   
 #abbreviation for easier reading  
 df = self.points  
 i = next\_cluster["i"]  
 j = next\_cluster["j"]  
   
 # what clusters do points i and j belong to?  
 i = df.at[i, "cluster\_record"]  
 j = df.at[j, "cluster\_record"]  
 print(i,j)  
   
 # join all of j's cluster to i's cluster  
 self.points.replace({"cluster\_record": j,}, i, inplace=True)  
   
 def next\_epoch(self):  
   
 # abbreviations  
 df = self.points  
   
 # find candidates for clustering   
 next\_cluster = self.next\_cluster()  
 print(next\_cluster)  
 print("\n")  
   
 # make the cluster and print the cluster\_record  
 self.join\_clusters(next\_cluster)  
 print(df.sort\_values("cluster\_record"))  
 print("\n")  
   
 # plot showing current clusters  
 plt.show(  
 sns.scatterplot(data=df, x="sepal\_width", y="sepal\_length", hue="cluster\_record", palette="Set2")  
 )

### d) Performing ‘Initial Setup’

#initialise Clustering object using our data  
c = Clustering(sepals)  
  
#show cluster record just created  
print(c.points)  
  
# plot of points coloured by cluster\_record  
plt.show(  
 sns.scatterplot(data=c.points, x="sepal\_width", y="sepal\_length", hue="cluster\_record", palette="Set2")  
 )

sepal\_length sepal\_width cluster\_record  
0 5.1 3.5 0.0  
1 4.9 3.0 1.0  
2 4.7 3.2 2.0  
3 4.6 3.1 3.0  
4 5.0 3.6 4.0  
5 5.4 3.9 5.0



png

The cendogram looks like: dendrogram-Cluster%200.svg

#show Distance Matrix just generated  
for i in c.distance\_matrix:  
 print(i)

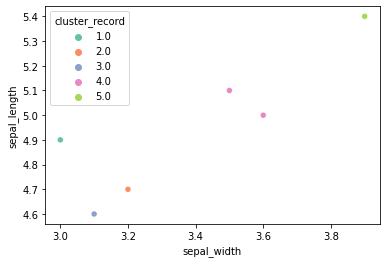
[]  
[0.8366600265340751]  
[0.8366600265340751, 0.6324555320336761]  
[0.9486832980505138, 0.6324555320336765, 0.44721359549995865]  
[0.44721359549995765, 0.8366600265340753, 0.8366600265340753, 0.948683298050514]  
[0.8366600265340759, 1.1832159566199232, 1.1832159566199232, 1.264911064067352, 0.8366600265340757]

### e) Clustering Epochs

#### Epoch 1

c.next\_epoch()

{'distance': 0.44721359549995765, 'i': 4, 'j': 0}  
  
  
4.0 0.0  
 sepal\_length sepal\_width cluster\_record  
1 4.9 3.0 1.0  
2 4.7 3.2 2.0  
3 4.6 3.1 3.0  
0 5.1 3.5 4.0  
4 5.0 3.6 4.0  
5 5.4 3.9 5.0

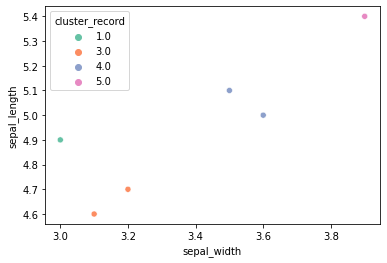


png

Dendogram looks like: dendrogram-Cluster%201.svg #### Epoch 2

c.next\_epoch()

{'distance': 0.44721359549995865, 'i': 3, 'j': 2}  
  
  
3.0 2.0  
 sepal\_length sepal\_width cluster\_record  
1 4.9 3.0 1.0  
2 4.7 3.2 3.0  
3 4.6 3.1 3.0  
0 5.1 3.5 4.0  
4 5.0 3.6 4.0  
5 5.4 3.9 5.0

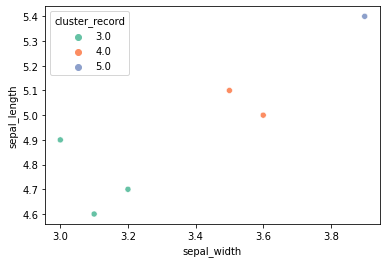


png

Dendogram looks like: dendrogram-Cluster%202.svg #### Cluster 3

c.next\_epoch()

{'distance': 0.6324555320336761, 'i': 2, 'j': 1}  
  
  
3.0 1.0  
 sepal\_length sepal\_width cluster\_record  
1 4.9 3.0 3.0  
2 4.7 3.2 3.0  
3 4.6 3.1 3.0  
0 5.1 3.5 4.0  
4 5.0 3.6 4.0  
5 5.4 3.9 5.0



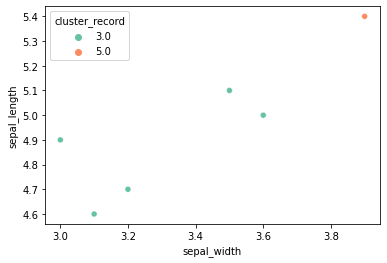
png

Dendogram looks like: dendrogram-Cluster%203.svg

#### Epoch 4

c.next\_epoch()

{'distance': 0.8366600265340751, 'i': 2, 'j': 0}  
  
  
3.0 4.0  
 sepal\_length sepal\_width cluster\_record  
0 5.1 3.5 3.0  
1 4.9 3.0 3.0  
2 4.7 3.2 3.0  
3 4.6 3.1 3.0  
4 5.0 3.6 3.0  
5 5.4 3.9 5.0



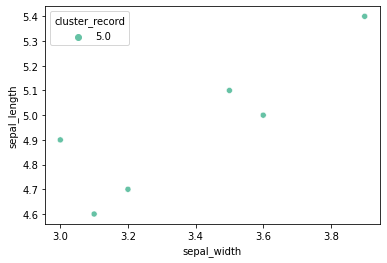
png

Dendogram looks like: dendrogram-Cluster%204-2.svg

#### Epoch 5

c.next\_epoch()

{'distance': 0.8366600265340757, 'i': 5, 'j': 4}  
  
  
5.0 3.0  
 sepal\_length sepal\_width cluster\_record  
0 5.1 3.5 5.0  
1 4.9 3.0 5.0  
2 4.7 3.2 5.0  
3 4.6 3.1 5.0  
4 5.0 3.6 5.0  
5 5.4 3.9 5.0

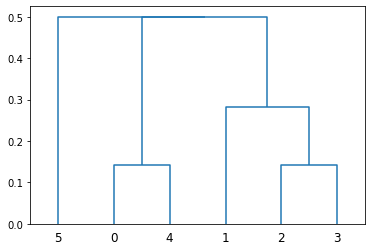


png

Dendogram looks like: dendrogram-Complete.svg

#### Check of my finished dendogram against another library

from scipy.cluster.hierarchy import dendrogram, linkage  
l = linkage(sepals)  
plt.show(dendrogram(l, color\_threshold=0))



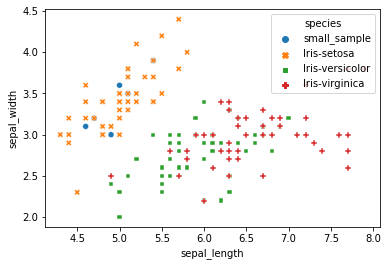
png

* My clustering, and the attached dendogram, matches that produced by SciPy in all except the final clustering.
* In my plot I joined cluster (1,2,3) to (0,4) before joining the new cluster (0,1,2,3,4) to (5).
* SciPy joined (1,2,3) to (0,4) at the same time as joining (5).
* This can be explained by differences in rounding and/or floating point errors:
  + The distance of p5:p4, as printed in Epoch 5: 0.8366600265340757
  + The distance of p2:p0, as printed in Epoch 4: 0.8366600265340751
  + The difference between these two numbers according to Python: 5.551115123125783e-16 ie 0.0000000000000005
  + The difference between these numbers according to Google: 0
* As this difference in calculation is so minute, and has no practical effect on our conclusion, this discrepancy has been ignored.

### Conclusion

* If we mark our cut-off point through the longest vertical line on the dendogram (excluding single value clusters, ie point 5) we create the following figure: dendrogram-Speciation.svg
* We can describe the contents of the figure as either 3 clusters, or as 2 clusters and an outlier.
* If we were to treat this small sample size as representative of all irises, we could infer either two or three species from our dendogram.
* It is clear, however, that our very small sample size, misrepresents the range of possible values. In doing so, it over-emphasizes the significance of differences between our six samples.
* If we load the sepals information for the entire iris dataset, a very different context emerges:

for i in range (6):  
 dataset.at[i, "species"] = "small\_sample"  
  
plt.show(  
 sns.scatterplot(data=dataset, x="sepal\_length", y="sepal\_width", hue="species", style="species")  
)



png

* All members of our small sample size fall within the range for Iris-Setosa.
* Checking the original dataset, it can be seen that all members of the small sample size do belong to Iris-Setosa.
* Only one species is shown on our dendogram, though its focus on such a small range might overstate the importance of distances within that species.