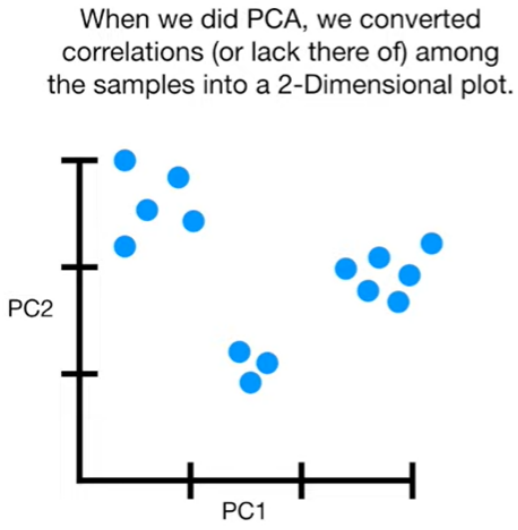
Multidimensional Scaling (MDS) is a non-linear technique for embedding data in a lower-dimensional space.

Multidimensional scaling is a visual representation of distances or dissimilarities between sets of objects.

As well as interpreting dissimilarities as distances on a graph, MDS can also serve as a **dimension reduction technique** for high-dimensional data.

Multidimensional scaling (MDS) is a means of visualizing the level of **similarity** of individual cases of a dataset.

MDS is used to translate "information about the pairwise 'distances' among a set of ${\textstyle n}$ objects or individuals" into a configuration of ${\textstyle n}$ points mapped into an abstract **Cartesian space**.



	Cell1	Cell2	Cell3	Cell4	...
Gene1	3	0.25	2.8	0.1	...
Gene2	2.9	0.8	2.2	1.8	...
Gene3	2.2	1	1.5	3.2	...
Gene4	2	1.4	2	0.3	...
Gene5	1.3	1.6	1.6	0	...
Gene6	1.5	2	2.1	3	...
Gene7	1.1	2.2	1.2	2.8	...
Gene8	1	2.7	0.9	0.3	...
Gene9	0.4	3	0.6	0.1	...

etc...

MDS and **PCoA** are very similar to **PCA** except that **instead of converting correlations** into a 2-D graph, they convert **distance among the samples** into a 2-D graph.

One very common way to calculate distances between two varables is **Euclidean Distance**.

With the **Euclidean Distance**, the graph would be identical to **PCA** graph.

In other words, clustering based on **minimizing the linear distances** is the same **maximizing the linear correlations**.

So there are multiple other methods to measure the distances. For example, another way to measure distances between cells is to calculate **the average of the absolute value of the log fold changes among variables**.

$$\text{Gene1} = \log\left(\frac{3}{0.25}\right) = 3.58 \rightarrow 3.58$$

$$\text{Gene2} = \log\left(\frac{2.9}{0.8}\right) = 1.86 \rightarrow 1.86$$

$$\vdots \qquad \vdots$$

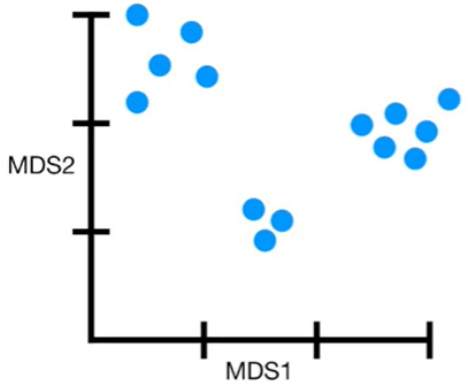
$$\text{Gene8} = \log\left(\frac{1}{2.7}\right) = -1.43 \rightarrow 1.43$$

Lastly, take the average of all the numbers. That's the average of the absolute value of the log fold changes among genes.

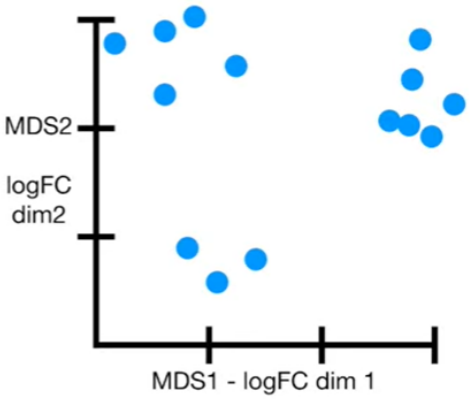
NOTE: We take the absolute value so that the negative fold changes don't cancel out positive ones.

Ultimately, we'll get graphs that look different!!!

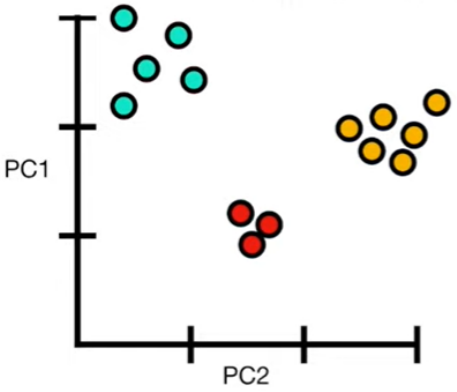
MDS plot using Euclidian distance



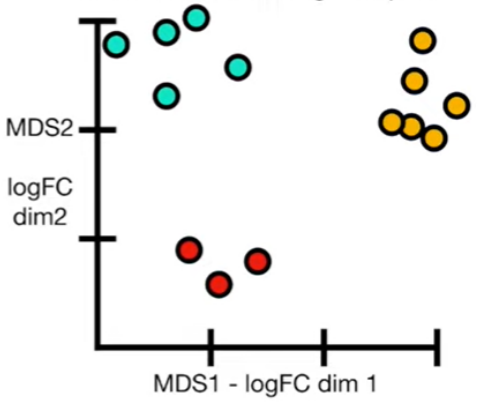
MDS plot using log fold change.

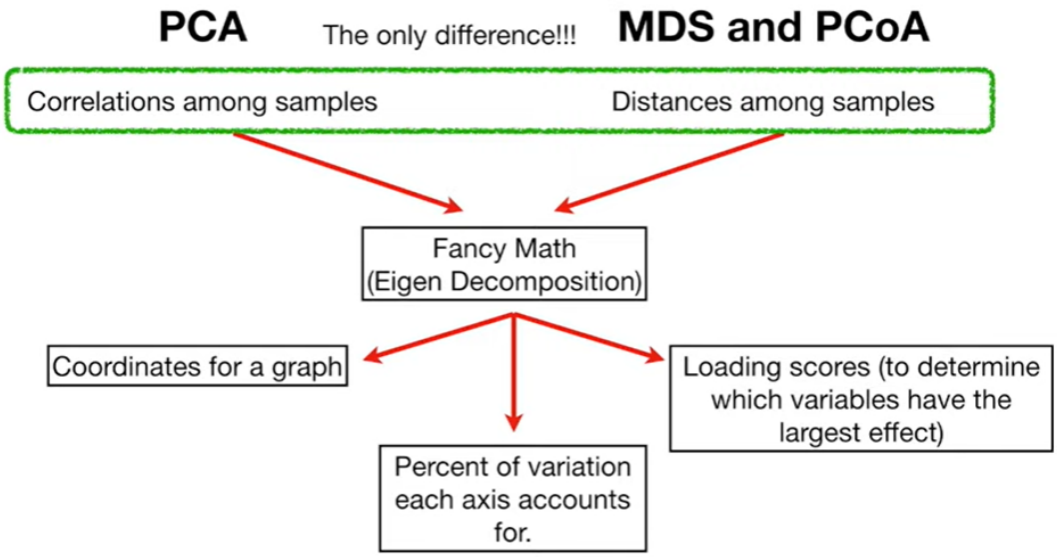


PCA creates plots based on correlations among samples.



MDS and PCoA create plots based on distances among samples





Ref: https://stackabuse.com/guide-to-multidimensional-scaling-in-python-with-scikit-learn/

https://en.wikipedia.org/wiki/Multidimensional_scaling

https://towardsdatascience.com/multidimensional-scaling-d84c2a998f72

MDS can be applied as a preprocessing step for dimensionality reduction in the case of classification as well as regression problems.

MDS is not only an effective technique for **dimensionality reduction** but also for **data visualization**.

It maintains the same clusters and patterns of high-dimensional data in the lower-dimensional space so you can boil down, say, a 5-dimensional dataset to a 3-dimensional dataset which you can interpret much more easily and naturally.

Normally the distance measure used in MDS is the **Euclidean distance**, however, any other suitable dissimilarity metric can be used when applying MDS.

There are two main ways to implement MDS:

- **Metric MDS / Classical MDS:** This version of MDS aims to preserve the **pairwise distance/dissimilarity measure** as much as possible.
- **Non-Metric MDS:** This method is applicable when only the **ranks of a dissimilarity metric** are known. MDS then maps the objects so that the ranks are preserved as much as possible.

Performing Multidimensional Scaling in Python with Scikit-Learn

The Scikit-Learn library's **sklearn.manifold** module implements **manifold learning** and **data embedding techniques**.

We'll be using the **MDS** class of this module.

The embeddings are determined using the **stress minimization using majorization (SMACOF)** algorithm.

Some of the important parameters for setting up the MDS object are (this is not an exhaustive list):

- *n_components:* Number of dimensions to map the points to. The default value is 2.
- *metric:* A Boolean variable with a default value of True for metric MDS and False for its non-metric version.
- *dissimilarity:* The default value is euclidean, which specifies **Euclidean pairwise distances**. The other possible value is **precomputed**. Using precomputed requires the computation of the pairwise distance matrix and using this matrix as an input to the *fit()* or *fit_transform()* function.

The four attributes associated with an MDS object are:

- *embedding_:* Location of points in the new space.
- *stress_:* Goodness-of-fit statistic used in MDS.
- *dissimilaritymatrix:* The matrix of pairwise distances/dissimilarity.
- *niter:* Number of iterations pertaining to the best goodness-of-fit measure.

Like all other classes for dimensionality reduction in scikit-learn, the MDS class also implements the fit() and fit_transform() methods.

In [1]:
```
from sklearn.manifold import MDS
from matplotlib import pyplot as plt
import sklearn.datasets as dt
import seaborn as sns
import numpy as np
from sklearn.metrics.pairwise import manhattan_distances, euclidean_dista
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
```

In [2]:
```
X = np.array([[0, 0, 0], [0, 0, 1], [1, 1, 1], [0, 1, 0], [0, 1, 1]])
```

In [3]:
```
mds = MDS(random_state=0)
X_transform = mds.fit_transform(X)
print(X_transform)
```

```
[[ 0.72521687  0.52943352]
 [ 0.61640884 -0.48411805]
 [-0.9113603  -0.47905115]
 [-0.2190564   0.71505714]
 [-0.21120901 -0.28132146]]
```

Another method of applying MDS is by **constructing a distance matrix** and **applying MDS** directly to this matrix as shown in the code below.

This method is useful when a distance measure other than Euclidean distance is required.

The code below computes the **pairwise Manhattan distances** (also called the **city block distance** or **L1 distance**) and transforms the data via MDS.

Note the dissimilarity argument has been set to **precomputed**:

In [4]:
```
dist_manhattan = manhattan_distances(X)
mds = MDS(dissimilarity='precomputed', random_state=0)
# Get the embeddings
X_transform_L1 = mds.fit_transform(dist_manhattan)
```

In [5]:
```
X_transform_L1
```

Out[5]:
```
array([[ 0.9847767 ,  0.84738596],
       [ 0.81047787, -0.37601578],
       [-1.104849  , -1.06040621],
       [-0.29311254,  0.87364759],
       [-0.39729303, -0.28461157]])
```

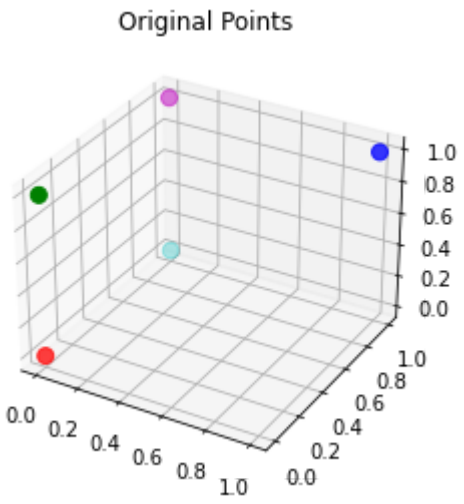
Though, this doesn't help us gain a good intuition as to what just happened.

To gain a better understanding of the entire process, let's plot the original points and their embeddings created by preserving Euclidean distances.

An original point and its corresponding embedded point are both shown in the same color:

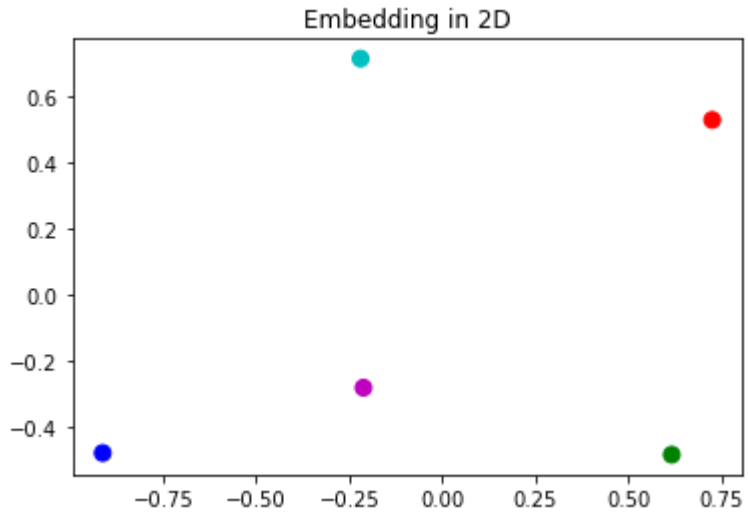
In [6]:
```
colors = ['r', 'g', 'b', 'c', 'm']
size = [64, 64, 64, 64, 64]
fig = plt.figure(2, (10,4))
ax = fig.add_subplot(121, projection='3d')
plt.scatter(X[:,0], X[:,1], zs=X[:,2], s=size, c=colors)
plt.title('Original Points')
```

Out[6]:
```
Text(0.5, 0.92, 'Original Points')
```



In [20]:
```
ax = fig.add_subplot(122)
plt.scatter(X_transform[:,0], X_transform[:,1], s=size, c=colors)
plt.title('Embedding in 2D')
```

```
fig.subplots_adjust(wspace=.4, hspace=0.5)
plt.show()
```



The plot keeps the relative distances generally intact - purple, green and blue are close together, and their relative position to each other is approximately the same when compared to cyan and red.

Practical Multidimensional Scaling On Olivetti Faces Dataset From AT&T

As a practical illustration of MDS, we'll use the **Olivetti faces dataset** from **AT&T** to show the embeddings in a space with dimensions as low as 2D.

The dataset has *10 64x64* bitmap images per person, each image acquired with varying facial expressions or lighting conditions.

MDS preserves the patterns in data so that different face images of the same person are close to each other in the 2D space and far away from another person's mapped face.

To avoid clutter, we'll take only the faces of 4 distinct people and apply MDS to them.

Before fetching the dataset and applying MDS, let's write a small function, **mapData()**, that takes the input arguments, i.e., the **pairwise distance matrix** *dist_matrix*, raw data matrix *X*, the class variable *y*, the Boolean variable *metric* and *title* for the graph.

The function applies MDS to the distance matrix and displays the transformed points in 2D space, with the same colored points indicating the mapped image of the same person. In a second figure, it also displays the image of each face on the graph where it is mapped in the lower-dimensional space.

We'll demonstrate MDS with different distance measures along with non-metric MDS:

In [7]:
```
def mapData(dist_matrix, X, y, metric, title):
    mds = MDS(metric=metric, dissimilarity='precomputed', random_state=0)
    # Get the embeddings
    pts = mds.fit_transform(dist_matrix)
    # Plot the embedding, colored according to the class of the points
    fig = plt.figure(2, (15,6))
```

```
    ax = fig.add_subplot(1,2,1)
    ax = sns.scatterplot(x=pts[:, 0], y=pts[:, 1],
                         hue=y, palette=['r', 'g', 'b', 'c'])

    # Add the second plot
    ax = fig.add_subplot(1,2,2)
    # Plot the points again
    plt.scatter(pts[:, 0], pts[:, 1])

    # Annotate each point by its corresponding face image
    for x, ind in zip(X, range(pts.shape[0])):
        im = x.reshape(64,64)
        imagebox = OffsetImage(im, zoom=0.3, cmap=plt.cm.gray)
        i = pts[ind, 0]
        j = pts[ind, 1]
        ab = AnnotationBbox(imagebox, (i, j), frameon=False)
        ax.add_artist(ab)
    plt.title(title)
    plt.show()
```

The code below fetches the Olivetti faces dataset and extracts examples with labels < 4:

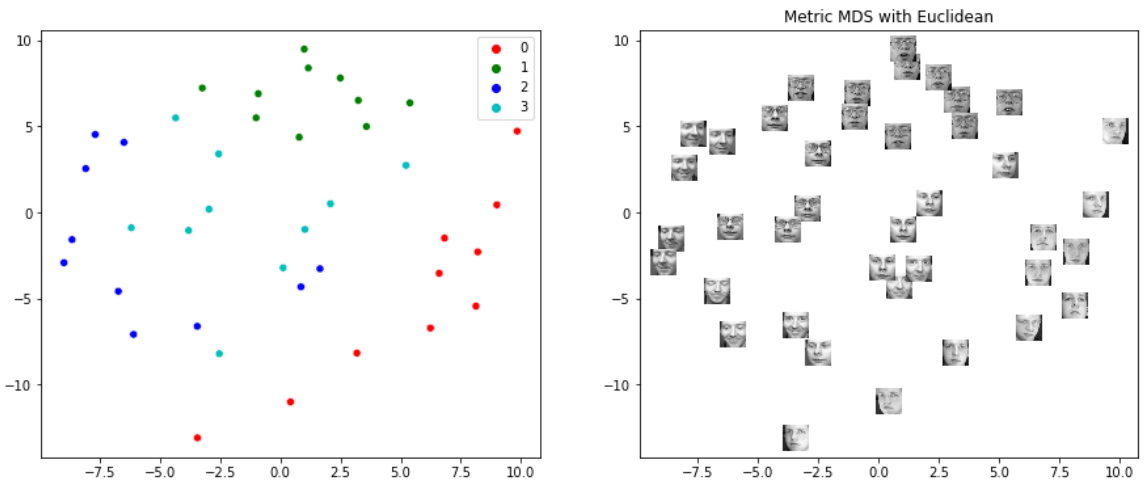
In [8]:
```
import sklearn.datasets as dt
faces = dt.fetch_olivetti_faces()
X_faces = faces.data
y_faces = faces.target
ind = y_faces < 4
X_faces = X_faces[ind,:]
y_faces = y_faces[ind]
```

And without further ado, let's load the data in and run our mapData() function on it!

Using the Euclidean Pairwise Distances

The mapping of the Olivetti faces dataset using Euclidean distances is shown below. Euclidean distance is the default distance for MDS because of how versatile and commonly used it is:

In [9]:
```
dist_euclid = euclidean_distances(X_faces)
mapData(dist_euclid, X_faces, y_faces, True,
        'Metric MDS with Euclidean')
```

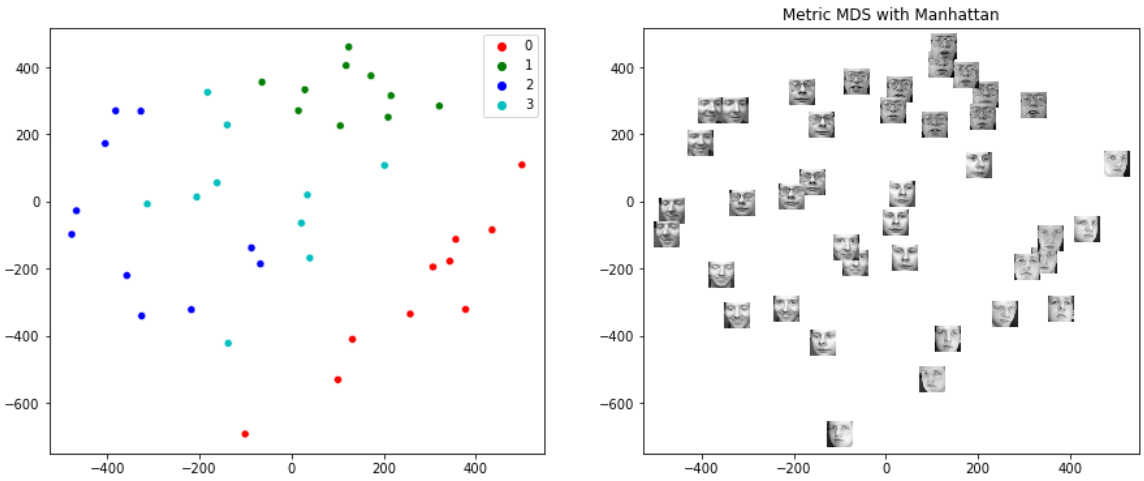


We can see a nice mapping of 64x64 images to a two-dimensional space, where the class of each image is well separated from the rest in most cases. It's worth taking a moment to appreciate the fact that images residing in a 64x64 dimension space can be reduced to a two dimensional space, and still retain their informational value.

Using the Manhattan Pairwise Distances

For comparison, we can perform MDS on the same data using the Manhatten pairwise distances. The code below uses the Manhatten distance matrix as an input to *mapData()*:

In [24]:
```
dist_L1 = manhattan_distances(X_faces)
mapData(dist_L1, X_faces, y_faces, True,
        'Metric MDS with Manhattan')
```

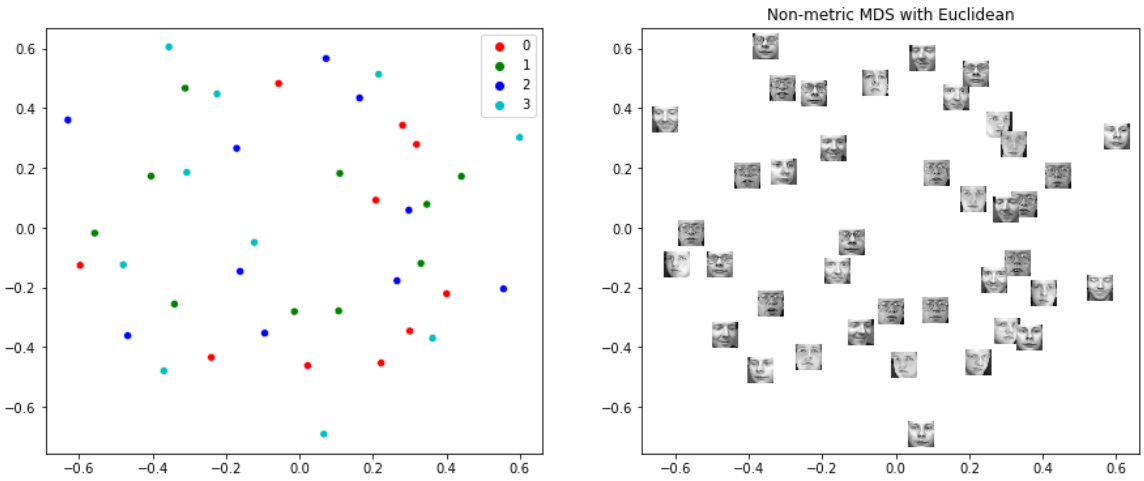


We can see the mapping is quite similar to the one obtained via Euclidean distances. Each class is nicely separated in the lower-dimensional space, though they're offset a bit differently on the plot.

Performing Non-Metric Multidimensional Scaling

As a final example, we'll show non-metric MDS on the same dataset using Euclidean distances and see how it compares with the corresponding metric version:

In [30]:
```
mapData(dist_euclid, X_faces, y_faces, False,
        'Non-metric MDS with Euclidean')
```



There are quite a lot of hiccups here. We can see that this version of MDS does not perform so well on the Olivetti faces dataset.

This is mainly because of the quantitative nature of data.

Non-metric MDS maintains the ranked distances between objects rather than the actual distances.

The n_components Parameter in MDS

One of the important **hyper-parameters** involved in MDS is **the size of the lower-dimensional space in which the points are embedded.**

This would be very relevant when MDS is used as a preprocessing step for dimensionality reduction.

The question arises:

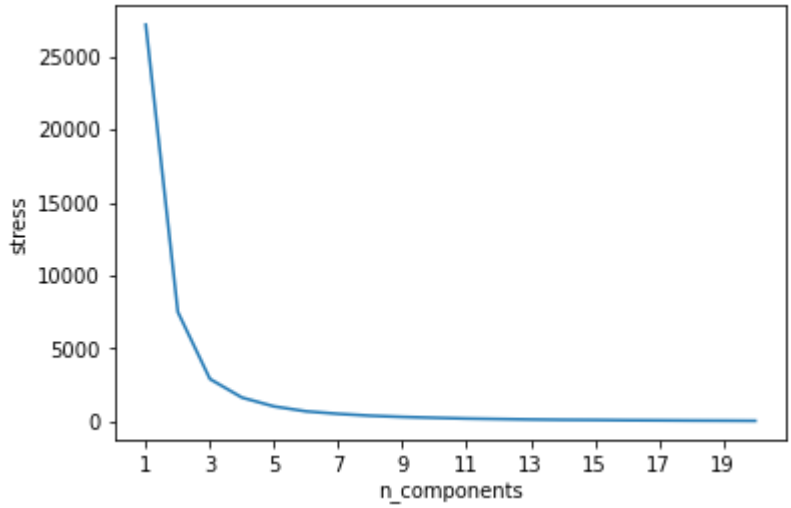
Just how many dimensions do you pick, so that you reduce the dimensionality the most you can, without losing important information?

A simple method to choose a value of this parameter is **to run MDS on different values of n_components** and **plot the stress_ value for each embedding.** Given that the stress *value decreases with higher dimensions - you pick a point that has a fair tradeoff between stress* and n_components.

The code below runs MDS by varying the dimensions from 1-20 and plots the corresponding stress_ attribute for each embedding:

In [31]:
```
stress = []
# Max value for n_components
max_range = 21
for dim in range(1, max_range):
    # Set up the MDS object
    mds = MDS(n_components=dim, dissimilarity='precomputed', random_state
    # Apply MDS
    pts = mds.fit_transform(dist_euclid)
    # Retrieve the stress value
    stress.append(mds.stress_)
# Plot stress vs. n_components
```

```
plt.plot(range(1, max_range), stress)
plt.xticks(range(1, max_range, 2))
plt.xlabel('n_components')
plt.ylabel('stress')
plt.show()
```



We can see that *increasing the value of **n_components** decreases the stress value at the beginning and then the curve levels off.* There's almost no difference between 18 and 19 dimensions, but there's a huge difference between 1 and 2 dimensions.

The elbow of the curve is a good choice for the optimal value of **n_components**. In this case the value can be taken at 4, which is an amazing 0.09% reduction of features/attributes.

In [ ]: