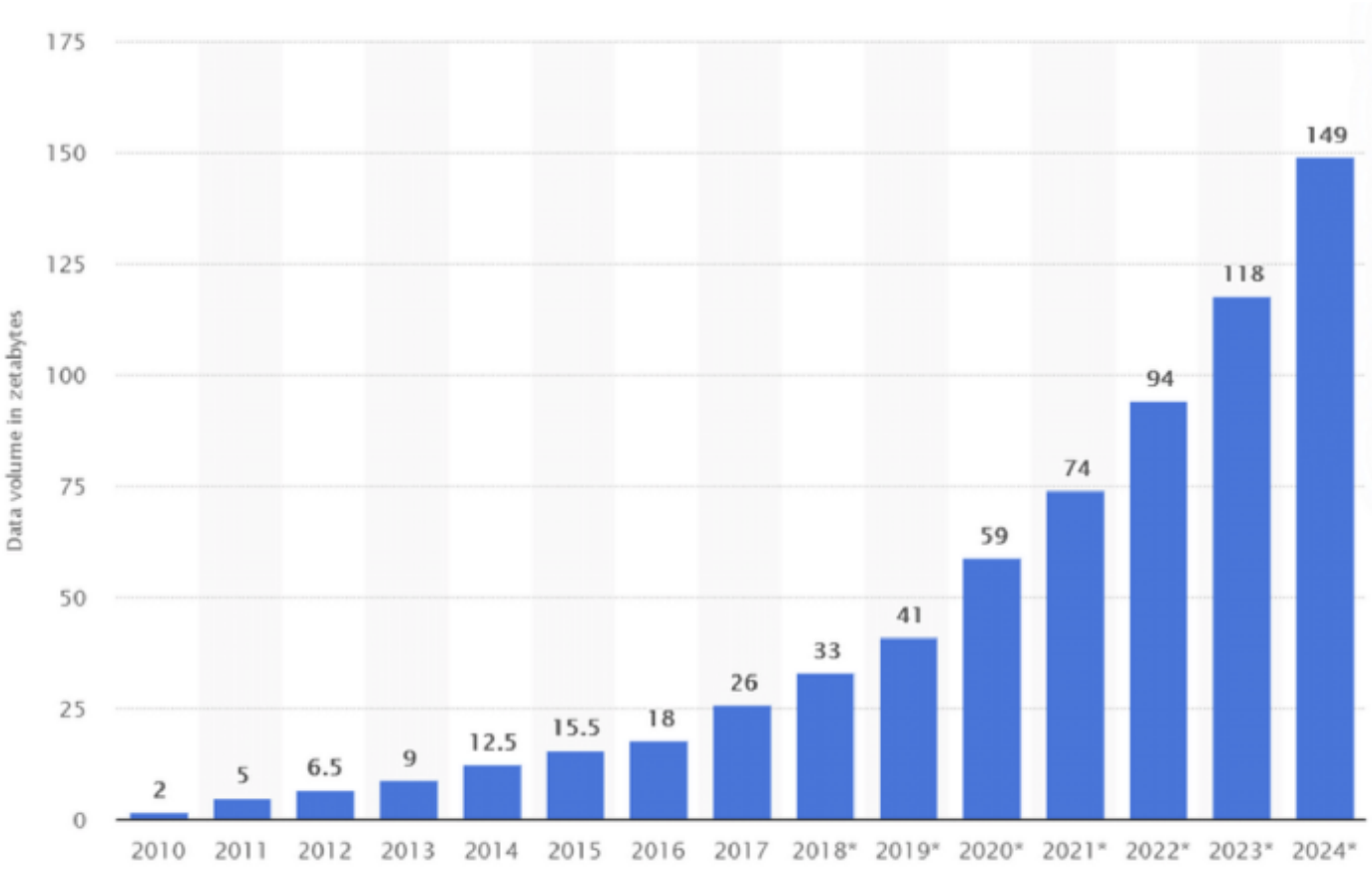
Topological Data Analysis (TDA)

TDA is an on-the-rise data science tool that looks the shape of data. It consists of a range of different approaches with an underlying theme of extracting structure (i.e. shapes) from unstructured data (i.e. point clouds). In this first article, I will give an accessible introduction to TDA that focuses less on the mathematical terminology and more on big picture ideas. Future posts will discuss two specific techniques under the umbrella of TDA: the Mapper algorithm and persistent homology.

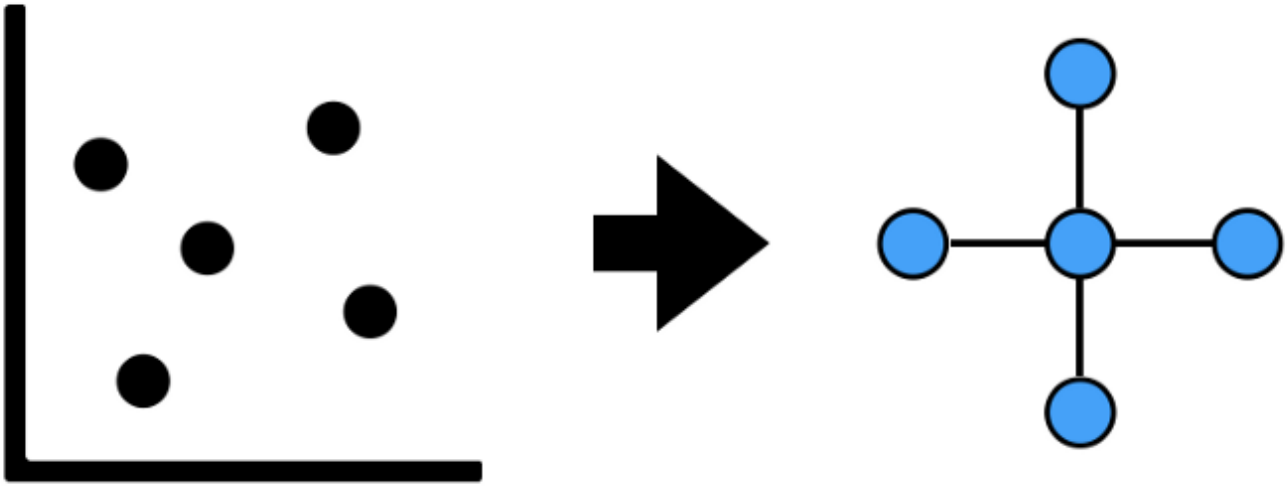
The Rise of Data

Data volumes across domains seem to be increasing at an accelerating rate. This has served as fuel for the many modern technologies e.g. NLP, image recognition, self-driving cars, etc. Although data has been a key for innovation, there's a devil in the details.



Mo' data, mo' problems. Any practitioner will tell you, data from the real-world is often noisy. A significant amount of care and effort is required to take what we measure and transform it into something we can analyze.

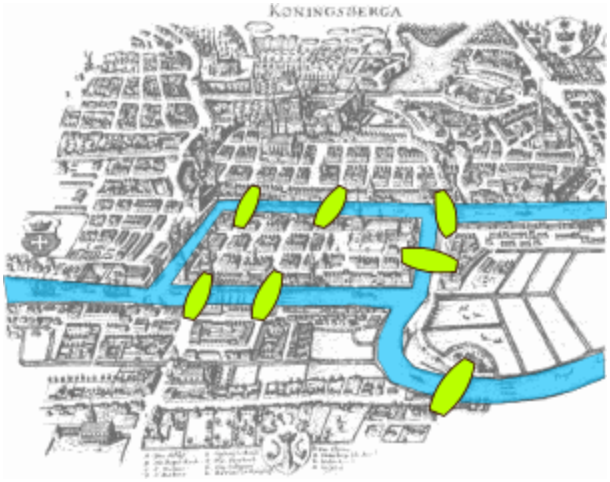
This is where **Topological Data Analysis (TDA)** can help. Rather than working with raw data directly, TDA **aims to extract the underlying shape of data.** From this shape, we can evaluate topological features which (typically) fair much better to noise than the raw data itself.



Topology

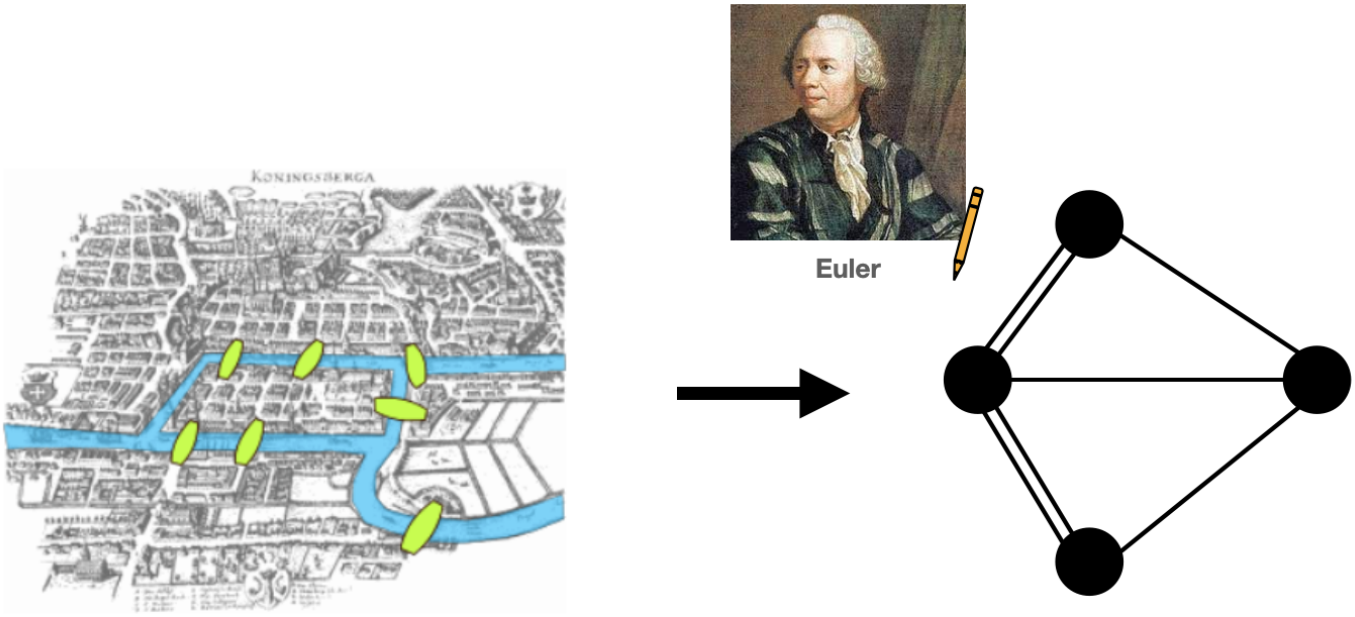
TDA is built on ideas from the mathematical field of topology. The story of topology goes back to a famous problem in math called the ***7 Bridges of Königsberg.**

Königsberg was a city have consisting of 4 land masses connected together by 7 bridges (pictured below). The famous problem is: how can one cross all 7 bridges in a single path, but only travel across each bridge once?



If you tried to find such a path with no luck, don't feel bad; no such path exists. However, demonstrating this lead famous mathematician Leonhard Euler to lay the foundation for modern day graph theory and topology.

So, how did he do it? What Euler did was draw a picture of Königsberg, but not like the map above, something much more basic. He drew what we now call a **graph**, which is just ***a set of dots connected by lines.*** The image below illustrates this.



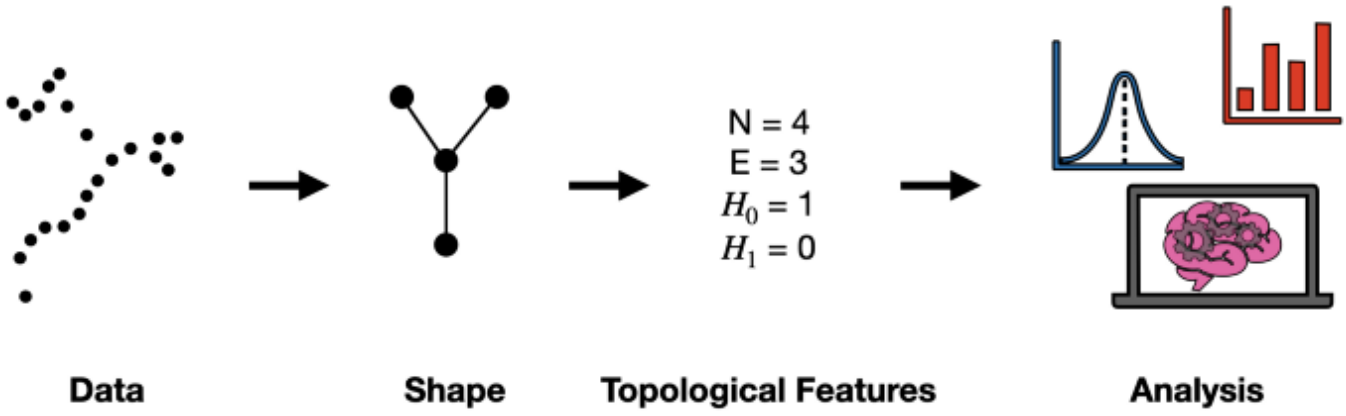
This **graph represents the essential elements of Königsberg** relevant to the problem. Each dot corresponds to a land mass in Königsberg, and two dots are connected by a line if the corresponding land masses are connected by a bridge. This simplified depiction of the problem may seem trivial, but it allowed Euler to solve the problem and change mathematics.

Topological Data Analysis (TDA)

The basic idea of TDA is reminiscent of what Euler did to solve the 7 Bridges of Königsberg problem. We take real-world data (e.g. a map of Königsberg) and extract the underlying shape that is most relevant to our problem (e.g. a graph representing Königsberg). The **key benefit** to doing this is we boil down the data to its core structure which is (hopefully) **invariant to noise.

The Pipeline

TDA isn't a single technique. Rather, it is a collection of approaches with a common theme of extracting shapes from data. A generic TDA pipeline is described in the paper by Chazal and Michel [1]. A visual overview of this pipeline is shown below.



The pipeline starts with data. Generally, we can think of any dataset as being an N-dimensional point cloud. This is comes from considering the N variables in the dataset as axes for an N-dimensional space in which data points live.

The next step is to generate shapes based on the point cloud. There are many ways to do this, which distinguishes different TDA approaches, as we will discuss in the next blogs in this series.

Once we have the shape of our data, we can characterize its topological features. There are again multiple way we can characterize shapes. For example, given a graph (like in the image above) we can count the number of dots and lines. We can also turn to something called **homology** and count the **number of "holes"** in the data.

As a final step, we can use these topological features to do analysis. This could be something as simple as categorizing data based on the statistics of topological features, or something more sophisticated, like using topological features as inputs to a machine learning model.

The Mapper Algorithm

Graph it like Euler

In this post, I will dive into a specific technique under the umbrella of **Topological data analysis (TDA)** called the Mapper algorithm. I will start by describing how the approach works, then walk through a concrete example with code.

Key Points

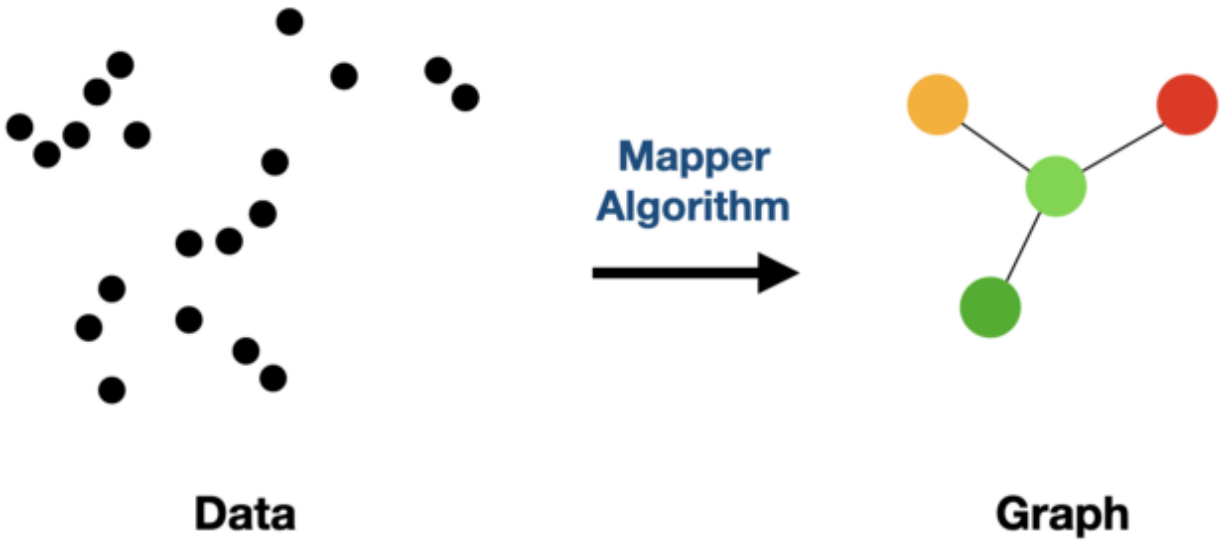
- The Mapper algorithm translates data into an interactive graph
- Mapper is well suited for exploratory data analysis and visualizing high-dimensional data

Graph it like Euler

In the previous post of this series, the 7 Bridges of Königsberg problem was presented. The famous problem was eventually solved by mathematician Leonhard Euler, who solved it by drawing a simplified version of the Königsberg map, which we now call a graph.

A graph is a picture consisting of dots connected by lines. Graphs are can be used to represent things from the real-world, for example, the city of Königsberg. In Euler's graph, the dots (nodes) correspond to land masses in Königsberg, and the lines (links) exist between nodes if the corresponding land masses share a bridge.

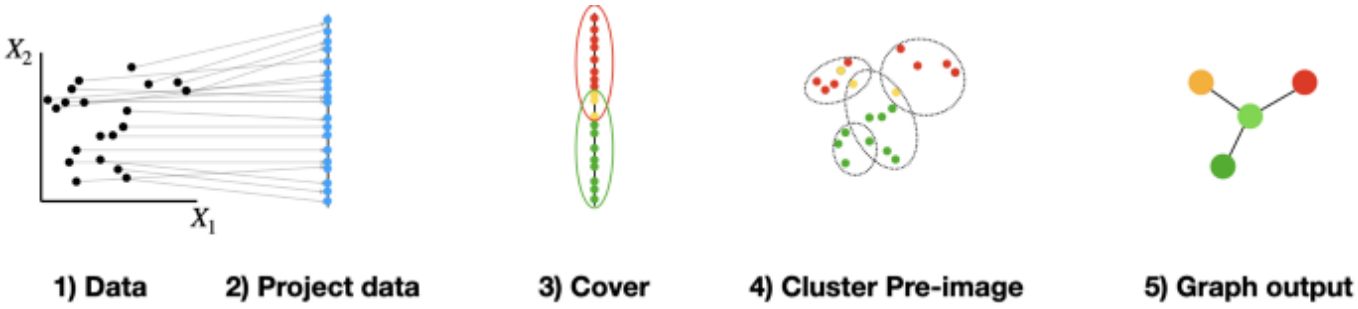
The basic idea of the Mapper algorithm is to construct a graph, just like Euler did. However, instead of starting from a map with land masses and bridges, we start with a set of points living in some N-dimensional space (i.e. data). A simple example of this is illustrated below for 2D data.



How does it work?

The mapper algorithm is a way to transform data into a graph. This is easy to understand, but how does it work? The process is a bit sophisticated, so I apologize in advance for resorting to the use of mathematical jargon. If you just want to dive right into a practical example feel free to skip ahead to the example code below.

> The Mapper algorithm can be broken down into **5 basic steps** as illustrated in the figure below.



- any dataset can be viewed as a point cloud by taking your N variables to be axes that define an N-dimensional space in which your data can live.

In the figure above, the data is 2-dimensional since we have 2 axes. However, there is no limit to the dimensionality of the input data, which makes the Mapper algorithm well suited for high-dimensional datasets.

- **The data is projected into a lower dimension**. The projection is governed by what is called a **lens function, filter**, or even just f. This function maps the higher dimensional input data into a lower dimensional representation.

There is no restriction on how one can define f. It can be as simple as just dropping all but one dimension, or a multi-step dimensionality reduction strategy using things like: PCA, ICA, UMAP, an autoencoder, etc. In the figure above, the 2-dimensional data is projected into 1 dimension.

- cover is defined for the projected data. All this means is we define overlapping subsets of the projected data points. In the figure above, the projected data is split into 2 subsets, indicated by a red and green circle, respectively. The subsets overlap, and the points in their intersection are colored yellow.

- we cluster the pre-image (really sorry about that). Now that's just math-speak for the process described in the next paragraph. Each point in the subsets from Step 3 (i.e. the red, green, and yellow points) corresponds to a point in the original dataset (i.e. Step 1), this is what we mean by pre-image.

What we mean by "cluster the pre-image", is to take a subset in the cover (e.g. the points in the red circle in Step 3), look at which points they correspond to in the original dataset (i.e. the red and yellow points in Step 4), and apply our favorite clustering algorithm to these points (e.g. k-means, DBSCAN).

We then repeat this process for every subset defined in Step 3. We can see in Step 4 of the above figure, that there will be clusters that share data points. Which brings us to the final step.

Finally, we **construct a graph based on the clusters**. More specifically, we draw a node corresponding to each of the clusters in Step 4, and connect two nodes together if they have any members in common.

And voila, after that 5-step process we have done what Euler did all those years ago, but now for a generic dataset. Additionally, each step of the process provides an incredible level of flexibility (e.g. choice of lens function, clustering algorithm, and cover definition), which is a double-edged sword.

It's nice because it makes the algorithm adaptable to different datasets, but presents a challenge in determining the best choice of hyperparameters. In practice, getting a good output graph will either take some trial and error or incredible insight or both. In the next section, I will walk through an example of what this might look like in practice.

Example Code: Exploratory Data Analysis of S&P 500

In this example, we will go through an application of the Mapper algorithm to financial data. More specifically, we will consider stock prices of companies in the S&P 500 from January 2020 to April 2022. The S&P 500 is an index which aggregates the performance of 500 top U.S. companies. We will then generate an interactive graph summarizing the high-dimensional stock data (i.e. ~500 variables) using the Mapper algorithm.

The first step is to import helpful Python libraries. The key library used here is Kepler Mapper, which is part of the scikit-tda ecosystem.

In [ ]:
```
!pip install yfinance
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
!pip install kmapper
```

In [ ]:
```
pip uninstall umap
```

In [ ]:
```
pip install umap-learn
```

In [1]:
```
import yfinance as yf    #FOR DATA

#FOR MAPPER
import kmapper as km
from kmapper.jupyter import display

#FOR LENS FUNCTION
import umap
import sklearn
import sklearn.manifold as manifold

#FOR CALCULATIONS AND PLOTTING
import numpy as np
import matplotlib.pyplot as plt
```

Next, we will use the yfinance library to load in stock data.

In [2]:
```
# read text file with ticker names
filename = open("SP500_tickernames.txt", "r")
raw_tickernames = filename.read()
ticker_names = raw_tickernames.split("\n")
ticker_names = ticker_names[:len(ticker_names)-1]

# define date range
start_date_string = "2020-01-01"
end_date_string = "2022-06-06"
```

```
# pull historical data
raw_data = yf.download(ticker_names, start=start_date_string, end=end_date_string)
```

```
[**********************100%**********************]  495 of 495 completed

4 Failed downloads:
- INFO: No data found, symbol may be delisted
- XLNX: No data found, symbol may be delisted
- PBCT: No data found, symbol may be delisted
- VIAC: No data found, symbol may be delisted
```

Then we tidy up the data to make it more suitable for analysis. Namely, we keep only daily closing prices, put the data in a numpy array, and scale each variable (i.e. company) to itself.

In [3]:

```
# get daily close prices and drop missing columns
df_close = raw_data['Adj Close'].dropna(axis='columns')

# convert pandas dataframe to numpy array, standardize ticker data, and transpose array
data = df_close.to_numpy()
data = data-np.mean(data, axis=0)/np.std(data, axis=0)
data = data.transpose() # so we can compare tickers and not days
```

Lastly, we compute the percent return of each ticker over the given time period, which will be used later to color the nodes in our output graph.

In [4]:

```
# calculate percent return of each ticker over entire date range
per_return = (df_close.to_numpy().transpose()[:,504] - df_close.to_numpy().transpose()[:,0
```

MAPPER

Up until this point, we've just loaded and prepped the data for analysis, which we can consider Step 1 from the earlier discussion. Now, we can finally get into the TDA stuff. We start by initializing an object that helps us implement the Mapper algorithm.

In [5]:

```
# initialize mapper
mapper = km.KeplerMapper(verbose=1)
```

```
KeplerMapper(verbose=1)
```

Next, we project our 495-dimensional data into 2D using isomap and UMAP. This corresponds to Step 2 from the previous visual overview.

In [6]:

```
#project data into 2D subscape via 2 step transformation,
#1>isomap
#2>UMAP

projected_data = mapper.fit_transform(data, projection=[manifold.Isomap(n_components=100,
```

```
..Composing projection pipeline of length 2:
        Projections: Isomap(n_components=100, n_jobs=-1)
                UMAP(random_state=1)
        Distance matrices: False
False
        Scalers: MinMaxScaler()
MinMaxScaler()
..Projecting on data shaped (490, 612)
```

```
..Projecting data using:
        Isomap(n_components=100, n_jobs=-1)


..Scaling with: MinMaxScaler()

..Projecting on data shaped (490, 100)

..Projecting data using:
        UMAP(random_state=1, verbose=1)

UMAP(random_state=1, verbose=1)
Tue Jun  7 09:45:27 2022 Construct fuzzy simplicial set
Tue Jun  7 09:45:27 2022 Finding Nearest Neighbors
Tue Jun  7 09:45:29 2022 Finished Nearest Neighbor Search
Tue Jun  7 09:45:31 2022 Construct embedding

Tue Jun  7 09:45:32 2022 Finished embedding

..Scaling with: MinMaxScaler()
```

Then, with one line of code we can do Steps 3–5. In other words, we generate a cover, cluster within the pre-image of each subset in the cover, and output a graph.

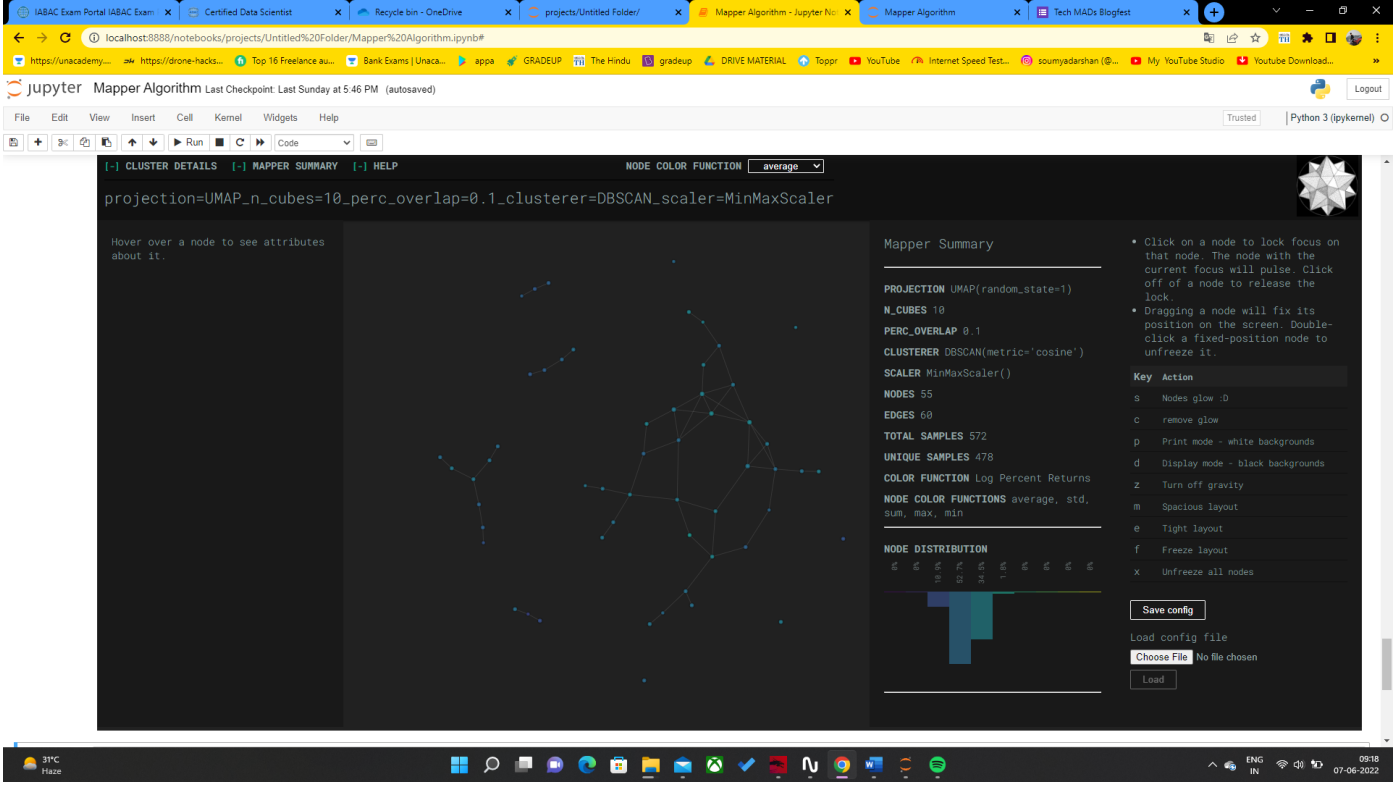
In [7]:
```
# cluster data using DBSCAN
G = mapper.map(projected_data, data, clusterer=sklearn.cluster.DBSCAN(metric="cosine"))
```

```
Mapping on data shaped (490, 612) using lens shaped (490, 2)

Creating 100 hypercubes.

Created 60 edges and 52 nodes in 0:00:00.025586.
```

Finally, we visualize the graph.

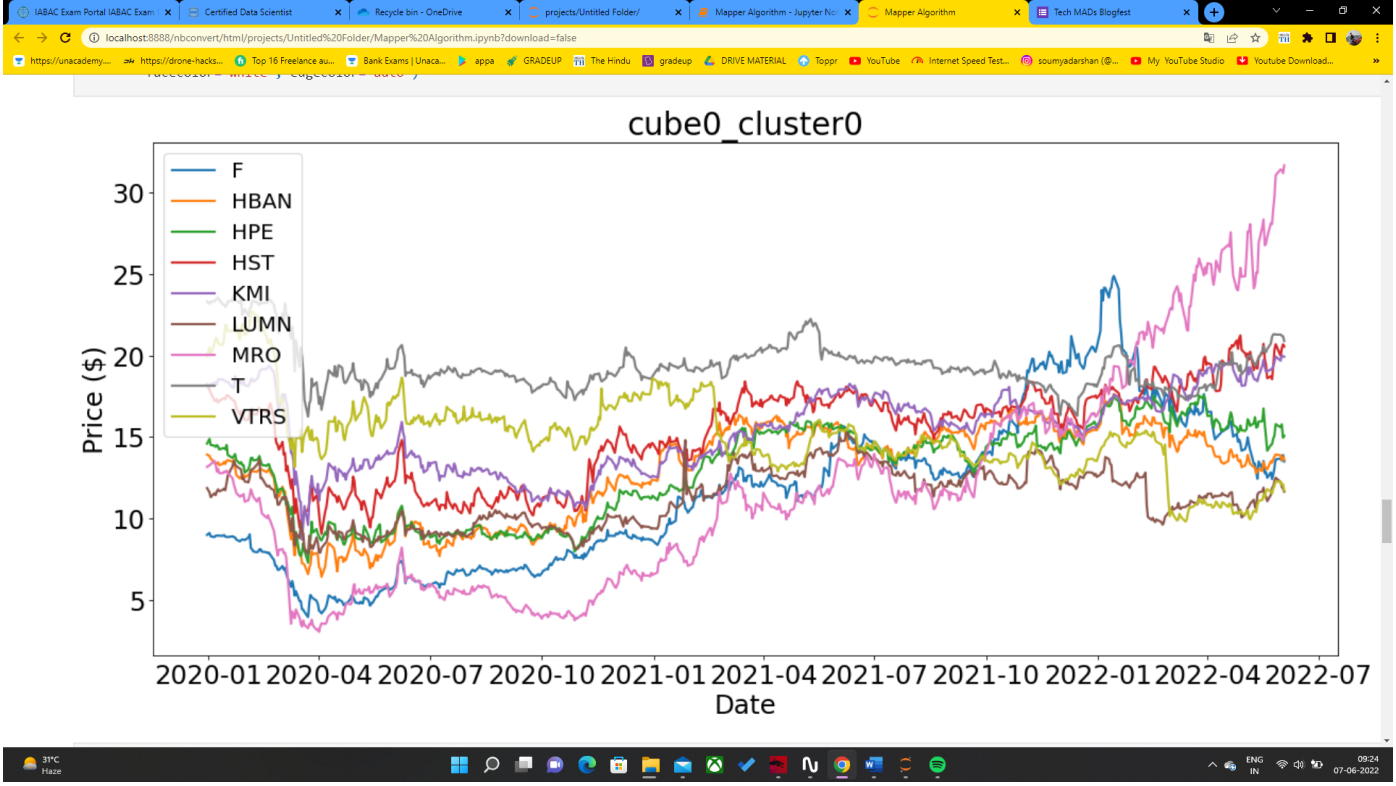
In [ ]:
```
# define an excessively long filename (helpful if saving multiple Mapper variants for sing
fileID = 'projection=' + G['meta_data']['projection'].split('(')[0] + '_' + \
'n_cubes=' + str(G['meta_data']['n_cubes']) + '_' + \
'perc_overlap=' + str(G['meta_data']['perc_overlap']) + '_' + \
'clusterer=' + G['meta_data']['clusterer'].split('(')[0] + '_' + \
'scaler=' + G['meta_data']['scaler'].split('(')[0]
```

In [ ]:
```
# visualize graph
mapper.visualize(G,
                 path_html="mapper_example_" + fileID + ".html",
                 title=fileID,
                 custom_tooltips = df_close.columns.to_numpy(),
                 color_values = np.log(per_return+1),
                 color_function_name = 'Log Percent Returns',
                 node_color_function = np.array(['average', 'std', 'sum', 'max', 'min']))

# display mapper in jupyter
km.jupyter.display("mapper_example_" + fileID + ".html")
```



In [ ]:

```
nodeid = 'cube0_cluster0'
node = G['nodes'][nodeid]


plt.figure(figsize=(18, 8), dpi=80)
plt.rcParams.update({'font.size': 22})

for i in node:
    plt.plot(df_close.iloc[:,i], linewidth=2)

plt.legend(list(df_close.columns[node]), fontsize=18)
plt.xlabel('Date')
plt.ylabel('Price ($)')
plt.title(nodeid)

plt.savefig("mapper_example_" + fileID + ".png", dpi='figure', format=None, metadata=None,
        bbox_inches=None, pad_inches=0.1,
        facecolor='white', edgecolor='auto')
```



In [ ]:

```
# convert notebook to python script
!jupyter nbconvert --to script mapper_example.ipynb
```

Conclusion

The Mapper algorithm provides a way translate data into a graph. This is well suited for things like exploratory data analysis and visualizing high-dimensional data. In this article, I walked through how the algorithm works and shared a concrete example with code of using Mapper to analyze market data.

SUBMITED BY:- SOUMYADARSHAN DASH