An Introduction to Topological Data Analysis

With the exorbitant amount of data available, including data about ourselves, methods for analyzing this data, such as linear regression, have gained eminence even outside of analysis professions. As Deloitte practitioners, we are often dealing with large datasets, such as healthcare data, where hundreds or thousands of dimensions are common. Deriving even basic data insights becomes much more difficult with this type of complex, high dimensional, large datasets. One technique to deal with complex, high dimensional data is Topological Data Analysis, also known as TDA.

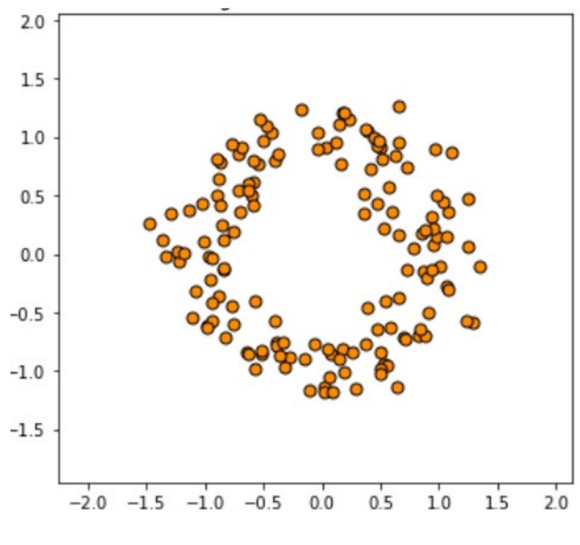
As an extension of Topology, a branch of mathematics that studies how space is connected and invariant through continuous deformations, TDA provides a compact geometric representation of data that reveals its underlying structure and can be used extract hidden insights from the data.

In the rest of the article, we will briefly review TDA, specifically two implementations: *persistent homology* and the *Mapper algorithm*. Persistent homology is a technique to identify topological features that persist when changing the measurements of relationships between data points and Mapper is an algorithm which provides a succinct summary of the shape of the data. We will even show how TDA has been used to bring a new perspective to basketball by reconstructing NBA player categories types based on their per-minute performance statistics.

Persistent Homology

In order to understand persistent homology, we will define the relevant topological terms while explaining the rationale behind this technique.

The following point cloud is clearly an approximation of a circle.

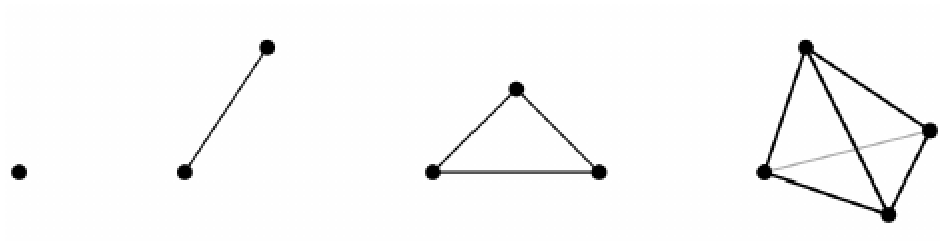


https://gjkoplik.github.io/pers-hom-examples/1d_pers_2d_data_widget.html

However, the question is how can we enable a program to identify it as such? To do so, we will build a series of distance relationships to connect the points, and track how changing the distance changes the relationships between all the points. First, we need to define some relevant terms.

Simplicial Complex: a simplicial complex is built from points, edges, triangular faces, etc. In the below diagram, from left to right, we have a 0-simplex, 1-simplex, 2-simplex, and 3-simplex.

Simplices Diagram

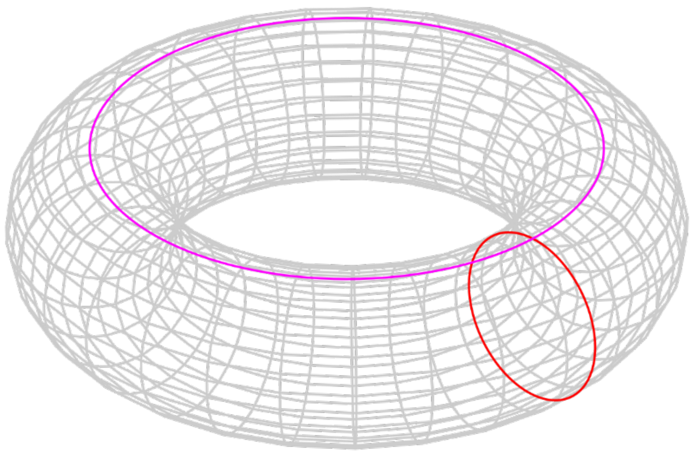


https://www.researchgate.net/figure/From-left-to-right-0-simplex-1-simplex-2-simplex-3-simplex_fig24_283091885

Homology: the main tool used to distinguish shapes and is computable via linear algebra, an i-dimensional homology H_i counts the number of i-dimensional holes

- H_0 can be thought of as the number of connected components, so in a point cloud with no connections the number of connected components would be equal to the number of points
- H_1 can be thought of as the number of one-dimensional holes
- H_2 can be thought of as the number of two-dimensional holes or voids enclosed by a surface

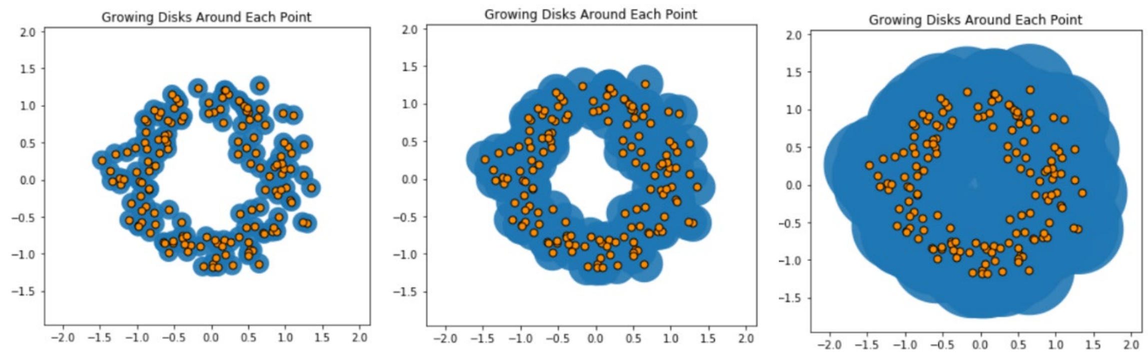
As an example, let us examine the torus. A torus is only one connected component, so it has H_0 of 1. There are two "holes", as shown in the diagram, so it has H_1 of 2. And finally, there is only one void enclosed by a surface – this could be filled with Bavarian cream and it would not spill out until you took a bite! So, it has H_2 of 1.



https://en.wikipedia.org/wiki/Torus#/media/File:Torus_cycles.svg

Now, returning to the point cloud, we choose a distance *d*, connect pairs of points that are not further apart than *d*, fill in complete simplices to obtain an abstract simplicial complex (the Rips-complex), then use homology to detect topological features. In the case of our point cloud, the topological feature we are looking for is a single one-dimensional hole.

In each of the diagrams below we see what the point cloud looks like when we use a different distance *d* to connect the points. In the center diagram, the one-dimensional hole is evident and can be found as a topological feature. In the last diagram, this one-dimensional hole has disappeared because the distance *d* is so large that all the points are connected to one another.

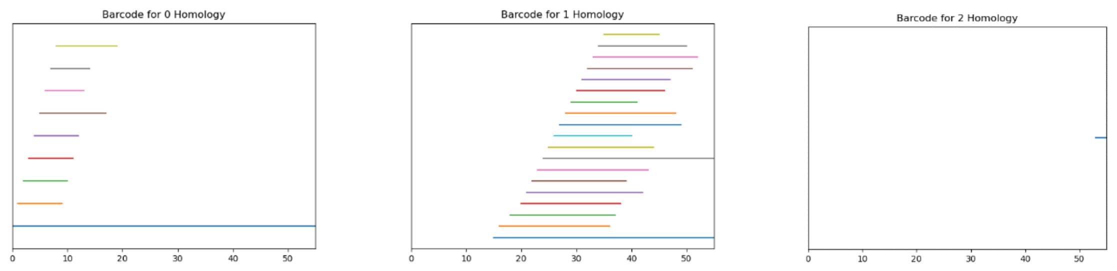


https://gjkoplik.github.io/pers-hom-examples/1d_pers_2d_data_widget.html

As we can see it might be difficult to choose *d* in a way that reconstructs our shape. We have a goldilocks problem: too small and we don't get any connected components, too big and everything is connected, so we need to get it just right so that we reconstruct the one-dimensional hole. Persistent homology gives us an interesting way around this problem: we choose all distances and track when topological features appear and disappear. In the case of the one-dimensional hole, it may appear at distance d_1 and disappear at distance d_2. This means that the persistence of this hole is (d_1, d_2).

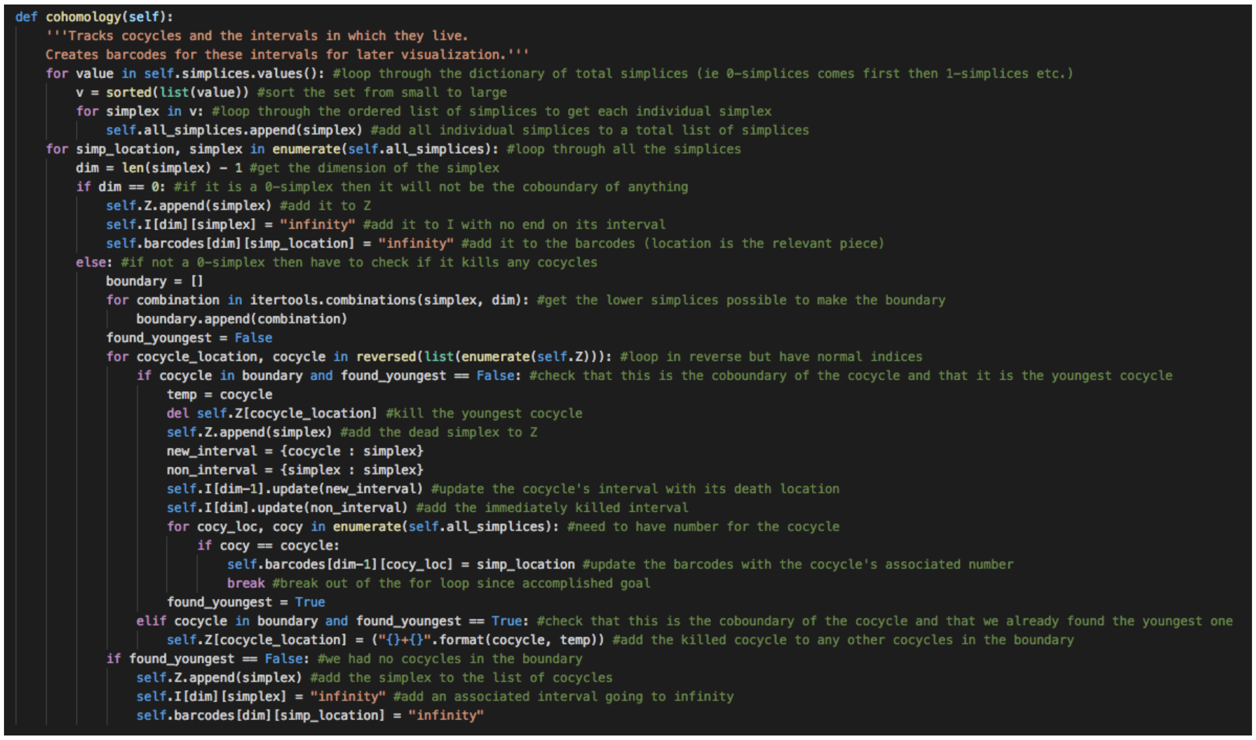
This (d_1, d_2) can be represented as a bar, and a collection of bars is called a barcode. Short bars, ones that do not persist for a long time as distance *d* changes, represent noise in the data. Whereas long bars represent features.

We can see here the barcodes for a torus. The lines that go to the far-right edge of the graph are indicative of cycles that never end, that is, they persist throughout the life of the torus. These lines are commonly known as homology generators and they are used to calculate the homology of a shape. For a torus, the dimension of H_0 is 1, the dimension of H_1 is 2, and the dimension of H_2 is 1. For each of these homologies we can see that the number of lines that go to the far-right edge of the graphs corresponds with the dimension of the homology. For example, the dimension of H_2 is 1 and there is one line that goes to the far-right edge of the graph. On the graphs, the horizontal axis represents the measurement distance *d* while the vertical axis is an arbitrary ordering of homology generators.



This is persistent homology. It takes an input of increasing spaces and provides an output of the barcode. The genius of a barcode representation is the ability to qualitatively filter out topological noise and capture significant events. Persistent homology not only tells us what homological features exist at which scales, as demonstrated in the barcodes, but also is a form of generalizing clustering. It includes higher-order homological features in addition to the 0-dimensional homology, the number of connected components, which can be thought of as the clusters.

A persistence barcode is computable via linear algebra and has at worst *$O(n^3)$* computational complexity, where *n* is the number of simplices. The most efficient method for computing a persistence barcode utilizes *cohomology*, which can be thought of as homology with more mathematical structure, rather than homology as discussed above. But the methodologies are similar whether using cohomology or homology. If you are interested, the commented code snippet below walks through the barcode computation and gives an idea of the mathematics powering persistence barcodes.

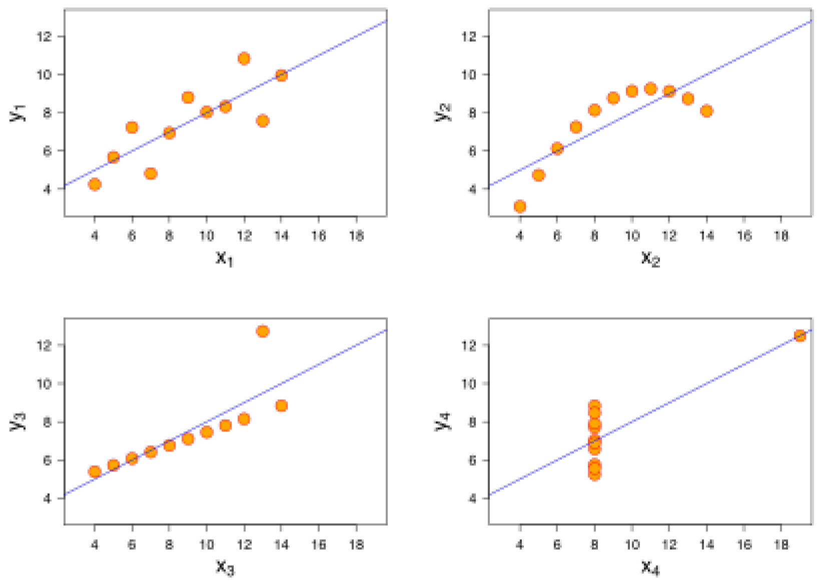


```python
def cohomology(self):
    '''Tracks cocycles and the intervals in which they live.
    Creates barcodes for these intervals for later visualization.'''
    for value in self.simplices.values(): #loop through the dictionary of total simplices (ie 0-simplices comes first then 1-simplices etc.)
        v = sorted(list(value)) #sort the set from small to large
        for simplex in v: #loop through the ordered list of simplices to get each individual simplex
            self.all_simplices.append(simplex) #add all individual simplices to a total list of simplices
    for simp_location, simplex in enumerate(self.all_simplices): #loop through all the simplices
        dim = len(simplex) - 1 #get the dimension of the simplex
        if dim == 0: #if it is a 0-simplex then it will not be the coboundary of anything
            self.Z.append(simplex) #add it to Z
            self.I[dim][simplex] = "infinity" #add it to I with no end on its interval
            self.barcodes[dim][simp_location] = "infinity" #add it to the barcodes (location is the relevant piece)
        else: #if not a 0-simplex then have to check if it kills any cocycles
            boundary = []
            for combination in itertools.combinations(simplex, dim): #get the lower simplices possible to make the boundary
                boundary.append(combination)
            found_youngest = False
            for cocycle_location, cocycle in reversed(list(enumerate(self.Z))): #loop in reverse but have normal indices
                if cocycle in boundary and found_youngest == False: #check that this is the coboundary of the cocycle and that it is the youngest cocycle
                    temp = cocycle
                    del self.Z[cocycle_location] #kill the youngest cocycle
                    self.Z.append(simplex) #add the dead simplex to Z
                    new_interval = {cocycle : simplex}
                    non_interval = {simplex : simplex}
                    self.I[dim-1].update(new_interval) #update the cocycle's interval with its death location
                    self.I[dim].update(non_interval) #add the immediately killed interval
                    for cocy_loc, cocy in enumerate(self.all_simplices): #need to have number for the cocycle
                        if cocy == cocycle:
                            self.barcodes[dim-1][cocy_loc] = simp_location #update the barcodes with the cocycle's associated number
                            break #break out of the for loop since accomplished goal
                    found_youngest = True
                elif cocycle in boundary and found_youngest == True: #check that this is the coboundary of the cocycle and that we already found the youngest one
                    self.Z[cocycle_location] = ("{}+{}".format(cocycle, temp)) #add the killed cocycle to any other cocycles in the boundary
            if found_youngest == False: #we had no cocycles in the boundary
                self.Z.append(simplex) #add the simplex to the list of cocycles
                self.I[dim][simplex] = "infinity" #add an associated interval going to infinity
                self.barcodes[dim][simp_location] = "infinity"
```

Mapper

Another tool within Topological Data Analysis, that again leverages topological properties of shapes to derive insight, is the Mapper algorithm. One of the primary uses of Mapper, data visualization of complex, high dimensional, large datasets, can be justified by the example of Abscombe's Quartet.

As shown below, it is possible to have extremely different datasets that produce the same summary statistics. Without visualizing the data, statistical summaries could be easily construed to derive an incorrect conclusion. Stating that the Abscombe's Quartet datasets are equivalent because they have the same summary statistics, would be ignoring the vast differences in the data points themselves and their relationship to one another—they have important topological differences.



https://en.wikipedia.org/wiki/Anscombe%27s_quartet#/media/File:Anscombe's_quartet_3.svg

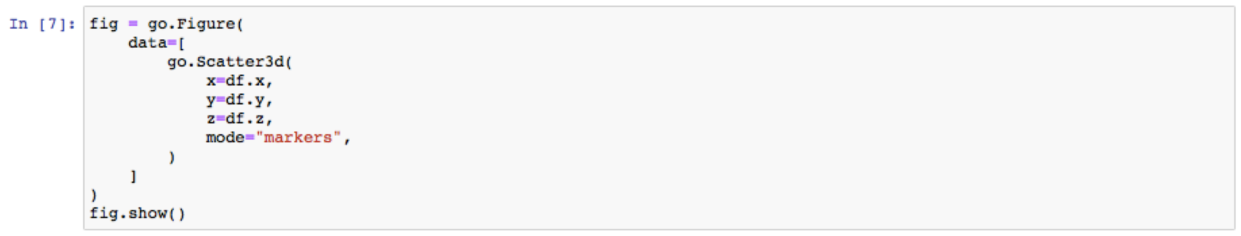
This is where Mapper comes in. Mapper provides a succinct summary of the shape of a dataset so that data visualization and clustering issues can be alleviated.

In general terms, Mapper fits a simplicial complex to data, with the goal being either clustering or data visualization. One key distinction between Mapper and general clustering algorithms, such as k-means, is that clustering partitions the data into disjoint subsets whereas Mapper is an open cover of the range. This means that the intervals overlap and together cover the whole dataset, connecting overlapping clusters. This is important because many clusters in real data are not disconnected components and so k-means is making unrealistic assumptions. Often the clusters are branches of some single connected component.

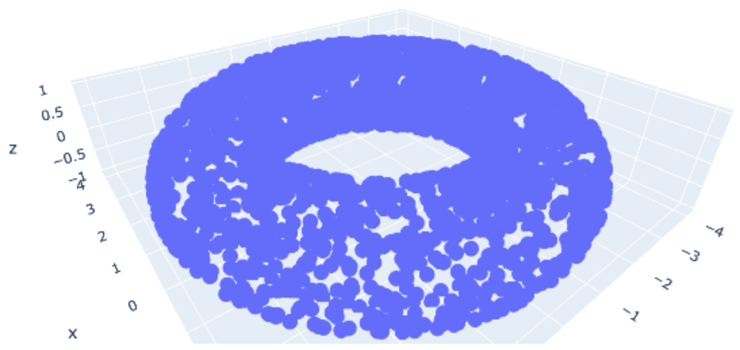
Mapper can be understood through three main steps:

1. Data points are mapped to the real numbers $\mathbb{R}$ using a filter function *f*.
2. The filter values are covered with overlapping intervals.
3. For each interval, a clustering algorithm is applied to the observations that were mapped into that interval from the first two steps.

For an example, we can use the Giotto-tda Python package, that is built on top of scikit-learn, to apply the Mapper function to a dataset of points representing a torus. A visualization of the generated torus points is shown below.



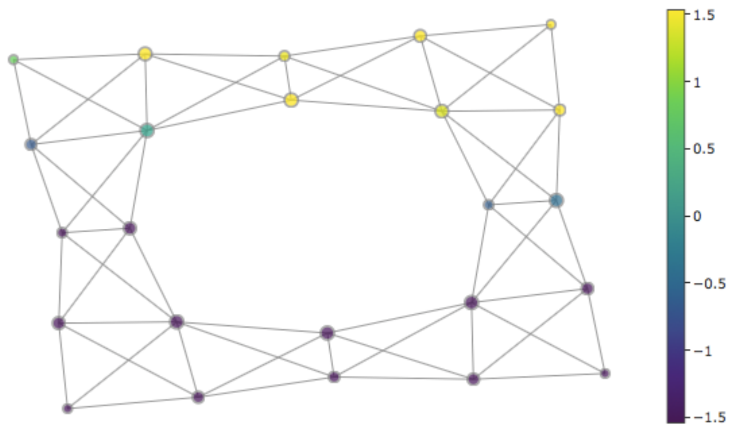
```
In [7]: fig = go.Figure(
            data=[
                go.Scatter3d(
                    x=df.x,
                    y=df.y,
                    z=df.z,
                    mode="markers",
                )
            ]
        )
        fig.show()
```



After importing the relevant packages and generating the torus points, the next step is to build a plotting function. Then we will configure the Mapper pipeline, where specifications, such as the cover type (this was the second step in the main Mapper steps listed previously), are decided.

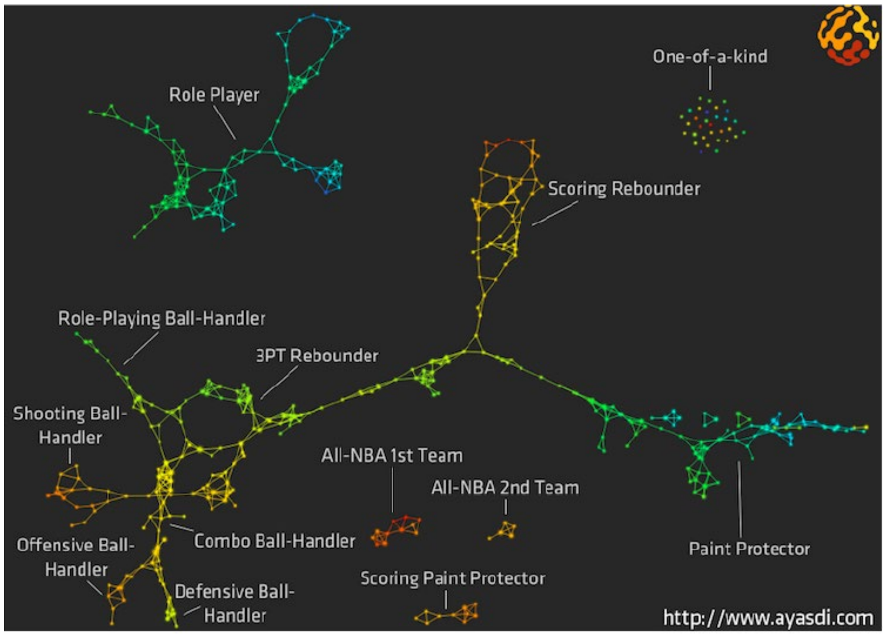
```
In [47]: # configure Mapper pipeline
         pipeline = make_mapper_pipeline(
             filter_func=None,
             cover=CubicalCover(n_intervals=5, overlap_frac=0.20),
             clusterer=DBSCAN(eps=.6),
         )
         # generate figure
         plot_interactive_mapper_graph(pipeline, coords)
```

kind	uniform	n_intervals	5	overlap_frac	0.2
algorithm	auto	eps	0.6	leaf_size	30
metric	euclidean	min_samples	5		



In this visualization, we can see that we have distilled the numerous data points to a much smaller number, but we have still captured the important top features of the torus. While there may appear to be a striking difference between our original torus and this new torus, they are topological the same. If you imagine a much larger, complex dataset that is too complex to visualize, Mapper would enable you to visualize this data without losing the topological features of the data.

There are numerous applications of TDA, such as time series analysis and forecasting, signal processing, feature extraction, and image analysis. It has even been applied to derive new insights for NBA basketball players. The NBA basketball application is one of the most accessible, as it is with relatively understandable data compared to many of the complex applications in physics and medical research. In the NBA basketball player application, players were categorized through a similarity network that was based on their in-games, per-minute performance statistics. The players were categorized into 13 new positions, compared to the standard 5 positions. This was a completely new way of thinking about NBA players and showed that there may be information lost about players when they are forced to be categorized in the standard 5 positions.



http://www.ayasdi.com/wp-content/uploads/_downloads/Redefining_Basketball_Through_Topological_Data_Analysis.pdf

In summation, Topological Data Analysis is a technique for dealing with complex, high dimensional data and has a vast array of disparate applications. Despite its mathematical complexity, due to the growing interest in TDA, there Python and R packages that make TDA much more accessible to those without a background in topology.

References:

https://www.youtube.com/watch?v=XfWibrh6stw&feature=youtu.be

https://jsseely.github.io/notes/TDA/

Persistent Homology: A Non-Mathy Introduction with Examples | by Gary Koplik | Towards Data Science

Applications of TDA to the Understanding of Disease and Drug Discovery | Institute for Mathematics and its Applications (umn.edu)

Topological Data Analysis taught by Anibal Medina Mardones

https://www.thekerneltrip.com/statistics/topological-data-analysis-tutorial/

https://towardsdatascience.com/visualising-high-dimensional-data-with-giotto-mapper-897fcdb575d7

https://github.com/giotto-ai/giotto-tda/blob/master/examples/mapper_quickstart.ipynb

https://towardsdatascience.com/getting-started-with-giotto-learn-a-python-library-for-topological-machine-learning-451d88d2c4bc

https://sauln.github.io/blog/tda_explanations/