MARKOWITZ PORTFOLIO THEORY USING TOPOLOGICAL NETWORKS

ROMIE BANERJEE

1. Introduction

Modern Portfolio Theory is a theory about how investors construct portfolios that maximise their expected returns for given levels of risk. The key insight of MPT was that risks(variance) and returns of securities should not be analysed individually, but as a whole portfolio in terms of how different securities move with respect to one another (expressed as covariance and correlation).

The fundamental principle of this theory is the possibility for investors to construct an efficient set of portfolios" that offers the maximum expected returns for a given level of risk.

The Efficient frontier: Different combinations of securities produce different levels of return. The efficient frontier represents the best of these securities combinations – those that produce the maximum expected return for a given level of risk.

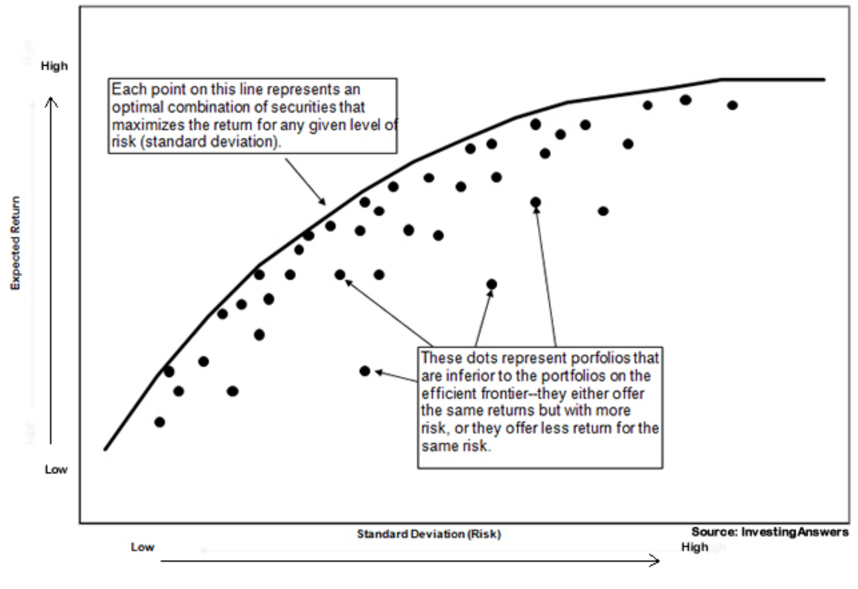

Figure 1. Efficient Frontier, Source: Investing Answers

The relationship securities have with each other is an important part of the efficient frontier. Some securities' prices move in the same direction under similar circumstances, while others move in opposite directions. The more out of sync the securities in the portfolio are (that is, the lower their covariance), the smaller the risk of the portfolio that combines them.

Date: November 23, 2018.

1.1. **Basic terminology with examples.** Every security/stock/risky asset is analysed by means of historical daily values. This can interpreted as a time series, or as a random variable (on time). The returns are the daily percentage differences from the previous day's values. This also forms a time series.

Given m-stocks $X_1, \dots, X_m$ and n numbers $b_1, \dots, b_m$ the linear combination $\Sigma_{i=1}^m b_i X_i$ is the linear combination of the corresponding time series, and hence can be interpresed as new security. Similarly with the returns time series.

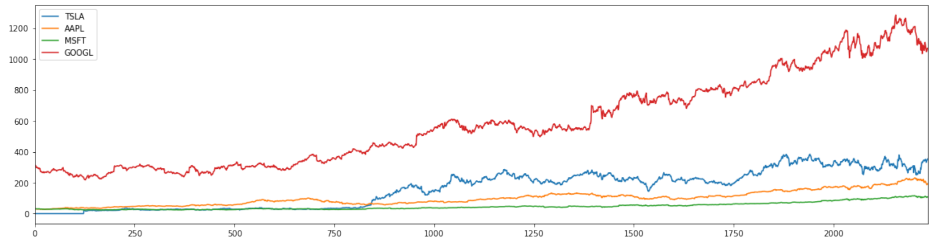

FIGURE 2. Sample historical daily stock prices

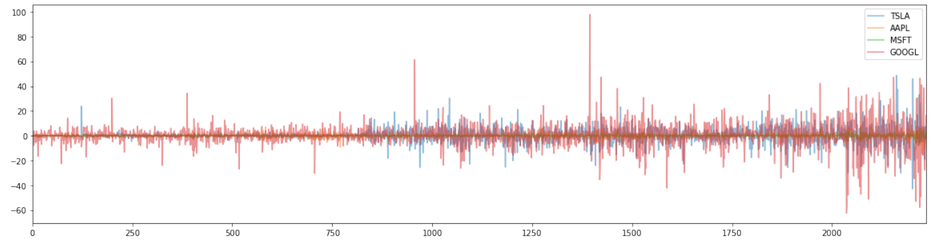

FIGURE 3. and their daily returns

A *portfolio* of m stocks is a security of the form $\Sigma_{i=1}^m b_i X_i$ where $0 \leq b_i \leq 1$ and $\Sigma_i b_i = 1$. Here b_i is to be interpreted as a the fraction of wealth allocated to asset X_i. We can allow negetive fractional values in case we allow shorting of stocks.

The *Expected Return* of a stock is the mean/average of the corresponding returns. The *risk/volatilty* of a stock is the variance of the corresponding returns.

1.1.1. *Computing portfolio returns and risk.*

(1) Expection/mean is linear: $\mathbb{E}(\Sigma_{i=1}^m b_i X_i) = b_i \Sigma_{i=1}^m \mathbb{E}(X_i)$
(2) The variance of a sum of two random variables is given by $\text{Var}(aX + bY) = a^2\text{Var}(X) + b^2\text{Var}(Y) + 2ab\text{Cov}(X, Y)$. In general,

$$\text{Var}(\Sigma_{i=1}^m b_i X_i) = \Sigma_{i=1}^m b_i^2 \text{Var}(X_i) + 2\Sigma_{1 \leq i < j \leq n} \text{Cov}(X_i, X_j)$$

As a consequence, if the random variables X_i are uncorrelated, then $\text{Var}(\Sigma_{i=1}^m b_i X_i) = \Sigma_{i=1}^m b_i^2 \text{Var}(X_i)$. The vectorized formula for the general case is

$$\text{Var}(\Sigma_{i=1}^m b_i X_i) = \mathbf{b}^T \mathbf{\Sigma} \mathbf{b}$$

where $\mathbf{b} = (b_1, \dots, b_m)^T$ is the weight vector and $\mathbf{\Sigma}$ is the $n \times n$ covariance matrix, $\mathbf{\Sigma}_{i,j} = \text{Cov}(X_i, X_j)$.

2. Markowitz Mean Variance Analysis

The problem: Form an optimal portfolio (minimize risk, maximize expected return) out of a collection of risky assets.

2.1. **The input:**

- m- risky assets: $\{X_i, i = 1, 2, \dots, m\}$
- Asset returns is m-variate random vector
$$R = (R_1, R_2, \dots, R_m)^T$$
- Mean and Covariance of returns
$$\mathbb{E}[R] = \boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_m)^T,\ \mathrm{Cov}[R] = \boldsymbol{\Sigma} = \begin{pmatrix} \Sigma_{1,1} & \dots & \Sigma_{1,n} \\ \dots & \dots & \dots \\ \Sigma_{n,1} & \dots & \Sigma_{n,n} \end{pmatrix}$$
- Portfolio: m-vector of weights representing fraction of portfolio wealth allocated in each asset (negetive weight amounts to shorting a stock).
$$\mathbf{w} = (w_1, \dots, w_m)^T,\ \Sigma w_i = 1$$
- Portfolio of assets:
$$R_{\mathbf{w}} = \mathbf{w}^T.R = \Sigma_1^m w_i R_i$$
Portfolio expected return:
$$\mathbb{E}[R_{\mathbf{w}}] = \mathbf{w}^T.\mathbb{E}[R] = \Sigma_1^m = w_i \alpha_i$$
Portfolio variance:
$$\mathrm{var}[R_{\mathbf{w}}] = \mathbf{w}^t \boldsymbol{\Sigma} \mathbf{w}$$

2.2. **Aim of Markowitz theory:** Evaluate different portfolios $\mathbf{w}$ using the mean-variance pair $(\mathbb{E}[R_{\mathbf{w}}], \mathrm{var}[R_{\mathbf{w}}])$ with preference for

- Higher expected return $\mathbb{E}[R_{\mathbf{w}}]$
- Lower variance $\mathrm{var}[R_{\mathbf{w}}]$

2.3. **Problem I: Risk Minimization.** For a given choice of target mean return α_0, choose a portfolio $\mathbf{w}$ to

(1) Mimimize: $\mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w}$
(2) Subject to: $\mathbf{w}^T.\boldsymbol{\alpha} = \alpha_0$ and, $\mathbf{w}^T.\mathbf{1}_m = 1$

2.4. **Solution (closed form):** Apply the method of Lagrange multipliers to the convex optimization problem subject to linear constraints. The optimal portfolio is given by
$$\mathbf{w_0} = \lambda_1 \boldsymbol{\Sigma}^{-1} \boldsymbol{\alpha} + \lambda_2 \boldsymbol{\Sigma}^{-1} \mathbf{1}_m$$
where the constants λ_1 and λ_2 are given by
$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = \begin{pmatrix} a & b \\ b & c \end{pmatrix}^{-1} \begin{pmatrix} \alpha_0 \\ 1 \end{pmatrix}$$
where, $a = \boldsymbol{\alpha}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\alpha}, b = \boldsymbol{\alpha}^T \boldsymbol{\Sigma}^{-1} \mathbf{1}_m$, and $c = \mathbf{1}_m^T \boldsymbol{\Sigma}^{-1} \mathbf{1}_m$.

2.5. **Variance of the optimal portfolio with return α_0:** With the given values of λ_1 and λ_2 the variance of the optimal portfolio $\mathbf{w_0}$ is given by
$$\mathrm{var}[R_{\mathbf{w_0}}] = \mathbf{w_0}^T \boldsymbol{\Sigma} \mathbf{w_0} = \frac{1}{ac - b^2} \left(c\alpha_0^2 - 2b\alpha_0 + a \right)$$

2.6. **Problem II: Expected Return Maximization:** For a given choice of target return variance σ_0^2 choose a portfolio to

(1) Maximize: $\mathbf{w}^T.\boldsymbol{\alpha}$
(2) Subject to: $\mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w} = \sigma_0^2$ and, $\mathbf{w}^T.\mathbf{1}_m = 1$

2.6.1. *Solution:* The optimal solution is obtained by application of Lagrange multipliers method to this optimization problem subject to quadratic constraints.

2.7. **Other related optimization problems.**

- Risk aversion optimization
- Mean-Variance optimization with risk-free asset

2.8. **The Efficient Frontier.** Collection of (E, V)-values for optimal portfolios. The efficient portfolio is defined as the points

$$\{(\mathbb{E}[R_{\mathbf{w_0}}], \mathrm{var}[R_{\mathbf{w_0}}]) \mid \mathbf{w_0} \text{ optimal}\}$$

in the E-V plane.

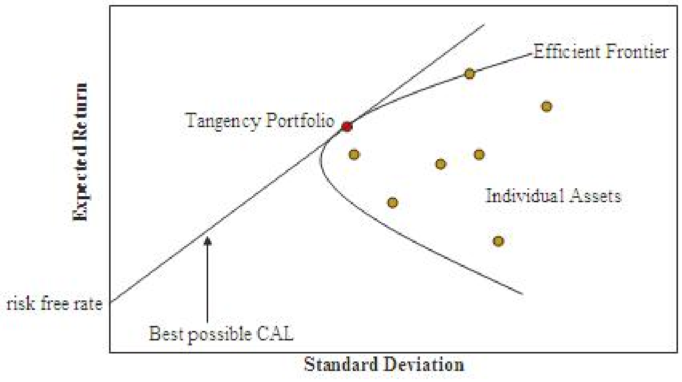

Figure 4. Efficient frontier (source: Wikipedia)

3. Computing Efficient Frontier using stock prices data in Python

We use historical daily stock prices of securities traded at NASDAQ from 31-12-2009 to present date. (Data source: Yahoo! Finance, Python code available separately)

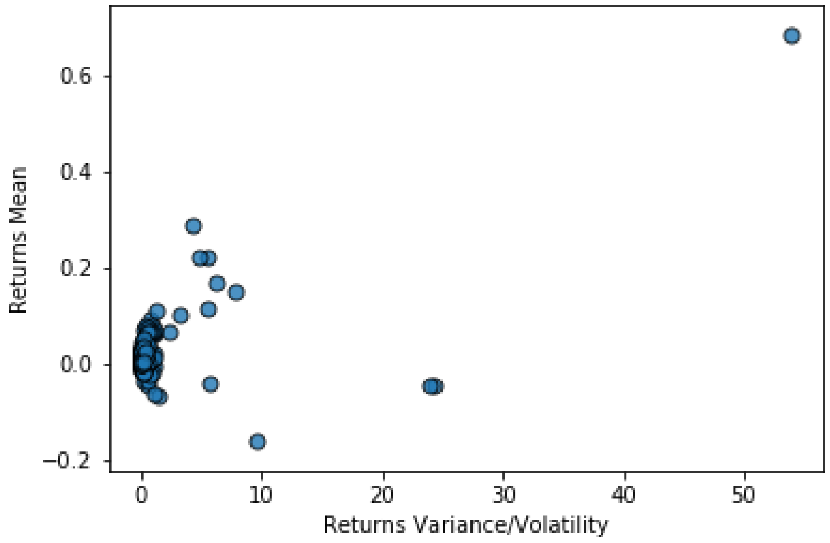

Figure 5. EV values for first 200 Nasdaq stocks(in terms of market capitalization)

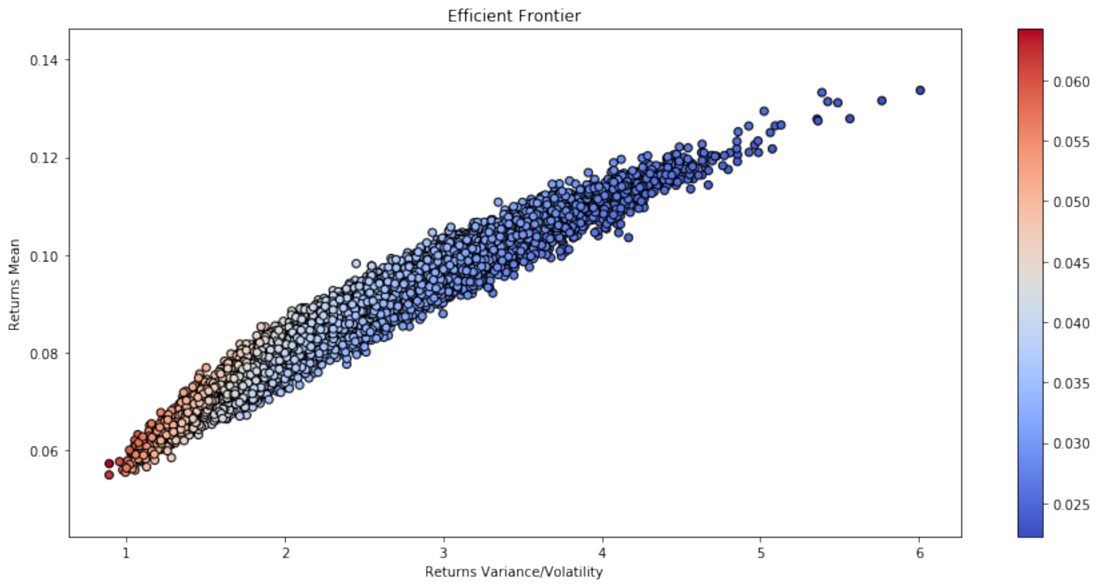

FIGURE 6. EV values for 10,000 random portfolios formed out of 50 stocks traded in Nasdaq. Color coded by the Sharpe ratios

4. Point Cloud Data

A *point cloud* is a collection of points (labels/data) with a notion of similarity/distance between any two points. This is made precise by the mathematical notion of a *metric space*.

Definition 4.1. A *metric space* is a set X is a function $d : X \times X \to \mathbb{R}$ satisfying the properties:

(1) $d(x, y) = 0$ if and only if $x = y$
(2) $d(x, y) = d(y, x)$ (symmetry)
(3) $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality)

A *point cloud* is a finite metric space.

Example: Any subset of $\mathbb{R}^n$ is a metric space with the metric induced by the l_2 norm in $\mathbb{R}^n$, $d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2$.

4.1. l_2 emdeddability.

Definition 4.2. A point cloud X is l_2^n embeddable if there is an isometric embedding of X in $\mathbb{R}^n$. In other words, for every point $x \in X$ there is a vector $\mathbf{x}$ such that this mapping is distance preserving.

Remark 4.1. l_2-embeddability of is useful for visualizing a point cloud.

4.1.1. *Condition for l_2-embeddability:* Let (X, d) be a metric space. Consider the matrix

$$M_{i,j} = \frac{1}{2}(d(1, x)^2 + d(1, y)^2 - d(x, y)^2).$$

Then X embeds in l_2^d, where d is the size of X, if M is positive semi-definite.

4.2. **Point Cloud to network.** : A network is a data structure consisting of a collection of vertices and edges. This can be expressed as a set $V = \{v_1, \ldots, v_n\}$ and edges $E = \{(v_i, v_j), \ldots\}$ consisting of pairs of vertices. The entire structure can be encoded by an adjacency matrix

$$A_{n\times n}(i,j) = 1 \,\text{or}\, 0$$

depending on the presence or absence of an edge.

4.2.1. *Point Cloud + scale = network.* Given a point cloud X and a real number (scale ≥ 0), we can construct a network N as follows:

- vertices of N = points of X
- for $x, y \in X$, the edge (x, y) exists in N if $d(x, y) <$ scale.

4.3. **Clustering a point cloud.** Clustering is an important category of unsupervised learning algorithms. We need a clustering mechanism to detect connected componets of a pount cloud. Some examples are

(1) DBSCAN, K-NN – applicable to point cloud data, detects connected components
(2) K-means – applciable to l_2-embedabble data (i.e. subsets of $\mathbb{R}^n$), partitions data into convex components

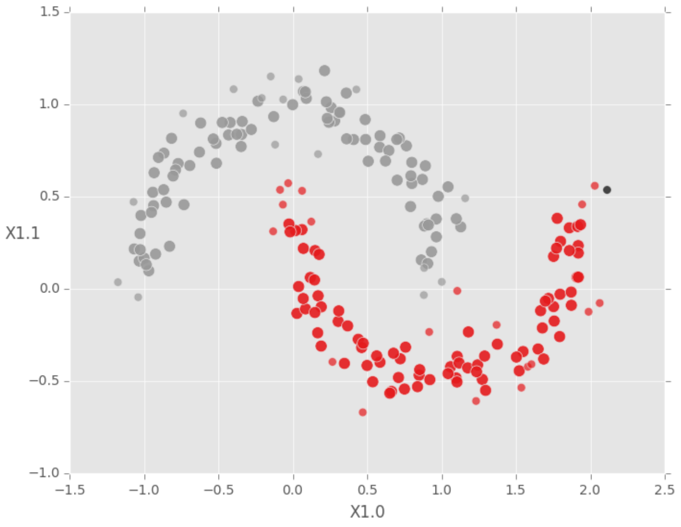

FIGURE 7. DBSCAN algorithm detects connected components

4.3.1. *Clustering using persistent homology.* Persistent homology is a method in topological data analysis that enables one to infer topological properties of a possible gemetric space from which a point cloud is sampled. The idea is to look at networks arising from a given point cloud and a choice of scale for different values of scales. The topological properties of the networks that persist through the varying values of scales are the persistent topological feature so of the point cloud.

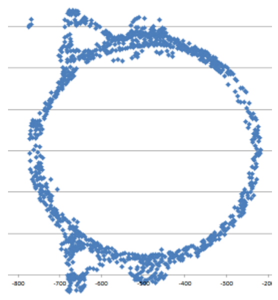

Figure 8. A point cloud having persistent topological features of a circle

5. Point cloud of Stocks

5.1. **Vector space of random variables.** The collection of random variables over a fixed probability space forms a normed vector space $\mathcal{RV}$. The k-norm on $\mathcal{RV}$ is defined by

$$\|X\|_k = \Big(\mathbb{E}[|X|^k]\Big)^{1/k}$$

This satisfies all the usual properties of a norm and define a metric on $\mathcal{RV}$,

$$d_k(X,Y) = \Big(\mathbb{E}[|X-Y|^k]\Big)^{1/k}$$

.

The 2-norm arises from an inner product on $\mathcal{RV}$ given by

$$\langle X, Y\rangle = \mathbb{E}(XY)$$

We'll denote the space $\mathcal{RV}$ with the 2-norm as $\mathcal{L}_2$. For $X, Y \in \mathcal{L}_2$,

- $\mathrm{Cov}(X,Y) = \langle X - \mathbb{E}(X), Y - \mathbb{E}(Y)\rangle$
- X and Y are uncorrelated if and only of $X - \mathbb{E}(X)$ and $Y - \mathbb{E}(Y)$ are orthogonal
- $\langle X, X\rangle = \|X\|_2^2 = \mathbb{E}(X^2)$

5.2. **Point cloud of stocks/securities.** Regarding a stock/security as a random variable of its historical daily returns, we have a normed vector space of stocks. Given a stock X and its corresponding returns R, its l_2-norm is calculated as

$$\|X\|_2 = \big(\mathbb{E}[|R|^2]\big)^{1/2}$$

Given two securities X and Y and their corresponding returns R_x and R_y, the *distance* between X and Y is calulated as

$$d(X,Y) = \big(\mathbb{E}[|R_x - R_y|^2]\big)^{1/2}.$$

We'll call this point cloud the *Stock Cloud.*

5.3. **Visualising the Stock Cloud.**

5.3.1. *By embedding in l_2^n.* : If the stock cloud in l_2 embdeddable in $\mathbb{R}^n$ (n = size of cloud) then the cloud can vizualized by applying PCA projection to 2 dimensions. The following diagram illustrates this using historical daily returns of 50 stocks traded in Nasdaq.

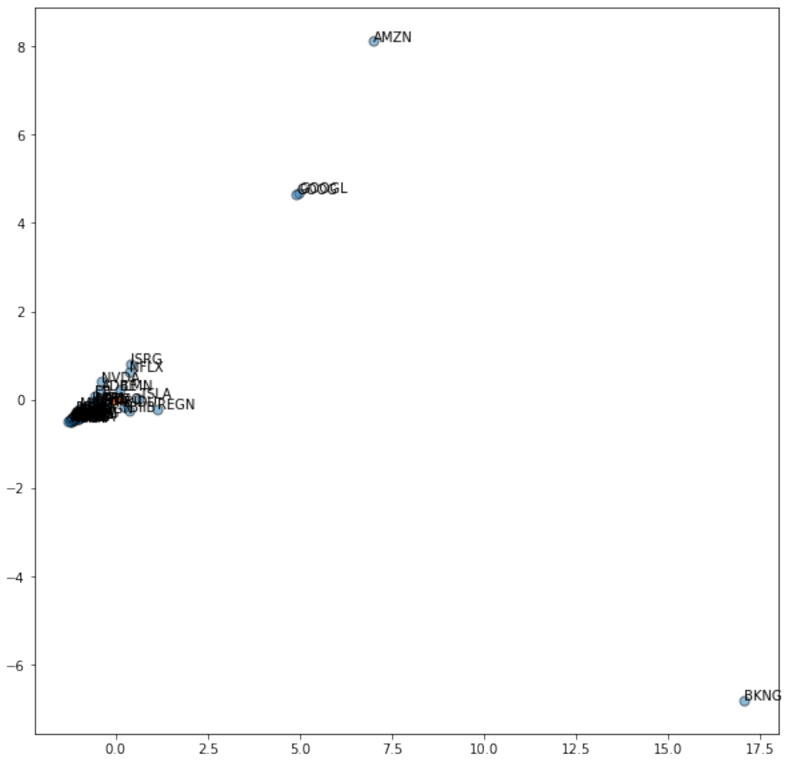

FIGURE 9. PCA projection of l_2 embedding of stock cloud of 50 Nasdaq stocks

5.4. **Visualizing the stock cloud using MAPPER.**

5.4.1. *MAPPER.* The Mapper is a topological mapping tool for point cloud data. In the simplest case, Mapper begins with a point cloud dataset X, and a continuous real valued function $f : X \to \mathbb{R}$, to produce a network. This function can be a function which reflects geometric properties of the dataset, such as the result of a density estimator, or can be a user definedfunction, which reflects properties of the data being studied. In the first case, one is attempting to obtain information about the qualitative properties of the dataset itself, and in the second case one is trying to understand how these properties interact with interesting functions on the dataset.

The basic idea behind Mapper can be referred to as partial clustering, in that a key step is to apply standard clustering algorithms to subsets of the original dataset, and then to understand the interaction of the partial clusters formed in this way with each other. That is, if U and V are subsets of the dataset, and $U \cap V$ is nonempty, then the clusters obtained from U and V respectively may have non-empty intersections, and these intersections are used in building a network.

5.4.2. *The mapper algorithm.*

(1) Begin with a continous map $f : X \to \mathbb{R}$ called a *Lens* and an finite open covering $\{\mathcal{U}_\alpha\}_{\alpha \in A}$ of the target
(2) The pullback $\{f^{-1}(\mathcal{U}_\alpha)\}_{\alpha \in A}$ is an open covering of X.
(3) For each α consider the decomposition of $f^{-1}\mathcal{U}_\alpha$ into connected components $\cup_i^{j_\alpha} V(\alpha, i)$, where j_α is the number of connected components.
(4) Let $W = \{W_\beta\}_{\beta \in B}$ be the set of all $V(\alpha, i)$ where $1 \leq i \leq j_\alpha$ and $\alpha \in A$.

(5) Construct the *Nerve* of the covering W, which is a network with a vertex for every open cover W_β and an edege between W_{β_1} and W_{β_2} if their intersection is nonempty.

5.5. **Mapper on Stock Cloud.** We use a density estimator for the Lens function on the stock point cloud for the mapper algorithm. A density estimator gives high values for points in the vicinity of a large number of points and lower values for points which are isolated. In particular we'll us the *Gaussian Kernel* function:

$$f(x) = \lambda \sum_{y \in X} \exp\left(\frac{-d(x,y)^2}{\epsilon}\right)$$

High values of f are assigned to points in a densely packed pubset of X. Therefore th Gaussian kernel helps identify dense regions within the point cloud.

After applying Mapper to the Stock cloud of 200 Nasdaq stocks with the function f to the Gaussian kernel, we get the following network. Here each node represents a cluster (or single) of stocks. The size of the node is proportional to the size of the cluster. The color is graded according to the value of the function f. In this case darker color means closer to the Gaussian mean.

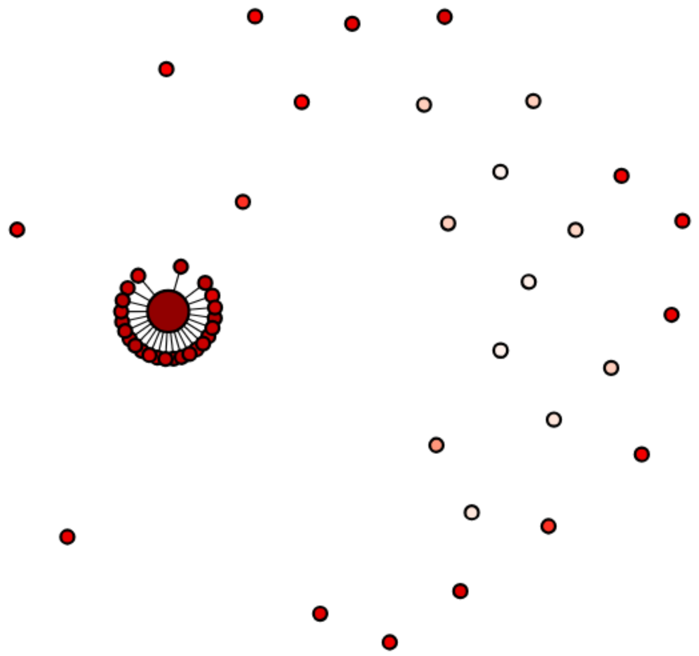

FIGURE 10. Mapper with Gaussian kernel on Stock Cloud

5.6. **Connection with Markowitz optimal portfolio.** The optimal solution for the minimum variance problem shows that a portfolio of stocks which have low levels of pairwise correlation have smaller risk. *In terms of the Stocks point cloud this amounts to saying that portfolio of stocks chosen from a dense subset of the point cloud have greater risk than a portfolio of stocks that are spread out along huge distances in the point cloud.*

The solution to minimum variance problem suggests allocation of more weight to stocks with small values of Gaussian kernel function in the mapper network for the stock cloud.

6. Code

6.1. **Python.** The following python code defines classes PointCloud and Netweok with methods for persistent components, l_2 embedding, Gaussian kernel mapper etc.

```
import numpy as np
import sympy as sp
import pandas as pd
import statistics as stats
import json
import math


def flatten(A): #flattens a nested list
        if A == []: return A
        if type(A[0]) == list:
            return flatten(A[0]) + flatten(A[1:])
        else: return [A[0]] + flatten(A[1:])


def cart(A,B): # cartesian product of two lists

    if A == []: return B
    elif B == []: return A
    else: return [flatten([x,y]) for x in A for y in B]

def mode(A): #Find most common element
    df = pd.DataFrame({"A":A})
    return df.mode().A.values.tolist()[0]



##.....Some linear agebra


def rref(A):
X = sp.Matrix(A).rref()[0]
return np.array(X).astype(np.float64)


def is_in_range(A,b):
    A = np.matrix(A)
    b = np.matrix(b)
    codomain_dim = np.shape(A)[0]
    domain_dim = np.shape(A)[1]
    if codomain_dim == np.shape(b)[1]: pass
    else: return None

    b_rref = rref(np.append(A,b.transpose(), axis=1))[:,-1]
    rank = np.linalg.matrix_rank(A)

    if rank == codomain_dim: return True
    elif np.array(b_rref[rank:]).tolist() == [0]*(codomain_dim - rank):
     return True
    else: return False


##......Network class

class Network():
def __init__(self, vertices = (), edges = []):
self.vertices = vertices
e1 = [edge for edge in edges if vertices.index(edge[0]) <= vertices.index(edge[1])]
e2 = [edge[::-1] for edge in edges if vertices.index(edge[0]) > vertices.index(edge[1])]
self.edges = e1 + e2

def adjacency_matrix(self):
V = self.vertices
n = len(self.vertices)
A = np.zeros((n,n))
for i in range(n):
for j in range(n):
if (V[i],V[j]) in self.edges or (V[j],V[i]) in self.edges:
A[i,j] = 1
else:
A[i,j] = 0
return A


def remove_vertices(self, vertex_subset):
V = [v for v in self.vertices if v not in vertex_subset]
E = [e for e in self.edges if set(vertex_subset).intersection(set(e)) == set()]
return Network(V,E)

def delta_0(self):
return np.zeros(len(self.vertices))

def delta_1(self):
m = len(self.vertices)
n = len(self.edges)
A = np.zeros((m,n))
for i in range(m):
for j in range(n):
for k in [0,1]:
if self.vertices[i] == self.edges[j][k]:
A[i,j] = (-1)**k
else: pass
return np.matrix(A)


def betti_0(self):
if len(self.edges) == 0: return len(self.vertices)
else:
return len(self.vertices) - np.linalg.matrix_rank(self.delta_1())

def betti_1(self):
```

```
if len(self.edges) == 0: return 0
else:
return len(self.edges) - np.linalg.matrix_rank(self.delta_1())

def is_connected(self, vertex1, vertex2):
if len(self.edges) == 0: return False
b = [0]*len(self.vertices)
V = self.vertices
path = [vertex1, vertex2]
if V.index(vertex1) <= V.index(vertex2): pass
else: path = path[::-1]

b[V.index(path[0])] = 1
b[V.index(path[1])] = -1

return is_in_range(self.delta_1(), b)

#def connected_component(self, vertex):
# if self.vertices == (): return ()
# connected_component = [vertex] + [v for v in self.vertices if self.is_connected(vertex,v)]
# return tuple(connected_component)

def connected_component(self,vertex):
if self.vertices == (): return ()
n = len(self.vertices)
index = self.vertices.index(vertex)
A = self.adjacency_matrix()
B = sum([np.linalg.matrix_power(A,i) for i in range(100)])
connected_indices = [j for j in range(n) if B[index,j] != 0]
return tuple([self.vertices[k] for k in connected_indices])

def components(self):
if len(self.vertices) == 0: return []
v = self.vertices[0]
component_v = self.connected_component(v)
X = self.remove_vertices(component_v)
return [component_v] + X.components()

def draw(self):
V = self.vertices
E = self.edges

nodes = [{"id": v, "label": v} for v in V]
links = [{"source": V.index(link[0]), "target": V.index(link[1]), "value": 1} for link in E]

viz = {"nodes":nodes, "links": links} #"paths": paths}

viz_json = json.dumps(viz)
file = open("network.json", 'w')
file.write(viz_json)
file.close()
```

```
## ........Point Cloud Class


class PointCloud():
def __init__(self, points = [], metric = {}):
self.points = points
self.pairs = [(x,y) for x in self.points for y in self.points
if self.points.index(x) < self.points.index(y)]
self.metric = {e:metric[e] for e in self.pairs}

def dist_matrix(self):
V = self.points
n = len(self.points)
A = np.zeros((n,n))
for i in range(n):
for j in range(n):
if i<j:
A[i,j] = self.metric[(V[i],V[j])]
A[j,i] = self.metric[(V[i],V[j])]
elif i == j:
A[i,j] = 0
else: pass
return A

def l2_inner(self):
n = len(self.points)
A = self.dist_matrix()
M = np.zeros((n,n))
for i in range(n):
for j in range(n):
M[i,j] = 0.5*(A[i,0]**2 + A[j,0]**2 - A[i,j]**2)
return np.matrix(M)

def l2_embed(self):
n = len(self.points)
M = self.l2_inner()
eig, U = np.linalg.eigh(M)
eig = np.absolute(eig)
D = np.zeros((n,n))
for i in range(n):
D[i,i] = eig[i]
S = np.sqrt(D)
V = np.dot(U,S)
return V


def sub_pc(self, points_subset):
points = points_subset
metric = {e:self.metric[e] for e in list(self.metric.keys())
if e[0] in points_subset and e[1] in points_subset}
return PointCloud(points, metric)
```

```
def network(self, scale):
V = self.points
E = [e for e in self.pairs if self.metric[e] < scale]
return Network(V,E)

def PH_0(self, steps = 100):
if self.points == []: return 0, 0
if len(self.metric) == 0: return 1,0
#n = min(list(self.metric.values()))
n = 0
m = max(list(self.metric.values()))
l = []
scales = np.arange(n,m,(m - n)/steps).tolist()
for scale in scales:
l.append(self.network(scale).betti_0())
ph0_scale = scales[l.index(mode(l))]
return mode(l), ph0_scale

def PH_1(self, steps = 100):
#n = min(list(self.metric.values()))
n = 0
m = max(list(self.metric.values()))
l = []
scales = np.arange(n,m,(m-n)/steps).tolist()
for scale in scales:
l.append(self.network(scale).betti_1())
ph1_scale = scales[l.index(mode(l))]
return mode(l), ph1_scale



def ph0_network(self, steps = 10):
return self.network(scale = self.PH_0(steps)[1])

def ph0_components(self, steps = 10):
return self.ph0_network(steps).components()

def rbfkernel(self):
A = np.exp(-np.square(self.dist_matrix()))
values = [np.sum(A[i,:]) for i in range(len(self.points))]
return values

def rbf_covering(self, rcover):
values = self.rbfkernel()

a,b = np.amin(values), np.amax(values)
N, overlap = rcover
eps = ((b - a)/N)*overlap
rcover = [a + (b - a)*i/N for i in range(N)] + [b]
print(rcover, eps)
covering = []
```

```
for i in range(N):
preimage = [x for x in self.points if values[self.points.index(x)]> rcover[i] - eps and
values[self.points.index(x)] < rcover[i+1] + eps]
print(preimage)
preimage_subpc = self.sub_pc(preimage)
print(preimage_subpc.PH_0())
components = preimage_subpc.ph0_components()
print(components)
for component in components:
covering.append(component)
return covering, N

def rbf_mapper(self, rcover = [5,0.3]):
V = self.rbf_covering(rcover)[0]
values = self.rbfkernel()

pairs = [(x,y) for x in V for y in V if V.index(x) < V.index(y)]
#print([len(v) for v in V])

E = [(x,y) for (x,y) in pairs if set(x).intersection(set(y)) != set()]
#return Network(V,E)

max_value = max(values)
max_weight = max([len(v) for v in V])
nodes = [{"id": v, "label": v, "weight": len(v), "rbf_value":
 math.floor(np.average([values[self.points.index(x)] for x in v])) } for v in V]

#print(nodes)

links = [{"source": V.index(link[0]), "target": V.index(link[1]), "value": 1} for link in E]

#print(links)

viz = {"max_rbfvalue":math.floor(max_value), "max_weight": max_weight, "nodes":nodes, "links": links}

viz_json = json.dumps(viz)
file = open("network.json", 'w')
file.write(viz_json)
file.close()
```

6.2. **Javascript.** Javasript D3 code for visualizing networks.

```
svg = d3.select("#networkd3"),
    width = +svg.attr("width"),
    height = +svg.attr("height");


var simulation = d3.forceSimulation()
  .force('charge', d3.forceManyBody().strength(-5))
  .force('center', d3.forceCenter(width / 2, height / 2))
  .force("link", d3.forceLink())
```

```
d3.json("network.json", function(error,  network){
  if (error) throw error;

  //Force Link Network//

  scaleRadius = d3.scaleLinear()
      .domain([0, network.max_weight])
      .range([20,200]);

  scaleColor = d3.scaleLinear()
      .domain([0, network.max_rbfvalue])
      .range([0,1]);

  simulation
      .nodes(network.nodes)
      .on("tick", ticked);

  simulation.force("link")
      .links(network.links);

 var line = d3.line()
  .x(function(d) { return network.nodes[d.node].x; })
  .y(function(d) { return network.nodes[d.node].y; })
  .curve(d3.curveLinear);


  function updateLinks() {
    var u = d3.select('.links')
      .selectAll('line')
      .data(network.links)

    u.enter()
      .append('line')
      .merge(u)
      .attr('x1', function(d) {return d.source.x;})
      .attr('y1', function(d) {return d.source.y;})
      .attr('x2', function(d) {return d.target.x;})
      .attr('y2', function(d) {return d.target.y;})

    u.exit().remove()
  }

  function updateNodes() {
    u = d3.select('.nodes')
      .selectAll('circle')
      .data(network.nodes)

    u.enter()
      .append('circle')
      .attr('stroke', 'black')
      .merge(u)
      .attr('cx', function(d) {return d.x;})
      .attr('cy', function(d) {return d.y;})
      .attr('r', function (d) {return Math.sqrt(scaleRadius(d.weight));})
      .attr('fill', function(d) {return d3.interpolateReds(scaleColor(d.rbf_value))})

      .on('mouseover', function(d) {

          d3.select(this).transition()
              .attr('stroke', 'white');
          d3.select(this)
            .append('svg:title')
            .text( function(d){ return "Ticker symbols: " + d.id; });
          })
      .on('mouseout', function(d) {
           // this.parentElement.appendChild(this);

            d3.select(this).transition()
              .attr('stroke', 'black');
      })
      .call(d3.drag()
          .on("start", dragstarted)
          .on("drag", dragged)
          .on("end", dragended));


    u.exit().remove()
  }

  function ticked() {
    //updatePaths()
    updateLinks()
    updateNodes()
   }
function dragstarted(d) {
  if (!d3.event.active) simulation.alphaTarget(0.3).restart();
  d.fx = d.x;
  d.fy = d.y;
}

function dragged(d) {
  d.fx = d3.event.x;
  d.fy = d3.event.y;
}
function dragended(d) {
  if (!d3.event.active) simulation.alphaTarget(0);
  d.fx = null;
  d.fy = null;
 }

})
```