Building a Minimally Correlated Portfolio with Data Science

The difficulty with modern portfolio theory is that correlations are non-stationary and harbours non-linear effects. On top of that financial data is incredibly sparse and Pearson's correlation only works on time series with equal dimensions. There is accumulating evidence that asset correlation networks, whose nodes are assets and whose edges are the pairwise correlations between the asset's historical returns follows a power-law distribution and show evidence of stationarity. In some sense one can say that these networks are governed by simple scale-free laws. It is known to be hard to forecast financial time series, but maybe the evolution of asset correlation networks is easier to predict if they are driven by simple laws, in which case these laws can be learned by a machine learning algorithm.

In the case of asset correlation networks, we are interested in how volatility spreads between assets between assets and how we can use these insights to optimize our portfolio. To do this we can look at the concept of communicability of complex networks to investigate how things spread and which nodes have the greatest influence in the process. Overall, we will use insights from network science to build centrality-based risk model to generate portfolio asset weights.

Summary

Using insights from Network Science, we build a centrality-based risk model for generating portfolio asset weights. The model is trained with the daily prices of 31 stocks from 2006-2014 and validated in years 2015, 2016, and 2017. As a benchmark, we compare the model with a portolfio constructed with Modern Portfolio Theory (MPT). Our proposed asset allocation algorithm significantly outperformed both the DIJIA and S&P500 indexes in every validation year with an average annual return rate of 38.7%, a 18.85% annual volatility, a 1.95 Sharpe ratio, a -12.22% maximum drawdown, a return over maximum drawdown of 9.75, and a growth-risk-ratio of 4.32. In comparison, the MPT portfolio had a 9.64% average annual return rate, a 16.4% annual standard deviation, a Sharpe ratio of 0.47, a maximum drawdown of -20.32%, a return over maximum drawdown of 1.5, and a growth-risk-ratio of 0.69.

Background

In this series we play the part of an Investment Data Scientist at Bridgewater Associates performing a go/no go analysis on a new idea for risk-weighted asset allocation. Our aim is to develop a network-based model for generating asset weights such that the probability of losing money in any given year is minimized. We've heard down the grapevine that all go-descisions will be presented to Dalio's inner circle at the end of the week and will likely be subject to intense scrutiny. As such, we work with a few highly correlated assets with strict go/no go criteria. We build the model using the daily prices of each stock (with a few replacements*) in the Dow Jones Industrial Average (DJIA). If our recommended portfolio

either (1) loses money in **any** year, (2) does not outperform the market **every** year, or (3) does not outperform the MPT portfolio---the decision is no go.

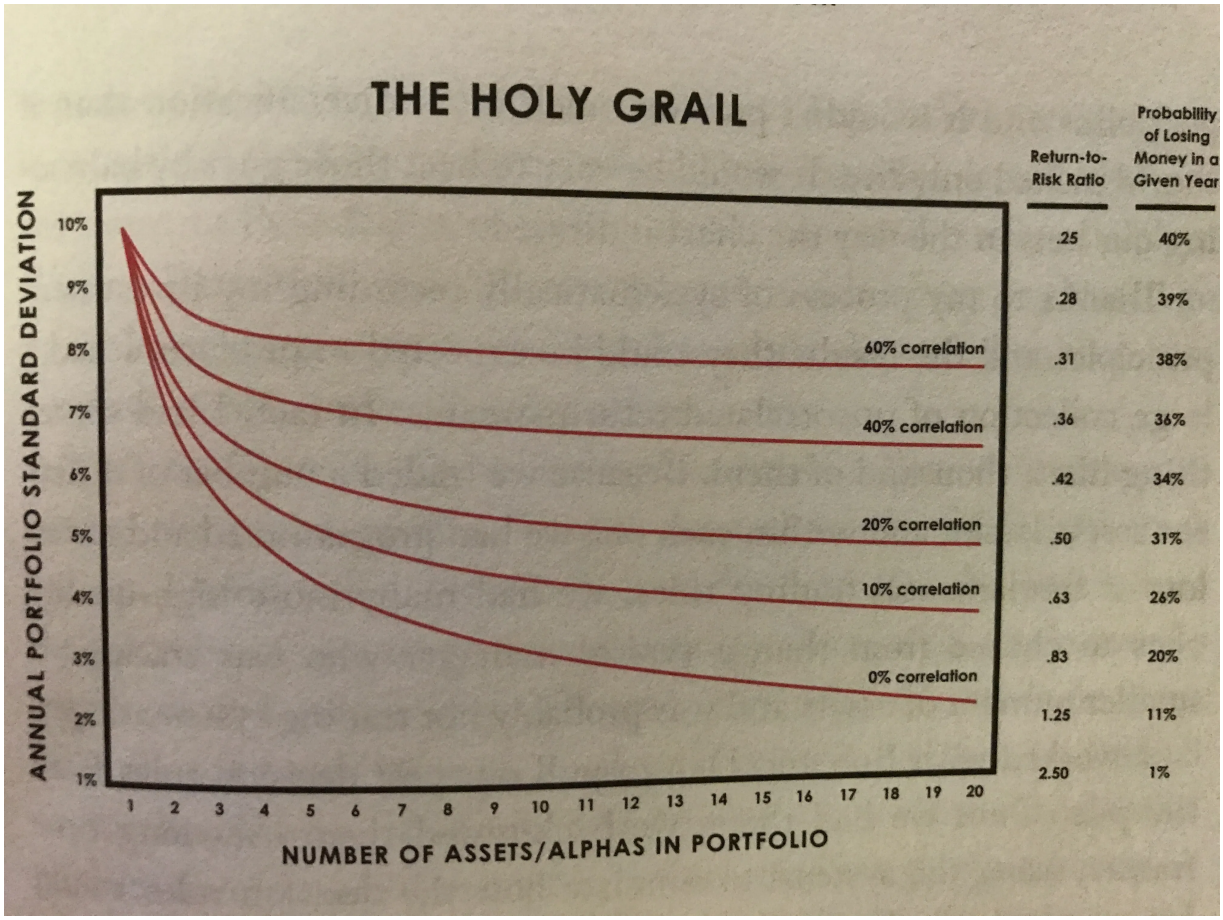
- We replaced Visa (V), DowDuPont (DWDP), and Walgreens (WBA) with three alpha generators: Google (GOOGL), Amazon (AMZN), and Altaba (AABA) and, for the sake of model building, one poor performing stock: General Electric (GE). The dataset is found on Kaggle.

Asset Diversification and Allocation

The building blocks of a portfolio are assets (resources with economic value expected to increase over time). Each asset belongs to one of seven primary asset classes: cash, equitiy, fixed income, commodities, real-estate, alternative assets, and more recently, digital (such as cryptocurrency and blockchain). Within each class are different asset types. For example: stocks, index funds, and equity mutual funds all belong to the equity class while gold, oil, and corn belong to the commodities class. An emerging consensus in the financial sector is this: a portfolio containing assets of many classes and types hedges against potential losses by increasing the number of revenue streams. In general the more diverse the portfolio the less likely it is to lose money. Take stocks for example. A diversified stock portfolio contains positions in multiple sectors. We call this *asset diversification*, or more simply *diversification*. Below is a table summarizing the asset classes and some of their respective types.

Cash	**Equity**	**Fixed Income**	**Commodities**	**Real-Estate**	**Alternativ Assets**
US Dollar	US Stocks	US Bonds	Gold	REIT's	Structurec Credit
Japenese Yen	Foreign Stocks	Foreign Bonds	Oil	Commerical Properties	Liquidatio
Chinese Yaun	Index Funds	Deposits	Wheat	Land	Aviation Assets
UK Pound	Mutual Funds	Debentures	Corn	Industrial Properties	Collectabl
•	•	•	•	•	•
•	•	•	•	•	•

An investor solves the following (asset allocation) problem: given X dollars and N assets find the best possible way of breaking X into N pieces. By "best possible" we mean maximizing our returns subject to minimizing the risk of our initial investment. In other words, we aim to consistently grow X irrespective of the overall state of the market. In what follows, we explore provocative insights by Ray Dalio and others on portfolio construction.



Source: Principles by Ray Dalio (Summary)

The above chart depicts the behaviour of a portfolio with increasing diversification. Along the x-axis is the number of asset types. Along the y-axis is how "spread out" the annual returns are. A lower annual standard deviation indicates smaller fluctuations in each revenue stream, and in turn a diminished risk exposure. The "Holy Grail" so to speak, is to (1) find the largest number of assets that are the **least** correlated and (2) allocate X dollars to those assets such that the probability of losing money any given year is minimized. The underlying principle is this: the portfolio most robust against large market fluctuations and economic downturns is a portfolio with assets that are the **most independent** of eachother.

[86]
```
#import data manipulation (pandas) and numerical manipulation
(numpy) modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sqlalchemy import create_engine, Table, Column, MetaData
from sqlalchemy.dialects.mysql import INTEGER, DOUBLE, VARCHAR,
DATETIME
%matplotlib inline

daily_columns = ['data_pregao', 'preco_abertura as Open',
'preco_min as Low', 'preco_max as High', 'preco_ultimo as
Close', 'vol_tot as Volume']

#silence warnings
```

```
import warnings
warnings.filterwarnings("ignore")
```

[96]
```
# Get the data
df = pd.read_csv(r"data/20130102_20200529_daily.csv",
index_col=0)
df.head()
```

	Ticker	Open	Low	High	Close	Volume	
Day							
2013-01-02	ABCB4	14.00	14.00	14.27	14.15	5224632.0	ABC
2013-01-02	ALPA4	15.10	14.98	15.30	15.16	2719722.0	ALPA
2013-01-02	AMAR3	32.55	32.54	33.01	32.63	7420976.0	LOJA MAR
2013-							

[90]
```
# Reads the dcor matrix
H_close = pd.read_csv(r"data/close_prices_dcor.csv")

#prints first 5 rows of the DataFrame
H_close.head()
```

	Unnamed: 0	ABCB4	ALPA4	AMAR3	BBAS3	BBDC4
0	ABCB4	1.000000	0.258812	0.242263	0.464469	0.483814
1	ALPA4	0.258812	1.000000	0.251698	0.292690	0.298628
2	AMAR3	0.242263	0.251698	1.000000	0.294654	0.292318
3	BBAS3	0.464469	0.292690	0.294654	1.000000	0.701440
4	BBDC4	0.483814	0.298628	0.292318	0.701440	1.000000

5 rows × 81 columns

[108]
```
stocks = open(r"data/selected_tickers.txt").read().split(",")
np.array(stocks)
```

```
array(['ABCB4', 'BBAS3', 'BBDC4', 'BRAP4', 'BRML3', 'BRPR3', 'BTOW3',
       'CCRO3', 'CMIG4', 'CPFE3', 'CSAN3', 'CSMG3', 'CSNA3', 'CYRE3',
       'DIRR3', 'DTEX3', 'ECOR3', 'ELET3', 'ENBR3', 'EQTL3', 'EVEN3',
       'EZTC3', 'GGBR4', 'GOAU4', 'GOLL4', 'HBOR3', 'HGTX3', 'HYPE3',
       'IGTA3', 'ITSA4', 'ITUB4', 'LAME4', 'LIGT3', 'LREN3', 'MRVE3',
       'MULT3', 'MYPK3', 'PETR4', 'POMO4', 'RAPT4', 'RENT3', 'SBSP3',
       'TCSA3', 'TIMP3', 'UGPA3', 'VALE3', 'VIVT4'], dtype='<U5')
```

[109]
```
# Filter data with the selected stocks
df = df.loc[df.Ticker.apply(lambda x: x in stocks)]
```

[110]
```
# Set the ranges for training and testing
TRAIN_RANGE = ('2013-01-02', '2018-12-28')
TEST_RANGE = ('2019-01-01', '2020-05-29')
df_train = df.loc[TRAIN_RANGE[0]:TRAIN_RANGE[1]]
df_train.tail()
```

	Ticker	Open	Low	High	Close	Volume	
Day							
2018-12-28	TCSA3	1.38	1.37	1.45	1.45	3678758.0	TECI
2018-12-28	TIMP3	11.78	11.68	11.93	11.85	42419784.0	TIM S/A
2018-12-28	UGPA3	52.07	52.07	53.48	53.20	92796710.0	ULTF
2018-							

[111]
```
#testing dataset
df_validate = df.loc[TEST_RANGE[0]:TEST_RANGE[1]]
df_validate.tail()
```

	Ticker	Open	Low	High	Close	Volume	
Day							
2020-05-29	TCSA3	0.74	0.72	0.75	0.75	6.657817e+06	TEC
2020-05-29	TIMP3	13.46	13.16	13.62	13.62	1.465600e+08	TIM S/A

	Ticker	Open	Low	High	Close	Volume	

It's always a good idea to check we didn't lose any data after the split.

[112]
```
#returns True if no data was lost after the split and False
otherwise.
df_train.shape[0] + df_validate.shape[0] == df.shape[0]
```

```
True
```

[113]
```
# sets each column as a stock and every row as a daily closing
price
df_validate = df_validate.pivot(columns='Ticker',
values='Close')
```

[114]
```
df_validate.head()
```

Ticker	ABCB4	BBAS3	BBDC4	BRAP4	BRML3	BRPR3	BT
Day							
2019-01-02	17.12	48.60	40.39	31.09	13.46	8.20	43.
2019-01-03	17.10	48.80	40.78	29.90	13.55	8.20	43.
2019-01-04	16.95	48.80	40.46	31.92	13.29	8.20	42.
2019-							

[115]
```
#creates a DataFrame for each time-series (see In [11])
df_train_close = df_train.pivot(columns='Ticker',
values='Close')

#makes a copy of the traning dataset
df_train_close_copy = df_train_close.copy()

df_train_close.head()
```

Ticker	ABCB4	BBAS3	BBDC4	BRAP4	BRML3	BRPR3	BT
Day							
2013-01-02	14.15	25.80	36.02	34.23	27.75	25.80	16.
2013-01-03	14.19	26.31	38.12	33.87	27.81	25.55	16.
2013-01-04	13.99	26.00	37.45	33.20	27.65	25.79	15.

[163]
```
df_train_close.isnull().sum()
```

```
Ticker
ABCB4    0
BBAS3    0
BBDC4    0
BRAP4    0
BRML3    0
BRPR3    0
BTOW3    0
CCRO3    0
CMIG4    0
CPFE3    0
CSAN3    0
CSMG3    0
CSNA3    0
CYRE3    0
```

[154]
```
#idx =
df_train_close.loc[df_train_close["RAPT4"].isnull()].index
#df_train_close.loc[idx].
df_train_close["RAPT4"].fillna(method="ffill", inplace=True)
```

[116]
```
#imports the dcor module to calculate distance correlation
import dcor

#function to compute the distance correlation (dcor) matrix
from a DataFrame and output a DataFrame
#of dcor values.
def df_distance_correlation(df_train):

    #initializes an empty DataFrame
    df_train_dcor = pd.DataFrame(index=stocks, columns=stocks)

    #initialzes a counter at zero
```

```
    k=0

    #iterates over the time series of eachstocks stock
    for i in stocks:

        #stores the ith time series as a vector
        v_i = df_train.loc[:, i].values

        #iterates over the time series of each stock subect to
the counter k
        for j in stocks[k:]:

            #stores the jth time series as a vector
            v_j = df_train.loc[:, j].values

            #computes the dcor coefficient between the ith and
jth vectors
            dcor_val = dcor.distance_correlation(v_i, v_j)

            #appends the dcor value at every ij entry of the
empty DataFrame
            df_train_dcor.at[i,j] = dcor_val

            #appends the dcor value at every ji entry of the
empty DataFrame
            df_train_dcor.at[j,i] = dcor_val

        #increments counter by 1
        k+=1

    #returns a DataFrame of dcor values for every pair of
stocks
    return df_train_dcor
```

[164]
```
df_train_dcor = df_distance_correlation(df_train_close)
df_train_dcor.head()
```

	ABCB4	BBAS3	BBDC4	BRAP4	BRML3	BR
ABCB4	1	0.848222	0.597899	0.677561	0.370743	0.300
BBAS3	0.848222	1	0.685005	0.713615	0.351099	0.322
BBDC4	0.597899	0.685005	1	0.544708	0.373804	0.241
BRAP4	0.677561	0.713615	0.544708	1	0.372271	0.410
BRML3	0.370743	0.351099	0.373804	0.372271	1	0.865

5 rows × 47 columns

Building a Time-Series Correlation Network with Networkx

[118]
```
#imports the NetworkX module
import networkx as nx

# takes in a pre-processed dataframe and returns a time-series
correlation
# network with pairwise distance correlation values as the
edges
def build_corr_nx(df_train, corr_threshold=0.325):

    # converts the distance correlation dataframe to a numpy
matrix with dtype float
    cor_matrix = df_train.values.astype('float')

    # Since dcor ranges between 0 and 1, (0 corresponding to
independence and 1
    # corresponding to dependence), 1 - cor_matrix results in
values closer to 0
    # indicating a higher degree of dependence where values
close to 1 indicate a lower degree of
    # dependence. This will result in a network with nodes in
close proximity reflecting the similarity
    # of their respective time-series and vice versa.
    sim_matrix = 1 - cor_matrix

    # transforms the similarity matrix into a graph
    G = nx.from_numpy_matrix(sim_matrix)

    # extracts the indices (i.e., the stock names from the
dataframe)
    stock_names = df_train.index.values

    # relabels the nodes of the network with the stock names
    G = nx.relabel_nodes(G, lambda x: stock_names[x])

    # assigns the edges of the network weights (i.e., the dcor
values)
    G.edges(data=True)

    # copies G
    ## we need this to delete edges or othwerwise modify G
    H = G.copy()

    # iterates over the edges of H (the u-v pairs) and the
weights (wt)
```

```
        for (u, v, wt) in G.edges.data('weight'):
            # selects edges with dcor values less than or equal to
0.33
            if wt >= 1 - corr_threshold:
                # removes the edges
                H.remove_edge(u, v)

            # selects self-edges
            if u == v:
                # removes the self-edges
                H.remove_edge(u, v)

        # returns the final stock correlation network
        return H
```

```
[165]  # builds the distance correlation networks for the training
       data
       H_close = build_corr_nx(df_train_dcor)
```

Plotting a Time-Series Correlation Network with Seaborn

```
[157]  # function to display the network from the distance correlation
       matrix
       def plt_corr_nx(H, title):

           # creates a set of tuples: the edges of G and their
       corresponding weights
           edges, weights = zip(*nx.get_edge_attributes(H,
       "weight").items())

           # This draws the network with the Kamada-Kawai path-length
       cost-function.
           # Nodes are positioned by treating the network as a
       physical ball-and-spring system. The locations
           # of the nodes are such that the total energy of the system
       is minimized.
           pos = nx.kamada_kawai_layout(H)

           with sns.axes_style('whitegrid'):
               # figure size and style
               plt.figure(figsize=(16, 9))
               plt.title(title, size=16)

               # computes the degree (number of connections) of each
       node
               deg = H.degree
```

```
        # list of node names
        nodelist = []
        # list of node sizes
        node_sizes = []

        # iterates over deg and appends the node names and
degrees
        for n, d in deg:
            nodelist.append(n)
            node_sizes.append(d)

        # draw nodes
        nx.draw_networkx_nodes(
            H,
            pos,
            node_color= "blue", #"#DA70D6",
            nodelist=nodelist,
            node_size= [(x+1) * 20 for x in node_sizes],
#np.power(node_sizes, 2.33),
            alpha=0.8,
            font_weight="bold",
        )

        # node label styles
        nx.draw_networkx_labels(H, pos, font_size=13,
font_family="sans-serif", font_weight='bold')

        # color map
        cmap = sns.cubehelix_palette(10, start=3.0, dark=0.1,
light=0.7, as_cmap=True, reverse=True)

        # draw edges
        nx.draw_networkx_edges(
            H,
            pos,
            edge_list=edges,
            style="solid",
            edge_color=weights,
            edge_cmap=cmap,
            edge_vmin=min(weights),
            edge_vmax=max(weights),
        )

        # builds a colorbar
        sm = plt.cm.ScalarMappable(
            cmap=cmap,
            norm=plt.Normalize(vmin=min(weights),
            vmax=max(weights))
        )
        sm._A = []
        plt.colorbar(sm)
```

```
        # displays network without axes
        plt.axis("off")


# function to visualize the degree distribution
def hist_plot(network, title, bins, xticks):

    # extracts the degrees of each vertex and stores them as a
list
    deg_list = list(dict(network.degree).values())

    # sets local style
    with plt.style.context('fivethirtyeight'):
        # initializes a figure
        plt.figure(figsize=(9,6))

        # plots a pretty degree histogram with a kernel density
estimator
        sns.distplot(
            deg_list,
            kde=True,
            bins = bins,
            color='darksalmon',
            hist_kws={'alpha': 0.7}

        );

        # turns the grid off
        plt.grid(False)

        # controls the number and spacing of xticks and yticks
        #xticks = range()
        plt.xticks(xticks, size=11)
        plt.yticks(size=11)

        # removes the figure spines
        sns.despine(left=True, right=True, bottom=True,
top=True)

        # labels the y and x axis
        plt.ylabel("Probability", size=15)
        plt.xlabel("Number of Connections", size=15)

        # sets the title
        plt.title(title, size=20);

        # draws a vertical line where the mean is
        plt.axvline(sum(deg_list)/len(deg_list),
                    color='darkorchid',
                    linewidth=3,
                    linestyle='--',
```

```
                        label='Mean =
{:2.0f}'.format(sum(deg_list)/len(deg_list))
                )

                # turns the legend on
                plt.legend(loc=0, fontsize=12)
```

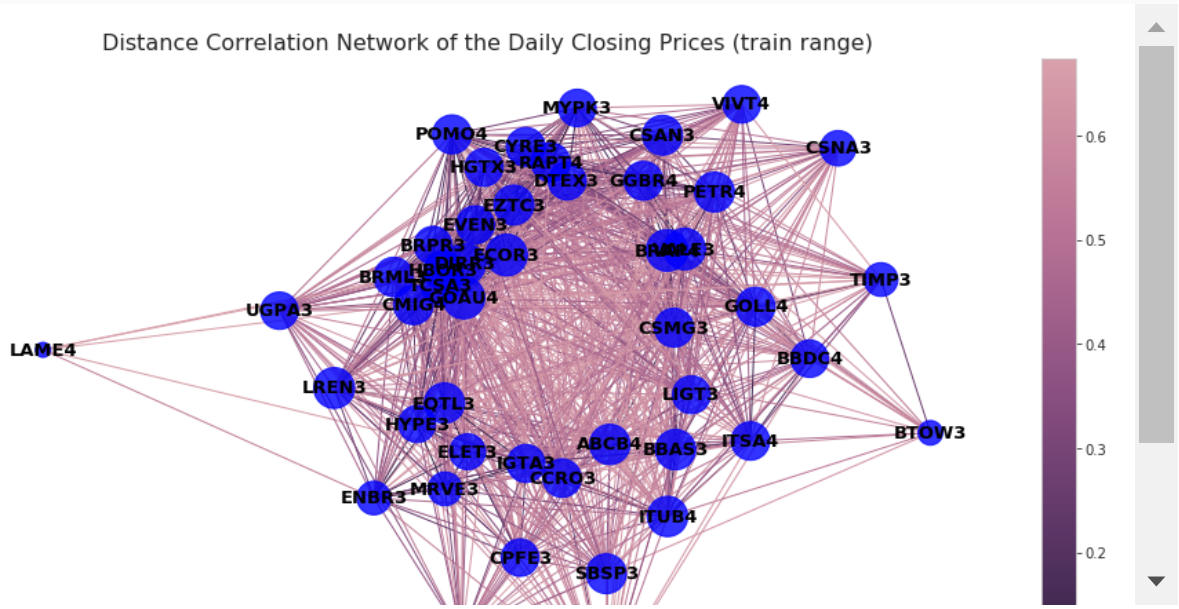
[166]
```
def is_irreducible(H):
    for node, weight in H.degree():
        if weight == 0:
            return False
    return True


def grid_search_threshold(df_dcor, threshold_list):
    for threshold in threshold_list:
        print("Testing for threshold
{:,.4f}:".format(threshold))
        H = build_corr_nx(df_dcor, corr_threshold=threshold)
        print("Result: {}".format("Irreducible!" if
is_irreducible(H) else "Not irreducible!"))
        print()


threshold_list = [0.0, 0.1, 0.15, 0.2, 0.25, 0.3, 0.325, 0.4,
0.45]
print("Testing for Close price: \n")
grid_search_threshold(df_train_dcor, threshold_list)
```

```
Testing for Close price:

Testing for threshold 0.0000:
Result: Irreducible!

Testing for threshold 0.1000:
Result: Irreducible!

Testing for threshold 0.1500:
Result: Irreducible!

Testing for threshold 0.2000:
Result: Irreducible!

Testing for threshold 0.2500:
```

Visualizing How A Portfolio is Correlated with Itself (with Physics)

The following visualizations are rendered with the Kamada-Kawai method, which treats each vertex of the graph as a mass and each edge as a spring. The graph is drawn by finding the list of vertex positions that minimize the total energy of the ball-spring system. The method treats the spring lengths as the weights of the graph, which is given by `1 - cor_matrix` where `cor_matrix` is the distance correlation matrix. Nodes seperated by large distances reflect smaller correlations between their time series data, while nodes seperated by small distances reflect larger correlations. The minimum energy configuration consists of vertices with few connections experiencing a repulsive force and vertices with many connections feeling an attractive force. As such, nodes with a larger degree (more correlations) fall towards to the center of the visualization where nodes with a smaller degree (fewer correlations) are pushed outwards. For an overview of physics-based graph visualizations see the Force-directed graph drawing wiki.

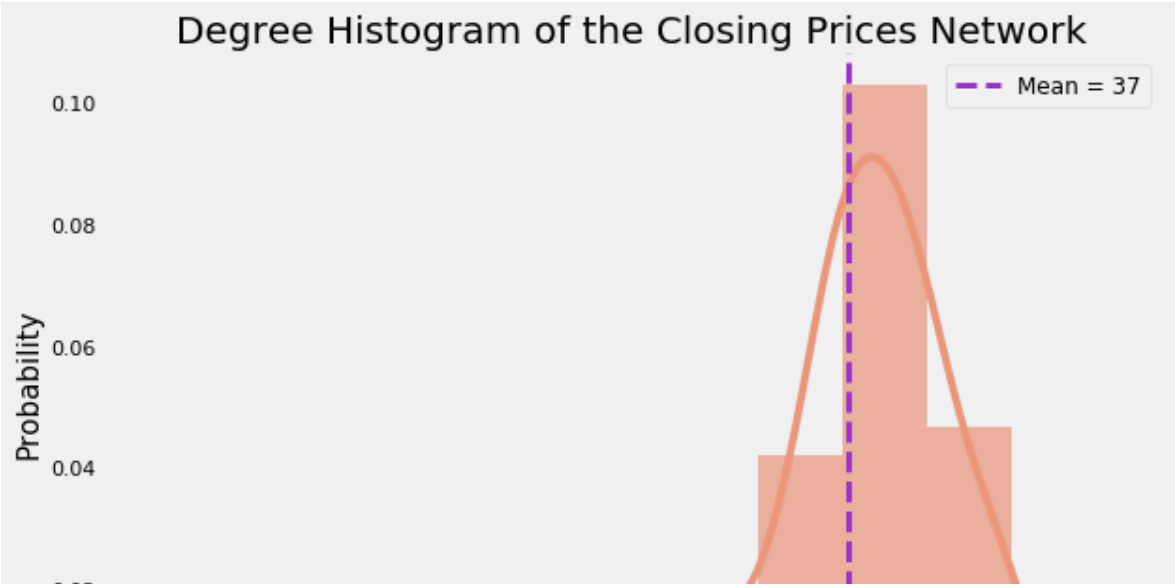
[167]
```
# plots the distance correlation network of the daily opening
prices from 2006-2014
plt_corr_nx(H_close, title='Distance Correlation Network of the
Daily Closing Prices (train range)')
```



In the above visualization, the sizes of the vertices are proportional to the number of connections they have. The colorbar to the right indicates the degree of disimilarity (the distance) between the stocks. The larger the value (the lighter the color) the less similar the stocks are. Keeping this in mind, several stocks jump out. **JBS**, **CSMG**, **HBOR3**, and all the ones that lie on the periphery of the network with the fewest number of correlations above $\rho_c = 0.325$. On the other hand **BBAS3**, **ITUB4**, **BBDC4**, and **LAME4** sit in the core of the network with the greatest number connections above $\rho_c = 0.325$. It is clear from the closing prices network that our asset allocation algorithm needs to reward vertices on the periphery and punish those nearing the center. In the next code block we build a function to visualize how the edges of the distance correlation network are distributed.

Degree Histogram

[168]
```
# plots the degree histogram of the closing prices network
hist_plot(
    H_close,
    'Degree Histogram of the Closing Prices Network',
    bins=9,
    xticks=range(0, 51, 5)
)
```



Communicability as a Measure of Relative Risk

We are now in a position to devise a method to compute the allocation weights of our portfolio. To recall, this is the problem:

Given the N assets in our portfolio, find a way of computing the allocation weights w_i, $\left(\sum_{i=1}^{N} w_i = 1\right)$ such that assets more correlated with each other obtain lower weights while those less correlated with each other obtain higher weights.

Theres an infinite number of possible solutions to the above problem. The asset correlation network we built contains information on how our portfolio is interrelated (whose connected to who), but it does not tell us how each asset *impacts* the other or how those impacts travel throughout the network. If, for example, Apple's stock lost 40% of its value wiping out, say, two years of gains, how would this impact the remaining assets in our portfolio? How easily does this kind of behaviour spread and how can we keep our capital isolated from it? We thus seek a measure of "relative risk" that quantifies not only the correlations between assets, but how those correlations mediate perturbations in the portfolio. Our aim, therefore, is twofold: allocate capital inversely proportional to (1) the correlations between assets and (2) proportional to the "impact resistence" of each asset. As luck would have it, there is a centrality measure that does just this! Let us define the relative risk as follows:

$$\text{Relative Risk of Asset} \ r = \dfrac{\omega_{r}}{\sum_{r'=1}^{N}\omega_{r'}},$$

where

$$\omega_{r} = \dfrac{1}{C}\sum_{p}\sum_{q}\dfrac{G_{prq}}{G_{pq}}$$

is the **Communicability Betweenness centrality** (Estrada, *et al.* (2009)) of node r. Here

$$G_{prq} = \big(\exp{\textbf{A}}\big)_{pq} - \big(\exp{(\textbf{A} - \textbf{E}(r))}\big)_{pq}$$

is the number of weighted walks involving only node r,

$$G_{pq} = \big(\exp{\textbf{A}}\big)_{pq}$$

is the so-called *communicability* between nodes p and q,

$$A_{pq} = \begin{cases} 1, & \text{if} \ \rho \geq \rho_{c} \\ 0, & \text{otherwise} \end{cases}$$

is the adjacency matrix induced by the distance correlation matrix Cor_{ij}, and $\textbf{E}(r)$ is a matrix such that when added to $\textbf{A}$, yields a new graph $G(r) = (V, E')$ with all edges connecting $r \in V$ removed. The constant $C = (n-1)^2 - (n-1)$ normalizes ω_{r} such that it takes values between 0 and 1. We can better understand what ω_{r} is counting by re-writing the matrix exponential as a taylor series:

$$\exp{\textbf{A} = \sum_{k=1}^{\infty}}\dfrac{\textbf{A}^{k}}{k!}$$

Rasing the adjacency matrix to the power of k counts all walks from p to q of length k. The matrix exponential therefore counts all possible ways of moving from p to q weighted by the inverse factorial of k. So the denominator of ω_{r} counts all weighted walks involving every node. Put simply,

$$\boxed{\text{Communicability Betweenees centrality} = \dfrac{\text{sum of all weighted walks involving node r}}{\text{sum of all weighted walks involving every node}}}$$

So the communicability betweenness centrality is proportional to the number of connections (correlations) a node has and therefore satisifies the first requirement of relative risk. Next, we explore how this measure quantifies the spread of impacts throughout the network, satisfying our second requirement.

The Physics of what Communicability Measures

Estrada & Hatano (2007) provided an ingenius argument showing the communicability of a network is identical to the Green's function of a network. That is, it measures how impacts (or more generally thermal fluctuations) travel from one node to another. Their argument works by treating each node as an oscillator and each edge as a spring (which is what we did to generate the visualization of our asset correlation network). Intuitively, we can draw an analogy between the movement of an asset's price and its motion in a ball-spring system. In this analogy, volatility is equivalent to how energetic the oscillator is. Revisiting the hypothetical scenerio of Apple losing 40% of its value: we can visualize this in our mind's eye as an impact to one of the masses---causing it to violently oscillate. How does this motion propagate throughout the rest of the ball-spring system? Which masses absorb the blow and which reflect it? Communicability betweeness centrality answers this question by counting all possible ways the impact can reach node r. Higher values indicate

the node has a greater susceptiblility to impacts whereas lower values denote just the opposite.

The Bottom Line

The communicability of a network quantifies how impacts spread from one node to another. In the context of an asset correlation network, communicability measures how volatility travels node to node. **We aim to position our capital such that it's the most resistant to the communicability of volatility**. Recall we seek a portfolio that (1) consistently generates wealth while minimizing potential losess and (2) is robust against large market fluctuations and economic downturns. Of course, generous returns are desired, but not in a way that threatens our initial investment. To this end, the strategy moving forward is this: allocate capital inversely proportional to its relative (or intraportfolio) risk.

Intraportfolio Risk

[169]
```
df_train_dcor.isnull().sum()
```

```
ABCB4    0
BBAS3    0
BBDC4    0
BRAP4    0
BRML3    0
BRPR3    0
BTOW3    0
CCRO3    0
CMIG4    0
CPFE3    0
CSAN3    0
CSMG3    0
CSNA3    0
CYRE3    0
DTPP3    0
```

[171]
```
zero_degree = []
nonzero_degree = []
for t, d in H_close.degree():
    if d == 0:
        zero_degree.append(t)
    else:
        nonzero_degree.append(t)

print(zero_degree)
print(len(zero_degree))
print(nonzero_degree)
print(len(nonzero_degree))
```

```
[]
0
['ABCB4', 'BBAS3', 'BBDC4', 'BRAP4', 'BRML3', 'BRPR3', 'BTOW3',
'CCRO3', 'CMIG4', 'CPFE3', 'CSAN3', 'CSMG3', 'CSNA3', 'CYRE3',
'DIRR3', 'DTEX3', 'ECOR3', 'ELET3', 'ENBR3', 'EQTL3', 'EVEN3',
'EZTC3', 'GGBR4', 'GOAU4', 'GOLL4', 'HBOR3', 'HGTX3', 'HYPE3',
'IGTA3', 'ITSA4', 'ITUB4', 'LAME4', 'LIGT3', 'LREN3', 'MRVE3',
'MULT3', 'MYPK3', 'PETR4', 'POMO4', 'RAPT4', 'RENT3', 'SBSP3',
'TCSA3', 'TIMP3', 'UGPA3', 'VALE3', 'VIVT4']
47
```

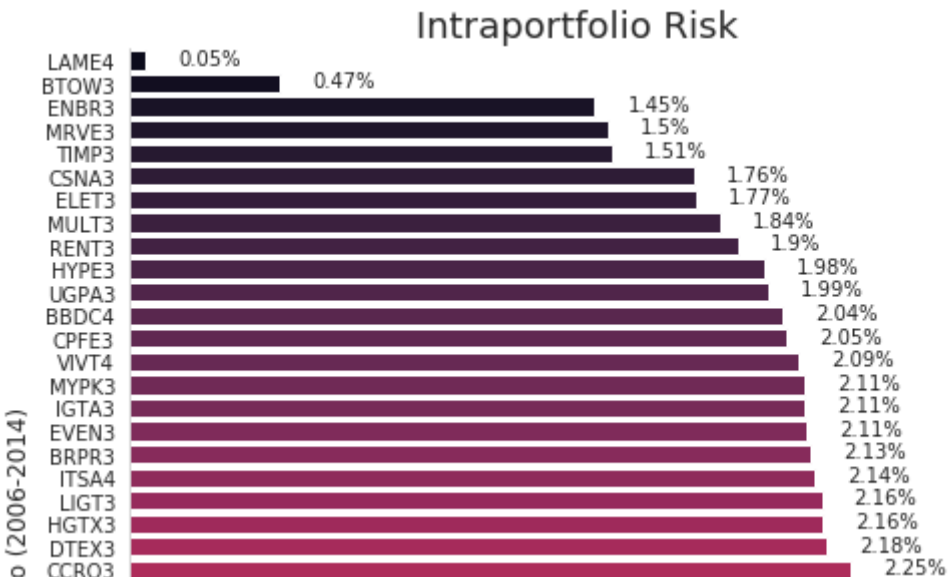
[173]
```
# master_zero_degree = ['ALPA4', 'AMAR3', 'BEEF3', 'EMBR3',
'FLRY3', 'GFSA3', 'GRND3', 'GUAR3', 'JBSS3', 'JHSF3', 'JSLG3',
'KLBN4', 'LOGN3', 'MDIA3', 'MGLU3', 'MILS3', 'MRFG3', 'OIBR3',
'POSI3', 'PSSA3', 'RADL3', 'SLCE3', 'SMTO3', 'TOTS3', 'TRPL4',
'WEGE3']
# master_nonzero_degree = ['ABCB4', 'BBAS3', 'BBDC3', 'BBDC4',
'BRAP4', 'BRFS3', 'BRML3', 'BRPR3', 'BTOW3', 'CCRO3', 'CIEL3',
'CMIG3', 'CMIG4', 'CPFE3', 'CSAN3', 'CSMG3', 'CSNA3', 'CYRE3',
'DIRR3', 'DTEX3', 'ECOR3', 'ELET3', 'ENBR3', 'EQTL3', 'EVEN3',
'EZTC3', 'GGBR4', 'GOAU4', 'GOLL4', 'HBOR3', 'HGTX3', 'HYPE3',
'IGTA3', 'ITSA4', 'ITUB3', 'ITUB4', 'LAME3', 'LAME4', 'LIGT3',
'LREN3', 'MRVE3', 'MULT3', 'MYPK3', 'PETR3', 'PETR4', 'POMO4',
'QUAL3', 'RAPT4', 'RENT3', 'SBSP3', 'TCSA3', 'TIMP3', 'UGPA3',
'VALE3', 'VIVT4']
# print(len(master_zero_degree))
# print(len(master_nonzero_degree))
```

[174]
```
len(zero_degree)
```

```
0
```

[175]
```
# calculates the communicability betweeness centrality and
returns a dictionary
risk_alloc = nx.communicability_betweenness_centrality(H_close)
#risk_alloc = nx.eigenvector_centrality(H_master)
# print(risk_alloc)
# converts the dictionary of degree centralities to a pandas
series
risk_alloc = pd.Series(risk_alloc)

# normalizes the degree centrality
risk_alloc = risk_alloc / risk_alloc.sum()

# resets the index
risk_alloc.reset_index()
```

```
    # converts series to a sorted DataFrame
    risk_alloc = (
        pd.DataFrame({"Stocks": risk_alloc.index, "Risk
    Allocation": risk_alloc.values})
            .sort_values(by="Risk Allocation", ascending=True)
            .reset_index()
            .drop("index", axis=1)
    )

    with sns.axes_style('whitegrid'):
        # initializes figure
        plt.figure(figsize=(8,10))

        # plots a pretty seaborn barplot
        sns.barplot(x='Risk Allocation', y='Stocks',
    data=risk_alloc, palette="rocket")

        # removes spines
        sns.despine(right=True, top=True, bottom=True)

        # turns xticks off
        plt.xticks([])

        # labels the x axis
        plt.xlabel("Relative Risk %", size=12)

        # labels the y axis
        plt.ylabel("Historical Portfolio (2006-2014)", size=12)

        # figure title
        plt.title("Intraportfolio Risk", size=18)

        # iterates over the stocks (label) and their numerical
    index (i)
        for i, label in enumerate(list(risk_alloc.index)):

            # gets the height of each bar in the barplot
            height = risk_alloc.loc[label, 'Risk Allocation']

            # gets the relative risk as a percentage (the labels)
            label = (risk_alloc.loc[label, 'Risk Allocation']*100
                        ).round(2).astype(str) + '%'

            # annotates the barplot with the relative risk
    percentages
            plt.annotate(str(label), (height + 0.001, i + 0.15))
```



[188]
```
df.loc[df.Ticker == "CSAN3"].Name.iloc[0]
```

```
'COSAN'
```

We read an intraportfolio risk plot like this: BTOW3 (*B2W Varejo*) is $\frac{0.47}{0.05} = 9.4$ times riskier than LAME4 (*Lojas Americanas*), ITUB4 (Itaú Unibanco) is $\frac{2.44}{2.04} = 1.20$ times more risky than BBDC4 (Banco Bradesco), ... , and GOAU4 (Gerdau) is $\frac{2.66}{0.05} = 53.2$ times riskier than LAME4! Conversely, EZTC3 (*Eztec*) is $\frac{2.44}{2.43} = 1.004$ times as risky as CSAN3 (*Cosan*), so on and so forth.

Intuitively, the assets that cluster in the center of the network are most susceptible to impacts, whereas those further from the cluster are the least susceptible. The logic from here is straightforward: take the inverse of the relative risk (which we call the "relative certainty") and normalize it such that it adds to 1. These are the asset weights. Formally,

$\text{w}_{r} = \dfrac{1}{\omega_{r} \sum_{r'}\omega_{r'}^{-1}}$

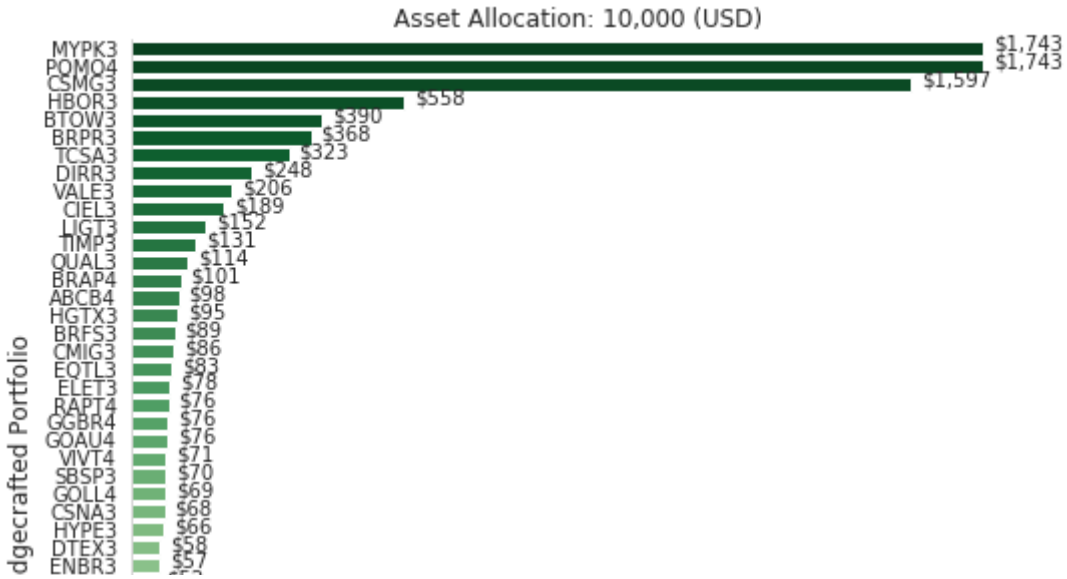
Next, Let's visualize the allocation of 10,000 (USD) in our portfolio.

Communicability-Based Asset Allocation

[189]
```
# calculates degree centrality and assigns it to investmnet_A
# investment_A =
nx.communicability_betweenness_centrality(H_master)
investment_A = nx.eigenvector_centrality(H_master)
```

```
    # calculates the inverse of the above and re-asigns it to
    investment_A as a pandas series
    investment_A = 1 / pd.Series(investment_A)

    # normalizes the above
    investment_A = investment_A / investment_A.sum()

    # resets the index
    investment_A.reset_index()

    # converts the above series to a sorted DataFrame
    investment_A = (
        pd.DataFrame({"Stocks": investment_A.index, "Asset
    Allocation": investment_A.values})
            .sort_values(by="Asset Allocation", ascending=False)
            .reset_index()
            .drop("index", axis=1)
    )

    with sns.axes_style('whitegrid'):
        # initializes a figure
        plt.figure(figsize=(8,9))

        # plot a pretty seaborn barplot
        sns.barplot(x='Asset Allocation', y='Stocks',
    data=investment_A, palette="Greens_r")

        # despines the figure
        sns.despine(right=True, top=True, bottom=True)

        # turns xticks off
        plt.xticks([])

        # turns the x axis label off
        plt.xlabel('')

        # fig title
        plt.title("Asset Allocation: 10,000 (USD)", size=12)

        # y axis label
        plt.ylabel("Historical Hedgecrafted Portfolio", size=12)

        # captial to be allocated
        capital = 10000

        # iterates over the stocks (label) and their numerical
    indices (i)
        for i, label in enumerate(list(investment_A.index)):

            # gets the height of each bar
            height = investment_A.loc[label, 'Asset Allocation']
```

```
            # calculates the capital to be allocated
            label = (investment_A.loc[label, 'Asset Allocation'] *
capital
                     ).round(2)

            # annotes the capital above each bar
            plt.annotate('${:,.0f}'.format(label), (height + 0.002,
i + 0.15))
```



[192]
```
investment_A.iloc[:, 1].cumsum()
```

```
0     0.174312
1     0.348624
2     0.508294
3     0.564050
4     0.603077
5     0.639868
6     0.672160
7     0.696916
8     0.717493
9     0.736378
10    0.751565
11    0.764634
12    0.776060
13    0.786137
14    0.795891
```

[194]
```
investment_A.iloc[:10]
```

	Stocks	Asset Allocation
0	MYPK3	0.174312
1	POMO4	0.174312

	Stocks	Asset Allocation
2	CSMG3	0.159670
3	HBOR3	0.055755
4	BTOW3	0.039027

[195]
```
investment_A.tail()
```

	Stocks	Asset Allocation
50	BBAS3	0.002542
51	ITUB4	0.002513
52	BBDC3	0.002497
53	BBDC4	0.002472
54	ITSA4	0.002455

MYPK3, POMO4, and CSMG3 recieve a little more than 50% of the invested capital, BTOW3 gets about 4%, VALE3 2%, CERL3 1.9%, and the remaining assets recieve less and less of our capital. To the traditional invester, this strategy may appear "risky" since 60% of our investment is with 5 of our 55 assets. While it's true *if* one of them is hit hard we'll lose a substantial amount of money, our algorithm predicts they are the *least* likey to take a hit if and when our other assets get in trouble.

It's worth pointing out that the methods we've used to generate the asset allocation weights differ dramatically from the contemporary methods of MPT and its extensions. The approach taken in this project makes no assumptions of future outcomes of a portfolio, i.e., the algorithm doesn't require us to make a prediction of the expected returns (as MPT does). What's more---we're not solving an optimization problem---there's nothing to be minimized or maximized. Instead, we observe the topology (interrelatedness) of our portfolio, predict which assets are the most susceptible to the communicability of volatile behaviour and allocate capital accordingly.

[196]
```
df_train_close_copy.index.max()
```

```
'2018-12-28'
```

[197]
```
# DataFrame of the prices we buy stock at
df_buy_in =
df_train_close_copy.loc[TRAIN_RANGE[1]].sort_index().to_frame('
```

```
        Buy In: {}'.format(TRAIN_RANGE[1]))
```

Alternative Allocation Strategy: Allocate Capital in the Maximum Independent Set

The maximum independent set (MIS) is the largest set of vertices such that no two are adjacent. Applied to our asset correlation network, the MIS is the greatest number of assets such that every pair has a correlation below ρ_c. The size of the MIS is inversely proportional to the threshold ρ_c. Larger values of ρ_c produce a sparse network (more edges are removed) and therefore the MIS tends to be larger. An optimized portfolio would therefore correspond to maximizing the size of the MIS subject to minimizing ρ_c. The best way to do this is to increase the universe of assets we're willing to invest in. By further diversifying the portfolio with many asset types and classes, we can isolate the largest number of minimally correlated assets and allocate capital inversely proportional to their relative risk. While generating the asset weights remains a non-optimization problem, generating the asset correlation network *becomes* one. We're really solving two sepreate problems: determing how to build the asset correlation network (there are many) and determining which graph invariants (there are many) extract the asset weights from the network. As such, one can easily imagine a vast landscape of portfolios beyond that of MPT and a metric fuck-tonne of wealth to create. **Unfortunately, solving the MIS problem is NP-hard. The best we can do is find an approximation.**

Using Expert Knowledge to Approximate the Maximum Independent Set

We have two options: randomly generate a list of maximal indpendent sets (subgraphs of G such that no two vertices share an edge) and select the largest one, or use expert knowledge to reduce the number of sets to generate and do the latter. Both methods are imperfect, but the former is far more computationally expensive than the latter. Suppose we do fundamentals research and conclude ITUB4 must be in our portfolio. How could we imbue the algorithm with this knowledge? Can we make the algorithm flexible enough for portfolio managers to fine-tune with goold-ole' fashioned research, while at the same time keeping it rigged enough to prevent poor decisions from producing terribe portfolios? We confront this problem in the code block below by extracting an approximate MIS by generating 100 random maximal indpendent sets containing ITUB4.

[198]
```
# a function to generate a random approximate MIS
### WARNING: rerunning kernel will produce different MISs
def generate_mis(G, sample_size, nodes=None):

    """Returns a random approximate maximum independent set.

    Parameters
    ----------
    G: NetworkX graph
```

```
        Undirected graph

    nodes: list, optional
        a list of nodes the approximate maximum independent set
must contain.

    sample_size: int
        number of maximal independent sets sampled from

    Returns
    -------
    max_ind_set: list or None
        list of nodes in the apx-maximum independent set
        NoneType object if any two specified nodes share an
edge

    """

    # list of maximal independent sets
    max_ind_set_list=[]

    # iterates from 0 to the number of samples chosen
    for i in range(sample_size):

        # for each iteration generates a random maximal
independent set that contains
        # UnitedHealth and Amazon
        max_ind_set = nx.maximal_independent_set(G,
nodes=nodes, seed=i)

        # if set is not a duplicate
        if max_ind_set not in max_ind_set_list:

            # appends set to the above list
            max_ind_set_list.append(max_ind_set)

        # otherwise pass duplicate set
        else:
            pass

    # list of the lengths of the maximal independent sets
    mis_len_list=[]

    # iterates over the above list
    for i in max_ind_set_list:

        # appends the lengths of each set to the above list
        mis_len_list.append(len(i))

    # extracts the largest maximal independent set, i.e., the
maximum independent set (MIS)
```

```
        ## Note: this MIS may not be unique as it is possible there
    are many MISs of the same length
        max_ind_set =
    max_ind_set_list[mis_len_list.index(max(mis_len_list))]

        return max_ind_set
```

[233]
```
max_ind_set = generate_mis(H_close, sample_size=100, nodes=
["ITUB4"])
print(max_ind_set)
```

```
['ITUB4', 'UGPA3', 'LAME4']
```

The `generate_mis` function generates a maximal independent set that approximates the true maximum independent set. As an option, the user can pick a list of assets they want in their portfolio and `generate_mis` will return the safest assets to complement the user's choice. Picking UNH and AMZN left us with VZ and MCD. The weights of these assets will remain directly inversely proportional to the communicability betweeness centrality.

[234]
```
# prices of shares to buy for the MIS
df_mis_buy_in = df_buy_in.loc[list(max_ind_set)]
df_mis_buy_in
```

	Buy In: 2018-12-28
Ticker	
ITUB4	35.5
UGPA3	53.2
LAME4	19.7

Backtesting with Modern Portfolio Theory

Now that we have a viable alternative to portfolio optimization, it's time to see how the Hedgecraft portfolio performed in the validation years (15', 16', and 17') with respect to the Markowitz portfolio (i.e., the efficient frontier model) and the overall market. To summarize our workflow thus far we:

1. Preprocessed historical pricing data of 31 stocks for time series analyses.
2. Computed the distance correlation matrix $\rho_D(X_i, X_j)$ for the `Open`, `High`, `Low`, `Close`, and `Close_diff` from 2006-2014.

3. Used the NetworkX module to transform each distance correlation matrix into a weighted graph.
4. Adopted the winner-take-all method by Tse, et al. and removed edges with correlations below a threshold value of $\rho_c = 0.325$.
5. Built a master network by averaging over the edge weights of the `Open`, `High`, `Low`, `Close`, and `Close_diff` networks.
6. Calculated the "relative risk" of each asset as the communicabality betweeness centrality assigned to each node.
7. Generated the asset weights as the normalized inverse of communicability betweeness centrality.

In addition to the above steps, we introduced a human-in-the-middle strategy, giving the user flexible control over the portfolio construction process. This is the extra step we added:

8. Adjust the asset weights for an approximate maximum independent set, either with or without human intervention.

To distinguish bewteen these two approaches we designate steps 1-7 as the *Hedgecraft* algo and steps 1-8 as the *Hedgecraft MIS* algo. Below we observe how these models perform with the Efficient Frontier as a benchmark.

Generating Hedgecraft Portfolio Weights

[235]
```
# calculates communicability betweeness centrality
weights = nx.communicability_betweenness_centrality(H_close)
#weights = nx.eigenvector_centrality(H_master)

# dictionary comprehension of communicability centrality for
the maximum independent set
mis_weights = {key: weights[key] for key in list(max_ind_set)}


# a function to convert centrality scores to portfolio weights
def centrality_to_portfolio_weights(weights):

    """Returns a dictionary of portfolio weights.

    Parameters
    ----------
    weights: dictionary
        NetworkX centrality scores

    Returns
    -------
    portfolio weights: dictionary
        normalized inverse of chosen centrality measure
```

```
    """

    # iterates over the key, value pairs in the weights dict
    for key, value in weights.items():

        # takes the inverse of the communicability betweeness
centrality of each asset
        weights[key] = 1/value

    # normalization parameter for all weights to add to 1
    norm = 1.0 / sum(weights.values())

    # iterates over the keys (stocks) in the weights dict
    for key in weights:

        # updates each key value to the normalized value and
rounds to 3 decimal places
        weights[key] = round(weights[key] * norm, 3)

    return weights



print(centrality_to_portfolio_weights(weights))
print('\n')
print(centrality_to_portfolio_weights(mis_weights))
```

```
{'ABCB4': 0.01, 'BBAS3': 0.01, 'BBDC4': 0.012, 'BRAP4': 0.009,
'BRML3': 0.01, 'BRPR3': 0.011, 'BTOW3': 0.051, 'CCRO3': 0.011,
'CMIG4': 0.01, 'CPFE3': 0.012, 'CSAN3': 0.01, 'CSMG3': 0.011,
'CSNA3': 0.013, 'CYRE3': 0.01, 'DIRR3': 0.009, 'DTEX3': 0.011,
'ECOR3': 0.009, 'ELET3': 0.013, 'ENBR3': 0.016, 'EQTL3': 0.01,
'EVEN3': 0.011, 'EZTC3': 0.01, 'GGBR4': 0.01, 'GOAU4': 0.009,
'GOLL4': 0.01, 'HBOR3': 0.01, 'HGTX3': 0.011, 'HYPE3': 0.012,
'IGTA3': 0.011, 'ITSA4': 0.011, 'ITUB4': 0.01, 'LAME4': 0.455,
'LIGT3': 0.011, 'LREN3': 0.009, 'MRVE3': 0.016, 'MULT3': 0.013,
'MYPK3': 0.011, 'PETR4': 0.01, 'POMO4': 0.01, 'RAPT4': 0.01, 'RENT3':
0.013, 'SBSP3': 0.01, 'TCSA3': 0.01, 'TIMP3': 0.016, 'UGPA3': 0.012,
'VALE3': 0.009, 'VIVT4': 0.011}


{'ITUB4': 0.02, 'UGPA3': 0.025, 'LAME4': 0.955}
```

Hedgecraft MIS allocates a staggering 91.4% of the investment to UNH. At first sight this portfolio appears far less diversified than Hedgecraft. However, if we recall, the relative risk of UNH was two orders of magnitude smaller than **every** other security (with the sole exception of Altaba). If our only options are the above 31 securities, the algorithm predicts UNH is the safest pick and allocates accordingly.

Allocating Shares to the Hedgecraft Portfolio

[ ]
```
# !pip install pyportfolioopt
```

[236]
```
# imports a tool to convert capital into shares
from pypfopt import discrete_allocation

# returns the number of shares to buy given the asset weights,
prices, and capital to invest
alloc = discrete_allocation.DiscreteAllocation(
    weights,
    df_buy_in['Buy In: {}'.format(TRAIN_RANGE[1])],
    total_portfolio_value=capital
)

# returns same as above but for the MIS
mis_alloc = discrete_allocation.DiscreteAllocation(
    mis_weights,
    df_mis_buy_in['Buy In: {}'.format(TRAIN_RANGE[1])],
    total_portfolio_value=capital
)
```

```
23 out of 47 tickers were removed
0 out of 3 tickers were removed
```

[237]
```
alloc = alloc.greedy_portfolio()[0]

mis_alloc = mis_alloc.greedy_portfolio()[0]
```

[238]
```
# converts above shares to a pandas series
alloc_series = pd.Series(alloc, name='Shares')

# names the series
alloc_series.index.name = 'Assets'

# resets index, prints assets with the shares we buy
alloc_series.reset_index
print(alloc_series)

print('\n')

# does same as above but for the MIS
mis_alloc_series = pd.Series(mis_alloc, name='MIS Shares')
mis_alloc_series.index.name = 'Assets'
mis_alloc_series.reset_index
```

```
print(mis_alloc_series)
```

```
Assets
LAME4    219
BTOW3     11
ENBR3     10
MRVE3     12
TIMP3     12
CSNA3     13
ELET3      5
MULT3      5
RENT3      4
BBDC4      3
CPFE3      3
HYPE3      3
UGPA3      2
```

[239]
```
# converts Hedgecraft shares series to a DataFrame
df_alloc = alloc_series.sort_index().to_frame('Shares')

# converts Hedgecraft MIS shares series to a DataFrame
df_mis_alloc = mis_alloc_series.sort_index().to_frame('MIS
Shares')
```

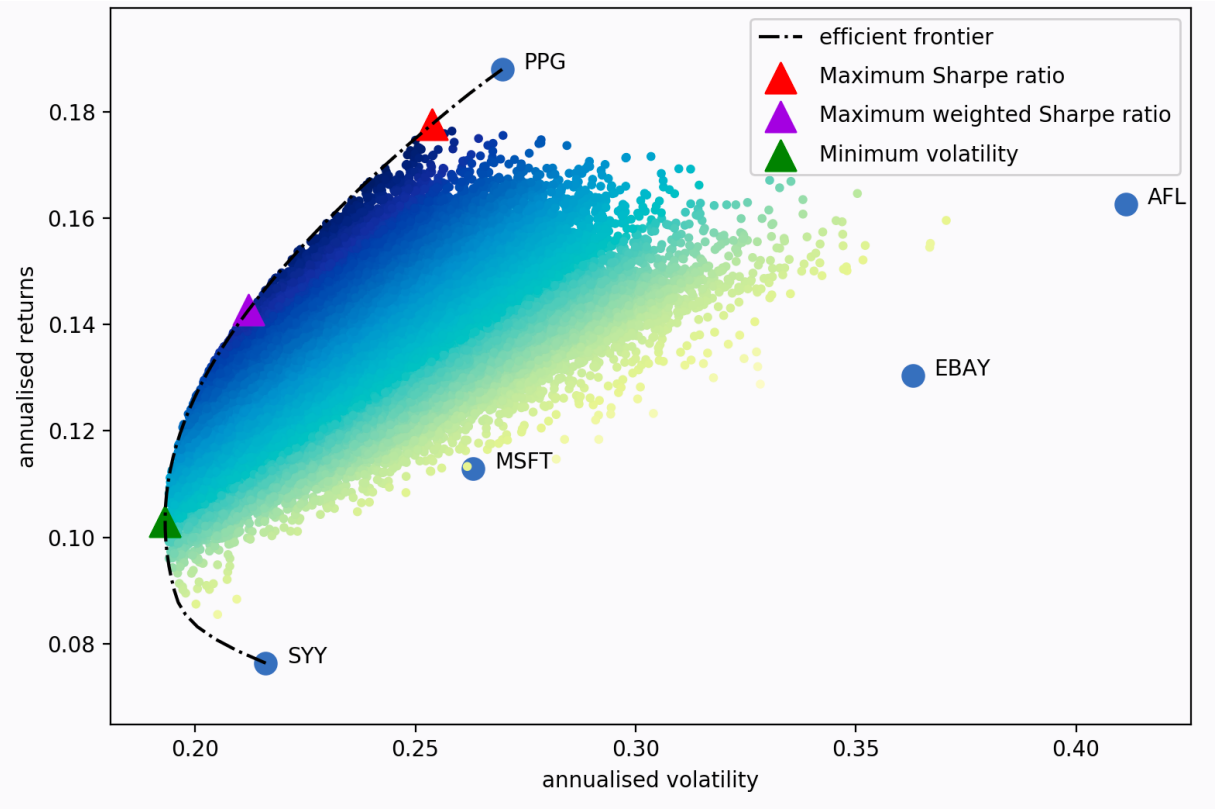
The Efficient Frontier

Harry Markowitz's 1952 paper Portfolio Selection transformed portfolio management from an art to a science. Markowitz's key insight came from diversification: by combinining uncorrelated assets with different expected returns and volatilities, one can calculate an optimal allocation.

The main idea is this: if w is the fraction of capital to be allocated to some asset, then the portfolio risk in terms of the covariance matrix Σ is given by $w^\top \Sigma w$ and its expected returns are given by $w^\top \mu$. The optimal portfolio can therefore be regarded as a convex optimization problem, and a solution can be found with quadratic programming. Denoting the target returns as $\mu*$ the optimization problem is mathmeatically expressed as follows:

$$\begin{split}\begin{equation*} \begin{aligned} & \underset{w}{\text{minimise}} & & w^T \Sigma w \\ & \text{subject to} & & w^T\mu \geq \mu^*\\ &&& w^T\mathbf{1} = 1 \\ &&& w_i \geq 0 \\ \end{aligned} \end{equation*}\end{split}$$

Varying the target return yields a different set of weights (i.e., a different portfolio). The set of all the portfolios with optimal weights is referred to as the **efficient frontier**.



Each dot in the above plot represents a different possible portoflio, with darker blue corresponding to better portfolios (in terms of the Sharpe ratio). The dotted black line is the frontier. The triangular markers denote the optimal portfolios subjected to different optimization objectives.

In the code blocks bellow we use the PyPortfolioOpt library to implement the Efficient Frontier algorithm. Rather than optimizing for a maximum Sharpe ratio, we optimize for a minimum volatility as these portfolios tend to outperform portfolio's with a maximized Sharpe ratio.

[240]
```
# imports a tool to calculate the mean historical return from a
# portfolio optimization package called pypfopt
from pypfopt.expected_returns import mean_historical_return

# returns the mean historical return of the training data
mu = mean_historical_return(df_train_close_copy)

# computes the covariance matrix
S = df_train_close_copy.cov()
```

[241]
```
# imports the efficient frontier model for asset allocation
from pypfopt.efficient_frontier import EfficientFrontier

# runs the efficient frontier (EF) algo
ef = EfficientFrontier(mu, S)

# computes portfolio weights subject to minimizing the
volatility (portfolio std)
weights_ef = ef.min_volatility()
```

```
# rounds and prints the weights
cleaned_weights = ef.clean_weights()
print(cleaned_weights)
```

```
{'ABCB4': 0.0, 'BBAS3': 0.0, 'BBDC4': 0.0, 'BRAP4': 0.0, 'BRML3':
0.0, 'BRPR3': 0.0, 'BTOW3': 0.0027, 'CCRO3': 0.0, 'CMIG4': 0.0,
'CPFE3': 0.0, 'CSAN3': 0.0, 'CSMG3': 0.0, 'CSNA3': 0.0, 'CYRE3': 0.0,
'DIRR3': 0.0, 'DTEX3': 0.0, 'ECOR3': 0.0, 'ELET3': 0.0, 'ENBR3':
0.29694, 'EQTL3': 0.0, 'EVEN3': 0.0, 'EZTC3': 0.0, 'GGBR4': 0.0,
'GOAU4': 0.0, 'GOLL4': 0.0, 'HBOR3': 0.0, 'HGTX3': 0.0, 'HYPE3': 0.0,
'IGTA3': 0.0, 'ITSA4': 0.0, 'ITUB4': 0.0, 'LAME4': 0.22828, 'LIGT3':
0.0, 'LREN3': 0.00102, 'MRVE3': 0.0, 'MULT3': 0.0, 'MYPK3': 0.0,
'PETR4': 0.0, 'POMO4': 0.0, 'RAPT4': 0.0, 'RENT3': 0.0, 'SBSP3': 0.0,
'TCSA3': 0.24299, 'TIMP3': 0.21584, 'UGPA3': 0.01223, 'VALE3': 0.0,
'VIVT4': 0.0}
```

Allocating Shares to the Markowitz Portfolio

[242]
```
# returns the number of shares to buy given asset prices,
weights, and capital to invest for the EF model
ef_alloc = discrete_allocation.DiscreteAllocation(
    weights_ef,
    df_buy_in['Buy In: {}'.format(TRAIN_RANGE[1])],
    total_portfolio_value=capital
)
```

```
42 out of 47 tickers were removed
```

[243]
```
ef_alloc = ef_alloc.greedy_portfolio()[0]
```

[244]
```
# converts EF shares to a pandas series
ef_alloc_series = pd.Series(ef_alloc, name='Shares')

# names the series
ef_alloc_series.index.name = 'Assets'

# resets index, prints assets and the shares we buy
ef_alloc_series.reset_index
```

```
<bound method Series.reset_index of Assets
ENBR3     191
TCSA3    1591
LAME4     110
```

```
TIMP3     173
UGPA3       2
Name: Shares, dtype: int64>
```

The Efficient Frontier produces a radically different portfolio than Hedgecraft or its MIS variant.

[245]
```
# converts EF shares series to a DataFrame
df_ef_alloc = ef_alloc_series.sort_index().to_frame('Shares')
```

Portfolio Performance: Hedgecraft vs. Efficient Frontier

In this section we write production (almost) ready code for portfolio analysis and include our own risk-adjusted returns score. The section looks something like this:

- We obtain the cumulative returns and returns on investment,
- extract the end of year returns and annual return rates,
- calculate the average annual rate of returns and annualized portfolio standard deviation,
- compute the Sharpe Ratio,
- Maximum Drawdown,
- Returns over Maximum Drawdown,
- and our own unique measure: the Growth-Risk Ratio.

Finally, we visualize the returns, drawdowns, and returns distribution of each model and analyze the results.

[246]
```
# total capital invested in the Hedgecraft portfolio after
buying shares
capital = (df_buy_in['Buy In:
{}'.format(TRAIN_RANGE[1])]*df_alloc['Shares']).sum()

# total capital invested in Efficient Frontier portfolio after
buying shares
ef_capital = (df_buy_in['Buy In:
{}'.format(TRAIN_RANGE[1])]*df_ef_alloc['Shares']).sum()

# total capital invested in Hedgecraft MIS portfolio after
buying shares
mis_capital = (df_mis_buy_in['Buy In:
{}'.format(TRAIN_RANGE[1])]*df_mis_alloc['MIS Shares']).sum()
```

```
# function to compute the cumulative returns of a portfolio
def cumulative_returns(shares_allocation, capital, test_data):

    """Returns the cumulative returns of a portfolio.

    Parameters
    ----------
    shares_allocation: DataFrame
        number of shares allocated to each asset in the
portfolio

    capital: float
        total amount of money invested in the portfolio

    test_data: DataFrame
        daily closing prices of portfolio assets

    Returns
    -------
    cumulative_daily_returns: Series
        cumulative daily returns of the portfolio

    """

    # list of DataFrames of cumulative returns for each stock
    daily_returns = []

    # iterates over every stock in the portfolio
    for stock in shares_allocation.index:

        # multiples shares by share prices in the validation
dataset

 daily_returns.append(shares_allocation.loc[stock].values *
test_data[stock])

    # concatenates every DataFrame in the above list to a
single DataFrame
    daily_returns_df = pd.concat(daily_returns,
axis=1).reset_index()

    # sets the index as the date
    daily_returns_df.set_index("Day", inplace=True)

    # adds the cumulative returns for every stock
    cumulative_daily_returns = daily_returns_df.sum(axis=1)

    # returns the cumulative daily returns of the portfolio
    return cumulative_daily_returns
```

```
# Hedgecraft cumulative daily returns
total_daily_returns = cumulative_returns(
    df_alloc,
    capital,
    df_validate
).rename('Hedgecraft Cumulative Daily Returns')


# Efficient Frontier cumulative daily returns
ef_total_daily_returns = cumulative_returns(
    df_ef_alloc,
    ef_capital,
    df_validate
).rename('EF Cumulative Daily Returns')


# Hedgecraft MIS cumulative daily returns
mis_total_daily_returns = cumulative_returns(
    df_mis_alloc,
    mis_capital,
    df_validate
).rename('MIS Cumulative Daily Returns')

# function to compute daily return on investment (roi)
def portfolio_daily_roi(shares_allocation, capital, test_data):

    """Returns the daily return on investment.

    Parameters
    ----------
    shares_allocation: DataFrame
        number of shares allocated to each asset

    capital: float
        total amount of money invested in the portfolio

    test_data: DataFrame
        daily closing prices of each asset

    Returns
    -------
    daily_roi: Series
        daily return on investment of the portfolio

    """

    # computes the cumulative returns
    cumulative_daily_returns = cumulative_returns(
        shares_allocation,
        capital,
```

```
            test_data
        )

        # calculates daily return on investment
        daily_roi = cumulative_daily_returns.apply(
            lambda returns: ((returns - capital) / capital)*100
        )

        # returns the daily return on investment
        return daily_roi



    # Hedgecraft daily return on investment
    daily_roi = portfolio_daily_roi(
        df_alloc,
        capital,
        df_validate
    ).rename('Hedgecraft Daily Returns')


    # Efficient Frontier daily return on investment
    ef_daily_roi = portfolio_daily_roi(
        df_ef_alloc,
        ef_capital,
        df_validate
    ).rename('EF Daily Returns')


    # Hedgecraft MIS daily return on investment
    mis_daily_roi = portfolio_daily_roi(
        df_mis_alloc,
        mis_capital,
        df_validate
    ).rename('MIS Daily Returns')
```

End of Year Returns

[247]
```
    # imports datetime manipluation library
    from datetime import datetime

    # function to extract the end of year returns
    def end_of_year_returns(model_roi, return_type, start, end):

        """Returns the end of year returns of a portfolio.
```

```
    Parameters
    ----------
    model_roi: Series
        portoflio returns on investment

    return_type: string
        'returns': returns roi
        'returns_rate': returns rate of returns

    start: int
        starting year to extract last trading day from

    end: int
        ending year to extract last trading day from

    Returns
    -------
    end_of_year_returns: dictionary
        each year's returns or rate of returns

    """

    # converts index of datetimes to a list of strings
    dates = model_roi.index.astype('str').tolist()

    # list comprehension of a string of dates between the
    # start and end dates
    years = [str(x) for x in range(start, end + 1)]

    # generates an empty list of lists for each year
    end_year_dates = [[] for _ in range(len(years))]

    # iterates over every date in the roi series
    for date in dates:

        # iterates over every year in the years list
        for year in years:

            # iterates over every year in each date
            if year in date:

                # converts each date string to a datime object
                datetime_object = datetime.strptime(date, '%Y-
%m-%d')

                # appends each date to its corresponding year
in the years list
                (end_year_dates[years.index(year)]
                    .append(datetime.strftime(datetime_object,
'%Y-%m-%d')))
```

```
    # finds the last date in each year
    end_year_dates = [max(x) for x in end_year_dates]

    # gets the rounded end of year returns
    returns = [round(model_roi[date], 1) for date in
end_year_dates]

    # shifts the returns list by 1 and appends 0 to the
beginning of the list
    return_rates = [0] + returns[:len(returns)-1]
    """Example: [a, b, c] -> [0, a, b]"""

    # converts returns list to an array
    returns_arr = np.array(returns)

    # converts the return_rates list to an array
    return_rates_arr = np.array(return_rates)

    # calculates the rounded rate of returns
    return_rates = [round(x, 1) for x in list(returns_arr -
return_rates_arr)]
    """Example: [a, b, c] - [0, a, b] = [a, b-a, c-b]"""

    # dictionary with the years as keys and returns as values
    returns = dict(zip(years, returns))

    # dictionary with the years as keys and return rates as
values
    return_rates = dict(zip(years, return_rates))

    if return_type == 'returns':
        return returns

    if return_type == 'return_rates':
        return return_rates


# Hedgecraft annual return rates
returns_dict = end_of_year_returns(
    daily_roi,
    'return_rates',
    2019,
    2020
)

# Efficient Frontier annual return rates
ef_returns_dict = end_of_year_returns(
    ef_daily_roi,
    'return_rates',
    2019,
    2020
)
```

```
# Hedgecraft MIS annual return rates
mis_returns_dict = end_of_year_returns(
    mis_daily_roi,
    'return_rates',
    2019,
    2020
)

# Hedgecraft annual returns
tot_returns_dict = end_of_year_returns(
    daily_roi,
    'returns',
    2019,
    2020
)

# Efficient Frontier annual returns
ef_tot_returns_dict = end_of_year_returns(
    ef_daily_roi,
    'returns',
    2019,
    2020
)

# Hedgecraft MIS annual returns
mis_tot_returns_dict = end_of_year_returns(
    mis_daily_roi,
    'returns',
    2019,
    2020
)
```

Average Annual Rate of Returns

[248]
```
# function to calculate avg annual portfolio returns
def avg_annual_returns(end_of_year_returns, mstat):

    """Returns average annual returns.

    Parameters
    ----------
    end_of_year_returns: dictionary
        annual returns

    mstat: string
```

```
            'arithmetic': returns the arithmetic mean
            'geometric': returns the geometric mean

        Returns
        -------
        average annual returns: float

        """

        # imports mean stats
        from scipy.stats import mstats

        # converts returns dict to an array (in decimal fmt)
        returns_arr =
    np.array(list(end_of_year_returns.values()))/100

        if mstat == 'geometric':

            # calculates the geometric mean
            gmean_returns = (mstats.gmean(1 + returns_arr) - 1)*100

            return round(gmean_returns, 2)

        if mstat == 'arithmetic':

            # calculates the arithmetic mean
            mean_returns = np.mean(returns_arr)

            return round(mean_returns, 2)


    # Hedgecraft avg annual returns
    gmean_returns = avg_annual_returns(returns_dict,
    mstat='geometric')

    # Efficient Frontier avg annual returns
    ef_gmean_returns = avg_annual_returns(ef_returns_dict,
    mstat='geometric')

    # Hedgecraft MIS avg annual returns
    mis_gmean_returns = avg_annual_returns(mis_returns_dict,
    mstat='geometric')

    print(gmean_returns)
    print(ef_gmean_returns)
    print(mis_gmean_returns)
```

```
14.75
-1.72
19.18
```

Annualized Portfolio Standard Deviation

The annualized standard deviation (a.k.a volatility) measures how far the portfolio's returns vary from the mean, that is, it measures how spread out the data is. It can be readily calculated as:

$$\sigma = \sqrt{252 \cdot \mathbf{w}^\top \mathbf{\Sigma} \mathbf{w}}$$

where 252 is the number of trading days in a year, $\mathbf{w}$ is the vector of portfolio weights, and $\mathbf{\Sigma}$ is the covariance matrix.

Annualized Sharpe Ratio

The Sharpe ratio is one of the most popular measures of risk-adjusted returns. It measures the excess of mean returns per unit deviation of the returns. The annualized Sharpe Ratio is given by,

$$ S = \dfrac{\bar{R} - \bar{R}_{f}}{\sigma} $$

where $\bar{R}$ is the average annual rate of returns, $\bar{R}_f$ is the average risk-free rate for the duration of the investment (usually taken as the 10-year treasurey rate), and σ is the annualized standard deviation. A Sharpe ratio greater than one usually signifies a portfolio of superior performance as opposed to a portfolio with a Sharpe ratio less than one. In general, a portfolio with a larger Sharpe ratio will outperform one with a smaller ratio. A snazzy article summarizing the use of the Sharpe ratio is given here.

[249]
```
# function to calculate annualized portoflio standard deviation
def portfolio_std(weights, test_data):

    """Returns annualized portfolio volatility.

    Parameters
    ----------
    weights: dictionary
        portfolio weights

    test_data: DataFrame
        validation data set

    Returns
    -------
    portfolio_std_dev: float
        annualized portfolio standard deviaion

    """
```

```
    # computes daily change in returns from 2015-2017
    daily_ret_delta = test_data.pct_change()

    # computes the covariance matrix of the above
    cov_matrix = daily_ret_delta.cov()

    # initializes weights
    weights_list = []

    # iterates over weights dict and appends above list
    for key, value in weights.items():
        weights_list.append(value)

    # converts weights list to numpy array
    weights_arr = np.array(weights_list)

    # calculates the annualized portfolio standard deviation
from 2015-2017 in pct format
    portfolio_std_dev = np.sqrt(
        np.dot(
            weights_arr.T,
            np.dot(
                cov_matrix,
                weights_arr
            )
        )
    )*np.sqrt(252)*100

    return round(portfolio_std_dev, 2)


# function to calculate annualized portfolio standard deviation
with a
# maximum independent set parameter
def mis_portfolio_std(weights, test_data,
maximum_independent_set):

    """Returns annualized portfolio volatility.

    Parameters
    ----------
    weights: dictionary
        portfolio weights

    test_data: DataFrame
        validation data set

    maximum_independent_set: list
        largest list of assets such that no two are adjacent

    Returns
    -------
```

```
        portfolio_std_dev: float
            annualized portfolio standard deviation

        """

        # computes daily change in returns from 2015-2017
        daily_ret_delta =
    test_data[maximum_independent_set].pct_change()

        # computes the covariance matrix
        cov_matrix = daily_ret_delta.cov()

        # initializes weights list
        weights_list = []

        # iterates over weights dict and appends above list
        for key, value in weights.items():
            weights_list.append(value)

        # converts weights list to numpy array
        weights_arr = np.array(weights_list)

        # calculates portfolio standard deviation from 2015-2017
        portfolio_std_dev = np.sqrt(
            np.dot(
                weights_arr.T,
                np.dot(
                    cov_matrix,
                    weights_arr
                )
            )
        )*np.sqrt(252)*100

        return round(portfolio_std_dev, 2)


    # Hedgecraft annualized volatility
    portfolio_std_dev = portfolio_std(
        weights=weights,
        test_data=df_validate
    )

    # Efficient Frontier annualized volatility
    ef_portfolio_std_dev = portfolio_std(
        weights=weights_ef,
        test_data=df_validate
    )

    # Hedgecraft MIS annualized volatility
    mis_portfolio_std_dev = mis_portfolio_std(
        weights=mis_weights,
        test_data=df_validate,
```

```
    maximum_independent_set=max_ind_set
)

print(portfolio_std_dev)
print(ef_portfolio_std_dev)
print(mis_portfolio_std_dev)
```

```
45.37
39.15
53.25
```

[250]
```
# function to compute the Sharpe ratio
def portfolio_sharpe_ratio(avg_annual_returns, portfolio_std,
risk_free_rate):

    """Returns Sharpe ratio.

    Parameters
    ----------
    avg_annual_returns: float
        portoflio avg annual returns

    portfolio_std: float
        annualized portfolio volatility

    risk_free_rate: float
        usually taken as the avg 10-year treasury rate over
investment period

    Returns
    -------
    portfolio_std_dev: float
        annualized portfolio standard deviaion

    """

    # calculates the Sharpe ratio
    sharpe_ratio = (avg_annual_returns - risk_free_rate) /
portfolio_std

    return round(sharpe_ratio, 2)

# Sharpe ratio of the Hedgecraft portfolio
## we use a 2% risk free rate
sharpe_ratio = portfolio_sharpe_ratio(
    gmean_returns,
    portfolio_std_dev,
    2
)

# Sharpe ratio of the Markowitz portfolio
```

```
ef_sharpe_ratio = portfolio_sharpe_ratio(
    ef_gmean_returns,
    ef_portfolio_std_dev,
    2
)

# Sharpe ratio of the Hedgecraft MIS portfolio
mis_sharpe_ratio = portfolio_sharpe_ratio(
    mis_gmean_returns,
    mis_portfolio_std_dev,
    2
)

print(sharpe_ratio)
print(ef_sharpe_ratio)
print(mis_sharpe_ratio)
```

```
0.28
-0.1
0.32
```

Drawdown Statistics

The drawdown is the extent to which the portfolio's cumulative returns falls below its historical maximum. Put plainly, it's the net loss since the previous peak. The drawdown can also be interpreted as the rate at which the portfolio shrinks and is commonly thought of as a measure of downside risk, that is, the risk associated with losses. The *drawdown duration* or 'time under water' is the length of peak-to-peak periods, or the time required for the portfolio to recoup its losses. As such, smaller drawdown durations enable a portfolio's growth to outpace its losses. Formally, the drawdown is defined in two steps: first, we find the rolling maximum up to time t,

$$M_{t} = \text{max}_{u \in [0, t]} \ R_{u}$$

where R_t is cumulative returns up to time t. The **drawdown** up to time t (as a percentage) is then,

$$DD_{t} = \dfrac{R_{t}}{M_{t}} - 1.$$

The **maximum drawdown** is thus the largest drop in cumulative returns from its rolling maximum over a given time frame:

$$MDD_{t} = \text{min}_{u \in [0, t]} \ DD_{u}.$$

Below we write code that returns the daily rolling maximum (`daily_rolling_max`) in a 252-day window. That is, we take $u \in [t_i, t_f]$ where t_i and t_f denote the beginning and end of each trading year, respectively. Then we compute the drawdown (`daily_rolling_drawdown`), the maximum drawdown (`max_daily_rolling_drawdown`), and the lifetime maximum drawdown (`lifetime_max_drawdown`) for each portfolio. For more information see Vercer, 2006.

[254]
```
# function to compute the 252-day daily rolling maximum
def daily_rolling_max(cumulative_returns, window=252,
min_periods=1):

    """Returns the daily running 252-day maximum.

    Parameters
    ----------
    cumulative_returns: Series
        portoflio's cumulative returns

    window: int, default 252
        size of the moving window.

    min_periods: int
        minimum number of observations in window required to
have a value

    Returns
    -------
    daily_rolling_max: Series

    """

    return cumulative_returns.rolling(
        window=window,
        min_periods=min_periods
    ).max()


# function to compute the 252-day rolling drawdown
def daily_rolling_drawdown(cumulative_returns, rolling_max):

    """Returns the daily running 252-day drawdown.

    Parameters
    ----------
    cumulative_returns: Series
        portoflio's cumulative returns

    rolling_max: Series
        rolling 252-day maximum

    Returns
    -------
    daily_rolling_drawdown: Series

    """
```

```
        return (cumulative_returns / rolling_max) - 1


    # function to compute the 252-day maximum daily drawdown
    def max_daily_rolling_drawdown(daily_drawdown, window=252,
    min_periods=1):

        """Returns the daily running 252-day maximum daily
    drawdown.

        Parameters
        ----------
        daily_drawdown: Series
           daily rolling 252-day drawdown

        window: int, default 252
            size of the moving window.

        min_periods: int
            minimum number of observations in window required to
    have a value

        Returns
        -------
        max_daily_rolling_drawdown: Series

        """

        return daily_drawdown.rolling(
            window=window,
            min_periods=min_periods
        ).min()


    # function to compute the lifetime maximum drawdown
    def lifetime_max_drawdown(daily_drawdown):

        """Returns the lifetime maximum drawdown.

        Parameters
        ----------
        daily_drawdown: Series
           daily rolling 252-day drawdown

        Returns
        -------
        lifetime_max_drawdown: float
            largest amount of money lost

        """

        return round(daily_drawdown.min()*100, 2)
```

```
# Hedgecraft 252-day daily rolling maximum, 252-day daily
rollng drawdown,
# 252-day daily rolling maximum drawdown, and lifetime max
drawdown
roll_max = daily_rolling_max(total_daily_returns)
daily_drawdown = daily_rolling_drawdown(
    total_daily_returns,
    roll_max
)
max_daily_drawdown = max_daily_rolling_drawdown(daily_drawdown)
max_drawdown = lifetime_max_drawdown(daily_drawdown)

# Efficient Frontier 252-day daily rolling maximum, 252-day
daily rollng drawdown,
# 252-day daily rolling maximum drawdown, and lifetime max
drawdown
ef_roll_max = daily_rolling_max(ef_total_daily_returns)
ef_daily_drawdown = daily_rolling_drawdown(
    ef_total_daily_returns,
    ef_roll_max
)
ef_max_daily_drawdown =
max_daily_rolling_drawdown(ef_daily_drawdown)
ef_max_drawdown = lifetime_max_drawdown(ef_daily_drawdown)

# Hedgecraft MIS 252-day daily rolling maximum, 252-day daily
rollng drawdown,
# 252-day daily rolling maximum drawdown, and lifetime max
drawdown
mis_roll_max = daily_rolling_max(mis_total_daily_returns)
mis_daily_drawdown = daily_rolling_drawdown(
    mis_total_daily_returns,
    mis_roll_max
)
mis_max_daily_drawdown =
max_daily_rolling_drawdown(mis_daily_drawdown)
mis_max_drawdown = lifetime_max_drawdown(mis_daily_drawdown)
```

Return Over Maximum Drawdown (Risk-Return-Ratio)

Deviation-based performance measures (like the Sharpe Ratio) implicity assume the distribution of returns follows a normal distribution. There's a growing a consensus the returns of hedge funds are not normal. As such, the Return Over Maximum Drawdown

(RoMaD) has become a popular risk-adjusted return metric for hedge funds. The metric attempts to answer the following question: is the invester willing to accept an occasional drawdown of $X\%$ in order to generate a return on investment of $Y\%$? Mathematically, we express the risk-return-ratio as follows,

$$RRR = \dfrac{R_{t}}{|MDD_{t}|}$$

where R_t is the return on investment at time t and $|MDD_t|$ is the absolute value of the maximum drawdown over some specified time interval. Usually, R_t is taken as either the return on investment to-date or the average annual return, while MDD_t is taken over the entire history of the portfolio, i.e., it's the lifetime maximum drawdown. We take R_t as the former. There are numerous variations of the metric, including the Calmar and Sterling ratio. In practice, portfolio managers aim to observe a risk-return-ratio of at least two or more. We provide the risk-return-ratio in the code block below. For more information see this article and Johnsson, 2010.

[255]
```
# calculates returns over lifetime maximum drawdown
def returns_over_max_drawdown(tot_returns_dict, year,
lifetime_maximum_drawdown):

    """Returns the lifetime maximum drawdown.

    Parameters
    ----------
    tot_returns_dict: dictionary
       cumulative annual portfolio returns

    year: int

    lifetime_maximum_drawdown: float
        largest amount of money lost

    Returns
    -------
    returns_over_max_drawdown: float
        cumulative returns divded by largest sum of money lost

    """

    return round(tot_returns_dict[year] /
abs(lifetime_maximum_drawdown), 2)


# Hedgecraft risk-return ratio
risk_return_ratio = returns_over_max_drawdown(
    tot_returns_dict,
    '2019',
    max_drawdown
)
```

```
# Efficient Frontier risk-return ratio
ef_risk_return_ratio = returns_over_max_drawdown(
    ef_tot_returns_dict,
    '2019',
    ef_max_drawdown
)

# Hedgecraft MIS risk-return ratio
mis_risk_return_ratio = returns_over_max_drawdown(
    mis_tot_returns_dict,
    '2019',
    mis_max_drawdown
)

print(risk_return_ratio)
print(ef_risk_return_ratio)
print(mis_risk_return_ratio)
```

```
0.8
0.77
0.66
```

Growth-Risk-Ratio

We are now in position to define our own risk-adjusted performance metric to evaluate and compare the Hedgecrafted and Markowitz portfolios. Like the risk-return-ratio, we aim to define a metric of downside-risk-adjusted returns that answers the following question: At what rate does the portfolio's growth outpace the frequency and magnitude of its largest losses? That is, how consistent is its rate of growth? To this end, let's define the growth-risk-ratio as,

$$\text{GRR} = \dfrac{\overline{R}}{|\overline{MDD_{t}}|}$$

where $\overline{R}$ is the average annual rate of returns and $\overline{MDD_t}$ is the average 252-day rolling maximum drawdown. The growth-risk-ratio is similiar to the Sterling ratio which is defined as the average annual rate of returns divided by the average annual maximum drawdown. Here MDD_t replaces the denominator of the Sterling ratio, giving a more fine-grained account of the portfolio's losses by tracking the dynamic behavior of the maximum drawdowns. If GRR < 1 the portfolio is at risk of losing all or most of its growth, whereas a GRR > 1 indicates the portfolio's growth outpaces the frequency and magnitude of its maximum drawdowns. The larger the value of GRR the more consistent the more consistent the growth rate. In what's to follow, we provide code for the growth-risk-ratio, collect the results of each metric, and discuss the performance of each portfolio.

[256]
```
# function to calculate the growth-risk ratio
def portfolio_growth_risk(avg_annual_returns,
max_daily_rolling_drawdown):
```

```
    """Returns the growth-risk ratio.

        Parameters
        ----------
        avg_annual_returns: float
           average annual returns

        max_daily_rolling_drawdown: Series
            252-day rolling maximum daily drawdown

        Returns
        -------
        portfolio_growth_risk: float
            average annual returns divided by average rolling
max daily drawdown

    """

    return round(avg_annual_returns /
abs(max_daily_rolling_drawdown.mean()*100), 2)

# growth-risk ratios for each model
growth_risk_ratio = portfolio_growth_risk(gmean_returns,
max_daily_drawdown)
ef_growth_risk_ratio = portfolio_growth_risk(ef_gmean_returns,
ef_max_daily_drawdown)
mis_growth_risk_ratio =
portfolio_growth_risk(mis_gmean_returns,
mis_max_daily_drawdown)

print(growth_risk_ratio)
print(ef_growth_risk_ratio)
print(mis_growth_risk_ratio)
```

```
0.62
-0.1
0.66
```

Collecting the Results

It's the moment of truth: How well did Hedgecraft spar with the Efficient Frontier? Scroll down to find out!

[258]
```
def collect_results(model):

    if model == 'hedgecraft':
```

```
        collection = [[], [], []]

        collection[0].append([str(x) + '%' for x in
list(returns_dict.values())])
        collection[1].append([str(x) + '%' for x in
list(tot_returns_dict.values())])
        collection[2].append([str(x) + '%' for x in
[gmean_returns, portfolio_std_dev, max_drawdown]])
        collection[2].append([sharpe_ratio, risk_return_ratio,
growth_risk_ratio])

        return collection

    if model == 'efficient_frontier':

        collection = [[], [], []]

        collection[0].append([str(x) + '%' for x in
list(ef_returns_dict.values())])
        collection[1].append([str(x) + '%' for x in
list(ef_tot_returns_dict.values())])

        collection[2].append([str(x) + '%' for x in [
            ef_gmean_returns,
            ef_portfolio_std_dev,
            ef_max_drawdown]
                             ]
                            )

        collection[2].append([ef_sharpe_ratio,
ef_risk_return_ratio, ef_growth_risk_ratio])

        return collection

    if model == 'hedgecraft_mis':

        collection = [[], [], []]

        collection[0].append([str(x) + '%' for x in
list(mis_returns_dict.values())])
        collection[1].append([str(x) + '%' for x in
list(mis_tot_returns_dict.values())])

        collection[2].append([str(x) + '%' for x in [
            mis_gmean_returns,
            mis_portfolio_std_dev,
            mis_max_drawdown]
                             ]
                            )
```

```
            collection[2].append([mis_sharpe_ratio,
mis_risk_return_ratio, mis_growth_risk_ratio])

            return collection

yearly_return_rates = collect_results(model = 'hedgecraft')[0]
[0]
yearly_returns = collect_results(model='hedgecraft')[1][0]

summary_stats = (collect_results(model='hedgecraft')[2][0]
    + collect_results(model='hedgecraft')[2][1])

ef_yearly_return_rates = collect_results(model =
'efficient_frontier')[0][0]
ef_yearly_returns = collect_results(model='efficient_frontier')
[1][0]
ef_summary_stats = (collect_results(model='efficient_frontier')
[2][0]
    + collect_results(model='efficient_frontier')[2][1])

mis_yearly_return_rates = collect_results(model =
'hedgecraft_mis')[0][0]
mis_yearly_returns = collect_results(model='hedgecraft_mis')[1]
[0]
mis_summary_stats = (collect_results(model='hedgecraft_mis')[2]
[0]
    + collect_results(model='hedgecraft_mis')[2][1])


# TODO: Get the Brazilain reference index (Bovespa)
#dow_yearly_return_rates=[0.10, 16.28, 27.97]
#dow_yearly_return_rates=[str(x) + '%' for x in
dow_yearly_return_rates]
#sp500_yearly_return_rates=[-0.73, 9.54, 19.42]
#sp500_yearly_return_rates=[str(x) + '%' for x in
sp500_yearly_return_rates]
```

[274]

```
# dictionary summary of model performances
returns_summary = {
                    'Close Returns': yearly_returns,
                    'Close MIS Returns': mis_yearly_returns,
                    'Efficient Frontier Returns':
ef_yearly_returns,
                    #'Dow Return Rates':
dow_yearly_return_rates,
                    #'S&P500 Return Rates':
sp500_yearly_return_rates,
                    'Close Return Rates': yearly_return_rates,
                    'Close MIS Return Rates':
mis_yearly_return_rates,
```

```
                        'Efficient Frontier Return Rates':
    ef_yearly_return_rates,
    }

    # converts above dict to a DataFrame
    returns_summary = pd.DataFrame.from_dict(returns_summary)

    # relabels indices as years
    returns_summary = returns_summary.rename(index={0:'2019',
    1:'2020'});

    # dictionary of stats
    backtest_stats = {
                      'Close': summary_stats,
                      'Close MIS': mis_summary_stats,
                      'Efficient Frontier': ef_summary_stats
    }

    # converts above dict to DataFrame
    backtest_stats =  pd.DataFrame.from_dict(backtest_stats)

    # renames the indices
    backtest_stats = backtest_stats.rename(index={
                                            0:'Avg Annual Rate of
    Returns',
                                            1:'Annual Volatility',
                                            2:'Maximum Drawdown',
                                            3:'Annualized Sharpe
    Ratio',
                                            4:'Returns Over Maximum
    Drawdown',
                                            5:'Growth-Risk Ratio'
                                    }
    );
```

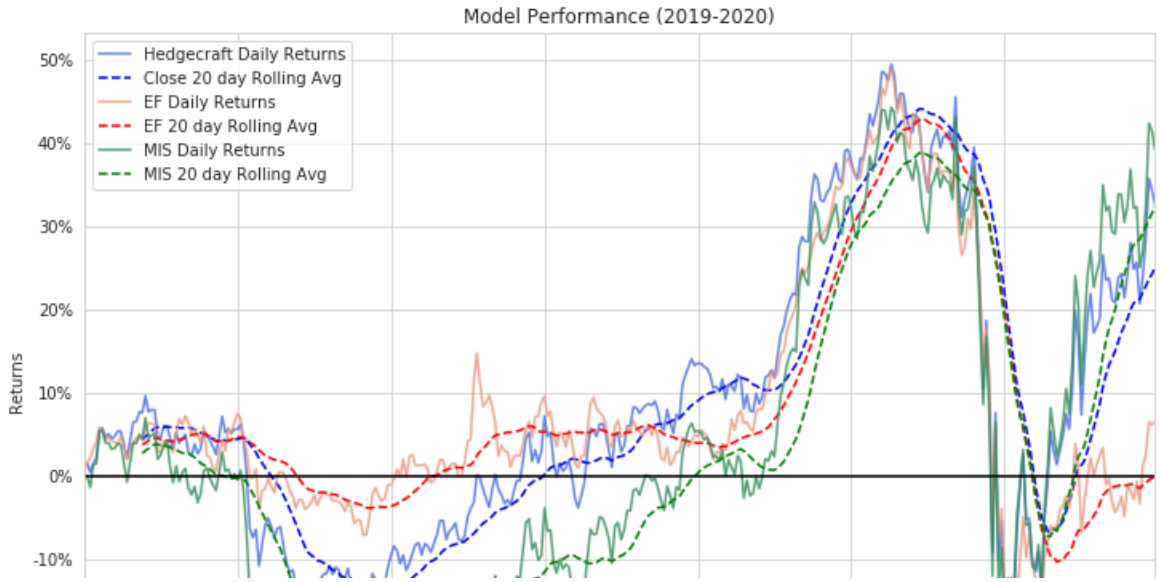
Visualizing the Returns

[261]
```
    # imports matplotlib submodule to aesthetically place date
    ticks
    from matplotlib.dates import AutoDateFormatter, AutoDateLocator

    # returns Hedgecraft 20 day moving avg
    rolling_20d_avg = (daily_roi.rename('Close 20 day Rolling Avg')
                               .rolling(20)
                               .mean()
    )
```

```
# returns Efficient Frontier 50 day moving avg
ef_rolling_20d_avg = (ef_daily_roi.rename('EF 20 day Rolling
Avg')
                              .rolling(20)
                              .mean()
)

# returns Hedgecraft MIS 50 day moving avg
mis_rolling_20d_avg = (mis_daily_roi.rename('MIS 20 day Rolling
Avg')
                                  .rolling(20)
                                  .mean()
)
```

[266]
```
with sns.axes_style('whitegrid'):
    # initializes figure and axis
    fig = plt.figure(figsize=(12,8))
    ax = fig.add_subplot(111)

    # gets xtick postions for datetime objects, set minimum
number of xticks to 3
    xtick_locator = AutoDateLocator(minticks=3)

    # aesthetically formats xticks
    xtick_formatter = AutoDateFormatter(xtick_locator)

    # returns pretty seaborn plot of hedgecraft returns and its
20 day rolling avg
    daily_roi.plot(color='royalblue', alpha=0.8)
    rolling_20d_avg.plot(color='blue', ls='dashed')

    # returns pretty seaborn plot of EF returns and its 20 day
rolling avg
    ef_daily_roi.plot(color='darksalmon', alpha=0.8)
    ef_rolling_20d_avg.plot(color='red', ls='dashed')

    # returns pretty seaborn plot of hedgecraft MIS returns and
its 20 day rolling avg
    mis_daily_roi.plot(color='seagreen', alpha=0.8)
    mis_rolling_20d_avg.plot(color='green', ls='dashed')

    # renders xticks, sets ylabel, and turns legend on
    #ax.xaxis.set_major_locator(xtick_locator)
    #ax.xaxis.set_major_formatter(xtick_formatter)
    ax.set_title('Model Performance (2019-2020)')
    ax.set_ylabel('Returns')
    ax.legend()

    # gets ytick labels and converts to pct format
```

```
        vals = ax.get_yticks()
        ax.set_yticklabels(['{:.0f}%'.format(x) for x in vals])

        # draws a horizontal line at 0, turns grid on
        plt.axhline(0, color='black');
        plt.tick_params(left=False, bottom=False)
```



Pictured above are the daily returns for Hedgecraft MIS (solid green curve), Hedgecraft (solid blue curve), and the Efficient Frontier portfolio (solid red curve) from January, 2019 to May, 2020. The color-coded dashed curves represent the 20 day rolling averages of the respective portfolios.

Several observations pop:

1. Efficient frontier performs well before the COVID-19 pandemia, but performs badly during the pandemia;
2. MIS is the opposite to EF;
3. Our risk portfolio `Close` shows more robusteness in the period, and obtain high returns during the pandemia period;
4. The results from Close and MIS suggests that this approach to minimize systemic risk performs well during crisis; and
5. It is good to note that the Brazilian market is not too stable in the recent years due to uncertainties in the political, economic, and social point of view.

Next, let's observe the annual returns for each portfolio and compare them with the market.

[267]
```
returns_summary
```

	Close	Close	Efficient	Close	Close MIS	Efficient Frontier

	Close Returns	MIS Close Returns	Efficient Frontier Returns	Close Return Rates	MIS Close Return Rates	Efficient Frontier Return Rates
2019	35.2%	29.0%	34.9%	35.2%	29.0%	34.8%
2020	32.6%	39.1%	6.5%	-2.6%	10.1%	-28.4%

TODO: Get the Bovespa!

In comparison, the Dow had a 0.10%, 16.28%, and 27.97% annual return rate in 2015, 2016, and 2017, respectively. The S&P500 had a -0.73%, 9.54%, and 19.42% annual return rate in the same respective years.

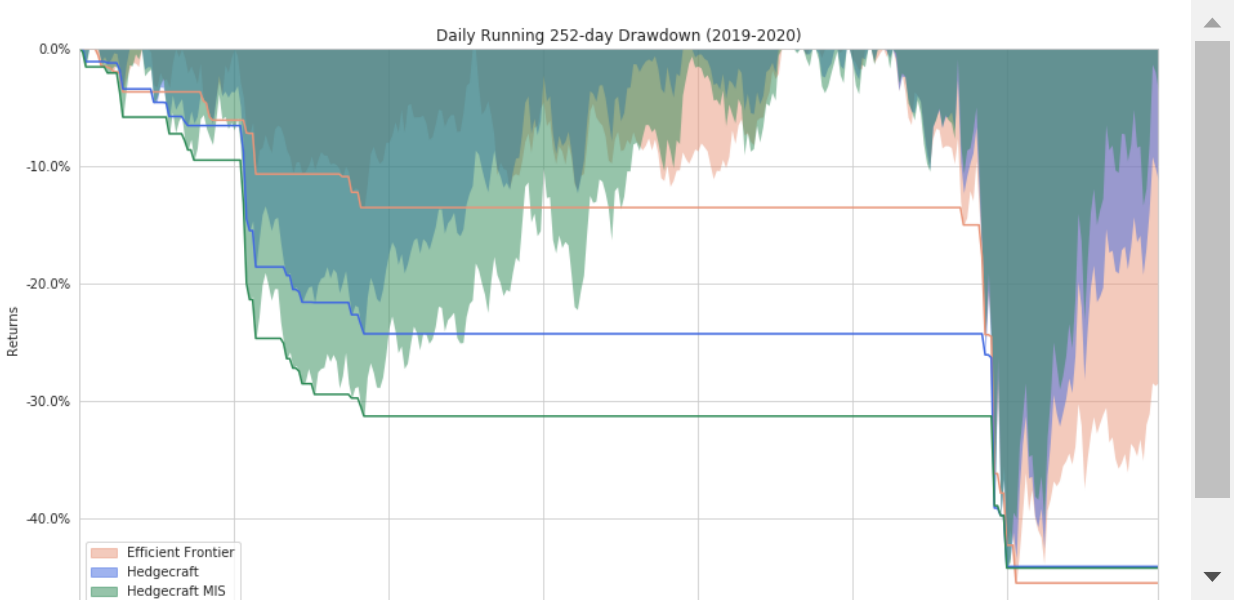
Close and Close MIS substantially outperformed both the market and the Efficient Frontier. Both Close portfolios grew at an impressive rate. What's more, Close MIS's return rates consistently increased about 20+% every year, whereas the return rates of Close and the Efficient Frontier were less consistent.

Close MIS clearly has the most consistent rate of growth. We'd expect this rapid growth to be accompanied with a large burden of risk---either manifested as a large degree of volatility, steep and frequent maximum drawdowns, or both. As we explore below, the Hedgecraft portfolios' sustained their growth rates with significantly less risk exposure than the Efficient Frontier.

Visualizing Drawdowns

[272]
```
# imports tool to build legends
import matplotlib.patches as mpatches

with sns.axes_style('whitegrid'):
    # initializes figure and axis
    fig = plt.figure(figsize=(15,8))
    ax = fig.add_subplot(111)

    # gets xtick postions for datetime objects, set minimum
number of xticks to 3
    xtick_locator = AutoDateLocator(minticks=3)

    # aesthetically formats xticks
    xtick_formatter = AutoDateFormatter(xtick_locator)

    ef_daily_drawdown.plot.area(ax=ax, linewidth=0, alpha=0.5,
color='darksalmon')
    ef_max_daily_drawdown.plot(color='darksalmon')
```

```
    daily_drawdown.plot.area(ax=ax, linewidth=0, alpha=0.5,
color='royalblue')
    max_daily_drawdown.plot(color='royalblue')

    mis_daily_drawdown.plot.area(ax=ax, linewidth=0, alpha=0.5,
color='seagreen')
    mis_max_daily_drawdown.plot(color='seagreen')

    # renders xticks, sets ylabel, and turns legend on
    #ax.xaxis.set_major_locator(xtick_locator)
    #ax.xaxis.set_major_formatter(xtick_formatter)

    ax.set_ylabel('Returns')

    # sets legend patches color and labels
    ef_patch = mpatches.Patch(color='darksalmon',
label='Efficient Frontier', alpha=0.5)
    patch = mpatches.Patch(color='royalblue',
label='Hedgecraft', alpha=0.5)
    mis_patch = mpatches.Patch(color='seagreen',
label='Hedgecraft MIS', alpha=0.5)

    # turns legend on with patches
    plt.legend(handles=[ef_patch, patch, mis_patch])

    plt.tick_params(left=False, bottom=False)
    plt.title('Daily Running 252-day Drawdown (2019-2020)')

     # gets ytick labels and converts to pct format
    vals = ax.get_yticks()
    ax.set_yticklabels([str(round(x*100,2)) + '%' for x in
vals])
```



Illustrated above is the daily rolling 252-day drawdown for Close MIS (filled sea green curve), Close (filled royal blue curve), and the Efficient Frontier (filled dark salmon curve) along with the respective rolling maximum drawdowns (solid curves).

Several observations stick out:

1. the Close portfolios have significantly smaller drawdowns than the portfolio generated from the Efficient Frontier;
2. All portfolios have roughly the same maximum drawdown (about 45%);
3. Close on average lost the least amount of returns; and
4. Close's rolling maximum drawdowns are, on average, less pronounced than Close MIS. These results suggest the communicability betweeness centrality has predictive power as a measure of relative or intraportfolio risk, and more generally, that network-based portfolio construction is a promosing alternative to the more traditional approaches like MPT.

Next, let's observe how each portfolio performed with the metrics we discussed earlier.

[275]
```
backtest_stats
```

	Close	Close MIS	Efficient Frontier
Avg Annual Rate of Returns	14.75%	19.18%	-1.72%
Annual Volatility	45.37%	53.25%	39.15%
Maximum Drawdown	-44.14%	-44.27%	-45.54%
Annualized Sharpe Ratio	0.28	0.32	-0.1
Returns Over Maximum Drawdown	0.8	0.66	0.77
Growth-Risk Ratio	0.62	0.66	-0.1

Close and its MIS variant dramatically outperformed the Efficient Froniter on every metric (save annual volatility). **These results give credence to the possibility that we are on to something substantial here as we have passed the criteria of our go/no go test**.

Outperforming MPT by these margins is no simple feat, but, the real test is whether or not they can consistently beat MPT on many randomly generated portfolios. To wrap up this notebook, let's take a look at how the returns for each portfolio are distributed and move to the conclusions.

Analyzing the Distribution of Returns

[280]
```
# probability of Hedgecraft losing money
```

```
prob_lose_money = (daily_roi[daily_roi < 0].shape[0]
                        / daily_roi.shape[0]
)

# probability of EF losing money
ef_prob_lose_money = (ef_daily_roi[ef_daily_roi < 0].shape[0]
                           / ef_daily_roi.shape[0]
)

# probability of Hedgecraft MIS losing money
mis_prob_lose_money = (mis_daily_roi[mis_daily_roi <
0].shape[0]
                            / mis_daily_roi.shape[0]
)

# Hedgecraft rolling 30 day avg
rolling_30d_avg = (daily_roi.rename('Hedgecraft 30 day Rolling
Avg')
                    .rolling(30)
                    .mean()
)

# EF rolling 30 day avg
ef_rolling_30d_avg = (ef_daily_roi.rename('EF 30 day Rolling
Avg')
                       .rolling(30)
                       .mean()
)

# Hedgecraft MIS rolling 30 day avg
mis_rolling_30d_avg = (mis_daily_roi.rename('MIS 30 day Rolling
Avg')
                        .rolling(30)
                        .mean()
)

# Hedgecraft rolling 90 day avg
rolling_90d_avg = (daily_roi.rename('Hedgecraft 90 day Rolling
Avg')
                    .rolling(90)
                    .mean()
)

# EF rolling 90 day avg
ef_rolling_90d_avg = (ef_daily_roi.rename('EF 90 day Rolling
Avg')
                       .rolling(90)
                       .mean()
)

# Hedgecraft MIS rolling 90 day avg
```

```
mis_rolling_90d_avg = (mis_daily_roi.rename('MIS 90 day Rolling
Avg')
                       .rolling(90)
                       .mean()
)

# Hedgecraft list of rolling avgs
rolling_avg_list = [rolling_30d_avg, rolling_50d_avg,
rolling_90d_avg]

# EF list of rolling avgs
ef_rolling_avg_list = [ef_rolling_30d_avg, ef_rolling_50d_avg,
ef_rolling_90d_avg]

# Hedgecraft MIS list of rolling avgs
mis_rolling_avg_list = [mis_rolling_30d_avg,
mis_rolling_50d_avg, mis_rolling_90d_avg]

# Hedgecraft probabilities of falling below 30, 50, and 90 day
rolling averages
prob_below_rolling_avgs = []
for i in rolling_avg_list:
    (prob_below_rolling_avgs.append(daily_roi[daily_roi <
i].shape[0]
                                    / daily_roi.shape[0]))

# Efficient Frontier probabilities of falling below 30, 50, and
90 day rolling averages
ef_prob_below_rolling_avgs = []
for i in ef_rolling_avg_list:

(ef_prob_below_rolling_avgs.append(ef_daily_roi[ef_daily_roi <
i].shape[0]
                                       /
ef_daily_roi.shape[0]))

# Hedgecraft MIS probabilities of falling below 30, 50, and 90
day rolling averages
mis_prob_below_rolling_avgs = []
for i in mis_rolling_avg_list:

(mis_prob_below_rolling_avgs.append(mis_daily_roi[mis_daily_roi
< i].shape[0]
                                        /
mis_daily_roi.shape[0]))

# worst day for the Hedgecraft model
max_loss = min(daily_roi)

# worst day for the Efficient Frontier model
ef_max_loss = min(ef_daily_roi)
```

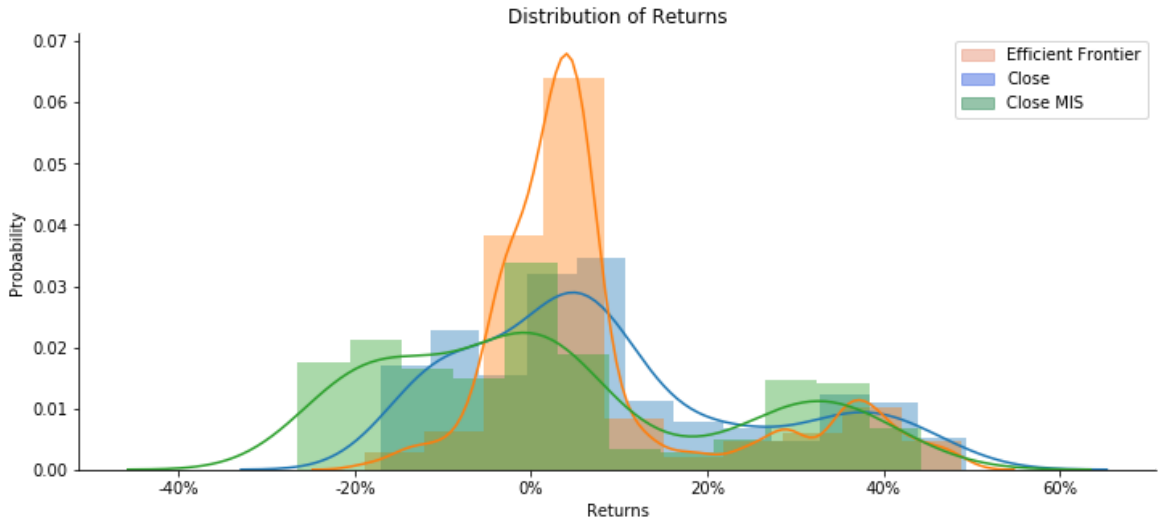
```
# worst day for the Hedgecraft MIS model
mis_max_loss = min(mis_daily_roi)

# avg amount of money lost by Hedgecraft when returns fell
below 0%
mean_loss = daily_roi[daily_roi < 0].mean()

# avg amount of money lost by EF when returns fell below 0%
ef_mean_loss = ef_daily_roi[ef_daily_roi < 0].mean()

# avg amount of money lost by Hedgecraft MIS when returns fell
below 0%
mis_mean_loss = mis_daily_roi[mis_daily_roi < 0].mean()

# Hedgecraft distribution of returns statistics
dist_stats = [
                prob_lose_money*100,
                max_loss,
                mean_loss,
                prob_below_rolling_avgs[0]*100,
                prob_below_rolling_avgs[1]*100,
                prob_below_rolling_avgs[2]*100
]

# rounds above list and annotates with % sign
dist_stats = [str(round(x, 2)) + '%' for x in dist_stats]

# Ef distribution of returns statistics
ef_dist_stats = [
                    ef_prob_lose_money*100,
                    ef_max_loss, ef_mean_loss,
                    ef_prob_below_rolling_avgs[0]*100,
                    ef_prob_below_rolling_avgs[1]*100,
                    ef_prob_below_rolling_avgs[2]*100
]

# rounds above list and annotates with % sign
ef_dist_stats = [str(round(x, 2)) + '%' for x in ef_dist_stats]

#  Hedgecraft MIS distribution of returns statistics
mis_dist_stats = [
                    mis_prob_lose_money*100,
                    mis_max_loss, mis_mean_loss,
                    mis_prob_below_rolling_avgs[0]*100,
                    mis_prob_below_rolling_avgs[1]*100,
                    mis_prob_below_rolling_avgs[2]*100
]

# rounds above list and annotates with % sign
mis_dist_stats = [str(round(x, 2)) + '%' for x in
mis_dist_stats]
```

```
# dictionary of distribution stats
dist_stats_summary = {
                         'Close': dist_stats,
                         'Close MIS': mis_dist_stats,
                         'Efficient Frontier': ef_dist_stats
}

# converts above dict to a DataFrame
dist_stats_summary = pd.DataFrame.from_dict(dist_stats_summary)

# renames the indices
dist_stats_summary = dist_stats_summary.rename(index={
                                  0:'probability of losing money
from initial investement',
                                  1:'maximum loss',
                                  2:'mean loss',
                                  3:'Probability of falling
bellow 30 day rolling avg',
                                  4:'Probability of falling
bellow 50 day rolling avg',
                                  5:'Probability of falling
bellow 90 day rolling avg'
                               }
);
```

Visualizing the Distribution of Returns

[278]
```
# function to plot many overlaping kde plots
def multi_distplot(rdist1, rdist2, rdist3, kde=True):

    # initializes figure and axis
    fig = plt.figure(figsize=(12,5))
    ax = fig.add_subplot(111)

    # pretty seaborn kde plots for each model
    sns.distplot(rdist1, bins=12, kde=bool)
    sns.distplot(rdist2, bins=10, kde=bool)
    sns.distplot(rdist3, bins=12, kde=bool)

    # gets xticks
    vals1 = ax.get_xticks()

    # reformats xticks to pcts
    ax.set_xticklabels(['{:.0f}%'.format(x) for x in vals1])

    # plot labels and title
```

```
    ax.set_ylabel('Probability')
    ax.set_xlabel('Returns')
    plt.title('Distribution of Returns')

    # removes spines
    sns.despine(top=True, right=True)

    # sets legend patches color and labels
    ef_patch = mpatches.Patch(color='darksalmon',
label='Efficient Frontier', alpha=0.5)
    patch = mpatches.Patch(color='royalblue', label='Close',
alpha=0.5)
    mis_patch = mpatches.Patch(color='seagreen', label='Close
MIS', alpha=0.5)

    # turns legend on with patches
    plt.legend(handles=[ef_patch, patch, mis_patch])
    #plt.legend(handles=[patch, mis_patch])

multi_distplot(rdist1 = daily_roi, rdist2 = ef_daily_roi,
rdist3 = mis_daily_roi)
#multi_distplot(rdist1 = daily_roi, rdist2 = None, rdist3 =
mis_daily_roi)
```



Above are the returns distribution for each portfolio: Efficient Frontier (in red), Close (in blue), and Close MIS (in green). The Efficient Frontier algorithm clearly produced a portfolio with a normal distribution of returns; the same can't be said of the Close portfolios.

It's important to emphasize that deviation-based measures of risk-adjusted performance implicitly assume the distribution of returns follows a normal distribution. As such, the Sharpe ratio isn't a suitable measure of performance since the standard deviation isn't a suitable measure of risk for the Close portfolios.

[281]
```
dist_stats_summary
```

	Close	Close MIS	Efficient Frontier
probability of losing money from initial investement	31.43%	50.86%	25.43%
maximum loss	-16.99%	-26.5%	-18.9%
mean loss	-8.37%	-12.51%	-4.37%
Probability of falling bellow 30 day rolling avg	33.43%	35.71%	42.57%
Probability of falling bellow 50 day rolling avg	27.71%	29.14%	39.71%

TODO: REVIEW

While Close had less pronounced maximum drawdowns it was more frequenty bellow 0% returns (1.59% of the time) than its MIS variant (0.53% of the time). These values dwarf that of the Efficient Frontier, which painfully experienced negative returns a third of the time. It's interesting to note that the maximum loss of the Close portoflios is an order of magnitude smaller than their maximum drawdowns.

This relationship is in contrast to the Efficient Frontier's maximum loss which is on the same order of magnitude as its maximum drawdown. It's also interesting to point out that Close has a lower probability of falling below its rolling 30, 50, and 90 averages than its MIS variant.

Taken together, Close's smaller average rolling maximum drawdown and smaller probabilities of falling below the above rolling averages indicate its growth is more consistent than its MIS variant. On the one hand, Close's *growth* is more consistent than its MIS variant while on the other hand, the MIS variant has a more consistent *growth rate*. Stated another way: Close's "velocity of returns" is more consistent than that of the MIS variant's, whereas Close MIS's "acceleration of returns" is more consistent than that of the Close portoflio.

Conclusion

In this notebook we built a novel algorithm for generating asset weights of a minimally correlated portoflio with tools from network science. Our approach is twofold: we first construct an asset correlation network with energy statistics (i.e., the distance correlation) and then extract the asset weights with a suitbale centrality measure. As an intermediate step we interpret the centrality score (in our case the communicability betweeness centrality) as a measure of relative risk as it quantifies the influence of each asset in the network. The communicability betweeness centrality is a particularly attractive choice since it captures the communicability of asset volatility. Recognizing the need for a human-in-the-middle variation of our proposed method, we modified the asset allocation algorithm to allow a user to pick assets subject to the constraints of the maximal

independent set. Both algorithms (Hedgecraft and Hedgecraft MIS, including the benchmark Efficient Frontier) were trained on a dataset of thirty-one daily historical stock prices from 2006-2014 and tested from 2015-2018. The portfolios were evaluated by cumulative returns, return rates, volatility, maximum drawdowns, risk-adjusted return metrics, and downside risk-adjusted performance metrics. **On all performance metrics, the Hedgecraft algorithm significantly outperformed both the portfolio generated by the Efficient Frontier and the market---passing our go/no go criteria.**

This cell has been deleted. Undo

```
[0]
```