Honors Thesis

Peter MacNeil*

January 2024

*Advised by Alex Iosevich

1 Introduction

In a world with an abundance of data, it has become apparent that only a fraction of it is useful. Much of the data we collect represents processes that have some flavor of inherent randomness. In industries where time series data is of utmost importance, such as the financial sector, meteorology, or in informing legislators who write economic policy, this abundance of data does not necessarily drive better forecasts. Too often it instead clutters the picture for data analysts looking to make informed conclusions. The time spent analyzing useless data would be much better spent investigating data we know we can draw conclusions from, and with increased and more effective data mining constantly being employed, the mass of this useless data is increasing exponentially. It is more important than ever to figure out how to separate the signal from the noise. We first must define what we mean by "useless" data. It must be made clear that by this phrasing we do not necessarily mean bad data, in that there were errors made in its collection. Rather, we mean data that we cannot draw significant conclusions from, with the definition of significance varying depending on what our goals are. Within the context of time-series forecasting, we define a useless time series as one that we can not consistently predict. Let us investigate a simple example. Suppose that we are playing a repeated game of heads and tails with a friend. We receive 1 dollar from our friend for every head, and we must pay your friend a dollar for every tail. It is easy to imagine that we can represent our winnings at each flip as time series data:

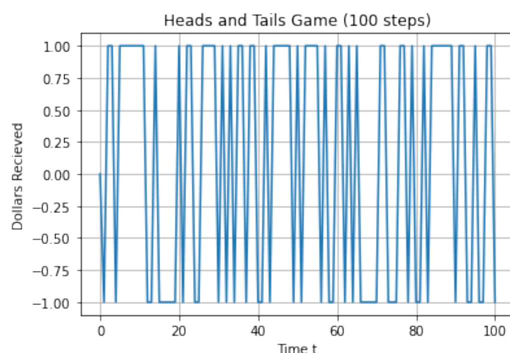

Figure 1: Coin Flips

The issue arises now when we try to forecast our future winnings. Any

forecasting technique would find some underlying pattern in our random data, and continue this pattern into the future, or give the best possible forecast which predicts our profits stabilizing indefinitely onwards. Still, both of these forecasts will fall short in any case any degree of accuracy is required.

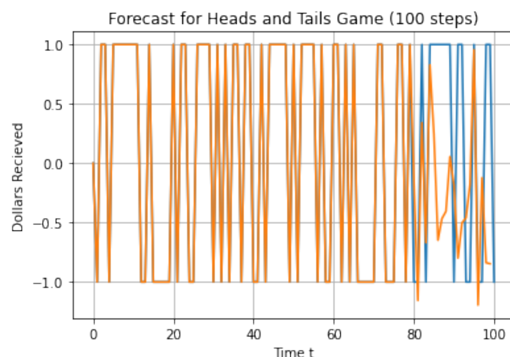

Figure 2: Trying to Forecast Coin Flips from past data (impossible)

This is even more obvious if we change our scheme to represent our profits over time from this game.

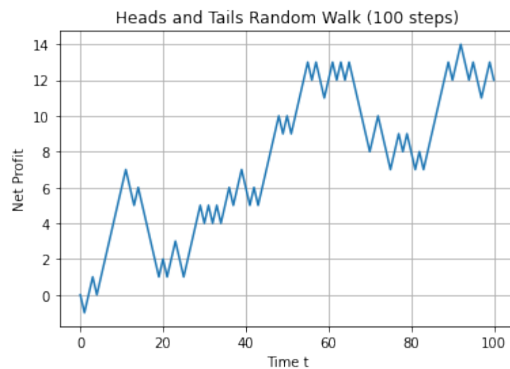

Figure 3: A Random Walk that happens to represent our profits over time

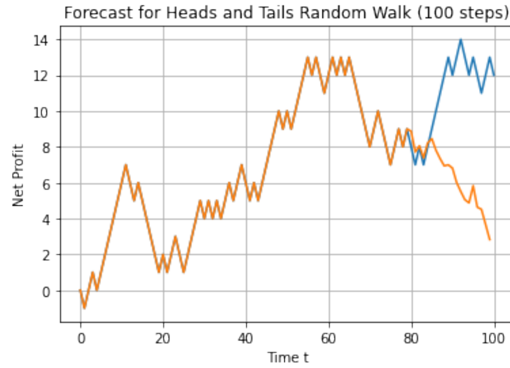

Figure 4: Forecast in Orange for our Random Walk

We can see that issues arise in forecasting when an underlying process is random. This is a trivial observation, but has more depth then one might think. This is because most things are not 100 percent random, or 100 percent deterministic, but somewhere in between. One may imagine a businesses total profit over time represented by a linear trend with random noise added to it. When asked to forecast this, our engine (roughly) succeeds at finding a trend, but struggles with the noise. Overall though, it is more successful with a task like this than the coin flip examples.

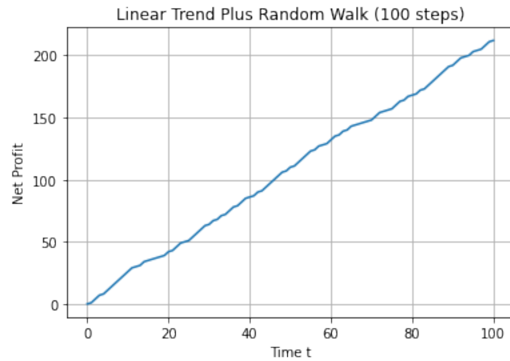

Figure 5: Linear Trend plus Random walk

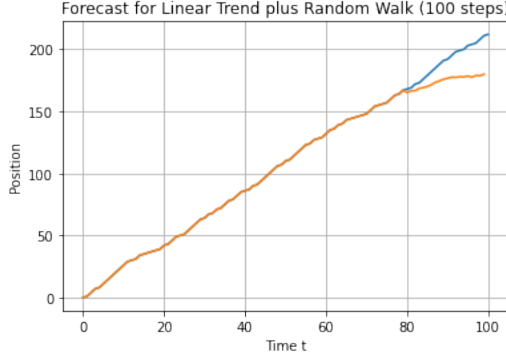

Figure 6: Forecast in Orange for our Linear trend plus Random walk

So how can we quantify how random something appears to be? We can't. But we can try. The following values, which we call $Q1$, $Q2$ and $Q3$, are currently employed in industry to approximate such a measure.

$$Q1 = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(f(i)-\mu)^2}}{\mu} \tag{1}$$

$$Q2 = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}|(f(i)-f(i-1)|-\mu_\Delta)^2}}{\mu_\Delta} \tag{2}$$

Where μ_Δ is defined as

$$\mu_\Delta = \frac{\sum_{i=1}^{N-1}|f(i)-f(i-1)|}{N-1} \tag{3}$$

and when S is the length of a season of interest (often the length of one year of time steps),

$$Q3 = \frac{\sqrt{\frac{1}{N}\sum_{i=S}^{N}|(f(i)-f(i-S)|-\mu_\Delta)^2}}{\mu_\Delta} \tag{4}$$

Where μ_Δ is defined as

$$\mu_\Delta = \frac{\sum_{i=S}^{N}|f(i)-f(i-S)|}{N-S} \tag{5}$$

These quantities are standard deviations of the time series, and differenced time series over their respective mean values. It is a natural quantity to come up with, as obviously those time series that differ from their average behavior often will be harder to predict.

2 Fractals and Neural Networks

Neural Networks are universal function approximaters. Given enough training data, a Neural Network can approximate all Borel Measurable functions arbitrarily close [1]. Special difficulties arise in time series forecasting, as we are predicting data beyond any point the model has trained. We model our value at time t as a function of the values coming before it.

$$V_t = f(V_{t-1}, V_{t-2}, ...V_{t-n}), n <= t \tag{6}$$

When forecasting using neural network architecture we must fix an n so we can have a fixed dimensionality of our training data. There are many measurable functions whose values match our time series at $t, t-1, .., t_0$ but have different values at $t+1, t+2, ...$ Those time series that are predictable are those that are close to one that our forecasting model is more biased towards.

A good analogy can be borrowed from real analysis. A function $f_1 : R \to R$ can be analytic at every point $x \in R$, with its respective power series expansion centered at any point having a radius of convergence equal to ∞. Another function $f_2 : R \to R$ is equal to f_1 at all $x \in R$ such that $x <= t$ for some $t > 0$. If we were to construct a forecasting engine around the idea of a Taylor expansion by first demanding it to calculate the best constant approximation for the function at 0, then the best quadratic expansion for the function at 0, then the best cubic and so on, then when we forecast f_1 past t, our forecasting engine will get us arbitrarily close to the true values of f_1, and while our engine will learn arbitrarily well the values of f_2 for $x <= t$, it will never approach f_2 past t. Our forecasting engine in this case is biased towards outputting analytic functions, and these are the functions that we can forecast indefinitely with it.

So what functions are neural networks biased to predict? Which ones are they not? A good starting place is investigating the functions that a feed forward neural network struggles to approximate. Let us introduce ourselves to a Cantor lattice:

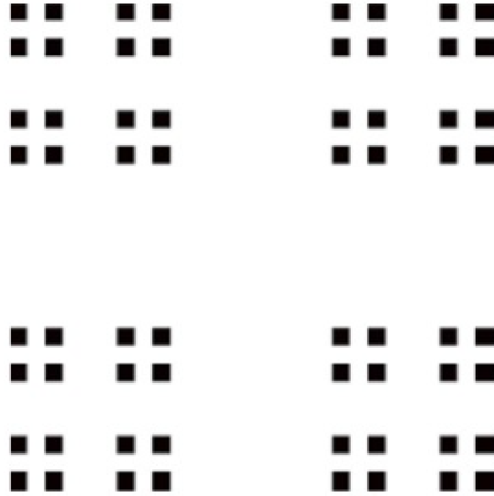

Figure 7: $\mathcal{C} \times \mathcal{C}$

Where $\mathcal{C}$ is the Cantor set, defined recursively by removing first the middle third of the unit interval, then the middle third of the remaining 2 intervals, then the middle third of the remaining 4, ad infinitum.

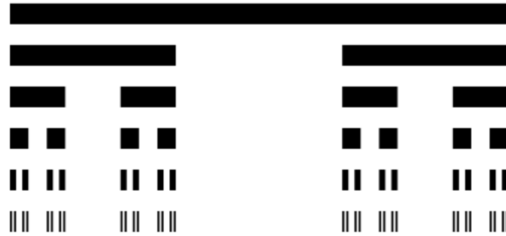

Figure 8: $\mathcal{C}^0, \mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3, \mathcal{C}^4, \mathcal{C}^5 ...$

Similar sets to $\mathcal{C}$ can be constructed using a recursive approach that involves removing sections of intervals, which we will still refer to as Cantor sets, thus we can construct more exotic Cantor Lattices as well. We denote these sets as $\mathcal{C}^k_{m,n}$, meaning at every step k, we divide the remaining intervals into n equal partitions, and then remove the same m partitions.

Theorem 1 *The Cantor set is uncountable.*

PROOF: The Cantor set is those numbers with 0 and 2 in their base 3 expansions. The theorem follows from a diagonalization argument.

Definition 1 *A Cantor lattice is a finite Cartesian product of Cantor-type sets:* $\mathcal{C}^k_{m_1,n_1} \times \mathcal{C}^k_{m_2,n_2} \times ... \mathcal{C}^k_{m_d,n_d}, k \in \mathbf{N} + \{\infty\}$

Theorem 2 *A Cantor lattice* $\mathcal{C}^d = \Pi_{i=0}^{d-1} \mathcal{C}_{m_i,2^d}$ *where* m_i *at the first iteration denotes the 2 intervals that start with* $.0$ *or* $.2^i \in [0,1]$ *in base* 2^d *has a bijection with [0,1]*

PROOF: For any $c = (c_0, c_1, ...c_{d-1}) \in \mathcal{C}^d, \sum_{i=0}^{d-1} c_i$ denotes a unique point $\in [0,1]$. Since every point in [0,1] has a base 2^d representation, the image is full.

This bijection is very easy for a neural network to learn taking $\mathcal{C}^d- > [0,1]$, but its inverse is very interesting. It is easy for us to work out on paper the $c \in \mathcal{C}^d$ corresponding to a point $x \in \mathbf{R}$, but can a FFNN?

We construct a python function that can generate 4 steps of cantor latices made up of d cantor types sets as in theorem 2, for $d \in 2, 3, 4, 5$ (total of 2^4 points in every cantor set that is then used to create the lattice. Our neural network will take as input $\sum_{i=0}^{d-1} c_i$ for some $c \in \mathcal{C}^d$ and output the d dimensional vector c. Our network will have 2 hidden layers of 100 and 50 neurons, and run 10 epochs of the full training set. After training we can see a clear pattern of increased MSE (Mean-Squared Error) with increased lattice dimension.

Cantor Lattice Dimension (d)	MSE after training on f: [0,1] -> C^d
2	0.187477537
3	0.205172178
4	0.218151553
5	0.225532654

Figure 9: MSE for each dimension d

Our Neural Network struggles learning fractal behavior embedded in higher dimensions, despite the underlying function being the same.

3 Discrete S-Energy

In this section we will introduce the reader to the idea of Discrete S-Energy developed in [2], and show how it is related to our cantor lattice experimentation. This will motivate its use in time series forecasting.

Let $P_n \in [0,1]^d, d >= 2$ be a finite point set of size n. For $s \in [0,d]$ we define the Discrete s-Energy I_s of P_n as follows:

Definition 2 *Discrete S-Energy*

$$I_s(P_n) = n^{-2} \sum_{p \neq q : p,q \in P_n} |p-q|^{-s} \tag{7}$$

Definition 3 *Discrete Hausdorff Dimension of point set P*

$$dim_H(P) = sup\{s \in [0,d] : sup_n I_s(P_n)\} < \infty \tag{8}$$

Where

$$P = \{P_n\} \tag{9}$$

Theorem 3 *Let $d \in Z$ Let $C^k_{m_1,n_1}, ..., C^k_{m_d,n_d}$ be discrete Cantor sets. Set $A_k = \Pi^d_{i=1} C^k_{m_i,n_i}$ then*

$$dim_H(A) = \sum_{i=1}^{d} \frac{ln(m_i)}{ln(n_i)} \tag{10}$$

From this theorem, we can see that the Hausdorff dimensionality of our Cantor data sets in Section 2 is increasing, which implies that for a fixed $s = 1$, We are seeing more error for smaller Discrete 1-Energy. This makes sense as we often say Discrete 1-Energy is a measure of "how 1 dimensional" a data set is.

This is motivation for a fractal perspective in estimating time series predictability.

Going back to our random walks, lets introduce a parameter $p \in [0,1]$ so that every step of the way, with probability p, we take one step forward (plus one difference) and with probability 1-p, we leave our next move up to 50/50 chance, either still taking one step forwards, or one step back. Let us see how these graphs vary in p:

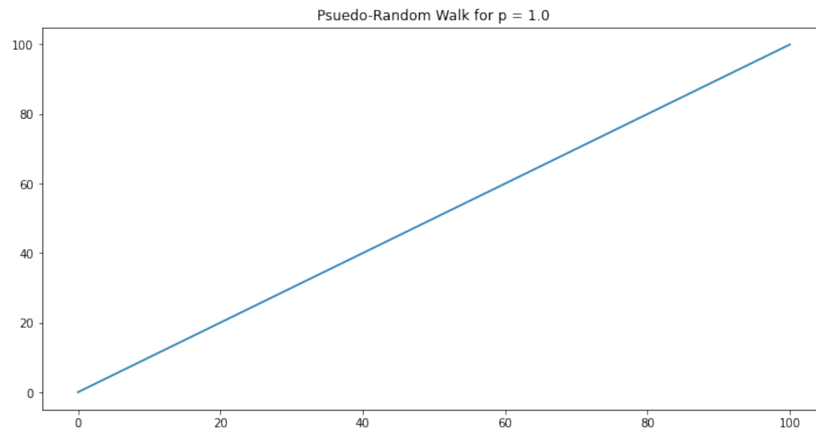

Figure 10: p=1

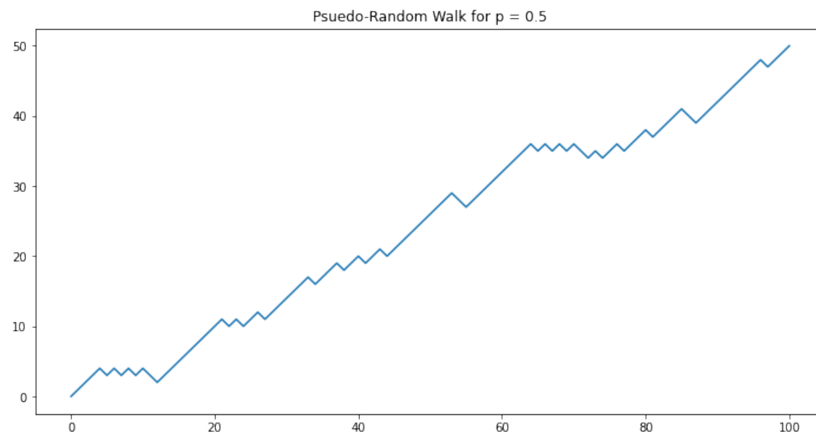

Figure 11: p=.5

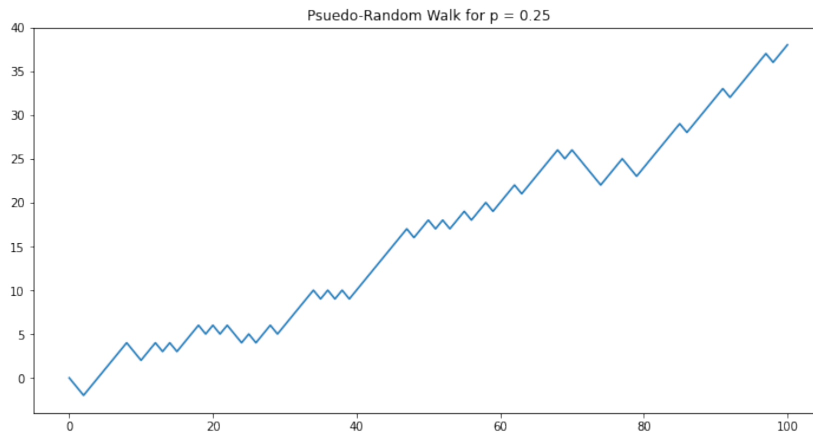

Figure 12: p=.25

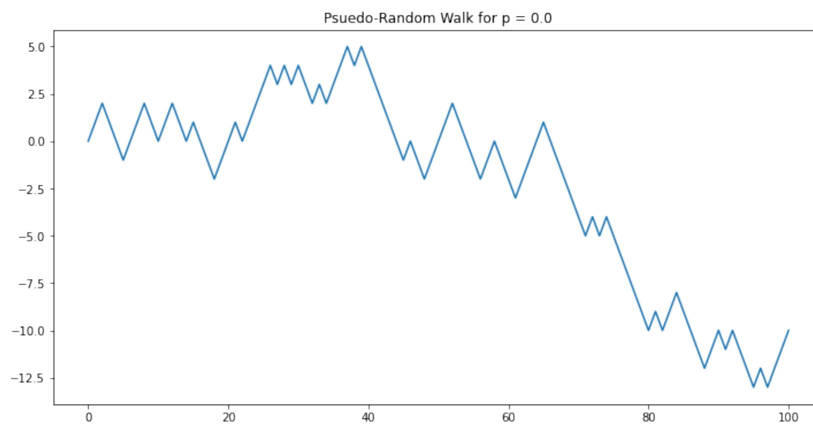

Figure 13: p=0

We can see that whatever determinism the time series has is lessened with decreasing p. We will see how this relates to forecastibility ;). Now when we apply a Discrete 1-Energy calculation, we can see that Discrete 1-Energy seems to be negatively correlated with random behavior:

```
['For p = 1.0 Discrete 1 Energy is 2.93853870826786',
'For p = 0.75 Discrete 1 Energy is 2.8438406512683163
'For p = 0.5 Discrete 1 Energy is 2.686350371733846',
'For p = 0.25 Discrete 1 Energy is 2.601679630539898'
'For p = 0.0 Discrete 1 Energy is 2.1160593227961235'
```

Figure 14: 1-Energy Calculations for Psuedo-Random Walks

4 Experimental Design

Our goal is to be able to take a time series, and calculate an indicator from the data that tells us whether or not it is worth it to forecast our data. This is a lofty goal, but a first step towards it is the following experiment.

4.1 Experiment 1

We construct 51 Psuedo-Random Walks of 100 steps with p parameters equally spaced in [0,1]. We then use a Feed Forward Neural Network Architecture with input dimension 20, one hidden layer of 10 neurons equipped with ReLu activations, and one output, to train on the first 80 steps of our time seris with lookback = 20, to then predict the last 20 steps of our time series. We will then calculate the errors (RMSE/SD), Discrete 1-Energy of the training set, and the Minimum over the industry standard values over the training set, and conduct an OLS regression statistical analysis to assess the signifigance of correlations between our 2 values and error.

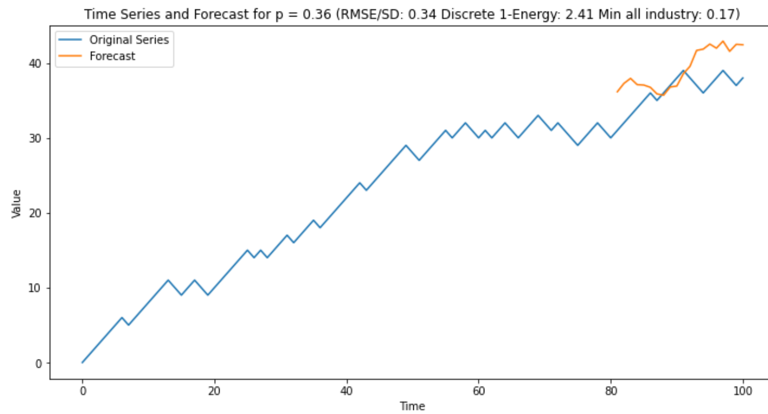

Figure 15: Example Psuedo-Random Walk and its respective forecast

After running statistical analysis, we can see how the two measurements compare.

OLS Regression Results

Dep. Variable:	RMSE/SD	R-squared:	0.413
Model:	OLS	Adj. R-squared:	0.401
Method:	Least Squares	F-statistic:	34.53
Date:	Wed, 10 Apr 2024	Prob (F-statistic):	3.63e-07
Time:	01:14:07	Log-Likelihood:	-30.997
No. Observations:	51	AIC:	65.99
Df Residuals:	49	BIC:	69.86
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	4.0889	0.572	7.147	0.000	2.939	5.239
Discrete 1-Energy	-1.3696	0.233	-5.876	0.000	-1.838	-0.901

Figure 16: OLS Regression results, Discrete 1-Energy values on RMSE/SD

```
[1] Standard Errors assume that the covariance matrix of the errors is correctly specifi
                            OLS Regression Results
==============================================================================
Dep. Variable:                RMSE/SD   R-squared:                       0.372
Model:                            OLS   Adj. R-squared:                  0.359
Method:                 Least Squares   F-statistic:                     29.02
Date:                Wed, 10 Apr 2024   Prob (F-statistic):           2.02e-06
Time:                        01:14:07   Log-Likelihood:                -32.737
No. Observations:                  51   AIC:                             69.47
Df Residuals:                      49   BIC:                             73.34
Df Model:                           1
Covariance Type:            nonrobust
=====================================================================================
                        coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------------
const                 0.4427      0.087      5.106      0.000       0.268       0.617
Min all Industry      1.8064      0.335      5.387      0.000       1.133       2.480
==============================================================================
```

Figure 17: OLS Regression results, Minimum of industry values on RMSE/SD

We can see that while both values are extremely significantly correlated with error, Discrete 1-Energy is slightly more significantly ($|t| = 5.876$ vs $|t| = 5.387$). While this difference isn't huge, it is promising that Discrete 1-Energy may have implications in forecastibility of a time series.

4.2 Experiment 2: Real World Data

We have been given 998 time series detailing sales of various products from a large retail store in CSV file format. Each time series is the sales data from specific product, sold at a specific location. The products are not unique, as many are sold at multiple locations. We are also given the state and a location id of the store from which each of these time series is from, as well as the department that sells it. The time series all stretch from the 5th week of 2015 to the 3rd week of 2017, with a data point for every week detailing the volume of sales. The important part of all this is that each time series is relatively similar. All the time series are from the same store, represents a select set of products, from a select set of locations, and are over the same time span. This gives us a good set of data for us to investigate.

	Unnamed: 0	product_id	location_id	state	department	_201405	_201406	_201407	_201408	_201409	...	_201646	_201647	_201648	_201649	_201650	_201651	_201652	_201701	_201702	_201703
0	0	7	516	NY	Clinique	0	0.00	0.00	0.00	0.00	...	192	180	564	0	0	0	0	0	0	0
1	1	10	353	NY	Kitchen Electrics	1106	287.04	1334.88	692.16	406.00	...	0	0	0	0	0	0	0	0	0	0
2	2	10	516	NY	Kitchen Electrics	70	33.28	75.60	56.00	85.84	...	0	0	0	0	0	0	0	0	0	0
3	3	14	8	NJ	Clinique	26	45.76	86.40	38.08	238.96	...	0	0	0	0	0	0	0	0	0	0
4	4	14	10	NY	Clinique	800	468.00	503.28	665.28	450.08	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
993	995	1796	516	NY	Clinique	0	0.00	0.00	0.00	0.00	...	984	492	1728	1992	1032	804	864	564	696	396
994	996	1803	353	NY	Clinique	144	128.96	116.64	176.96	113.68	...	0	0	0	0	0	0	0	0	0	0
995	997	1810	516	NY	Kitchen Electrics	0	0.00	0.00	0.00	0.00	...	0	0	0	0	0	0	0	0	0	0
996	998	1811	516	NY	Clinique	0	0.00	0.00	0.00	0.00	...	0	0	0	0	0	0	0	0	0	0
997	999	1821	516	NY	Kitchen Electrics	0	0.00	0.00	0.00	0.00	...	3096	3048	4896	5856	4668	3060	2172	588	1116	5664

998 rows × 160 columns

Figure 18: Preview of our dataset in Pandas dataframe

To clean our data, We load the data into a Pandas dataframe in Python. We then drop the columns corresponding to the time series number (Unnamed column), the *product_id*, *location_id*, *state*, and *department* of sale.

In order to conduct a sound experiment, we must unfortunately get rid of some of our time series. Many of the time series end in a string of 0's, indicating a pause or discontinuation of sales of a product at a specific location.

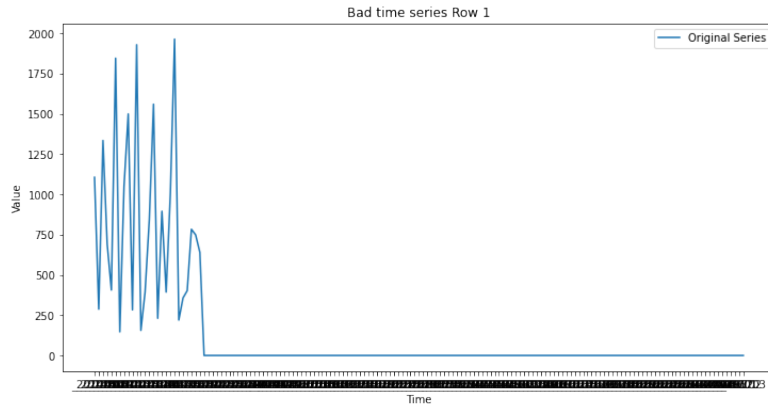

Figure 19: Bad Data

These time series are pointless to forecast. We cannot just cut off the time series at the point of discontinuation for use in our experiment as we would like to only compare time series of the same length. For these reasons we need to drop them from our data set.

Additionally, there are many time series in our dataset that start with a long string of zero sales.

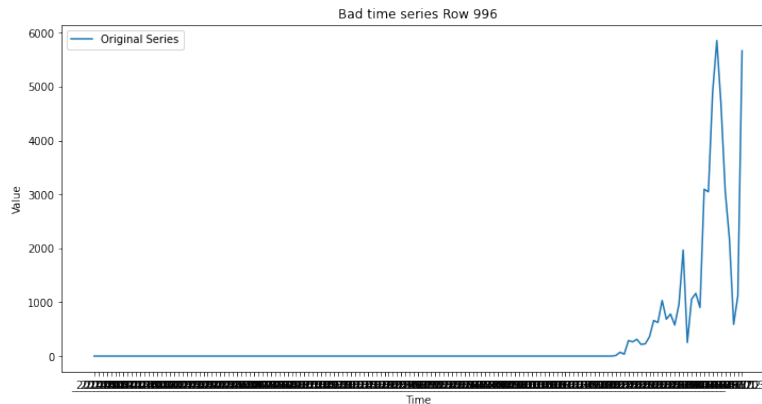

Figure 20: Bad Data

Similarly, it wouldn't be great to include these in our experiment, as our forecasting engine would receive less training data that is of interest to us. We must also drop these.

We do this by dropping the rows of our dataframe that start or end with a zero. After this process we are left with 144 usable time series.

We are now able to begin the bulk of our experimentation. The general outline of this experiment is as follows. Firstly, we will take each of our time series and utilize a forecasting engine to predict the tail end of it. Secondly, we will compare the values predicted by our forecasting engine with the true value of the time series at that time. Third, we will calculate our industry standard measurements and that of Discrete 1-Energy. Lastly we will investigate any correlations between our measurements and the error we observe from each time series

Every time series will be split into a training and test set. The training set will represent 80 percent of the available data points of each time series. This ends up being the sales from the first 124 weeks of our time period, with the remaining 31 making up the data that we will test our forecasts with.

Our forecasting model is an LSTM (Long Short Term Memory) Neural Network model. LSTMs are part of a class of RNNs (Recurrent neural networks), with not only directed edges forward to other neurons, but also directed edges between neurons within the same hidden layer, and even going backwards. The reason for this choice is because (a) It is still a relatively simple model, so it isn't a blackbox in our understanding, and (b) It can handle long term dependencies. LSTMs possess a memory cell, that lets the

model use previous observations to influence its output. This makes it advantageous for time series forecasting, especially within contexts where values at previous time steps may affect the present, such as ours.

The architecture of the model is a look back and therefore input dimension of 16, 2 hidden layers with 50 neurons each, and one output representing our sales at the next time step.

Along with the previous industry standards introduced, Q1, Q2, and Q3:

$$Q1 = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(f(i)-\mu)^2}}{\mu} \tag{11}$$

$$Q2 = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}|(f(i)-f(i-1)|-\mu_\Delta)^2}}{\mu_\Delta} \tag{12}$$

Where μ_Δ is defined as

$$\mu_\Delta = \frac{\sum_{i=1}^{N-1}|f(i)-f(i-1)|}{N-1} \tag{13}$$

and when S is 52, representing the 52 weeks in a year

$$Q3 = \frac{\sqrt{\frac{1}{N}\sum_{i=S}^{N}|(f(i)-f(i-S)|-\mu_\Delta)^2}}{\mu_\Delta} \tag{14}$$

Where μ_Δ is defined as

$$\mu_\Delta = \frac{\sum_{i=S}^{N}|f(i)-f(i-S)|}{N-S} \tag{15}$$

We will also test alternative versions:

$$altQ2 = |\frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(f(i)-f(i-1)-\mu_\Delta)^2}}{\mu_\Delta}| \tag{16}$$

Where μ_Δ is defined as

$$\mu_\Delta = \frac{\sum_{i=1}^{N-1}(f(i)-f(i-1))}{N-1} \tag{17}$$

and when S is 52, representing the 52 weeks in a year

$$altQ3 = |\frac{\sqrt{\frac{1}{N}\sum_{i=S}^{N}(f(i)-f(i-S)-\mu_\Delta)^2}}{\mu_\Delta}| \tag{18}$$

Where μ_Δ is defined as

$$\mu_\Delta = \frac{\sum_{i=S}^{N} (f(i) - f(i-S))}{N-S} \tag{19}$$

After training the LSTM model and producing a forecast for each of the 144 timeseries, we are able to run our statistical analysis.

OLS Regression Results

Dep. Variable:	RMSE/SD	R-squared:	0.035
Model:	OLS	Adj. R-squared:	0.028
Method:	Least Squares	F-statistic:	5.110
Date:	Tue, 02 Apr 2024	Prob (F-statistic):	0.0253
Time:	00:41:13	Log-Likelihood:	-89.166
No. Observations:	144	AIC:	182.3
Df Residuals:	142	BIC:	188.3
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.8006	0.192	4.170	0.000	0.421	1.180
Discrete 1-Energy	0.1537	0.068	2.261	0.025	0.019	0.288

Figure 21: OLS Regression, Discrete 1-Energy on RMSE/SD

We can see that the coefficient on Discrete 1-Energy has switched directions, which is unexpected. Even stranger is that it has remained significant at the 95% level ($|t| = 2.261$).

OLS Regression Results

Dep. Variable:	RMSE/SD	R-squared:	0.205
Model:	OLS	Adj. R-squared:	0.200
Method:	Least Squares	F-statistic:	36.69
Date:	Tue, 02 Apr 2024	Prob (F-statistic):	1.18e-08
Time:	00:41:13	Log-Likelihood:	-75.166
No. Observations:	144	AIC:	154.3
Df Residuals:	142	BIC:	160.3
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.8912	0.065	13.705	0.000	0.763	1.020
Min all Industry	0.2493	0.041	6.057	0.000	0.168	0.331

Figure 22: OLS Regression, Min Q1, Q2, Q3, altQ2, altQ3 on RMSE/SD

We can see that the performance of the industry standards has remained very significant ($|t| = 13.705$). This experiment suggests the industry standards may be better suited for assessing the predictibility of these types of

time series. But why is Discrete 1-Energy behaving so significantly different than in our first experiment? Lets look at some forecasts.

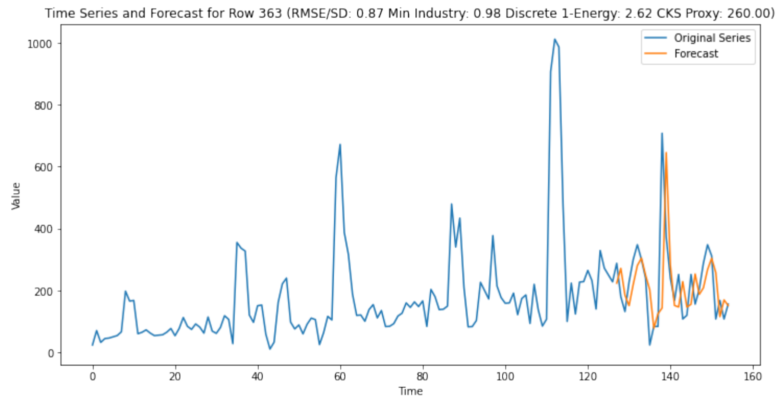

Figure 23: Good Forecast, Low 1-Energy

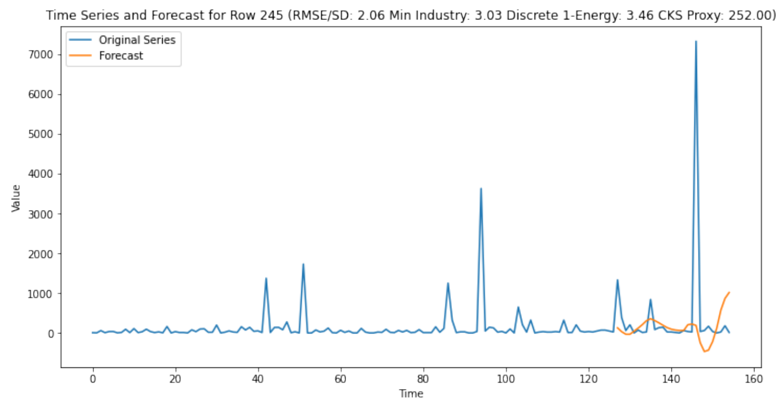

Figure 24: Bad Forecast, High 1-Energy

As we can see, the lower 1-Energy time series is much more oscillatory, which is as expected as we can think of 1-Energy as a measure of "How one-dimensional is this data set", as the Discrete 1-Energy sum diverges for a line set . The higher 1-Energy times series is much more constant, with a few relatively large spikes. The oscillatory nature of Figure 22, coupled with the ease of forecastibility suggests a strong seasonality within the time series. It is possible that the Seasonality of a time series is able to over power the predictive power of 1-Energy.

4.3 Experiment 3: Discrete 1-Energy without Seasonality

Since we believe that seasonality may be overpowering 1-Energy, we should calculate 1-Energy with seasonalities removed. To do this, we apply a 52 week difference operator to the training set that we calculate our predictibility metrics from, and then calculate 1-Energy from this differenced time series. We then forecast exactly how we did in Experiment 2.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 RMSE/SD   R-squared:                       0.065
Model:                             OLS   Adj. R-squared:                  0.058
Method:                  Least Squares   F-statistic:                     9.854
Date:                 Tue, 02 Apr 2024   Prob (F-statistic):            0.00206
Time:                         01:25:59   Log-Likelihood:                -91.286
No. Observations:                  144   AIC:                             186.6
Df Residuals:                      142   BIC:                             192.5
Df Model:                            1
Covariance Type:             nonrobust
========================================================================================================
                                          coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------------------------------
const                                   0.5884      0.209      2.820      0.005       0.176       1.001
Discrete 1-Energy without Seasonality   0.2735      0.087      3.139      0.002       0.101       0.446
```

Figure 25: OLS 1-Energy with seasonality removed on RMSE/SD

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 RMSE/SD   R-squared:                       0.197
Model:                             OLS   Adj. R-squared:                  0.191
Method:                  Least Squares   F-statistic:                     34.78
Date:                 Tue, 02 Apr 2024   Prob (F-statistic):           2.58e-08
Time:                         01:25:59   Log-Likelihood:                -80.343
No. Observations:                  144   AIC:                             164.7
Df Residuals:                      142   BIC:                             170.6
Df Model:                            1
Covariance Type:             nonrobust
====================================================================================
                       coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------------
const                0.8941      0.067     13.265      0.000       0.761       1.027
Min all Industry     0.2516      0.043      5.897      0.000       0.167       0.336
```

Figure 26: OLS Min all industry on RMSE/SD

We can see that removing yearly seasonality from the Discrete 1-Energy Calculation lowered the significance of our coefficient a bit. The coefficient remains to have the opposite sign of what is expected, and remains significant at the 95% level.

4.4 Experiment 4: Shorter Time Series

To see if we are asking too much of our forecasting model, we adapt it for shorter time series. To do this we change our data cleaning approach that was discussed in experiment 2. Before applying the restrictions on leading and trailing 0s, -indicating an abnormality in the sales of our time series, we first shorten the 998 time series to just the last 10 weeks. We then drop those series that have a leading or trailing 0 in this 10 week time span. After this process we are left with 299 usable time series. We then apply the same processes as in experiment 2. The LSTM model has to be changed slightly to fit these smaller time series. We shorten the look-back to 3 weeks, so our input dimension is 3. The two hidden layers have 10 and 3 neurons respectively, and then we have the same singular output. We also adjust the batch size to be 2, to handle the shorter time series. We also reduce the amount of training epochs to be 10.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                RMSE/SD   R-squared:                       0.037
Model:                            OLS   Adj. R-squared:                  0.034
Method:                 Least Squares   F-statistic:                     11.56
Date:                Sat, 13 Apr 2024   Prob (F-statistic):           0.000764
Time:                        16:19:18   Log-Likelihood:                -160.19
No. Observations:                 299   AIC:                             324.4
Df Residuals:                     297   BIC:                             331.8
Df Model:                           1
Covariance Type:            nonrobust
=====================================================================================
                        coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------------
const                 1.1474      0.215      5.327      0.000       0.724       1.571
Discrete 1-Energy    -0.7531      0.221     -3.401      0.001      -1.189      -0.317
```

Figure 27: OLS Discrete 1-Energy on RMSE/SD, Short Time Series

OLS Regression Results

Dep. Variable:	RMSE/SD	R-squared:	0.061
Model:	OLS	Adj. R-squared:	0.058
Method:	Least Squares	F-statistic:	19.30
Date:	Sat, 13 Apr 2024	Prob (F-statistic):	1.55e-05
Time:	16:19:18	Log-Likelihood:	-156.49
No. Observations:	299	AIC:	317.0
Df Residuals:	297	BIC:	324.4
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.5951	0.046	12.806	0.000	0.504	0.687
Min all Industry	-0.2170	0.049	-4.393	0.000	-0.314	-0.120

Figure 28: OLS Min all industry on RMSE/SD, Short Time Series

This experiment yields very interesting results. 1-Energy is now behaving as expected, with significance at the 95% level. Even stranger is the sign of the coefficient on Min all industry has switched away from what is expected, and has remained significant at the 95% level.

4.5 Experiment 5: Aggregation

Due to the fact that our data is very sparse, it might be advantageous to aggregate it to supply our forecasting model with better training data. To do this we go back to our data cleaning stage. Instead of dropping the columns that correspond to the time series number (Unnamed column), the *product_id*, *location_id*, *state*, and *department* of sale, we drop all but the *product_id*. We then sum our time series with matching *product_id*, and are left with 84 time series. There is no need to drop any now since there are none that start or end with 0 sales. We then forecast as we have done before, except reduce our batch size to 8, since the aggregation will smooth gradients.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                RMSE/SD   R-squared:                       0.001
Model:                            OLS   Adj. R-squared:                 -0.011
Method:                 Least Squares   F-statistic:                    0.1034
Date:                Sat, 13 Apr 2024   Prob (F-statistic):              0.749
Time:                        16:35:33   Log-Likelihood:                -40.785
No. Observations:                  84   AIC:                             85.57
Df Residuals:                      82   BIC:                             90.43
Df Model:                           1
Covariance Type:            nonrobust
=====================================================================================
                        coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------------
const                 1.0146      0.260      3.910      0.000       0.498       1.531
Discrete 1-Energy     0.0288      0.090      0.322      0.749      -0.149       0.207
```

Figure 29: OLS Discrete 1-Energy on RMSE/SD, Aggregated Time Series

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                RMSE/SD   R-squared:                       0.065
Model:                            OLS   Adj. R-squared:                  0.053
Method:                 Least Squares   F-statistic:                     5.670
Date:                Sat, 13 Apr 2024   Prob (F-statistic):             0.0196
Time:                        16:35:33   Log-Likelihood:                -38.030
No. Observations:                  84   AIC:                             80.06
Df Residuals:                      82   BIC:                             84.92
Df Model:                           1
Covariance Type:            nonrobust
====================================================================================
                       coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------------
const                0.9202      0.085     10.791      0.000       0.751       1.090
Min all Industry     0.1465      0.062      2.381      0.020       0.024       0.269
```

Figure 30: OLS Min all industry on RMSE/SD, Aggregated Time Series

As we can see from the regressions, we did not find a significant relationship between 1-Energy and error in this experiment. The minimum over the industry standards remains positively correlated with error at the 95% significance level.

5 Discussion and Conclusion

The findings from our experiments are very interesting. While we can not draw any specific conclusions from the results which may have been more desirable, we can say that Discrete 1-Energy seems to play some sort of role in the forecastibility of time series. This is evident in the multiple experiments

that found above 95% significance levels on the coefficient of Discrete 1-Energy. What is strange is the switching sign on this coefficient, as well as the coefficient on the Industry Standard. This implies that there may be something fundamentally different between the full length time series and the shortened one of 10 weeks.

A source of possible issues in our experimentation may be the data itself. The sales data, even when aggregating is very sparse, and seemingly very random. Similar experimentation may have to be done on richer data sets to draw more meaningful conclusions.

It also has to be said that our LSTM forecasting models are not perfectly calibrated for this application. Adjusting hyperparemeters is a tricky business, and we may have extremely different findings with appropriate adjustments to these values. It will also be fruitful to perform similar experimentation with other forecasting models, including FFNN, RNN, and facebook prophet neural network architechtures, along with more traditional statistical forecasting methodologies.

It seems like there is room for future work in designing a family of synthetic functions/time series for which increased Discrete 1-Energy is correlated with higher error, unlike the synthetic data that we investigated earlier in this paper. Additionally, doing the same for the minimum industry standards -finding a family of synthetic data for which the min over the industry standard measures is negatively correlated with error might give us foresight into what contexts the use of these standards is useful and when it is not. This might make it clearer when an alternative perspective, such as the fractal perspective with Discrete 1-Energy may be more useful in assessing forecastability.

At this point the industry standards behave more as expected, but these experiments show that they do not capture the full picture of forecastibility, if that is even possible, and that a fractal perspective may be useful for forecasters to possess.

6 Code Appendix

All of the code is written in Python within the context of a .ipynb file, so sometimes output statements will lack a print statement.

6.1 Discrete S-Energy

Awesome implementation with help from Svet

```
import numpy as np
import numpy.ma as ma
def DiscreteSEnergy(P, s):
    n = np.shape(P)[0]
    [Rows, Columns] = np.indices((n,n))

    VecDiff = np.abs(P[Rows,:] - P[Columns,:])
    MagDiff = np.power(np.sum(np.power(VecDiff,2), axis=2),1/2)
    #here we use L2 norm (Euclidean distance in Rd) to find vector magnitude
    Diff= np.triu(MagDiff)
    #remove duplicate differences by considering only those differences above
    the main diagonal
    MaskedDiff = ma.masked_where(Diff==0, Diff, copy=True)
    #mask all zero differences so we are not dividing by zero

    SumMat = np.ma.power(MaskedDiff, -s)
    sum = np.sum(np.sum(SumMat))

    discSEnergy = np.power(float(n), -2)*sum
    return discSEnergy
```

6.2 Psuedo-Random Walks

```
import random
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
#Psuedorandom Walks to test discrete 1 energy on
def PsuedoRandomWalks(Parray):
    allPRW = np.arange(101)
    for p in(Parray):
        PRW = [0]
        for i in range(100):
            q = random.random()
```

```
            if (q < p):
                PRW.append(PRW[i] + 1)
            else:
                r = random.random()
                if(r<.5):
                    PRW.append(PRW[i]+1)
                else:
                    PRW.append(PRW[i]-1)

        allPRW = np.vstack((allPRW, PRW))
    return allPRW

PsuedoRandomWalk = PsuedoRandomWalks([.75, .5, .25, 0])

for i in range(PsuedoRandomWalk.shape[0]):
    p= 1-(.25*i)
    # Plotting
    plt.figure(figsize=(12, 6))
    plt.title(f"Psuedo-Random Walk for p = {p}")
    plt.plot(PsuedoRandomWalk[i])



    import numpy.ma as ma
def DiscreteSEnergy(P, s):
    n = np.shape(P)[0]
    [Rows, Columns] = np.indices((n,n))

    VecDiff = np.abs(P[Rows,:] - P[Columns,:])
    MagDiff = np.power(np.sum(np.power(VecDiff,2), axis=2),1/2)
    #here we use L2 norm (Euclidean distance in Rd)
    to find vector magnitude
    Diff= np.triu(MagDiff) #remove duplicate differences by
    considering only those differences above the main diagonal
    MaskedDiff = ma.masked_where(Diff==0, Diff, copy=True) #mask all zero
    differences so we are not dividing by zero

    SumMat = np.ma.power(MaskedDiff, -s)
```

```
    sum = np.sum(np.sum(SumMat))

    discSEnergy = np.power(float(n), -2)*sum
    return discSEnergy

# Turn 1 dim y value data to 2 dim x,y data
def prepforsenergy(timeSeries):
    # Convert timeSeries to NumPy array if it’s not
    if type(timeSeries) != np.ndarray:
        timeSeries = np.array(timeSeries)

    xVal = np.arange(len(timeSeries))

    # Normalize xVal to map to [0, 1]
    xVal_normalized = xVal / max(xVal)

    # Normalize timeSeries to map to [0, 1]
    timeSeries_normalized = (timeSeries - min(timeSeries)) / (max(timeSeries) - mi

    # Create a 2D array
    twoDimTimeSeries = np.column_stack((xVal_normalized, timeSeries_normalized))

    return twoDimTimeSeries

OneEnergies = []

for i in range(PsuedoRandomWalk.shape[0]):
    p= 1-(.25*i)
    TwoDimPRW = prepforsenergy(PsuedoRandomWalk[i])
    OneEnergies.append(f"For p = {p} Discrete 1 Energy is {DiscreteSEnergy(TwoDimP

OneEnergies
```

6.3 Experiment 1: Synthetic Data

```
import pandas as pd
import numpy as np
```

```
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

def forecast_next_values(timeSeries, k, p):
    """
    This function trains a feedforward neural network on a sequence of numbers to

    Args:
    numbers (list): List of numbers forming a time series.
    k (int): The number of future values to predict.

    Returns:
    forecasted_values (list): The forecasted future 'k' values.
    """

    numbers = timeSeries[:80]
    timeSeriestest = np.array(timeSeries[81:])

    # Set the random seeds for reproducibility
    np.random.seed(0)
    tf.random.set_seed(0)

    # Prepare the data for the feedforward network input
    X, y = [], []
    for i in range(len(numbers) - k):
        X.append(numbers[i:i+k])
        y.append(numbers[i+k])

    # Convert the data to numpy arrays
    X = np.array(X)
    y = np.array(y)

    # Create the feedforward model
    model = Sequential()
```

```
model.add(Dense(10, input_dim=k, activation='relu'))  # Hidden layer with 10 n
model.add(Dense(1))  # Output layer

# Compile the model with MSE as the loss function and Adam optimizer
model.compile(loss='mean_squared_error', optimizer='adam')

# Train the model
model.fit(X, y, epochs=50, batch_size=20, verbose=0)

# Use the model to forecast the next k values
last_k_values = numbers[-k:]  # Get the last 'k' observed values
forecasted_values = []
for _ in range(k):
    input_data = np.reshape(last_k_values, (1, k))  # Reshape the data to fit
    next_value = model.predict(input_data)  # Use the model to forecast the ne
    numbers.append(next_value[0, 0])  # Append the forecasted value to the lis
    last_k_values = np.append(last_k_values[1:], next_value[0, 0])  # Update t

# Calculate RMSE on normalized data
rmse = np.sqrt(mean_squared_error(timeSeriestest, numbers[80:]))
sd =  np.std(timeSeries)
error = rmse/sd
# calculate metrics

metricArray = obtainMetrics(timeSeries[:80], error)

 #Plotting
plt.figure(figsize=(12, 6))
plt.title(f"Time Series and Forecast for p = {p} (RMSE/SD: {error:.2f} Discret
plt.xlabel("Time")
plt.ylabel("Value")

    # Plotting the original series
plt.plot(timeSeries, label='Original Series')

# Plotting the forecasted part
forecast_index = np.arange(81, len(timeSeries))
plt.plot(forecast_index, numbers[80:], label='Forecast')
```

```
    plt.legend()
    plt.show()




    return metricArray
```

6.4 Experiment 2: Real World Data 1

6.4.1 Data Cleaning

```
    df = pd.read_csv("cleansalesdata.csv")

columns_to_drop = ['Unnamed: 0', 'product_id', 'location_id', 'state', 'department

df.drop(columns=columns_to_drop, axis=1, inplace=True)
df


def drop_rows_with_zero_first_last(df):
    # Drop rows where the first or last entry is 0
    df = df[(df.iloc[:, 0] != 0) & (df.iloc[:, -1] != 0)]
    return df
df = drop_rows_with_zero_first_last(df)
```

6.4.2 Predictability Metrics

```
    def obtainMetrics(timeSeries, error):

    s_Energy = DiscreteSEnergy(prepforsenergy(timeSeries), 1)
    industryMetric = minQ1Q2Q3(timeSeries)
    industryAlt = minQ1Q2Q3alt(timeSeries)
    totalIndustry = minAllQ(timeSeries)
    compressionMetric = compressionStringLength(timeSeries)
    Error = error
```

```
    return[Error,compressionMetric, industryMetric, industryAlt, totalIndustry, s_

    import numpy as np
import pandas as pd
import ast
import matplotlib as mpl
import matplotlib.pyplot as plt
import lzma
import numpy.ma as ma


def prepforuse(timeSeriesStr):
    timeSeries = ast.literal_eval(timeSeriesStr)
    return timeSeries


# Turn 1 dim y value data to 2 dim x,y data
def prepforsenergy(timeSeries):
    # Convert timeSeries to NumPy array if it’s not
    if type(timeSeries) != np.ndarray:
        timeSeries = np.array(timeSeries)

    xVal = np.arange(len(timeSeries))

    # Normalize xVal to map to [0, 1]
    xVal_normalized = xVal / max(xVal)

    # Normalize timeSeries to map to [0, 1]
    timeSeries_normalized = (timeSeries - min(timeSeries)) / (max(timeSeries) - mi

    # Create a 2D array
    twoDimTimeSeries = np.column_stack((xVal_normalized, timeSeries_normalized))

    return twoDimTimeSeries


#Parameters
```

```
  #P: nxd ndarray of n d-dimensional datapoints
  #s: real nonegative float s
#Returns:
  #discSEnergy: Float containing the discrete-s energy of P
  #s parameter

def DiscreteSEnergy(P, s):
    n = np.shape(P)[0]
    [Rows, Columns] = np.indices((n,n))

    VecDiff = np.abs(P[Rows,:] - P[Columns,:])
    MagDiff = np.power(np.sum(np.power(VecDiff,2), axis=2),1/2) #here we use L2 no
    Diff= np.triu(MagDiff) #remove duplicate differences by considering only those
    MaskedDiff = ma.masked_where(Diff==0, Diff, copy=True) #mask all zero differen

    SumMat = np.ma.power(MaskedDiff, -s)
    sum = np.sum(np.sum(SumMat))

    discSEnergy = np.power(float(n), -2)*sum
    return discSEnergy


def diffDiscreteSEnergy(timeSeries):
    timeSeries=np.array(timeSeries)
    deltaf=[]

    for i in range(52,len(timeSeries)):
        deltaf.append(timeSeries[i]-timeSeries[i-52])

    return DiscreteSEnergy(prepforsenergy(deltaf), 1)


def Q1(timeSeries):
    timeSeries=np.array(timeSeries)
    mean=np.mean(timeSeries)
    mse=np.sum(np.square(timeSeries-mean))/len(timeSeries)
    return mse**(0.5)/mean
```

```
def Q2(timeSeries):
    timeSeries=np.array(timeSeries)
    deltaf=[]

    for i in range(1,len(timeSeries)):
        deltaf.append(timeSeries[i]-timeSeries[i-1])


    deltaf = np.array(deltaf)
    delmean=np.mean(abs(deltaf))

    #delmean = (outputf[-1]-outputf[0])/(len(outputf)-1)


    mse= np.sum(np.square(abs(deltaf-delmean))/len(deltaf))

    if delmean == 0:
            return mse**(0.5)/(delmean+0.000001)

    return mse**(0.5)/delmean

def Q3(timeSeries):
    timeSeries = np.array(timeSeries)
    deltaf52=[]

    for i in range(52, len(timeSeries)):
        deltaf52.append(timeSeries[i]-timeSeries[i-52])

    deltaf52 = np.array(deltaf52)
    delmean52 = np.mean(abs(deltaf52))

    mse = np.sum(np.square(abs(deltaf52- delmean52)))/len(deltaf52)


    Q3 = mse**(.5)/delmean52
    return Q3
```

```
def Q2alt(timeSeries):
    timeSeries=np.array(timeSeries)
    deltaf=[]

    for i in range(1,len(timeSeries)):
        deltaf.append(timeSeries[i]-timeSeries[i-1])

    delmean=np.mean(deltaf)

    #delmean = (outputf[-1]-outputf[0])/(len(outputf)-1)


    mse= np.sum(np.square((deltaf-delmean))/len(deltaf))

    if delmean == 0:
            return mse**(0.5)/(delmean+0.000001)

    return mse**(0.5)/delmean

def Q3alt(timeSeries):
    timeSeries = np.array(timeSeries)
    deltaf52=[]

    for i in range(52, len(timeSeries)):
        deltaf52.append(timeSeries[i]-timeSeries[i-52])

    delmean52 = np.mean((deltaf52))

    mse = np.sum(np.square((deltaf52- delmean52)))/len(deltaf52)


    Q3 = mse**(.5)/delmean52
    return Q3


def minQ1Q2Q3(timeSeries):
    return min(abs(Q1(timeSeries)), abs(Q2(timeSeries)), abs(Q3(timeSeries)))
```

```
def minQ1Q2Q3alt(timeSeries):
    return min(abs(Q1(timeSeries)), abs(Q2alt(timeSeries)), abs(Q3alt(timeSeries))

def minAllQ(timeSeries):
    return min(abs(Q1(timeSeries)), abs(Q2alt(timeSeries)), abs(Q3alt(timeSeries))


import struct

def compressionStringLength(timeSeries):
    # Convert floats to integers
    sequence_int = [int(round(x)) for x in timeSeries]

    #convert bytes
    sequence_bytes = b''.join(struct.pack('i', num) for num in sequence_int)

    # Compress the sequence
    compressed_data = lzma.compress(sequence_bytes)
    #with py7zr.SevenZipFile(sequence, 'w') as archive:
        #archive.writeall("target/")
    compressed_length = len(compressed_data)
    return compressed_length
```

6.4.3 LSTM

```
import numpy as np
import pandas as pd
import ast
import matplotlib as mpl
import matplotlib.pyplot as plt
import lzma
import numpy.ma as ma


def prepforuse(timeSeriesStr):
    timeSeries = ast.literal_eval(timeSeriesStr)
    return timeSeries
```

```
# Turn 1 dim y value data to 2 dim x,y data
def prepforsenergy(timeSeries):
    # Convert timeSeries to NumPy array if it's not
    if type(timeSeries) != np.ndarray:
        timeSeries = np.array(timeSeries)

    xVal = np.arange(len(timeSeries))

    # Normalize xVal to map to [0, 1]
    xVal_normalized = xVal / max(xVal)

    # Normalize timeSeries to map to [0, 1]
    timeSeries_normalized = (timeSeries - min(timeSeries)) / (max(timeSeries) - mi

    # Create a 2D array
    twoDimTimeSeries = np.column_stack((xVal_normalized, timeSeries_normalized))

    return twoDimTimeSeries


#Parameters
  #P: nxd ndarray of n d-dimensional datapoints
  #s: real nonegative float s
#Returns:
  #discSEnergy: Float containing the discrete-s energy of P
  #s parameter

def DiscreteSEnergy(P, s):
    n = np.shape(P)[0]
    [Rows, Columns] = np.indices((n,n))

    VecDiff = np.abs(P[Rows,:] - P[Columns,:])
    MagDiff = np.power(np.sum(np.power(VecDiff,2), axis=2),1/2) #here we use L2 no
    Diff= np.triu(MagDiff) #remove duplicate differences by considering only those
    MaskedDiff = ma.masked_where(Diff==0, Diff, copy=True) #mask all zero differen

    SumMat = np.ma.power(MaskedDiff, -s)
```

```
    sum = np.sum(np.sum(SumMat))

    discSEnergy = np.power(float(n), -2)*sum
    return discSEnergy




def Q1(timeSeries):
    timeSeries=np.array(timeSeries)
    mean=np.mean(timeSeries)
    mse=np.sum(np.square(timeSeries-mean))/len(timeSeries)
    return mse**(0.5)/mean

def Q2(timeSeries):
    timeSeries=np.array(timeSeries)
    deltaf=[]

    for i in range(1,len(timeSeries)):
        deltaf.append(timeSeries[i]-timeSeries[i-1])


    deltaf = np.array(deltaf)
    delmean=np.mean(abs(deltaf))

    #delmean = (outputf[-1]-outputf[0])/(len(outputf)-1)


    mse= np.sum(np.square(abs(deltaf-delmean))/len(deltaf))

    if delmean == 0:
            return mse**(0.5)/(delmean+0.000001)

    return mse**(0.5)/delmean

def Q3(timeSeries):
    timeSeries = np.array(timeSeries)
    deltaf52=[]
```

```
    for i in range(52, len(timeSeries)):
        deltaf52.append(timeSeries[i]-timeSeries[i-52])

    deltaf52 = np.array(deltaf52)
    delmean52 = np.mean(abs(deltaf52))

    mse = np.sum(np.square(abs(deltaf52- delmean52)))/len(deltaf52)


    Q3 = mse**(.5)/delmean52
    return Q3


def Q2alt(timeSeries):
    timeSeries=np.array(timeSeries)
    deltaf=[]

    for i in range(1,len(timeSeries)):
        deltaf.append(timeSeries[i]-timeSeries[i-1])

    delmean=np.mean(deltaf)

    #delmean = (outputf[-1]-outputf[0])/(len(outputf)-1)


    mse= np.sum(np.square((deltaf-delmean))/len(deltaf))

    if delmean == 0:
            return mse**(0.5)/(delmean+0.000001)

    return mse**(0.5)/delmean

def Q3alt(timeSeries):
    timeSeries = np.array(timeSeries)
    deltaf52=[]

    for i in range(52, len(timeSeries)):
```

```
        deltaf52.append(timeSeries[i]-timeSeries[i-52])

    delmean52 = np.mean((deltaf52))

    mse = np.sum(np.square((deltaf52- delmean52)))/len(deltaf52)


    Q3 = mse**(.5)/delmean52
    return Q3


def minQ1Q2Q3(timeSeries):
    return min(abs(Q1(timeSeries)), abs(Q2(timeSeries)), abs(Q3(timeSeries)))

def minQ1Q2Q3alt(timeSeries):
    return min(abs(Q1(timeSeries)), abs(Q2alt(timeSeries)), abs(Q3alt(timeSeries))

def minAllQ(timeSeries):
    return min(abs(Q1(timeSeries)), abs(Q2alt(timeSeries)), abs(Q3alt(timeSeries))


import struct

def compressionStringLength(timeSeries):
    # Convert floats to integers
    sequence_int = [int(round(x)) for x in timeSeries]

    #convert bytes
    sequence_bytes = b''.join(struct.pack('i', num) for num in sequence_int)

    # Compress the sequence
    compressed_data = lzma.compress(sequence_bytes)
    #with py7zr.SevenZipFile(sequence, 'w') as archive:
        #archive.writeall("target/")
    compressed_length = len(compressed_data)
    return compressed_length
```

6.5 Statistical Analysis

```
    import seaborn as sns
import statsmodels.api as sm

def statAnalysis(datatoGraph):

    # Convert the NumPy array to a DataFrame
    df_to_graph = pd.DataFrame(datatoGraph, columns = ['RMSE/SD','Compression Metr

    print(df_to_graph)

    #Create a pair plot
    sns.pairplot(df_to_graph)

    #All linear regressions

    Y = df_to_graph['RMSE/SD']
    X = df_to_graph['Discrete 1-Energy']

    X = sm.add_constant(X)

    model = sm.OLS(Y,X).fit()

    predictions = model.predict(X)

    print_model = model.summary()

    print(print_model)

    Y = df_to_graph['RMSE/SD']
    X = df_to_graph['Min Q1, Q2, Q3']

    X = sm.add_constant(X)

    model = sm.OLS(Y,X).fit()

    predictions = model.predict(X)
```

```
print_model = model.summary()

print(print_model)

Y = df_to_graph['RMSE/SD']
X = df_to_graph['Min Q1, Alt Q2, Alt Q3']

X = sm.add_constant(X)

model = sm.OLS(Y,X).fit()

predictions = model.predict(X)

print_model = model.summary()

print(print_model)

Y = df_to_graph['RMSE/SD']
X = df_to_graph['Min all Industry']

X = sm.add_constant(X)

model = sm.OLS(Y,X).fit()

predictions = model.predict(X)

print_model = model.summary()

print(print_model)

return
```

7 Experiment 3: No Seasonality in 1-Energy calculation

7.1 Predictability Metrics

```
def obtainMetrics2(timeSeries, error):

    diff_s_Energy = diffDiscreteSEnergy(timeSeries)
    industryMetric = minQ1Q2Q3(timeSeries)
    industryAlt = minQ1Q2Q3alt(timeSeries)
    totalIndustry = minAllQ(timeSeries)
    compressionMetric = compressionStringLength(timeSeries)
    Error = error

    return[Error,compressionMetric, industryMetric, industryAlt, totalIndustry, di

def diffDiscreteSEnergy(timeSeries):
    timeSeries=np.array(timeSeries)
    deltaf=[]

    for i in range(52,len(timeSeries)):
        deltaf.append(timeSeries[i]-timeSeries[i-52])

    return DiscreteSEnergy(prepforsenergy(deltaf), 1)
```

7.2 Statistical Analysis

```
import seaborn as sns
import statsmodels.api as sm

def statAnalysis2(datatoGraph):

    # Convert the NumPy array to a DataFrame
    df_to_graph = pd.DataFrame(datatoGraph, columns = ['RMSE/SD','Compression Metr

    print(df_to_graph)
```

```
#Create a pair plot
sns.pairplot(df_to_graph)

#All linear regressions

Y = df_to_graph['RMSE/SD']
X = df_to_graph['Discrete 1-Energy without Seasonality']

X = sm.add_constant(X)

model = sm.OLS(Y,X).fit()

predictions = model.predict(X)

print_model = model.summary()

print(print_model)

Y = df_to_graph['RMSE/SD']
X = df_to_graph['Min Q1, Q2, Q3']

X = sm.add_constant(X)

model = sm.OLS(Y,X).fit()

predictions = model.predict(X)

print_model = model.summary()

print(print_model)

Y = df_to_graph['RMSE/SD']
X = df_to_graph['Min Q1, Alt Q2, Alt Q3']

X = sm.add_constant(X)

model = sm.OLS(Y,X).fit()
```

```
    predictions = model.predict(X)

    print_model = model.summary()

    print(print_model)

    Y = df_to_graph['RMSE/SD']
    X = df_to_graph['Min all Industry']

    X = sm.add_constant(X)

    model = sm.OLS(Y,X).fit()

    predictions = model.predict(X)

    print_model = model.summary()

    print(print_model)

    return
```

8 Experiment 4: Shorter time series

8.1 Data Cleaning

```
import pandas as pd

df = pd.read_csv("cleansalesdata.csv")

columns_to_drop = ['Unnamed: 0', 'product_id', 'location_id', 'state', 'department

df.drop(columns=columns_to_drop, axis=1, inplace=True)
columns_to_drop2 = (df.columns[0:-10].values)

df.drop(columns=columns_to_drop2, axis=1, inplace=True)
```

```
def drop_rows_with_zero_first_last(df):
    # Drop rows where the first or last entry is 0
    df = df[(df.iloc[:, 0] != 0) & (df.iloc[:, -1] != 0)]
    return df
df = drop_rows_with_zero_first_last(df)

df
```

8.2 LSTM

```
    import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error


# Assuming 'df' is your DataFrame where each row is a time series


look_back = 3

def create_dataset(series, look_back):
    X, y = [], []
    for i in range(len(series) - look_back):
        a = series[i:(i + look_back), 0]
        X.append(a)
        y.append(series[i + look_back, 0])
    return np.array(X), np.array(y)

#look_back = 12  # Number of previous time steps to consider
forecasts = []  # To store forecasts for each time series
```

```
def forecastSeries(df):

    datatoGraph = []
    for index, row in df.iterrows():
        # Process each time series
        series = row[~pd.isna(row)].values.reshape(-1, 1)
        scaler = MinMaxScaler(feature_range=(0, 1))
        series_scaled = scaler.fit_transform(series)

        timeSeriesforStats = series.T[0]
        ##print(timeSeriesforStats)

        trainIndex = int(.8*(timeSeriesforStats.size))

        timeSeriesforStats = timeSeriesforStats[:trainIndex]



        # Create dataset
        X, y = create_dataset(series_scaled, look_back)
        X = np.reshape(X, (X.shape[0], X.shape[1], 1))

        # Split into training and testing sets (80-20 split)
        split_idx = int(len(X) * 0.8)
        X_train, X_test = X[:split_idx], X[split_idx:]
        y_train, y_test = y[:split_idx], y[split_idx:]

        # Build and compile the LSTM model
        model = Sequential()
        model.add(LSTM(10, return_sequences=True, input_shape=(look_back, 1)))
        model.add(LSTM(3))
        model.add(Dense(1))
        model.compile(optimizer='adam', loss='mean_squared_error')

        # Train the model
        model.fit(X_train, y_train, epochs=10, batch_size=2)

        # Forecasting34
```

```
        forecast = model.predict(X_test)

        # Inverse transform to original scale for both forecast and y_test
        forecast_original_scale = scaler.inverse_transform(forecast)
        y_test_original_scale = scaler.inverse_transform(y_test.reshape(-1, 1))


        # Calculate RMSE on normalized data
        rmse = np.sqrt(mean_squared_error(y_test, forecast.flatten()))
        sd =  np.std(series_scaled)
        error = rmse/sd


        #Calculate our Experimental Measures
        experimentalMeasures = obtainMetrics(timeSeriesforStats, error)
        datatoGraph.append(experimentalMeasures)



        # Plotting
        plt.figure(figsize=(12, 6))
        plt.title(f"Time Series and Forecast for Row {index} (RMSE/SD: {error:.2f}
        plt.xlabel("Time")
        plt.ylabel("Value")

        # Plotting the original series
        plt.plot(series, label='Original Series')

        # Plotting the forecasted part
        forecast_index = np.arange(len(series) - len(forecast_original_scale), len
        plt.plot(forecast_index, forecast_original_scale.flatten(), label='Forecas

        plt.legend()
        plt.show()
    return datatoGraph

datatoGraph = forecastSeries(df)
```

9 Experiment 5: Aggregation

9.1 Data Cleaning

```
import numpy as np
import pandas as pd
import ast
import matplotlib as mpl
import matplotlib.pyplot as plt
import lzma
import numpy.ma as ma

df = pd.read_csv("cleansalesdata.csv")

columns_to_drop = ['Unnamed: 0', 'location_id', 'state', 'department']

df.drop(columns=columns_to_drop, axis=1, inplace=True)


def drop_rows_with_zero_first_last(df):
    # Drop rows where the first or last entry is 0
    df = df[(df.iloc[:, 1] != 0) & (df.iloc[:, -1] != 0)]
    return df
df = drop_rows_with_zero_first_last(df)


# Group by the product id and sum the values
df_grouped = df.groupby('product_id').sum().reset_index()

df_grouped

df_grouped.drop(columns= 'product_id', axis=1, inplace=True)
df_grouped
```

9.2 LSTM

```
import pandas as pd
import numpy as np
```

```
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error


# Assuming 'df' is your DataFrame where each row is a time series


look_back = 16

def create_dataset(series, look_back):
    X, y = [], []
    for i in range(len(series) - look_back):
        a = series[i:(i + look_back), 0]
        X.append(a)
        y.append(series[i + look_back, 0])
    return np.array(X), np.array(y)

#look_back = 12  # Number of previous time steps to consider
forecasts = []  # To store forecasts for each time series

def forecastSeries(df):

    datatoGraph = []
    for index, row in df.iterrows():
        # Process each time series
        series = row[~pd.isna(row)].values.reshape(-1, 1)
        scaler = MinMaxScaler(feature_range=(0, 1))
        series_scaled = scaler.fit_transform(series)

        timeSeriesforStats = series.T[0]
        ##print(timeSeriesforStats)

        trainIndex = int(.8*(timeSeriesforStats.size))

        timeSeriesforStats = timeSeriesforStats[:trainIndex]
```

```
# Create dataset
X, y = create_dataset(series_scaled, look_back)
X = np.reshape(X, (X.shape[0], X.shape[1], 1))

# Split into training and testing sets (80-20 split)
split_idx = int(len(X) * 0.8)
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]

# Build and compile the LSTM model
model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape=(look_back, 1)))
model.add(LSTM(50))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=8)

# Forecasting34
forecast = model.predict(X_test)

# Inverse transform to original scale for both forecast and y_test
forecast_original_scale = scaler.inverse_transform(forecast)
y_test_original_scale = scaler.inverse_transform(y_test.reshape(-1, 1))


# Calculate RMSE on normalized data
rmse = np.sqrt(mean_squared_error(y_test, forecast.flatten()))
sd =  np.std(series_scaled)
error = rmse/sd


#Calculate our Experimental Measures
experimentalMeasures = obtainMetrics(timeSeriesforStats, error)
```

```
        datatoGraph.append(experimentalMeasures)



        # Plotting
        plt.figure(figsize=(12, 6))
        plt.title(f"Time Series and Forecast for Row {index} (RMSE/SD: {error:.2f}
        plt.xlabel("Time")
        plt.ylabel("Value")

        # Plotting the original series
        plt.plot(series, label='Original Series')

        # Plotting the forecasted part
        forecast_index = np.arange(len(series) - len(forecast_original_scale), len
        plt.plot(forecast_index, forecast_original_scale.flatten(), label='Forecas

        plt.legend()
        plt.show()
    return datatoGraph

datatoGraph = forecastSeries(df_grouped)
```

10 Cantor Lattice Experiment

```
import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras import backend
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics
import math
from math import log
```

```
import numpy as np
from numpy.linalg import norm
import matplotlib
import matplotlib.pyplot as plt



from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
import itertools
from itertools import product

import random

x=[]
kgap=4  # determines how many time the iteration will go, will generate 2^kgap num
def xinAi(k,coor,power,dimension):  # for the case x in Ai, dimension is the dim o
    for an in [0, 2**(dimension-1)]:
        coor = coor + an*((2**dimension)**(-power))

        if power < k:
            xinAi(k, coor, power + 1, dimension)

        else :
            x.append(coor)

xinAi(kgap,0,1,2)

def IdentitySum(HighDimVar):
  HighDimVar = np.array(HighDimVar)
  sum = 0
  for i in range(len(HighDimVar)):
      sum = sum + HighDimVar[i]
  return sum

oneDimlist=[]

HighDimList=[]
```

```
outputlist=[]

x=[]

CantorList=[]

def HighDimCantorSet(CantorDimension):
  for i in range(2,CantorDimension+2):

    xinAi(kgap,0,1,i)
    xcopy=x.copy()
    #print(x)
    CantorList.append(xcopy)

    x.clear()


HighDimCantorSet(4)

print(CantorList)


CantorList=[]
def HighDim(NumofCarProdut,function):  #NumofCarProdut, it makes sense for trainin

  #CantorList.clear()

  #CantorListCopy=CantorList.copy()
  HighDimCantorSet(NumofCarProdut)
  CarProdut=list(itertools.product(*CantorList))

  #CantorList.clear()

  for i in range(len(CarProdut)):    # HighDimList list of high dim points on the
      HighDimList.append(CarProdut[i])
      #print(CarProdut)
```

```
  for i in range(len(CarProdut)):    # sum of each point's coordinate and return a
        #print(sum(CarProdut[i]))
        oneDimlist.append(sum(CarProdut[i]))

  for i in range(len(HighDimList)):   # the function which input is each point in
      outputlist.append(function(HighDimList[i]))


#print(HighDimList)

print(x)

oneDimlist=[]

HighDimList=[]

outputlist=[]

CantorList=[]

def increaseDiscreteDimDECOMP(finalDim):
  for i in range(2,finalDim+1):


    oneDimlist.clear()
    HighDimList.clear()
    outputlist.clear()

    CantorList.clear()

    HighDim(i,IdentitySum)

    #print(HighDimList)
    #print(CarProdut)

    #Distencefunction
    #HighDim(xaxis,i,IdentitySum)
```

```
    #HighDim(xaxis,i,SqrtAllComp)

    z1 = np.array(oneDimlist)
    k1 = np.array(HighDimList)

    #print(HighDimListCopy)

    z1_train, z1_test, k1_train, k1_test = train_test_split(z1,k1, test_size=0.2,

    backend.clear_session()
    modelh = Sequential()
    modelh.add(Dense(100, input_dim=1, activation='relu')) # Hidden 1  dim increas
    modelh.add(Dense(50, activation='relu')) # Hidden 2
    modelh.add(Dense(i)) # Output dimension
    modelh.compile(loss='mean_squared_error', optimizer='adam')
    modelh.fit(z1_train,k1_train,verbose=2,epochs=10, batch_size=10)

    pred2 = modelh.predict(z1_test)
    score2 = np.sqrt(metrics.mean_squared_error(pred2,k1_test))
    print("**********", score2)



increaseDiscreteDimDECOMP(5)
```

11 Code for various figures

```
import numpy as np
import matplotlib.pyplot as plt

def random_walk(steps):
    # Initialize the position at 0
    position = 0
    # Create an array to store the positions
    positions = [position]
```

```
    # Simulate the random walk
    for _ in range(steps):
        # Generate a random step of -1 or 1
        step = np.random.choice([-1, 1])
        # Update the position
        position = step
        # Append the new position to the list
        positions.append(position)

    return positions

# Number of steps in the random walk
steps = 100

# Simulate the random walk
positionsHeadsTails = random_walk(steps)

# Plot the random walk
plt.plot(positionsHeadsTails)
plt.title("Heads and Tails Game (100 steps)")
plt.xlabel("Time t")
plt.ylabel("Dollars Recieved")
plt.grid(True)
plt.show()

# Calculate resultant random walk


def sumTimeSeries(positionsHeadsTails):
    positionsRandomWalk = [positionsHeadsTails[0]]
    for i in range(1,len(positionsHeadsTails)):
        positionsRandomWalk.append(positionsRandomWalk[i-1] + positionsHeadsTails[
    return(positionsRandomWalk)


positionsRandomWalk = sumTimeSeries(positionsHeadsTails)
# Plot the random walk
plt.plot(positionsRandomWalk)
```

```
plt.title("Heads and Tails Random Walk (100 steps)")
plt.xlabel("Time t")
plt.ylabel("Net Profit")
plt.grid(True)
plt.show()

import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def forecast_next_values(timeSeries, k):
    """
    This function trains a feedforward neural network on a sequence of numbers to

    Args:
    numbers (list): List of numbers forming a time series.
    k (int): The number of future values to predict.

    Returns:
    forecasted_values (list): The forecasted future 'k' values.
    """

    numbers = timeSeries[:80]

    # Set the random seeds for reproducibility
    np.random.seed(0)
    tf.random.set_seed(0)

    # Prepare the data for the feedforward network input
    X, y = [], []
    for i in range(len(numbers) - k):
        X.append(numbers[i:i+k])
        y.append(numbers[i+k])

    # Convert the data to numpy arrays
    X = np.array(X)
    y = np.array(y)
```

```
    # Create the feedforward model
    model = Sequential()
    model.add(Dense(10, input_dim=k, activation='relu'))  # Hidden layer with 10 n
    model.add(Dense(1))  # Output layer

    # Compile the model with MSE as the loss function and Adam optimizer
    model.compile(loss='mean_squared_error', optimizer='adam')

    # Train the model
    model.fit(X, y, epochs=100, batch_size=1, verbose=0)

    # Use the model to forecast the next k values
    last_k_values = numbers[-k:]  # Get the last 'k' observed values
    forecasted_values = []
    for _ in range(k):
        input_data = np.reshape(last_k_values, (1, k))  # Reshape the data to fit
        next_value = model.predict(input_data)  # Use the model to forecast the ne
        numbers.append(next_value[0, 0])  # Append the forecasted value to the lis
        last_k_values = np.append(last_k_values[1:], next_value[0, 0])  # Update t

    return numbers


forecastHeadsTails = forecast_next_values(positionsHeadsTails,20)

# Plot the random walk
plt.plot(positionsHeadsTails)
plt.plot(forecastHeadsTails)


plt.title("Forecast for Heads and Tails Game (100 steps)")
plt.xlabel("Time t")
plt.ylabel("Dollars Recieved")
plt.grid(True)
plt.show()

forecastRandomWalk = sumTimeSeries(forecastHeadsTails)
```

```
plt.plot(positionsRandomWalk)
plt.plot(forecastRandomWalk)

plt.title("Forecast for Heads and Tails Random Walk (100 steps)")
plt.xlabel("Time t")
plt.ylabel("Net Profit")
plt.grid(True)
plt.show()

# Make linear trend with random component from random walk
for i in range(len(positionsRandomWalk)):
    positionsRandomWalk[i] = positionsRandomWalk[i] + i
plt.plot(positionsRandomWalk)
plt.title("Linear Trend Plus Random Walk (100 steps)")
plt.xlabel("Time t")
plt.ylabel("Net Profit")
plt.grid(True)
plt.show()

forecastLinearTrendNoise = forecast_next_values(positionsRandomWalk,20)

plt.plot(positionsRandomWalk)
plt.plot(forecastLinearTrendNoise)

plt.title("Forecast for Linear Trend plus Random Walk (100 steps)")
plt.xlabel("Time t")
plt.ylabel("Position")
plt.grid(True)
plt.show()
```

```
import pandas as pd
import numpy as np

df = pd.read_csv("cleansalesdata.csv")
```

```
columns_to_drop = ['Unnamed: 0', 'product_id', 'location_id', 'state', 'department

df.drop(columns=columns_to_drop, axis=1, inplace=True)
df

def get_bad_timeseries1(df):
    # Get rows where firs entry is 0
    dfbad = df[(df.iloc[:, 0] == 0)]
    return dfbad
dfbad1 = get_bad_timeseries1(df)
dfbad1

def get_bad_timeseries2(df):
    # Drop rows where the last entry is 0
    dfbad = df[(df.iloc[:, -1] == 0)]
    return dfbad

dfbad2 = get_bad_timeseries2(df)
dfbad2

import matplotlib.pyplot as plt
# Plotting
plt.figure(figsize=(12, 6))
plt.title("Bad time series Row 996")
plt.xlabel("Time")
plt.ylabel("Value")

# Plotting the original series
plt.plot(dfbad1.iloc[-1], label='Original Series')

plt.legend()
plt.show()

# Plotting
plt.figure(figsize=(12, 6))
plt.title("Bad time series Row 1")
plt.xlabel("Time")
```

```
plt.ylabel("Value")

# Plotting the original series
plt.plot(dfbad2.iloc[1], label='Original Series')

plt.legend()
plt.show()
```

12 References

....

[1] Hornik, Kurt and Stinchcombe, Maxwell B. and White, Halbert, *Multilayer Feedforward Networks are Universal Approximators*, Neural Networks, 1989, Elsevier Science Ltd. 10.1016/0893-6080(89)90020-8

[2] Betti et al, Fractal dimension, approximation and data sets, 2022, 2209.12709, arXiv