Data exploration , basic statistics , filters :

---

Disclaimer: Only 4 pages are allowed therefore I wrote another notebook which is longer and more detailed..
If you think something is lacking I would be very glad if you check there , but maybe I just write too much
since this is only my 2nd report (excluding ICA since it wasn't really a report)

## Preprocessing tasks:

---

### Reading the data :

I used Matan published code to read the data from the file with genres for each movie, the initial data set is as follows :

| Dim | Data |
|---|---|
| 480,000 | Users |
| 17,770 | Movies |
| 27 | Genres |

$480,000 * 17,770 * 27 \approx$ 250 billion slots! I think I finally found something Jeff Bezos can't saturate.

### Filtering the data :

My main goal is to decrease the dimension of the data in a meaningful way that will preserve most of its patterns. First I filtered by thresholds.

| | Movies | Users |
|---|---|---|
| Filters | Must be associated with a genre | 500 Votes or more |

**Users** with at least 500 votes highly decrease our chances of using unrepresentative samples.

**Movies** with genres included will greatly contribute to characterize users.

- Each genre is a one-hot encoded , movie with more than one genre is an addition of the corresponding one-hot-vectors.

Im now left with 12,000~ movies and 54,000~ users , i.e. , each user is a sample of size 12,000~.

# Feature engineering :

---

Each using being represented by a 12,000~ orthogonal vector is a bit wasteful and unpractical for any diagonalize a based algorithm.

**Solution** : We can use the movie genres, and the user rating to profile each user. Assume Yossi is our current user, the following table describe which feature is placed in each index.

| Indexes in feature vector / Users (only Yossi for now but we'll make it someday.) | 0 | 1 | 2 | 3 | 4 until 31 | 31 until 58 |
|---|---|---|---|---|---|---|
| Yossi features | Average ratings | SD ratings | Average genre viewing distribution | Standard deviation genre viewing distribution | Viewing distribution | Rating distribution |

## Math explaining the extraction of the feature vector :

Assume $u_i$ is a user , $\forall i \ 0 \leq i \leq 54,000$ , $m_j$ is a movie , $\forall j \ \ 0 \leq j \leq 12,000$ . $u_i$ watched $A_i$ movies and rated each movie by a ranking $R_i$

so we can profile each user $u_i$ by $A_i, R_i$ , i.e. the movies he watched and how much he rated each. Next time is to reduce the dimension.

## *Combining $A_i$ and $R_i$*

Each move in $A_i$ has a mapping to the corresponding addition of one hot embedding , each one-hot corresponds to the each of the movie genres. , i.e. $A_i \in R^{12000}$ is a one hot vector that indicate which movies he watch. assume $a_1, \ldots, a_k$ equal 1 and the rest are 0. therefor define the following indicator :

$$v_{ij} = \begin{cases} 1 & movie\, i\, include\, genre\, j \\ 0 & o.w. \end{cases}$$

$$g(a_i) = v_i = \sum_j v_{ij}$$

therefor $g(x) : \{1, 2, \ldots, 12000\} \to \{0, 1\}^{27}$ , where 27 is the number of genres. The operator is linear and therefor we can define $f(x) : \{1, 0\}^{12000} \to \{1, 0\}^{27}$

$$f(A_i) = \sum_i g(a_i) = \sum_i v_i = \sum_{ij} v_{ij}$$

Extending the idea to rating by changing $v_{ij}$ to

$$r_{ij} = \begin{cases} 1 & if\ \ u_i\ \ ranked\ movie\ j\ 1 \\ 2 & if\ \ u_i\ \ ranked\ movie\ j\ 2 \\ 3 & if\ \ u_i\ \ ranked\ movie\ j\ 3 \\ 4 & if\ \ u_i\ \ ranked\ movie\ j\ 4 \\ 5 & if\ \ u_i\ \ ranked movie\ j\ 5 \\ 0 & o.w. \end{cases}$$

Therefor extending the idea with the a corresponding $p(x)$ and $m(x)$ to $f(x)$ and $g(x)$ we get :

$$p(R_i) = \sum_i m(r_i) = \sum_i r_i = \sum_{ij} r_{ij}$$

Both vectors are of dim 27 and represent the semantic information captured in the original matrices , $R_i$ and $A_i$. The expression $D_i = \frac{R_i}{A_i}$ is the rating distribution per genre. the expression $A_i$ represent the viewing distribution per genre unnormalized ,. i.e. $K_i = A_i/(A_i.sum())$ is the distribution.

| **Feature vector indexes and meaning** | **How to extract them** |
|---|---|
| **index 0** : Average ratings | $D_i.mean()$ |
| **index 1** standard deviation ratings | $D_i.std()$ |
| **index 2** average viewing distribution | $K_i.avg()$ |
| **index 3** standard deviation viewing distribution | $K_i.std()$ |
| **index's 4-31** Viewing distribution | $K_i * 50$ |
| **Indexes 31-57** Rating distribution normalized | $D_i - 2.5$ |

- I centered the rating distribution since I want to low scores to indicate the negative semantic of a low score ,on the other hand, viewing distribution wasn't normalized since being indifferent to a genre isn't the same as actively disliking him.
- $K_i * 50$ In order to get the same order of magnitude for the distribution in accordance to the remaining features.

## *Current data size :*

Personally I think its amazing , starting with initial shape of $((480,000), (17,770), (27))$ was reduced by feature engineering and meaningful filters to shape $((54,000), 58)$ . The implication are enormous computation wise. $54,000 * 58 \approx 3,000,000$. So in proportion , assuming my feature enringing is valid and only lose a small amount of information , the ratio between the size :

$$\frac{250,000,000,000}{3,000,000} = 83,333$$

Algorithms will run 83,333 times faster when they're *linear* .

# Denoising autoencoders

---

Using autoencoders to reduce the dimension of the feature vector and afterwards applying the manifolds techniques , autoencoders tries to map the data into a manifold in the latent variable space , hopefully then after mapping diffusion map and LLE would extract the underlying manifold. Afterwards I'll compare PCA with AE.

General fitting process description :

1. Picked a shallow auto encoder and tired to get a general idea about the data and training autoencoder , MSE plateaued at 0.4 for latent space of 5

2. Adding dropout to improve accuracy,  playing with different layers of dropout and the corresponding p values , I found the best one at 0.5 for the larger layers and 0.1 for the smaller ones. and MSE managed to reach 0.324
3. Proceeding to denoising autoencoder to make the model more robust by forcing him to fit a noise input to a denoised one we prevent overfitting and help him to generalize.
4. Gradually making the network deeper and increasing the noise accordingly , since deeper network will be able to extract more noise , capping at 0.22 noise for a 30 layers network which yielded the best performance.

The model architecture :

```
LATENT_SPACE_DIM=3
class Denoising_AE(nn.Module):
    def __init__(self):
        super(Denoising_AE, self).__init__()
        self.dropout=nn.Dropout(p=0.5)
        self.dropout_weaker=nn.Dropout(p=0.1)

        # encoder
        self.enc0 = nn.Linear(in_features=58, out_features=55)
        self.enc1 = nn.Linear(in_features=55, out_features=50)
        self.enc2 = nn.Linear(in_features=50, out_features=45)
        self.enc3 = nn.Linear(in_features=45, out_features=40)
        self.enc4 = nn.Linear(in_features=40, out_features=35)
        self.enc5 = nn.Linear(in_features=35, out_features=30)
        self.enc6 = nn.Linear(in_features=30, out_features=25)
        self.enc7 = nn.Linear(in_features=25, out_features=20)
        self.enc8 = nn.Linear(in_features=20, out_features=20)
        self.enc9 = nn.Linear(in_features=20, out_features=20)
        self.enc10 = nn.Linear(in_features=20, out_features=20)
        self.enc11 = nn.Linear(in_features=20, out_features=20)
        self.enc12 = nn.Linear(in_features=20, out_features=20)
        self.enc13 = nn.Linear(in_features=20, out_features=20)
        self.enc14 = nn.Linear(in_features=20, out_features=15)
        self.enc15 = nn.Linear(in_features=15, out_features=LATENT_SPACE_DIM)

        # decoder
        self.dec0 = nn.Linear(in_features=LATENT_SPACE_DIM, out_features=15)
        self.dec1 = nn.Linear(in_features=15, out_features=20)
        self.dec2 = nn.Linear(in_features=20, out_features=20)
        self.dec3 = nn.Linear(in_features=20, out_features=20)
        self.dec4 = nn.Linear(in_features=20, out_features=20)
        self.dec5 = nn.Linear(in_features=20, out_features=20)
        self.dec6 = nn.Linear(in_features=20, out_features=20)
        self.dec7 = nn.Linear(in_features=20, out_features=20)
        self.dec8 = nn.Linear(in_features=20, out_features=25)
        self.dec9 = nn.Linear(in_features=25, out_features=30)
        self.dec10 = nn.Linear(in_features=30, out_features=35)
        self.dec11 = nn.Linear(in_features=35, out_features=40)
        self.dec12 = nn.Linear(in_features=40, out_features=45)
        self.dec13 = nn.Linear(in_features=45, out_features=50)
        self.dec14 = nn.Linear(in_features=50, out_features=55)
        self.dec15 = nn.Linear(in_features=55, out_features=58)

    def encoder(self,x):
        x = F.relu(self.enc0(x))
        x=self.dropout(x)
        x = F.relu(self.enc1(x))
        x=self.dropout(x)
        x = F.relu(self.enc2(x))
        x=self.dropout(x)
        x = F.relu(self.enc3(x))
        x=self.dropout_weaker(x)
        x = F.relu(self.enc4(x))
        x=self.dropout_weaker(x)
        x = F.relu(self.enc5(x))
        x=self.dropout_weaker(x)
        x = F.relu(self.enc6(x))
        x=self.dropout_weaker(x)
        x = F.relu(self.enc7(x))
        x = F.relu(self.enc8(x))
        x = F.relu(self.enc9(x))
        x = F.relu(self.enc10(x))
        x = F.relu(self.enc11(x))
        x = F.relu(self.enc12(x))
        x = F.relu(self.enc13(x))
        x = F.relu(self.enc14(x))
        x = F.relu(self.enc15(x))
        return x

    def decoder(self,x):
        x = F.relu(self.dec0(x))
        x = F.relu(self.dec1(x))
        x = F.relu(self.dec2(x))
        x = F.relu(self.dec3(x))
        x = F.relu(self.dec4(x))
        x = F.relu(self.dec5(x))
        x = F.relu(self.dec6(x))
        x = F.relu(self.dec7(x))
```

```
        x = F.relu(self.dec8(x))
        x=self.dropout_weaker(x)

        x = F.relu(self.dec9(x))
        x=self.dropout_weaker(x)

        x = F.relu(self.dec10(x))
        x=self.dropout_weaker(x)

        x = F.relu(self.dec11(x))
        x=self.dropout_weaker(x)

        x = F.relu(self.dec12(x))
        x=self.dropout(x)

        x = F.relu(self.dec13(x))
        x=self.dropout(x)

        x = F.relu(self.dec14(x))
        x=self.dropout(x)

        x = F.relu(self.dec15(x))
        return x
```

## *What does the MSE mean ?*

One the main obstacles I faced was how to determine if my loss is good enough? MSE isn't accuracy like we saw in previous exercises and I have no visual guidance like a denoised image. An approximation I came up with was

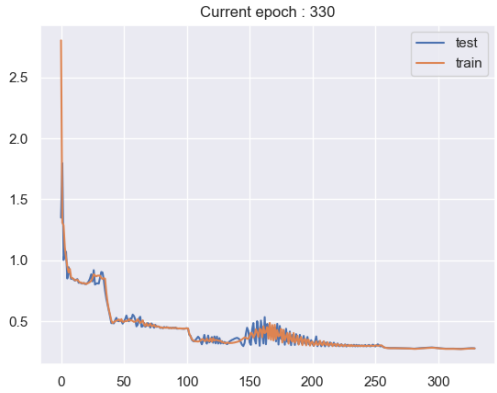
$$MSE = 58 * (C) \iff C = MSE/58$$

Given an MSE and solving that equation $C$ would be the average MSE per feature , therefor $\sqrt{C}$ would be the average distance per feature. for a given MSE of 0.273 , $\sqrt{C}$=0.07 , i.e.

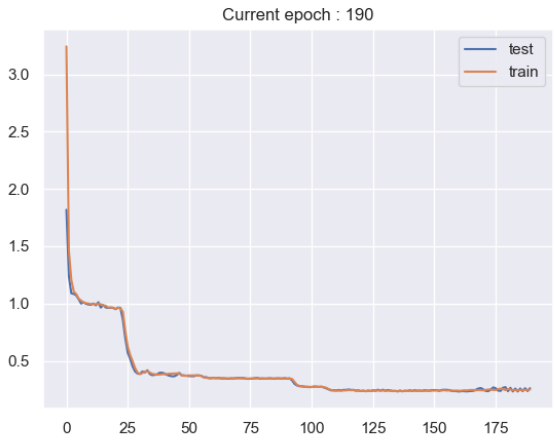
$$0.273 \approx (0.07)^2 * 58$$

Taking into account the fact that our features are between [-2.5,10] (In extreme cases some features may reach at most 10 , the average maximum feature per user is around 5) a difference of 0.07 doesn't mean that much.

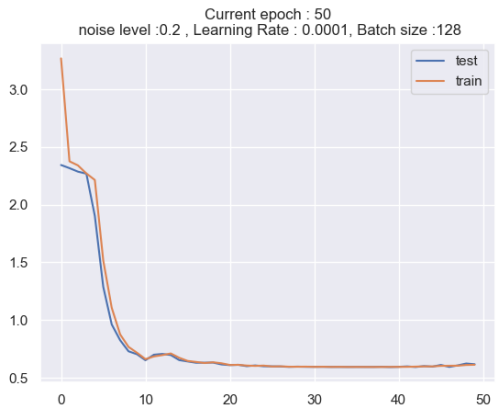
## Understating the valley :

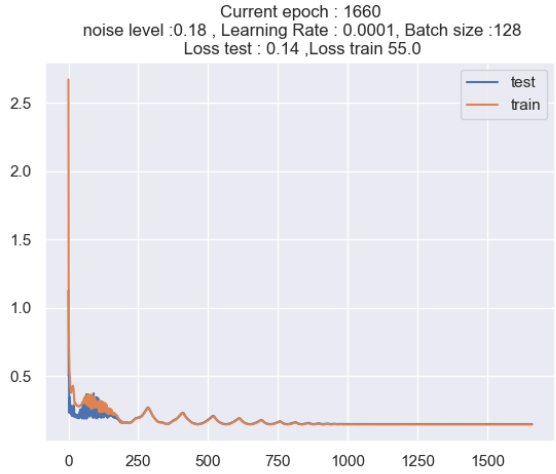



Convergence of a shallow (6 Layers)Denoising Autoencoder with latent variable space of 10 , convergence MSE : 0.3




Convergence of a deep Denoising Autoencoder with latent variable space of 3 , Noise level : 0.1 , convergences MSE : 0.235




Convergence of a deep Denoising Autoencoder with latent variable space of 5 with noise 0.3 , failed compare to the rest at 0.5 MSE , mainly due to high noise.

Adding multiple (14 in total) dropout layers , with increasing noise a bit to 0.22 results in the best result thus far , $MSE = 0.147$

Couldn't figure out why the error convergence in valleys , I think it means that he figured out some feature mapping in the latent space that he didn't manage before and the fast convergence is just optimizing the weights based on the feature he figured out.

Best MSE is $0.14 = (0.05)^2 * 58$ , therefor $\sqrt{C} = 0.05$ over a latent space of dimension 3.

## Checking the results :

A grave lesson was learned today. a neural network will find a way to do her task , even if its a trivial way , on top of that I should always sample latent variables during training.

The NN managed to map each and every input to 0 and fix the biases in the decoder part to minimize the loss. i.e. for every user I received a 3 dimensional vector : [0,0,0] , such a magnificent manifold indeed.

### *My assumptions about the above:*

The error was actually misleading , huge batch size (128) on top of severe dropout channels and noising the input resulted in her inability to learn anything , so she decided that her best guess would but not to guess anything. The decoder and encoder worked together in an elaborate plot to broaden my knowledge and in the process expose my own fallacies. The encoder only job was sending every input to zero ,the decoder job was more complicated. The decoder changed the weights and biases from there(His first layer) on to reach a constant vector which minimizes the distances between all the users output , i.e. he actually solved the following minimization problem :

$$C \in R^{58}, min_C \left\| \sum_{i=1}^{n} f_{u_n} - C \right\|^2$$

So I literary wasted a day optimizing the exact opposite of my goal.

### *Attempting to remedy it :*

- Dropout discarded
- Noise level in the input is zero
- Increase latent variable dimension to 10.
- Batch size to 4 from 128

**Remark** : Im certain that's not the optimal solution , using denoising encoders and dropout while monitoring the latent space more carefully would yield optimal results , but given the time constraints I'm forced to move.

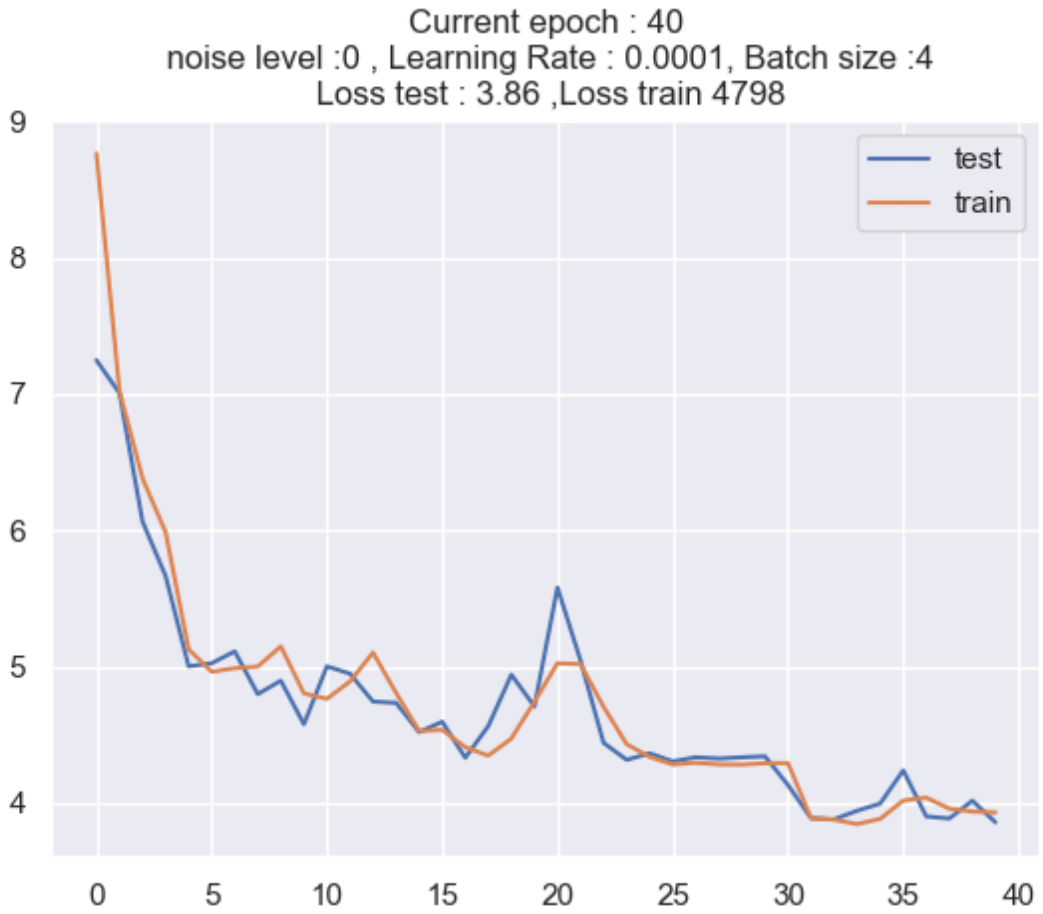
All of those are contributing to the variance in the data therefor I decreased them , Unfortunately I don't have time to optimize as I did before so my results won't be optimal. After those changes I observed the training latent variable and saw a good disparity between each input. on top of that while the latent dimension I set was 10 he constantly only used 3 dimension , which I think indicate that he doesn't need more then 3 dimensions? But maybe he'll increase it in a prolong training if he understand that's the only way to increase his accuracy.

| **Epoch/Sample latent variable** | **Latent variable** |
|---|---|
| 10 | $[7.7414e+26, 7.6903e+26, 7.1145e+26, 0, 0, 0, 0, 0, 0, 0]$ |
| 10 | $[8.3318e+26, 8.2775e+26, 7.6576e+26, 0, 0, 0, 0, 0, 0, 0]$ |
| 20 | $[[3.7136e+29, 3.6801e+29, 3.4068e+29, 0, 0, 0, 0, 0, 0, 0]$ |

- So even in this case , I'm still at 3 dimension right now , therefor summing a randomly sampled 1000 latent variables results in :

$$[5.854e+345.793e+345.371e+340.000e+000.000e+000.000e+000.000e+000.000e+000.000e+000.000e+00]$$

  which mean that my  manifold extrinsic  dimension is 3 with very high probability , because given a dimension between 1 to 10  to project the data into the autoencoder chose 3.
- As the epoch increases the values increases , i.e. he's mapping them farther a part to find more meaningful projection. So  it should indicate good things?




---

Training with out a trivial solution ,as we can see the MSE is much higher then before and the results are more meaningful in return Don't forget to put final convergence graph here.

| **Epochs\Metrics** | **20** | **30** | **40** |
|---|---|---|---|
| Average abs distance per feature ($\sqrt{C}$ ) | 0.287 | 0.275 | 0.2556 |
| MSE | 4.7 | 4.3 | 3.9 |

## Current data size :

Using the encoder to encode all the datasets and then slicing all the zero columns out leaves us with a dataset of  shape $((54,000),3)$ , i.e. he's of size $162,000$

$$\frac{3,000,000}{54,000*3}=19.33$$

Autoencoders manage to help us reduce the dataset size 20 times , while costing on average 0.25 (**CHANGE IF IMPORVED**)abs distance per feature, the cherry on the top is the greatly reduced effect of the curse of dimensionality.  Therefor the dataset would look as follows for 2 randomly samples users :

| **Features/Users** | **x** | **y** | **z** |
|---|---|---|---|
| 1 | $8.3318e+26$ | $8.2775e+26$ | $7.6576e+26$ |
| 2 | $8.1216e+26$ | $7.2118e+26$ | $7.6788e+26$ |

I made another version of the dataset which include the normalized values of the latent variables since I'm not sure how the big numbers will react , i.e. a sample from the second dataset , but since the data is so sparse It could be a bad idea , since those numbers are extremely close.

| **Features/Users** | **x** | **y** | **z** |
|---|---|---|---|

| Features/Users | x | y | z |
|---|---|---|---|
| 1 | 0.81883124 | 0.81883109 | 0.81883061 |
| 2 | −0.65894032 | −0.65894021 | −0.65894025 |

## Preprocessing summary :

| Stage | Stage summary | Data dimension |
|---|---|---|
| 1 | Initial data | $((480,000),(17,770),(27))$ |
| 2 | user rating amount bigger then 500 | $((54,000),(17,770),(27))$ |
| 3 | movies with genres data available | $((54,000),(12,220),(27))$ |
| 4 | Feature engineering where user is a sample : | $((54,000),58)$ |
| 5 | Deep denoising autoencoders | $((54,000),3)$ |

None trivial filters , combining with feature engineering and deep autoencoders managed to reduce the data set size by a 1.5 million.

$$\frac{250,000,000,000}{54,000*3}=1,543,209$$

All that's left now is to hope that my feature engineering logic was correct , otherwise the next part will be a complete failure. (This part was written before I realized my autoencoder mapped everything to 0)

# Subsampling

# Manifold learning techniques

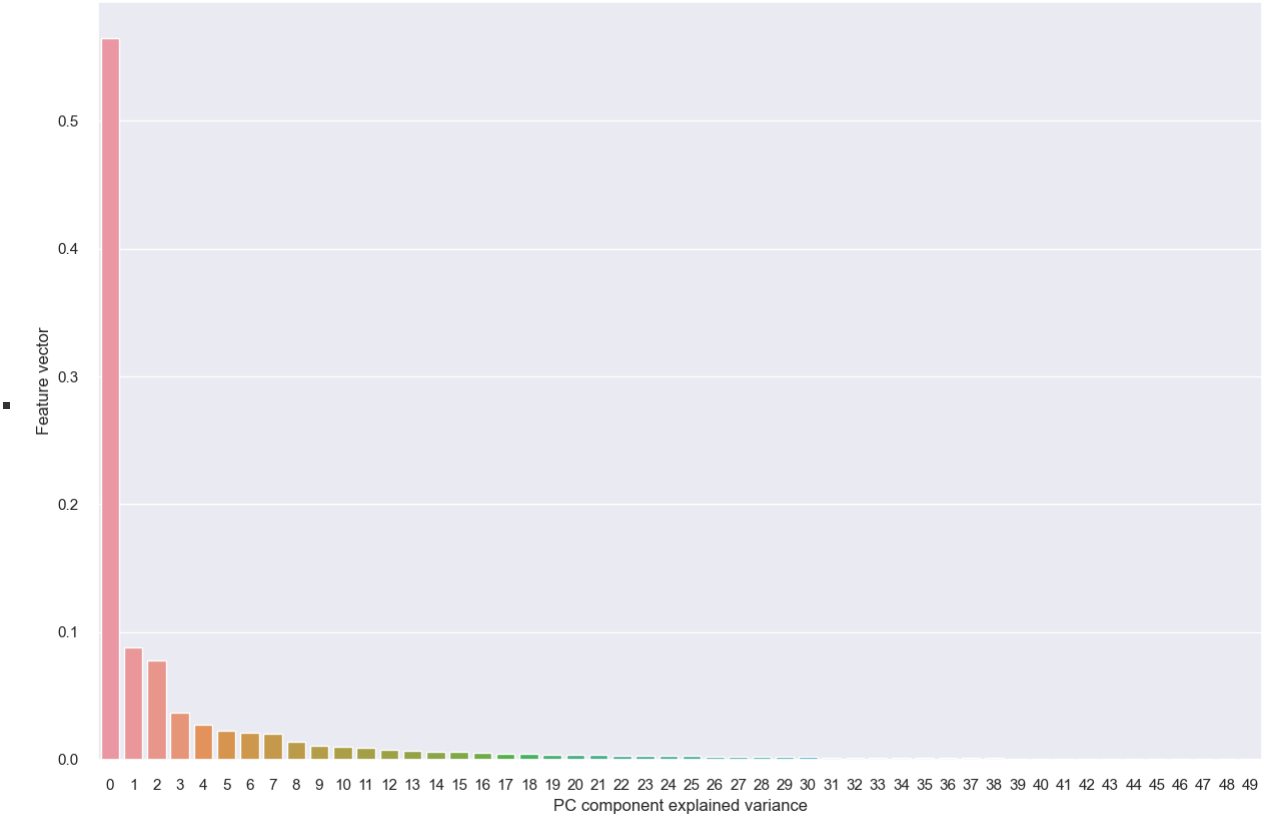
.

## Autoencoders results :

Notation : Im going to call the 58 dim feature vector I designed by feature vector data and the autoencoder by AE data \ latent space.

Suboptimal results regarding the auto encoder dimension  reduction , the results stems from my lack of experience with the model . still using the line (which is **very** sparse) have to include some patterns since I validated the training wasn't placebo and average distance per feature was 0.25. I'll first PCA the dataset to achieve the 1 dim line with out losing any information afterwards I'll seek patterns. The eigenvalues explained variance behave as expected :

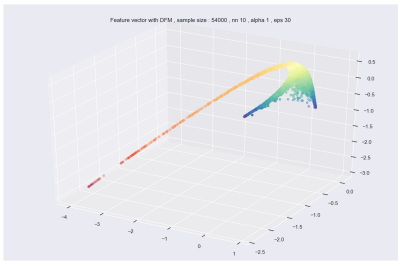
| E.V. / Explained variance | $\lambda_0$ | $\lambda_1$ | $\lambda_2$ |
|---|---|---|---|
| Explained Variance | 1 | $3.04717761e-14\approx 0$ | $2.98594610e-14\approx 0$ |



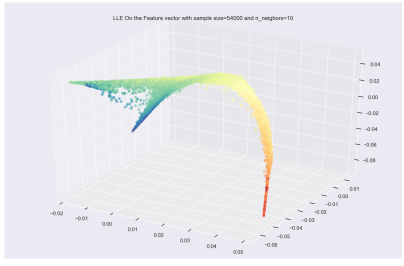

- The first E.V. with zero in the feature vector explained variance is E.V. number 42 , the 10 first components holds 91% of the explained variance. In other words the features are not meaningless if in linear methods we need 42 dimension to remain lossless.

## Comparing results :

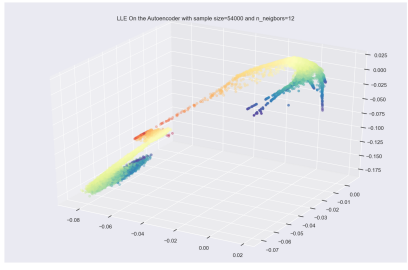
Diffusion map couldn't extract meaningful data from the AE data and always outputted a line , sometimes a curved line which happened because of high NN. therefor he's ignore from now on.



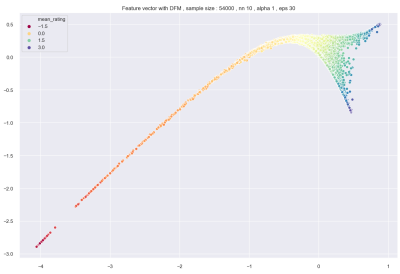
The local density is preserved much better , but seems like the DFM lost more global structure by the combining "horns" in the left

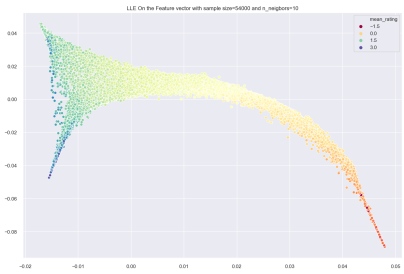


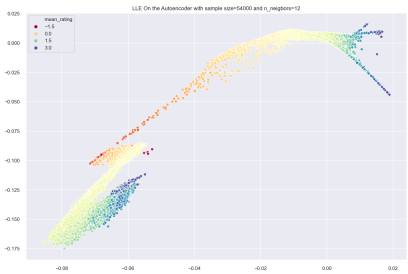
Global structure is preserved much better but global structure isn't as sparse as DFM.



LLE managed to derive the intrinsic dimension from the AE data and expand it to 3 dimension , even thou 99.99% of the explained variance was in the first vector, *very* surprising results







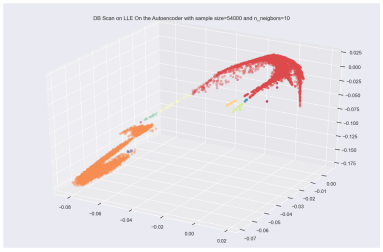
Its worth noting that it is magic that LLE managed to extract a 3d graph with complex patterns from a 1D line with miniscule variance in axis y,z. From here on I'll compare and research those graphs based on the feature vector.

# Different structure different meaning :

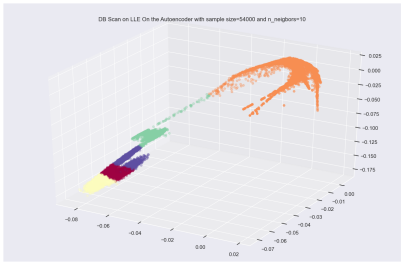
I'll go over meaningful features in the feature vector and use them as the color for the graphs to find patterns. While trying to use clusters DB Scan and K Means they found a nice general structure to the data but coloring by feature vectors revealed much more intriguing things. In the following compression table I'll switch between 3d and 2d graphs based visualization clarity , and showcase failures and success of models . The first 3 rows are intuitionally chosen , from there on I sorted the features by their std , the 4 most dominants ones are

```
Talk-Show Avg rat     0.60786
Film-Noir Avg rat     0.66601
News Avg rat          0.79646
Reality-TV Avg rat    0.85629
```

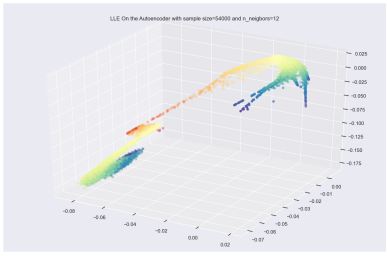
Taking high variance features will hopefully allow me to understand the dimension reduction quality.



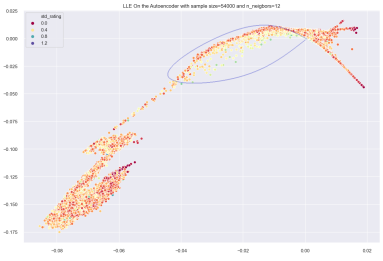
DB Scan actually managing to cluster the parts pretty well. Orange and red correspond to high average rating users for example , and the middle yellow and green are low rating users.



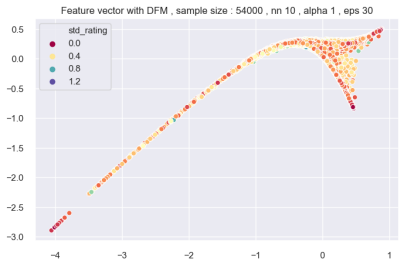
I chose K=5 to separate by the colors of the average rating approx which should have been doable given the distribution. But the purple points mix Blue and yellow and they're far apart



Mean rating distribution was captured perfectly by every version of the LLE or the DFM.



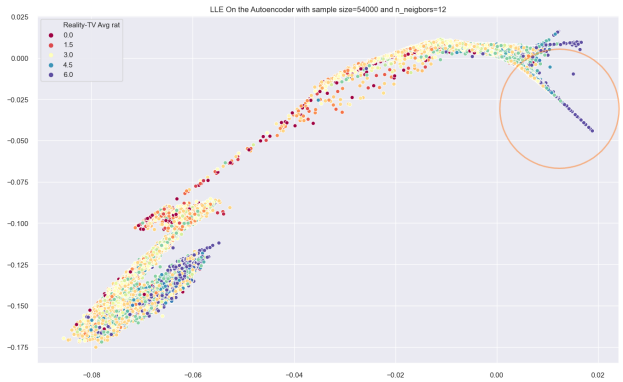
Std rating distribution clearly declare that users tend to vote the same way . A complete separate area can be seen in the top middle right where users vary in their rating.




Failure of the DFM Feature vector to understand the std rating. LLE Feature vector also managed to distinguish clearly.

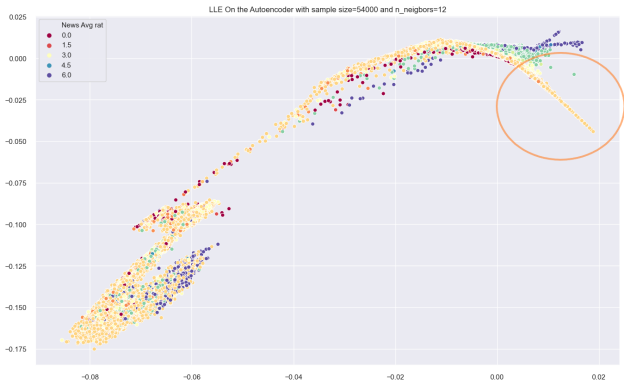
This picture examples a failure of the feature scaling on my part , The variance in the feature vector of Drama popularity is high (0.4) , but its high since he's popular and therefor all his values are more extreme , i.e. , not complex patterns exists that justify the variance because of sub optimal feature scaling.

**Comparing high variance genres : Genre 1**



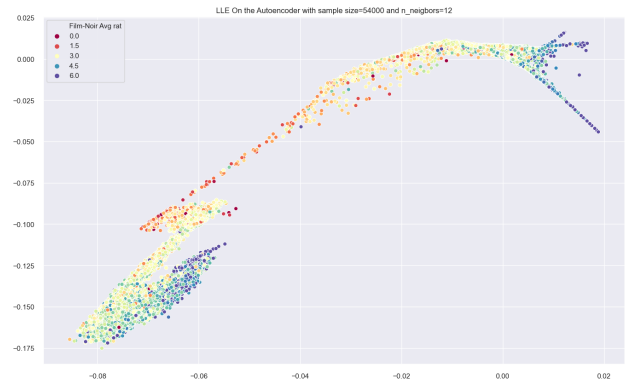

A clear separation based on the Reality's TV Average rating of the user. Circled wing in demonstrate a separation that was learned by the algorithms in the dimensionality reduction.,

**Genre 2**



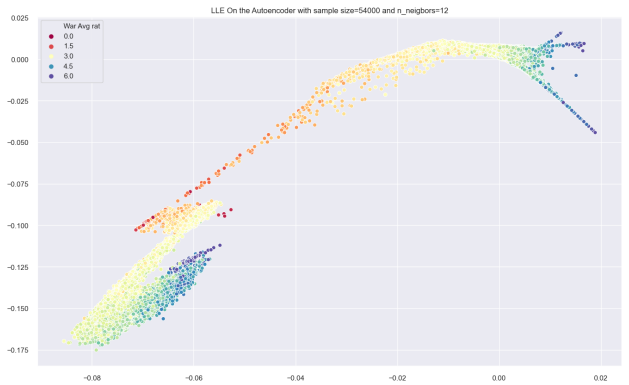

A meaningful distribution also applies here , we can see that News genre is less liked then the Reality's genre , its a shame that Trump valiant efforts to combine those genres doesn't reflect here.

**Comparing high variance genres : Genre 1**

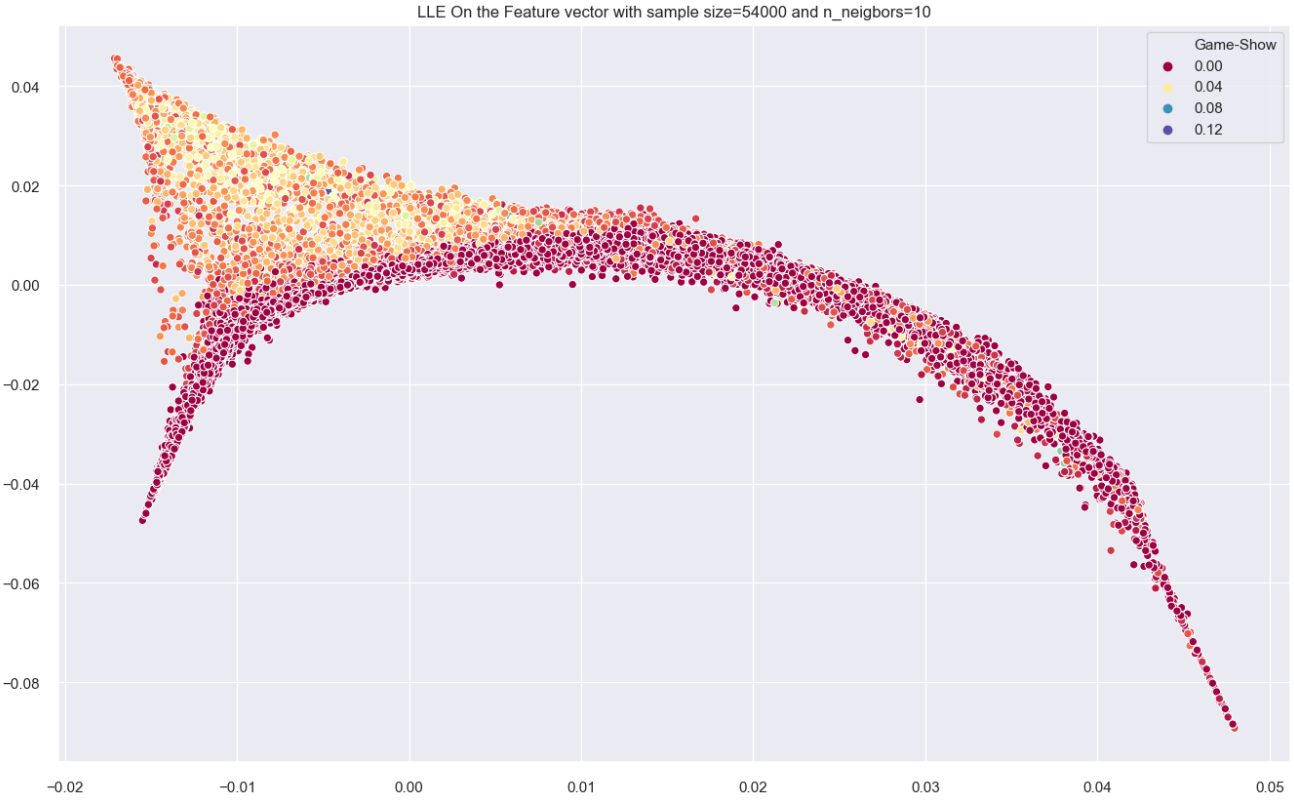



Users rating distribution here is more continuous , this genre attracts almost all users , and every user have different opinions on him.

**Genre 2**




An example of a complexly continuous distribution captured by the model , therefor the model approximate the War average rating Perfectly.




Game show  popularity is also an interesting pattern that's captured , which indicate that even among the Game-Show fans he's only a niche , and most of the population doesn't watch it.

When I examined the other features most of them exhibited the same patterns that weren't notable enough to graph.

# Summary & Conclusion

Using initial filters by statistics , feature engineering results in a feature vector of 58 dimensions . I trained an autoencoder I designed from scratch and It was a roller-coaster that made me scratch my head  a lot but I also really enjoyed the process and prepared myself to next week hopefully.  PCA and autoencoders were compared both with regard to their dimension reduction quality. Afterwards the performance of LLE and DFM on reduced dimension by both of them was compared.

DFM couldn't figure out the patterns in AE latent space and rightfully so , the AE mapped all the data into a 3d line with PC1 explained variance of 0.99999 , which corresponds to the global structure of the manifold , LLE managed to pull a rabbit out of a hat and understand the miniscule variance in PC2 and PC3 explained variance. in regard to the feature vector , DFM managed to preserve global structure while LLE managed to preserve a balance between global and local.

Clustering was done managed to separate different regions of rating distribution but the results were general. Afterwards examining the interstice  dimension produced by algorithms based on the original feature vector as the color ,trying to understand which patterns preserved the dimeonsality reduction.

Therefor reduction the feature vector of each user from 17,770*27 to 3 while preserving a lot of the patterns (Autoencoder loss suggest that the average distance between each feature upon reconstruction is 0.25 In the 58 -> 3 section) and finding meaningful patterns on in the 2\3 dim data as described in the section above is amazing in my opinion.