Unsupervised Learning Report

Rishabh Kaushick
NU ID: 002808996
College of Engineering
Northeastern University
Toronto, ON
kaushick.r@northeastern.edu

Assignment 3

Abstract

Dataset #	Dataset	Unsupervised Learning Algorithm
Dataset #1	Near Earth Objects	k-Means Clustering
		Gaussian Mixture Model (GMM)
		Principal Component Analysis (PCA)
		Independent Component Analysis (ICA)
Dataset #2	Heart Attack Risk Prediction	k-Means Clustering
		Gaussian Mixture Model (GMM)
		Principal Component Analysis (PCA)
		Independent Component Analysis (ICA)

Table 1: Datasets & algorithms covered.

Datasets

Dataset	Name	Dataset Characteristics	Attribute Characteristics	Associated Task	Number of Instances	Number of Attributes
1	NASA Near Earth Objects (NEOs)	Multivariate	Real	Clustering & Dimensionality Reduction	904	27
2	Heart Attack Risk Prediction	Multivariate	Real	Clustering & Dimensionality Reduction	8764	25

Table 2: Dataset Details

Why I Find Both the Datasets Interesting

Both the datasets were chosen with different aspects in mind. As an Astro-Physics enthusiast, the first one containing the data about the Near-Earth Objects (NEOs) was strikingly interesting. Earth, the only known planet with life, must be protected from threats including foreign objects in our galaxy. The data of NEOs collected by NASA can help for us to be prepared and devise solutions if an asteroid or another natural satellite approaches in a potentially hazardous route towards the Earth. The second dataset was chosen because it aligns with my values to positively give back to the society.

Unlike the thread of NEOs perhaps many light years away, the threat of a heart attack is much more urgent. Machine Learning can aid doctors to be largely able to detect those patients having a high risk of heart attacks to take preventive measures.

Additional Exploratory Data Analysis (EDA)

As part of the Assignment 2A & 2B, we have used the same dataset, therefore the data visualization remains the same. However, below I have gone further deeper into the data before training the unsupervised models & feature reduction algorithms.

Redundant Columns

Let us revisit the redundant columns. Firstly, we can see that Min Diameter and Max Diameter attributes are repeatedly present in different units as shown in the below figure.

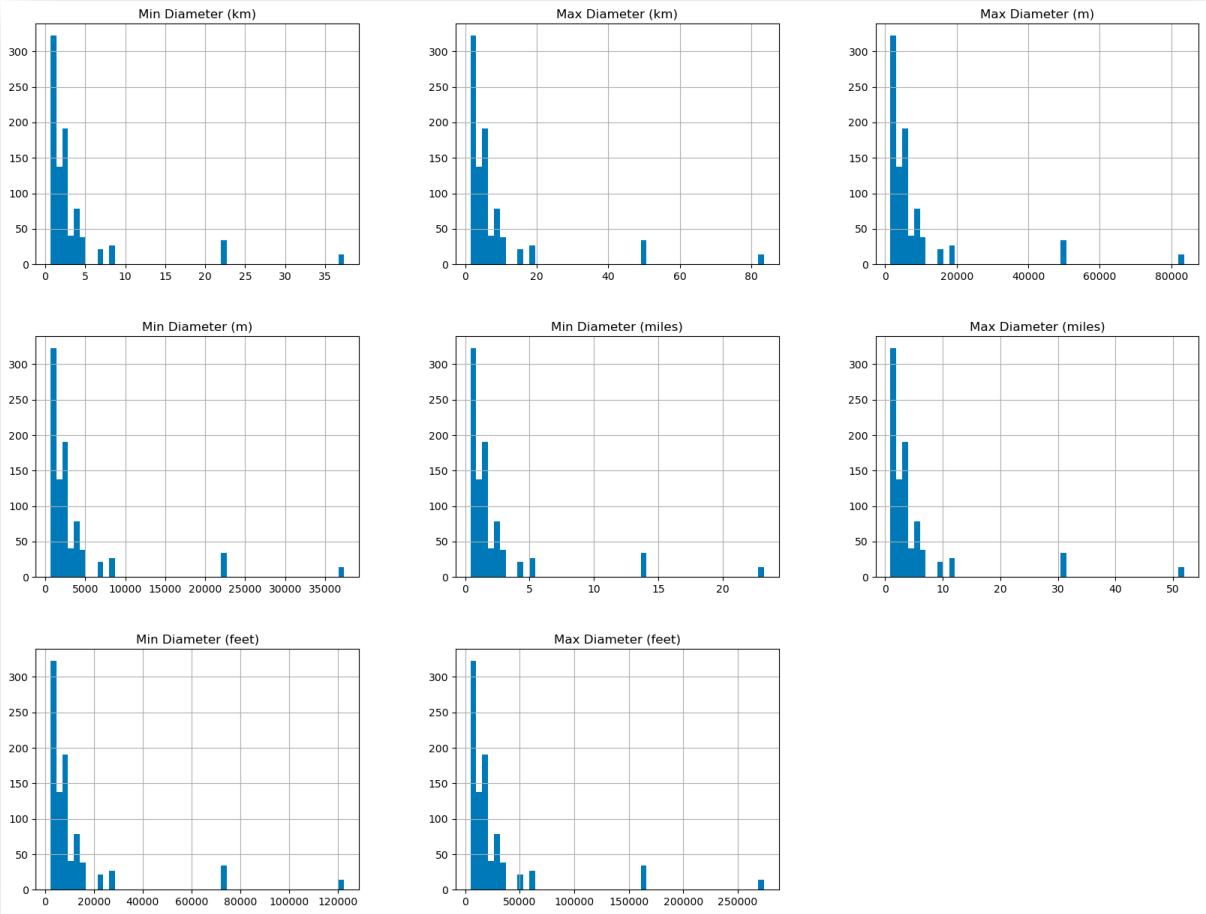


Figure 1: Min Diameter & Max Diameter features in different units.

Since 1 mile is larger than 1 kilometer, 1 meter and 1 foot, the values of min diameter and max diameter will be smallest in the unit of miles and largest in the unit of feet. The figure below shows the drastic difference in values and mean between the minimum diameter in miles and feet.

```
neo_df['Min Diameter (miles)'].describe()
[14]  ✓ 0.0s
...  count    904.000000
     mean       2.280249
     std        3.643036
     min        0.405421
     25%        0.794156
     50%        1.258653
     75%        2.177238
     max       23.222339
     Name: Min Diameter (miles), dtype: float64
```

```
neo_df['Min Diameter (feet)'].describe()
[16]  ✓ 0.0s
...  count       904.000000
     mean      12039.717721
     std       19235.235316
     min        2140.622213
     25%        4193.146421
     50%        6645.689217
     75%       11495.821927
     max      122613.990772
     Name: Min Diameter (feet), dtype: float64
```

Figure 2: Drastic difference in values between miles and feet

Training the machine learning models will be easier on smaller data values compared to much larger ones. Therefore, we decide to drop the following attributes:

- Min Diameter (km)
- Max Diameter (km)
- Min Diameter (m)
- Max Diameter (m)
- Min Diameter (feet)
- Max Diameter (feet)

Similarly relative velocity is also in different units as shown in the figure below. Here the km/s velocity values are the smallest we will follow the same principal as stated above and drop the other columns - Relative Velocity (km/h) and Relative Velocity (miles/h).

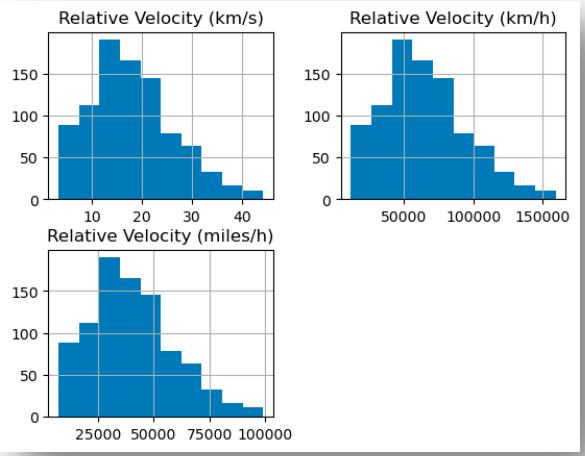


Figure 3: Relative Velocity columns.

Similarly, we have Miss Distance column which is present in units - astronomical, lunar, km, and miles. From the figure below, we can see that the units km and miles are very large in the order of magnitude of 10^8. The astronomical unit is the one which is between 0 and 2. This would be the best one to keep for the machine learning model, therefore we will drop the other 3 columns.

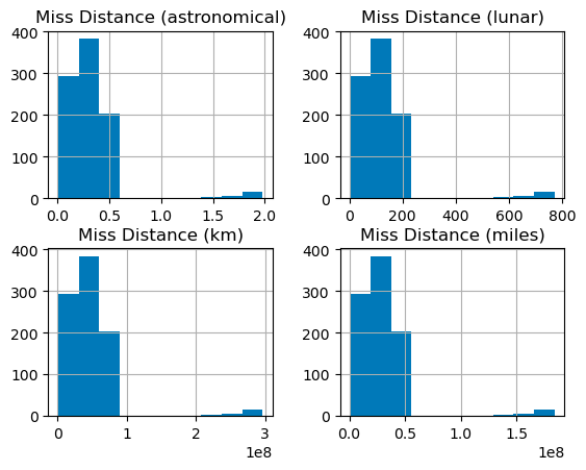


Figure 4: Miss Distance columns.

Preparing The Data for the Models

Categorical Data- Orbiting Body

For the orbiting body feature we have the following categorical data:

1. Earth
2. Venus
3. Juptr
4. Mars
5. Merc

```
neo_df['Orbiting Body'].value_counts()
[26]  ✓ 0.0s
...   Orbiting Body
      Earth    787
      Venus     66
      Juptr     25
      Mars      17
      Merc       9
      Name: count, dtype: int64
```

Figure 5: Orbiting Body categorical values

We must encode the data for this column. However, the way that Label Encoders do these is by converting the text into categorical numbers. For example, Earth – 0, Venus – 1, Juptr – 2, and so forth. This potentially could be misleading because the model might take different inferences based on the value of the encoded numbers. For this reason, in the previous assignment, I did not use Label Encoding, however I used different numbers based on the size of the orbiting body:

Furthermore, the machine learning models cannot understand string values. In our dataset, we have columns such as 'Name' and 'Orbiting Body' which both contain string values. Therefore, we must convert them into some numeric values. For the Orbiting body, based on my intuition, I have set the values for each of the orbiting body to have a value from 0 to 4 based on their size. Therefore:

- *# 0 = Merc*
- *# 1 = Mars*
- *# 2 = Venus*
- *# 3 = Earth*
- *# 4 = Juptr*

Figure 6: Screenshot of handling Orbiting Body feature in the previous assignment

I realized that this is not a very accurate representation of the different planets. Therefore, instead of this, we are going to use One Hot Encoders to represent this data. In our case One Hot Encoders are a good option, since there are only 5 categorical data values. In the case when there are many categories for a column, One Hot Encoders would not be a good option.

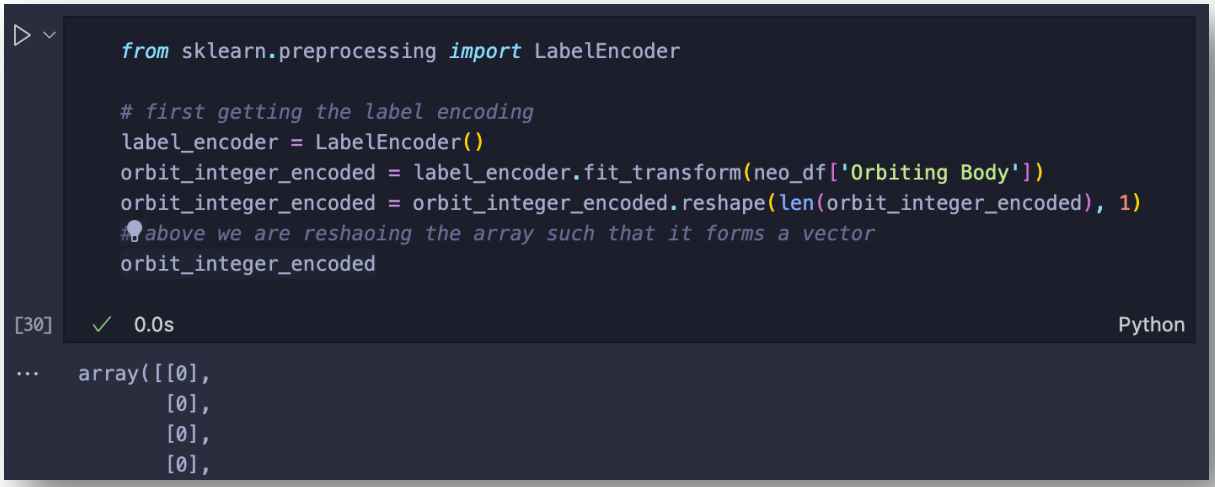
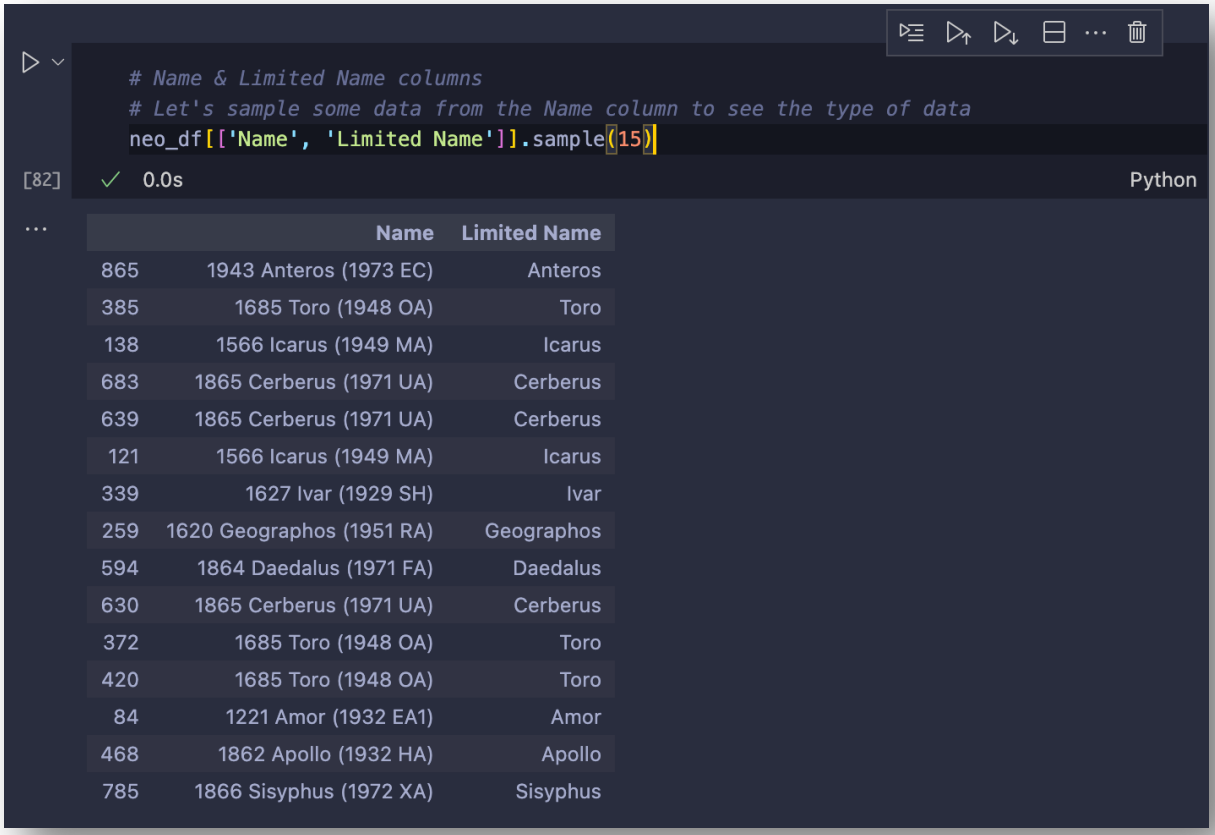


Figure 7: Using Label Encoding on the Orbiting Body feature first.

Figure 8: Using One Hot Encoding next.

String Value Columns- Name & Limited Name

When we sampled some data of the columns ‘Name’ and ‘Limited Name’, we could see that there were multiple repetitions. I realized that the Name column and the Limited Name column are subsets of each other as viewed from the below figure.



	Name	Limited Name
865	1943 Anteros (1973 EC)	Anteros
385	1685 Toro (1948 OA)	Toro
138	1566 Icarus (1949 MA)	Icarus
683	1865 Cerberus (1971 UA)	Cerberus
639	1865 Cerberus (1971 UA)	Cerberus
121	1566 Icarus (1949 MA)	Icarus
339	1627 Ivar (1929 SH)	Ivar
259	1620 Geographos (1951 RA)	Geographos
594	1864 Daedalus (1971 FA)	Daedalus
630	1865 Cerberus (1971 UA)	Cerberus
372	1685 Toro (1948 OA)	Toro
420	1685 Toro (1948 OA)	Toro
84	1221 Amor (1932 EA1)	Amor
468	1862 Apollo (1932 HA)	Apollo
785	1866 Sisyphus (1972 XA)	Sisyphus

Figure 9: Sample data from Name & Limited Name columns.

From the above figure we can also see that name and limited name columns are repeating multiple times. These columns can be treated as categories. Therefore, let us see how many such categories of data is present in the Limited Name column:

```
neo_df['Limited Name'].value_counts()
```

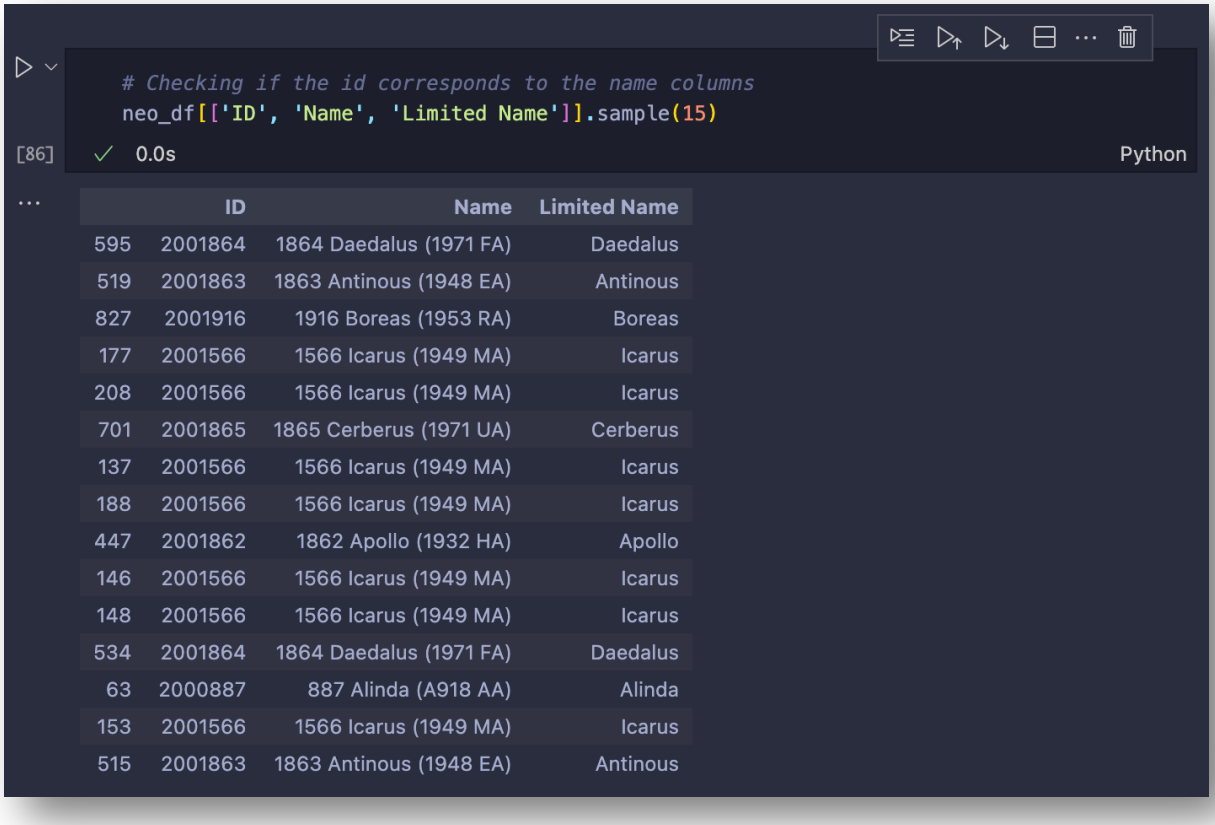
[85] ✓ 0.0s Python

```
Limited Name
Cerberus        150
Icarus          115
Apollo           84
Daedalus         81
Toro             78
Geographos       76
Anteros          37
Eros             34
Amor             29
Quetzalcoatl     29
Sisyphus         27
Betulia          25
Tezcatlipoca     24
Antinous         22
Ivar             21
Albert           17
Cuyo             15
Ganymed          14
Alinda           14
Boreas           12
Name: count, dtype: int64
```

There are a total of 20 categories of data in the Limited Name column.
If we want to convert this data - it will not be wise to use a One Hot Encoding as that will increase the model complexity tenfold.

Figure 10: Limited Name column – categorical values.

Additionally, I noticed that the ID column was also related to the Name column as shown in the Figure below:



```
# Checking if the id corresponds to the name columns
neo_df[['ID', 'Name', 'Limited Name']].sample(15)
```

[86] ✓ 0.0s Python

	ID	Name	Limited Name
595	2001864	1864 Daedalus (1971 FA)	Daedalus
519	2001863	1863 Antinous (1948 EA)	Antinous
827	2001916	1916 Boreas (1953 RA)	Boreas
177	2001566	1566 Icarus (1949 MA)	Icarus
208	2001566	1566 Icarus (1949 MA)	Icarus
701	2001865	1865 Cerberus (1971 UA)	Cerberus
137	2001566	1566 Icarus (1949 MA)	Icarus
188	2001566	1566 Icarus (1949 MA)	Icarus
447	2001862	1862 Apollo (1932 HA)	Apollo
146	2001566	1566 Icarus (1949 MA)	Icarus
148	2001566	1566 Icarus (1949 MA)	Icarus
534	2001864	1864 Daedalus (1971 FA)	Daedalus
63	2000887	887 Alinda (A918 AA)	Alinda
153	2001566	1566 Icarus (1949 MA)	Icarus
515	2001863	1863 Antinous (1948 EA)	Antinous

Figure 11: Sample of the ID, Name, Limited Name columns.

For example, above we can see that the Limited Name – '*Icarus*' will always have the Name as '*1566 Icarus (1949 MA)*' and ID as '*2001566*'. Therefore, this is the same data represented in many ways. Therefore, we can remove the Name & Limited Name columns and we can keep just the ID column while training the models.

Dropping Columns

Some columns do not have any significance for the data science model, and therefore we can remove them to reduce the model complexity. These columns are:

- NASA JPL URL

Since this dataset is the same one which was used for supervised learning, we must also remove the target field, i.e. 'Is Potentially Hazardous' before training the unsupervised machine learning models.

Correlation Matrix

Once all the unnecessary columns are removed, the correlation matrix is as follows:

```
# Let's find the correlation between all the fields
corr_matrix = neo_df.corr()
corr_matrix.style.background_gradient()
```

[108] ✓ 0.1s Python

	Designation	Absolute Magnitude (H)	Min Diameter (miles)	Max Diameter (miles)	Relative Velocity (km/s)	Miss Distance (astronomical)	Orbit Earth	Orbit Juptr	Orbit Mars	Orbit Merc	Orbit Venus	Close Approach Date (datetime)
Designation	1.000000	0.495540	-0.604571	-0.604571	0.270055	-0.107703	0.008650	-0.151902	0.002290	-0.027549	0.093906	0.031315
Absolute Magnitude (H)	0.495540	1.000000	-0.852578	-0.852578	0.228146	-0.068325	-0.138415	0.006976	0.007181	0.072215	0.142890	0.008876
Min Diameter (miles)	-0.604571	-0.852578	1.000000	1.000000	-0.242810	0.121466	0.034679	0.069299	0.050279	-0.040929	-0.099061	-0.016042
Max Diameter (miles)	-0.604571	-0.852578	1.000000	1.000000	-0.242810	0.121466	0.034679	0.069299	0.050279	-0.040929	-0.099061	-0.016042
Relative Velocity (km/s)	0.270055	0.228146	-0.242810	-0.242810	1.000000	-0.130966	-0.057351	-0.212659	-0.017885	0.253031	0.120821	0.042882
Miss Distance (astronomical)	-0.107703	-0.068325	0.121466	0.121466	-0.130966	1.000000	-0.186815	0.879698	-0.125250	-0.081380	-0.217003	-0.004912
Orbit Earth	0.008650	-0.138415	0.034679	0.034679	-0.057351	-0.186815	1.000000	-0.437391	-0.359052	-0.260078	-0.727854	-0.010047
Orbit Juptr	-0.151902	0.006976	0.069299	0.069299	-0.212659	0.879698	-0.437391	1.000000	-0.023347	-0.016912	-0.047329	-0.013921
Orbit Mars	0.002290	0.007181	0.050279	0.050279	-0.017885	-0.125250	-0.359052	-0.023347	1.000000	-0.013883	-0.038852	0.002828
Orbit Merc	-0.027549	0.072215	-0.040929	-0.040929	0.253031	-0.081380	-0.260078	-0.016912	-0.013883	1.000000	-0.028142	-0.015723
Orbit Venus	0.093906	0.142890	-0.099061	-0.099061	0.120821	-0.217003	-0.727854	-0.047329	-0.038852	-0.028142	1.000000	0.026262
Close Approach Date (datetime)	0.031315	0.008876	-0.016042	-0.016042	0.042882	-0.004912	-0.010047	-0.013921	0.002828	-0.015723	0.026262	1.000000

Figure 12: Correlation Matrix.

From the above figure, we can see a few columns are strongly correlated among each other:

- Designation & Absolute Magnitude (H)
- Miss Distance (astronomical) & Orbit Jupiter
- Minimum Diameter and Maximum Diameter

Designation vs Absolute Magnitude

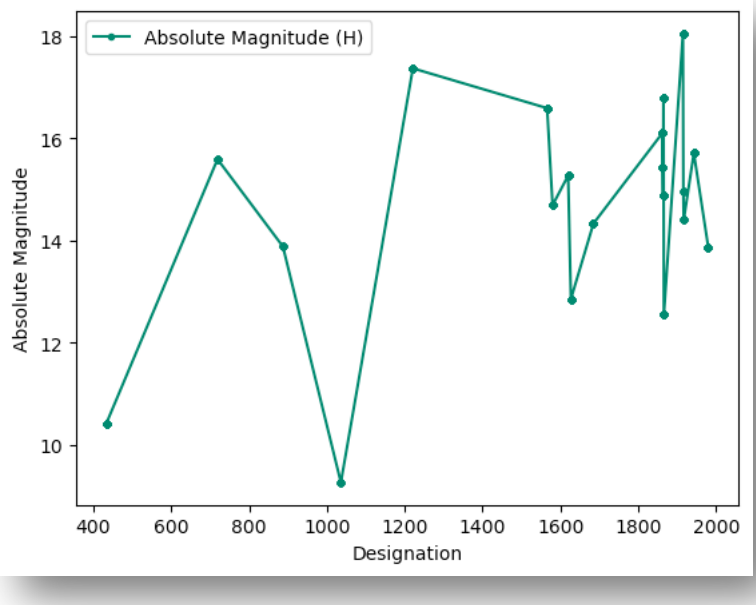


Figure 13: Designation vs Absolute Magnitude (H)

From the above figure, we can see that Designation is only slightly correlated to Absolute Magnitude. It looks like for smaller values of designation, the absolute magnitude is smaller, but there are some exceptions.

Miss Distance (astronomical) vs Orbit Jupiter

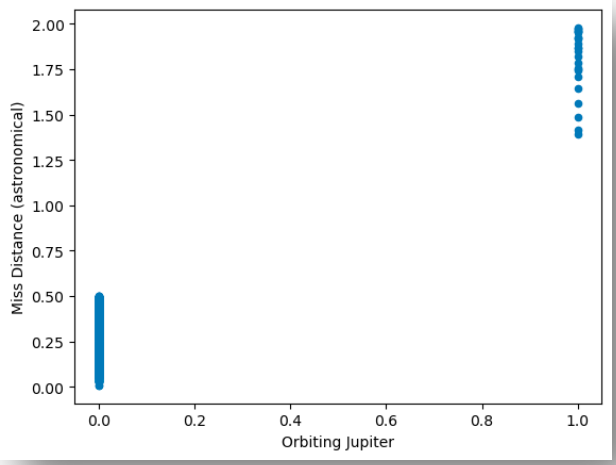


Figure 14

From this we can infer that those NEOs which are usually orbiting Jupiter have a higher miss distance. This logically makes sense, because Jupiter is much farther away from the Earth, and those NEOs orbiting Jupiter have a less likelihood to collide with Earth.

Feature Engineering

Minimum Diameter & Maximum Diameter

From the Figure 12 of the correlation matrix, we can see that the Minimum Diameter and Maximum Diameter have a corelation value of 1. This means that they are directly correlated with each other. Therefore plotting a graph gives us a straight line as shown below:

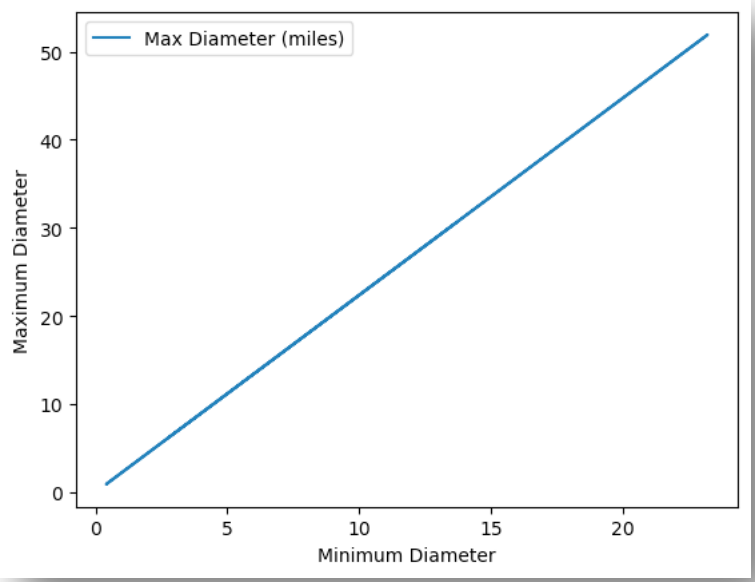


Figure 15: Min & Max Diameter are directly correlated to each other.

Instead of having two features which are directly correlated, I have decided to calculate the average between the minimum and maximum diameter.

```
neo_df['Average Diameter (miles)'] = (neo_df['Max Diameter (miles)']+neo_df['Min Diameter (miles)'])/2
```

[52] ✓ 0.0s Python

```
neo_df[['Max Diameter (miles)', 'Min Diameter (miles)', 'Average Diameter (miles)']].head()
```

[53] ✓ 0.0s Python

	Max Diameter (miles)	Min Diameter (miles)	Average Diameter (miles)
0	30.576724	13.674327	22.125526
1	30.576724	13.674327	22.125526
2	30.576724	13.674327	22.125526
3	30.576724	13.674327	22.125526
4	30.576724	13.674327	22.125526

Figure 16: Min, Max Diameter, and new Average Diameter features side by side.

Normalizing The Data

Additionally, let us try to normalize the data in these columns. Normalization is the process of changing the scale of the data. Machine learning models do not work very well with large values. If the values are very large, then the coefficients of the model can also be large and at times inaccurate. The models work best when the data is between 0 and 1. Therefore if we convert the existing data and scale them to between 0 and 1. This is also known as feature scaling.

Scaling the features of all the columns except the following:

- Orbit Earth
- Orbit Juptr
- Orbit Mars
- Orbit Merc
- Orbit Venus

Reason being the above columns are contain binary data, and do not require feature scaling.

```
# Normalize all the features
from sklearn import preprocessing
min_max_scalar = preprocessing.MinMaxScaler()
neo_scaled = min_max_scalar.fit_transform(neo_df.drop(columns=['Orbit Earth',
                                                               'Orbit Juptr',
                                                               'Orbit Mars',
                                                               'Orbit Merc',
                                                               'Orbit Venus']))
neo_df_scaled = pd.DataFrame(neo_scaled)
neo_df_scaled.head()
```

[56] ✓ 0.0s

	0	1	2	3	4	5
0	0.0	0.13083	0.002668	0.056507	0.155560	0.581538
1	0.0	0.13083	0.025467	0.027606	0.234985	0.581538
2	0.0	0.13083	0.056921	0.037913	0.249075	0.581538
3	0.0	0.13083	0.079782	0.032525	0.178317	0.581538
4	0.0	0.13083	0.102737	0.064860	0.084099	0.581538

Figure 17: Min Max Scalar.

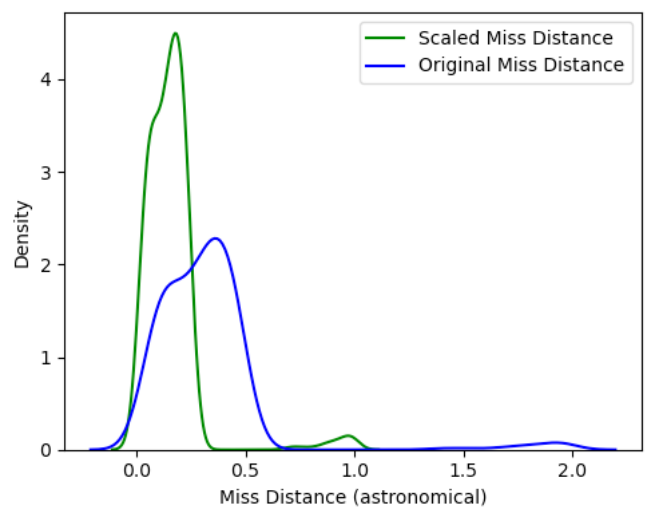


Figure 18: Plot displaying the normalized & original values of the Miss Distance.

The above graph shows the scaled values of the Miss Distance as compared to the original values. The KDE (Kernel Density Estimate) plot is useful to represent distribution of observations in a dataset. The green KDE plot shows the scaled values between 0 to 1 and the blue KDE plot shows the original miss distance value in astronomical units.

Training & Testing Split

In this dataset we do not have too many instances for training and testing since it has only 904 records. Therefore I will have decided to consider the train & test split as 80 - 20 respectively.

```
# Train and test split

# creating a function which splits the data randomly
def split_train_test(data, test_ratio):
    np.random.seed(23) # setting the random number generator's seed for generating the same test & train sets each time.
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

[140] ✓ 0.0s

```
# In this dataset we do not have too many instances to test on - since it has only 904 records.
# therefore I will have the train & test split as 80 - 20 respectively

neo_df_train, neo_df_test = split_train_test(neo_df_scaled, 0.2)
```

[141] ✓ 0.0s

Figure 19: Train & Test Split function.

K-Means Clustering

Now the data has been prepared, scaled and ready to run the K-Means clustering algorithm. K-Means clustering is a method in which we try to find clusters based on the distance between the points which are close to each other. However, as the dimensionality increases, it gets much more complex to be able to classify points in different clusters. In this dataset, as displayed on Figure 20 below, we can see that we have a total of 11 columns. This leads me to worry about the curse of dimensionality.

```
[70]  neo_df_train.info()
      ✓ 0.0s

...   <class 'pandas.core.frame.DataFrame'>
      Index: 724 entries, 530 to 595
      Data columns (total 11 columns):
       #   Column                        Non-Null Count  Dtype
      ---  ------                        --------------  -----
       0   Designation                   724 non-null    float64
       1   Absolute Magnitude (H)        724 non-null    float64
       2   Epoch Date Close Approach     724 non-null    float64
       3   Relative Velocity (km/s)      724 non-null    float64
       4   Miss Distance (astronomical)  724 non-null    float64
       5   Average Diameter (miles)      724 non-null    float64
       6   Orbit Earth                   724 non-null    float64
       7   Orbit Juptr                   724 non-null    float64
       8   Orbit Mars                    724 non-null    float64
       9   Orbit Merc                    724 non-null    float64
       10  Orbit Venus                   724 non-null    float64
      dtypes: float64(11)
      memory usage: 67.9 KB
```

Figure 20: Final Set of Features.

Another challenge is that when we have a data which has a lot of features, it also gets increasingly difficult to visualize them in 2-D plots or graphs. Therefore, after training the model, we will try to visualize the clusters by taking the permutations and combinations of different features. This will help us to make a better sense of the clusters.

Let us try to train the unsupervised model first to find two clusters first. I chose the k=2 value because in the supervised learning assignment, the target variable had 2 values – Is Potentially Dangerous and Is Not Potentially Dangerous.

```
# import KMeans
from sklearn.cluster import KMeans

# initially let us start by creating 2 clusters -
kmeans_neo = KMeans(n_clusters=2)
kmeans_neo.fit(neo_df_train)
```

[72] ✓ 0.1s Python

```
KMeans
KMeans(n_clusters=2)
```

Figure 21: K Means model with k = 2.

Visualizing the Clusters

Since our model has more than 2 features, we cannot visualize it in a 2-dimentional x-y graph. Therefore, in order to visualize the same, I have considered two different features each time:

Miss Distance vs Average Diameter

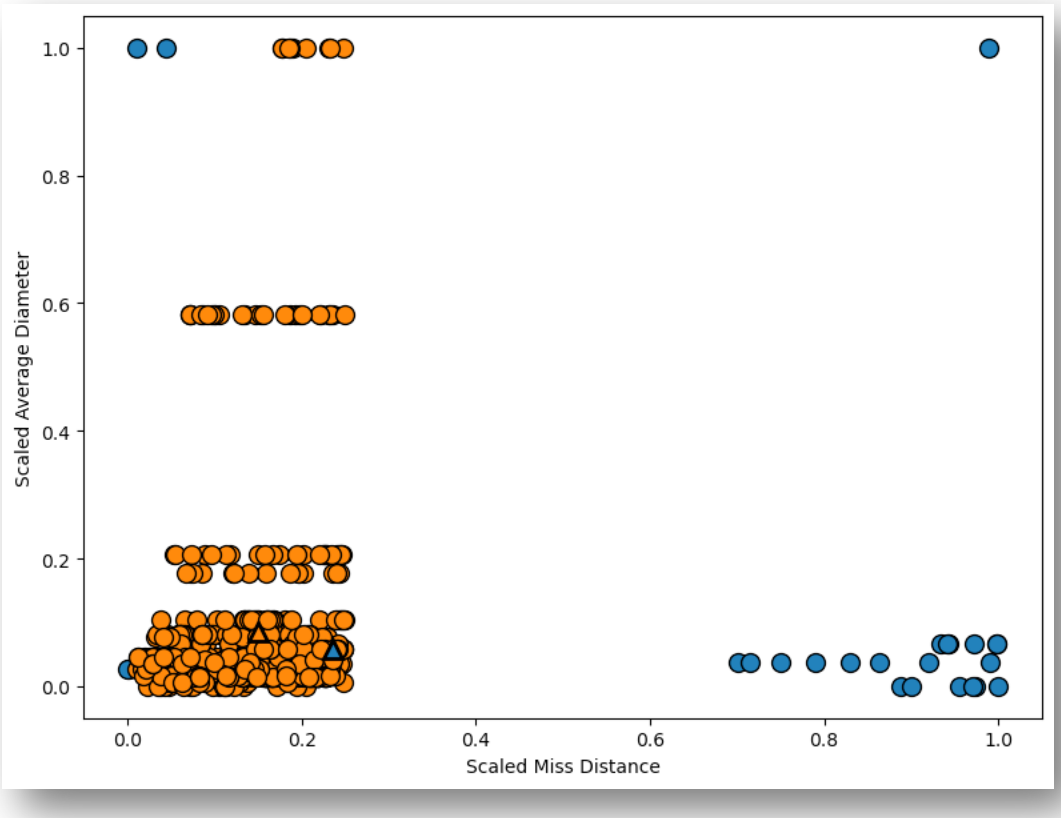


Figure 22: Visualization of K Means model with k = 2.

From the graph above we can see two distinct groups. The points which are near the origin are most likely classified in one cluster (with a few exceptions) and the ones which are having high values of

scaled miss distance are classified in another cluster. The centroids of the graphs have been plotted with a triangle shape.

From my understanding, since the K-Means model is more complex than just these two features, the 2-D representation is not enough to see the separation of the clusters. However, the above features do look like they have been separated well for the most part – except for a few blue points, and the centroid of the blue cluster.

Relative Velocity vs Miss Distance

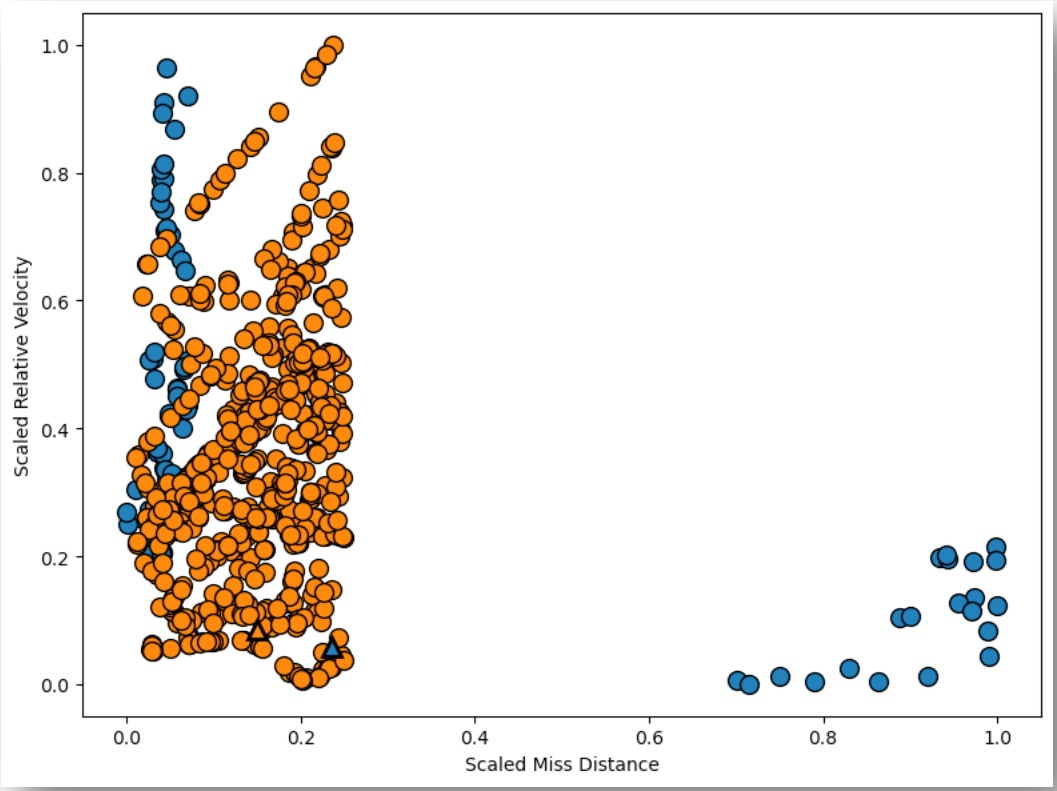


Figure 23: Visualization of K Means model with k = 2.

In the above figure, we can see distinctly two clusters. It looks like a lot of the points in the orange cluster are correctly getting classified. However, there are some points in blue which do not look correctly classified. As mentioned in the previous graph, we cannot say whether these points are correctly clustered, because the K-Means algorithm was trained on data which has a much higher dimension than 2-D.

From figures 22 and 23, we can see that a much larger set of values are in the orange cluster as compared to the blue cluster.

Evaluating K-Means Model with Inertia

There are different ways to evaluate clustering algorithms. Inertia is one such method which calculates a summation of the distance between the points and the centroid of the respective cluster. The figure 24 below shows the inertia for the model where k=2.

```
kmeans_neo.inertia_
[128]  ✓  0.0s
...    244.28538406631614
```

Figure 24: Inertia of K Means model with k = 2

Let us try to find out which value of k will be better with respect to the inertia. We know that the lower the inertia model will be the best model, as the cluster points will be very close to the cluster centroid.

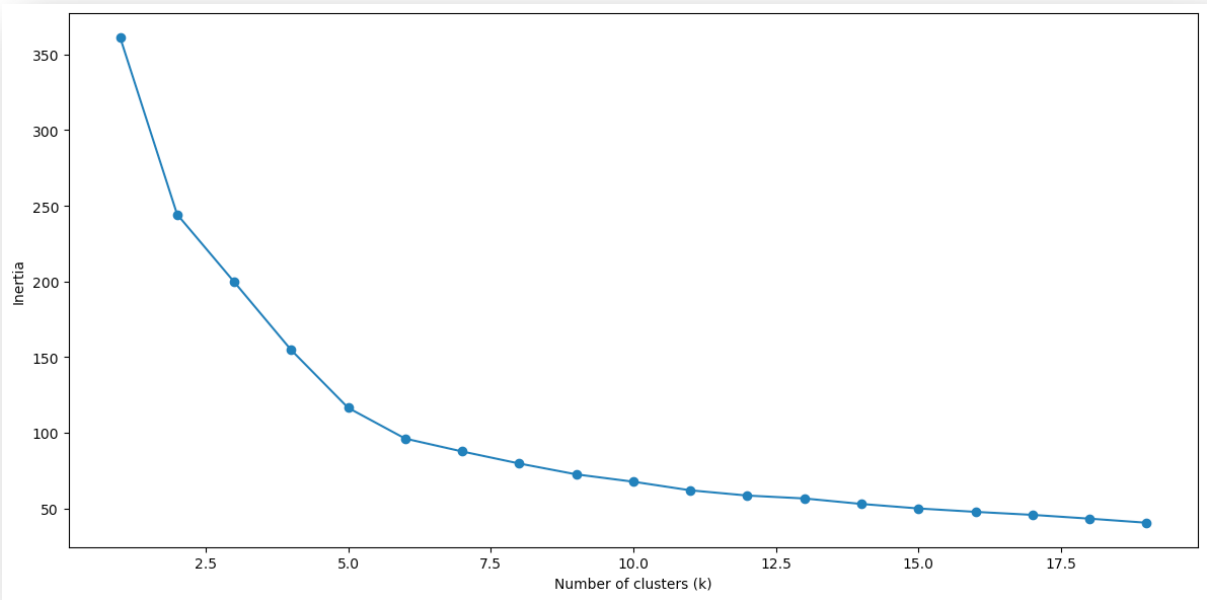


Figure 25: Number of clusters vs Inertia.

We can use the elbow-method to take the best value of k. From the figure above, we can see that k = 6 is a good value of k. Therefore let us retrain our model and visualize the same features as we did previously.

We have trained the K-Means model again with k=6, and now let us try to compare the visualization of the same features which we did when k =2.

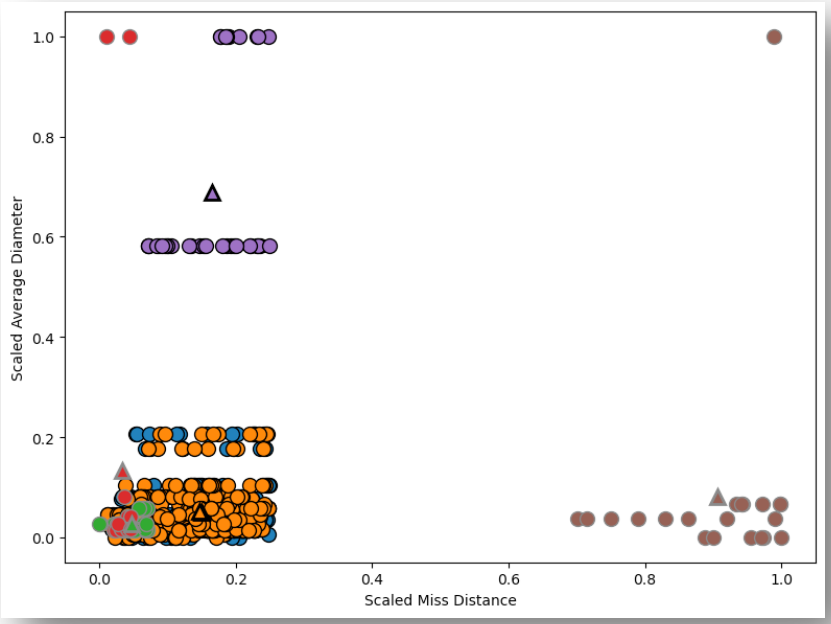


Figure 26: Visualization of K Means model with k = 6.

From the above plot we can see that there are a few distinct clusters – for example the brown, purple, and the orange cluster. However, we can see that the blue, red and green clusters are very much close to the orange cluster.

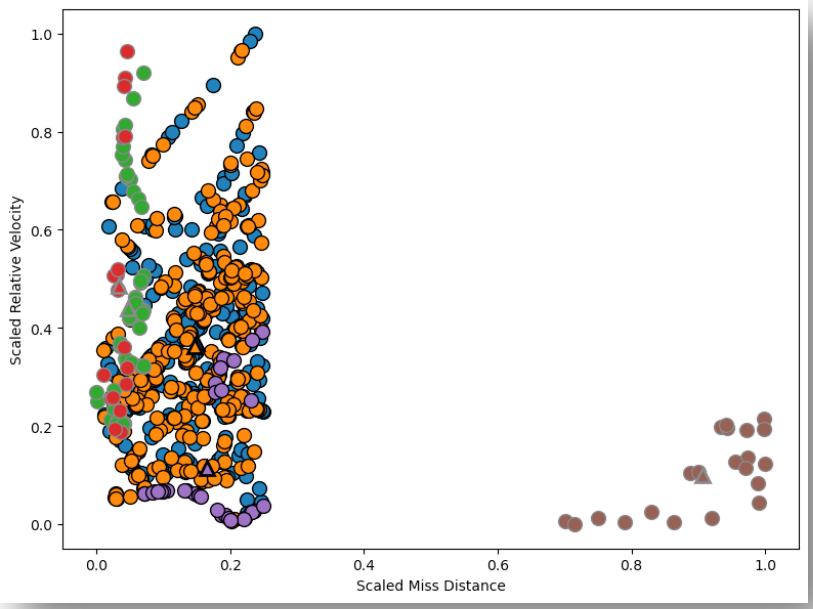


Figure 27: Visualization of K Means model with k = 6.

While viewing the Scaled Miss Distance vs Scaled Relative Velocity columns, we can see that mostly the brown cluster is distinct. There could be a possibility that in a higher dimension, a hyper-plane would be able to separate out the clusters well.

When k=6, the value of the computed inertia is around 96 units.

Gaussian Mixed Models (GMM)

There is one issue with K-Means clustering method which is that it looks at the average/ mean distance of all the points from their respective centroid. However, it does not include the variance. GMMs are probability distribution models which take into consideration both the mean and the standard deviation. Let us train a GMM model for the same dataset with value of n_estimators = 2 first.

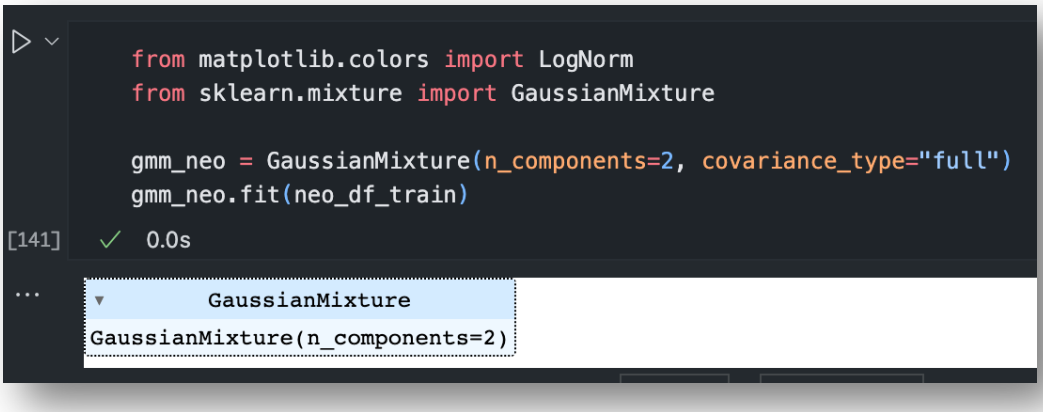


Figure 28: GMM with n_estimators = 2.

Let us try to again visualize how the model has classified the data using the same features that we used in the k-Means model. Since GMM is a probability distribution model, we must plot the Density estimation based on the negative log-likelihood.

Principle Component Analysis (PCA)

When we are training lots of models – especially models which rely on their neighbor to make predictions, the curse of dimensionality is something to always take into consideration. When there are lots of features, the dimension of the data is high, and therefore the model becomes very complex. In this case our dataset has 11 features, therefore it has 11 dimensions. This is too high; however, we cannot just remove features as they may be important for the models. In such cases, there are methods in which we can reduce the dimensionality of data without losing many important features. I will be using PCA to precisely reduce the model dimensionality.

```
from sklearn.decomposition import PCA
pca_neo = PCA(n_components=2)
pca_neo.fit(neo_df_train)
X_pca_neo = pca_neo.transform(neo_df_train)

print("Original shape: {}".format(str(neo_df_train.shape)))
print("Reduced shape: {}".format(str(X_pca_neo.shape)))
```

[176] ✓ 0.0s

```
Original shape: (724, 11)
Reduced shape: (724, 2)
```

Figure 29: PCA model

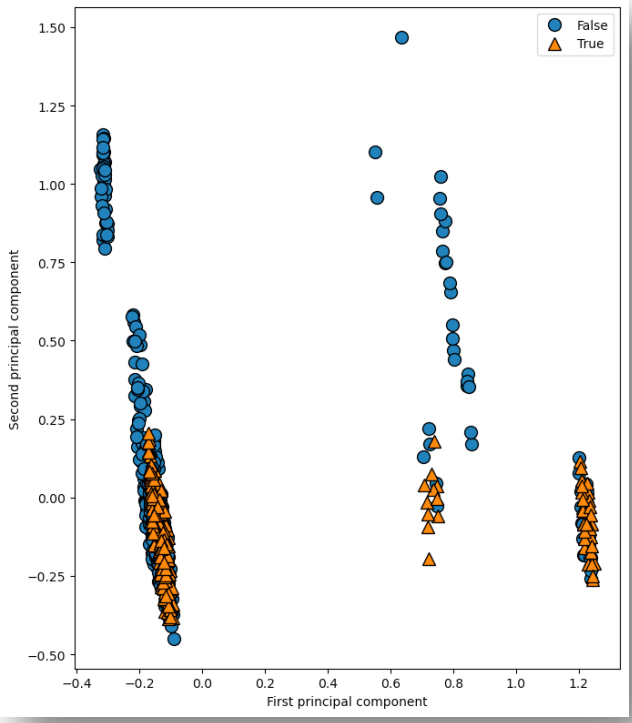


Figure 29: PCA model with values from the target variable – 'Is Potentially Hazardous'

We can see that although the PCA model reduced the dimensionality, it does not exactly coincide with the target value.

Independent Component Analysis (ICA)

Another algorithm to for dimensionality reduction is ICA. Using ICA let us first try to reduce the dimensions from 11 down to 2 and visualize how the data looks like.

```
from sklearn.decomposition import FastICA
ica_neo = FastICA(n_components=2, random_state=12)
X_ica =ica_neo.fit_transform(neo_df_train.values)

print("Original shape: {}".format(str(neo_df_train.shape)))
print("Reduced shape: {}".format(str(X_ica.shape)))
```

[198] ✓ 0.0s

```
Original shape: (724, 11)
Reduced shape: (724, 2)
```

Figure 30: ICA model with n_components = 2.

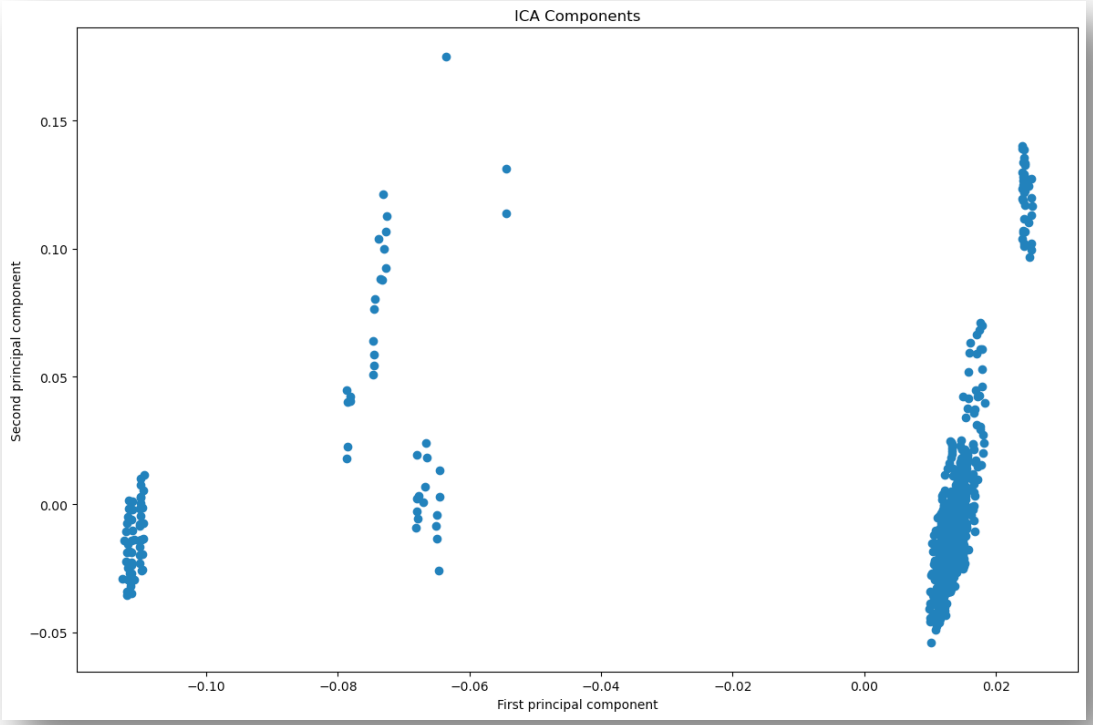


Figure 31: Visualizing ICA model.

Let us try to see how the plots will look like if we increase the value of n components and furthermore let us visualize. Please find below the figure 30 and 31 wherein n = 3.

```
    ica_neo_3 = FastICA(n_components=3, random_state=12)
    X_ica_3 =ica_neo_3.fit_transform(neo_df_train.values)

    print("Original shape: {}".format(str(neo_df_train.shape)))
    print("Reduced shape: {}".format(str(X_ica_3.shape)))
[203]  ✓  0.0s

...  Original shape: (724, 11)
     Reduced shape: (724, 3)
```

Figure 32: ICA model with n_components = 3

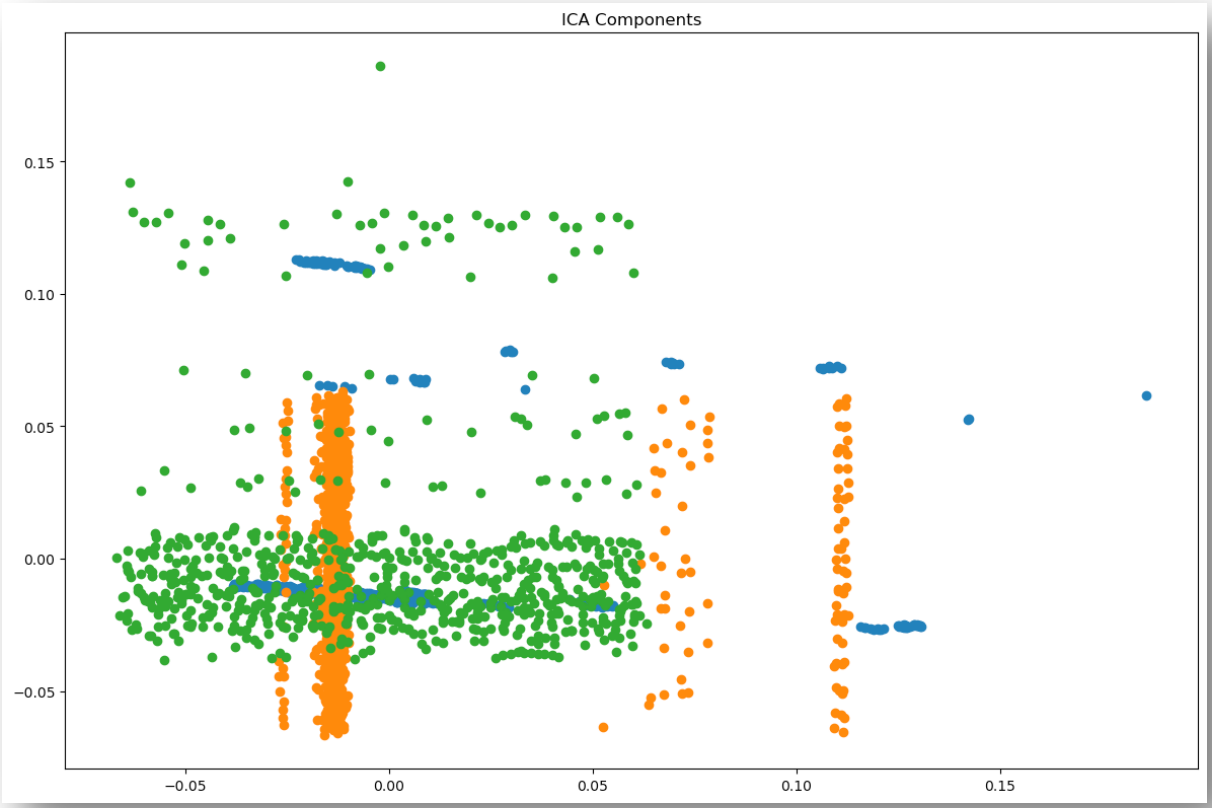


Figure 33: Visualizing the ICA model.

From the above figure 33, we can see that the ICA Is trying to group together data in the vertical axis as orange, in the horizontal axis as green. There is also a group of blue which looks like it must be representing data along a 3rd axis.

Re-running Clustering Algorithms After Feature Reduction

K-Means Clustering after PCA

Since previously for PCA dimensionality reduction algorithm we reduced the dimensions from 11 down to 2, let us run the k-means algorithm such that value of k = 2.

```
    # X_pca_neo
    pca_kmeans_neo = KMeans(n_clusters=2)
    pca_kmeans_neo.fit(X_pca_neo)
    # here we are training the data on the result of the PCA algorithm.
[205]  ✓  0.0s
...     KMeans
        KMeans(n_clusters=2)
```

Figure 34: K-Means followed by PCA.

Let us now visualize how the clusters have been formed for this data.

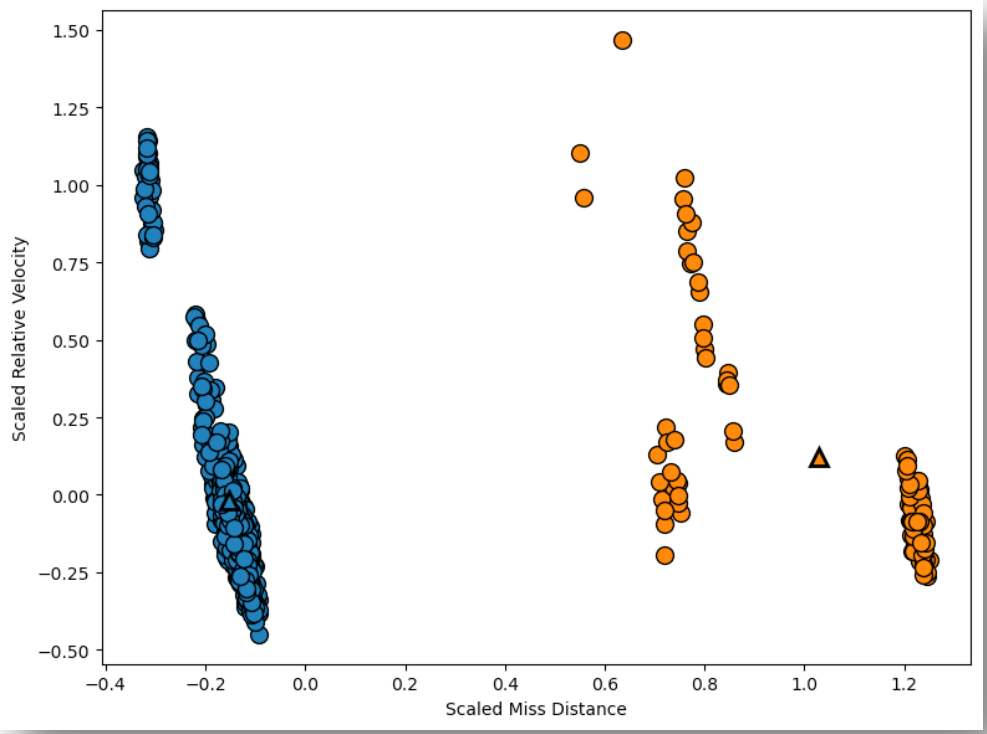


Figure 35: Visualizing K-Means followed by PCA.

The figure 35 shows that our K-Means model has correctly clustered data which are near each other into orange and blue colors. Notice the triangle in both orange and blue – denoting that it is the centroid of the data points. Undoubtedly this this model is much better than the highly complex K-Means model which we had trained on 11 features previously. After running the PCA, when the dimension was down to 2 features, the K-Means model has performed significantly better.

Re-running Neural Network Model from Assignment 2

For the dataset #1, we made a few changes by performing feature engineering and feature scaling. Therefore, we will re-run the neural network model (created in Assignment 2) again first. With this as a benchmark, we will perform feature reduction and then re-run the neural network model.

Neural Network Model

```
import tensorflow as tf
from tensorflow import keras

ann_model = keras.models.Sequential()
ann_model.add(keras.layers.Dense(50, activation="relu"))
ann_model.add(keras.layers.Dense(25, activation="relu"))
ann_model.add(keras.layers.Dense(10, activation="relu"))
ann_model.add(keras.layers.Dense(5, activation="relu"))
ann_model.add(keras.layers.Dense(1, activation="sigmoid"))
```

[212] ✓ 9.1s Python

```
# similar to the above decision tree example, we are adding the loss to be binary cross en
ann_model.compile(loss="binary_crossentropy", optimizer="adam", metrics=['accuracy'])

# now running the training on the ANNs
# we will train the neural network 20 times with a batch size of 5.
ann_model.fit(neo_df_train, neo_df_target_train, epochs=20, batch_size=5)
```

[213] ✓ 2.3s Python

```
loss, accuracy = ann_model.evaluate(neo_df_train, neo_df_target_train)
print(accuracy)
```

[214] ✓ 0.1s Python

```
23/23 [==============================] - 0s 533us/step - loss: 0.4820 - accuracy: 0.7970
0.7969613075256348
```

The ANN model has a training accuracy of 79.69%. We can see that in the previous assignment 2 – the training accuracy of the ANN model was just 69.75%. Therefore by performing feature engineering and feature scaling, the neural network model has improved.

From the below figure we can see that the testing accuracy is 82.22%. This is also higher than the test accuracy of 68.88% which was seen in Assignment 2.

```
test_loss, test_accuracy = ann_model.evaluate(neo_df_test, neo_df_target_test)
test_accuracy
```

[215] ✓ 0.0s Python

```
6/6 [==============================] - 0s 865us/step - loss: 0.4597 - accuracy: 0.8222

0.8222222328186035
```

Running Neural Network Model on Just PCA Data

```
# X_pca_neo
pca_ann_model = keras.models.Sequential()
pca_ann_model.add(keras.layers.Dense(50, activation="relu"))
pca_ann_model.add(keras.layers.Dense(25, activation="relu"))
pca_ann_model.add(keras.layers.Dense(10, activation="relu"))
pca_ann_model.add(keras.layers.Dense(5, activation="relu"))
pca_ann_model.add(keras.layers.Dense(1, activation="sigmoid"))
```

[218] ✓ 0.0s Python

```
# similar to the above decision tree example, we are adding the loss to be binary cross en
pca_ann_model.compile(loss="binary_crossentropy", optimizer="adam", metrics=['accuracy'])

# now running the training on the ANNs
# we will train the neural network 20 times with a batch size of 5.
pca_ann_model.fit(X_pca_neo, neo_df_target_train, epochs=20, batch_size=5)
```

[219] ✓ 2.0s Python

```
loss_pca_ann, accuracy_pca_ann = pca_ann_model.evaluate(X_pca_neo, neo_df_target_train)
print(accuracy_pca_ann)
```

[220] ✓ 0.0s Python

```
23/23 [==============================] - 0s 517us/step - loss: 0.5496 - accuracy: 0.7182
0.7182320356369019
```

This ANN model has a training accuracy of 71.82%. Comparing this with the previous neural network model (which had a score 79.69%), this model has a lower training accuracy score. However, from assignment 2, this score is still better than 69.75%. Also as seen in the figure below, the testing accuracy of just the PCA + ANN model is 69.99%. Similarly this is not better than the benchmarked ANN model previously, however it is better than the ANN model which was created as a part of Assignment 2.

```
test_loss_pca_ann, test_accuracy_pca_ann = pca_ann_model.evaluate(X_pca_neo_test, neo_df_t
test_accuracy_pca_ann
```

[224] ✓ 0.0s Python

```
6/6 [==============================] - 0s 867us/step - loss: 0.5882 - accuracy: 0.7000

0.699999988079071
```

Running Neural Network Model on All Features + K Means Cluster ID (k=6)

```
kmeans_neo_six_train_labels = neo_df_train
kmeans_neo_six_train_labels['label'] = kmeans_neo_six.labels_
kmeans_neo_six_train_labels.info()
```

[225] ✓ 0.0s Python

```
<class 'pandas.core.frame.DataFrame'>
Index: 724 entries, 530 to 595
Data columns (total 12 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   Designation                  724 non-null    float64
 1   Absolute Magnitude (H)       724 non-null    float64
 2   Epoch Date Close Approach    724 non-null    float64
 3   Relative Velocity (km/s)     724 non-null    float64
 4   Miss Distance (astronomical) 724 non-null    float64
 5   Average Diameter (miles)     724 non-null    float64
 6   Orbit Earth                  724 non-null    float64
 7   Orbit Juptr                  724 non-null    float64
 8   Orbit Mars                   724 non-null    float64
 9   Orbit Merc                   724 non-null    float64
 10  Orbit Venus                  724 non-null    float64
 11  label                        724 non-null    int32
dtypes: float64(11), int32(1)
memory usage: 86.9 KB
```

```
# training the ANN

k_6_ann_model = keras.models.Sequential()
k_6_ann_model.add(keras.layers.Dense(50, activation="relu"))
k_6_ann_model.add(keras.layers.Dense(25, activation="relu"))
k_6_ann_model.add(keras.layers.Dense(10, activation="relu"))
k_6_ann_model.add(keras.layers.Dense(5, activation="relu"))
k_6_ann_model.add(keras.layers.Dense(1, activation="sigmoid"))

# similar to the above decision tree example, we are adding the loss to be binary cross en
k_6_ann_model.compile(loss="binary_crossentropy", optimizer="adam", metrics=['accuracy'])

# now running the training on the ANNs
# we will train the neural network 20 times with a batch size of 5.
k_6_ann_model.fit(kmeans_neo_six_train_labels, neo_df_target_train, epochs=20, batch_size=
```

[226] ✓ 2.3s Python

```
loss_k_6, accuracy_k_6 = k_6_ann_model.evaluate(kmeans_neo_six_train_labels, neo_df_target
print(accuracy_k_6)
```

[228] ✓ 0.0s Python

```
23/23 [==============================] - 0s 491us/step - loss: 0.3858 - accuracy: 0.8135
0.8135359287261963
```

The ANN model containing all the features + the K-Means Cluster ID has a training accuracy of 81.35%. This is the best training accuracy that we have seen so far compared to previous neural network models.

```
test_loss_k_6, test_accuracy_k_6 = k_6_ann_model.evaluate(kmeans_neo_six_test_labels, neo_
test_accuracy_k_6
```

[233] ✓ 0.0s Python

```
6/6 [==============================] - 0s 911us/step - loss: 0.4284 - accuracy: 0.8000

0.800000011920929
```

From the test accuracy of 80.00%, we can see that this is the best test accuracy from the previous neural network models.

We can see that the ANN model which was trained on all the features along with the Cluster ID of the K-Means outperforms all the other ANN models in both training and testing accuracy. Apart from the accuracy performance, I noticed that the while running time of the neural network models was the same.

Dataset #2 – Heart Attack Risk Prediction

Understanding the Data

In this Heart Attack Risk Prediction dataset, there are a total of 8763 rows. From below figures we can see that for each column there are 8763 non-null rows. Additionally, this data set does not contain any columns with null or NaN data values.

```
# Fist 5 rows
heart_df.head()
```

[6] Python

	Patient ID	Age	Sex	Cholesterol	Blood Pressure	Heart Rate	Diabetes	Family History	Smoking	Obesity	...	Sedentary Hours Per Day	Income	BMI	Triglycerides	Physical Activity Days Per Week	Sleep Hours Per Day	Cou
0	BMW7812	67	Male	208	158/88	72	0	0	1	0	...	6.615001	261404	31.251233	286	0	6	Arger
1	CZE1114	21	Male	389	165/93	98	1	1	1	1	...	4.963459	285768	27.194973	235	1	7	Car
2	BNI9906	21	Female	324	174/99	72	1	0	0	0	...	9.463426	235282	28.176571	587	4	4	Fra
3	JLN3497	84	Male	383	163/100	73	1	1	1	0	...	7.648981	125640	36.464704	378	3	4	Car
4	GFO8847	66	Male	318	91/88	93	1	1	1	1	...	1.514821	160555	21.809144	231	1	5	Thai

5 rows × 26 columns

```
heart_df.info()
```

[93] ✓ 0.0s

```
RangeIndex: 8763 entries, 0 to 8762
Data columns (total 26 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Patient ID                      8763 non-null   object
 1   Age                             8763 non-null   int64
 2   Sex                             8763 non-null   object
 3   Cholesterol                     8763 non-null   int64
 4   Blood Pressure                  8763 non-null   object
 5   Heart Rate                      8763 non-null   int64
 6   Diabetes                        8763 non-null   int64
 7   Family History                  8763 non-null   int64
 8   Smoking                         8763 non-null   int64
 9   Obesity                         8763 non-null   int64
 10  Alcohol Consumption             8763 non-null   int64
 11  Exercise Hours Per Week         8763 non-null   float64
 12  Diet                            8763 non-null   object
 13  Previous Heart Problems         8763 non-null   int64
 14  Medication Use                  8763 non-null   int64
 15  Stress Level                    8763 non-null   int64
 16  Sedentary Hours Per Day         8763 non-null   float64
 17  Income                          8763 non-null   int64
 18  BMI                             8763 non-null   float64
 19  Triglycerides                   8763 non-null   int64
 20  Physical Activity Days Per Week 8763 non-null   int64
 21  Sleep Hours Per Day             8763 non-null   int64
 22  Country                         8763 non-null   object
 23  Continent                       8763 non-null   object
 24  Hemisphere                      8763 non-null   object
 25  Heart Attack Risk               8763 non-null   int64
```

As we explored this dataset in the previous assignment, we will not be performing any exploratory data analysis on this dataset.

Preparing The Data for the Unsupervised Learning Tasks

Removing the Target Variable

```
heart_df_label = heart_df["Heart Attack Risk"]
heart_df.drop(columns=['Heart Attack Risk'], inplace=True)
heart_df_label
```

[246] ✓ 0.0s Python

```
0       0
1       0
2       0
3       0
4       0
       ..
8758    0
8759    0
8760    1
8761    0
8762    1
Name: Heart Attack Risk, Length: 8763, dtype: int64
```

Updating Categorical Data

Gender

```
# now performing One Hot Encoding on Gender column

label_encoder_heart = LabelEncoder()
gender_integer_encoded = label_encoder_heart.fit_transform(heart_df['Sex'])
gender_integer_encoded = gender_integer_encoded.reshape(len(gender_integer_encoded), 1)
# above we are reshaoing the array such that it forms a vector
print(gender_integer_encoded)

one_hot_encoder_heart = OneHotEncoder(sparse=False)
gender_one_hot_encoded = one_hot_encoder_heart.fit_transform(gender_integer_encoded)
print(gender_one_hot_encoded)

# one_hot_encoder = OneHotEncoder(sparse=False)
# orbit_body_one_hot_encoded = one_hot_encoder.fit_transform(orbit_integer_encoded)
# orbit_body_one_hot_encoded
```

[247] ✓ 0.0s Python

```
[[1]
 [1]
 [0]
 ...
 [1]
 [1]
 [0]]
[[0. 1.]
 [0. 1.]
 [1. 0.]
 ...
 [0. 1.]
 [0. 1.]
 [1. 0.]]
```

The machine learning models cannot understand string or text values; therefore we must encode the data. I chose OneHotEncoder over the LabelEncoder for the gender column. This is because, the label encoder will change string to a numerical value for example Female – 0 and Male – 1. However, the model may misinterpret the value of this data. However this will not occur using OneHotEncoders.

Data Cleaning on Blood Pressure Column

```
# data cleaning for blood pressure

# converting separate these two values into two columns – 'Blood Pressure Systolic' & 'Blood Pressure Diastolic'
heart_df[['Blood Pressure Systolic', 'Blood Pressure Diastolic']] = heart_df['Blood Pressure'].str.split('/', expand=True)
# expand = True: Expands the split strings into separate columns
```

[251] ✓ 0.0s Python

```
heart_df[['Blood Pressure', 'Blood Pressure Systolic', 'Blood Pressure Diastolic']].head()
```

[252] ✓ 0.0s Python

	Blood Pressure	Blood Pressure Systolic	Blood Pressure Diastolic
0	158/88	158	88
1	165/93	165	93
2	174/99	174	99
3	163/100	163	100
4	91/88	91	88

Once we do the data cleaning, as shown in the above image, we can drop the Blood Pressure column from the data frame.

Location Data

In this dataset, we have the location of the patients which are their country, continent and hemisphere. We cannot use One Hot Encoding for such type of values because each of these columns have multiple types of categorical data. If we used One Hot Encoding, then the model will become very complex. Therefore as shown below, I have used Label Encoders.

```
# Let us use Label Encoders to encode the data for these three columns
label_encoder_country = LabelEncoder()
heart_df['Country'] = label_encoder_country.fit_transform(heart_df['Country'])

label_encoder_continent = LabelEncoder()
heart_df['Continent'] = label_encoder_continent.fit_transform(heart_df['Continent'])

label_encoder_hemisphere = LabelEncoder()
heart_df['Hemisphere'] = label_encoder_hemisphere.fit_transform(heart_df['Hemisphere'])
```

[262] ✓ 0.0s Python

Train & Test Split

We are splitting the training and the testing data by 25% as shown below using the train_test_split function provided by sci-kit learn library.

```
from sklearn.model_selection import train_test_split

heart_X_train, heart_X_test, heart_y_train, heart_y_test = train_test_split(heart_df, heart_df_label, random_state=0)
# by default setting the train size to be 25 %
```

[263] ✓ 0.0s Python

Add Markdown Cell

Running Unsupervised Learning Models

Based on the below figure, we can see that our model which has 26 columns. This is too large to run an unsupervised learning model. Just because of such large dimensionality, the model will be very complex and it will fail due to the curse of dimensionality.

```
heart_X_train.info()
```

[284] ✓ 0.0s Python

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6572 entries, 0 to 6572
Data columns (total 26 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   Age                             6572 non-null   int64  
 1   Cholesterol                     6572 non-null   int64  
 2   Heart Rate                      6572 non-null   int64  
 3   Diabetes                        6572 non-null   int64  
 4   Family History                  6572 non-null   int64  
 5   Smoking                         6572 non-null   int64  
 6   Obesity                         6572 non-null   int64  
 7   Alcohol Consumption             6572 non-null   int64  
 8   Exercise Hours Per Week         6572 non-null   float64
 9   Diet                            6572 non-null   float64
 10  Previous Heart Problems         6572 non-null   int64  
 11  Medication Use                  6572 non-null   int64  
 12  Stress Level                    6572 non-null   int64  
 13  Sedentary Hours Per Day         6572 non-null   float64
 14  Income                          6572 non-null   int64  
 15  BMI                             6572 non-null   float64
 16  Triglycerides                   6572 non-null   int64  
 17  Physical Activity Days Per Week 6572 non-null   int64  
 18  Sleep Hours Per Day             6572 non-null   int64  
 19  Country                         6572 non-null   int64  
 20  Continent                       6572 non-null   int64  
 21  Hemisphere                      6572 non-null   int64  
 22  Blood Pressure Systolic         6572 non-null   int64  
 23  Blood Pressure Diastolic        6572 non-null   int64  
 24  Is Female                       6572 non-null   float64
 25  Is Male                         6572 non-null   float64
dtypes: float64(6), int64(20)
memory usage: 1.4 MB
```

Therefore, before we can run any unsupervised model on this dataset we must run some dimensionality reduction algorithms.

PCA

```
pca_heart = PCA(n_components=2)
pca_heart.fit(heart_X_train)
X_pca_heart = pca_heart.transform(heart_X_train)

print("Original shape: {}".format(str(heart_X_train.shape)))
print("Reduced shape: {}".format(str(X_pca_heart.shape)))
```

[287] ✓ 0.1s Python

```
Original shape: (6572, 26)
Reduced shape: (6572, 2)
```

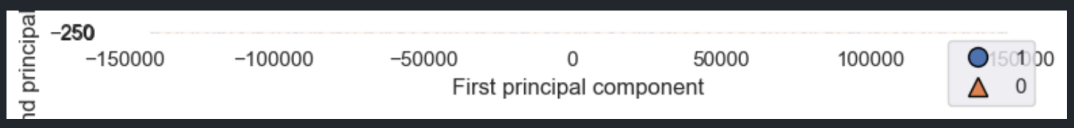
We have now reduced the features from 26 down to 2.

However, plotting the graph does not show any results as we can see below. This is because before we reduced the dimension of the data down to 2, we dd not perform any feature scaling. Now the data values are all very small numbers, hence it is not showing up on the graph.

```
# plotting the first vs. second principal component, colored by target class
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca_heart[:, 0], X_pca_heart[:, 1], heart_y_train)
plt.legend(heart_y_train.unique(), loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

[289] ✓ 0.1s Python

```
Text(0, 0.5, 'Second principal component')
```




```
X_pca_heart
```

[291] ✓ 0.0s Python

```
array([[ 1.15924750e+05,  1.35363367e+02],
       [-7.47252455e+04, -5.17223073e+01],
       [ 8.61667576e+04, -3.74600880e+02],
       ...,
       [-8.69602517e+04,  3.69797228e+02],
       [-1.37331246e+05, -1.61826738e+02],
       [-1.35756252e+05,  3.61400876e+02]])
```

Upon doing the feature scaling, we will get better graph for the PCA data.

Conclusion

In this assignment we covered the different clustering algorithms like K-Means and GMM and how we can use them to segment or club together different data-points. We also covered dimensionality reduction techniques like PCA and ICA. Furthermore, we used the Cluster ID from the K-Means clustering algorithm as an input to our supervised learning model and noticed it has the best accuracy of 80% in both training and testing.

References

1. Verma, J. (2023, February 9). *How to normalize data using scikit-learn in Python*. DigitalOcean. https://www.digitalocean.com/community/tutorials/normalize-data-in-python
2. Team, G. L. (2023, November 8). *Label encoding in python - 2024*. Great Learning Blog: Free Resources what Matters to shape your Career! https://www.mygreatlearning.com/blog/label-encoding-in-python/#:~:text=Label%20encoding%20is%20a%20technique,only%20operate%20on%20numerical%20data.
3. Brownlee, J. (2020, June 30). *Why one-hot encode data in machine learning?*. MachineLearningMastery.com. https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/
4. Brownlee, J. (2019, August 14). *How to one hot encode sequence data in python*. MachineLearningMastery.com. https://machinelearningmastery.com/how-to-one-hot-encode-sequence-data-in-python/
5. *4 ways to add a column in Pandas*. Built In. (n.d.). https://builtin.com/data-science/pandas-add-column
6. *Data to fish*. Data to Fish. (n.d.). https://datatofish.com/count-nan-pandas-dataframe/
7. Long, A. (2020, February 2). *Machine learning with Datetime feature engineering: Predicting Healthcare appointment no-shows*. Medium. https://towardsdatascience.com/machine-learning-with-datetime-feature-engineering-predicting-healthcare-appointment-no-shows-5e4ca3a85f96
8. *Pandas.to_datetime#*. pandas.to_datetime - pandas 2.1.3 documentation. (n.d.). https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html
9. *Datetime - basic date and time types*. Python documentation. (n.d.). https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior
10. *Pandas.dataframe.plot#*. pandas.DataFrame.plot - pandas 2.1.3 documentation. (n.d.). https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html
11. *Choosing colormaps#*. Choosing Colormaps - Matplotlib 3.8.2 documentation. (n.d.). https://matplotlib.org/stable/users/explain/colors/colormaps.html
12. *6.3. Preprocessing Data*. scikit. (n.d.). https://scikit-learn.org/stable/modules/preprocessing.html#scaling-features-to-a-range
13. *Pandas.dataframe.rename#*. pandas.DataFrame.rename - pandas 2.1.3 documentation. (n.d.). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rename.html
14. *Seaborn.kdeplot#*. seaborn.kdeplot - seaborn 0.13.0 documentation. (n.d.). https://seaborn.pydata.org/generated/seaborn.kdeplot.html#seaborn.kdeplot
15. *Visualizing distributions of data#*. Visualizing distributions of data - seaborn 0.13.0 documentation. (n.d.). https://seaborn.pydata.org/tutorial/distributions.html
16. Sharma, P. (2023, November 3). *The Ultimate Guide to K-means clustering: Definition, methods and applications*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/#Implementing_K-Means_Clustering_in_Python
17. *Sklearn.cluster.kmeans*. scikit. (n.d.-b). https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
18. *Sklearn.mixture.gaussianmixture*. scikit. (n.d.-c). https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html
19. Pineau, J. (n.d.). COMP 551 –applied machine learning lecture 13 ... - McGill University. https://www.cs.mcgill.ca/~jpineau/comp551/Lectures/13Unsupervised.pdf
20. *Numpy.linspace#*. numpy.linspace - NumPy v1.26 Manual. (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.linspace.html#numpy.linspace

21. *Density Estimation for a gaussian mixture*. scikit. (n.d.-b). https://scikit-learn.org/stable/auto_examples/mixture/plot_gmm_pdf.html#sphx-glr-auto-examples-mixture-plot-gmm-pdf-py
22. Sharma, P. (2023b, November 5). *The Ultimate Guide to 12 dimensionality reduction techniques (with python codes)*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2018/08/dimensionality-reduction-techniques-python/#h-3-9-independent-component-analysis
23.