

COMP4702 – Machine Learning

Assignment Report

Johanes Steven – 4728272

I. Data Cleaning and Exploratory Data Analysis

The dataset for this project is the 'Rel_2_Nutrient_file' dataset provided on Blackboard. The spreadsheet provided contains two different data sheets, one where the data is measured per 100g, and one where the data is measured per 100mL. For this project, I will be using only the first one, where it is measured per 100g.

A quick look at the data on Jupyter Notebook reveals many features containing nulls and zeros. For that reason, I will remove columns where the nulls and zeros exceed 60% of its entirety, as those columns will not bring any useful value anyway (some columns even having 1000+ nulls).

df.isnull().mean().round(4).mul(100).sort_values(ascending=False)	
Oligosaccharides \n(g)	99.94
Erythritol \n(g)	99.94
Glycerol \n(g)	99.94
Maltitol \n(g)	99.94
C12:2 (g)	99.94
Xylitol \n(g)	99.94
Fumaric acid \n(g)	99.94
C12:2 (%T)	99.94
Polydextrose \n(g)	99.94
C20:4 (g)	99.81
Shikimic acid \n(g)	99.81
C20:4 (%T)	99.81
Dextrin \n(g)	99.57
Glycogen \n(g)	99.44
Xanthophyl \n(ug)	99.32
Delta tocotrienol \n(mg)	99.26
Maltodextrin (g)	99.20
Gamma tocotrienol \n(mg)	99.13
Stachyose \n(g)	99.01
C22:2 (%T)	98.95
Mannitol \n(g)	98.89
Raffinose \n(g)	98.82
Maltotrios \n(g)	98.82
Succinic acid \n(g)	98.70
C22:5w6 (%T)	98.58
Beta tocotrienol \n(mg)	98.58
Alpha tocotrienol \n(mg)	98.58
C16:2w4 (%T)	98.58
C21:5w3 (%T)	98.58
C20:3 (%T)	98.58
C20:2 (%T)	98.58
C18:4w1 (%T)	98.58
C18:3w4 (%T)	98.58
C12:1 (%T)	98.58
C16:3 (%T)	98.58
Cobalt (Co) \n(ug)	98.51
C20:4w3 (%T)	98.45
Lycopene \n(ug)	98.45
C23 (%T)	98.33
Lutein \n(ug)	98.33
C23 (g)	98.14
C21 (%T)	98.14
...	...

Figure 1.1. Null percentage of columns

For columns with less than 60% nulls (but still having nulls), I first imputed the missing values using the averages of that column using Pandas' built-in `fillna()` function.

After all the missing values are taken care of, I took a closer look at the provided data, namely how different types of foods can be classified. I then grouped the different types of foods together into 5 main groups at first, as suggested by the Australian government, which are:

- Vegetables and Legumes/Beans
- Fruit

- Grain (cereal) foods
- Lean meats and poultry, fish, eggs, tofu
- Dairy and alternatives

However, there are some foods in the data that does not really fit into any of the five groups, such as beverages, oils, condiments, commercial snacks, and spices. Therefore, I put those into their own separate groups to differentiate them from the five main groups. By putting these foods into their own classes, I can now apply a classification problem to this dataset.

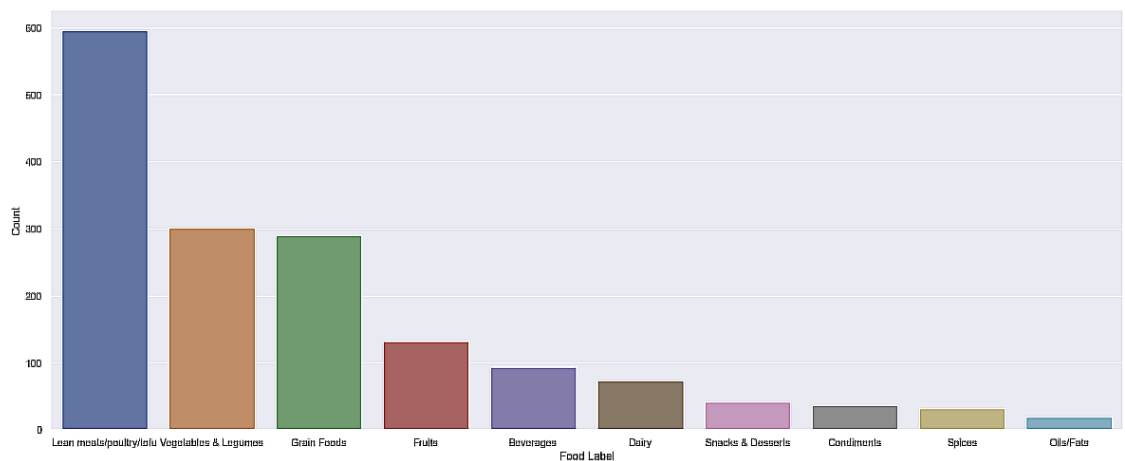


Figure 1.2. Class Division of Foods

I then used a Scikit-learn's LabelEncoder to encode the above classes into a numerical label, as shown by the figure below:

```
Out[407]: Food Label      label-encoded
Lean meats/poultry/tofu    5          596
Vegetables & Legumes       9          301
Grain Foods                4          290
Fruits                     3          132
Beverages                  0           94
Dairy                      2           73
Snacks & Desserts          7           42
Condiments                 1           37
Spices                     8           32
Oils/Fats                  6           19
dtype: int64
```

Figure 1.3. Label Encoding of Food Labels

Now that I have an identifying label, I will be dropping the other non-numerical labels, as those will not be used in the classification anyway. This leaves us with a dataframe that has 1616 rows and 110 columns.

II. Principle Component Analysis (PCA)

I will now be using PCA to reduce the clean dataset's dimensionality. This is mainly because the dataset has 110 dimensions which is a very high number. I will do so to avoid the 'Curse of Dimensionality,' where high numbers of dimensions increase the complexity of the data points, which can reduce the accuracy of the models. As dimensionality increases, the data points will become more sparsely distributed, which reduces the effectiveness of distance-calculating metrics, such as the k-Nearest Neighbors model that I will be using later, making it even more important why I used PCA to reduce the data's dimensionality.

First step is to standardize the data by scaling it. This is done by removing the mean and scaling each data point to its unit variance (using its z-score), as explained by the following formula:

$$z = \frac{x - \mu}{s}$$

where μ denotes the mean of the sample and s denotes the variance of the sample. This can be done easily by using Scikit-learn's StandardScaler method. The main reason why scaling the data is important is because some features of the data is measured in grams, some in micrograms, milligrams, etc., which all have different scaling and thus can reduce the accuracy of the model.

After scaling, I needed to find the right number of dimensions, big enough to explain an adequate percentage of the data's variance, while also being small enough to reduce the dimensionality. To do so, I first called Scikit-learn's PCA function using the original number of columns (110) to observe each principal component's percentage of variance explained.

```
In [163]: from sklearn.decomposition import PCA

pca_110 = PCA(n_components=110, random_state=2023)
pca_110.fit(X_scaled)
X_pca_110 = pca_110.transform(X_scaled)

In [164]: sum(pca_110.explained_variance_ratio_ * 100)

Out[164]: 99.99999999999999
```

Figure 2.1. All 110 features explain 99.999% of the data's variance

```
In [165]: pca_cumsum = np.cumsum(pca_110.explained_variance_ratio_ * 100)
pca_cumsum

Out[165]: array([ 10.26521504, 19.72705776, 27.70348856, 33.28122003,
 37.30173665, 41.04468204, 44.25009412, 47.39775508,
 50.25838101, 52.87456111, 55.26739454, 57.56975015,
 59.66424493, 61.69678592, 63.58410971, 65.39752658,
 66.96441944, 68.4653672 , 69.84075583, 71.19228986,
 72.52218591, 73.82181533, 75.03305773, 76.15480785,
 77.26165397, 78.3295954 , 79.31453599, 80.28515734,
 81.22835963, 82.13365848, 83.00820077, 83.84547544,
 84.62992618, 85.38835316, 86.11741019, 86.8278421 ,
 87.50755635, 88.1837224 , 88.78888954, 89.35812104,
 89.9078202 , 90.42599898, 90.91725624, 91.3996941 ,
 91.87533314, 92.31742944, 92.7474219 , 93.14981632,
 93.54454362, 93.93009824, 94.30512239, 94.66611028,
 95.00818803, 95.32966295, 95.63545981, 95.92926811,
 96.20939635, 96.48131025, 96.73630285, 96.98202683,
 97.20770733, 97.42118357, 97.61468989, 97.80562791,
 97.98949134, 98.16148877, 98.32203508, 98.4627537 ,
 98.60282614, 98.7310677 , 98.84844765, 98.95924915,
 99.05749123, 99.14873344, 99.22991383, 99.30871711,
 99.37804685, 99.44517453, 99.50620317, 99.56028278,
 99.61403363, 99.66587651, 99.71538242, 99.76103948,
 99.80207876, 99.83693483, 99.87073829, 99.90121386,
 99.92032403, 99.93774579, 99.95259801, 99.96610661,
 99.97422534, 99.98206796, 99.98803707, 99.9913471 ,
 99.99429977, 99.99701575, 99.9980981 , 99.99882818,
 99.99937709, 99.99981715, 99.99996632, 99.99998503,
 99.9999918 , 99.99999606, 99.99999959, 99.99999997,
 100.          , 100.          ])
```

Figure 2.2. Cumulative percentages of variance explained by each component

The above figure can be illustrated more clearly in a plot, as seen below:

```
In [221]: import matplotlib.pyplot as plt
sns.set(rc={'figure.figsize':(10,8.27)})

plt.plot(pca_cumsum)
plt.xlabel('No. of Components')
plt.ylabel('Explained Variance')
plt.show()
```

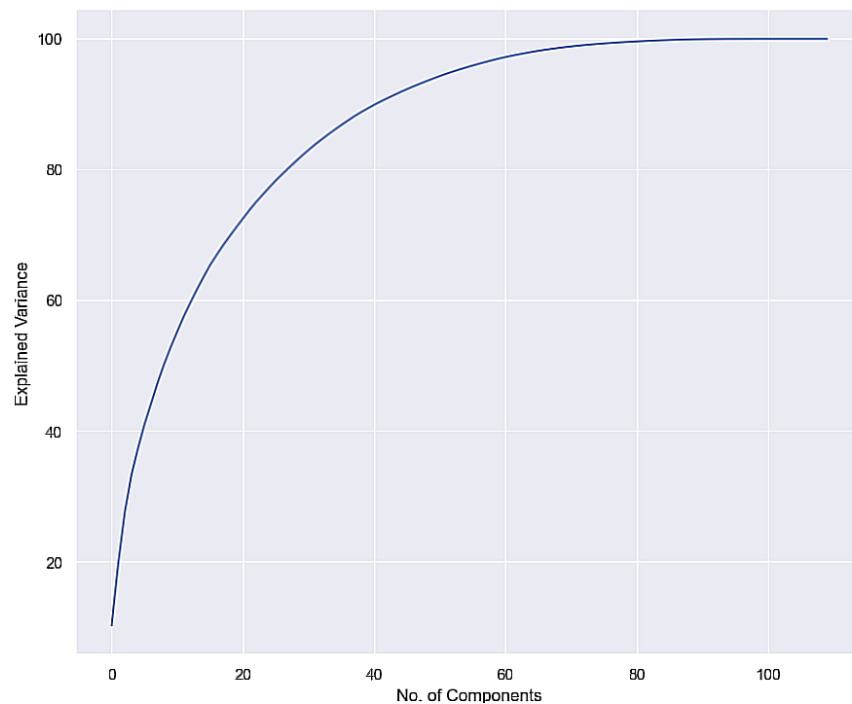


Figure 2.3. Plot of variance explained by principal components

The figure shows a clear diminishing return, meaning as soon as the number of components reach a certain point, the increase in explained variance reduces. The ideal percentage I will be aiming for in this PCA is 95%, which by using Figures 2.2 and 2.3, I was able to know that the ideal number of components is 53.

```
In [239]: pca_53 = PCA(n_components=53, random_state=2023)
pca_53.fit(X_scaled)
X_pca_53 = pca_53.transform(X_scaled)
np.cumsum(pca_53.explained_variance_ratio_ * 100)

Out[239]: array([10.26521504, 19.72705776, 27.70348856, 33.28122003, 37.30173665,
41.04468204, 44.25009412, 47.39775508, 50.25838101, 52.87456111,
55.26739453, 57.56975015, 59.66424493, 61.69678591, 63.58410969,
65.39752651, 66.96441935, 68.46536708, 69.84075571, 71.19228972,
72.52218556, 73.82181409, 75.03305636, 76.15480619, 77.26165213,
78.3295935 , 79.31453387, 80.2851549 , 81.22835581, 82.13365201,
83.00819296, 83.84546019, 84.6299033 , 85.38832893, 86.11737888,
86.82780731, 87.50750006, 88.18366441, 88.7887933 , 89.35802123,
89.90771143, 90.42587155, 90.91703184, 91.39943391, 91.87494704,
92.31610514, 92.74602006, 93.14786315, 93.54211444, 93.92644633,
94.30100293, 94.66163902, 95.00211773])
```

Figure 2.4. 95% of variance explained with 53 principal components

Thus, I now have dataframe with 53 dimensionality instead of 110, which significantly reduces the effect of the ‘curse of dimensionality’ mentioned before.

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9
0	0.397803	1.114769	-2.776373	2.874371	0.028558	0.890800	-0.783110	-2.159021	-3.513700
1	1.966444	2.351598	-6.264710	2.623304	7.242332	5.027993	-5.510545	-7.660898	-7.556610
2	-1.054505	1.833634	-1.412131	2.852883	0.715572	1.453174	-1.513697	-1.981362	-3.826090
3	0.898495	1.726940	-2.841921	1.771292	0.145900	3.932701	-1.676964	-1.778178	-3.733120
4	1.659378	0.892937	-3.452589	2.973281	1.021921	0.911148	-1.046371	-3.329016	-4.311240
...
1611	-2.427184	0.184246	1.387666	-1.110539	1.277969	-0.300030	-0.521542	-0.846076	-0.139140
1612	-2.470248	0.199712	1.490130	-1.224157	1.153923	-0.358281	-0.483579	-0.844940	-0.055530
1613	-2.894924	0.146730	1.839084	-1.446729	2.055375	-0.531842	0.132576	-0.686395	-0.323260
1614	-2.202397	-0.049491	1.291603	-1.052005	1.248667	-0.415186	-0.400661	-0.891809	-0.010030
1615	-2.267272	-0.009686	1.381165	-1.150135	1.171738	-0.448508	-0.388201	-0.864593	0.074920

1616 rows × 54 columns

Figure 2.5. Dataset after PCA

III. Classification using k-Nearest Neighbors (kNN)

The new dataframe can be split into train and test datasets, using Scikit-learn’s `train_test_split()` method, with a ratio of 70:30. I then fit the kNN classifier model into the training dataset, initially with a value of $k = 3$. The model will then be used

to predict the values of both the test and train datasets to find the test and train accuracy scores.

```
In [451]: y_pred = knn.predict(X_test)

from sklearn.metrics import accuracy_score
acc_test_class = accuracy_score(y_test, y_pred)
print("Test accuracy:", acc_test_class)

Test accuracy: 0.8989690721649485

In [452]: y_pred2 = knn.predict(X_train)

from sklearn.metrics import accuracy_score
acc_train_class = accuracy_score(y_train, y_pred2)
print("Train accuracy:", acc_train_class)

Train accuracy: 0.9522546419098143
```

Figure 3.1. Test and Train Accuracy Scores of the kNN Model

The resulting accuracy scores are quite high, but by varying the value of k, I can try and find a better performing model. Below is a chart showing the accuracy scores for kNN models with values of k from 1-1000:

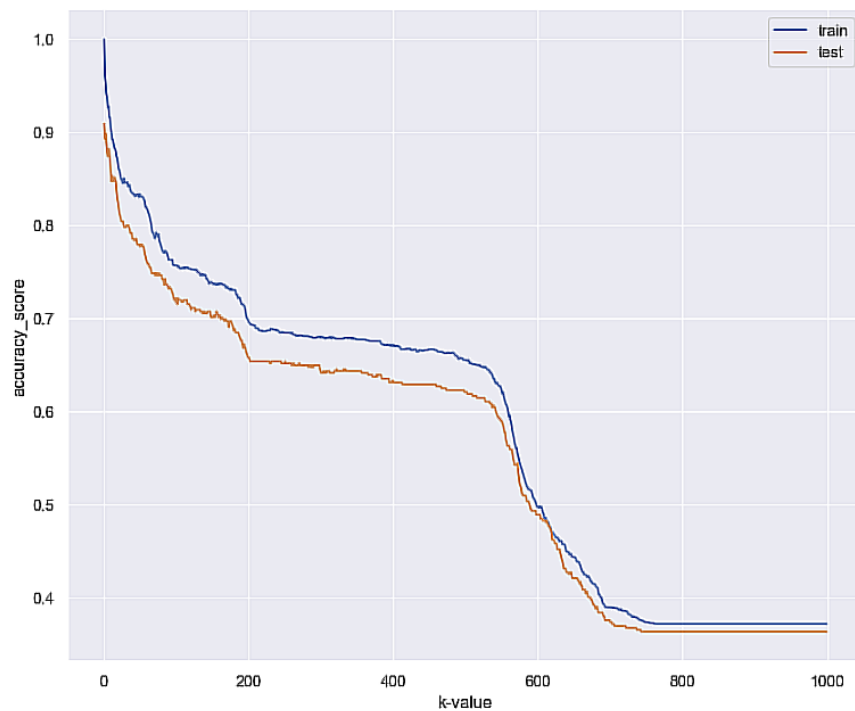


Figure 3.2. Train and Test Accuracy Scores of Varying k-values

The trend of accuracy scores decreases as k-value increases, which is the expected behaviour as higher values of k will make the model more likely to underfit, decreasing its accuracy score. A lower value of k however will mean the model is more likely to overfit to the training data's noise, making it difficult for the model to generalize. It is difficult to find the 'sweet spot' where the model is neither underfitting nor overfitting. Making a

guess based on the chart, the ideal value of k for the best-performing model is at $k = 200$, where both training and test accuracy scores seem to be more stable (less fluctuation in value).

To further test the ideal value of k , I utilized a 5-fold cross-validation technique using Scikit-learn's `cross_val_score()` function. Cross-validation will find the average of several train-test splits, which will be a better method of finding out the ideal k -value. Cross-validation also utilizes stratified sampling, which will balance the representation of each class.

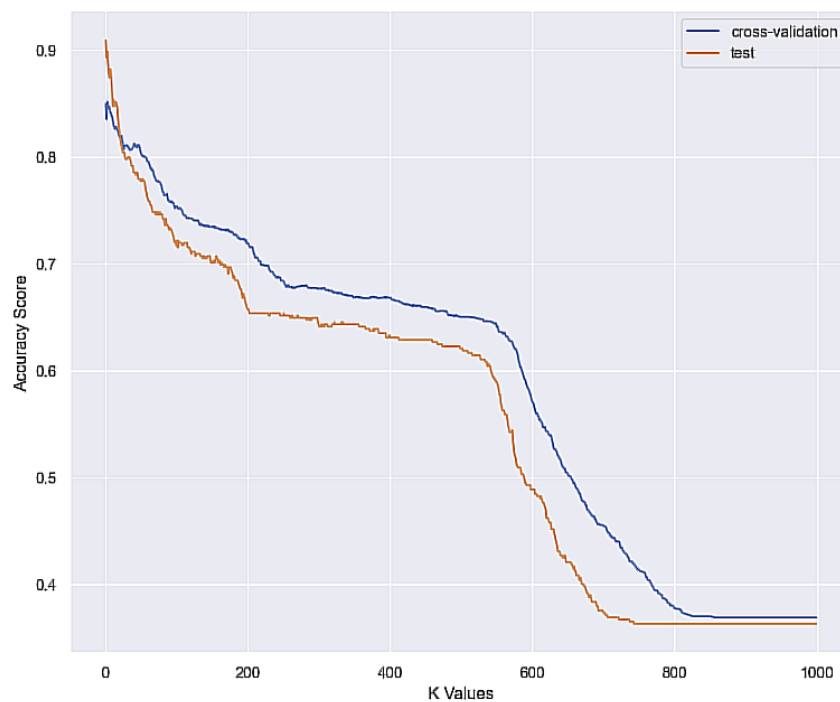


Figure 3.3. Cross-validation vs. Test Accuracy Scores

The chart above shows that the cross-validation and test accuracy scores have a similar trend, but it looks like the cross-validation score suggests that the ideal k -value is slightly bigger than the previous method's suggestion, at $k = 250$.

These tests can be repeated with a different randomization of train-test splits, with the following results:

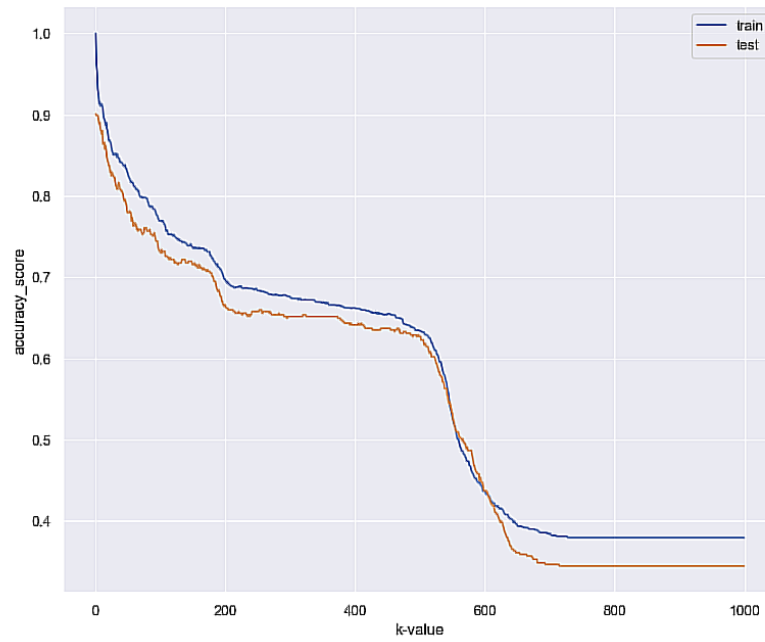


Figure 3.4. 2nd Repetition of Test vs. Train Accuracy Scores with Varying k-values

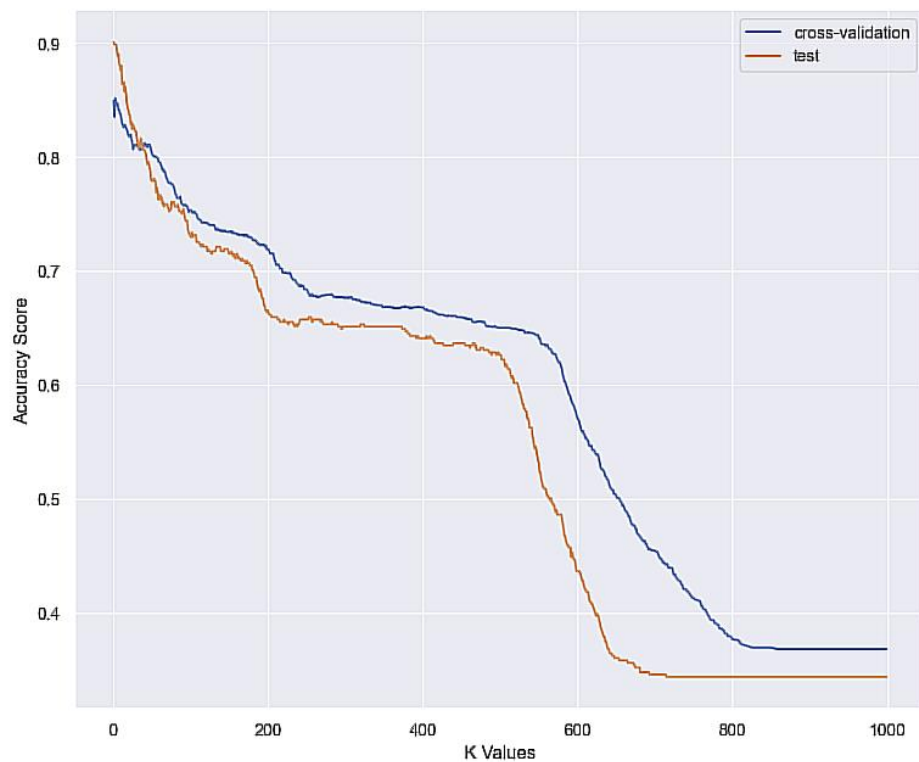


Figure 3.4. 2nd Repetition of Cross-validation vs. Test Accuracy Scores with Varying k-values

Both charts demonstrate similar trends with their previous iterations, where k-values ranging from 200-500 seem to produce a more constant accuracy score. These two charts also suggest that the ideal k-value seem to be at $k = 250$, at the point where the values seem to be more stable.

IV. Regression

Next is seeing if I can fit regression models onto the dataset. First, I will be fitting a linear regression model to the data, then I will be comparing the performance of the linear regression model to polynomial regression models with varying degrees.

- Linear Regression:

```
In [609]: from sklearn.linear_model import LinearRegression

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=True)
regr = LinearRegression()

regr.fit(X_train, y_train)

Out[609]: LinearRegression()

In [612]: y_pred = regr.predict(X_test)

from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.metrics import r2_score

mae = mean_absolute_error(y_true=y_test, y_pred=y_pred)

r2score = r2_score(y_test, y_pred)
print("Mean absolute error:", mae)
print("r2 score:", r2score)

Mean absolute error: 0.6116991783208621
r2 score: 0.596362799582558
```

Figure 4.1. Linear Regression Results

The Mean Absolute Error (MAE) of the model is 0.612 (which is reasonable considering the scale of our dataset), while its R^2 score is 0.596, which means the linear regression model is a decent fit to the dataset, but let us experiment to see if there is a better fitting polynomial regression model.

I will be varying the value of degrees for the polynomial regression model to see the best fitting model for the data.

2nd Degree Polynomial Regression:

Polynomial Regressions can be done by using Scikit-learn's PolynomialFeatures function. The results of the second-degree polynomial regression are as follows:

```

poly_model = PolynomialFeatures(degree=2, include_bias=False)
poly_x = poly_model.fit_transform(X)
poly_x_values = poly_model.fit_transform(X_train)
poly_x_test_values = poly_model.fit_transform(X_test)

poly_model.fit(poly_x_values, y_train)

regression_model = LinearRegression()

regression_model.fit(poly_x_values, y_train)
y_pred = regression_model.predict(poly_x_test_values)

```

```

mae = mean_absolute_error(y_true=y_test, y_pred=y_pred)

r2score = r2_score(y_test, y_pred)
print("Mean absolute error:", mae)
print("r2 score:", r2score)

```

Mean absolute error: 1.952200631348672
r2 score: -50.354421992488476

Figure 4.2. 2nd Degree Polynomial Regression

The R^2 score is negative, which means that the model does not suit the dataset at all. It could also mean that the mean of the test dataset is incredibly different to the mean of the train dataset. Another possible reason why this happens is because 2nd order polynomials cannot fit the dataset due to the sheer size of it (53 principal components is incredibly large). This result can be validated by using 5-fold cross-validation score:

```

np.mean(cross_val_score(regression_model, poly_x_values, y_train, cv=5))
-5.22533216121259

```

Figure 4.3. 2nd Degree Polynomial Regression Cross-validation Score

The cross-validation score still produces a negative value, thus supporting the argument that 2nd degree polynomial regression does not fit the dataset at all. Fitting a 3rd degree polynomial regression model to the dataset also produces a negative R^2 score:

```

poly_model = PolynomialFeatures(degree=3, include_bias=False)
poly_x_values = poly_model.fit_transform(X_train)
poly_x_test_values = poly_model.fit_transform(X_test)

poly_model.fit(poly_x_values, y_train)

regression_model = LinearRegression()

regression_model.fit(poly_x_values, y_train)
y_pred = regression_model.predict(poly_x_test_values)

mae = mean_absolute_error(y_true=y_test, y_pred=y_pred)

r2score = r2_score(y_test, y_pred)
print("Mean absolute error:", mae)
print("r2 score:", r2score)

```

```

Mean absolute error: 1.9696928661701811
r2 score: -53.06622626289298

```

Figure 4.4. 3rd Degree Polynomial Regression Result

So will the 5-fold cross-validation score:

```

In [771]: np.mean(cross_val_score(regression_model, poly_x_values, y_train, scoring='r2', <
Out[771]: -5.223544237711011

```

Figure 4.5. 3rd Degree Polynomial Regression Result Cross-validation Score

4th degree polynomial regression only decreases the score even further.

```

In [772]: poly_model = PolynomialFeatures(degree=4, include_bias=True)
poly_x_values = poly_model.fit_transform(X_train)
poly_x_test_values = poly_model.fit_transform(X_test)

poly_model.fit(poly_x_values, y_train)

regression_model = LinearRegression()

regression_model.fit(poly_x_values, y_train)
y_pred = regression_model.predict(poly_x_test_values)

mae = mean_absolute_error(y_true=y_test, y_pred=y_pred)

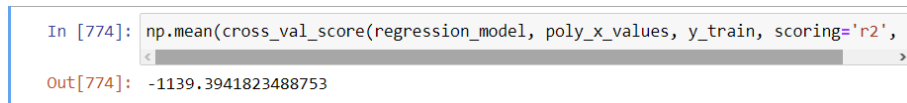
r2score = r2_score(y_test, y_pred)
print("Mean absolute error:", mae)
print("r2 score:", r2score)

```

```

Mean absolute error: 12.553017897470205
r2 score: -5781.957380079043

```



```
In [774]: np.mean(cross_val_score(regression_model, poly_x_values, y_train, scoring='r2',  
Out[774]: -1139.3941823488753
```

Figure 4.5. 4th Degree Polynomial Regression Results

Unfortunately, increasing the degree even further is not possible due to a lack of memory that can be allocated for the array, because of the sheer size of the dataset itself.

```
MemoryError: Unable to allocate 38.6 GiB for an array with shape (1131, 458211  
6) and data type float64
```

Figure 4.6. Memory Error when Trying 5th Degree Polynomial Regression

However, from the clear downwards trend, it is safe to assume that any other degrees of polynomial past 4 will produce similarly terrible results in terms of the model's fitness. Therefore, for regression problems, the Linear Regression (1st degree) is the best-fit for this dataset.