HW1 Report (Group 29)

Group Members: 張柏豐(111651070)、王睿綸(110511290)

A. Summary of K-Nearest Neighbors (KNN) Processing Steps

1. **Import Necessary Functions and Download the Specified Kaggle Dataset**

   Begin by importing the required functions and downloading the designated Kaggle dataset.

2. **Remove Data with a Mix of English and Numerals (Having Minimal Impact on Classification Features)**

   Exclude data instances that combine English and numerals, as these have minimal influence on classification features.

3. **Calculate the Feature Correlation Matrix Using the Feature Correlation Matrix**

   Utilize the feature correlation matrix to assess the relationships between each feature and the classification labels.

4. **Select the Most Relevant Features from the Training and Testing Sets**

   Identify the most highly correlated features with the training and testing datasets.

5. **Evaluate Accuracy Using a KNN Model**

   Assess accuracy by feeding these features into a KNN model for testing.

6. **Enhance Precision with Normalization and Weight**

   Improve precision by introducing normalization and weighting techniques.

B. Detailed Code Explanation

1. Import Function

```
# import function
!pip install kaggle
from google.colab import files
import pandas as pd
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Z-Score Normalization
from sklearn.preprocessing import StandardScaler

# Robust Normalization
from sklearn.preprocessing import RobustScaler
```

First, import all the libraries that will be used in the following code. The 'files' library is imported to allow the upload of the Kaggle.json file, which enables us to use Kaggle's API to download the dataset for this assignment. The 'KNeighborsClassifier' library is used for our k-nearest neighbors (k-NN) model, and the 'accuracy_score' function is imported to evaluate accuracy. Finally, the last two libraries are used for normalizing the training and test datasets.

2. Download data in Kaggle

```
# Download data in Kaggle
uploaded = files.upload() # kaggle.json
!mkdir -p ~/.kaggle/
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json # Setting Permissions of the File (kaggle.json), only readable by the current user
!kaggle competitions download -c groupchallenge-knn-competition-pds
!unzip groupchallenge-knn-competition-pds.zip
```

Next, we start by uploading the 'kaggle.json' file to the Google Colab workspace. We then move this file to a location where we can access it and set the permissions to be accessible only by the current user to prevent others from using the API to download datasets from this account. After that, we proceed to download and unzip the dataset for this assignment, preparing for the next steps.

3. Data formatting

```
# For Train
df = pd.read_csv("train.csv", sep=',', header=None, dtype=str, skipinitialspace=True)
df = df.fillna(0)
train_array = df.values

# For Test
df = pd.read_csv("test.csv", sep=',', header=None, dtype=str, skipinitialspace=True)
df = df.fillna(0)
test_array = df.values
```

To facilitate the upcoming data processing, we first read the original CSV file and convert it into an array, recording the data in each cell. Each entry in the CSV file is separated by a comma (",") and stored in a matrix. The "dtype" parameter specifies the data type of the stored data. Finally, the "skipinitialspace" option is used to skip any whitespace before and after the commas.

Next, we use the "df.fillna(0)" function to replace missing values ("nan") in the data table with the numerical value 0. Finally, we convert the DataFrame into a Numpy array format.

4. Data Cleaning

```
# Change Y, N to 1, 0 in columns 7, 8
train_array[:, 7] = np.where(train_array[:, 7] == "Y", 1, 0)
train_array[:, 8] = np.where(train_array[:, 8] == "Y", 1, 0)
test_array[:, 7] = np.where(test_array[:, 7] == "Y", 1, 0)
test_array[:, 8] = np.where(test_array[:, 8] == "Y", 1, 0)

# Segment IDX
train_array = train_array[1:, :]
test_array = test_array[1:, :]

# Remove mixed alphanumeric data (low-relevance)
# [IDX, id, full_name, pdes, name, prefix, orbit_id, equinox, class]
delete_columns = [0, 1, 3, 4, 5, 6, 13, 17, 44]
train_array = np.delete(train_array, delete_columns, axis=1)
test_array = np.delete(test_array, delete_columns, axis=1)
```

Next, to prepare the data, we'll start by addressing the columns 7 and 8 in both the train_array and test_array. These columns contain only three possible values: "N," "Y," and "nan" (which we treat as 0). We will use One-Hot Encoding to encode these values as follows: "N" becomes 0, "Y" becomes 1, and "nan" becomes 0.

Next, we remove the first row, which contains labels, and retain only the data.

Finally, for the data that consists of a mix of English and numeric values, since most of it is primarily numeric and does not significantly affect the KNN prediction results, we can remove it. This is mainly because both the Feature Correlation Matrix and the KNN model cannot handle non-numeric data, so it must be excluded. The selected columns for removal include: [IDX, id, full_name, pdes, name, prefix, orbit_id, equinox, class]

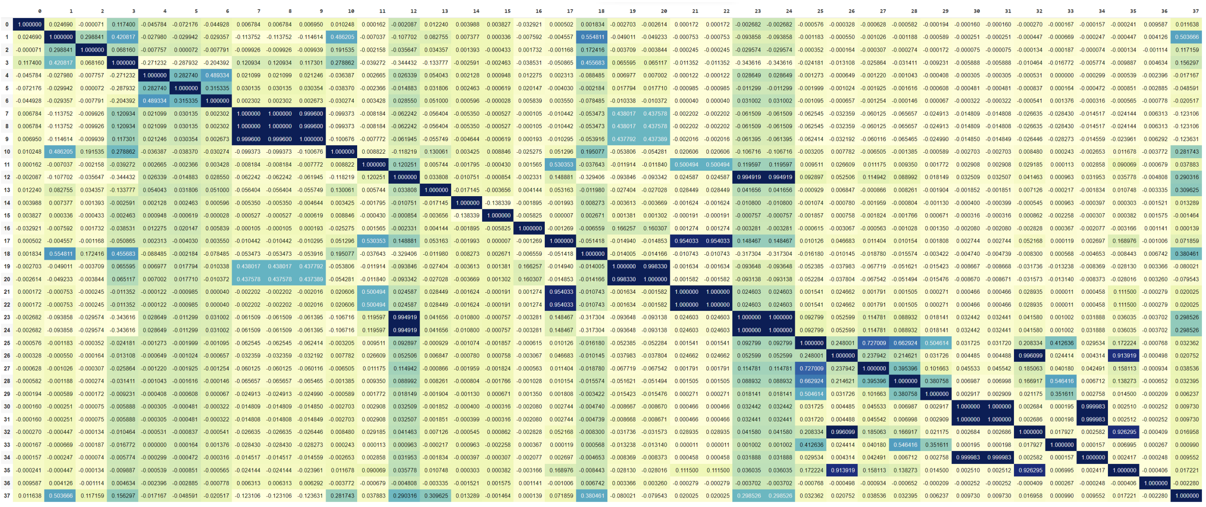
As a result, the train_array and test_array matrices we obtain will consist entirely of numerical values.

5. Feature Correlation Matrix

```
# Creat feature correlation matrix
train_df = pd.DataFrame(train_array.astype("float"))
correlation_matrix = train_df.corr(numeric_only=True)
display(correlation_matrix.style.background_gradient(cmap='YlGnBu').set_properties(2))

# Segment labels and answers
train_data = train_array[:, :-1]
train_label = train_array[:, -1]
test_data = test_array[:, :-1]
test_label = test_array[:, -1]
```

Next, we'll export a Feature Correlation Matrix from the previously obtained train_array to understand the relationships between each feature. It's essential to note that the data matrix we obtained earlier consists of both integers and floating-point numbers. Different data types can prevent us from conducting feature correlation tests. Therefore, we need to convert all values to floating-point numbers (float) first before generating the chart. We configure the chart's color gradient using the "YlGnBu" color format and set the font size for the correlation values of each feature to level 2 (.set_properties(2)).

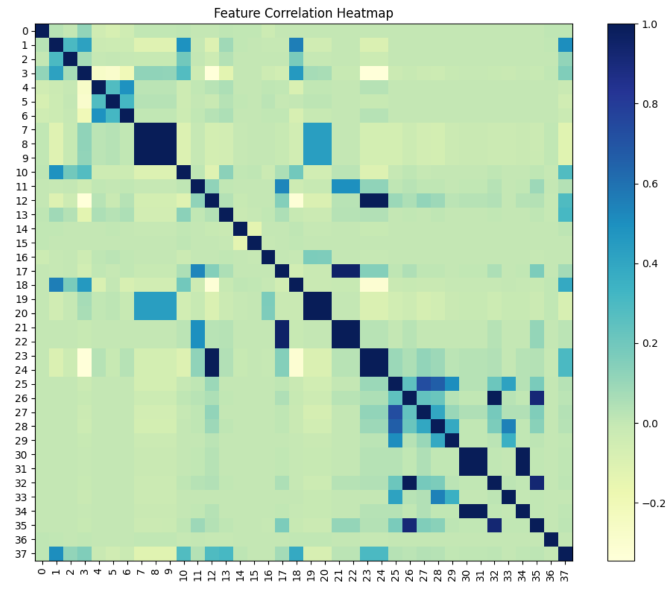
After generating the chart, we proceed to separate the answers and features in both the train_array and test_array, storing them in two separate matrices.



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1.000000	0.024690	-0.000071	0.117400	-0.045784	-0.072176	-0.044928	0.006784	0.006784	0.006950	0.010248	0.000162	-0.002087
1	0.024690	1.000000	0.298841	0.420817	-0.027980	-0.029942	-0.029357	-0.113752	-0.113752	-0.114614	0.486205	-0.007037	-0.107702
2	-0.000071	0.298841	1.000000	0.068160	-0.007757	0.000072	-0.007791	-0.009926	-0.009926	-0.009939	0.191535	-0.002158	-0.035647
3	0.117400	0.420817	0.068160	1.000000	-0.271232	-0.287932	-0.204392	0.120934	0.120934	0.117301	0.278862	-0.039272	-0.344432
4	-0.045784	-0.027980	-0.007757	-0.271232	1.000000	0.282740	0.489334	0.021099	0.021099	0.021246	-0.036387	0.002665	0.026339
5	-0.072176	-0.029942	0.000072	-0.287932	0.282740	1.000000	0.315335	0.030135	0.030135	0.030354	-0.038370	-0.002366	-0.014883
6	-0.044928	-0.029357	-0.007791	-0.204392	0.489334	0.315335	1.000000	0.002302	0.002302	0.002673	-0.030274	0.003428	0.028550
7	0.006784	-0.113752	-0.009926	0.120934	0.021099	0.030135	0.002302	1.000000	1.000000	0.999600	-0.099373	-0.008184	-0.062242
8	0.006784	-0.113752	-0.009926	0.120934	0.021099	0.030135	0.002302	1.000000	1.000000	0.999600	-0.099373	-0.008184	-0.062242
9	0.006950	-0.114614	-0.009939	0.117301	0.021246	0.030354	0.002673	0.999600	0.999600	1.000000	-0.100676	-0.007772	-0.061945
10	0.010248	0.486205	0.191535	0.278862	-0.036387	-0.038370	-0.030274	-0.099373	-0.099373	-0.100676	1.000000	0.008822	-0.118219
11	0.000162	-0.007037	-0.002158	-0.039272	0.002665	-0.002366	0.003428	-0.008184	-0.008184	-0.007772	0.008822	1.000000	0.120251
12	-0.002087	-0.107702	-0.035647	-0.344432	0.026339	-0.014883	0.028550	-0.062242	-0.062242	-0.061945	-0.118219	0.120251	1.000000
13	0.012240	0.082755	0.034357	-0.133777	0.054043	0.031806	0.051000	-0.056404	-0.056404	-0.055749	0.130061	0.005744	0.033808
14	0.003988	0.007377	0.001393	-0.002591	0.002128	0.002463	0.000596	-0.005350	-0.005350	-0.004644	0.003425	-0.001795	-0.010751
15	0.003827	0.000336	-0.000433	-0.002463	0.000948	-0.000619	-0.000028	-0.000527	-0.000527	-0.000619	0.008846	-0.000430	-0.000854
16	-0.032921	-0.007592	0.001732	-0.038531	0.012275	0.020147	0.005839	-0.000105	-0.000105	0.000193	-0.025275	0.001565	-0.002331
17	0.000502	-0.004557	-0.001168	-0.050865	0.002313	-0.004030	0.003550	-0.010442	-0.010442	-0.010295	0.051296	0.530353	0.148881
18	0.001834	0.554811	0.172416	0.455683	-0.088485	-0.002184	-0.078485	-0.053473	-0.053473	-0.053916	0.195077	-0.037643	-0.329406
19	-0.002703	-0.049011	-0.003709	0.065595	0.006977	0.017794	-0.010338	0.438017	0.438017	0.437792	-0.053806	-0.011914	-0.093846
20	-0.002614	-0.049233	-0.003844	0.065117	0.007002	0.017710	-0.010372	0.437578	0.437578	0.437389	-0.054281	-0.011840	-0.093342
21	0.000172	-0.000753	-0.000245	-0.011352	-0.000122	-0.000985	0.000040	-0.002202	-0.002202	-0.002016	0.020606	0.500494	0.024587
22	0.000172	-0.000753	-0.000245	-0.011352	-0.000122	-0.000985	0.000040	-0.002202	-0.002202	-0.002016	0.020606	0.500494	0.024587
23	-0.002682	-0.093858	-0.029574	-0.343616	0.028649	-0.011299	0.031002	-0.061509	-0.061509	-0.061395	-0.106716	0.119597	0.994919
24	-0.002682	-0.093858	-0.029574	-0.343616	0.028649	-0.011299	0.031002	-0.061509	-0.061509	-0.061395	-0.106716	0.119597	0.994919
25	-0.000576	-0.001183	-0.000352	-0.024181	-0.001273	-0.001999	-0.001095	-0.062545	-0.062545	-0.062414	-0.003205	0.009511	0.092897
26	-0.000328	-0.000550	-0.000164	-0.013108	-0.000649	-0.001024	-0.000657	-0.032359	-0.032359	-0.032192	0.007782	0.026609	0.052506
27	-0.000628	-0.001026	-0.000307	-0.025864	-0.001220	-0.001925	-0.001254	-0.060125	-0.060125	-0.060116	-0.006505	0.011175	0.114942
28	-0.000582	-0.001188	-0.000274	-0.031411	-0.001043	-0.001616	-0.000146	-0.065657	-0.065657	-0.065465	-0.001385	0.009350	0.088992
29	-0.000194	-0.000589	-0.000172	-0.009231	-0.000408	-0.000608	0.000067	-0.024913	-0.024913	-0.024990	-0.000589	0.001772	0.018149
30	-0.000160	-0.000251	-0.000075	-0.005888	-0.000305	-0.000481	-0.000322	-0.014809	-0.014809	-0.014850	-0.002703	0.002908	0.032509
31	-0.000160	-0.000251	-0.000075	-0.005888	-0.000305	-0.000481	-0.000322	-0.014808	-0.014808	-0.014849	-0.002703	0.002908	0.032507
32	-0.000270	-0.000447	-0.000134	-0.010464	-0.000531	-0.000837	-0.000541	-0.026635	-0.026635	-0.026446	0.008480	0.029185	0.041463
33	-0.000167	-0.000669	-0.000187	-0.016772	0.000000	0.000164	0.001376	-0.028430	-0.028430	-0.028273	0.000243	0.000113	0.000963
34	-0.000157	-0.000247	-0.000074	-0.005774	-0.000299	-0.000472	-0.000316	-0.014517	-0.014517	-0.014559	-0.002653	0.002858	0.031953
35	-0.000241	-0.000447	-0.000134	-0.009887	-0.000539	-0.000851	-0.000565	-0.024144	-0.024144	-0.023961	0.011678	0.090069	0.035778
36	0.009587	0.004126	-0.001114	0.004634	-0.002396	-0.002885	-0.000778	0.006313	0.006313	0.006292	-0.003772	-0.000679	-0.004808
37	0.011638	0.503666	0.117159	0.156297	-0.017167	-0.048591	-0.020517	-0.123106	-0.123106	-0.123631	0.281743	0.037883	0.290316

	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0.012240	0.003988	0.003827	-0.032921	0.000502	0.001834	-0.002703	-0.002614	0.000172	0.000172	-0.002682	-0.002682	-0.000576
1	0.082755	0.007377	0.000336	-0.007592	-0.004557	0.554811	-0.049011	-0.049233	-0.000753	-0.000753	-0.093858	-0.093858	-0.001183
2	0.034357	0.001393	-0.000433	0.001732	-0.001168	0.172416	-0.003709	-0.003844	-0.000245	-0.000245	-0.029574	-0.029574	-0.000352
3	-0.133777	-0.002591	-0.002463	-0.038531	-0.050865	0.455683	0.065595	0.065117	-0.011352	-0.011352	-0.343616	-0.343616	-0.024181
4	0.054043	0.002128	0.000948	0.012275	0.002313	-0.088485	0.006977	0.007002	-0.000122	-0.000122	0.028649	0.028649	-0.001273
5	0.031806	0.002463	-0.000619	0.020147	-0.004030	-0.002184	0.017794	0.017710	-0.000985	-0.000985	-0.011299	-0.011299	-0.001999
6	0.051000	0.000596	-0.000028	0.005839	0.003550	-0.078485	-0.010338	-0.010372	0.000040	0.000040	0.031002	0.031002	-0.001095
7	-0.056404	-0.005350	-0.000527	-0.000105	-0.010442	-0.053473	0.438017	0.437578	-0.002202	-0.002202	-0.061509	-0.061509	-0.062545
8	-0.056404	-0.005350	-0.000527	-0.000105	-0.010442	-0.053473	0.438017	0.437578	-0.002202	-0.002202	-0.061509	-0.061509	-0.062545
9	-0.055749	-0.004644	-0.000619	0.000193	-0.010295	-0.053916	0.437792	0.437389	-0.002016	-0.002016	-0.061395	-0.061395	-0.062414
10	0.130061	0.003425	0.008846	-0.025275	0.051296	0.195077	-0.053806	-0.054281	0.020606	0.020606	-0.106716	-0.106716	-0.003205
11	0.005744	-0.001795	-0.000430	0.001565	0.530353	-0.037643	-0.011914	-0.011840	0.500494	0.500494	0.119597	0.119597	0.009511
12	0.033808	-0.010751	-0.000854	-0.002331	0.148881	-0.329406	-0.093846	-0.093342	0.024587	0.024587	0.994919	0.994919	0.092897
13	1.000000	-0.017145	-0.003656	0.004144	0.053163	-0.011980	-0.027404	-0.027028	0.028449	0.028449	0.041656	0.041656	-0.000929
14	-0.017145	1.000000	-0.138339	-0.001895	-0.001993	0.008273	-0.003613	-0.003669	-0.001624	-0.001624	-0.010800	-0.010800	-0.001074
15	-0.003656	-0.138339	1.000000	-0.005825	0.000007	0.002671	0.001381	0.001302	-0.000191	-0.000191	-0.000757	-0.000757	-0.001857
16	0.004144	-0.001895	-0.005825	1.000000	-0.001269	-0.006559	0.166257	0.160307	0.001274	0.001274	-0.003281	-0.003281	-0.000615
17	0.053163	-0.001993	0.000007	-0.001269	1.000000	-0.051418	-0.014940	-0.014853	0.954033	0.954033	0.148467	0.148467	0.010126
18	-0.011980	0.008273	0.002671	-0.006559	-0.051418	1.000000	-0.014005	-0.014166	-0.010743	-0.010743	-0.317304	-0.317304	-0.016180
19	-0.027404	-0.003613	0.001381	0.166257	-0.014940	-0.014005	1.000000	0.998330	-0.001634	-0.001634	-0.093648	-0.093648	-0.052385
20	-0.027028	-0.003669	0.001302	0.160307	-0.014853	-0.014166	0.998330	1.000000	-0.001582	-0.001582	-0.093138	-0.093138	-0.052284
21	0.028449	-0.001624	-0.000191	0.001274	0.954033	-0.010743	-0.001634	-0.001582	1.000000	1.000000	0.024603	0.024603	0.001541
22	0.028449	-0.001624	-0.000191	0.001274	0.954033	-0.010743	-0.001634	-0.001582	1.000000	1.000000	0.024603	0.024603	0.001541
23	0.041656	-0.010800	-0.000757	-0.003281	0.148467	-0.317304	-0.093648	-0.093138	0.024603	0.024603	1.000000	1.000000	0.092799
24	0.041656	-0.010800	-0.000757	-0.003281	0.148467	-0.317304	-0.093648	-0.093138	0.024603	0.024603	1.000000	1.000000	0.092799
25	-0.000929	-0.001074	-0.001857	-0.000615	0.010126	-0.016180	-0.052385	-0.052284	0.001541	0.001541	0.092799	0.092799	1.000000
26	0.006847	-0.000780	0.000758	-0.003067	0.046683	-0.010145	-0.037983	-0.037804	0.024662	0.024662	0.052599	0.052599	0.248001
27	-0.000866	-0.001959	-0.001824	-0.000563	0.011404	-0.018780	-0.067719	-0.067542	0.001791	0.001791	0.114781	0.114781	0.727009
28	0.008261	-0.000804	-0.001766	-0.001028	0.010154	-0.015574	-0.051621	-0.051494	0.001505	0.001505	0.088932	0.088932	0.662924
29	-0.001904	-0.001130	0.000671	0.001350	0.001808	-0.003422	-0.015423	-0.015476	0.000271	0.000271	0.018141	0.018141	0.504614
30	-0.001852	-0.000400	-0.000316	-0.002080	0.002744	-0.004740	-0.008667	-0.008670	0.000466	0.000466	0.032442	0.032442	0.031725
31	-0.001851	-0.000399	-0.000316	-0.002080	0.002744	-0.004739	-0.008668	-0.008671	0.000466	0.000466	0.032441	0.032441	0.031720
32	0.007126	-0.000545	0.000862	-0.002828	0.052168	-0.008300	-0.031736	-0.031573	0.028935	0.028935	0.041580	0.041580	0.208334
33	-0.000217	0.000963	-0.002258	0.000367	0.000119	0.000568	-0.013238	-0.013140	0.000011	0.000011	0.001002	0.001002	0.412636
34	-0.001834	-0.000397	-0.000307	-0.002077	0.002697	-0.004653	-0.008369	-0.008373	0.000458	0.000458	0.031888	0.031888	0.029534
35	0.010748	0.000303	0.000382	-0.003166	0.168976	-0.008443	-0.028130	-0.028016	0.111500	0.111500	0.036035	0.036035	0.172224
36	-0.003335	-0.001521	0.001575	0.001141	-0.001006	0.006742	0.003366	0.003260	-0.000279	-0.000279	-0.003702	-0.003702	-0.000768
37	0.309625	0.013289	-0.001464	0.000139	0.071859	0.380461	-0.080021	-0.079543	0.020025	0.020025	0.298526	0.298526	0.032362

	26	27	28	29	30	31	32	33	34	35	36	37
0	-0.000328	-0.000628	-0.000582	-0.000194	-0.000160	-0.000160	-0.000270	-0.000167	-0.000157	-0.000241	0.009587	0.011638
1	-0.000550	-0.001026	-0.001188	-0.000589	-0.000251	-0.000251	-0.000447	-0.000669	-0.000247	-0.000447	0.004126	0.503666
2	-0.000164	-0.000307	-0.000274	-0.000172	-0.000075	-0.000075	-0.000134	-0.000187	-0.000074	-0.000134	-0.001114	0.117159
3	-0.013108	-0.025864	-0.031411	-0.009231	-0.005888	-0.005888	-0.010464	-0.016772	-0.005774	-0.009887	0.004634	0.156297
4	-0.000649	-0.001220	-0.001043	-0.000408	-0.000305	-0.000305	-0.000531	0.000000	-0.000299	-0.000539	-0.002396	-0.017167
5	-0.001024	-0.001925	-0.001616	-0.000608	-0.000481	-0.000481	-0.000837	0.000164	-0.000472	-0.000851	-0.002885	-0.048591
6	-0.000657	-0.001254	-0.000146	0.000067	-0.000322	-0.000322	-0.000541	0.001376	-0.000316	-0.000565	-0.000778	-0.020517
7	-0.032359	-0.060125	-0.065657	-0.024913	-0.014809	-0.014808	-0.026635	-0.028430	-0.014517	-0.024144	0.006313	-0.123106
8	-0.032359	-0.060125	-0.065657	-0.024913	-0.014809	-0.014808	-0.026635	-0.028430	-0.014517	-0.024144	0.006313	-0.123106
9	-0.032192	-0.060116	-0.065465	-0.024990	-0.014850	-0.014849	-0.026446	-0.028273	-0.014559	-0.023961	0.006292	-0.123631
10	0.007782	-0.006505	-0.001385	-0.000589	-0.002703	-0.002703	0.008480	0.000243	-0.002653	0.011678	-0.003772	0.281743
11	0.026609	0.011175	0.009350	0.001772	0.002908	0.002908	0.029185	0.000113	0.002858	0.090069	-0.000679	0.037883
12	0.052506	0.114942	0.088992	0.018149	0.032509	0.032507	0.041463	0.000963	0.031953	0.035778	-0.004808	0.290316
13	0.006847	-0.000866	0.008261	-0.001904	-0.001852	-0.001851	0.007126	-0.000217	-0.001834	0.010748	-0.003335	0.309625
14	-0.000780	-0.001959	-0.000804	-0.001130	-0.000400	-0.000399	-0.000545	0.000963	-0.000397	0.000303	-0.001521	0.013289
15	0.000758	-0.001824	-0.001766	0.000671	-0.000316	-0.000316	0.000862	-0.002258	-0.000307	0.000382	0.001575	-0.001464
16	-0.003067	-0.000563	-0.001028	0.001350	-0.002080	-0.002080	-0.002828	0.000367	-0.002077	-0.003166	0.001141	0.000139
17	0.046683	0.011404	0.010154	0.001808	0.002744	0.002744	0.052168	0.000119	0.002697	0.168976	-0.001006	0.071859
18	-0.010145	-0.018780	-0.015574	-0.003422	-0.004740	-0.004739	-0.008300	0.000568	-0.004653	-0.008443	0.006742	0.380461
19	-0.037983	-0.067719	-0.051621	-0.015423	-0.008667	-0.008668	-0.031736	-0.013238	-0.008369	-0.028130	0.003366	-0.080021
20	-0.037804	-0.067542	-0.051494	-0.015476	-0.008670	-0.008671	-0.031573	-0.013140	-0.008373	-0.028016	0.003260	-0.079543
21	0.024662	0.001791	0.001505	0.000271	0.000466	0.000466	0.028935	0.000011	0.000458	0.111500	-0.000279	0.020025
22	0.024662	0.001791	0.001505	0.000271	0.000466	0.000466	0.028935	0.000011	0.000458	0.111500	-0.000279	0.020025
23	0.052599	0.114781	0.088932	0.018141	0.032442	0.032441	0.041580	0.001002	0.031888	0.036035	-0.003702	0.298526
24	0.052599	0.114781	0.088932	0.018141	0.032442	0.032441	0.041580	0.001002	0.031888	0.036035	-0.003702	0.298526
25	0.248001	0.727009	0.662924	0.504614	0.031725	0.031720	0.208334	0.412636	0.029534	0.172224	-0.000768	0.032362
26	1.000000	0.237942	0.214621	0.031726	0.004485	0.004488	0.996099	0.024414	0.004314	0.913919	-0.000498	0.020752
27	0.237942	1.000000	0.395396	0.101663	0.045533	0.045542	0.185063	0.040180	0.042491	0.158113	-0.000934	0.038536
28	0.214621	0.395396	1.000000	0.380758	0.006987	0.006998	0.166917	0.546416	0.006712	0.138273	-0.000652	0.032395
29	0.031726	0.101663	0.380758	1.000000	0.002917	0.002909	0.021175	0.351611	0.002758	0.014500	-0.000209	0.006237
30	0.004485	0.045533	0.006987	0.002917	1.000000	1.000000	0.002684	0.000195	0.999983	0.002510	-0.000252	0.009730
31	0.004488	0.045542	0.006998	0.002909	1.000000	1.000000	0.002686	0.000198	0.999983	0.002512	-0.000252	0.009730
32	0.996099	0.185063	0.166917	0.021175	0.002684	0.002686	1.000000	0.017927	0.002582	0.926295	-0.000409	0.016958
33	0.024414	0.040180	0.546416	0.351611	0.000195	0.000198	0.017927	1.000000	0.000157	0.006995	0.000267	0.000990
34	0.004314	0.042491	0.006712	0.002758	0.999983	0.999983	0.002582	0.000157	1.000000	0.002417	-0.000248	0.009552
35	0.913919	0.158113	0.138273	0.014500	0.002510	0.002512	0.926295	0.006995	0.002417	1.000000	-0.000406	0.017221
36	-0.000498	-0.000934	-0.000652	-0.000209	-0.000252	-0.000252	-0.000409	0.000267	-0.000248	-0.000406	1.000000	-0.002280
37	0.020752	0.038536	0.032395	0.006237	0.009730	0.009730	0.016958	0.000990	0.009552	0.017221	-0.002280	1.000000



6. Reserve Correlated Column

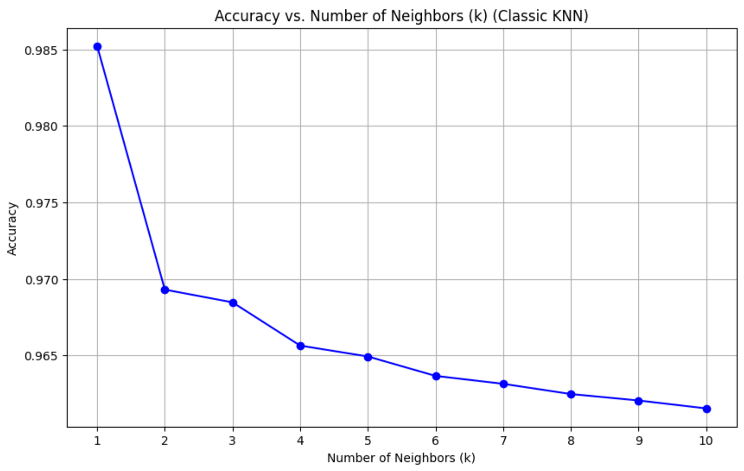
```
# Select features with higher correlation
reserved_column = [1, 10, 12, 13, 18, 23, 24] #[neo, e, q, i, n, moid, moid_ld]
train_data = train_data[:, reserved_column]
test_data = test_data[:, reserved_column]
```

From the previously generated feature correlation matrix, we can observe that features in columns 1, 10, 12, 13, 18, 23, and 24 (neo, e, q, i, n, moid, moid_ld) exhibit the highest correlations with the last column (class_id). The correlation values range from 30% to 50%. Therefore, we will only select these specific features from train_data and test_data for training and prediction using k-nearest neighbors (KNN).

7. Classic KNN w/ k=1

```
# Train & Predict (None)
accuracies = []

for k in range(1, 11):
  knn = KNeighborsClassifier(n_neighbors = k)
  knn.fit(train_data, train_label)
  test_pred_none = knn.predict(test_data)
  accuracy = accuracy_score(test_label, test_pred_none)
  accuracies.append(accuracy)

plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), accuracies, marker='o', linestyle='-', color='b')
plt.title('Accuracy vs. Number of Neighbors (k) (Classic KNN)')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.grid(True)
plt.xticks(range(1, 11))
plt.show()
```

```
Accuracy: [0.9852202701386861, 0.9693033704739251, 0.9684722280181948, 0.9656449526353128, 0.9649355256019697,
           0.9636662076256451, 0.9631480476846249, 0.9624803516532432, 0.9620595640501328, 0.9615414041091127]
```

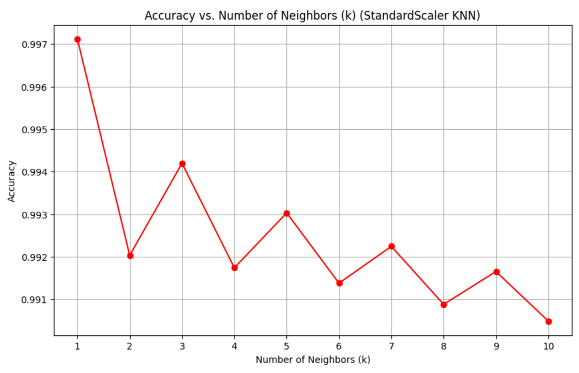


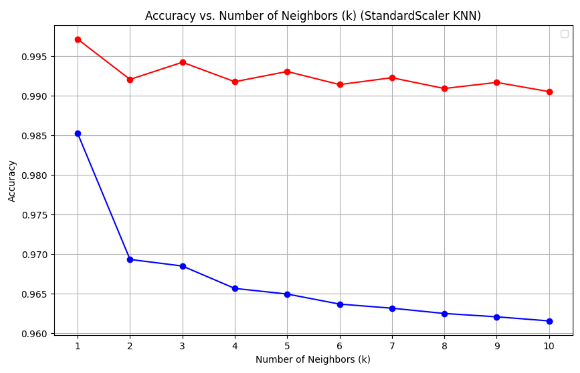
Next, we move on to the evaluation of training and prediction. Here, we perform KNN training without adding weights or normalizing the data. We directly train the KNN model. We also vary the parameter n_neighbors from 1 to 10 and record the accuracy for each case to create the chart as shown above. We can observe that when n_neighbors = 1, the highest accuracy is achieved (approximately 0.9852). We speculate that this phenomenon occurs because, after feature correlation analysis, we have already selected the most relevant features. Therefore, the nearest neighbors are more likely to be the correct class, reducing the chances of misclassification due to less relevant features.

8. StandardScaler Method

```
# Train & Predict (StandardScaler)

scaler = StandardScaler()
scaled_train_data = scaler.fit_transform(train_data)
scaled_test_data = scaler.fit_transform(test_data)

accuracies_Standard = []
for k in range(1, 11):
  knn = KNeighborsClassifier(n_neighbors = k)
  knn.fit(scaled_train_data, train_label)
  test_pred_standard = knn.predict(scaled_test_data)
  accuracy = accuracy_score(test_label, test_pred_standard)
  accuracies_Standard.append(accuracy)

plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), accuracies_Standard, marker='o', linestyle='-', color='r')
plt.title('Accuracy vs. Number of Neighbors (k) (StandardScaler KNN)')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.grid(True)
plt.xticks(range(1, 11))
plt.show()
```

```
Accuracy: [0.9971170832811695, 0.9920328562088776, 0.9941959131438746, 0.991740739195148, 0.9930309226724534,
           0.991379070511483, 0.9922449888021811, 0.9908817760714435, 0.9916537996077286, 0.9904888091363073]
```



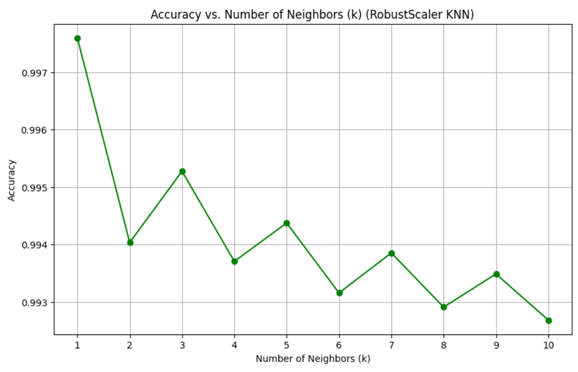


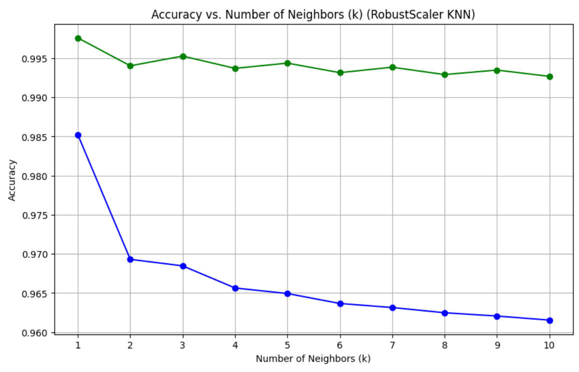
Next, we introduce a standardization technique called StandardScaler, which is one of the most common methods for standardization. It involves subtracting the mean and dividing by the standard deviation. After applying this technique to both the train_data and test_data, we conduct the same evaluation and create the chart as shown above. It's evident that data accuracy significantly improves after applying StandardScaler, with the highest accuracy still observed at k=1. Notably, the accuracy line exhibits a sawtooth pattern as k increases, with odd values achieving higher accuracy. However, the overall trend still shows a gradual decrease as k increases.

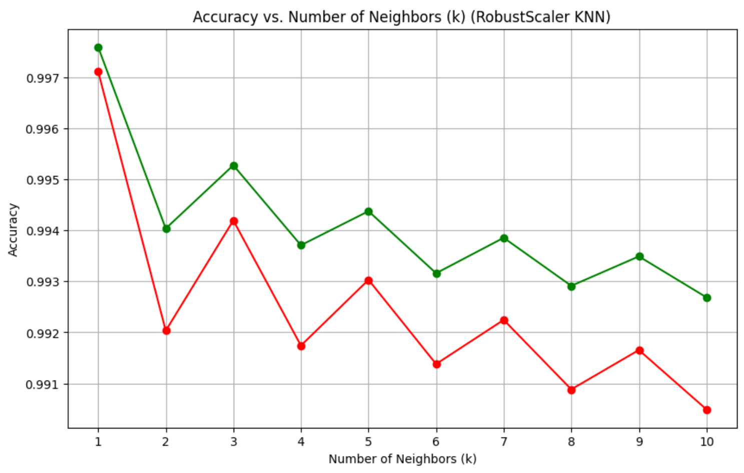
9. RobustScaler Method

```
# Train & Predict (RobustScaler)

scaler = RobustScaler()
scaled_train_data = scaler.fit_transform(train_data)
scaled_test_data = scaler.fit_transform(test_data)

accuracies_Robust = []
for k in range(1, 11):
  knn = KNeighborsClassifier(n_neighbors = k)
  knn.fit(scaled_train_data, train_label)
  test_pred_Robust = knn.predict(scaled_test_data)
  accuracy = accuracy_score(test_label, test_pred_Robust)
  accuracies_Robust.append(accuracy)

plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), accuracies_Robust, marker='o', linestyle='-', color='g')
plt.title('Accuracy vs. Number of Neighbors (k) (RobustScaler KNN)')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.grid(True)
plt.xticks(range(1, 11))
plt.show()
```

```
Accuracy: [0.9975969898037252, 0.9940359443030227, 0.9952774416113731, 0.9937090514543254, 0.9943767474857071,
           0.9931595932618342, 0.9938551099611902, 0.9929126848335629, 0.9934899636940283, 0.9926866419062722]
```







Next, we introduce another standardization technique called RobustScaler, which computes the center and range of the data using the median and quartiles. The key difference compared to the previous method lies in RobustScaler's ability to automatically exclude outliers to prevent them from affecting feature interpretation erroneously.

After applying this technique to both the train_data and test_data, we generate the chart as shown above. We can observe that this method yields significantly higher accuracy compared to KNN without standardization, and it also outperforms StandardScaler. The accuracy trend still exhibits a sawtooth pattern with decreasing accuracy as k increases.

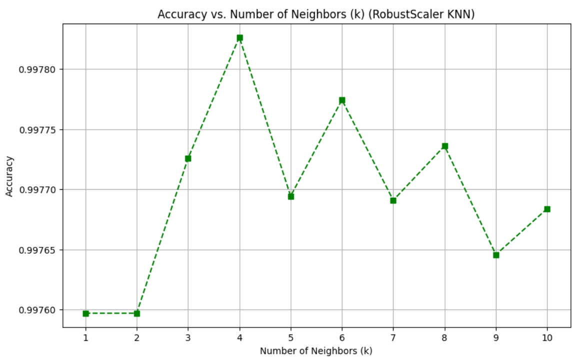
Since this data preprocessing approach results in the most significant accuracy improvement, we choose to use RobustScaler along with weighted k-NN for prediction.

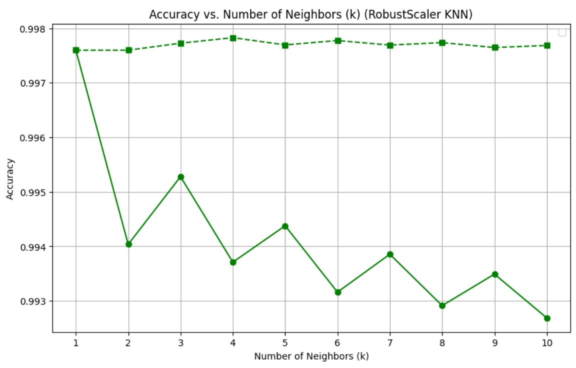
10. RobustScaler + Weight Method

```
# Train & Predict (RobustScaler + Weight)

scaler = RobustScaler()
scaled_train_data = scaler.fit_transform(train_data)
scaled_test_data = scaler.fit_transform(test_data)

accuracies_Robust_weight = []
for k in range(1, 11):
  knn = KNeighborsClassifier(n_neighbors = k, weights='distance')
  knn.fit(scaled_train_data, train_label)
  test_pred_Robust_weight = knn.predict(scaled_test_data)
```

```
    accuracy = accuracy_score(test_label, test_pred_Robust_weight)
    accuracies_Robust_weight.append(accuracy)

plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), accuracies_Robust, marker='o', linestyle='-', color='g')
plt.plot(range(1, 11), accuracies_Robust_weight, marker='s', linestyle='--', color='g')
plt.title('Accuracy vs. Number of Neighbors (k) (RobustScaler KNN)')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.grid(True)
plt.xticks(range(1, 11))
plt.legend()
plt.show()
```

```
Accuracy: [0.9975969898037252, 0.9975969898037252, 0.9977256603931061, 0.9978265103145126, 0.997694362141635,
           0.997774346562061, 0.9976908845581383, 0.9977360931435963, 0.9976456759726801, 0.9976839293911447]
```





We set the weight to "distance," meaning that closer data points have higher weights. After introducing weighted k-NN on top of RobustScaler, we observe a further increase in accuracy compared to using RobustScaler alone. Interestingly, the highest accuracy is achieved at k=4, not k=1. We believe that this phenomenon occurs because when each data point in space is weighted based on proximity, the model considers not only the nearest neighbor but also nearby data points with a degree of proximity. This helps prevent overfitting that might occur when considering only the nearest neighbor.

After reaching the highest accuracy at k=4, the accuracy gradually decreases as k increases, still exhibiting a sawtooth pattern. Notably, the accuracy remains higher than when considering only 1 nearest neighbor.

11. Organize The Output

```
# Save as .csv file

pred = test_pred_Robust_weight
idx_values = np.arange(len(pred))
result = np.column_stack((idx_values, pred))
result = pd.DataFrame(result, columns=['IDX', 'Target'])
result.to_csv('test_pred_Robust_weight.csv', index=False)
files.download('test_pred_Robust_weight.csv')
```

Based on the experiments conducted, we have observed that the accuracy significantly improves when the data is standardized and weighted between data points in the k-NN model. Ultimately, we choose the RobustScaler with distance-weighted k-NN, as it achieved the highest accuracy. We can now finalize the predictions and export them as a CSV file.