PDS Challenge1 Group 8

Our video: PDS_KNN_group8

1. Problem Setting: Data preprocessing:

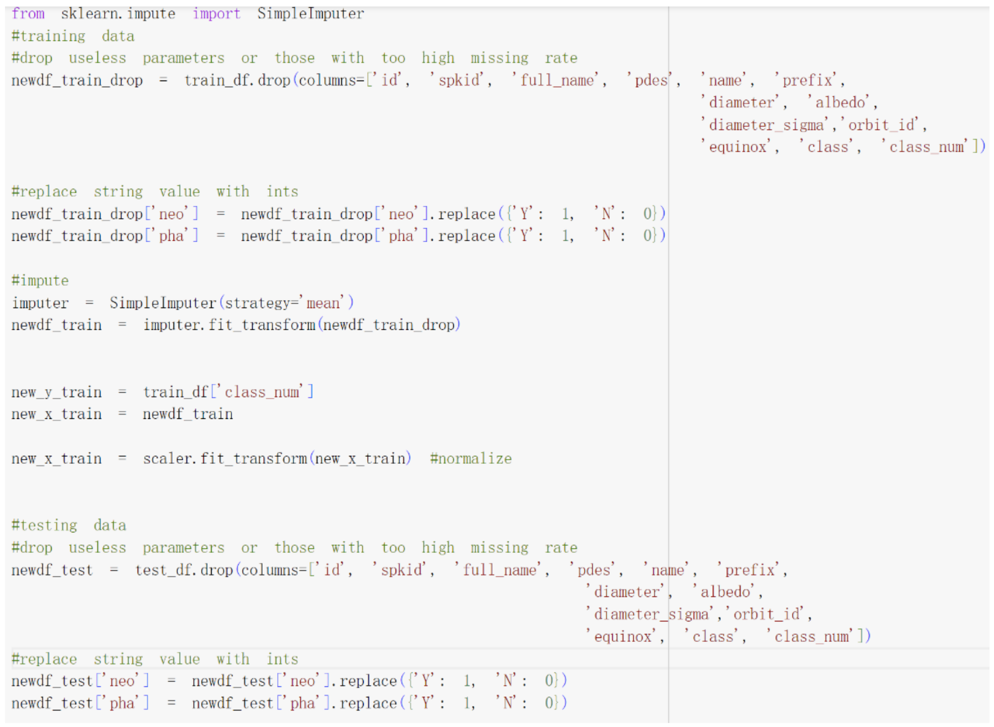
   We have 2 approaches in this step. The first approach uses the PCA method to reduce the dimensionality of a dataset, while the second approach is more rooted to domain knowledge and identifying the importance of the features in the dataset.

   However, generally, the 2 approaches stick to the 3 data-processing steps: extract features from the train and test dataset, imputing missing values, and feature standardization(normalization).

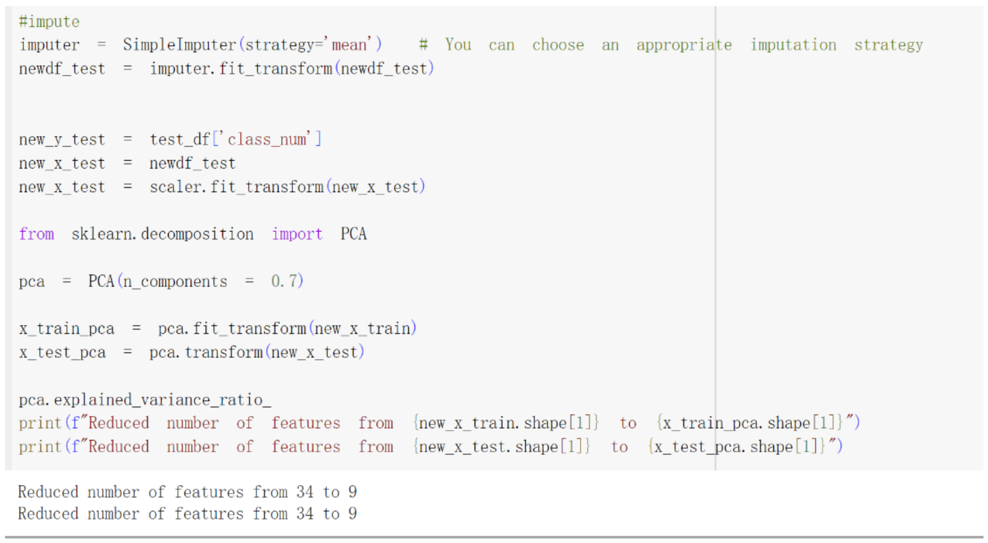
   The result is that Approach 2 gave better accuracy, so we sticked to it in algorithm developments.

   - Approach 1:

   (1) Drop apparently useless features such as 'ID' and 'name' etc. Also, features with too high missing rates are dropped, including 'diameter' and 'albedo', with missing rate `85.79527756208577 and 85.91212384513695` respectively.
   (2) Replace string values with ints
   (3) Impute missing values with mean values
   (4) Apply PCA (Principal component analysis) method:
   The amount of variance is set to 0.7 to tradeoff between computing time and correctness. After PCA, the features left to be computed are reduced from 34 to 9, which makes the code run efficiently.



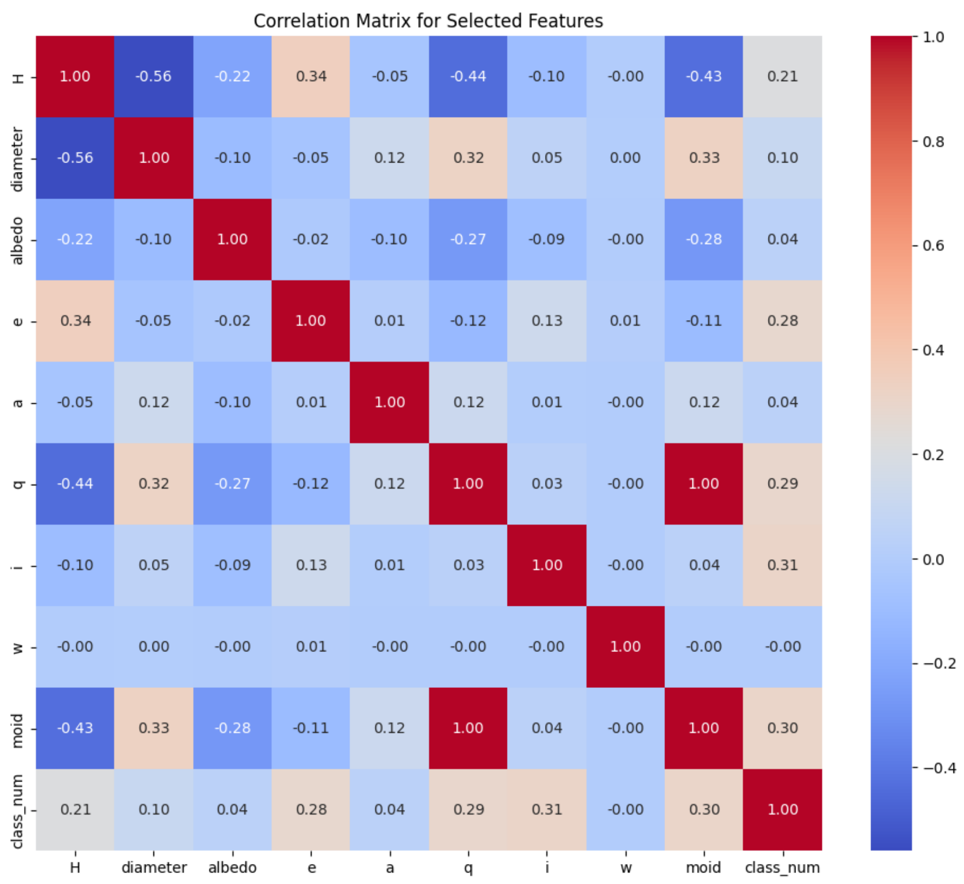
```
from sklearn.impute import SimpleImputer
#training data
#drop useless parameters or those with too high missing rate
newdf_train_drop = train_df.drop(columns=['id', 'spkid', 'full_name', 'pdes', 'name', 'prefix',
                                          'diameter', 'albedo',
                                          'diameter_sigma','orbit_id',
                                          'equinox', 'class', 'class_num'])


#replace string value with ints
newdf_train_drop['neo'] = newdf_train_drop['neo'].replace({'Y': 1, 'N': 0})
newdf_train_drop['pha'] = newdf_train_drop['pha'].replace({'Y': 1, 'N': 0})

#impute
imputer = SimpleImputer(strategy='mean')
newdf_train = imputer.fit_transform(newdf_train_drop)


new_y_train = train_df['class_num']
new_x_train = newdf_train

new_x_train = scaler.fit_transform(new_x_train) #normalize


#testing data
#drop useless parameters or those with too high missing rate
newdf_test = test_df.drop(columns=['id', 'spkid', 'full_name', 'pdes', 'name', 'prefix',
                                   'diameter', 'albedo',
                                   'diameter_sigma','orbit_id',
                                   'equinox', 'class', 'class_num'])
#replace string value with ints
newdf_test['neo'] = newdf_test['neo'].replace({'Y': 1, 'N': 0})
newdf_test['pha'] = newdf_test['pha'].replace({'Y': 1, 'N': 0})
```



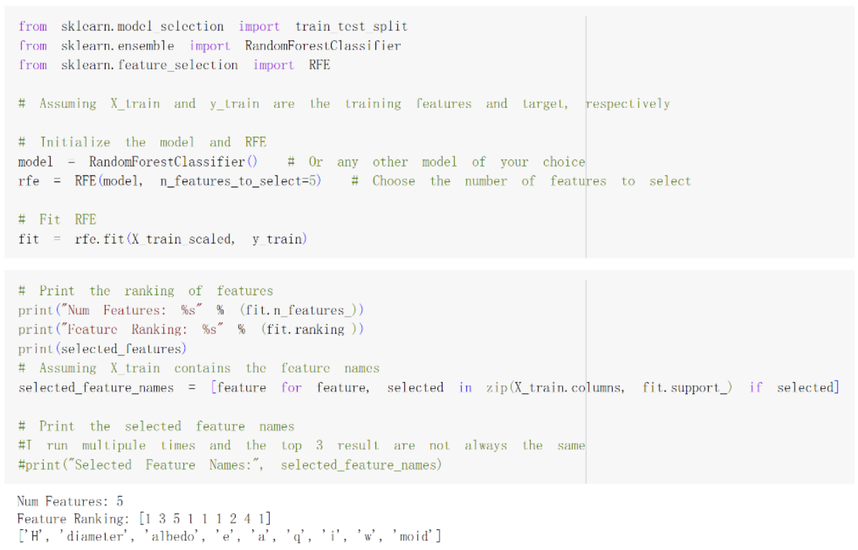
```
#impute
imputer = SimpleImputer(strategy='mean')  # You can choose an appropriate imputation strategy
newdf_test = imputer.fit_transform(newdf_test)


new_y_test = test_df['class_num']
new_x_test = newdf_test
new_x_test = scaler.fit_transform(new_x_test)

from sklearn.decomposition import PCA

pca = PCA(n_components = 0.7)

x_train_pca = pca.fit_transform(new_x_train)
x_test_pca = pca.transform(new_x_test)

pca.explained_variance_ratio_
print(f"Reduced number of features from {new_x_train.shape[1]} to {x_train_pca.shape[1]}")
print(f"Reduced number of features from {new_x_test.shape[1]} to {x_test_pca.shape[1]}")
```

```
Reduced number of features from 34 to 9
Reduced number of features from 34 to 9
```

- Approach 2:
  (1) We trimmed the datasets, only leaving 9 features: 'H', 'diameter', 'albedo', 'e', 'a', 'q', 'i', 'w', and 'moid'. The decision is based on our understanding of features by ChatGPT and the description from UMD (https://pdssbn.astro.umd.edu/data_other/objclass.shtml).
  (2) Try to get more information from the Correlation Matrix on the 9 selected features. However, we didn't get much sense about the relations between the selected features.



(3) Rather, we used RFE(Recursive Feature Elimination) to look for important features. We found that H, e, a, q, moid are most influential. After some trials , we concluded that e, a, q may be the best features. Therefore, we trimmed the dataset the second time, leaving only 3 features. This dataset is used in our algorithms later.

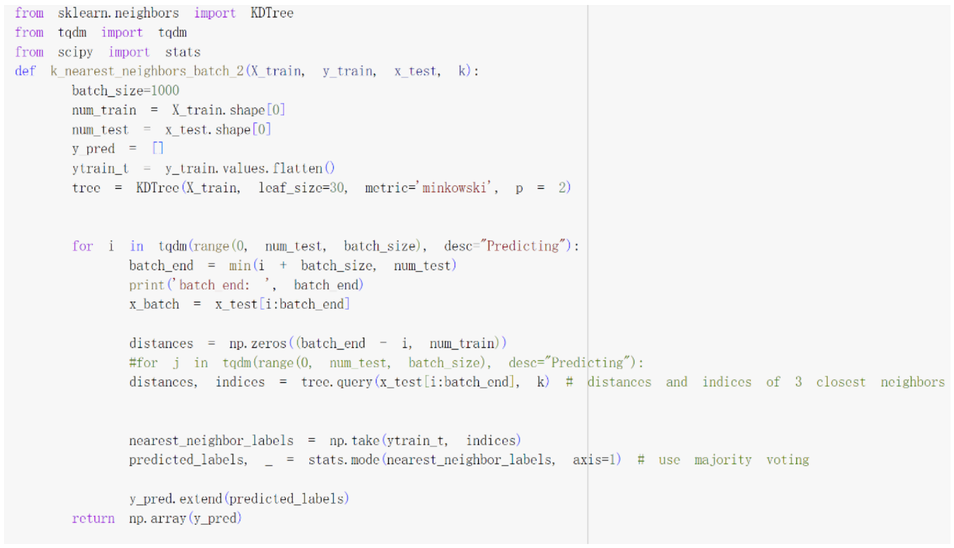


```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import RFE

# Assuming X_train and y_train are the training features and target, respectively

# Initialize the model and RFE
model = RandomForestClassifier()  # Or any other model of your choice
rfe = RFE(model, n_features_to_select=5)  # Choose the number of features to select

# Fit RFE
fit = rfe.fit(X_train_scaled, y_train)
```

```
# Print the ranking of features
print("Num Features: %s" % (fit.n_features_))
print("Feature Ranking: %s" % (fit.ranking_))
print(selected_features)
# Assuming X_train contains the feature names
selected_feature_names = [feature for feature, selected in zip(X_train.columns, fit.support_) if selected]

# Print the selected feature names
#I run multiple times and the top 3 result are not always the same
#print("Selected Feature Names:", selected_feature_names)
```

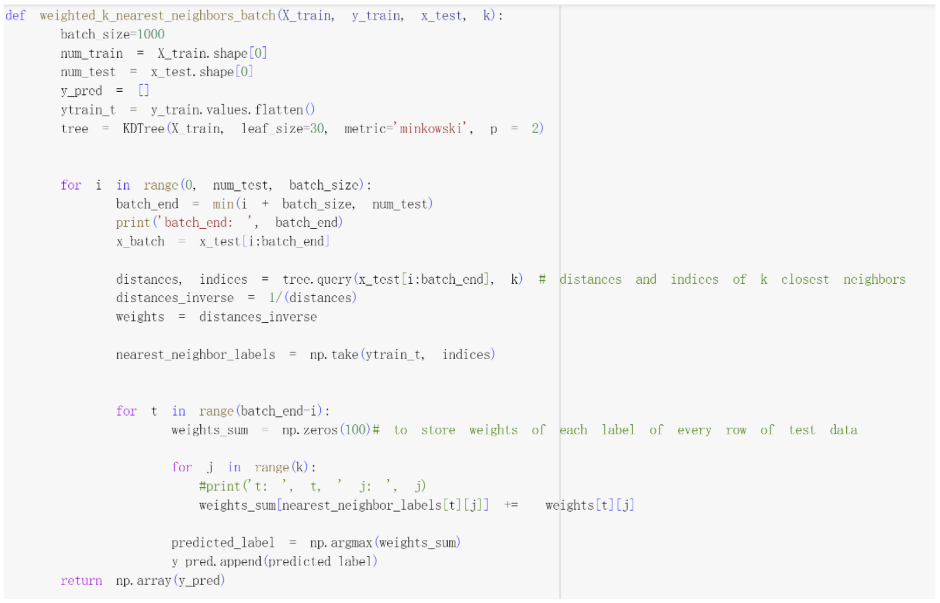
```
Num Features: 5
Feature Ranking: [1 3 5 1 1 1 2 4 1]
['H', 'diameter', 'albedo', 'e', 'a', 'q', 'i', 'w', 'moid']
```

2. Classic kNN Algorithm Development
   The classic kNN algorithm is developmented with a KDTree and batch computing (batch size = 1000) to achieve better performance. For k>1, we use majority voting to decide prediction results.
   - The kDTree returns the k smallest Euclidean distance between each entry in the test dataset with the training dataset.
   - Then we do majority voting.



```
from sklearn.neighbors import KDTree
from tqdm import tqdm
from scipy import stats
def k_nearest_neighbors_batch_2(X_train, y_train, x_test, k):
        batch_size=1000
        num_train = X_train.shape[0]
        num_test = x_test.shape[0]
        y_pred = []
        ytrain_t = y_train.values.flatten()
        tree = KDTree(X_train, leaf_size=30, metric='minkowski', p = 2)


        for i in tqdm(range(0, num_test, batch_size), desc="Predicting"):
                batch_end = min(i + batch_size, num_test)
                print('batch end: ', batch end)
                x_batch = x_test[i:batch_end]

                distances = np.zeros((batch_end - i, num_train))
                #for j in tqdm(range(0, num_test, batch_size), desc="Predicting"):
                distances, indices = tree.query(x_test[i:batch_end], k) # distances and indices of 3 closest neighbors


                nearest_neighbor_labels = np.take(ytrain_t, indices)
                predicted_labels, _ = stats.mode(nearest_neighbor_labels, axis=1) # use majority voting

                y_pred.extend(predicted_labels)
        return np.array(y_pred)
```

3. Weighted kNN Algorithm Development
   In our user-defined function, we used “inverse weights technique” to calculate the weights. We compute the inverse of each distance, then each weight is a voted for its associated class. Other implementation strategies are similar to the classic kNN algorithm.

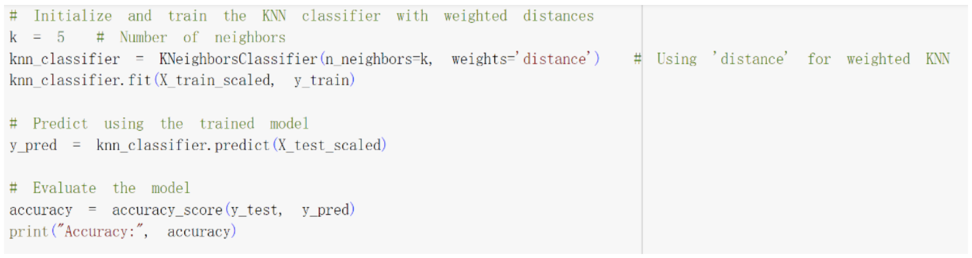


```python
def  weighted_k_nearest_neighbors_batch(X_train,  y_train,  x_test,  k):
        batch_size=1000
        num_train  =  X_train.shape[0]
        num_test  =  x_test.shape[0]
        y_pred  =  []
        ytrain_t  =  y_train.values.flatten()
        tree  =  KDTree(X_train,  leaf_size=30,  metric='minkowski',  p  =  2)


        for  i  in  range(0,  num_test,  batch_size):
                batch_end  =  min(i  +  batch_size,  num_test)
                print('batch_end:  ',  batch_end)
                x_batch  =  x_test[i:batch_end]

                distances,  indices  =  tree.query(x_test[i:batch_end],  k)  #  distances  and  indices  of  k  closest  neighbors
                distances_inverse  =  1/(distances)
                weights  =  distances_inverse

                nearest_neighbor_labels  =  np.take(ytrain_t,  indices)


                for  t  in  range(batch_end-i):
                        weights_sum  =  np.zeros(100)#  to  store  weights  of  each  label  of  every  row  of  test  data

                        for  j  in  range(k):
                            #print('t:  ',  t,  '  j:  ',  j)
                            weights_sum[nearest_neighbor_labels[t][j]]  +=    weights[t][j]

                        predicted_label  =  np.argmax(weights_sum)
                        y_pred.append(predicted_label)
        return  np.array(y_pred)
```

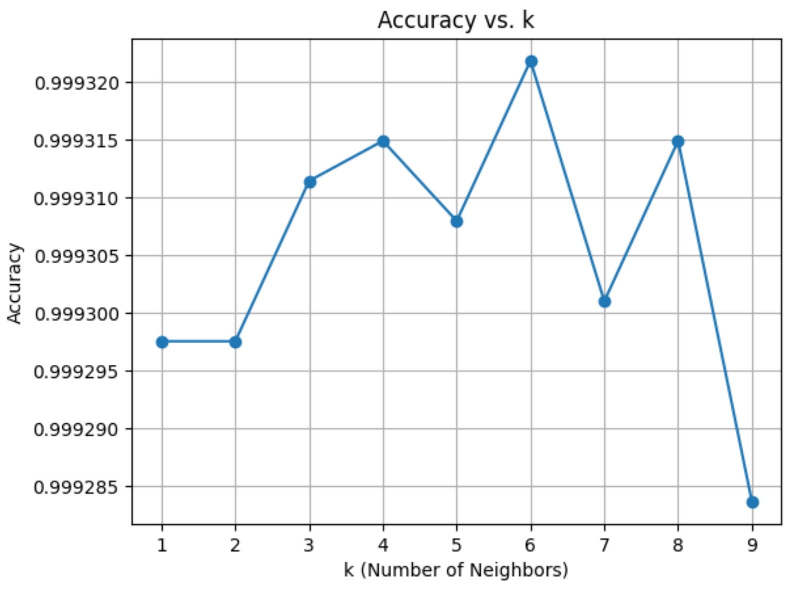
Besides, another simpler approach is to apply the `KNeighborsClassifier()` provided by python. We set parameters: `n_neighbors = k, weights = 'distance'`.

- `'distance'`: weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

Algorithm evaluation and classification report is provided below.



```python
#  Initialize  and  train  the  KNN  classifier  with  weighted  distances
k  =  5    #  Number  of  neighbors
knn_classifier  =  KNeighborsClassifier(n_neighbors=k,  weights='distance')    #  Using  'distance'  for  weighted  KNN
knn_classifier.fit(X_train_scaled,  y_train)

#  Predict  using  the  trained  model
y_pred  =  knn_classifier.predict(X_test_scaled)

#  Evaluate  the  model
accuracy  =  accuracy_score(y_test,  y_pred)
print("Accuracy:",  accuracy)
```



4. Evaluating and Visualizing the Performance

   (1) Performance Metrics

   To evaluate the performance of our classification model, we use common classification metrics from scikit-learn. The assessment metrics we used and the results provided below.

   Accuracy: 0.9613780919956739

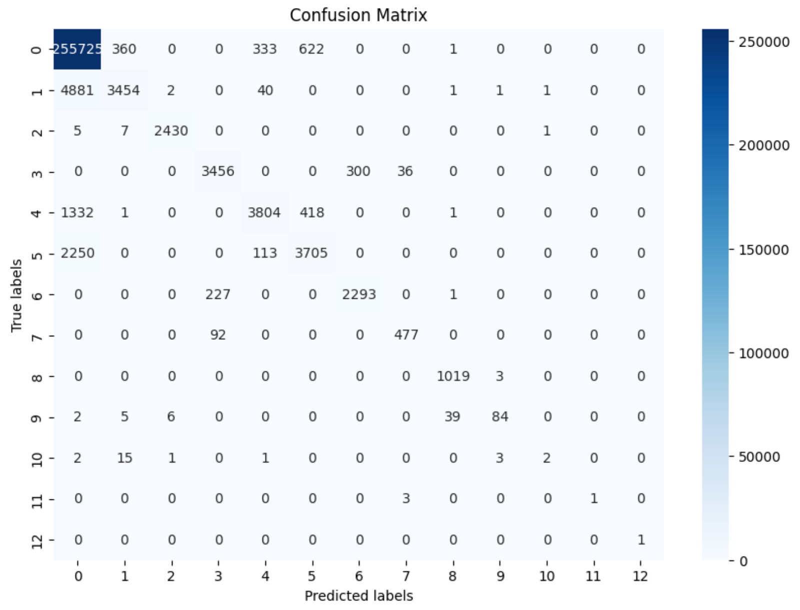
   Precision: 0.9590435018283396

   Recall: 0.9613780919956739

   F1 Score: 0.9569043541608467

   (2) Confusion Matrix Visualization

   After that, we generate a confusion matrix and visualize it using a heatmap. Each cell in the matrix represents a combination of predicted and true labels, making it easy to identify areas where the model excels and where it may need improvement.

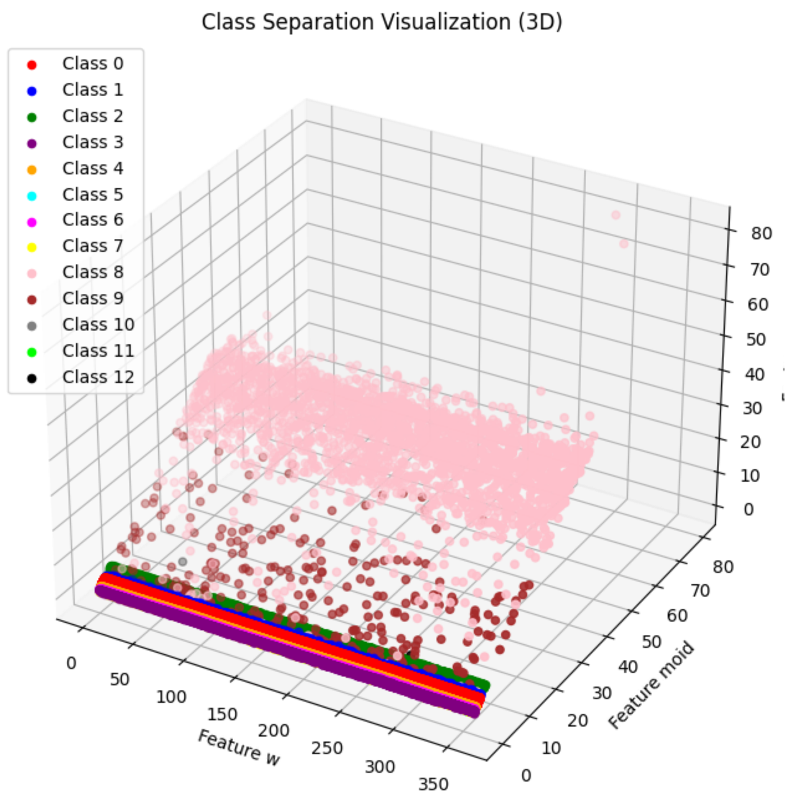


Confusion Matrix

True labels \ Predicted labels	0	1	2	3	4	5	6	7	8	9	10	11	12
0	255725	360	0	0	333	622	0	0	1	0	0	0	0
1	4881	3454	2	0	40	0	0	0	1	1	1	0	0
2	5	7	2430	0	0	0	0	0	0	0	1	0	0
3	0	0	0	3456	0	0	300	36	0	0	0	0	0
4	1332	1	0	0	3804	418	0	0	1	0	0	0	0
5	2250	0	0	0	113	3705	0	0	0	0	0	0	0
6	0	0	0	227	0	0	2293	0	1	0	0	0	0
7	0	0	0	92	0	0	0	477	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1019	3	0	0	0
9	2	5	6	0	0	0	0	0	39	84	0	0	0
10	2	15	1	0	1	0	0	0	0	3	2	0	0
11	0	0	0	0	0	0	0	3	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1

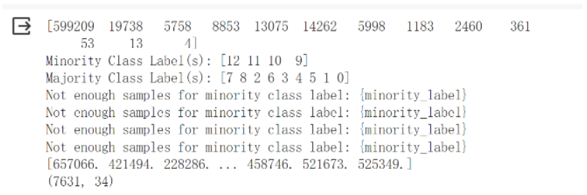
   (3) Class Separation Visualization

   To gain deeper insights into the classification task, we take advantage of the mpl_toolkits library to create a 3D scatter plot. Each class is represented by a unique color, and three features (w, moid, q) are visualized in a three-dimensional space. This provides a vivid illustration of how well-separated the classes are based on these features.



5. Regularized kNN for Balanced Classification

To understand how imbalanced our dataset is, we counted the number of entries in each class label. Among the 13 kinds of labels, label 0 is the most dominant, having 599209 entries, while only 4 entries belong to label 12.

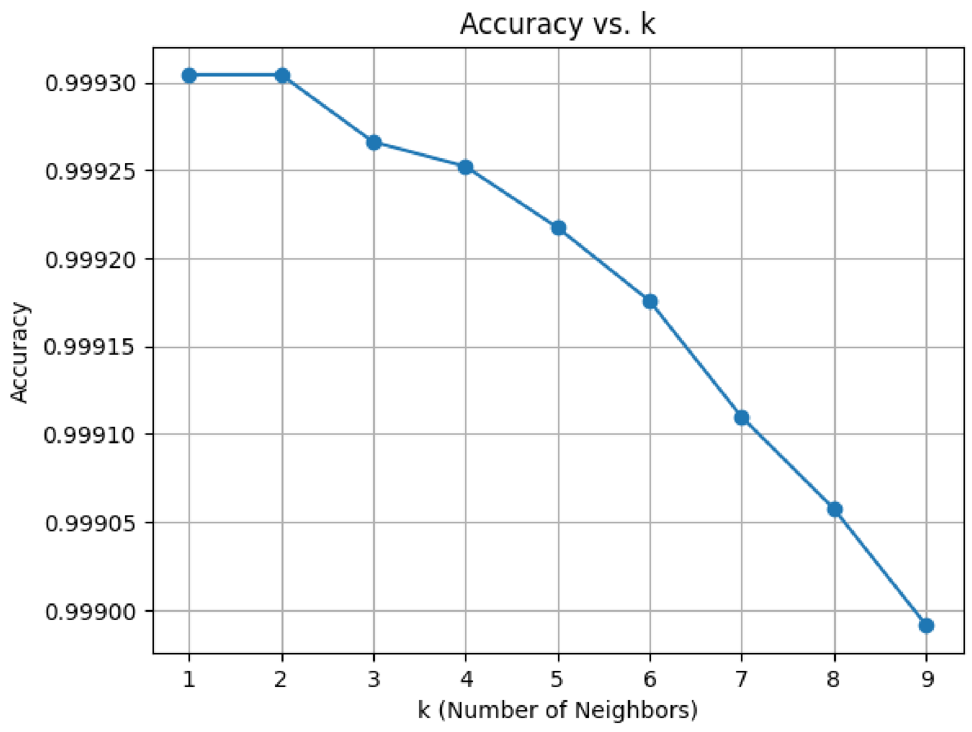


```
[599209  19738   5758   8853  13075  14262   5998   1183   2460    361
     53     13      4]
Minority Class Label(s): [12 11 10  9]
Majority Class Label(s): [7 8 2 6 3 4 5 1 0]
Not enough samples for minority class label: {minority_label}
Not enough samples for minority class label: {minority_label}
Not enough samples for minority class label: {minority_label}
Not enough samples for minority class label: {minority_label}
[657066. 421494. 228286. ... 458746. 521673. 525349.]
(7631, 34)
```

To address potential bias that may arise due to disparities in class label frequencies, we tried to apply oversampling. Oversampling involves increasing the number of instances in the minority class by generating synthetic examples to balance the class distribution and improve the model's ability to make accurate predictions for both the majority and minority classes.

In implementation, we applied the `BorderlineSMOTE()` function to our training data. The accuracy improved comparing to the weighted kNN algorithm without oversampling. The best case is when k = 1, with accuracy `0.999304485719353`.

```
from imblearn.over_sampling import SMOTE, BorderlineSMOTE, ADASYN
# Instantiate the SMOTE class
#smote = SMOTE(sampling_strategy='auto',k_neighbors=3, random_state=42)
borderline_smote = BorderlineSMOTE(sampling_strategy='auto', k_neighbors=3, random_state=42, kind='borderline-1')

# Fit and apply SMOTE to your training data
x_train_resampled, y_train_resampled = borderline_smote.fit_resample(X_train_scaled, y_train)
```


Accuracy vs. k
Accuracy
k (Number of Neighbors)