



## kNN Classification Algorithm on Iris Dataset

### ▼ Author - Abderrahman Alami-Afilal

```
#importing the required libraries needed to set up and executre the algorithm
import pandas as pd
import numpy as np
import operator
import matplotlib.pyplot as plt

#reading iris data from the file using pandas
from google.colab import files
uploaded = files.upload()
import io
data = pd.read_csv(
    io.BytesIO(uploaded['iris.csv']),
    names=['sepal_length','sepal_width', 'petal_length','petal_width', 'class'])
print(data)
```

  iris.csv

- **iris.csv**(text/csv) - 3907 bytes, last modified: n/a - 100% done

Saving iris.csv to iris.csv

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
2	4.7	3.2	1.3	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
4	5.0	3.6	1.4	0.2	Setosa
..	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Virginica
146	6.3	2.5	5.0	1.9	Virginica
147	6.5	3.0	5.2	2.0	Virginica
148	6.2	3.4	5.4	2.3	Virginica
149	5.9	3.0	5.1	1.8	Virginica

[150 rows x 5 columns]

### ▼ Part 1)

Processing the data into development(ie, training) and and test.

```
#initialize indices randomization
indices = np.random.permutation(data.shape[0])
div = int(0.75 * len(indices))
development_id, test_id = indices[:div], indices[div:]
```

```
#we divide the dataset using the randomized indices
development_set, test_set = data.loc[development_id,:], data.loc[test_id,:]
print("Development Set:\n", development_set, "\n\nTest Set:\n", test_set)
mean_development_set = development_set.mean()
mean_test_set = test_set.mean()
std_development_set = development_set.std()
std_test_set = test_set.std()
```

```
Development Set:
      sepal_length  sepal_width  petal_length  petal_width      class
42              4.4          3.2           1.3           0.2      Setosa
39              5.1          3.4           1.5           0.2      Setosa
113             5.7          2.5           5.0           2.0  Virginica
36              5.5          3.5           1.3           0.2      Setosa
132             6.4          2.8           5.6           2.2  Virginica
..             ...           ...           ...           ...      ...
23              5.1          3.3           1.7           0.5      Setosa
93              5.0          2.3           3.3           1.0  Versicolor
20              5.4          3.4           1.7           0.2      Setosa
83              6.0          2.7           5.1           1.6  Versicolor
74              6.4          2.9           4.3           1.3  Versicolor
```

```
[112 rows x 5 columns]
```

```
Test Set:
      sepal_length  sepal_width  petal_length  petal_width      class
97              6.2          2.9           4.3           1.3  Versicolor
124             6.7          3.3           5.7           2.1  Virginica
63              6.1          2.9           4.7           1.4  Versicolor
140             6.7          3.1           5.6           2.4  Virginica
34              4.9          3.1           1.5           0.2      Setosa
80              5.5          2.4           3.8           1.1  Versicolor
38              4.4          3.0           1.3           0.2      Setosa
107             7.3          2.9           6.3           1.8  Virginica
15              5.7          4.4           1.5           0.4      Setosa
70              5.9          3.2           4.8           1.8  Versicolor
59              5.2          2.7           3.9           1.4  Versicolor
26              5.0          3.4           1.6           0.4      Setosa
21              5.1          3.7           1.5           0.4      Setosa
11              4.8          3.4           1.6           0.2      Setosa
142             5.8          2.7           5.1           1.9  Virginica
128             6.4          2.8           5.6           2.1  Virginica
84              5.4          3.0           4.5           1.5  Versicolor
5              5.4          3.9           1.7           0.4      Setosa
65             6.7          3.1           4.4           1.4  Versicolor
27             5.2          3.5           1.5           0.2      Setosa
4              5.0          3.6           1.4           0.2      Setosa
131            7.9          3.8           6.4           2.0  Virginica
87             6.3          2.3           4.4           1.3  Versicolor
8              4.4          2.9           1.4           0.2      Setosa
51             6.4          3.2           4.5           1.5  Versicolor
53             5.5          2.3           4.0           1.3  Versicolor
79             5.7          2.6           3.5           1.0  Versicolor
```

28	5.2	3.4	1.4	0.2	Setosa
29	4.7	3.2	1.6	0.2	Setosa
25	5.0	3.0	1.6	0.2	Setosa
116	6.5	3.0	5.5	1.8	Virginica
86	6.7	3.1	4.7	1.5	Versicolor
14	5.8	4.0	1.2	0.2	Setosa
134	6.1	2.6	5.6	1.4	Virginica
1	4.9	3.0	1.4	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
60	5.0	2.0	3.5	1.0	Versicolor
125	7.2	3.2	6.0	1.8	Virginica

```
<ipython-input-5-9a2792f01e2d>:8: FutureWarning: Dropping of nuisance columns
mean_development_set = development_set.mean()
```

```
<ipython-input-5-9a2792f01e2d>:9: FutureWarning: Dropping of nuisance columns
mean_test_set = test_set.mean()
```

## ▼ Part 2)

Implement kNN using the following hyperparameters:

number of neighbor

\* 1,3,5,7

distance metric

\* euclidean distance  
 \* normalized euclidean distance  
 \* cosine similarity

Retrieving the 'class' column from the development and test sets and storing it in separate lists.  
 Calculating the mean and standard deviation of the development set and test set for normalizing the data.

```

# extract the class column from the test and dev set
test_class = list(test_set.iloc[:,-1])
dev_class = list(development_set.iloc[:,-1])

# calculate the mean of the development and test set
mean_development_set = development_set.mean()
mean_test_set = test_set.mean()

# calculate the standard deviation of the development and test set
std_development_set = development_set.std()
std_test_set = test_set.std()

<ipython-input-6-b3cdcc5dd217>:6: FutureWarning: Dropping of nuisance columns
mean_development_set = development_set.mean()
<ipython-input-6-b3cdcc5dd217>:7: FutureWarning: Dropping of nuisance columns
mean_test_set = test_set.mean()
<ipython-input-6-b3cdcc5dd217>:10: FutureWarning: Dropping of nuisance column
std_development_set = development_set.std()
<ipython-input-6-b3cdcc5dd217>:11: FutureWarning: Dropping of nuisance column
std_test_set = test_set.std()

```

Functions for computing the Euclidean Distance, Normalized Euclidean Distance, Cosine Similarity and k Nearest Neighbor to determine the 'class' for a given input instance.

```

# define function that calculates euclidean distance between two data points
def euclideanDistance(data_1, data_2, data_len):
    dist = 0
    for i in range(data_len):
        dist = dist + np.square(data_1[i] - data_2[i])
    return np.sqrt(dist)

# this function finds the normalized euclidean distance between data points
def normalizedEuclideanDistance(data_1, data_2, data_len, data_mean, data_std):
    n_dist = 0
    for i in range(data_len):
        n_dist = n_dist + (np.square(((data_1[i] - data_mean[i])/data_std[i]) - ((
    return np.sqrt(n_dist)

# Title: Cosine Similarity between 2 Number Lists
# Author: dontloo
# Date: 03.27.2017
# Code version: 1
# Availability: https://stackoverflow.com/questions/18424228/cosine-similarity-between-two-arrays
def cosineSimilarity(data_1, data_2):
    dot = np.dot(data_1, data_2[:,-1])
    norm_data_1 = np.linalg.norm(data_1)
    norm_data_2 = np.linalg.norm(data_2[:,-1])

```

```
cos = dot / (norm_data_1 * norm_data_2)
return (1-cos)
```

```
# method to generate most frequent class
```

```
def knn(dataset, testInstance, k, dist_method, dataset_mean, dataset_std):
    distances = {}
    length = testInstance.shape[1]
    if dist_method == 'euclidean':
        for x in range(len(dataset)):
            dist_up = euclideanDistance(testInstance, dataset.iloc[x], length)
            distances[x] = dist_up[0]
    elif dist_method == 'normalized_euclidean':
        for x in range(len(dataset)):
            dist_up = normalizedEuclideanDistance(testInstance, dataset.iloc[x], 1)
            distances[x] = dist_up[0]
    elif dist_method == 'cosine':
        for x in range(len(dataset)):
            dist_up = cosineSimilarity(testInstance, dataset.iloc[x])
            distances[x] = dist_up[0]
    # Sort values based on distance
    sort_distances = sorted(distances.items(), key=operator.itemgetter(1))
    neighbors = []
    # Extracting nearest k neighbors
    for x in range(k):
        neighbors.append(sort_distances[x][0])
    # Initializing counts for 'class' labels counts as 0
    counts = {"Iris-setosa" : 0, "Iris-versicolor" : 0, "Iris-virginica" : 0}
    # Computing the most frequent class
    for x in range(len(neighbors)):
        response = dataset.iloc[neighbors[x]][-1]
        if response in counts:
            counts[response] += 1
        else:
            counts[response] = 1
    # Sorting the class in reverse order to get the most frequent class
    sort_counts = sorted(counts.items(), key=operator.itemgetter(1), reverse=True)
    return(sort_counts[0][0])
```

## ▼ Part c)

Using the development data set

Iterating all of the development data points and computing the class for each k and each distance metric

```

# Creating a list of list of all columns except 'class' by iterating through the d
row_list = []
for index, rows in development_set.iterrows():
    my_list = [rows.sepal_length, rows.sepal_width, rows.petal_length, rows.petal_w
    row_list.append([my_list])
# k values for the number of neighbors that need to be considered
k_n = [1, 3, 5, 7]
# Distance metrics
distance_methods = ['euclidean', 'normalized_euclidean', 'cosine']
# Performing kNN on the development set by iterating all of the development set da
obs_k = {}
for dist_method in distance_methods:
    development_set_obs_k = {}
    for k in k_n:
        development_set_obs = []
        for i in range(len(row_list)):
            development_set_obs.append(knn(development_set, pd.DataFrame(row_list[
            development_set_obs_k[k] = development_set_obs
# Nested Dictionary containing the observed class for each k and each distance
obs_k[dist_method] = development_set_obs_k
print(dist_method.upper() + " distance method performed on the dataset for all
#print(obs_k)

    EUCLIDEAN distance method performed on the dataset for all k values!
    NORMALIZED_EUCLIDEAN distance method performed on the dataset for all k value
    COSINE distance method performed on the dataset for all k values!

```

Computing the accuracy for the development data set and finding the optimal hyperparameters

```

# Calculating the accuracy of the development set by comparing it with the develo
# initialize a dictionary that will hold our accuracy results
accuracy = {}
for key in obs_k.keys():
    accuracy[key] = {}
    for k_value in obs_k[key].keys():

        count = 0
        for i,j in zip(dev_class, obs_k[key][k_value]):
            if i == j:
                count = count + 1
            else:
                pass
        accuracy[key][k_value] = count/(len(dev_class))

# Storing the accuracy for each k and each distance metric into a dataframe
df_res = pd.DataFrame({'k': k_n})
for key in accuracy.keys():

```

```

for key in accuracy.keys():
    value = list(accuracy[key].values())
    df_res[key] = value
print(df_res)

# Plotting a Bar Chart for accuracy
draw = df_res.plot(x='k', y=['euclidean', 'normalized_euclidean', 'cosine'], kind=
draw.set(ylabel='Accuracy')

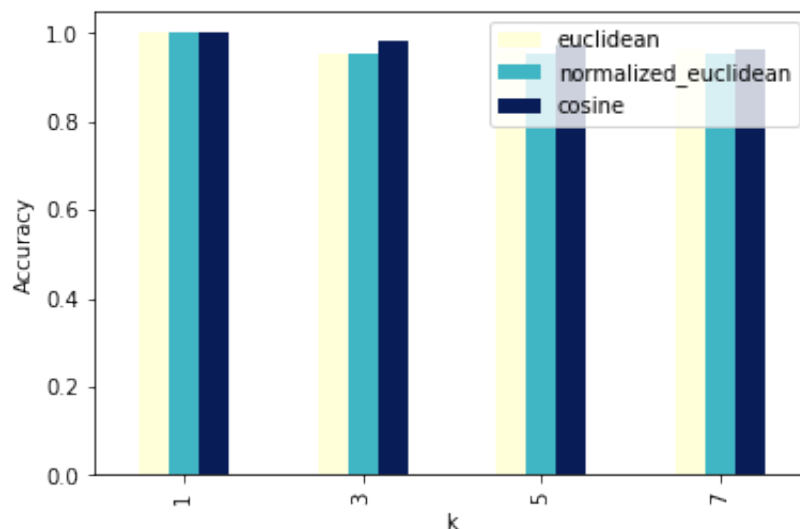
# Ignoring k=1 if the value of accuracy for k=1 is 100%, since this mostly implie
df_res.loc[df_res['k'] == 1.0, ['euclidean', 'normalized_euclidean', 'cosine']] =

# Fetching the best k value for using all hyper-parameters
# In case the accuracy is the same for different k and different distance metric
column_val = [c for c in df_res.columns if not c.startswith('k')]
col_max = df_res[column_val].max().idxmax(0)
best_dist_method = col_max
row_max = df_res[col_max].argmax()
best_k = int(df_res.iloc[row_max]['k'])
if df_res.isnull().values.any():
    print('\n\n\nBest k value is\033[1m', best_k, '\033[0mand best distance metri
else:
    print('\n\n\nBest k value is\033[1m', best_k, '\033[0mand best distance metri

```

	k	euclidean	normalized_euclidean	cosine
0	1	1.000000	1.000000	1.000000
1	3	0.955357	0.955357	0.982143
2	5	0.964286	0.955357	0.973214
3	7	0.964286	0.955357	0.964286

Best k value is **3** and best distance metric is **cosine** . Ignoring k=1 if the va



## ▼ Part d)

Using the test dataset

```
print('\n\n\nBest k value is\033[1m', best_k, '\033[0mand best distance metric is\
```

```
Best k value is 3 and best distance metric is cosine
```

Using the best k value and best distance metric to determine the class for all rows in the test dataset

```
# Creating a list of list of all columns except 'class' by iterating through the
row_list_test = []
for index, rows in test_set.iterrows():
    my_list = [rows.sepal_length, rows.sepal_width, rows.petal_length, rows.petal_
    row_list_test.append([my_list])
test_set_obs = []
for i in range(len(row_list_test)):
    test_set_obs.append(knn(test_set, pd.DataFrame(row_list_test[i]), best_k, bes
#print(test_set_obs)

count = 0
for i,j in zip(test_class, test_set_obs):
    if i == j:
        count = count + 1
    else:
        pass
accuracy_test = count/(len(test_class))
print('Final Accuracy of the Test dataset is ', accuracy_test)
```

```
Final Accuracy of the Test dataset is 1.0
```



[Colab paid products](#) - [Cancel contracts here](#)

---

✓ 0s completed at 4:53 PM

