In [33]:
```
import pandas as pd
import numpy as np
import operator
import matplotlib.pyplot as plt
```

In [ ]:
```
# reads a CSV file named iris data andnstores in Pandas DataFrame
data = pd.read_csv('iris.csv', header=None, names=['sepal_length', 'se
pal_width', 'petal_length', 'petal_width', 'class'])
print(data)
```

```
     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]
```

In [ ]:
```
# generates a randomized array of integers from 0 to the number of row
s
indices = np.random.permutation(data.shape[0])

# calculating the index on which data will be divided
div = int(0.75 * len(indices))

# dividing the indexes into two array
development_id, test_id = indices[:div], indices[div:]

# using loc method of pandas dataframe which will select a particular
row and its all columns
# its just like splitting into training set and testing set

# ????????????????? i think we should exclude last column here ??????
????????????? if not then we should in euclidean distance

development_set, test_set = data.loc[development_id,:], data.loc[test_
id,:]
print("Development Set:\n", development_set, "\n\nTest Set:\n", test_s
et)
```

Development Set:

```
Development Set:
      sepal_length  sepal_width  petal_length  petal_width       cla
ss
37             4.9          3.6           1.4          0.1      Setos
a
77             6.7          3.0           5.0          1.7  Versicolo
r
119            6.0          2.2           5.0          1.5   Virginic
a
8              4.4          2.9           1.4          0.2      Setos
a
145            6.7          3.0           5.2          2.3   Virginic
a
..             ...          ...           ...          ...         ..
.
102            7.1          3.0           5.9          2.1   Virginic
a
149            5.9          3.0           5.1          1.8   Virginic
a
76             6.8          2.8           4.8          1.4  Versicolo
r
139            6.9          3.1           5.4          2.1   Virginic
a
143            6.8          3.2           5.9          2.3   Virginic
a

[112 rows x 5 columns]

Test Set:
      sepal_length  sepal_width  petal_length  petal_width       cla
ss
4              5.0          3.6           1.4          0.2      Setos
a
86             6.7          3.1           4.7          1.5  Versicolo
r
22             4.6          3.6           1.0          0.2      Setos
a
40             5.0          3.5           1.3          0.3      Setos
a
117            7.7          3.8           6.7          2.2   Virginic
a
118            7.7          2.6           6.9          2.3   Virginic
a
142            5.8          2.7           5.1          1.9   Virginic
a
85             6.0          3.4           4.5          1.6  Versicolo
r
103            6.3          2.9           5.6          1.8   Virginic
a
16             5.4          3.9           1.3          0.4      Setos
```

a

112	6.8	3.0	5.5	2.1	Virginica
136	6.3	3.4	5.6	2.4	Virginica
100	6.3	3.3	6.0	2.5	Virginica
93	5.0	2.3	3.3	1.0	Versicolor
42	4.4	3.2	1.3	0.2	Setosa
95	5.7	3.0	4.2	1.2	Versicolor
32	5.2	4.1	1.5	0.1	Setosa
66	5.6	3.0	4.5	1.5	Versicolor
98	5.1	2.5	3.0	1.1	Versicolor
55	5.7	2.8	4.5	1.3	Versicolor
0	5.1	3.5	1.4	0.2	Setosa
35	5.0	3.2	1.2	0.2	Setosa
65	6.7	3.1	4.4	1.4	Versicolor
26	5.0	3.4	1.6	0.4	Setosa
45	4.8	3.0	1.4	0.3	Setosa
29	4.7	3.2	1.6	0.2	Setosa
60	5.0	2.0	3.5	1.0	Versicolor
11	4.8	3.4	1.6	0.2	Setosa
122	7.7	2.8	6.7	2.0	Virginica
6	4.6	3.4	1.4	0.3	Setosa
106	4.9	2.5	4.5	1.7	Virginica
111	6.4	2.7	5.3	1.9	Virginica
120	6.9	3.2	5.7	2.3	Virginica
113	5.7	2.5	5.0	2.0	Virginica
49	5.0	3.3	1.4	0.2	Setos

```
a
71             6.1          2.8           4.0          1.3  Versicolo
r
94             5.6          2.7           4.2          1.3  Versicolo
r
104            6.5          3.0           5.8          2.2   Virginic
a
```

In [ ]:

```
# extracting the class labels for development and testing data
test_class = list(test_set.iloc[:,-1])
dev_class = list(development_set.iloc[:,-1])

# calculating mean and standard deviation for both development set and
testing 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()
```

```
<ipython-input-30-f2c127012c5f>:6: FutureWarning: Dropping of nuisan
ce columns in DataFrame reductions (with 'numeric_only=None') is dep
recated; in a future version this will raise TypeError.  Select only
valid columns before calling the reduction.
  mean_development_set = development_set.mean()
<ipython-input-30-f2c127012c5f>:7: FutureWarning: Dropping of nuisan
ce columns in DataFrame reductions (with 'numeric_only=None') is dep
recated; in a future version this will raise TypeError.  Select only
valid columns before calling the reduction.
  mean_test_set = test_set.mean()
<ipython-input-30-f2c127012c5f>:8: FutureWarning: Dropping of nuisan
ce columns in DataFrame reductions (with 'numeric_only=None') is dep
recated; in a future version this will raise TypeError.  Select only
valid columns before calling the reduction.
  std_development_set = development_set.std()
<ipython-input-30-f2c127012c5f>:9: FutureWarning: Dropping of nuisan
ce columns in DataFrame reductions (with 'numeric_only=None') is dep
recated; in a future version this will raise TypeError.  Select only
valid columns before calling the reduction.
  std_test_set = test_set.std()
```

In [ ]:

```
# finding Euclidean Distance
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)

# Formula for Normalized Euclidean Distance
# d(p, q) = sqrt(sum(((pi - mu_i) / sigma_i - (qi - mu_i) / sigma_i) *
```

```
* 2))
# pi and qi are features of data_1 and data_2 and mu is mean and sigma
_i is standard deviation
def normalizedEuclideanDistance(data_1, data_2, data_len, data_mean, d
ata_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]) - ((data_2[i] - data_mean[i])/data_std[i])))
    return np.sqrt(n_dist)


def cosineSimilarity(data_1, data_2):
# computes the dot product of data_1 and data_2 without considering th
e last element of 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])

# It computes the cosine similarity between data_1 and data_2, dividin
g dot by the product of the two Euclidean norms.
    cos = dot / (norm_data_1 * norm_data_2)
    return (1-cos)


# This function calculates the distance between the test instance and
all instances
# Then it finds the k nearest neighbors and returns the class

# For K-nearest neighbours
def knn(dataset, testInstance, k, dist_method, dataset_mean, dataset_s
td):
    distances = {}

    # ???? why length is depending on test instance ????

    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, datase
t.iloc[x], length, dataset_mean, dataset_std)
            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-virginic
a" : 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 frequest clas
s
    sort_counts = sorted(counts.items(), key=operator.itemgetter(1), r
everse=True)
    return(sort_counts[0][0])
```

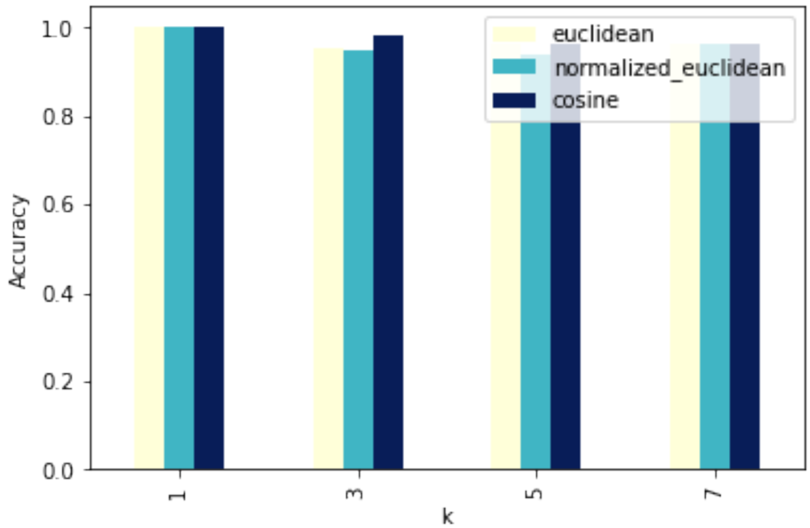
In [36]:
```
# Now we implememt KNN algorithm on development set for various values
of K
# It iterates through each data point in the development set
# to determine its predicted class label based on the k nearest neighb
ors

row_list = []
for index, rows in development_set.iterrows():
    my_list =[rows.sepal_length, rows.sepal_width, rows.petal_length,
rows.petal_width]
    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 develo
pment set data points and for each k and each distance metric
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[i]), k, dist_method, mean_development_set, std_development_s
et))
        development_set_obs_k[k] = development_set_obs
    # Nested Dictionary containing the observed class for each k and e
ach distance metric (obs_k of the form obs_k[dist_method][k])
    obs_k[dist_method] = development_set_obs_k
    print(dist_method.upper() + " distance method performed on the dat
aset for all k values!")
```

```
EUCLIDEAN distance method performed on the dataset for all k values!
NORMALIZED_EUCLIDEAN distance method performed on the dataset for al
l k values!
COSINE distance method performed on the dataset for all k values!
```

In [38]:
```
accuracy = {}
for key in obs_k.keys():
    accuracy[key] = {}
    for k_value in obs_k[key].keys():
        #print('k = ', key)
        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 data frame
df_res = pd.DataFrame({'k': k_n})
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="bar", colormap='YlGnBu')
draw.set(ylabel='Accuracy')

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

# Fetching the best k value for using all hyper-parameters
# In case the accuracy is the same for different k and different distance metric selecting the first of all the same
column_val = [c for c in df_res.columns if not c.startswith('k')]
col_max = df_res[column_val].max().idxmax()
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 metric is\033[1m', best_dist_method, '\033[0m. Ignoring k=1 if the value of accuracy for k=1 is 100%, since this mostly implies overfitting')
else:
    print('\n\n\nBest k value is\033[1m', best_k, '\033[0mand best distance metric is\033[1m', best_dist_method, '\033[0m.')
```

	k	euclidean	normalized_euclidean	cosine
0	1	1.000000	1.000000	1.000000
1	3	0.955357	0.946429	0.982143
2	5	0.964286	0.937500	0.964286
3	7	0.964286	0.964286	0.964286

Best k value is **3** and best distance metric is **cosine** . Ignoring k=1 if the value of accuracy for k=1 is 100%, since this mostly implies overfitting



In [39]:
```
print('\n\n\nBest k value is\033[1m', best_k, '\033[0mand best distance metric is\033[1m', best_dist_method, '\033[0m')
```

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

In [40]:
```
# Creating a list of list of all columns except 'class' by iterating t
hrough the development set
row_list_test = []
for index, rows in test_set.iterrows():
    my_list =[rows.sepal_length, rows.sepal_width, rows.petal_length,
rows.petal_width]
    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, best_dist_method, mean_test_set, std_test_set))
#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
```