In [1]:
```python
import matplotlib as plt
import pandas as pd
import numpy as np
from math import sqrt
```

Load and process Data.

In [2]:
```python
#read file from dataset directory
filename = 'Dataset/iris.data'
df = pd.read_csv(filename, header = None)

#create columns
df.columns = ["sepal length", "sepal width", "petal length", "petal width", "Class"]

# #Convert class names into values
df.Class.replace(('Iris-setosa','Iris-versicolor','Iris-virginica'),(1,2,3), inplace=True)
```

In [3]:
```python
#Dataset to be used for normal Euclidean distance and Cosine Similarity
df
```

Out[3]:

	sepal length	sepal width	petal length	petal width	Class
0	5.1	3.5	1.4	0.2	1
1	4.9	3.0	1.4	0.2	1
2	4.7	3.2	1.3	0.2	1
3	4.6	3.1	1.5	0.2	1
4	5.0	3.6	1.4	0.2	1
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	3
146	6.3	2.5	5.0	1.9	3
147	6.5	3.0	5.2	2.0	3
148	6.2	3.4	5.4	2.3	3
149	5.9	3.0	5.1	1.8	3

150 rows × 5 columns

In [4]:
```python
#data chosen for normalization
data = df.iloc[0:,0:4]
x = data.values.tolist()
```

In [5]:
```python
#Calculate MinMax
# df_list = df.values.tolist()
minmax = list()
for i in range(len(x[0])):
    col_values = [row[i] for row in x]
    value_min = min(col_values)
    value_max = max(col_values)
    minmax.append([value_min, value_max])

#Calculate Normalized dataset
normalizedList = list(x)
for row in normalizedList:
    for i in range(len(row)):
        row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])
```

In [6]:
```python
df_normalized = pd.DataFrame(normalizedList)
```

In [7]:
```python
df_normalized['Class'] = df['Class'].values
df_normalized.columns = ["sepal length", "sepal width", "petal length", "petal width", "Clas
s"]
```

In [8]:
```python
##Dataset to be used for Normalized Euclidean distance and Cosine Similarity
df_normalized
```

Out[8]:

	sepal length	sepal width	petal length	petal width	Class
0	0.222222	0.625000	0.067797	0.041667	1
1	0.166667	0.416667	0.067797	0.041667	1
2	0.111111	0.500000	0.050847	0.041667	1
3	0.083333	0.458333	0.084746	0.041667	1
4	0.194444	0.666667	0.067797	0.041667	1
...	...	...	...	...	...
145	0.666667	0.416667	0.711864	0.916667	3
146	0.555556	0.208333	0.677966	0.750000	3
147	0.611111	0.416667	0.711864	0.791667	3
148	0.527778	0.583333	0.745763	0.916667	3
149	0.444444	0.416667	0.694915	0.708333	3

150 rows × 5 columns

In [9]:
```python
from sklearn.model_selection import train_test_split
```

Divide the dataset as development and test. Because kNN does not require training you don’t have a train dataset. Make sure randomly divide the dataset.

In [10]:
```python
#Dividing the data for development and test data split
x = df.iloc[0:,0:4]
y = df.iloc[0:,-1]
x_normalized = df_normalized.iloc[0:4]
y_normalized = df_normalized.iloc[0:,-1]
```

In [11]:
```python
X_dev, X_test, y_dev, y_test = train_test_split(x,y,test_size = 0.4)
X_normalized_dev, X_normalized_test, y_normalized_dev, y_normalized_test = train_test_split(
x,y,test_size = 0.4)
```

In [12]:
```python
df_dev = X_dev
df_test = X_test
df_dev_normalized = X_normalized_dev
df_test_normalized = X_normalized_test
```

In [13]:
```python
df_dev['Class'] = y_dev
df_test['Class'] = y_test
df_dev_normalized['Class'] = y_normalized_dev
df_test_normalized['Class'] = y_normalized_test
```

```
<ipython-input-13-5cbd31fc1e64>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  df_dev['Class'] = y_dev
<ipython-input-13-5cbd31fc1e64>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  df_test['Class'] = y_test
<ipython-input-13-5cbd31fc1e64>:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  df_dev_normalized['Class'] = y_normalized_dev
<ipython-input-13-5cbd31fc1e64>:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy
  df_test_normalized['Class'] = y_normalized_test
```

In [14]:
```python
df_dev_list = df_dev.values.tolist()
df_dev_normalized_list = df_dev_normalized.values.tolist()
df_normalized_list = df_normalized.values.tolist()
df_test_list = df_test.values.tolist()
df_test_normalized_list = df_test_normalized.values.tolist()
```

Implement kNN using the following hyperparameters

number of neighbor K -> 1,3,5,7

In [15]:
```python
k_neighbors = [1,3,5,7]
```

Distance metrics:

In [16]:
```python
from sklearn.metrics.pairwise import cosine_similarity
```

In [17]:
```python
class KNN(object):
    def __init__(self,k):
        self.k = k

#     #Cosine similarity for datasets
#     def cos_sim(self, row1, row2):
#         for i in range(len(row)-1):
#             dot_product = np.dot(row1[i], row2[i])
#             norm_a = np.linalg.norm(row1[i])
#             norm_b = np.linalg.norm(row1[i])
#             a = dot_product / (norm_a * norm_b)
# #             print(a)
#         return (1-a)

    #Cosine similarity for datasets
    def cos_sim(self,row1, row2):
        sum = 0
        sum1 = 0
        sum2 = 0
        for i in range(len(row)-1):
#             print(i)
            for i,j in zip(row1,row2):
#                 print(j)
                sum1 += i*i
                sum2 += j*j
                sum += i*j
            cos = sum/((sqrt(sum1))*(sqrt(sum2)))
        return (1-cos)


    #Calculate Euclidean Distance between 2 vectors/rows.
    def calculateEuclideanDistance(self,row1, row2):
        for i in range(len(row1)-1):
            sum_sq = np.sum(np.square(row1[i] - row2[i]))
            a = (np.sqrt(sum_sq))
        return a

    #Calculate the nearest neighbors.
    def calculateNearestNeighborsUsingCos(self,train, test_row, num_neighbors):
        distances = list()
        for train_row in train:
            dist = self.cos_sim(train_row, test_row)
            distances.append((train_row,dist))
        distances.sort(key = lambda tup : tup[1])
        neighbors = list()
        for i in range(num_neighbors):
            neighbors.append(distances[i][0])
        return neighbors

    #Calculate the nearest neighbors.
    def calculateNearestNeighborsUsingEuclidean(self,train, test_row, num_neighbors):
        distances = list()
        for train_row in train:
            dist = self.calculateEuclideanDistance(train_row, test_row)
            distances.append((train_row, dist))
        distances.sort(key = lambda tup : tup[1])
        neighbors = list()
        for i in range(num_neighbors):
            neighbors.append(distances[i][0])
        return neighbors

    #make predictions for euclidean.
    def makePredictionsforeuclidean(self,train, test_row, num_neighbors):
        neighbors = self.calculateNearestNeighborsUsingEuclidean(train, test_row, num_neighb
ors)
        outputValues = [row[-1] for row in neighbors]
        prediction = max(set(outputValues), key = outputValues.count )
        return prediction

    #make predictions for cosine distances.
    def makePredictionsforcosine(self,train, test_row, num_neighbors):
        neighbors = self.calculateNearestNeighborsUsingCos(train, test_row, num_neighbors)
        outputValues = [row[-1] for row in neighbors]
        prediction = max(set(outputValues), key = outputValues.count )
        return prediction

    #KNN Algorithm for euclidean
    def kNN_Algorithmforeuclidean(self,train, test, num_neighbors):
        predictions = list()
        for row in test:
            output = self.makePredictionsforeuclidean(train, row, num_neighbors)
            predictions.append(output)
        return predictions

    #KNN Algorithm for cosine
    def kNN_Algorithmforcosine(self,train, test, num_neighbors):
        predictions = list()
        for row in test:
            output = self.makePredictionsforcosine(train, row, num_neighbors)
            predictions.append(output)
        return predictions

    #evaluate algorithm using a cross validation split to check score
    def evaluateAlgorithmforeuclidean(self,dataset,*args):
        score = list()
        testSet = list()
        for row in dataset:
            rowCopy = list(row)
            testSet.append(rowCopy)
            rowCopy[-1] = None
        actual = [row[-1] for row in dataset]
        predicted = self.kNN_Algorithmforeuclidean(dataset, testSet, *args)
        accuracy = self.accuracyMetric(actual, predicted)
        score.append(accuracy)
        return score

    #evaluate algorithm using a cross validation split to check score
    def evaluateAlgorithmforcosine(self,dataset,*args):
        score = list()
        testSet = list()
        for row in dataset:
            rowCopy = list(row)
            testSet.append(rowCopy)
#             rowCopy[-1] = None
        actual = [row[-1] for row in dataset]
        predicted = self.kNN_Algorithmforcosine(dataset, testSet, *args)
        accuracy = self.accuracyMetric(actual, predicted)
        score.append(accuracy)
        return score

    #calculate accuracy percentage
    def accuracyMetric(self,actual, predicted):
        correct = 0.0
        for i in range (len(actual)):
            if actual[i] == predicted[i]:
                correct += 1
        percentage = correct / float(len(actual)) *100.0
        return percentage
```

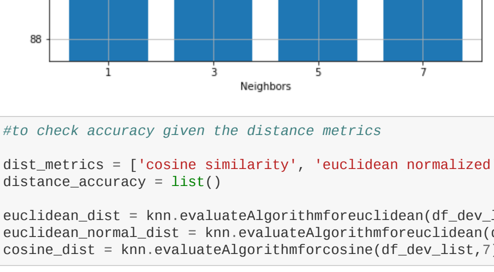
In [18]:
```python
knn = KNN(5)
```

In [19]:
```python
import matplotlib.pyplot as plt

scores = list()
for i in k_neighbors:
    scores.append(knn.evaluateAlgorithmforeuclidean(df_dev_list, i))

score = [ item for elem in scores for item in elem]
```

Calculate accuracy by iterating all of the development data point

Find optimal hyperparameters

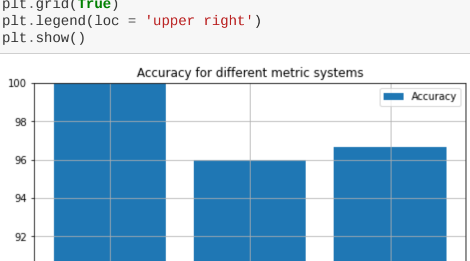
In [20]:
```python
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(k_neighbors, score, width =1.5, label = 'Accuracy')
ax.set_ylabel('Accuracy')
ax.set_xlabel('Neighbors')
ax.set_title('Accuracy for K = 1,3,5,7')
ax.set_xticks(k_neighbors)
plt.ylim((87,100))
plt.grid(True)
plt.legend(loc = 'upper left')
plt.show()
```



In [21]:
```python
#to check accuracy given the distance metrics

dist_metrics = ['cosine similarity', 'euclidean normalized', 'euclidean']
distance_accuracy = list()

euclidean_dist = knn.evaluateAlgorithmforeuclidean(df_dev_list,7)
euclidean_normal_dist = knn.evaluateAlgorithmforeuclidean(df_normalized_list,7)
cosine_dist = knn.evaluateAlgorithmforcosine(df_dev_list,7)
```

In [22]:
```python
distance_accuracy.append(cosine_dist)
distance_accuracy.append(euclidean_normal_dist)
distance_accuracy.append(euclidean_dist)

score_dist = [ item for elem in distance_accuracy for item in elem]
```

In [23]:
```python
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(dist_metrics, score_dist, label = 'Accuracy')
ax.set_title('Accuracy for different metric systems')
plt.ylim((85,100))
plt.grid(True)
plt.legend(loc = 'upper right')
plt.show()
```



From the above inferences. I can conclude that Cosine similarity with a k value 7 is the optimal hyperparameter for this iteration. Notice that for each iteration of the whole model, the optimal parameter changes

Use the optimal hyperparameters you found in the step c, and use it to calculate the final accuracy

In [24]:
```python
test_score = knn.evaluateAlgorithmforcosine(df_test_list,7)
```

In [25]:
```python
print('Scores: %s' %test_score)
```

```
Scores: [98.33333333333333]
```