In [2]:
```python
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

Lab 2

Dominic Sagers - i6255473

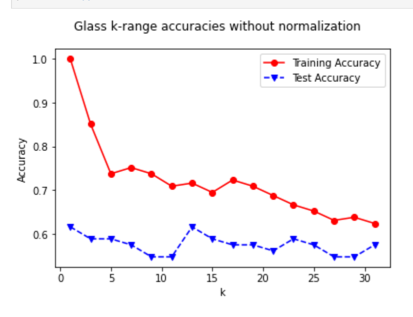
In [34]:
```python
# Class of k-Nearest Neigbor Classifier


class kNN():
    def __init__(self, k = 3, exp = 2):
    # constructor for kNN classifier
    # k is the number of neighbor for local class estimation
    # exp is the exponent for the Minkowski distance
        self.k = k
        self.exp = exp

    def fit(self, X_train, Y_train):
    # training k-NN method
    # X_train is the training data given with input attributes. n-th row correponds to n-th instance.
    # Y_train is the output data (output vector): n-th element of Y_train is the output value for n-th instance in X_train.
        self.X_train = X_train
        self.Y_train = Y_train

    def getDiscreteClassification(self, X_test):
    # predict-class k-NN method
    # X_test is the test data given with input attributes. Rows correpond to instances
    # Method outputs prediction vector Y_pred_test:  n-th element of Y_pred_test is the prediction for n-th instance in X_test

        Y_pred_test = [] #prediction vector Y_pred_test for all the test instances in X_test is initialized to empty list []


        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances = []  #list of distances of the i-th test_instance for all the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances in X_train
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance, train_instance) #distance between i-th test instance and j-th training instance
                distances.append(distance) #add the distance to the list of distances of the i-th test_instance

            # Store distances in a dataframe. The dataframe has the index of Y_train in order to keep the correspondence with the classes of the training instances
            df_dists = pd.DataFrame(data=distances, columns=['dist'], index = self.Y_train.index)

            # Sort distances, and only consider the k closest points in the new dataframe df_knn
            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]

            # Note that the index df_knn.index of df_knn contains indices in Y_train of the k-closed training instances to
            # the i-th test instance. Thus, the dataframe self.Y_train[df_knn.index] contains the classes of those k-closed
            # training instances. Method value_counts() computes the counts (number of occurencies) for each class in
            # self.Y_train[df_knn.index] in dataframe predictions.
            predictions = self.Y_train[df_knn.index].value_counts()

            # the first element of the index predictions.index contains the class with the highest count; i.e. the prediction y_pred_test.
            y_pred_test = predictions.index[0]

            # add the prediction y_pred_test to the prediction vector Y_pred_test for all the test instances in X_test
            Y_pred_test.append(y_pred_test)

        return Y_pred_test

    def getPrediction(self, X_test):
      #This method reuses the code from getDiscreteClassification and instead of predicting a classification, returns a dataframe containing the regression values for each instance of X_test
      #instance in the X_test data's corresponding output from the Y/Class output list

        Y_regression = pd.DataFrame(columns=['regression_values']) #prediction vector Y_pred_test for all the test instances in X_test is initialized to empty list []


        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances =  [] #list of distances of the i-th test_instance for all the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance, train_instance)
                distances.append(distance)

            df_dists = pd.DataFrame(data=distances, columns=['dist'], index = self.Y_train.index)

            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]


            predicted_values = self.Y_train[df_knn.index]
            #var = {'regression_values' : predicted_values.sum()/self.k}
            #Y_regression = Y_regression.append(var, ignore_index=True)
            Y_regression = Y_regression.append({'regression_values' : predicted_values.sum()/self.k}, ignore_index=True)

        return Y_regression

    def getClassProbs(self, X_test):

      #This method reuses the code from getDiscreteClassification and instead of predicting a classification, returns a dataframe containing the posterior probabilities of each
      #instance in the X_test data's corresponding output from the Y/Class output list

        Y_pred_test = pd.DataFrame(columns = self.Y_train.unique()) #prediction vector Y_pred_test for all the test instances in X_test is initialized to empty list []


        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances =  [] #list of distances of the i-th test_instance for all the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances in X_train
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance, train_instance) #distance between i-th test instance and j-th training instance
                distances.append(distance) #add the distance to the list of distances of the i-th test_instance

            # Store distances in a dataframe. The dataframe has the index of Y_train in order to keep the correspondence with the classes of the training instances
            df_dists = pd.DataFrame(data=distances, columns=['dist'], index = self.Y_train.index)

            # Sort distances, and only consider the k closest points in the new dataframe df_knn
            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]

            # Note that the index df_knn.index of df_knn contains indices in Y_train of the k-closed training instances to
            # the i-th test instance. Thus, the dataframe self.Y_train[df_knn.index] contains the classes of those k-closed
            # training instances. Method value_counts() computes the counts (number of occurencies) for each class in
            # self.Y_train[df_knn.index] in dataframe predictions.
            class_counts= self.Y_train[df_knn.index].value_counts()

            class_values = self.Y_train.unique()
            posterior_probabilities = {}#Here we create a dictionary to append all of our probabilities to this instance as that was the most convenient way i could find to do it.

            for class_n in class_values:
              if class_n in class_counts.index:
                posterior_probabilities[class_n] = class_counts[class_n]/class_counts.sum()
              else:
                posterior_probabilities[class_n] = 0

            Y_pred_test = Y_pred_test.append(posterior_probabilities, ignore_index=True)

        return Y_pred_test


    def Minkowski_distance(self, x1, x2):
    # computes the Minkowski distance of x1 and x2 for two labeled instances (x1,y1) and (x2,y2)

        # Set initial distance to 0
        distance = 0

        # Calculate Minkowski distance using the exponent exp
        for i in range(len(x1)):
            distance = distance + abs(x1[i] - x2[i])**self.exp

        distance = distance**(1/self.exp)

        return distance

    def normalize(self, X_test): #My normalize method normalizes the training set and then uses the same min and max from the training set to normalize the test set, as that after days is
                                 #of searching I have found is the most correct method (TA approved).
      min = self.X_train.min()
      max = self.X_train.max()

      self.X_train = (self.X_train-min)/(max-min)
      X_test = (X_test-min)/(max-min)
      return self.X_train, X_test
```

Add to class kNN method normalize that normalizes the input attributes of the training data X_train and test data X_test. We note that attribute normalization is important since all the attributes receive equal weights when instance distances are being computed.

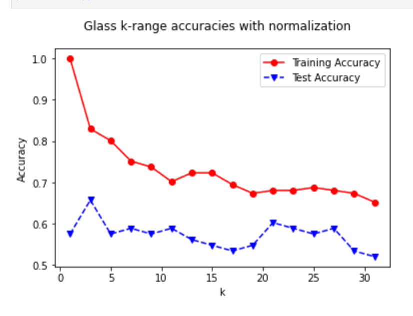
Answer: After figuring out how python uses objects and changes lists (took about 3 hours) I finally was able to normalize the data upon the formula x - x_MIN)/(x_MAX - x_MIN) which produced in some areas relatively more accurate test set.

In [17]:
```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from numpy.random import random
from sklearn.metrics import accuracy_score

##################################################
# Hold-out testing: Training and Test set creation
##################################################

data = pd.read_csv('/content/drive/MyDrive/ML Lab Files/Lab2/glass.csv')
data_diabetes = pd.read_csv('/content/drive/MyDrive/ML Lab Files/Lab2/diabetes.csv')



data.head()
Y = data['class']
X = data.drop(['class'],axis=1)

Yg = data_diabetes['class']
Xg = data_diabetes.drop(['class'], axis=1)


X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34, random_state=10)

Xg_train, Xg_test, Yg_train, Yg_test = train_test_split(Xg, Yg, test_size=0.34, random_state=10)

# range for the values of parameter k for kNN

k_range = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]

trainAcc = np.zeros(len(k_range))
testAcc = np.zeros(len(k_range))

trainAcc_diabetes = np.zeros(len(k_range))
testAcc_diabetes = np.zeros(len(k_range))
```

In [ ]:
```python
#Glass without normalization
index = 0
for k  in  k_range:
    clf = kNN(k)
    clf.fit(X_train, Y_train)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1


plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--')
plt.suptitle('Glass k-range accuracies without normalization')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.show()
```

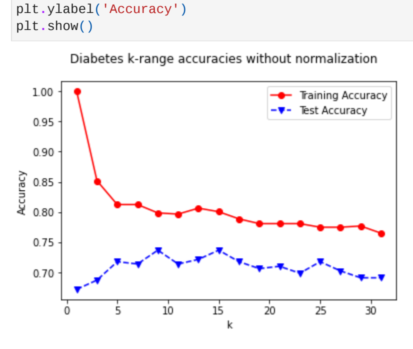


In [18]:
```python
#Glass with normalization
index = 0
for k  in  k_range:
    clf = kNN(k)
    clf.fit(X_train, Y_train)
    X_train, X_test = clf.normalize(X_test)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1

plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--')
plt.suptitle('Glass k-range accuracies with normalization')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.show()
```

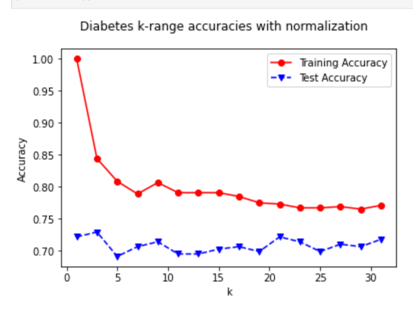


For the glass dataset: We observe above that through normalization the graph will have more consistent data and consistently higher accuracy rather than without normalization.

In [ ]:
```python
#Diabetes without normalization
index = 0
for k  in  k_range:
    clf = kNN(k)
    clf.fit(Xg_train, Yg_train)
    Yg_predTrain = clf.getDiscreteClassification(Xg_train)
    Yg_predTest = clf.getDiscreteClassification(Xg_test)
    trainAcc_diabetes[index] = accuracy_score(Yg_train, Yg_predTrain)
    testAcc_diabetes[index] = accuracy_score(Yg_test, Yg_predTest)
    index += 1


plt.plot(k_range,trainAcc_diabetes,'ro-',k_range,testAcc_diabetes,'bv--')
plt.suptitle('Diabetes k-range accuracies without normalization')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.show()
```



In [19]:
```python
#Diabetes with normalization
Xg_train, Xg_test, Yg_train, Yg_test = train_test_split(Xg, Yg, test_size=0.34, random_state=10)


index = 0
for k  in  k_range:
    clf = kNN(k)
    clf.fit(Xg_train, Yg_train)
    Xg_train, Xg_test = clf.normalize(Xg_test)
    Yg_predTrain = clf.getDiscreteClassification(Xg_train)
    Yg_predTest = clf.getDiscreteClassification(Xg_test)
    trainAcc_diabetes[index] = accuracy_score(Yg_train, Yg_predTrain)
    testAcc_diabetes[index] = accuracy_score(Yg_test, Yg_predTest)
    index += 1

plt.plot(k_range,trainAcc_diabetes,'ro-',k_range,testAcc_diabetes,'bv--')
plt.suptitle('Diabetes k-range accuracies with normalization')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('k')
plt.ylabel('Accuracy')
plt.show()
```

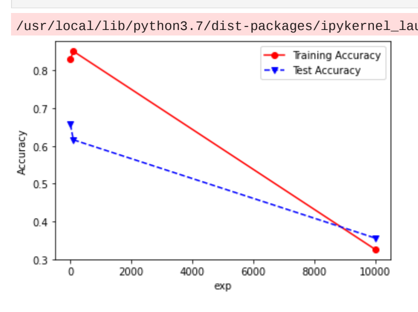


Interestingly enough in the case of the diabetes dataset, accuracy becomes more consistent but does not necessarly increase as much as seen in the glass dataset.

Test the kNN classifier on the glass classification data sets the data is normalized for different values of the exp parameter of the Minkowski distance. Indicate whether the training and hold-out accuracy rates changes due to exp. For this task you might use the second testing script provided in the Jupiter note.

In [23]:
```python
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import accuracy_score
from numpy.random import random

##################################################
# Hold-out testing: Training and Test set creation
##################################################

data = pd.read_csv('/content/drive/MyDrive/ML Lab Files/Lab2/glass.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)


X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34, random_state=10)


# range for the values of parameter exp for kNN

exp_range = [2,  100, 10000]

trainAcc = np.zeros(len(exp_range))
testAcc = np.zeros(len(exp_range))


index = 0
for exp  in  exp_range:
    clf = kNN(k = 3, exp = exp)
    clf.fit(X_train, Y_train)
    X_train, X_test = clf.normalize(X_test)
    Y_predTrain = clf.getDiscreteClassification(X_train)
    Y_predTest = clf.getDiscreteClassification(X_test)
    trainAcc[index] = accuracy_score(Y_train, Y_predTrain)
    testAcc[index] = accuracy_score(Y_test, Y_predTest)
    index += 1


#########################################
# Plot of training and test accuracies
#########################################

plt.plot(exp_range,trainAcc,'ro-',exp_range,testAcc,'bv--')
plt.legend(['Training Accuracy','Test Accuracy'])
plt.xlabel('exp')
plt.ylabel('Accuracy')
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:140: RuntimeWarning: overflow encountered in double_scalars
```



The graph plots a few points of which rapidly decreases around exp=100 in accuracy as the exp climbs to 10000. This may occur as when exp reaches an upper bound (say infinity), the SUP function of the Minkowski distance causes the its return to use the supremum of all combined distances which creates a decline in overall accuracy.

Add to class kNN method getClassProbs that computes for all the test instances in X_test the posterior class probabilities. This means that the method computes for each row (instance) in X_test a row with probability of class 1, probability of class 2, and probability of class N. Combine the rows of the posterior class probabilities in pandas.DataFrame object that will be the output of the method getClassProbs:

Answer:

In [ ]:
```python
Y = data['class']
X = data.drop(['class'],axis=1)

Yg = data_diabetes['class']
Xg = data_diabetes.drop(['class'], axis=1)



X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34, random_state=10)

test_glass = kNN(5)

test_glass.fit(X_train,Y_train)
class_probs = test_glass.getClassProbs(X_test)
display(class_probs)



Xg_train, Xg_test, Yg_train, Yg_test = train_test_split(Xg, Yg, test_size=0.34, random_state=10)

test_diabetes = kNN(5)
test_diabetes.fit(Xg_train, Yg_train)
class_probs_diabetes = test_diabetes.getClassProbs(Xg_test)
display(class_probs_diabetes)
```

	'build wind float'	'build wind non-float'	headlamps	'vehic wind float'	containers	tableware
0	0.8	0.0	0.0	0.2	0.0	0.0
1	1.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.8	0.0	0.2	0.0
3	0.2	0.8	0.0	0.0	0.0	0.0
4	0.8	0.2	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...
68	0.6	0.0	0.0	0.4	0.0	0.0
69	0.2	0.2	0.2	0.4	0.0	0.0
70	0.0	0.2	0.6	0.0	0.0	0.2
71	0.0	1.0	0.0	0.0	0.0	0.0
72	0.4	0.6	0.0	0.0	0.0	0.0

73 rows × 6 columns

	tested_positive	tested_negative
0	0.8	0.2
1	0.4	0.6
2	0.6	0.4
3	0.0	1.0
4	0.4	0.6
...	...	...
257	0.0	1.0
258	0.8	0.2
259	1.0	0.0
260	0.0	1.0
261	0.2	0.8

262 rows × 2 columns

As seen in the above dataframe diplays, my method calculates for each instance the probabilities of the surrounding k-neighbor classes.

Test the method getPrediction on the autoprice data set which is a regression data set (see Appendix A). For that purpose you can adapt the test script tha you have already used for Task B. Please use mean absolute error as the main metric for estimating regression performance1 instead of the accuracy rate. To compute the mean absolute error you can use method mean_absolute_error from sklearn.metrics.

Answer:

In [26]:
```python
from sklearn.metrics import mean_absolute_error
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
mean_absolute_error(y_true, y_pred)
```

Out[26]: 0.5

In [36]:
```python
data = pd.read_csv('/content/drive/MyDrive/ML Lab Files/Lab2/autoprice.csv')
data.head()
Y = data['class']
X = data.drop(['class'],axis=1)
data.head()
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34, random_state=10)

k_range = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]

trainAcc = np.zeros(len(k_range))
testAcc = np.zeros(len(k_range))

index = 0
for k  in  k_range:
    clf = kNN(k)
    clf.fit(X_train, Y_train)
    X_train, X_test = clf.normalize(X_test)
    Y_predTrain = clf.getPrediction(X_train)
    Y_predTest = clf.getPrediction(X_test)
    trainAcc[index] = mean_absolute_error(Y_train, Y_predTrain)
    testAcc[index] = mean_absolute_error(Y_test, Y_predTest)
    index += 1



plt.plot(k_range,trainAcc,'ro-',k_range,testAcc,'bv--')
plt.suptitle('Autoprice Dataset Mean Absolute Errors')
plt.legend(['Training Mean Absolute Error','Test Mean Absolute Error'])
plt.xlabel('k')
plt.ylabel('Mean Absolute Error')
plt.show()
```