Make The Training Arrays and Give Them Corresponding Y-Cap Values

In [1]:
```python
import numpy as np
from sklearn.neighbors import KNeighborsRegressor
from matplotlib import pyplot as plt

#Required, Pre-defined Global Vars to work with
rng = np.random.default_rng(seed=22)
TOTAL_SAMPLE_SIZE = 100
high = 10
n = 7
d = 1
K = 3
m = 5
s = 2

#Matplot lib is bugged with the version I am using
#the remove function for lines and collections will not'
#do what it's supposed to and remove plot lines, it instead
#clears the entire plot no matter what. This function is to
#rebuild all the stupid settings for the plot it was not meant
#to so carelessly delete.
def setupPlot():
    #getting plot information
    fig, ax = plt.subplots()

    #clear old plot and rebuild
    fig.clear()

    #resize
    plt.figure().set_figwidth(6)
    plt.figure().set_figheight(5)

    # Labeling the axes
    plt.xlabel("X")
    plt.ylabel("y")

    #set x&y-axis for better viewing
    listOf_Yticks = np.arange(-1.5, 2.5, .25)
    plt.yticks(listOf_Yticks)
    listOf_Xticks = np.arange(0, 11, 1)
    plt.xticks(listOf_Xticks)

#print_dataframe():
#prints the dataframe from the top
#and bottom
def print_dataframe(frame):
    #get size of frame
    framesize = frame.size

    #print formatted data
    for i in range(0, 5):
        print(str(frame[i]))
    for i in range(0, frame[i].size):
        print("...")
    for i in range(framesize-5, framesize):
        print(str(frame[i]))
    return

#generate the possible vectors via the given globals
def initializeDataStructure():
    return(rng.random((TOTAL_SAMPLE_SIZE,1)) * high)

#generate the Y-Cap corresponding results
def initializeDataYCap(x_train):

    #Generate a gausian noise and add to natural log of each element
    #Mean of 0 and standard deviation of +-.1
    #result is Y cap
    calcY = lambda t: np.log(t) + (rng.random((1,1)) / 10)[0]
    return(np.array(list(map(calcY, x_train))))

#make the actual array to use
def makeDataSet():
    x_train = initializeDataStructure()
    y_train = initializeDataYCap(x_train)
    return(x_train, y_train)

#make dataset
X_train, y_train = makeDataSet()
print("Training Dataset (X):")
print_dataframe(X_train)
print("\nTraining Y-caps:")
print_dataframe(y_train)
```

```
Training Dataset (X):
[3.66346915]
[1.99295379]
[0.88558373]
[6.53191688]
[4.59337045]
...
[1.49818758]
[7.61340915]
[5.98017057]
[3.29598789]
[8.49757297]

Training Y-caps:
[1.30266095]
[0.71397191]
[-0.0714847]
[1.91672393]
[1.56308961]
...
[0.4246537]
[2.10229177]
[1.87576487]
[1.1964912]
[2.23525261]
```

Make The Testing Dataset

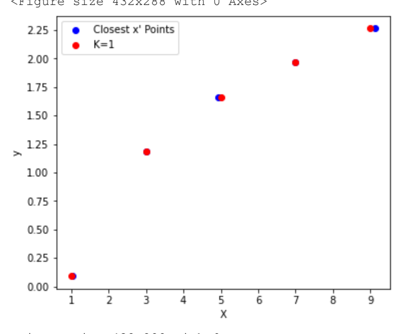
In [2]:
```python
#make the test data-set
def makeTestDataSet():
    return(np.array([1, 3, 5, 7, 9]), np.array([0, 0, 0, 0, 0]))

#make test points
X_test, y_test = makeTestDataSet()
print("Test Datapoints:")
print(X_test)
print("")

#get the closest data points to our test points:
x_closest = [0.0,0.0]
y_closest = [0.0,0.0]

#find closest points
currentIndex = 0
for x in X_test:

    #get index of closest match
    idx = (np.abs(X_train - x)).argmin()

    #push to array of closest values & update index
    x_closest.append(X_train[idx])
    y_closest.append(y_train[idx])
    currentIndex = currentIndex + 1

#remove garbage placeholder values
x_closest = x_closest[2:]
y_closest = y_closest[2:]
```

```
Test Datapoints:
[1 3 5 7 9]
```

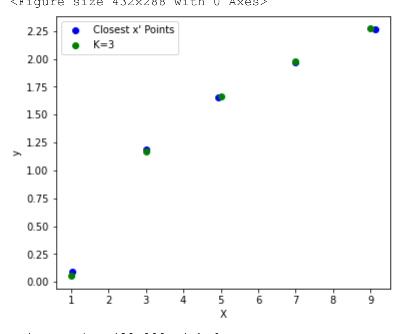
K neighbors contribute equally (1, 3, 50)

In [3]:
```python
def calcDistances(TrainingSet_X, TrainingSet_Y, inputs, neighbors):

    #nparray of values to return
    finalValues = np.array(0)

    #make determinations
    for i in range(0, inputs.size):
        for j in range(0, neighbors.size):
            neigh = KNeighborsRegressor(n_neighbors=neighbors[j])
            neigh.fit(TrainingSet_X, TrainingSet_Y)
            KNeighborsRegressor(...)
            finalValues = np.append(finalValues, neigh.predict([[inputs[i]]]))

    #return all values
    return(np.resize(np.delete(finalValues, 0), (5,3)))

#test first euclidiandistance
print("Results when all neighbors have equal weight:")
neighborPoints = np.array([1, 3, 50])
returnedValues = calcDistances(X_train, y_train, X_test, neighborPoints)
print("K = 1 : " + str(tuple(zip(X_test, returnedValues[:,0]))))
print("K = 3 : " + str(tuple(zip(X_test, returnedValues[:,1]))))
print("K = 50 : " + str(tuple(zip(X_test, returnedValues[:,2]))))

#Plot the graph for the first K value
setupPlot()
plt.scatter(x_closest,y_closest, label='Closest x\' Points',color='b')
plt.scatter(X_test,returnedValues[:,0], label='K=1',color='r')

# function to show the plot
plt.legend()
plt.show()

#Plot the graph for the second K value
setupPlot()
plt.scatter(x_closest,y_closest, label='Closest x\' Points',color='b')
plt.scatter(X_test,returnedValues[:,1], label='K=3',color='g')

# function to show the plot
plt.legend()
plt.show()

#Plot the graph for the third K value
setupPlot()
plt.scatter(x_closest,y_closest, label='Closest x\' Points',color='b')
plt.scatter(X_test,returnedValues[:,2], label='K=50',color='y')

# function to show the plot
plt.legend()
plt.show()
```

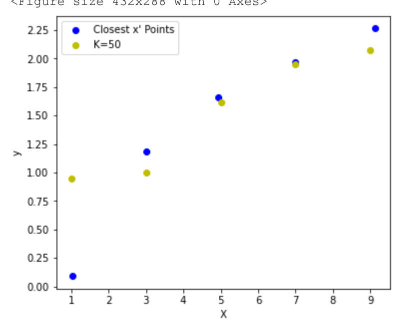
```
Results when all neighbors have equal weight:
K = 1 : ((1, 0.08870372315944877), (3, 1.188042230085945), (5, 1.6570342183115738), (7, 1.9669742276398698), (9, 2.2647329986460223))
K = 3 : ((1, 0.05222124920942698), (3, 1.169516485872208), (5, 1.6591939000407374), (7, 1.9804511371025058), (9, 2.2708714815282414))
K = 50 : ((1, 0.9433687322026917), (3, 1.0040555930754969), (5, 1.6164132965802025), (7, 1.9538006553855862), (9, 2.0698855324236494))
<Figure size 432x288 with 0 Axes>
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
<Figure size 432x288 with 0 Axes>
```



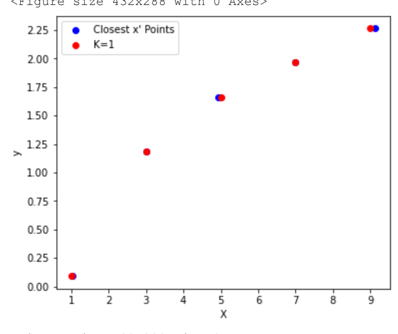

```
<Figure size 432x288 with 0 Axes>
<Figure size 432x288 with 0 Axes>
```



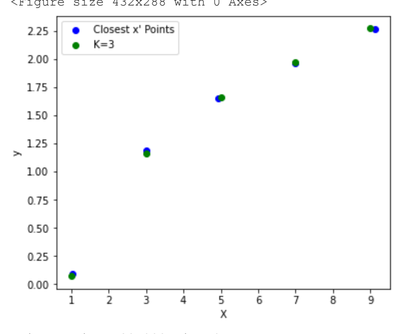
K neighbors contribute inversely by distance (1, 3, 50)

In [4]:
```python
def calcDistances(TrainingSet_X, TrainingSet_Y, inputs, neighbors):

    #nparray of values to return
    finalValues = np.array(0)

    #make determinations
    for i in range(0, inputs.size):
        for j in range(0, neighbors.size):
            neigh = KNeighborsRegressor(n_neighbors=neighbors[j], weights='distance')
            neigh.fit(TrainingSet_X, TrainingSet_Y)
            KNeighborsRegressor(...)
            finalValues = np.append(finalValues, neigh.predict([[inputs[i]]]))

    #return all values
    return(np.resize(np.delete(finalValues, 0), (5,3)))

#test first euclidiandistance
print("Results when neighbors' weight is inversely proportional to their distance:")
neighborPoints = np.array([1, 3, 50])
returnedValues = calcDistances(X_train, y_train, X_test, neighborPoints)
print("K = 1 : " + str(tuple(zip(X_test, returnedValues[:,0]))))
print("K = 3 : " + str(tuple(zip(X_test, returnedValues[:,1]))))
print("K = 50 : " + str(tuple(zip(X_test, returnedValues[:,2]))))

#Plot the graph for the first K value
setupPlot()
plt.scatter(x_closest,y_closest, label='Closest x\' Points',color='b')
plt.scatter(X_test,returnedValues[:,0], label='K=1',color='r')

# function to show the plot
plt.legend()
plt.show()

#Plot the graph for the second K value
setupPlot()
plt.scatter(x_closest,y_closest, label='Closest x\' Points',color='b')
plt.scatter(X_test,returnedValues[:,1], label='K=3',color='g')

# function to show the plot
plt.legend()
plt.show()

#Plot the graph for the third K value
setupPlot()
plt.scatter(x_closest,y_closest, label='Closest x\' Points',color='b')
plt.scatter(X_test,returnedValues[:,2], label='K=50',color='y')

# function to show the plot
plt.legend()
plt.show()
```

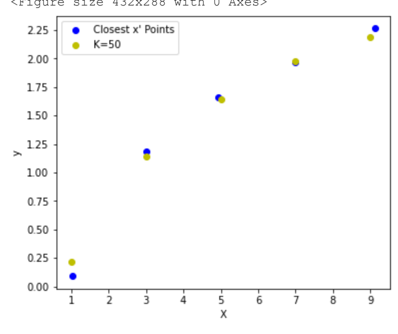
```
Results when neighbors' weight is inversely proportional to their distance:
K = 1 : ((1, 0.08870372315944877), (3, 1.188042230085945), (5, 1.6570342183115736), (7, 1.9669742276398698), (9, 2.2647329986460223))
K = 3 : ((1, 0.07129757778815825), (3, 1.1596333908860648), (5, 1.658259784386586), (7, 1.9723031326905218), (9, 2.274049764593811))
K = 50 : ((1, 0.2163176575411153), (3, 1.140822199613423), (5, 1.6430137656382582), (7, 1.9732190691315237), (9, 2.188547350667337))
<Figure size 432x288 with 0 Axes>
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
<Figure size 432x288 with 0 Axes>
```



K neighbors all points contribute, with each contribution proportional to e^(−(1/2)*d^2), where 'd' is distance

In [5]:
```python
def calcDistances(TrainingSet_X, TrainingSet_Y, inputs, neighbors):

    #nparray of values to return
    finalValues = np.array(0)

    #using 'e' as the weight formula
    eWeight = lambda d: np.exp((-1/2)*(d*d))

    #make determinations
    for i in range(0, inputs.size):
        neigh = KNeighborsRegressor(n_neighbors=neighbors, weights = eWeight)
        neigh.fit(TrainingSet_X, TrainingSet_Y)
        KNeighborsRegressor(...)
        finalValues = np.append(finalValues, neigh.predict([[inputs[i]]]))

    #return all values
    return(np.delete(finalValues, 0))

#test first euclidiandistance
print("Results when all neighbors' weight is accounted for:")
neighborPoints = 100
returnedValues = calcDistances(X_train, y_train, X_test, neighborPoints)
print("K = N (100) : " + str(tuple(zip(X_test, returnedValues))))

#resize
plt.figure().set_figwidth(6)
plt.figure().set_figheight(5)

#Plot the graph
plt.scatter(x_closest,y_closest, label='Closest x\' Points',color='b')
plt.scatter([1,3,5,7,9],returnedValues, label='K=N',color='r')

# Labeling the axes
plt.xlabel("X")
plt.ylabel("y")

#set x&y-axis for better viewing
listOf_Yticks = np.arange(-1.5, 2.5, .25)
plt.yticks(listOf_Yticks)
listOf_Xticks = np.arange(0, 11, 1)
plt.xticks(listOf_Xticks)

# function to show the plot
plt.legend()
plt.show()
```

```
Results when all neighbors' weight is accounted for:
K = N (100) : ((1, 0.20885371159846228), (3, 1.108475256510394), (5, 1.643742971856342
5), (7, 1.978165064387419), (9, 2.206261493656699))
<Figure size 432x288 with 0 Axes>
```