In [157]:

```
import pandas as pd
import numpy as np
import math


VALIDATION_BLOCK_SIZE = 200
kVals = [1, 5, 10, 50, 100]
df_train = pd.read_csv("train.csv", nrows = 1000)
trainingData = df_train.drop("label", axis = 1).values
trainingResults = df_train["label"].values
df_test = pd.read_csv("test.csv", nrows = 500)


# @param: point1, point2 - arrays of pixel data for two points
# @return: euclidean distance between the two points
# Note: features are unweighted
def getDistance(point1, point2):
    assert(len(point1) == len(point2))
    distance = math.sqrt(
        np.sum(np.square([point1[i] - point2[i] for i in range(len(point1))])))
    return distance

# @param: point - array of pixel data from MNIST dataset relating to query point
# @param: data - the dataset to finding the k closest neighbors from
# @param: k - number of numbers to find
# @return: an array holding the k nearest neighbors of the query point, an array
def getKNN(point, data, k):
    neighbors = []  # array holding indexes of the k closest neighbors
    distances = [] # array holding distances of respective neighbors
    for i in range(len(data)):
        neighbors.append(i)
        distances.append(getDistance(point, data[i]))
        # Limit the number of neighbors to just k
        if len(neighbors) > k:
            # Invariant: neighbors, distances size is k + 1
            sortedZip = sorted(zip(distances, neighbors))
            neighbors = [neighbor for (distance, neighbor) in sortedZip]
            distances = [distance for (distance, neighbor) in sortedZip]
            neighbors.pop()
            distances.pop()
            #Invariant: neighbors, distances size is k
    return neighbors, distances

# @param: neighbors - array of indices of points
# @param: results - array of output classifications
# @param: distances - array of distances of each respective point in neighbors fr
# @return: a prediction of classification based off the classification with the l
def makePrediction(neighbors, results, distances):
    predictionMap = {}
    for i in range(len(neighbors)):
        neighbor = neighbors[i]
        distance = distances[i]
        neighborPrediction = results[neighbor]
        # Create a dictionary relating each possible prediction to a list of dist
        if neighborPrediction not in predictionMap:
            predictionMap[neighborPrediction] = []
```

```
        predictionMap[neighborPrediction].append(distance)
    # Relate each prediction to an average distance (among the neighbors)
    for prediction in predictionMap:
        predictionMap[prediction] = np.sum(predictionMap[prediction])/len(predict
    return min(predictionMap, key = predictionMap.get)

# @param: validationBlock - array of arrays holding point data for the validation
# @param: validationResults - array of integers corresponding to the respective c
# @param: nonValidationBlock - array of arrays holding point data for the trainin
# @param: nonValidationResults - array of integers corresponding to the respectiv
# @return: the ratio of incorrectly classified numbers in the validation set
def getValidationError(validationBlock, validationResults, nonValidationBlock, no
    misclassified = 0
    for i in range(len(validationBlock)):
        point = validationBlock[i]
        neighbors, distances = getKNN(point, nonValidationBlock, k)
        prediction = makePrediction(neighbors, nonValidationResults, distances)
        if prediction != validationResults[i]:
            misclassified += 1
    return misclassified/len(validationBlock)
```

In [158]:

```
##### Run cross validation
validationErrors = []
for i in range(len(kVals)):
    k = kVals[i]
    # Build the validation set
    validationStart = VALIDATION_BLOCK_SIZE * i
    validationEnd = VALIDATION_BLOCK_SIZE * (i + 1)
    validationBlock = trainingData[validationStart : validationEnd]
    validationResults = trainingResults[validationStart : validationEnd]
    # Build the nonvalidation set
    nonValidationBlock = []
    nonValidationResults = []
    for j in range(len(trainingData)):
        if j < validationStart or j >= validationEnd:
            nonValidationBlock.append(trainingData[j])
            nonValidationResults.append(trainingResults[j])
    # Calculate validation error
    validationError = getValidationError(validationBlock, validationResults, nonV
    validationErrors.append(validationError)
```

In [159]:
```
from matplotlib import pyplot as plt

plt.plot(kVals, validationErrors)
plt.title("k-Choice and validation error on k-NN digit classification")
plt.xlabel("k value")
plt.xscale('log')
plt.ylabel("validation error")
plt.show()
```