Lab 4: Clustering and Linear Regression

Name: Shamith Achanta (shamith2)

Due September 23, 2019 11:59 PM

Logistics and Lab Submission

See the course website. Remember that all labs count equally, despite the labs being graded from a different number of total points).

What You Will Need to Know For This Lab

- K-means clustering
- Vector Quantization
- Nearest Neighbors Classification
- Linear Regression

Preamble (don't change this)

In [1]:
```
%pylab inline
import numpy as np
from sklearn import neighbors
from numpy import genfromtxt
import scipy.spatial.distance as dist
import random
from sklearn.cluster import KMeans
from PIL import Image
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

```
Populating the interactive namespace from numpy and matplotlib
```

Problem 1: Selecting the number of clusters (30 points)

Write a function which implements K-means clustering.

You will be given as input:

- A (N, d) numpy.ndarray of unlabeled data (with each row as a feature vector), data
- A scalar K which indicates the number of clusters
- A scalar representing the number of iterations, *niter* (this is your stopping criterion/criterion for convergence)

Your output will be a tuple consisting of a vector of length N containing which cluster $(0, \ldots, K-1)$ a feature vector is in and a (K, d) matrix with the rows containing the cluster centers.

Do not use scikit-learn or similar for implement K-means clustering. You may use `scipy.spatial.distance.cdist` to calculate distances. Initialize the centers randomly without replacement with points from the data set. `random.sample` or `np.random.choice` may be useful for this. **(10 points)**

In [2]:
```
def kMeans(data,K,niter):
    #Put your code here
    centers  = data[np.random.choice(range(data.shape[0]), K, replace=False), :]
    for i in range(niter):
        sq_dists = np.sum((centers[: ,np.newaxis, :] - data)**2,axis=2)
        closest  = np.argmin(sq_dists, axis=0)
        for j in range(K):
            centers[j,:] = data[closest==j,:].mean(axis=0)

    return closest, centers
```

The K-means clustering problem tries to minimize the following quantity by selecting $\{z_i\}_{i=1}^N$ and $\{\mu_k\}_{k=1}^K$: $J_K(\{z_i\}_{i=1}^N, \{\mu_k\}_{k=1}^K) = \sum_{i=1}^N$ I where \mathbf{\mu}_{z_i} is the center of the cluster to which \mathbf{x}_i is assigned.

One visual heuristic to choose the number of clusters from the data (where the number of clusters is not known a priori) is to estimate the optimal value of J_K(\{z_i\}_{i=1}^N ,\{\mu_k\}_{k=1}^K), J^*(K) , for different values of K and look for an "elbow" or "knee" in the curve of J^* versus K and choose that value of K.

In this part of the problem, you will run K-means 100 times for each K=2,\ldots,10 and calculate J_K(\{z_i\}_{i=1}^N ,\{\mu_k\}_{k=1}^K) for the clustering given by K-means. Use the smallest value of J_K(\{z_i\}_{i=1}^N ,\{\mu_k\}_{k=1}^K) in the runs of K-means for each value of K to form an estimate of J^*(K). **Plot this estimate versus K (with correct labels!). Which K should you pick by this heuristic?** Use niter=100 for each run of K-means.

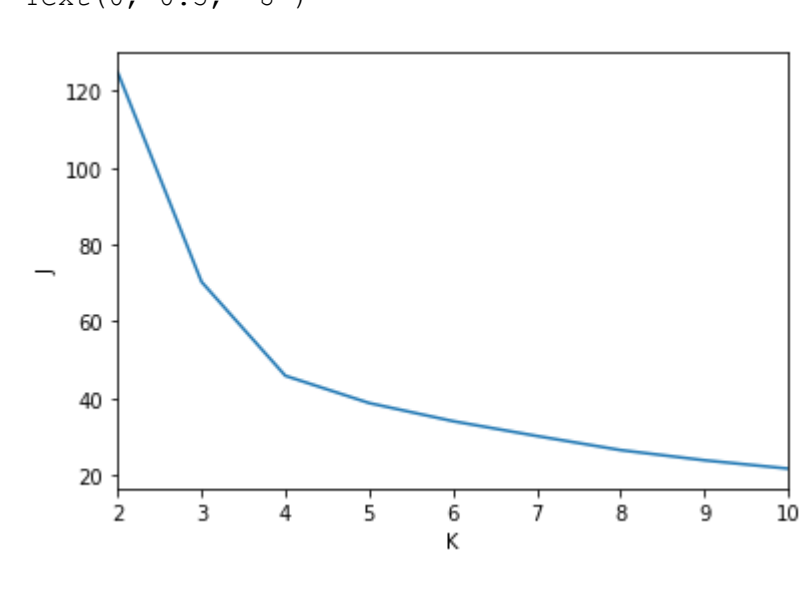
For an attempt to formalize this heuristic, see Tibshirani, Robert, Guenther Walther, and Trevor Hastie. "Estimating the number of clusters in a data set via the gap statistic." Journal of the Royal Statistical Society: Series B (Statistical Methodology) 63.2 (2001): 411-423. Sometimes, an elbow does not exist in the curve or there are multiple elbows or the K value of an elbow cannot be unambiguously identified. Further material can be found on Wikipedia as well.

Note: Your code should be relatively quick -- a few minutes, at worst. **(10 points)**

In [3]:
```
# Load up some data, which we will store in a variable called problem1
data_p1 = genfromtxt('problem1.csv', delimiter=',')
```

In [4]:
```
# Put your code here
# constants
k_range = range(2, 11)
n_iter = 100

J = np.zeros(len(k_range))
distortions = np.zeros(10)
runs = np.zeros(n_iter)

for k in k_range:
    for i in range(n_iter):
        closest, center = kMeans(data_p1, k, n_iter)

        for k2 in range(center.shape[0]):
            if k2 in closest:
                distortions[k2] = np.linalg.norm(data_p1[closest == k2] - center[k2]) ** 2

            else:
                continue

        runs[i] = np.sum(distortions)
        distortions[:] = 0

    J[k-2] = np.min(runs)
    runs[:] = 0

plot(np.arange(2,11), J)
xlim(2,10)
xlabel("K")
ylabel("J")
```

Out[4]:
```
Text(0, 0.5, 'J')
```

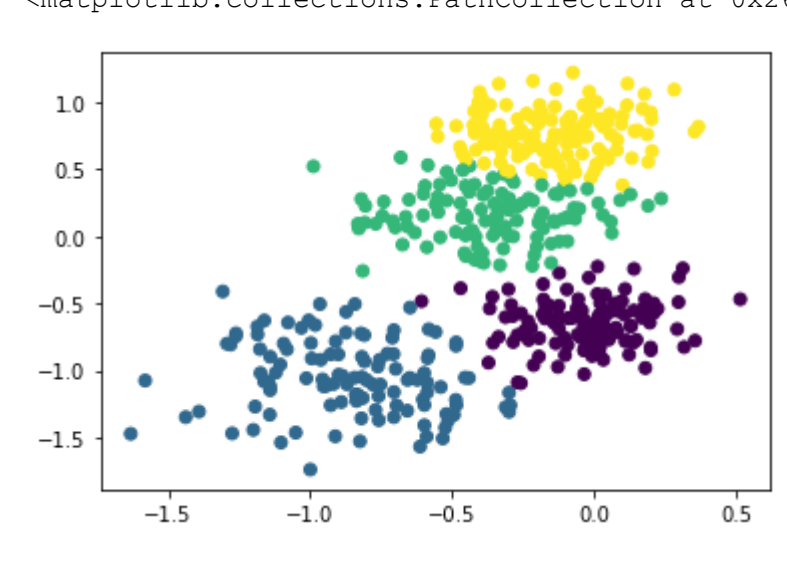


[K = 4]

Using the value of K you determined from the elbow, perform K-means clustering on the data. Plot it as a scatter plot with the colors given by the labels. You don't need to show the legends. (5 points)

In [5]:
```
# Put your code here
groups, centers = kMeans(data_p1, K=4, niter=100)
scatter(data_p1[:,0], data_p1[:,1], c=groups)
```

Out[5]:
```
<matplotlib.collections.PathCollection at 0x268179995c8>
```



Should you pick the K such that J^*(K) is minimized? Why or why not? (5 points)

[No, picking a K that minimizes J^*(K) splits true clusters into additional clusters, whenever there is a small decrease in J^*(K). Therefore, the optimal K is where the marginal change in J^*(K) is equal to 0]

Problem 2: Vector Quantization (40 points)

In this problem, you will implement vector quantization. You will use `sklearn.cluster.KMeans` for the K-means implementation and use *k-means++* as the initialization method. See Section 4.2.1 in the notes for details.

Write a function to generate a codebook for vector quantization. You will be given inputs:

- A (N,M) numpy.ndarray representing a greyscale image, called *image*. (If we want to generate our codebook from multiple images, we can concatenate the images before running them through this function).
- A scalar B, for which you will use B \times B blocks for vector quantization. You may assume N and M are divisible by B.
- A scalar K, which is the size of your codebook

You will return:

- The codebook as a (K,B^2) numpy.ndarray. **(10 points)**

In [6]:
```
def trainVQ(image,B,K):
    #Put your code here
    codebook = np.zeros((K,B**2))
    N,M = image.shape
    horz = np.split(image, image.shape[0]/B)
    stack = []

    for subary in horz:
        stack.extend(np.asarray(np.split(subary, image.shape[1]/B, axis=1)))

    reshaped = []
    for mtx in stack:
        reshaped.append(mtx.reshape(B**2))

    return KMeans(n_clusters=K, init="k-means++").fit(np.asarray(reshaped)).cluster_centers_
```

Write a function which compresses an image against a given codebook. You will be given inputs:

- A (N,M) numpy.ndarray representing a greyscale image, called image. You may assume N and M are divisible by B.
- A (K,B^2) codebook called codebook
- Block width B

You will return:

- A (N/B,M/B) numpy.ndarray consisting of the indices in the codebook used to approximate the image.

You can use the nearest neighbor classifier from scikit-learn if you want (though it is not necessary) to map blocks to their nearest codeword. **(10 points)**

In [7]:
```
def compressImg(image,codebook,B):
    #Put your code here
    N, M = image.shape
    result = np.zeros((N//B,M//B))

    horz = np.split(image, image.shape[0]/B)
    loss = []

    for i in range(len(horz)):
        row = np.split(horz[i], image.shape[1]/B, axis=1)

        for j in range(len(row)):
            for center in codebook:

                loss.append(np.linalg.norm(center - row[j].reshape(B**2)))

            result[i][j] = np.argmin(loss)
            del loss[:]

    return result
```

Write a function to reconstruct an image from its codebook. You will be given inputs:

- A (N/B,M/B) numpy.ndarray containing the indices of the codebook for each block called indices
- A codebook as a (K,B^2) numpy.ndarray called codebook
- Block width B

You will return a (N,M) numpy.ndarray representing the image. **(10 points)**

In [8]:
```
def decompressImg(indices,codebook,B):
    #Put your code here
    img = np.zeros((indices.shape[0]*B, indices.shape[1]*B))

    for i in range(indices.shape[0]):
        for j in range(indices.shape[1]):
            img[(B*i):(B*i)+B,(B*j):(B*j)+B] = codebook[indices[i][j].astype("int")].reshape(B,B)

    return img
```

Run your vector quantizer with 5 \times 5 blocks on the provided image with codebook sizes K=2,5,10,20,50,100,200 (i.e. generate codebooks from this image of those sizes, compress the image using those codebooks and reconstruct the images). **Display (for each K) and comment on the reconstructed images (you may be quantitative (e.g. PSNR) or qualitative). Which code book would you pick? Why?** Make sure to take into account the bits per pixel used by the compressor.

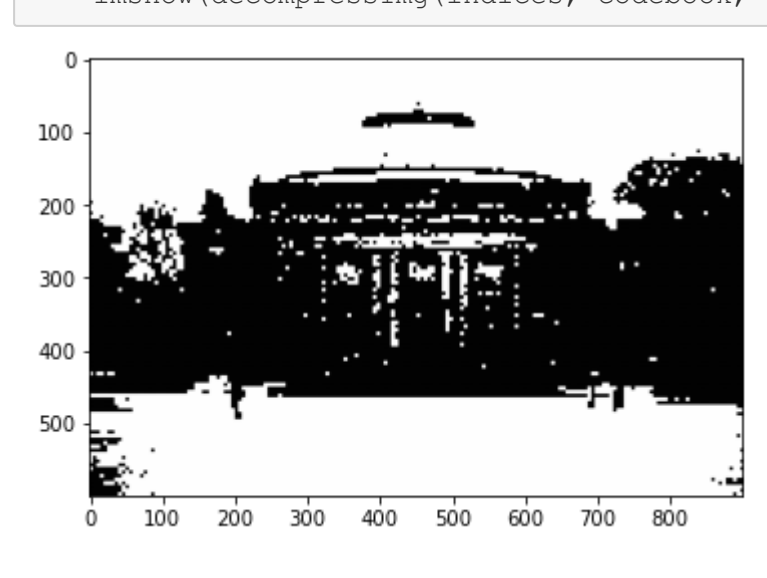
Note the number of bits per pixel can be approximated as \frac{\log_2 K}{25} and the codebook takes approximately 200K bits (assuming each pixel is stored as 8 bits). Some good ideas on quantitative arguments for codebook size can be found in Gonzalez & Woods, Digital Image Processing 3e or Gersho & Gray, Signal Compression & Vector Quantization. It is not necessary to look at these references for quantitative arguments, though. **(10 points)**

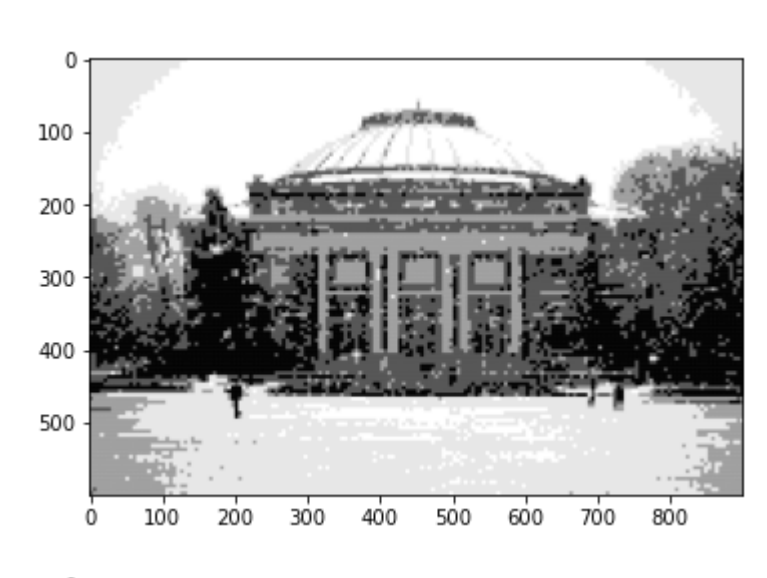
The image used is under fair use from Daily Illini.

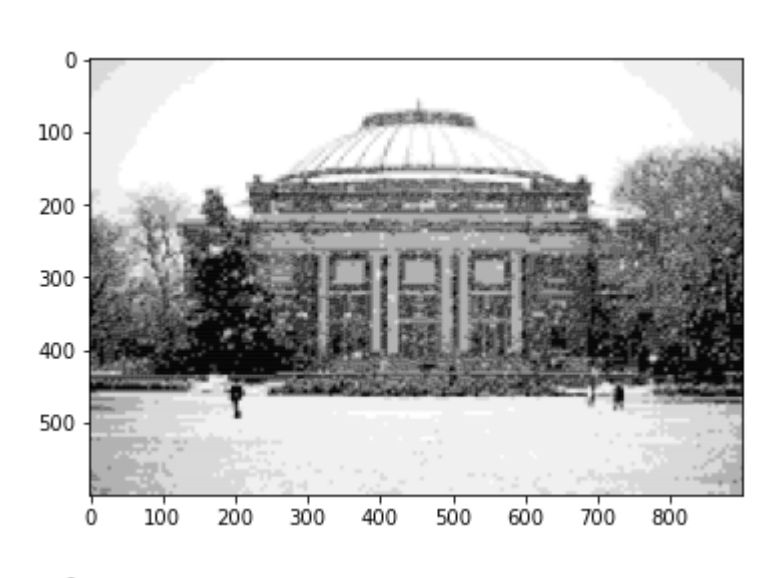
In [9]:
```
# The provided image is stored in image
image = np.asarray(Image.open("mrtb.jpg").convert("L"))
imshow(image, cmap = cm.Greys_r)
```

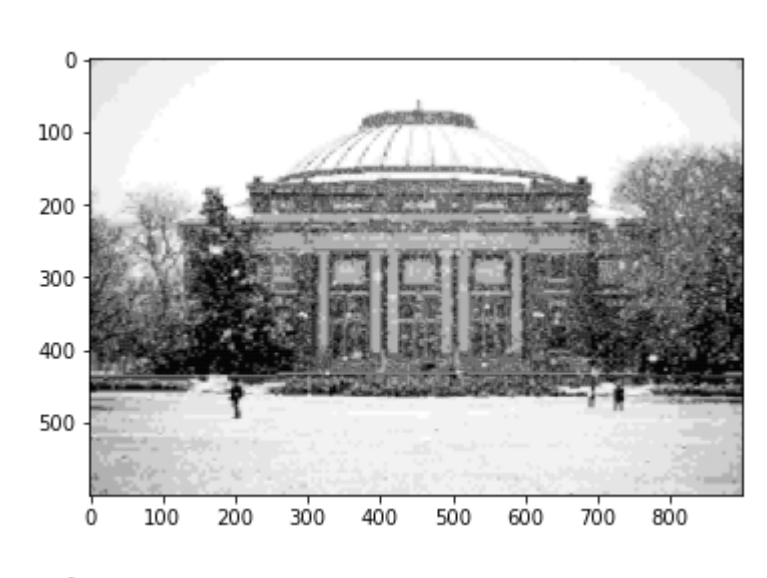
Out[9]:
```
<matplotlib.image.AxesImage at 0x26817a15688>
```

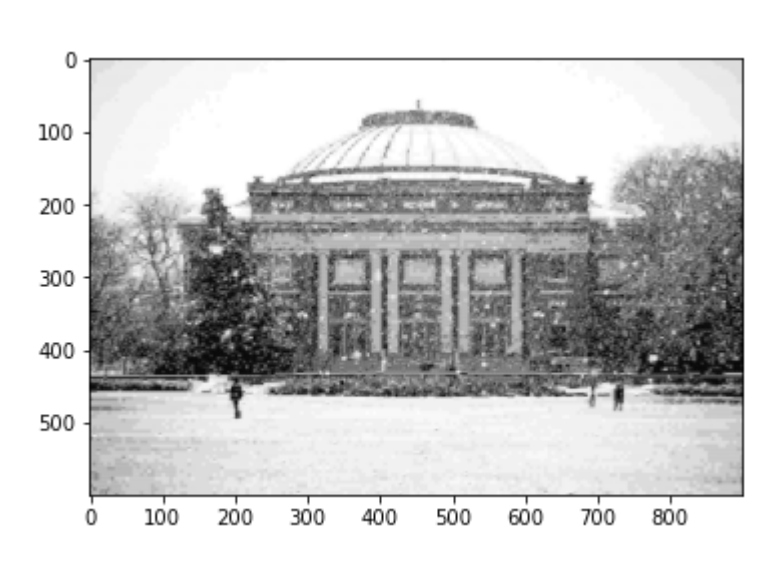
In [10]:
```
# Put your code here
for k in [2, 5, 10, 20, 50, 100, 200]:
    codebook = trainVQ(image, 5, k)
    indices = compressImg(image,codebook,5)
    pylab.figure()
    imshow(decompressImg(indices, codebook, 5), cmap=cm.Greys_r)
```

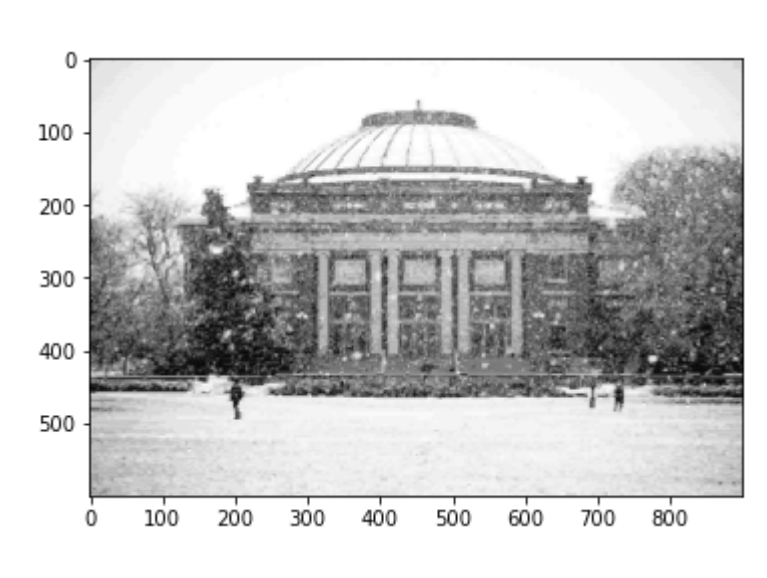


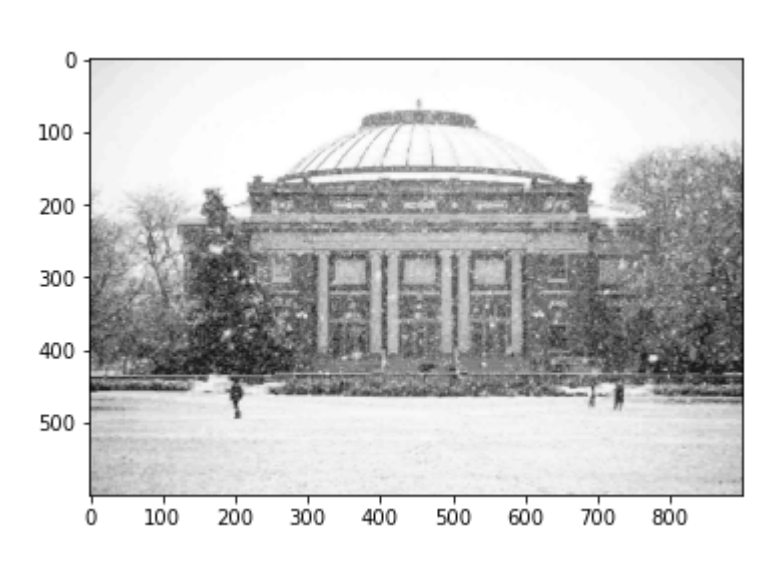












[I would pick the codebook where K = 50 (with ~0.226 bpp) because the loss of detail in the image and the run-time of the compression and decompression algorithms are tolerable, when compared to the faster compressions/decompressions when K < 50 and more detailed images when K > 50]

Problem 3: Using K-means to Accelerate Nearest Neighbors (20 points)

In this problem, you will use K-means clustering to accelerate nearest neighbors, as outlined in the notes (Algorithm 7). Use `sklearn.neighbors.KNeighborsClassifier` for nearest neighbor classification and `sklearn.cluster.KMeans` for the K-means implementation with *k-means++* as the initialization method.

You will write a function to generate prototypes from labeled data. It will have input:

- Training features as (N,d) numpy.ndarray called *traindata*
- Training labels as a length N vector called *trainlabels*
- K, the number of prototypes under each class

You will return a tuple containing:

- The prototypes selected as a (K*\text{number of classes},d) numpy.ndarray
- The corresponding labels as a K*\text{number of classes} length vector

You may assume there are at least K examples under each class. `set(trainlabels)` will give you the set of labels. **(10 points)**

In [11]:
```
def generatePrototypes(traindata,trainlabels,K):
    # Put your code here
    classes = np.unique(trainlabels)
    clusterdata = np.zeros((K * classes.size, traindata.shape[1]))
    clusterlabels = np.zeros(K * classes.size)

    for i in range(classes.size):
        km = KMeans(n_clusters = K, init='k-means++').fit(traindata[trainlabels == i])
        clusterdata[i*K:(i+1)*K] = km.cluster_centers_
        clusterlabels[i*K:(i+1)*K] = i

    return clusterdata, clusterlabels
```

Train a nearest neighbor classifier (i.e. 1-NN) with 1,10,50,100 and 200 prototypes per class for the digits data set from Lab 2. **Comment on the validation error and computational complexity versus the nearest neighbor classifier from Lab 2 (error=0.056) and the LDA classifier (error=0.115) from Lab 2. Which classifier would you pick? Why?**

Note that this data set is generated from zip code digits from US mail, and the US Postal Service processes hundreds of millions of pieces of mail a day, so a small improvement in error can lead to tremendous savings in terms of mis-routed packages (which cost a lot of money and time to re-transport). **(10 points)**

In [12]:
```
# Load the digits data set

#Read in the Training Data
traindata_tmp= genfromtxt('zip.train', delimiter=' ')
#The training labels are stored in "trainlabels", training features in "traindata"
trainlabels=traindata_tmp[:,0]
traindata=traindata_tmp[:,1:]


#Read in the Validation Data
valdata_tmp= genfromtxt('zip.val', delimiter=' ')
#The validation labels are stored in "vallabels", validation features in "valdata"
vallabels=valdata_tmp[:,0]
valdata=valdata_tmp[:,1:]
```

In [13]:
```
# Put your code here
for i in [1, 10, 50, 100, 200]:
    generatePrototypes(traindata, trainlabels, i)
    prototypedata, prototypelabels = generatePrototypes(traindata, trainlabels, i)
    KNN = neighbors.KNeighborsClassifier(n_neighbors = 1)
    KNN.fit(prototypedata, prototypelabels)

    # print results
    start_time = time.time()
    print("Error with {} prototype(s): {}".format(i, (1 - KNN.score(valdata, vallabels))))
    print("Predicition Time: {} s \n".format(time.time() - start_time))
```

```
Error with 1 prototype(s): 0.1858495266567015
Predicition Time: 0.09977889060974121 s

Error with 10 prototype(s): 0.08470353761833582
Predicition Time: 0.2302405834197998 s

Error with 50 prototype(s): 0.06427503736920781
Predicition Time: 1.014786958694458 s

Error with 100 prototype(s): 0.06527154957648229
Predicition Time: 1.455094814300537 s

Error with 200 prototype(s): 0.05879422022919778
Predicition Time: 2.9690682888031006 s
```

[Given the results of the above test, I would pick the nearest neighbor classifier. The most accurate result of the nearest neighbors produced an error of 5.879% and took nearly 2 seconds to test. This means that the nearest neighbor algorithm had a 0.279% lower error and had a lower time complexity, taking only ~0.512 seconds to classify when using the brute force method. LDA, by comparison, only performed when compared to an accelerated approach of < 10 prototypes, however only took about 0.0046 seconds to classify]

Problem 4: Linear Regression (35 points)

In this problem, you will do model selection for linear regression using Ordinary Least Squares, Ridge Regression and the LASSO.

The dataset you will use has 8 features:

```
lcavol - log cancer volume
lcaweight - log prostate weight
age
lbph - log of amount of benign prostatic hyperplasia
svi - seminal vesicle invasion
lcp - log capsular penetration
gleason - Gleason score
pgg45 - percent of Gleason scores 4 or 5
```

and you will predict the level of a prostate-specific antigen. The data set was collected from a set of men about to receive a radical prostatectomy. More details are given in Section 3.2.1 in Elements of Statistical Learning 2e by Hastie et al.

In [14]:
```
# Load the data
trainp= genfromtxt('trainp.csv', delimiter=',')

# Training data:
trainfeat=trainp[:,:-1] #Training features (rows are feature vectors)
trainresp=trainp[:,-1] #Training responses

valp= genfromtxt('valp.csv',delimiter=',')
# Validation data:
valfeat=valp[:,:-1] #Validation Features (rows are feature vectors)
valresp=valp[:,-1] #Validation Response

# Standardize and center the features
ftsclr=StandardScaler()
trainfeat = ftsclr.fit_transform(trainfeat)
valfeat= ftsclr.transform(valfeat)
# and the responses (note that the example in the notes has centered but not
#                     standardized responses, so your numbers won't match up)
rsclr=StandardScaler()
trainresp = (rsclr.fit_transform(trainresp.reshape(-1,1))).reshape(-1)
valresp= (rsclr.transform(valresp.reshape(-1,1))).reshape(-1)
```

In [15]:
```
# The training features are in trainfeat
# The training responses are in trainresp
# The validation features are in valfeat
# The validation responses are in valresp
```

Since we centered the responses, we can begin with a benchmark model: Always predict the response as zero (mean response on the training data). Calculate the validation RSS for this model. **(5 points)**

If another model does worse than this, it is a sign that something is amiss.

Note: The RSS on a data set with V samples is given by \frac{1}{V} \lVert \mathbf{y} - \hat{\mathbf{y}} \rVert^2 where \mathbf{y} is a vector of the responses, and \hat{\mathbf{y}} is the predicted responses on the data.

In [16]:
```
# Put your code here
features = np.asarray(["lcavol","lcaweight", "age","lbph","svi","lcp", "gleason", "pgg45"])

def RSS(V, y_actual, y_predict):
    return (1/V)*(np.linalg.norm(y_actual - y_predict)**2)

regr = linear_model.LinearRegression()
regr.fit(trainfeat, trainresp)
predresp = regr.predict(valfeat)

print("RSS with Benchmark: {}".format(RSS(predresp.size, np.zeros(predresp.size), predresp)))
```

```
RSS with Benchmark: 0.3637162154335814
```

[RSS Validation Score: 0.364]

First, you will try (Ordinary) Least Squares. Use `sklearn.linear_model.LinearRegression` with the default options. Calculate the validation RSS. **(5 points)**

Note: The .score() method returns an R^2 value, not the RSS, so you shouldn't use it anywhere in this problem.

In [17]:
```
# Put your code here
lm = linear_model.LinearRegression().fit(trainfeat, trainresp)
predicted = lm.predict(valfeat)
print("RSS with OLS: {}".format(RSS(predicted.size, predicted, valresp)))
```

```
RSS with OLS: 0.36230709903819774
```

[RSS Validation Score: 0.362]

Now, you will apply ridge regression with `sklearn.linear_model.Ridge`.

Sweep the regularization/tuning parameter \alpha=0,\ldots,100 with 1000 equally spaced values.

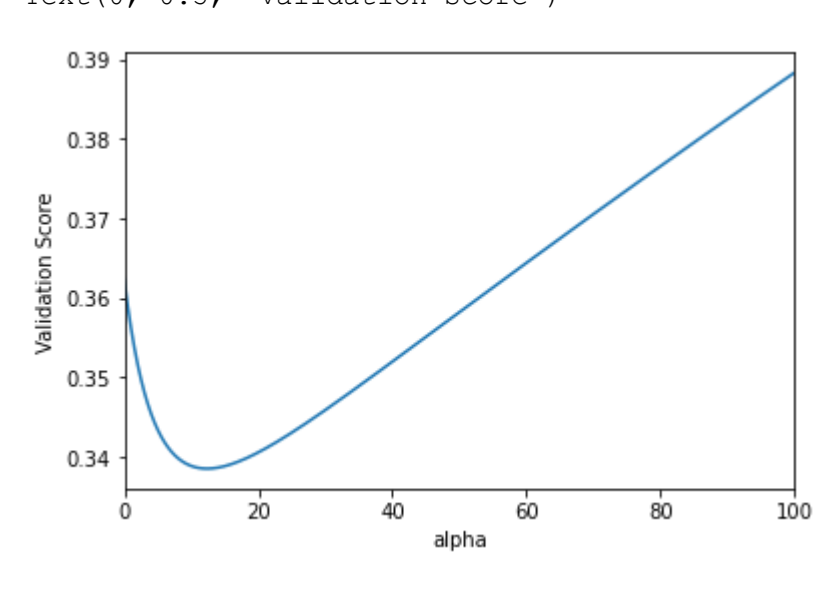
Make a plot of the RSS on the validation set versus \alpha. What is the minimizing \alpha, corresponding coefficients and validation error?

Note: Larger values of \alpha shrink the weights in the model more. \alpha=0 corresponds to the LS solution. **(10 points)**

In [18]:
```
# Put your code here
result = []
alphas = np.linspace(0,100,num=1000, endpoint=True)

for a in alphas:
    clf = linear_model.Ridge(alpha = a)
    clf.fit(trainfeat, trainresp)

    predresp = clf.predict(valfeat)
    result.append(RSS(predresp.size, predresp, valresp))

print("Minimizing Alpha: {}".format(alphas[np.argmin(result)]))
print("Corresponding Coefficients: {}".format(np.argmin(result)))
print("Validation Score: {}".format(np.min(result)))

print("Selected Features: {}".format(features[(linear_model.Ridge(alpha = alphas[np.argmin(result)]).fi
t(trainfeat, trainresp).coef_ >= 0)]))

pylab.plot(alphas,result)
pylab.xlim(0,100)
xlabel("alpha")
ylabel("Validation Score")
```

```
Minimizing Alpha: 12.312312312312313
Corresponding Coefficients: 123
Validation Score: 0.3384853327816166
Selected Features: ['lcavol' 'lcaweight' 'lbph' 'svi' 'gleason' 'pgg45']
```

Out[18]:
```
Text(0, 0.5, 'Validation Score')
```



[Minimum Alpha: 12.312312312312313; Corresponding Coefficients: 123; Validation Score: 0.3384853327816166]

Now, you will apply the LASSO with `sklearn.linear_model.Lasso`.

Sweep the tuning/regularization parameter \alpha=0,\ldots,1 with 1000 equally spaced values.

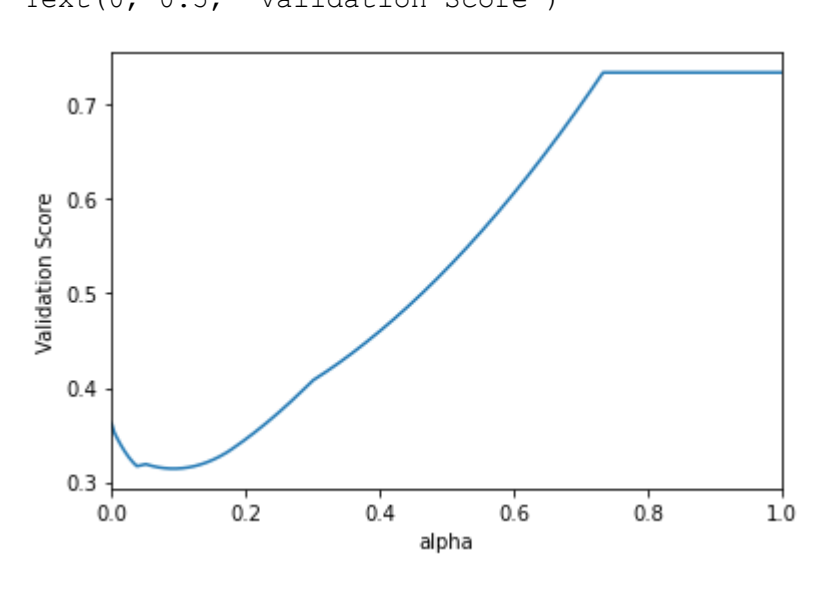
Make a plot of the RSS on the validation set versus \alpha. What is the minimizing \alpha, corresponding coefficients and validation error?

Note: Larger values of \alpha lead to sparser solutions (i.e. less features used in the model), with a sufficiently large value of \alpha leading to a constant prediction. Small values of \alpha are closer to the LS solution, with \alpha=0 being the LS solution. **(10 points)**

In [19]:
```
# Put your code here
result = []
alphas = np.linspace(0,1,num=1000, endpoint=True)

for a in alphas:
    clf = linear_model.Lasso(alpha = a)
    clf.fit(trainfeat, trainresp)

    predresp = clf.predict(valfeat)
    result.append(RSS(predresp.size, predresp, valresp))

print("Minimizingn Alpha: {}".format(alphas[np.argmin(result)]))
print("Corresponding Coefficients: {}".format(np.argmin(result)))
print("Validation Score: {}".format(np.min(result)))

print("Selected Features: {}".format(features[(linear_model.Lasso(alpha = alphas[np.argmin(result)]).fi
t(trainfeat, trainresp).coef_ != 0)]))

pylab.plot(alphas,result)
pylab.xlim(0,1)
xlabel("alpha")
ylabel("Validation Score")
```

```
Minimizingn Alpha: 0.09309309309309309
Corresponding Coefficients: 93
Validation Score: 0.3142651111631038
Selected Features: ['lcavol' 'lcaweight' 'lbph' 'svi' 'pgg45']
```

Out[19]:
```
Text(0, 0.5, 'Validation Score')
```



[Minimizingn Alpha: 0.09309309309309309; Corresponding Coefficients: 93; Validation Score: 0.3142651111631038]

Which features were selected by Ridge Regression when minimizing the RSS on the validation set? Which features were selected by LASSO when minimizing the RSS on the validation set? Which model would you choose and why? You reason should include both the error and the model complexity. (5 points)

**[Features Selected by Ridge: lcavol, lcaweight, lbph, svi, gleason, pgg45

Features Selected by Lasso: lcavol, lcaweight, lbph, svi, pgg45

I would choose the LASSO because when compared to the benchmark, OLS, and Ridge regressions, it produced the smallest validation error]**

And this concludes Lab 4! Congratulations!