COMP5318 - Machine Learning and Data Mining: Assignment 1

Due: Wednesday 14 Oct 2020 11:59PM

The goal of this assignment is to build a classifier to classify some grayscale images of the size 28x28 into a set of categories. The dimension of the original data is large, so you need to be smart on which method you gonna use and perhaps perform a pre-processing step to reduce the amount of computation. Part of your marks will be a function of the performance of your classifier on the test set.

Hardware and software specifications

MacBook Pro (Retina, 13-inch, Mid 2014)
Processor: 2.8 GHz Intel Core i5
Memory: 8 GB 1600 MHz DDR3
Graphics: Intel Iris 1536 MB
OS: macOS Mojave Version 10.14.6

Code is written in Visual Studio Code with Python Extension for Visual Studio Code by Microsoft

Load Libraries

In [1]:
```
import h5py
import numpy as np
import os
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
from bisect import bisect
from scipy.spatial import distance
from math import exp, sqrt, pi
print(os.listdir("./Input/train"))
print(os.listdir("./Input/test"))
# train_files = [name for name in os.listdir("./Input/train") if no
t name.endswith('DS_Store')]
# test_files = [name for name in os.listdir("./Input/test") if not
name.endswith('DS_Store')]
# print(train_files)
# print(test_files)
```

```
['images_training.h5', 'labels_training.h5']
['.DS_Store', 'images_testing.h5', 'labels_testing_2000.h5']
```

Load Data

In [2]:
```
with h5py.File('./Input/train/images_training.h5','r') as H:
    data_train = np.copy(H['datatrain'])
with h5py.File('./Input/train/labels_training.h5','r') as H:
    label_train = np.copy(H['labeltrain'])

# using H['datatest'], H['labeltest'] for test dataset.
print(data_train.shape,label_train.shape)
```

```
(30000, 784) (30000,)
```

In [3]:
```
with h5py.File('./Input/test/images_testing.h5','r') as H:
    data_test = np.copy(H['datatest'])
with h5py.File('./Input/test/labels_testing_2000.h5','r') as H:
    label_test = np.copy(H['labeltest'])

# using H['datatest'], H['labeltest'] for test dataset.
print(data_test.shape,label_test.shape)
```

```
(5000, 784) (2000,)
```

Data Pre-processing

Principal Component Analysis

Sourced: Tutorial 2 - Matrix Decomposition
Sourced: https://stats.stackexchange.com/questions/125172/pca-on-train-and-test-datasets-should-i-run-one-pca-on-traintest-or-two-separa
https://towardsdatascience.com/pca-with-numpy-58917c1d0391
https://stackoverflow.com/questions/10818718/principal-component-analysis

Computing the Eigenvectors and Eigenvalues

In [4]:
```
data = data_train   #Create copy of training data
data = data - np.mean(data, axis=0) #Centre observations by zero me
an
print(data.shape)
covariance_matrix = np.cov(data.T)  #Create the covariance matrix b
etween the 784 features
#print(covariance_matrix)
eig_val, eig_vec = np.linalg.eig(covariance_matrix) #From linear al
gebra, retrive the eiganvalue and eigan vectors from the covariance
matrix
print("First 20 Eigenvalues: \n", eig_val[:20], "\n")
```

```
(30000, 784)
First 20 Eigenvalues:
 [19.8611004  12.10997382  4.1078177   3.3719857   2.61461635  2.3
5693788
  1.61184549  1.28149922  0.92593176  0.89463432  0.67365696  0.62
224642
  0.52434522  0.44943814  0.41495554  0.4023021   0.37964251  0.36
276613
  0.31522305  0.31177036]
```

Picking Principal Components

In [5]:
```
variance_explained = [] #Declare list to hold all variances explaine for each feature.
for i in eig_val:
    variance_explained.append((i/sum(eig_val))*100)    #Get proportion of eiganvalue of each feature and multiply by 100 to get percentage
print("First 10 Variance Explained: \n", variance_explained[:10], "\n")

cumulative_variance_explained = np.cumsum(variance_explained)   #Gets list of cummulative varianccces
print("First 10 Cummulative Variance Explained: \n", cumulative_variance_explained[:10], "\n")
```

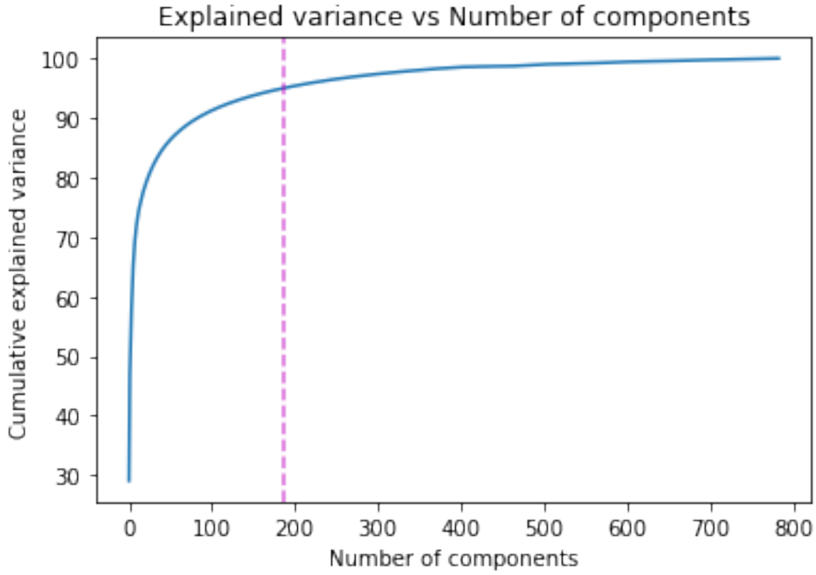
```
First 10 Variance Explained: 
 [29.077053310731294, 17.72924699012503, 6.013928330120427, 4.936655363514209, 3.8278512994969107, 3.4506048031981504, 2.3597744497586266, 1.876140819639972, 1.3555828542550032, 1.309762764368793]

First 10 Cummulative Variance Explained: 
 [29.07705331 46.8063003  52.82022863 57.75688399 61.58473529 65.0353401
 67.39511455 69.27125537 70.62683822 71.93660099]
```

Plot Cumulative explained variance to find elbow point

In [6]:
```
comp = bisect(cumulative_variance_explained, 95) #Number of components at 95% explained variance
sns.lineplot(x = np.arange(data.shape[1]), y=cumulative_variance_explained) #Plot numbercomponents by cummulative variance
plt.axvline(comp, c='m', linestyle='--', alpha=0.6) #abline to show where we are cutting for 95% variance explained
plt.xlabel("Number of components")
plt.ylabel("Cumulative explained variance")
plt.title("Explained variance vs Number of components")
```

Out[6]:
```
Text(0.5, 1.0, 'Explained variance vs Number of components')
```



As seen from the diagram above, the number of components for 95% variance explained is 187 components. Below we set this globablly.

In [7]:
```
n_component = comp
print("Number of components for 95% Explained Variance:", n_component)
```

```
Number of components for 95% Explained Variance: 187
```

Project Data Onto Lower-Dimensional Linear Subspace

</br> We now will create our projection matrix by reducing our array of eiganvectors by the number of components found above

Project onto training data

In [8]:
```
projection_matrix = (eig_vec.T[:][:n_component]).T
X_train_pca = data.dot(projection_matrix)
print(X_train_pca.shape)
```

```
(30000, 187)
```

Project on to test data

To keep consistency the same projecetion matrix is applied onto the test data.

In [9]:
```
data = data_test
data = data - np.mean(data, axis=0)
X_test_pca = data.dot(projection_matrix)
print(X_test_pca.shape)
```

```
(5000, 187)
```

Split Data

Split training data into train and validaiton sets on 70:30

In [10]:
```
train_pct_index = int(0.7 * X_train_pca.shape[0])
X_train, X_val = X_train_pca[:train_pct_index], X_train_pca[train_p
ct_index:]
y_train, y_val = label_train[:train_pct_index], label_train[train_p
ct_index:]
```

Classification Models

kNN Classificaiton Algorithm

Sourced: COMP5318 Tutorial 5 - Classification I

In [11]:
```
def calc_knn(X, y, K, X_q, distance_method):
    if distance_method == "squared" :   #Calculate distance between 
X_q and each training point
        dis = ((X - X_q)**2).sum(axis=1) #Sum the distance of entir
e row, in other words by columns.
    elif distance_method =='euclidean':
        dis = np.sqrt(((X - X_q)**2).sum(axis=1))
    elif distance_method == "manhattan":
        dis = (X - X_q).sum(axis=1)
    else:
        raise ValueError('Undetermined distance')
    arg_ascending = np.argsort(dis)      #Sort distanc matrix
    return stats.mode(y[arg_ascending[:K]]).mode    #Take the neigh
bour that occurs the most
```

Cross Validation

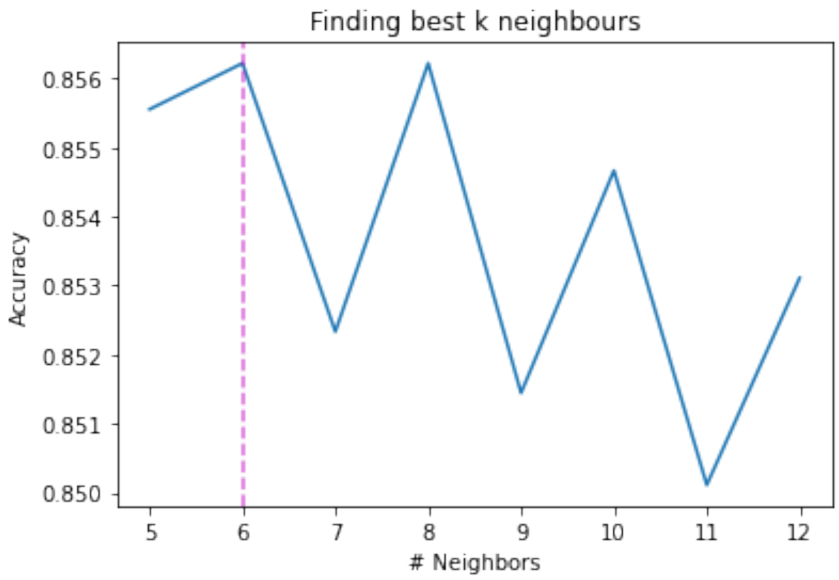
Find best k method

In [12]:
```
%%time
method = "squared"
k_list = np.arange(5, 13)
score_list = []
for k in k_list:    #iterate through the range of k neighbours
    y_pred = np.empty((len(X_val)))     #Declare empty arrary to hol
d predictions
    print("neighbour", k)
    for i in range(len(y_val)):     #Iterate each sample
        y_pred[i]= calc_knn(X_train, y_train, k, X_val[i,:],method) 
#Get prediction for given k
    score_list.append(np.mean(y_pred == y_val))     #append score in 
to a score list
score_list = np.array(score_list)
```

```
neighbour 5
neighbour 6
neighbour 7
neighbour 8
neighbour 9
neighbour 10
neighbour 11
neighbour 12
CPU times: user 19min 2s, sys: 3min 18s, total: 22min 21s
Wall time: 23min 22s
```

Plot scores for each k and retrieve k with highest score

In [13]:
```
fig, ax = plt.subplots()
n_comp_list = np.arange(5, 13)
ax.plot(k_list, score_list)
ax.axvline(n_comp_list[np.argmax(score_list)], c='m', linestyle='--
', alpha=0.6)
ax.set_xlabel('# Neighbors')
ax.set_ylabel('Accuracy')
ax.set_xticks(k_list)
ax.set_title('Finding best k neighbours')
plt.show()
```



Set the k that yields the best accuracy

In [14]:
```
k = k_list[np.argmax(score_list)]
```

In [15]:
```
%%time
y_pred = np.empty((len(X_val)))
for i in range(len(y_val)):
    y_pred[i] = calc_knn(X_train, y_train, k, X_val[i,:], method)
```

```
CPU times: user 2min 21s, sys: 24.3 s, total: 2min 45s
Wall time: 2min 51s
```

In [16]:
```
y_true = y_val
y_pred = y_pred

y_true = pd.Series(y_true, name='Actual')
y_pred = pd.Series(y_pred, name='Predicted')
confusion_matrix = pd.crosstab(y_true, y_pred)
print(confusion_matrix)
kNN_train_acc = np.sum(y_true == y_pred[:y_true.shape[0]])/len(y_tr
ue)
```

```
Predicted  0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0
Actual
0          797    2   13   22    7    0   90    0    6    0
1            6  835    6   18    0    0    3    0    1    0
2           12    0  676   11   99    0   68    0    2    0
3           28    6    7  801   32    0   18    0    2    0
4            3    2  105   35  686    0   80    0    1    0
5            1    0    1    0    0  848    0   55    5   41
6          165    0  123   15   70    0  500    0   10    0
7            0    0    0    0    0   18    0  910    1   24
8            5    0   15    6    4    3    9    3  838    2
9            0    0    0    0    0    4    0   29    0  815
```

Calculate Test Dataset

Sourced: COMP5318 Tutorial 5 - Classification I

In [17]:
```
%%time
knn_y_pred = np.empty((len(X_test_pca)))
for i in range(len(label_test)):
    knn_y_pred[i] = calc_knn(X_train_pca, label_train, k, X_test_pc
a[i,:], method)
```

```
CPU times: user 44 s, sys: 8.29 s, total: 52.3 s
Wall time: 53.5 s
```

Confusion Matrix on Test Dataset

In [18]:
```
y_true = label_test
y_pred = knn_y_pred[:y_true.shape[0]]

y_true = pd.Series(y_true, name='Actual')
y_pred = pd.Series(y_pred, name='Predicted')
knn_confusion = pd.crosstab(y_true, y_pred,margins=True)
print(knn_confusion)
```

```
Predicted  0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0   All
Actual
0          162    0    5    4    3    1   12    0    5    0   192
1            0  182    1    1    0    0    0    0    0    0   184
2            5    1  161    4   20    0   15    0    0    0   206
3            8    1    4  170   13    0   10    0    1    0   207
4            1    0   29   13  160    0   17    0    0    0   220
5            0    0    0    0    0  163    0   17    0   10   190
6           36    1   24    4   14    0  107    0    4    0   190
7            0    0    0    0    0    3    0  182    0    7   192
8            0    0    2    1    2    0    2    0  219    1   227
9            0    0    0    0    0    1    0    9    0  182   192
All        212  185  226  197  212  168  163  208  229  200  2000
```

kNN Accuracy on test set

In [19]:
```
np.sum(y_true == y_pred[:y_true.shape[0]])/len(y_true)

knn_accuracy = np.mean(y_pred == y_true)
```

Gaussian Naive Bayes Classifier

Sourced: COMP5318 Tutorial 5 - Classification I

Firstly for each class in our test set, the mean and variance is calculated for each feature.

In [20]:
```
def get_label_stats(X, y):
    labels = set(y)     #Get distinct classes from our test set
    label_stats = dict()    #Declare the dictionary that will hold a list of statistics
    for label in labels:    #Loop to iterate each class
        x_labels = X[y == label]    #Retreive the observations(x) that fall corresponds to the current class
        feature_stats = []      #Declare list to hold the statistic
        for i in range(X.shape[1]): #Iterate each row of our obeservations
            feature_x = x_labels[:,i]   #Select the column
            feature_stats.append([np.mean(feature_x), np.var(feature_x)])   #Append results to our list
        label_stats[label] = feature_stats  #Set the value of our class to the list of statistics
    return label_stats
```

The priors probability is calculated by finding the proportion of each class in the training set.

In [21]:
```
def get_prior(X, y):    #Get the prior probabilty of our classes
    labels = set(y)
    prior = dict()      #Declare dictionary where key is our class and value its proportion
    for label in labels:
        prior[label] = float(len(y[y == label])) / len(y)   #Find number of labels in our dataset and divide by dataset length
    return prior
```

Apply the gaussian formula to get our conditional probabilities

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{x-\mu}{2\sigma^2}}$$

In [22]:
```
def get_p_given_c(Xq, mean, var):
    return  exp( -((Xq-mean)**2 / (2 * var )) ) / (sqrt(2 * pi * var))
```

Calculate the probability of each image for each class

In [23]:
```
def get_posterior(label_stats, row, prior, labels):
    prob = dict()    #Declare dictionaly to hold our probabilities
    for label in labels:    #Interate through our classes
        # prob[label] = prior[label]
        likelihood = prior[label]    #Set the current likelihood as
our prior probability
        for i in range(row.shape[0]):
            mean, var = label_stats[label][i][0], label_stats[label
][i][1]  #For each feature in our image we calculate the proability
of each feature in the current class.
            likelihood *= get_p_given_c(row[i], mean, var)
        prob[label] = likelihood    #Append the posterier to our di
citonry
    return prob
```

Prediction is made by taking the class with the highest posterior probability

In [24]:
```
def get_predictions(posterior):
    predictions = []
    for row in posterior:   #Interate through our posterior diction
ary where key is the label and value is the posterioer.
        predictions.append(max(row, key=row.get))    #Return the key
(class) that has the highest value (posterior)
    return predictions  #Return an arrary of our predicitons
```

The fucntion below incorporates all funcitons above to make the classifier

In [25]:
```
def NB_Classifier(X_train, y_train, X_test):
    #Train Data
    label_stats = get_label_stats(X_train, y_train) #Get statistics
    prior = get_prior(X_train, y_train) #Gets our priors
    posteriors = []
    for X_q in X_test:
        posteriors.append(get_posterior(label_stats, X_q, prior, la
bel_set))    #Get posteriers of each row in our new data set
    y_pred = get_predictions(posteriors)     #Get prediciton for our
new set
    return y_pred
```

kNN Train Model and predict on validation

In [26]:
```
%%time
label_set = set(y_train)
y_pred = NB_Classifier(X_train, y_train, X_val)
```

```
CPU times: user 31.7 s, sys: 214 ms, total: 31.9 s
Wall time: 33.2 s
```

Training Confusion Matrix

In [27]:
```
y_true = pd.Series(y_val, name='Actual')
y_pred = pd.Series(y_pred, name='Predicted')
confusion_matrix = pd.crosstab(y_true, y_pred)
print(confusion_matrix)
nb_train_acc = np.mean(y_pred == y_true)
print("\nAccuracy:", np.mean(y_pred == y_true))
```

```
Predicted    0    1    2    3    4    5    6    7    8    9
Actual
0          668    4   21   75    3   13   59    1   91    2
1            6  762    8   51    2    5    9    0   26    0
2           16    0  572   11  110    8   76    1   73    1
3           25   29   11  711   24    6   28    1   59    0
4            2    1   79   35  627    3   95    0   70    0
5            8    0   11    0    1  679   53  174   20    5
6          124    2   81   24   73   14  447    0  118    0
7            0    0    0    0    0   89    8  814    4   38
8           18    0   10    8   10   19   31   22  765    2
9            0    0    1    0    0   10   16   72   14  735

Accuracy: 0.7533333333333333
```

Predict on Test Dataset

Sourced: COMP5318 Tutorial 5 - Classification I

Prediciton is made on whole training data

In [28]:
```
%%time
nb_y_pred = NB_Classifier(X_train_pca, label_train, X_test_pca)
```

```
CPU times: user 17.6 s, sys: 142 ms, total: 17.8 s
Wall time: 18.5 s
```

Test Confusion Matrix

In [29]:
```
y_true = pd.Series(label_test, name='Actual')
y_pred = pd.Series(nb_y_pred[:2000], name='Predicted')
nb_confusion = pd.crosstab(y_true, y_pred,margins=True)
print(nb_confusion)
print("\nAccuracy:", np.mean(y_pred == y_true))
nb_accuracy = np.mean(y_pred == y_true)
```

```
Predicted    0    1    2    3    4    5    6    7    8    9   All
Actual
0          127    2    3   13    2    3   18    0   24    0   192
1            0  165    1    8    0    1    2    0    7    0   184
2            3    0  125    1   23    0   38    0   16    0   206
3           13    2    1  159    6    3    9    0   14    0   207
4            1    1   21   10  137    0   29    0   21    0   220
5            0    0    2    0    0  129    5   45    9    0   190
6           20    0   14   13   13    2  101    0   27    0   190
7            0    0    0    0    0   21    0  160    0   11   192
8            7    0    4    1    1    7   12    8  186    1   227
9            0    0    0    0    0    5    4   10    7  166   192
All        171  170  171  205  182  171  218  223  311  178  2000

Accuracy: 0.7275
```

Multinominal Logistic Regression Classifier

Sourced: COMP5318 Tutorial 8 - Multinomial Logistic Regression

In multinominal Logistic Regression a softmax function is used to replace the sigmoid funciton.

$$h_w(x) = p(y = c|x; w_1, \ldots, w_c) = \frac{\exp(w_c^T x)}{\sum_{c=1}^{C} \exp(w_c^T x)}$$

In [30]:
```
def softmax(Z):
    exp_z = np.exp(Z)
    return exp_z / exp_z.sum(axis = 1, keepdims = True)
```

Softmax Gradient

$$w_j \leftarrow w_j - \alpha(\sum_{i=1}^{n}(h_w(x^i) - y^i)x^i + \lambda w_j)$$

In [31]:
```
def softmax_grad(X, y, W):
    A = softmax(X.dot(W))      # shape of (N, C)
    id0 = range(X.shape[0])    # number of train data
    A[id0, y] -= 1             # A - Y, shape of (N, C)
    return X.T.dot(A)/X.shape[0] #+ (1/2) * np.sum(W) ** 2
```

Loss

$$\text{loss}(\mathbf{w}) = -l(\mathbf{w}) = -\sum_{k=1}^{C} 1\{y = k\}\Big(\log(\frac{\exp(w_c^T x)}{\sum_{c=1}^{C} \exp(w_c^T x)}\Big)$$

Sourced: http://web.stanford.edu/~jurafsky/slp3/5.pdf

In [32]:
```
def softmax_loss(inputs, targets, weights):
    A = softmax(inputs.dot(weights))
    id0 = range(inputs.shape[0])
    return -np.mean(np.log(A[id0, targets]))
```

From our week 8 tutorial, fitting the multi-nominal logistic regression by batches

In [33]:
```
# building learning fuction using softmax gradient decent
def softmax_fit(inputs, targets, weights, lr, epoches, batch_size):
    tol = 1e-5
    weights_prev = weights.copy()
    loss_hist = [softmax_loss(inputs, targets, weights)] # store hi
story of loss
    N = inputs.shape[0]
    nbatches = int(np.ceil(float(N)/batch_size)) #number of batches
for given batch size
    for epoch in range(epoches):
        mix_ids = np.random.permutation(N) # mix data
        for i in range(nbatches):
            # get the i-th batch
            batch_ids = mix_ids[batch_size*i : min(batch_size*(i+1)
, N) ]
            X_batch, y_batch = inputs[batch_ids], targets[batch_ids
]
            weights -= lr * softmax_grad(X_batch, y_batch, weights)
# update gradient descent
        loss_hist.append(softmax_loss(inputs, targets, weights))
        if np.linalg.norm(weights - weights_prev)/weights.size < to
l:
            break
        weights_prev = weights.copy()
    return weights, loss_hist
```

Predictins are made by taking the highest probability along each row

In [34]:
```
def get_prediction(weights, inputs):
    A = softmax(inputs.dot(weights)) #Use trained weights to find p
robabilites on test data.
    return np.argmax(A, axis = 1)   #Return classes with highest pr
obability
```

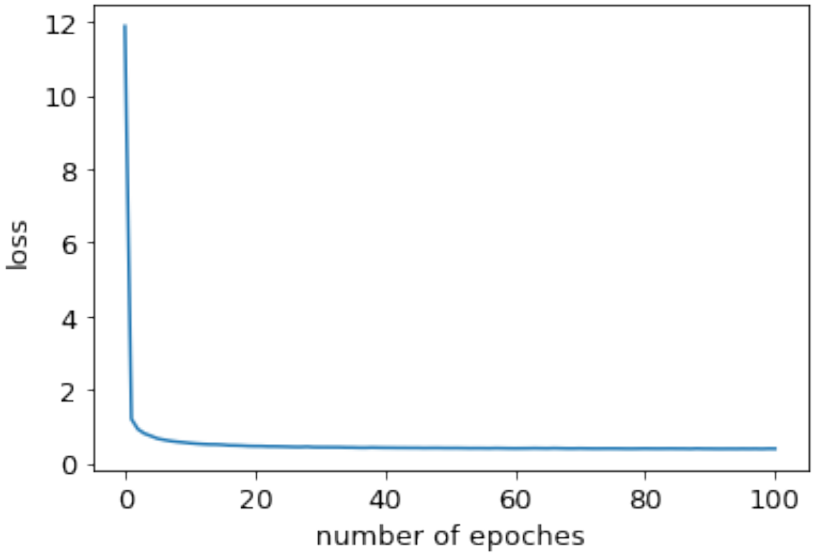
Fit training data

In [35]:
```
%%time
C=len(np.unique(y_val))
weights = np.random.randn(X_train.shape[1], C)
weights, loss_hist = softmax_fit(X_train, y_train, weights, batch_size = 10, epoches = 100, lr = 0.03)
```

```
CPU times: user 33.9 s, sys: 374 ms, total: 34.3 s
Wall time: 18.9 s
```

Plot loss against epoches to find optimal number of epoches

In [36]:
```
plt.plot(loss_hist)
plt.xlabel('number of epoches', fontsize = 13)
plt.ylabel('loss', fontsize = 13)
plt.tick_params(axis='both', which='major', labelsize=13)
plt.show()
```



Can see from graph that there is a sharp elbow point. I have selected 10.

Confusion Matrix of training data agaist validation

In [37]:
```
y_pred = get_prediction(weights,X_val)
y_true = pd.Series(y_val, name='Actual')
y_pred = pd.Series(y_pred, name='Predicted')
confusion_matrix = pd.crosstab(y_true, y_pred)
print(confusion_matrix)
logit_train_acc = np.mean(y_pred == y_true)
print("\nAccuracy:", np.mean(y_pred == y_true))
```

```
Predicted    0    1    2    3    4    5    6    7    8    9
Actual
0          758    5   21   40    3    5   93    1   10    1
1            8  830    6   19    2    0    3    0    1    0
2           13    1  656   16  107    2   65    1    7    0
3           28    9    9  789   28    1   24    2    3    1
4            2    5   72   29  733    0   65    1    4    1
5            1    0    0    1    0  870    1   44   10   24
6          130    2  108   26  104    1  494    1   17    0
7            0    0    0    0    0   25    0  888    3   37
8            5    2    7   10    2    8   11    3  830    7
9            0    0    0    0    0   10    0   19    0  819

Accuracy: 0.8518888888888889
```

Fit whole data and predict on test data

In [38]:
```
%%time
C=len(np.unique(label_train))
weights = np.random.randn(X_train_pca.shape[1], C)
weights, loss_hist = softmax_fit(X_train_pca, label_train, weights,
batch_size = 1, epoches = 10, lr = 0.02)
```

```
CPU times: user 38.4 s, sys: 364 ms, total: 38.8 s
Wall time: 20.6 s
```

Predict Test data

In [39]:
```
%%time
logist_y_pred = get_prediction(weights,X_test_pca)
```

```
CPU times: user 5.8 ms, sys: 2.25 ms, total: 8.05 ms
Wall time: 7.62 ms
```

Confusion Matrix

In [40]:
```
y_true = pd.Series(label_test, name='Actual')
y_pred = pd.Series(logist_y_pred[:2000], name='Predicted')
logist_confusion = pd.crosstab(y_true, y_pred,margins=True)
print(logist_confusion)
print("\nAccuracy:", np.mean(y_pred == y_true))
logist_accuracy = np.mean(y_pred == y_true)
```

```
Predicted    0    1    2    3    4    5    6    7    8    9   All
Actual
0          153    3    5    4    2    0   23    0    2    0   192
1            0  181    0    3    0    0    0    0    0    0   184
2            4    3  148    4   22    0   24    1    0    0   206
3           11    7    3  167    4    1   12    0    2    0   207
4            1    2   31   13  152    0   21    0    0    0   220
5            0    0    0    0    0  179    0    9    1    1   190
6           33    1   20    9   24    0  100    0    3    0   190
7            0    0    0    0    0   12    0  170    0   10   192
8            0    0    0    3    1    3    5    2  212    1   227
9            0    0    0    0    0    3    0    6    0  183   192
All        202  197  207  203  205  198  185  188  220  195  2000

Accuracy: 0.8225
```

Model Anaylsis

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

In [41]:
```
print("kNN Accuracy:", knn_accuracy)
knn_confusion
```

```
kNN Accuracy: 0.844
```

Out[41]:

Predicted Actual	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	All
0	162	0	5	4	3	1	12	0	5	0	192
1	0	182	1	1	0	0	0	0	0	0	184
2	5	1	161	4	20	0	15	0	0	0	206
3	8	1	4	170	13	0	10	0	1	0	207
4	1	0	29	13	160	0	17	0	0	0	220
5	0	0	0	0	0	163	0	17	0	10	190
6	36	1	24	4	14	0	107	0	4	0	190
7	0	0	0	0	0	3	0	182	0	7	192
8	0	0	2	1	2	0	2	0	219	1	227
9	0	0	0	0	0	1	0	9	0	182	192
All	212	185	226	197	212	168	163	208	229	200	2000

In [42]:
```
print("Naive Bayes Accuracy:", nb_accuracy)
nb_confusion
```

```
Naive Bayes Accuracy: 0.7275
```

Out[42]:

Predicted	0	1	2	3	4	5	6	7	8	9	All
Actual											
0	127	2	3	13	2	3	18	0	24	0	192
1	0	165	1	8	0	1	2	0	7	0	184
2	3	0	125	1	23	0	38	0	16	0	206
3	13	2	1	159	6	3	9	0	14	0	207
4	1	1	21	10	137	0	29	0	21	0	220
5	0	0	2	0	0	129	5	45	9	0	190
6	20	0	14	13	13	2	101	0	27	0	190
7	0	0	0	0	0	21	0	160	0	11	192
8	7	0	4	1	1	7	12	8	186	1	227
9	0	0	0	0	0	5	4	10	7	166	192
All	171	170	171	205	182	171	218	223	311	178	2000

In [43]:
```
print("Multinominal Logistic Accuracy:", logist_accuracy)
logist_confusion
```

```
Multinominal Logistic Accuracy: 0.8225
```

Out[43]:

Predicted / Actual	0	1	2	3	4	5	6	7	8	9	All
0	153	3	5	4	2	0	23	0	2	0	192
1	0	181	0	3	0	0	0	0	0	0	184
2	4	3	148	4	22	0	24	1	0	0	206
3	11	7	3	167	4	1	12	0	2	0	207
4	1	2	31	13	152	0	21	0	0	0	220
5	0	0	0	0	0	179	0	9	1	1	190
6	33	1	20	9	24	0	100	0	3	0	190
7	0	0	0	0	0	12	0	170	0	10	192
8	0	0	0	3	1	3	5	2	212	1	227
9	0	0	0	0	0	3	0	6	0	183	192
All	202	197	207	203	205	198	185	188	220	195	2000

In [46]:
```
score_comparison = pd.DataFrame({"Train Accuracy":[kNN_train_acc, nb_train_acc, logit_train_acc],"Test Accuracy":[knn_accuracy,nb_accuracy,logist_accuracy]})
score_comparison.index = ['kNN', 'Naive Bayes', 'Multi-Logistic']
score_comparison
```

Out[46]:

	Train Accuracy	Test Accuracy
kNN	0.856222	0.8440
Naive Bayes	0.753333	0.7275
Multi-Logistic	0.851889	0.8225

Output best prediction to Output folder

In [45]:
```
with h5py.File('Output/predicted_labels.h5','w') as H:
    H.create_dataset('Output',data=y_pred)
```