Mini-project 2
CMPSCI 689 Spring 2015
Due: Tuesday, April 07, in class

Guidelines

Submission. Submit a hardcopy of the report containing all the figures and printouts of code in class. For readability you may attach the code printouts at the end of the solutions. Submissions may be 48 hours late with 50% deduction. Submissions more than 48 hours after the deadline will be given zero. Late submissions should be emailed to the TA as a pdf.

Plagiarism. We might reuse problem set questions from previous years, covered by papers and webpages, we expect the students not to copy, refer to, or look at the solutions in preparing their answers. Since this is a graduate class, we expect students to want to learn and not google for answers.

Collaboration. The homework must be done individually, except where otherwise noted in the assignments. 'Individually' means each student must hand in their own answers, and each student must write their own code in the programming part of the assignment. It is acceptable, however, for students to collaborate in figuring out answers and helping each other solve the problems. We will be assuming that, as participants in a graduate course, you will be taking the responsibility to make sure you personally understand the solution to any work arising from such a collaboration.

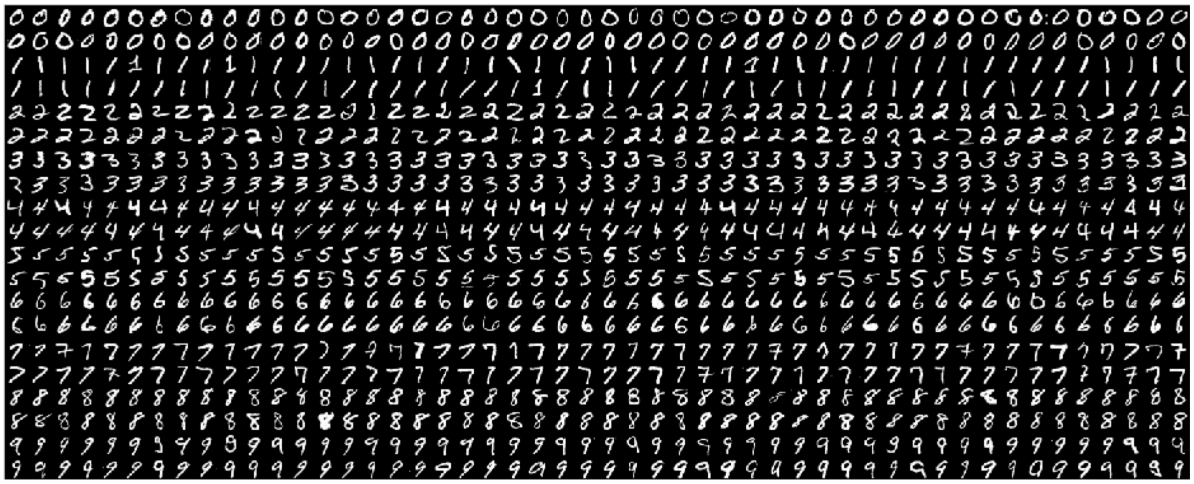
Using other programming languages. All of the starter code is in Matlab which is what we expect you to use. You are free to use other languages such as Octave or Python with the caveat that we won't be able to answer or debug non Matlab questions.

The MNIST dataset

The dataset for all the parts of this homework can be loaded into Matlab by typing `load(digits.mat)`. This is similar to the one used in mini-project 1, but contains examples from all 10 digit classes. In addition, the data is split into `train, val` and `test` sets.

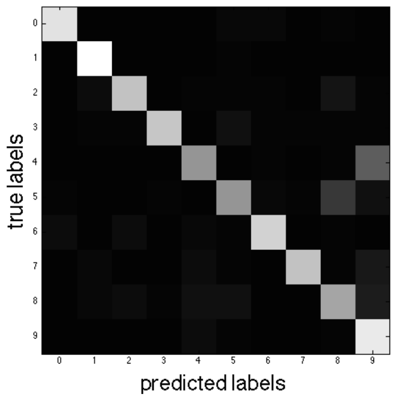
For each split, the features and labels are in variables `x` and `y` respectively. For e.g., `data.train.x` is an array of size 784×1000 containing 1000 digits, one for each column of the matrix. There are 100 digits each from classes $0, 1, \ldots, 9$. Class labels are given by the variable `data.train.y`. The `val` set contains 50 examples from each class, and the `test` set contains 100 examples from each class. This is a subset of the much larger MNIST dataset [1].

You can visualize the dataset using `montageDigits(x)` function. For example, here is the output of `montageDigits(data.train.x)`. *Tip:* `montageDigits(x)` *internally uses the* `montage` *command in Matlab which has a* `size` *option controlling the number of rows and columns of the output.*



Evaluation

For various parts of the homework you will be asked to report the accuracy and confusion matrix. The accuracy is simply the fraction of labels you got right, i.e., $\frac{1}{N}\sum_{k=1}^{N}[y_k = ypred_k]$. The confusion matrix C is a 10×10 array where $C_{ij} = \sum_{k=1}^{N}[y_k = i] \cdot [ypred_k = j]$. The function provided in the codebase called `[acc, conf] = evaluateLabels(y, ypred, display)` returns the accuracy and confusion matrix. If `display=true`, the code also displays the confusion matrix as seen below:



[1] `http://yann.lecun.com/exdb/mnist/`

1 Multiclass to binary reductions

In the previous homework you implemented a binary classifier using `averagedPerceptonTrain` and `preceptronPredict` functions. In this part you will extend these to the multiclass setting using one-vs-one and one-vs-all reductions. The codebase contains an implementation of the above two functions. You are welcome to use your own implementation, but in case you decide to use the provided functions here are the details.

The function `model = averagedPerceptronTrain(x, y, param)` takes as input `x, y` and runs the averaged preceptron training for `param.maxiter` iterations. The output `model` contains the learned weights and some other fields that you can ignore at this point.

The function `[y,a]=preceptronPredict(model, x)` returns the predictions `y` and the activations `a = model.w`^{T}`x`.

The entry code for this part is in the first half of the file `runMulticlassReductions.m`. It loads the data and initializes various parameters.

1.a Multiclass training

Implement a function `model = multiclassTrain(x, y, param);` that takes training data `x,y` and return a multi-class model. Right now you will train linear classifiers using `averagedPerceptronTrain`, but later you will extend this to use non-linear classifiers. Your code will do this by setting the parameter `param.kernel.type='linear'`.

1. *[10 points]* When `param.type='onevsall'` the code should return 10 one-vs-all perceptron classifiers, one for each digit class. You may find cell arrays in Matlab useful for this part.

2. *[10 points]* When `param.type='onevsone'` the code should return $(10 \text{ choose } 2) = 45$ one-vs-one perceptron classifiers, one for each pair of digit classes.

1.b Multiclass prediction

Implement a function `ypred = multiclassPredict(model, x, param);` that returns the class labels for the data `x`.

1. *[5 points]* When `param.type='onevsall'` the code should predict the labels using the one-vs-all prediction scheme, i.e., pick the class which has the highest activation for each data point.

2. *[5 points]* When `param.type='onevsone'` the code should predict the labels using the one-vs-one prediction scheme, i.e., pick the class that wins the most number of times for each data point.

1.c Experiments

On the `train` set learn one-vs-one and one-vs-all classifiers and answer the following questions:

1. *[2 points]* On the `val` set estimate the optimal number of iterations. You may find that the accuracy saturates after a certain number of iterations. Report the optimal value you found for the `onevsone/onevsall` classifiers? Set `param.maxiter` to that value.

2. *[2 points]* What is the `test` accuracy of `onevsone/onevsall` classifier on the dataset?

3. *[2 points]* Show the confusion matrices on the `test` set for `onevsone/onevsall` classifiers. What pair of classes are the most confused for each classifier, i.e., i and j that have the $\max_{ij}\{C_{ij} + C_{ji}\}$?

2 Kernelized perceptrons

In this part you will extend the perceptron training and prediction to use polynomial kernels. A polynomial kernel of degree d is defined as:

$$K^d_{(poly)}(\mathbf{x}, \mathbf{z}) = (a + b\mathbf{x}^T\mathbf{z})^d \tag{1}$$

Here, $a \geq 0, b > 0$ and $d \in \{1, 2, \ldots, \infty\}$ are hyperparamters of the kernel.

2.a Training kernelized perceptrons

[10 points] In this part you will implement a kernelized version of averaged perceptron in the function `model = kernelizedAveragedPerceptronTrain(x, y, param)`. In class we extended the perceptron training to require only dot products, or kernel evaluations between data points. However, it is often more efficient to precompute the kernel values between all pairs of input data to avoid repeated kernel evaluations.

Write a function `K=kernelMatrix(x, z, param)` that returns a matrix with $K_{ij} = \text{kernel}(\mathbf{x}_i, \mathbf{z}_j)$. The kernel is specified by `param.kernel.type`. It is important to implement this efficiently, for example by avoiding for loops. You may find the Matlab function `pdist` useful for this.

The output of the training is a set of weights α for each training example. However, this alone is not sufficient for prediction since it requires dot products with the training data. You may find it convenient to store the training data for each classifier in the model itself, for example in a field such as `model.x`, and the weights α in `model.a`. The entry code for this part is in the second half of the file `runMulticlassReductions.m`.

Tip: When the kernel is linear then the classifier trained using kernelized averaged perceptron should match the results (up to numerical precision) using averaged perceptron. With polynomial kernels you can do this by setting $a = 0$, $b = 1$ *and* $d = 1$.

2.b Predictions using kernelized preceptrons

[5 points] Implement `[y,a] = kernelizedPerceptronPredict(model, x, param)`, which takes the model and features as input and returns predictions and activations. Once again you may first compute the kernel matrix using `K=kernelMatrix(model.x, x, param)` function you implemented earlier and then multiply `model.a` to compute activations.

2.c Going multiclass

[5 points] Integrate the above two functions with `multiclassTrain` and `multiclassPredict` functions you implemented in problem 1 to use kernelized perceptrons as the binary classifier. The code should use perceptrons when `param.kernel.type='linear'` and kernelized perceptrons otherwise. The code passes the parameters of the polynomial kernel in the field `param.kernel.poly`.

2.d Experiments

Train a multiclass classifier using a polynomial kernel with $a = 1, b = 1$ and $d = 4$ as the kernel parameters. This is a degree 4 polynomial kernel. Run the kernelized perceptron for 100 iterations during training and answer the following questions.

1. *[2 points]* What is the `test` accuracy of `onevsone/onevsall` classifier on the dataset?
2. *[2 points]* Show the confusion matrices on the `test` set for `onevsone/onevsall` classifiers. What classes are the most confused?

3 Multiclass naive Bayes classifier

For this part you will train a naive Bayes classifier for multiclass prediction. Assume that for a given data $\mathbf{x}$ the i^{th} feature $x^{(i)}$, for class j, is generated according to a Gaussian distribution with class specific means μ_{ij} and variances σ_{ij}^2, i.e.,

$$P(x^{(i)}|Y=j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} \exp\left(\frac{-||x^{(i)} - \mu_{ij}||^2}{2\sigma_{ij}^2}\right). \tag{2}$$

In addition each class is generated using a multinomial, i.e,

$$P(Y=j) = \theta_j. \tag{3}$$

1. *[4 points]* Write down the log-likelihood of the training data $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_n, y_n)$ in terms of the parameters.
2. *[2 points]* What is the maximum likelihood estimate of the parameter θ_j?
3. *[2 points]* What is the maximum likelihood estimate of the parameter μ_{ij}?
4. *[2 points]* What is the maximum likelihood estimate of the parameter σ_{ij}?

3.a Experiments

You will implement a naive Bayes model for mulitclass digit classification. The entry code for this part is in `runNaiveBayes.m`. Before you implement the model, there is one issue to take care of. Suppose that pixel i is always zero for a class j, then the MLE of the σ_{ij} for that pixel and class will be zero. This leads to probabilities that are zero for any pixel value not equal to zero. Thus a little bit of noise in the pixel can make the total probability zero. One way to deal with this problem is to add a small positive constant τ to the MLE of the σ_{ij} parameter, i.e.,

$$\hat{\sigma}_{ij}^2 \leftarrow \hat{\sigma}_{ij}^2 + \tau. \tag{4}$$

This will assign a small non-zero probability to any noisy pixel you might see in the test data. In terms of Bayesian estimation this is a maximum a-posteriori (MAP) estimate because τ acts as a prior to the σ parameter.

[10 points] Implement the function `model=naiveBayesTrain(x, y,` τ) that estimates the parameters of the naive Bayes model using the smoothing parameter τ.

[5 points] Implement the function `ypred=naiveBayesPredict(model,x)` that predicts the class with the highest joint probability $P(\mathbf{x}, Y=k)$.

Train these models on the `train` set using $\tau = 0.01$ and report accuracies on the `val, test` set.

Tip: Compute log probabilities to avoid numerical underflow.

4 Multiclass logistic regression

We can easily extend the binary logistic regression model to handle multiclass classification. Let's assume we have K different classes, and posterior probability for class k is given by:

$$P(Y = k|X = x) = \frac{\exp(\mathbf{w}_k^T\mathbf{x})}{\sum_{i=1}^{K}\exp(\mathbf{w}_i^T\mathbf{x})}. \tag{5}$$

Our goal is to estimate the weights using gradient ascent. We will also define regularization on the parameters to avoid overfitting and very large weights.

1. *[4 points]* Write down the log conditional likelihood of the labels, $\mathcal{L}(\mathbf{w}_1, ..., \mathbf{w}_K)$ with L_2 regularization on the weights. Use λ as the tradeoff parameter between the $\mathcal{L}$ and regularization, i.e., objective $= \mathcal{L} - \lambda \times$ regularization. Show your steps.
2. *[4 points]* Note that there is not a closed form solution to maximize the log conditional likelihood, $L(\mathbf{w}_1, ..., \mathbf{w}_K)$ with respect to $\mathbf{w}_k$. However, we can still find the solution with gradient ascent by using partial derivatives. Derive the expression for the i^{th} component in the vector gradient of $L(\mathbf{w}_1, ..., \mathbf{w}_K)$ with respect to $\mathbf{w}_k$.
3. *[2 points]* Beginning with the initial weights of 0, write down the update rule for $\mathbf{w}_k$, using η for the step size.
4. *[2 points]* Will the solution converge to a global maximum?
5. *[20 points]* Train a multiclass logistic regression using $\eta = 0.01$ and $\lambda = 0.01$ on the training set and run batch gradient accent for `param.maxtier=100` iterations. Recall that in batch gradients, we sum the gradients across all training examples. In Matlab you can do all the computations efficiently using matrix multiplications. For example, on my laptop the entire training (100 iterations) takes less a second. Report accuracy on the `test` set. The entry code for this part is in `runMulticlassLR.m`.

5 Two layer neural network

Consider a two layer neural network with a hidden layer of H units and K output units, one for each class. Assume that each hidden unit is connected to all the inputs and a bias feature. Use the sigmoid function as the link function.

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{6}$$

The output units combines the hidden units using multi-class logistic regression as in the earlier problem.

1. *[4 points]* Write down the log-likelihood of the labels given all the weights with a L_2 regularization on the weights. Use λ as the tradeoff parameter between the $\mathcal{L}$ and regularization, i.e., objective $=$ $\mathcal{L} - \lambda \times$ regularization. Show your steps.
2. *[4 points]* Derive the equations for gradients for all the weights. The gradients of the second layer weights resemble the earlier problem, while that for the hidden layer can be obtained by back-propagation (or chain-rule of gradients).
3. *[2 points]* Suppose you run gradient ascent, will the solution converge to a global maximum?
4. *[20 points]* Train a two layer network with 100 hidden units using $\eta = 0.001$ and $\lambda = 0.01$ on the training set and run batch gradient ascent for `param.maxtier=1000` iterations. Once again you can implement all the computations using matrix multiplications. My implementation takes about 30 seconds for the entire training. *Tip: you can do entry-wise matrix multiplication using .* operation*. Initialize the weights randomly to small weights using the Matlab function `randn(.,.)*0.01`. Report accuracy on the `test` set. *Tip: You can implement this part with small modifications to the multiclass LR code from the earlier part*. The entry code for this part is in `runMulticlassNN.m`.

Tip: For the previous problem and this one, make sure your objective increases after each iteration.

Checkpoints

To help with debugging here are the outputs of my implementation. The neural network results might vary slightly due to randomness in initialization. You might be able to get better performance by tuning the hyperparameters. It might be tempting to tune them on the test data, but that would be cheating! Only use the validation data for this.

```
>> runMulticlassReductions
OneVsAll:: Perceptron (linear):: Validation accuracy: 75.80%
OneVsAll:: Perceptron (linear):: Test accuracy: 75.70%
OneVsOne:: Perceptron (linear):: Validation accuracy: 86.40%
OneVsOne:: Perceptron (linear):: Test accuracy: 84.30%
OneVsAll:: Perceptron (poly):: Validation accuracy: 88.60%
OneVsAll:: Perceptron (poly):: Test accuracy: 87.30%
OneVsOne:: Perceptron (poly):: Validation accuracy: 88.20%
OneVsOne:: Perceptron (poly):: Test accuracy: 87.00%

>> runNaiveBayes
NaiveBayes:: Validation accuracy: 75.60%
NaiveBayes:: Test accuracy: 76.00%

>> runMulticlassLR
Multiclass LR:: Validation accuracy: 85.40%
Multiclass LR:: Test accuracy: 82.70%

>> runMulticlassNN
Multiclass LR:: Validation accuracy: 87.00%
Multiclass LR:: Test accuracy: 85.20%
```