Mini-project 4: Recognition

CMPSCI 670, Fall 2016, UMass Amherst
Due: November 28, 5:00 PM
Instructor: Subhransu Maji
TA: Tsung-Yu Lin

Guidelines

Submission. Submit a *single pdf document* via moodle that includes your solutions, figures and printouts of code. For readability you may attach the code printouts at the end of the solutions. You could have 24 hours late submission with a 50% mark down. Late submission beyond 24 hours will not be given *any* credits.

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. 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 you will be taking the responsibility to make sure you personally understand the solution to any work arising from such a collaboration.

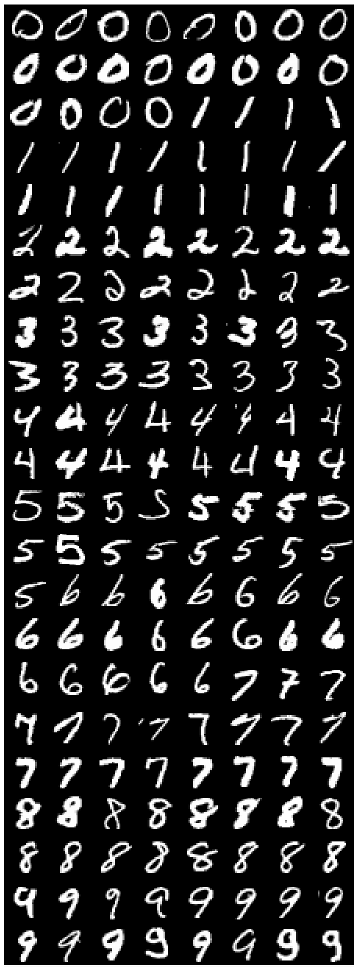
Using other programming languages. All of the starter code is in Matlab which is what we expect you to use. You are free to use other languages such as Octave or Python with the caveat that we won't be able to answer or debug non Matlab questions.

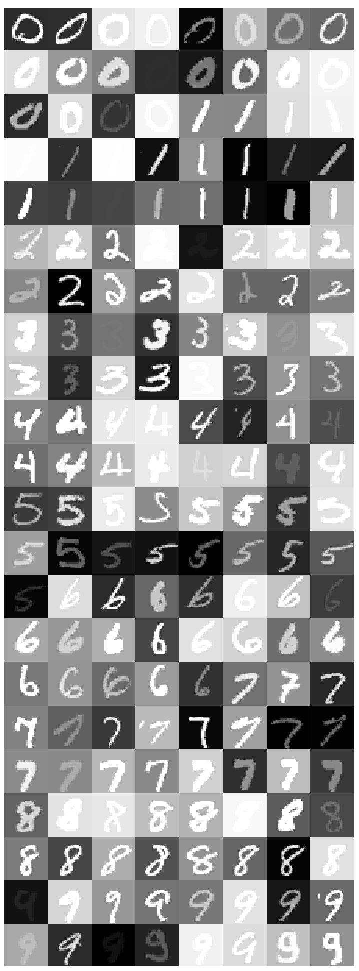
The MNIST dataset

In this homework you will implement several image features and evaluate their performance using logistic-regression classifiers on three variants of the MNIST digits dataset:

- **normal** – the standard MNIST dataset (File `digits-normal.mat`)
- **scaled** – where the pixel values of each image in the normal dataset are replaced as $I \leftarrow a * I + b$ where a and b are random numbers $\in [0, 1]$ (File `digits-scaled.mat`).
- **jittered** – where the image is translated within each image randomly between $[-10, 10]$ pixels both in the horizontal and vertical directions (File `digits-jittered.mat`).

The "normal" dataset can be loaded into Matlab by typing `load(../data/digits-normal.mat)`. This loads a structure called `data` which has the images, labels and the splits in variables `x`, `y`, and `set` respectively. E.g., `data.x` is an array of size $28 \times 28 \times 1 \times 2000$ containing 2000 digits. Class labels are given by the variable `data.y` $\in \{0, 1, \ldots, 9\}$. Thus, pixel `(i,j)` in the the `k'th` training example is `data.x(i,j,1,k)` with label `data.y(k)`. The `data.set` has values $\in \{1, 2, 3\}$ corresponding to train, val, and test splits respectively.


(a) normal


(b) scaled

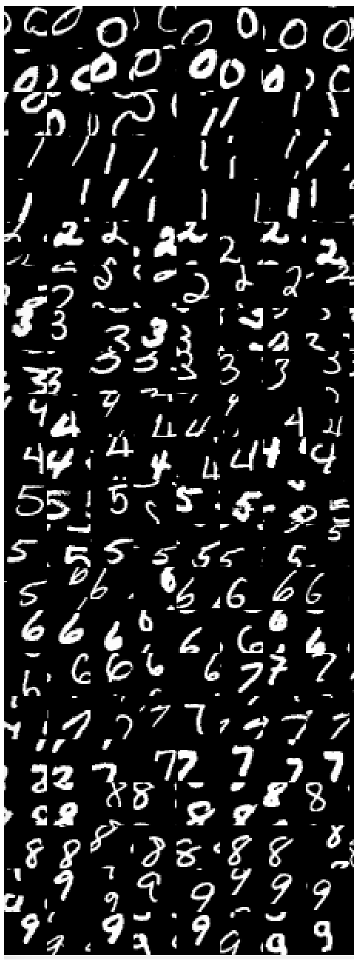

(c) jittered

Figure 1: Example images from different variants of the MNIST dataset. You can visualize the images using `montageDigits` function included in the codebase.

1 Image representations

Implement three different image representations described below.

1.a Pixel features

Implement the function `features = extractDigitFeatures(x, 'pixel')` that takes as input the images `x` and returns their pixel values as a feature. You have to simply reshape the 28×28 pixel image into a vector of size 784×1. The variable `features` is a array of size $784 \times N$ where N is the number of images in `x`.

1.b Histogram of oriented gradients

Implement the function `features = extractDigitFeatures(x, 'hog')` that takes as input the images `x` and returns their histogram of oriented gradients (HoG) as a feature. Recall the steps in the HOG feature computation are:

1. Apply non-linear mapping to input image
2. Compute horizontal and vertical gradients gx and gy at each pixel using derivative operators. Compute the magnitude $m = \sqrt{gx^2 + gy^2}$ and orientation $\theta = \arctan(gy/gx)$ at each pixel.
3. Given a spatial binSize and number of orientation bins (numOri) define a grid of binSize$\times$ binSize across the image and compute the orientation histogram weighted by gradient magnitudes within each grid cell by quantizing the orientation into numOri bins. Concatenate these histograms to obtain an image representation.
4. Local and global normalization steps.

To begin with you can ignore initial non-linear mapping (Step 1) and local normalization (part of Step 4). You will investigate the effect of global normalization in the next section.

1.c Local binary patterns

Implement the function `features = extractDigitFeatures(x, 'lbp')` that takes as input the images `x` and returns a local binary pattern (LBP) histogram for each image. Local binary patterns are a class of highly effective feature representations.

You will implement a simplifed version that maps each 3x3 patch into a number between 0 and 255 as follows. Consider the pixel values in the 3x3 patch. Subtract from each pixel the value at the center. Compute the sign of resulting value at each of the 8 positions (exclude the center). This gives us a 8 bit number which the representation of the 3x3 patch. The scheme is shown in the Figure below.

You can apply this scheme to each pixel in the image (excluding the border) and obtain a number between 0 and 255 for it. The overall LBP histogram is obtained by counting how many times each value appears in the image.

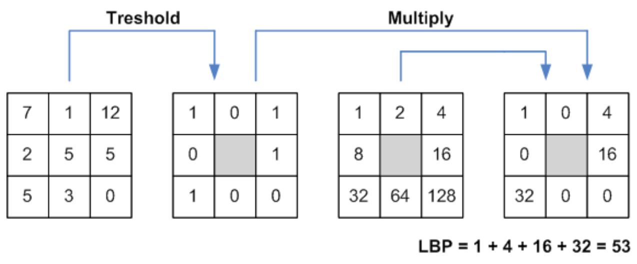


Figure 2: Illustration of the LBP feature computation for a 3x3 patch.

2 Normalization

After you implement these features you may find that the scales of the features can vary dramatically on some datasets for some features. For example, on the "scaled" version of the digits the raw pixel values are randomly perturbed making the pixel representation less robust. Implement the following steps:

1. **Square-root scaling** – replace the value of each feature by its square-root, i.e., $x \leftarrow \sqrt{x}$.
2. **L2-normalization** – scale each feature vector to unit length, i.e., if $\mathbf{x}$ is a vector then the L2-normalized version would be $\mathbf{x} \leftarrow \mathbf{x}/\sqrt{\sum_i \mathbf{x}_i^2}$

The effectiveness of one or both of these steps depends on the dataset and feature types.

3 Classifiers

You can use the logistic-regression classifier included in the code base. Take a look at the function `model = trainModel(x, y)`, which in turn calls the function `model = multiclassLRTrain(x, y, param)`. The `multiclassLRTrain` method runs batch gradients using a fixed learning rate for a given number of iterations to compute the model parameters. You might have to adjust these hyperparamters for different feature types to obtain good performance. Once trained the prediction from this model can be obtained using `ypred = multiclassLRPredict(model,x)`.

4 Training, cross-validation, testing

The entry code for the homework is in the file `testMNIST.m`. It loops over different variants of the dataset, calls the function `extractDigitFeatures` for three different feature types $\in$ {'pixel', 'hog', 'lbp'}, trains a logistic-regression model on the training set and evaluates the model on the validation set. The function `[acc, conf] = evaluateLabels(y, ypred, false)` returns the accuracy and confusion matrix (`conf`) given the true labels `y`, and predictions `ypred`. When the last parameter is `true` it additionally displays the confusion matrix. You may find this helpful for intermediate debugging.

Currently the `extractDigitFeatures` simply returns zero features for all methods and all methods perform at 10% accuracy on all datasets which is the chance performance. You will implement your approach here. Each training step has several hyperparameters, such as normalization method, size of bin for oriented gradient features, as well as classifier-specific ones such as learning rates, regularization term. These should be tuned on the validation set (`testSet=2`). Once you have found the optimal values, you can change the `testSet=3` to evaluate the method on the test set.

The confusion matrix (c) is a 10x10 array where the $c(i, j)$ entry counts how many times a class i got classified as class j. A perfect classifier should have all non-zero values in the diagonal. You can use this to visualize which pair of classes get confused most by different features.

5 Questions [80 points]

Based on your implementation answer the following questions:

1. (14 points) Describe your experiments using pixel features. In particular you should:
   - Report the accuracy on the validation and test set?
   - Describe what normalization steps were effective on what datasets?
   - Include the values of all the hyper parameters that worked best on the validation set.
2. (30 points) Answer all the above questions for HoG features. Don't forget to include the hyperparameters specific to HoG features such as numOri and binSize.
3. (30 points) Answer all the above questions for LBP features.
4. (2 points) Which feature works the best on normal digits? Why?
5. (2 points) Which feature works the best on scaled digits? Why?
6. (2 points) Which feature works the best on jittered digits? Why?

In your submission include your implementation of `extractDigitFeatures`. As a reference my implementation of various features with suitable normalization obtains the following accuracy on the validation set. With careful cross-validation you may be able to outperform these numbers.

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------
dataset           pixel hog   lbp
-------------------------------------
digits-normal.mat 87.00 82.00 74.00
digits-scaled.mat 77.40 79.00 74.00
digits-jitter.mat 20.40 34.00 71.60
```

I don't expect you to match these results exactly, but if your numbers are significantly off that might suggest a bug in the code. If that happens, I expect you to investigate why your results are lower in order to obtain partial credit.

6 Extentions [10 points]

Implement at least one of the following to obtain full points. You may implement multiple for extra credit.

- Train a random forest classifier (an ensemble of decision trees) on this dataset. Investigate what features work best. You can use an exisiting decision tree library to do this. For example, take a look at https://www.mathworks.com/help/stats/classification-trees.html for a Matlab implementation.
- Train a convolution network on this dataset. There is an excellent Matlab implementation here http://www.vlfeat.org/matconvnet/ that you can use to build and experiment with various architectures. To familiarize with the matconvnet take a look at the included tutorials (including the LeNet network on MNIST) which you can follow.
- Implement and report results using a state-of-the-art image representation from literature.