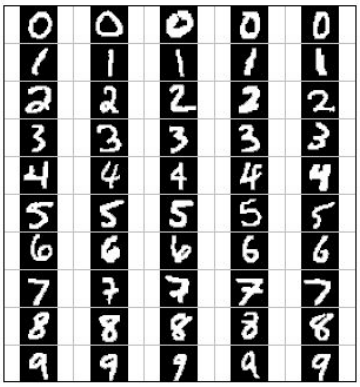
CS 386: Lab Assignment 2

(TAs in charge: Mihir Kulkarni and Anand Dhoot)

Acknowledgement: This lab assignment is based on Project 5: Classification, which is a part of a recent offering of CS188 at UC Berkeley. The code and resources provided here are almost entirely drawn from the Berkeley project. We thank the authors at Berkeley for making their project available to the public.

Optical Character Recognition (OCR) is the task of extracting text from image sources. In this assignment, you will design two **multi-class** classifiers on a set of scanned handwritten digit images: a **Perceptron** classifier, and a large-margin classifier called **MIRA**. Subsequently, you will create your own enchanced feature extractor, which will supplement the simple features provided to further improve the performance of your classifiers.



(Image source: https://inst.eecs.berkeley.edu/~cs188/sp08/projects/project8/webpage/img2.gif.)

The data set on which you will run your classifiers is a collection of handwritten numerical digits (0–9). OCR on digits is a very commercially useful technology, similar to the technique used by the US post office to route mail by zip codes. There are systems that can perform with over 99% classification accuracy (see LeNet-5 for an example system in action). Details about your tasks, and the files you must submit as a part of this assignment, are mentioned in the sections below.

Code

The base code for this assignment is available in this zip file. You need to use these files from the `classification` directory.

File Name	Description
classificationMethod.py	Abstract super class for the classifiers you will write. You **should** read this file carefully to see how the infrastructure is set up. Do **not** edit this file.
createData.sh	Bash script that runs the perceptron classifier on several different sizes of the training set.
createGraphIterations.gnuplot	Gnuplot script to plot accuracy versus number of training examples seen.
createGraphTraining.gnuplot	Gnuplot script to plot accuracy versus size of the training set.
data.zip	Contains the training, validation and test data sets.
dataClassifier.py	The wrapper code that will call your classifiers. You will also write your enhanced feature extractor here.
graph_answers.txt	Blank text file for you to enter answers from Task 2.
mira.py	The file where you will write your MIRA classifier.
perceptron.py	The file where you will write your perceptron classifier.
samples.py	I/O code to read in the classification data. Do **not** edit this file.
util.py	Code defining some useful tools, that will save you a lot of time. Do **not** edit this file.

Task 0: Understanding the Multi-class Perceptron (Ungraded)

A Perceptron keeps a weight vector w^y corresponding to each class y (y is a suffix, not an exponent). Given a feature list f (a vector with the same number of dimensions as the weight vectors), the perceptron computes the class y whose weight vector is most similar to the input vector f. Formally, given a feature vector f (in our case, a map from pixel locations to indicators of whether they are on), we score each class y with:

$$\text{score}(f, y) = \sum_i f_i w_i^y,$$

where i iterates over the dimensions in the data. Then we choose the class with highest score as the predicted label for data instance f. In the code, we will represent w^y as a `Counter` (defined in the file `util.py`).

Learning the weights of the Perceptron

In the basic multi-class Perceptron, we scan over the training data one instance at a time. When we come to an instance (f, y), we find the label with highest score: $y' = arg\max_{y''} score(f, y'')$, breaking ties arbitrarily. We compare y' to the true label y. If $y' = y$, we have correctly classified the instance, and we do nothing. Otherwise, we predicted y' when we should have predicted y. That means that w^y has scored f lower, and/or $w^{y'}$ has scored f higher, than what would have been ideal. To avert this error in the future, we update these two weight vectors accordingly:

$w^y = w^y + f$, and

$w^{y'} = w^{y'} - f$.

Task 1: Implementing the Multi-class Perceptron (4 marks)

In this task, you have to implement the multi-class Perceptron classifier. Fill out code in the `train()` function at the location indicated in `perceptron.py`. Using the addition, subtraction, and multiplication functionality of the `Counter` class in `util.py`, the perceptron updates should be relatively easy to code. Certain implementation issues have been taken care of for you in `perceptron.py`, such as the following.

- Iterating over the training data.
- Setting up the `weights` data structure. This can be accessed using `self.weights` in the `train()` method.
- Predictions for each label/class are performed by a separate weight vector, as described above. Therefore, each legal label needs its own `Counter` of weights.

Run your code with the following command.

```
python dataClassifier.py -c perceptron -t 1000 -s 1000
```

This will print out a host of details, such as the classifier that is being trained, the training set size, if enhanced features are being used (more on this in Task 5), etc. After this, the classifier gets trained for a default of 3 iterations. `dataClassifier.py` would then print the accuracy of your model on the validation and test data sets.

The `-t` option passed to the Perceptron training code specifies the number of training data points to read in from memory, while the `-s` option specifies the sizes of the validation and test sets (the train, validation, and test data sets are mutually independent). Since iterating over 1000 points may take some time, you can use 100 data points instead while developing and testing your code. However, note that all our evaluations will involve train, validation, and test data sets with 1000 points.

Evaluation: Your classifier will be evaluated by for its accuracy on the test set after the Perceptron has been trained for the default 3 iterations. You will be awarded 4 marks if the accuracy exceeds 65%, otherwise 3 marks if accuracy exceeds 55%, otherwise 2 marks if accuracy exceeds 50%, otherwise 0 marks.

Task 2: Analysing the Perceptron's performance (4 marks)

One of the problems with the perceptron is that its performance is sensitive to several practical details, such as how many iterations you train it for, the order you use for the training examples, and how many data points you use. The current code uses a default value of 3 training iterations. You can change the number of iterations for the perceptron with the `-i iterations` option. In practice, you would use the performance on the validation set to figure out when to stop training, but you don't need to implement this stopping criterion for this task.

Instead, you need to analyse and comment on how the validation accuracy changes (1) as the total number of data points seen during training change, and (2) as the total size of training set changes. See details below.

1. **Variation with number of training data points 'seen':** In this sub-task, you need to plot the validation accuracy as increasing numbers of data points are 'seen' (that is, a point is examined to determine whether an update needs to be made to the Perceptron, or the point is already correctly classfied, and so no update is needed) by the classifier.

   You have been provided helper code for this purpose: the `train()` function in `perceptron.py` writes `<examples seen>,<validation accuracy>` as a comma-separated pair into the file `perceptronIterations.csv` 2 times in every iteration. The plotting script `createGraphIterations.gnuplot` uses this csv file to create the required plot and save it in a file by name `plot_iterations.png`. You can modify either of the two code files mentioned above to modify the plot being generated. Feel free to experiment with the various switches and write a description of your observations and a description of the nature of the plot in the file `graph_answers.txt` in plain text.

   Below are commands to generate the required graph.

   - `python dataClassifier.py -c perceptron -t 1000 -s 1000 -v`
   - `gnuplot createGraphIterations.gnuplot`

2. **Variation with training set size in each iteration:** Here, you will see the influence of training set size on the accuracy of validation set, when trained for the same number of iterations.

   Every time `dataClassifier.py` is run, it writes a comma-seperated pair of numbers—<training set size>,<accuracy>—for the validation and the test sets to files `perceptron_valid.csv` and `perceptron_valid.csv`. Use `createData.sh` (supplied in the zip file) to run the perceptron training on 100, 200, 300, ..., 1000 examples. Create a plot named `plot_training.png` by running `createGraphTraining.gnuplot`. Below are commands to generate the required graph.

   - `./createData.sh`
   - `gnuplot createGraphTraining.gnuplot`

   Append your observations and description of the plot obtained in the file `graph_answers.txt` (to which you had already written in the previous sub-task). Additionally, answer the following questions.

   1. With a 1000 data points, is your test accuracy close to 100%? Why (if it is) and why not (if it is not)?
   2. Imagine a point on the x axis with 0 training points: that is, a classifier that must make predictions based on no training data at all! How would such a classifier make predictions? On this data set, what would be the accuracy of such a classifier?

   Note: The bash script to generate the data for the graph might take a long time to run. You are advised to proceed to subsequent tasks while the script is running.

Evaluation: Sub-tasks 1 and 2 will each be evaluated for a total of 4 marks, with the plots and the corresponding explanations and answers all taken into consideration.

Task 3: Understanding MIRA (Ungraded)

MIRA is an on-line learner which is closely related to both the Support Vector Machine and the Perceptron classifiers. Similar to a multi-class Perceptron classifier, multi-class MIRA also keeps a weight vector w^y for each label y. We also scan over the data, one instance at a time. As before, when we come to an instance (f, y), we find the label with highest score:

$$y' = arg\max_{y''} score(f, y'').$$

We compare y' to the true label y. If $y' = y$, we have classified the instance correctly, and so we do nothing further. Otherwise, we predicted y' incorrectly, instead of y. The difference between MIRA and Perceptron is in the update rule. Under MIRA, we update the weight vectors of labels with a variable step size:

$w^y = w^y + \tau f$, and

$w^{y'} = w^{y'} - \tau f$,

where

$$\tau = \min(C, \frac{(w^{y'} - w^y) \cdot f + 1}{2||f||_2^2}),$$

with C being a positive constant that is tuned. It exceeds the scope of this assignment to delve into the semantics and the derivation of τ. The interested student is referred to the original paper on MIRA. For the moment, it suffices to remark that MIRA is designed learn linear separators with a larger **margin** (separating region between points from different classes) than can be guaranteed by Perceptron.

Task 4: Training using MIRA (3 marks)

A skeleton implementation of the MIRA classifier is provided for you in `mira.py`. Implement `trainAndTune()` in `mira.py`. This method should train a MIRA classifier using each value of *C* in `Cgrid`. Evaluate accuracy on the held-out validation set for each *C* value and choose the one with the highest validation accuracy. In case of ties, prefer the *lowest* value of *C*. Test your MIRA implementation by running this command.

```
python dataClassifier.py -c mira --autotune -t 1000 -s 1000
```

Here are some useful suggestions for this task.

- You can use your Perceptron code from Task 1 with relevant edits.
- Pass through the data `self.max_iterations` times during training.
- Store the weights learned using the best value of *C* at the end in `self.weights`, so that these weights can be used to test your classifier.
- To use a fixed value of *C=0.001*, remove the `--autotune` option from the command above.
- It might save some debugging time if the +1 term in the definition of τ (see Task 3) is implemented as +1.0, due to division truncation of integer arguments. Depending on how you implement this, it might not matter.

Evaluation: (1) Your implementation of MIRA will be assessed: if it is correct and it achieves an accuracy exceeding 65% on the test set, you will receive 2 marks. If not, your marks (1 or 0) will depend on the degree of correctness of your implementation and the accuracy. (2) An additional 1 mark is reserved for the successful running of your auto-tuning function. This requires you to check validation accuracy scores for each choice of the parameter *C* and to choose the weights corresponding to the best such *C* obtained.

Task 5: Feature Design (4 marks)

Building classifiers is only a small part of getting a good system working for a task. Indeed, the main difference between a good classification system and a bad one is usually not the classifier itself (e.g. Perceptron or MIRA), but rather the quality of the features used. So far, we have only used the simplest possible features: the identity of each pixel (being on/off).

To increase your classifier's accuracy further, you will need to extract more useful features from the data. The `EnhancedFeatureExtractorDigit()` in `dataClassifier.py` is your new playground. When analysing your classifiers' results, you should look at some of your errors and look for characteristics of the input that would give the classifier useful information about the label. You can add code to the `analysis()` function in `dataClassifier.py` to inspect what your classifier is doing.

As a concrete illustration, consider the digit data you have used. In each training point (a binary matrix corresponding to an image), consider the number of separate, connected regions of white pixels. This quantity would vary by digit type. 1, 2, 3, 5, and 7 tend to mostly have one contiguous region of white space, while the loops in 0, 6, 8, and 9 create more. The number of white regions in a 4 depends on the writer. This is an example of a feature that is not directly available to the classifier from the per-pixel information.

If your feature extractor adds new features such as the quantity described above (think of others, too!), the classifier will be able exploit them. Note that some features may require non-trivial computation to extract, so write efficient and correct code. Add your new binary features for the digit dataset in the `EnhancedFeatureExtractorDigit()` function. Note that you can encode a feature which takes 3 values [1,2,3] by using 3 binary features, of which only one is on for any given instance, indicating which of the original feature values has occured.

We will test your classifier with the following command.

```
python dataClassifier.py -c perceptron -f -t 1000 -s 1000
```

Evaluation: If your new features result in any improvement in accuracy over your Perceptron classifier with the basic features, you will receive 2 marks. If your new features give you a test accuracy exceeding 80% using the command above, you will get 2 additional marks. (Note: we don't expect breaching the 80% mark to be easy!).

Submission

You are not done yet! Place all files in which you have written code in or modified in a directory named your roll number (say 12345678). Tar and Gzip the directory to produce a single compressed file (12345678.tar.gz). It must contain the following files:

1. `perceptron.py`
2. `graph_answers.txt`
3. `mira.py`
4. `dataClassifier.py`
5. `createData.sh`
6. `createGraphIterations.gnuplot`
7. `createGraphTraining.gnuplot`
8. `plot_iterations.png`
9. `plot_training.png`

Submit this compressed file on Moodle, under Lab Assignment 2.