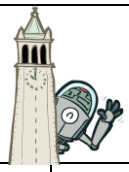
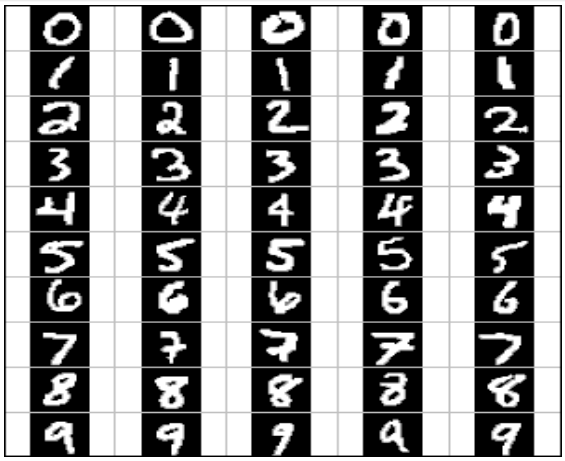


CMPT310
Classification Project:

Perceptron vs Naïve Bayes vs Mira .

Table of Contents



Introduction

In this project, you will design three classifiers: a naïve Bayes Classifier, a perceptron classifier, and a large-margin (MIRA) classifier. You will test these classifiers on a set of scanned handwritten digit images. Even with simple features, your classifiers will be able to do quite well on these tasks when given enough training data.

Optical character recognition (OCR) is the task of extracting text from image sources. The data set on which you will run your classifiers is a collection of handwritten numerical digits (0-9). This is a very commercially developed technology used in many domains, like the technique used by the US post office to route mail by zip codes or detecting vehicles' plate number and so on. There are systems that can perform with over 99% classification accuracy these days.

The code for this project includes the following files and data, available as a zip file.

Data file

digitdata	Data file, including the digit images

Files you will edit

naiveBayes.py	The location where you will write your naive Bayes classifier.
perceptron.py	The location where you will write your perceptron classifier.
mira.py	The location where you will write your MIRA classifier.
dataClassifier.py	The wrapper code that will call your classifiers. You will also write your enhanced feature extractor here. You will also use this code to analyze the behavior of your classifier.
answers.py	Answer to Question 2 goes here.

Files you should read but NOT edit

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.)
samples.py	I/O code to read in the classification data.
util.py	Code defining some useful tools. You may be familiar with some of these by now, and they will save you a lot of time.
mostFrequent.py	A simple baseline classifier that just labels every instance as the most frequent class.

Files to Edit and Submit: You will fill in portions of perceptron.py, mira.py, answers.py, and dataClassifier.py (only) during the assignment, and submit them. You should submit only these files in your submission as a zipped folder. Please *do not* change the other files in this distribution or submit any of the original files other than this file.

Evaluation: Your code will be graded for technical correctness. Please *do not* change the names of any provided functions or classes within the code. You will not need to add any new file or functions or classes. So it is very important to stay in this boundary to make the grading standard fair to everyone. Grading is out of 100 for total mark. The breakup of the grades per sections follows bellow.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the T.A.s for help. Office hours, and the discussion forum are there for your support.

Spoiler Alert!!: Please be careful not to post spoilers.

Question 1 (30 points): Perceptron

A skeleton implementation of a perceptron classifier is provided for you in `perceptron.py`. In this part, you will fill in the `train` function.

Unlike the naive Bayes classifier, a perceptron does not use probabilities to make its decisions. Instead, it keeps a weight vector w^y of each class y (y is an identifier, not an exponent). Given a feature list f, 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 with:

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

Then we choose the class with highest score as the predicted label for that data instance. In the code, we will represent w^y as a `Counter`.

Learning weights

In the basic multi-class perceptron, we scan over the data, one instance at a time. When we come to an instance (f, y), we find the label with highest score:

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

We compare y' to the true label y. If $y' == y$, we've gotten the instance correct, so we do nothing. Otherwise, we guessed y' but we should have guessed y. That means that w^y should have scored f higher, and $w^{y'}$ should have scored f lower, in order to prevent this error in the future. We update these two weight vectors accordingly:

$$w^y = w^y + f$$

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

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 handling iterations over the training data and ordering the update trials. Furthermore, the code sets up the `weights` data structure for you. Each legal label needs its own `Counter` full of weights.

Question

Fill in the `missing parts specified` in `perceptron.py`. Run your code in PyCharm (with Python 3.9) using the run configuration named Percept (-c perceptron -d digits -t 1000 -w -1 3 -2 6).

Hints and observations:

- The command above should yield validation accuracies around 82% and test accuracy about 75% (with the default 3 iterations). These ranges are wide because the perceptron is a lot more sensitive to the specific choice of tie-breaking than naive Bayes.
- 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, and the order you use for the training examples (in practice, using a randomized order works better than a fixed order. There is a commented-out code in train() function which you can use for playing with randomized data). The current code uses a default value of 3 training iterations and the order of processing the samples is fixed. You can change the number of iterations for the perceptron with the `-i iterations` option. Try different numbers of iterations and see how it influences performance. 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 assignment.

Question 2 (10 point): Perceptron Analysis

Visualizing weights

Perceptron classifiers, and other discriminative methods, are often criticized because the parameters they learn are hard to interpret. To see a demonstration of this issue, we can write a function to find features that are characteristic of one class. (Note that, because of the way Perceptron is trained, it is not as crucial to find odds ratios.)

Question

Fill in `findHighWeightFeatures(self, label)` in `perceptron.py`. It should return a list of the 100 features with highest weight for that label. You can display the 100 pixels with the largest weights using the configuration Percept_w_k2.5

Use this command to look at the weights' image in the output terminal, and answer the following question. Which of the following sequence of weights is most representative of the perceptron?

(a)

(b) 

Answer the question `answers.py` in the method `q2`, returning either 'a' or 'b'.

Question 3 (20 points): MIRA

A skeleton implementation of the MIRA classifier is provided for you in `mira.py`. MIRA is an online learner which is closely related to both the support vector machine and perceptron classifiers. You will fill in the missing sections in `mira.py`.

Note: If you have done Question 2 above, you should reproduce the same code for `findHighWeightFeatures(self, label)` this time in `Mira.py`.

Theory

Similar to a multi-class perceptron classifier, multi-class MIRA classifier also keeps a weight vector w^y for each label y. We also scan over the data, one instance at a time. When we come to an instance f, we create a list of pairs (f, y''), y'' being the label MIRA computes for f. Finally, we find the label y' which is the highest in this list:

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

We compare y' to the true label y. If $y' == y$, we've gotten the instance correct, and we do nothing. Otherwise, we guessed y' but we should have guessed y. Unlike the perceptron, we update the weight vectors of these labels with a variable step size:

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

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

where $\tau \geq 0$ is chosen such that it minimizes

$$\min_{w'} \frac{1}{2} \sum_c ||(w')^c - w^c||_2^2$$

subject to the condition that $(w')^y \cdot f \geq (w')^{y'} \cdot f + 1$

which is equivalent to

$$\min_{\tau} ||\tau f||_2^2 \text{ subject to } \tau \geq \frac{(w^{y'} - w^y).f + 1}{2||f||_2^2} \text{ and } \tau \geq 0$$

Note that $w^{y'} \cdot f \geq w^y \cdot f$, so the condition $\tau \geq 0$ is always true given $\tau \geq \frac{(w^{y'} - w^y).f + 1}{2||f||_2^2}$ Solving this simple problem, we then have

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

However, we would like to cap the maximum possible value of τ by a positive constant C, which leads us to

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

Question

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* and choose the *C* with the highest validation accuracy. In case of ties, prefer the *lowest* value of *C*. Test your MIRA implementation by running configuration MIRA. This configuration runs MIRA with fixed C=0.001. You must also be able to run tuning using the sequence of C's (as outlined in the code and report on each C's accuracy. Try to interpret the accuracy's possible dependence on C value. To try your code in autotuning mode run the configuration MIRA_autotune.

Hints and observations:

- 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.
- The same code for returning high odds features in your perceptron implementation should also work for MIRA if you're curious what your classifier is learning.

Question 4 (15+5 + 5 points): Naïve Bayes

A skeleton implementation of the Naïve Bayes classifier is provided for you in `naiveBayes.py`. We covered the theory of Naïve Bayes statistical classifier in detail in class. The provided code for this classifier is more complete, compared the other classifiers in this assignment. You will only need to fill a small part in the *`trainAndTune`* function, and in *`calculateLogJointProbabilities`*

-15 points is for *trainAndTune* and its related function *calculateLogJointProbabilities*. To run your code, use the configuration NB_k2.5. This configuration use Laplace smoothening with factor k=2.5

-5 point for implementing *findHighOddsFeatures*(): A good way to see what features are prominent in one label vs another, is to implement the function *findHighOddsFeatures* (label1, label2). This function returns a list of top 100 features prominent in label 1 compared to label2. This can help in theory to improve on our algorithm further. To test this part, you can run configuration NB_k2.5_Odd. You can also choose the labels to compare by adding the command line as follows: -o -1 3 -2 6, which means use label1=3 and label2=6 in *findHighOddsFeatures*.

-5 point for autotuning: We also going to run autotuning on K with running NB_Autotune. In this mode you should report accuracy for each k, to see what value of k gives the best accuracy.

Question 5 (8+7 points): Digit Feature Design

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 vs. naive Bayes), but rather the quality of the features used. So far, we have 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 analyzing 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. For instance, in the digit data, consider the number of separate, connected regions of white pixels, which varies by digit type. 1, 2, 3, 5, 7 tend to have one contiguous region of white space while the loops in 6, 8, 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 that encode these properties, the classifier will be able exploit them. Note that some features may require non-trivial computation to extract, so write efficient and correct code.

Note: You will be working with digits, so make sure you are using DIGIT_DATUM_WIDTH and DIGIT_DATUM_HEIGHT, for image width and height.

Question

Add new binary features for the digit dataset in the *`EnhancedFeatureExtractorDigit`* function. Note that you can encode a feature which for example taking 3 values [1,2,3] by using 3 binary features, of which only one is on at a time, to indicate which of the three possibilities you have. In theory, features aren't conditionally independent as naive Bayes requires, but your classifier can still work well in practice. We will test your classifier with running the configuration NB_Feature.

With the basic features (without the `-f` option), your optimal choice of smoothing parameter should yield 82% on the validation set with a test performance of 78%. You will receive 8 points for implementing new feature(s) which yield an improvement of at least 1 point over the basic feature case. You will receive 7 additional points if your new feature(s) give you a test performance improvement of %84 or higher on the test data.

Submission

Make sure you submit only those files specified at the beginning of this file.