Supervised Learning (COMP0078) – Coursework 2

Mark Herbster

Due : 11 January 2021.
CW2_20v3

Submission

You should produce a report about your results. You will not only be assessed on the **correctness/quality** of your answers but also on **clarity of presentation**. Additionally make sure that your code is *well commented*. Please submit on moodle i) your report as well as a ii) zip file with your source code. Regarding the use of libraries, you should implement regression using matrix algebra directly. Likewise any basic machine learning routines such as cross-validation should be implemented directly. Otherwise libraries are okay.

Questions please e-mail `sl-support@cs.ucl.ac.uk` .

1 PART I [50%]

1.1 Kernel perceptron (Handwritten Digit Classification)

Introduction: In this exercise you will train a classifier to recognize hand written digits. The task is quasi-realistic and you will (perhaps) encounter difficulties in working with a moderately large dataset which you would not encounter on a "toy" problem.

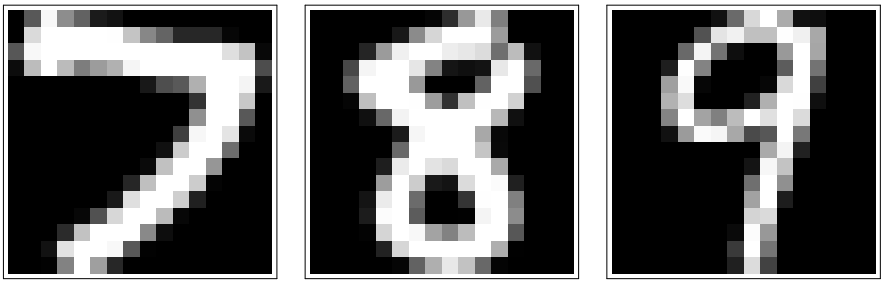

Figure 1: Scanned Digits

You may already be familiar with the perceptron, this exercise generalizes the perceptron in two ways, first we generalize the perceptron to use *kernel* functions so that we may generate a nonlinear separating surface and second, we generalize the perceptron into a majority network of perceptrons so that instead of separating only two classes we may separate k classes.

Adding a kernel: The *kernel* allows one to map the data to a higher dimensional space as we did with basis functions so that class of functions learned is larger than simply linear functions. We will consider a single type of kernel, the polynomial $K_d(\boldsymbol{p}, \boldsymbol{q}) = (\boldsymbol{p} \cdot \boldsymbol{q})^d$ which is parameterized by a positive integer d controlling the dimension of the polynomial.

Training and testing the kernel perceptron: The algorithm is *online* that is the algorithms operate on a single example $(\boldsymbol{x}_t, y_t)$ at a time. As may be observed from the update equation a single kernel

function $K(\boldsymbol{x}_t, \cdot)$ is added for each example scaled by the term α_t (may be zero). In online training we repeatedly cycle through the training set; each cycle is known as an *epoch.* When the classifier is no longer changing when we cycle thru the training set, we say that it has converged. It may be the case for some datasets that the classifier never converges or it may be the case that the classifier will *generalize* better if not trained to convergence, for this exercise I leave the choice to you to decide how many epochs to train a particular classifier (alternately you may research and choose a method for converting an online algorithm to a batch algorithm and use that conversion method). The algorithm given in the table correctly describes training for a single pass thru the data (*1st epoch*). The algorithm is still correct for multiple epochs, however, explicit notation is not given. Rather, latter epochs (additional passes thru the data) is represented by repeating the dataset with the $\boldsymbol{x}_i$'s renumbered. I.e., suppose we have a 40 element training set $\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ..., (\boldsymbol{x}_{40}, y_{40})\}$ to model additional epochs simply extend the data by duplication, hence an m epoch dataset is

$$\underbrace{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_{40}, y_{40})}_{\text{epoch 1}}, \underbrace{(\boldsymbol{x}_{41}, y_{41}), \ldots, (\boldsymbol{x}_{80}, y_{80})}_{\text{epoch 2}}, \ldots, \underbrace{(\boldsymbol{x}_{(m-1)\times 40+1}, y_{(m-1)\times 40+1}), \ldots, (\boldsymbol{x}_{(m-1)\times 40+40}, y_{(m-1)\times 40+40})}_{\text{epoch m}}$$

where $\boldsymbol{x}_1 = \boldsymbol{x}_{41} = \boldsymbol{x}_{81} = \ldots = \boldsymbol{x}_{(m-1)\times 40+1}$, etc. Testing is performed as follows, once we have trained a classifier $\boldsymbol{w}$ on the training set, we simply use the trained classifier with only the *prediction* step for each example in test set. It is a mistake when ever the prediction $\hat{y}_t$ does not match the desired output y_t, thus the test error is simply the number of mistakes divided by test set size. Remember in testing the *update* step is never performed.

	Two Class Kernel Perceptron (training)
Input:	$\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\} \in (\Re^n, \{-1, +1\})^m$
Initialization:	$\boldsymbol{w}_0 = \mathbf{0}$ $(\alpha_0 = 0)$
Prediction:	Upon receiving the tth instance $\boldsymbol{x}_t$, predict $\hat{y}_t = \text{sign}(\boldsymbol{w}_t(\boldsymbol{x}_t)) = \text{sign}(\sum_{i=0}^{t-1} \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_t))$
Update:	`if` $\hat{y}_t = y_t$ `then` $\alpha_t = 0$ `else` $\alpha_t = y_t$ $\mathbf{w}_{t+1}(\cdot) = \mathbf{w}_t(\cdot) + \alpha_t K(\boldsymbol{x}_t, \cdot)$

Generalizing to k classes: Design a method (or research a method) to generalise your two-class classifier to k classes. The method should return a vector $\boldsymbol{\kappa} \in \Re^k$ where κ_i is the "confidence" in label i; then you should predict either with either a label that maximises confidence or alternately a randomised scheme.

I'm providing you with mathematica code for a 3-classifier and a demonstration on a small subset of the data. First, however, my mathematica implementation is flawed and is relatively inefficient for large datasets. One part of your goal is improve my code so that it can work on a larger datasets, the mathematical logic of the algorithm will not change, however either and/or the program logic and data structures will need to change. Also, I suspect that it will be considerable easier to implement sufficiently fast code in Matlab (or the language of your choice) rather than Mathematica.

Files: From `http://www0.cs.ucl.ac.uk/staff/M.Herbster/SL/misc/`, you will find files relevant to this assignment. These are,

`poorCodeDemoDig.nb`	demo mathematica code
`dtrain123.dat`	mini training set with only digits 1,2,3 (329 records)
`dtest123.dat`	mini testing set with only digits 1,2,3 (456 records)
`zipcombo.dat`	full data set with all digits (9298 records)

each of the data files consists of records (lines) each record (line) contains 257 values, the first value is the digit, the remaining 256 values represent a 16×16 matrix of grey values scaled between -1 and 1. In attempting to understand the algorithms you may find it valuable to study the mathematica code. However, remember the demo code is partial (it does not address model selection, and though less efficient implementations are possible it is not particularly efficient.) Improving the code may require thought and observation of behaviour on the given data, there are many distinct types of implementations for the kernel perceptron.

Experimental Protocol: Your report on the main results should contain the following (errors reported should be percentages not raw totals):

1. Basic Results: Perform 20 runs for $d = 1, \ldots, 7$ each run should randomly split `zipcombo` into 80% train and 20% test. Report the mean test and train errors as well as well as standard deviations. Thus your data table, here, will be 2×7 with each "cell" containing a mean$\pm$std.

2. Cross-validation: Perform 20 runs : when using the 80% training data split from within to perform 5-fold cross-validation to select the "best" parameter d^* then retrain on full 80% training set using d^* and then record the test errors on the remaining 20%. Thus you will find 20 d^* and 20 test errors. Your final result will consist of a mean test error$\pm$std and a mean d^* with std.

3. Confusion matrix: Perform 20 runs : when using the 80% training data split that further to perform 5-fold cross-validation to select the "best" parameter d^* retrain on the 80% training using d^* and produce a *confusion matrix*. Here the goal is to find "confusions" thus if the true label was "7" and "2" was predicted then a "mistake" should recorded for "(7,2)"; the final output will be a 10×10 matrix where each cell contains a confusion error and its standard deviation. Note the diagonal will be 0.

4. Within the dataset relative to your experiments there will be five hardest to predict correctly "data items." Print out the visualisation of these five digits along with their labels. Is it surprising that these are hard to predict?

5. Repeat 1 and 2 (d^* is now c and $\{1, \ldots, 7\}$ is now S) above with a Gaussian kernel

$$K(\boldsymbol{p}, \boldsymbol{q}) = e^{-c\|\boldsymbol{p}-\boldsymbol{q}\|^2},$$

   c the width of the kernel is now a parameter which must be optimised during cross-validation however, you will also need to perform some initial experiments to a decide a reasonable set S of values to cross-validate c over.

6. Choose (research) an alternate method to generalise to k-classes then repeat 1 and 2.

7. Choose two more algorithms to compare to the kernel perceptron each of these algorithms will have a parameter vector θ and you will need to determine a cross-validation set S_θ with this information repeat 1 and 2 for your new algorithms.

Assessment: In your report you will not only be assessed on the correctness/quality of your experiment (e.g., sound methods for choosing parameters, reasonable final test errors) but also on the clarity of presentation and the insightfulness of your observations. You may view this as practice scientific writing for your MSc thesis. Thus the aim is that your report is sufficiently detailed so that the reader could largely repeat your experiments based on the description in your report alone. The report should also contain the following.

- A discussion of any parameters of your method which were not cross-validated over.
- A discussion of the two methods chosen for generalising 2-class classifiers to k-class classifiers.
- A discussion comparing results of the Gaussian to the polynomial Kernel.
- A discussion comparing the results of the three algorithms (Kernel Perceptron and the two that you have selected) with respect to both time complexity and test error.
- A discussion of your implementation of the kernel perceptron. This should at least discuss how the sum $\boldsymbol{w}(\cdot) = \sum_{i=0}^{m} \alpha_i K(\boldsymbol{x}_i, \cdot)$ was i) represented, ii) evaluated and iii) how new terms are added to the sum during training.

Note 1 (further comments on assessment) : Your score in Part I is not the "percentage correct," rather it will be a qualitative judgement of the report's *scientific excellence*. Thus a report that is merely correct/good as a baseline can expect 30-40 points out of 50. An excellent report will receive 40-50 points. An outstanding report can receive up to 70 points. Some ways to excel include interesting/challenging selection of comparison algorithms, creating your own implementations of difficult to implement algorithms

(e.g. SVMs), good visualisations or interesting/surprising experimental observations. If you receive over 50 points the additional points will go toward Part II, the score for the coursework will be capped at 100.
Note 2 (use of libraries) : You may use libraries. Thus for example you could use an SVM library. Note, however, there will be the possibility to receive additional credit by implementing the algorithm yourself. A 50/50 score is possible without hand-coding any of the additional algorithms from scratch. A caveat on the use of libraries is as follows. The aim of the report is scientifically reproducibility so any method used must be documented, i.e., some times methods in packages are not well-documented or detailed, i.e., it is not enough to say we used the default parameters of "PKlearn metaBoostTrees", the aim is that some could reproduce your experiments but without access to that library (up to numerical issues). A key example for the distinction here is hard-margin SVMs versus Random Forests in libraries:

a) Hard-margin SVM has a well defined definition but depending on library used you may get slightly different results due to hidden implementation details.

b) Random Forests in libraries often do not have a well-defined definition so different libraries are often slightly or majorly different in implementations. So in this case you need to either completely the document the library's algorithm or produce your own code and documentation.
Note 3 (reducing computation time) : As an alternative to working with the full ∼9300 item dataset. You may do the following save time.

1. Create a new 1000 item dataset by sampling uniformly at random without replacement 100 digits from each of the 10 classes.

2. Each time the instructions ask you run 20 times you may instead run 5 times.

If you choose this option you will lose 10 points on Part I but in general for an excellent report 70/50 will still be possible for Part I

2 PART II [50%]

2.1 Questions

1. **[(a,b,c,d 28pts, e 22pts)]** *Sparse learning:*

   The 'just a little bit' problem. In the following problem we will consider the *sample complexity* of the `perceptron, winnow, least squares, and 1-nearest neighbours` algorithms for a specific problem.

   Problem ('just a little bit'): The m patterns $\boldsymbol{x}_1, \ldots \boldsymbol{x}_m$ are sampled *uniformly* at random from $\{-1, 1\}^n$, and each label is defined as $y_i := x_{i,1}$, i.e., the label of a pattern $\boldsymbol{x}$ is just its first coordinate. Thus for example here is a typical data set with $m = 4$ examples in $n = 3$ dimensions,

$$X = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{pmatrix} \quad Y = \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

   We are concerned with estimating the sample complexity as a function of the dimension (n) of the data of this problem. Where our "working definition" of sample complexity is the minimum number of examples (m) to incur no more than 10% generalisation error (on average).

   (a) In this part, you will implement the four classification algorithms and then use them to estimate the sample complexity of these algorithms. Here you will plot m (left axis) versus n (bottom axis). As an illustration I include an example plot[1] of estimated sample complexity for least squares. Please include sample complexity plots for all four all four algorithms.

[1] In the figure I have included "error bars" which indicate the standard deviation for the estimates of m, it is not necessary for you to include them.

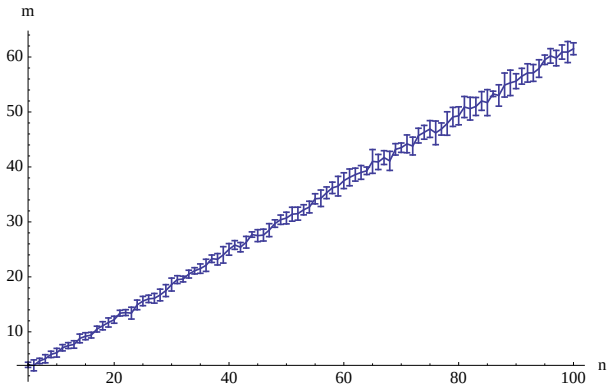

Figure 2: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for least squares.

(b) Computing the sample complexity "exactly" by simulation would be extremely expensive computationally. Thus for your method in part (a) it is necessary to trade-off accuracy and computation time. Hence, i) Please describe your method for estimating sample complexity in detail. ii). please discuss the tradeoffs and biases of your method.

(c) Please estimate how m grows as a function of n as $n \to \infty$ for each of the four algorithms based on experimental or any other "analytical" observations. Here the use of $O(\cdot), \Omega(\cdot), \Theta(\cdot)$ will be useful. For example experimentally from the plot given for least squares it seems that sample complexity grows linearly as a function of dimension, i.e., it is $m = \Theta(n)$. Discuss your observations and compare the performance of the four algorithms.

(d) Now additionally, suppose we sample an integer $s \in \{1, \ldots, m\}$ uniformly at random. Derive a non-trivial upper bound $\hat{p}_{m,n}$ on the probability that the perceptron will make a mistake on the sth example after being trained on examples $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_{s-1}, y_{s-1})$. The derived upper bound $\hat{p}_{m,n}$ should be function of only m and n (i.e., it is independent of s) and should be with respect to the dataset decribed above. Justify your derivation, analytically.

(e) **[Challenge]** Find a *simple* function $f(n)$ which is a *good* lower bound of the sample complexity of `1-nearest neighbor` algorithm for the 'just a little bit' problem. Prove $m = \Omega(f(n))$. *There is no partial credit for this problem.*

Notes

1. `Winnow:` Use $\{0,1\}^n$ for the patterns and $\{0,1\}$ for the labels. This follows our presentation in the notes.

2. `Linear Regression:`

   (a) We use regression vector $\boldsymbol{w}$ to define a classifier $f_{\boldsymbol{w}}(\boldsymbol{x}) := \text{sign}(\boldsymbol{w}^\top \boldsymbol{x})$.

   (b) For this problem we are usually in the *underdetermined* case. We use the convention that the linear regression solution is a limiting case of the ridge regression solution. A technical presentation of this is given `http://www.cs.ucl.ac.uk/staff/M.Pontil/courses/2-gi07.pdf`, equation (6) and details on p25-p26]. The practical "take away," is that in matlab we may use $w = pinv(X) * y$ to compute the "w" of minimal norm that is consistent with the data. This is contrary to the usual advice to use the `left division` operator in matlab for regression. This is so that we have consistent definition of linear regression across programming languages. Finally, note computing pseudoinverse is still inefficient as a method to solve for the minimal norm solution in the underdetermined case, however for this exercise the efficiency of the implementation is not the focus.

3. **Sample Complexity:** For this problem, let $\mathcal{S}_m$, denote a set of m patterns drawn uniformly at random from $\{-1,1\}^n$ with their derived labels. Then let $\mathcal{A}_{\mathcal{S}}(\boldsymbol{x})$ denote the prediction of an algorithm $\mathcal{A}$ trained from data sequence $\mathcal{S}$ on pattern $\boldsymbol{x}$. Thus the generalisation error is then

$$\mathcal{E}(\mathcal{A}_{\mathcal{S}}) := 2^{-n} \sum_{\boldsymbol{x}\in\{-1,1\}^n} I[\mathcal{A}_{\mathcal{S}}(\boldsymbol{x}) \neq x_1]$$

and thus the sample complexity on average at 10% generalisation error is

$$\mathcal{C}(\mathcal{A}) := \min\{m \in \{1,2,\ldots\} : \mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] \leq 0.1\}\,.$$

With sufficient computational resources we may compute this exactly via,

$$\mathcal{C}(\mathcal{A}) = \min\{m \in \{1,2,\ldots\} : (2^{-nm} \sum_{S\subseteq\{-1,1\}^{n^m}} \mathcal{E}(\mathcal{A}_{\mathcal{S}})) \leq 0.1\}\,.$$