CSCI 635: Introduction to Machine Learning
Homework 2: Multiclass Classification

Due Date: Sunday, November 11, 11:59pm, *Late:* Tuesday, November 13, 11:59pm

Instructions: The assignment is out of 100 points. Submit your assignment through the Dropbox in MyCourses as two files zipped into one container (named according to the convention `<your_last_name>-hw2.zip`). The individual files are:

1. A *.pdf file for the write-up, named `a2.pdf`. Furthermore, while your Python scripts will allow you to save your plots to disk, please copy/insert the generated plots into your document again mapped to the correct problem number. For answers to various questions and recorded metrics throughout, please copy the question(s) being asked and write your answer/content underneath (as well as the problem number).
2. The *.py files for your answers/implementations to Questions 1a, 1b, 1c, and 2, named `q1a.py`, `q1b.py`, `q1c.py`, and `q2.py`.

Grade penalties will be applied (but not limited to) for: 1) not submitting the write-up as instructed above, 2) submitting code with incorrect file names, and 3) using any machine-learning-specific library for your code, e.g., TensorFlow, PyTorch.

In this assignment, you will implement, in **raw numpy/scipy**, and apply two types of multi-class classifiers, i.e., maximum entropy and naïve Bayes, as well as their parameter update procedures, i.e., (stochastic) gradient descent and maximum likelihood estimation. Your focus will be on multiclass (multinomial) classification –note that this is different from multi-label classification, where each data point could belong to two or more distinct label sets.

Problem #1: Learning a Maximum Entropy Model (60 points)

In order to learn a parametrized model, as we saw in class, we need a method for calculating partial derivatives of our loss with respect to model parameters and an update rule that alters model parameters using these computed partial derivatives. For this problem, you will implement these. You will use the gradients you calculate to update weights via mini-batch gradient descent.

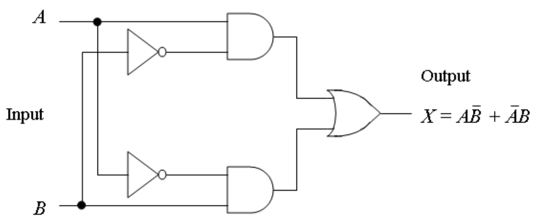
The logical exclusive-OR (XOR) problem is a classical problem that is linearly inseparable and useful for motivating the need for hidden processing units. It is fully specified via the following truth table depicted to the right. You will work with this classical problem as you develop your basic maximum entropy classifier. In Figure 1, you can see the classical circuit depicted as well as a non-learnable/hard-coded perceptron that can directly solve the problem. In the next homework, you will revisit this problem when you design an artificial neural network. The data for XOR is in `xor.dat`.

$\mathbf{in}_1$	$\mathbf{in}_2$	$\mathbf{out}$
0	0	0
0	1	1
1	0	1
1	1	0

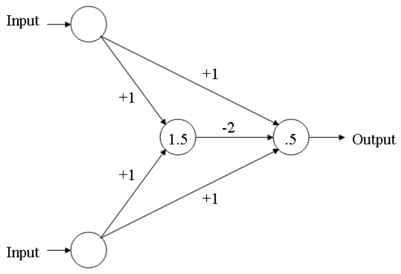
Problem #1a: Tackling the XOR Problem (30)

Implementing a gradient update rule (which is an application of the chain rule from differential calculus) is rather straightforward once a loss function and model structure have been defined. To begin, we will first design the a simple linear model – maximum entropy, otherwise known softmax or multinoulli[1] regression. This model could be interpreted as an artificial neural network with no hidden units. Our parameters are housed in the construct $\Theta = \{W, \mathbf{b}\}$ (where W is a weight matrix of dimensions $(d \times k)$ and b is a bias vector of dimensions $(1 \times k)$, where d is the number of observed variables and k is the number of classes we want to predict). Our dataset is $\mathbf{X}$ which is of dimension $(N \times d)$ (N is the number of samples) and y (which is a column vector of

[1]The multinoulli distribution (also known as the categorical distribution) is a generalization of the Bernoulli distribution. If you perform an experiment with *k* outcomes, you have a multinoulli distributed random variate.



(a) The exclusive-or (XOR) circuit.



(b) An XOR perceptron gate.

Figure 1: *In Figure 1a we see the full XOR gate's digital circuit diagram and in Figure 1b we see the same gate implemented as a hard-coded (technically multi-layer) perceptron. The center-most unit of the perceptron-gate is hidden from outside influence and is connected only to input or output units. The value of 1.5 for the threshold of the internal unit ensures that it will be turned on only when both input units are on. The value of 0.5 for the output unit ensures that it will turn on only when the net positive input is greater than 0.5. The weight of -2 from the hidden unit to the output unit ensures that the output unit will* not *turn on when both inputs are on.*

class indices, dimension $(N \times 1)$, where each entry is an integer representing the class index, i.e., $y_i \in \{0, 1, \cdots, c, c+1, \cdots, (C-1)\}$ where C is the number of classes – note you will encode each label into a one-hot encoding, $\mathbf{y}_i = (\mathbf{1}_{y_i = c})$. We retrieve a particular datapoint $(y_i, \mathbf{x}_i)$ from the dataset $\mathcal{D} = (y, \mathbf{X})$ via the integer index i.

We will learn parameters by minimizing the negative log likelihood of our model's predictive posterior. If we say that $\mathbf{p}$ is our model's vector of normalized output probabilities, we define the cost as follows:

$$\mathbf{p} = \frac{\exp(\mathbf{f})}{\sum_j \exp(\mathbf{f}[j])}, \qquad \mathcal{J} = -\frac{1}{N} \sum_i \Big(\log(\mathbf{p}[i, y_i]) \Big) + \frac{\lambda}{2} \sum_d \sum_k (W[d, k])^2 \tag{1}$$

noting that we have written a regularized loss (imposing a Gaussian prior distribution over the input-to-hidden parameters W).[2] The computation of $\mathbf{p}$ is simply the application of the softmax link function presented to you in class, where $\mathbf{f} = W \cdot \mathbf{X} + \mathbf{b}$ is the computation of the logits (or log odds/class scores).

We first seek $\partial \mathcal{J} / \partial \mathbf{f}$, which is the gradient of the loss with respect to the maximum entropy model's pre-activation. Implementing the gradient calculation for softmax regression turns out to be quite simple once one works through the derivations (by judiciously applying the chain rule of calculus[3]). It turns out that most things cancel out in the derivation, yielding the elegant and very simple expression:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{f}} = \mathbf{p} - \mathbf{y} \tag{2}$$

which directly gives us the partial derivatives of our loss with respect to the output logits $\mathbf{f}$.

We can think of this as simply subtracting one from the probability indexed at the correct class c (or your probability vector minus the one-hot encoding of your class label). To obtain the partial derivatives of the loss with respect to parameters, we will need to take the derivative above and transmit it backwards through the model, yielding the gradients:

$$\frac{\partial \mathcal{J}}{\partial W} = \mathbf{X}^T * \Big(\frac{1}{N} \frac{\partial \mathcal{J}}{\partial \mathbf{f}} \Big) + \lambda(W) = \mathbf{X}^T * \Big(\frac{1}{N} (\mathbf{p} - \mathbf{y}) \Big) + \lambda(W) \tag{3}$$

$$\frac{\partial \mathcal{J}}{\partial \mathbf{b}} = \sum_{i=1}^{N} \frac{1}{N} \frac{\partial \mathcal{J}}{\partial \mathbf{f}}[i, ?] = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{p} - \mathbf{y})[i, ?] \tag{4}$$

where we also see that the gradient of the Gaussian prior over the weights W is equally straightforward (as discussed in class, we do not regularize the biases). Note that the notation used here has been massaged a bit to be more implementation-oriented (for a row-major language like Python). ? is the

[2]For Equation 1, you can also perform the same calculcation with your one-hot encodings instead, i.e., $\log(\mathbf{p}[i, y_i]) = \sum_j (\mathbf{y} \otimes \log(\mathbf{p}))[i, j]$, where $\otimes$ is the Hadamard product.

[3]This is a rewarding derivation to work through but is left as an optional, ungraded exercise to the reader.

wildcard index, which allows us to slice a matrix to retreive the desired row or column vector, i.e., $\mathbf{p}[i, ?]$ means we retreive the ith row of matrix $\mathbf{p}$.

Gradient-checking the code you have written for the partial derivatives is absolutely essential. Oftentimes, it is possible that your model will appear to "learn" even with a buggy implementation, however, you will find that it never quite behaves the way literature claims it should. While for the last homework assignment, gradient-checking was optional (though suggested), you are required to do so in this assignment to verify your learning algorithm's correctness.

As discussed in class, one can approximate the partial derivatives desired by instead invoking a secant approximation – this is known as the method of finite differences. To calculate this numerical gradient, all we require is the prediction routine and the cost function routine to be correctly written. Specifically, we may estimate the gradient by simply recording how the loss function's value changes each time we perturb a parameter in a certain direction, by a certain amount, ϵ. The secant approximation to the derivatives we seek follow from the definition of a limit:

$$\frac{\partial \mathcal{J}(\Theta)}{\partial \Theta} = \lim_{\epsilon \to 0} \frac{\mathcal{J}(\Theta + \epsilon) - \mathcal{J}(\Theta - \epsilon)}{2\epsilon} \tag{5}$$

where now our derivatives can be approximated by simply using the above equation and setting ϵ to a reasonably small value (i.e., 10^{-4} or 10^{-3} – do not set this to too small a value, otherwise you will encounter numerical instabilities). The essence of implementing the numerical gradient calculation involves iterating over each and every single parameter scalar value (for example, each value inside the W matrix). As an example, let us image we have flattened Θ (W and $\mathbf{b}_j$) into a single long vector. To find its numerical gradient, you would loop through the parameter vector and do the following to each scalar value (Θ_j) within Θ:

1. Add ϵ to Θ_j, put Θ_j back into Θ, giving us $(\Theta + \epsilon)$ at j
2. Given perturbed Θ_j (all the rest of the values in Θ are fixed), calculate $\mathcal{J}(\Theta + \epsilon)$
3. Reset Θ_j to its original state
4. Subtract ϵ from Θ_j
5. Given perturbed Θ_j, calculate $\mathcal{J}(\Theta - \epsilon)$
6. Estimate the scalar derivative, $\left(\frac{\partial \mathcal{J}(\Theta)}{\partial \Theta_j}\right)_j$ using Equation 5
7. Reset b to its original state, returning $\sim \nabla_{\Theta_j}$
8. Repeat Steps 1-7 for each Θ_j in Θ

Your implemented numerical gradient algorithm will return an approximation ($\sim \nabla_{\Theta_j}$) of $\frac{\partial \mathcal{J}(\Theta)}{\partial \Theta_j}$. Once you have obtained the full set of numerical partial derivatives, you will then want to compare them against your own routine's calculation of the exact derivatives.

You should observe agreement between your analytic solution and the numerical solution, up to a reasonable amount of decimal points in order for this part of the problem to be counted correct. Make sure you check your gradients before you proceed with the rest of this problem. Make sure the program returns "CORRECT" for each parameter value in W and $\mathbf{b}$, i.e., for any specific parameter θ_j in Θ, we return "CORRECT" if: $\sim \nabla_{\theta_j} < 1e-4$.

Once you have finished gradient-checking, fit your maximum entropy model to the XOR dataset, found in the file. Describe what hyper-parameters you selected, your process for choosing them, and observations made related to your model's ability to fit the data and its loss per epoch. Record your model's final accuracy. Code goes into a file you will name `q1a.py`.

Problem #1b: Softmax Regression & the Spiral Problem (15)

Now it is time to fit a model to a multi-class problem (generally defined as $k > 2$, or beyond binary classification). The spirals dataset (which we artificially generated for you offline to simulate an "unknown" data generating process) represents an excellent toy problem for observing nonlinearity.

Instead of the XOR data, you will now load in the spiral dataset file, `spiral_train.dat`. Train your maximum entropy model on the data and then plot its decision boundary. Please save and update your answer document with the generated plot once you have fit the model as best you can to the data. Make sure you comment on your observations and describe any steps taken in tuning the model. You will certainly want to re-use the code as you work through sub-problems to minimize implementation bugs/errors. Do NOT forget to report your accuracy. Code goes into a file you will name `q1b.py`.

Problem #1c: The IRIS Maximum Entropy Model (15)

You will now re-appropriate your maximum entropy code to a more standard machine learning problem, where a separate validation set is given in addition to the training set. In this scenario, you are no longer concerned directly with pure optimization (as we discussed in class), but with generalization (or out-of-sample performance). As such, you will have two quantities to track – the training loss and the validation loss.

Fit/tune your maximum entropy model to the IRIS dataset, `iris_train.dat`. Note that since you are concerned with out-of-sample performance, you must estimate generalization accuracy by using the validation/development dataset we have also provided for you, `/problems/HW2/data/iris_test.dat`. Code goes into a file you will name `q1c.py`.

Note that for this particular problem, you will want to make sure your parameter optimization **operates with mini-batches** instead of the full batch of data to calculate gradients. As a result, you will have to write a mini-batch creation function, i.e., a *create_mini_batch(·)* routine, where you draw randomly **without replacement** M data vectors from $\mathbf{X}$ (and their respective labels from y). This will be necessary in order to write a good training loop.

Besides recording your accuracy for both your training and development sets, track your loss as a function of epoch. Create a plot with both of these curves superimposed. Furthermore, consider the following questions: What ultimately happens as you train your model for more epochs? What phenomenon are you observing and why does this happen? What is a way to control for this?

Problem #2: Learning a Naïve Bayes Classifier (40 points)

The data set given in `q2.csv` is synthetically generated, but labeled in an intentional way. Note that it includes both discrete and real-valued parameters. Use the naïve Bayes model for the data, with Spam being the parent class (assume that each real-valued parameter has a Gaussian distribution).

1. Using the data in `q2.csv`, compute the maximum-likelihood parameters for this network.
2. For each row of the file `q2b.csv`, use the values of the features other than Spam to compute $P(Y|featurevalues)$ for each row. Compute an overall classification error rate based on a threshold $P(Y)$ of 0.5.
3. **Playtime!** Repeat the preceding analysis, but ignoring some or all of the features (columns), and compare the classification accuracy. Choose a subset of the features that is small but seems to give good results.

Your code shall go into a file you will name as `q2.py`. This file should have (at least):

- a function that performs the parameter computation for the full feature set
- a function that performs the classification for new data samples using the full feature set
- a function that performs classification based on your chosen subset of the features
- Documentation explaining how to use these functions!

In your writeup, report the parameters for your model as computed in part a, the classification error you achieved in part b, and (most importantly) explain the process that you undertook in part c) - both how you modified your code to handle this problem, as well as how you chose features to use and/or remove and what classification error your simpler classifier obtained.

Note that for all sub-problems, make sure you place every Python script and documentation/answers/plots into a single file that is zipped. The zip file is to be named `<your_last_name>-hw2.zip`