Machine Learning Fall 2017 Homework 3

Homework must be submitted electronically on Canvas. Make sure to explain you reasoning or show your derivations. Except for answers that are especially straightforward, you will lose points for unjustified answers, even if they are correct.

General Instructions

You are allowed to work with at most one other student on the homework. With your partner, you will submit only one copy, and you will share the grade that your submission receives. You should set up your partnership on Canvas as a two-person group.

Submit your homework electronically on Canvas. We recommend using LaTeX, especially for the written problems. But you are welcome to use anything as long as it is neat and readable.

For the programming portion, you should only need to modify the Python files. You may modify the iPython notebook, but you will not be submitting them, so your code must work with our provided iPython notebook.

Relatedly, cite all outside sources of information and ideas.

Written Problems

1. Machine learning methods can be viewed as function estimators. Consider the logical functions AND, OR, and XOR. Using a signed representation for Boolean variables, where input and output variables are in $\{+1, -1\}$, these functions are defined as

$$\mathrm{AND}(x_1, x_2) = \begin{cases} +1 & \text{if } x_1 = +1 \wedge x_2 = +1 \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

$$\mathrm{OR}(x_1, x_2) = \begin{cases} +1 & \text{if } x_1 = +1 \\ +1 & \text{if } x_2 = +1 \\ -1 & \text{otherwise} \end{cases} \tag{2}$$

$$\mathrm{XOR}(x_1, x_2) = \begin{cases} +1 & \text{if } x_1 = +1 \wedge x_2 = -1 \\ +1 & \text{if } x_2 = +1 \wedge x_1 = -1 \\ -1 & \text{otherwise} \end{cases} \tag{3}$$

   (a) (6 points) Which of these three logical functions can be expressed as a linear classifier of the form

$$f(\boldsymbol{x}; \boldsymbol{w}) = \mathrm{sign}(w_1 x_1 + w_2 x_2 + b), \tag{4}$$

   and show weights w_1, w_2, and bias values b that mimic these logical functions. Which of these functions cannot be expressed as a linear classifier, and why not?

   (b) (6 points) For any logical functions that cannot be expressed as a linear classifier, show how and with what weights it can be expressed as a two-layered perceptron of the form

$$f(\boldsymbol{x}; \boldsymbol{w}) = \mathrm{sign}(\boldsymbol{w}^{\mathrm{out}\top} \boldsymbol{h} + b^{\mathrm{out}}) \tag{5}$$

$$\boldsymbol{h} = [h_1, h_2]^\top \tag{6}$$

$$h_1 = \mathrm{sign}(\boldsymbol{w}^{(1)\top} \boldsymbol{x} + b^{(1)}) \tag{7}$$

$$h_2 = \mathrm{sign}(\boldsymbol{w}^{(2)\top} \boldsymbol{x} + b^{(2)}) \tag{8}$$

   For this problem, you must specify three weight vectors $(\boldsymbol{w}^{\mathrm{out}}, \boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)})$ and three biases $(b^{\mathrm{out}}, b^{(1)}, b^{(2)})$.

Programming Assignment

For this programming assignment, we have provided a lot of starter code. Your tasks will be to complete the code in a few specific places, which will require you to read and understand most of the provided code, but will only require you to write a small amount of code yourself.

In `syntheticData.mat`, there are 10 synthetic, two-dimensional datasets. All but the first dataset are not linearly classifiable. We provided the main experiment notebook `run_synthetic_experiment.ipynb` for you to use. For each model, the notebook uses cross-validation on the training data to choose learning parameters, trains on the full training set using those parameters, and evaluates the accuracy on the test set.

We have also provided a unit test class in `test_mlp_and_svm.py`, which contains unit tests for each type of learning model. These unit tests may be easier to use for debugging in an IDE like PyCharm than the iPython notebook. A successful implementation should pass all unit tests and run through the entire iPython notebook without issues. You can run the unit tests from a *nix command line with the command

```
python -m unittest -v test_mlp_and_svm
```

or you can use an IDE's unit test interface.

Before starting all the tasks, examine the entire codebase. Follow the code from the iPython notebook to see which methods it calls. Make sure you understand what all of the code does.

1. (6 points) Finish the back-propagation operation in the `mlp_objective` function in `mlp.py`.

   You should implement the matrix-form of back-propagation that we covered in class, which we replicate here. First, forward propagation computes the predicted values. We refer to the neural activation values as $\boldsymbol{h}$ vectors. For notational convenience, for input $\boldsymbol{x}$, let $\boldsymbol{h}_0 = \boldsymbol{x}$. Then forward propagation computes $\boldsymbol{h}_i \leftarrow s\left(\mathbf{W}_1 \boldsymbol{h}_{i-1}\right)$ for i from 1 to the depth of the MLP (`num_layers`). If `num_layers` $= m$, the output is $\boldsymbol{h}_m$. Forward propagation also computes the derivatives of the squashing function and stores them in `squash_derivatives`, i.e.,

   $$\texttt{squash\_derivatives[i]} := s'\left(\mathbf{W}_i \boldsymbol{h}_{i-1}\right).$$

   Back-propagation should start by computing the "errors" δ—i.e., the gradients of the loss with respect to each layer's pre-squashed inputs—backwards from the output layer (δ_m) back to the input layer (δ_1). Use the chain-rule recursion, which starts with the last layer $\delta_m = \ell'(f(\boldsymbol{x}, \mathbf{W}))$, and for all other layers,

   $$\delta_i = \left(\mathbf{W}_{i+1}^\top \delta_{i+1}\right) \circ s'\left(\mathbf{W}_i \boldsymbol{h}_{i-1}\right) \tag{9}$$

   where the $\circ$ operator is the element-wise product (the default product for `numpy` `ndarrays`).

   Finally, now that the errors have been computed, compute the gradient for each layer with the formula

   $$\nabla_{\mathbf{W}_i} \ell = \delta_i \boldsymbol{h}_{i-1}^\top \tag{10}$$

   You should implement these two equations in the marked `TODO` lines in `mlp.py`.

2. (6 points) Finish the support-vector machine setup in `kernel_svm_train` and `kernel_svm_predict` in `kernelsvm.py`.

   The dual quadratic program for kernel SVM is

   $$\begin{aligned} \min_{\boldsymbol{\alpha}} \ & \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{K} \boldsymbol{\alpha} - \sum_{i=1}^n \alpha_i \\ \text{s.t.} \ & \sum_{i=1}^n \alpha_i y_i = 0, \ \ \alpha_i \in [0, C] \end{aligned} \tag{11}$$

   where $\mathbf{K}$ is the Gram matrix, y_i is the label of the ith point, C is the regularization parameter, and the free vector $\boldsymbol{\alpha}$ is the dual variables. For the linear kernel, all of the input quantities have been

computed for you. Your job is to put them into the correct form to solve the quadratic program in the `TODO` in `kernel_svm_train`. You will be using our custom wrapper for the open-source `quadprog` solver, which you will need to install to run (usually `pip install quadprog` will work).

Make sure to examine the code after the quadratic program completes. This code saves the information necessary for prediction with the returned `model` object, including the nonzero support vectors and the bias b. Next, you should finish the linear SVM by implementing the predictor in the `TODO` in `kernel_svm_predict`. The formula for prediction $f(\boldsymbol{x})$ is

$$f(\boldsymbol{x}) = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i^\top K(\boldsymbol{x}_i, \boldsymbol{x})\right). \tag{12}$$

As usual, you should be able to manipulate this equation to compute the predictions for a batch of data without loops.

Once you complete this second `TODO`, the linear SVM should pass its tests, and the part of the iPython notebook that tests the linear kernel should run (and perform poorly because the data is not linearly separable). You may see some of the quadratic programs exit with warnings. These warnings are okay; they result from the problem becoming poorly scaled and should not happen as often with the non-linear kernels in the next problem.

3. (6 points) Finish the two kernel functions `polynomial_kernel` and `rbf_kernel` in `kernelsvm.py`.

   This final task requires approximately two lines of code. You should return the Gram matrix for each of these famous kernel functions. The formula for the polynomial kernel of order k is

   $$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^\top \boldsymbol{x}_j + 1)^k . \tag{13}$$

   And the formula for the Gaussian radial-basis function (RBF) kernel is

   $$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(\frac{1}{\sigma}(\boldsymbol{x}_i - \boldsymbol{x}_j)^\top (\boldsymbol{x}_i - \boldsymbol{x}_j)\right). \tag{14}$$

   Both formulas must be computed without loops for full credit, i.e., your code should directly compute these from the inputs `row_data` and `col_data` without slicing to compute over individual columns of these. The polynomial kernel is fairly straightforward to convert into matrix operations. For the RBF kernel, one hint is that it helps to expand the squared distance

   $$(\boldsymbol{x}_i - \boldsymbol{x}_j)^\top (\boldsymbol{x}_i - \boldsymbol{x}_j)$$

   into

   $$\boldsymbol{x}_i^\top \boldsymbol{x}_i + \boldsymbol{x}_j^\top \boldsymbol{x}_j - 2\boldsymbol{x}_i^\top \boldsymbol{x}_j .$$

   This expanded form is easier to reason about via matrices and vectors. Also, see the class slides for some matrix formulas for these kernel computations.