CS 4824 / ECE 4424 HW 3 Programming Portion

General Instructions:

- 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 to implement multilayer perceptron and kernel SVM. You will be testing these classifiers on `syntheticData.mat`, which contains 10 synthetic two-dimensional datasets. All but the first dataset are not linearly classifiable.
- You only need to upload two Python files: `mlp.py` and `kernelsvm.py`, that you will be modifying for this assignment. We will be testing your code while keeping all other Python files the same as provided in the original folder. Please make sure that your code works with our provided iPython notebook and the unit test class. At the top of both of your Python submission files (on line 5), please include your name and VT email id as a comment, so that it is easy for us to manage your submissions during the grading process.
- We have 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. The tests only check for the accuracy of the classifiers but you should make sure that your code completes within a reasonable time. If your code takes more than 30 minutes to complete all tests on a standard computer, you will not receive any marks. You should avoid long and nested loops as much as posssible and instead use matrix operations to make your code efficient and also reduce the number of lines.
- We have also provided the main experiment notebook `run_synthetic_experiment.ipynb` for you to analyze the classifiers and their results on the 10 datasets. 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.
- 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.

Programming Tasks:

1. (10 points) Finish the back-propagation operation in the `mlp_objective` function in `mlp.py`.

   You should implement the matrix-form of back-propagation that can be described as follows. Let us denote the set of inputs as a matrix $\boldsymbol{X}$ of size $d \times n$ where d is the number of input dimensions and n is the number of data points (we append a row of ones to $\boldsymbol{X}$ to account for the bias term in neural network weights). The labels correspond to a binary vector y of length n with values in $\{-1, +1\}$. For a neural network with L layers, let us denote the set of activations at layer i on all data points as the matrix $\boldsymbol{A}_i$ of size $m_i \times n$, where m_i is the number of nodes at layer i. For notational convenience, let $\boldsymbol{A}_0 = \boldsymbol{X}$, which is the input matrix and $m_0 = d$. Further, the output of the neural network is $\boldsymbol{A}_L$ of size $m_L \times n$ where $m_L = 1$. The weight vector at layer i is given by the matrix $\mathbf{W}_i$ of size $m_{i+1} \times m_i$.

   During forward-propagation, the activations at layer i are computed as $\boldsymbol{A}_{i+1} = \sigma(\mathbf{W}_i \boldsymbol{A}_i)$ for i ranging from 0 to $L-1$, where σ denotes the sigmoid activation function. Forward propagation also computes the derivatives of the sigmoid activation function $\boldsymbol{S}_i$ and stores them in `activation_derivatives`, i.e.,

   $$\texttt{activation\_derivatives[i]} := \boldsymbol{S}_i = \boldsymbol{A}_{i+1} \circ (1 - \boldsymbol{A}_{i+1}),$$

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

Back-propagation should start by computing the "errors" δ_i—i.e., the gradients of the loss with respect to each layer's activations, $\boldsymbol{A}_i$—backwards from the output layer (δ_{L-1}) back to the first hidden layer (δ_0). In our code, the errors δ_i are stored in `layer_delta[i]`. Use the chain-rule recursion to compute these errors, which starts with the last layer, $\delta_{L-1} = \partial\ \texttt{Cross-entropy-Loss}/\partial \boldsymbol{a}_L$, and for all other layers,

$$\delta_i = \mathbf{W}_{i+1}^\top (\delta_{i+1} \circ S_{i+1}) \tag{1}$$

where $\boldsymbol{a}^\top$ denotes the transpose of $\boldsymbol{a}$.

Once δ_i have been computed for all layers i ranging from 0 to $L-1$, compute the gradients for each layer and store them in `layer_gradients` using the formula

$$\texttt{layer\_gradients[i]} := \nabla_{\mathbf{W}_i} \texttt{Cross-entropy-Loss} = (\delta_i \circ \boldsymbol{S}_i)\, \boldsymbol{A}_i^\top \tag{2}$$

You should implement these two equations in the marked `TODO` lines in `mlp.py`. You will receive 1 bonus point each if Equations 1 and 2 are implemented in a single line.

2. (10 points) Finish the two kernel functions `polynomial_kernel` and `rbf_kernel` in `kernelsvm.py` for implementing non-linear SVMs. You are already given the code for the linear kernel in the function `linear_kernel` and the SVM setup in `kernel_svm_train` and `kernel_svm_predict` functions.

   The dual quadratic program for kernel SVM is given by

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^\top (\mathbf{K} \circ \boldsymbol{y}\boldsymbol{y}^\top)\boldsymbol{\alpha} - \sum_{i=1}^{n} \alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0, \quad \alpha_i \in [0, C] \end{aligned} \tag{3}$$

   where $\mathbf{K}$ is the Gram matrix, $\boldsymbol{y}$ is the vector of labels where y_i is the label of the ith point, C is the regularization parameter, and the learnable vector $\boldsymbol{\alpha}$ is the set of Lagrange multipliers. To solve the dual problem, we will be using a custom wrapper for the open-source `quadprog` solver, which you will need to install to run.

   To install `quadprog`, `pip install quadprog` will sometimes work depending on your system, or you can clone the repository at `https://github.com/rmcgibbo/quadprog` and use `python setup.py install` to install it directly from source. Also, `conda install -c omnia quadprog` is another way to install it.

   Once `quadprog` has been installed, you should be able to run SVM with linear kernel in the iPython notebook without any errors. 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 you will be implementing.

   Implementing the polynomial kernel and RBF kernel requires approximately 2 to 5 lines of code. You should return the Gram matrix for each of these kernel functions. The formula for the polynomial kernel of order k is

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^\top \boldsymbol{x}_j + 1)^k\ . \tag{4}$$

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

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{1}{2\sigma^2}(\boldsymbol{x}_i - \boldsymbol{x}_j)^\top (\boldsymbol{x}_i - \boldsymbol{x}_j)\right). \tag{5}$$

   Avoid using loops while computing both these formulas, 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. <u>You will receive 1 bonus point each if Equations 4 and 5 are implemented without any loops.</u>