CIS 520: Machine Learning *Due:* Mar. 2, 2021 (11:59 AM) EST

Problem Set 2

Submission Instructions:

- **PDF Submission:** This is your main submission, and must contain your solutions to **all problems that you are attempting, including both mathematical problems and programming problems.** It must be submitted as a single PDF file to **Gradescope**, compiled in LaTeX using the LaTeX template provided on Canvas. Each problem should be on a new page. Mathematical problems can be either typed in LaTeX or written by hand and **scanned** into images included in your LaTeX solution, but it must be **readable** by our staff to be graded; we recommend that you not take photos of your hand-written solutions. For programming problems, in addition to including any plots, observations, or explanations as requested, be sure to include a snippet of your code in your LaTeX solution; for this, we encourage you to use `lstlisting` environment in `listings` packages for LaTeX.

  Special instruction for submitting on Gradescope: For each problem, select the pages containing your solution for that problem.

- **Code Submission:** Submit all your (Python) code files (**.py files**) to the problem set's code submission component on **Gradescope**. If you use Jupyter notebook, please export them to the corresponding **PS*-P*.py** files and include them in your code submission. Note that these are **in addition** to writing your code inline in the PDF file above. Do not rename any given python files.

- **Python version and libraries:** Make sure that your code runs with **Python 3.8**. Your implementation should **not** use any external libraries other than numpy, scipy and those that are explicitly listed in the handout.

Summary: The PDF file and Python files should be submitted to the corresponding assignments on Gradescope. All components of problem set must be received in the right places and in the right formats before the submission deadline. Plan to submit early!

Collaboration Policy:

You are allowed to discuss problems in groups, but you must write all your solutions and code **individually, on your own**. All collaborators with whom problems were discussed **must** be listed in your PDF submission.

1. (35 points) **Programming Exercise: Support Vector Machines.** Complete the implementation of the support vector machine class in `support_vector_machines.py`. You need to implement the following functions, with more detailed instructions in `support_vector_machines.py`:

   - `fit`
   - `predict`

   In this problem, you will implement the SVM algorithm for binary classification and will apply it to both synthetic and real data. Complete a PYTHON implementation of the SVM classifier: your program should take as input a training set $(\mathbf{X}, \mathbf{y})$, where $\mathbf{X}$ is an $m \times d$ matrix (m instances, each of dimension d) and $\mathbf{y}$ is an m-dimensional vector (with $y_i \in \{\pm 1\}$ being a binary label associated with the i-th instance in $\mathbf{X}$), parameter C, kernel type (which can be 'linear', 'polynomial' or 'rbf'), and kernel parameter (which you can take here to be a single real number r; this is used as the degree parameter q in the polynomial kernel $K(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}' + 1)^q$, and as the parameter γ in the RBF kernel $K(\mathbf{x}, \mathbf{x}') = e^{-\gamma ||\mathbf{x}-\mathbf{x}'||_2^2}$). The program should learn an SVM model (consisting of the kernel type and parameters, the support vectors, the coefficients corresponding to the support vectors and the bias term). You are provided with two data sets for this problem: a synthetic 2-dimensional data set (Synthetic data set), and a real data set (Spam data set); each data set is split into training and test sets.

   (a) **Data set 1 (synthetic 2-dimensional data).** This data set contains 200 training examples and 1800 test examples, all generated i.i.d. from a fixed probability distribution. (See folder `P1/Synthetic-Dataset` under the folder for this problem for relevant files; labels y are in the last column.)

      i. **Linear SVM.** Use your implementation of SVM to learn a linear classification model from the given training data, selecting C from the range $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 10^2\}$ via 5-fold cross-validation on the training set (the train and test data for each fold are provided in a separate folder in `P1/Synthetic-Dataset/CrossValidation`). Show a plot of the training, test, and cross-validation errors as a function of C. Report the selected value of C, and the corresponding training and test error. Also show a plot of the resulting decision boundary.

      ii. **Kernel SVM: polynomial kernel.** Use your implementation of SVM to learn a polynomial kernel classifier from the given training data. Consider degree-q polynomial kernels for $q \in \{1, 2, 3, 4, 5\}$. For each q, select C from the range $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 10^2\}$ via 5-fold cross-validation on the training set; further, also select q using cross-validation. (In other words, for each q, select the best C in terms of cross-validation error as above; then further select the best q via cross-validation, i.e. select a value of q for which the best value of C gives the lowest cross-validation error). Use the same cross-validation folds as above. Show a plot of the training, test, and cross-validation errors as a function of q (for each value of q, use the best value of C based on cross-validation). Report the selected value of q (and C), and the corresponding training and test error. Again, also show a plot of the resulting decision boundary.

      iii. **Kernel SVM: RBF kernel.** Use your implementation of SVM to learn an RBF kernel classifier from the given training data. Consider RBF kernels with parameter γ in the range $\{10^{-2}, 10^{-1}, 1, 10, 10^2\}$. For each γ, select C from the range $\{10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 10^2\}$ via 5-fold cross-validation on the training set; further, also select γ using cross-validation. (In other words, for each γ, select the best C in terms of cross-validation error as above; then further select the best γ via cross-validation, i.e. select a value of γ for which the best value of C gives the lowest cross-validation error). Use the same cross-validation folds as above. Show a plot of the training, test, and cross-validation errors as a function of γ (for each value of γ, use the best value of C based on cross-validation). Report the selected value of γ (and

C), and the corresponding training and test error. Also show a plot of the resulting decision boundary.

iv. Summarize your results in the following table:

Algorithm	Parameters selected	Training Error	Test Error
Linear SVM	$C =$		
Kernel SVM, Polynomial Kernel	$q =; C =$		
Kernel SVM, RBF Kernel	$\gamma =; C =$		

(b) **Data set 2 (real data).** This is a spam classification data set, where each instance is an email message represented by 57 features and is associated with a binary label indicating whether the email is spam (+1) or non-spam (−1).[1] The data set is divided into training and test sets; there are 250 training examples and 4351 test examples. The goal is to learn from the training set a classifier that can classify new email messages in the test set. (See folder `P1/Spam-Dataset` for relevant files.) Repeat all steps (i)-(iv) in part (a) for this data set (except for plotting the decision boundaries, which is hard in 57 dimensions!); use the same parameter ranges as in part (a).

Python Instructions:

(a) *Use CVXOPT quadratic program solver* `cvxopt.solvers.qp` *to solve the SVM dual problem. You should NOT need to use any special cvxopt parameters/settings in your implementation. Convert your variables using* `cvxopt.matrix` *before passing them into* `cvxopt.solvers.qp`.

(b) *You may need to threshold α values from the* `cvxopt.solvers.qp` *output for calculating SV1 and SV due to the numbers being very small (effectively 0). Generally, put α in the range $(C*10^{-6}, C*(1-10^{-6}))$.*

(c) *You can use the provided function* `plot_contour` *for plotting the decision boundary for part (a). Read the comments in the function carefully.*

(d) *You can use the provided functions in* `utils.py` *for loading training-testing data, loading cross validation data and picking cross validation fold, and computing classification error.*

(e) *Please do NOT modify the signature (function name, parameter name, parameter order) of any of the class functions.*

(f) *Include all of the codes you used for experiments and plotting in* `PS2-P1.py` *and follow submission instructions when submitting.*

2. (15 points) **Kernel Functions.** Let $K_1 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ and $K_2 : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be two symmetric, positive semi-definite kernel functions, and for simplicity, assume that each implements dot products in some finite-dimensional space, so that there are vector mappings $\phi_1 : \mathcal{X} \to \mathbb{R}^{d_1}$ and $\phi_2 : \mathcal{X} \to \mathbb{R}^{d_2}$ for some $d_1, d_2 \in \mathbb{Z}_+$ such that

$$K_1(x, x') = \phi_1(x)^\top \phi_1(x') , \quad K_2(x, x') = \phi_2(x)^\top \phi_2(x') \quad \forall x, x' \in \mathcal{X} .$$

For each of the following functions $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, determine whether or not K is a valid kernel. In each case, justify your answer by either finding a vector mapping $\phi : \mathcal{X} \to \mathbb{R}^d$ for some suitable $d \in \mathbb{Z}_+$ such that $K(x, x') = \phi(x)^\top \phi(x')$ $\forall x, x'$, or explaining why such a mapping cannot exist.

(a) $K(x, x') = c \cdot K_1(x, x')$, where $c > 0$

(b) $K(x, x') = K_1(x, x') + K_2(x, x')$

(c) $K(x, x') = K_1(x, x') - K_2(x, x')$

[1]UCI Machine Learning Repository: `http://archive.ics.uci.edu/ml/datasets/Spambase`

(d) $K(x,x') = K_1(x,x') \cdot K_2(x,x')$

(e) $K(x,x') = K_1(f(x), f(x'))$, where $f : \mathcal{X} \to \mathcal{X}$ is any function.

3. (35 points) **Programming Exercise: Neural Networks for Classification.** Complete the implementation of the `NeuralNetworkClassification` class in `neural_network.py`. You need to implement the following functions, with more detailed instructions in `neural_network.py`:

   - `back_propagate`
   - `predict`

   In this problem, you will implement a neural network learning algorithm for binary classification (with one hidden layer) and will apply it to a real data set. Implement a one-hidden-layer neural network for binary classification using batch gradient descent. You need to implement the class `NeuralNetworkClassification` which inherits from `NeuralNetworkBase` Your neural network object should take as input a training set $(\mathbf{X}, \mathbf{y})$, where $\mathbf{X}$ is an $m \times d$ matrix (m instances, each of dimension d) and $\mathbf{y}$ is an m-dimensional vector (with $y_i \in \{\pm 1\}$ being a binary label associated with the i-th instance in $\mathbf{X}$), a step size (learning rate) parameter η, and the desired number of iterations; it should output a neural network model that can be used to make predictions on new instances. For this problem, you may find it helpful to convert the labels $y_i \in \{\pm 1\}$ to labels $\widetilde{y}_i \in \{0,1\}$ via $\widetilde{y}_i = (y_i + 1)/2$.

   Recall that a neural network model for binary classification produces class probability estimates $\widehat{\eta}(\mathbf{x})$; a one-hidden-layer model with d_1 hidden units can be written as

$$\widehat{\eta}(\mathbf{x}) = g_\sigma(f_{\mathbf{W},\mathbf{b}}(\mathbf{x}))$$

   with

$$f_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \mathbf{W}^{(2)} g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + b^{(2)} ,$$

   where $\mathbf{W}^{(1)} \in \mathbb{R}^{d_1 \times d}$ is a matrix of weights connecting the input layer to the hidden layer; $\mathbf{W}^{(2)} \in \mathbb{R}^{1 \times d_1}$ is a row vector of weights connecting the hidden layer to the output layer; $\mathbf{b}^{(1)} \in \mathbb{R}^{d_1}$ is a vector of bias terms associated with the hidden units; $b^{(2)}$ is a bias term associated with the output unit; g is the activation function used by the hidden units; and g_σ is the logistic sigmoid function. The optimization objective is given by

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{m}\sum_{i=1}^{m}\big(- \widetilde{y}_i \ln(\widehat{\eta}(\mathbf{x}_i)) - (1 - \widetilde{y}_i) \ln(1 - \widehat{\eta}(\mathbf{x}_i))\big) .$$

   Denoting

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{a}^{(1)} = g(\mathbf{z}^{(1)}) ,$$

   the derivatives of the objective with respect to the parameters can be written as

$$\begin{aligned}
\frac{\partial J(\mathbf{W},\mathbf{b})}{\partial w^{(1)}_{jk}} &= \frac{1}{m}\sum_{i=1}^{m}\Big(g_\sigma\big(f_{\mathbf{W},\mathbf{b}}\mathbf{x}_i)\big) - \widetilde{y}_i\Big) \cdot w^{(2)}_{1j} \cdot g'(z^{(1)}_{ij}) \cdot x_{ik} \\
\frac{\partial J(\mathbf{W},\mathbf{b})}{\partial b^{(1)}_{j}} &= \frac{1}{m}\sum_{i=1}^{m}\Big(g_\sigma\big(f_{\mathbf{W},\mathbf{b}}(\mathbf{x}_i)\big) - \widetilde{y}_i\Big) \cdot w^{(2)}_{1j} \cdot g'(z^{(1)}_{ij}) \\
\frac{\partial J(\mathbf{W},\mathbf{b})}{\partial w^{(2)}_{1j}} &= \frac{1}{m}\sum_{i=1}^{m}\Big(g_\sigma\big(f_{\mathbf{W},\mathbf{b}}(\mathbf{x}_i)\big) - \widetilde{y}_i\Big) \cdot a^{(1)}_{ij} \\
\frac{\partial J(\mathbf{W},\mathbf{b})}{\partial b^{(2)}} &= \frac{1}{m}\sum_{i=1}^{m}\Big(g_\sigma\big(f_{\mathbf{W},\mathbf{b}}(\mathbf{x}_i)\big) - \widetilde{y}_i\Big) .
\end{aligned}$$

You are provided with a spam classification data set, where each instance is an email message represented by 57 features and is associated with a binary label indicating whether the email is spam (+1) or non-spam (−1).[2] The data set is divided into training and test sets; there are 250 training examples and 4351 test examples. The goal is to learn from the training set a classifier that can classify new email messages in the test set. (See folder `P3/Spam-Dataset` for relevant files.)

(a) **Synthetic data set**

i. **Sigmoid units.** Use your implementation of the neural network learning algorithm to learn a one-hidden-layer neural network classification model from the given training data. Take all d_1 hidden units to have a **logistic sigmoid** activation function. Consider d_1 in the range $\{1, 5, 10, 15, 25, 50\}$, and perform 5-fold cross validation to select d_1 from this range (the training and test data for each fold are provided in separate folders in `P3/Synthetic-Dataset /CrossValidation`). Show a plot of the training, test, and cross-validation errors as a function of the number of hidden units d_1. Use the following settings:

Number of hidden nodes d_1	$\{1, 5, 10, 15, 25, 50\}$
Initial **W** and **b**	For each d_1, use files provided in the corresponding folder under `P3/Synthetic-Dataset/InitParams/sigmoid`
Number of iterations	8000
Learning rate	0.1

ii. **ReLU units.** Use your implementation of the neural network learning algorithm to learn a one-hidden-layer neural network classification model from the given training data. Take all d_1 hidden units to have a **ReLU** activation function. Consider d_1 in the range $\{1, 5, 10, 15, 25, 50\}$, and perform 5-fold cross validation to select d_1 from this range (the training and test data for each fold are provided in separate folders in `P3/Synthetic-Dataset/CrossValidation`). Show a plot of the training, test, and cross-validation errors as a function of the number of hidden units d_1. Use the following settings:

Number of hidden nodes d_1	$\{1, 5, 10, 15, 25, 50\}$
Initial **W** and **b**	For each d_1, use files provided in the corresponding folder under `P3/Synthetic-Dataset/InitParams/relu`
Number of iterations	8000
Learning rate	0.01

iii. Summarize your results in the following table:

Algorithm	Parameters selected	Training Error	Test Error	Training Time
Neural net, ReLU units	$d_1 =$			
Neural net, sigmoid units	$d_1 =$			

(b) **Real data set**

i. **Sigmoid units.** Repeat the steps in (a)(i), using the following settings:

Number of hidden nodes d_1	$\{1, 5, 10, 15, 25, 50\}$
Initial **W** and **b**	For each d_1, use files provided in the corresponding folder under `P3/Spam-Dataset/InitParams/sigmoid`
Number of iterations	8000
Learning rate	0.12

[2]UCI Machine Learning Repository: `http://archive.ics.uci.edu/ml/datasets/Spambase`

ii. **ReLU units.** Repeat the steps in (a)(ii), using the following settings:

Number of hidden nodes d_1	$\{1, 5, 10, 15, 25, 50\}$
Initial $\mathbf{W}$ and $\mathbf{b}$	For each d_1, use files provided in the corresponding folder under `P3/Spam-Dataset/InitParams/relu`
Number of iterations	8000
Learning rate	0.1

iii. Summarize your results in the following table:

Algorithm	Parameters selected	Training Error	Test Error	Training Time
Neural net, ReLU units	$d_1 =$			
Neural net, sigmoid units	$d_1 =$			

Python Instructions:

(a) *By default,* `NeuralNetworkClassification` *inherits* `fit` *function from* `NeuralNetworkBase`*. This function has been implemented for you.*

(b) *You may use Python's* `time` *library to track your training elapsed time.*

(c) *You may use the learned neural network to predict values for* `numpy.linspace(0,1)` *to draw your learned non-linear model.*

(d) *As much as you can, avoid computing matrices entry by entry and using nested for loops. Matrix-matrix and matrix-vector multiplication is the way to go.*

(e) *Include all of the codes you used for experiments and plotting in* `PS2-P3.py` *and follow submission instructions when submitting.*

4. (15 points) **Convolutional Neural Networks.** A convolutional neural network (CNN) is a type of feed-forward neural network that is often used in computer vision and image processing tasks; in such networks, the hidden layers often involve performing *convolution* operations. Figure 1 shows an example of a simple one-hidden layer CNN that takes a 5×4 pixel image as an input, and applies a single 3×3 convolutional *filter* (sometimes a called convolutional *kernel*) to produce a 3×2 "image" (hidden representation) in the hidden layer.

In general, *convolution* is a mathematical operation on two functions that produces a third function expressing how the shape of one is modified by the other. In case of the convolution operation between an image and a filter, it defines the process of adding each element of the image to its local neighbors, weighted by the filter values. Specifically, consider an image $\mathbf{X}$ represented as a 2D matrix $x_{i,j}$, $i = 0, \ldots, d_h - 1$, $j = 0, \ldots, d_w - 1$, where d_h is the image height and d_w is the width in pixels. Given an $M \times N$ filter $\mathbf{W}$, the convolution operation $\mathbf{X} * \mathbf{W}$ defines a new "image" $\mathbf{Z}$ as follows:[3]

$$z_{s,t} = [\mathbf{X} * \mathbf{W}]_{s,t} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{s+m,t+n}\, w_{m,n} \tag{1}$$

So in each $M \times N$ block of pixels in the input image $\mathbf{X}$, each pixel gets multiplied by the corresponding entry of the filter and then such values get summed up. The sum becomes a new "pixel" (or hidden node value) in $\mathbf{Z}$.

[3]One can find similar definitions of the convolution operation but with opposite signs, e.g. $x(s-m,\ t-n)$ or different index bounds, e.g. index m running from $-M/2$ to $M/2$. Such definitions are equivalent and depend on how the filter is centered, if the filter is flipped and how the indices are defined in general (some prefer to define the center of the image/filter as (0,0) point).

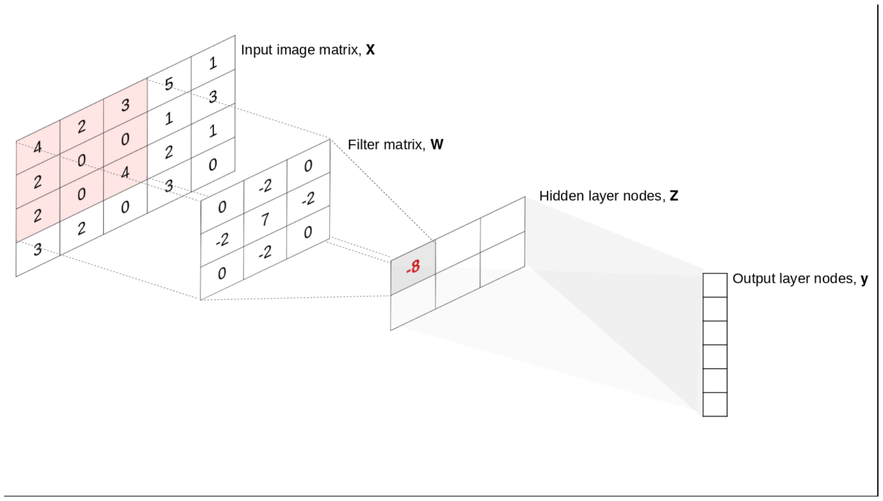

Figure 1: Example of a simple one-hidden-layer CNN that involves convolution of a 5×4 input image with a given 3×3 filter.

For our particular example presented in the Figure 1,

$$z_{0,0} = \sum_{m=0}^{2} \sum_{n=0}^{2} x_{m,n} w_{m,n} = 2 \cdot (-2) + 2 \cdot (-2) = -8$$

(a) Perform the convolution $\mathbf{Z} = \mathbf{X} * \mathbf{W}$ between the given image $\mathbf{X}$ and the given filter $\mathbf{W}$ in the Figure 1, i.e. obtain the remaining $z_{s,t}$ values. Show your calculations.

(b) Can you say what a convolutional filter "intuitively" does? E.g. what does the 3×3 filter in Figure 1 do – when does it produce high values?

(c) The CNN in Figure 1 has only one 3×3 filter $\mathbf{W}$ that is applied to the 5×4 input image $\mathbf{X}$ to produce the hidden layer containing $3 \times 2 = 6$ hidden nodes. To train the CNN, one would therefore need to learn the $3 \times 3 = 9$ parameters defining the filter $\mathbf{W}$ (in addition to any parameters associated with the output layer). If the CNN had six 3×3 filters, how many parameters would need to be learned for the input-to-hidden layer (assume no bias terms)? How many hidden nodes would there be?

(d) Now consider a larger input image of size 600×500 pixels. Again, consider a CNN with six 3×3 filters. How many hidden nodes would there be? How many parameters would need to be learned for the input-to-hidden layer (again, no bias terms)? If instead of the CNN we used a fully connected neural network with 15 hidden nodes (i.e. with independent connections between all pairs of input and hidden nodes), then how many parameters would need to be learned for the input-to-hidden layer (no bias terms)? Based on this, what can you say about some of the advantages of using CNNs when working with images as compared to fully connected feed-forward neural networks?