CS 4100: Artificial Intelligence (Fall 2023) — Lawson L.S. Wong
Northeastern University — Due November 17, 2023

Programming Assignment 4: Machine Learning

1. **5 points.** *Linear regression.*
   Implement the methods in `model.PolynomialRegressionModel` for the linear regression model with polynomial features, presented in Lecture 15. Specifically, we are trying to fit $y = h_\theta(x)$, where $x, y \in \mathbb{R}$, using powers of x up to a specified degree as features.

   - `__init__(degree, learning_rate)`: Instantiate class variables here.
   - `get_features(x)`: Returns a list of features for input x.
     In this model, this is $[x, x^2, \ldots, x^p]$, where p is the degree specified in the class constructor.
     Consider including a "dummy" feature with value 1 (for any input) to capture the bias term.
   - `get_weights()`: Returns the current parameter values.
   - `hypothesis(x)`: Returns $h_\theta(x)$.
   - `predict(x)`: Returns the prediction y for input x.
     In regression, $y = h_\theta(x)$, so this method simply calls `hypothesis(x)`.
   - `loss(x, y)`: Returns the loss on a single sample $L(h_\theta(x), y)$.
     In linear regression, this is the squared-error loss.
   - `gradient(x, y)`: Returns a list of partial derivatives of the loss function with respect to each weight, evaluated at the sample (x, y) and the current weights θ.
     Mathematically, this is the *gradient*: $\left[\frac{\partial L(h_\theta(x),y)}{\partial \theta_0}, \frac{\partial L(h_\theta(x),y)}{\partial \theta_1}, \ldots, \frac{\partial L(h_\theta(x),y)}{\partial \theta_p}\right]$
   - `train(dataset, evalset)`: Performs the training loop using the supplied `dataset` (ignore `evalset` for now). Use stochastic gradient descent (SGD) (i.e., one weight update per sample).
     Choose an appropriate number of iterations for the training loop: set it as a constant that you think is high enough for the data in the next question, instead of explicitly checking for convergence.

2. **4 points.** *Linear regression: Application and analysis.*
   This question involves both programming and written responses. Include generated plots in your write-up.

   (a) Apply linear regression on the `sine_train` dataset, with degree $p = 1$ and learning rate $\alpha = 10^{-4}$.
   Report the final hypothesis found and the average loss of this hypothesis on the training dataset.
   Plot the data and the hypothesis found.
   Helpful functions: `util.get_dataset`, `util.Dataset.compute_average_loss`, `util.RegressionDataset.plot_data`

   (b) Plot the *training loss curve*, which plots the average **training** loss at regular points during training.
   To obtain this curve, you will need to modify `model.PolynomialRegressionModel.train` to keep track of relevant statistics during training.
   Helpful functions: `util.RegressionDataset.plot_loss_curve`
   Hint: If evaluating the average loss at every training iteration is too slow, you may choose to only evaluate once every T iterations, as long as the curve does not change qualitatively. Similarly, the `step` argument in `util.Dataset.compute_average_loss` allows you to compute the average loss on a subset of the dataset.

   (c) To apply linear regression, we had to manually specify the degree and learning rate. Use the *validation dataset* `sine_val` to perform a *hyperparameter search* over the degree and learning rate hyperparameters.
   For different combinations of (degree, learning rate), apply linear regression on the `sine_train` dataset, then evaluate the final result with the `sine_val` dataset (which was not used during training).
   Try at least 10 combinations. Which combination was the best (lowest average validation loss)?
   Report the combinations you tried, and the average training and validation losses on each.
   Hint: For the learning rate, it is often useful to search at regular intervals in the log domain.

   (d) **(Optional)** After identifying the best set of hyperparameters using the validation set, typically we would re-train the model with the combined training + validation datasets (`sine_train_val`), since the validation data is usually considered part of the training set that is only temporarily held out.
   Re-train the model on the `sine_train_val` dataset with the best hyperparameter combination you found in part (c), report and plot the hypothesis found, and plot the new training loss curve.
   Evaluate the model on the *test set* `sine_test`: report the average test loss and plot the test loss curve.

3. **5 points.** *Binary logistic regression.*
Implement the methods in `model.BinaryLogisticRegressionModel` for the binary logistic regression model with image-pixel features, presented in Lecture 17. Specifically, we are trying to classify MNIST digits between two classes (e.g., "0" vs. "1"), using the image pixels as the only features.
MNIST images are of size 28×28, with grayscale values between 0.0 and 1.0.

   - `__init__(num_features, learning_rate)`: Instantiate class variables here.
   - `get_features(x)`: Returns a list of features for input x.
     For MNIST, x is a list of lists, where each inner list corresponds to a row of 28 pixels.
   - `get_weights()`: Returns the current parameter values.
   - `hypothesis(x)`: Returns $h_\theta(x) = \mathbb{P}(y = 1|x; \theta)$.
   - `predict(x)`: Returns the prediction y for input x.
     In binary classification, y must be either 0 or 1 (i.e., not the same as `hypothesis(x)`).
   - `loss(x, y)`: Returns the loss on a single sample $L(h_\theta(x), y)$.
     In binary classification, this is the cross-entropy loss.
   - `gradient(x, y)`: Returns a list of partial derivatives of the loss function with respect to each weight.
   - `train(dataset, evalset)`: Performs the SGD training loop using the supplied `dataset`.

4. **4 points.** *Binary logistic regression: Application and analysis.*
This question involves both programming and written responses. Include generated plots in your write-up.

   Extract the provided MNIST dataset `mnist_png.tar.gz`. By default, the starter code assumes that the `mnist_png` directory will be located in the same directory as the other files (`model.py` and `util.py`).
   If that is not the case, you can change the path to `mnist_png` in `util.get_dataset`.

   Hint: Loading the MNIST dataset can take a while ($\sim$1 minute or less). The loading time appears to decrease significantly after the first time, perhaps due to caching. If loading the data is too slow, you can temporarily create a smaller MNIST dataset by removing a fraction of the images from each folder in the dataset.
   During development and debugging, it is advisable to have a sufficiently small dataset / low number of training iterations so that model training completes in under 1 minute, and ideally within a few seconds.

   (a) Apply logistic regression on the `mnist_binary_train` dataset, which only contains MNIST data for the "7" and "9" classes. Note that the `BinaryMNISTDataset` object labels these classes as "0" and "1" respectively. Plot the training *accuracy* curve (not loss) and the *test* accuracy curve (on `mnist_binary_test`).
   Helpful functions: `util.Dataset.compute_average_accuracy`, `util.MNISTDataset.plot_accuracy_curve`
   Hint: To obtain the test accuracy curve, you will need to evaluate the average **test** accuracy at regular points during training. This requires `model.BinaryLogisticRegressionModel.train` to have access to the test set `mnist_binary_test`, which can be passed in via the `evalset` argument.

   (b) Plot the *confusion matrix*, which for a binary classification is the following matrix of counts:

Num. actual class 0, predicted class 0	Num. actual class 0, predicted class 1
Num. actual class 1, predicted class 0	Num. actual class 1, predicted class 1

   Entries along the diagonal are correct predictions, whereas off-diagonal entries are incorrect.
   The confusion matrix categorizes the errors made and is more informative than a single accuracy number. Assuming the model is correct on most predictions, the entries on the diagonal will dominate the off-diagonal entries. For the purposes of analyzing errors, diagonal entries are therefore typically omitted.
   Helpful functions: `util.MNISTDataset.plot_confusion_matrix`

   (c) Plot the non-bias-term weights. How should we interpret the learned model? Does it make sense?
   Helpful functions: `util.MNISTDataset.plot_image`

   (d) Plot and inspect at least 10 erroneous predictions (or all errors, if fewer than 10).
   Why did the model make errors on these cases? What does the model fail to capture?
   Suggest some additional features that may help the model make correct predictions on these error cases.
   Helpful functions: `util.MNISTDataset.plot_image`

5. **5 points.** *Multinomial logistic regression.*
   Implement the methods in `model.MultiLogisticRegressionModel` for the multinomial logistic regression model with image-pixel features, presented in Ex4 Q6. Specifically, we are trying to classify MNIST digits between all 10 classes ("0" to "9"), using the image pixels as the only features.

   - `__init__(num_features, num_classes, learning_rate)`: Instantiate class variables here.
   - `get_features(x)`: Returns a list of features for input x.
   - `get_weights()`: Returns the current parameter values.
   - `hypothesis(x)`: Returns a list of probabilities that x belongs to each class k.
     This is $[h_1(x), h_2(x), \ldots, h_K(x)]$, where $h_k(x) = \mathbb{P}(y = k|x; \theta_k)$, and K is the number of classes.
   - `predict(x)`: Returns the prediction y for input x.
     In multi-class classification, y must be an integer between 1 and K, where K is the number of classes.
   - `loss(x, y)`: Returns the loss on a single sample $L(h_\theta(x), y)$.
   - `gradient(x, y)`: Returns a list of partial derivatives of the loss function with respect to each weight.
   - `train(dataset, evalset)`: Performs the SGD training loop using the supplied `dataset`.

6. **2 points.** *Multinomial logistic regression: Application.*
   This question involves both programming and written responses. Include generated plots in your write-up.

   Apply multinomial logistic regression on the `mnist_train` dataset.
   Plot the training and test accuracy curves (on `mnist_train` and `mnist_test` respectively).
   Plot the confusion matrix and the learned weights. Do the results seem reasonable?

   Hint: With all 10 classes, the MNIST dataset contains 60K training images and 10K test images. Training and testing on the entire dataset can take a long time. You should be able to get good performance just by training on a fraction of the data. See the final section of the Lecture 17 slides to get a sense of what to expect.

Submit code (`model.py` and `util.py`) and write-up to PA4 on Canvas.

7. **Extra credit.** Implement ℓ_2-regularized linear regression (see Ex4 Q4).
   Apply this model on the data from Q2 and analyze the results.
   How does it compare with standard linear regression? Do the results match the analysis from Ex4 Q4?

8. **Extra credit.** Apply binary logistic regression on the data from Ex4 Q5(a) (see `util.get_dataset`).
   Verify your answers from Ex4 Q5(a) – do your features and parameters classify all data points correctly?
   Using the features you defined, what weights does logistic regression learn? How do they compare with parameters you identified? Do the learned models classify all data points correctly?

9. **Extra credit.** Perform the same error analysis from Q4(d) on the multinomial logistic regression setting (Q6).
   Implement the features you come up with. Do they actually lead to a more accurate model?
   Using only the 28×28 image pixels as features, we were able to obtain 91.93% test accuracy using multinomial logistic regression (see final slide of Lecture 17). Can you outperform our model?