CS M146: Introduction to Machine Learning
Prof. Aditya Grover

UCLA Fall Quarter 2023
28th September 2023

Homework #1

Due: 16th October 2023, Monday, before 11:59 pm

Problem 1 (Short answer question)

[10 points total] For each of the statements below, state True (T) or False (F), or pick the correct option. Explain your answer in 1-2 sentences. For every part below, 1 point for correct assertion and 1 point for correct explanation.

(a) T/F: Ridge regression does not have a closed-form solution. F

(b) T/F: If we have m values for a hyperparameter of a classifier and we use k fold cross-validation for hyperparameter tuning, then we train the classifier mk times from scratch. T

(c) T/F: Consider a linear function basis $\phi(\mathbf{x}) = [1, \log x_1, x_2^3 x_3]$ where $\mathbf{x} \in \mathbb{R}^3$. There exists a closed form solution for minimizing the mean squared error for the hypothesis $h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^T \phi(\mathbf{x})$. If True, state the solution. Otherwise, give a justification. T

(d) Which of the following is a major drawback of linear regression?

A. It assumes a linear relationship between the independent and dependent variables

B. It can handle only numerical data

C. It is not suitable for high-dimensional data

D. It is prone to overfitting

E. None of the above

E

(e) Which of the following is a common method for preventing overfitting in machine learning?

A. Increasing the complexity of the model

B. Decreasing the size of the training data set

C. Regularization

D. Using a simpler evaluation metric

E. None of the above

C

Problem 2 (Mean Absolute Error)

[10 points total]

In class, we considered optimizing the Mean Square Error (MSE) loss. In practice, there are other choices of loss functions as well. For this problem, we will consider linear regression using Mean Absolute Error (MAE) as our loss function. Specifically, the MAE loss is given as:

$$J_{\text{MAE}}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} |\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}|. \tag{1}$$

(a) **[4 points]** Derive the partial derivative $\frac{\partial J_{\text{MAE}}(\boldsymbol{\theta})}{\partial \theta_j}$.

Hint: For this question, you don't need to worry about $J = 0$. You can use the vector-valued *sign* function s.t. $sign(\hat{\boldsymbol{y}} - \boldsymbol{y}) = \begin{cases} +1 & \hat{y}^{(i)} \geq y^{(i)}, \\ -1 & \text{otherwise.} \end{cases}$

Solution:

$$\begin{aligned} \frac{\partial J_{\text{MAE}}(\boldsymbol{\theta})}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j}\frac{1}{n}\sum_{i=1}^{n} |\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}| \\ &= \frac{\partial}{\partial \theta_j}\frac{1}{n}\sum_{i=1}^{n} sign(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)})(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}) \\ &= \frac{1}{n}\sum_{i=1}^{n} sign(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)} \end{aligned}$$

(b) **[2 points]** Write the vectorized solution for the gradient of the loss function, i.e., $\nabla_{\boldsymbol{\theta}} J_{\text{MAE}}(\boldsymbol{\theta})$.

Solution:

$$\nabla_{\boldsymbol{\theta}} J_{\text{MAE}}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} sign(\boldsymbol{\theta}^T \boldsymbol{x}^{(i)} - y^{(i)}) \boldsymbol{x}^{(i)}$$

(c) **[4 points]** Given the following dataset of 8 points (listed without bias feature $x_0^{(i)}$), what is the value of of $J_{\text{MAE}}(\boldsymbol{\theta})$ and $J_{\text{MSE}}(\boldsymbol{\theta})$ at $\boldsymbol{\theta} = [1.0, 1.0, 1.0]^T$? How about their gradients $\nabla_{\boldsymbol{\theta}} J_{\text{MAE}}(\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} J_{\text{MSE}}(\boldsymbol{\theta})$?

i	1	2	3	4	5	6	7	8
$\boldsymbol{x}^{(i)}$	[4,0]	[1,1]	[0,1]	[-2,-2]	[-2,1]	[1,0]	[5,2]	[3,0]
$y^{(i)}$	12	3	1	6	3	6	8	7

Solution: $J_{\text{MAE}}(\boldsymbol{\theta}) = 3.375$, $J_{\text{MSE}}(\boldsymbol{\theta}) = 10.3125$,
$\nabla_{\boldsymbol{\theta}} J_{\text{MAE}}(\boldsymbol{\theta}) = [-0.25, 0.25, 0.625]^T$, $\nabla_{\boldsymbol{\theta}} J_{\text{MSE}}(\boldsymbol{\theta}) = [-3.125, -2.125, 2]^T$

Problem 3 (Programming Exercise: Polynomial Regression)

[34 points total] In this exercise, you will work through linear and polynomial regression. Our data consists of (scalar) inputs $x^{(i)} \in \mathbb{R}$ and outputs $y^{(i)} \in \mathbb{R}, i \in \{1, \ldots, n\}$, which are related through a target function $y^{(i)} = f(x^{(i)})$. Your goal is to learn a linear predictor $h_{\boldsymbol{\theta}}(x)$ that best approximates $f(x)$.

---

code and data

- code : `regression.py, Notebook.ipynb`
- data : `regression_train.csv`, `regression_valid.csv`, `regression_test.csv`

Visualization

As we learned last week, it is often useful to understand the data through visualizations. For this data set, you can use a scatter plot to visualize the data since it has only two properties to plot (x and y).

(a) **[2 points]** Visualize the training and test data using the `plot_data(...)` function. Simply by looking at the dataset, can you make an educated guess on the effectiveness of linear regression in predicting the label y using the default feature x?

Solution: From the training data, it does not show a strong linear relationship unless we are assuming a very large noise. We would expect simple linear regression to be a poor predictor, but polynomial regression might work.

Linear Regression

Recall that linear regression attempts to minimize the objective function

$$J(\boldsymbol{\theta}) = \frac{1}{2n} \sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)} \right)^2.$$

In this problem, we will use the matrix-vector form where

$$\mathbf{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{pmatrix}, \qquad \boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}^{(1)T} \\ \boldsymbol{x}^{(2)T} \\ \vdots \\ \boldsymbol{x}^{(n)T} \end{pmatrix}, \qquad \boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{pmatrix}$$

and each instance $\boldsymbol{x}^{(i)} = \left(1, x_1^{(i)}, \ldots, x_d^{(i)}\right)^T$.

Rather than working with this fully generalized, multivariate case, let us start by considering a simple linear regression model with $d = 1$. Similar to what we did in class, we first augment an extra dimension $x_0 = 1$ to vectorize our hypothesis.

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{x} = \theta_0 + \theta_1 x_1$$

The file `regression.py` contains the skeleton code for the class `Regression`. Objects of this class can be instantiated as `model = Regression(m)` where m is the degree of the polynomial feature vector we will touch upon soon. Polynomial feature vectors are a class of vectors having the form $[1, x_1^{(i)}, {x_1^{(i)}}^2, \ldots, {x_1^{(i)}}^m]^T$. Setting $m = 1$ instantiates a feature vector for instance i as $[1, x_1^{(i)}]^T$.

(b) **[4 points]** Modify `get_poly_features()` in `Regression.py` for the case $m = 1$ to create the matrix $\boldsymbol{X}$ for a simple linear model. Include a screenshot of your code in the writeup.

Solution:

```
def get_poly_features(self, X):
    """
    Inputs:
    - X: A numpy array of shape (n,1) containing the data.
    Returns:
    - X_out: an augmented training data as an mth degree feature vector
    e.g. [1, x, x^2, ..., x^m], x \in X.
    """
    n,d = X.shape
    m = self.m
    X_out= np.zeros((n,m+1))
    if m==1:
        # ============================================================ #
        # YOUR CODE HERE:
        # IMPLEMENT THE MATRIX X_out with each entry = [1, x]
        # ============================================================ #
        X_out=np.c_[np.ones((n,1)),X]
        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #
```

Figure 1: Implementation for feature extension.

(c) **[4 points]** Before tackling the harder problem of training the regression model, complete `predict()` in `Regression.py` to predict $\mathbf{y}$ from $\boldsymbol{X}$ and $\boldsymbol{\theta}$. Include a screenshot of your code in the writeup.

Solution: See Fig. 2

(d) In the lecture, we studied optimizing the loss for linear regression through gradient descent (GD) and its extension called mini-batch gradient descent (MBGD).

Recall that the parameters of our model are the θ_j values. These are the values we will iteratively adjust to minimize $J(\boldsymbol{\theta})$. In MBGD, each iteration we sample a batch of B points D_B at random from the full dataset D without replacement and perform the update

$$\theta_j \leftarrow \theta_j - \alpha \frac{1}{B} \sum_{(\boldsymbol{x}^{(i)}, y^{(i)}) \in D_B} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^i) - y^{(i)} \right) x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j\text{)}.$$

With each step of MBGD, we expect our updated parameters θ_j to gradually come closer to the parameters that will achieve the lowest value of $J(\boldsymbol{\theta})$.

```python
def predict(self, X):
    """
    Inputs:
    - X: n x 1 array of training data.
    Returns:
    - y_pred: Predicted targets for the data in X. y_pred is a 1-dimensional
      array of length n.
    """
    y_pred = np.zeros(X.shape[0])
    m = self.m
    if m==1:
        # ================================================================ #
        # YOUR CODE HERE:
        # PREDICT THE TARGETS OF X
        # ================================================================ #
        X_out = self.get_poly_f
        y_pred = np.dot(X_out, self.theta)
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
```

(parameter) self: Self@Regression

Figure 2: Implementation of predict().

- **[4 points]** As we perform gradient descent, it is helpful to monitor the convergence by computing the loss, *i.e.*, the value of the objective function J. Complete `loss_and_grad()` to calculate $J(\boldsymbol{\theta})$, and the gradient. Complete `train_LR()` and plot the loss history.
  Solution: Loss: 1.0455416122950603. Gradient: $[1.33142275, 2.65167278]$.

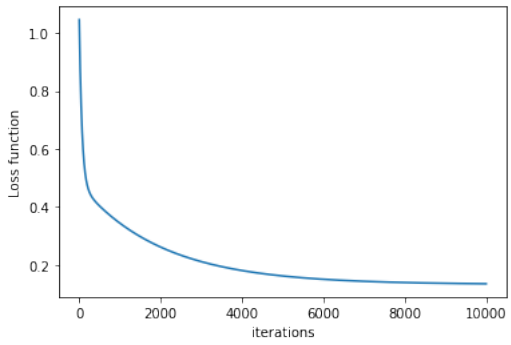

Figure 3: Learning curve with the default learning rate.

We will use the following specifications for the gradient descent algorithm:

  - We run the algorithm for $10,000$ iterations.
  - We will use a fixed step size.

- **[2 points]** So far, you have used a default learning rate (or step size) of $\alpha = 0.01$. Try different $\alpha = 10^{-4}, 10^{-3}, 10^{-1}$, and make a table of the learning rates and the final value of the objective function. Do all the learning rates lead to convergence?
  Solution: When the learning rate is small, the model does not converge.

α	final training loss
10^{-1}	0.1320896910198222
10^{-2}	0.13208969101982218
10^{-3}	0.13458201902518097
10^{-4}	0.34341432816283046

Fig. 4 is the learning curve for demonstration purposes, which is not a requirement.

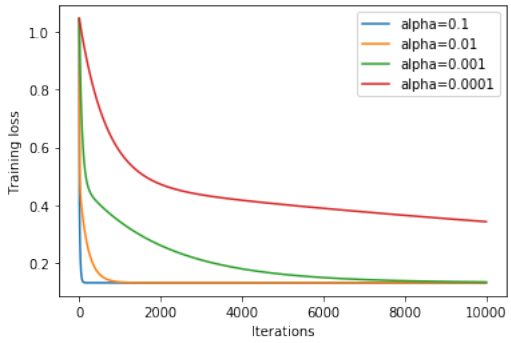

Figure 4: Learning curves of different step sizes.

- **[2 points]** Now let's fix $\alpha = 0.01$. Try different $B = 1, 10, 20, n$, where n is the size of the training dataset. Plot the loss history for different values of B on the same figure (i.e., 4 curves in total for $B = 1, 10, 20, n$).
  Solution: See Fig. 5

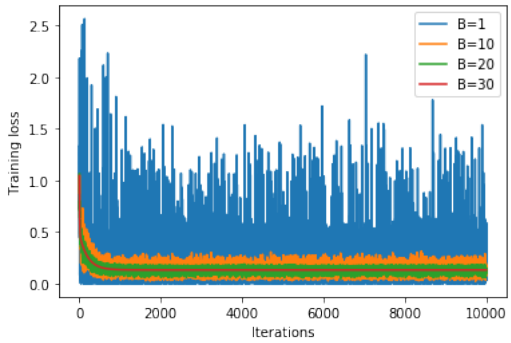

Figure 5: Learning curves of different batch sizes.

(e) **[4 points]** In class, we learned that the closed-form solution to linear regression is

$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\mathbf{y}.$$

Using this formula, you will get an exact solution in one step. There is no iterative repetition of parameter updates like in gradient descent.

- Implement the closed-form solution `closed_form()`.
- Run your `closed_form()` to get solution. Report the result of parameters and the loss. How do they compare to those obtained by MBGD?
  Solution:
  Optimal solution loss 0.1320896910198222.
  Optimal solution weight $[-0.37906992, 0.8852483]$.

Polynomial Regression

Now let us consider the more complicated case of polynomial regression. Here, we first define a polynomial feature map: $\phi(x) = [1, x, x^2, \cdots, x^m]^T$

Consequently, our hypothesis for linear regression is:

$$h_{\boldsymbol{\theta}}(x) = \boldsymbol{\theta}^T \phi(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_m x^m.$$

Note, here we have NOT augmented our input x with an additional dimension, as done previously, as the vectorization can be induced directly after applying the feature map ϕ.

(f) **[6 points]** Recall that linear basis function regression (in this case, using a polynomial basis) can be considered as an extension of linear regression in which we replace our input matrix $\boldsymbol{X}$ with

$$\boldsymbol{\Phi}(\boldsymbol{X}) = \begin{pmatrix} \phi(x^{(1)})^T \\ \phi(x^{(2)})^T \\ \vdots \\ \phi(x^{(n)})^T \end{pmatrix},$$

where $\phi(x)$ is a vector function such that $\phi_j(x) = x^j$ for $j = 0, \ldots, m$.

Update `get_poly_features()` for the case when $m \geq 2$. Submit a screenshot of your code for this question.

For $m = \{0, \ldots, 10\}$, use the closed-form solver to determine the best-fit polynomial regression model on the training data, and with this model, calculate the loss on both the training data and the test data. Generate a plot depicting how loss varies with model complexity (polynomial degree) – you should generate a single plot with training, validation and test error, and include this plot in your writeup. Which degree polynomial would you say best fits the data? Was there evidence of under/overfitting the data? Use your plot to justify your answer.

Solution: Being the best fit means the model performs the best on the validation set, which will be expected to generalize to the test set also. As shown in Fig. 6, 2nd, 3rd, and 4th, 5th degree polynomials fit the data best. For $m > 5$, there is clear evidence of overfitting – the test error increases greatly but training error decreases. For $m < 2$, there is evidence of underfitting – the training error is very high.

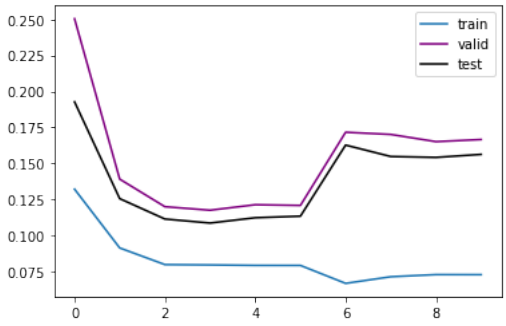

Figure 6: Training, validation and testing error as a function of model complexity.

Regularization

Finally, we will explore the role of regularization. For this problem, we will use L_2-regularization so that our regularized objective function is

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}||\boldsymbol{\theta}_{[1:m]}||^2,$$

again optimizing for the parameters $\boldsymbol{\theta}$.

(g) **[6 points]** Modify `loss_and_grad()` to incorporate ℓ_2-regularization using the closed-form solution. Submit a screenshot of your code for this question. Use your updated solver to find the coefficients that minimize the error for a tenth-degree polynomial ($m = 10$) given regularization factor $\lambda = 0, 10^{-8}, 10^{-7}, \ldots, 10^{-1}, 10^{0}$. Now we want to check how λ affects the loss (unregularized) on the training data, validation data, and test data. Generate a plot depicting how the loss error varies with λ (for your x-axis, let $x = [1, 2, \ldots, 10]$ correspond to $\lambda = [0, 10^{-8}, 10^{-7}, \ldots, 10^{0}]$ so that λ is on a logarithmic scale, with regularization increasing as x increases), and include this plot in your writeup. Which λ value appears to work best?

Solution: Working the best means having the least testing error. In this case, the influence of the regularization parameter is almost negligible on the testing error, as shown in Fig. 7. $\lambda = 10^{-1}$ appears to have the least testing error. You need to be cautious not to plot the regularized loss.

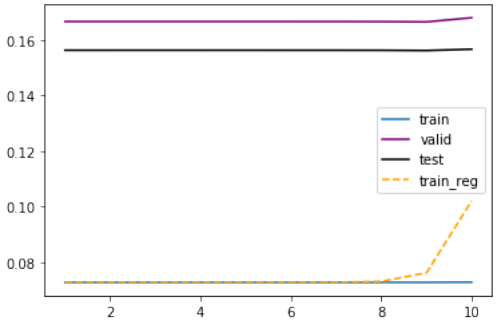

Figure 7: Training, validation and testing error as a function of the regularization parameter. The orange curve is not a correct answer.