CSCI 740 **Spring 2021**

Assignment 2 : Gradient Descent

Due Date: 11:30 pm, March 21, 2021

Note: Read the entire assignment completely and think about how the different parts are connected before working on the solution.

This assignment is intended to build the following skills:

1. Implementation of the iterative optimization Gradient Descent algorithms for solving a **Linear Regression problem**
2. Implementation of various regularization techniques
3. Polynomial regression
4. Learning curve

Read and Follow Assignment Instructions Carefully

1. This is an individual assignment. All work submitted must be your own.
2. First read the entire assignment description to get the big picture; make notes down the control flow, expected functionality of the various methods and why you are being asked to implement specific items.
3. Read and understand Jupyter Workbooks 6, 7 and 8.
4. You are Not **allowed** to use any Scikit-Learn **models or functions.**
5. Download the wine quality dataset from:

   http://archive.ics.uci.edu/ml/datasets/Wine+Quality

   You will be using the white wine dataset: "winequality-white.csv"

6. Deliverables:
   - **a.** The code and answers for Sections A, B and C should be written in a Jupyter notebook named `<lastname>_<firstname>_assignment2.ipynb.
   - **b.** The answers for Section D should be written in a pdf file named `<lastname>_<firstname>_assignment2.pdf`.
   - **c.** These two files **should then be compressed and submitted as `<lastname>_<firstname>_assignment2.zip`**

7. Make sure you copy each question with the question number as a Markdown Cell in Jupyter and have the code response right below it (as shown in the workbooks assigned in class). Points will be deducted if it is difficult to locate the question and response.
8. Make sure you comment your code. Points will be deducted if code logic is not apparent.
9. The written sections will be graded on correctness and preciseness while programming code will be graded on structure, implementation and correctness.

Score Distribution

Part A (Model Code): 40 points
Part B (Data Processing): 5 points
Part C (Model Evaluation): 30 points
Part D (Written Report): 25 pts

Total: 100 points

Part A: Model Code [40 pts]

1. Implement the following function that generates the polynomial and interaction features for a given degree of the polynomial. **[5 pts]**

polynomialFeatures(X, degree)

Arguments:

X : ndarray
A numpy array with rows representing data samples and columns representing features (d-dimensional feature).

degree : integer
The degree of the polynomial features. Default = 1.

Returns:

A new feature matrix consisting of all polynomial combinations of the features with degree equal to the specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are [a, b, a^2, ab, b^2].

2. Implement the following function to calculate and return the mean squared error (mse) of two vectors. **[2 pts]**

mse(Y_true, Y_pred)

Arguments:

Y_true : ndarray
1D array containing data with "float" type. True y values.

Y_pred : ndarray
1D array containing data with "float" type. Values predicted by your model.

Returns:

cost : float
It returns a float value containing mean squared error between Y_true and Y_pred.
Note: these 1D arrays should be designed as column vectors.

3. Implement the following function to compute training and validation errors. It will be used to plot **learning curves**. The function takes the feature matrix X (usually the training data matrix) and the training size (from the "train_size" parameter) and by using cross-validation computes the average mse for the training fold and the validation fold. It iterates through the entire X with an increment step of the "train_size". **[10 pts]**

For example, if there are 50 samples (rows) in X and the "train_size" is 10, then the function will start from the first 10 samples and will successively add 10 samples in each iteration. During each iteration it will use k-fold cross-validation to compute the average mse for the training fold and the validation fold. Thus, for example, for 50 samples there will be 5 iterations (on 10, 20, 30, 40, and 50 samples) and for each iteration it will compute the cross-validated average mse for the training and the validation fold. For training the model (using the "fit" method) it will use the model parameters from the function argument. The function will return two arrays containing training and validation **root-mean-square error** (rmse) values.

learning_curve(model, X, Y, cv, train_size=1, learning_rate=0.01, epochs=1000, tol=None, regularizer=None, lambda=0.0, **kwargs)

Arguments:

model : object type that implements the "fit" and "predict" methods. An object of that type which is cloned for each validation.

X : ndarray
A numpy array with rows representing data samples and columns representing features.

Y : ndarray
A 1D numpy array with labels corresponding to each row of the feature matrix X.

cv : int
integer, to specify the number of folds in a k-fold cross-validation.

train_sizes : int or float
Relative or absolute numbers of training examples that will be used to generate the learning curve. If the data type is float, it is regarded as a fraction of the maximum size of the training set (that is determined by the selected validation method), i.e. it has to be within (0, 1]. Otherwise it is interpreted as absolute sizes of the training sets.

learning_rate : float
It provides the step size for parameter update.

epochs : int
The maximum number of passes over the training data for updating the weight vector.

tol : float or None
The stopping criterion. If it is not None, the iterations will stop when (error > previous_error - tol). If it is None, the number of iterations will be set by the "epochs".

regularizer : string
The string value could be one of the following: l1, l2, None.
If it's set to None, the cost function without the regularization term will be used for computing the gradient and updating the weight vector. However, if it's set to l1 or l2, the appropriate regularized cost function needs to be used for computing the gradient and updating the weight vector.

lambda : float
It provides the regularization coefficient. It is used only when the "regularizer" is set to l1 or l2.

Returns:

train_scores : ndarray
root-mean-square error (rmse) values on training sets.

val_scores : ndarray
root-mean-square error (rmse) values on validation sets.

5. Implement a **Linear_Regression** model class. It should have the following three methods. Note the that "fit" method should implement the **batch gradient descent** algorithm. **[23pts]**

a. fit(self, X, Y, learning_rate=0.01, epochs=1000, tol=None, regularizer=None, lambda=0.0, **kwargs)

Arguments:

X : ndarray
A numpy array with rows representing data samples and columns representing features.

Y : ndarray
A 1D numpy array with labels corresponding to each row of the feature matrix X.

learning_rate : float
It provides the step size for parameter update.

epochs : int
The maximum number of passes over the training data for updating the weight vector.

tol : float or None
The stopping criterion. If it is not None, the iterations will stop when (error > previous_error - tol). If it is None, the number of iterations will be set by the "epochs".

regularizer : string
The string value could be one of the following: l1, l2, None.
If it's set to None, the cost function without the regularization term will be used for computing the gradient and updating the weight vector. However, if it's set to l1 or l2, the appropriate regularized cost function needs to be used for computing the gradient and updating the weight vector.

Note: you may define two helper functions for computing the regularized cost for "l1" and "l2" regularizers.

lambda : float
It provides the regularization coefficient. It is used only when the "regularizer" is set to l1 or l2.

Returns:

return value necessary.

Note: the "fit" method should use a weight vector "theta_hat" that contains the parameters for the model (one parameter for each feature and one for bias). The "theta_hat" should be a 1D column vector.

Finally, it should update the model parameter "theta" to be used in "predict" method as follows.
self.theta = theta_hat

b. predict(self, X)

Arguments:

X : ndarray
A numpy array containing samples to be used for prediction. Its rows represent data samples and columns represent features.

Returns:

1D array of predictions for each row in X.
The 1D array should be designed as a column vector.

Note: the "predict" method uses the **self.theta** to make predictions.

c. __init__(self)

It's a standard python initialization function so we can instantiate the class. Just "pass" this.

Part B: Data Processing [5 pts]

6. Read in the **winequality-red.csv** file as a Pandas data frame.
7. Summarize each of the variables in the dataset in terms of mean, standard deviation, and quartiles. **Include this in your report**.
8. Shuffle the rows of your data. You can use def = df.sample(frac=1) as an idiomatic way to shuffle the data in Pandas without losing column names.
9. Generate pair plots using the seaborn package. This will be used to identify and report the redundant features, if there is any.

Part C: Model Evaluation [30 pts]

10. **Model selection via Hyperparameter tuning**: Use the **kFold** function (known as sFold function from previous assignment) to evaluate the performance of your model over each combination of lambda, learning_rate and regularizer from the following sets: **[15 pts]**

    a. lambda = [1.0, 0, 0.1, 0.01, 0.001, 0.0001]
    b. learning_rate = [0.1, 0.01, 0.001, 0.001]
    c. regularizer = [l1, l2]
    d. Store the returned dictionary for each and **present it in the report**.
    e. Determine the **best model** (model selection) based on the overall performance (lowest average error). For the ***error_function*** argument of the kFold function (known as sFold function from previous assignment), use the "mse" function. For the model selection **don't augment the features**. In other words, your model selection procedure should use the data matrix X as it is.

11. Evaluate your model on the **test data** and report the mean squared error. **[5 pts]**
12. Using the best model plot the learning curve. Use the rmse values obtained from the "learning_curve" function to plot this curve. **[5 pts]**
13. Determine the best model hyperparameter values for the training data matrix with polynomial **degree 3** and plot the learning curve. Use the rmse values obtained from the "learning_curve" function to plot this curve. **[5 pts]**

Part D: Written Report [25 pts]

12. Describe whether or not you used feature scaling and why or why not. **[3 pts]**
13. Describe whether or not you dropped any feature and why or why not. **[3 pts]**
14. In the lecture we have studied two types of Linear Regression algorithm: closed- form solution and iterative optimization. Which algorithm is more suitable for the current dataset? Justify your answer. **[4 pts]**
15. Would the batch gradient descent and the stochastic gradient descent algorithm learn similar values for the model weights? Justify your answer. Let's say that you used a large learning rate. Would that make any difference in terms of learning the weights by both algorithms? **[5 pts]**
16. Consider the learning curve of your model (degree 1). What conclusion can you draw from the learning curve about (a) whether your model is overfitting/underfitting and (b) its generalization error. Justify your answer. **[5 pts]**
17. Consider the learning curve of the 3rd degree polynomial data matrix. What conclusion can you draw from the learning curve about (a) whether your model is overfitting/underfitting and (b) its generalization error. Justify your answer. **[5 pts]**