AI534 — Implementation Assignment 1 —

General instructions.

1. Please use Python 3 (preferably version 3.6+). You may use packages: Numpy, Pandas, and matplotlib, along with any from the standard library (such as 'math', 'os', or 'random' - for example).

2. You can collaborate in group of up to three people. Please do not share code with other groups, or copy program files/structure from any outside sources like Github. Each group's work should be your own.

3. Submit your report on Canvas and your code on TEACH following this link: `https://teach.engr.oregonstate.edu/teach.php?type=assignment`.

4. Your code should follow the structure and function specification of the provided skeleton code.

5. Please test your code before submission on the EECS servers (i.e. babylon) to make sure that it runs correctly on the server and produce the correct results as reported in the your report.

6. Be sure to answer all the questions in your report. You will be graded based on your code as well as the report. The report should be clear and concise, with figures and tables clearly labeled with necessary legend and captions. The quality of the report and is worth 10 pts. The report should be a PDF document.

7. In your report, the **results should always be accompanied by discussions** of the results. Do the results follow your expectation? Any surprises? What kind of explanation can you provide?

IMPORTANT! If your code fails to run during our testing, you will be notified with the failure result and you will have 24 hours to correct the issue.

Report will not be graded unless the accompanying code runs successfully.

Linear regression
(total points: 90 pts + 10 report pts)

For the first part of the assignment, you will implement linear regression, which learns from a set of N training examples $\{\mathbf{x}_i, y_i\}_{i=1}^N$ an weight vector $\mathbf{w}$ that optimize the following Mean Squared Error (MSE) objective:

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{w}^T\mathbf{x}_i)^2 \tag{1}$$

To optimize this objective, you will implement the gradient descent algorithm. Because some features have very large values, for part of the assignment you are asked to normalize the features. This will have an impact on the convergence behavior of gradient descent.

Data. This data contains historic data on houses sold between May 2014 to May 2015. You need to build a linear regression that can be used to predict the house's price based on a set of features. You are provided with two data files: **train** and **dev** (validation), in csv format. You are provided with a description of the features as well. The last column stores the target y values for each example. You need to learn from the training data and tune with the provided validation data to chose the best model.

Part 0 (5 pts) : Data preprocessing. Excluding the ID, this data contains 19 features that are mixed in types. For this part, we will simply treat these features as numerical. Note that there may be better ways of encoding some of them and you can explore them in PART 4. To prepare your data, perform the following pre-processing steps:

(a) Remove the ID column

(b) Split the `date` feature into three separate features `year, month, day`. This gives us a total 21 usable features that are encoded in the numerical form.

(c) Add to the data a dummy feature with constant value of 1 for all examples. This will allow us to learn the intercept or bias term w_0 for the linear model $y = w_0 + w_1x_1 + ... + w_dx_d$.

(d) Feature `yr_renovated` has value 0 if the house has not been renovated. This creates an inconsistent meaning to the numerical values. We will replace it with a new feature `age_since_renovated` to replace `yr_renovated` as follows. If `yr_renovated` is zero,

$$age\_since\_renovated = year - yr\_built$$

Otherwise

$$age\_since\_renovated = year - yr\_renovated$$

(e) Normalize all features (excluding `waterfront`, which is binary and we will leave it as is) using z-score normalization based on the training data. Do not normalize `price` as it is the target `y`.
To normalize a feature x using z-score normalization, the formula is $z = \frac{x-\mu}{\sigma}$, where μ and σ are the mean and standard deviation of x respectively. The normalized feature z will have zero mean and unit standard deviation. Note that when applying the model learned from normalized data to test/validation data, you should make sure that you normalize the inputs exactly the same way. That is, you must save the training μ and σ for each feature to be used for normalization of the test data.

Part 1 (45 pts). Implement batch gradient descent and explore different learning rates. For this part, you will work with the pre-processed and normalized data, and consider at least the following values for the learning rate: $10^i, i = 0, 1, \cdots, 4$.

(a) (35 pts) Train your model for up to 4000 iterations using different learning rates. Stop early if it has converged (training MSE changes very little) or diverged (training MSE increases with more iterations) before reaching 4000 iterations. Plot the MSE as a function of iterations for each learning rate. You can generate a single figure for all curves (excluding the diverging ones), using different colors with legends to indicate which color for what learning rate.

> Question: Which learning rate or learning rates did you observe to be good for this particular dataset? What learning rates (if any) make gradient descent diverge?

(b) (5 pts) For each learning rate that did not diverge, compute and report the MSE of final model on the validation data.

> Question: Which learning rate leads to the best validation MSE? Between different convergent learning rates, how should we choose one if the validation MSE is nearly identical?

(c) (5 pts) Pick the best model that minimizes the validation MSE, and report the learned weights for each feature (excluding w_0).

> Question: learned feature weights are often used to understand the importance of the features. What features are the most important in deciding the house prices according to the learned weights?

Part 2 (32 pts). (Let's explore this a bit with some additional experiments.)

(a) **Training with *non-normalized data*(24 pts)**. Here we will use the pre-processed data but skip the normalization step and train with batch gradient descent for up to 4000 epochs. For this part, you will need to search for a learning rate that works well. Note that here the learning rate will need to be orders of magnitude smaller than what had worked for part 1. You can begin your search assuming the learning rate $= 10^{-i}$, and search for the smallest i that avoids divergence using linear (slower) or exponential (faster) search starting from $i = 1$. You can then further expand your search to consider other values that are not in the form of 10^{-i}.

In your report, please provide the following:

(a) a list all the learning rates that you experimented with for this part and whether the learning is converging or diverging.

(b) the best learning rate that you have found and its resulting MSE on the training and validation data respectively

(c) the learned feature weights using the best learning rate (excluding w_0)

> Questions:
> 1. Why do you think the learning rate needs to be much smaller for the non-normalized data?
> 2. Compare between using the normalized and the non-normalized versions of the data, which one is easier to train and why?
> 3. Compare the learned weights to those of 1(c), what key difference do you observe? How do you think this impacts the validity of using weights as the measure of feature importance?

(b) **Redundancy in features (8 pts).** Note that the two features `sqrt_living` and `sqrt_living15` are fairly correlated. When features are correlated, they can offer redundant information. For this part, please use the normalized data but drop the feature `sqrt_living15` and train the model with a learning rate of your choice and report the validation MSE and the learned feature weights.

Question: how does this new model compare to the one in part 1 (c)? What do you observe when comparing the weight for `sqrt_living` in both versions? Consider the situation in general when two features x_1 and x_2 are redundant, what do you expect to happen to the weights (w_1 and w_2) when learning with both features, in comparison with w_1 which is learned with just x_1? How does this phenomenon influence the interpretation of feature importance?