Homework 2: Linear Regression and Logistic Regression

Principles of Machine Learning (Wentworth Institute of Technology)

Fall 2024 Semester

Although they are some of the simplest machine learning algorithms when looking at their template, linear regression and logistic regression can both perform quite well with careful feature engineering. Despite their similarities, their implementations do not align code-wise (get it, equations with a linear component, which are lines, and 'align' sounds like 'line?') because we can compute one exactly while we must approximate the other. Please respond to each item below, which can be a question to answer and/or task to perform. *You must show your work in order to receive full credit for a correct response, and your work will determine partial credit for incorrect responses.* If you choose to work with others in the class, please remember that everyone must turn in their own version of the responses to the items in this assignment. You must mention with whom you collaborated as well (and for that matter, please remember to include your name on the submission).

When working on problems in the wild, be it for work or pleasure, the world will neither tell you the specific problem nor what to do to solve this problem. Customers and/or clients come to you because they do not know how to do it themselves, which means they can only give a lot of context attempting to describe what they want you to achieve. The world itself has lots of things you can observe and loves to throw life challenges at us. Some of the information from these sources will be useful, and the rest is either a red herring or irrelevant. These problems are phrased in a similar way, and your solutions will need you to identify both the problem(s) and which information is relevant to solve them.

1. ***(20 points)*** Implementing efficient matrix data structures can be complicated due to all the different features of data (sparsity, size, etc.) and computer architecture options (CPU vs. GPU, available RAM, etc.), but using an efficient matrix data structure makes algorithms that use them far less complicated! Choose a matrix-based programming language (Matlab, Octave, Julia, R, etc.) or package (Python's numpy and scipy, C++'s Boost, etc.) of choice, and then we will implement several variants of linear regression. However, we want to make the user's life as simple as possible such that they only need to think about a *single linear regression function call/thing.* One ideal way to think about and prepare this code is as follows, and the steps are individual chunks that should be able to run successfully before moving on. It is highly encouraged you TEST that your code works as intended after each step so that any problems in the next step are easier to diagnose. *Good programming practices you should use in this assignment are listed in the back as an appendix.*

   (a) There are other benefits to keeping things bundled together when possible and carefully planning out our software design: ***modularity***! A module is a unit of code that we can easily swap out for another unit of code, which means they need to connect to the rest the code in the same way. The Python-based machine learning library Scikit-Learn uses modularity a lot so that users can easily switch the machine learning algorithm in a few lines of code without needing to change anything else, for example. In two-to-three sentences, how can abstract superclasses in object-oriented programming facilitate modularity? For non-object-oriented languages that use data structures and functions on their own, what would modular components (usually contents in one file that users import into their code) require documentation-wise so that people know how each module connects[1]?

   (b) Whether your programming language of choice represents 'things' as classes or a data structures (let's be honest, classes are a special data structure in object-oriented programming), create a

[1] This information should also be documented if you write your code using object-oriented programming languages, but good programming practices for class inheritance should document this information by default.

'thing' that represents a machine learning algorithm's template with the following: a variable that stores whether the template has assigned parameters yet (and would thus become a model), a variable that stores the parameters learned, a variable that stores any hyperparameter settings (for later reference), a function that performs training, a function that performs prediction (testing or learned model execution), and functions that access each of the variables (get methods). If we do not provide functions that manipulate each of the variables (set methods), then how do you think these variables receive their assignments? For a machine learning algorithm template, why should we prefer this approach over functions that manipulate each variable?

(c) If we store *all* hyperparameters and *all* learned parameters each in a single variable, then we need to make sure they have some organizational structure to keep track of individual hyperparameters and parameters. Compare a hash map/dictionary to an array/matrix implementation for each of these variables—which is more appropriate for hyperparameters, which is more appropriate for parameters, and why?

(d) Now that we have a general class or data structure that should describe the properties and capabilities of any machine learning algorithm, we can implement a more specific (sub)class or data structure per algorithm. For linear regression, the training and prediction functions should follow these specifications:

Train Function

i. The inputs to the train function should always be the hyperparameters (as an array or hash map based on your response to 1.c.) and the training data as inputs X (a matrix) and ground truth outputs $\overrightarrow{y}$ (a vector, which is effectively a matrix with exactly one column).

ii. Let one of the hyperparameters the user sets be a choice of MAP Estimation vs. MLE vs. regularization to choose the training variant for linear regression. Check this hyperparameter to determine which specific ***helper function*** to call for the actual training process. Helper functions are additional functions the programmer writes to make their code easier to read, write, or debug; this 'helps' the programmer instead of being additional functionality for the user who gets the code as a package, library, API, etc.

iii. All the helper functions for linear regression are similar so that you should be able to avoid writing too much code—code reuse is essential to being efficient and reducing errors (if you write the same thing twice, you could make a mistake in only one of them or make different mistakes in each). In fact, we will only need one helper function (MAP Estimation) instead of three because we can manipulate the function inputs due to how related they are. Declare variables σ^2 and b^2 before the condition checks on the hyperparameter for the user's choice of linear regression variant, and put a single helper function call at the end of your condition checks on the hyperparameter that takes these variables as its inputs. Do not forget that the MLE helper function will also need the training datas additional function inputs.

iv. The MAP Estimation case seems the simplest because σ^2 and b^2 should be defined in the given hyperparameters. If you want to be super efficient, then you could avoid checking for the MAP Estimation variant and simply initialize the declared variables with those hyperparameter values. However, this efficiency trick can also be problematic if the user did not set these hyperparameters as expected. Thus, given how small the time savings is for a single conditional branch, it is better for our implementation's average situation that we check for the MAP Estimation variant and sanity check that the user provided these hyperparameter values within the case (and that they are non-negative).

v. The MLE case does not use any hyperparameters, which means we want the values of the σ^2 and b^2 variables to work out mathematically that the hyperparemeter terms in the MAP Estimation equation go away. Conveniently, $\sigma^2 = 0$ and $b^2 = 1$ will do the trick because $\left(\sigma^2/b^2\right) I_{dim(X^T X)} = 0_{dim(X^T X)}$. If you want to use the efficiency trick of skipping one conditional case through initializing the declared variables, then MLE is the best choice here because the assigned values are fixed in the code without risk of users providing incorrect input. However, this means your code assumes MLE is the

default variant of linear regression to run; you must note such an assumption in your documentation so that users are aware of what the code does when they either provide an invalid variant of linear regression or do not specify a variant at all.

vi. The regularization case uses the σ^2 and λ hyperparameters. Besides sanity checking that the user provided these hyperparameter values (and that they are non-negative), we need to assign the value of the b^2 variable to work out mathematically that the equality $\left(\sigma^2/b^2\right) = \lambda$ holds. Conveniently, $b^2 = \lambda/\sigma^2$ as long as σ^2 is non-zero, which is a slight modification to the non-negative check above.

vii. To test the train function without the implemented helper function, simply define the helper function to print something informative such as "Training: MLE (Inputs include...)" and return a dummy output that has the correct format such as $\overrightarrow{0}$ (vector format) or 0 (real number format).

viii. If you did not implement it for the generalized template, set the variable that indicates whether the parameters have been learned yet. If the helper function does not store the learned parameters, then be sure to store them as well (and make sure the helper function returns the learned parameters).

Predict Function

i. The input to the predict function should always be the inputs X.

ii. Compute and return $\overrightarrow{z} = X\,\overrightarrow{\theta}$ as your predicted output number(s). $\overrightarrow{\theta}$ are your learned parameters, which is why linear regression is such a simple machine learning model.

iii. An important sanity check, if you did not implement this for the generalized template, is that the parameters have been learned already. If not, be sure to inform the user and terminate the program gracefully (throw an error/exception or trigger an assertion violation).

Helper Function: MAP Estimation

i. The input to the MAP Estimation function should always be the hyperparameters and the training data as inputs X and ground truth outputs $\overrightarrow{y}$.

ii. Compute and store the matrices X^T (get this one first because you will use it again later) and $X^TX + \left(\sigma^2/b^2\right) I_{dim(X^TX)} = M$ (get this one second because it can use the X^T that you already stored).

iii. M should have an inverse. You could consider a sanity check in the code based on your work on the previous homework problem, but it is not mandatory if you are confident that the user will rarely violate the condition. What you must do is compute and store that matrix's inverse M^{-1}.

iv. Now you can compute the parameters $M^{-1}X^T\overrightarrow{y}$ and store them internally and/or return them to the calling function on the runtime stack.

2. ***20 points*** Continuing to build our software suite of machine learning algorithms, we will next implement logistic regression. Using the general class or data structure you implemented for machine learning algorithms, we can implement yet another specific (sub)class or data structure. For logistic regression, the training and prediction functions should follow these specifications. *Good programming practices you should use in this assignment are listed in the back as an appendix.*

Train Function

(a) The inputs to the train function should always be the hyperparameters (as an array or hash map based on your response to 1.c.) and the training data as inputs X (a matrix) and ground truth outputs $\overrightarrow{y}$ (a vector, which is effectively a matrix with exactly one column). However, in order to learn an additional parameter for the y-intercept, we must augment X with one additional feature. Append a column at the end of the matrix where every entry is set to the value 1[2].

[2]You can use the augmentation method for linear regression as well if you want a y-intercept. As it is not always necessary, you can consider offering this feature to the user as yet another hyperparameter.

(b) Let the hyperparameters the user sets affect the gradient descent process. These include step size α, convergence threshold τ, and maximum number of iterations m. Please use appropriately descriptive names for your hyperparameter variables, including key names if using a hash map data structure, when writing the code.

(c) If you have support for function pointers in your programming language (C, C++, Python, etc.), then you can write a generic gradient descent helper function that receives the gradient as an input parameter. However, we will assume that is not the case because there are no language requirements for this homework assignment. Simply call a more specific helper function that runs gradient descent just for the log likelihood of the logistic regression function, which we create in a later step. Pass all the inputs into this helper function call because it will use them.

(d) To test the train function without the implemented helper function, simply define the helper function to print something informative such as "Training: Gradient Descent (Inputs include...)" and return a dummy output that has the correct format such as $\overrightarrow{0}$ (vector format) or 0 (real number format).

(e) If you did not implement it for the generalized template, set the variable that indicates whether the parameters have been learned yet. If the helper function does not store the learned parameters, then be sure to store them as well (and make sure the helper function returns the learned parameters).

Predict Function

(a) The input to the predict function should always be the inputs X. However, in order to have an additional input to go with the parameter for the y-intercept, we must augment X with one additional feature. Append a column at the end of the matrix where every entry is set to the value 1.

(b) Before computing the class label predictions, we need the predicted probability that the inputs will each be assigned class label 0, which will be a vector $\overrightarrow{p}$. Pass the input X into the predict probability helper function, which we will implement in a later step. We will use this helper function again, which is why we do not include those computations directly inside the predict function.

(c) To test the predict function without the implemented helper function, simply define the helper function to print something informative such as "Predicting: Probability of class label 0 (Inputs include...)" and return a dummy output that has the correct format such as $\overrightarrow{0}$ (vector format) or 0 (real number format).

(d) Compute and return the actual class label predictions in a vector by comparing each entry in $\overrightarrow{p}$ against 0.5, which is the midpoint between 0 and 1 (the minimum and maximum possible probability values). Keep the meaning of these numbers in mind to avoid "magic numbers" lying around your code. Any $p_i < 0.5$ gets classification label 0, and any $p_i \geq 0.5$ gets classification label 1. Just like the computation of p_i, some programming languages and matrix libraries will support using inequality operators between a matrix and a single integer without needing a loop, again serving as a convenience to the programmer. Otherwise, consider performing this comparison in the same loop that computes each p_i because the comparison only requires the current p_i to be computed first. Separate loops are not computationally efficient unless they are absolutely necessary because computer chips have to clear the execution pipeline between iterations (reseting the next lines of code that will execute to either be the code in the loop or the code after the loop), and this takes more time than running a single arithmetic or load/store operation.

(e) An important sanity check, if you did not implement this for the generalized template, is that the parameters have been learned already. If not, be sure to inform the user and terminate the program gracefully (throw an error/exception or trigger an assertion violation).

Helper Function: Predict Probability Function

(a) The input to the predict probability helper function should always be the inputs X.

(b) Compute and store $\overrightarrow{z} = X\overrightarrow{\theta}$ as your predictions of the ***log-odds*** (logarithm of the likelihood ratio) between binary class labels 0 and 1, which we assume is a linear equation for logistic regression's template. $\overrightarrow{\theta}$ are your learned parameters, just like linear regression.

(c) If your programming language and/or matrix package supports mathematical operations over a matrix, then this step can be done with some mathematics operations. However, many programming languages and matrix libraries lack this support because it is mathematically incorrect notation and more of a convenience feature for programmers. For those in the latter group, use a loop over the entries of $\overrightarrow{z}$ to compute the actual likelihood of the class label being 1 for each input:

$$p_i = \mathrm{P}\left(\mathrm{class}_i = 1 \,\middle|\, \overrightarrow{X_{i,.}}, \overrightarrow{\theta}\right) = 1/\left(1 + e^{z_i}\right).$$

The non-loop version for languages that do support these operations for convenience will simply exclude the i subscript from this equation—double-check that it still yields a vector $\overrightarrow{p}$.

(d) That is it; return the probabilities.

Helper Function: Gradient Descent for Logistic Regression

(a) The inputs to the gradient descent helper function should always be the same as the inputs to the train function. Despite being the same, it is important to document what each parameter is just in case future changes lead to a divergence between the train function's inputs and gradient descent function's inputs.

(b) Without the guarantee of function pointers, we need to run gradient descent with the specific gradient written into the helper function. However, we can still attempt to make duplicating the helper function for future uses of gradient descent as copy-paste-error proof as possible because a function pointer is simply a variable that stores the memory address for the code that performs the function[3], and nothing is stopping us from just hard-coding a function call that computes the gradient. This is another way to think of modularity because we can switch the function name that computes the gradient if you copy-paste the gradient descent function elsewhere to approximate the optima of a different function. This means we will soon have a helper function for this helper function, but nothing specific to write in code yet.

(c) In order for gradient descent to begin its iterative process, we first need a ***starting point*** from which we begin to move towards the function's minimum. There are no parameter constraints to consider that would bound what values are possible, which means $\overrightarrow{\theta}$ can be anything. Simply initialize each of the parameters in this vector to some random floating point number. Do not forget that *logistic regression does use the intercept coefficient in addition to each feature's scalar*—to avoid putting something specific to logistic regression inside the generic gradient descent code, consider setting the number of parameters beforehand, such as in the train function before it passes the information into the gradient descent function call.

(d) To avoid a risk of an infinite loop, execute the iterative step of gradient descent m times using a loop. If we need fewer iterations of the loop because we found the optimal parameters for the training data and hyperparameters, then we can use a break statement to stop things early. The iterative step first backs up the current parameter assignment values into some duplicate variable along the lines of $\overrightarrow{\theta}_{prev}$ so that we can compute the change in $\overrightarrow{\theta}$ later.

(e) With the back-up ready, it is now safe to reassign $\overrightarrow{\theta}$ using the additional helper function to compute the gradient $\nabla\mathcal{L}$:

$$\overrightarrow{\theta} = \overrightarrow{\theta}_{prev} - \left(\alpha \cdot \nabla\mathcal{L}\left(\overrightarrow{\theta}_{prev}\right)\right).$$

This update can become more intricate if we implement variations of the gradient descent algorithm that improve the rate of convergence or reduce oscillation about the solution, but we will not program those variations at this time.

[3]Although one can think of this like a goto statement in Basic-based programming languages or a branch command in assembly, the specifications require function pointers to keep track of the runtime stack just like a function call. This means the compiler or interpreter does all the difficult bookkeeping for you to backtrack after the function completes its execution.

(f) To determine whether we can break from the loop early, compare $\overrightarrow{\theta}$ and $\overrightarrow{\theta}_{prev}$ using a distance function. When the distance is below the specified threshold τ, we assume that the result ***converged*** because the modification $-\left(\alpha \cdot \nabla\mathcal{L}\left(\overrightarrow{\theta}_{prev}\right)\right)$ is small enough to imply that the gradient is very close to 0 as desired. The most common choice for measuring distance in this convergence check is the L_2 norm (Euclidean distance) because the parameters are each a dimension in ***parameter space***; $\overrightarrow{\theta}$ and $\overrightarrow{\theta}_{prev}$ are effectively coordinates. Because these vectors can compute the L_2 norm naturally, avoid using a loop since it is not computationally efficient: just check whether $\left|\left|\overrightarrow{\theta} - \overrightarrow{\theta}_{prev}\right|\right|_2^2 \leq \tau$ where $||\overrightarrow{v}||_2^2$ is the squared magnitude of a vector $\overrightarrow{v}$. Alternatively, the squared magnitude is just the dot product of a vector with itself, $||\overrightarrow{v}||_2^2 = \overrightarrow{v}^T \cdot \overrightarrow{v}$—if you use the dot product approach, be sure to store the difference between the parameter vectors as $\overrightarrow{v}$ to avoid computing it twice!

(g) Whether the code's control flow is past the loop because gradient descent converged early or it ran for all m iterations, we are done and take the most recent $\overrightarrow{\theta}$ as our parameters that set the gradient to 0/optimize the function of interest. Return this vector as the result. If you include print statements for debugging, then it would not hurt to have some here that mention whether the loop terminated early or needed all m iterations.

Helper Function: Gradient Computation for Logistic Regression

(a) The inputs to the gradient computation for logistic regression helper function should be the training data, X and $\overrightarrow{y}$, and the current parameters $\overrightarrow{\theta}$. We do not need hyperparameters for this computation.

(b) The gradient for $\mathcal{L}$, the log likelihood of error under logistic regression, is a vector $\nabla\mathcal{L}$, but each entry in this vector is a function instead of a number: $\nabla\mathcal{L}_i\left(\overrightarrow{\theta}\right) = \partial\mathcal{L}\left(\overrightarrow{\theta}\right)/\partial\theta_i$. Specifically, each function $\nabla\mathcal{L}_i\left(\overrightarrow{\theta}\right)$ is the partial derivative of the logistic regression-based negative log likelihood function with respect to the i^{th} parameter θ_i:

$$\nabla\mathcal{L}_i\left(\overrightarrow{\theta}\right) = \frac{\partial\mathcal{L}\left(\overrightarrow{\theta}\right)}{\partial\theta_i} = \sum_{r=1}^{R} x_{r,i}\left(\mathrm{P}\left(\text{label} = 0 \,\middle|\, \overrightarrow{x}_r, \overrightarrow{\theta}\right) - y_r\right).$$

Because we have inputs $\overrightarrow{\theta}$, these functions will evaluate to numbers. Unfortunately, there is no simple matrix to compute for this gradient. We will need to use nested loops over i and r respectively to compute each gradient. However, we can at least reduce the number of computations in these loops to avoid wasting time with duplicate computations. Before starting those loops, create a vector that will store the gradient and set all its entries to 0. 0 is important here because it is the identity element of addition for the real numbers; anything plus 0 is itself.

(c) The outer loop will go over each row (sample) in our training data X and $\overrightarrow{y}$. Why go over the data first if we ultimately need the gradient with respect to each parameter? Whether you asked or not, great question! Rick Freedman (the instructor who wrote this assignment) has an answer: each ***summand*** (each item added towards a sum) per partial derivative is a product of two ***multiplicands***: $x_{r,i}$ and $\left(\mathrm{P}\left(\text{label} = 0 \,\middle|\, \overrightarrow{x}_r, \overrightarrow{\theta}\right) - y_r\right)$. The left multiplicand depends on both loop variables r and i, but the right multiplicand only depends on the r loop variable since we use all the entries of $\overrightarrow{\theta}$ rather than a single one[4]. If we iterated over r inside a loop iterating over i, then we would have to recompute the right multiplicand when restarting the inner r loop for the next value of i ... as the right multiplicand uses a lot of arithmetic operations, that is an expensive computation to repeat frequently. A single load will be far less computational effort in comparison, and that means we need to store each row's computation *before* iterating over each i. This order of operations is why we choose r for the outer loop.

[4]The use of all parameters in the probability computation is exactly why we cannot get an exact partial derivative computation like with linear regression.

(d) Compute and store $d = \left(\mathrm{P}\left(\text{label} = 0 \,\middle|\, \overrightarrow{x}_r, \overrightarrow{\theta}\right) - y_r\right)$ within the outer loop before starting the inner loop. To avoid writing the same code multiple times with a risk of getting it wrong the second time, keep in mind that $\mathrm{P}\left(\text{label} = 0 \,\middle|\, \overrightarrow{x}_r, \overrightarrow{\theta}\right)$ is a prediction of the probability (NOT the label) for input $\overrightarrow{x}_r$ (the r^{th} row/sample of X) using the input parameters $\overrightarrow{\theta}$. Rather than write a lot of fresh code, this means you can use your logistic regression class's predict probability helper function with the correct inputs! Wait, the predict probability helper function only allows $\overrightarrow{x}_r$ as inputs? This is where you use a "hack" and set the class instance's parameters to $\overrightarrow{\theta}$ just before calling the predict probability helper function. If you want your program to be ***secure***, then you will need to clear out the parameter setting after the function call. If you want your program to be optimal, then you can avoid clearing out the parameter setting and hope you set it to the correct parameter setting before something else accesses that data[5]. y_r simply comes from the ground truth labels, and subtracting it from the predicted probability should be trivial after the other code you have written so far.

(e) The inner loop will go over each index in $\overrightarrow{\theta}$, which we have been calling i in the math; feel free to use a more descriptive variable name in your code.

(f) Within the inner loop, multiply the stored computation of d with $x_{r,i}$ from the training data and *accumulate* it to the i^{th} gradient computation. Accumulation is something like the `+=` operator, which some languages lack. If your language of choice does not have accumulation, then you can add the product to the current value: $\nabla\mathcal{L}_i\left(\overrightarrow{\theta}\right) = \nabla\mathcal{L}_i\left(\overrightarrow{\theta}\right) + (d \cdot x_{r,i})$. A computer will load the value currently stored there before replacing it with the updated value; do not worry about this doing something weird.

(g) Once both loops completed their computation, the gradient vector should be ready to return.

3. ***50 points*** Now that we have our own linear regression and logistic regression machine learning systems, let us try them out with some data and think about what our models learn. Close your eyes and imagine the following situations, but then open your eyes so you can read the rest the assignment, write the code, and prepare your responses—please keep imagining, though.

   (a) You are taking on the role of an analyst for housing prices under the employment of a national real estate company[6]. They want to make sure they evaluate houses accurately to reflect the proper value of potential clients' homes, which attracts clients who will get what their home is worth (and the agent who strikes the deal usually gets a commission proportional to the price of the sale).

      i. We will use the California Housing Dataset[7], whose features are explained at https://inria.github.io/scikit-learn-mooc/python_scripts/datasets_california_housing.html. https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html has a copy of the dataset that you can download for use in the programming language of your choice via file reading. You might need to format the data from the read-in file in order to store it into a matrix data structure.

      ii. Let's assume that there is no feature engineering necessary. Then we can begin training on the data right away after selecting how much training data we wish to use from the dataset. To explore whether we can find the trade-off between general model improvement and overfitting, try learning linear regression models (all three variants you implemented in Problem 1 above) over multiple training dataset sizes (minimum 5 models, and making this a parameter of a function would be most practical). Set the remainder of the California Housing Dataset dataset not used for training each model aside for testing, partitioning the dataset into two different collections of rows/samples. Remember that our classes store each learned model in

[5] Your own code will have no issues here because you do not use the parameters until they get set from the calling train function, but what if someone else uses your class like a library and just calls the gradient code on its own? Then they might have junk data in the parameters that will allow the "fake model" to run logistic regression.

[6] Check out https://en.wikipedia.org/wiki/Real_estate_agent if you want some quick details about what your employer generally does.

[7] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_california_housing.html

the constructed regression instance; so, store each learned model to compare in later parts! This also means you should store all the training and testing sets in a way that they are associated with their respective linear regression model—you can save memory by storing the row indeces per partition per model rather than copying the dataset and splitting it per model.

iii. In order to assess each learned linear regression model with its respective testing set, we can consider the ***root mean squared error*** between the ground truth values $\overrightarrow{y}$ and model prediction $\overrightarrow{\hat{y}}$:

$$RMSE\left(\overrightarrow{y}, \overrightarrow{\hat{y}}\right) = \sqrt{dim\left(\overrightarrow{y}\right)^{-1} \cdot \sum_{n=1}^{dim\left(\overrightarrow{y}\right)} \left(\overrightarrow{y} - \overrightarrow{\hat{y}}\right)^2} = \sqrt{dim\left(\overrightarrow{y}\right)^{-1}} \cdot \left|\overrightarrow{y} - \overrightarrow{\hat{y}}\right|.$$

This effectively measures how extreme the differences are collectively over all the test data points. Compute the RMSE per learned linear regression model and generate a plot comparing the RMSE (y-axis) to the percentage of the dataset used for training (x-axis). Based on your plot, when do you think there was too much training data to cause overfitting? Why? What concerns might you have about your conclusions given that you only learned one model per hyperparameter and training dataset size?

iv. Learning the 'best' model possible is a great start, but that does not provide us with all the insights that the real estate company needs to know. It might be great to simply predict the price of the house, but the potential clients will want to know *why* their home is valuated at its worth. Take a look at the learned parameters for your best, but not overfitted, model. Recall that these form the line whose equation is

$$\hat{y} = \sum_{n=1}^{dim(\vec{\theta})} (\theta_n \cdot x_n)$$

where one of those x_n is fixed at 1 if your model learns a y-intercept. Reading this equation, what does a positive, negative, and zero value for some θ_n tell us about the n^{th} feature's impact on the real estate value in California? How can you determine which of these features have the greatest impact on the real estate value?

(b) Because the United States runs on capitalism, the real estate company for which you work wants to go one step further than having machine learning determine the value of their potential clients' homes. They also want to determine the inflection point where the price of their client's home will encourage versus discourage sales—that is, how much can they ask someone to pay before the buyer decides it is too expensive? Although this sounds like a strategy for the company (and their client) to make more money, purchasing real estate works more like a negotiation in the United States so that the buyer can suggest a lower price that they are willing to pay. Thus, if the real estate company suggests a higher price than the home is worth, then there is a greater chance that the buyer's suggested negotiation value (which will obviously be equal or lesser because the buyer wants to save money) will be closer to the home's actual value.

i. In reality, we would ideally want to know the recent transaction history of properties in each region to better determine what people are willing to pay. However, the California Housing Dataset does not have this information; some homes might not have been sold for a while to have accurate prices anyways. So, we will perform feature engineering to create new features that *estimate* the most someone will pay for a home, and your linear regression models from the previous task will be helpful here! Because each model predicts the price of each real estate property, we can take the predictions' comparison to the ground truth (as you used in the RMSE computation) to approximate what a buyer is willing to pay as though it is the buyer's evaluation of the home's worth. Specifically, for your linear regression model of choice that is most accurate without overfitting (which one do you choose, and why?), predict each property's value as one new feature. Generate this prediction feature for all rows/samples in

the dataset regardless of whether it was in the training or testing partition. Then, compare that predicted value to the actual median house value to create a "willing to purchase" feature. If the prediction $\hat{y}$ is greater than the ground truth y for some datapoint, then set its "willing to purchase" value to 1 for 'yes' (disclaimer: this is an arbitrary assumption and not based on any actual information). Otherwise, set its "willing to purchase" value to 0 for 'no.' How many of the datapoints received a 1 versus a 0? Is this what you expected the proportion to be? Why?

ii. With this augmented dataset, we now have enough information to classify whether someone would or would not be willing to purchase a particular real estate property based on its usual features and predicted price. Why is it not a good idea to include the property's ground truth value as an input feature as well?

iii. Because this is a classification task where the class transition from "not willing to purchase" to "willing to purchase" takes place along a linear function, we can learn a logistic regression model. Use the logistic regression class you implemented to get various training and testing datasets of varying sizes (minimum 5) and keep track of the models with their respective dataset partitions. Generate a ***confusion matrix*** for each trained model that represents the following outcomes from testing as both a count and percentage of the testing data:

	Prediction is 1	Prediction is 0
Ground Truth is 1	True Positives	False Negatives
Ground Truth is 0	False Positives	True Negatives

iv. Generate a plot comparing each metric in the confusion matrix as a percentage of the testing data (y-axis) to the percentage of the dataset used for training (x-axis). You may compute four separate plots for each metric if that is easier to read. Based on your plot(s), when do you think there was too much training data to cause overfitting? Why? What concerns might you have about your conclusions given that you only learned one model per training dataset size? What concerns might you have about your conclusions given that you do not have any hyperparameters that can alter what the model learns (adjusting how gradient descent performs should not change the learned parameters too drastically, if at all)?

v. Take a look at the learned parameters for your best, but not overfitted, model. Recall that these form the logistic curve whose equation is

$$\mathrm{P}\left(output = 1\right) = \left(1 + exp\left(-\sum_{n=1}^{dim(\vec{\theta})}\left(\vec{\theta}_n \cdot \vec{x}_n\right)\right)\right)^{-1}$$

where one of those x_n is fixed at 1 for the y-intercept parameter θ_n. Reading this equation, what does the y-intercept parameter tell us about the inflection point for when people will change their mind about purchasing a particular home in California? Similarly, what does a positive, negative, and zero value for the coefficient parameters tell us about the n^{th} feature's impact on this inflection point? Consider 0.5 to be the threshold probability because this is a binary decision.

4. ***10 points*** Our naïve friend Flubsy overheard that our gradient descent function cannot easily generalize without function pointers or replacing the function call directly for a different gradient computation. Flubsy understands this issue, but does not understand why we are creating a function that computes the gradient in the first place. "You already know the function f you want to optimize if you are able to compute its derivative, right? Then why bother with programming the gradient's computation in the first place once you know it is such a mess? Think about it: a derivative is just the rate of change at a single point, which we compute via limits that get really close to that point from both sides. So, all we need is that original function and its inputs. To compute the partial derivative in some dimension for θ_n, we take that parameter with some offset $\theta_n - \epsilon$ and $\theta_n + \epsilon$[8] and plug it in with

[8] ϵ is a mathematical way of saying "a super tiny amount that is almost meaningless," which is actually meaningful for getting close to a number without reaching that number.

the other parameters and inputs fixed to their current values. Then, we have the ratio

$$\frac{f\left(\theta_n + \epsilon, \text{the rest}\right) - f\left(\theta_n - \epsilon, \text{the rest}\right)}{2\epsilon}$$

as that entry of the gradient; rinse and repeat for all the dimensions. Nothing can possibly go wrong!" Flubsy is not really wrong, but things can certainly go wrong thanks to computers being computers. Be nice to Flubsy, but please explain the concerns to the them through the following examples.

(a) The most alarming issue that we can more easily catch by hand is when a function has no partial derivative at a point where differentiability breaks. Show Flubsy how their approach would compute a gradient at the following functions and points that actually do not have a partial derivative when we examine the function in the real world:

- The absolute value function at the sharp point: $f\left(\overrightarrow{x}\right) = |\theta_1 x_1| + |\theta_2 x_2|$ when $\theta_1 = \theta_2 = 0$ and $\overrightarrow{x}$ is any vector with two real-valued entries.
- The jump in a piecewise function: $f\left(\overrightarrow{x}\right) = \theta_2 x_2$ if $x_1 \geq \theta_1, \theta_3 x_3$ otherwise when $\theta_1 = x_1$, $\theta_2 > \theta_3$, and $x_2 > x_3$.
- The vertical asymptote in a rational function that can divide by 0: $f\left(\overrightarrow{x}\right) = \theta_3 / \left(\theta_1 x_1 - \theta_2 x_2\right)$ when $\theta_1 = x_2$, $\theta_2 = x_1$, and θ_3 is any real number.

(b) Another issue that can become a concern if ϵ becomes small enough is ***underflow***, which we know is problematic from multiplying probabilities in linear regression's MLE and MAP Estimation variants. Describe how underflow would disrupt our ability to compute partial derivatives if we use Flubsy's proposed approach.

Appendix: Good Programming Practices

When writing code, you must follow these good programming practices:

- When turning in code, keep in mind that Rick Freedman (the instructor who wrote this assignment) has a golden rule: *Good code with bad comments is bad code, PERIOD.* Make sure comments are clear and describe what will happen per significant moment in the code. If the code does something trivial over several lines, explain the idea just before the first of those lines. If the code does something complicated on a single line, then explain what that line does. Use your judgement, but also remember you might need to reuse and understand this code later—your future self will thank you if you document it better now while the content is fresh in your mind!

- Documenting functions you create (purpose, inputs, outputs, and possible error throws) is also helpful. There is a reason most APIs that you read include these details as part of the manual.

- Avoid "magic numbers" when encoding choices like the linear regression approaches. It is tempting to use an integer such as 0 meaning MLE, etc., and the idea works well if done carefully. If you simply write 0 everywhere in the code, then how will you know which 0's mean the MLE choice, which 0's mean the number 0, and which 0's mean some other choice between different options? Instead, set a variable (ideally constant, if your language supports them, to avoid accidentally changing the assignment) to the number, and then use that variable everywhere in the code. It is easier to understand (compare `approach = 0` to `approach = MLE_LINREG`) and easier to update later if something changes in your code. New option to add in that slot, which means you have to change the previous option's assignment? Just change the variable to another number and do not worry because the variable effectively replaced it everywhere else in the code. Found a bug in the code that requires you to change the option you chose somewhere in the code? Now you know which 0's are for the option and which 0's are not so you do not accidentally change the wrong things (creating more errors that were not broken before).

- Be careful naming the output file to which you write because the program could overwrite another file that your program previously wrote by mistake! It is usually good practice to add a date+timestamp in the filename so that the output filenames are unique (unless you call the function really fast multiple times in a row).

- Store intermediate computations that you are able to reuse rather than compute everything from scratch in each line of code. Although you can compute things multiple times (such as X^T), it costs *real-world time* in an era where computer memory is much cheaper on your average computer (satellites and embedded systems are a different story, as is time spent retrieving cached data... these are all beyond the scope of this assignment). Save the time by storing the intermediate computation steps, and then call the variable storing it in every case that you use it afterwards. Storing the intermediate computation also lessens the chance of making a typo when recomputing it next time. This is similar to the benefits of avoiding "magic numbers."

- Break out sequences of computations that you will reuse frequently into functions rather than rewrite those lines of code from scratch each time you need it. Similar to storing intermediate computations, this lessens the chance of a typo when rewriting all that code each time. Furthermore, the function makes your code easier to read if it is descriptive—consider a function name replacing a lot of redundant code like an abridged comment placed above its block of code.

- Make debugging accessible even after your program is finished. You never know when you will need to look at the code or modify it again, and you do not want to have to redo your entire debugging setup. In particular, if you have any print statements in your code to explore the control flow ("entering function F," "exiting loop L in function F," etc.) or inspect variables ("the value of variable V is currently:," "compare variable V's value to expected E:," etc.), then those should stay in your code. However, to avoid those print statements cluttering the screen, put them all inside conditional blocks (if statements) where the condition to check is for a global constant variable such as DEBUG. Make sure to define this global constant variable DEBUG at the start of your code, and then set it appropriately before your compile and run your code. If DEBUG is boolean, then use TRUE when you want the debug information to print and FALSE when you do not want the debug information on the screen. If DEBUG is an integer, then define a hierarchy of verbosity/priorities where 0 means 'print nothing,' 1 means 'print the basic information,' 2 means 'print some additional details,' etc. (make sure to define each verbosity/priority level with its own constants to avoid "magic numbers"). You can also create more elaborate debugging variables in more complex code, such as separate DEBUG_X constant variables for different parts of the program X. For an assignment like this, one boolean variable should be sufficient.

Rick has lots of other rules and pet peeves about writing good code, but you can talk about those at office hours sometime if you are interested. They do not likely apply in this assignment.