CPSC 340 Assignment 3 (Due 2021-05-28 at 9:25am)

Important: Submission Format

Please make sure to follow the submission instructions posted on Piazza. **We are entitled to deduct 50% of your marks if the submission format is incorrect.**

1 Matrix Notation and Minimizing Quadratics

1.1 Converting to Matrix/Vector/Norm Notation

Using our standard supervised learning notation (X, y, w) express the following functions in terms of vectors, matrices, and norms (there should be no summations or maximums).

1. $\max_{i\in\{1,2,\dots,n\}} |w^Tx_i - y_i|$.

   Answer: $\|Xw - y\|_\infty$

2. $\sum_{i=1}^n v_i(w^Tx_i - y_i)^2 + \frac{\lambda}{2}\sum_{j=1}^d w_j^2$.

   Answer: $(Xw-y)^TV(Xw-y) + \frac{\lambda}{2}\|w\|^2$ (the first term can also be written $||V^{1/2}(Xw-y)||^2$).

3. $\left(\sum_{i=1}^n |w^Tx_i - y_i|\right)^2 + \frac{1}{2}\sum_{j=1}^d \lambda_j|w_j|$.

   Answer: $\|Xw-y\|_1^2 + \frac{1}{2}\|\Lambda w\|_1$.

Note that in part 2 we give a *weight* v_i to each training example and the value λ is a non-negative scalar, whereas in part 3 we are regularizing the parameters with different weights λ_j. You can use V to denote a diagonal matrix that has the values v_i along the diagonal, and Λ as a diagonal matrix that has the λ_j values along the diagonal. You can assume that all the v_i and λ_i values are non-negative.

1.2 Minimizing Quadratic Functions as Linear Systems

Write finding a minimizer w of the functions below as a system of linear equations (using vector/matrix notation and simplifying as much as possible). Note that all the functions below are convex so finding a w with $\nabla f(w) = 0$ is sufficient to minimize the functions (but show your work in getting to this point).

1. $f(w) = \frac{1}{2}\|w - v\|^2$ (projection of v onto real space).

   Answer: $f(w) = \frac{1}{2}\|w\|^2 - w^Tv + \frac{1}{2}\|v\|^2$ so $\nabla f(w) = w - v$ and setting it equal to 0 we have $w = v$

2. $f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{1}{2}w^T\Lambda w$ (least squares with weighted regularization).

   Answer: $\nabla f(w) = X^TXw - X^Ty + \Lambda w$ so we need to solve the linear system $(X^TX + \Lambda)w = X^Ty$

3. $f(w) = \frac{1}{2}\sum_{i=1}^n v_i(w^Tx_i - y_i)^2 + \frac{\lambda}{2}\|w - w^0\|^2$ (weighted least squares shrunk towards non-zero w^0).

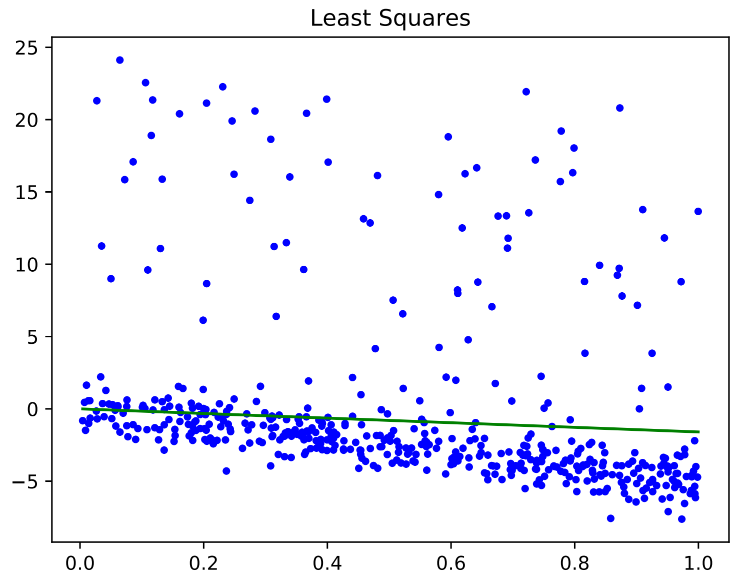
   Answer: $f(w) = \frac{1}{2}(Xw - y)^TV(Xw - y) + \frac{\lambda}{2}(w - w^0)^T(w - w^0)$ and $\nabla f(w) = X^TVXw - X^TVy + \lambda w - \lambda w^0$, so we need to solve the linear system $(X^TVX + \lambda I)w = X^TVy + \lambda w^0$

Above we assume that v and w^0 are d by 1 vectors, and Λ is a d by d diagonal matrix (with positive entries along the diagonal). You can use V as a diagonal matrix containing the v_i values along the diagonal.

Hint: Once you convert to vector/matrix notation, you can use the results from class to quickly compute these quantities term-wise. As a sanity check, make that the dimensions match for all quantities/operations: in order to make the dimensions match you need to introduce an identity matrix. For example, $X^TXw + \lambda w$ can be re-written as $(X^TX + \lambda I)w$.

2 Robust Regression and Gradient Descent

If you run `python main.py -q 2`, it will load a one-dimensional regression dataset that has a non-trivial number of 'outlier' data points. These points do not fit the general trend of the rest of the data, and pull the least squares model away from the main downward trend that most data points exhibit:



Note: we are fitting the regression without an intercept here, just for simplicity of the homework question. In reality one would rarely do this. But here it's OK because the "true" line passes through the origin (by design). In Q3.1 we'll address this explicitly.

2.1 Weighted Least Squares in One Dimension

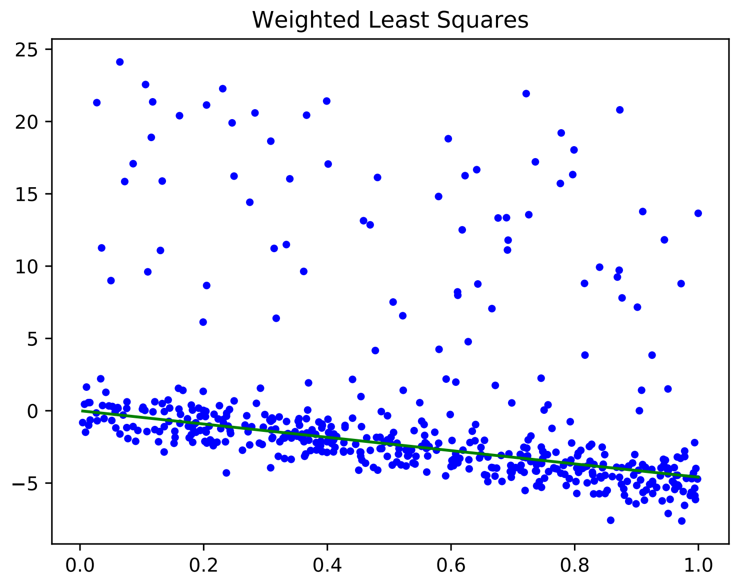
One of the most common variations on least squares is *weighted* least squares. In this formulation, we have a weight v_i for every training example. To fit the model, we minimize the weighted squared error,

$$f(w) = \frac{1}{2}\sum_{i=1}^{n} v_i(w^T x_i - y_i)^2.$$

In this formulation, the model focuses on making the error small for examples i where v_i is high. Similarly, if v_i is low then the model allows a larger error. Note: these weights v_i (one per training example) are completely different from the model parameters w_j (one per feature), which, confusingly, we sometimes also call "weights".

Complete the model class, `WeightedLeastSquares`, that implements this model (note that Q1.2.3 asks you to show how a few similar formulation can be solved as a linear system). Apply this model to the data containing outliers, setting $v = 1$ for the first 400 data points and $v = 0.1$ for the last 100 data points (which are the outliers). Hand in your code and the updated plot.

Answer: See code. The figure, when weighting the data points, now looks like this:



It now models the overall trend in the non-outlier points.

2.2 Smooth Approximation to the L1-Norm

Unfortunately, we typically do not know the identities of the outliers. In situations where we suspect that there are outliers, but we do not know which examples are outliers, it makes sense to use a loss function that is more robust to outliers. In class, we discussed using the sum of absolute values objective,

$$f(w) = \sum_{i=1}^{n} |w^T x_i - y_i|.$$

This is less sensitive to outliers than least squares, but it is non-differentiable and harder to optimize. Nevertheless, there are various smooth approximations to the absolute value function that are easy to optimize. One possible approximation is to use the log-sum-exp approximation of the max function[1]:

$$|r| = \max\{r, -r\} \approx \log(\exp(r) + \exp(-r)).$$

Using this approximation, we obtain an objective of the form

$$f(w) = \sum_{i=1}^{n} \log\left(\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)\right).$$

which is smooth but less sensitive to outliers than the squared error. Derive the gradient ∇f of this function with respect to w. You should show your work but you do not have to express the final result in matrix notation.

Answer: Let's first take the derivative of the approximation,

$$\frac{d}{dr}[\log(\exp(r) + \exp(-r))] = \frac{\exp(r) - \exp(-r)}{\exp(r) + \exp(-r)}$$

which if you think about it is a reasonable approximation to the sign of r ($r/|r|$) as long as r isn't too close to zero. An individual partial derivative will have the form

$$\frac{\partial f}{\partial w_j} = \sum_{i=1}^{n} x_{ij} \frac{\exp(w^T x_i - y_i) - \exp(y_i - w^T x_i)}{\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)}.$$

The gradient will be the vector formed from these elements.
(If you wanted to express it in matrix notation, we could define a vector r with elements

$$r_i = \frac{\exp(w^T x_i - y_i) - \exp(y_i - w^T x_i)}{\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)},$$

and with this notation we have

$$\nabla f(w) = X^T r,$$

which is similar to the gradient for least squares but where we've replaced the signed residuals $(Xw - y)$ with an approximation of the signs of the residuals.)

2.3 Gradient Descent: Understanding the Code

Recall gradient descent, a derivative-based optimization algorithm that uses gradients to navigate the parameter space until a locally optimal parameter value is found. In `optimizers.py`, you will see my implementation of gradient descent, taking the form of a class named `OptimizerGradientDescent`. This class has a similar design pattern as PyTorch, a popular differentiable programming and optimization library. One step of gradient descent is defined as:

$$w^{t+1} = w^t - \alpha^t \nabla_w f(w^t)$$

Look at the methods named `step()` and `break_yes()`, and answer each of these questions, one sentence per answer:

1. Which variable is equivalent to α^t, the step size at iteration t?

   Answer: `self.learning_rate`. For vanilla gradient descent, this is a constant hyper-parameter supplied to the optimizer.

[1]Other possibilities are the Huber loss, or $|r| \approx \sqrt{r^2 + \epsilon}$ for some small ϵ.

2. Which variable is equivalent to $\nabla_w f(w^t)$ the current value of the gradient vector?

   Answer: `g_old`, which is computed by calling `fun_obj.evaluate()` the first time.

3. Which variable is equivalent to w^t, the current value of the parameters?

   Answer: `self.parameters`. This variable is attached to the optimizer's state and gets updated every time `step()` is called.

4. What is the method `break_yes()` doing?

   Answer: `break_yes()` returns true if a termination condition has been reached. The boolean return value is returned by `step()` to communicate with an outer loop that iteratively calls `step()`.

2.4 Robust Regression

The class `LinearModelGradientDescent` is the same as `LeastSquares`, except that it fits the least squares model using a gradient descent method. If you run `python main.py -q 2.4` you'll see it produces the same fit as we obtained using the normal equations.

The typical input to a gradient method is a function that, given w, returns $f(w)$ and $\nabla f(w)$. See `FunObjLeastSquares` inside `fun_obj.py` for an example. Note that the `FunObj` class also has a numerical check that the gradient code is approximately correct, since implementing gradients is often error-prone.[2]

An advantage of gradient-based strategies is that they are able to solve problems that do not have closed-form solutions, such as the formulation from the previous section. The class `LinearModelGradientDescent` has all of the implementation of a gradient-based strategy for fitting a generic linear regression model. Your task is to implement robust regression with gradient descent, whose objective function uses the log-sum-exp approximation.

2.4.1 Implementing the Objective Function

Optimizing robust regression parameters is the matter of implementing a function object and using an optimizer to minimize the function object. The only part missing is the function and gradient calculation inside `fun_obj.py`. Inside `fun_obj.py`, complete `FunObjRobustRegression` to implement the objective function and gradient based on the smooth approximation to the absolute value function (from the previous section). Hand in your code, as well as the plot obtained using this robust regression approach.

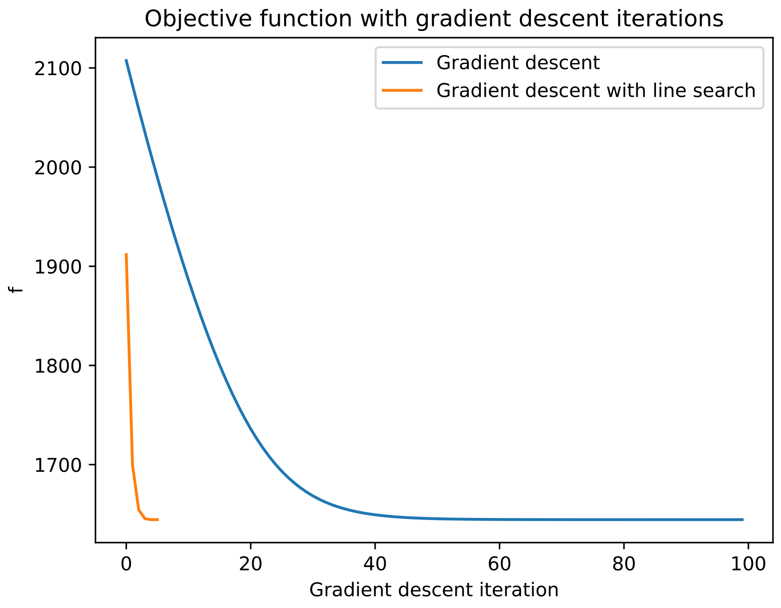
Answer: We obtain similar results to the weighted least squares model with this model: However, in this case we didn't need to know which points were the outliers.

2.4.2 The Learning Curves

Using the same dataset as the previous sections, produce the plot of "gradient descent learning curves" to compare the performances of `OptimizerGradientDescent` and `OptimizerGradientDescentLineSearch` for robust regression, where **one hundred (100) iterations** of gradient descent are on the x-axis and the **objective function value** corresponding to each iteration is visualized on the y-axis (see gradient descent lecture). Use the default `learning_rate` for `OptimizerGradientDescent`. Submit this plot. According to this plot, which optimizer is more "iteration-efficient"?

Answer: See the plot. The line search optimizes for the step size at each iteration, and hence terminates much earlier at fewer than 10 iterations.

[2]Sometimes the numerical gradient checker itself can be wrong. See CPSC 303 for a lot more on numerical differentiation.



3 Linear Regression and Nonlinear Bases

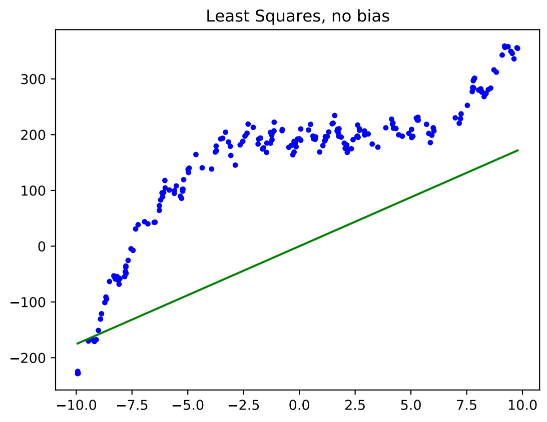
In class we discussed fitting a linear regression model by minimizing the squared error. In this question, you will start with a data set where least squares performs poorly. You will then explore how adding a bias variable and using nonlinear (polynomial) bases can drastically improve the performance. You will also explore how the complexity of a basis affects both the training error and the test error.

3.1 Adding a Bias Variable

If you run `python main.py -q 3`, it will:

1. Load a one-dimensional regression dataset.
2. Fit a least-squares linear regression model.
3. Report the training error.
4. Report the test error (on a dataset not used for training).
5. Draw a figure showing the training data and what the linear model looks like.

Unfortunately, this is an awful model of the data. The average squared training error on the data set is over 28000 (as is the test error), and the figure produced by the demo confirms that the predictions are usually nowhere near the training data:



The y-intercept of this data is clearly not zero (it looks like it's closer to 200), so we should expect to improve performance by adding a *bias* (a.k.a. intercept) variable, so that our model is

$$y_i = w^T x_i + w_0.$$

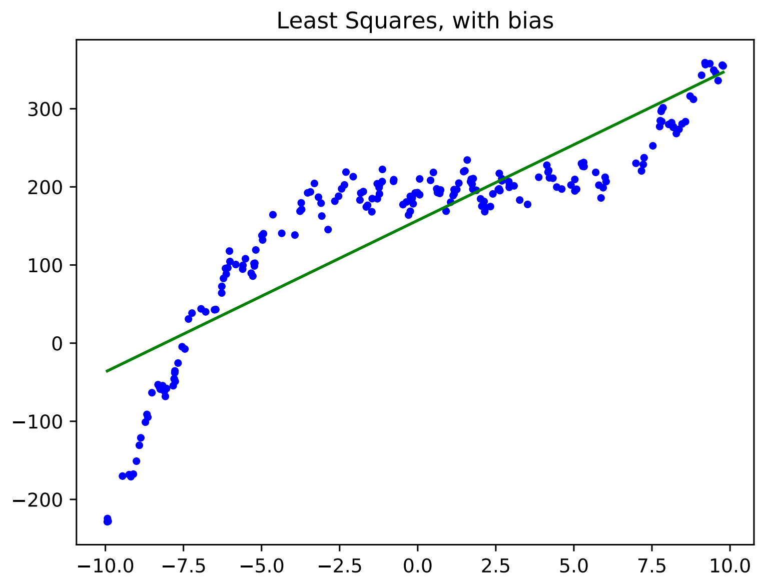
instead of

$$y_i = w^T x_i.$$

In file `linear_models.py`, complete the class `LeastSquaresBias`, that has the same input/model/predict format as the `LeastSquares` class, but that adds a *bias* variable (also called an intercept) w_0 (also called β in lecture). Hand in your new class, the updated plot, and the updated training/test error.

Hint: recall that adding a bias w_0 is equivalent to adding a column of ones to the matrix X. Don't forget that you need to do the same transformation in the `predict` function.

Answer: See accompanying code. Adding a bias drastically reduces the training and test error to 3551.35 and 3393.87 (respectively), and the new plot now looks much better:



3.2 Polynomial Basis

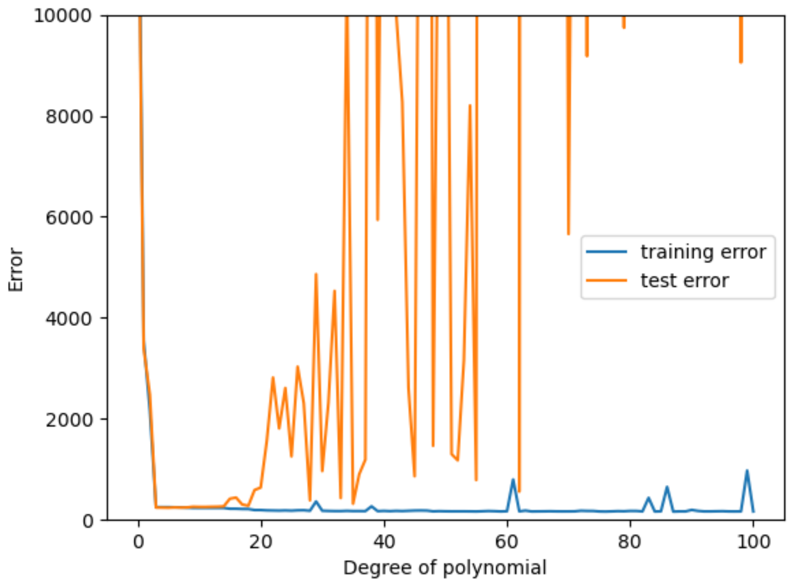
Adding a bias variable improves the prediction substantially, but the model is still problematic because the target seems to be a *non-linear* function of the input. Complete `LeastSquarePoly` class, that takes a data vector x (i.e., assuming we only have one feature) and the polynomial order p. The function should perform a least squares fit based on a matrix Z where each of its rows contains the values $(x_i)^j$ for $j = 0$ up to p. E.g., `LeastSquaresPoly.fit(x,y)` with $p = 3$ should form the matrix

$$Z = \begin{bmatrix} 1 & x_1 & (x_1)^2 & (x_1)^3 \\ 1 & x_2 & (x_2)^2 & (x_2)^3 \\ \vdots & & & \\ 1 & x_n & (x_n)^2 & (x_N)^3 \end{bmatrix},$$

and fit a least squares model based on it. Submit your code, and submit the plot showing training and test error curves for $p = 0$ through $p = 100$. Clearly label your axes. Explain the effect of p on the training error and on the test error. NOTE: large values of p may cause numerical instability. Your solution may look different from others' even with the same code depending on the OS. As long as your training and test error curves behave as expected, you will not be penalized.

Note: you should write the code yourself; don't use a library like sklearn's `PolynomialFeatures`.

Answer: My training and test error curves with y-axis limited at $1e4$ looks like this:



The resulting plots will be different depending on the operating systems. We will give full marks as long as the results are reasonable in terms of how training and test errors behave. The errors under different orders of the polynomial are:

Degree	Training	Test
0	15481	14391
1	3551	3394
2	2168	2481
3	252	243
4	251	242
5	251	240
6	249	246
7	247	243
8	241	246
9	236	259
10	235	256

Your values may differ slightly from the above, depending on the version of Python/Numpy. We see that the *training error goes down as d increases* and the model gets more complicated. On the other hand, the *test error goes down but then later goes up as d increases* (with some oscillation) indicating that we start to overfit as the model gets more complicated.

4 Very-Short Answer Questions

1. Suppose that a training example is global outlier, meaning it is really far from all other data points. What is the difference between how k-means and density-based clustering treat this example?

   Answer: K-means assigns it to the closest mean (no matter how far away the mean is). Density-based clustering doesn't assign it to any cluster.

2. Why do need random restarts for k-means but not for density-based clustering?

   Answer: Density-based clustering is not influenced by the initialization (up to boundary point issues).

3. Can hierarchical clustering find non-convex clusters?

   Answer: Yes (though it may depend on the distance function).

4. For model-based outlier detection, list an example method and problem with identifying outliers using this method.

   Answer: Example is z-score, but would fail on bimodal data with outlier in the middle.

5. For graphical outlier detection, list an example method and problem with identifying outliers using this method.

   Answer: Example is scatterplot, but would fail if you need to see more than 2 dimensions.

6. For supervised outlier detection, list an example method and problem with identifying outliers using this method.

   Answer: Example is decision trees with examples of outliers, but would fail if you have an outlier that doesn't look like the outliers from training.

7. If we want to do linear regression with 1 feature, explain why it would or would not make sense to use gradient descent to compute the least squares solution.

   Answer: The closed-form solution costs $O(n)$ and gradient descent costs $O(nt)$ for t iterations.

8. Why do we typically add a column of 1 values to X when we do linear regression? Should we do this if we're using decision trees?

   Answer: You do this so that the y-intercept is not forced to be zero. This would have no effect on decision trees.

9. Why do we need gradient descent for the robust regression problem, as opposed to just using the normal equations? Hint: it is NOT because of the non-differentiability. Recall that we used gradient descent even after smoothing away the non-differentiable part of the loss.

   Answer: Linear least squares is a special case where everything works out beautifully as a linear system. Almost all other problems don't have that amazing property and need a more general-purpose optimization method.

10. What is the problem with having too small of a learning rate in gradient descent? What is the problem with having too large of a learning rate in gradient descent?

    Answer: If too small, convergence would be extremely slow. If too large, the optimization could not converge (aka diverge).

11. What is the purpose of the log-sum-exp function and how is this related to gradient descent?

    Answer: Smoothes the max function so we can apply gradient descent.

12. What type of non-linear transform might be suitable if we had a periodic function?

    Answer: Trigonometric functions like sines and cosines.