CPSC 340 Assignment 3 (see course page for due date)

Important: Submission Format [5 points]

Please make sure to follow the submission instructions posted on the course website. We will deduct marks if the submission format is incorrect, or if you're not using LaTeX and your handwriting is *at all* difficult to read – at least these 5 points, more for egregious issues. Compared to assignment 1, your name and student number are no longer necessary (though it's not a bad idea to include them just in case, especially if you're doing the assignment with a partner).

1 Matrix Notation and Minimizing Quadratics [12 points]

1.1 Converting to Matrix/Vector/Norm Notation [6 points]

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).

Answer: During class, we defined the following notation. I will be using these conventions throughout this assignment:

$$X = \begin{bmatrix} - & x_1^T & - \\ - & x_2^T & - \\ & \vdots & \\ - & x_n^T & - \end{bmatrix}, w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \text{ and each of } x_i = \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{bmatrix}$$

1. $\max_{i \in \{1,2,\ldots,n\}} |w^T x_i - y_i|$. This is "brittle regression".

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

2. $\sum_{i=1}^n v_i (w^T x_i - y_i)^2 + \frac{\lambda}{2} \sum_{j=1}^d w_j^2$. This is regularized least squares with a *weight* v_i for each training example: Hint: You can use V to denote a diagonal matrix that has the values v_i along the diagonal. What does $a^T V b$ look like in summation form (for some arbitrary vectors a, b)?

   Answer: Let V be $\begin{bmatrix} v_1 & & \\ & \ddots & \\ & & v_n \end{bmatrix}$. Then, we can rewrite the first term as $(Xw-y)^T \times V \times (Xw-y)$.

   The second term is more straightforward, as it is the sum of squared terms of w. Hence, we can write that as $\frac{\lambda}{2}\|w\|^2$.

   Hence, the final expression is $(Xw-y)^T \times V \times (Xw-y) + \frac{\lambda}{2}\|w\|^2$.

3. $\left(\sum_{i=1}^{n} |w^T x_i - y_i|\right)^2 + \frac{1}{2}\sum_{j=1}^{d} \lambda_j |w_j|$. This is L1-regularized least squares with a different regularization strength for each dimension: Hint: You can use Λ to denote a diagonal matrix that has the λ_j values along the diagonal.

   Answer: Let $\Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}$. We can first write the first term as the square of the L1-norm. Thus, the first term is equal to $(\|Xw - y\|_1)^2$. Then, the second term involves the L1-norm again, but this time including λ_j. We can write part as $\frac{1}{2}\|\Lambda w\|_1$. Hence, the final expression is $(\|Xw-y\|_1)^2 + \frac{1}{2}\|\Lambda w\|_1$.

Note: you can assume that all the v_i and λ_i values are non-negative.

1.2 Minimizing Quadratic Functions as Linear Systems [6 points]

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:

$$f(w) = \frac{1}{2}\left\| \begin{bmatrix} w_1 - v_1 \\ w_2 - v_2 \\ \vdots \\ w_n - v_n \end{bmatrix} \right\|^2 = \frac{1}{2}\sum_{i=1}^{n}(w_i - v_i)^2 = \frac{1}{2}\left((w_1 - v_1)^2 + (w_2 - v_2)^2 + \ldots + (w_n - v_n)^2\right).$$

   If one took the partial derivative with respect to w_1, we would see that:

$$\frac{\partial f}{\partial w_1} = (w_1 - v_1)$$

   If we did this with all, we would see that:

$$\frac{\partial f}{\partial w_n} = (w_n - v_n)$$

   Therefore, in vector form:

$$\nabla f(w) = (w - v)$$

   To optimize, $w - v = 0, w = v$.

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

   Answer: The first term in this expression (call it f_1) is the sum of squared residuals expression shown in class. The gradient of this term is:

$$\nabla f_1 = X^T(Xw - y) = X^T X w - X^T y$$

   The second term (f_2) is essentially a quadratic w term, so we end up with a linear expression of w.

$$f_2 = \frac{1}{2}w^T \Lambda w = \frac{1}{2}\begin{bmatrix} w_1 & \cdots & w_n \end{bmatrix} \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \frac{1}{2}(\lambda_1 w_1^2 + \lambda_2 w_2^2 + \cdots + \lambda_n w_n^2)$$

When taking the partial derivatives, all other terms are dropped and we are left with $\frac{\partial f_2}{\partial w_n} = \lambda_n w_n$.

$$\nabla f_2 = \Lambda w$$

Hence,

$$\nabla f(w) = X^T X w - X^T y + \Lambda w$$

To optimize, we would solve the matrix linear equation $(X^T X + \Lambda)w = X^T y$.

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

Answer: Let V be $\begin{bmatrix} v_1 & & \\ & \ddots & \\ & & v_n \end{bmatrix}$. Then, we can write the first term (f_1) as

$$\frac{1}{2}\left(v_1(w^T x_1 - y_1)^2 + \cdots + v_n(w^T v_n + y_n)^2\right)$$

When differentiating, this is similar to the example in class, as the v_is do not interfere with the derivative with respect to w. For example,

$$\frac{\partial f_1}{\partial w_1} = v_1(w^T x_1 - y_1)(x_{1,1}) + \cdots + v_2(w^T x_n - y_n)(x_{1,n})$$

And hence

$$\frac{\partial f_1}{\partial w_n} = v_1(w^T x_1 - y_1)(x_{n,1}) + \cdots + v_2(w^T x_n - y_n)(x_{n,n}) = \langle V(Xw - y), x_n \rangle$$

We can rewrite the collection of all these partial derivatives such that $\nabla f_1(w) = X^T V(Xw - y)$.

The second term is again the norm-squared of the difference of two vectors. This is similar to question 1.2.1, except $v = w^{(0)}$ in this case. Therefore, the gradient of the second term is $\lambda(w - w^{(0)})$. Combining that together, we get that:

$$\nabla f(w) = X^T V(Xw - y) + \lambda(w - w^{(0)})$$

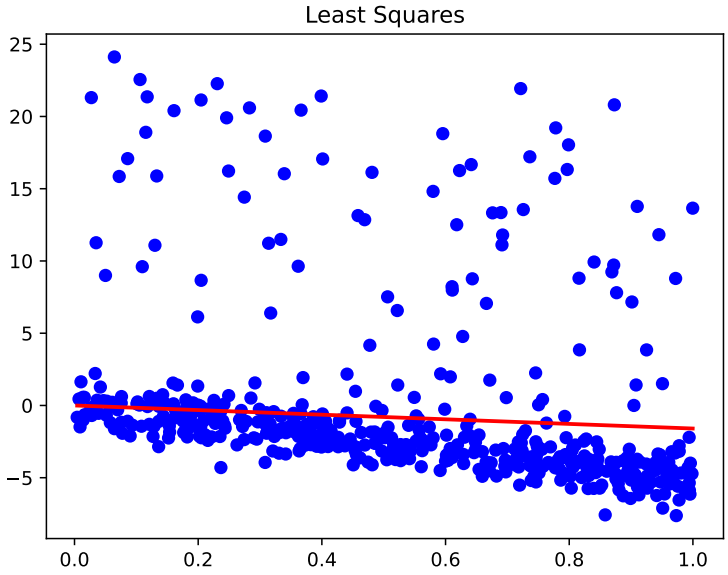
Then, to optimize this value, we would solve the equation $(X^T V X + \lambda I)w = X^T V X y + \lambda w^{(0)}$.

Above we assume that v and $w^{(0)}$ are $d \times 1$ vectors, and Λ is a $d \times d$ 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 spot check, make sure that the dimensions match for all quantities/operations: to do this, you may need to introduce an identity matrix. For example, $X^T X w + \lambda w$ can be re-written as $(X^T X + \lambda I)w$.

2 Robust Regression and Gradient Descent [41 points]

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.

A coding note: when we're doing math, we always treat y and w as column vectors, i.e. if we're thinking of them as matrices, then shape $n \times 1$ or $d \times 1$, respectively. This is also what you'd usually do when coding things in, say, Matlab. It is *not* what's usually done in Python machine learning code, though: we usually have `y.shape == (n,)`, i.e. a one-dimensional array. Mathematically, these are the same thing, but if you mix between the two, you can really easily get confusing answers: if you add something of shape `(n, 1)` to something of shape `(n,)`, then the NumPy broadcasting rules give you something of shape `(n, n)`. This is a very unfortunate consequence of the way the broadcasting rules work. If you stick to either one, you generally don't have to worry about it; **we're assuming shape `(n,)` here**. Note that you can ensure you have something of shape `(n,)` with the `utils.ensure_1d` helper, which basically just uses `two_d_array.squeeze(1)` (which checks that the axis at index 1, the second one, is length 1 and then removes it). You can go from `(n,)` to `(n, 1)` with, for instance, `one_d_array[:, np.newaxis]` (which says "give me the whole first axis, then add another axis of length 1 in the second position").

2.1 Weighted Least Squares in One Dimension [8 points]

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." The v_i are sometimes called *sample weights* or *instance weights* to help distinguish them.

Complete the model class, `WeightedLeastSquares` (inside `linear_models.py`), to implement this model. (Note that Q1.2.3 asks you to show how a 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: As calculated as part of the answer to question 1.2.3, we can put all values of v into the diagonal of an $n \times n$ matrix, V. Then, the gradient for this weighted least squares objective function is:

$$\nabla f(w) = X^T V(Xw - y)$$

To find the minimum, we set this equal to zero, and we solve the following equation using numpy's linalg.solve() function:

$$(X^T V X)w = X^T V y$$

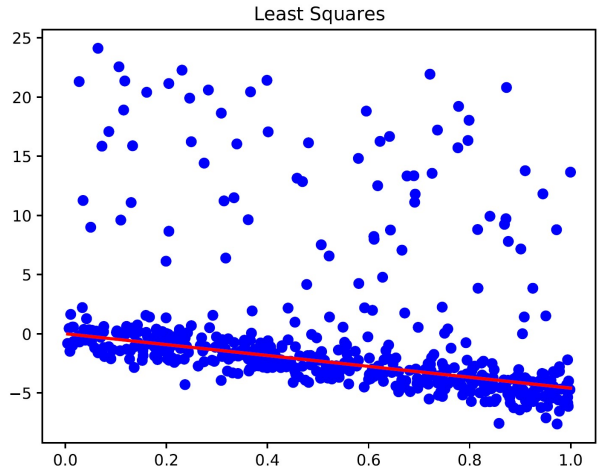
That is exactly what the code below does, in the class `WeightedLeastSquares`.

```
class WeightedLeastSquares(LeastSquares):

    def fit(self, X, y, v):
        self.w = solve(X.T @ v @ X, X.T @ v @ y)
```

Then, in main, to make the v vector, the following code was used:

```
[...]
v = np.ones(500)
v[:400] = 1
v[400:] = 0.1
V = np.diag(v, 0)
[...]
model = linear_models.WeightedLeastSquares()
model.fit(X, y, V)

## + Code for plotting (not shown)
```

The updated plot is shown below:



2.2 Smooth Approximation to the L1-Norm [8 points]

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: The answer is derived as below:

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

$$f(w) = \log(\exp(w^T x_1 - y_1) + \exp(y_1 - w^T x_1)) + \cdots + \log(\exp(w^T x_n - y_n) + \exp(y_n - w^T x_n))$$

To see an example, I will take the partial derivative with respect to w_1.

$$\frac{\partial f}{\partial w_1} = \frac{\exp(w^T x_1 - y_1)(x_{1,1}) + \exp(y_1 - w^T x_1)(-x_{1,1})}{\exp(w^T x_1 - y_1) + \exp(y_1 - w^T x_1)} + \cdots + \frac{\exp(w^T x_n - y_n)(x_{1,n}) + \exp(y_n - w^T x_n)(-x_{1,n})}{\exp(w^T x_n - y_n) + \exp(y_n - w^T x_n)}$$

Here, $x_{1,j}$ is the j-th term in the vector x_1. We can notice that this is similar to the notation for an inner product.

$$\frac{\partial f}{\partial w_1} = \left\langle \frac{\exp(w^T x - y) - \exp(y - w^T x)}{\exp(w^T x - y) + \exp(y - w^T x)} \ , \ x_1 \right\rangle$$

If you were to collect together all of $\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \cdots, \frac{\partial f}{\partial w_n}$, we will get

$$\nabla f(w) = X^T \left(\frac{\exp(w^T x - y) - \exp(y - w^T x)}{\exp(w^T x - y) + \exp(y - w^T x)} \right) = X^T \left(\frac{\exp(Xw - y) - \exp(y - Xw)}{\exp(Xw - y) + \exp(y - Xw)} \right)$$

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

2.3 Gradient Descent: Understanding the Code [5 points]

Recall gradient descent, a derivative-based optimization algorithm that uses gradients to navigate the parameter space until a locally optimal parameter is found. In `optimizers.py`, you will see our 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 `get_learning_rate_and_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: alpha

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

   Answer: `g_old`

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

   Answer: `w_old`

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

   Answer: The `break_yes()` method is checking when the optimization can end. It will return True (and therefore **should** end) when we are at a certain threshold of optimality, or that we have done the desired number of steps.

2.4 Robust Regression [20 points]

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 `funObj` in `LinearModelGradientDescent` for an example. Note that the `fit` function of `LinearModelGradientDescent` 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 most of the implementation of a gradient-based strategy for fitting the robust regression model under the log-sum-exp approximation.

2.4.1 Implementing the Objective Function [15 points]

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.

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

Answer: As shown in my answer to question 2.2, the gradient of our objective function is

$$X^T \left(\frac{\exp(Xw - y) - \exp(y - Xw)}{\exp(Xw - y) + \exp(y - Xw)} \right)$$

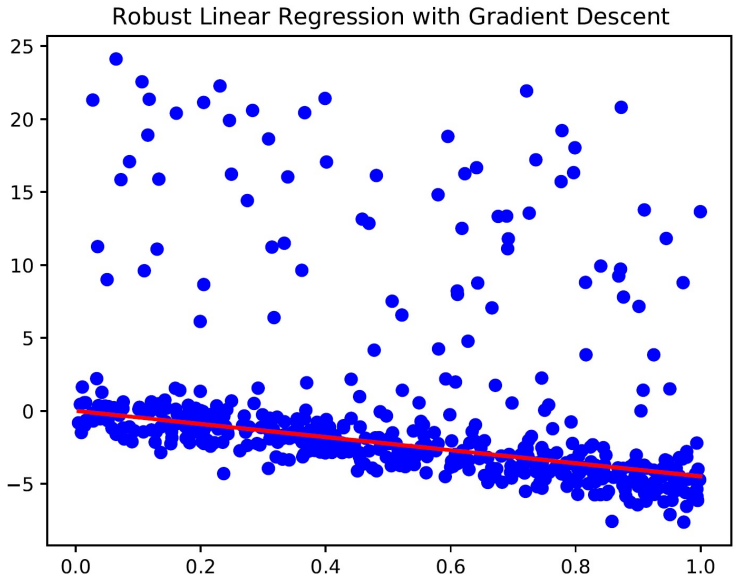
This can be used to to calculate and return the gradient at each w, as demonstrated in the code below:

```
## Gradient
numerator = np.exp(X @ w - y) - np.exp(y - X @ w)
denominator = np.exp(X @ w - y) + np.exp(y - X @ w)
quotient_vector = np.divide(numerator, denominator)
g = X.T @ quotient_vector
```

The objective function **value** also needs to be returned. We can directly substitute in our X, w and y to get the value of f.

```
fn_vector = np.log(np.exp(X @ w - y) + np.exp(y - X @ w))
f = np.sum(fn_vector)
return f, g
```

The plot that uses the robust regression is shown below:

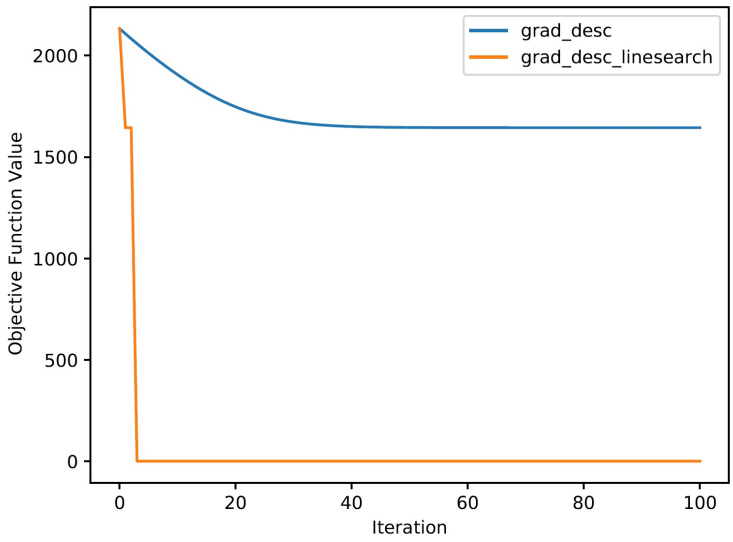


We can see that, even without hard-coding in the outlier points, the regression is somewhat-ignoring the outliers and focusing on the main bulk of data points, just as we wanted it to do.

2.4.2 The Learning Curves [5 points]

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: The plot generated is below.



From the plot, it is clear that the LineSearch method is much more iteration-efficient. In fact, the LineSearch method only underwent 3-4 iterations to get to its best value.

3 Linear Regression and Nonlinear Bases

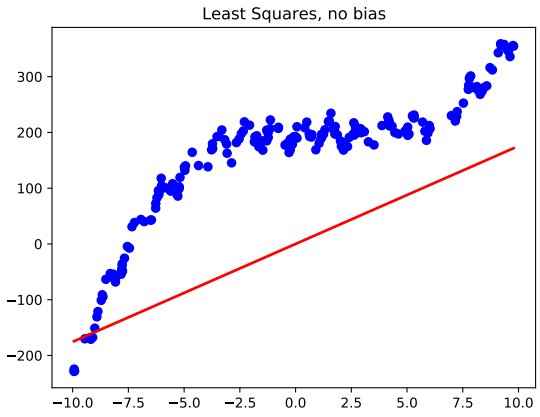
In class we discussed fitting a linear regression model by minimizing the squared error. In this question, you will start with a data set where least squares performs poorly. You will then explore how adding a bias variable and using nonlinear (polynomial) bases can drastically improve the performance.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

3.1 Adding a Bias Variable [8 points]

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 validation error.
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 validation 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/validation 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.

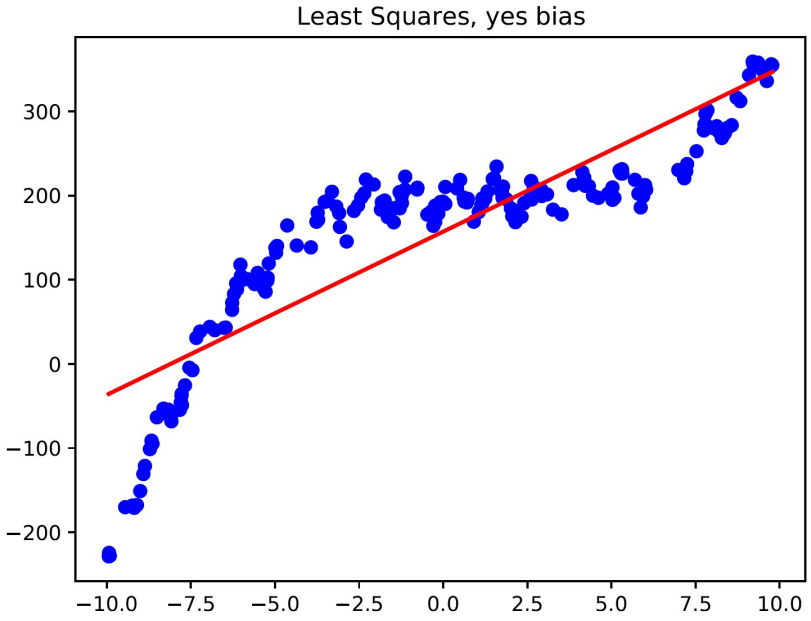
Answer: My code for the `LeastSquaresBias` class is below. The code becomes a lot simpler with the np.vander() function, as it makes the transformed matrix directly.

```
class LeastSquaresBias:
    "Least Squares with a bias added"
    def fit(self, X, y):

        transformed_X = np.vander(X.squeeze(), increasing = True, N=2)
        self.w = solve(transformed_X.T @ transformed_X, transformed_X.T @ y)

    def predict(self, X_pred):

        transformed_Xpred = np.vander(X_pred.squeeze(), increasing=True, N=2)
        return transformed_Xpred @ self.w
```

The training error found was: 3551.3

The validation error was: 3393.9

The plot of the updated plot is below:



3.2 Polynomial Basis [10 points]

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 a plot showing training and validation error curves for the following values of p: $0, 1, 2, 3, 4, 5, 10, 20, 30, 50, 75, 100$. Clearly label your axes, and use

a logarithmic scale for y by `plt.yscale("log")` or similar, so that we can still see what's going on if there are a few extremely large errors. Explain the effect of p on the training error and on the validation 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 and other factors. As long as your training and validation error curves behave as expected, you will not be penalized.

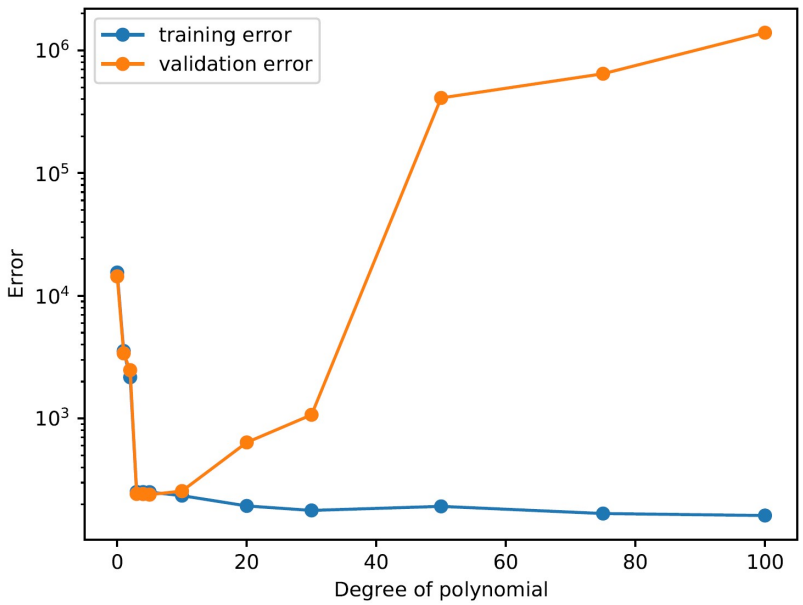
Note: you should write the code yourself; don't use a library like sklearn's `PolynomialFeatures`.

Note: in addition to the error curves, the code also produces a plot of the fits themselves. This is for your information; you don't have to submit it.

Answer: Similar to question 3.1, we can use the np.vander() function here to construct the Z matrix. The `LeastSquarePoly` class is implemented below:

```
class LeastSquaresPoly:
    "Least Squares with polynomial basis"
    def __init__(self, p):
        self.leastSquares = LeastSquares()
        self.p = p

    def fit(self, X, y):
        Z = np.vander(X.squeeze(), increasing= True, N = self.p+1)
        self.w = solve(Z.T @ Z, Z.T @ y)

    def predict(self, X_pred):
        Z_pred = np.vander(X_pred.squeeze(), increasing=True, N=self.p + 1)
        return Z_pred @ self.w
```

The error curves for training/validation error plotted against p is shown below:



From the plot, we can see that p is a parameter that can suffer the issues of overfitting as well. In training error, larger and larger p leads to less and less error, and we can imagine that if we increased p further, we might get 0 training error.

However, in validation error, we can see that even though high p values have close to zero training error, they also have very large validation error, showing that their polynomials are overfitted to the data and cannot generalize well. The validation error also shows a pattern that as p increases from 0, it first decreases, till it reaches its "lowest error point", then starts to increase after that, a typical display of overfitting hyperparameters.

4 Very-Short Answer Questions [24 points]

Answer the following questions (in a sentence or two).

1. Suppose that a training example is global outlier, meaning it is really far from all other data points. How is the cluster assignment of this example set by k-means? And how is it set by density-based clustering?

   Answer: K-means will label as it's own cluster, while density-based methods will not assign it a cluster, it will just be an outlier that belongs to no cluster.

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

   Answer: K-means clustering is very sensitive to the initialization. This is partially because the clustering algorithm **relies** on the "centers" to carry out its algorithm and reach the assignments. Therefore, different initializations can have strong implications for the end result.

   On the flip side, density-based methods are based on the proximity measure ε and distance calculations, but these are **only dependant on the points themselves**. Eventually, all the points will be evaluated in the same way, so the order does not matter as much.

3. Can hierarchical clustering find non-convex clusters?

   Answer: Yes. Part of hierarchical clustering is the **combination** of smaller clusters within the data. In some cases, when you combine two convex clusters you can end up with a non-convex cluster. A simple example is two rectangular-shaped mini-clusters (convex) combining into a "L" shape that is not convex.

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

   Answer: One method is to use Z-scores. The problem with this method arises when the data modeled is **not** normally distributed. For example, using Z-scores on bimodal data (where most data is **not** around the sample mean) can lead to poor outlier detection.

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

   Answer: One example is to use histograms, or variable-pair plots. The problem with both of these methods is that with **higher dimensions**, it becomes very difficult to visualize. If we try to projecting a multi-dimensioned dataset into different planes can cause losses of the context that lead to inaccurate outlier detection.

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

   Answer: This method is done by labelling some data points as "outliers", and thens we train a model to be able to make identify outlier-like points. A problem with this is that the model (however accurate and good) will not be able to identify *new* types of outliers when they come in.

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: No, it would not make sense. With 1 feature, we have very fast and simple methods to do linear regression (like the normal equations). Gradient descent will also likely get the right answer, but is unnecessarily complicated.

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: This is to introduce the bias term, or the constant term in the linear regression. This would **not** make sense with decision trees, as that column is not actually part of our data, but would look like a "feature" of every example. This "pseudo-feature" of the column of 1s would (at best) have no effect on our decision trees since the value would be 1 for every example and every class and thus have no explanatory power; at worst, it may even lead our classification in the wrong direction even though it is not part of our real data.

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: Gradient descent may be good for situations where we do not have closed form solutions to the problem, but we are able to numerically approximate the optimum value.

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 the learning rate is too small, the learning becomes very slow and the running time will be much higher even for simple problems. If the learning rate is too large, then we won't have as much fine tune control on the approximation, and may not be able to find a good solution because we keep taking too "big" of a step over it.

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

    Answer: The log-sum-exp function is a way to approximate a function in its non-differentiable areas as a differentiable function. Gradient descent operates on *the gradient*, so the function we are optimizing must be differentiable at all places - which the log-sum-exp can achieve.

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

    Answer: We can transform it with sine or cosine funtions within the Z matrix.