CPSC 340 Assignment 3

Talisha Griesbach (54645544), Alberto Escobar Mingo (92377860)

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.

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).

1. $\left(\sum_{i=1}^{n} |w^T x_i - y_i|\right)^2$.

   Answer: $||Xw - y||_1^2$.

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: $V||(Xw - y)||_2^2 + \frac{\lambda}{2}||w||_2^2$.

3. $\max_{i \in \{1,2,\ldots,n\}} |w^T x_i - y_i| + \frac{1}{2} \sum_{j=1}^{d} \lambda_j |w_j|$. This is L1-regularized brittle regression 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: $||Xw - y||_\infty + \frac{1}{2}||\Lambda w||_1$.

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:

$$f(w) = \frac{1}{2}\|w - v\|^2$$
$$f(w) = \frac{1}{2}(w - v)^T(w - v)$$
$$f(w) = \frac{1}{2}(w^T w - 2w^T v + v^T v)$$
$$\nabla f(w) = \frac{1}{2}(2w - 2v)$$
$$0 = w - v$$
$$w = v$$

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

   Answer:

$$f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{1}{2}w^T \Lambda w$$
$$f(w) = \frac{1}{2}\left(w^T X^T X w - 2w^T X^T y + y^T y + w^T \Lambda w\right)$$
$$\nabla f(w) = \frac{1}{2}\left(2X^T X w - 2X^T y + 2\Lambda w\right)$$
$$0 = X^T X w - X^T y + \Lambda w$$
$$\left(X^T X + \Lambda\right) w = X^T y$$
$$w = \left(X^T X + \Lambda\right)^{-1} X^T y$$

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:

$$f(w) = \frac{1}{2}\sum_{i=1}^{n} v_i(w^T x_i - y_i)^2 + \frac{\lambda}{2}\|w - w^{(0)}\|^2$$

$$f(w) = \frac{1}{2}V\|Xw - y\|^2 + \frac{\lambda}{2}\|w - w^{(0)}\|^2$$

$$f(w) = \frac{1}{2}(w^T X^T V X w - 2w^T X^T V y + y^T V y) + \frac{\lambda}{2}(w^T w - 2w^T w^{(0)} + w^{(0)T} w^{(0)})$$

$$\nabla f(w) = \frac{1}{2}(2X^T V X w - 2X^T V y) + \frac{\lambda}{2}(2w - 2w^{(0)})$$

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

$$(X^T V X + \lambda I)w = X^T V y + \lambda w^{(0)}$$

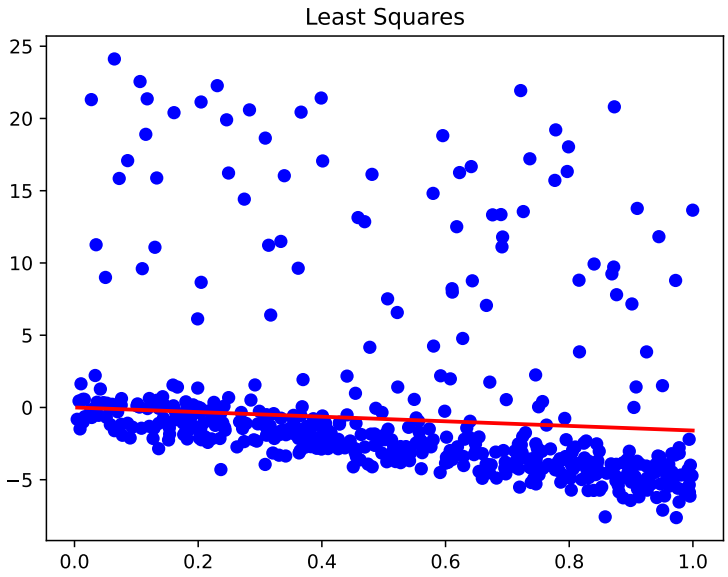
$$w = (X^T V X + \lambda I)^{-1}(X^T V y + \lambda w^{(0)})$$

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 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,

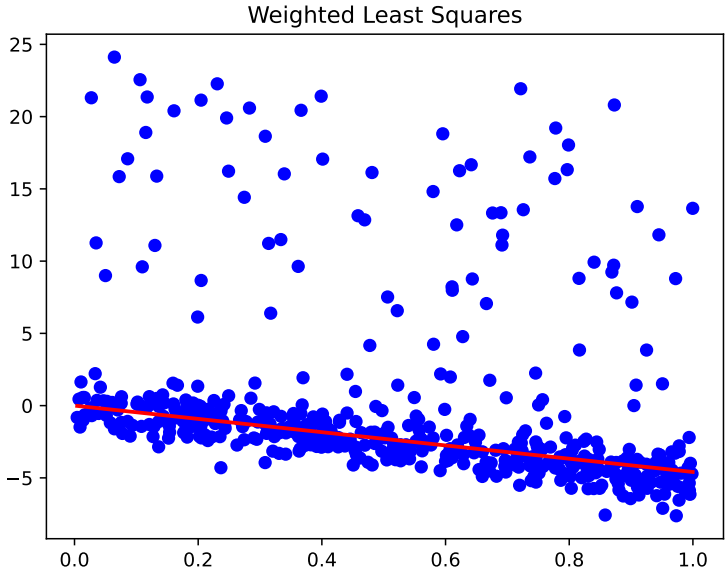
$$f(w) = \frac{1}{2}\sum_{i=1}^{n} v_i(w^T x_i - y_i)^2.$$

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:

```python
#linear_models.py
class WeightedLeastSquares(LeastSquares):
    # inherits the predict() function from LeastSquares
    def fit(self, X, y, v):
        V = np.diag(v)
        self.w = solve(X.T @ V @ X, X.T @ V @ y)

#main.py
@handle("2.1")
def q2_1():
    data = load_dataset("outliersData.pkl")
    X = data["X"]
    y = data["y"].squeeze(1)

    v = np.ones(500)

    # Set the last 100 data points to have a weight of 0.1
    v[400:] = 0.1
    model = WeightedLeastSquares()
    model.fit(X, y, v)
    print(model.w)

    test_and_plot(
        model, X, y, title="Weighted Least Squares",
        ↪  filename="weighted_least_squares_outliers.pdf"
    )
```



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:

$$g_i(w) = w^T x_i - y_i = \sum_{j=1}^{d} w_j x_{ij} - y_i$$

$$\frac{\partial g_i}{\partial w_j} = x_{ij}$$

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

$$\frac{\partial f}{\partial g_i} = \sum_{i=1}^{n} \frac{\exp(g_i) - \exp(-g_i)}{\exp(g_i) + \exp(-g_i)}$$

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

$$\nabla f = \begin{bmatrix} \sum_{i=1}^{n} \frac{\exp(w^T x_i - y_i) - \exp(y_i - w^T x_i)}{\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)} x_{i1} \\ \sum_{i=1}^{n} \frac{\exp(w^T x_i - y_i) - \exp(y_i - w^T x_i)}{\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)} x_{i2} \\ \vdots \\ \sum_{i=1}^{n} \frac{\exp(w^T x_i - y_i) - \exp(y_i - w^T x_i)}{\exp(w^T x_i - y_i) + \exp(y_i - w^T x_i)} x_{id} \end{bmatrix}$$

[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 `GradientDescent`. 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 method breaks the optimization process and therefore the gradient descent when it returns True. The method will return true when either the gradient norm is less than the predefined model optimality tolerance meaning the problem has been solved to the required optimization requirements, or when the number of iterations has reached the predefined maximum number of iterations.

2.4 Robust Regression [20 points]

The class `LinearModel` is like `LeastSquares`, except that it fits the least squares model using a gradient descent method. If you run `python main.py 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 `fun_obj.py` for some examples. Note that the `fit` function of `LinearModel` 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 `LinearModel` has most of the implementation of a gradient-based strategy for fitting the robust regression model under the log-sum-exp approximation.

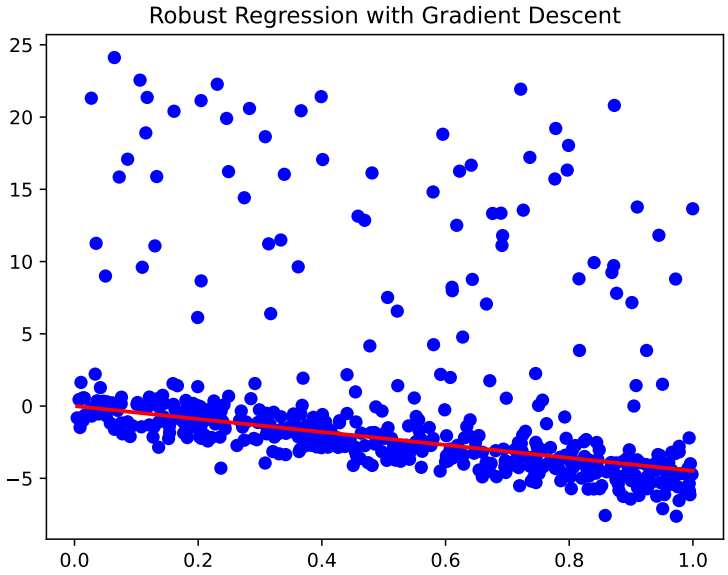
[2]Sometimes the numerical gradient checker itself can be wrong. See CPSC 303 for a lot more on numerical differentiation.

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 `RobustRegressionLoss` 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:

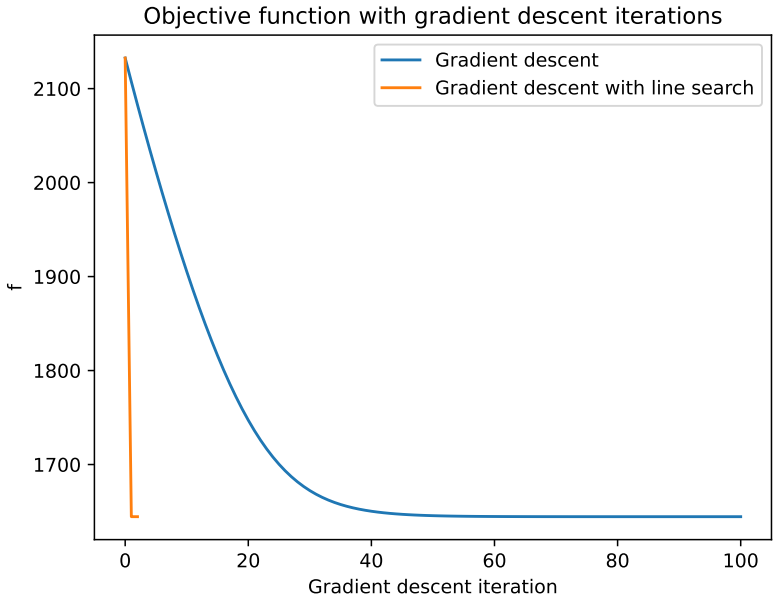
```python
# fun_obj.py
class RobustRegressionLoss(FunObj):
    def evaluate(self, w, X, y):
        """
        Evaluates the function and gradient of ROBUST least squares objective.
        """
        # help avoid mistakes (as described in the assignment) by
        # potentially reshaping our arguments
        w = ensure_1d(w)
        y = ensure_1d(y)

        n, d = X.shape
        f = 0
        partial_fg = np.zeros(n)
        for i in range(n):
            residual = w.T@X[i,:] - y[i]
            f += np.log(np.exp(residual) + np.exp(-residual))
            partial_fg[i] = (np.exp(residual) - np.exp(-residual)) / (np.exp(residual) +
            ↪  np.exp(-residual))

        g = X.T@partial_fg
        return f, g

# main.py
@handle("2.4.1")
def q2_4_1():
    data = load_dataset("outliersData.pkl")
    X = data["X"]
    y = data["y"].squeeze(1)

    fun_obj = RobustRegressionLoss()
    optimizer = GradientDescentLineSearch(max_evals=100, verbose=False)
    model = LinearModel(fun_obj, optimizer)
    model.fit(X, y)
    print(model.w)

    test_and_plot(
        model,
        X,
        y,
        title="Robust Regression with Gradient Descent",
        filename="robust_regression_gd.pdf",
    )
```


Robust Regression with Gradient Descent
25
20
15
10
5
0
−5
0.0
0.2
0.4
0.6
0.8
1.0

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 `GradientDescent` and `GradientDescentLineSearch` 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 `GradientDescent`. Submit this plot. According to this plot, which optimizer is more "iteration-efficient"?

Answer: GradiantDescentLineSearch is more iteration-efficent as it fines the minimum of the objective in less than 5 iterations compared to GradientGescent which takes 50 iterations.



3 Linear Regression and Nonlinear Bases

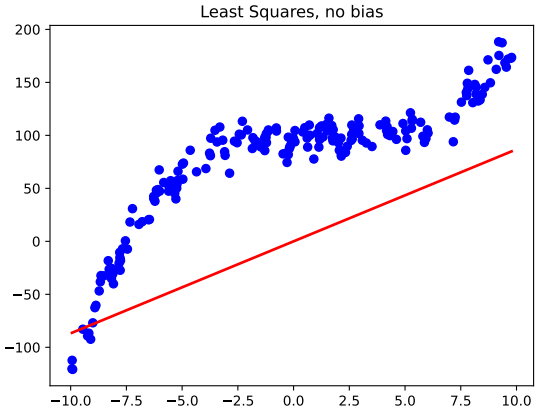
In class we discussed fitting a linear regression model by minimizing the squared error. In this question, you will start with a data set where least squares performs poorly. You will then explore how adding a bias variable and using nonlinear (polynomial) bases can drastically improve the performance. 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 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 7000 (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

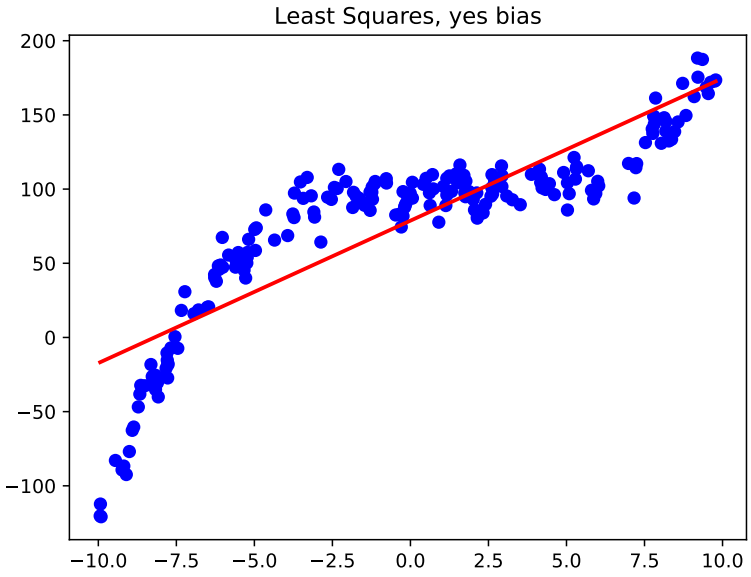
$$y_i = w^T x_i.$$

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:

```
class LeastSquaresBias:
    "Least Squares with a bias added"

    def fit(self, X, y):
        n = X.shape[0]
        ones_column = np.ones((n,1))
        X_with_ones_column = np.hstack([X, ones_column])

        self.model = LeastSquares()
        self.model.fit(X_with_ones_column, y)

    def predict(self, X_pred):
        n = X_pred.shape[0]
        ones_column = np.ones((n,1))
        X_pred_with_ones_column = np.hstack([X_pred, ones_column])

        y_hat = self.model.predict(X_pred_with_ones_column)
        return y_hat
```



Training error = 938.4
Validation error = 844.4

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

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

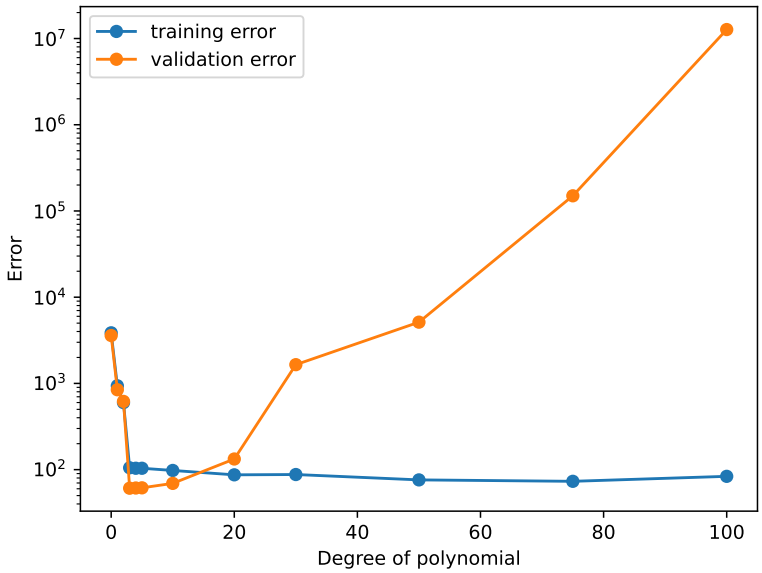
and fit a least squares model based on it. Submit your code, and a plot showing training and validation error curves for the following values of p: $0, 1, 2, 3, 4, 5, 10, 20, 30, 50, 75, 100$. Clearly label your axes, and use a logarithmic scale for y by `plt.yscale("log")` or similar, so that we can 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.

```
class LeastSquaresPoly:
    "Least Squares with polynomial basis"

    def __init__(self, p):
        self.leastSquares = LeastSquares()
        self.p = p

    def fit(self, X, y):
        Z = self._poly_basis(X)
        print(Z[0])
        self.leastSquares.fit(Z, y)

    def predict(self, X_pred):
        Z_pred = self._poly_basis(X_pred)
        y_hat = self.leastSquares.predict(Z_pred)
        return y_hat

    # A private helper function to transform any X with d=1 into
    # the polynomial basis defined by this class at initialization.
    # Returns the matrix Z that is the polynomial basis of X.
    def _poly_basis(self, X):
        n = X.shape[0]
        Z = np.ones([n,1])
        for i in range(1, self.p+1):
            X_power_j = np.power(X, i)
            Z = np.append(Z, X_power_j, axis=1)
        return Z
```



Answer: When p is small, both the training and validation error are large, visible in p = 0 to p = 2. This is a clear indicator that the model is under fit because the polynomial order cannot capture the complexity of the data and therefore cannot make an accurate predictions on the test data.

As p increases, from p = 3 to p = 10, the model can capture the complexity of the training data so the training error decreases, and since the model is better able to capture the complexity of the training data, it can make more accurate predictions on test data which results in a decreasing validation error.

As p increases to even larger values, visible from p = 10 onwards, the complexity of the model increases and results in over fitting the data. Here the training error will be low or zero because the polynomial order will capture all the training data points, i.e overfit to the training points. When new unseen validation data is given, the validation error will be large because the model is too complex, even if training error is low.

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: In k-means, the example will be classified to the closest cluster mean point. In density-based clustering, the example would not be classified to any cluster and just treated as noise.

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

   Answer: K-means algorithm needs to randomize the initialization of cluster centers for its iterative process, and multiple restart attempts result in more optimal clustering and higher accuracy. On the other hand, density-based clustering does not require initialization of centers, and only look at the density of the data points – therefore no random restarts are needed.

3. Can hierarchical clustering find non-convex clusters?

   Answer: Yes, because the method for clustering is bottom-up which allows for non-convex clusters. Hierarchical clusters are made based on a distance metric without assumption of any shapes, thus can capture any shape, including non-convex ones.

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

   Answer: An example method and problem for model-based outlier detection is using a normal distribution model using z-scores, which assumes that the dataset fits a normal distribution model and that outliers lie above a specific z-value threshold. The problem is that this assumption does not account for non-normal distribution datasets, such as a bimodal distribution.

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

   Answer: An example method is using a box plot on your data. Box plots are limited to one variable, so you cannot observe outliers on multiple variables at once.

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

   Answer: An example method would be using a decision tree trained on data classifying examples as outliers or not. A problem with outlier detection is that if a new type of outlier occurs that is not captured by the training data, it could possibly never be detected by the decision tree.

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: It would not make sense, because gradient descent is more computationally heavy compared to the solving the linear system of equations when only 1 feature is involved. Gradient descent would make sense with a high number of features, larger datasets or regularization.

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: A 1 column is added to X for linear regression to obtain a y-intercept (the bias variable) and not force the intercept to be zero during fitting. There is no need to do this for decision trees, since we are making decisions based on one feature at a time and do not rely on linear combinations.

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: Robust regression objective function cannot be solved as a linear system compared to least squares. Because of this, an iterative solution such as gradient descent is needed.

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, it will take a long time for the model to converge to the minimum during fitting. If the learning rate is too large, during training the model might become non-converging and/or oscillatory.

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

    Answer: Log-sum-exp function is used in smoothing the objective function to make it differentiable and help find the gradient, usually when involving maximum algorithms.

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

    Answer: Transforms that use trigonometric functions like sine and cosine would be suitable.