Foundations of Machine Learning
DA5400
ASSIGNMENT 1

SUBMITTED BY

GLEN PHILIP SEQUEIRA
DA24C005

Contents

1. INTRODUCTION

The assignment has been coded in Python. Basic libraries like numpy, pandas, and matplotlib have been used for tasks like matrix multiplications, data handling and visualisation.

The source code contains two python files:

- **Main.py**: It is the main file for demonstration and viewing the results
- **Algorithms.py**: It contains the code for implementation of classes of various regression algorithms

To run the source code smoothly, make sure that both the above python files are in the same directory. Two variables named `train_path` and `test_path` have been defined in **Main.py** right after importing libraries. Make sure to change these variables values accordingly so that the train and test dataset can be loaded.

Throughout the code, Mean Squared Error (MSE) has been used for measuring the error (wherever applicable) since it provides an "average" error, indicating how much the predicted values are farther from the true values of y.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where

- n: number of data points
- y_i: actual value for the target variable at the i-th data point.
- $\hat{y}_i$: predicted value for the i-th data point.

A dataset named `FMLA1Q1Data_train.csv` consisting of 1000 points in $(\mathbb{R}^2, \mathbb{R})$ has been provided. The dataset is stored in a variable named `train_data`. We name the first two columns of `data` as `x1`, `x2` and the last column as `y`. We add a column named `bias` in `X`, which consists of all 1's. This is done to fit the linear regression model appropriately so that the intercept/bias term is captured in the weight vector. We extract `x1`, `x2` and `bias` of `train_data` into a $n \times d$ feature matrix `X`. The last column of `train_data` is extracted into a $n \times 1$ matrix `y`, which stores the output/target values.

A similar process as above is done to create a variable called `test_data`, which is created using`FMLA1Q1Data_test.csv`. `test_data` is used to test the effiency of the models on unseen data.

A user-defined function named `split_dataset(X, y, fold, num_folds)` has been used for splitting dataset into k folds and return $k-1$ folds as the training set and 1 fold as the validation set. It is used in both `ridge_regression` class and `kernel_regression` class during cross-validation.

2. ANALYTICAL SOLUTION

APPROACH

In linear regression, we try to find a weight or coefficient vector **w** that minimizes the difference between the predicted values and the actual values. The **least squares solution** gives an analytical formula for finding the best-fitting line by minimizing the SSE.

The **Least Squares solution**, also known as **Ordinary Least Squares (OLS)** solution is the value of **w** that minimizes the **sum of squared errors (SSE)** between the predicted and actual values:

$$\text{SSE} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \|y - X\mathbf{w}\|^2$$

To minimize the SSE, we differentiate it with respect to **w** and set the derivative to zero. From this, we obtain the closed-form **analytical solution**:

$$\mathbf{w}_{\text{ML}} = (X^T X)^{-1} X^T y \tag{a}$$

$\mathbf{w}_{\text{ML}}$ is called the **Maximum Likelihood (ML)** solution.

To obtain $\mathbf{w}_{\text{ML}}$, we use the Fisher's Likelihood function and try to estimate the parameter **w** by maximising the log-likelihood. The solution obtained by using this method is equivalent to the Least squares solution as linear regression with squared error is equivalent to assuming a Gaussian model where the noise in the observed data is normally distributed with mean zero and fixed variance.

CODE

`w_ml` is the maximum likelihood solution (least squares solution), which is calculated using the analytical method. The predicted values, `y_pred` are calculated by performing a matrix multiplication of the feature matrix `X` and `w_ml`. The Mean Squared Error (MSE) between the actual values `y` and `y_pred` is calculated and printed.

OBSERVATIONS

$$\mathbf{w}_{\text{ML}} = \begin{bmatrix} 9.894 \\ 1.766 \\ 3.522 \end{bmatrix}$$

Mean Squared Error (MSE): 123.365

3. GRADIENT DESCENT APPROACH

Gradient descent is used in linear regression while using large datasets. Directly computing the inverse of $\mathbf{X}^\top\mathbf{X}$ in the closed-form solution (equation (a)) for linear regression is a $O(d^3)$ operation, where d is the number of features. This becomes impractical when d is very large. Gradient descent updates the weight vector, $\mathbf{w}$ during each iteration, avoiding the need to compute the inverse directly. This approach has a lower cost,$O(nd)$ per iteration, where n is the number of samples.

The cost function $f(\mathbf{w})$ using MSE is:

$$f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y}),$$

To find the gradient of $f(\mathbf{w})$ with respect to $\mathbf{w}$, we take the derivative:

$$\nabla f(\mathbf{w}) = \frac{\partial f(\mathbf{w})}{\partial \mathbf{w}} = \frac{2}{n}\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}),$$

The weight vector is updated during each epoch, based on the following update rule:

$$w_{t+1} = w_t - \eta_t \nabla f(w_t)$$

- w_t is the weight vector at iteration t,
- η_t is the step size or learning rate,
- $\nabla f(w_t)$ is the gradient of the loss function at w_t.

Choice of Step-Size

In gradient descent algorithm, the choice of step size, η_t must be made carefully to ensure it converges to a solution efficiently without oscillating or diverging.

$\eta_t = \frac{1}{t}$ is chosen as the step-size. This is justified as the following conditions are met:

$$\sum_{t=1}^{\infty}\eta_t = \sum_{t=1}^{\infty}\frac{1}{t} = \infty$$

Divergent sum condition ensures that the algorithm continues making updates and does not stop before converging towards the solution.

$$\sum_{t=1}^{\infty}\eta_t^2 = \sum_{t=1}^{\infty}\frac{1}{t^2} < \infty$$

Convergent sum condition ensures that the step sizes decrease quickly enough to prevent overshooting and oscillations in order to allow convergence.

CODE

`gradient_descent` class implements the gradient descent algorithm.

Attributes

- `X` is the feature matrix of training data with dimensions $n \times d$, where n represents the number of data points and d represents the number of features.
- `y` The target vector with dimensions $n \times 1$
- `epochs` specifies the number of times for the entire dataset is processed during gradient descent
- `w` is the weight vector initialized to zeros, with dimensions $d \times 1$
- `w_ml` is the analytical solution

Methods

- `calculate_weights` performs the gradient descent algorithm, depending on number of epochs given. During each epoch t, it calculates the step size $\frac{1}{t}$. It updates the weights based on the gradient and step size. After each update, it calculates $\|w_t - w_{\text{ML}}\|_2$ and appends this value to `errors`. It returns the final weight vector and the list of weight errors for all iterations.
- `mse` calculates and returns MSE between the predicted values $\mathbf{Xw}$ and the actual values `y`.

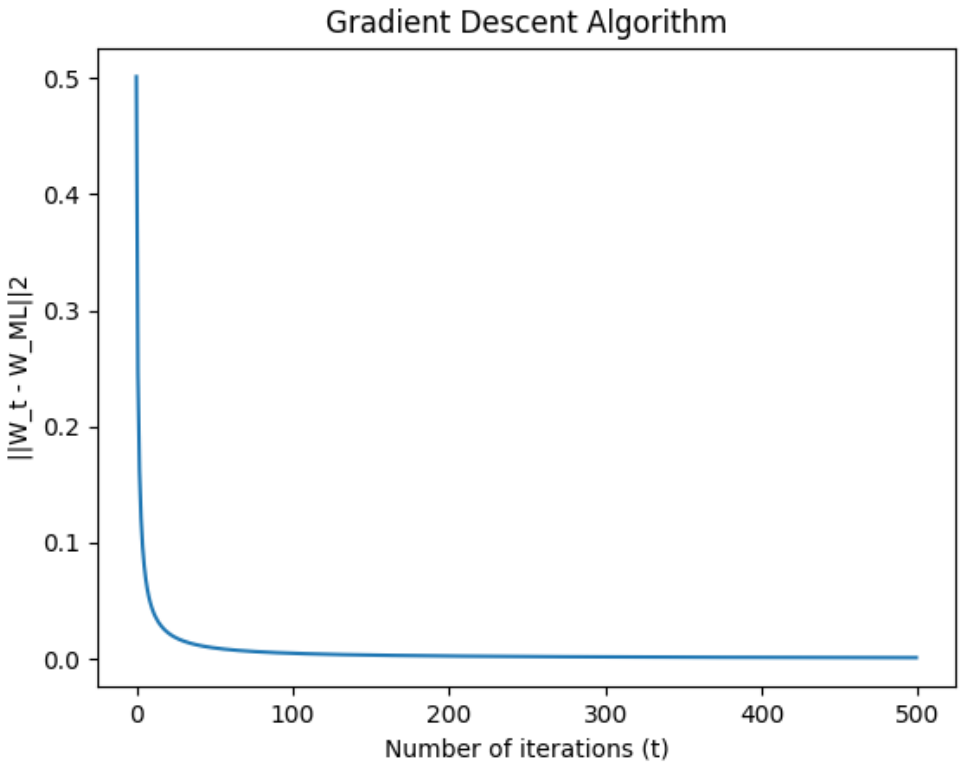
NOTE:

The terms epoch and iterations have been used in this report. These words refer to different things. One epoch is completed when the entire data-set has been processed or used in the algorithm. But one iteration is completed when the update rule has been applied. In case of Gradient Descent, the number of iterations and epochs are same but in Stochastic Gradient Descent number of iterations is equal to the product of number of epochs and batch size.

OBSERVATIONS

$$\mathbf{w}_{\mathrm{ML}} = \begin{bmatrix} 9.894 \\ 1.765 \\ 3.522 \end{bmatrix}$$

Mean Squared Error for training data(MSE): 123.365



The above graph visualizes $\|w_t - w_{\mathrm{ML}}\|_2$ as a function of number of iterations(t). We observe that the graph is smooth. This is because in gradient descent, the update rule is calculated using the entire dataset. So, any randomness in the data will average out, giving a smooth value for the updates.

Also, the graph is steep initially, it drops suddenly but after a few iterations, it is somewhat flat. The weight vector is initialised with an arbitrary value, in our case [0 0 0]. This value, (w_0) is usually far from the actual solution (w_{ml}). We know -$\nabla f(w_t)$ points in the direction of steepest descent. So, during initial iterations, the gradient values are large, leading to large updates in weight vector. Thus, values of $\|w_t - w_{\mathrm{ML}}\|_2$ drop significantly during intial iterations. As the value of $\|w_t - w_{\mathrm{ML}}\|_2$ approaches 0, the gradient values become smaller because the cost function is flatter near the minimum. This leads to smaller updates to the weights. If we choose a suitable step-size and number of epochs, the gradient descent will eventually converge towards w_{ML}, which is clear through the graph.

4. STOCHASTIC GRADIENT DESCENT APPROACH

Stochastic Gradient Descent (SGD) is used for large datasets. It is done to avoid calculating the X^TX matrix, which is a $(O(n^2))$ operation. This is very expensive computationally when the number of data-points n becomes very large.

Instead of performing calculations on the entire data, SGD performs calculations on a smaller subset of data called a **mini-batch**. The size of a mini-batch is called **batch-size** The gradient is calculated using the mini-batch. Some noise is introduced in the calculations but over many iterations, this noise averages out, converging to the optimal solution.

In gradient descent, the gradient is calculated as:

$$\nabla_w f(w) = -\frac{2}{n} X^T (y - Xw)$$

Let us define the mini-batch feature matrix as $\tilde{X}$ and mini-batch of target vector,$\tilde{y}$.The gradient for each mini-batch is:

$$\nabla_w f(w) \approx -\frac{2}{m} \tilde{X}^T (\tilde{y} - \tilde{X}w)$$

CODE

`stochastic_gradient_descent` is used to implement the stochastic gradient descent algorithm.

Attributes

- `X`, `y`, `epochs`, `w`, and `w_ml` are as defined in `gradient_descent` class
- `batch_size` is the number of data points in each mini-batch. Default is taken as 100

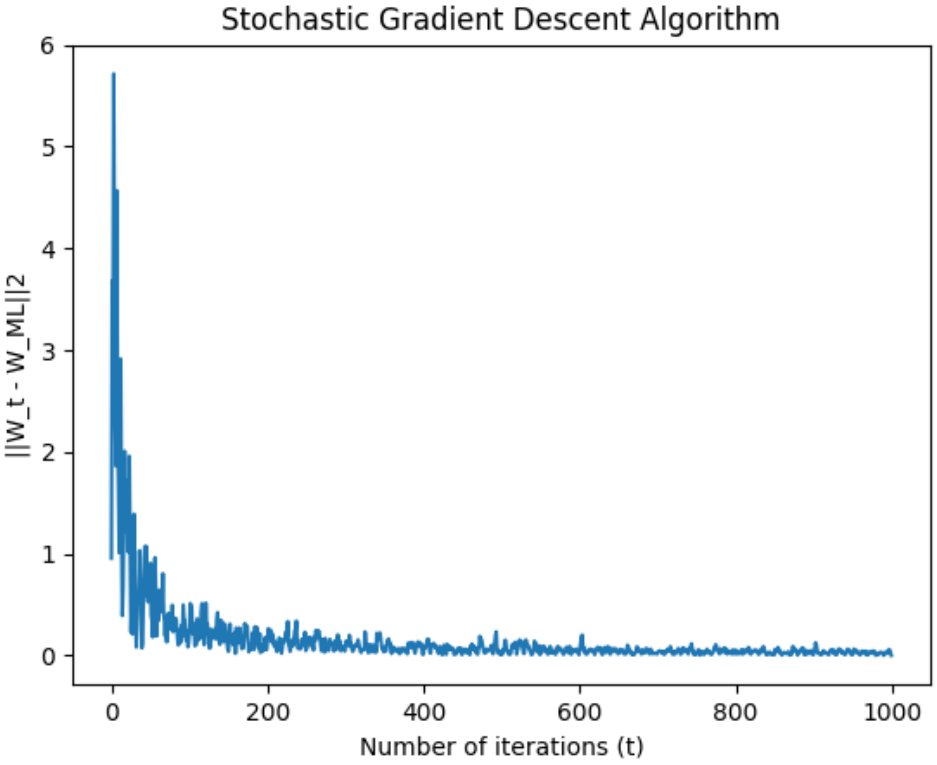
Methods

- `calculate_weights` executes the stochastic gradient descent algorithm. During each epoch t, it randomly shuffles the data indices array, `indices` and divides the data into mini-batches of size `batch_size`. The feature matrix of each batch is X_batch. The weights are updated using gradient and step size $\frac{1}{t}$. It calculates $\|w_t - w_{\mathrm{ML}}\|_2$ and appends it to `errors`. It returns the final weight vector and the list of weight errors for all iterations,`errors`.
- `mse` is similar to the method used in `gradient_descent` class

OBSERVATIONS

$$\mathbf{w}_{\mathrm{ML}} = \begin{bmatrix} 9.895 \\ 1.765 \\ 3.522 \end{bmatrix}$$

Mean Squared Error for training data(MSE): 123.365



The above graph represents $\|w_t - w_{\mathrm{ML}}\|_2$ as a function of the number of iterations(t). The graph of SGD is not smooth, it has a zig-zag path. Also, it takes more number of iterations to converge to optimal solution. But, the behaviour of dropping steeply initially and then flattening is similar to GD graph.

In stochastic gradient descent, the weight updates are calculated using a mini-batch, instead of the entire dataset. This leads to randomness or noise in the updates of the weight vector. Since, we are sampling the data points randomly, direction of current update can be very different from the previous udpate. The graph of SGD has a noisy path whereas regular GD has a smooth graph. But, the noise averages out and over many iterations, SGD converges to the optimal solution.

Though we have taken only 100 epochs, the number of iterations for SGD are 1000, whereas GD converges in 500 iterations. This shows that SGD takes more iterations to converge to w_{ML} than regular gradient descent.

5. RIDGE REGRESSION

APPROACH

In ordinary linear regression, the goal is to minimize the mean squared error (MSE) between predicted and actual values:

$$\text{MSE} = \frac{1}{n}\|Xw - y\|^2$$

In **Ridge Regression**, we add an $L2$ penalty by adding λ, the regularization parameter, to the cost function to penalize large weights. The cost function can be expressed as:

$$J(w) = \frac{1}{n}\|Xw - y\|^2 + \lambda\|w\|^2$$

The gradient of the cost function with respect to w is given by:

$$\nabla_w J(w) = \frac{1}{n}X^T(Xw - y) + \lambda w$$

We solve the ridge regressioin problem using gradient descent:

$$w^{(t+1)} = w^{(t)} - \eta\left(\nabla_w J(w)\right)$$

Cross-validation for λ

Choosing the right value of λ is very important since a very small choice of λ reduces the problem to a regular linear regression model and very large values of λ lead to underfitting.

To find the best value of λ for our ridge regression model, we do **cross-validation** for a given set of λ values. The value of λ which gives the least error in cross-validation is chosen to be the best value of λ. We take 0-50 as λ values for the first step of cross-validation. After we get an optimal λ, we try to find a better value of λ in the interval $[\lambda - 2, \lambda + 2]$ by doing cross-validation again.

CODE

`ridge_regression` class is used to implement ridge regression.

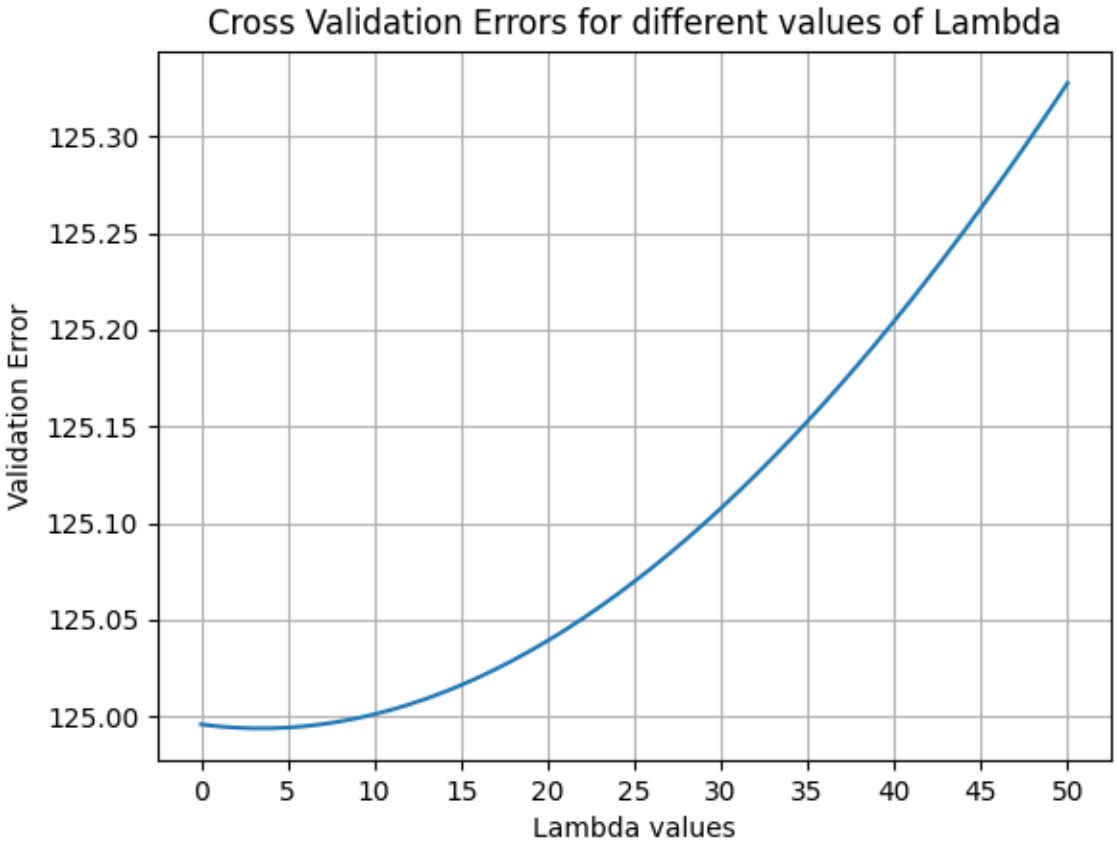
Attributes

- `X`, `y` and `epochs` are as defined in the previous classes
- `X_test` and `y_test` are the feature matrix and target vector, respectively for the test data-set.
- `num_folds` is the number of folds for cross-validation. Default value is 5.
- `w_ridge` is solution of ridge regression

Methods

- `ridge` gives the weight vector, $w_R$for a given value of λ. To calculate this, it uses gradient descent algorithm. It returns the final value of w_R after gradient descent.
- `lambda_cv` does cross-validation to find the best λ from a given list of λ values. For each λ, it splits the dataset into training and validation sets using the number of folds specified by `num_folds`. For this, it uses a user-defined function named `split_dataset(X_train, y_train, fold, num_folds)`. It calculates the validation error for each fold and then returns the average cross-validation error for each λ.
- `calculate_weights` does cross-validation twice to find the best value of λ. It first uses a broad range of λ values from `lambdas1`. Cross-validation is done again to find a better value of λ around the optimal value of λ found from `lambdas1`. It calculates $\mathbf{w_R}$ using the best λ. It returns $\mathbf{w_R}$, the best λ. It also returns the average cross-validation error for each λ value, which is used to plot a graph.

OBSERVATIONS



$$\mathbf{w}_{\mathrm{R}} = \begin{bmatrix} 9.86 \\ 1.758 \\ 3.511 \end{bmatrix}$$

Best value of λ: 3.45

Test data error using w_R: 65.953

Test data error using w_{ML}**:**: 66.005

We observe that the test error using w_R is slightly less compared to w_{ML} because ridge regression uses a L2 penalty term in the cost function. It reduces the length of the weight vector by shrinking the coeffecients. If there is any redundancy in the features, ridge regression retains the necessary features while shrinking the redundant features close to 0. This reduces overfitting on training data, which helps ridge regression to perform relatively well on unseen test data as compared to standard linear regression.

6. KERNEL REGRESSION APPROACH

We have found out the solution using analytical method, gradient descent method, stochastic gradient descent and ridge regression but we obtain a relatively high MSE in these methods. These relatively high error values suggest that the features and the target variable have a non-linear relationship. Thus, kernel regression is a suitable way to find the solution to the regression model.

Polynomial Kernel

The justification for choice of polynomial kernel is given below in the *Observations* section. The kernel matrix K is calculated by applying the polynomial kernel function to every pair of points x_i and x_j in the training data.

$$K[i, j] = (x_i \cdot x_j + 1)^d$$

$k(X_i, X_{\text{test}})$ calculates the similarity between X_i and X_{test}. The vector α represents the coefficients that determine how important each training point is in the prediction process.α can be calculated using the equation:

$$\alpha = K^{-1}y$$

- K^{-1} is the inverse of the kernel matrix K,
- y is the target vector

To predict the output for a new data point X_{test}, we apply the formula:

$$\hat{y}_{\text{test}} = \sum_{i=1}^{n} \alpha_i k(X_i, X_{\text{test}})$$

- α_i is the i-th element of the α vector,
- $k(X_i, X_{\text{test}})$ is the kernel function applied to training point X_i and test point X_{test}.

Cross validation for degree of polynomial kernel

Here, degree, d of the polynomial kernel is a hyper-parameter. Hence, we perform cross-validation to determine the optimal degree of the polynomial kernel. We provide a list of degrees (0-10) for which the kernel matrix, α and the polynomial kernel is calculated. Then, we calculate the training error for each degree. The degree which gives minimum error is chosen as the best degree of the polynomial.

CODE

`kernel_regression` is designed to implement polynomial kernel regression with cross-validation to select the best degree of the polynomial kernel. It includes the following attributes and methods:

Attributes:

- `X`, `y`, `X_test`, `y_test` and `num_folds` are similar to what has been defined in `ridge_regression` class

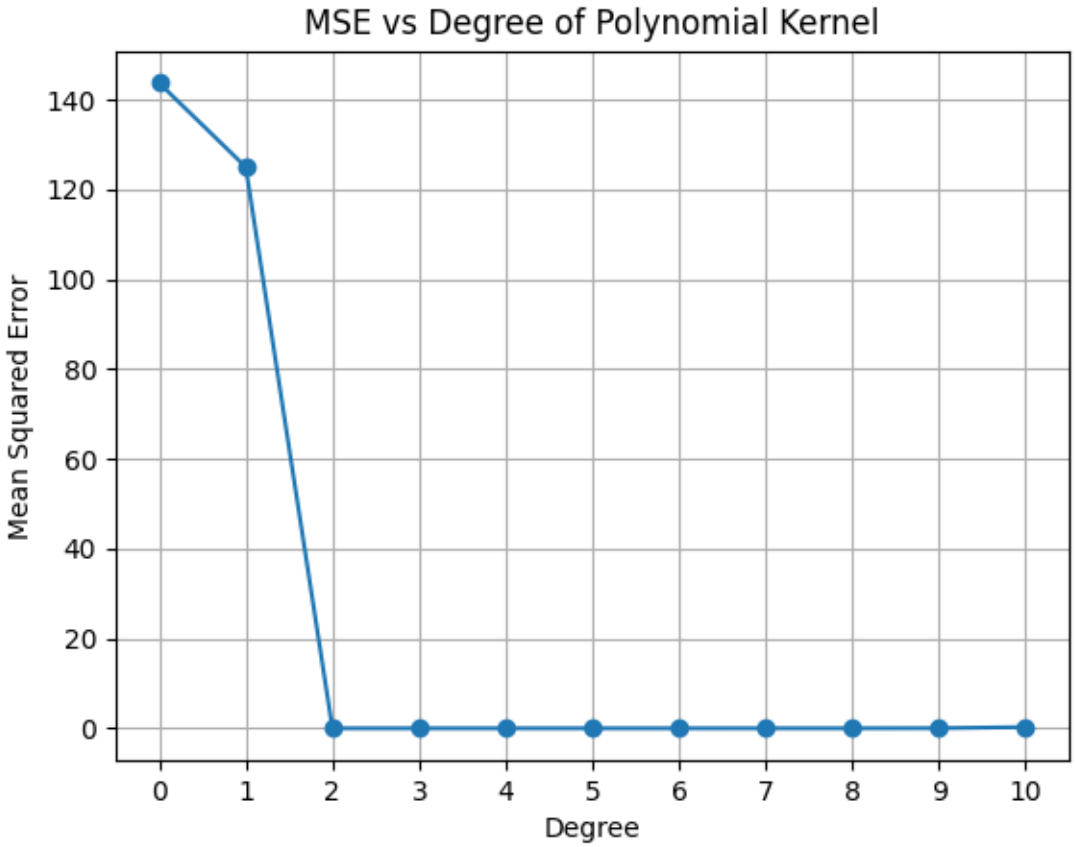
Methods

- `cv` does cross-validation to find the best degree of the polynomial kernel. We initialise the list of degrees to cross-validate for from 0-10. For each degree, it splits the data into testing and validation sets using `split_dataset` function. It then calculates the kernel matrix and α using `X_train` and `y_train`. It predicts the values and calculates MSE by comparing the prediction values with `y_val`. The MSE for each fold is stored in the list `error` and then the average value for each degree is stored in the `mse` list. For each degree, we compare the MSE value with the least MSE and find the best degree. This method returns the best degree and the `mse` list which store average cross-validation error for each degree.

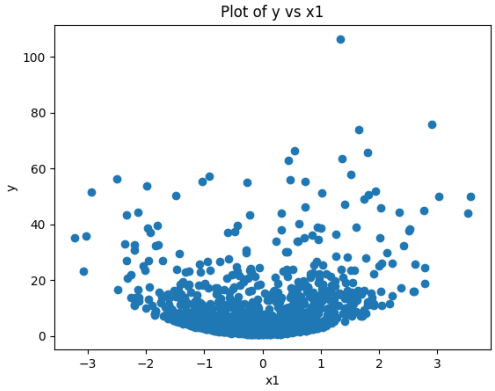
- `predict` predicts the output for `X_test` using the polynomial kernel with the given α and degree.

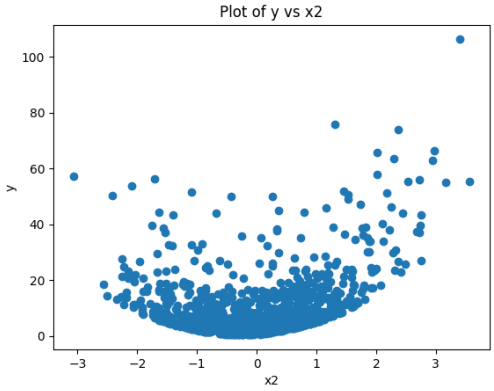
- `alpha` method finds the best degree of the polynomial kernel using the `cv`. It returns the best degree, the corresponding value of
  alpha and the list of cross-validation errors.

OBSERVATIONS



Best polynomial kernel is of degree: 2 **with MSE**: 0.0129





Choice of Kernel Function

We have a few choices for the type of kernel, namely, Gaussian, Polynomial, Sigmoid, Laplacian and Exponential kernel. We visualise our dataset by plotting each of the feature variable, `x1` and `x2` against `y`. From the plots above, we get an intuition that the features have a quadratic-like relationship with the target variable. Hence, we choose a Polynomial Kernel for our regression model as it will help us model these quadratic interactions. It also captures cross-terms like `x1` $\cdot$ `x2`.

Polynomial kernel is better than the standard least squares regression as the standard least squares regression cannot capture non-linear relationship between the feature variables. It can only capture linear relationships by finding the line of best fit. This is strongly backed by the test errors of polynomial kernel and w_{ML} as given below:

Test data error using w_{ML}:: 66.005

Test Error(MSE) using polynomial kernel of degree 2: 0.0093