Linear regression and gradient descent

by Anshuman Singh

Task-1

Write a piece of code to obtain the least squares solution wML to the regression problem using the analytical solution.

To solve the regression problem using the least squares analytical solution, we need to find the optimal weight vector w^*that minimizes the error in the linear regression model. This can be done by solving the following equation:

$$w^* = (X^T X)^{-1} X^T Y$$

where:

- X is the matrix of input features of shape n x d, where n is the number of samples and d is the number of features,
- Y is the target vector,
- w^* is the vector of weights (coefficients) that minimize the squared error

We have used pseudo-inverse of (X * X^T) term to calculate w* in the code. This is because **pseudo-inverse** generalizes the concept of the inverse to handle all cases where (X * X^T) matrix is non-invertible (due to multicollinearity) or if its inverse calculation is numerically unstable due to round-off errors. The pseudo-inverse is more stable and provides a more accurate solution when dealing with such matrices.

Our training dataset contains 1000 points and test dataset contains 100 data points. Both have two features, $feature_0$ and $feature_1$ along with the target variable y.

For our dataset, the X is a matrix containing $feature_0$ and $feature_1$ as columns. Also, bias term column of 1's was added to the X matrix as the first column, thus making X a matrix of dimensions 1000 × 3. Y has the dimensions 1000 × 1.

Using the above function, we get the following value for w^*:

```
w_ml: [[9.89400832]
 [1.76570568]
 [3.5215898 ]]
```

Once $w*$ is obtained, we can make the predictions using $w*$ on the test dataset using the following equation:

$$predictions = X\_test * w^*$$

This gives predictions in the form of n x 1 matrix, where n are the number of sample points in the test dataset.

> 💡 *NOTE: Source code for all algorithms/functions are given at the end of the report*

For this assignment, **MSE (mean squared error)** has been used as the error metric. MSE is a widely used metric in regression problems to measure the difference between the actual (true) values and the predicted values. It quantifies the average squared difference between the predicted values (from a model) and the actual values (ground truth).

Mathematically, MSE is defined as:

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

where:

- n is the number of data points
- y_i is the actual value (ground truth) for the i-th data point
- $\hat{y}_i$ is the predicted value for the i-th data point

The MSE for predictions made using w^* obtained from least squares solution on the test dataset is **66.005**.

Task-2

Code the gradient descent algorithm with suitable step size to solve the least squares algorithms and plot $||w^t - w_m l||_2$ as a function of t. What do you observe?

The gradient descent algorithm finds the w^* value by iteratively moving towards the steepest descent direction defined by the negative of the gradient. It's widely used in machine learning and statistics, especially for optimizing loss functions.

General form of gradient descent algorithm is as follows:

$$w = w - \eta \nabla L(w)$$

Where η is the step size (learning rate) and $\nabla L(w)$ is the gradient.

The gradient for the objective function for linear regression is given by:

$$\nabla L(w) = 2X^T(Xw - y).$$

The gradient descent algorithm function used in the source code has a maximum number of iterations limit of 10^4. This is done so that if our gradient descent algorithm doesn't converge after 10^4 iterations, the algorithm will stop running and the final w^t will be given as output. Also, the algorithm will be considered to have converged when the gradient $\nabla L(w)$ becomes smaller than 10^{-6}. This is done to avoid un-necessary iterations.

The gradient descent algorithm was first applied using a decreasing step size given by $\text{step} = \frac{1}{\text{iters}+1}$, where $iters$ is the current iteration. But, it was observed that this step size was very large causing very large updates, and the algorithm didn't converge, giving very large values of w. Therefore, another parameter in the form of $step\_size\_multiplier$ was added as a multiplicative co-efficient to the step size function to keep the step size small. Thus, the final step size function used in the gradient descent algorithm is given by:

$$\text{step} = \frac{1}{\text{iters} + 1} * step\_size\_multiplier$$

Therefore, we have to find the appropriate $step\_size\_multiplier$ value for the gradient descent algorithm. For that, we have plotted the values taken by weights of w^t corresponding to $feature_0$ and $featuer_1$ for different values of $step\_size\_multiplier$ during the gradient descent algorithm in figure-1. The x-axis shows the value of the weight coefficient for $feature_0$, while the y-axis displays the corresponding value for $feature_1$. The plot clearly highlights the dynamic behaviour of the weights during the gradient descent process for each value of $step\_size\_multiplier$. The following observations can be made from figure-1:

1. When the value of $step\_size\_multiplier$ is 10^{-1}, w^t takes on very large values, ranging up to 10^{62}, before it converges after 215 iterations. This is because the step size is very large, leading to very large updates in the value of w^t at each iteration, causing large oscillations in the value of w^t. This shows that the optimization algorithm is not stable and $step\_size\_multiplier$ value is too large.
2. For $step\_size\_multiplier$ = 10^{-2}, the number of iterations taken by gradient descent algorithm for convergence is 32, but the range of values taken by w^t are still very large. This indicates volatility, and suggests that we should choose a lower value for $step\_size\_multiplier$.
3. For $step\_size\_multiplier$ = $10^{-2.5}$ and 10^{-3} the range of values taken by w^t is now less than 100, but it still shows that the large step size is causing w^t to oscillating between large values before converging.
4. $step\_size\_multiplier$ = $10^{-3.5}$ has the least amount of oscillations in the w^t values and the final w^t value is very near to the w^* as obtained from least squares solution. The w^t takes a smooth path towards the minima w^*. Hence, this value of $step\_size\_multiplier$ seems to be the most optimum for our problem.
5. $step\_size\_multiplier$ = 10^{-4} the step_size becomes too small, leading to very small updates in w^t after each iteration. Consequently, w^t is unable to converge within the maximum number of iterations provided (10^4).

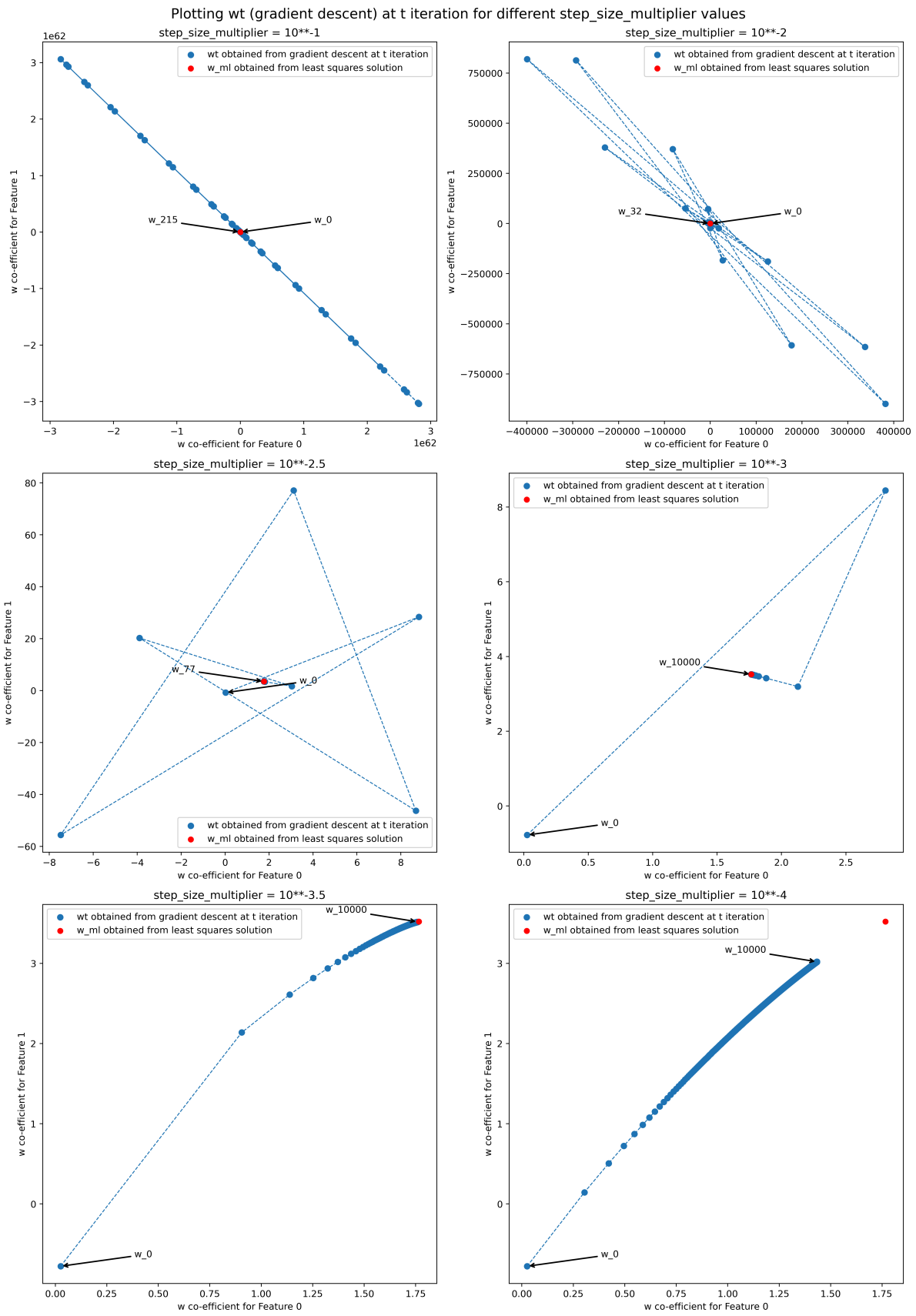

Figure - 1: Plot showing values taken by w^t during gradient descent algorithm for different values of step_size_multiplier. Blue dots are w^t values and red dot is the w^* value as obtained from least squares solution. w_0 is the value of w at the start of the

iteration and w_x is the final value after x iterations (max iterations is set to 10^4). x-axis is w coefficient for $feature_0$ and y-axis is the value of w coefficient for $feature_1$ for each sub-plot.

Hence, we will choose the value of $step\_size\_multiplier = 10^{-3.5}$.

The value of $w*$ as obtained from gradient descent is as follows:

```
w_gd: [[9.8829622 ]
[1.76047697]
[3.5190822 ]]
```

The $w*$ obtained from gradient descent (w_gd) and least squares solution (w_ml) are very close.

MSE on test dataset based on predictions made using $w*$ obtained from gradient descent is **65.983**, which is also only slightly less than the mse of $w*$ obtained from least squares solution. This slight difference can be accounted to the noise in the dataset. While w^*, obtained from the closed-form least squares solution, is the optimal estimator for the training dataset, the presence of noise in the data affects the accuracy of this estimate. As a result, the performance of the model is influenced by this noise, leading to minor differences in MSE between the two approaches. In a different scenario, the MSE for the gradient descent algorithm on the test dataset could have been more than the MSE for least squares solution, because it depends on the datapoints present in the test dataset.

Figure 2 shows the plot of $||w^t - w_{ml}||_2$ as a function of t (iteration). As the gradient descent algorithm proceeds and the number of iterations increase, the value $||w^t||$ gets closer and closer to $||w_ml||$, hence the norm of their difference also decreases with each iteration. After few iterations, the difference in their norms gets very small, reaching almost values as low as 10^-4 as shown in figure-2 log plot. The decrease in the difference is the more during the initial stages, because of two reasons:

1. The step size is largest during the initial stages. This is because the step size is proportional to the inverse of the number of iterations.
2. The weight updates from the gradient function are also large during the initial stages when w^t is farthest from the optimal $w*$.

As w^t gets closer to the optimal w^*, the size of the weight updates from the gradient decrease and also the step_size decreases, giving almost a straight line after the initial iterations.

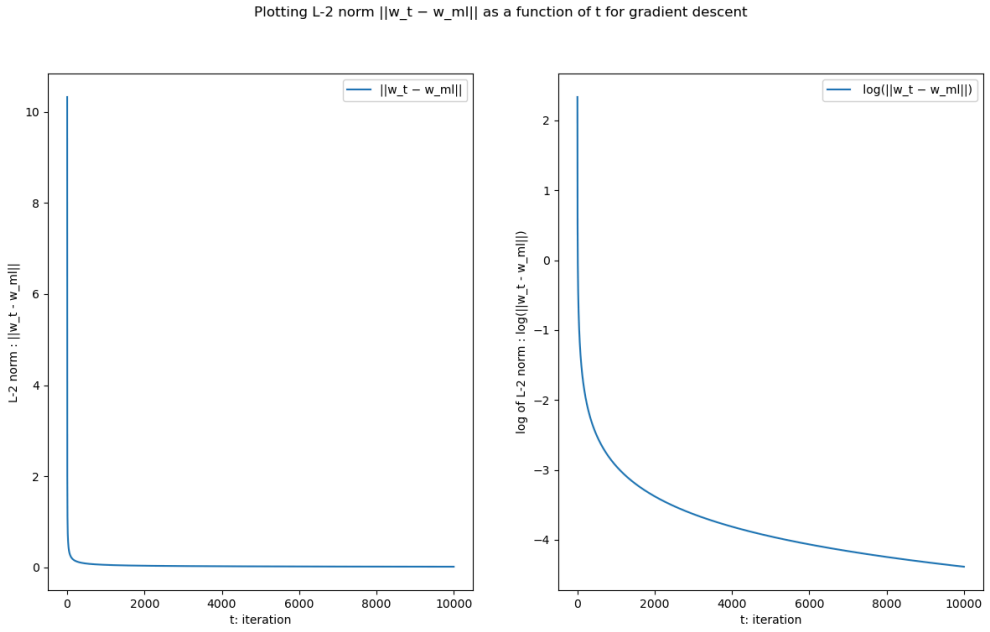

Figure-2: Plot of $||w^t - w_{ml}||_2$ as a function of t (iteration) for gradient descent (left plot is $||w^t - w_{ml}||_2$ vs t; right plot is log $||w^t - w_{ml}||_2$ vs t)

Task-3

Code the stochastic gradient descent algorithm using batch size of 100 and plot $\|w^t - w_{ml}\|_2$ as a function of t. What are your observations?

The stochastic gradient descent (SGD) algorithm was implemented using a batch size of 100 to solve for the weight vector w^* in the linear regression problem. Stochastic gradient descent algorithm is almost similar to the gradient descent algorithm with two major differences:

1. The gradient calculation for updating w^t is done using a batch of datapoints and not the entire dataset. The batch size is pre-determined and the data points are selected randomly for this batch.
2. To obtain the final value of w^*, all the values of w^t from the first iteration onwards are averaged:

   $$w^* = (1/N) \times \Sigma_t^N w^t$$

   where N is the total number of iterations carried out.

The $step\_size\_multiplier$ value was taken to be 10^-2 for stochastic gradient descent as this value gave the best results during experimentation. For a higher $step\_size\_multiplier$ value, the oscillations were very large, and for a lesser $step\_size\_multiplier$ value the algorithm failed to converge within 10^4 iterations.

The value of $w*$ as obtained from stochastic gradient descent is as follows:

```
w_sgd: [[9.89412528]
 [1.74028523]
```

```
[3.50684393]]
```

The $w*$ obtained from gradient descent (w_sgd) and least squares solution (w_ml) are also very close.

MSE on test dataset based on predictions made using $w*$ obtained from stochastic gradient descent is **66.006**, which is also very close to the mse of $w*$ obtained from least squares solution.

Figure-3 shows the values taken by w^t during the stochastic gradient descent algorithm. The trajectory of w^t is not as smooth as it was in gradient descent algorithm. The path is more random during the initial iterations, and it gets more directed as the number of iterations increase. This is because the gradient calculation is not done on the entire dataset, but on a subset of the dataset, thus introducing randomness into the gradient calculation. But, despite using just 100 samples for computation of gradient update, the algorithm is still able to decrease the value of the loss function to a respectable point. The final value of $w*$ after obtained after the final iteration is close to the w^* obtained from least squares solution.

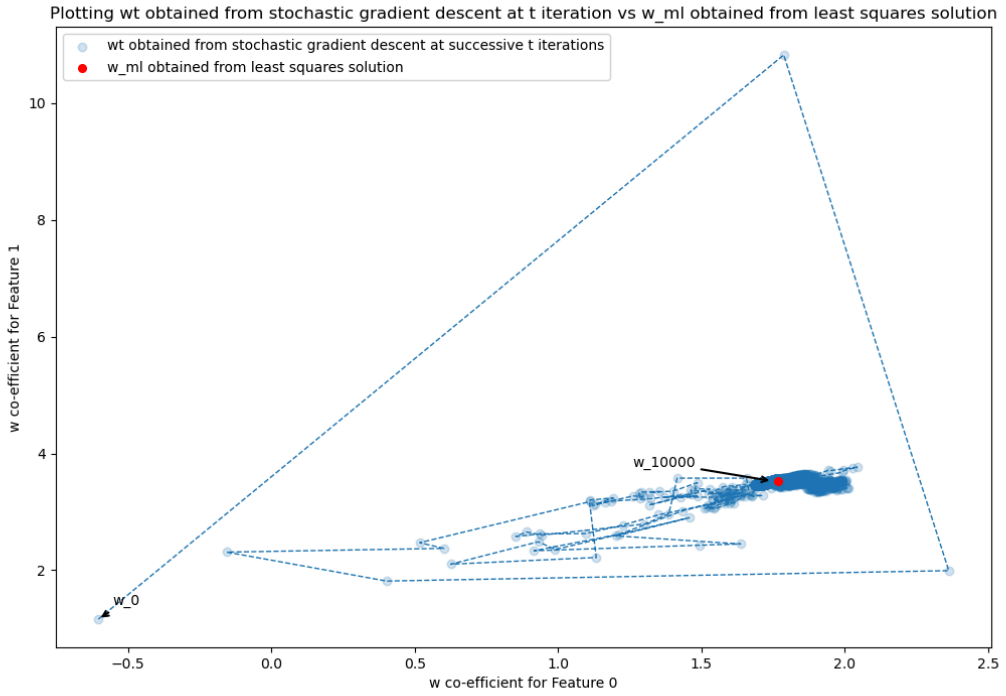

Figure-3: Plot showing values taken by w^t during stochastic gradient descent algorithm. Blue dots are w^t values and red dot is the w^* value as obtained from least squares solution. w_0 is the value of w at the start of the iteration and w_{10000} is the final value after 10000 iterations. x-axis is w coefficient for $feature_0$ and y-axis is the value of w coefficient for $feature_1$ for each sub-plot.

The plot of $||w_t - w_{ml}||_2$ as a function of the number of iterations t for stochastic gradient descent is provided in figure-4.

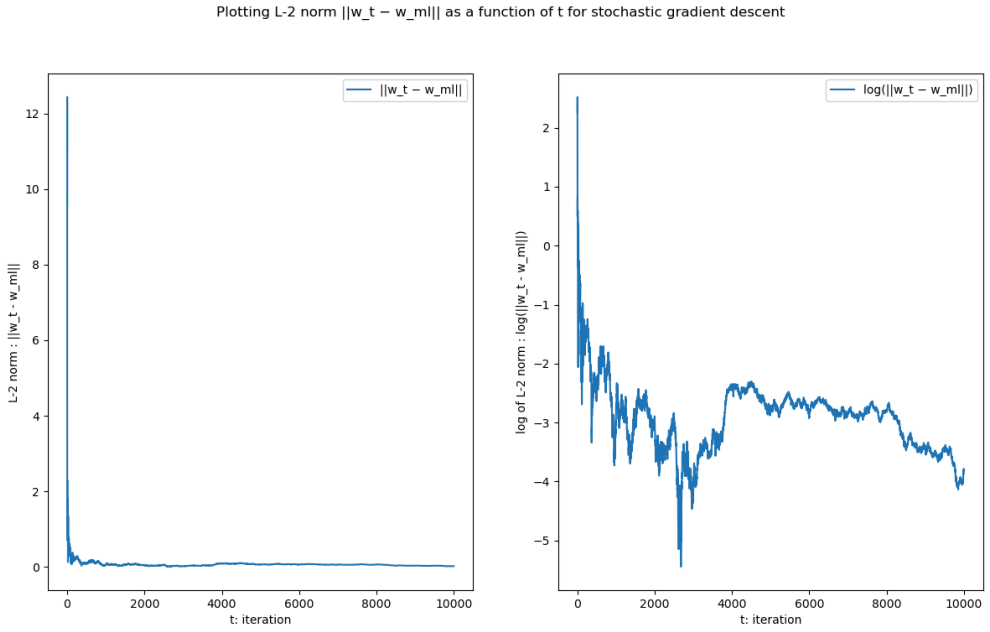

Figure-4: Plot of $||w^t - w_{ml}||_2$ as a function of t (iteration) for stochastic gradient descent (left plot is $||w^t - w_{ml}||_2$ vs t; right plot is log $||w^t - w_{ml}||_2$ vs t)

Figure-4 shows the shows the plot of $||w^t - w_{ml}||_2$ as a function of t (iteration) for SGD. As the stochastic gradient descent algorithm proceeds and the number of iterations increase, the value of $||w^t||$ gets closer and closer to $||w_m l||$, hence the norm of their difference also decreases with each iteration.

The decreasing trend in the value of $||w^t - w_{ml}||_2$ is jagged and uneven unlike in Figure-2. This is because of the nature of the stochastic gradient descent where the updates are made on a randomly selected subset of the dataset. Due to this randomness, the w^t is not always able to follow the actual path of steepest descent, which happens in the case of gradient descent. But the overall direction of descent of w^t from stochastic gradient descent is still in the direction of the minima, causing w^t to get closer to w_{ml}.

Task-4

Code the gradient descent algorithm for ridge regression. Cross-validate for various choices of λ and plot the error in the validation set as a function of λ. For the best λ chosen, obtain wR. Compare the test error (for the test data in the file FMLA1Q1Data test.csv) of wR with wM L. Which is better and why?

Ridge Regression is a type of linear regression that introduces L2 regularization to prevent overfitting by adding a penalty term proportional to the square of the magnitude of the coefficients (weights). When we apply gradient descent to solve Ridge Regression, we optimize the weights by minimizing the regularized cost function, which includes both the error term (residuals) and a regularization term that penalizes large weights.

The objective function for Ridge Regression is:

$$L(w) = ||Xw - Y||^2 + \lambda||w||^2$$

Where:

- X is the matrix of input features of shape n x d, where n is the number of samples and d is the number of features,
- Y is the target vector,
- w is the vector of weights (coefficients)
- λ (or C) is the regularization parameter that controls the strength of regularization.
  - When λ = 0, Ridge Regression reduces to ordinary least squares (OLS).
  - Larger λ values enforce stricter regularization and penalize large weights more.

The gradient of the Ridge Regression cost function with respect to the weight vector w is:

$$\nabla L(w) = 2X^T(Xw - Y) + 2\lambda w$$

In order to find the optimum value of λ, the Ridge Regression model was implemented using gradient descent with various values of λ by doing 5-fold cross-validation. Note that λ has been denoted as C in the source code. The goal to perform cross-validation was to find the best value of the parameter C. The value of C thus obtained was then used to train the final ridge regression model and its performance was evaluated on the test dataset.

Cross-validation is a technique used to evaluate the performance of models by dividing the available data into training and validation subsets. It helps in ensuring that a model generalizes well to unseen data and doesn't overfit to the training dataset. In k-fold cross-validation, the data is split into k equal-sized parts (folds). The model is trained k times, each time leaving out one fold for validation and training on the remaining k-1 folds.

The plot of C (ridge regularization parameter) against the MSE obtained from the 5-fold cross-validation is given in Figure-5. From this plot, we can see that the MSE first decreases till it reaches the lowest value at C = 2.1, after which it keeps on increasing. Thus, C = 2.1 is the best value of C.

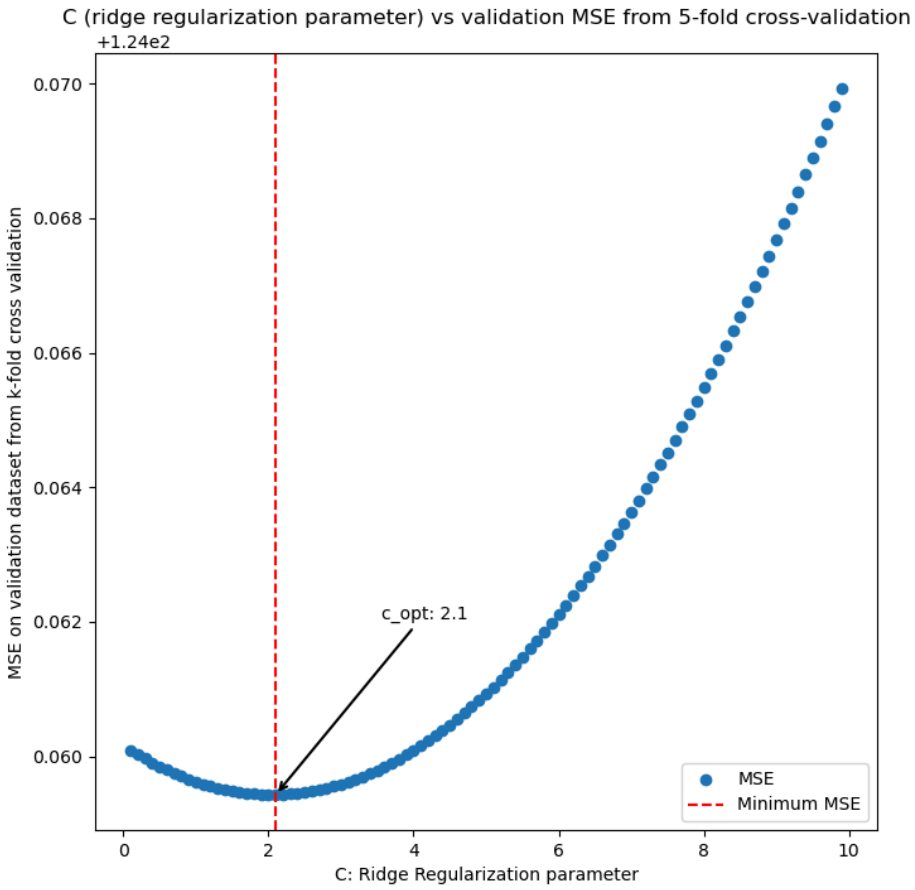

Figure-5: C (ridge regularization parameter) against the MSE obtained from k-fold cross-validation

Using the value of C = 2.1, the Ridge Regression model was trained on the entire training dataset using gradient descent. The weight vector w_r obtained from the final model of ridge regression is:

```
w_ridge = [[9.86252978]
 [1.75611844]
 [3.51244339]]
```

MSE on test dataset based on predictions made using w_r obtained from ridge regression is **65.954**, which is the least MSE that we have obtained till now - although the reduction is marginal. It is 0.8% lower than the MSE of **66.005** obtained from w_{ml} (least squares solution). Hence, the ridge regression model is performing better on the test dataset as compared to the standard least squares solution. This improvement can be explained by considering the bias-variance tradeoff, as follows:

$$MSE = Bias^2 + Variance$$

The above equation tells us that there is always a trade-off between the bias and variance for a model. The least squares solution is obtained from the maximum likelihood estimator, and hence it an unbiased estimator with zero bias.

Ridge regression introduces bias in the model by assuming that the weight coefficients, given by the vector w, come from a normal distribution with zero mean (and non-zero variance). This assumption increases the bias of the model but reduces the variance by a larger amount, causing an overall higher reduction in the error (MSE) of the model. This addition of $||w||^2$ term in the loss function shrinks the coefficients, controlling the magnitude of the weights . The smaller weight coefficients indirectly prevent the model from learning overly large weights that may fit the training data too well (overfitting) but generalize poorly to new data, leading to lower MSE on unseen test data. That is why ridge regression performs better on the test data as compared to the least squares solution.

Task-5

Assume that you would like to perform kernel regression on this dataset. Which Kernel would you choose and why? Code the Kernel regression algorithm and predict for the test data. Argue why/why not the kernel you have chosen is a better kernel than the standard least squares regression.

Kernel regression is a non-parametric technique that extends linear regression to handle nonlinear relationships between the input features and the target variable. Instead of fitting a linear model directly on the input space, kernel regression uses a kernel function to implicitly map the input features into a higher-dimensional space.

Feature Analysis:

- Figure-6 is a plot showing correlation between $feature_0$, $feature_1$ and target variable y respectively. Both the plots have a quadratic shape when plotted against y. This suggests that the features might be quadratically related with y.
- From the contour plot in Figure-7, the elliptical shape of the contours suggests that the target variable is influenced by a nonlinear relationship involving both $feature_0$ and $feature_1$, possibly including quadratic terms like $feature_0^2$, $feature_1^2$, and an interaction term $feature_0 \times feature_1$. The value of y (target variable) is small near the origin and increases as we move away from the origin in the form of elliptical contours, indicating that higher values of y are associated with large absolute values of both features. This contour plot implies a quadratic or nonlinear interaction between the features and the target variable.

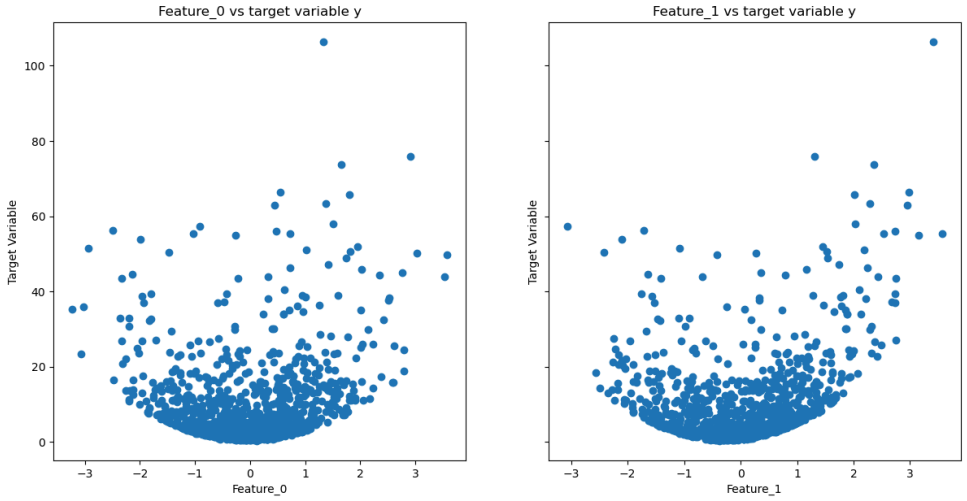

Figure-6: Plot showing correlation between feature_0, feature_1 and target variable.

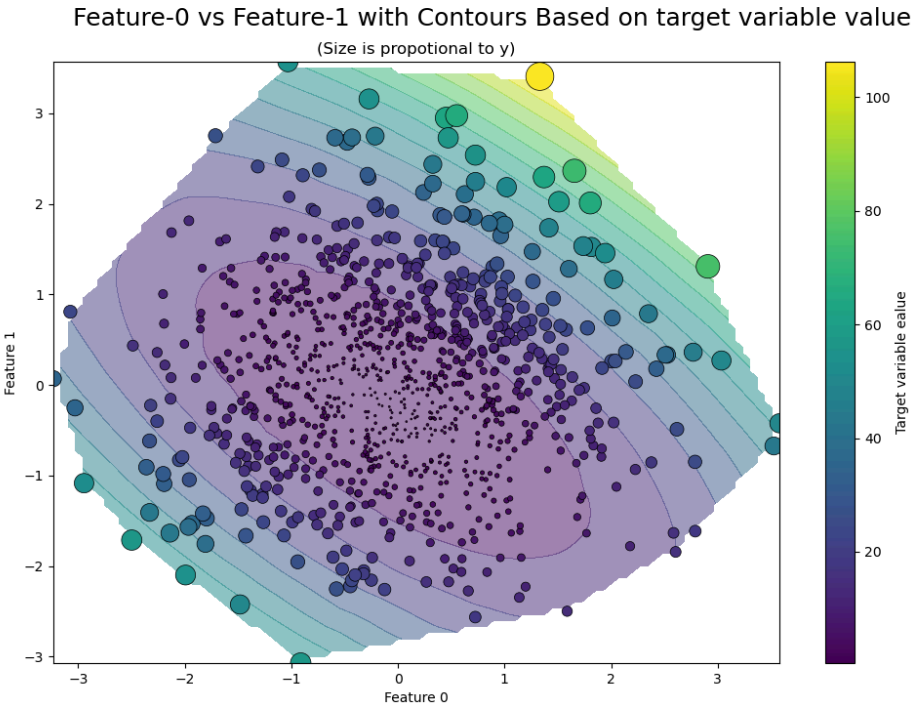

Figure-7: Contour plot showing data points in the training dataset. x-axis and y-axis are $feature_0$ and $feature_1$ respectively. The size of the data point is proportional to the target variable.

Given these observations, a **polynomial kernel of degree 2** seems to be a reasonable option to try. The polynomial kernel can effectively capture the nonlinear relationships (e.g., the product of features), making it a better choice compared to a linear model or standard least squares regression.

The **kernel function** defines the similarity between two points. The polynomial kernel is defined as:

$$K(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}' + 1)^{\deg}$$

Where:

- $\mathbf{x}$, $\mathbf{x}'$ are feature vectors
- $\deg$ is the degree of the polynomial

The
kernel matrix is calculated for all pairs of training data points. This matrix has size $n \times n$ where n is the number of training samples, and each entry in the matrix is computed as:

$$K_{ij} = (\mathbf{x}_i^\top \mathbf{x}_j + 1)^{\deg}$$

Once the kernel matrix
K is computed, we solve for the **alpha coefficients** by using the following equation:

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

This step involves finding the pseudo-inverse of the kernel matrix K and then multiplying it by the target values Y to obtain α.

After the alpha coefficients are learned, predictions for new test points are made using the kernel function. Specifically, the prediction for a test point $\mathbf{x}_{\text{test}}$ is:

$$\hat{y}_{\text{test}} = \Sigma_i \alpha_i K(\mathbf{x}_i, \mathbf{x}_{\text{test}})$$

MSE on test dataset based on predictions made using polynomial kernel regression of degree 2 is **0.012**, which is a very significant improvement from simple linear regression models. This is almost a 100% reduction in the MSE. The low MSE value says for itself that the polynomial kernel regression model is the one more suited for this dataset as compared to the linear regression model.

The polynomial kernel regression is better because:

1. **Capturing Nonlinear Relationships**:
   The polynomial kernel can capture
   nonlinear interactions between features, which are evident in the dataset. Least squares regression assumes a linear relationship, which cannot capture such interactions.
2. **Better Fit**:
   By mapping the input features into a higher-dimensional space, the polynomial kernel allows the model to find a better fit to the data. This results in a lower
   Mean Squared Error (MSE) compared to standard least squares regression, indicating better generalisation.

Thus, using kernel regression with a polynomial kernel significantly improves the model's ability to capture the underlying relationships in the data, leading to better predictive performance.

Source Codes

▼ **Source code for the function to calculate w* from least squares solution is as follows:**

```
import numpy as np

def least_squares_soln(X, y):
    """
    This function calculates the optimal weight vector `w*` for linear regression
    using the least squares.

    Parameters:
    -----------
    X : numpy.ndarray
        A 2D array of shape (n_samples, n_features) representing the input data,
        where each row corresponds to a data point and each column corresponds to a feature.
    y : numpy.ndarray
        A 1D array of shape (n_samples,) containing the target values for the input data.

    Returns:
    --------
    numpy.ndarray
        The optimal weight vector `w*` that minimizes the least squares error, shape
        will be (n_features,).

    Notes:
    ------
    The least squares solution is obtained by solving the normal equation:
        w* = (X^T X)^(-1) X^T y

    """

    lhs = np.linalg.pinv(X.T @ X)
    rhs = X.T @ y
    return lhs @ rhs
```

▼ **Source code for the function to calculate mse is as follows:**

```
import numpy as np

def mse(true, preds):
    """
    Calculate the Mean Squared Error (MSE) between the actual values and the predicted values.

    Parameters:
    -----------
    true : numpy.ndarray
        The actual values of the target variable (ground truth). It is a 1D array.
    preds : numpy.ndarray
        The predicted values of the target variable. It is a 1D array.
```

```
    Returns:
    --------
    mean_squared_error : float
        The mean squared error, which is the average of the squared differences between the actual
        and predicted values.
    """
    return np.mean(np.power(true - preds, 2), axis=0)
```

▼ **The source code for gradient descent function is as follows:**

```
import numpy as np

def gradient_descent(X, y, max_iter = 10**4, step_size_multiplier = 0.01, intermediate_w = False):
    """
    Perform gradient descent to solve linear regression.

    This function implements the gradient descent algorithm to minimize the squared loss
    for a linear regression problem. It iteratively updates the weight vector `w` using
    the gradient of the loss function with respect to `w`, until convergence or the maximum
    number of iterations is reached.

    Parameters:
    -----------
    X : numpy.ndarray
        A 2D array of shape (n_samples, n_features) representing the input data, where each row
        corresponds to a data point and each column corresponds to a feature.
    y : numpy.ndarray
        A 1D array of shape (n_samples,) containing the target values for the input data.
    max_iter : int, optional
        Maximum number of iterations for the gradient descent algorithm. Default is 10^4.
    step_size_multiplier : float, optional
        Multiplier for the step size, which controls the rate of convergence. Default is 0.01.
    intermediate_w : bool, optional
        If True, returns a list of the weight vectors `w` at each iteration. Default is False.

    Returns:
    --------
    w : numpy.ndarray
        The optimized weight vector `w` of shape (n_features, 1) obtained after running
        gradient descent.
    w_array : numpy.ndarray, optional
        If `intermediate_w` is True, returns a 2D array containing the weight vectors `w`
        at each iteration. Shape is (iterations, n_features).

    Notes:
    ------
    - The gradient of the loss function is computed as:
        ∇L(w) = 2 * X^T (X*w - y)
    - Gradient descent stops when the norm of the gradient (∇L(w)) becomes smaller than
      10^(-6), indicating convergence. If convergence is reached, the number of iterations
      is printed.
```

```
    - The step size decreases as the number of iterations increases, with a step size at
      iteration `i` being `1 / (i + 1)`.

    """

    def del_fx(X, y, w):
        """
        Returns the value of the derivative of the loss function for a particular w
        """
        return 2*(X.T @ X @ w) - 2*(X.T @ y)

    iters = 0
    step = 1
    np.random.seed(23)
    w = np.random.randn(X.T.shape[0], 1)
    w_array = [w]

    while (iters < max_iter) and np.linalg.norm(del_fx(X, y, w)) > 10**-6:
        w = w - step * step_size_multiplier * del_fx(X, y, w)
        w_array.append(w)
        iters += 1
        step = 1/(iters + 1)

    if iters < max_iter and not np.isnan(np.linalg.norm(w)):
        print(f"Gradient descent converged with {iters} iterations")

    if intermediate_w:
        return w, np.array(w_array)

    return w
```

▼ **The source code of stochastic gradient descent is as follows:**

```
import numpy as np

def stochastic_gd(X, y, batch = 100, max_iter = 10**4, step_size_multiplier = 0.01, intermediate_w
= False):
    """
    Perform stochastic gradient descent (SGD) for linear regression.

    This function implements the stochastic gradient descent algorithm, where a subset (batch)
    of the data is randomly selected at each iteration to compute the gradient, rather than
    using the entire dataset. This can lead to faster convergence for large datasets.

    Parameters:
    -----------
    X : numpy.ndarray
        A 2D array of shape (n_samples, n_features) representing the input data, where each row
        corresponds to a data point and each column corresponds to a feature.
    y : numpy.ndarray
        A 1D array of shape (n_samples,) containing the target values for the input data.
```

```
    batch : int, optional
        The number of data points to be sampled at each iteration to compute the gradient. Default is
100.
    max_iter : int, optional
        Maximum number of iterations for the stochastic gradient descent algorithm. Default is 10^4.
    step_size_multiplier : float, optional
        Multiplier for the step size, which controls the rate of convergence. Default is 0.01.
    intermediate_w : bool, optional
        If True, returns a 2D array of the weight vectors `w` at each iteration. Default is False.

    Returns:
    --------
    w : numpy.ndarray
        The final weight vector `w`, averaged over all iterations. Shape is (n_features, 1).
    w_array : numpy.ndarray, optional
        If `intermediate_w` is True, returns a 2D array containing the weight vectors `w` at each iterati
on.
        Shape is (iterations, n_features).

    Notes:
    ------
    - At each iteration, a random batch of data points is sampled to compute the gradient.
    - The gradient of the loss function is computed as:
        ∇L(w) = 2 * X_batch^T (X_batch*w - y_batch)
    - The algorithm will stop after a maximum number of iterations or when convergence is manuall
y
      checked using the returned weight vectors.
    - The step size decreases as the number of iterations increases, with the step size at
      iteration `i` being `1 / (i + 1)`.
    """

    def del_fx(X, y, w):
        """
        Returns the value of the derivative of the loss function for a particular w
        """
        return 2*((X.T @ X) @ w) - 2*(X.T @ y)

    iters = 0
    step = 1
    np.random.seed(69)
    w = np.random.randn(X.T.shape[0], 1)
    w_array = [w]

    while (iters < max_iter):
        points = np.random.randint(0, len(X), batch)   ## Selecting random samples from the dataset
        X_ = X[points]
        y_ = y[points]
        w = w - step * step_size_multiplier * del_fx(X_, y_, w)
        w_array.append(w)
        iters += 1
        step = 1/(iters + 1)
```

```
    w_array = np.array(w_array)

    if intermediate_w:
        return np.mean(w_array, axis = 0), w_array

    return np.mean(w_array, axis = 0)
```

▼ **The source code for ridge regression is as follows:**

```
import numpy as np

def gradient_descent_ridge(X, y, C = 1, max_iter = 10**4, step_size_multiplier = 0.01, intermediate_
w = False):
    """
    Perform gradient descent for ridge regression (L2-regularized linear regression).

    This function implements the gradient descent algorithm to solve ridge regression,
    a form of linear regression that includes an L2 regularization term to penalize large
    weight values. It iteratively updates the weight vector `w` using the gradient of the
    ridge regression loss function until convergence or the maximum number of iterations
    is reached.

    Parameters:
    -----------
    X : numpy.ndarray
        A 2D array of shape (n_samples, n_features) representing the input data,
        where each row corresponds to a data point and each column corresponds to a feature.
    y : numpy.ndarray
        A 1D array of shape (n_samples,) containing target values for the input data.
    C : float, optional
        The regularization constant (equivalent to lambda in ridge regression). This controls
        the strength of the L2 regularization, with larger values enforcing greater regularization.
        Default is 1.
    max_iter : int, optional
        Maximum number of iterations for the gradient descent algorithm. Default is 10^4.
    step_size_multiplier : float, optional
        Multiplier for the step size, which controls the rate of convergence. Default is 0.01.
    intermediate_w : bool, optional
        If True, returns a 2D array of the weight vectors `w` at each iteration. Default is False.

    Returns:
    --------
    w : numpy.ndarray
        The final weight vector `w` obtained after running gradient descent. Shape is (n_features, 1).
    w_array : numpy.ndarray, optional
        If `intermediate_w` is True, returns a 2D array containing the weight vectors `w`
        at each iteration. Shape is (iterations, n_features).

    Notes:
    ------
```

```
- Ridge regression adds an L2 regularization term to the standard linear regression
  loss function to prevent overfitting. The loss function for ridge regression is:
    L(w) = ||Xw - y||^2 + C * ||w||^2
  where `C` is the regularization constant.
- The gradient of the ridge regression loss function is computed as:
    ∇L(w) = 2 * X^T (Xw - y) + 2 * C * w
- The gradient descent stops when the norm of the gradient becomes smaller than 10^(-6),
  indicating convergence.
- The step size decreases as the number of iterations increases, with the step size at
  iteration `i` being `1 / (i + 1)`.
"""

def del_fx(X, y, w, C):
    """
    Returns the value of the derivative of the loss function for a particular w
    """
    return 2*((X.T @ X) @ w) - 2*(X.T @ y) + 2*C*w

iters = 0
step = 1
np.random.seed(23)
w = np.random.randn(X.T.shape[0], 1)
w_array = [w]

while (iters < max_iter) and np.linalg.norm(del_fx(X, y, w, C)) > 10**-6:
    w = w - step * step_size_multiplier * del_fx(X, y, w, C)
    w_array.append(w)
    iters += 1
    step = 1/(iters + 1)

# if iters < max_iter and not np.isnan(np.linalg.norm(w)):
#     print(f"Ridge gradient descent converged with {iters} iterations")

if intermediate_w:
    return w, np.array(w_array)

return w
```

▼ **The source code of cross-validation function is as follows:**

```
from gradient_descent import gradient_descent
from stochastic_gd import stochastic_gd
from gradient_descent_ridge import gradient_descent_ridge
from mean_squared_error import mse
import numpy as np

def cross_validation(model, X, y, k, max_iter = 10**4, step_size_multiplier = 0.01, C = None, batch
= None):
    """
    Perform k-fold cross-validation on the given dataset using the specified model.
```

```
    This function divides the dataset into `k` folds and trains the model `k` times, each time
    using a different fold as the validation set and the remaining `k-1` folds as the training set.
    It returns the average mean-squared error (MSE) across all folds, as well as the list of MSE
    values for each fold.

    Parameters:
    -----------
    model : str
        Specifies the model to use for training. Should be one of:
        - "gd" : Gradient Descent for linear regression
        - "sgd" : Stochastic Gradient Descent for linear regression
        - "ridge" : Gradient Descent for Ridge regression
    X : numpy.ndarray
        A 2D array of shape (n_samples, n_features) representing the input data, where each row
        corresponds to a data point and each column corresponds to a feature.
    y : numpy.ndarray
        A 1D array of shape (n_samples,) containing the target values for the input data.
    k : int
        The number of folds for k-fold cross-validation.
    max_iter : int, optional
        Maximum number of iterations for the model's optimization algorithm. Default is 10^4.
    step_size_multiplier : float, optional
        Multiplier for the step size, which controls the rate of convergence. Default is 0.01.
    C : float, optional
        Regularization constant for Ridge regression. This parameter is required if `model="ridge"`.
    batch : int, optional
        Batch size for Stochastic Gradient Descent (SGD). This parameter is required if `model="sg
d"`.

    Returns:
    --------
    mean_mse : float
        The average mean-squared error (MSE) across all folds.
    mse_list : list of floats
        A list containing the MSE for each fold in the k-fold cross-validation.

    Notes:
    ------
    - If `model` is "ridge", the `C` parameter (regularization coefficient) must be specified.
    - If `model` is "sgd", the `batch` parameter (batch size) must be specified.
    - The function uses the `gradient_descent`, `stochastic_gd`, and `gradient_descent_ridge` functi
ons
      from imported modules for the respective models.

    """

    model_dict = {"gd" : gradient_descent,
    "sgd" : stochastic_gd,
    "ridge": gradient_descent_ridge}

    if model not in model_dict.keys():
```

```
        print("model parameter not correctly passed. Please specify a model from (gd) gradient desc
ent, (sgd) stochastic gradient descent, and (ridge) ridge regression.")
        return
    elif model == "ridge" and C is None:
        print("also pass the C parameter for the ridge model")
        return
    elif model == "sgd" and batch is None:
        print("also pass the batch parameter for the sgd model")
        return
    else:
        model_fun = model_dict[model]

    ## Setting a random seed
    np.random.seed(23)
    sample_perm = np.random.permutation(np.arange(0, len(X)))

    l = len(X)//k  ## Specifies the number of sample points in each fold

    X_batches = []  ## list will contain training batches
    y_batches = []
    val_batches = []  ## list will contain validation batches
    y_val_batches = []

    for i in range(k):
        sample_fold = sample_perm[i*l:(i+1)*l]
        val_batches.append(X[sample_fold])
        y_val_batches.append(y[sample_fold])
        X_batches.append(X[np.setdiff1d(sample_perm, sample_fold)])
        y_batches.append(y[np.setdiff1d(sample_perm, sample_fold)])

    mse_list = []

    for i in range(k):
        if model == "ridge":
            w = model_fun(X = X_batches[i], y = y_batches[i], C = C, max_iter = max_iter, step_size_m
ultiplier = step_size_multiplier)
        elif model == "sgd":
            w = model_fun(X = X_batches[i], y = y_batches[i], batch = batch, max_iter = max_iter, step
_size_multiplier = step_size_multiplier)
        else:
            w = model_fun(X = X_batches[i], y = y_batches[i], max_iter = max_iter, step_size_multiplier
= step_size_multiplier)

        preds = np.matmul(val_batches[i], w)
        mse_list.append(mse(y_val_batches[i], preds))

    return np.mean(mse_list), mse_list
```

Note: The cross-validation function has been written so that it can accept any of the variants of linear regression (gradient descent, stochastic gradient descent or ridge gradient descent) for performing cross-validation.

▼ **The source code for implementation of kernel regression is as follows:**

```
import numpy as np

# Polynomial kernel regression function
def kernel_regression_poly(X, y, deg=2):
    """
    Performs polynomial kernel regression and returns the alpha coefficients
    and the kernel matrix.
    """
    # Compute the polynomial kernel matrix
    kernel_matrix = np.power(X @ X.T + 1, deg)
    # Solve for alpha using the pseudo-inverse of the kernel matrix
    alpha = np.matmul(np.linalg.pinv(kernel_matrix), y)

    return alpha, kernel_matrix

# Function to make predictions using the learned alpha coefficients
def kernel_pred_poly(alpha, X, X_test, deg=2):
    """
    Predicts target values for test data using the polynomial kernel and learned alpha coefficients.
    """
    # Compute the kernel matrix between training data X and test data X_test
    pred_kernel_matrix = np.power(X @ X_test.T + 1, deg)
    # Compute the predictions by applying the learned alpha values
    preds = np.sum(alpha * pred_kernel_matrix, axis=0).T

    return preds

# Using the kernel regression on training data
deg = 2
alpha, kernel_matrix = kernel_regression_poly(X_train, y_train, deg)

# Making predictions on test data
preds = kernel_pred_poly(alpha, X_train, X_test, deg)

# Calculate the Mean Squared Error (MSE) on test data
mse_kernel = mse(y_test.ravel(), preds.ravel())
print(f"MSE for kernel regression using polynomial kernel (degree = {deg}): {mse_kernel}")
```