Week 3 - Lasso & Ridge Regression
Machine Learning

David Esteso Calatrava

October 10, 2024

School of Computer Scinece and Statics

3D Plot of Data

(a) Scatter plot analysis

In this section, we visualize the data using a 3D scatter plot, where the first feature is represented on the x-axis, the second feature on the y-axis, and the target variable on the z-axis.

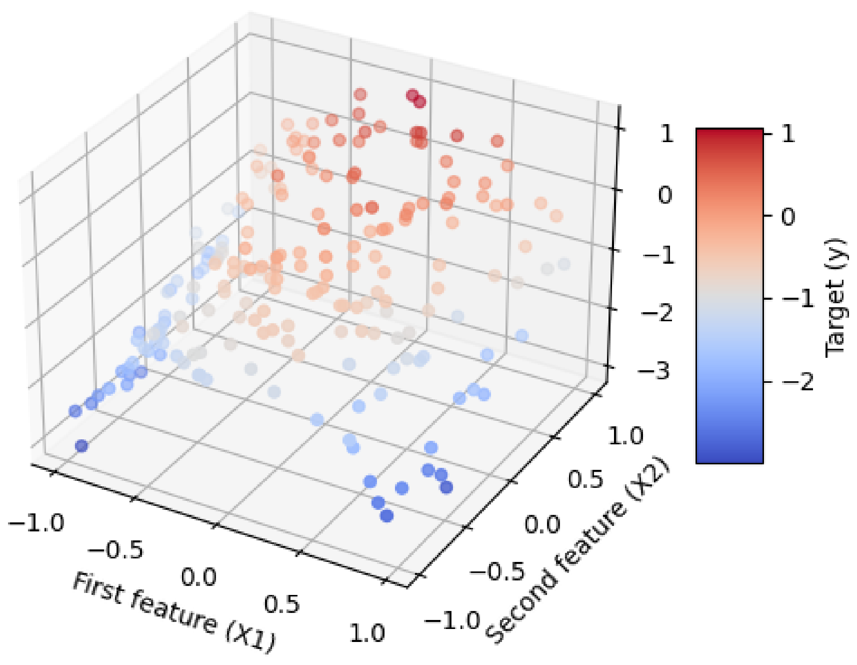

Figure 1: Dataset with id:7–14-7

To determine whether the data forms a plane or a curve, let's examine each axis.

Starting with the **X1 axis**, we observe that the target values initially increase and then decrease, indicating a **curved shape** that brings to mind the $-X^2$ equation (which is a concave curve). This suggests that in the polynomial models we will train, the polynomial feature $X1^2$ is likely to have a significant negative weight.

Next, when we analyze the **X2 axis**, we see that the target values consistently increase as $X2$ increases. This relationship appears to be more linear, indicating that $X2$ will likely have a positive weight and will dominate the model, requiring no polynomial features.

In conclusion, the data **resembles a curve** more than a plane. We can describe it as a curve in the axis of feature X_1 that ascends along the axis of feature X_2.

(b) Lasso model with polynomial features

Before discussing the details of this task, it is important to note that we will focus on **regressors**, unlike previous tasks where we used classifiers. The key difference between the two is that regressors predict **continuous values**, while classifiers assign data points to discrete categories.

Here, we expand the dataset by adding polynomial features up to power 5 and train Lasso regression models for different values of the penalty parameter C.

That is what this Python code does:

```
# Create polynomial features
poly = PolynomialFeatures(degree=5)
X_poly = poly.fit_transform(X)

for C in C_values:
    alpha = 1 / (2 * C)
    model = Lasso(alpha=alpha)
    model.fit(X_poly, y)
```

The ‘PolynomialFeatures‘ class transforms the original features into higher-degree interaction terms. For each regularization strength C, it computes the corresponding Lasso regression parameter α and fits the Lasso model using the polynomial features.

Before examining how changes in C have affected the model, we must understand its effect on Lasso regression. Lasso regression employs a linear model with a **mean square cost function and an L1 penalty**. The weight parameter for the L1 penalty is C (set to $1/(2C)$ in sklearn).

This means that as C increases, the penalty for large coefficients decreases, allowing more features to be included in the model. Conversely, when C is smaller, the penalty increases, leading to more coefficients being set to zero. This is exactly what our tests illustrate.

The table below summarizes the intercept, coefficients for each feature, model score, and the number of non-zero coefficients for each model.

C	Intercept	Non-Zero Coefficients	Model Score	Key Coefficients
1	-0.7593	0	0.0000	None
10	-0.2244	2	0.8874	X_2: 0.81, X_1^2: -1.43
100	-0.0476	2	0.9373	X_2: 0.95, X_1^2: -1.93
1000	-0.0368	11	0.9392	X_2: 0.93, X_1^2: -1.94

Table 1: Lasso Regression Results for Various Values of C

With $C = 1$, the model seems to be **under-trained**, failing to identify any patterns in the data, as indicated by a **model score of 0.0000** and **0 non-zero coefficients**. This lack of learning suggests that the model is **overly regularized**, which prevents it from capturing meaningful relationships between the features and the target.

When we increase C to 10, the model starts to show improvement. We observe **2 non-zero coefficients** with a **model score of 0.8874**. This indicates that the model is beginning to align with our previous findings, where X_2 and X_1^2 are identified as significant features with coefficients **0.81** and **-1.43**, respectively.

Further increasing C to 100 leads to an even better model performance, with a **model score of 0.9373** and the same **2 non-zero coefficients** as before. The values for X_2 and X_1^2 improve to **0.95** and **-1.93**, respectively, indicating that the model has captured more details in the relationship between these features and the target.

However, when we set $C = 1000$, we see a substantial increase in the number of non-zero coefficients, from **2 to 11**. This increase does not correspond to an improved model score, which raises concerns about **overfitting**. Additionally, the **9 coefficients** that are not shown in the table have magnitudes less than **0.1**. This suggests that the model may be capturing **noise** or features that other models have not been able to identify.

In terms of the intercept, we see that it **decreases** as we increase C, in contrast to the coefficients, which tend to be larger. This behavior suggests that the model is relying more on the features to explain the target variable, reducing the need for a large intercept to achieve accurate predictions.

(c) Predictions on feature grid

In this section, we generate predictions for the target variable on a grid of feature values that extends beyond the original dataset. The predictions are visualized as a surface, while the training data is represented as scatter points. We made the decision to split the predictions into two separate plots to distinguishd better the behavior of each model as C changes.

```
padding = 2
grid_x1 = np.linspace(x1_min - padding, x1_max + padding, 100)
grid_x2 = np.linspace(x2_min - padding, x2_max + padding, 100)

fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')

# 3D scatter plot of training data
ax.scatter(X[:, 0], X[:, 1], y, c=colors, marker='o', label='Training␣Data', alpha=0.5)

predictions = model.predict(Xtest_poly)
predictions = predictions.reshape(100, 100)

ax.plot_surface(grid_x1.reshape(100, 1), grid_x2.reshape(1, 100), predictions,
                alpha=0.3, color=color, label=f'Predictions␣(C={C})')
```

In this code, two grids **grid_x1** and **grid_x2** are generated to span the feature space with some padding. A 3D scatter plot of the training data is created, followed by a surface plot showing the model's predictions.

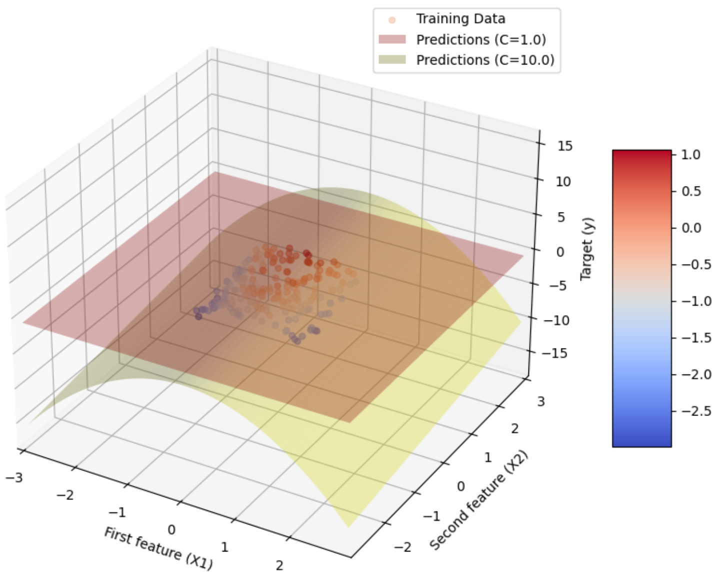

Figure 2: Lasso predictions with C=1 and C=10

In the two 3D plots, we observe how the model's predictions change as the value of the regularization parameter C is varied. Figure 2 shows the predictions for $C = 1$ and $C = 10$, represented by the red and yellow surfaces, respectively. Notably, in the previous section, we saw that $C = 1$ corresponds to a model without features and with a score of 0, resulting in a

flat plane. As C increases from 1 to 10, this plane begins to curve more, indicating a **better fit** to the data.

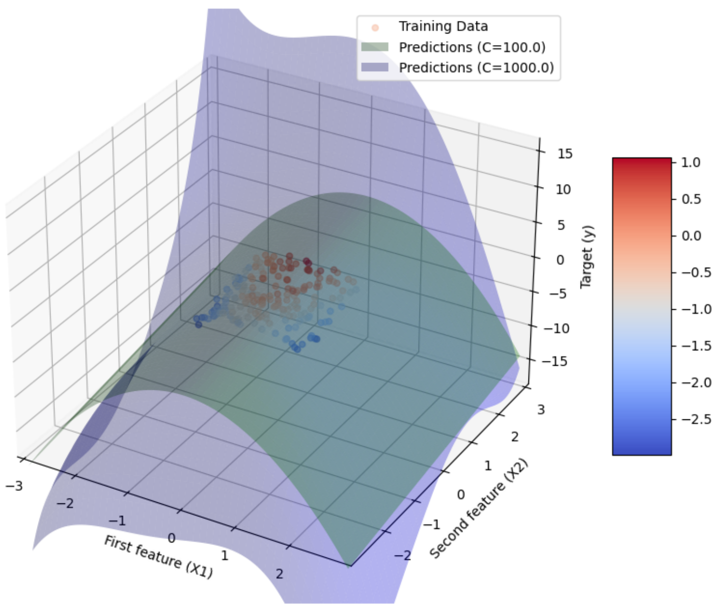

Figure 3: Lasso predictions with C=100 and C=1000

Figure 3 illustrates the predictions for $C = 100$ and $C = 1000$, with the green and blue surfaces. Similar to the first plot, we observe that increasing C makes the predictions **fit the data more closely** when raising C to 100. However, when $C = 1000$, we notice that in areas without any training data points, the blue surface shows more **extreme predictions**. This unusual behavior seems to be influenced by the feature X_2. As X_2 decreases to values outside the training range, the predictions drop sharply, while increasing X_2 leads to a significant rise in predicted values. This pattern likely results from the **new model's coefficients**, which don't accurately reflect the actual data patterns; instead, they appear to have captured relationships that are either too complex or, in some cases, just noise. Although these coefficients have relatively small values, we see that the predictions differ noticeably from those of the other models. To better understand whether the model is truly capturing noise and how well it scales, we'll explore these issues using cross-validation in the following sections.

(d) Overfitting and underfitting

Underfitting happens when a model is too simple to pick up on the actual patterns in the data. As a result, it doesn't perform well on either the training set or the test set. You can often spot underfitting when the model shows a high error rate, even with the training data. For instance, if you try to use a linear regression model to fit a dataset that actually has a quadratic relationship, you're likely to run into underfitting. The model just isn't complex enough to capture the intricacies of the data, leading to big errors in its predictions.

On the flip side, we have **overfitting**. This occurs when a model learns not only the true signals in the data but also the random noise. This can result in the model performing really well on the training set but struggling when it encounters new, unseen data. A classic example of overfitting is a decision tree that's too deep. It might perfectly fit the training data by capturing every little detail and noise, but this makes it poor at generalizing to new data.

What we have observed in sections *(b)* and *(c)* **illustrates these two situations** (The analysis conducted in the cross-validation section allows us to affirm that, since the model with $C = 1000$ does not scale well, we have likely captured irrelevant patterns). In section *(b)*, we find that a low C leads to coefficients being set to zero, indicating a lack of learning from the data. As C increases, the situation improves; however, at $C = 1000$, we observe not only a significant increase in the magnitude of the coefficients but also a higher number of non-zero coefficients. Although these coefficients are relatively small, they seem to represent random fluctuations in the training data rather than reflecting the underlying data structure.

Section *(c)* reveals that the pattern corresponding to the lowest C value resembles a flat plane, diverging considerably from the actual shape of the data. In contrast, at $C = 1000$, the overfitted model demonstrates extreme behavior when predicting values for X_1 and X_2 beyond the range encountered during training, that, as it has been said, are caused by the number of non-zero coefficients.

Both underfitting and overfitting are critical issues in model training that highlight the importance of tuning hyperparameters like C and employing techniques such as cross-validation. These methods help identify the model that strikes the right balance between capturing the main features in the data without fitting too closely to the noise. The ideal model lies between those that underfit and those that overfit, making it essential to experiment with different values and approaches to achieve optimal performance.

(e) Ridge regression comparison

Repeating parts (b) and (c) for Ridge regression models, we observe how the parameters and predictions behave when using an L2 penalty instead of L1. This section compares the effects of changing C in both Lasso and Ridge regression.

C	Intercept	Non-Zero Coefficients	Model Score	Key Coefficients
0.01	-0.4874	20	0.7121	X_2: 0.0340, X_1^2: 0.3826
0.1	-0.2170	20	0.9204	X_2: 0.0425, X_1^2: 0.6867
1.0	-0.0944	20	0.9378	X_2: -0.0005, X_1^2: 0.8509
10.0	-0.0335	20	0.9403	X_2: -0.0874, X_1^2: 0.9447

Table 2: Ridge Regression Results for Various Values of C

Both Ridge and Lasso regression methods demonstrate **similar trends when incrementing the regularization parameter** C. Starting from a very low value of C, both methods can result in underfitted models. As C increases, the model performance generally improves, although we will encounter overfitting at high C values.

However, notable differences appear as we analyze the magnitude of C needed for optimal performance. Lasso regression generally exhibits a higher optimal value for C compared to Ridge regression, suggesting that Lasso is more sensitive to regularization, resulting in more significant coefficient decrease as C increases. Conversely, Ridge regression achieves its optimal model score with a lower value of C, indicating that it may require less regularization to perform better, while Lasso benefits from more substantial penalization.

In regression analysis, Lasso employs an L1 penalty, while Ridge utilizes an L2 penalty, expressed mathematically as:

$$\text{Cost}_{\text{Lasso}} = \text{RSS} + \lambda \sum_{j=1}^{p} |\beta_j|$$

$$\text{Cost}_{\text{Ridge}} = \text{RSS} + \lambda \sum_{j=1}^{p} \beta_j^2$$

Moreover, the way coefficients behave shows more differences between the two methods. Lasso is good at driving many coefficients to zero, which helps in selecting important features. At higher values of C, Lasso keeps **only the most relevant features** to the model, making it simpler. On the other hand, Ridge regression keeps all coefficients non-zero, even when it is least regularized. This means Ridge is less aggressive in reducing coefficients. In fact, Lasso tends to assign higher absolute values to the coefficients of the features it trusts the most, while Ridge **spreads its confidence across all features**, resulting in smaller coefficients.

The term $|\beta_j|$ denotes the absolute value of the coefficient β_j, representing the L1 penalty that drives many coefficients to zero, facilitating feature selection by eliminating irrelevant predictors from the model. In contrast, the term β_j^2 indicates the square of the coefficient β_j and corresponds to the L2 penalty, which reduces all coefficients towards zero without eliminating any, thus retaining all features in the model while controlling their influence.

Now, we will apply the same method as in section (c) but with Ridge regression. We will generate predictions for the target variable on a grid of feature values that extends beyond the original dataset. The predictions will be visualized as a surface, while the training data will be represented as scatter points. As before, we will split the predictions into two separate plots to better distinguish the behavior of each model as the regularization parameter C changes.

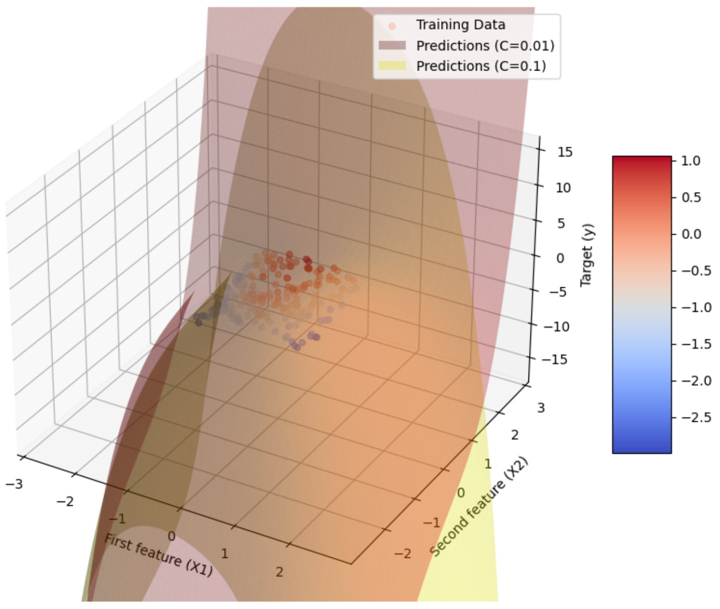

Figure 4: Ridge predictions with C=0.001 and C=0.01

The results from the Ridge regression are quite ihnteresting. Notably, the predictions from the model suspected of overfitting with Lasso resemble those from Ridge when C is low. This is understandable since Lasso can be overly simplistic by setting many coefficients to zero,

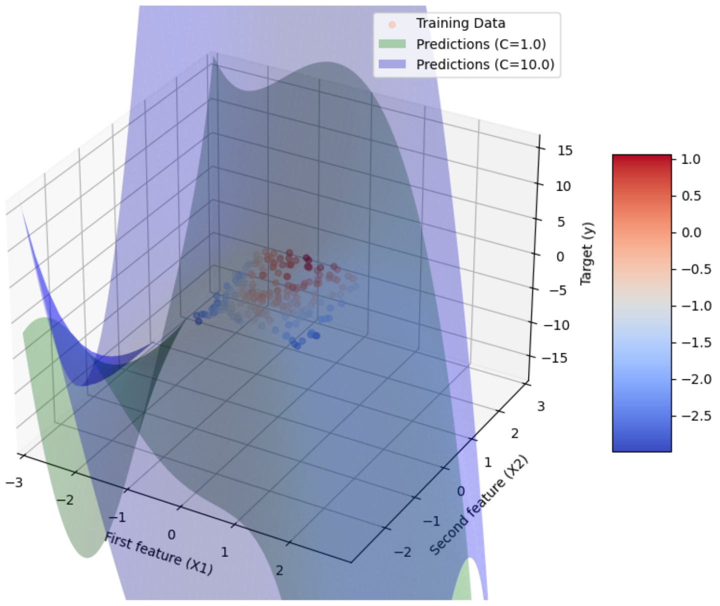

Figure 5: Ridge predictions with C=1 and C=10

while Ridge retains all coefficients, allowing it to adapt better in regions outside the training predictions.

At first glance, none of the Ridge models appear to be very underfitted, unlike the low C value we observed with Lasso. As we increase C, the planes fit the shape of the data more closely. However, an interesting change occurs at $C = 1$; instead of predicting lower values when feature x_1 decreases, the model predicts higher values. This change happens because some coefficients related to x_1 switch from positive to negative. For example, the coefficient for $x_1^2 x_2$ went from 0.065 to -0.143.

In conclusion, while both Ridge and Lasso regression exhibit similar patterns when varying C, they differ significantly in their regularization strengths and the impact on coefficients. Lasso's capacity to eliminate coefficients results in simpler models, whereas Ridge retains all predictors, which may be beneficial in certain contexts.

Cross-validation to select C

(a) Cross-validation error

Using **5-fold cross-validation**, we plot the mean and standard deviation of the prediction error as a function of C. In this method, the dataset is divided into five subsets, allowing the model to be trained on four subsets while testing it on the remaining one. This process is repeated five times, ensuring that each subset serves as the test set once. The range of values for C is selected based on the behavior observed in the previous sections. The following snippet aims to accomplish this.

```
plt.errorbar(C_values, mean_scores, yerr=std_scores, fmt='o-', capsize=5, label='MSE␣␣
    STD')
```

The graphs showing the average prediction error and the standard deviation versus C are essential for understanding the model's behavior. These graphs allow us to visualize **how the Mean Squared Error varies with different values of C.** The best value of C is determined by examining the Mean Squared Error, which calculates **the average squared difference between the predicted and actual values**:

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

A lower MSE indicates better predictive performanc. Additionally, the standard deviation indicates **how dispersed the MSE values are around the average value for each C.** This information gives insight into the variability of the model and the confidence we can have in the MSE estimates.

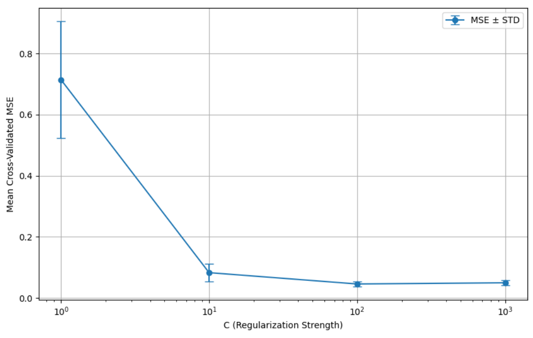

Figure 6: Mean and STD vs C in Lasso

Analyzing Figure 6, we observe that the MSE decreases sharply as C increases. However, there is an increase in MSE when moving from $C = 100$ to $C = 1000$, which is undesirable as it suggests potential overfitting. On the other hand, the standard deviation consistently tends to decrease, indicating that the variability of the model's predictions reduces with higher values of C, which is a positive sign of model stability.

(b) Recommended value of C

The following Python code performs 5-fold cross-validation to calculate the mean and standard deviation of the negative mean squared error:

```
scores = cross_val_score(model, X_poly, y, cv=5, scoring='neg_mean_squared_error')
mean_scores.append(-scores.mean())
std_scores.append(scores.std())
```

In this code, `cross_val_score` is used to evaluate the model using 5-fold cross-validation. The scoring function returns negative mean squared error, so the results are negated to get positive values.

Based on the cross-validation results, we determine the best value of C for the Lasso regression model and provide a detailed explanation for this choice.

In the table, the Mean MSE is lowest at $C = 100$ with a value of 0.0462. This analysis highlights the importance of using cross-validation when selecting the regularization parameter C. Although score may suggest that $C = 1000$ provides better performance without cross-validation, as seen in previous sections, the model shows deficiencies when confronted with data

C Value	Mean MSE	Std MSE
1	0.7095	0.1787
10	0.0828	0.0276
100	0.0460	0.0070
1000	0.0499	0.0062

Table 3: Cross-Validation Results for Lasso

it has not been trained on. Therefore, the choice of $\mathbf{C = 100}$ effectively balances regularization and model complexity, leading to a more reliable model that generalizes well to new data.

(c) Ridge regression cross-validation

We repeat parts (a) and (b) for Ridge regression.

C Value	Mean MSE	Std MSE
0.01	0.2418	0.0669
0.1	0.0658	0.0126
1	0.0524	0.0058
10	0.0534	0.0061

Table 4: Cross-Validation Results for Ridge

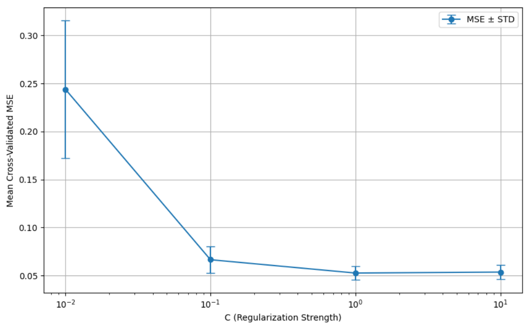

Figure 7: Mean and STD vs C in Ridge

Upon examining Figure 7, we observe that it closely resembles Figure 6. In both cases, the mean squared error decreases until it reaches a value of some value of C, 1 in this case, after which it starts to rise.

As it has been said, the optimal value of C is determined by examining the Mean Squared Error. The lowest Mean MSE occurs at $C = 1$ with a value of 0.0527.

Choosing $C = 1$ ensures the model captures important relationships without overfitting, making it a robust option for this dataset.

Comparison with the Baseline Predictor

The following Python code trains a dummy regressor using the 'mean' strategy and calculates the mean squared error (MSE):

```
dummy = DummyRegressor(strategy='mean')
dummy.fit(X_poly, y)

y_pred = dummy.predict(X_poly)

mse = mean_squared_error(y, y_pred)
```

We will compare both the Lasso and Ridge regression models with the baseline predictor. The baseline predictor has a Mean Squared Error (MSE) of 0.7044, which is significantly higher than the MSE of the final models we have selected. Notably, if we examine the Lasso model with $C = 1$, which we classified as underfitted, we find that its performance is almost equal to that of the baseline predictor, with a comparable MSE and a score of 0 in both cases.

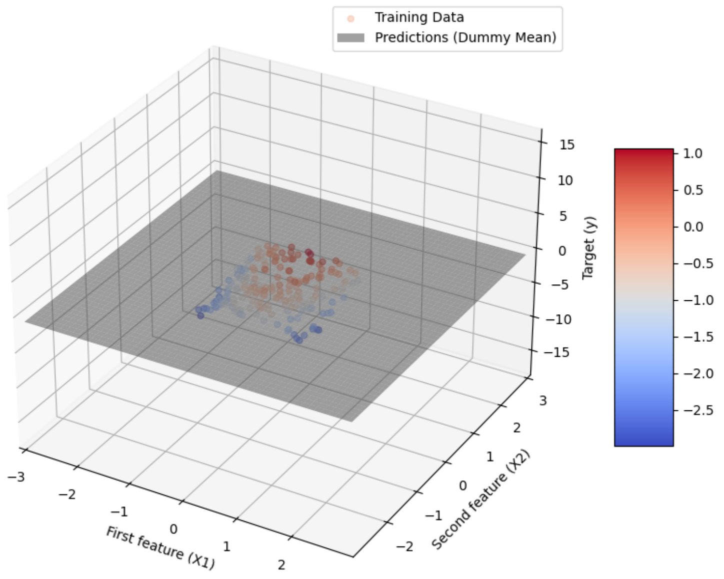

Figure 8: Dummy predictions

As we can see in the graph, the predictions of the baseline correspond to a flat plane, similar to those of the Lasso model at $C = 1$. This comparison clearly indicates that **both Lasso and Ridge regression models outperform the baseline predictor.** Thus, we can conclude that our chosen models provide better predictive performance, showing their effectiveness in capturing the underlying patterns in the data.

Additionally, when comparing Lasso with Ridge in terms of MSE, Lasso outperforms Ridge, as the MSE for Lasso (0.0462) is lower than that for Ridge (0.0527). Therefore, to conclude, **we can confidently choose the Lasso model with $C = 10$ as the best-performing model.**

Appendix

exercise.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Lasso, Ridge
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.dummy import DummyRegressor

def plot_3d(df):
    """
    Generate 3D and 2D scatter plots from a DataFrame with features and target.
    """
    X = df.iloc[:, :2].values
    y = df.iloc[:, 2].values

    # Normalize target values for color mapping
    norm = plt.Normalize(y.min(), y.max())
    colors = plt.cm.coolwarm(norm(y))

    # Create 3D scatter plot
    fig_3d = plt.figure()
    ax_3d = fig_3d.add_subplot(111, projection='3d')
    ax_3d.scatter(X[:, 0], X[:, 1], y, c=colors, marker='o')
    ax_3d.set_xlabel('First␣feature␣(X1)')
    ax_3d.set_ylabel('Second␣feature␣(X2)')
    ax_3d.set_zlabel('Target␣(y)')

    # Add color bar for 3D plot
    color_map = plt.cm.ScalarMappable(cmap='coolwarm', norm=norm)
    color_map.set_array([])
    cbar = fig_3d.colorbar(color_map, ax=ax_3d, shrink=0.5, aspect=5)
    cbar.set_label('Target␣(y)')


    plt.savefig('../output/i/3d_scatter_plot.png')


def train_model_with_polynomial_features(model_type, df, C_values=None):
    """
    Train a regression model with polynomial features (up to degree 5)
    and varying C values for Lasso and Ridge regression.
    """

    X = df.iloc[:, :2].values
    y = df.iloc[:, 2].values

    # Create polynomial features
    poly = PolynomialFeatures(degree=5, include_bias=False)
    X_poly = poly.fit_transform(X)

    if model_type in ['lasso', 'ridge']:
        output_file = f'../output/i/{model_type}_results.txt'
        models = []
        with open(output_file, 'w') as f:
            for C in C_values:
                alpha = 1 / (2 * C)

                # Create and train the specified model
                if model_type == 'lasso':
                    model = Lasso(alpha=alpha)
                elif model_type == 'ridge':
                    model = Ridge(alpha=alpha)

                model.fit(X_poly, y)

                f.write(f"\n{model_type.capitalize()}␣with␣C␣=␣{C}␣(alpha␣=␣{alpha}):\n")
                f.write(f"Intercept:␣{model.intercept_}\n")
                f.write("Coefficients:\n")

                for feature, coef in zip(poly.get_feature_names_out(['X1', 'X2']), model.
                    coef_):
                    f.write(f"{feature}:␣{coef}\n")

                f.write(f"Model␣Score:␣{model.score(X_poly,␣y):.4f}\n")
                num_nonzero = np.sum(model.coef_ != 0)
                f.write(f"Non-zero␣coefficients:␣{num_nonzero}\n")
                f.write("-" * 50 + "\n")

                models.append(model)

        return models

    elif model_type == 'dummy_mean':
        output_file = '../output/i/dummy_mean_results.txt'

        # Train DummyRegressor with 'mean' strategy
        dummy = DummyRegressor(strategy='mean')
        dummy.fit(X_poly, y)

        y_pred = dummy.predict(X_poly)

        mse = mean_squared_error(y, y_pred)

        with open(output_file, 'w') as f:
            f.write("Dummy␣Regressor\n")
            f.write(f"Score:␣{dummy.score(X_poly,␣y):.4f}\n")
            f.write(f"Mean␣Squared␣Error:␣{mse:.4f}\n")

        return dummy


def plot_model_predictions(models, df):
    """
    Generate predictions from trained models and plot them with training data.
    Save each group of two models into separate files with different colors for each batch.
    """

    X = df.iloc[:, :2].values
    y = df.iloc[:, 2].values

    model_name = type(models[0]).__name__

    x1_min, x1_max = X[:, 0].min(), X[:, 0].max()
```

```
x2_min, x2_max = X[:, 1].min(), X[:, 1].max()

# Create a grid of feature values for plotting
padding = 2
grid_x1 = np.linspace(x1_min - padding, x1_max + padding, 100)
grid_x2 = np.linspace(x2_min - padding, x2_max + padding, 100)

Xtest = []
# Create a grid of feature values
for i in grid_x1:
    for j in grid_x2:
        Xtest.append([i, j])
Xtest = np.array(Xtest)

norm = plt.Normalize(y.min(), y.max())
colors = plt.cm.coolwarm(norm(y))

custom_colors = ['red', 'yellow', 'green', 'blue']

batch_size = 2

for batch_idx in range(0, len(models), batch_size):
    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(111, projection='3d')

    # 3D scatter plot of training data
    ax.scatter(X[:, 0], X[:, 1], y, c=colors, marker='o', label='Training␣Data', alpha
        =0.5)

    ax.set_xlabel('First␣feature␣(X1)')
    ax.set_ylabel('Second␣feature␣(X2)')
    ax.set_zlabel('Target␣(y)')

    # Iterate through the current batch of models
    for idx, model in enumerate(models[batch_idx:batch_idx+batch_size]):
        # Calculate C from alpha
        if model_name != 'DummyRegressor':
            alpha = model.alpha
            C = 1 / (2 * alpha)

        poly = PolynomialFeatures(degree=5, include_bias=False)
        Xtest_poly = poly.fit_transform(Xtest)

        predictions = model.predict(Xtest_poly)

        predictions = predictions.reshape(100, 100)

        # Use custom colors for each model
        color = custom_colors[batch_idx + idx]

        # Plot the surface for each model
        if model_name == 'DummyRegressor':
            ax.plot_surface(grid_x1.reshape(100, 1), grid_x2.reshape(1, 100), predictions
                ,
                            alpha=0.5, color='gray', label='Predictions␣(Dummy␣Mean)')
        else:
            ax.plot_surface(grid_x1.reshape(100, 1), grid_x2.reshape(1, 100), predictions
                ,
                            alpha=0.3, color=color, label=f'Predictions␣(C={C})')
        # Set limits for the axes
        ax.set_xlim(X[:, 0].min() - padding, X[:, 0].max() + padding)
        ax.set_ylim(X[:, 1].min() - padding, X[:, 1].max() + padding)

        # Set z-limits to accommodate all models' predictions
        ax.set_zlim(y.min() - 15, y.max() + 15)

        sm = plt.cm.ScalarMappable(cmap='coolwarm', norm=norm)
        sm.set_array([])
        fig.colorbar(sm, ax=ax, shrink=0.5, aspect=5)

        ax.legend()

        plt.savefig(f'../output/i/{model_name.lower()}_predictions_batch_{batch_idx//
            batch_size␣+␣1}.png')

        plt.close(fig)


def plot_prediction_error_vs_C(models, C_values, df):
    """
    Plot mean and standard deviation of prediction error (MSE) vs C values
    using 5-fold cross-validation for a list of models.
    """

    X = df.iloc[:, :2].values
    y = df.iloc[:, 2].values

    poly = PolynomialFeatures(degree=5, include_bias=False)
    X_poly = poly.fit_transform(X)

    model_name = type(models[0]).__name__

    output_file = f'../output/i/{model_name.lower()}_cross_validation.txt'

    mean_scores = []
    std_scores = []

    # Perform 5-fold cross-validation for each model
    for model in models:
        scores = cross_val_score(model, X_poly, y, cv=5, scoring='neg_mean_squared_error')
        mean_scores.append(-scores.mean())
        std_scores.append(scores.std())

    mean_scores = np.array(mean_scores)
    std_scores = np.array(std_scores)

    with open(output_file, 'w') as f:
        f.write("C␣values,␣Mean␣MSE,␣Std␣MSE\n")
        for C, mean, std in zip(C_values, mean_scores, std_scores):
            f.write(f"{C},␣{mean},␣{std}\n")

    plt.figure(figsize=(10, 6))
    plt.errorbar(C_values, mean_scores, yerr=std_scores, fmt='o-', capsize=5, label='MSE␣␣
        STD')
    plt.xscale('log')
    plt.xlabel('C␣(Regularization␣Strength)')
    plt.ylabel('Mean␣Cross-Validated␣MSE')
    plt.legend()
    plt.grid()
    plt.savefig(output_file.replace('.txt', '.png'))
```

main.py

```
import exercise as i
import pandas as pd


if __name__ == "__main__":
    data_filepath = '../data/week3.csv'
    df = pd.read_csv(data_filepath)
    i.plot_3d(df)

    dummy = i.train_model_with_polynomial_features('dummy_mean', df)
    i.plot_model_predictions([dummy], df)

    C_values_lasso = [1, 10, 100, 1000]
    lasso = i.train_model_with_polynomial_features('lasso', df, C_values_lasso)
    i.plot_model_predictions(lasso, df)

    C_values_ridge = [0.01, 0.1, 1, 10]
    ridge = i.train_model_with_polynomial_features('ridge', df, C_values_ridge)
    i.plot_model_predictions(ridge, df)

    i.plot_prediction_error_vs_C(ridge, C_values_ridge,df)
    i.plot_prediction_error_vs_C(lasso, C_values_lasso, df)
```