UNIVERSITÀ DEGLI STUDI DI MILANO

PROJECT REPORT

RIDGE REGRESSION FROM SCRATCH

Statistical Methods for Machine Learning

Rakhyshev Zhanassyl 986862

1 Introduction

Ridge Regression is technique used to analyze multiple regression data that exhibit multicollinearity, which occurs when predictor variables are highly correlated. This multicollinearity can lead to unstable estimates of regression coefficients in ordinary least squares (OLS) regression, making them sensitive to small changes in the model or data. Ridge Regression introduces regularization term to the Empirical Risk Minimization (ERM) functional. This regularization term is designed to increase the model's bias slightly, but significantly reduce its variance, which overall improves the model's performance on unseen data. This is trade-off central to the concept of bias-variance tradeoff in machine learning.

Key Concepts of Ridge Regression:

1. Regularization: Ridge Regression incorporates regularization term into the optimization objective, which is essentially penalty on the size of coefficients. This helps prevent the coefficients from reaching large values, which is common problem in models suffering from multicollinearity.
2. Bias-Variance Tradeoff: By introducing bias into the model (through the regularization term), Ridge Regression aims to reduce the variance of the coefficient estimates. This trade-off often leads to model that generalizes better to unseen data, even though it might not fit the training data as closely as an ordinary least squares model.

The key difference between standard linear regression and Ridge Regression is the addition of the regularization term to the loss function, where $\alpha > 0$ is the regularization parameter, and represents the L2 norm of the weight vector.

$$\boldsymbol{w}_{S,\alpha} = \underset{\boldsymbol{w}\in\mathbb{R}^d}{\operatorname{argmin}} \|S\boldsymbol{w} - \boldsymbol{y}\|^2 + \alpha \|\boldsymbol{w}\|^2$$

Picture 1. The formula of Ridge Reggression

As α approaches 0, the Ridge Regression solution converges to the standard linear regression solution. As α approaches infinity, the solution tends towards the zero vector, indicating that higher values of α enforce stronger penalties on the size of the coefficients. It is mentioned that the gradient of the loss function (including the regularization term) equals $2S^T(Sw - y) + 2\alpha w$. The solution where this gradient vanishes gives us the optimal weight vector $w_{S,\alpha}$. Importantly, the addition of α helps ensure $S^T S+\alpha I$ that is always invertible by shifting the eigenvalues away from zero, thus overcoming the problem $S^T S$ of being nearly singular.

Ridge Regression is especially useful when dealing with datasets where predictors are correlated, which can cause ordinary least squares regression to overfit the data and produce unstable coefficient estimates. By choosing an appropriate regularization term, it can significantly improve the model's performance and generalizability.

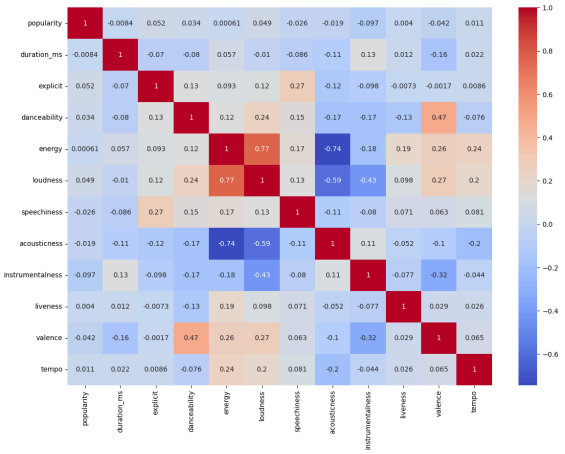
2 Preprocessing stage before Ridge Regression

Before starting the project, it's imperative to obtain the dataset from Kaggle and process analysis to facilitate future Ridge Regression. This analysis involves checking for errors, missing values in dataset and ensuring data readiness for subsequent processing.

1. ‘df.isnull().sum()’ checks for null (missing) elements in the dataset stored in the dataframe variable df.
2. ‘duplicates = df[df.duplicated()]’ following the code snippet identifies duplicate rows within the dataframe by using the duplicated() function. It creates new dataframe called duplicates containing rows that are duplicates of others in the original dataframe. The print statements then output these duplicate rows. Duplicate data can introduce bias in the model's coefficient estimation. Removing duplicates before conducting ridge regression ensures that the model is trained on reliable and non-redundant data, leading to more accurate and interpretable results.
3. ‘df = df.drop(['Unnamed: 0', 'track_id', 'artists', 'album_name', 'track_name'], axis=1)’ removes specific columns or features in dataframe, because these columns are unnecessary for analysis and contain irrelevant information for Ridge Reggression.
4. ‘df = df[df['speechiness'] <= 0.7] filters the df to include only rows, where the value in the 'speechiness' column is less than or equal to 0.7. Remove data points that don't meet certain criteria (in our case remove rows higher than 0.7).
5. ‘discrete_numeric=[feature for feature in numeric_cols if df[feature].nunique()<20]’ creates list called discrete_numeric, which contains the names of numeric features (columns) from the df that have fewer than 20 unique values. Updating discrete numeric features ensures that they are properly prepared for analysis and modeling, leading to more accurate and reliable results. Trying to separate numeric and categorical features to enhance performance of ridge regression.

Analyzing the dataset before applying Ridge Regression (or any regression model) can provide valuable insights into the relationships between features and the target variable, as well as the distribution of data. There are several examples of plots that help to better understand the dataset:

1. Correlation heatmap shows the correlation coefficients between numerical variables. It can help identify which features are most strongly correlated with the target variable (popularity in our case) and with each other. High correlation between features might indicate multicollinearity, which Ridge Regression can help address.

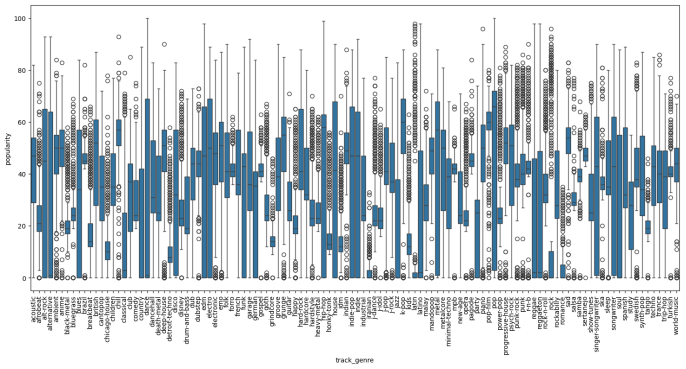


Picture 2. The correlation heatmap plot based on the given dataset

The correlation coefficient values range from -1 to 1, where: “1” indicates perfect positive linear relationship, “-1” indicates perfect negative linear relationship, “0” indicates no linear relationship. By examining the correlation coefficients, we can identify which variables have

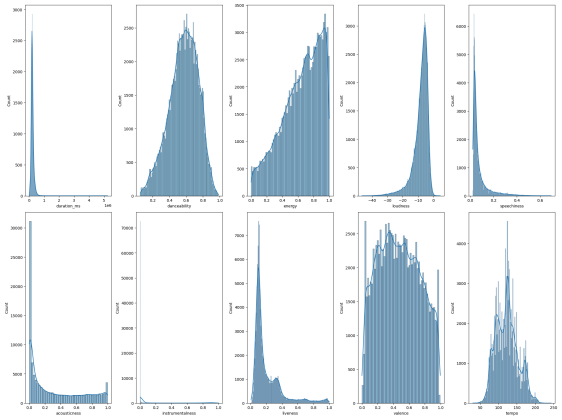
strong linear relationship with the target variable (popularity). These variables might be good predictors in regression model. In the heatmap, strong positive or negative correlation (close to “1” or “-1”, respectively) between two features would suggest multicollinearity. For example, “energy” and “loudness” have high positive correlation (0.77), which means they are multicollinear.

2. Box plots can help understand the distribution of target variable across different categories in categorical feature. For example, significant changes in popularity based on “track_genre” feature.



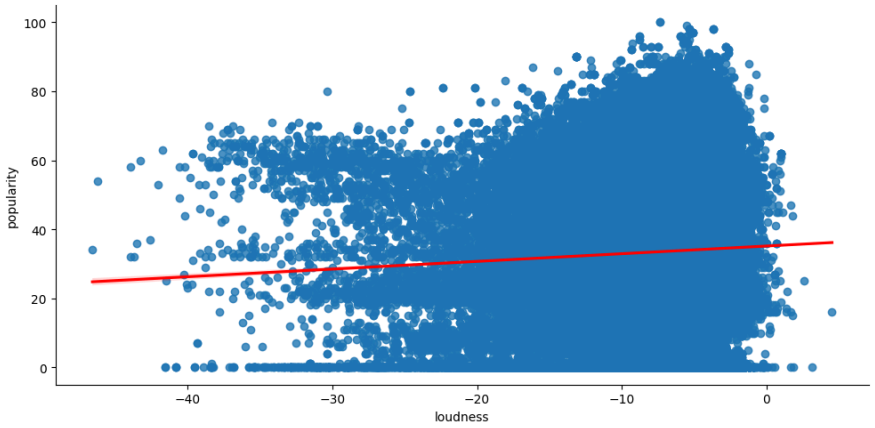
Picture 3. The box plot based on features of the given dataset

3. Understanding the distribution of numerical features can be helpful, especially to identify features that might need scaling or normalization. Ridge Regression, like other linear models, can benefit from features being on similar scale.



Picture 4. Distribution plot of numerical features of the given dataset

4. For key features identified from the correlation heatmap or pair plots, scatter plots is created against the target variable to visualize the linear relationship. Adding line of best fit will help understand how well linear model might perform.



Picture 5. The scatter plot against the target variable with line of best fit

3 Implementation of Ridge Regression for numerical features

Before performing Ridge Regression, the data should be divided into numerical and categorical features, along with separating the target variable.

1. 'numerical_features = df.select_dtypes(include=[np.number])' selects only the columns from the df that contain numerical data types (e.g., integers or floating-point numbers).
2. 'X_numerical = numerical_features.drop("popularity", axis=1)' creates new dataframe X_numerical containing all the numerical features except for the "popularity" column. It uses the drop() method to remove the "popularity" column along the specified axis (axis=1 indicates columns).
3. 'y_numerical = numerical_features["popularity"]' creates y_numerical containing only the values from the "popularity" column. It selects this column from the numerical_features dataframe, effectively isolating the target variable for the regression model.

X_numerical represents the input features (independent variables) for the ridge regression model, while y_numerical represents the target variable (dependent variable) that the model aims to predict. This separation allows for the independent and dependent variables to be appropriately handled during model training and evaluation.

3.1 Ridge Regression based on sklearn library

This code leverages functionalities provided by sklearn library to perform various steps involved in training, validating, and evaluating ridge regression model based on numerical features, along with reporting of performance metrics. its used to observe the optimal outcomes and understanding for implementing ridge regression from scratch.

```
X_train_num, X_test_num, y_train_num, y_test_num = train_test_split(X_numerical, y_numerical, test_size=0.2, ra

# train Ridge regression model
ridge_model_num = Ridge(alpha=1.0)
ridge_model_num.fit(X_train_num, y_train_num)

# perform cross-validation
cv_scores_num = cross_val_score(ridge_model_num, X_numerical, y_numerical, cv=5, scoring='neg_mean_squared_erro
cv_rmse_num = np.sqrt(-cv_scores_num)

# evaluate model on the test set
y_pred_num = ridge_model_num.predict(X_test_num)
mse_num = mean_squared_error(y_test_num, y_pred_num)
rmse_num = np.sqrt(mse_num)

print("Numerical Features based on Ridge Library:")
print(f"CV RMSE: {cv_rmse_num.mean()}")
print(f"Test RMSE: {rmse_num}")
```

3.2 Implementation of Ridge Regression from scratch

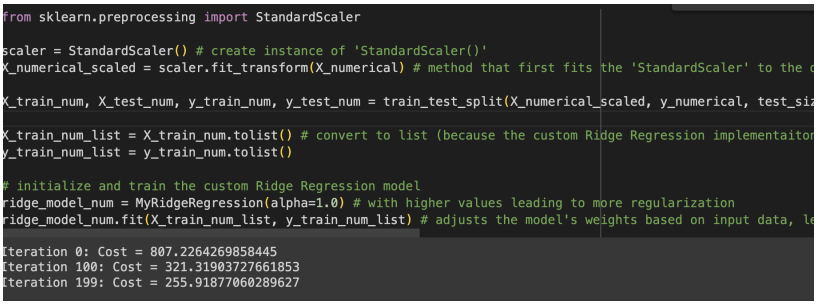
In this section of code, custom Ridge Regression class are defined that includes methods for fitting the model to data and making predictions. This class implements Ridge Regression manually without relying on external libraries like NumPy for matrix operations.

1. 'def __init__(self, alpha=1.0, num_iters=100, learning_rate=0.01):' – constructor initializes the Ridge Regression model with the specified hyperparameters:

a. alpha: The alpha parameter controls the regularization strength in Ridge Regression. Regularization is technique used to prevent overfitting by penalizing large coefficients in the model. Without regularization, model might fit the training data too closely, capturing noise in the training data as if it were true signal. This can lead to poor generalization performance on unseen data. By adjusting alpha, we can control the trade-off between fitting the training data well and keeping the model coefficients small, which helps improve the model's generalization ability.
b. num_iters: The number of iterations for which the gradient descent optimization algorithm will run. During each iteration, gradient descent updates the model's coefficients (weights) in an attempt to minimize the loss function. The number of iterations determines how long the algorithm will try to optimize the coefficients. Too few iterations might result in an under-optimized model, while too many iterations can lead to wasted computational resources and, in some cases, overfitting if the learning rate is not appropriately adjusted.
c. learning_rate: The learning rate controls how much we adjust the model's weights with respect to the loss gradient for each iteration. It essentially sets the step size of the updates during the optimization process. The learning rate is critical for the convergence of gradient descent. If it's too high, the algorithm might overshoot the minimum and diverge. If it's too low, the algorithm will converge very slowly, requiring more iterations to reach an optimal solution.
d. self.theta: placeholder for the model coefficients, initially set to 'None'. This represents the coefficients or weights assigned to each feature in the dataset, including the intercept. The values in theta determine the influence of each feature on the prediction. The model's weights are what the learning algorithm is trying to optimize. They are central to the model's predictions, as they quantify the relationship between each feature and the target variable. The learning process involves adjusting these weights to minimize the loss function, which includes both the fit to the data and the regularization term.

2. 'def add_intercept(self, X):' – modifies the feature matrix X by adding column of ones at the beginning. This column represents the intercept term in the linear model, allowing the model to fit data that does not necessarily pass through the origin.
3. 'def dot(self, x, y):' – calculates the dot product of two vectors x and y. This operation is fundamental component of matrix operations, such as calculating predictions and gradients.
4. 'def mat_vec_dot(self, X, y):' – performs matrix-vector multiplication between matrix X and vector y using the previously defined dot method. This is used to compute predictions from the model.
5. 'def transpose(self, X):' – transposes matrix X, flipping its rows and columns. This operation is crucial for calculating the gradients during model fitting.
6. 'def fit(self, X, y):' – fits the Ridge Regression model to the training data X and y. It includes initializing the model weights (theta), adding an intercept term to X, and iteratively updating the weights using gradient descent. The cost, including the regularization term, is calculated at each iteration to monitor the model's performance.
7. 'for i in range(len(self.theta)): …' – iterates over each coefficient in theta to update it based on the calculated gradient and the regularization term. Regularization is applied to all coefficients except for the intercept.
8. 'if iteration % 100 == 0: … ' – prints the cost function value every 100 iterations to monitor the model's learning progress, providing insight into whether the model is converging.

9. ‘def predict(self, X): …’ – makes predictions on new data X by adding an intercept term to X and then using matrix-vector multiplication to compute the predicted values based on the learned model coefficients (theta).



Different features in dataset can be on entirely different scales. Such disparity in scales can cause machine learning algorithms to weigh larger values more heavily than smaller ones, skewing the learning process. StandardScaler transforms each feature to have mean of 0 and standard deviation of 1, ensuring uniformity in scale across features. This code snippet is structured approach to prepare numerical data for training custom Ridge Regression model. It involves scaling the data, splitting it into training and testing sets, converting the data into list format, initializing Ridge Regression model, and then training it.

3.3 Model training and validation

The validation implements process for evaluating the performance of custom Ridge Regression model using K-Fold cross-validation, followed by testing the model on separate test set. The main parts of CV code snippet:

1. Setting up Cross-Validation (‘kf = KFold(n_splits=5, shuffle=True, random_state=42)’) – initializes KFold instances with 5 splits, enabling shuffling to ensure data points are randomized before splitting, which helps mitigate bias in the cross-validation process. A fixed random_state ensures that the results are reproducible.
2. CV process – iterates through each fold defined by KFold, separating the data into training and validation sets for each fold. This approach ensures that the model is trained and validated on different segments of the data across all folds, providing comprehensive assessment of its performance.
3. Model training and validation – custom Ridge Regression model is instantiated and trained on each training set. Then, it makes predictions on the corresponding validation set. This step is repeated for each fold, accumulating validation performance metrics.
4. Performance Evaluation – calculates the mean squared error for predictions in each validation fold, appending these scores to list. After completing all folds, the root mean squared error (RMSE) is computed across all validation folds, providing measure of the model's performance on unseen data.
5. Model testing – beyond cross-validation, the model is also tested on separate test set (not used during the cross-validation process), and the test RMSE is calculated.

This CV process can be used to fine-tune model hyperparameters (like alpha in Ridge Regression) by identifying settings that minimize the cross-validation RMSE.

4 Implemetation of Ridge Regression for both numerical and categorical features

4.1 Additional preprocessing steps

Additional preprocessing steps are required for Ridge Regression, which also involve handling categorical features.

This code snippet is part of data preprocessing routine typically used in machine learning workflows, particularly when dealing with datasets that contain both numerical and categorical features.

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import Pipeline

numerical_features = X_numerical.columns.tolist()

categorical_features = df.select_dtypes(exclude=[np.number]).columns.tolist()
categorical_features
```

```
['explicit', 'key', 'mode', 'time_signature', 'track_genre']
```

‘categorical_features = df.select_dtypes(exclude=[np.number]).columns.tolist()’ identifies categorical features by selecting columns in the dataframe that do not have numerical data type. This list will be used to specify which columns the OneHotEncoder should be applied to. There are five categorical features in the dataframe.

This code snippet defines “preprocessor” object using scikit-learn's ColumnTransformer. The preprocessor is designed to apply specific transformations to different subsets of features (columns) in dataset. “ColumnTransformer” is flexible tool for applying distinct preprocessing steps to different columns within dataset. It is particularly useful in handling datasets that contain mix of numerical and categorical features, each requiring different types of preprocessing. Each item in the list is tuple that specifies particular transformation to be applied to certain columns. “OneHotEncoder(handle_unknown='ignore')” converts categorical variables into numerical format

```
# ColumnTransformer allows to specify which transformations to apply to which colu
preprocessor = ColumnTransformer(
    # define the list of transformers - each transformer is tuple containing name
    transformers=[
        ('num', StandardScaler(), numerical_features), # transformer for numerica
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features) # t
    ])
```

that machine learning algorithms can work with more effectively. It does so by creating a binary column for each category of the variable and encoding the presence of each category with 1 or 0 (hot or not). The handle_unknown='ignore' parameter tells the encoder to ignore categories that were not seen during training, avoiding errors during prediction. Suppose the dataset with categorical feature "Color" that can take on the values "Red", "Blue", and "Green". After applying OneHotEncoder, this feature is transformed into three features: "Color_Red", "Color_Blue", and "Color_Green". For an observation with Color = "Red", the encoded features will be Color_Red = 1, Color_Blue = 0, and Color_Green = 0. Similarly, for Color = "Blue", the encoded features will be Color_Red = 0, Color_Blue = 1, and Color_Green = 0.

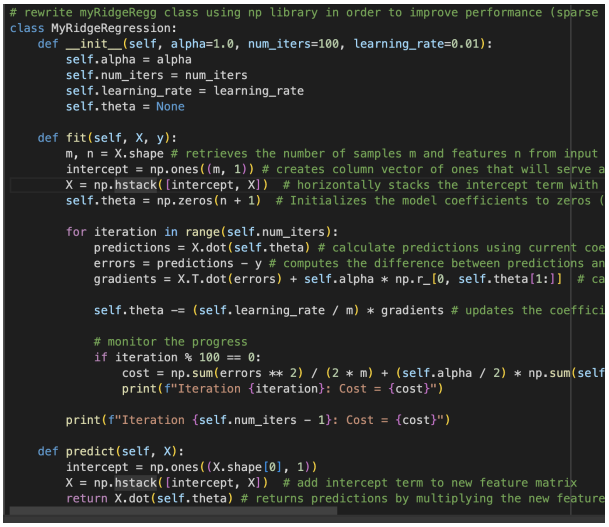
```
# create new pipeline consisting of:
my_ridge_pipeline = make_pipeline(
    preprocessor,
    to_dense,
    MyRidgeRegression(alpha=1.0, learning_rate=0.01, num_iters=100)
)
```

This code snippet creates machine learning pipeline named my_ridge_pipeline using scikit-learn's "make_pipeline" function. This pipeline integrates preprocessing steps and custom Ridge Regression model into single, seamless workflow. 'to_dense' is created using FunctionTransformer that useful to convert sparse matrix outputs (common when dealing with large, encoded categorical variables) into dense arrays. The conversion to dense arrays might be necessary for compatibility with certain operations that do not support sparse input. The 'to_dense' step ensures that data passed through the pipeline maintains compatibility with all subsequent steps. 'MyRidgeRegression' is custom implementation of Ridge Regression and initialized with specific hyperparameters.

4.2 Explanation of Ridge Regression implementation for numerical and categorical features

The code snippet below outlines custom implementation of Ridge Regression using the NumPy library to enhance performance, particularly for operations involving sparse matrices.

1. 'def fit(self, X, y)' – fits the model to the training data X (features) and y (target values).
2. 'm, n = X.shape' retrieves the dimensions of X, where m is the number of samples and n is the number of features.
3. 'intercept = np.ones((m, 1))' creates column vector of ones to serve as the intercept term, allowing the model to fit data not centered around the origin.
4. 'X = np.hstack([intercept, X])' horizontally concatenates the intercept vector to the feature matrix, effectively adding an intercept term to X.
5. 'self.theta = np.zeros(n + 1)' initializes the model's coefficients (including the intercept) as zeros.
6. Within the 'for' loop, predictions are calculated and errors are determined. Gradients are computed using both the errors and the regularization term, but the intercept term is excluded from regularization.
7. 'self.theta -= (self.learning_rate / m) * gradients' updates the coefficients in the direction that reduces the cost, with the update magnitude controlled by the learning rate and normalized by the number of samples.
8. The cost is computed and printed every 100 iterations to monitor the model's convergence during training.
9. 'def predict(self, X)' makes predictions on new data X. Similar to the fitting process, an intercept term is added to the feature matrix of new data. Predictions are computed by multiplying the modified feature matrix by the model's coefficients.



```python
# rewrite myRidgeRegg class using np library in order to improve performance (sparse
class MyRidgeRegression:
    def __init__(self, alpha=1.0, num_iters=100, learning_rate=0.01):
        self.alpha = alpha
        self.num_iters = num_iters
        self.learning_rate = learning_rate
        self.theta = None

    def fit(self, X, y):
        m, n = X.shape # retrieves the number of samples m and features n from input
        intercept = np.ones((m, 1)) # creates column vector of ones that will serve a
        X = np.hstack([intercept, X])  # horizontally stacks the intercept term with
        self.theta = np.zeros(n + 1)  # Initializes the model coefficients to zeros (

        for iteration in range(self.num_iters):
            predictions = X.dot(self.theta) # calculate predictions using current coe
            errors = predictions - y # computes the difference between predictions an
            gradients = X.T.dot(errors) + self.alpha * np.r_[0, self.theta[1:]]  # ca

            self.theta -= (self.learning_rate / m) * gradients # updates the coeffici

            # monitor the progress
            if iteration % 100 == 0:
                cost = np.sum(errors ** 2) / (2 * m) + (self.alpha / 2) * np.sum(self
                print(f"Iteration {iteration}: Cost = {cost}")

        print(f"Iteration {self.num_iters - 1}: Cost = {cost}")

    def predict(self, X):
        intercept = np.ones((X.shape[0], 1))
        X = np.hstack([intercept, X])  # add intercept term to new feature matrix
        return X.dot(self.theta) # returns predictions by multiplying the new feature
```

Picture 6. Ridge Regression implementation based on NumPy library for matrix operations

There are the results of the following code snippet based on specific hyperparameters.

```
Iteration 0: Cost = 807.373096206314
Iteration 0: Cost = 806.7332897331588
Iteration 0: Cost = 808.4618623720773
Iteration 0: Cost = 807.0946745102707
Iteration 0: Cost = 807.1145369581848
Cross-Validation RMSE: 22.41497981403211
Iteration 0: Cost = 807.3554918594988
Test RMSE: 22.43385859909111
```

Picture 7. The results of CV RMSE and Test RMSE based on numerical and categorical features

5 Differences between two implementation of Ridge Regression in the project

First Implementation (manual matrix operations) approach uses loop-based methods for mathematical operations (e.g., dot, mat_vec_dot). its slower due to the use of Python loops instead of optimized matrix operations. This approach is less efficient, especially with larger datasets.

In the manual implementation, operations such as updating the model's weights (theta) are performed iteratively through explicit loops. Each weight in theta is updated one at a time within the loop. This approach is required because:

1. Each element of theta needs to be accessed and potentially updated individually, taking into account its specific gradient and regularization adjustment.
2. The regularization term is not applied to the intercept (the first element of theta), necessitating conditional check (if i > 0) within the loop.
3. Without using NumPy or similar library that supports vectorized operations, performing these updates requires manually iterating over the elements.

Second Implementation (using NumPy for matrix operations) approach uses NumPy library for matrix operations, which are highly optimized and vectorized. Operations like np.dot for matrix multiplication and np.hstack for adding an intercept are used. It demonstrates how leveraging NumPy can significantly improve the efficiency of custom machine learning algorithms. its much faster and more efficient due to NumPy's optimizations. This version is better suited for handling larger datasets.

While both implementations theoretically perform the same Ridge Regression algorithm, the second implementation is preferred for practical applications due to its use of NumPy for efficient matrix operations. When using NumPy, the explicit loop for updating theta is replaced by vectorized

operations that allow for the simultaneous update of all weights according to the gradient descent rule. NumPy's operations are designed to efficiently compute over entire arrays at once.
This code snippet achieves the same effect as the loop in the manual implementation but leverages NumPy's ability to:

1. Perform calculations across the entire theta array without needing to iterate through its elements explicitly.
2. ‘np.r_[0, self.theta[1:]]’ is concise way to exclude the intercept from regularization by creating new array that starts with 0 (no regularization for the intercept) and follows with the rest of theta (where regularization is applied).

In summary, the loop in the manual implementation is necessary to individually update each weight, including handling regularization selectively. The NumPy implementation benefits from vectorized operations, allowing for more efficient expression of the same logic without explicit loop over each weight.

6 Conclusion

This experimental project helps in deepening understanding of ridge regression implementation and provides practice in implementing it from scratch, enhancing comprehension of each aspect of ridge regression and the process of model evaluation and validation.