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.

In the project, two different methods are used: gradient descent and the closed-form solution (an explicit formula for the coefficients). Now, the description will be provided in a simpler way. Gradient descent is an iterative optimization method used to minimize the objective function. The gradient descent update rule for Ridge Regression is: $\beta \leftarrow \beta - \eta (-2X^T(y-X\beta) + 2 \lambda \beta)$, where η is learning rate. The explicit solution for Ridge Regression can be derived by setting the gradient of the objective function to zero and solving β. The closed form solution is: $\beta = (X^TX + \lambda I)^{-1} * X^Ty$, where I is identity matrix.

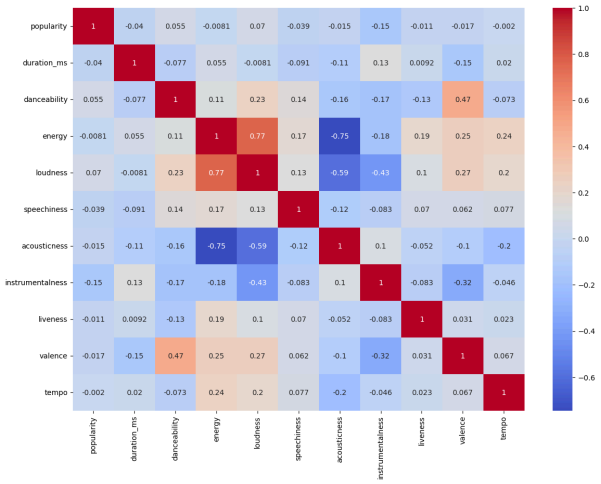
2 Preprocessing stage before Ridge Regression

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.

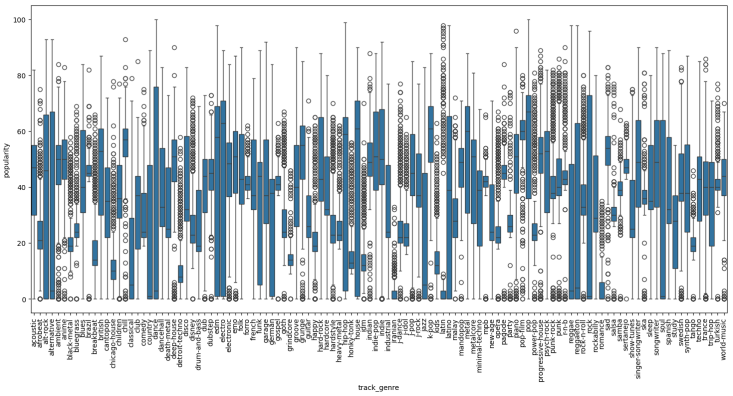


Picture 2. The correlation heatmap plot based on the given dataset

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

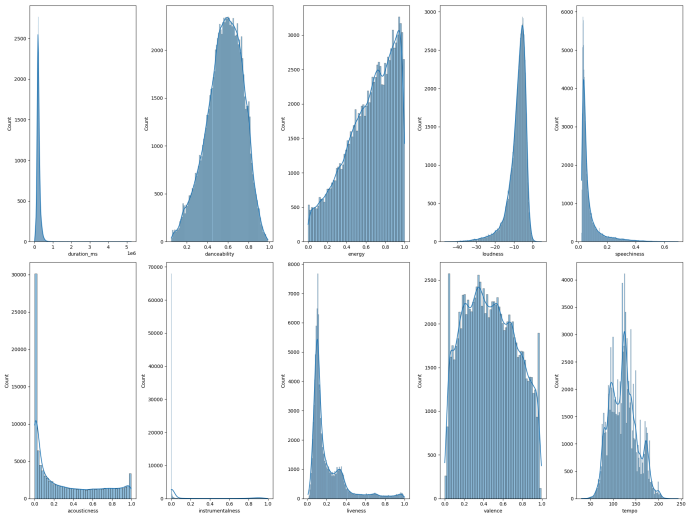
strong linear relationship with the target variable (popularity). These variables might be good predictors in regression model. In the heatmap, strong positive or negative correlation (close to “1” or “-1”, respectively) between two features would suggest multicollinearity. For example, “energy” and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



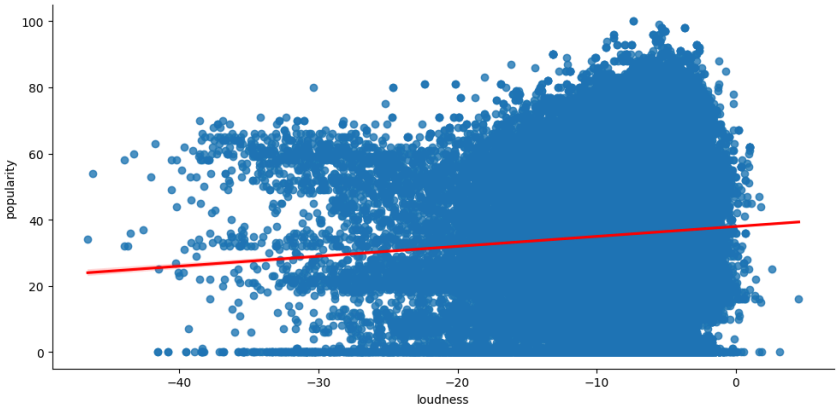
Picture 3. The box plot based on features of the given dataset

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, method='closed_form'):' – 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.
e. method: choosing between closed_form and gradient descent solutions.

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 mat_mult(self, A, B):' – performs matrix A and matrix B multiplication.
7. 'def mat_add(self, A, B):' – performs matrix A and matrix B addition.
8. 'def compute_cost(self, X, y, theta):' – computes the cost function (mean squared error + regularization). In the MyRidgeRegression class, the cost function was integrated to assess the performance of the regression model during training. It measures how well the model's predictions match the actual data, incorporating a regularization term to prevent overfitting. Monitoring the cost function helps to figure out whether the model is learning correctly.
9. Added time.time() in the methods to measure the duration for both gradient descent and closed-form solutions.

10. ‘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 or closed-form solution.
11. ‘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.
12. ‘if iteration == 0 or iteration == self.num_iters - 1:’ – prints the cost function value to monitor the model's learning progress, providing insight into whether the model is converging.
13. ‘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).

Computing the cost in machine learning models like ridge regression serves several important purposes:

1. Performance Evaluation: The cost function provides a measure of how well the model is performing. It quantifies the difference between the predicted values and the actual values, typically using mean squared error (MSE).
2. Model Optimization: During training, especially in gradient descent, the cost function is used to adjust the model parameters (weights) to minimize the cost. This process iteratively improves the model's accuracy.
3. Regularization: In ridge regression, the cost function includes a regularization term that helps prevent overfitting by penalizing large coefficients. This helps to improve the model's generalizability to new data.

By computing and monitoring the cost, we can ensure that the model is learning correctly, adjust hyperparameters, and select the best model configuration.

3.3 Model training and validation

The following steps in the code ensure there is no data leakage during the feature scaling process:

1. Data Splitting: The dataset is split into training and test sets before any scaling to prevent data leakage.
2. Feature Scaling: StandardScaler is fit on the training data only. The trained scaler is then used to transform both the training and test sets. 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.
3. Training and Evaluation: Both closed-form and gradient descent methods are used to train the model. Root Mean Squared Error (RMSE) is calculated and printed for both methods.

```
from sklearn.preprocessing import StandardScaler

X_train_num, X_test_num, y_train_num, y_test_num = train_test_split(X_numerical, y_numerical,

# create instance of 'StandardScaler()'
scaler = StandardScaler()

X_train_num_scaled = scaler.fit_transform(X_train_num).tolist()
X_test_num_scaled = scaler.transform(X_test_num).tolist()

# initialize and train the custom Ridge Regression model
ridge_model_num = MyRidgeRegression(alpha=1.0, method='gradient_descent')

# adjusts the model's weights based on input data, learning to predict the target variable
ridge_model_num.fit(X_train_num_scaled, y_train_num)
```

```
Iteration 0: Cost = 848.243924768654
Iteration 99: Cost = 307.6630408191464
Gradient Descent Time: 54.81380319595337 seconds
```

```
# predict the popularity label by ridge regression
y_pred_num = ridge_model_num.predict(X_test_num_scaled)

# calculate root-mean-square-error
mse_num = mean_squared_error(y_test_num, y_pred_num)
rmse_num = np.sqrt(mse_num)
print(f"Test RMSE: {rmse_num}")
```

```
Test RMSE: 24.627665470472092
```

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.

```
Processing fold 1...
Closed-form solution: Cost = 221.1065497026199
Closed-form Solution Time: 4.738535165786743 seconds
Processing fold 2...
Closed-form solution: Cost = 221.18543023194067
Closed-form Solution Time: 8.290873050689697 seconds
Processing fold 3...
Closed-form solution: Cost = 220.09638599778995
Closed-form Solution Time: 5.184112548828125 seconds
Processing fold 4...
Closed-form solution: Cost = 220.98597736383618
Closed-form Solution Time: 5.192025661468506 seconds
Processing fold 5...
Closed-form solution: Cost = 220.5069983911736
Closed-form Solution Time: 4.543147563934326 seconds
Cross-Validation RMSE: 21.016144984269825
```

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.

3.4 Differences between implementation of custom Ridge Regression using two solutions: gradient descent and explicit formula for coefficients (closed-form solution)

1. Gradient Descent Solution:
    a. Cost at Initial Iteration (0): Around 845-848 across all folds, indicating a high initial error.
    b. Cost at Final Iteration (99): Around 306-307, showing significant error reduction after 100 iterations.
    c. Gradient Descent Time: Approximately 53-55 seconds per fold, indicating the computational effort required for iterative optimization.
    d. Cross-Validation RMSE: 24.78
    e. Test RMSE: 24.63
2. Closed-Form Solution (explicit formula for coefficients):
    a. Cost: Consistently around 220-221 across all folds, indicating a lower and more stable error.
    b. Closed-Form Solution Time: 4.5-8.3 seconds per fold, significantly faster than gradient descent.
    c. Cross-Validation RMSE: 21.02
    d. Test RMSE: 20.95
3. Performance Assessment:
    a. Accuracy: The closed-form solution has a lower RMSE both in cross-validation (21.02) and test set (20.95) compared to gradient descent (24.78 and 24.63). This indicates better predictive performance.
    b. Computational Efficiency: The closed-form solution is much faster (4.5-8.3 seconds per fold) than gradient descent (53-55 seconds per fold), making it more efficient for this dataset.
4. Theoretical Justification:
    a. Closed-Form Solution:
        i. The closed-form solution directly solves the normal equation, $\theta = (XTX + \alpha I)^{-1} * XTy$, which ensures the global minimum of the cost function. This guarantees optimal coefficients in one step, leading to more accurate and consistent results.
        ii. Ridge regression incorporates a regularization term (αI) to penalize large coefficients, which helps in preventing overfitting and improves generalization. The closed-form solution handles this elegantly by adding αI directly to XTX.
    b. Gradient Descent:
        i. Gradient descent is an iterative optimization algorithm that updates coefficients step-by-step using the gradient of the cost function. The learning rate (η) and number of iterations determine the convergence speed and accuracy. In this case, even after 100 iterations, it does not achieve the global minimum as effectively as the closed-form solution.
        ii. The performance of gradient descent is sensitive to the learning rate. If the learning rate is too high, the algorithm may overshoot the minimum, while if it is too low, convergence can be very slow.
        iii. Gradient descent approximates the solution, and depending on the complexity of the data and the cost surface, it might get stuck in local minima or take many iterations to approach the global minimum.

c. Computational Complexity of two solutions:
    i. The closed-form solution involves matrix inversion, which has a time complexity of O(n^3) for nxn matrices. For large datasets, this can be computationally expensive, but it's one-time computation. Ideal for small to medium-sized datasets where the inversion of XTX is computationally feasible like in our case.
    ii. Gradient descent has a time complexity of O(knm), where k is the number of iterations, n is the number of features, and m is the number of samples. While each iteration is cheaper, it requires multiple iterations to converge. Preferred for very large datasets where computing the closed-form solution is impractical due to memory constraints or computational cost.

In summary, the closed-form solution's superior performance in both accuracy and computational efficiency in this scenario is due to its ability to directly minimize the cost function and effectively incorporate regularization, whereas gradient descent provides an iterative approximation that may not fully converge within the given iterations.

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 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

```
# 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
    ])
```

Color_Green = 0. Similarly, for Color = "Blue", the encoded features will be Color_Red = 0, Color_Blue = 1, and Color_Green = 0.

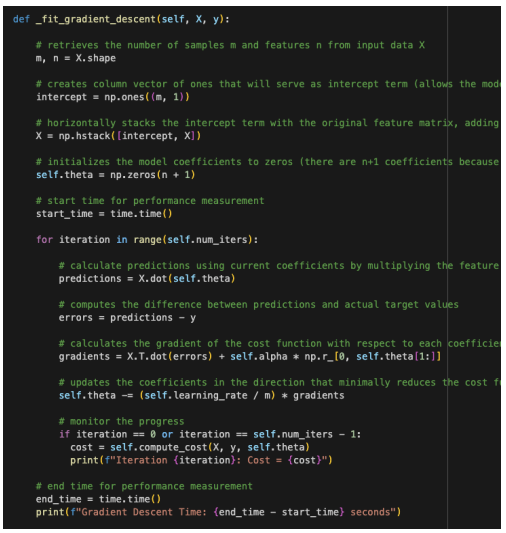
```
# create new pipeline consisting of:
my_ridge_pipeline = make_pipeline(
    preprocessor,
    to_dense,
    MyRidgeRegression(alpha=1.0, learning_rate=0.01, num_iters=100)
)
```

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 using gradient descent solution

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

1. 'def *_fit*_gradient_descent(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 function is called and printed the cost value for the first iteration and the last iteration 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.



```
def _fit_gradient_descent(self, X, y):

    # retrieves the number of samples m and features n from input data X
    m, n = X.shape

    # creates column vector of ones that will serve as intercept term (allows the mod
    intercept = np.ones((m, 1))

    # horizontally stacks the intercept term with the original feature matrix, adding
    X = np.hstack([intercept, X])

    # initializes the model coefficients to zeros (there are n+1 coefficients because
    self.theta = np.zeros(n + 1)

    # start time for performance measurement
    start_time = time.time()

    for iteration in range(self.num_iters):

        # calculate predictions using current coefficients by multiplying the feature
        predictions = X.dot(self.theta)

        # computes the difference between predictions and actual target values
        errors = predictions - y

        # calculates the gradient of the cost function with respect to each coefficie
        gradients = X.T.dot(errors) + self.alpha * np.r_[0, self.theta[1:]]

        # updates the coefficients in the direction that minimally reduces the cost f
        self.theta -= (self.learning_rate / m) * gradients

        # monitor the progress
        if iteration == 0 or iteration == self.num_iters - 1:
          cost = self.compute_cost(X, y, self.theta)
          print(f"Iteration {iteration}: Cost = {cost}")

    # end time for performance measurement
    end_time = time.time()
    print(f"Gradient Descent Time: {end_time - start_time} seconds")
```

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.

```
Processing fold 1...
Iteration 0: Cost = 820.7495293402354
Iteration 99: Cost = 229.52279030740127
Gradient Descent Time: 2.2874438762664795 seconds
Processing fold 2...
Iteration 0: Cost = 819.4271609853769
Iteration 99: Cost = 228.8604894602559
Gradient Descent Time: 1.6480932235717773 seconds
Processing fold 3...
Iteration 0: Cost = 817.618957530649
Iteration 99: Cost = 228.47573134083243
Gradient Descent Time: 1.6772003173828125 seconds
Processing fold 4...
Iteration 0: Cost = 819.6999414240354
Iteration 99: Cost = 227.85609902117102
Gradient Descent Time: 2.8912875652313232 seconds
Processing fold 5...
Iteration 0: Cost = 820.8903576151471
Iteration 99: Cost = 228.27749396973343
Gradient Descent Time: 1.8708624839782715 seconds
Cross-Validation RMSE: 21.385372336633527
```

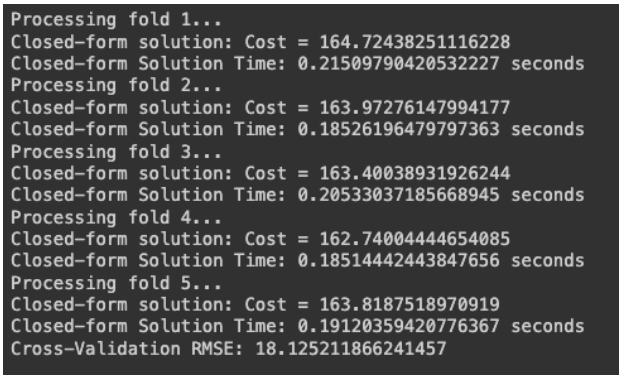
Picture 7. The results of CV RMSE based on numerical and categorical features using gradient descent

4.3 Explanation of Ridge Regression implementation for numerical and categorical features using closed-form solution

The code snippet below outlines custom implementation of Ridge Regression using the closed-form solution with help of NumPy library to enhance performance.

1. ‘def _fit_closed_form(self, X, y):’– fits the model to the training data X (features) and y (target values).
2. ‘m, n = X.shape’ – gets the number of samples and features.
3. ‘intercept = np.ones((m, 1))’ – creates column vector of ones for the intercept term.
4. ‘X = np.hstack([intercept, X])’ – add intercept term to the feature matrix.
5. ‘A = X.T.dot(X) + self.alpha * np.eye(n + 1)’ – computes matrix A = X^T * X + alpha * I and np.eye(n + 1) creates identity matrix of size (n+1), ensuring regularization is applied to all coefficients except the intercept.
6. ‘A[0, 0] -= self.alpha’ – adjusts for not regularizing the intercept term, the top-left element of A is reduced by self.alpha, so the intercept term is not regularized.
7. ‘b = X.T.dot(y)’ – computes the vector b = X^T * y.
8. ‘self.theta = np.linalg.inv(A).dot(b)’ – compute the closed-form solution for theta = inv(A) * b, it first computes the inverse of matrix A and then takes the dot product with vector b.

9. ‘cost = self.compute_cost(X, y, self.theta)’ – calls the cost function and prints the cost value + regularization.



Picture 8. The results of CV RMSE based on numerical and categorical features using explicit formula for coefficients

Key aspects that demand careful consideration are evident when using both solutions:

1. In ridge regression, we add regularization term to the cost function to prevent overfitting by penalizing large coefficients. This regularization term is applied to all coefficients, including the intercept term. However, the intercept term should not be regularized because it is not part of the feature space and does not influence the model complexity. By subtracting self.alpha from the top-left element of matrix A, we ensure that the intercept term is excluded from regularization.
2. In the fit_closed_form method implemented before, the regularization term is added manually through the creation of an identity matrix. This ensures that the regularization term is added to the diagonal elements of XTX, excluding the intercept term from regularization since XTX[0][0] remains unchanged by this specific process.
3. 'gradients = X.T.dot(errors) + self.alpha * np.r_[0, self.theta[1:]]'v: Computes the gradients of the cost function with respect to each coefficient. Regularization '(self.alpha * np.r_[0, self.theta[1:]])' is added to the gradients of all coefficients except the intercept. 'np.r_[0, self.theta[1:]]' ensures that the intercept term is not regularized.

5 Differences between two implementation of Ridge Regression (custom implementation and implementation using NumPy library) in the project

Custom implementation of MyRidgeRegression class uses Python built-in functions and manual implementations of matrix operations, which can be slower and less efficient. The second implementation of MyRidgeRegression class (because of working with sparse matrices (OneHotEncoder <- "category" features)) uses NumPy's optimized linear algebra routines, resulting in faster computations.

NumPy implementation directly uses X.dot(theta) for predictions, benefiting from NumPy's optimized operations. Custom implementation uses mat_vec_dot for matrix-vector multiplication or other methods, which is manually implemented.

- Gradient Descent in the custom implementation, iterative updates are manually implemented using Python loops. in the Numpy implementation, iterative updates are handled using NumPy, making the operations faster.
- Closed-form solution in the custom implementation, manually implements matrix inversion and multiplication. Numpy uses np.linalg.inv and np.dot for efficient matrix operations.

NumPy significantly reduces computation time and simplifies the code, making it easier to maintain and less prone to errors, especially working with sparse matrices (large datasets).

One more benefit of using NumPy in gradient descent solution or why np library is used:
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 a loop. This approach is required because,

- Each element of theta needs to be accessed and potentially updated individually, taking into account its specific gradient and regularization adjustment.
- The regularization term is not applied to the intercept (the first element of theta), necessitating a conditional check (if i > 0) within the loop.

Without using NumPy or a similar library that supports vectorized operations, performing these updates requires manually iterating over the elements.
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.

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 a new array that starts with 0 (no regularization for the intercept) and follows with the rest of theta (where regularization is applied).

6 Results of implementing Ridge Regression using both numerical and categorical features with the help of the NumPy library, compared to a custom implementation using only numerical features based on closed-form solution

1. Closed-Form Solution with Numerical and Categorical Features:
    a. Cost: Approximately 163-165 across all folds, indicating lower errors compared to using only numerical features.
    b. Solution Time: Around 0.18-0.21 seconds per fold, significantly faster than using only numerical features.
    c. Cross-Validation RMSE: 18.13.
    d. Test RMSE: 18.10
2. Closed-Form Solution with Only Numerical Features:
    a. Cost: Approximately 220-221 across all folds, indicating higher errors.
    b. Solution Time: Around 4.5-8.3 seconds per fold.
    c. Cross-Validation RMSE: 21.02.
    d. Test RMSE: 20.95.
3. Performance Assessment:
    a. Accuracy – using both numerical and categorical features leads to lower cost, and lower RMSE indicates better model performance and more accurate predictions.
    b. Computational Efficiency – the solution time is significantly lower, indicating faster computations by using numerical and categorical features (mostly because of using np libraries in the solution).
4. Theoretical Justification:
    a. Feature Inclusion: numerical and categorical features provides more information, leading to more accurate model. Categorical features can capture additional variability in the data that numerical features alone might miss.
    b. Dimensionality and Regularization: regularization term helps manage the increased complexity from additional features, preventing overfitting and improving

generalization. While simpler, the model may miss important patterns present in the categorical features, leading to underfitting.

Using both numerical and categorical features results in more accurate and computationally efficient model for ridge regression. This demonstrates the importance of utilizing all relevant features in the dataset to enhance model performance. The same result will be received for the gradient descent solution with an RMSE of 21.32 and a CV RMSE of 21.38, instead of 24.63 and 24.78.

In the project, one more comparison is made between two different implementations: custom implementation (for numeric and categorical features) and implementation using the sklearn library. Both the custom implementation using closed_form solution (numeric and category features) and sklearn.linear_model.Ridge effectively use numerical and categorical features, leading to similar test RMSE values. The custom implementation using closed-form solution slightly outperforms in cross-validation RMSE, suggesting it may handle training data variations better in this specific scenario. The sklearn implementation remains highly reliable, optimized, and easier to use for practical purposes. The final comparison of solutions serves to validate the accuracy of the implementation of Ridge Regression.

7 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.

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.