Statistical Learning Experimental Project - (Kernel) Ridge Regression

Elena Dal Savio, matr. 07223A

May 2024

Introduction

This experimental project involves implementing Ridge Regression from scratch to analyze the features of a Spotify dataset. The primary purpose of this project is to predict the tracks' popularity which can improve the user experience and engagement on music platforms. The project is divided into several steps to achieve this goal. Firstly, essential functions used throughout the project are defined. Then, simulations and experiments are conducted using only numerical features, followed by another set of experiments incorporating categorical features. Third step covers cross-validation, and finally, concluding remarks are presented.

Let's begin by examining the input data. This Spotify dataset is composed of various audio features such as `track_name`,`duration_ms`, `danceability`, `track_genre`, and more.

	track_id	artists
0	5SuOikwiRyPMVoIQDJUgSV	Gen Hoshino
1	4qPNDBW1i3p13qLCt0Ki3A	Ben Woodward
2	1iJBSr7s7jYXzM8EGcbK5b	Ingrid Michaelson;ZAYN
3	6lfxq3CG4xtTiEg7opyCyx	Kina Grannis
4	5vjLSffimiIP26QG5WcN2K	Chord Overstreet

	album_name	track_name
0	Comedy	Comedy
1	Ghost (Acoustic)	Ghost - Acoustic
2	To Begin Again	To Begin Again
3	Crazy Rich Asians (Original Motion Picture Soundtrack)	Can't Help Falling In Love
4	Hold On	Hold On

	popularity	duration_ms	explicit	danceability	energy	key
0	73	230666	False	0.676000	0.461000	1
1	55	149610	False	0.420000	0.166000	1
2	57	210826	False	0.438000	0.359000	0
3	71	201933	False	0.266000	0.059600	0
4	82	198853	False	0.618000	0.443000	2

	loudness	mode	speechiness	acousticness	instrumentalness
0	-6.746000	0	0.143000	0.032200	0.000001
1	-17.235000	1	0.076300	0.924000	0.000006
2	-9.734000	1	0.055700	0.210000	0.000000
3	-18.515000	1	0.036300	0.905000	0.000071
4	-9.681000	1	0.052600	0.469000	0.000000

	liveness	valence	tempo	time_signature	track_genre
0	0.358000	0.715000	87.917000	4	acoustic
1	0.101000	0.267000	77.489000	4	acoustic
2	0.117000	0.120000	76.332000	4	acoustic
3	0.132000	0.143000	181.740000	3	acoustic
4	0.082900	0.167000	119.949000	4	acoustic

The tables above represent the first rows of the Spotify dataset. The features can be categorized into two types: numerical and categorical, such as `track_id`, `artist`, `album_name`, `track_name`, `explicit`, `track_genre`. We will discuss how to handle these categorical features later on. The main idea is to convert these features into numerical features using appropriate techniques; I chose the label encoding.

In Fig. 1, there's a heatmap called the *Correlation Matrix*. It shows correlations between different features, both numerical and appropriately converted categorical features. A correlation matrix is like a table showing how much two things are connected. The *correlation coefficient*, which is a number between -1 and 1, tells us how strong the relationship is between two sets of data. If it's close to 1, it means they're strongly connected. If it's close to -1, there's inverse correlation between the two variables. If it's close to 0, there's not much connection.

As we can see in Fig. 1, `popularity` has minimal correlation with most of the other features. This suggests that there are no single features that are strongly predictive of 'popularity'. Because of this, the only option is that popularity is influenced by a combination of multiple features; Ridge Regression is usually effective in this scenario. Moreover, we can observe that there are features strongly correlated to each other (for example energy and loudness); this fact can lead do overfitting. Again, Ridge Regression usually works well in this context.

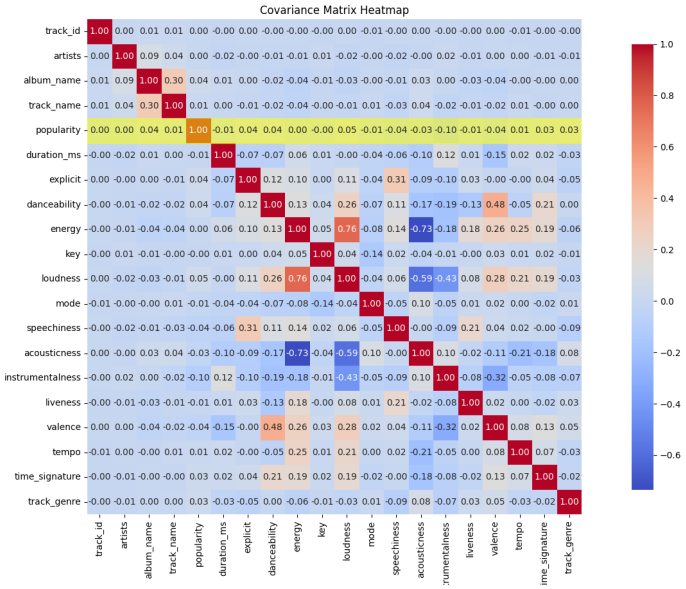


Figure 1: Correlation Matrix

1 Training the model

Let's start by defining some functions that we'll use through the project. First of all, we should import the libraries we'll use in this paper.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error,
mean_absolute_error,r2_score
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import cross_val_score
from sklearn.base import BaseEstimator, RegressorMixin
from sklearn.linear_model import Ridge
import ipywidgets as widgets
from IPython.display import display
import seaborn as sns
from matplotlib.ticker import ScalarFormatter
from ipywidgets import fixed
```

Secondly, we define a function called `getNumericalDataFrame()` which turns

the CSV dataset into a dataframe. We start focusing on the numerical features and we use Pandas' `drop()` function to remove the categorical ones. It's important to normalize the data to improve the performance and stability of the model being defined.

```
def getNumericalDataFrame():
    df = pd.read_csv('dataset.csv', encoding="utf8", index_col=0)

    datasetDf = df.copy()
    categoricalColumns = ['track_id','artists','album_name',
    'track_name','explicit','track_genre']
    numericDf = datasetDf.drop( columns =
    categoricalColumns , axis = 1, inplace = False)

    #scaling
    dfMean = numericDf.mean()
    dfStd = numericDf.std()
    numericDf = (numericDf - dfMean) / dfStd
    return numericDf
```

Next, let's define the `splitTrainTestSets()` function. This function randomly divides the input dataframe into training and testing parts based on the specified size for the test part. We set a seed to ensure reproducibility of the experiment. Additionally, this function calculates the training predictor variable (X), the training target variable (y), as well as the test set predictors (xTest) and the test set target (yTest).

```
def splitTrainTestSets(df, testSize):
    np.random.seed(8) #seed per riproducibilità
    arrayIndices = np.arange(0, len(df), 1)
    arrayTest = np.random.choice(arrayIndices,
    int(len(df)*testSize), replace=False)
    arrayTrain = np.setdiff1d(arrayIndices, arrayTest)

    arrayTest.sort()
    arrayTrain.sort()

    test, train = df.iloc[arrayTest], df.iloc[arrayTrain]

    X = train.copy()
    X = X.drop('popularity', axis = 1, inplace = False)
    X.insert(0, 'intercept', np.ones((X.shape[0],1)))
    y = train['popularity'].copy()

    xTest = test.copy()
    yTest = test['popularity'].copy()
    xTest = xTest.drop('popularity', axis=1, inplace = False)
```

```
    xTest.insert(0, 'intercept', np.ones((xTest.shape[0],1)))

    return X, y, xTest, yTest
```

1.1 Handling the categorical features

When incorporating categorical features into our model, it's essential to transform these features into measurable quantities. For this purpose, we'll employ a label encoding technique. I've chosen a label encoding over a one-hot encoding because the one-hot encoding significantly increases the size of the dataframe. One-hot encoding creates a separate column for each category in every categorical variable, potentially leading to a more complex and slower-to-train model, as well as issues with sparsity and overfitting.

On the contrary, label encoding operates by converting all unique values of a feature into strings, then mapping these strings to integers. We'll use a function called `np.searchsorted()`; typically, the `np.searchsorted()` function is used to find the indices in a sorted array such that if elements from another array are inserted before these indices, the order of the first array would remain preserved.

In this specific scenario, following the suggestion from [5] and [6], we collect all unique values into an array called `uniqueValues`. Then, the `np.searchsorted()` function iterates through the elements of `dummiesDf[feature]` to locate the corresponding index of each element in `uniqueValues`. Once again, we standardize the data to enhance the performance and stability of the model under development.

```
def getDummiesDataFrame():
    df = pd.read_csv('dataset.csv', encoding="utf8",
    index_col=0)

    dummiesDf = df.copy()
    categoricalColumns = ['track_id','artists','album_name',
    'track_name','explicit','track_genre']

    for feature in categoricalColumns:
        dummiesDf[feature] = dummiesDf[feature].astype(str)
        uniqueValues = np.unique(dummiesDf[feature])
        dummiesDf[feature] = np.searchsorted(uniqueValues,
        dummiesDf[feature])

    #scaling
    dfMean = dummiesDf.mean()
    dfStd = dummiesDf.std()
    dummiesDf = (dummiesDf - dfMean) / dfStd
    return dummiesDf
```

1.2 Ridge Regression Model Implementation

Ridge Regression [1] is an extension of Linear Regression that addresses multicollinearity by adding a regularization term to the cost function. Instead of minimizing the Residual Sum of Squares (RSS) as in ordinary least squares (OLS), Ridge Regression minimizes:

$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \alpha \sum_{j=1}^{p} \beta_j^2, \tag{1}$$

where $\alpha \geq 0$ is a *tuning parameter*. Here, n is the number of observations, p is the number of features, y_i is the target value for the i-th observation, x_{ij} is the value of the j-th predictor for the i-th observation, and β_j are the coefficients corresponding to each predictor. The term $\alpha \sum_{j=1}^{p} \beta_j^2$ is a shrinkage penalty that shrinks the coefficients β_j towards zero, helping to prevent overfitting. When $\alpha = 0$, Ridge Regression becomes OLS; as α increases, the impact of the penalty grows, pushing the coefficients towards zero.

To find the coefficients θ (which includes β_0 and β_j), Ridge Regression solves the following equation:

$$\theta = (X^T X + \alpha I)^{-1} X^T y, \tag{2}$$

where X is the $n \times p$ matrix of input features from the training set, y is the $n \times 1$ vector of target values from the training set, I is the $p \times p$ identity matrix, and αI is the regularization term that prevents overfitting by shrinking the coefficients.

This optimization ensures that the model parameters are adjusted to make accurate predictions on the training data.

To complete our model, we need to define a function for Ridge Regression. Instead of a regular function, I chose to define a class because it offers more flexibility, especially when we want to use features from the `scikit- learn` package. For example, having this class allows us to utilize Cross Validation, a feature we'll use later on. I'm following the user guide [3] to make sure my class follows the rules for an estimator.

The class I created consists of five main objects: `init(), fit(), predict(), get_params() e set_params()`.

- The `init()` function initializes the hyperparameter (alpha) and self. Self is a reference to the instance of the class.
- The `fit()` function trains the model using the training data. It optimizes the model parameters to make accurate predictions.
- The `predict()` function predicts values based on the input data and the trained model.
- Finally, `get_params()` is used to retrieve the current model parameters, while and `set_params()` allows setting new parameter values.

```
class MyCustomModelRidgeRegression(BaseEstimator, RegressorMixin):
    def __init__(self, alpha):
        self.alpha = alpha
        self.theta = None

    def fit(self, X, y):
        # Regularization matrix
        A = np.identity(X.shape[1])
        A[0,0] = 0
        aBiased = self.alpha*A
        self.theta = np.linalg.inv(X.T.dot(X) + aBiased).dot(X.T).dot(y)
        return self

    def predict(self, X):
        predictions = X.dot(self.theta)
        return predictions

    #def score(self, xTest, yTest):
     #   predictions = np.array(self.predict(xTest))
      #  mse = mean_squared_error(yTest, predictions)
       # return mse

    def get_params(self, deep = True):
        return {"alpha": self.alpha}

    def set_params(self, **params):
        for parametro, valore in params.items():
            setattr(self, parametro, valore)
        return self
```

2 Correctness of the model

Now let's compare the model we've just defined from scratch with the `sklearn.linear_model.Ridge` class from the scikit-learn package. This will help us to estimate correctness and accuracy of our model.

For this purpose I'm taking fixed values for aplha ($alpha = 10$) and testSize ($testSize = 0.8$) and I will compare the predictions of the two models.

```
#init params
alpha=10
testSize = 0.8

#init dataframe
outDf = getNumericalDataFrame()
X, y, xTest, yTest = splitTrainTestSets(outDf, testSize)
```

```
#scikit Ridge
classicRidge = Ridge(alpha)
classicRidge.fit(X,y)
classicPredictions = classicRidge.predict(xTest)

#Ridge from scratch
scratchRidge = MyCustomModelRidgeRegression(alpha)
scratchRidge.fit(X,y)
scratchPredictions = scratchRidge.predict(xTest)

# Differences between the predictions of the two models
modelError = classicPredictions - scratchPredictions
meanModelError = modelError.mean()
print("The mean error between the ridge predictions
from scikit-learn package and my model is:
",meanModelError)
plt.xlabel('custom model error')
plt.hist(modelError)
```

Using the code above I calculated the predictions of the two models and the difference between the two predictions. The histogram representing these differences is shown in Fig. 2. As we can see, the differences are very small and the mean of those quantities is $1.6125842970281659e^{-18}$. Thus, the model `MyCustomModelRidgeRegression` accurately represents Ridge Regression.

I also varied the values of the parameters (alpha and testSize) and used `getDummiesDataFrame()` instead of `getNumericalDataFrame()`; the results remained consistent.

3 Experiments

Now it's time to do some experiments with our model. First, we define the function `UpdatePlot(testSize, dataFrameType, maxAlpha)`. This function calculates the mean square error (MSE), the mean absolute error (MAE), and the r-squared for each value of alpha and creates the respective plots. Moreover, it allows us to choose which dataframe to consider (the numerical one or the categorical one) and define the upper limit of the range for the alphaArray.

```
def updatePlot(testSize, dataFrameType, maxAlpha):
    #init params
    alphaArray = np.linspace(0.1, maxAlpha, num = 500)
    mseArray = []
    maeArray = []
    rSquaredArray = []
    bestScoreMSE = float('inf')
```

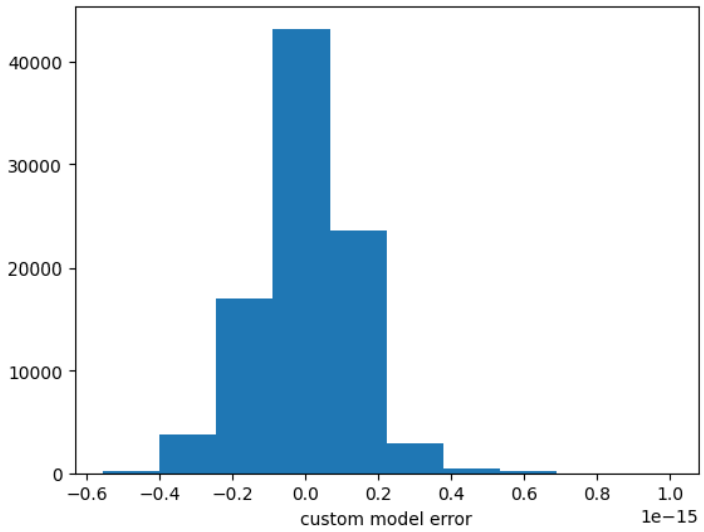


Figure 2: Differences between custom model and Ridge Regression

```
bestScoreMAE = float('inf')
bestScoreRSquared = -float('inf')
bestAlphaMSE = None
bestAlphaMAE = None
bestAlphaRSquared = None

#init dataframes
if dataFrameType == 'numerical':
    outDf = getNumericalDataFrame()
    X, y, xTest, yTest = splitTrainTestSets(outDf, testSize)
elif dataFrameType == 'categorical':
    outDf = getDummiesDataFrame()
    X, y, xTest, yTest = splitTrainTestSets(outDf, testSize)
else :
    print("invalid option for dataFrameType")


#alpha loop
for alpha in alphaArray:
    ridgeRegression = MyCustomModelRidgeRegression(alpha)
    ridgeRegression.fit(X, y)
    predictions = ridgeRegression.predict(xTest)

    mse = np.mean((yTest - predictions) ** 2)
    mseArray.append(mse)
```

```
    if mse < bestScoreMSE:
        bestScoreMSE = mse
        bestAlphaMSE = alpha


    RSS = np.sum((yTest - predictions) ** 2)
    TSS = np.sum((yTest - np.mean(yTest)) ** 2)
    r2 = 1 - (RSS / TSS)
    rSquaredArray.append(r2)
    if r2 > bestScoreRSquared:
        bestScoreRSquared = r2
        bestAlphaRSquared = alpha


    mae = np.mean(np.abs(yTest - predictions))
    maeArray.append(mae)
    if mae < bestScoreMAE:
        bestScoreMAE = mae
        bestAlphaMAE = alpha


#construction of the plot
plt.figure(figsize=(22, 6))
plt.yscale('linear')

#construction of the subplot mse
plt.subplot(1, 3, 1)
plt.plot(alphaArray, mseArray, label='MSE')
plt.xlabel('alpha')
plt.ylabel('MSE')
plt.title('Plot of MSE with respect to alpha')
plt.legend()
plt.grid(True)
mseArray = np.array(mseArray)
print("mean MSE: ",mseArray.mean())
print("best MSE: ", bestScoreMSE)
print("best alpha MSE: ", bestAlphaMSE, "\n")

#construction of the subplot r^2
plt.subplot(1, 3, 2)
plt.plot(alphaArray, rSquaredArray, label='R^2')
plt.xlabel('alpha')
plt.ylabel('R^2')
plt.title('Plot of R-squared with respect to alpha')
plt.legend()
plt.grid(True)
```

```
rSquaredArray = np.array(rSquaredArray)
print("mean R-Squared: ",rSquaredArray.mean())
print("best R-Squared: ", bestScoreRSquared)
print("best alpha R-Squared: ", bestAlphaRSquared, "\n")

#construction of the aubplot mae
plt.subplot(1, 3, 3)
plt.plot(alphaArray, maeArray, label='MAE')
plt.xlabel('alpha')
plt.ylabel('MAE')
plt.title('Plot of MAE with respect to alpha')
plt.legend()
plt.grid(True)
maeArray = np.array(maeArray)
print("mean MAE: ",maeArray.mean())
print("best MAE: ", bestScoreMAE)
print("best alpha MAE: ", bestAlphaMAE, "\n")
```

3.1 Experiments with numerical features only

Let's start considering numerical features only; to do this we use the function `updatePlot` with $dataFrameType =' numerical'$.

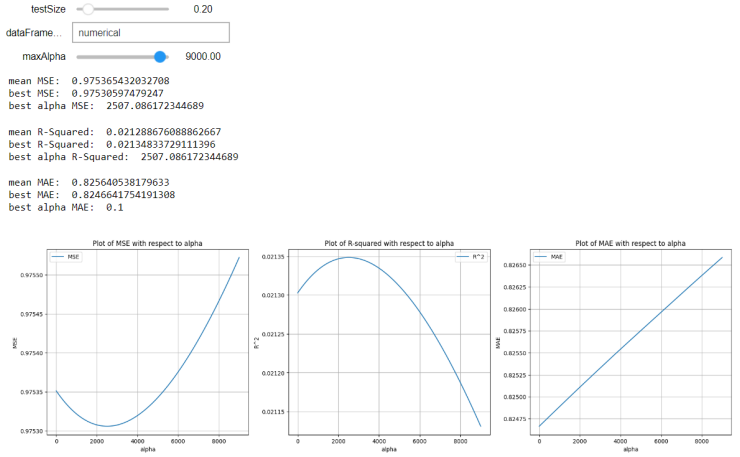
```
testSizeSlider = widgets.FloatSlider(value=0.8, min=0.1, max=0.9,
step=0.1, description='testSize')
maxAlphaSlider = widgets.FloatSlider(value=5000,
min=0, max=10000, step=100, description='maxAlpha')
interactivePlot = widgets.interactive(updatePlot,
testSize=testSizeSlider, dataFrameType='numerical',
maxAlpha=maxAlphaSlider)
display(interactivePlot)
```

The code above allows us to display the different plots and two sliders to vary the values of the hyperparameter alpha and testSize. I used the guide in [4] to build these objects.

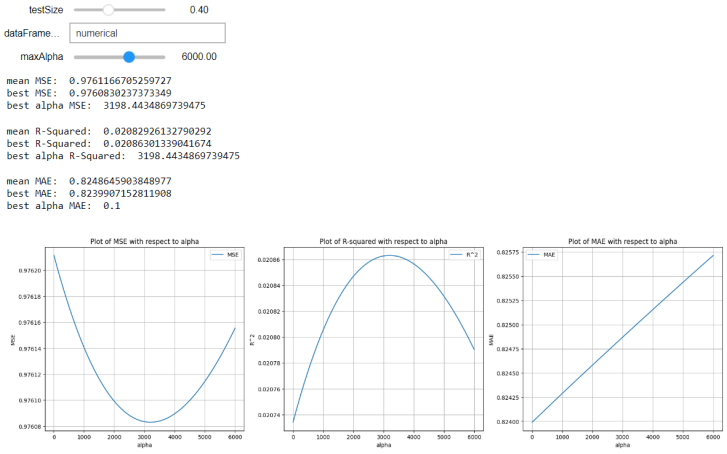
We can observe from Fig. 3 that as the test size increases, the alpha with the best MSE decreases. The same trend is seen for the alpha with the best r-squared. Additionally, the alpha values that yield the best MSE and best r-squared are almost identical. Differently, the MAE value consistently increases with alpha and does not depend on test size.

The decreasing best MSE alpha values with larger test sizes suggest possible overfitting. Despite using ridge regularization to reduce overfitting, if the model is too complex for the available data, it may not perform well on new unseen test data.

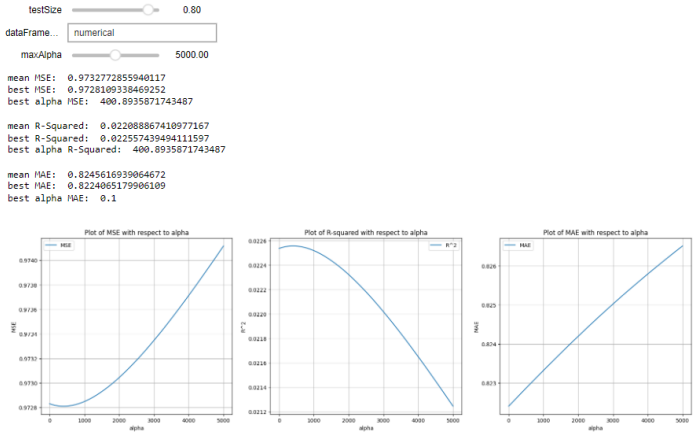
Furthermore, the r-squared value never exceeds 0.22. This indicates a weak correlation between the predictive features and the target variable, which could



(a) Experiments with testsize=0.2

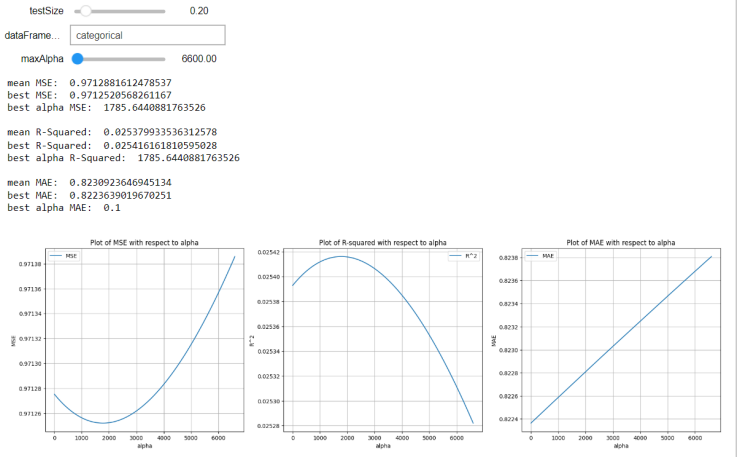


(b) Experiments with testsize=0.4

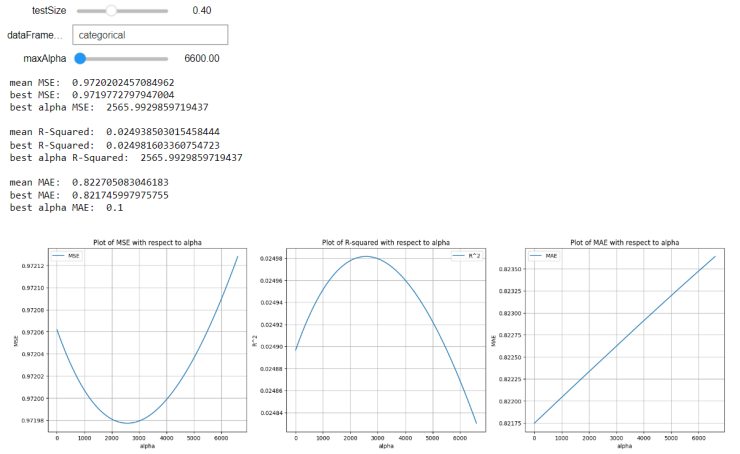


(c) Experiments with testsize=0.8

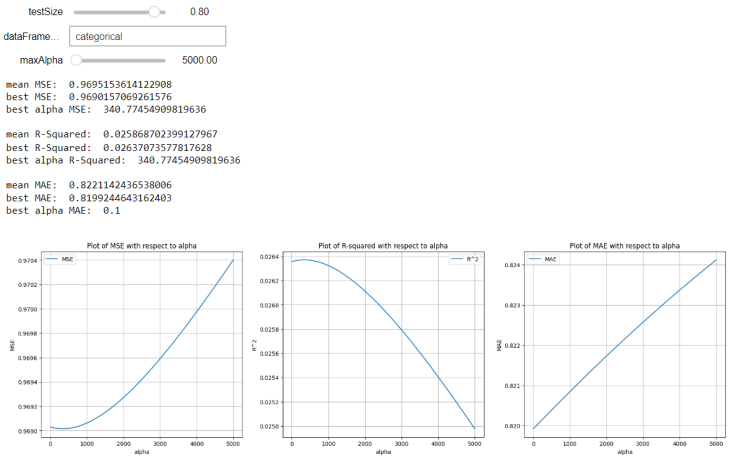
Figure 3: Numerical Experiments



(a) Experiments with testsize=0.2



(b) Experiments with testsize=0.4



(c) Experiments with testsize=0.8

Figure 4: Categorical Experiments

limit the model's predictive accuracy and the maximum achievable r-squared.

3.2 Experiments with numerical features and categorical features

Let's see if there's something different including categorical features and varying testSize and alpha as did before.

```
testSizeSlider = widgets.FloatSlider(value=0.8, min=0.1, max=0.9,
step=0.1,
description='testSize')
maxAlphaSlider = widgets.FloatSlider(value=5000,
min=0, max=200000, step=100, description='maxAlpha')
interactivePlot = widgets.interactive(updatePlot,
testSize=testSizeSlider, dataFrameType='categorical',
maxAlpha=maxAlphaSlider)
display(interactivePlot)
```

Considering the same values used before we observe a slightly different behaviour. Indeed we can observe that bestAlpha has a small value (340) when testSize is big while it has a bigger value (1785) when testSize is smaller. Thus, bestAlpha value depends on the amount of training data available. A larger test size requires less regularization to avoid underfitting, while a smaller test size requires more regularization to avoid overfitting. This indicates that the model is sensitive to the amount of data, and using ridge regularization is crucial to balance the trade-off between bias and variance.

Also, adding encoded categorical variables increases the amount of information available to the model: more information can help the model make better predictions with less need for regularization. If these new encoded variables are good at showing the variance in the training data, the model won't overfit. So, we can use a smaller alpha without risking the model's overall performance.

3.3 Cross Validation and Risk Estimates

Let's now present 5 Cross Validation approach to tune the hyperparameters and see if we can obtain better results. Indeed, Crss Validation is a statistical method that involves partitioning the data into subsets, training the model on some subsets, and testing it on the remaining subsets. This process is repeated many times, and the results are averaged. This enable us to provide a stronger estimate of the model's performance. Following [2] and [7] we should implement a function with the following structure:

- dataset splitting: divide the training set into 5 folds.
- training and evaluation: For each fold, the model is trained on the rest of the 4 folds and tested on the remaining fold.
- error calculation: the error, measured by MSE,is computed on each fold.

- risk estimation: average of errors across all folds is calculated

I'll use the scikit-learn package to implement this function and I'll use MSE as the error metric.

```
def updatePlotCV(testSize, maxAlpha, dataFrameType):
    #make some var global to use them outside the function
    global X, y, xTest, yTest, bestAlphaMSE, bestAlphaMAE,
    bestAlphaRSquared
    global bestAlphaMSE, bestAlphaMAE, bestAlphaRSquared,
    rSquaredArrayScore, rSquaredArray


    #init params
    alphaArray = np.linspace(0.1, maxAlpha, num=500)
    mseArray = []
    maeArray = []
    rSquaredArray = []
    bestScoreMSE = float('inf')
    bestScoreMAE = float('inf')
    bestScoreRSquared = -float('inf')
    bestAlphaMSE = None
    bestAlphaMAE = None
    bestAlphaRSquared = None
    coefs = []

    #init dataframes
    if dataFrameType == 'numerical':
        outDf = getNumericalDataFrame()
        X, y, xTest, yTest = splitTrainTestSets(outDf, testSize)
    elif dataFrameType == 'categorical':
        outDf = getDummiesDataFrame()
        X, y, xTest, yTest = splitTrainTestSets(outDf, testSize)
    else:
        print("invalid option for dataFrameType")
        return


    #alpha loop
    for idAlpha, alpha in enumerate(alphaArray):

        ridgeRegression = MyCustomModelRidgeRegression(alpha)
        kf = KFold(n_splits=5)
        mseArrayScore = []
        maeArrayScore = []
        rSquaredArrayScore = []
        coefsFold = []
```

```
#fold loop CV
for trainCV, testCV in kf.split(X):
    trainCV.sort()
    testCV.sort()
    XCV, XTestCV = X.iloc[trainCV], X.iloc[testCV]
    yCV, yTestCV = y.iloc[trainCV], y.iloc[testCV]

    #local ridge
    ridgeRegression.fit(XCV, yCV)
    predictionsCV = ridgeRegression.predict(XTestCV)

    #MSE
    mse = np.mean((yTestCV-predictionsCV)**2)
    mseArrayScore.append(mse)

    #MAE
    mae = np.mean(np.abs(yTestCV-predictionsCV))
    maeArrayScore.append(mae)

    #R2
    RSS = np.sum((yTestCV - predictionsCV) ** 2)
    TSS = np.sum((yTestCV - np.mean(yTestCV)) ** 2)
    r2 = 1 - (RSS / TSS)
    rSquaredArrayScore.append(r2)

    #folder coef
    coefsFold.append(ridgeRegression.theta)

meanMse = np.mean(mseArrayScore)
meanMae = np.mean(maeArrayScore)
meanR2 = np.mean(rSquaredArrayScore)
mseArray.append(meanMse)
maeArray.append(meanMae)
rSquaredArray.append(meanR2)

coefsFold = np.array(coefsFold)
coefs.append(np.mean(coefsFold, axis=0))

if meanMse < bestScoreMSE:
    bestScoreMSE = meanMse
    bestAlphaMSE = alpha

if meanMae < bestScoreMAE:
    bestScoreMAE = meanMae
    bestAlphaMAE = alpha
```

```
    if meanR2 > bestScoreRSquared:
        bestScoreRSquared = meanR2
        bestAlphaRSquared = alpha


coefs = np.array(coefs)

fig, axs = plt.subplots(1, 4, figsize=(25, 8))

#coef plot
for i in range(X.shape[1]):
    axs[0].plot(alphaArray, coefs[:, i], label=X.columns[i])
axs[0].set_xscale('log')
axs[0].set_xlabel('alpha')
axs[0].set_ylabel('Coefficient Value')
axs[0].set_title('Ridge coefficients as a
function of the regularization')
axs[0].legend()
axs[0].grid(True)

#construction of the plot
axs[1].plot(alphaArray, mseArray, label='MSE')
axs[1].set_xlabel('alpha')
axs[1].set_ylabel('MSE')
axs[1].set_title('Plot of MSE CV with respect to alpha')
axs[1].legend()
axs[1].grid(True)
mseArray = np.array(mseArray)
print("mean MSE: ", mseArray.mean())
print("best MSE: ", bestScoreMSE)
print("best alpha: ", bestAlphaMSE)

# Plot R-squared
axs[2].plot(alphaArray, rSquaredArray, label='R-squared')
axs[2].set_xlabel('alpha')
axs[2].set_ylabel('R-squared')
axs[2].set_title('Plot of R-squared CV with respect to alpha')
axs[2].legend()
axs[2].grid(True)
rSquaredArray = np.array(rSquaredArray)
print("mean R-squared: ", rSquaredArray.mean())
print("best R-squared: ", bestScoreRSquared)
print("best alpha: ", bestAlphaRSquared)
```

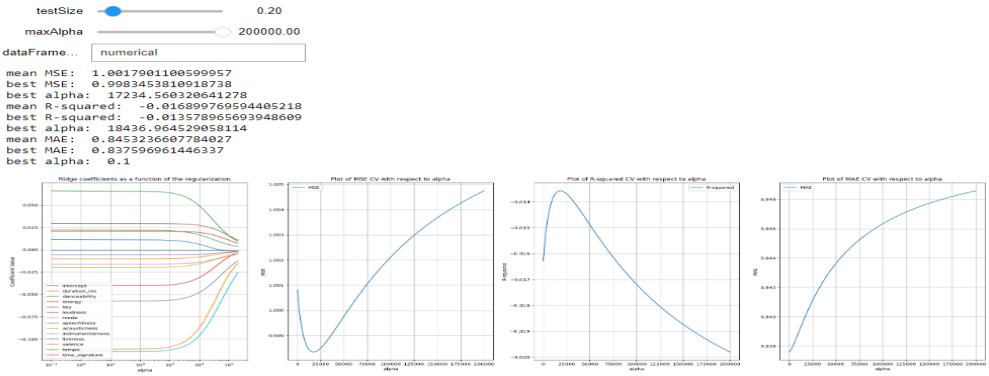
```
    # Plot MAE
    axs[3].plot(alphaArray, maeArray, label='MAE')
    axs[3].set_xlabel('alpha')
    axs[3].set_ylabel('MAE')
    axs[3].set_title('Plot of MAE CV with respect to alpha')
    axs[3].legend()
    axs[3].grid(True)
    maeArray = np.array(maeArray)
    print("mean MAE: ", maeArray.mean())
    print("best MAE: ", bestScoreMAE)
    print("best alpha: ", bestAlphaMAE)

    plt.tight_layout()
    plt.show()
```

The function above calculates the 5 Cross-Validation on the training set and plot the MSE, r-squared and MAE with respect to alpha. Moreover it shows the values of each coefficient in relation to alpha. After running the UpdatePlotCV function, we fit the training data with the alpha corresponding to the best MSE value obtained from UpdatePlotCV. Lastly, we make predictions on the test set and we calculate the risk estimate (MSE error in this case).

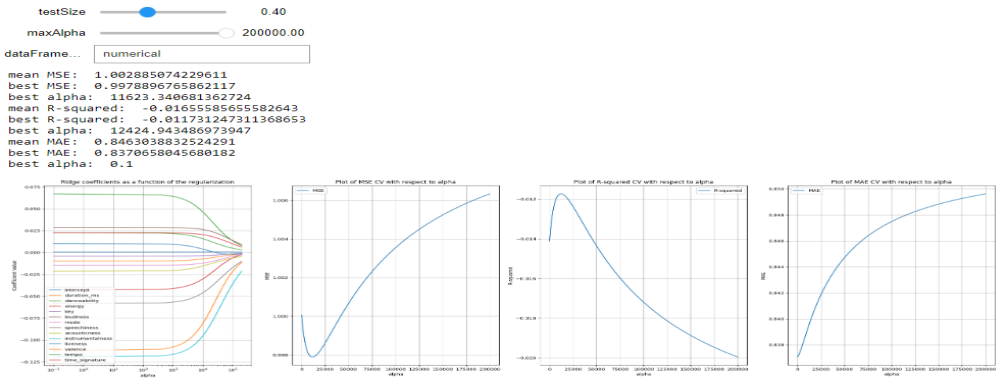
```
print("the best MSE alpha obtained through CV is: ",bestAlphaMSE)
ridgeRegression = MyCustomModelRidgeRegression(bestAlphaMSE)
ridgeRegression.fit(X, y)
predictions = ridgeRegression.predict(xTest)
mse = np.mean((yTest-predictions)**2)
print("the risk estimate is: ",mse)
RSS = np.sum((yTest- predictions) ** 2)
TSS = np.sum((yTest - np.mean(yTest)) ** 2)
print("r-squared is: ",1 - (RSS / TSS))
mae = np.mean(np.abs(yTest-predictions))
print("mae is: ",mae)
```

I ran this code with the same testSize I used without Cross Validation and I noticed different behaviors.

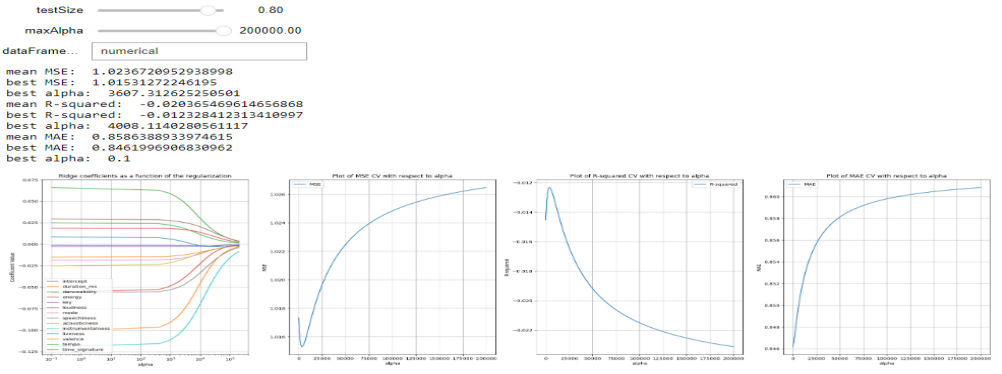
When using Cross-Validation, I observed lower bestAlpha values compared to when both numerical and categorical features are included. Additionally, the risk estimates (in this case MSE error) obtained through Cross Validation are quite similar regardless of whether categorical variables are included, while the r-squared values show more significant differences. For example in Fig. 5 and 6, we can observe differences in the best MSE alpha values when considering only numerical features versus when including both numerical and categorical features. When *testSize=0.2*, the best MSE alpha is 17234 for the numerical case, while it increases to 23246 when including categorical variables. Additionally, the risk estimates vary slightly, with values of 0.97613 and 0.97286



(a) CV experiments with testsize=0.2

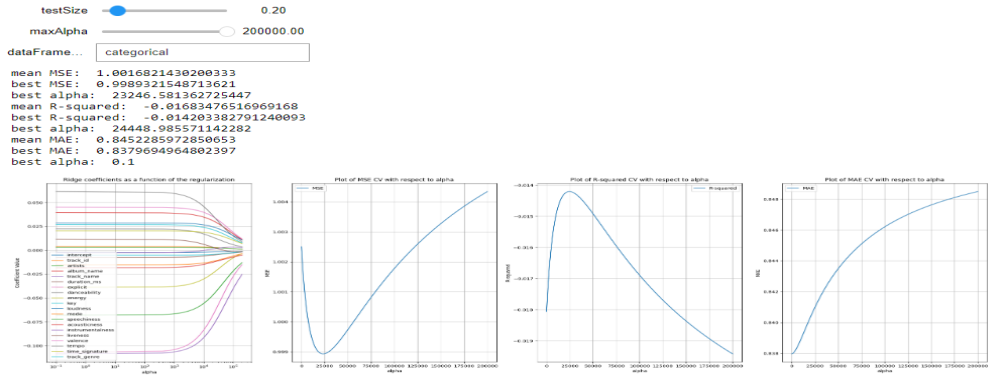


(b) CV experiments with testsize=0.4

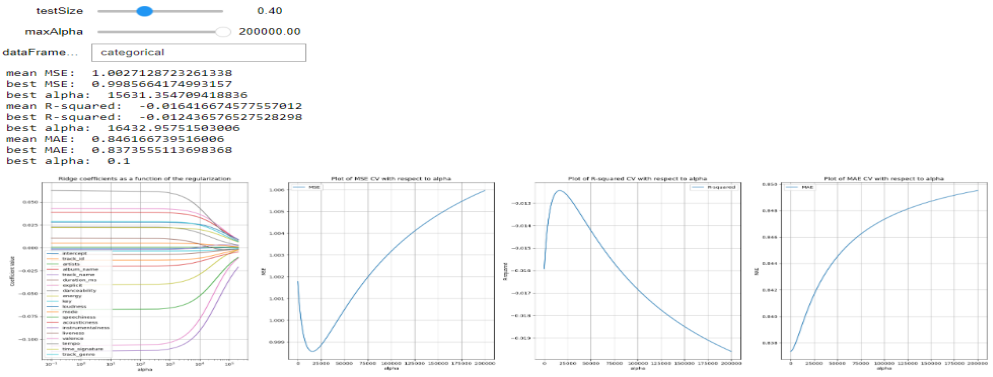


(c) CV experiments with testsize=0.8

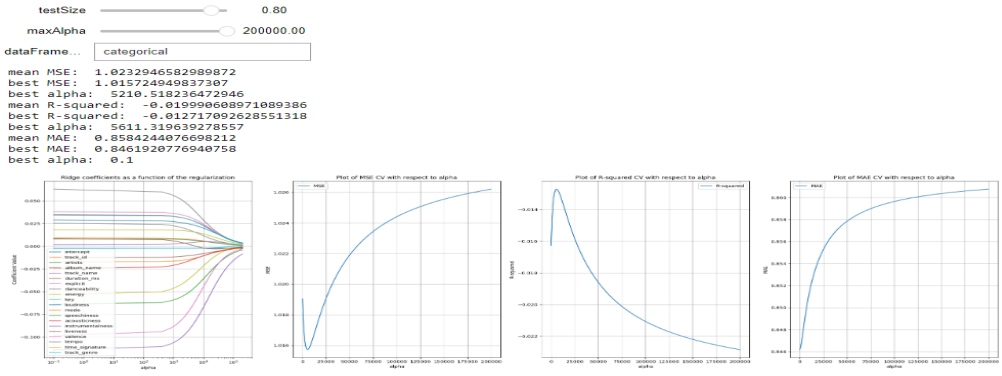
Figure 5: Numerical Experiments with Cross Validation



(a) CV experiments with testsize=0.2



(b) CV experiments with testsize=0.4



(c) CV experiments with testsize=0.8

Figure 6: Categorical Experiments with Cross Validation

respectively, while the corresponding r-squared values are 0.02049 and 0.02380. Similar trends are observed across different test sizes.

It's worth noting that the r-squared values obtained during the evaluation step of Cross Validation are negative. This is expected since we're calculating r-squared with a limited amount of training data in each fold. However, the overall r-squared is comparable to the one obtained without Cross Validation.

Another observation is that the best MSE alpha and best R-squared alpha are close to each other but not identical when using Cross Validation. This discrepancy may arise because MSE and r-squared are obtained through averaging values from different splits instead of just one.

Moreover, the best alphas obtained through Cross Validation are significantly larger than those without Cross Validation. This suggests stronger regularization, which is essential for preventing overfitting, particularly in high-dimensional datasets.

These insights underscore the importance of employing Cross Validation for more robust model evaluation and hyperparameter tuning, especially in scenarios involving complex datasets with multiple features.

I think that all these differences between results obtained with and without Cross Validation depend on various factors. On one hand, using Cross Validation we evaluate the model through different splits; this provides provides a more robust estimate, highlighting a better overall alpha. On the other hand, using one single data split (without Cross Validation), the model may not genarize well. Additionally, including categorical variables increase the size of the data resulting in smaller coefficients and greater regularization. Cross Validation is used to detect overfitting; thus, it suggest larger alpha for better generalization in high dimensions. Finally, Cross Validation provides a more reliable estimate of evaluation metrics, reducing the chance of selecting a suboptimal alpha.

Another detail to pay attention can be found in Fig. 5 and 6. Let's consider the plots regarding the Ridge coefficients as a function of the regularization. This graphs show how the coefficients of the ridge regression model change with the regularization parameter alpha. We can observe that when alpha is small, there is little regularization and the coefficients are larger. Conversely, when alpha is large, the coefficients approach zero. This is consistent with the model because when alpha is large, to keep the term (1) small, we need to penalize the size of the coefficients.

The graph helps us understand which features are more or less important with different levels of regularization. For example, if a feature's coefficient gets smaller as alpha increases, it means that feature has less influence on 'popularity' when more regularization is applied.

Also, depending on the type of experiment chosen, we can see how the significant variables (those that stay away from zero for longer) depend on the test size and whether categorical variables are used. This can be seen by analyzing how the coefficients change with different alpha values.

4 Conclusions

From the experiments on Ridge Regression with and without Cross Validation, we learned several important points.

First of all, Ridge Regression is good at handling complex datasets with lots of variables (both numerical and categorical). It helps prevent overfitting and balances the trade-off between simplicity and accuracy. Secondly, an important role is being done by the dataset size;the size of the dataset, especially the test set, affects the results. A larger test set needs a lower alpha to avoid underfitting, while a smaller test set needs higher alpha to avoid overfitting. Thirdly, including categorical variables affects the results. It leads to slightly different optimal alpha values and makes the model more complex. Finally, Cross Validation is important. Indeed, using Cross Validation gives more reliable estimates of the model's performance and helps find the best parameters. It reduces the risk of picking the wrong alpha value and gives a better understanding of how well the model works. In fact, we observed different optimal alpha values when using or not using Cross Validation; although the alpha and respective MSE appeared better without Cross Validation, these estimates were likely not as accurate and robust. Moreover, it's important to note that these results might vary depending on how we set up the experiments, so we need to be careful when interpreting them. For example, we introduced the parameter *maxAlpha*, which sets the maximum value of the alpha array $alphaArray = np.linspace(0.1, maxAlpha, num = 500)$. However, this also introduces inaccuracy because, for computational reasons, we set num=500. Varying *maxAlpha* increases or decreases the spacing between alpha values, making them more or less precise accordingly.

Despite Ridge Regression and cross-validation being useful, the model's r-squared remained relatively low.

Overall, this highlights the importance of carefully evaluating the model, including handling categorical variables and using techniques like Cross Validation. These practices contribute to building more accurate and reliable models, particularly when dealing with complex datasets.

References

[1] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, *An Introduction to Statistical Learning with Applications in R*, Springer, 2013. `https://www.stat.berkeley.edu/users/rabbee/s154/ISLR_First_Printing.pdf`

[2] Nicolo Cesa-Bianchi, *Cross-Validation and Risk Estimation*, `https://cesa-bianchi.di.unimi.it/MSA/Notes/crossVal.pdf`, 2023

[3] Scikit-learn Developers, *Developer Guide*, `https://scikit-learn.org/stable/developers/develop.html`, 2024

[4] Jupyter Development Team, *ipywidgets: Interactive HTML Widgets*, https://ipywidgets.readthedocs.io/en/stable/reference/ipywidgets.html, 2024

[5] Stack Overflow, *Label encoding across multiple columns with same attributes in scikit-learn*, https://stackoverflow.com/questions/50264334/label-encoding-across-multiple-columns-with-same-attributes-in-sckit-learn, 2024

[6] Pandas, *pandas.core.categorical.Categorical.searchsorted*, https://pandas.pydata.org/pandas-docs/version/0.15/generated/pandas.core.-Categorical.Categorical.searchsorted.html 2024

[7] scikit-learn, *Cross-validation: evaluating estimator performance*, https://scikit-learn.org/stable/modules/cross_validation.html, 2024