Ridge Regression to predict tracks' popularities on Spotify

Fabrizio Cattozzo

July 19, 2023

Chapter 1

Introduction

For my experimental project for the course of Statistical Methods for Machine Learning, I chose to implement the Ridge Regression algorithm to predict the popularity of tracks on the streaming platform Spotify, by using as training data a dataset found on Kaggle.com containing 114k different tracks.
The project required to write the Ridge Regression algorithm from scratch, to train the model only on numerical features and then also with categorical features, to implement 5-fold Cross Validation in order to compute the risk estimates and find the best hyperparameters for both the only numerical trained model and the model trained on all the available data. I decided to write all my code on the Jupyter Notebook and using Python.

Chapter 2

Data description

As said before, the training dataset given contains 114k different Spotify tracks, over a range of 125 different genres, and for each track it provides categorical data such as:

- the *track_id*
- the *artist* name
- the *album_name*
- the *track_name*
- the *track_genre*

as well as numerical data and audio features like:

- *popularity* of the track, a value that goes from 0 (not popular) to 100 (popular)
- *duration_ms* is the lenght of the track in milliseconds
- *explicit* is "true" if the song contains explicit language, "false" otherwise
- *danceability* describe how suitable the track is for dancing, 0.0 is least danceable and 1.0 is most danceable
- *energy* is a value from 0.0 to 1.0 that represents a perceptual measure of intensity and activity
- *key* is the key of the track
- *loudness* of the track expressed in decibels
- *mode* is the modality of the track, 0.0 for minor and 1.0 for major
- *speechiness* represents the amount of spoken words in a track, it's 1.0 if the track is strictly spoken, 0.0 if there are no spoken words in the track
- *acousticness* is 0.0 if the track is not acoustic, 1.0 otherwise
- *instrumentalness* goes from 0.0 to 1.0, the closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content
- *liveness* goes from 0.0 to 1.0, and a value closer to 1.0 represent a strong likelihood that the track has been recorded live
- *valence* is measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track
- *time_signature* is a notational convention to specify how many beats are in each measure and it ranges from 3 to 7 indicating time signatures of 3/4, to 7/4
- *tempo* is the speed or pace, in beat per minute (BPM) of a given piece and derives directly from the average beat duration

Chapter 3

The Ridge Regression algorithm

3.1 Overview

The Ridge Regression algorithm tries to regularize a linear model, this is done by penalizing the presence of larger coefficients. A classic Linear Regression algorithm outputs a predictor $h : \mathbb{R}^d \to \mathbb{R}$, where $\mathbb{R}^d$ is our feature space, of the form

$$h(x) = w^T x$$

where $w \in \mathbb{R}^d$ is a vector of real coefficients.
Given a training set $(x_1, y_1), \dots, (x_m, y_m)$ we find w by minimizing the following cost function:

$$\sum_{t=1}^{m} (w^T x_t - y_t)^2$$

If we call S the $m \times d$ design matrix such that $S^T = [x_1, \dots, x_m]$ we could also write the cost function as

$$||Sw - y||^2$$

where $||x|| = \sqrt{\sum_{t=1}^{m} x_t^2}$.
In Ridge Regression, this cost function get penalized for having larger coefficients w, so the cost function becomes as follows:

$$||Sw - y||^2 + \alpha ||w||^2$$

where $\alpha > 0$ is the regularization parameter. A grater α makes the values of the coefficients increasingly small. In this way, we increase the approximation error (bias) of the model while reducing the variance, thus having a more stable predictor. If we compute the gradient of the previous cost function, we find that it vanishes for

$$w = (\alpha I + S^T S)^{-1} S^T y$$

This is called the closed form formula of Ridge Regression, and gives us the value of w that minimizes the cost function, so the w value that we are searching for.

3.2 Implementation of the algorithm

I chose to implement the Ridge Regression algorithm by using the closed form formula. To do so, I crated a new class **My_Ridge**, receiving as input when initialized only the value of the regularization parameter α.
Inside the class I coded the *fit* method that fits the model by using the training set X and the training labels y provided.
To do so I added a column of ones at the beginning of the training set, this is the *intercept column* which handle the bias (this column gets added only if it's not already present) and then I computed the closed form formula of the Ridge Regression and saved the coefficients w into the parameter

coefficients of the class.
I also coded the *predict* method, that receives as input the test set, adds to it a column of ones in the beginning to have the intercept, and then returns the predicted value by computing $w^T x$.
Lastly I coded two method, *r2score* and *mse_ score*, that compute rapidly the r squared value and the rooted mean squared error of the predicted value with respect to the actual labels.
The r squared value is defined as follows: given $(y_1, \ldots, y_m)$ a set of labels, and given $(\hat{y_1}, \ldots, \hat{y_m})$, their predictions, the R2 r squared value is computed by:

$$R^2 = 1 - \frac{\sum_{t=1}^{m}(y_t - \hat{y_t})^2}{\sum_{t=1}^{m}(y_t - \bar{y_t})^2}$$

where $\bar{y_t}$ is the mean of the labels. This score goes from 0 to 1 and a higher value of R2 means that our predictive model explain well the variation of the target variable around its mean.
The rooted mean squared error is instead defined by the following formula:

$$RMSE = \sqrt{\sum_{t=1}^{m}(\frac{(y_t - \hat{y_t})^2}{m}}$$

where y_t and $\hat{y_t}$ have the same meaning as in the R2 definition.
RMSE is a measure of the mean error made by our predictions.
So these are two useful metrics that give us an idea on the effectiveness of the model. This is the code of the class:

[ ]:
```python
class My_Ridge :
    def __init__(self,alpha) :
        self.alpha = alpha
        self.coefficients = None

    def fit(self, X, y):
        if "intercept" not in X:
            X.insert(0,"intercept",1, True)

        n_feat= X.shape[1]
        I=np.eye(n_feat)
        I[0][0]=0

        A = np.dot(X.T, X) + self.alpha * I
        b = np.dot(X.T, y)
        self.coefficients = np.dot(np.linalg.inv(A),b)

    def predict(self, X):
        if "intercept" not in X:
            X.insert(0,"intercept",1, True)
        return np.dot(X,self.coefficients)

    def r2score(self,Y_pred, Y_act):
        return r2_score(Y_act,Y_pred)

    def mse_score(self,Y_pred, Y_act):
        return mean_squared_error(Y_act,Y_pred,squared=False)
```

Chapter 4

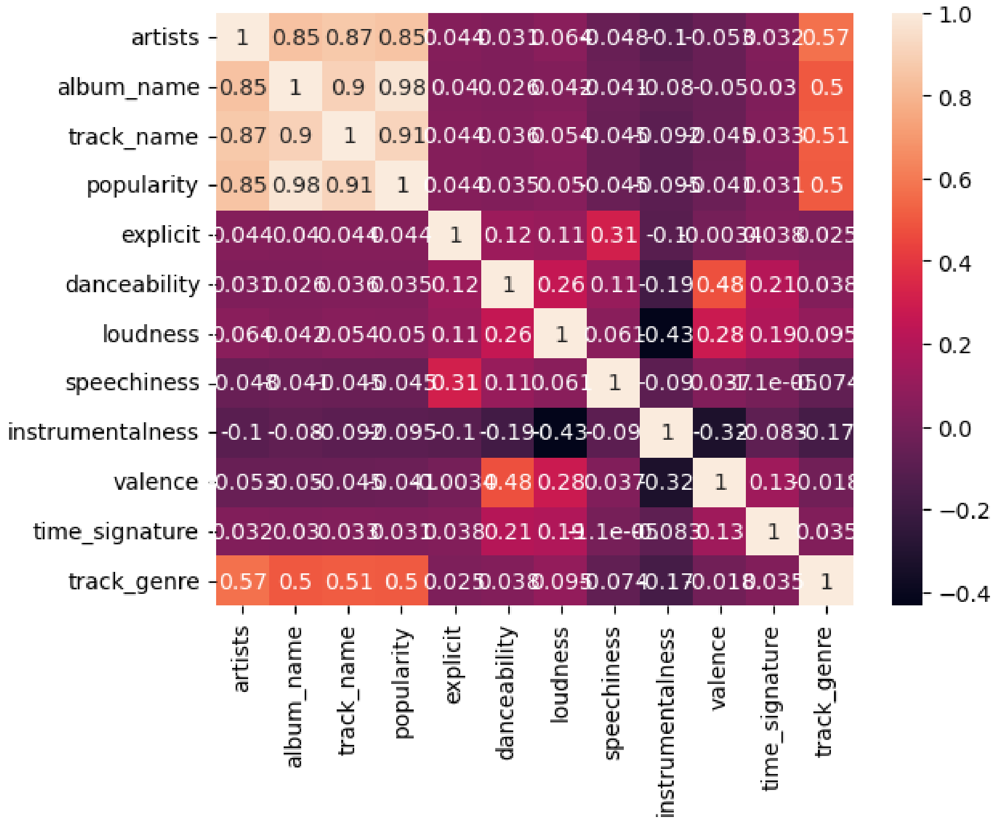
Data Preprocessing

After reading the dataset, I immediately dropped two columns, *Unnamed: 0* and *track_ id*, because they are just indexing columns. Then I checked the correlation between the numerical variables and the target feature *popularity* only and I found the following results:

popularity 1.000000
duration_ms -0.007101
explicit 0.044082
danceability 0.035448
energy 0.001056
key -0.003853
loudness 0.050423
mode -0.013931
speechiness -0.044927
acousticness -0.025472
instrumentalness -0.095139
liveness -0.005387
valence -0.040534
tempo 0.013205
time_signature 0.031073

We see that numerical features don't have high correlations with the *popularity* column, so we can guess that they won't have strong predictive power.
I decided to drop the features that have an absolute value of the correlation with *popularity* <= 0.02. This caused the discarding of the following features: *acousticness, tempo, duration_ ms, energy, key, mode, liveness.*

I then had to deal with the categorical features of the dataset. To encode them I opted for Target Encoding, because other encoding methods like One-Hot Encoding (OHE) were not really suitable for this dataset due to the fact that each categorical feature had a massive amount of different possible values, so by using OHE I created a really big dataset that was difficult to handle and store.
Target encoding on the other hand seemed like a better option, because it replaces each categorical column with only one numerical column. To do Target Encoding i used the encoder *TargetEncoder* from the library *category_ encoders.* The way this encoder works, is that it replaces each feature with a blend of the expected value of the target given particular categorical value and the expected value of the target over all the training data, this blending is controlled by a parameter *smoothing* that I decided to set to 300 after testing various different values, I will explain the reasoning behind this choice later during the report. An higher smoothing means that the replacing value is going to be more similar to the expected value of the target over all the training data.
I initially used Target Encoding with all the categorical features present in the dataset in order to then check the correlation between these encoded variables and the *popularity* values. What I got is the following heatmap:



I noticed that the *artists, track_ name, album_ name* features were all really highly correlated to each other, so I decided to drop *track_ name* and *album_ name.* I decided to maintain *artists* only, because I thought that when a new track gets published, the most determining factor out of these three in deciding whether or not the song is going to be popular it's the artist name, due to the fact that tracks from artist who already had popular songs will more likely be listened by the users, whereas if a new song gets published with the same *track_ name* or the same *album_ name* of a popular song, but different *artists*, it's not as likely that it will become popular. So, even though training with also *album_ name* will lead to better results on this dataset, due to the really high correlation between *album_ name* and *popularity* of 0.98, I decided to delete these two columns in order to not create wrong biases in the model when working with brand new data.
The last things I did during preprocessing was to save the *popularity* feature column in a label vector and then I normalized the training dataset by subtracting its mean and dividing by its standard deviation, while the normalization of the test dataset was done by subtracting the training dataset mean and dividing by the standard deviation of the training dataset.

Chapter 5

Training with numerical features only

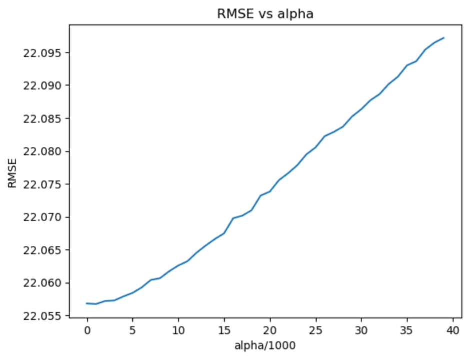
The first kind of training that I tried has been only on the features that were numerical from the start. To do so, I created the *spotify_num* dataset by dropping from the training data also the *artists*, *track_name* and *album_name* features, then I implemented the code to compute the 5-fold Cross Validation and repeated the cross validation for values of the parameter α ranging from 0 to 40000, with steps of 1000. The code for this is the following:

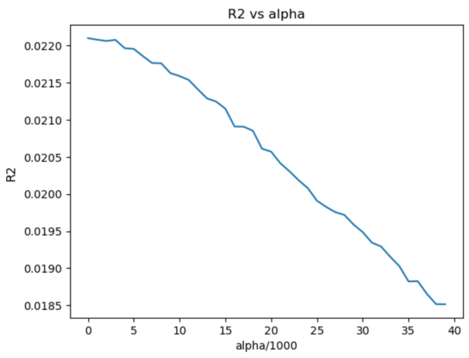
```
[]: k_Fold=KFold(n_splits=5,shuffle=True)
```

```
[]: i=0
     for alphas in range(0,40000,1000): #I cycle on the alpha to find the one that␣
      ↪gives us the best scores
         average_rscore=0.0
         average_msescore=0.0
         ridge = My_Ridge(alpha=alphas)
         for train_index, test_index in k_Fold.split(spotify): #cross validation
             train,test = (spotify_num.T[train_index]).T, (spotify_num.
      ↪T[test_index]).T

             train=train.dropna(axis=0)
             test=test.dropna(axis=0)

             y = train["popularity"].copy() #I save the labels vector
             yt=(np.array([y])).T
             yt=yt.astype(np.int64)

             X = (train.drop(columns= 'popularity'))
             X_test=(test.drop(columns= 'popularity'))

             x_mean=X.mean() #normalizing training and testing dataset
             x_std=X.std()
             X = (X - x_mean) / x_std
             X_test=(X_test-x_mean)/x_std
             X=X.astype(np.float64)
             X_test=X_test.astype(np.float64)

             ridge.fit(X,yt)
             predictions=ridge.predict(X_test)

             expect= np.array(test["popularity"]) #actual testing popularity values
```

```
        expect = np.reshape(expect,(-1,1))

        rscore=ridge.r2score(predictions,expect) #r2 score of the prediction
        msescore=ridge.mse_score(predictions,expect) #rmse score of the␣
↪prediction
        average_rscore += rscore
        average_msescore += msescore
    av[i]=average_msescore/5 #computing the average of the r2 and rmse scores␣
↪on the 5 folds
    avr[i]=average_rscore/5
    i=i+1
    print(alphas)
```

The resulting rmse and r squared variations are as follows:





We see that the best α (i.e. the one that is closest to minimizing the RMSE and maximizing the R2 scores at the same time) for this model is of 1000, and it provides an $RMSE = 22.05672747$ and a $R2 = 0.02208097$.
These are really bad, a R2 score so low means that our model isn't capable of explaining the variability observed in the target variable *popularity*. This is due to the fact that, as we saw in the beginning, the numerical features had very low correlations with the *probability* feature, so their predictive power is very low and a model trained on them tends to always predict a *popularity* value really close to the mean of the target variable in the training data.

Chapter 6

Training with both numerical and categorical features

I now trained my model on the whole preprocessed dataset described in Chapter **4**. To do so, i followed a really similar method as the one i used when training only on numerical data: I computed 5-Fold Cross Validation for each value of α in the 0 to 40000 range, with steps of 1000, and plotted the variations of the rmse and r2 scores.
The code I implemented to do so is the following:

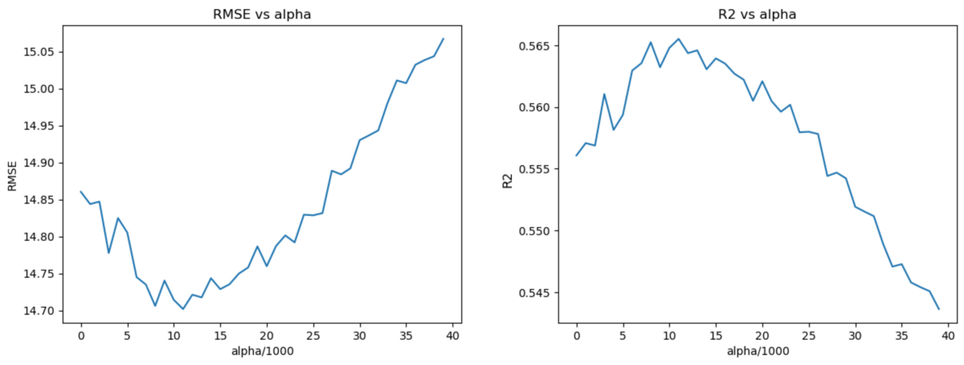
```
[ ]: encoder = TargetEncoder(smoothing=300)
     cat_cols =['artists','explicit','track_genre']
```

```
[ ]: k_Fold=KFold(n_splits=5,shuffle=True)
```

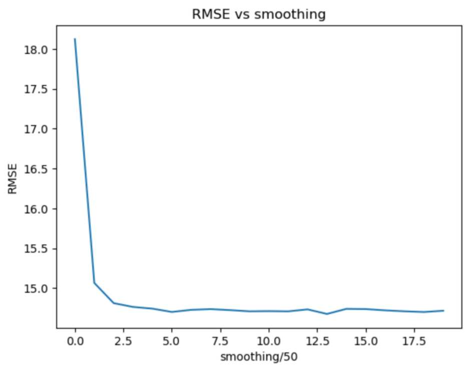
```
[ ]: i=0
     for alphas in range(0,40000,1000): #I cycle on the alpha to find the one that␣
      ↪gives us the best scores
         average_rscore=0.0
         average_msescore=0.0

         ridge = My_Ridge(alpha=alphas)
         for train_index, test_index in k_Fold.split(spotify): #cross validation
             train,test = (spotify.T[train_index]).T, (spotify.T[test_index]).T

             train[cat_cols] = encoder.fit_transform(train[cat_cols],␣
      ↪train["popularity"]) #encoding of the categorical features
             test[cat_cols] = encoder.transform(test[cat_cols])
             train=train.dropna(axis=0)
             test=test.dropna(axis=0)

             y = train["popularity"].copy()
             yt=(np.array([y])).T
             yt=yt.astype(np.int64)

             X = (train.drop(columns= 'popularity'))
             X_test=(test.drop(columns= 'popularity'))

             x_mean=X.mean() #normalizing training and testing data
             x_std=X.std()
             X = (X - x_mean) / x_std
             X_test=(X_test-x_mean)/x_std
```

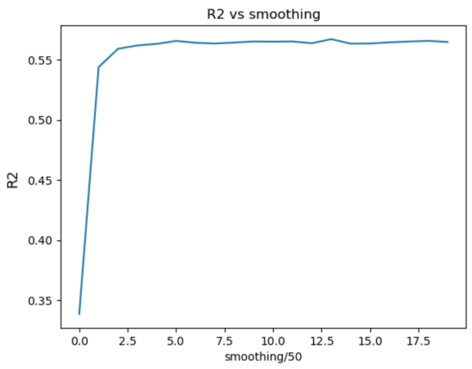
```
        X=X.astype(np.float64)
        X_test=X_test.astype(np.float64)

        ridge.fit(X,yt)
        predictions=ridge.predict(X_test)

        expect= np.array(test["popularity"]) #actual training labels
        expect = np.reshape(expect,(-1,1))

        rscore=ridge.r2score(predictions,expect) #r2 score of the prediction
        msescore=ridge.mse_score(predictions,expect) #rmse score of the␣
↪prediction
        average_rscore += rscore
        average_msescore += msescore
    av[i]=average_msescore/5 #computing the average of the r2 and rmse scores␣
↪over the 5 folds
    avr[i]=average_rscore/5
    i=i+1
    print(alphas)
```

By doing so I obtained the following plots:



I tried repeating this code multiple times and all of them the optimal α (i.e. the one that is closest to minimizing the RSME and maximizing the R2 scores at the same time) was near the value of 12000, so I decided to use the value of $\alpha = 12000$ as the best possible choice for the α hyperparameter.
In the case displayed in the graphs, this value of α provided a $RMSE = 14.721176$ and a $R2 = 0.5643638$. Both scores are significantly better than the ones obtained by training the model using only numerical features. This is intuitive, because categorical features showed a significantly higher correlation to the *popularity* feature than the numerical ones, so by training also on these features, the model gained much more predictive power.
Finally, to also set an optimal value for the hyperparameter *smoothing*, i repeated the previous code, but this time i set the value of α to 1200, and performed a 5-fold Cross Validation for every possible value of *smoothing* inside the range of 0 to 1000, with steps of 50, so that for every cross validation I performed, the encoding of the categorical features was done with a different value of *smoothing*. This is what I obtained:





We see that from the value of $smoothing = 150$ on, the performances of the model don't increase. So I decided to pick an optimal value of $smoothing = 300$ to not blend too much the expected value of the target given particular categorical value and the expected value of the target over all the training data.

Chapter 7

Conclusion

To conclude, I obtained that when training on only the numerical features of the dataset, the best hyperparameter setting is $\alpha = 1000$ and the related scores that I obtained after a 5-fold Cross Validation are $RMSE = 22.05672747$ and a $R2 = 0.02208097$.
On the other hand, when training with both categorical and numerical variables, the best hyperparameters setting is $\alpha = 12000$ and $smoothing = 300$. With this choice of hyperparameters, the model obtains the following scores: $RMSE = 14.721176$ and a $R2 = 0.5643638$.
I think that the results I got when training only with numerical features were not satisfying, primarly for the very low R2 that means that the predictions didn't really reflect the variations of the target feature around its mean value.
On the other hand, when training with also categorical features, the results become more satisfying, because a RMSE of 14.7 is not bad considering the fact that we are doing a prediction on a target feature that ranges from 0 to 100. Moreover, the much higher R2 value of 0.564, shows that the model explain more than half of the variability of the target variable.
Such a better performance of the model when trained also on the categorical features is explained by the low correlation between the numerical features and the target variable *popularity*, and the high correlation instead of the categorical features with *popularity*.

Disclaimer