UNIVERSITA' DEGLI STUDI DI MILANO

DATA SCIENCE AND ECONOMICS (DSE)

Module Machine Learning

Prof. Nicolò Cesa Bianchi

Experimental Project

Stefano Zanlucchi Matr.No. 22288A

Accademic Year 2022-2023

Contents

List of Figures

List of Tables

Abstract

In this project of the Module of Machine Learning the scope is to present both ridge regression and kernel ridge regression from scratch and to compare the developed algorithm with the already build-in libraries in order to understand if they can be reasonable in terms of outputs, running time and memory workload.
The dataset used is the Spotify Tracks Dataset and in addition I carry out cross validation technique and investigate the tuning of the hyperparameters.
I really enjoyed making a good effort to deepen these topics, because it lets understand how we can face problems thanks to a strong theory background and at then end we can solve them with coding and reasoning skills.

Key words: Python, Ridge Regression, Kernel Ridge Regression, Cross Validation, Latex.

CHAPTER 1

Experimental project

This experimental project, that is developed in Python [3], is structured in 4 main sections:

1. Dataset preparation: the dataset is manipulated in order to have 2 datasets for the analysis
2. Ridge Regression: it includes its implementation adding cross validation and hyper parameter tuning
3. Kernel Ridge Regression: it explores how the Ridge regression can be kernalized
4. Conclusion: it summarizes the project outcomes

1.1 Dataset preparation

The dataset used for in this project is the Spotify Tracks Dataset [2] available from *Kaggle* which contains more than 100.000 songs with different features both numerical and categorical. The purpose of this project is to predict the variable popularity through the others attributes and in fig.1.1 the types of features are presented and also the number of their unique values.

However before developing from scratch the algorithms for this project, in this first section parts I would like to focus on some relevant steps that need to be done on the dataset:

- data cleaning
- preparation of *dataset 1* that considers only numerical features
- preparation of *dataset 2* which includes both numerical and categorical features

```
track_id            object
artists             object
album_name          object
track_name          object
popularity           int64
duration_ms          int64
explicit             int64
danceability       float64
energy             float64
key                  int64
loudness           float64
mode                 int64
speechiness        float64
acousticness       float64
instrumentalness   float64
liveness           float64
valence            float64
tempo              float64
time_signature       int64
track_genre         object
```

(a) Types of features

```
No.of.unique values in each column :
 track_id            89741
artists              31437
album_name           46589
track_name           73608
popularity             101
duration_ms          50697
explicit                 2
danceability          1174
energy                2083
key                     12
loudness             19480
mode                     2
speechiness           1489
acousticness          5061
instrumentalness      5346
liveness              1722
valence               1790
tempo                45653
time_signature           5
track_genre            114
dtype: int64
```

(b) Unique values for each feature

Figure 1.1: Clarification about data types and occurrence of unique values

1.1.1 Data cleaning

In fig.1.1b if we look to the `track_id` feature we can clearly see that there are repetitions of same songs because unique values are less that the total number of the rows in the dataset. Therefore the first step is to analyze the root causes of these duplicates and get rid of them:

	track_id	track_genre	artists	album_name	track_name	popularity	duration_ms	explicit	danceability	energy	key
15028	001APMDOI3qtx1526T11n1	chill	Pink Sweat$,Kirby	New RnB	Better	0	176320	0	0.613	0.471	1
103211	001APMDOI3qtx1526T11n1	soul	Pink Sweat$,Kirby	New RnB	Better	0	176320	0	0.613	0.471	1
85578	001YQlnDSduXd5LgBd66gT	punk-rock	Soda Stereo	Soda Stereo (Remastered)	El Tiempo Es Dinero - Remasterizado 2007	38	177266	0	0.554	0.921	2
100420	001YQlnDSduXd5LgBd66gT	ska	Soda Stereo	Soda Stereo (Remastered)	El Tiempo Es Dinero - Remasterizado 2007	38	177266	0	0.554	0.921	2
91801	003vvx7Niy0yvhvHt4a68B	rock	The Killers	Hot Fuss	Mr. Brightside	86	222973	0	0.352	0.911	1
...	...	...	...	...	...	...	...	...	...	...	...
72679	7zv2vmZq8OjS54BxFzI2wM	metalcore	Attila	Soundtrack to a Party (Bonus)	Lets Start the Party	25	125859	1	0.592	0.932	1
22326	7zv2vmZq8OjS54BxFzI2wM	death-metal	Attila	Soundtrack to a Party (Bonus)	Lets Start the Party	25	125859	1	0.592	0.932	1
2004	7zwn1eykZtZ5LODrf7c0tS	alt-rock	The Neighbourhood	Hard To Imagine The Neighbourhood Ever Changing	You Get Me So High	83	153000	0	0.551	0.881	7
3100	7zwn1eykZtZ5LODrf7c0tS	alternative	The Neighbourhood	Hard To Imagine The Neighbourhood Ever Changing	You Get Me So High	83	153000	0	0.551	0.881	7
91401	7zwn1eykZtZ5LODrf7c0tS	rock	The Neighbourhood	Hard To Imagine The Neighbourhood Ever Changing	You Get Me So High	83	153000	0	0.551	0.881	7

(a) Duplicated tracks with different genre

track_id	popularity
0000vdREvCVMxbQTkS888c	44
000CC8EParg64OmTxVnZ0p	47
000lz0K615UepwSJ5z2RE5	22
000RDCYioLteXcutOjeweY	62
000qpdoc97IMTBvF8gwcpy	19
...	...
7zxHiMmVLt4LGWpOMqOpUh	56
7zxpdh3EqMq2JCkOl0EqcG	23
7zyYmIdjqqiX6kLryb7QBx	61
7zybSU9tFO9HNlwmGF7stc	54
7zz7iNGlWhmfFE7zlXkMma	22

89741 rows × 1 columns

(b) Duplicated tracks with different popularity score

Figure 1.2: Sources of duplicates

1. `track_genre`: each song is duplicated if it belongs to more than one genre as presented in fig.1.2a. The strategy is to keep just one row for each song and we can do that applying one hot encoding that creates a new column for each different genre, this methods will be as also discussed in paragraph 1.1.3

2. `popularity`: this variable shows duplicates of same songs differing just for a small change of the popularity value as visible in fig.1.2b, therefore we decide just to keep one row for each unique song with the average of this score

Thanks to these transformations of the dataset we get only unique songs in rows and their features are listed in column.

1.1.2 Dataset 1: numerical features

Now we would keep only the numerical features, it means floats or integers. For the sake of simplicity we transform integer variables into float and then we apply a filter on the types of the feature keeping just floating numbers. The obtained *dataset 1* has following dimension (89741, 10): only 10 features, 9 of them will be used as independent variables, 1 as dependent that we would like to predict.

In this project we are not interested understanding distribution of numerical variables,

track_id	danceability	energy	loudness	speechiness	acousticness	instrumentalness	liveness	valence	tempo	popularity
5SuOikwiRyPMVoIQDJUgSV	0.676	0.4610	-6.746	0.1430	0.0322	0.000001	0.3580	0.7150	87.917	73.0
4qPNDBW1i3p13qLCt0Ki3A	0.420	0.1660	-17.235	0.0763	0.9240	0.000006	0.1010	0.2670	77.489	55.0
1iJBSr7s7jYXzM8EGcbK5b	0.438	0.3590	-9.734	0.0557	0.2100	0.000000	0.1170	0.1200	76.332	57.0
6lfxq3CG4xtTiEg7opyCyx	0.266	0.0596	-18.515	0.0363	0.9050	0.000071	0.1320	0.1430	181.740	71.0
5vjLSffimiIP26QG5WcN2K	0.618	0.4430	-9.681	0.0526	0.4690	0.000000	0.0829	0.1670	119.949	82.0
...	...	...	...	...	...	...	...	...	...	...
2C3TZjDRiAzdyViavDJ217	0.172	0.2350	-16.393	0.0422	0.6400	0.928000	0.0863	0.0339	125.995	21.0
1hIz5L4lB9hN3WRYPOCGPw	0.174	0.1170	-18.318	0.0401	0.9940	0.976000	0.1050	0.0350	85.239	22.0
6x8ZfSoqDjuNa5SVP5QjvX	0.629	0.3290	-10.895	0.0420	0.8670	0.000000	0.0839	0.7430	132.378	22.0
2e6sXL2bYv4bSz6VTdnfLs	0.587	0.5060	-10.889	0.0297	0.3810	0.000000	0.2700	0.4130	135.960	41.0
2hETkH7cOfqmz3LqZDHZf5	0.526	0.4870	-10.204	0.0725	0.6810	0.000000	0.0893	0.7080	79.198	22.0

89741 rows × 10 columns

Figure 1.3: Dataset 1 including only numerical features

e.g. whether skewness is present or not, and even not about their correlation with the target variable `popularity` as was done instead in the Statistical learning module. However we can perform a quick analysis looking at the correlation matrix and we can recognize that there is very low correlation of the features with the variable `popularity` as shown in fig.1.4. In this experimental project instead we will focus mostly on the development of the learning algorithms from scratch.

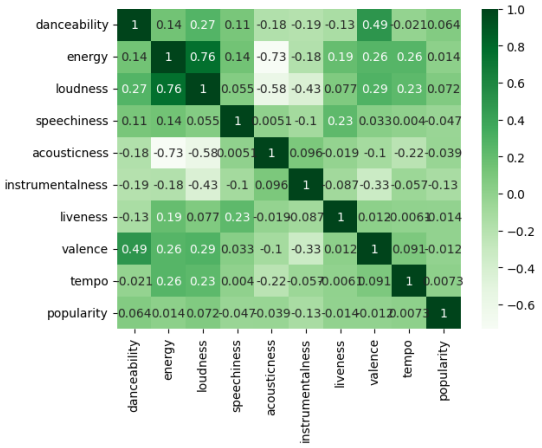

Figure 1.4: Dataset 1: correlation matrix

1.1.3 Dataset 2: numerical and categorical features

Firstly we can start looking at categorical feature exttttrack_genre. It is clear that the genre is a nominal attribute therefore we can apply the one hot encoding technique to rearrange the genre by columns (each column identify a genre). Grouping by `track_id` we can get unique values of songs in rows as presented in fig.1.5.

The second step of this paragraph is to check and analyze the remaining categorical

	track_id	acoustic	afrobeat	alt-rock	alternative	ambient	anime	black-metal	bluegrass	blues	...	spanish	study	swedish	synth-pop	tango	techno	trance	trip-hop	turkish	world-music
0	0000vdREvCVMxbQTkS888c	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	000CC8EParg64OmTxVnZ0p	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	000lz0K615UepwSJ5z2RE5	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	000RDCYioLteXcutOjeweY	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	000qpdoc97IMTBvF8gwcpy	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
89736	7zxHiMmVLI4LGWpOMqOpUh	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
89737	7zxpdh3EqMq2JCkOI0EqcG	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
89738	7zyYmldjqqiX6kLryb7QBx	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
89739	7zybSU9tFO9HNlwmGF7stc	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
89740	7zz7iNGlWhmfFE7zlXkMma	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

89741 rows × 115 columns

Figure 1.5: One hot encoding done on `track_genre` and aggregated to unique `track_id`

variables to understand if they can be considered nominal or ordinal. Fig.1.6 shows the list of the categorical features:

- `time_signature` and `key` need to be handled also as nominal attributes since they refer respectively to musical keys (usually identified with letters A, B, etc.) and to time (better expressed with values as 3/4, 4/4, etc.), so we use one hot encoding
- instead the other two variables `explicit` and `mode` can be considered similar to a boolean variable, so no transformation is needed at all

```
Feature -> explicit: [0 1]
Feature -> mode: [0 1]
Feature -> key: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
Feature -> time_signature: [0, 1, 3, 4, 5]
```

Figure 1.6: Categorical variables in addition to already analyzed `track_genre`

The full *dataset 2* has new dimensions (89741, 145) and the detailed explanations of these previous steps can be also see directly in Jupyter project [3].

1.1.4 Training and test set

The last steps of this first part of data manipulation are:

- to create *dataset 1** using the rescaling technique `StandardScaler` on the numerical variables in order to reduce the influence of features with large values
- to divide the dataset into two parts: 3/4 training set and 1/4 test set

One last remark about section 1.3 in which we are further simplifying the datasets due to memory issue because the kernel matrix becomes expensive to be stored in memory

1.2 Ridge regression

1.2.1 Brief theory introduction

Using the background of this course [1] as basis, the coefficient of a linear regression can be obtained with the normal equation in vector and matrix notation, that is:

$$\hat{w} = (S^T S)^{-1} S^T y \tag{1.1}$$

where S is the design matrix $m \times d$, which contains the training data points in rows $x_1, ..., x_m \in \mathbb{R}^d$ and y is the vector collecting the label of the training set $y_t \in \mathbb{R}$.
However when $S^T S$ is nearly singular the vector $\hat{w}$ becomes sensitive to any small perturbations of the training set, that leads to an increase in the variance error of the prediction. We can develop a more robust predictor that is called Rigde Regression, just adding a regularization term that increases the bias error. The previous equation now becomes:

$$\hat{w_{S,\alpha}} = (\alpha I + S^T S)^{-1} S^T y \tag{1.2}$$

This formula 1.2 is the one at that need to be implemented in this first part of the project.

1.2.2 Coding part

The purpose it to implement from scratch the algorithm for the Ridge Regression and then evaluated it to onto the two datasets. To do that we prepare just few functions presented in fig.1.7:

```python
def ridge_regression(X, y, regu):
  # add first column for intercept in the S matrix
  S = np.concatenate([np.ones((len(X), 1)), X], axis=1)
  # Apply ridge regression formula (alpha*I+S.T S)^-1 S.T y
  w_s = (np.linalg.inv(regu*np.identity(S.shape[1])+S.T@S)@S.T)@y
  return w_s
```

```python
def predict(XX, beta):
  # get the prediction given the coefficients
  XX = np.concatenate([np.ones((len(XX), 1)), XX], axis=1)
  y_pred = XX @ beta
  return y_pred
```

```python
def calculate_metrics(y_pred,y):
  # calculate Root Mean Squared Errore
  rmse = math.sqrt(((y - y_pred)** 2).mean())
  # calculate R^2
  u = ((y - y_pred)** 2).sum()
  y = ((y - y.mean()) ** 2).sum()
  R_squared = 1 - u/y
  return rmse, R_squared
```

Figure 1.7: Ridge regressions functions developed in Python

- `ridge_regression`: to compute the coefficients of the predictor
- `predict`: to perform the prediction
- `calculate_metrics`: to calculate the metrics R^2 and Root Mean Squared Error

Fixing the value of α we can check how this algorithm behaves in comparison to the same one from `scikit-learn` package and so we can evaluate its correctness. The results obtained with this analysis are summarized in table 1.1, from which it is evident that this

algorithm is very closed (or even exactly) to the baseline from scikit-learn, thus it can be used for our next steps.

Results of *dataset 1 and 1** do not differ from each other, so the `StandardScaler` is

Source	Dataset	R^2 train	R^2 test	RMSE train	RMSE test
scikit-learn	1	0.0299	0.0317	-	-
scikit-learn	1*	0.0299	0.0317	-	-
scratch	1*	0.0299	0.0317	20.4192	20.4192
scikit-learn	2	0.2970	0.2954	-	-
scratch	2	0.2970	0.2954	17.1999	17.4186

Table 1.1: Ridge regression results overview with $\alpha = 0.5$, different datasets and different source of the algorithm

not much useful in this case. It is clear that the R^2 values of the regressions are very low especially on *dataset 1 and 1** and this could be assumed from the correlation matrix of fig.1.4. Also searching on the internet we can get the confirmation of very poor outcome when performing prediction on this dataset with simple algorithms as ours. We can then notice that regressions on *dataset 2* have higher R^2 values because the obtained models have a better explanation of the variance of the dataset variance itself thanks to the added categorical variable (e.g. `track_genre`).

1.2.3 Cross validation

In order to avoid overestimation of the risk of a learning algorithm, we can use a technique called K fold (external) cross validation. In case the learning algorithm has also hyper-parameters, as in our case, we assume α as fixed, and thus we can follow these general Cross Validation (CV) steps:

1. dividing the training set S into K subset $D_1, ..., D_k$ of size m/K
2. considering the training part $S^{(i)} = S \backslash D_i$ and the validation part D_i
3. looping for $i = 1, ...K$ and obtain h_i predictors computed on the training part and then calculate the errors on each validation part
4. averaging the estimate of these errors $l_S^{CV}(A) = \frac{1}{K} \sum_{i=1}^{K} l_{D_i}(h_i)$

The algorithm of CV can be implemented in this project focusing on two main parts and functions as shown in fig.1.8 and 1.9 :

- `create_split`: used to create sets of indexes according to the K-Folds partitioning
- `K_fold_cv_ridge`: is the main loop that processes the different sets, computes predictions and the error, and lastly averages the error

Since from previous section the *dataset 1* is very poor for the prediction, we can just take *dataset 2* to perform this step.
We can try two solutions to perform CV:

```
def create_splits(X,n_splits):
  # function that perform n_splits of the indexes given in X
  # check the number of splits and samples
  n_samples = X.shape[0]
  if n_splits > n_samples:
    sys.exit("Number of splits cannot be higher than number of samples")
  # set up
  fold_size = n_samples//n_splits # size of fold, used floor divison
  k_index = np.arange(n_samples) #assign an index to the samples
  fold_train =[]
  fold_test = []
  fold = []
  # for loop to create splits
  for j in range(n_splits):
    test_indices = k_index[j*fold_size:(j+1)*(fold_size)] #index of test set
    train_indices_pre = k_index[:j*fold_size] # indices pre test indices
    train_indices_post = k_index[(j+1)*(fold_size):] # indices post test indices
    train_indices = np.concatenate((train_indices_pre, train_indices_post)) # concatenate
    # store training and test indices
    fold_train.append(train_indices)
    fold_test.append(test_indices)
    fold.append((train_indices,test_indices)) # fold indices
  return fold
```

Figure 1.8: Code to get indices for K-fold partitioning

```
def K_fold_cv_ridge(kf_split,X,y,alpha,n_split):
  # function to perform cross validation on ridge regression
  # starting from splits obtained with K fold partitioning
  # create score matrix to store metrics ()
  score_cv_tr = np.zeros((n_split,2))
  score_cv_ts = np.zeros((n_split,2))
  i = 0
  # perform Cross Validation on K_fold splitted indexes
  for train_index, test_index in kf_split:
    # print("TRAIN:", train_index, "TEST:", test_index)
    X_train_cv = X.iloc[train_index]
    X_test_cv = X.iloc[test_index]
    y_train_cv = y[train_index]
    y_test_cv = y[test_index]
    # perform ridge regression on the K-fold training set
    ww_cv =ridge_regression(X_train_cv, y_train_cv, alpha)
    # get the prediction on the training set and testing set
    y_train_pred = predict(X_train_cv,ww_cv)
    y_test_pred = predict(X_test_cv,ww_cv)
    # # calculate metrics of training and testing error
    met_train_cv = calculate_metrics(y_train_pred,y_train_cv)
    met_test_cv = calculate_metrics(y_test_pred,y_test_cv)
    # save the result of the metrics
    score_cv_tr[i,:] = met_train_cv
    score_cv_ts[i,:] = met_test_cv
    i +=1
  # return score_cv
  return score_cv_tr, score_cv_ts
```

Figure 1.9: Code of CV with Ridge Regression algorithm

- scratch 1: it is a hybrid algorithm, in deed it uses K-fold indexes created through `scikit-learn` function, but applies the own code `K_fold_cv_ridge` from scratch
- scratch 2: it is full developed from scratch using both functions `create_split` and `K_fold_cv_ridge`

Looking at table 1.2 it is visible that also in this case the results are exactly the same to the ones obtained from `scikit-learn` package.

In this way we can assess the performance in a more accurate way instead of just using a

Source	Dataset	R^2 with cv = 5	
scikit-learn	2	0.2940	-
scratch 1	2	0.2940	17.2363
scratch 2	2	0.2940	17.2363

Table 1.2: Results of cross validation implementation

random split of training and testing data that could lead us to overestimation of the risk.

1.2.4 Hyperparameter tuning

As shown in equation 1.2 the regularization term α is the hyperparameter of the Ridge Regression, and in this project we consider a finite set of 50 values of α that are spaced evenly on a logspace.

From fig.1.10 we can understand how the coefficients of the estimators on both datasets

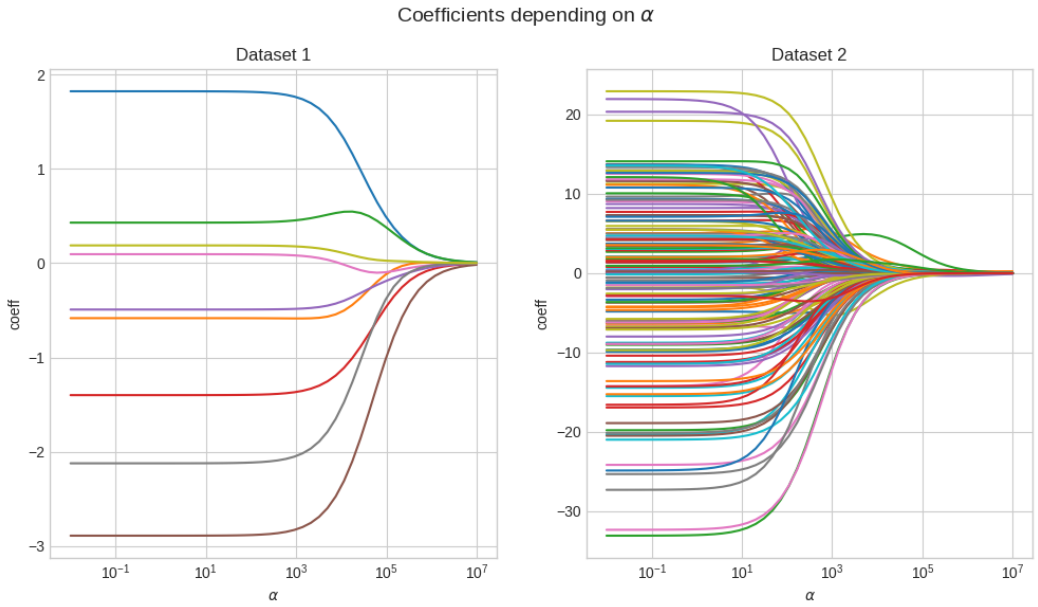

Figure 1.10: Coefficients depending on α for both datasets

depends on α, when α is closed to 0 we are reaching the linear regression estimator, when α is big instead all the coefficients shrink towards 0, meaning towards the null hypothesis. In this section we would like to find the optimal value of α that minimizes the risk estimate computed on the training data. To do that we can follow these steps:

- we set as usual the baseline using `GridSearchCV` technique available from `scikit-learn` package on both datasets *1** and *2*

- we can search for the optimal parameter α_{opt} using a for loop that explores every value of α and recalls the function `K_fold_cv_ridge` to store the scores of RMSE and R^2 of training and validation. This short part of code is shown in fig.1.11.

```
# lists to store scores with different hyper parameters
m_cv_hyp1_tr = np.zeros((n_alphas,2))
m_cv_hyp1_ts = np.zeros((n_alphas,2))
i=0
for a in alphas:
  # dataset 1 with a = alpha[i] and cv = 5
  score1_cv_hyp_tr,score1_cv_hyp_ts = K_fold_cv_ridge(Xk1,X1_train,y1_train,a,n_s)
  m_cv_hyp1_tr[i,:] = np.mean(score1_cv_hyp_tr,axis=0) #training scores
  m_cv_hyp1_ts[i,:] = np.mean(score1_cv_hyp_ts,axis=0) #testing scores
  i+=1
```

Figure 1.11: Part of code to search for best value of α

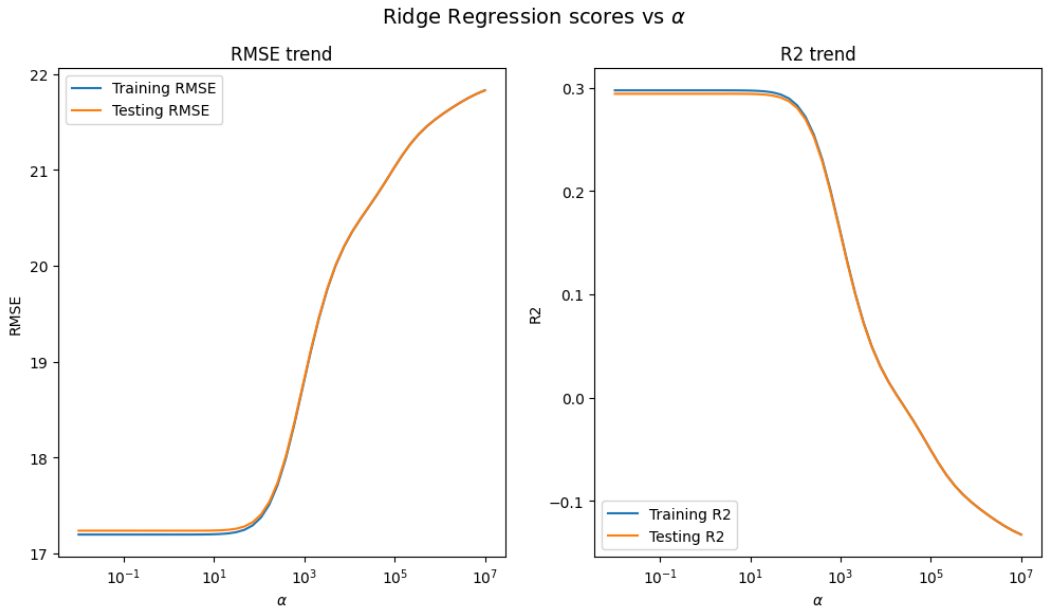

Figure 1.12: RMSE and R^2 of Ridge Regression algorithm depending of α on *dataset 2*

The outcome from the hyperparameter tuning can be summarized in the following table 1.3 and it is done for both datasets, although *dataset 1** could be neglected since not usable for interesting predictions. It is clear that these approach is giving back a results very close to the one obtained with grid search algorithm, there are just some minor differences about the best alpha value due to either approximation errors or different score metrics to be optimized, but basically the R^2 is maximized to the same value. In fig. 1.12 we notice that both training score and the validation score are quite low, so it means that the estimator will be underfitting on *dataset 2*. Then increasing α it is even worsening the approximation error.

In the end we can use the value α_{opt} to fit the model again on the training set and evaluate the result on testing set that represents our source of unseen data.

Source	Dataset	α_{opt}	R^2	RMSE
scikit-learn	1*	3.7276	0.0297	-
scratch	1*	1.2915	0.0297	20.2072
scikit-learn	2	2.7826	0.2940	-
scratch	2	2.1544	0.2940	

Table 1.3: Results of hyperparameter tuning on α

1.3 Kernel Ridge regression

In this section I would like to explore another way to obtain the prediction of the variable popularity by means of the Kernel Ridge regression (KRR).

1.3.1 Short theoretical introduction

Linear predictors lead usually to high approximation error because of the limit of the number of coefficients that is limited to the number of feature d. We can overcome this restriction considering a feature expansion function $\phi : \mathbb{R}^d \to H$, meaning that ϕ maps the data points $x \in R^d$ into a higher dimensional space H.
This method is equivalent to train a non linear predictor in the original space, but the computation of θ becomes unfeasible when n is large. Therefore we can use the kernel trick that helps us finding a kernel function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. Here also an important reminder: any symmetric function is a kernel if an only if the matrix K is positive semi-definite, that is $z^T K z \geq 0$ for all $z \in \mathbb{R}^d$.
The Ridge Regression can be easily kernelized and we can explain how to do that starting from previous eq.1.2 and considering the identity:

$$(\alpha I + S^T S)^{-1} S^T = S^T (\alpha I + S^T S)^{-1} \tag{1.3}$$

In this way we can represent the ridge regression predictor in a Reproducing Kernel Hilbert space (RKHS) by $y^T(\alpha I + K)^{-1} k(\cdot)$. And in this part of the project it is requested to build the Gaussian Kernel from scratch, which its function is $K_\gamma = \exp(-\frac{1}{2\gamma}\|x - x'\|)$. The parameters used in this learning algorithm are:

- alpha: is the regularization strength, that helps improving the conditioning of the problem and reduces the variance of the estimates.
- gamma: is the width of the Gaussian centered on the training point: small values means that the gaussian is very small and it can cause overfitting, large value means that the gaussian is very wide and it can cause underfitting

In our project we will call instead γ as the inverse of the radius of the Gaussian kernel itself.

1.3.2 Project application and coding

In the Python code described in [3], we start exploring and developing two ways to obtain the matrix K:

1. we can build K through nested for loops computing each element in the matrix one at a time as shown in fig.1.13. This is very slow computationally, so it performs very bad when the number of samples start increasing. In fact in the coding part we limit it to a few tousand or less samples to get quick results

```
def gauss_kernel(z1,z2,gamma):
    return np.exp(-gamma*np.linalg.norm(z1-z2)**2)

# kernel trick to build Gaussian Kernel
for i in range(m):
  for j in range(m):
    K_k[i,j] = gauss_kernel(x_k[i,:], x_k[j,:], gamma_fix)
k_k_T = time.time() - t0
```

Figure 1.13: Solution 1: nested for loops to calculate matrix K

2. we can use the broadcasting technique available in `numpy` library to process more efficiently the looping steps directly in C rather than in Python. In [3] it is explained step by step what is behind the simple line shown in fig.1.14 that lets us obtain the K matrix in faster way than the first possibility. In addition we are able check its correctness comparing it to the `rbf_kernel` function available in `scikit-learn`.

```
# compute the Gaussian kernel matrix --> K(X,X)
k_gauss = np.exp(-gamma*np.sum((X_tr - X_tr[:,np.newaxis])**2,axis=-1))
```

Figure 1.14: Solution 2: with broadcasting to obtain matrix K

After having identified an efficient way to obtain K matrix, the second step is to write the whole KRR algorithm (see 1.15 and the `KernelRidgeReg` function) that led us to perform the prediction of target variable `popularity`. We can thus write a function that takes in input the training and testing set and their labels, the hyperparameters α and γ and that returns in output the score of the prediction in term of R^2 and RMSE.
An important remark still related to the matrix K is that K may become computational costly for the main memory, in fact its size depends on the number of training samples n according to the relation $\mathcal{O}(n^2)$, therefore with just more than 20k items we could not be able to compute K because the main memory could not be able to store it. In addition in the `KernelRidgeReg` function there is high computational workload both for memory and time because we need to compute the inverse of a big matrix. We could think about alternatives instead of using the standard inverse, for example as trying Cholesky decomposition, but this investigation is out of the project scope.
As done in previous section, it is important to understand if this developed algorithm outputs results closed to the already tools available in `scikit-learn` module, so we do as following:

```python
def KernelRidgeReg(X_tr,y_tr,X_ts,y_ts,gamma,alpha, print_out=True):
  # Kernel ridge regression function
  # X_tr: training dataset | y_tr: training values --> (simplified X  and Y)
  # X_ts: testing dataset |y_ts: testing values --> (simplified x and y)
  # the Ridge Regression predictor in RHKS is given
  # y = Y'(alpha*I+K(X,X))^{-1}*k(x,X)

  # compute the Gaussian kernel matrix --> K(X,X)
  t0 = time.time() # start time
  k_gauss = np.exp(-gamma*np.sum((X_tr - X_tr[:,np.newaxis])**2,axis=-1))
  KK_new_T = time.time() - t0 # end time

  # compute the coefficients --> w = Y'(alpha*I+K(X,X))^{-1}
  KinvAlpha = np.linalg.inv(alpha*np.eye(X_tr.shape[0])+k_gauss)
  w_kern = y_tr.T @ KinvAlpha

  # ON TRAINING SET
  k_gauss_train = k_gauss # kernel matrix --> k() = k(x,X)
  y_pred_train = k_gauss_train @ w_kern # predictor --> y_pred = w * K(x, X')
  score_train = calculate_metrics(y_pred_train,y_tr) # get the score

  # ON TEST SET
  k_gauss2 = np.exp(-gamma_fix*np.sum((X_ts - X_tr[:,np.newaxis])**2,axis=-1))
  y_pred_test = np.dot(k_gauss2.T,w_kern) #y_pred = w * K(x, X')
  score_test = calculate_metrics(y_pred_test,y_ts)
```

Figure 1.15: KRR implemetation in Python from scratch

- we assume γ (the parameter of the gaussian kernel) and α (the one of the regularization) as given and fixed and we reduce the datasets to smaller ones with just 3000 samples in the training part and 600 samples in the testing part in order to avoid memory issues
- we start calculating the baseline using KRR from `scikit-learn` on the *dataset 1**and *2*
- we run the KRR from scratch on both datasets
- we check the results looking to RMSE and R^2 and computational time

These outcome of above approach is summarized in table 1.4 and with this first example KRR application we can cleary see that we can reach high score on training set and a very bad prediction on the validation set, this means that we are having a problem of overfitting.

Source	Dataset	Time [s]	R^2 train	RMSE train	R^2 test	RMSE test
scikit-learn	1*	2.082	0.656	12.285	-0.213	23.097
scikit-learn	2	3.113	0.623	12.853	-2.184	37.417
scratch	1*	3.854	0.656	12.285	-0.213	23.097
scratch	2	15.539	0.623	12.853	-2.184	37.417

Table 1.4: Results of KRR algorithms obtained with fixed $\alpha = 0.5$ and $\gamma = 1$ and n.of training samples = 3000 and n.of testing samples = 600

1.3.3 Cross Validation

The purpose of this part is to implement a 5-fold CV that helps us giving a better estimation of the risk as already done in ridge regression paragraph 1.2.3. Therefore we can adapt previous CV algorithm to Kernel Ridge Regression case just changing the function recalling the new `KernelRidgeReg` instead of the previous `ridge_regression`.

For the sake of simplicity and to avoid memory issues, we choose to go ahead only

Source	Dataset	R^2 with cv = 5
scikit-learn	1*	-0.358
scratch	1*	-0.358

Table 1.5: Results of cross validation for KRR on *Dataset 1**

with *dataset 1** and when we compare our algorithm of KRR with CV (`K_fold_cv_KRR`) developed from scratch with the one available from `scikit-learn` package we will get the same results as shown also in table 1.5. We do not explain in details how CV works or it is implemented because it would be a repetitions, but we can clearly notice that R^2 is very poor even worse than a straight hyperplane that should represent the null hypothesis.

1.3.4 Hyperparameters tuning

In this section we perform experiments with different training parameters γ and α that are spaced evenly on a log-space. We are not choosing a lot of values for these parameters, but only 5 to keep it simple. We can then follow these steps:

- we use `GridSearchCV` of `scikit-learn` package to get the baseline for the best parameters, this methods as defaults has also cross validation in it. It is evident that the results obtained aren't of much interest since R^2 is very low 0.057 as visible in table 1.6.

Source	Dataset	α_{opt}	γ_{opt}	R^2	RMSE
scikit-learn	1*	0.3162	0.0178	0.057	-
scratch	1*	0.3162	0.0178	0.057	20.3284

Table 1.6: Results of hyperparameter tuning on α and γ

- we then explore with a nested for loop our hyperparameters search from scratch saving the scores of R^2 and RMSE included in previous table 1.6. This step takes around the 5 times more in comparison to `GridSearchCV` because we are using the matrix inversion as it is and not implementing some tricks as Cholesky decomposition (because K is positive semi-definite) or other solution that could help to boost the computational time.

- we can create useful charts when we compare the training and validation scores depending separately on γ and on α. In fig.1.16 we can see that when γ (inverse of

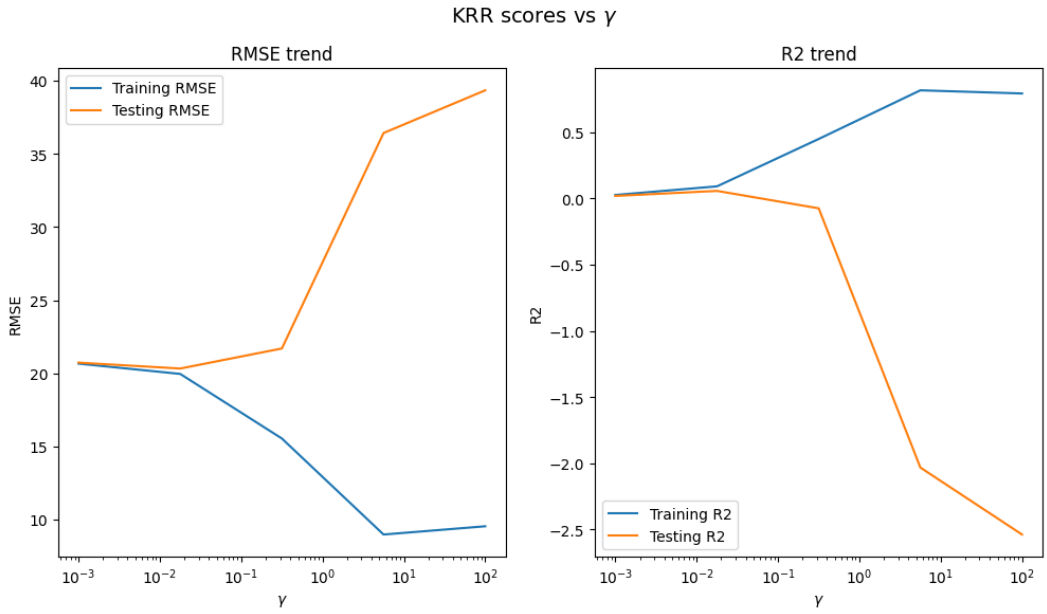

Figure 1.16: KRR training and testing scores depending on γ

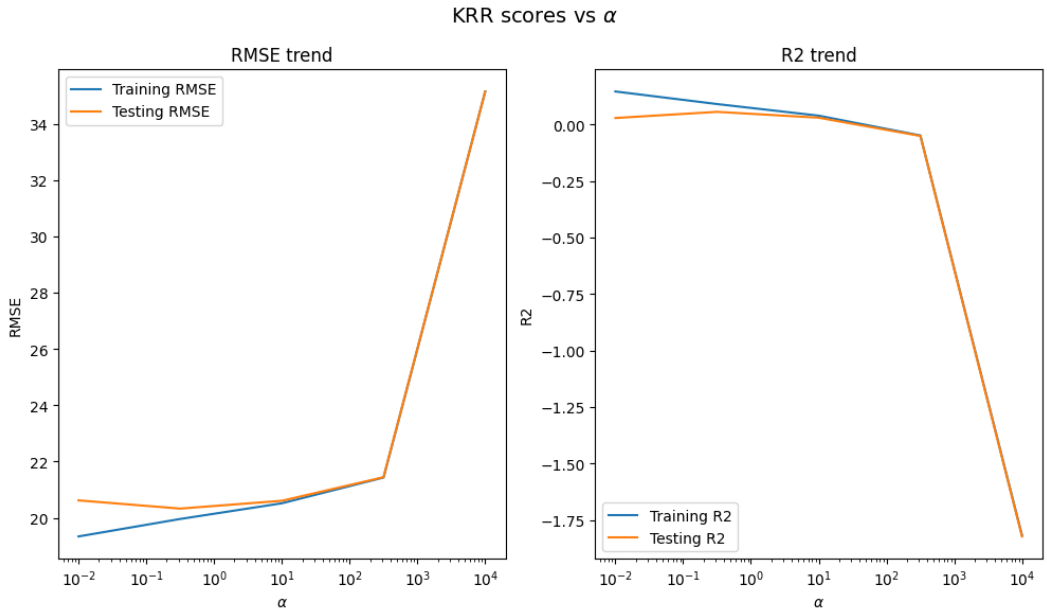

Figure 1.17: KRR training and testing scores depending on α

radius of the Gaussian kernel) increases, the training error decrease and R^2 increases, but at the contrary we are worsening the results on the validation set: this is the problem of overfitting. Then looking at the charts of fig.1.17 as α grows instead, we can find that both training and testing errors increase and this is a clear situation of underfitting. In general we do not have good prediction with this KRR algorithm since the results of R^2 are nearly closed to 0.

In the end we can use the retrieved optimal values α_{opt} and γ_{opt} to in the KRR model fitted again on the training set and then evaluate the model on the test set with on unseen data.

1.4 Conclusion

In conclusion we did a very interesting project exploring Ridge Regression and Kernel Ridge Regression passing through a short theoretical introduction, then jumping into the implementation from scratch of the learning algorithms, adding some techniques as cross validation for better estimation of the errors and in the end exploring the role of training parameters. I think it is a very nice exercise in order to be able to master what is under the hood of a machine learning algorithm and this approach could ne extended also to other algorithms of ML worlds.
However, on one hand I would have got more satisfaction experimenting it onto a dataset that could led me to better prediction results. On the other hand I think that this is useful to remind us that we are not always able to get good regression models and indeed it could occur very often in a data science daily life.
The results obtained with algorithm developed from scratch are closed to `scikit-learn` outcomes so they can be used as an efficient alternative.
In general simple model do not risk to overfit and this is satisfied by Ridge Regression. Instead Kernel Ridge Regression clearly overfits and when choosing the best parameters that avoid this problem we got anyway extremely poor results.

Bibliography

[1] Prof. Nicolò Cesa Bianchi. Material from module machine learning, 2023.

[2] Kaggle. Spotify tracks dataset. https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset/data.

[3] Stefano Zanlucchi. Project - github. https://github.com/Stravij/DSE_MachineLearning/blob/main/Project_Machine.ipynb, 2023-2024.