Statistical Methods for Machine Learning

Alessandro Minoli 20202A

A.Y. 2022-2023

Contents

1 Introduction

1.1 Project specifications

(Kernel) Ridge Regression

Download the Spotify Tracks Dataset and perform ridge regression to predict the tracks' popularity. Note that this dataset contains both numerical and categorical features. The student is thus required to follow these guidelines:

- first, train the model using only the numerical features,
- second, appropriately handle the categorical features (for example, with one-hot encoding or other techniques) and use them together with the numerical ones to train the model,
- in both cases, experiment with different training parameters,
- use 5-fold cross validation to compute your risk estimates,
- thoroughly discuss and compare the performance of the model

The student is required to implement from scratch (without using libraries, such as Scikit-learn) the code for the ridge regression, while it is not mandatory to do so for the implementation of the 5-fold cross-validation.
Optional: Instead of regular ridge regression, implement kernel ridge regression using a Gaussian kernel.

1.2 Project overview

The project starts with the exploration and cleaning of the dataset.
Then, after some feature engineering, we will present theoretical notions about both Ridge and Kernel Ridge regression along with their implementation in Python. These models are finally tuned, tested and evaluated on the dataset. The predictions are initially made considering only the numerical features of the data and then also including the categorical ones.

1.3 Declaration

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

2 Dataset analysis and cleaning

The dataset is located at the path *datasets/spotify_tracks.csv*. All the operations described in this section are performed in the notebook *dataset_analysis_cleaning.ipynb*.

2.1 Features exploration

Table 1 lists the features of the dataset along with their name, type and domain.

name	type	domain
popularity	numerical	$\mathbb{N} \in [0, 100]$
track_id	categorical	string
artists	categorical	string
album_name	categorical	string
track_name	categorical	string
duration_ms	numerical	$\mathbb{R}_{>0}$
explicit	categorical	$\in \{true, false\}$
danceability	numerical	$\mathbb{R} \in [0, 1]$
energy	numerical	$\mathbb{R} \in [0, 1]$
key	categorical	$\mathbb{N} \in [0, 11]$
loudness	numerical	$\mathbb{R}$
mode	categorical	$\in \{0, 1\}$
speechiness	numerical	$\mathbb{R} \in [0, 1]$
acousticness	numerical	$\mathbb{R} \in [0, 1]$
instrumentalness	numerical	$\mathbb{R} \in [0, 1]$
liveness	numerical	$\mathbb{R} \in [0, 1]$
valence	numerical	$\mathbb{R} \in [0, 1]$
tempo	numerical	$\mathbb{R}_{\geq 0}$
time_signature	categorical	$\mathbb{N} \in [0, 5]$
track_genre	categorical	string

Table 1: Features overview

2.2 Data cleaning

The dataset has 114000 rows and 20 features. Only one row has missing values, we can drop it. There are also some duplicate rows we can drop.

Rearrangement of the *track_genre* attribute

The examples are uniformly divided by *track_genre* and some of them have the same *track_id*. There are rows that are completely equal except for their

track_genre. We aggregate these rows by joining their *track_genre* attributes with a semicolon. After this operation the dataset will be reduced to 90460 rows but some pairs of examples with the same *track_id* are still remaining. All these pairs are the same row except for *track_genre* and *popularity*. Again, we aggregate these pairs of rows by joining their *track_genre* attributes with a semicolon and taking the mean of their *popularity*. Finally *track_id* has become a unique identifier of the songs, we can now drop this feature since it has no predictive power. Removing *track_id* introduces a few duplicate rows we can drop; the dataset now has 89665 rows. Currently the *track_genre* attribute of the examples consists of a string that lists all the genres a song belongs to (e.g. "rock;fusion;jazz"). It has 1584 distinct values, while at the beginning there were only 114 distinct single genres. For each single genre, we introduce in the dataset a boolean feature named *GENRE_{genre_name}* and we set it to 1 when the song belongs to that specific genre. This operation is performed row by row looking at the *track_genre* attributes. When the new features have been initialized we can drop the *track_genre* feature.

The cleaned dataset eventually has 89665 rows and 132 features. It is saved at the path *datasets/spotify_tracks_cleaned.csv* and is ready to be used.

2.3 Distribution of popularity values

Figure 1 represents the distribution of the values of the target feature in the cleaned dataset. There are 9352 examples with popularity = 0 (10.43 % of the total) and 9797 examples with popularity $\geq$ 60 (10.93 % of the total).
Q1 = 19.0, Q2 = 33.0, Q3 = 49.0 and consequently IQR = 30.0 .

3 Feature engineering

3.1 Categorical features encoding

Categorical features needs to be encoded into numerical ones before being used to make predictions. The binary feature *mode* and the boolean feature *GENRE_{genre_name}* are already mapped in {0,1}. For the boolean feature *explicit* we only need to map *true* values to 1 and *false* values to 0.

One-hot encoding

Performing one-hot encoding on a categorical feature consists in adding to the dataset a column of zeros for all its distinct values and then, for each row, setting to 1 the column corresponding to the value of the encoded feature in that row. An example of application of this technique in shown in table 2. To encode a feature with n distinct values we need to add to the dataset n columns.
We applied one-hot encoding on the features *time_signature* and *key*.
The amount of added columns is acceptable because these features only have respectively 5 and 12 distinct values.

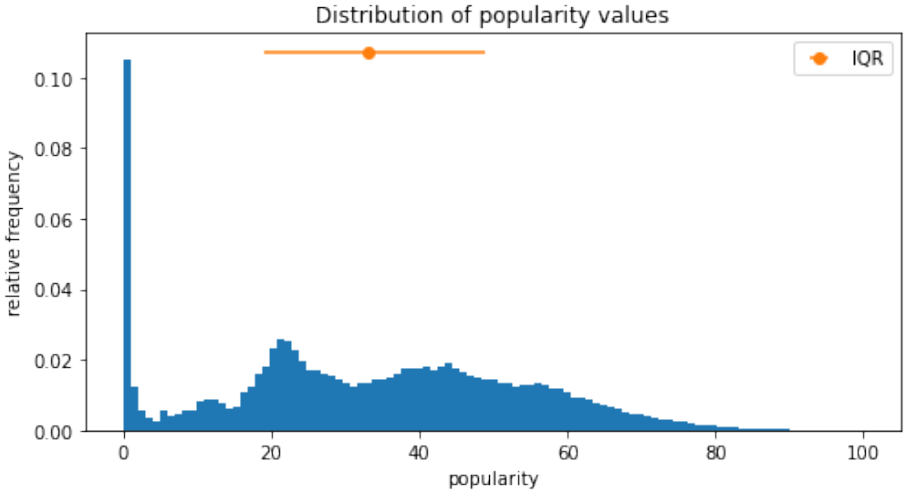

Figure 1: Distribution of popularity values with IQR

feature	**value-A**	**value-B**	**value-C**
A	1	0	0
C	0	0	1
A	1	0	0
B	0	1	0
B	0	1	0
C	0	0	1

Table 2: Example of one-hot encoding

Binary encoding

The features *artists*, *album_name* and *track_name* have respectively 31437, 46589 and 73608 distinct values. In this case one-hot encoding is not feasible anymore, we choose binary encoding. This technique consists in assigning to each distinct value of the feature a binary integer and to place its digits into new separate columns. An example of application of this technique in shown in table 3. To encode a feature with n distinct values we need to add to the dataset $\lceil log_2 n \rceil$ columns. To encode our features we used respectively 15, 16 and 17 columns.

After features encoding we split the dataset in training set and test set with a test size of 20%. The split is done stratifying the data according to the target variable.

feature	new-col-1	new-col-2	new-col-3
A	0	0	0
C	0	1	0
B	0	0	1
D	0	1	1
B	0	0	1
E	1	0	0
A	0	0	0
F	1	0	1

Table 3: Example of binary encoding

3.2 Features selection

Performing feature selection consists of using only a subset of all data features to make predictions. The purpose of feature selection is to reduce computational and memory requirements of the models without worsening their predictive performances. A simple feature selection approach is to discard the features that have a low correlation with the target variable *popularity*. Looking at our training set we can measure the correlation between all pairs of features, figure 2 shows the correlation matrix of the data. Moreover, the Python *sklearn* library contains a class that scores features with respect to how much relevant they are for a regression task. The result of the application of this tool is shown in figure 3. Given the matrix, the scores and some experiments, we decided to ignore the features *tempo* and *liveness* because they are not powerful at all.

3.3 Features standardization

The last step before fitting our models is to standardize the data, namely to shift and scale them in order to have mean $\mu = 0$ and standard deviation $\sigma = 1$. Given a feature X, we standardize it following the formula:

$$X' = \frac{X - \mu}{\sigma} \tag{1}$$

Standardization, in general, improves the effectiveness of the models and ensures that data are on the same scale. When applying this transformation it's important to fit the scaler on the training data only because we want to avoid having scaling parameters that depend on the test set.

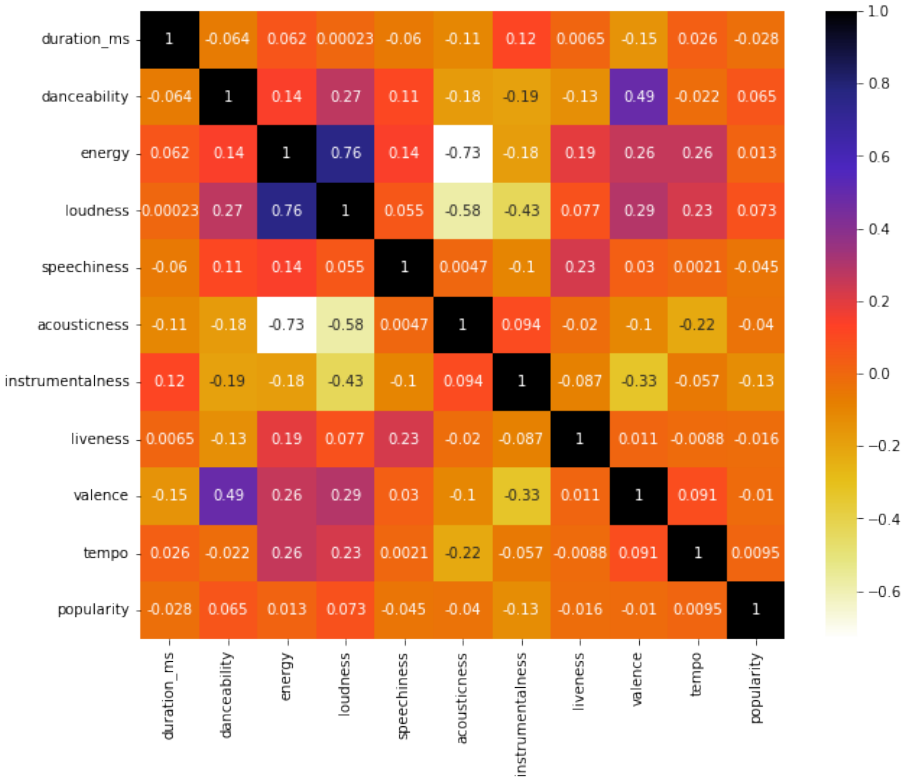

Figure 2: Correlation between training set features

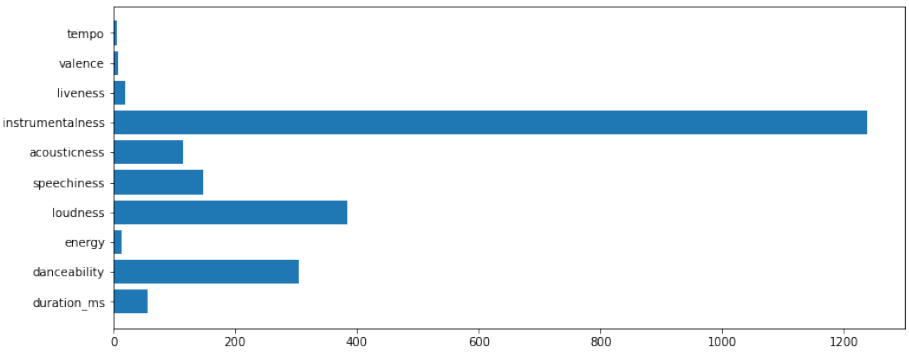

Figure 3: Feature selection scores

4 Ridge regression

4.1 Theory recap

In linear regression, predictors are linear functions $h : \mathbb{R}^d \to \mathbb{R}$ of the form $h(x) = w^T x$, where $w \in \mathbb{R}^d$. Given a training set $(x_1, y_1), ..., (x_m, y_m) \in \mathbb{R}^d \times \mathbb{R}$, we define the standard linear regression predictor as the ERM with respect to the square loss :

$$w_S = \underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \sum_{t=1}^{m} (w^T x_t - y_t)^2 \tag{2}$$

In (3) we rewrite (2) in matricial form, given that S is the $m \times d$ design matrix such that $S^T = [x_1, \ldots, x_m]$. Since we are dealing with a convex function, by putting the gradient to 0, we finally obtain a closed-form expression of the predictor. This holds if $S^T S$ is invertible.

$$w_S = \underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \|Sw - y\|^2 = (S^T S)^{-1} S^T y \tag{3}$$

Ridge regression extends standard linear regression by introducing a regularizer in the ERM, as written in (4) with $\alpha \geq 0$. This predictor has more bias than the standard one but lower variance. It's more stable and less prone to overfitting. Another time, we can express the predictor in closed-form. The more α tends to 0, the more we are moving towards the standard linear regression.

$$w_{S,\alpha} = \underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \|Sw - y\|^2 + \alpha \|w\|^2 = (\alpha I + S^T S)^{-1} S^T y \tag{4}$$

4.2 Python implementation

My implementation of Ridge regression is the class *RR* in figure 4. The library *NumPy* has been used for linear algebra operations. The class inherits from two classes in *sklearn* in order to be compatible with all the other tools of the library (e.g. *GridSearchCV*). The class contains the following methods :

- ***__init__*** : receives λ and initializes the predictor
- ***fit*** : receives training data and computes w as in equation (4)
- ***predict*** : receives test data and computes the prediction $w^T x$

5 Kernel Ridge regression

5.1 Theory recap

By training a linear predictor on a feature-expanded training set that lives in a high-dimensional space $\mathcal{H}$, we're able to learn a more complex nonlinear

```python
import numpy as np
from sklearn.base import BaseEstimator, RegressorMixin

class RR(BaseEstimator, RegressorMixin):

    def __init__(self, λ = 1.0):

        if λ < 0: raise ValueError("λ must be >= 0")
        self.λ = λ

        self.w_ = None

    def fit(self, X, y):

        X = np.insert(X, 0, np.ones(X.shape[0]), axis=1)

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

        self.w_ = np.linalg.inv(X.T @ X + self.λ * I) @ X.T @ y

    def predict(self, X):

        if self.w_ is None:
            raise RuntimeError('Model is still to fit')

        X = np.insert(X, 0, np.ones(X.shape[0]), axis=1)

        return X @ self.w_
```

Figure 4: *ridge_regression.py* , Ridge regression implementation

predictor in the original space and obtain a smaller bias error. The data points are expanded using some function $\phi : \mathbb{R}^d \to \mathcal{H}$. Kernel functions are functions of the form $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ such that $K(x, x') = \phi(x)^T \phi(x')$ for all $x, x' \in \mathbb{R}^d$. An example of kernel function is the Gaussian kernel :

$$K_\gamma(x, x') = exp\Big(\frac{-1}{2\gamma}\|x - x'\|^2\Big) \tag{5}$$

where $\gamma > 0$. Predictors that use Gaussian kernel predict any point as a combination of Gaussians centered on the training points and evaluated at x. The parameter γ controls the width of these Gaussians. Choosing γ too small leads to overfitting and, viceversa, choosing γ too large leads to underfitting. Learning algorithms that use Gaussian kernels can be consistent : the expected risk of their predictors converges to the Bayes risk as the sample size grows to infinity.

Given the $m \times d$ design matrix S and the vector of training labels $y \in \mathbb{R}^m$, we can represent the Ridge regression predictor $w^T x = y^T S(\alpha I + S^T S)^{-1} x$ in kernel space as

$$\langle g, \phi_K(x) \rangle_K = y^T(\alpha I + K)^{-1} k(x) \tag{6}$$

where K is the $m \times m$ matrix with entries $K_{i,j} = K(x_i, x_j)$ and $k(\cdot)$ is the vector $(K(x_1), \cdot), \ldots, K(x_m, \cdot))$ of functions $K(x_t, \cdot) = \langle \phi_K(x_t), \cdot \rangle_K$.

5.2 Python implementation

My implementation of Kernel Ridge regression is the class *KRR* in figure 5. The class contains the following methods :

- ***__init__*** : receives λ, γ and initializes the predictor
- ***gaussian_kernel*** : receives the matrices X, Y and computes (5) for each pair of rows of those matrices
- ***fit*** : receives training data and computes the costant part $y^T(\alpha I + K)^{-1}$ of equation (6)
- ***predict*** : receives test data and finishes the computation of equation (6) for each test point

6 Performance evaluation

6.1 Metrics

We can use several metrics to evaluate the performance of regressors, here we present the ones that we will consider.
When we talk about residuals we are referring to the differences $y - \hat{y}$ between actual target values y and predicted target values $\hat{y}$.

Mean Squared Error

Mean Squared Error (MSE) is the average of squared residuals :

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y_i})^2 \tag{7}$$

where n is the test set size.

Coefficient of determination

Coefficient of determination (R^2) is defined as :

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y_i})^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{8}$$

where n is the test set size and $\bar{y}$ is the mean of the actual target values. When the predicted values perfectly match the actual ones then $R^2 = 1$. Models that

```python
import numpy as np
from scipy.spatial import distance
from sklearn.base import BaseEstimator, RegressorMixin

class KRR(BaseEstimator, RegressorMixin):

    def __init__(self, λ = 1.0, γ = 1.0):

        if λ < 0: raise ValueError("λ must be >= 0")
        if γ <= 0: raise ValueError("γ must be > 0")
        self.λ = λ
        self.γ = γ

        self.X_ = None
        self.c_ = None

    def gaussian_kernel(self, X, Y):
        return np.exp((-0.5 / self.γ) * np.square(distance.cdist(X, Y, 'euclidean')))

    def fit(self, X, y):

        self.X_ = X.values
        K = self.gaussian_kernel(self.X_, self.X_)
        self.c_ = y.T @ np.linalg.inv(K + self.λ * np.identity(self.X_.shape[0]))

    def predict(self, X):

        if self.c_ is None or self.X_ is None:
            raise RuntimeError('Model is still to fit')

        return self.c_ @ self.gaussian_kernel(self.X_, X.values)
```

Figure 5: *kernel_ridge_regression.py* , Kernel Ridge regression implementation

always predict $\bar{y}$ will have $R^2 = 0$ and models that are worse than this simple model will have negative R^2 values. This metric is very often used to evaluate regression models.

Residuals analysis

Given the residuals, we can place them in a histogram to see their distribution and understand if we are systematically over-predicting or under-predicting the target variable. The median and some dispersion measure of the residuals can also be calculated. For example we will use the interquartile range $IQR = Q3 - Q1$, which is the range in which the middle 50% of the residuals lie.

6.2 Hyperparameters tuning

As we have seen in sections 4 and 5, our models have hyper-parameters that need to be tuned. To accomplish this task we use the class *GridSearchCV*

of *sklearn*. This class permits to search for the best hyper-parameters values among user-defined candidate values and, at the same time, to cross-validate our model. We choose to use 5-fold cross validation.

7 Experiments and final results

The developed models have been tested on the dataset in the notebooks *predictions_numerical.ipynb* and *predictions_all.ipynb*.
Kernel Ridge regression execution is computationally heavy, so we could not apply this model to the whole dataset. The hyper-parameters of Kernel Ridge regression are tuned only on a sample of the training set of size 5000. The best model is then trained on a sample of the training set of size 20000 and finally tested on a sample of the test set of size 4000.
Both RR and KRR have been trained considering only the numerical features and then considering all the features of the data.
Figures 6, 8, 10 and 12 represent the regions where we found the best hyper-parameters for the models in the four experiments.
Figures 7, 9, 11 and 13 represent the distributions of the residuals of the models applied on the test set in the four experiments. We observe that, when we consider only the numerical features, our models have the tendency to over-predict the tracks' popularity values.
Table 4 provides the performance of the models in terms of MSE and R^2, these results are also plotted in figure 14.

experiment	MSE	R^2
RR numerical features	410.58	0.03
RR all features	258.81	0.39
KRR numerical features	368.04	0.09
KRR all features	204.33	0.49

Table 4: Models comparison

The predictions of our models have been quite satisfactory when considering all the features of the data and pretty inaccurate when considering only the numerical ones. In both cases, as expected, KRR performed better than RR according to the metrics we analyzed. The only drawback of KRR is that its execution is slow and, consequently, it's not convenient to apply this predictor on very large datasets.

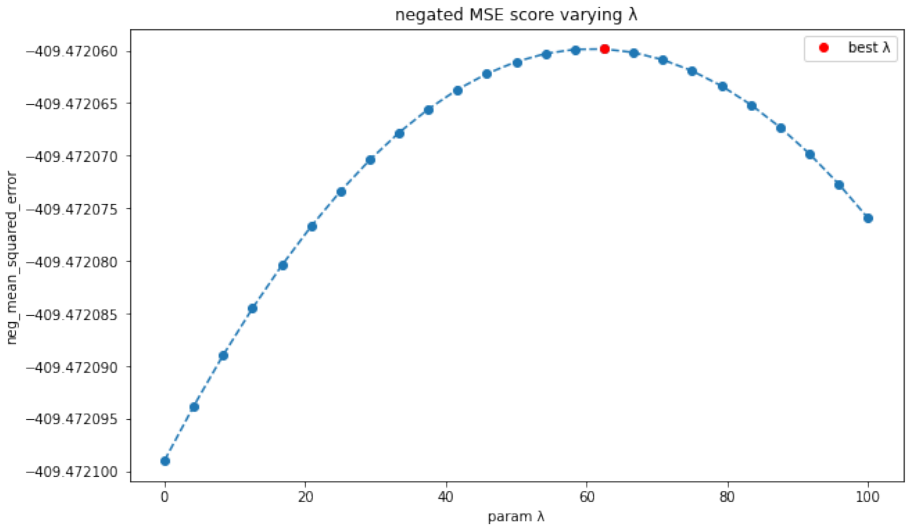

Figure 6: Ridge regression using numerical features,
hyper-parameters tuning gives that best $\lambda = 62.5$

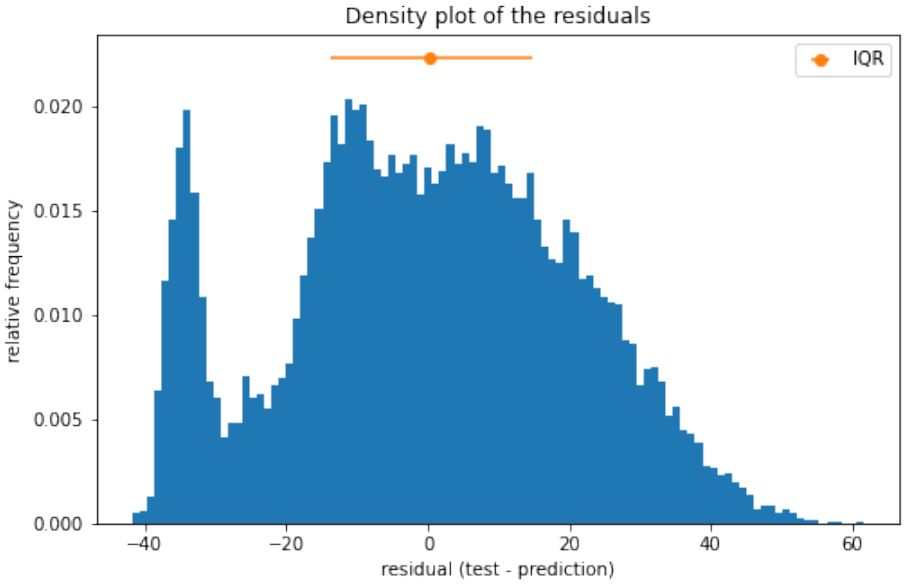

Figure 7: Ridge regression using numerical features on the test set,
Q1 = -13.72, Q2 = 0.18, Q3 = 14.73 and IQR = 28.45

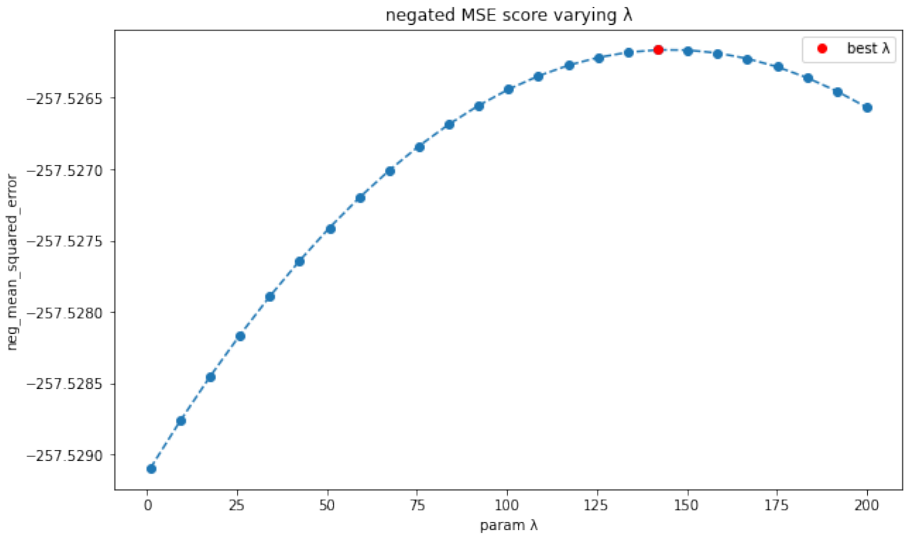

Figure 8: Ridge regression using all features,
hyper-parameters tuning gives that best $\lambda = 141.96$

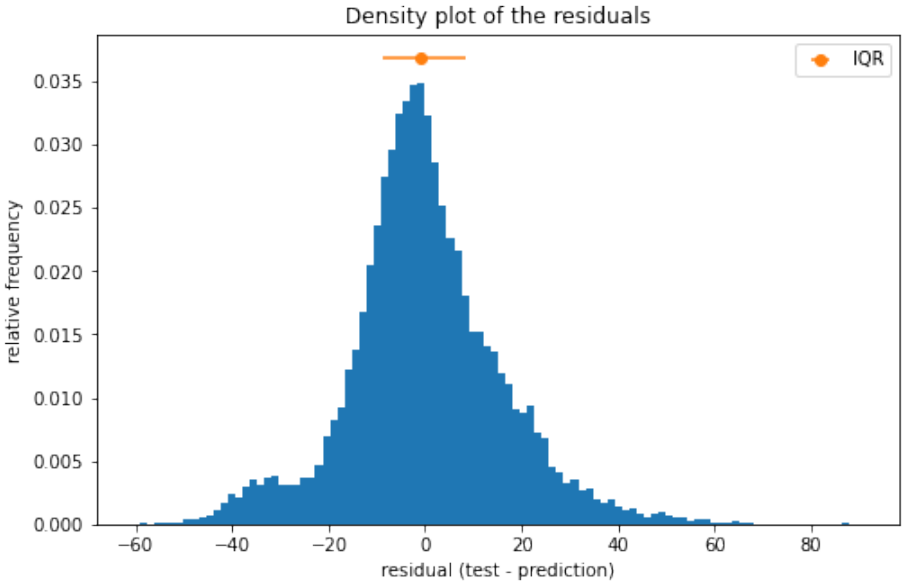

Figure 9: Ridge regression using all features on the test set,
Q1 = -8.85, Q2 = -1.01, Q3 = 8.56 and IQR = 17.41

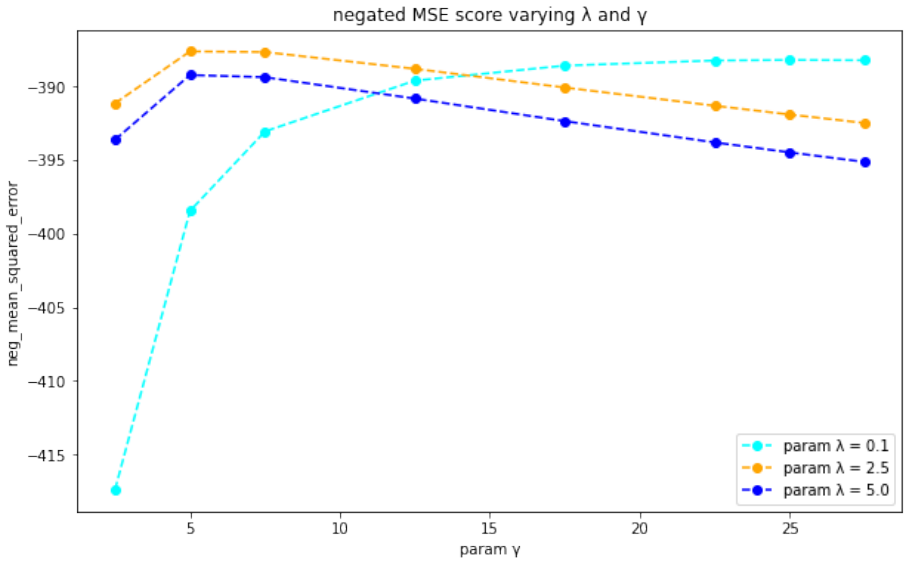

Figure 10: Kernel Ridge regression using numerical features, hyper-parameters tuning gives that best $\lambda = 2.5$ and best $\gamma = 5.0$

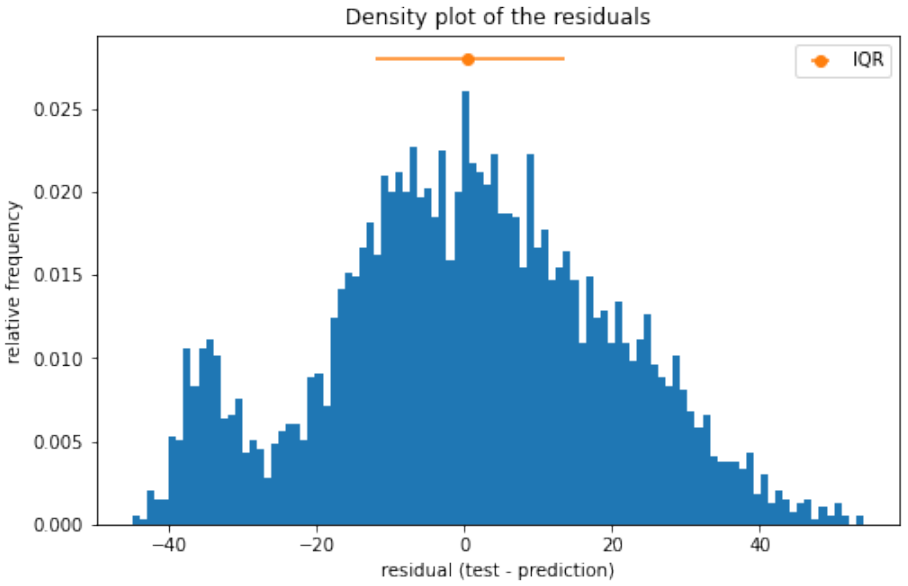

Figure 11: Kernel Ridge regression using numerical features on the test set, Q1 = -12.05, Q2 = 0.32, Q3 = 13.61 and IQR = 25.67

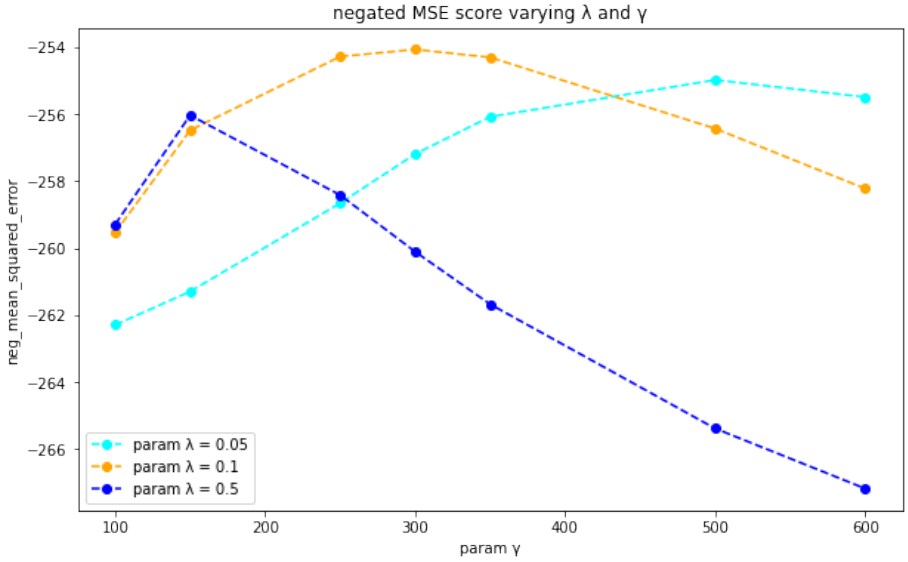

Figure 12: Kernel Ridge regression using all features,
hyper-parameters tuning gives that best $\lambda = 0.1$ and best $\gamma = 300.0$

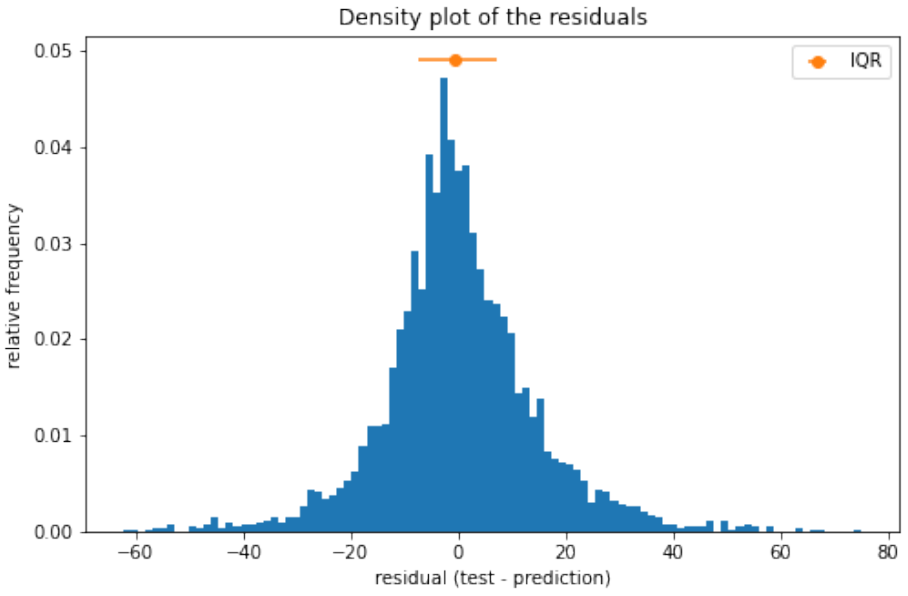

Figure 13: Kernel Ridge regression using all features on the test set,
Q1 = -7.43, Q2 = -0.76, Q3 = 7.30 and IQR = 14.73

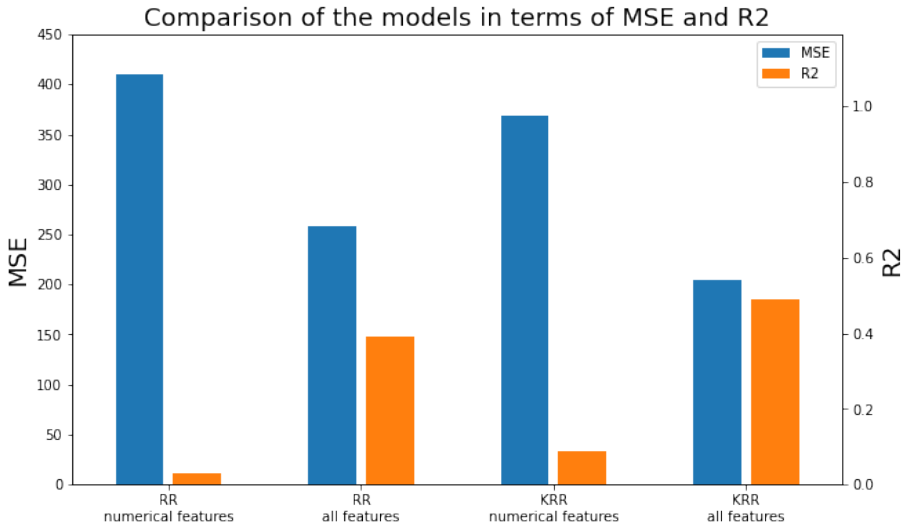

Figure 14: Models comparison

References

[1] Nicolò Cesa-Bianchi, *Linear predictors*, lecture notes (2022-23)

[2] Nicolò Cesa-Bianchi, *Kernel functions*, lecture notes (2022-23)

[3] Shai Shalev-Shwartz, Shai Ben-David *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press (2014)