Popularity Prediction of Spotify Tracks

Luca Bellani and Andrea Longoni

Statistical Methods for Machine Learning

Abstract

In this paper we develop a prediction system that estimates the popularity of a song. The flow of this work starts with the manipulation of the dataset: "Spotify Track Dataset". The training and estimation parts, together with k-fold nested cross validation and kernel functions, are developed from scratch and compared with the `scikit-learn` operations.

1 Introduction

Ridge Regression is a linear prediction where vectors of real coefficients are approximated by an hyperparameter called "regularizer". This choice is especially useful in a situation in which the training set is perturbed, thus achieving a more stable predictor for reducing the variance of the risk.
In order to get the best hyperparameter α, the k-fold nested cross validation process is built.

K-Fold Nested Cross Validation splits the dataset in K parts and computes the estimate by dividing further the training data, iterates on the set of hyperparameters for each fold predictor and delivers in the end the average of their performance as output. For experimental purpose, we have implemented an additional part: the kernel trick. This allows us to deal with higher dimensional space data, without the explicit necessity of calculating it. In fact, this process would be computationally expensive for higher-dimensional spaces.

The aim of this project is to analyze these different techniques in order to predict the popularity of tracks, with the purpose of determining which models and hyperparameters fit better.

2 Dataset

"Spotify Track Dataset" is a CSV dataset of Spotify tracks over a range of 125 different genres. The data was collected and cleaned using Spotify's Web API and Python. Each track has some audio features associated with it, both numerical and categorical, with a total number of 21 features that describe exactly 114000 datapoints [1].

2.1 Feature Description

The attributes of the dataset represent the inertial aspects of the tracks: most of the numerical features, like `instrumentalness`, `liveness` or `speechiness`, detect the presence of the characteristics they specify, indicating the probability of how much the phenomenon occurs. The categorical metadata show for the majority nominal characteristics: in fact, `track_id`, `artists`, `album_name`, `track_name` labels are not functional and discarded from the `DataFrame` for discovering the popularity of a track. Features such as `key`, `mode` and `time_signature` are represented as numbers but they can be considered as categorical, since are labels mapped to integer values.

2.2 Data Manipulation

The first step is to convert the `.csv` document into a `DataFrame` object. The issue of the Spotify Track Dataset is that the rows are ordered alphabetically by musical genre: after we have read the file, a randomization of the datapoint position is applied, in order to enhance the prediction perfomance. Another correlated problem is that some rows are perfectly equal except for the `track_genre` attribute: the solution is explained in the subsection 2.3.

NumPy and Pandas libraries are imported for their data structure and functions they offer, in particular for the data management and calculus.

Next, the data are divided randomly into two sets, training (70%) and test (30%). The project focuses on training both numerical and all features, so we have momentarily dropped the categorical metadata for the first part and, as already mentioned in the section above, `key`, `mode` and `time_signature` too. For a more accurate analysis, we have randomized the row positions in three distinct modes, this simply by using different seed generations. In this way we can view how much variance occurs in the estimations.

2.3 Manage a unique track with multiple genres

One noticeable thing that shows up in the information set is that there are more rows with the same `track_id` (this can be seen as a primary key) that perfectly match except for their `track_genre` attribute. The solution implemented for this particular problem is managed as follows:

1. build a new `DataFrame` object with `track_id` and `track_genre` features, in which One-Hot Encoding procedure is applied on `track_genre`;
2. group by `track_id` and aggregate the one-hot encoded features into a single line;
3. create a temporary `DataFrame` with all the features, except for the "non functional features" (cited in subsection 2.1) and `track_genre`;
4. merge the temporary `DataFrame` with the "`track_genre DataFrame`" through inner-join, in order to achieve the final object used for the next operations.

In general, if we have the same track with multiple genres, this will be saved as multiple lines at the beginning. This method aggregates these rows into a single one, simply by keeping 1 in the `track_genre` cells there were before.

In this manner, we have decreased the amount of datapoints down to 89740.

2.4 Normalization Range and One-Hot Encoding

A `min-max` normalization procedure is applied for `loudness`, `tempo` and `duration_ms`, because they are the only features that are not comprised in the range 0-1.
In order to compute the ridge regression on all the columns, it has been performed the `One-Hot Encoding` approach: this simply converts the categorical data into binary ones. For each non-numerical data point, a binary vector is created of length equal to the number of unique categories in the variable. Each vector has a 1 in the position corresponding to the category label and 0s in all other positions. This 1 indicates the presence of that category for that particular datapoint. The result of this process expands the number of attributes up to 146, but since that the `explicit` column is binary, we keep only the true one, thus it will be considered 1 as explicit and 0 not. In the end, an additional normalization has been utilized: we achieved a $[0, 100]$ interval on all features aimed at taking the same range of `popularity.`

3 Ridge Regression

Linear regression is an approach in which predictors are linear functions $h : \mathbb{R}^d \to \mathbb{R}$, each parameterized by a vector $\boldsymbol{w} \in \mathbb{R}$ of real coefficients: $h(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$. Hence, the ERM with respect to the square loss is:

$$w_s = \underset{\boldsymbol{w} \in \mathbb{R}^d}{\operatorname{argmin}} \sum_{i=1}^{m} (\boldsymbol{w}^\top \boldsymbol{x}_t - y_t)^2 \tag{1}$$

where m are the number of datapoints in a training set.

In a matrix context, we have a closed-form expression for the predictor (the function that we are dealing with is convex, so its gradient can vanish to 0):

$$\boldsymbol{w} = (S^\top S)^{-1} S^\top \boldsymbol{y} \tag{2}$$

where

- S is a $m \times d$ design matrix (m is the number of training points and d is the number of features);
- $S^\top S$ should be nonsingular, which means that one or more of its rows/columns are not definable as a linear combination of all or some other of its rows/columns;
- $\boldsymbol{y} = (y_1, \ldots, y_m)$, the label vector of the training set.

However, when $S^\top S$ is nearly singular, the variance error could grow. In order to enhance the predictor, a regularizer $\alpha > 0$ is added for increasing the stability [2]:

$$\boldsymbol{w} = (\alpha I + S^\top S)^{-1} S^\top \boldsymbol{y} \tag{3}$$

3.1 Implementation

The function delegated to calculate the formula above (`ridge_regression`) takes as arguments the hyperparameter α and the data to fit. Next, the `popularity` vector ($\boldsymbol{y}$) is extracted from the training set. Through $NumPy$ facilities, we have computed the hyperplane $\boldsymbol{w}$ via ridge regression and converted it into a $DataFrame$ with $Pandas$. With the aim of predicting and see how this model works, another two functions are written: `predict`, which outputs the estimate, and `avg_square_loss`, that gives as a result the mean square error (MSE) of the hyperplane $\boldsymbol{w}$ on a given test set:

$$MSE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \tag{4}$$

where, in our case, $\hat{y}_i = \boldsymbol{w}^\top \boldsymbol{x}_i$, $\boldsymbol{w} \in \mathbb{R}^d$.

3.2 Results

From now on, the results written in this report refer to the seed generation with number 0.

3.2.1 Numerical features

The MSE (4) calculated with the regularizer α set to 0 (the standard linear regression) only on numerical features is 414.79 with a variance of 15.86, whereas the `scikit-learn` Ridge predictor gives a MSE of 408.01 with a variance of 14.20. The training and test set are manipulated as written in the subsection 2.2. In the figure 1 we have considered a set of values of α with its resulting square losses in the ridge regression fashion. The first plot figures the ridge prediction developed from scratch, which is compared with the second plot that represents the ridge regression model of `scikit-learn`: `sklearn.linear_model.Ridge`.

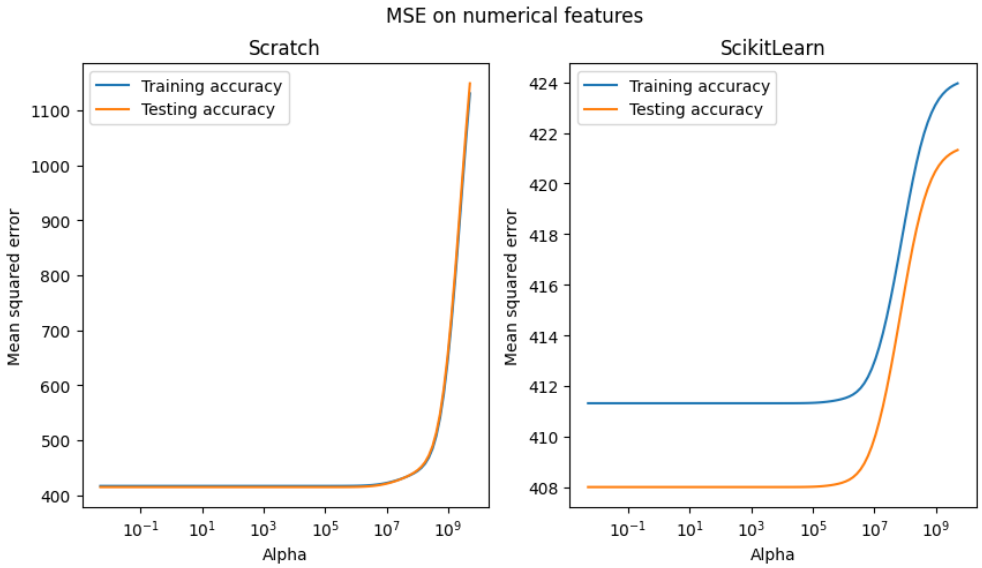

Figure 1: These plots depict the training and test losses, based on only numerical features in a ridge regression context. The X-Axis represents the alpha coefficients used in the estimate, while the Y-Axis represents the mean squared error given by the prediction. Note that the Y-Axis scales are different from one another.

3.2.2 All features

The MSE (4) calculated with the regularizer α set to 0 (the standard linear regression) on all the features is 273.34, with a variance of 31.91, whereas the `scikit-learn` Ridge predictor gives a MSE of 273.33 with a variance of 20.36. The training and test set are manipulated as written in the subsection 2.2. In the figure 2 we have considered a set of values of α with its resulting square losses in the ridge regression fashion. The first plot figures the ridge prediction developed from scratch, which is compared with the second plot that represents the ridge regression model of `scikit-learn`: `sklearn.linear_model.Ridge`.

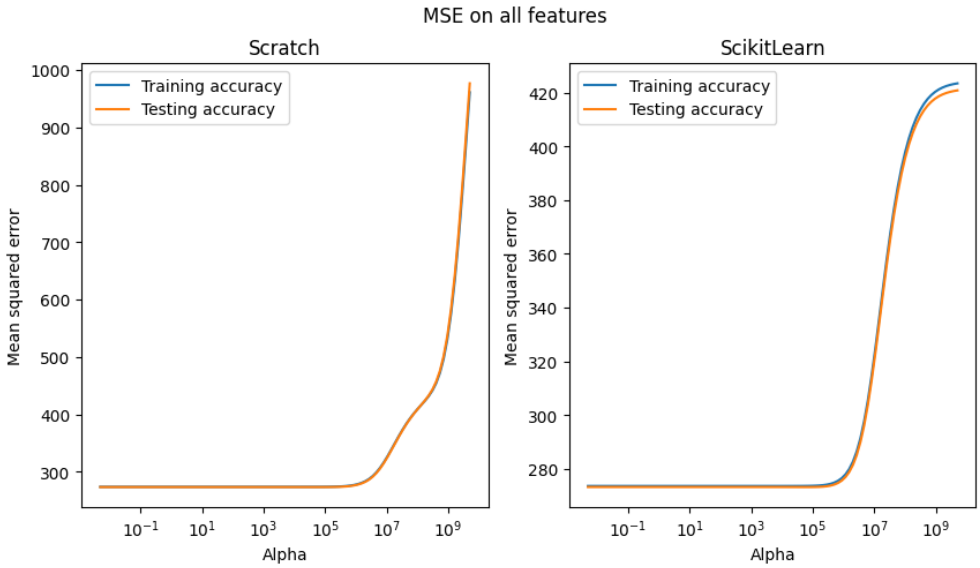

Figure 2: These plots depict the training and test losses, considering all the dataset features in a ridge regression context. The X-Axis represents the alpha coefficients, used in the estimate, while the Y-Axis represents the mean squared error given by the prediction. Note that the Y-Axis scales are different from one another.

4 K-Fold Nested Cross Validation

Cross-validation is a technique to provide a more accurate estimate of a model's performance and to prevent overfitting. This is done by testing an entire set of hyperparameters $\Theta_0 \subset \Theta$, where Θ is the set of all possible hyperparameter values. The nested CV is an extension of traditional cross-validation and is particularly useful when you want to tune hyperparameters. The code partitions the entire dataset S in K parts or folds, denoted by $i = 1, \ldots, K$. The algorithm computes the loss of the correspondent predictor on the test set S_i and it is executed on each training part $S_{-i} = S \setminus S_i$. The estimate is calculated by splitting further the training data in two subsets, S_{train} and S_{dev}. The development set S_{dev} (also called validation set) is used as a surrogate test set. The algorithm is run on S_{train} once for each value of the hyperparameter in Θ_0. The resulting predictors are tested on the S_{dev} set. In order to obtain the final predictor, the learning algorithm is run once more on the original training set S_{-i} using the value of the θ corresponding to the predictor with smallest error on the validation set. In the end, the cross validation estimate is calculated by averaging the test errors of the predictors [3].

4.1 Implementation

The Nested CV method picks as input the list of the hyperparameters *alphas*, the K number of folds and the dataset S. As mentioned above, the data are subdivided into a test set S_{test} and a training set $S_{train} = S \setminus S_{test}$ for each iteration $i = 1, \ldots, K$. In the nested part, we cut up S_{train} in CV_{train} and CV_{dev} (validation set). Then, for each $alpha \in alphas$, we predict $\boldsymbol{w}$ through ridge regression (3). We take into account the *alpha* that has generated the minor loss in the estimate. At the end of the external loop, a last prediction is made by taking up the entire training set and tested with S_{test}. Finally, after have examined all the folds, the average of the K losses is calculated and returned by the method, together with the hyperparameter *alpha* and the best predictor $\boldsymbol{w}$.

4.2 Results

The hyperparameter tuning provides an α equal to 53613, with a resulting MSE (4) of 275.39. The classifier `sklearn.linear_model.RidgeCV` delivers a MSE value of 273.35. As shown in figure 2, the choice of α does not significantly affect the rate of the MSE (9) until the order of magnitude of about 10^6.

5 Kernel Ridge Regression

The kernel trick facilitates the feature expansion without effectively transform data. This procedure makes us avoid an exponential computational cost of $N = \Theta(d^n)$, where d is the number of features and n is the number of dimensions we want to reach. In general, a kernel can be defined in this way: consider a function $\boldsymbol{\phi} : \mathbb{R}^d \to \mathcal{H}$ that maps data points on an higher feature dimensional space $\mathcal{H}$ and a general linear predictor learned through the Perceptron:

$$h(x) = sgn\left(\sum_{s \in S} y_s \boldsymbol{x_s}^\top \boldsymbol{x}\right) \tag{5}$$

where S is the set of indices of training examples $(\boldsymbol{x}_s, \boldsymbol{y}_s)$ on which the Perceptron made an update. In the space $\mathbb{R}^N \equiv \mathcal{H}$, the classifier becomes:

$$h_\phi(x) = sgn\left(\sum_{s \in S} y_s \boldsymbol{\phi(x_s)}^\top \boldsymbol{\phi(x)}\right). \tag{6}$$

The faster way to enhance the calculation speed of $\boldsymbol{\phi(x_s)}^\top \boldsymbol{\phi(x)}$ is by introducing a kernel function such that: $K(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{\phi(x)}^\top \boldsymbol{\phi(x')}$.

Some kernel functions are:

- Linear Kernel: $K_1(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{x}^\top \boldsymbol{x'}$ (7)
- Polynomial Kernel: $K_2(\boldsymbol{x}, \boldsymbol{x'}) = (1 + \boldsymbol{x}^\top \boldsymbol{x'})^n$ (8)

- Gaussian Kernel: $K_3(\boldsymbol{x}, \boldsymbol{x'}) = exp\left(-\frac{1}{2\gamma}\|\boldsymbol{x} - \boldsymbol{x'}\|^2\right)$ (9)

To "kernelize" the ridge regression predictor $\boldsymbol{w} = (\alpha I + S^\top S)^{-1} S^\top \boldsymbol{y}$, we can represent it in a generic Reproducing Kernel Hilbert Space (RKHS) [4]:

$$\langle g, \phi_K(\boldsymbol{x})\rangle_K = \boldsymbol{y}^\top (\alpha I + \boldsymbol{K})^{-1} \boldsymbol{k}(\boldsymbol{x}) \tag{10}$$

where K is the $m \times m$ matrix with entries $\boldsymbol{K}_{i,j} = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and $\boldsymbol{k}(\boldsymbol{x})$ is the vector $(K(\boldsymbol{x}_1, \boldsymbol{x}), \ldots, (K(\boldsymbol{x}_m, \boldsymbol{x}))$ of functions $K(\boldsymbol{x}_t, \boldsymbol{x}) = \langle \phi_K(\boldsymbol{x}_t, \boldsymbol{x})\rangle_K$ [5].

5.1 Implementation

In this project has been decided to deal with the Gaussian Kernel (9). For experimental purpose, we have tested two possible researches for the hyperparameter γ: the cross validation and the determination of the average norm on a number of datapoints. The last solution just cited has been carried out for the reason that if γ is relatively small to the typical euclidean squared distances between training and test points, this will imply a training error equal or close to zero, with a resulting overfitting of the predictor. At the contrary, large γ values with respect to the euclidean squared distances cause to the resulted predictors to underfit [5]. In order to respect so, we have analyzed whether or not the investigation of the average distance of the datapoints is meaningful.

5.1.1 Kernel Functions

The code part for the Kernel Ridge Regression is made up of four functions:

- `gaussian_kernel`: this function takes the hyperparameter γ and the two vectors for calculating the formula of the Gaussian Kernel (9);
- `kernel_ridge_regression`: it considers as input the dataset and the hyperparameters α and γ. The vector $\boldsymbol{y}$ is extracted from the dataset (10) by taking the `popularity` column. Then, an iteration will be made in order to take the entries and calculate the matrix $\boldsymbol{K}$ (10). It's observable that $\boldsymbol{K}(x_i, x_j) = \boldsymbol{K}(x_j, x_i)$ with $i, j = 1, \ldots, m$ for the square norm in the Gaussian Kernel formula (9). Hence, to halve the computation, we have calculated only the triangle above the matrix diagonal and summed it to its transpose part. Finally, we have summed the matrix $\boldsymbol{K}$ with the identity matrix scaled by α, then we have inverted and multiplied it by the vector $\boldsymbol{y}$.
- `kernel_predict`: in this method we take the vector $\boldsymbol{x}$ and, examining all the training points, we determining $\boldsymbol{k}(\boldsymbol{x})$ (10). In the end we obtain the estimate by factorize the `kernel_ridge_regression` output with $\boldsymbol{k}(\boldsymbol{x})$.
- `kernel_avg_square_loss`: the process of this functionality is predicting the estimate on the test set values and then calculating the MSE (4).

5.1.2 Results

In this section we are focusing on the values $\alpha = 1$ and γ that is achieved by calculating, for each number of samples in table 5.1.2, the average distance of the first n datapoints. The considered γ for the current analysis is the one got from 10000 samples. The sizes of the sets are 7500 for training and 2500 for testing.

Samples	Average Distance
100	71913
500	68506
1000	68946
5000	68477
10000	68300

The test made on the numerical features gives an MSE (9) of 392.38, with a variance of 14.57, whereas considering all the features decreases the average loss to 281.30 with a variance of 55.32. The `scikit-learn` predictor `KernelRidge` provides a test error of 281.31 with a variance of 55.57, given `RBF` as kernel function [6]. This kernel function differs from (9) because:

$$K(\boldsymbol{x}, \boldsymbol{x'}) = exp\left(-\frac{1}{2\gamma^2}\|\boldsymbol{x} - \boldsymbol{x'}\|^2\right) \tag{11}$$

The unique difference is that γ is squared. In the test phase we have simply passed to the function the root squared average norm considered for the precedent tests.

5.2 Kernel Cross Validation

The hyperparameter tuning is now done on both γ and α. The procedure is almost the same: the only difference is that the prediction and the loss are made by the kernel functions (see section 5.1.1). In order to estimate the two best hyperparameters, it has been implemented an outside loop from the nested cross validation that checks for each α its best γ value.

5.2.1 Results

For this topic, the hyperpameters are chosen from a set of values generated at random. The cross validation process is built with a number of 1000 samples, due to the huge amount of computation.

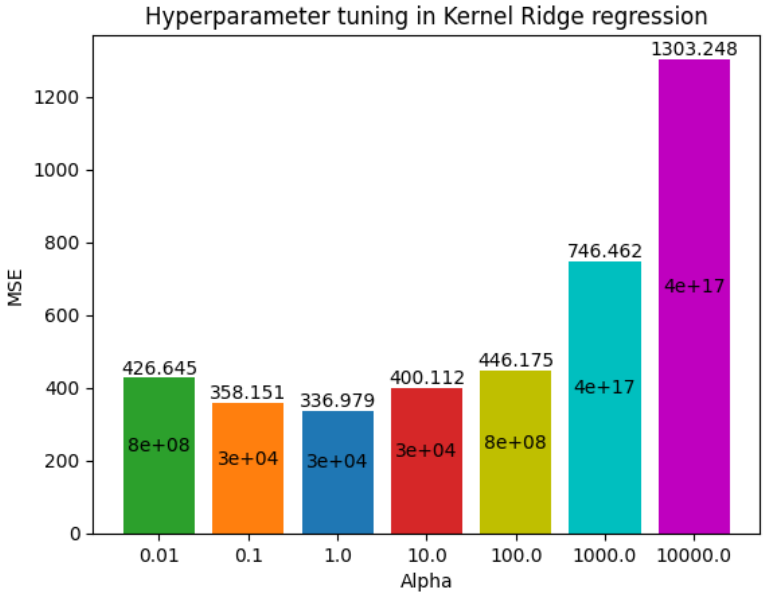

Figure 3: The histogram reveals, for each α, the respect average square loss considering the γ value that minimizes the MSE itself.

Once we have selected the best α and γ values, we re-train the model with 7500 datapoints and tested with 2500. The MSE obtained is 276.14.

6 Conclusion

The results show a sensible decrease when all the features have been considered in respect of examining only the numerical ones in the ridge regression fashion. In the figures 1 and 2 it has been compared the MSE (9) considering a wide range of the hyperparameter α, using two different ridge regression predictors. Both of them depict almost the same loss in the first α's orders of magnitude: in fact, when we deal with nested cross validation, the results are approximately equal, as illustrated in the figure 4.

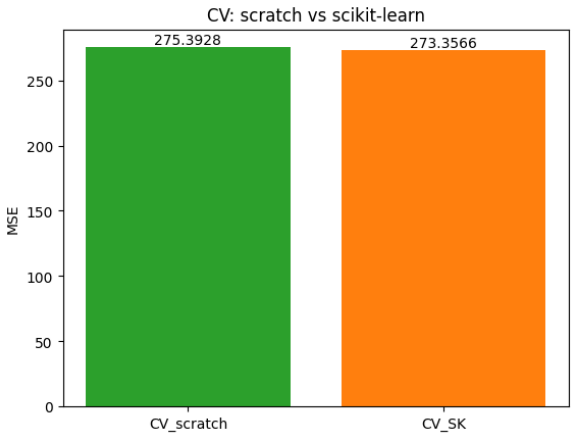

Figure 4: The histogram shows the MSEs comparison between the ridge regression model developed from scratch and `scikit-learn` ridge regressor considering all dataset features and all the datapoints.

The resultant values illustrated by the kernel computations in the figure 5 do not reflect neither a significant worsening nor an improvement in terms of loss compared to the standard linear ridge prediction. Thus, it should be identical choosing one of the two models in terms of yield. However, the ridge regression predictor takes less time to compute with respect to the kernel one. In addition, in the first one we consider all of the datapoints in the estimation, which is literally inconceivable for the kernel to process. One improvement of this work could be to take into account the kernel model considering the whole dataset, in order to verify whether or not this predictor might refine its estimations.

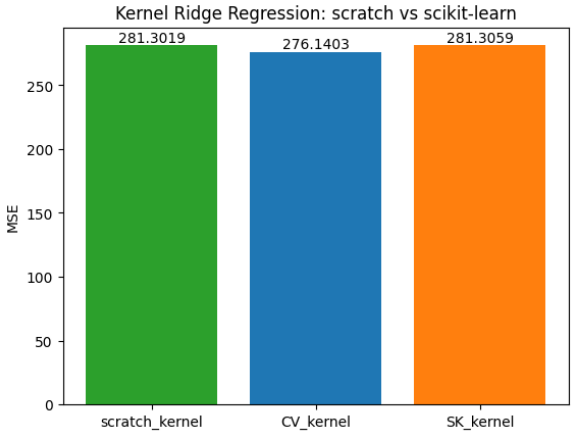

Figure 5: The histogram shows the MSEs comparison among the kernel developed from scratch (with the average square distance), the kernel with the hyperparameters achieved by cross validation and `scikit-learn` RBF kernel [6] considering all dataset features.

The experiment with contemplating the average datapoint distance as the bandwith parameter of the Gaussian Kernel γ (9) works. Indeed, as figured in the picture 5, the order of magnitude of the MSE value is equal to the one in which a nested cross validation is computed. This confirms what mentioned in the subsection 5.1: a γ close to the distance of the examined datapoints is an ideal rate for the predictor.

References

[1] Maharshi Pandya. Spotify tracks dataset, 2022.

[2] Nicolò Cesa-Bianchi. Linear predictors, 2023.

[3] Nicolò Cesa-Bianchi. Hyperparameter tuning and risk estimates, 2023.

[4] T. Kailath. Rkhs approach to detection and estimation problems–i: Deterministic signals in gaussian noise. *IEEE Transactions on Information Theory*, 17(5):530–549, 1971.

[5] Nicolò Cesa-Bianchi. Kernel functions, 2023.

[6] Rbf - scikit-learn documentation, 2023.