Università degli Studi di Milano

California Housing Prices

Statistical Methods for Machine Learning

Student
Francesco Lazzara

Degree Course
Data Science and Economics

ID
942830

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 offenses 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/us or any other person for assessment on this or any other course of study.

Contents

1 Introduction **2**

2 Theoretical Background **3**

2.1 Linear Prediction 3

2.2 Hyperparameter tuning with Cross-Validation 5

2.3 Principal Component Analsysis (PCA) 6

2.4 Nested Cross-Validation 6

3 Experimental Applications **8**

3.1 Exploratory Data Analysis and Features Transformation 10

3.2 Hyperparameter Tuning with training/test split and Cross-Validation 13

3.3 Dimensionality Reduction method: Principal Component Analysis 17

3.4 Hyperparameter Tuning with Nested Cross-Validation 21

4 Conclusion **23**

1 Introduction

SHAMPER

2 Theoretical Background

In the first section of the project, we would give reasoning about the tasks that would be performed throughout the analysis, starting from an introduction of the theoretical background for *Linear Prediction*, *Hyperparameter tuning* with *Cross-Validation* and *Principal Component Analysis*, in combination with the techniques to estimate the risk and obtain a measure of the overall performance of the predictor output by the learning algorithm.

2.1 Linear Prediction

In this setting, **Linear predictors** are learned by parametric learning algorithms, under the form of linear functions of the data points expressed as vectors of real numbers (integer/floatable data types). In mathematical terms, a linear predictor is a function $h : \mathbb{R}^d \rightarrow \mathbb{R}$ such that $h(x) = f(w^T x)$ for $w \in \mathbb{R}^d$. While, in geometrical terms a linear classifier defines an *hyperplane* $\{x \in \mathbb{R}^d : w^T \cdot x = c\}$, described by the pair (w, c), that intersects the vector w at distance $\frac{c}{||w||}$ from the origin. However, we can distinguish between *homogeneous* and *non-homogeneous* linear classifiers, depending whether or not they pass through the origin ($c = 0$). The hyperplanes of the latter class could be turned into *homogeneous* by slightly transforming our vector representation of the data points. For instance, we would learn *homogeneous* linear predictors by adding an extra feature to our data points as a column vector of ones $(1, 1, ..., 1)_{m\times 1}$, transforming the domain of the examples (x, y) in $x \in \mathbb{R}^{d+1}$. In summary, the aim of linear prediction is to learn w such that the linear predictions, computed for each observation, are as good as possible compared to the corresponding true values of the label (*ground truth*).

In a **Linear Regression** context, the predictors are parametrized by a vector of real coefficients $w \in \mathbb{R}^d$, each associated to one feature describing the data points, where the element corresponding to the extra dimension previously added could be interpreted as the *intercept* of the model. Given a training set S_m of size m, the linear regression predictor is the *Empirical Risk Minimizer* with respect to the square loss:

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

The *Bayes optimal predictor* that minimizes the statistical risk for regression problems with square loss is $f(x^*) = E[Y|X = x]$. A Loss function typically depends on the difference between the predictions and the actual labels, such that it could be considered as the "interface" between the algorithm and the data. In *Supervised Learning* scenarios, it is easier to compute the loss of a predictor since the mistakes can be defined properly, with the **Square Loss** which is defined as the average of the square of the errors:

$$\ell_D(y, \hat{y}) = (y - \hat{y})^2 \tag{2}$$

In particular, the square loss grows up quadratically for $\hat{y} \neq y$ and its first derivative appears to be more informative for the algorithm than the *Absolute Loss* one $\ell_D(y, \hat{y}) = |y - \hat{y}|$, which only gives binary information. In particular, the square loss is differentiable in every point and indicates the amount and sign of the prediction mistake $\ell'(y, \hat{y}) = -2(y - \hat{y})$, which is used to adjust the predictions. Next, we can define the vector of linear predictions as $v = (w^T \cdot x_1, ..., w^T \cdot x_m) \in \mathbb{R}^m$ and rewrite the convex optimization problem as:

$$\hat{w} = \underset{w \in \mathbb{R}^d}{argmin} \, ||v - y||^2 \tag{3}$$

If we set $F(w) = ||v - y||^2$, the ERM of linear regression is computed by solving $\nabla F(w) = 0$:

$$\hat{w} = (S^T S)^{-1} S^T y \tag{4}$$

This finding, which coincides to the *Least square solution*, holds every time the matrix $S^T S$ is *invertible*, or full-rank (nonsingular), even though it might not always be invertible. In mathematical terms, this situation corresponds to multiple solutions to the minimization problem given by the presence of *linearly dependent* vectors in $S^T S$. Still, even in the case the vector $\hat{w}$ has a closed form, it could be highly unstable to some perturbation of the training set. In other words, when $S^T S$ is barely invertible a change of few examples on S_m could affect significantly the elements of $\hat{w}$, which worsen the variance error and could potentially lead to a predictor with high bias and variance (poor overall performances). A solution to that, **Ridge Regression** introduces a regularizer term $\alpha > 0$ to control the bias of the algorithm and obtain a more stable predictor:

$$\hat{w} = \underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \|S \cdot w - y\|^2 + \alpha \cdot \|w\|^2 \tag{5}$$

The main idea of Ridge regression is to inject some bias in the model, in a controlled way, hoping to produce a larger decrease in the variance, such that creating a biased predictor could provide a more stable and resistant solution to overfitting. Of course, when $\alpha \to 0 : \hat{w}_\alpha \to \hat{w}$ the two regression problems have the same solution, while for $\alpha \to \infty : \hat{w}_\alpha = (0, 0, ..., 0)$ the solution becomes the zero vector since most of the loss is given by the second component. In other words, as α increases the elements of the $\hat{w}_\alpha$ are shrunk towards zero, thus we could think of α as a measure of the stability of the predictor. As before, we compute the gradient and we set it equal to zero:

$$\nabla(\|S \cdot w - y\|^2 + \alpha \cdot \|w\|^2) = 2S^T \cdot (Sw - y) + 2\alpha w \tag{6}$$

And we solve for w:

$$w = \hat{w}_\alpha = (\alpha I + S^T S)^{-1} \cdot S^T y \tag{7}$$

Now, the $S^T S + \alpha \cdot I$ matrix is *positive definite* and, since the eigenvalues $\lambda_1, ..., \lambda_d \geq 0$ of $S^T S$ are all non-negative, it is always invertible since its eigenvalues $(\lambda_1 + \alpha, ..., \lambda_d + \alpha)$ are all positive provided that $\alpha > 0$. In statistical terms, when all the regression coefficients in $\hat{w}_\alpha$ are penalized towards zero, the variables becomes de-correlated, thus Ridge regression can be used as a solution to the problem of *multicollinearity*, given that variables which are highly linearly related tend to provide redundant information to the model. Though, the linear regularizer term would set close to zero the elements of $\hat{w}$ corresponding to unimportant features, but not be exactly zero unless there is perfect multicollinearity.

The intercept term is included in the model, as a result of the dimension added to the data to learn homogeneous linear predictors, but typically we would prefer not to apply the regularization to this term since the regularizer term of the Ridge regression encourages the intercept $\hat{w}_0$ to be small, or equivalently shrunk towards zero. Regardless of the number of features describing the data points, having a large intercept in magnitude, is not indicative of overfitting, thus it does not make sense to shrink also the $\hat{w}_0$ element just because the model contains a large number of variables. Therefore, a small change would be made to the algorithm in order to ensure that the intercept term is not affected by the regularizer term α. In particular, the first element of the $\alpha\mathbf{I}$ matrix, which is highlighted in red, would be set equal to zero as the following image shows:

$$\begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{8}$$

Likewise, when Ridge regression is applied to a standardized training set S_m, where the features have been centered to have mean zero and standard deviation one, the estimated coefficient of the intercept $\hat{w}_0$ would be equal to the mean value of the label $\bar{y} = \frac{1}{m}\sum_{i=1}^{m} y_i$. Intuitively, the only reason we want to remove the intercept term from the model is that we would need $y_i = 0$ when $x_i = 0$ for $i = 1, ..., d$, for this reason we would exclusively focus on homogenous linear predictors on transformed features of the data points through the experimental sections.

2.2 Hyperparameter tuning with Cross-Validation

Now, we would focus on the problem of estimating $E[\ell_D(A_\alpha)]$, that is the **Expected Risk** (average) of the algorithm A_α that is run on a random training set S_m of size m (in short S), from which the expectation is taken. In particular, it measures the performance of the algorithm with a fixed value for the hyperparameter A_α on a typical training set S with a given size m. This is because when an algorithm is evaluated on a particular train set S, we are not scoring the algorithm itself but the predictor $h = A(S)$ which results from a selected train set. In other words, the performance of a predictor on a fixed training set could be better than its actual risk because of random fluctuations of data points that could make S an "good" training set, even though its risk would actually be larger if S is composed by "bad" examples, let's say. Hence, the expectations taken over all possible training set S of size m are used to estimate the risk of typical predictor output by a learning algorithm A.

In this setting, **K-fold Cross-Validation** is a powerful technique that results in a less biased model. It ensures that every observation contained in the dataset has the chance of appearing in the training and test sets. In theory, the best approximation to the generalization error of the model is when $K = m$, so that each fold in the cross validation contains exactly one observation, with the algorithm that is trained on $m - 1$ examples and tested on the held-out data point. This technique is a special case of the K-fold cross validation and it is called *Leave-One-Out Cross Validation* (LOOCV). However, the main drawbacks of this procedure is that it is computationally intensive, as it requires to fit m different models for each α in the grid of values for the hyperparameter tuning task. Instead, a widely accepted compromise with respect to the bias-variance trade-off is obtained by setting $K = \{5, 10\}$ and using the average of the resulting validation errors as an estimate for the risk. Although, this approximation tends to underestimate the risk of the learning algorithm, but typically this difference is small, so that we can rely on it.

For instance, to estimate the $E[\ell_D(A_\alpha)]$ for each algorithm $\{A_\alpha : \alpha \in \mathbf{A}\}$, where $\mathbf{A}$ is the set of possible values of the hyperparameter, we would use *K-fold (external) cross-validation*. Given S the entire dataset, we partition it in K-number of chunks $D_1, ..., D_k$, each of approximately the same size $\frac{m}{K}$. The testing part (or validation part) is D_k and is composed by the k-th fold, while $S^{(k)} = S \setminus D_k$ is the training part which contains all the data points in the folds but the ones in the k-th chunk, used for the evaluation of the predictor. Therefore, the K-fold CV estimate of $E[\ell_D(A_\alpha)]$ is computed as the mean value of the errors on the testing part $\hat{\ell}_{D_k}$ resulting at each iteration:

$$E[\ell_D(A_\alpha)] = \frac{1}{K}\sum_{k=1}^{K} \hat{\ell}_{D_k} \tag{9}$$

In particular, the best value of the hyperparameter in the grid $\hat{\alpha} \in \mathbf{A}$, used to run the algorithm A_α on the training folds, would solve the following minimization problem:

$$\hat{\alpha} = \underset{\alpha \in \mathbf{A}}{min}\, E[\ell_D(A_\alpha)] \tag{10}$$

2.3 Principal Component Analsysis (PCA)

Actually, to improve the risk estimate of the predictor output by the learning algorithm $E[\ell_D(A_{\hat{\alpha}})]$, where $\hat{\alpha}$ is the best value of the hyperparameter obtained with K-fold cross-validation, we can perform **Principal Component Analysis** (PCA). The main idea of PCA is to reduce the dimensionality of the features to a new artificial subspace, whose dimensions account for a sufficient percentage of variance explained. Coherently, we would apply a linear transformation to the features matrix and then select smaller number of directions $\mathbb{R}^d \to \mathbb{R}^{d'}$ where $d' < d$, so that we would not lose too much information with respect to the original observations in $\mathbb{R}^d$. For instance, the data points belong to a $d = 13$-dimensional space, thus we would reduce their dimensionality by projecting them onto a new space spanned by a subset of the *linearly independent* vectors, orthogonal to each other. In mathematical terms, we would exploit the *Singular Value Decomposition* (SVD), which is the generalization to non-square matrices of the *Spectral Decomposition*, in order to define the feature matrix $\mathbf{X}$ as the product of the three distinct matrices, each of them with specific characteristics:

$$\mathbf{X} = U \Sigma V^T \tag{11}$$

The matrix U contains the eigenvectors of the matrix XX^T as columns, while Σ is a diagonal matrix with the eigenvalues of the matrix $\sqrt{X^T X}$. Moreover, V is a matrix that contains the eigenvectors of $X^T X$ as columns, which are the same of U and by definition orthogonal to each other, that form a basis for a d-dimensional space. Both matrices U and V are orthonormal, namely the dot product between their column vectors is zero and they all have norm equal to one ($\|X_i\| = 1$ for $i = 1, ..., d$), while the eigenvalues in the diagonal of Σ are ordered with respect to their magnitude, which gives information about the geometry of the data. Coherently, the eigenvectors contained in U, which are associated to the correspondent eigenvalues in the Σ matrix, represent the directions onto which the data vary the most, so we could apply them as a linear transformation to the original features in order to project the data onto a new low-dimensional space spanned by the orthogonal eigenvectors which account for the largest fraction of variability of the data.

2.4 Nested Cross-Validation

Now, the minimization problem is shifted to estimate the expected risk of a learning algorithm $A_\alpha(S)$ on a training set S of fixed size m, whose examples are randomly drawn:

$$E[\underset{\alpha \in \mathbf{A}}{min}\, \ell_D(A_\alpha(S))] \tag{12}$$

The objective function in (12) measures the performance of A_α on a general training set of fixed size, among every possible subset randomly drawn from the dataset. This estimate of the risk is computed as the average validation error over all the training sets S of a given size m: $\underset{\alpha \in \mathbf{A}}{min}\, \ell_D(A_\alpha(S))$, which is equivalent to the result obtained when tuning the hyperparameters on a given training/test split.

A more computationally intensive option to estimate the risk, in terms of time, is computed through of **Nested Cross Validation**, which basically consists in using CV to solve the hyperparameter optimization problem. The main idea behind this procedure is to perform two cross-validations, one at an internal level (*inner cv*) the other at the external level (*outer cv*) on the whole set of data points at disposal. This procedure, even though is more costly with respect to a simple external cross-validation, does not return a final model, which can be use for prediction, but an unbiased generalization performance. In other words, nested cross-validation is aimed at reducing overfitting and evaluating the performance of an algorithm through a more reliable estimate, which can be achieved with a particular optimization strategy, or a less optimistic one ("more honest"), by removing the problem to choose the best hyperparameter $\hat{\alpha}$.

Therefore, we would subset the entire dataset in K-folds and we would run an inner cross-validation on the training part for each $A_\alpha : \alpha \in \mathbf{A}$, where $\mathbf{A}$ defines a grid of values for the regularizer term of the Ridge learner. Then, among all the cross-validation estimates for each A_α, we would pick the $\hat{\alpha}$ associated to the lowest risk and we would run $A_{\hat{\alpha}}$ on the entire training part of the external folds $S^{(k)} = S \setminus D_k$. Consequently, the resulting predictor would be evaluated on the outer validation fold D_k at each iteration and them the resulting validation errors would be averaged to produce the nested cross-validation estimate of the risk. The choice of the best hyperparameter is done on each fold of the external cross-validation, which does not test $A_{\hat{\alpha}}$ for a given value in $\hat{\alpha} \in \mathbf{A}$ since different values of the hyperparameter can be potentially chosen to run the algorithm on the outer level. Hence, with nested cross-validation we are not measuring the goodness of a predictor generated by the algorithm for a given $\hat{\alpha}$, instead we obtain a measure of the average risk of the different predictors when the hyperparameter is optimized on the training part through an internal cross-validation. Note that, it is not necessary that the number of folds of the outer cross-validation must be equal to the internal one, given that typically the latter is in general lower than the former $K_{inner} < K_{outer}$, even though it is not mandatory.

3 Experimental Applications

Since the beginning of our analysis we import the libraries needed to build the algorithms from scratch, which are all listed below:

- **numpy** ('np'): library useful to deal with nd-arrays and large matrices.
  - **linalg** ('LA'): sub-module of numpy that contains efficient linear algebra algorithms.
- **scipy.stats**: module mainly used for statistical purposes as it contains large number of probability distributions and statistical functions.
- **pandas** ('pd'): data-analysis and manipulation tool which uses Dataframe/Series data types.
- **matplotlib.pyplot** ('plt'): library to create and customize plots.
- **seaborn** ('sns'): data visualization library based on matplotlib.
- **random** ('rd'): module to generate pseudo-random numbers from various distributions.

Once we have downloaded the original dataset *California Housing Prices* from *Dropbox* [1], we import the ".csv" file as DataFrame object. The whole dataset contains a total of nine features, eight of them which are numeric and one categorical, plus a label *Median house prices*. The variables measured for each observation contain information about house blocks located all over the state of California, collected during the 1990 census. The variables that describe the data points do not have all the same unit of measure, for instance 'longitude' and 'latitude' are used to geographically locate the blocks according to how they are located far from north/south or east/west, respectively. While, variables such as 'total_rooms' and 'total_bedrooms' measure the sum of rooms and, more specific, bedrooms of the houses which belong to the same district. Therefore, since the features have all different unit of measures, we could think about some scaling methods to be applied before the training of the Ridge learner. Here it is a detailed list of the features and the label contained in the dataset:

- *longitude*: A measure of how far west a house is; a higher value is farther west.
- *latitude*: A measure of how far north a house is; a higher value is farther north.
- *housing_median_age*: Median age of a house within a block; a lower number is a newer building.
- *total_rooms*: Total number of rooms within a block.
- *total_bedrooms*: Total number of bedrooms within a block.
- *population*: Total number of people residing within a block.
- *households*: Total number of households, a group of people residing within a home unit, for a block.
- *median_income*: Median income for households within a block of houses (measured in tens of thousands of US Dollars).
- *ocean_proximity*: Location of the house w.r.t ocean/sea.
- *median_house_value*: Median house value for households within a block (measured in US Dollars).

[1] URL-link to the Dataset

The dataset contains a total of 20640 data points for each feature, even though 'total_bedrooms' contains 207 null values (*NaN*). We would deal with the presence of missing value by remove the entire row from the features matrix, mainly because it is not informative and can affect the techniques applied in the following sections, such as the scaling of the variables. Moreover, to directly remove the values in question we would use the `remove_null()` function, which matches the observations with *NaN* as value from a dataframe, given as only argument, and resets the index of each row. Of course, we could have used other solutions to deal with the missing values, such as setting the mean value of feature instead of the *NaN* ones, but we would prefer not to make any assumptions about their values. Therefore, once the null values are removed, the final dataset contains 20433 observations.

Now, we need to take into account the presence of the categorical attribute 'ocean_proximity', which contains five different classes: "*<1H OCEAN*", "*INLAND*", "*NEAR OCEAN*", "*NEAR BAY*" and "*ISLAND*", ordered with respect to their absolute frequency in the dataset. In regression tasks, categorical variables are commonly turned into *dummies* that represent binary variables with values $x_i \in \{0, 1\}$ for those features depending on whether a condition is satisfied or not. For example, if a house block is less than one-hour far from the Pacific Ocean, then the corresponding observation in the dataset would take value one for the dummy variable "*<1H OCEAN*", otherwise it would be zero. However, since we are dealing with five classes of the categorical variable in question, creating one feature for each of them would result in a collinearity issue, namely one or more variables would be predicted from the others (*Collinearity Trap*). This is because the last category would be already indicated by having zero on all the other four dummies, thus keeping it inside the feature matrix would add redundant information, thus it can be used as reference group. Still, since the Ridge learner would take care about any collinearity issue, we would replace the categorical attribute 'ocean_proximity' with the five dummies previously created. At this stage, we would plot the density of the house blocks over a *Latitude Longitude Map* of California, by exploiting their geographical coordinates described in the 'latitude' and 'longitude' features, from which we could see that most of the blocks are concentrated alongside the coast.

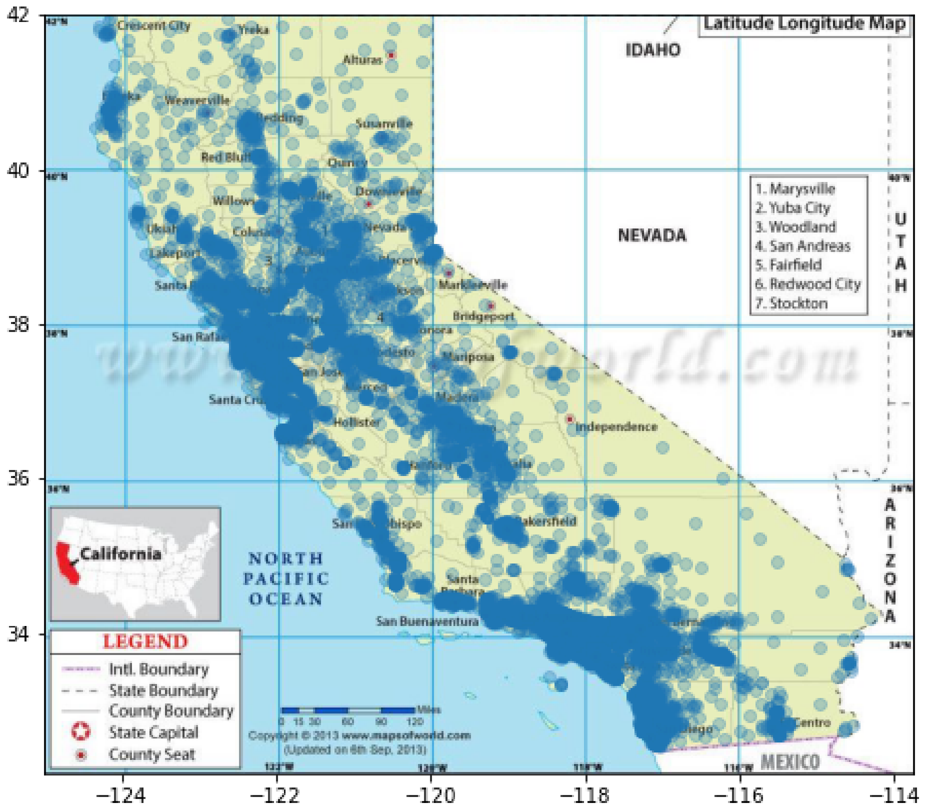

Figure 1: Density map of house blocks in California.

3.1 Exploratory Data Analysis and Features Transformation

At this stage, we would inspect the distribution of each attribute and the label by displaying their corresponding histograms, where the five dummies would be surely concentrated on the two extreme points of the unit interval. In particular, the plots referred to the two attributes "housing_median_age" and "median_house_value" revealed that they are both capped at a certain value, so that ages and prices never go beyond that limit. The attribute "housing_median_age" is capped at 52 years old, while "median_house_value" at 500.001 USD. This situation could raise the doubt of those extreme data points to be outliers, since they are far from the median value of both distributions. However, even though the techniques that are based on the squared loss function are particularly sensitive to outliers, we decide to keep the problematic observations in the dataset, because removing them without being sure about their status, could end up in reducing the available information from the data at disposal and compromise our learning procedure.

The degree of linear relationship between the numerical features and the label could be observed through the correlation matrix, which is a symmetric matrix having as entries the *Pearson Correlation Index* ρ computed between each pair of variables, interpreted as a measure of statistical association. The domain of the correlation index is between $\rho \in [0, 1]$, where $\rho = 1$ indicates perfect positive/negative linear correlation, while $\rho = 0$ corresponds to the absence of any linear correlation. From the correlation matrix it can be seen that the feature "median_income" is moderately correlated with the label with $\rho = 0.64$, while there is some strong positive and negative linear association among other attributes. However, having two or more strongly correlated variables could end up in a multicollinearity issue, since their simultaneous presence would bring redundant information when predicting the response variable. This does not only affect the estimation of the coefficients of those involved variables, but all the ones contained in the resulting predictor vector, which would affect the performance of the model. Coherently, we retrieve the variables that were strongly correlated in absolute value, having $|\rho| \geq 0.75$, for example "total_bedrooms" and "households", "total_rooms" and "total_bedrooms" or "longitude" and "latitude". Actually, we can exclude among those highly correlated variables the two geo-localization attributes because their strong negative correlation is implied by the proper shape of California, as displayed in the previous section. The density plot of house blocks upon the map of California clearly solves the doubt regarding their association measure, which could be seen as *spurious* since they do not explain each other in this case.

Regarding the other four highly correlated features, which have all $\rho > 0.85$, the description of the variables in the dataset clearly explains that when the total number of bedrooms increases even the total number of rooms does so, or when the number of households in a block increases even the population of that block does so. Then, it seems a good solution to remove three out of four variables since they all bring redundant information about the magnitude of the blocks contained in the dataset, but we know that Ridge regression will take care about this collinearity issue through the regularizer term α. This is because the main concern with multicollinearity is the variability of the estimates, given that a small fluctuation of data points in the training set S could greatly change the estimates of the coefficients in the predictor vector. In fact, what the penalty α does is biasing towards zero the estimates of the real coefficients, with the aim to reduce their variability, thus we decide to keep all the 13 features in the dataset and continue the analysis accordingly.

Now, we would introduce some notation and split the total number of observations in two non-overlapping sets. In a supervised learning scenario, a **training set** is the set of examples $S = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$ that are given as input to the learning algorithm A, which returns as outcome a predictor: $A(S) = h$. A **predictor** (or classifier) $h : X \rightarrow Y$ is a function that maps data points to labels, which is learned by the algorithm to predict the label of new data through the examples accessed in S. Coherently, a **test set** $S' = \{(x'_1, y'_1), (x'_2, y'_2), ..., (x'_n, y'_n)\}$ with size n where in general $n < m$, is used to have a measure of how the predictor behaves with unseen data, which come from the same source but are typically disjointed with respect to the training set. In particular, it is mandatory to keep the two sets separate and independent, so that the algorithm does not have access to any information about the test examples. Actually, even though the test set is not given as input to A, we are interested in designing an algorithm that outputs a predictor with a small test error, namely which has good performance when evaluated on examples not contained in the training set. Coherently, we would partition the whole set of data points into train/test set at random, by means of the `train_test()` function, created from scratch, that takes the following arguments:

- **features**: *DataFrame*; set of features describing the data points.
- **labels**: *DataFrame* or *Series*; column vector of labels corresponding to the data points.
- **test_set _proportion**: *integer*; the proportion of data points to be assigned to the test set with respect to the whole number of data points.
- **seed**: *integer*; used for the random assignments of data points to the two sets, for reproducibility of the results.

Typically, three-fourth of the observations are assigned to the training set, or equivalently one-fourth are left to the test set. Hence, we would set the argument "test_set_proportion = 0.25" that leads to a training set with cardinality $|S| = 15324$ and a test set with $|S'| = 5109$ number of examples, which would be initially kept separated in order to train our learning algorithm and evaluate the performance of the output predictor on the test portion. Before building from scratch the Ridge Regression algorithm, we would construct a class of functions to perform some data pre-processing and feature transformation, in order to compare the numerical features with different scales and units. By running the algorithm on scaled data points, the estimated weights (or the coefficients of the separating hyperplane for linear predictors) would update similarly during the training process hoping to get more accurate predictions. However, encoding the data is not a trivial task and implies some degree of arbitrariness, which depends on the type of scaling method used, and might discard some information contained in the data. Graphically, the histogram plots of the scaled features would coincide to the non-scaled ones, except for the domain of the data points, which is now smaller and would mitigate the presence of outliers. We would consider three different transformation methods to be applied to the features matrix: *Standardization*, *Normalization* and *Unit-length Scaling*.

Standardization, commonly used to compute the *Z-scores*, aims at rescaling the values of each feature X_j for $j = 1, ..., d$, by subtracting the mean and dividing by the standard deviation, such that the data are centered around the average value $\mu_j = 0$ with a unit standard deviation $\sigma_j = 1$, as the following formula shows:

$$x_j(i)' = \frac{x_j(i) - \mu_j}{\sigma_j} \tag{13}$$

Normalization, also known as Min-Max scaling, would reduce the domain of the values of each feature inside the $X_j \in [0, 1]$ interval and could mitigate the presence of outliers, by subtracting the minimum value of the features and dividing by the difference between the maximum and the minimum, as shown in the next formula:

$$x_j(i)' = \frac{x_j(i) - min(X_j)}{max(X_j)) - min(X_j))} \tag{14}$$

Unit-length Scaling is obtained by dividing the values of each predictor by the norm of the vector, which we measured according to the L_1-norm (*Manhattan distance*) in this case, since in some contexts it could be suitable, such as with the presence of histogram features or outliers among the data points:

$$x_j(i)' = \frac{x_j(i)}{||X_j||_1} \tag{15}$$

Considering that our dataset contains five dummy variables, we could worry about the alteration of their values when applying standardization and unit-length scaling, given that normalization would not change their domain as they already lie in the unit interval. Actually, standardization and unit-length scaling would not affect the relative ordering among the the values, namely their relation would not change. Though, if they are not transformed then there could be some kind of different degree of penalization made by the regularizer term α through their corresponding weights, with a lower penalty for larger ranges.

In this regression scenario, the standardization of dummy variables would not affect the results, since it would not impact on their relationships with the label. For instance, it would have concerned for *Clustering algorithms* since they are scale-dependent as well as for distance algorithms, such as *K-NN* and *SVM*, which are most affected by the range of the features to determine the similarity between data points. On the other hand, unit-length scaling would result in features with values significantly smaller in absolute terms, pretty close towards zero, as the norm of the vectors is way larger than their actual values. Still, we know that dummy variables have mean equal to $\mu_{dummy} = p$ and variance $\sigma^2_{dummy} = p \cdot (1 - p)$, where "$p$" is the proportion of one in the vector, which in most of the cases do not affect the fit of the model since are both close to zero and one respectively. In practice, since the difference between scaling or not the dummy variables would only slightly affect the regularization and the fitting of the model, we would stick to the original features and continue our analysis accordingly.

As we have previously seen, even in the feature transformation step it is essential to maintain the independence between the training and test set, by separately scaling the data points in the two samples. In particular, we would create a scaler using the `features_transformation()` class that only takes one argument:

- **scale_transform**: *string* or *NoneType*; the scaling method to be applied on the features describing the data points. Allowed values are "standardization", "normalization", "unit_length" and None for no transformation required.

Once a scaler object is created for each scaling method previously introduced, the two modules of the class `fit()` and the `test_transform()` would be applied to the training and test features, respectively, using the following arguments:

- `.fit(`**features**`)`: *DataFrame*; set of training features describing the data points.
- `.test_transform(`**test_features**`)`: *DataFrame*; set of test features to be scaled using the parameters computed by the `fit()` method.

Consistently, the `fit()` method would be used on the training data and would save the scaling parameters of the three transformation methods: the mean/standard deviation for standardization, max/min for normalization and the norm of the features for the unit-length scaling. On the contrary, the `test_transform()` method would be applied to the data points in the test set and would scale them using the saved parameters of the training portion, in order to avoid any data leakage during the model testing process. In other words, if we would use the scaling parameters of the test set, we would give some information to our learning algorithm which should have not been known before the predictor is generated. This would of course compromise the statistical significance of our results given that the independence between the two samples is violated. Accordingly, once the scaler is defined and the training features are transformed, the module in question would be applied to the complementary test set without any threat to their respective independence.

3.2 Hyperparameter Tuning with training/test split and Cross-Validation

The training data points in S would be used to output a predictor from the Ridge learner, while the test portion (S' or S_{test}) to get a proxy of its performance on unknown data through the test error. The resulting estimates would be considered to get some initial insights about the scaling methods which better describe the data and lead to the best performances in terms of predictions. Actually, the behavior of our learning algorithm could be highly variable depending on the fact that the features are scaled or not, thus we would evaluate the results in all possible cases, even though we expect the ridge learner to have obtained from the scaled features to perform better than the one from the original ones.

Now, we would move to the problem of estimating the risk of the predictor output by the Ridge learner, with an optimally chosen value of the regularizer term α on a random training set S of size m. Because of the domain of the hyperparameter $\alpha \in \mathbf{A}$ that could be very large in practice, potentially it could tend to infinity, we would choose a finite subset of equally spaced values to perform hyperparameter tuning on S. Accordingly, we would at first partition the entire dataset in training and test set with proportion 75% and 25%, respectively, then the *Validation Set* would be obtained from the training set by setting again the argument "test_set_proportion = 0.25". The three resulting sets would have cardinality $|S_{train}| = 11493$, $|S_{val}| = 1831$ and $|S'| = 5109$. At this time, we would create the three scaler objects, one for each transformation method, and we would fit them on S_{train} to save the scaling parameters and apply the `test_transform()` method both on the validation and test set, since the predictor would then be tested on S_{val} to select the best hyperparameter $\hat{\alpha}$.

To perform the **Ridge Regression** algorithm we would build the Ridge learner from scratch under the form of the class `Ridge()`, where we translated into code the theoretical procedure described in Section 2.1. The `Ridge()` object would be assigned to a variable, once the following two arguments are defined:

- **intercept** $= True$: *boolean*; "True" if the intercept should be included in the regression model.
- **alpha**: *floatable*; the value of the regularizer term (hyperparameter) of the Ridge Regression algorithm. Note that, if "alpha = 0" the $\alpha I + S^T S$ matrix might be singular (non-invertible) and the subsequent `fit()` method would return an error.

Once the `Ridge()` object is defined, then the `fit()` module would train the algorithm on a given training set S and would save in the `reg_coef` element the resulting predictor under the form of a column vector of real coefficients $w_{(d+1)\times 1}$ (when "intercept = True"), when the following parameters are specified:

- **train_scaled_features**: *DataFrame*; training features in their original form or scaled, after the application of the `features_transformation()` class.
- **labels**: *DataFrame* of *Series*; corresponding labels (response variable) of the data points in the training set.

Then, the resulting predictor would be evaluated on a test portion of data points through the `predict()` module, with the only argument "test_scaled_features" which requires the testing features under the form of a *DataFrame*, and the `pred_err()` one, applied to the actual testing labels as *DataFrame* or *Series* ("true_labels"), to return the test error of the predictor used as an estimate of the risk.

Coherently, we would exploit the `Ridge()` class and its methods to construct a loop over the values in the grid defined by $\alpha \in \mathbf{A}_{grid} = \{1, 264.1, 527.2, ..., 5000\}$, namely twenty linearly spaced values starting from 1 to 5000. In particular, we would at first track the validation errors of $h_\alpha(S_{train})$, obtained with each value of the hyperparameter in the grid, and retrieve the best one " $\hat{\alpha}$ " that leads to the lowest error when evaluated in the surrogated test set S_{val}. Next, we would run the learning algorithm with the best hyperparameter $\hat{\alpha}$ on the complete training set $S = S_{train} \bigcup S_{val}$ and evaluate the final predictor $h_S^{\hat{\alpha}}$ on the independent test set S_{test}, scaled according to the new saved parameters of S instead of S_{train}. In particular, when performing this analysis on the scaled and original features, the resulting best hyperparameter for the each transformation method was $\hat{\alpha}_{scaled} = 1.0$, while for the original variables it was equal to $\hat{\alpha}_{non-scaled} = 264.11$. The resulting test error of the predictor trained on the standardized, normalized and original features is of the e^{+09} order of magnitude, with the predictor obtained from the unit-length scaled data points which performed the worst (e^{+10}). The predictor learned from the original features had the the lowest risk estimate $\ell_D(h_S^{\hat{\alpha}}) = 4.5e^{+09}$, very close to the one obtained in the normalized case, but still those results could be an under/overestimate of the risk since they depend on the random component of the given training set S, namely on its composition and random fluctuations of the data points which could make it a "good" or "bad" training set for the learning algorithm. In this situation, the actual risk is not averaged over all possible training set of size m, but the risk estimate of the predictor highly depends on the stochastic component of the random draw of observations in the training sample. Therefore, we would perform *K-fold Cross-Validation* to estimate the expected risk of a predictor on a typical training set S, removing the random component of the sample draw by averaging the validation errors over all S of fixed size m.

As before, to perform the **K-fold Cross-Validation** algorithm we would build the `ValidationCurve()` class from scratch, by translating the theoretical concepts described in Section 2.2 into code, that would take the following arguments:

- **features**: *DataFrame*; the features describing all the data points at disposal in the dataset, without any transformation applied.
- **labels**: *DataFrame* or *Series*; corresponding labels (response variable) of the data points of the "features" argument.
- **feat_scaling**: *string* or *NoneType*; the scaling method to be applied on the features describing the data points. Allowed values are "standardization", "normalization", "unit_length" and None for no transformation required. Note that, at each iteration of the main loop the k-th test fold would be scaled according to the saved parameter(s) of the $k-1$ training folds.
- **include_intercept** $= True$: *boolean*; "True" if the intercept should be included in the regression model.

To perform *Hyperparameter Tuning* with K-fold cross-validation for the Ridge learner and inspect graphically the resulting *Validation Curve*, we would exploit the `val_curve_ridge()` method of the above class by defining the following arguments:

- **grid_alpha**: *ndarray*; the grid of values of the hyperparameter $\alpha \in \mathbf{A}_{grid}$ to be used in the "alpha" argument of the `Ridge()` class.
- **k_folds**: *integer*; the number of chunks to be created when splitting the features and labels.
- **seed**: *integer*; used for the random assignments of data points to the two sets, for reproducibility of the results.
- **show_plot** $= True$: *boolean*; "True" if the plot of the validation curve should be displayed, with the green and red shaded areas corresponding to the standard deviations of the estimates.

The method `val_curve_ridge()` would return two matrices containing the training and validation (or test) errors for each value of $\alpha \in \mathbf{A}_{grid}$ at each iteration for $k = 1, ..., K$, which would be used to draw the points of the validation curve (if "show_plot = True"), and would print a dictionary containing the best hyperparameter $\hat{\alpha}$ which leads to the lowest estimate of the risk. The graph would display the pattern of the average training and test errors at each iteration $k = 1, ..., K$ for all the different values of the hyperparameter α given assigned to the "grid_alpha" argument. In this case, we would perform *5-fold Cross-Validation* by using the previously defined grid $\alpha \in \mathbf{A}_{grid} = \{1, 264.1, 527.2, ..., 5000\}$ and focusing on homogenous linear predictors, by leaving the "include_intercept" argument to its default value. Therefore, we would at first create four objects of the `ValidationCurve()` class, corresponding to the three scaling methods and the original features, and inspect the shape of the curves for an optimally chosen subset of the initial $\mathbf{A}_{grid}$, to graphically see the distance between the training and test errors:

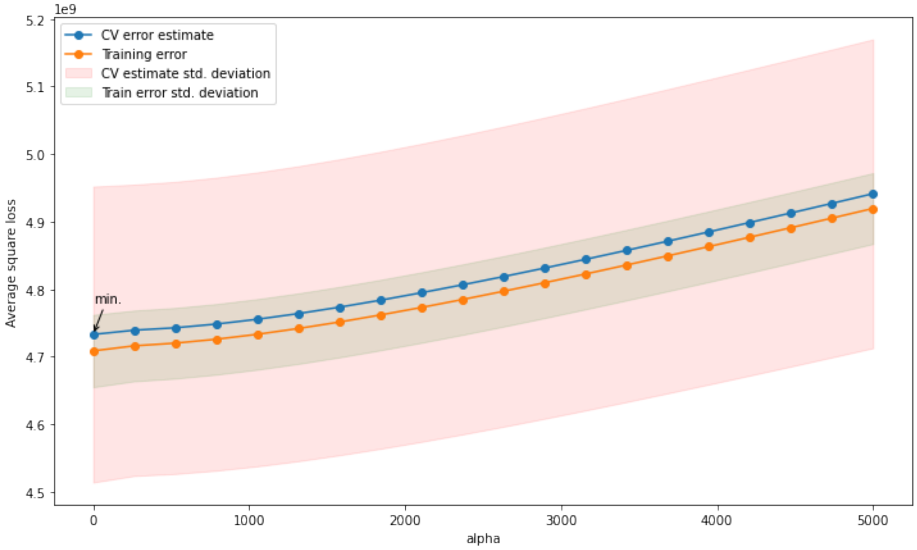

Figure 2: Validation Curve of the *original* features.

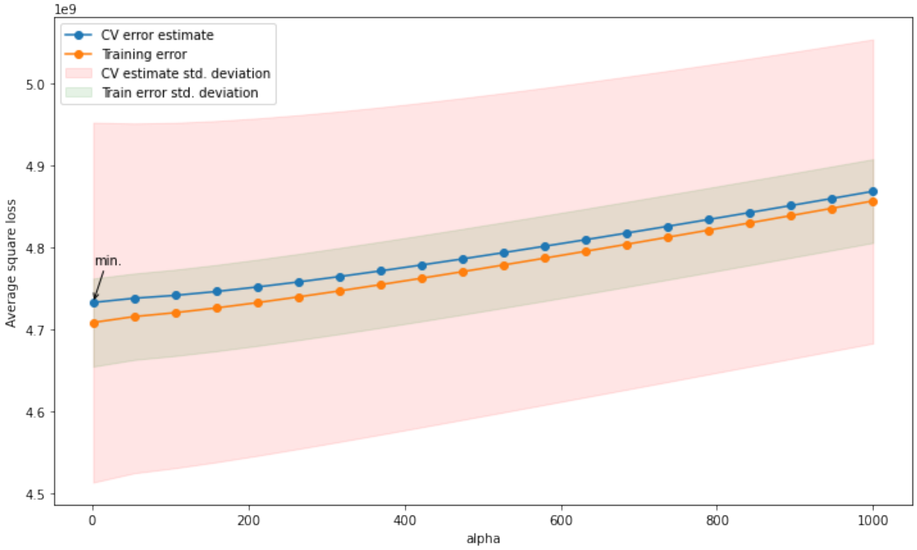

Figure 3: Validation Curve of the *standardized* features.

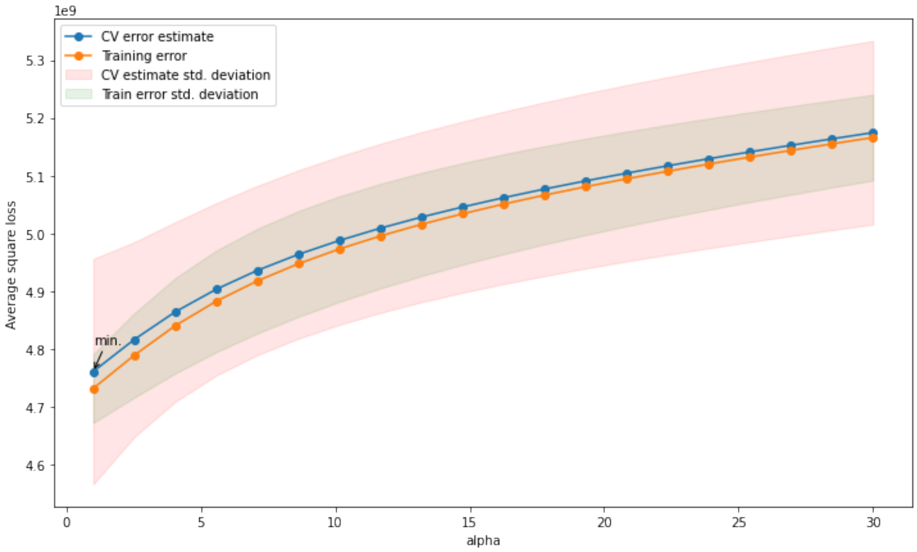

Figure 4: Validation Curve of the *normalized* features.

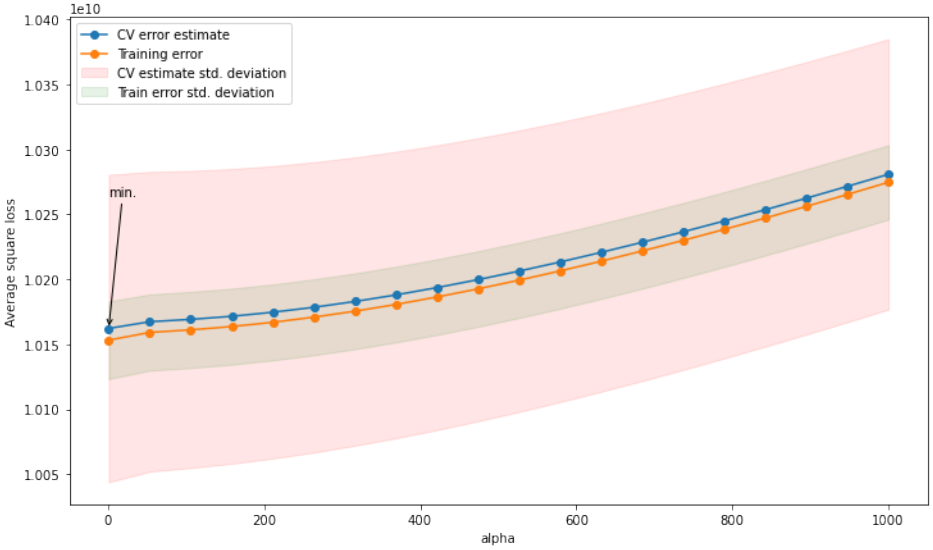

Figure 5: Validation Curve of the *unit-length scaled* features.

Surprisingly, the best value of the hyperparameter in all the four cases seems to be $\hat{\alpha} = 1.0$, even though the cross-validation estimates of the risk and the shapes of the curves are quiet different depending on the scaling method (or not), while the standard deviation of the test error estimates (red shaded area) is always larger than the training error one (green shaded area). As we can see from Figure 5, the average square loss for the unit-length scaling is the highest with respect to the others, of the same order of the one estimated before with training/test split $E[\ell_D(h_S^{\hat{\alpha}})] = 1.02e^{+10}$. Regarding the predictors obtained from the normalized features in Figure 4, we reduced the range of values for the grid of α to visually inspect the difference in estimates of the training and test errors. The corresponding validation curve is concave increasing and the resulting risk is larger than before $E[\ell_D(h_S^{\hat{\alpha}})] = 4.76e^{+09}$, with the two errors that tend to be closer as α increases, up to the point in which they coincide. Finally, the risk estimates of the remaining predictors, trained on the standardized and the original features, are approximately the same $E[\ell_D(h_S^{\hat{\alpha}})] = 4.73e^{+09}$, even though the former is slightly lower, thus we could conclude that using the non-scaled features did not result in bad overall performances of the corresponding predictor.

3.3 Dimensionality Reduction method: Principal Component Analysis

At this stage, we would exploit the dimensionality reduction method explained in Section 2.3 and build from scratch the algorithm that performs the **Principal Component Analysis** (PCA). We would try to improve the risk estimates of the predictor by building a lower subspace of artificial new dimensions, namely *principal components*, that would describe a relatively high portion of variability of the data points, without losing too much information. Coherently, to perform PCA we would build from scratch the homonym class `PCA()` that would be assigned to four variables, thus creating four `PCA()` objects according to the three transformation methods and the original features. In particular, to defined a `PCA()` object we would need a single argument representing the "scaled_features" of the training set, since the variance explained by the resulting dimensions are highly affected by variables with different units of measure, but we would return to this topic later. Once the objects are created, the `singular_values()` method would be recalled with no arguments and would perform the *Singular Value Decomposition* (SVD), by returning three matrices such that the one of training data points would be decomposed as: $\mathbf{X} = U\Sigma V^T$. In addition, the Σ matrix returned as output would be used as argument of the `variance_explained()` module, which would print the cumulative percentage of variance explained by each principal component and would display the corresponding *Screeplot*. In fact, a widely used criterion for selecting an optimal number of principal components for projecting the features is to inspect the screeplot and look for an elbow, namely a point in which the subsequent explained variance drops off relevantly and retrieve all the number of components up to that singular value. After the inspection of the screeplot, the `projected_features()` method would allow to reduce the dimensionality of the data points by applying a linear transformation to the features $T_{pca} : \mathbb{R}^d \to \mathbb{R}^{d'}$, with $d' < d$ and depends on the number of selected principal components, to be indicated as follows:

- **n_components**: *integer*; the number of principal components onto which the data points would be projected, allowed values are "n_components $\in [2, 12]$" since $d' = d$ would be meaningless.
- **show_plot** $= False$: *boolean*; "True" if the two dimensional plot of the projected features should be displayed, in combination with the condition "$n\_components = 2$".

In particular, this method would return the new low-dimensional features under the form of *DataFrame* and would graphically display the data points projected along the first two principal components, for fixed values of the two arguments indicated above. Of course, it might be that when the data are described by a large number of features, then the first two principal components would likely explain a small fraction of the overall variability, thus being not so informative. For instance, the new dimensions would be ranked in the V^T matrix on the basis of their explained variance, thus it would not be the best strategy to select the components starting from last ones, as they account for the lowest cumulative percentage of explained variance. As a consequence, plotting the new features onto the last two eigenvectors would not be so informative, resulting in a cloud of data points along the two axes. The last method of the class is `project_test()` that would be used to project the features of the test points according to the same number of dimensions selected with the previous function for the training portion, thus it only takes the test data as argument in the form of *DataFrame*.

Now, we would focus on reducing the dimensionality of the data by using the four `PCA()` objects on the whole set of data points and select the optimal number of dimensions by firstly inspecting the screeplot and then the validation curve, obtained with *5-fold Cross-Validation*, for a range of value of principal components $d' = \{2, 3, 4, ..., 12\}$. This is because the estimates of the risk obtained would highly depend on the new artificial features selected, thus it would be a more accurate strategy to inspect the behavior of the test and training errors for increasing values of d', by setting as hyperparameter of the Ridge learner the corresponding best $\hat{\alpha}$, coming from the previous analysis, for each scaling method (or original features). Coherently, we would start by inspecting the amount of variance explained by the principal components for each transformation method and in the case of the original features. However, when finding the new dimensions, PCA gives emphasis to the computation of the variance for each attribute, thus it is highly sensitive to the magnitude of the square deviations from the mean, thus we expect to see the first two principal components obtained from the original features to account for the majority of the variability. Hence, we would focus mostly on the application of principal component analysis on scaled data and inspect how the new low-dimensional subspaces approximate the data, but we would also take a look at the erroneous case of the non-scaled data.

As expected, once we compute the singular values and the cumulative explained variance for the original variables, we see that the first two principal components together account for 93% of the total variance. This is mainly driven by the fact that features like "*total_rooms*", "*population*" and "*households*" have larger mean and standard deviation with respect to the other variables, in absolute terms, thus the algorithms misleads any deviations from their average value as a direction with large variability. On the contrary, the three screeplot resulting from the application of PCA to the scaled features show that there is quietly more homogeneous distribution of the overall variability between the principal components rather than in the last case, namely there is no single principal component which account for the majority or larger percentage of explained variance. Indeed, by inspecting the corresponding screeplot of each scaling method, a slightly larger number of principal components seem to account for a sufficient amount of variance, as adding more and more dimensions do not increase relevantly the cumulative percentage of explained variance. In particular, for the standardized method four artificial dimensions account for approximately 66% of total variance explained, while this cumulative percentage increases for the normalized transformation to 78%. Though, in the case of unit-length scaling, the first two principal components are sufficient to explain 63% of the overall variability of the data, which increases to almost 99% by adding two more dimensions. This could signal the fact that unit-length scaling could not be the most appropriate transformation method to describe the data points and keep much of their original information.

We should mention that the new coordinates of the data points do not represent the original attributes anymore so, even though they may improve the performance of the models, reducing the dimensions of the features would simultaneously reduce the interpretability of the coefficients in the predictor vector, since the new directions would be computed as linear combinations of the original features. In addition, we know that PCA is not the best strategy for dealing with categorical variables as it assumes a linearly relationship between their classes, even though it might be non-linear. However, this topic is behind the scope of the project, thus we would adhere to the application of Principal Component Analysis to the features.

In alternative to the screeplot criterion, we could solve the problem of choosing the optimal number of principal components by inspecting the pattern of the risk estimate through a slightly different version of the validation curve. The parameter to tune would be the number of principal components, by computing the test and training errors for increasing number of new dimensions $d' = \{2, 3, 4, ..., 12\}$, while we would set as hyperparameter of the Ridge learner the optimal $\hat{\alpha}$ found in the four cases analyzed with the standard validation curve, for each transformation method used. Coherently, we would build from scratch the function `val_curve_pca()`, included in the `ValidationCurve()` class, that can be recalled as a method of the four objects created in the previous section where we tuned the regularizer term of the Ridge learner. The design of the class is suitable to save the best value of the regularizer term $\hat{\alpha}$, obtained from the previous `val_curve_ridge()` module, that would be used as hyperparameter of the algorithm learned on the training folds with *5-fold Cross-Validation*, for increasing number of dimensions d' at each iteration $k = 1, ..., K$. In addition, once the transformation method is indicated in the main object from the `ValidationCurve()` class, then the scaled features would be saved even for the `val_curve_pca()` method:

- **grid_princomp**: *ndarray*; the grid of values of the principal components d' to project the features onto a lower dimensional space. Allowed values are $d' \in [2, 12]$.
- **k_pcafolds** : *integer*; the number of folds created when splitting the features and labels of the whole dataset.
- **seed**: *integer*; used for the random assignments of data points to the k-folds, for reproducibility of the results.
- **show_plot** $= True$: *boolean*; "True" if the plot of the validation curve should be displayed.

What this function does is to perform PCA firstly on the scaled $k-1$ training folds, then on the k-th test fold, transformed according to the saved parameters, for each iteration of the algorithm. Before recalling the methods for the four different cases, we would expect the estimated test errors to be on certain intervals a decreasing function of the number of the principal components, since adding subsequent dimensions to the data points would initially increase the explained variance, thus reducing the estimated errors. However, as we have commented the screeplots before, there are some principal components whose explained variance is relatively low or it actually approximates zero, so it could be the case that increasing the number of dimensions would overfit the data as $d' \to 12$. Accordingly, we would recall the `val_curve_pca()` module for the four different cases, even for the erroneous one with the original features, that would automatically display the corresponding validation curves:

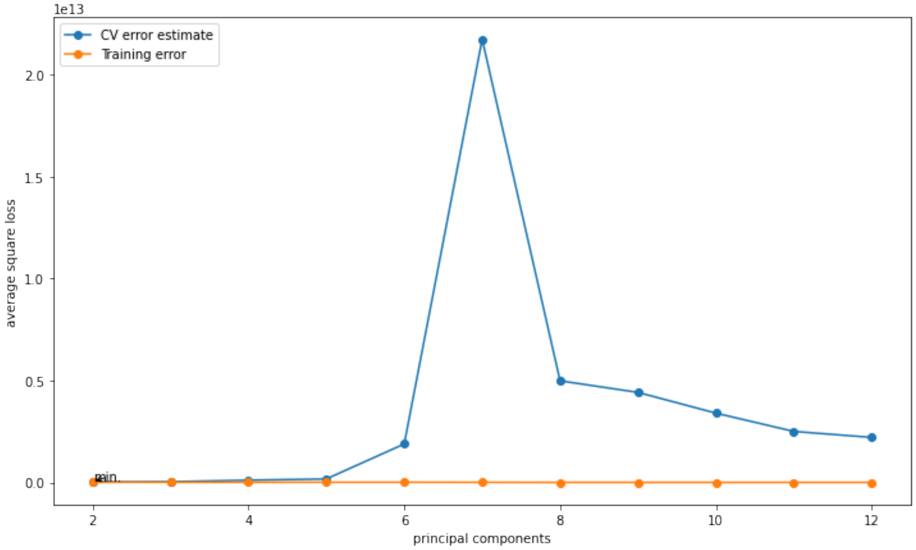

Figure 6: Principal Component Validation Curve of the *original* features.

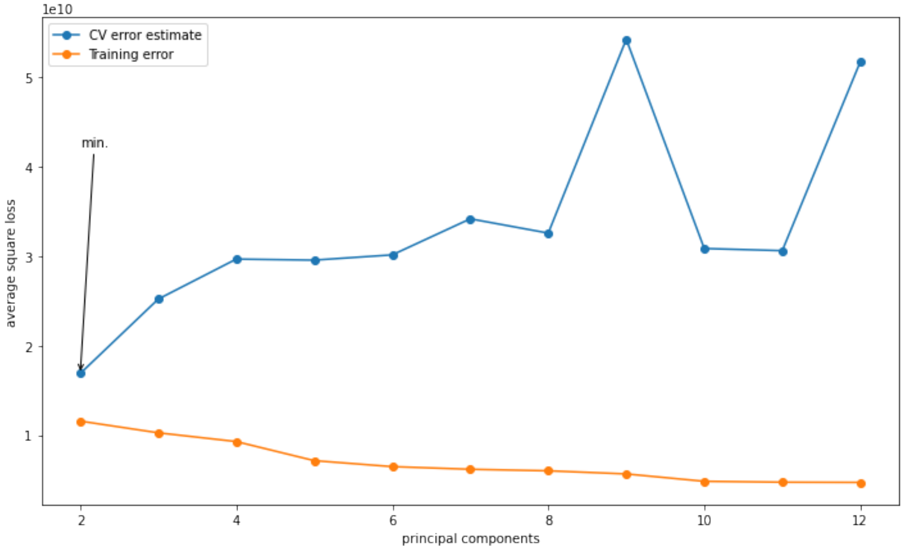

Figure 7: Principal Component Validation Curve of the *standardized* features.

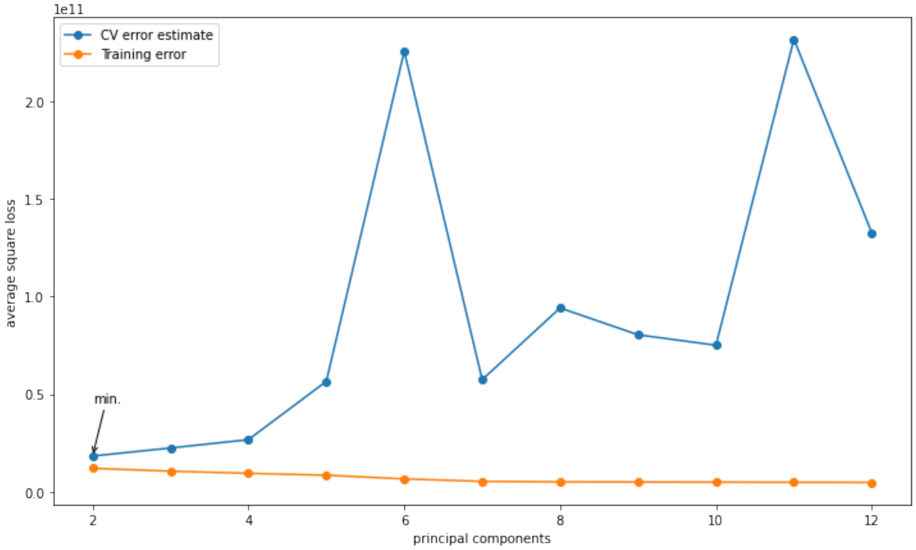

Figure 8: Principal Component Validation Curve of the *normalized* features.

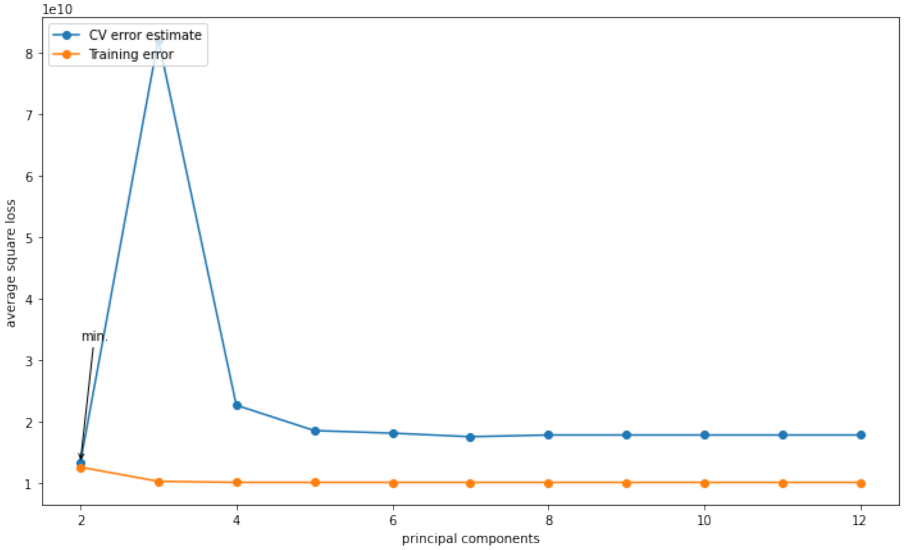

Figure 9: Principal Component Validation Curve of the *unit-length scaled* features.

The reason why the best value of the hyperparameter $\hat{\alpha}$ would be fixed for this principal component tuning is to simplify the optimization problem to a one-dimensional grid. In fact, adding the regularizer term to the grid would increase its dimensionality and make the problem more computationally intensive as the number of elements in the grid grows. Despite this fact, the best estimates of the average risk in all the four cases are obtained when the data points are projected onto the first two principal components, with the overfitting area that starts as the number of dimensions grows, where the training error which decreases regularly and the test error increases or fluctuates above its minimum. As expected, from Figure 6 we could see that the training error remains constant as the the number of dimensions increases, given by the fact that the first two principal components accounted for 93% of the total variance, with a corresponding risk estimate of $E[\ell_D(h_S^{\hat{\alpha}})] = 2.92e^{+10}$, which is the worst performance and gets even worse for $d' = 7$ where the test error reaches a peak.

Moreover, the test error of the validation curve in the unit-length scaled case (Figure 9) increases dramatically when adding one more dimension to the first two principal components, namely when the total variance explained is almost entirely accounted by the new dimensions. In accordance, before the overfitting area, the predictor reaches the lowest risk estimate among the others $E[\ell_D(h_S^{\hat{\alpha}})] = 1.33e^{+10}$, even though is still larger than all the ones obtained when tuning the hyperparameter α. On the contrary, for the standardized and normalized cases the training error decreased more smoothly for larger values of d'. Simultaneously, the overfitting of the predictors seems to be driven both by the composition of the data points in the held-out fold of each iteration (testing chunk) and by the complexity of the model, which increases as more dimensions are included, as it can be seen from the greater fluctuations of the test error. At this stage, we could conclude that the reduction in dimensionality brought by PCA, by selecting only the most informative artificial dimensions, did not improve the risk estimate, especially in the case of non-scaled features as we have clearly stated before the analysis.

3.4 Hyperparameter Tuning with Nested Cross-Validation

In this last part of the project, we would perform a more computationally intensive estimate of the average risk $E[\ell_D(h_S^{\hat{\alpha}})]$ through a **Nested Cross-Validation**. As explained in Section 2.4, this technique works by performing a two cross-validations, one at an inner level while the other at an outer one. In particular, the inner cross validation is performed at each step on the $k-1$ training folds of the outer level, with the k'-number of inner folds which could be either the same as the one of the external loop ($k' = k$) or different ($k' > k \mid k' < k$). The inner loop is exploited to reduce the bias of the ridge learner, which is trained on a smaller number of observations, and for hyperparameter tuning, in order to optimize the choice of the regularization term. Therefore, the inner cross validation would be performed for $\forall \alpha \in \mathbf{A}_{grid}$ and the hyperparameter corresponding to the model with the lowest risk estimate would be trained on the $k-1$ outer training folds, while it is evaluated on the k-th outer validation fold. Accordingly, the external cross validation would be used to evaluate the performance of the Ridge learner, with the average estimate of the validation error that would be used as nested cross-validation estimate.

To perform this final task we would build from scratch the algorithm under the form of a single function defined as `nested_cv()` that takes seven positional arguments and a default one, defined in the following list:

- **features**: *DataFrame*; the features describing all the data points at disposal in the dataset, without any transformation applied.
- **labels**: *DataFrame* or *Series*; corresponding labels (response variable) of the data points.
- **f_scaling**: *string* or *NoneType*; the scaling method to be applied on the features describing the data points. Allowed values are "standardization", "normalization", "unit_length" and None for no transformation required. Note that, at each iteration of the main loop the k-th test fold would be scaled according to the saved parameter(s) of the $k-1$ training folds.
- **grid_alpha**: *ndarray*; the grid of values of the hyperparameter $\alpha \in \mathbf{A}_{grid}$ to be used in the "alpha" argument of the `Ridge()` class.
- **k_inner**: *integer*; the number of folds/chunks to be created when splitting the training features and labels at the internal level.
- **k_outer**: *integer*; the number of folds/chunks to be created when splitting the features and labels of the whole dataset at the external level.
- **seed**: *integer*; used for the random assignments of data points to the two sets, for reproducibility of the results.
- **include_intercept** $= True$: *boolean*; "True" if the intercept should be included in the regression model.

The function would be recalled once for each transformation method including "None" for the original features, by setting the same grid used in the first cross-validation $\alpha \in \mathbf{A}_{grid} = \{1, 264.1, 527.2, ..., 5000\}$. In addition, we would split the training folds for the internal cross-validation in a smaller number of chunks with respect to the external one, so we would set "*k_inner* = 3" and "*k_outer* = 5". Note that, the function would not return a predictor that could be used for predicting new data points of an independent test set, but it would only compute an estimate of the average risk of predictors which could potentially be trained with different values of $\hat{\alpha}$ at each iteration in the outer level. Therefore, the k'-iterations of the internal cross-validation could generate a set of best hyperparameters $\hat{\alpha}_1, \hat{\alpha}_2, ..., \hat{\alpha}_k$ that would be used to run the algorithm on the outer training set and averaging the performance of the resulting predictors on the held-out test fold. The following table summarizes the risk estimates obtained with nested cross-validation:

Transformation method	**Nested Cross-Validation estimate**
None (original features)	$4.73e^{+09}$
Standardization	$4.73e^{+09}$
Normalization	$4.76e^{+09}$
Unit-length scaling	$1.02e^{+10}$

Table 1: Risk estimates from the Nested Cross-Validation.

On average, the function takes 3 seconds to run according to the parameters in our setting, while the time increases as more number of inner/outer folds are created and more elements are added to the grid of the hyperparameter $\mathbf{A}_{grid}$. As it can be seen from the Table 1, the resulting estimates appear to be almost identical to the ones obtained when tuning the hyperparameter, where $\hat{\alpha} = 1.0$, thus it could be that the best hyperparameter selected in the inner cross-validation is the same in most of the iterations, as the previous validation curve might signal since the test error is always an increasing function of α. Accordingly, the previous hierarchy of the estimates is respected, with the ones resulting from the standardized and original features which are really close to each other, while the worst average performance comes from the predictors obtained by running the algorithm on the unit-length scaled data.

4 Conclusion

SHAMPER