Università degli Studi di Milano

2019/2020

Data Science and Economics

Machine Learning Assignment

Python Application for Supervised Learning Algorithm: Ridge Regression, estimation of the risk related to it

Francesca Grazia Radatti

944128

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.

Abstract

In the following paper I am going to show the implementation of an algorithm from scratch, the Ridge Regression, and how it is possible to make some predictions and evaluate the accuracy of the algorithm.

To perform the analysis, I used Jupyter Notebook, an open-source software very useful for scientific computing in Python. Particularly, I exploited the usage of some libraries: Pandas and NumPy for linear algebra and data manipulation and Matplotlib, Seaborn, Warning and Simply for data visualization.

I studied this Supervised Learning Algorithm in order to perform some predictions, where it is necessary to fit a model that relates the response variable to features. Indeed, the main characteristic differentiating Supervised Algorithm from Unsupervised ones, is the presence of the response variable.

The algorithm taken into account is the Ridge Regression. It is a shrinkage, or regularizer, method: it literally shrinks linear coefficients, estimated with the Linear Regression, towards zero in order to regularize them.

So, using the Ridge Regression it is possible to solve some problems typically present in Linear Regression Algorithm, such as the multicollinearity.

To be honest, the Ridge Regression is not the only solution we have for “adjusting” Linear Models. It is also possible to implement some useful variable selection methods. They give an important contribution: understanding what the significant features are to perform the analysis.

Finally, I am going to explain the application of the Principal Component Analysis that allows for a small number of variables explaining a high percentage of the total variability explained by the original variables; and, as a consequence, limit the multicollinearity among variables.

Statement of the Problem

In order to reach my aim of showing how it is possible to make some predictions applying Machine Learning Algorithm, I predicted the price of houses in California.

Before going in depth to the real goal of the analysis, I explored the dataset and I prepared the data in order to avoid every incoherence in my analysis. So, I inspected firstly the variable types and their structures, then I operated some descriptive analysis in order to understand the relationship between target variable and input variables.

Then I computed the Ridge Regression and I evaluated the risk related to the prediction done with this technique. Finally, I compared the Ridge Regression with the PCR considering the accuracy of the two models.

1. An overview on the dataset

The **HousePricing Dataset**, with **20640 observations**, contains information about the price of houses in California in 1990 based on several features, including:

Variable Name	Description	Type
Longitude	measure of how far west a house is	numeric
Latitude	measure of how far north a house is	numeric
HousingMedianAge	median age of a house within a block	numeric
TotalRooms	total number of rooms within a block	numeric
TotalBedrooms	total number of bedrooms within a block	numeric
Population	total number of people residing within a block	numeric
Households	total number of households, for each home unit, for a block	numeric
MedianIncome	median income for households within a block	numeric
MedianHouseValue	Median house value for households within a block; - numeric	numeric
OceanProximity	location of the house with respect to ocean and/or sea	categorical

So, the Dataset includes 10 attributes: 9 predictor variables (marked in black in the table) and a response variable (blue mark in the table).

Requiring *"datasetname.info()"* in Jupyter, it is possible to investigate the number of observations, the variable types and the number of null values.

```
housing = pd.read_csv("C:/Users/franc/OneDrive/Desktop/DS
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Indeed, from the output it is possible to notice that:

-there are 10 attributes(columns).

-there are 20640 entries(rows).

-the variable "totalBedrooms" is the only variable that contains NULL VALUES (207 Null Values and 20433 entries).

So, I found a way to manage these null values. Since null values represent a little percentage of the variables, it is not advisable to remove the variable containing the NA values: there could be lack of information. It is possible to impute them with several techniques, on which I'll came back later on.

2. Pre-processing Data

Variable types

From the *"dataset.head()"* command it is also possible to look at the variable types: excluding "ocean_proximity" variable, all variables are numeric.

So, I had to adjust the categorical variable in order to avoid mistakes. A way to make the variable ready to be analysed is to get it to a dummy variable.

From "*housing["ocean_proximity"].value_counts()*" command it is possible to observe that the variable could have five possible values.

So, I created 5 dummies corresponding to each possible value.

Data Partitioning: training/test set splitting

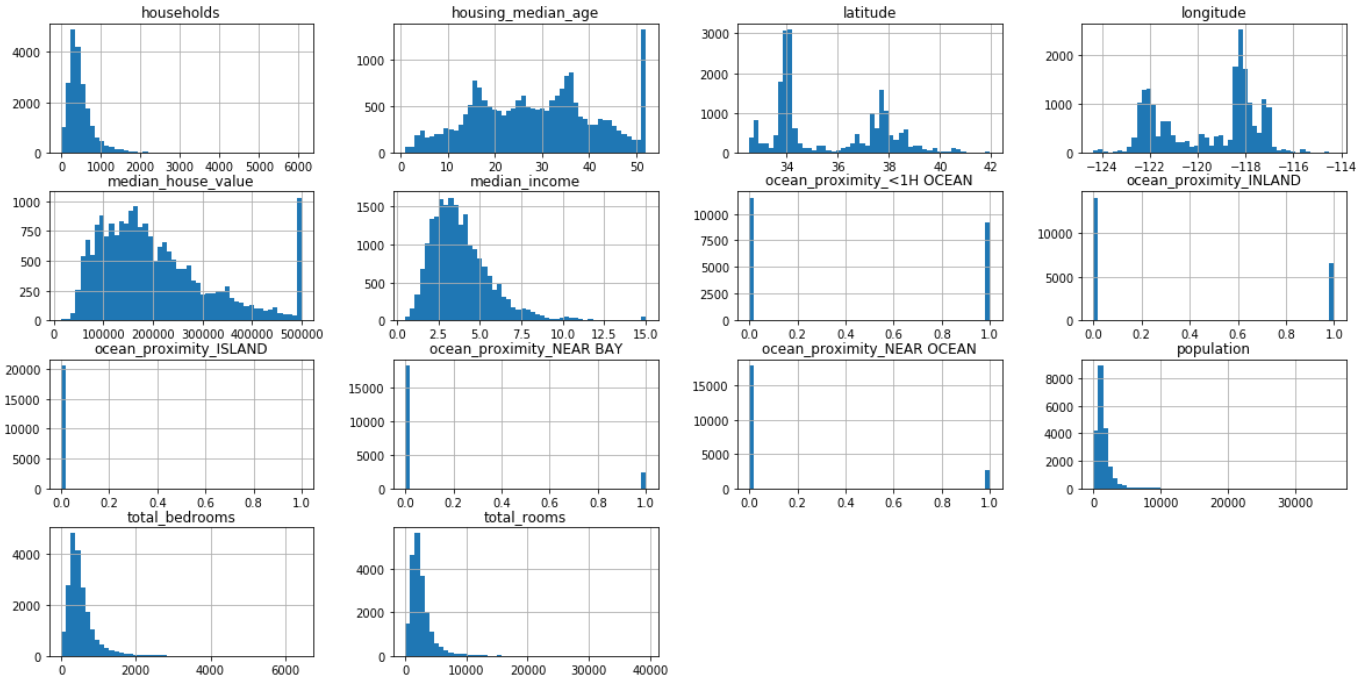
After understood the variable types, it is extremely important to **split the dataset in training and test set before doing any king of analysis**, since two types of subset are needed: one to run the algorithm and another one to test its performance.

Hence, after set the seed in order to have the same results in case of reproducibility of the dataset, I split the dataset in training and test set: I built the training set with the 60% of the total observations and the test set with the remaining part.

Descriptive Analysis

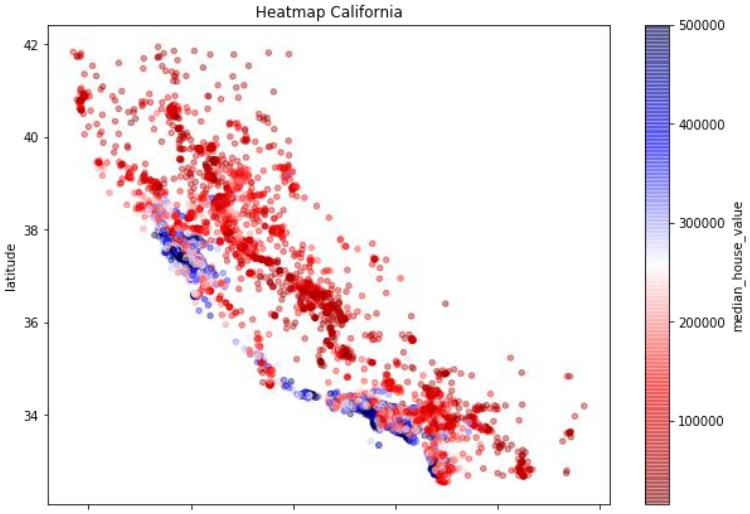
Then, using the training set observations, I required some plots in order to investigate some **information about distribution**:

1.Histograms on attributes



From the histogram it is possible to understand the **skewed distribution** of the variables: this allows to notice that the variables have different scale, so it is advisable to rescale values. Furthermore, one can notice from these histograms the **presence of outliers**, with which I'll deal later on.

2. Dataset Mapping:



The aim of the map is showing **how prices vary relatively to location characteristics**. To do this, I used a further option in the command: "cmap" allowed me to indicate different prices with several colours. Imagining the California map, it's notable that - on average - there is an **higher median house value for houses near to the ocean**(blue points); on the contrary, **moving towards inland**, the **median value of the house is lower**(red points). Based on these differences, it is reasonable to say that the variable **"ocean_proximity"** plays an **important role** in predicting median house values. Furthermore, looking at the plot, it is possible to observe that some locations near to the ocean, along California's northern most coast, have lower median house prices than other housing located also along the coast.

Considering other characteristics, such as the house square feet and population it is possible to make other consideration; I chose to use locations to have a clear visual idea about dataset.

Missing Values

Then, I proceeded with my analysis and I treated missing values.

The following output shows the missing values for training and test set:

```
In [8]: #cleaning data
        training_set.isna().sum(), test_set.isna().sum()

Out[8]: (longitude                   0
         latitude                    0
         housing_median_age          0
         total_rooms                 0
         total_bedrooms            120
         population                  0
         households                  0
         median_income               0
         median_house_value          0
         ocean_proximity_<1H OCEAN   0
         ocean_proximity_INLAND      0
         ocean_proximity_ISLAND      0
         ocean_proximity_NEAR BAY    0
         ocean_proximity_NEAR OCEAN  0
         dtype: int64,
         longitude                   0
         latitude                    0
         housing_median_age          0
         total_rooms                 0
         total_bedrooms             87
         population                  0
         households                  0
         median_income               0
         median_house_value          0
         ocean_proximity_<1H OCEAN   0
         ocean_proximity_INLAND      0
         ocean_proximity_ISLAND      0
         ocean_proximity_NEAR BAY    0
         ocean_proximity_NEAR OCEAN  0
         dtype: int64)
```

As I anticipated in the dataset description, the variable **"total_bedrooms" has 207 observations with missing data: 120 in the training set (left panel) and 87 in the test set (right panel)**, respecting the proportions of the splitting.

Having null values can lead to wrong predictions, so it is crucial to manage them. One way to manage null values is using the *fillna()* method (from Pandas), that allows to replace the null values with other values. One can substitute the null values with 0, with the mean of the corresponding column, with the median of the corresponding column, with the last non-value in the same column and so on.

Among all possible approaches, after testing them, I chose to fill the NA observations with the median of the corresponding column because, if one looks at the histogram, the major part of the observation are very close and few of them are very far(outliers). Indeed, this way is the most compatible to the dataset.

Normalizing variables: Min-Max Scaling

As It can be observed from the histograms, variables have different measurement units and, if not rescaled, they can cause biases. So, I rescaled the original values normalizing variables between 0 and 1: normalization allows to have the exact same measurement units for regression coefficients. To do this, I applied the following formula to all variables, except the dummies (already between 0 and 1):

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

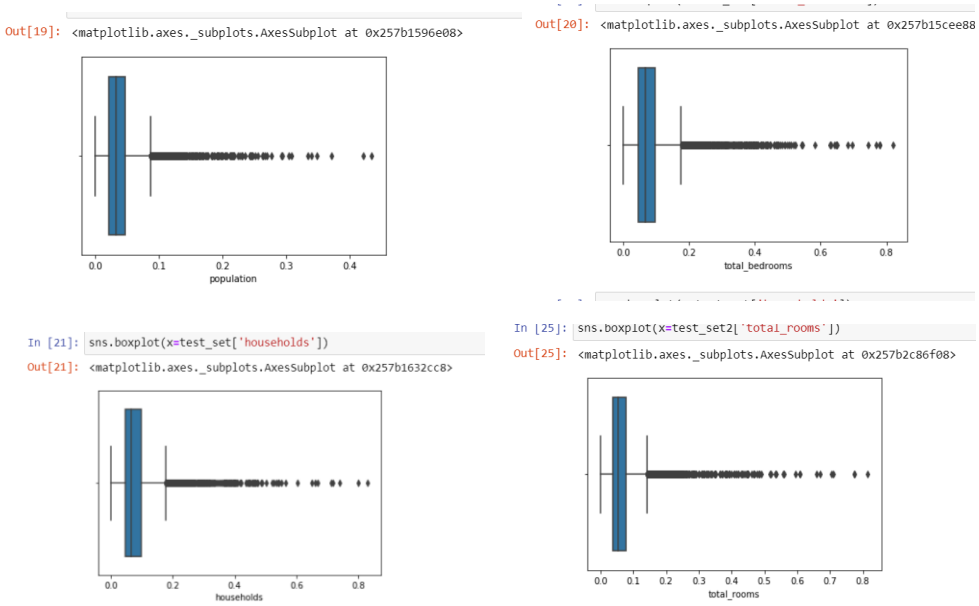
(where z_i is the new normalized variable and x_i the original one).

The new normalized values are obtained separately for training and test set. To avoid bias due to the dataset partition, for normalizing observation values belonging to the test set I used the minimum and maximum values taken from the training set: in this way it is possible to test the performance of the model combined with the pre-processing (crucial in prediction task). This ensures consistency with the transformation performed on the training data and makes it possible to evaluate if the model can generalize well.

The reason for which I chose to normalize, and not to standardize variables, is related to the interpretability of the results. For regression topics, it is often recommended to centre the variables so that the predictor variables have mean 0. This makes it so the intercept term is interpreted as the expected value of Yi when the values of the predictor variables are set to their means. Otherwise, the intercept is interpreted as the expected value of Yi when the predictor variables are set to 0, which may not be a realistic or interpretable situation.

Outliers

It's possible to notice that normalized values in the test set are not perfectly included in the range [0,1]. The reason could be t to the fact that, to normalize them, there are used the minimum and maximum values evaluated on the training set, that contains outliers. Indeed one of the main problems of Min-Max Scaling is related to the management of the outliers. Since I don't know the data distribution, I generated some box plots on variables in order to check outliers presence:



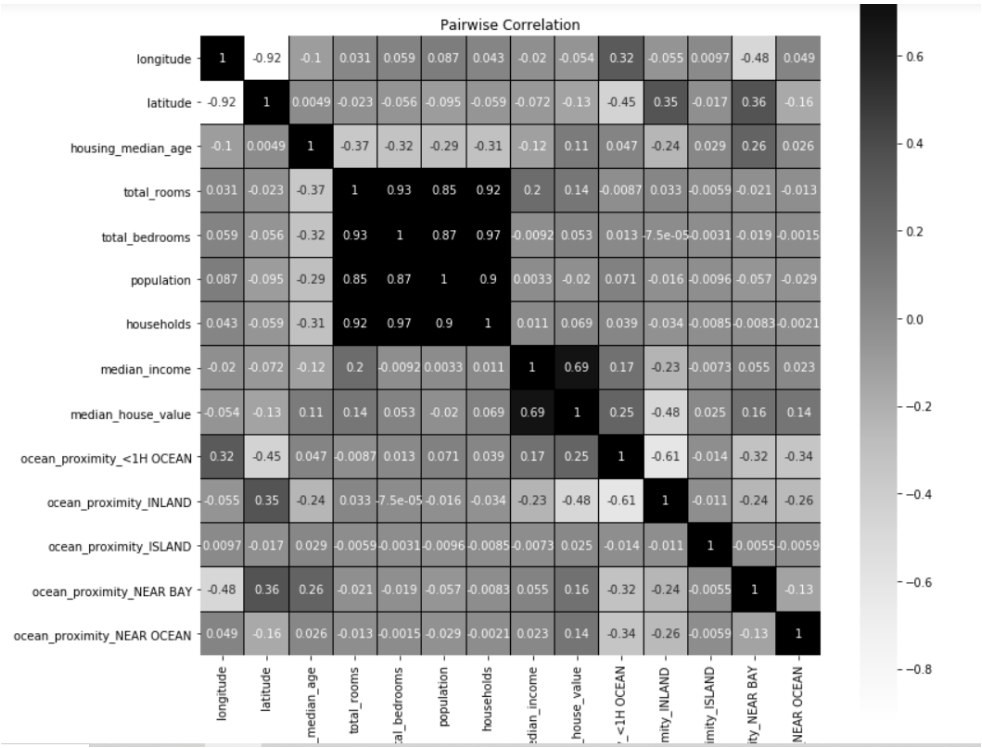
Boxplots display the distribution of data based on five numbers: minimum, first quartile (Q1), median, third quartile (Q3), and maximum.

From the boxplots shown above, it can be seen that **4 variables have outliers and there are the points outside the whiskers** (whiskers correspond to minimum and maximum values).

So, I made some proofs to delete the outliers, but I noticed from analysis that the results don't change. So, I decided to not remove them.

Pairwise correlation between features: from matrix to heatmap

The following heatmap shows the correlation between variables performed applying the *corr()* function to the training set:



The pairwise correlation could have a value included in the interval [-1;+1]: there is perfect negative correlation when the index is equal to -1, perfect positive correlation when the index is equal to 1 and if the result is equal to zero variables are independent.

Looking at the heatmap on the correlation matrix, we can conclude that there is high correlation between the following variables:

- total_bedrooms with total_rooms, households and population;
- total_rooms with population and households;
- population with households;

Requiring the correlation indices higher than 0.75 (that indicates a strong correlation), it is possible to have a clearer visualization about significant correlation:

```
In [93]: corr[((corr > 0.75) | (corr < -0.75)) & (corr != 1.0)].dropna(axis='index', how='all')
```

Out[93]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_
longitude	NaN	-0.924656	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
latitude	-0.924656	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
total_rooms	NaN	NaN	NaN	NaN	0.925057	0.850026	0.916848	NaN	NaN	
total_bedrooms	NaN	NaN	NaN	0.925057	NaN	0.865660	0.972584	NaN	NaN	
population	NaN	NaN	NaN	0.850026	0.865660	NaN	0.902337	NaN	NaN	
households	NaN	NaN	NaN	0.916848	0.972584	0.902337	NaN	NaN	NaN	

So, having a look at the correlation indices obtained, it could seem to be reasonable to omit some of the correlated variables; however, I prefer not to omit variables because they influence the house pricing and omitting variables there could be bias caused by lack of information.

Furthermore, since I performed the prediction using the Ridge Regression, I haven't had any multicollinearity problem.

2. Ridge Regression

The Ridge Regression is a **regularized loss minimization**(RML) machine learning problem which add a **stabilizer to the empirical risk minimizer (ERM**). More conventionally is called Shrinkage Method and it has the aim of adjusting **the Bias-Variance trade off:** shrinking coefficients towards zero, the tuning parameter increases the bias with respect OLS estimate(with quasi-null bias and very high variance), but this increase is compensated by the variance cut down.

So, this approach is very similar to least squares, the difference is referred to **the tuning parameter (λ)** added to RSS (residuals sum of squares) for calculating the coefficients. Indeed, when λ=0, it has not any effect on the coefficients, while, as the term goes to infinity, the effect of regularization grows and the coefficient become smaller and smaller. But the coefficients never reach the zero value (as it can happen in the Lasso Regression). The tuning parameter, λ, regularizes the functional Empirical Risk Minimizer making more stable the model.

2.1 Variables definition

Firstly, I separated the label (y) I wanted to predict, from the predictor variables set (X) for both the training and the test set. (with 12384 entries and 8256 respectively).

2.2 Going in depth: Ridge Regression

After defining examples, I run the Ridge Regression algorithm on the training part, and then I predicted and evaluated the performance on the test part.

The aim of my analysis consists in the implementation of the algorithm from scratch using Pandas and NumPy libraries and applying it. The first step consists in estimated the ridge coefficients varying λ; in order to estimate the ridge coefficients, the following formula is needed:

$$\beta_R = (X^T X + \lambda \mathrm{I})^{-1} * X^T * y$$

From results given by the *bhat_ridge(lamb)* formula, it's possible to look at how coefficients are shrinked to 0 from the tuning parameter, as is shown below:

```
In [40]: bhat1=bhat_ridge(0)
         bhat2=bhat_ridge(0.05)
         bhat3=bhat_ridge(0.5)
         bhat4=bhat_ridge(5)
         bhat5=bhat_ridge(50)
         bhat6=bhat_ridge(100)
         bhat7=bhat_ridge(500)
         bhat8=bhat_ridge(1000)
         bhat9=bhat_ridge(10000)

         bhat1,bhat2,bhat3,bhat4,bhat5,bhat6,bhat7,bhat8,bhat9

Out[40]: (array([-0.46638947, -0.41485449,  0.12141633, -0.40864922,  0.91268218,
                 -2.55636757,  0.92528627,  1.17100221,  0.42078215,  0.3348342 ,
                  0.41694092,  0.43852317]),
          array([-0.46304906, -0.41142592,  0.12156993, -0.40409988,  0.91556559,
                 -2.49506215,  0.88997677,  1.17067504,  0.4180308 ,  0.33156865,
                  0.41447818,  0.43606881]),
          array([-0.43380124, -0.38217595,  0.12273327, -0.35520655,  0.88913359,
                 -2.06455023,  0.67620331,  1.16636399,  0.39475062,  0.30397215,
                  0.39338488,  0.41508732]),
          array([-0.26107494, -0.21662764,  0.12694929, -0.0827918 ,  0.47707558,
                 -0.77039583,  0.27156581,  1.12874967,  0.26625851,  0.15125848,
                  0.27325352,  0.29601165]),
          array([-0.01077388,  0.01871655,  0.11449174,  0.09810372,  0.12902035,
                 -0.07901934,  0.10234718,  0.93285457,  0.13512682, -0.0246608 ,
                  0.14881652,  0.16909857]),
          array([ 0.04493551,  0.06966054,  0.10778115,  0.09725539,  0.09283322,
                 -0.02735808,  0.08113834,  0.79364069,  0.13424752, -0.0405875 ,
                  0.14629211,  0.16408207]),
          array([ 0.1290155 ,  0.13282978,  0.13164822,  0.06450238,  0.05570236,
                  0.01169983,  0.05423245,  0.40514597,  0.1550006 , -0.03585453
```

Then, I predicted the housing price value on the test set typing in Jupyter *"y= b@x_test"*

In order to be computationally faster, I included all steps in the *ridge_regression(lamb)* function:

```
In [60]: def ridge_regression(lamb):
             y_pred_ridge = X_test @ (((LA.inv((X_train_t@X_train)+lamb*I))@X_train_t)@y_train)
             return y_pred_ridge
```

Considering "lamb" (referred to λ values) the parameter of the function, I have the possibility to observe how estimated values vary as λ moves from 0(Linear Regression) to ∞ (never reached).

2.3 Cross-validation risk estimate

The next step consists in evaluating what is the optimal value for λ such that the cross-validation error is the smallest.

What's the Cross-Validation? The Cross-validation is a statistical method used to estimate the test error associated to a machine learning algorithm and its predictive performance.

What's the test error? The test error is the average error that results from using a machine learning method to predict the response on a new observation. Formally it corresponds to the average loss of the predictor $\hat{f}$ on the test set. Assuming that the test set is generated through independent draws from D, the test error corresponds to the sample mean of the risk of the algorithm.

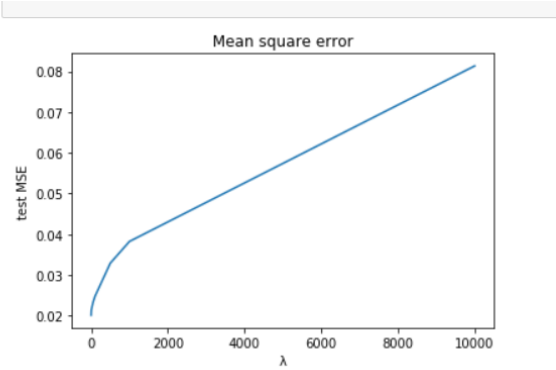
How is it possible to estimate the test error?

One of the several ways to estimate the test error is The Validation Set Approach: being an exhaustive method, it's based on partitioning the dataset in training and test set. The model is fit on the training part and predictions are assessed on the test part. Often, and much more in the case of quantitative response variables, **the test error is estimated using the test Mean Square Error**: the

average squared prediction error on the test observations (x_t, y_t). It is measured **with the formula of the quadratic loss function**:

$$\text{Test MSE} = E[(Y_t - \hat{f}(X_t))^2] \qquad (2)$$

In the following graph it is possible to observe how the test MSE changes as a function of λ and so how the **cross-validation risk estimate depends on the values given to the tuning parameter**. As λ increases test MSE increases. At **λ=0**, the **MSE reaches the lowest value** and this indicates that **0 is the optimal value for λ**. When λ is equal to zero, the Ridge Regression coincides with the Linear Regression and do in this case **it's better to apply a linear regression**.



So, I checked the Test Mean Square Error creating the function *"mse(lamb)"* and the prediction is evaluated across all new unseen example pairs (X_t, Y_t). In order to reduce the risk, the goal is to select the tuning parameter for which the test MSE is the lowest.

Overfitting/Underfitting

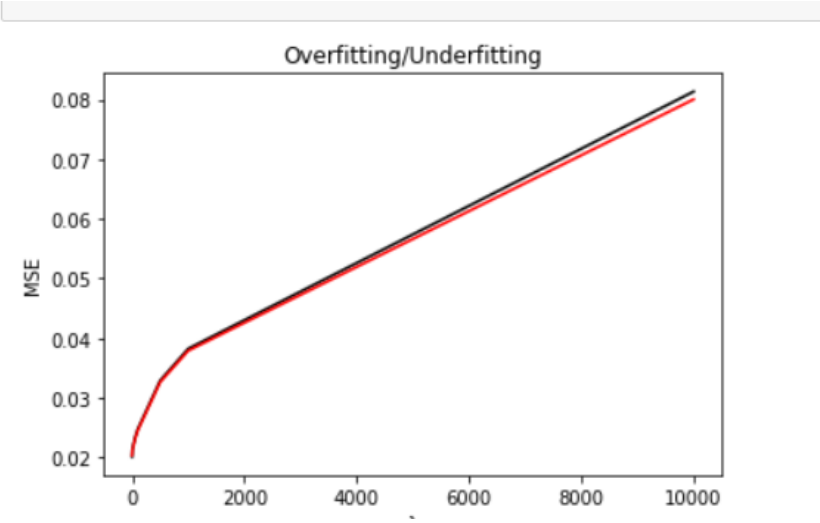
Unfortunately, it is difficult to calculate the test MSE. When it's not possible to compute it, one can evaluate algorithm performances measuring the Training MSE, but not always: there is no guarantee that the algorithm with the lowest training MSE will also be the model with the lowest test MSE because of the bias-variance trade-off. For this reason, the **comparison between training error and test error is important in order to choose a tuning parameter that avoid underfitting and overfitting**, since they can lead poor algorithm performance. **Overfitting** refers to a model that fit the training data too well and so the **variance error is higher** with respect to the bias error. **Underfitting** refers to a model that can neither model the training data nor generalize to new data and so the **bias error prevails** in the Bias-Variance trade off.

As I said previously, the test error is equal to the average squared prediction error on the test observations (x_t, y_t) (2)

On the contrary, the training error is expressed by the following formula:

$$\text{MSE}_{\text{training}} = \frac{1}{n}\sum_{t=1}^{N}((Y_t - \hat{f}(X_t))^2 \qquad (3)$$

The following plot shows as the performance of the estimator on the unknown test set is different from the performance on the training part.



It can be observed that, **for any λ value**, the **training error** (red curve) is **lower** or equal to the test error and this is reasonable, since the training error is based on known data.

Moreover, in the **extreme left part**(for low values of the tuning parameter), the training error and the test error are very low and **they almost overlap**: there's a good estimate of the test error and, as a consequence, there's almost no variance with curves very near each other. For **λ>5**, the **black curve begins to step away and both the errors are increasing rapidly**: there is **underfitting** and so bias increases.

3. Principal Component Analysis

Then, I checked a possible **way to reduce the risk of the prediction**, hence I computed the **PCA**. Often, it could be a good technique for improving the risk of prediction.

This approach involves the **identification of first M principal components**, $Z_1, \ldots\ldots, Z_M$ and then **using these components as the predictor variables**. The aim of the approach is to use a small number of principal components **that explain most of the variability in the data**. If fitting the model to $Z_1, \ldots\ldots, Z_M$ lead to better results than fitting a least squares model to $X_1, \ldots\ldots, X_p$ original predictor variables, most or all the information in the data that relates to the response is contained in $Z_1, \ldots\ldots, Z_M$ and only M≪p coefficients are needed.

Hence, I continued my analysis projecting data onto a lower-dimensional space without losing too much information.

3.1 First step: singular values decomposition (SVD):

The **SVD is a is a factorization of a real or complex matrix** that generalizes the eigen decomposition of a square normal matrix to any *m x n* matrix **via an extension of the polar decomposition**.

It could be applied on symmetric and square matrices.

Analytically, the **SVD applying the Spectral Theorem Decomposition**, is the following:

$$X = U\Sigma V^T = \sum_{i=1}^{d} \sigma_i u_i v_i^T \quad (4)$$

Where:

- V and U are orthonormal matrices;
- the columns of **U** are the **eigenvectors** of XX^T;
- the columns of V are the eigenvectors of $X^T X$;
- the diagonal elements of Σ are the **eigenvalues** of $\sqrt{XX^T}$ or, identically, $\sqrt{X^T X}$;

So, I did the SVD in Python using the NumPy function: *"LA.svd"* (from Linear Algebra) for both training and test set. From the function **I required eigenvalues**:

```
Matrix(np.diag(np.round(s, decimals=2)))
```

Out[55]:

115.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	68.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	47.55	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	41.57	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	25.34	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	22.62	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	13.18	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	10.43	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.64	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.64	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.59	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.99

```
Matrix(np.diag(np.round(s_test, decimals=2)))
```

Out[64]:

94.03	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	56.07	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	39.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	34.26	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	20.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	18.46	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	10.79	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	8.49	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.17	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.28	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.16	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.72

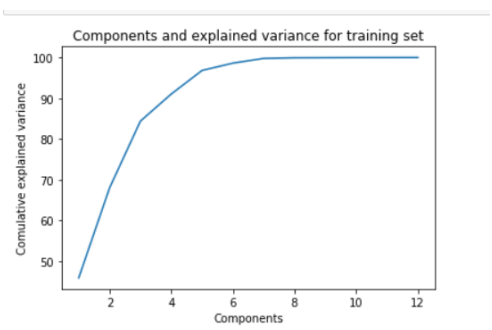
The table shows, on the diagonal of the Σ matrix, the square root of the eigenvalues of the original matrix and its transpose (they have the same eigenvalues due to symmetrisation).

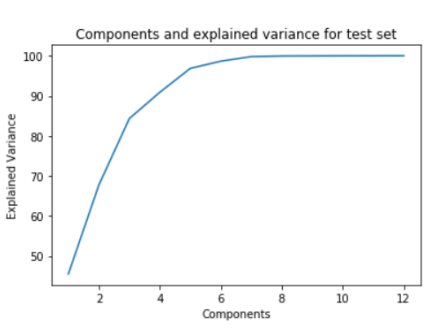
Looking at the table it is possible to observe that, going from left to right of the matrix, the **singular values are decreasing** and it could be a good information about the geometry and direction of the data: in the first direction, the data matrix is more elongated than the others.

3.2 Principal components choice

The **choice of the principal components** is taken **looking at the explained cumulative variance**.

So, I computed the cumulative variance for both the training set and the test set:





The first **plot shows variability explained, step by step, when an additional component is included in the model** for the training set; while, looking at the second graph it is observable the cumulative variance explained by test set predictor variables. The plots look very similar and the variances explained are the following:

```
In [61]: cum_variance_explained = np.cumsum(variance_explained)
         print(cum_variance_explained)

[ 45.87966377  67.96347043  84.36154277  91.00735498  96.8056277
  98.59827965  99.75733067  99.90613328  99.91527891  99.95056624
  99.97463354 100.         100.        ]
```

```
In [71]: cum_variance_explained_test = np.cumsum(variance_explained_test)

         cum_variance_explained_test

Out[71]: array([ 45.4839943 ,  67.6877567 ,  84.29854992,  90.86101032,
                 96.82545184,  98.6263192 ,  99.76386127,  99.91528563,
                 99.95038935,  99.9736354 ,  99.99271899, 100.        ,
                100.        ])
```

The first output reports the cumulative variances measured on the training set and in the second one are displayed the cumulative variances explained computed on the test set. To what concern the training set, the **first principal components it's the most informative**: it explains the 45,88% of the total variance; then, as the second component is added, the total variance explained raises to 67,96%, so **the additional variance explained by the second component is much smaller than the variance explained by the first component**. Continuing adding a new component in the model, the total variance explained increases and the model with all the components explains the 100% of the variance.

By the way, the **principal component analysis has the aim of explaining a good quantity of the variance with as less as possible features**. Indeed it can be seen, from the first picture, that adding a third principal component increase the explained variance to 84,36% and adding the 4^{th} the increase is not significant(less than 7%), so the 4^{th} principal components it's not useful for our aim. So, I select PCs until they add a significative explanation of the variance, then stop. So, **PC1, PC2 and PC3**, projected on the dataset, **could well replace the original 13 predictor variables** since they explain most of all the information.

Another way to choose the useful and significant principal components is following the **"Elbow Rule"** in the graphs about cumulative variances: the optimal number of components is indicated by the elbow in the plot of the cumulative variances. Indeed, it can be observed that the elbow occurs when the 3^{rd} component is added.

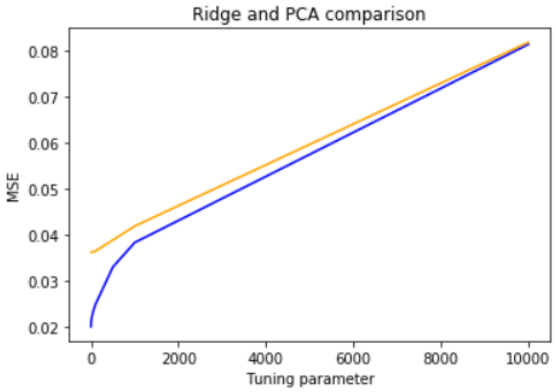
However, there is **not an absolutely rule** and the best **choice depends on the aim**: in my case the aim is that of **minimizing the risk of prediction**. So, after various proofs with different number of principal components that could well replace the original features, the **number of principal components to which is associate the lowest error is six**, to which correspond the 98,59% of the total variability.

For the test set, can be used the same logic and also, six principal component are that which guarantee the minimum risk of the prediction.

Is this technique improving with respect to Ridge Regression?

Now my aim is to check if adding six components could improve the performance with respect to ridge regression algorithm on the original predictor variables.

So, I **projected the components on the training and test set creating new**, reduced, **sets of data**. Then, I computed the **MSE test to evaluate the performance** of the model and I compared it with MSE test of the ridge regression:



λ	MSE_{RIDGE}	MSE_{PCA}
0	0.0199420	0.03614135
0.05	0.0199605	0.03614131
0.5	0.020137	0.0361410
5	0.0212757	0.03613866
50	0.02324	0.0361866
100	0.0247248	0.0363489
500	0.0328918	0.038763
1000	0.0382267	0.041791
10000	0.08133	0.081768

The graph shows how MSE_{RIDGE} and MSE_{PCA} vary as λ increases from 0 to ∞. The blue curve reports MSE_{RIDGE} values and the MSE_{PCA} results are summarized by the orange curve.

Looking also at the table on the right, it is possible to conclude that there is not an improvement: the **MSE obtained from Ridge is always lower**. So, I can conclude, for my data and my results that **principal components cannot efficiently replace the original features in order to reduce the risk**, not in Linear Regression (λ=0), nor in Ridge Regression (λ>0).

Conclusion

Using the Californian Housing Pricing, I predicted the value of the mean housing price (my target variable) running a Ridge Regression Learning Algorithm that allowed me to relate the input variables to the target variable. Then, I evaluated the risk of the prediction using different values of the tuning parameter(λ); the addition of λ aims to solve some problems typical of the least square estimate. Finally, I selected the model with a value of λ such that the resulting cross validation error is the lowest, this kind of approach permits me to maximize the accuracy of the prediction. Then I tried to further reduce the risk evaluated with the previous approach, using the Principal Component Analysis. Hence, I firstly selected the useful principal components and then I projected these new components to my data obtaining a new dataset, with less observation but almost the same information explained. Secondly, I investigated how the models differ upon changing the values of shrinkage parameter. Finally, I compared the MSE resulting from Ridge Regression and Principal Component Regression.

The evaluation of several models is necessary to make some good analyses since there does not exist an absolutely best model: the choice of the best model depends on the dataset characteristics and the aim of the analysis.

GitHub link:

https://github.com/FrancescaGraziaRadatti/ml_project_francescagraziaradatti/blob/main/ml_project_francescagraziaradatti_RR.ipynb

Openining the link it's possible to have access to the repository containing the codes.