Algorithms for massive datasets
Statistical methods for ML

Joint project for 2019-20

Matilde Sacchi, Luigi Tropiano

January 2021

Contents

1 Introduction

The task of the project is to implement from scratch a learning algorithm for regression with square loss (e.g., ridge regression). The techniques used to infer the predictor is required to be time and space efficient, and to scale up to larger datasets. The project is implemented in Python 3.

2 Dataset

The project is based on the analysis of the "2013 American Community Survey" dataset published on Kaggle and released under the public domain license (CC0). There are two types of survey data provided: housing and population. In population survey each row is a person and each column is a property such as age, gender etc., in housing survey each row is a housing unit and each column is a propriety such as rented, owned etc.

The dataset is divided into four files, two files for housing survey "ss13husa.csv" and "ss13husb.csv" and two files for population survey, "ss13pusa.csv" and "ss13pusb.csv", respectively containing 1613672 and 1519123 records, for a total of 3132795 records. In every file of type "a", for each survey, there are states from 1 to 25 and in every file of type "b" there are states from 26 to 50.

The total number of attributes is 509. The label to be predicted must be selected among the following 5 attributes, removing the remaining 4 from the dataset:

- PERNP (Person's earnings)
- PINCP (Person's income)
- WAGP (Wages or salary income past 12 months)
- HINCP (Household income)
- FINCP (Family income)

3 Data organization

We choose to implement the project using the Apache Spark framework, v3.1.1 prebuilt for Hadoop3.2, and its python implementation PySpark. In Python data analysis Pandas is usually the most straightforward choice because of its user-friendly interface and its flexibility. On the other hand PySpark, with the distributed nature of its Dataframe and RDD, is more effective when dealing with dataset that have huge dimensions, also allowing to distribute horizontally the computations, and allowing the creation of dynamic and heterogeneous clusters that can easily be scaled if more hardware resources should be needed. This is also one of the requirements for the implementation of the project. At startup, our project downloads the dataset's *.csv* files from Kaggle website, unzips and then load them into a spark dataframe.

We chose to predict PINCP (Person's income). As PINCP is a column from *population* data, we joined the the housing dataframes with a left join on the population ones, using the SERIALNO column as key.

There are 550753 of records where PINCP is not defined. We filtered those rows out and the remaining records were 2582042. Then we removed some feature columns that were not significant in predicting PINCP.

Those columns are

- RT (Record Type (Housing or Person record))
- VACS (Vacancy status)
- SERIALNO (Housing unit/GQ person serial number)
- DIVISION (Division code)
- REGION (Region code)
- ADJINC (Adjustment factor for income and earnings dollar amounts)
- ADJHSG (Adjustment factor for housing dollar amounts)
- WGTP (Housing Weight)
- SPORDER (Person number)
- PWGTP (Person's weight)
- PUMA (Public use microdata area code)
- pwgtp[1-80] (Person's Weight replicate 1-80)
- wgtp [1-80] (Housing Weight replicate 1-80)
- F[xyz]P (Allocation flag)

The ADJINC/ADJHSG columns are useful only if the user want to adjust the reported income/housing cost to 2013 dollars change. The goal of the project is to predict a chosen label; in this case we thought inflation-adjusts was not significant for our goal. Also, for our study, all the weight columns and allocation flags were not considered because they are strictly of statistical meaning.

Total number of columns after *data cleansing* is 207. From the data documentation we identified 31 numerical features and 176 categorical features, 17 of which may be considered ordinal features, like YBL (Year when the structure was first built). SPORDER and SERIALNO are id numbers for people and housing units which are not relevant for our prediction. VACS column was dropped because every value was null.

4 Data pre-processing

We split our dataframe in two sets of different size: training set, which is the 0.7 of the entire set, and test set which is the 0.3 percent. The split was implemented with the *randomSplit()* method provided by PySpark. Dataframe was randomly shuffled before splitting.

We handled the three types of features identified earlier in three different ways:

- The *numerical* features were managed by filling with the value 0 all the rows that had a null value.
- The *ordinal* categorical features, that are categorical data which have an order and may be represented using an ordinal scale, were indexed with a StringIndexer.
- The *non-ordinal* categorical data were indexed with a StringIndexer and encoded with a OneHotEncoder.

Due to the different nature of the data, they were centered and scaled to unit standard deviation fitting a *Standard Scaler* model only on the training set and then applying the transformation on both the training and the test set. This procedure was implemented to reduce the risk of data leakage, if the model was fitted on the whole dataset the information on the test set wouldn't be new for the ridge regression model, because the Standard Scaler would compute standard deviation taking into account test set's data mean and variance which for the model must be completely unknown not to incur in overfitting problems.

One-Hot-Encoding used for *non-ordinal* categorical has the advantage of producing a binary result rather than ordinal and that everything sits in an orthogonal vector space. The disadvantage is that for high cardinality, the feature space can really blow up quickly and you start fighting with the curse of dimensionality. Indeed, after One-Hot-Encoding (the vector of categorical features has more than 4000 values)

In these cases, it may be helpful to employ one-hot-encoding followed by Principal Component Analysis (PCA) for dimensionality reduction. Thus, we also tried to increase the performance of the algorithm by integrating a PCA technique for features extraction, in order to run our algorithm implementation on a lower number of features.

5 The algorithm implementation

We chose to implement the Ridge Regression algorithm. Ridge regression is the regularized form of linear regression. *Regression* denotes any modeling task that involves predicting a numerical value given a set of input features. In other words, regression tries to estimate the expected target value when we provide the known input features. Linear Regression makes use of Ordinary Least Square (OLS) method in order to predict the value of a target variable based on given predictor variables. That means, Linear Regression tries to minimize the sum of the squares of the differences between the observed dependent variable (the values of the variable being observed) in the given dataset and those predicted by the linear function of the independent variable.

Ordinary Least Square method finds the coefficients that best fit the data. However, it also finds an unbiased coefficients. Here unbiased means that OLS doesn't consider which independent variable is more important than others. It simply finds the coefficients for a given dataset by considering all variables of the same importance. However, this could easily lead to overfitting issues, that is, the model will not perform as well with new data because it is built specifically for the given data.

Ridge Regression tries to improve linear regression by making use of the L2 penalty. This penalty shrinks the coefficients of those input variables which

have contributed less in the prediction task. This helps in building a model that gives more importance to variables that are more significant for the prediction task. Thus, Ridge Regression may be considered an improvement over linear regression as it allows to obtain a solution that is more resistance to over-fitting and with less variance error.

In order to scale up with data size as best as possible, we decided to implement our Ridge Regression Algorithm using the Apache Spark framework, with an approach as similar as possible to the MapReduce paradigm. In particular, we implemented the Ridge Regression's closed form, that may be formalized as follow:

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

We split the previous formula in three different steps:

1. In the first step we computed

$$M = (\alpha I + S^T S)$$

2. In the second step we computed

$$\hat{M} = M^{-1}$$

3. In the third step we computed

$$\hat{M} S^T y$$

The trickiest part of the first step was computing $S^T S$, because of the size of our input dataframe. Indeed, it had about 2 million rows, and given a size of each value of 8 bytes, it means a total of 16MB of RAM just for storing one column of the matrix S. Assuming a number of 1000 columns, that would require 16 GB of RAM for the full matrix S, and 16 more GB just to store its transposed. This makes that computation not feasible to be computed even on a machine with 32GB of RAM, considering the amount of memory required to run the operating system plus the memory required for storing all the intermediate computations. For this reason, a MapReduce approach was the ideal solution. The Apache Spark framework provides some very convenient tools for data preprocessing and allows to implement a custom MapReduce algorithm using a distributed file system, thus providing to distribute the computation on multiple machines and to scale up with the dataset size. The implementation, as shown in Figure 1, involves the use of the flatMap() method, in order to compute the squared matrix $x_i^T \times x_i$ for every row x_i. We then assigned an incremental key to the rows of each computed squared matrix, such that the first rows of all matrices have key 0, the second rows have key 1 and so on. After that, we applied a reduceByKey function in order to sum up all the rows with the same keys, that means, all the first rows of the matrices, all the second rows, etc. That lead us to the matrix resulting from $S^T S$.

The final part of the first step was adding together that final matrix and αI, which is the identity matrix multiplied by the regularization parameter α.

The second step, that is the computation of the inverse of the singular matrix resulted from the first step, was done using the *numpy.linalg* library, that comes

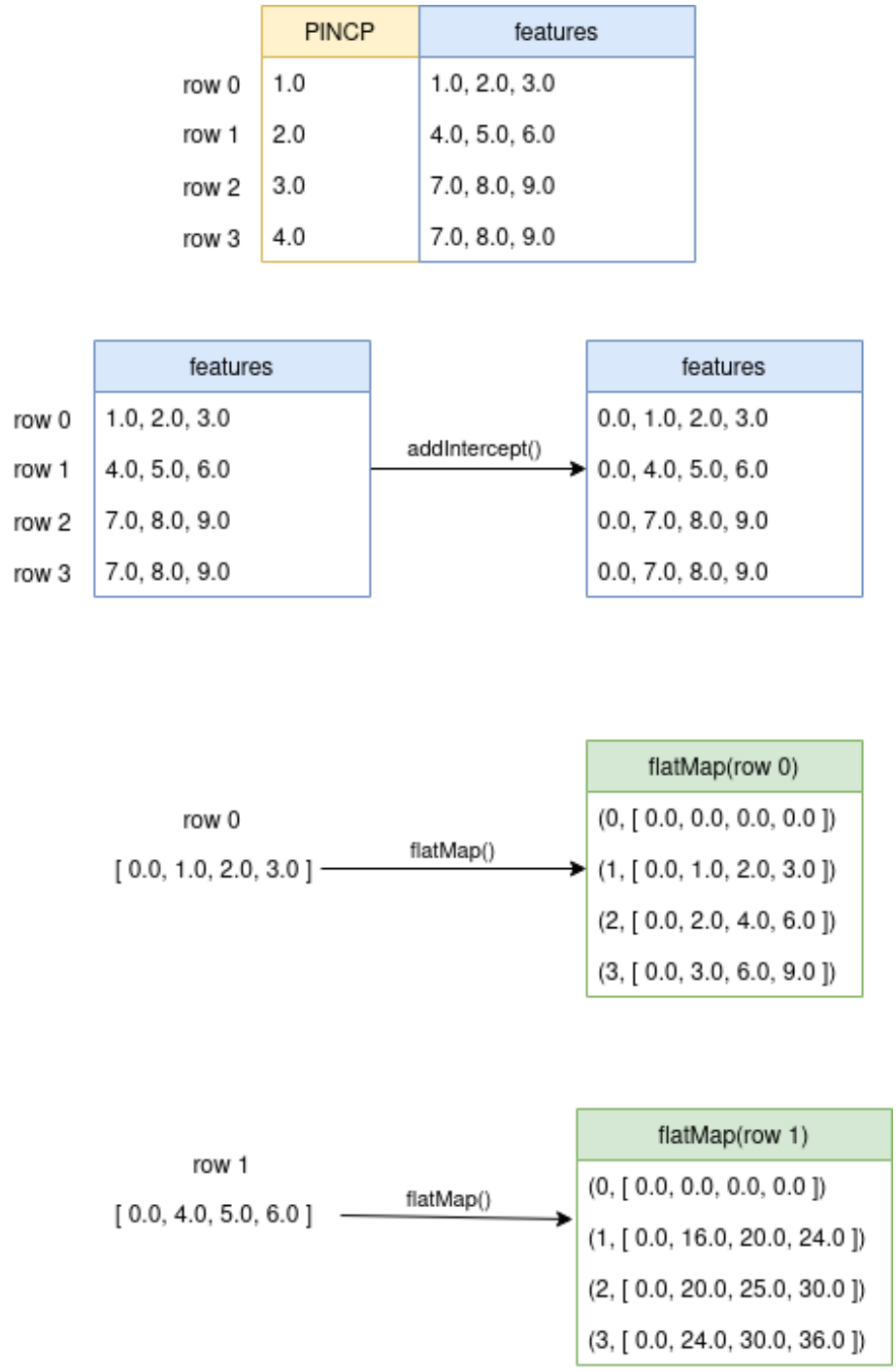

Figure 1: example of the flatMap implementation of our algorithm

with a convenient method already implemented for the computation of inverse matrices. This is the only operation with matrices done on a single node, without

using Apache Spark. All the other operations with matrices were done using a MapReduce approach in order to allow the computation to be distributed on different nodes, which is mandatory, given the size of the matrices. Note that the singular matrix resulting from the step 1 is a square matrix with shape $m \times m$, where m is the number of the columns of the initial dataframe. That means that the algorithm implementation is not limited by this choice, as the amount of input data samples, that is the total number of rows of the input dataframe, does not influence the amount of data required to be stored in RAM after the step 1. Assuming, for example, a dataframe with 5000 columns, the matrix after the step 1 would be a matrix with shape (5000 x 5000), and assuming a size of 8 bytes for each value of the matrix, that would require a total of $5000 * 5000 * 8$ bytes of RAM, that is just 200MB of RAM. So, even if the inverse of the matrix is computed in a non distributed way, the algorithm still scales up perfectly with the number of input samples.

The third step requires both a matrix-matrix and a matrix-vector multiplication; in particular, the inverted matrix resulted from step 2 must be multiplied by the transposed form of the matrix of the whole input dataframe, and the resulting matrix must then be multiplied by the vector of the target labels. Again, we implemented this computations through Apache Spark with a MapReduce approach, in order to allow a distributed computation of the result. The matrix-matrix product has size

$$(m \times m) \times (m \times n) = m \times n \tag{1}$$

while the matrix-vector multiplication has size

$$(m \times n) \times (n \times 1) = m \times 1 \tag{2}$$

Product (1) was done by multiplying the matrix resulting from step 2 for each row of the dataframe, that means, we performed n products each with size $(m \times m) \times (m \times 1) = m \times 1$. Then, we mapped each $m \times 1$ row with a different key in order to keep track of their ordinal number; this is mandatory in order to implement product (2) efficiently. Indeed, we applied product (2) before computing the $m \times n$ matrix resulting from product (1). Keys were added using the zipWithIndex() method, which takes as input a row vector v and creates a tuple (k, v) for each of the rows of the dataframe, where k is the sequential number of the row. The same zipWithIndex() method was used for the labels vector y, in order to have the same sequential keys between the labels and the rows of the dataframe. Then we joined all the tuples with the same keys, in order to have a single list of tuples of the form

$$(k_i, (v_i, y_i))$$

At this point the product between each row vector v_i and each label value y_i was straightforward, requiring only one more final reduce step in order to sum up all the resulting vectors. The output of the previous operation is the final vector of the weights, with shape $m \times 1$.

6 Experiments

The initial dataset was split randomly with a 70%-30% split in order to simulate a training and a test set. Also, as explained earlier, data were organized in three

categories, that are *numericals*, *ordinals* and *categoricals* and then were passed into three different pipelines:

- *Numericals* were standardized using the *StandardScaler* included within Apache Spark framework.
- *Ordinals* were indexed with a *StringIndexer* and standardized with a *StandardScaler*.
- *Categoricals* were indexed with a *StringIndexer*, one-hot encoded with a *OneHotEncoder* and then standardized with a *StandardScaler*.

The final vectors computed by joining *numericals*, *ordinals* and *categoricals* had almost 5000 features. We performed three different experiment: For the first one we used a subset of all the available features, excluding from the experiment all the categoricals features. For the second one and the third one, again, we used a subset of all the available features but this time we included 25 randomly selected of all the 150 categoricals features, for the second, and 50 randomly selected of all categoricals for the third one. Finally, we used Principal Component Analysis in order to extract the most relevant features among all, to try to increase the performance of our algorithm. To summarize, we ran:

1. Experiment 1: Only numericals and ordinals features (no categoricals)
2. Experiment 2: All numericals + all ordinals + 25 of 150 categorical features
3. Experiment 3: All numericals + all ordinals + 100 of 150 categorical features
4. Experiment 4: All features + PCA

In all our experiments we fit the Standard Scalers (and also PCA) only on training set, and then we use the fitted models to apply the same transformation both to training an test set.

We also used a CrossValidation technique in order to tune the Ridge Regression regularization parameter α for both the experiments. The results of the experiments, however, are not directly comparable as each run was performed with a different random split of the dataset into training/test set.

We used cross validation to choose the best regularization parameter between three values: 0.01, 0.1 and 1. Cross validation was done by splitting the training set into 5 folds. That means we had to run our algorithm 5 times (one for each excluded fold) for each of the 3 regularization values, that is a total number of 15 times.

In Experiment 2 we selected 25 of 150 categorical features randomly. The results we show below were obtained with the following columns:

```
['DDRS', 'NATIVITY', 'HFL', 'HANDHELD', 'WKEXREL', 'VEH',
'RACASN', 'MARHM', 'RACAIAN', 'FS', 'SEX', 'STOV', 'OCCP',
'DRIVESP', 'CIT', 'SCH', 'FES', 'BLD', 'RACBLK', 'JWTR',
'BATH', 'DEYE', 'HUPARC', 'FER', 'RAC1P']
```

In Experiment 3 we selected 100 of 150 categorical features randomly. The results we show below were obtained with the following columns:

['HISP', 'MRGT', 'LANX', 'TYPE', 'MLPB', 'QTRBIR', 'SATELLITE', 'RWAT', 'GCL', 'RACAIAN', 'NWRE', 'FIBEROP', 'LNGI', 'HICOV', 'FS', 'LAPTOP', 'SMX', 'WIF', 'POWSP', 'HINS3', 'REFR', 'NAICSP', 'SSMC', 'MULTG', 'NWLA', 'MRGI', 'KIT', 'NPP', 'HINS4', 'FER', 'HINS5', 'BUS', 'DSL', 'RC', 'MLPCD', 'MIG', 'DEYE', 'NOP', 'MIGSP', 'HHL', 'LANP', 'NWLK', 'HINS1', 'WKEXREL', 'SEX', 'GCM', 'RACBLK', 'DEAR', 'BROADBND', 'FHINS4C', 'PARTNER', 'RESMODE', 'POBP', 'ACR', 'RACPI', 'ACCESS', 'GCR', 'R18', 'DREM', 'MIL', 'SRNT', 'MLPJ', 'SINK', 'OC', 'POWPUMA', 'DDRS', 'COMPOTHX', 'HUPAOC', 'MAR', 'HANDHELD', 'MARHM', 'FHINS5C', 'R65', 'SCIENGP', 'DOUT', 'RACNUM', 'MLPFG', 'ENG', 'DIS', 'MSP', 'WRK', 'MRGX', 'RELP', 'ANC1P', 'VPS', 'COW', 'HINS2', 'BLD', 'MODEM', 'CIT', 'FOD1P', 'STOV', 'SCIENGRLP', 'RACWHT', 'ANC', 'MARHW', 'RWATPR', 'PUBCOV', 'RACNH', 'PSF']

7 Results

To evaluate our model we used the coefficient of determination R^2. R^2 score provides a measure of how well observed outcomes are replicated by the model, based on the proportion of total variation of outcomes explained by the model. R^2 score may be formalized as follows:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y_i})^2}{\sum_i (y_i - \mu)^2}$$

where

$$\mu = \frac{1}{n} \sum_{i=1}^{n} y_i$$

and $\hat{y_i}$ is the predicted value for a given input i.

We ran all the experiments on a Spark Cluster made of 3 Spark Worker Instances each with 40GB of RAM and 7 cores, and a total number of 3 Spark Executor Instances each with 36GB of RAM and 6 cores.

7.1 Experiment 1: all numericals + all ordinals features

The total number of features after the data pre-processing was 49. That means, we fed our algorithm implementation with about 2 million rows each with 49 values of 8 bytes each. Total job time was 31 minutes.

The R^2 scores of cross validation for each different regularization parameter α value were:

- $\alpha = 0.01$, $R^2 = 0.56155106146$
- $\alpha = 0.1$, $R^2 = 0.56155106149$
- $\alpha = 1$, $R^2 = 0.56155106176$

Given these result, we selected $\alpha = 1$. Final score on test set using $\alpha = 1$ was

$$R^2 = 0.55787021104$$

As R^2 score baseline is 0.0, the performance of our final model were very good overall.

7.2 Experiment 2: all numericals + all ordinals + 25 of 150 categorical features

The total number of features after the data pre-processing turned out to be 643. That means, we fed our algorithm implementation with about 2 million rows each with 643 values of 8 bytes each.

Total job time was 3h52m. The most of the time was used to perform cross validation in order to tune the Ridge Regression regularization parameter. Each run of the algorithm took about 15 minutes to fit our training set. As cross-validation requires the algorithm to be fit 15 different times, the total time spent just for running cross-validation was about 3h30m hours.

The R^2 scores of cross validation for each different regularization parameter α value were:

- $\alpha = 0.01$, $R^2 = 0.63454325884$
- $\alpha = 0.1$, $R^2 = 0.63454325932$
- $\alpha = 1$, $R^2 = 0.63454326499$

Given these result, we selected $\alpha = 1$. Final score on test set using $\alpha = 1$ was

$$R^2 = 0.63212228784$$

As R^2 score baseline is 0.0, the performance of our final model are very good overall.

7.3 Experiment 3: all numericals + all ordinals + 100 of 150 categorical features

The total number of features after the data pre-processing turned out to be 1734. That means, we fed our algorithm implementation with about 2 million rows each with 1734 values of 8 bytes each.

Total job time was 29h37m. This may seems a lot of time, but the most of the time was used to perform cross validation in order to tune the Ridge Regression regularization parameter. We think this time can be further reduced by tuning the Spark configuration and by adding more computational power to the cluster. Each run of the algorithm took about 1h30m to fit our training set. As cross-validation requires the algorithm to be fit 15 different times, the total time just for running cross-validation was about 23 hours.

The R^2 scores of cross validation for each different regularization parameter α value were:

- $\alpha = 0.01$, $R^2 = 0.62640002738$
- $\alpha = 0.1$, $R^2 = 0.62640002723$
- $\alpha = 1$, $R^2 = 0.62640002857$

Given these result, we selected $\alpha = 1$. Final score on test set using $\alpha = 1$ was

$$R^2 = 0.625615977602$$

As R^2 score baseline is 0.0, the performance of our final model are very good overall.

7.4 Experiment 4: all features + PCA(k=75)

This experiment was performed by taking into accounts all the features left into the dataframe after the cleansing process, and by using a PCA technique in order to extract only the K most relevant features. We expected this to be particularly useful in order to speed up the fitting time of the algorithm without losing important information about the features, especially when performing cross-validation, that in our experiments requires the algorithm to be run 15 different times just to tune the regularization parameter α. For some reason, the total time of the job was 31h24m, which was higher then the one required for experiment 2. As in the previous experiment, but the most of the time was used to perform cross validation in order to tune the Ridge Regression regularization parameter. PCA fitting time was no more than 3 hours. Each run of the algorithm took about 1h50m to fit our training set. As cross-validation requires the algorithm to be fit 15 different times, the total time spent just for running cross-validation was about 27 hours. The total number of features resulting from PCA was fixed at 75. That means, we fed our algorithm implementation with about 2 million rows each with 75 values of 8 bytes each. We do not actually know why the algorithm took so long to fit with only 75 input features, but we think that may be related to a non-optimal spark configuration. The R^2 scores of cross validation for each different regularization parameter α value were:

- $\alpha = 0.01$, $R^2 = 0.37152532566$
- $\alpha = 0.1$, $R^2 = 0.371525325670$
- $\alpha = 1$, $R^2 = 0.371525325691$

Given these result, we selected $\alpha = 1$. Final score on test set using $\alpha = 1$ was

$$R^2 = 0.3709365289515$$

Even though the score is not directly comparable to the results of experiment 1, experiment 2 and experiment 3, as the dataframe was split in a random way for each experiment, this score is clearly lower than the ones before. That may be related to the high number of categorical features and to PCA that performs better with numerical features and is less meaningful when used with one-hot-encoded data.

" *We declare that this material, which We now submit for assessment, is entirely our 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 our work. We 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 us or any other person for assessment on this or any other course of study.* "