Analysis on the American Community Survey dataset

Lassandro Marco 945907

Contents

1 Introduction

In this report will be presented the experiments done for the task of finding the person's salary of the past 12 months using the "American Community Survey" dataset.

After the organization of the data it has been implemented from scratch two ml algorithms for a regression problem: one about **Linear Regression** (with batch gradient descent and L2 regularization) and then another for the construction of a **Decision Tree** for regression. For this purpose it has been used the **Spark** framework to manage the data and distribute the operations to achieve a scalability property in terms of time and space complexity.

2 Dataset

The American Community Survey is an ongoing survey from the US Census Bureau. In this survey, approximately 3.5 million households per year are asked detailed questions about who they are and how they live.

The topic of the experiments was to predict the label **WAGP** about wages or salary of the past 12 months of a worker. Indeed it has been considered the population data set of this survey, in particular the CSV **ss13pusa.csv** present in **Kaggle** website.

2.1 ss13pusa.csv

This csv file have a size of **1.4 GB** in which there are about **1600000 records** with information of American's individuals. There are **283 columns** representing features like age, gender, whether they work, method/length of commute, etc.

All columns are numerical, there are missing data in most of the features but the **N/A values** represents individual circumstances (like age less than 17, he/she doesn't have a car, he is not a worker, etc.) for which the person couldn't answer at specific questions.

3 Pre-processing

The features **PERNP** (Person's earnings), **PINCP** (Person's income), **HINCP** (Household income), **FINCP** (Family income) have been removed from the dataset because they represents similar information respect to the feature "WAGP" to be predicted.

Records with **AGE** < 17 have been deleted from dataset in order to focus on data persons that could had a salary in the past 12 months.

All missing values have been considered meaningful, indeed to better manage the data operations the **N/A values** of all the features was replaced with a **zero value**, shifting the other values by one.

3.1 Spearman Correlation

Due to the huge number of the features present in the dataset a **Spearman** correlation function has been implemented in order to reduce the features' dimensionality, selecting the ones that are most related with the feature to be predicted.

The **Spearman** correlation coefficient is a measure of the strength of the monotonic relationship between two continuous or ordinal variables. In a monotonic relationship, the variables tend to change together, but not necessarily at a constant rate. The **Spearman** correlation coefficient is based on the ranked values for each variable rather than the raw data.

The output of this function is a number between **-1** and **1** that indicates the extent to which two variables are related. So $+\mathbf{1}$ indicates a perfect association of ranks, a $\mathbf{r_s}$ of zero indicates no association between ranks and a $\mathbf{r_s}$ of **-1** indicates a perfect negative association of ranks. The closer $\mathbf{r_s}$ is to zero, the weaker the association between the ranks.

For the implementation of this function was used the **Pearson** correlation formula with ranked variables:

$$r_s = \frac{conv(rg_x, rg_y)}{\sigma_{rg_x}\sigma_{rg_y}} \tag{1}$$

- $\mathbf{r_s}$ is the correlation coefficient
- $\mathbf{conv}(\mathbf{rg_x}, \mathbf{rg_y})$ is the covariance between two variables, in a data sample, formula used:

$$conv(rg_x, rg_y) = \Sigma rg_x rg_y - n\overline{rg_x rg_y} \tag{2}$$

  - $\mathbf{n}$ is the size of the data sample
  - $\mathbf{rg_x}$ and $\mathbf{rg_y}$ are the rank variables
  - $\overline{\mathbf{rg_x}}$ and $\overline{\mathbf{rg_y}}$ are the means of the ranked variables

- $\sigma_{\mathbf{rg_x}}$ and $\sigma_{\mathbf{rg_y}}$ are respectively the standard deviation of $\mathbf{rg_x}$ and $\mathbf{rg_y}$, the formula used for the data sample was:

$$\sigma_{rg_x} = \sqrt{\Sigma rg_x^2 - n\overline{rg_x}^2} \tag{3}$$

So the resulting formula is:

$$r_s = \frac{\Sigma rg_x rg_y - n\overline{rg_x rg_y}}{\sqrt{\Sigma rg_x^2 - n\overline{rg_x}^2}\sqrt{\Sigma rg_y^2 - n\overline{rg_y}^2}} \tag{4}$$

3.2 Feature selected

It has been taken the **0.2%** of the whole dataset to compute the **Spearman** correlation between the feature **WAGP** and the other features.

Feature with $|\mathbf{r_s}| > \mathbf{0.1}$ have been chosen for the experiments, that are 64 features, the top five related feature with **WAGP** are:

Feature	Spearman score	Description
WKHP	0.8	Usual hours worked per week past 12 months. Values from 01 to 99, N/A value indicates people with age less than 16 or people didn't work in the past 12 months
ESR	0.73	Employment status. Values from 0 to 6, N/A values indicates people with $age <$ 16
WKL	0.69	When last worked. Values from 1 to 3. N/A value indicates people with $age <$ 16
JWAP	0.56	Time of arrival at work - hour and minute. Values from 1 to 285. N/A value indicates person isn't a worker
JWMNP	0.53	Travel time to work. Values from 1 to 200. N/A value indicates people
NWLA	0.52	On layoff from work. Values from 1 to 3. N/A value indicates people with $age <$ 16

4 Regression algorithms

4.1 Linear Regression

As first approach it has been implemented a linear regression model in order to find a linear relationship between the dependent variable **Y** and independent variables **X**. The equation of the linear regression is:

$$Y = Xw + q \tag{5}$$

Where **w** is a vector of coefficients for every independent variable and **q** is the intercept of **Y**.

To calculate the **error** done by the model in order to predict the variable **y** of the observations, it has been applied the Squared Sum Loss function, that is the squared difference between the real values **y** and the predicted values $\hat{\mathbf{y}}$.

$$SSE = \sum_{i=0}^{n} (y_i - \hat{y_i})^2 \tag{6}$$

where **n** is the number of the observations and $\hat{\mathbf{y}} = (\mathbf{wx_i} + \mathbf{q})$

4.1.1 Gradient Descent algorithm

To minimize the cost function of the model so to have a better accuracy, it is necessary to update the values of **w** and **q**.

To do that it has been implemented the iterative optimization algorithm that is the Gradient Descent. So after every iteration the independent variables' coefficients **w** and the intercept **q** will be updated to minimise the cost function SSE:

- **w** and **q** are initially equals to 0, then the partial derivatives of the loss function SSE with respect to **w** and **q** is computed:

  Partial derivative for **w**:

  $$P_w = 2 \sum_{i=0}^{n} x_i (y_i - \hat{y_i}) \tag{7}$$

  Partial derivative for **q**:

  $$P_q = 2 \sum_{i=0}^{n} (y_i - \hat{y_i}) \tag{8}$$

  where $\hat{y_i} = w * x_i + q$

- Than **w** and **q** are updated as:

  $$w_i = w_{i-1} - lrP_w \tag{9}$$

$$q_i = q_{i-1} - lrP_q \tag{10}$$

where **lr** is an hyperparameter to be set for controlling the length of the steps that gradient descent algorithm does toward the optimal solution.

4.1.2 Ridge regression

It has been added a penalty factor at the cost function that determine the importance loss of some features for a better stability of the model on outlier observations and so to avoid overfitting.

It has been applied an **L2 penalty** to the loss function, so the new loss function is:

$$E = \sum_{i=0}^{n}(y - (wx_i + q))^2 + \lambda \sum_{j=0}^{d} w_j^2 \tag{11}$$

The strength of the penalty is managed through the hyperparameter λ, with $\lambda = \mathbf{0.0}$ the loss function will be the same as [Eq. 6]

Because of this changes the [Eq. 9] and [Eq. 10] will become:

$$w_i = w_{i-1} - l(2\sum_{i=0}^{n} x_i(y - \hat{y_i}) + 2\lambda \sum_{j=0}^{d} w_j) \tag{12}$$

$$q_i = q_{i-1} - l(2\sum_{i=0}^{n}(y - \hat{y_i}) + 2\lambda \sum_{j=0}^{d} w_j) \tag{13}$$

4.1.3 Implementation

Figure 1: Add of a new column in the training and validation data

```
#Itialization of the weights with zeros, adding 1 more weight that will correspond to the  intercept
weights = np.zeros(sizeOfFeatures+1)

#Added a new column of all 1 to the training data for the intercept updates
trainingData = trainingData.withColumn("intercept", F.lit(1))
assembler = VectorAssembler(inputCols=["intercept", featureCol] , outputCol="featuresI")
trainingData = assembler.transform(trainingData)

#Added a new column of all 1 to the validation data for the intercept
valData = valData.withColumn("intercept", F.lit(1))
valData = assembler.transform(valData)
featureCol = "featuresI"
```

To save time for the computation of the partial derivatives, it has been integrated the intercept **q** into the array of the weights [Figure 1] in order to unify the two equations [Eq. 11] and [Eq. 12].

For this a **new column** has been added to the training data and to the validation data with values equals to **1**, than a new set of features has been extracted to train the model.

Figure 2: Computation of the RMSE on the validation data

```
for i in range(maxIterations):
    #Calculation of the accuracy of the model during the iteration to better understand the improvents of the m
    #The accuracy is calculated through the compute of root mean square error on the validation data
    labelsAndPredsTrain = valData.rdd.map(lambda p: getLabeledPrediction(weights, p, featureCol, labelCol))
    rmse = calcRMSE(labelsAndPredsTrain)
    trainingErrors.append(rmse)

    print("Iteration:"+str(i)+" rmse:"+str(trainingErrors[i]))
```

At every iteration a **root mean square error** is calculated on the validation data to view the progress during the training of the model. The operation is done through a **map job** with **spark** [Fig. 2].

Figure 3: Check if it's still good continue to train

```
#The difference between the error of the current iteration and the error of the previous iteration
#has to be greater than a threshold fixed to 1
if(i > 0 and (trainingErrors[i-1] - trainingErrors[i] <= minImprovement)):
    print("Error doesn't go down. Stopped the training.")
    break
```

The algorithm will stop if the difference between the error of a previous iteration and the actual iteration are less than a certain threshold, the variable **minImprovement**, fixed to **1**. In this way it has been saved time when the training didn't go any further.

This has been done also to stop a possible divergences of the model during the update of the weights, for instance when the learning rate parameter is set too big [Fig. 3].

Figure 4: Computation of the gradient for which update the weights

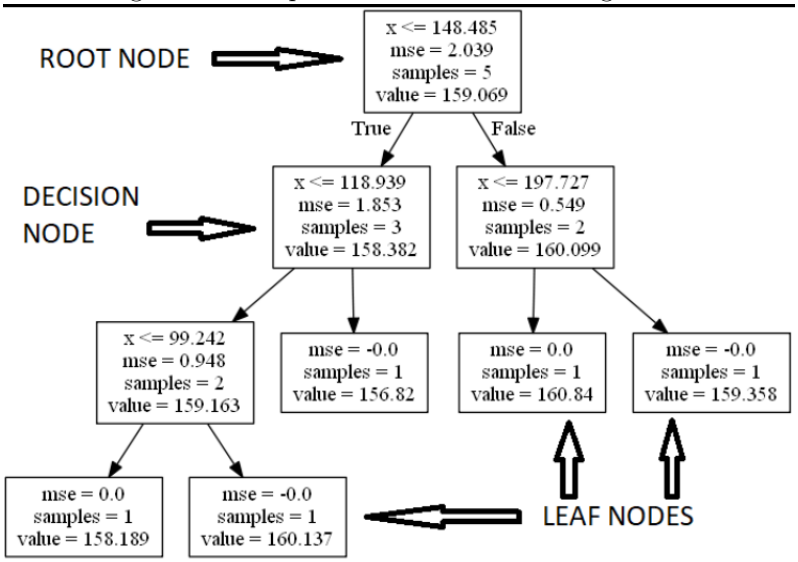
```
#To calculate the gradient of the current iteration will be executed a map job through spark, will be calculated for every
gradient = sum(trainingData.rdd.map(lambda p: (DenseVector(gradientSummand(weights, p, featureCol, labelCol))))
               .collect())

weights = weights - learningRate * (2 * gradient + 2*alpha*sum(weights))
```

Another **map job** is launched to calculate the gradient summand and finally the weights will be updated with the **L2 regularization** [Fig. 4].

4.2 Decision Tree for Regression

Figure 5: Example of a decision tree for regression



A tree predictor has the structure of an ordered and rooted tree. The algorithm produces a recursive binary partitioning of the features, every node is a decision point labeled as "$\mathbf{x} <= \mathbf{y}$" or "$\mathbf{x} = \mathbf{y}$" where $\mathbf{x}$ is the name of the feature and $\mathbf{y}$ is the value for which an observation will be driven to the left or right child of the node through the condition "<=" or "=" [Fig. 5].

The choose of the feature $\mathbf{x}$ and the value $\mathbf{y}$, in order to make the best split of the data, is done by evaluating the weighted sum of the variances for the left and right part of the splitted data.

Variance:

$$V = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2 \tag{14}$$

where $\mathbf{N}$ is the number of the observations.

So the weighted sum of the variances will be:

$$impurity = \frac{1}{N_{left}} V_{left} + \frac{1}{N_{right}} V_{right} \tag{15}$$

After all the candidates splits have been evaluated the one that minimize the [Eq. 15] is chosen and a new decision point will be created.

The candidates feature's value will be selected in two different way based on the type of the feature, if it is a categorical feature or a continuous feature.

- For categorical features the candidates are chosen simply by taking and sorting the unique values of that feature.
- For continuous features it has been chosen to not taking as candidates the unique values but to take in consideration an approximate set of values based on a tunable parameter.

In case the split is based on a categorical feature the observations will go to left child if $\mathbf{x} != \mathbf{y}$ is true and to the right child otherwise. If the split is based on a continuous feature the observations will go to left child if $\mathbf{x} <= \mathbf{y}$ is true and to the right child otherwise.

The generation of other nodes will continue until the meet of one of this stopping criteria:

- The tree's branches reached a depth limit controlled by a hyperparameter.
- The observations prensent in a node are less than a fixed number of instances
- The impurity of a split is equal to 0.
- The size of the left or the right part of the data is 0.

When one of this conditions is true a leaf will be created and will represent the label of the observations that reach that terminal node. The leaf is the mean of the labels of the training data at that point.

4.2.1 Implementation

Figure 6: Retrieving the types of the features

```
# data preparations
if counter == 0:
    global COLUMN_HEADERS, FEATURE_TYPES
    data = data.toPandas()
    COLUMN_HEADERS = featureList
    FEATURE_TYPES = getTypeOfFeature(data, featureList, maxCatValues)
    print(FEATURE_TYPES)
```

As first step of the Decision Tree algorithm[1] the function "getTypeOfFeature" will provide if a feature is **continuous** or **categorical**, if the distinct values of a feature is > than the parameter **maxCatValues** (default is 15) the feature will result "continuous" otherwise will be "categorical" [Fig. 6].

Figure 7: Base cases of the function

```
# base cases
if (len(data) < minSamples) or (counter == maxDepth):
    leaf = createLeaf(data, labelCol)
    return leaf
```

As base case [Fig. 7] of the recursive function there are two stopping criteria described above that are the reach of a max depth (default 5) of a tree's branch and the minimum amount of observations required to continue the splits. If one of this is condition is true a leaf will be created and the recursion will be stopped at that level.

[1]The implementation of the algorithm is based (then modified) on the notebook *https://github.com/SebastianMantey/Decision-Tree-from-Scratch/blob/master/notebooks/Video%2011%20-%20Regression%202.ipynb*

Figure 8: Operations for the split of the data

```
# recursive part
else:
    counter += 1

    print("Depth:"+str(counter))

    #Taking the cadidates for which evaluate the split of the data
    potentialSplits = getPotentialSplits(data, featureList, bins)

    #Finding the best cadidate for the split
    splitColumn, splitValue, metric = getBestSplit(data, potentialSplits, labelCol)

    #Split the data with the best cadidate
    dataLeft, dataRight = splitData(data, splitColumn, splitValue)
```

In the recursive part of the algorithm the three main operations are:

- getPotentialSplits:

Figure 9: execution of the mapping job for the values' selection

```
def getCandidates(data, feature, bins = 32):

    candidates = [feature]

    if FEATURE_TYPES[feature] == "continuous":
        minValue = min(data[feature])
        maxValue = max(data[feature])

        if len(data) < bins:
            bins = len(data)

        step = (maxValue - minValue)/bins

        value = minValue
        candidates = [feature]
        while(value <= maxValue):
            candidates.append(value)
            value += step

    else:
        distinctValues = np.unique(data[feature])
        distinctValues.sort()
        for values in distinctValues:
            candidates.append(values)

    return candidates

def getPotentialSplits(data, featureNameList, bins = 32):
    featureList = sc.parallelize(featureNameList)
    candidates = featureList.map(lambda f: getCandidates(data, f, bins))

    return candidates
```

A map job will be executed to retrieve the candidates' value for every feature. The parameter "bins" indicates how much values generate for every continuous feature. This is done calculating a step that will be added to the min value of the feature until the reach of the max value [Fig. 9].

- getBestSplit:

Figure 10: Execution of the map job for the research of the best split

```
def getBestMetric(data, feature, values, labelCol):
    firstIteration = True
    gotBestMetric = False

    for value in values:
        dataLeft, dataRight = splitData(data, splitColumn=feature, splitValue=value)
        metric = totalVariance(dataLeft, dataRight, labelCol)

        if firstIteration or metric < bestMetric:
            gotBestMetric = True
            firstIteration = False
            bestSplitValue = value
            bestMetric = metric
            if(bestMetric == 0.0):
                break
        elif gotBestMetric:
            break

    return [feature, bestSplitValue, bestMetric]

def getBestSplit(data, potentialSplits, labelCol):
    metrics = potentialSplits.map(lambda candidates: getBestMetric(data, candidates[0], candidates[1:], labelCol)).collect()

    firstIteration = True
    for m in metrics:
        if firstIteration or m[2] < bestMetric:
            firstIteration = False
            bestSplitColumn = m[0]
            bestSplitValue = m[1]
            bestMetric = m[2]

    return bestSplitColumn, bestSplitValue, bestMetric
```

A **map job** will be executed for every feature to retrieve from the candidates the best metric, than the feature with the minimum metric will be selected [Fig. 10].

- splitData:

Figure 11: Split of the data with the best feature and value selected

```
def splitData(data, splitColumn, splitValue):

    typeOfFeature = FEATURE_TYPES[splitColumn]
    if typeOfFeature == "continuous":
        dataLeft = data.loc[data[splitColumn]<= splitValue]
        dataRight = data.loc[data[splitColumn] > splitValue]
    # feature is categorical
    else:
        dataLeft = data.loc[data[splitColumn] == splitValue]
        dataRight = data.loc[data[splitColumn] != splitValue]

    return dataLeft, dataRight
```

The data will be divided based on the new decision point selected [Fig. 11].

Figure 12: Construction of the tree's structure

```
# decision point construction
featureName = [feature for feature in COLUMN_HEADERS if feature == splitColumn]
typeOfFeature = FEATURE_TYPES[splitColumn]
if typeOfFeature == "continuous":
    question = "{} <= {}".format(featureName[0], splitValue)

# if feature is categorical
else:
    question = "{} = {}".format(featureName[0], splitValue)

print(question)

# instantiate sub-tree
subTree = {question: []}

# recursion
yesAnswer = myDecisionTreeRegression(dataLeft, featureList, labelCol, counter, minSamples, maxDepth, maxCatValues, bins)
noAnswer = myDecisionTreeRegression(dataRight, featureList, labelCol, counter, minSamples, maxDepth, maxCatValues, bins)

subTree[question].append(yesAnswer)
subTree[question].append(noAnswer)
```

The structure of the total tree will be represented by dictionaries where the key is a node of the tree, it is represented by a string $\mathbf{x} <= \mathbf{y}$ or $\mathbf{x} = \mathbf{y}$, and the value of the key will be an array with the two result of the recursive call for the left and right part of the splitted data [Fig. 12].

4.2.2 Test the model

Figure 13: Construction of the tree's structure

```
def predictLabel(observation, tree, labelCol):
    question = list(tree.keys())[0]
    featureName, operator, value = question.split(" ")

    # ask question
    if condition == "<=":
        if observation[featureName] <= float(value):
            answer = tree[question][0]
        else:
            answer = tree[question][1]

    # feature is categorical
    else:
        if str(observation[featureName]) == value:
            answer = tree[question][0]
        else:
            answer = tree[question][1]

    # base case
    if not isinstance(answer, dict):
        return (example[labelCol], answer)

    # recursive part
    else:
        residual_tree = answer
        return predictLabel(observation, residual_tree, labelCol)
```

To test the tree model, a recursive function was implemented and will navigate through the structure of the tree and as result it will return the label predicted for the observation of the test data [Fig. 13].

5 Experiments

For the experiments the ss13pusa.csv dataset was token in consideration. To evaluate the accuracy of the two algorithms proposed, it has been chosen to use the Root Mean Square error on the testing data.

Root Mean Square error:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2} \tag{16}$$

For the first experiments the 50% of the whole dataset has been considered to train and test the proposed approaches. This experiments have been executed on a Google Colab environment. The dataset was downloaded through the Kaggle API.

Then it has been tested the scalability of the two approaches using the whole dataset on a cluster of machines with the use of Google Cloud Platform and the Dataproc API.

For all the experiments the dataset is divided in 70% for training and 30% for testing.

5.1 Ridge regression GD experiments

For the Ridge Regression model the features have been scaled in order to help the Gradient Descent algorithm converges in a shorter period of time to the solution.

The learning rate has tuned through a grid search where the max iterations was 10 and the values for the learning rate were [0.1, 0.001, 0.0001, 0.00001, 0.000001, 0.0000001, 0.00000001]

The learning rate 0.0000001 has given the best result, values greater than this make the training diverges from the optimal solution as the step is too large.

Then a training with lr 0.0000001 and max iterations 1000 has launched, it took 58 iterations to update the weights until the RMSE didn't change anymore.

The best RMSE obtained was 36834.96

5.2 Decision Tree regression experiments

Also for the training of the Decision Tree model a grid search has been executed in order to tune the hyperparameters **maxDepth** (maximum depth reachable by the tree), the parameter **maxCatValues** that is the threshold with which consider a feature categorical or continuous and the parameter **bins** that generates n values from continuous features to use as candidates for the split of the data.

The best model obtained was using maxDepth = 15 maxCatValues = 2 and bins = 4 with RMSE = 31665.61

5.3 Google Cloud Platform experiments

With the same hyperparameters used for the previous experiments it has been trained the two models using the whole dataset on two types of Spark cluster.

The first one was composed by three machines, one for the master role of the spark framework and two for the worker role. Then another cluster is based on four machines, one for the master role and three for the worker role.

Every machine is a 4 core machine with 15GB RAM.

Cluster	Model	RMSE	Time of training
3-machines	RidgeRegressionWithGD	36917.53	62 min for 53 iterations
4-machines	RidgeRegressionWithGD	37114.12	40 min for 56 iterations
3-machines	DecisionTree	31009.40	11 min
4-machines	DecisionTree	30605.11	8 min

6 Final Considerations

The results show that the Decision Tree model has achieved the best results for the proposed task, this maybe because some of the features has not a strong linear relationship with the label **WAGP** and others have a non-linear relationship that can be caught only by the Decision Tree model. The Decision Tree algorithm had better result also in terms of time to train and construction of the model.

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