UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Data Mining and Machine Learning

AirBnb Price Estimation using Machine Learning

Project Documentation

TEAM MEMBERS:
Edoardo Fazzari
Mirco Ramo

Academic Year: 2020/2021

Contents

1 — Introduction

Housing prices are an important reflection of the economy, and pricing a rental property on Airbnb, therefore, can be a challenging task for the owner as it determines the number of customers for the place. On the other hand, customers have to evaluate an offered price with minimal knowledge of an optimal value for the property. [1]

1.1 Goals

The aim of this paper is to explain the choices and the strategies we adopted on the project and development of **AirBnb Price Estimator**, whose aim is to help owners to decide the most correct price for their *BnB*. In order to accomplish it, we started from web-scraped data, we performed all the preprocessing needed for having a suitable dataset and then we built several classifiers, using different strategies, in order to determine the one that predicts best the class attribute. All these classifiers have been tested using more than one method and the analysis of the results guided us in the choice of the best classifier. Since the class attribute is numeric, we had two possible choices:

1. Discretize the attribute, choosing the most appropriate algorithm
2. Keep it numeric, using regression algorithms for the classification purposes

The first approach is surely easier, but it would not be as helpful as the second one for our application purposes: suggesting a precise value to an owner will give him/her a more accurate advice rather than a range.

The regression model that generalizes best the class feature has then been chosen as the "heart" of AirBnB Price Estimator: the application asks users to input the required fields that correspond to the attributes needed by the classificatory. On these fields bases, it simply outputs to the user the suggested price for night.

1.2 Initial Dataset

The fee of a real estate is strictly related to the city where it is located, thus would force us to recompute for every different city each step we make over and over, wasting time. Thus we decide to consider only one location (i.e, city), big enough to have a huge number of records. We chose *New York City*. The dataset, related to all the **BnB** situated in *NYC*, is taken from `http://insideairbnb.com/get-the-data.html` [2]. The scraping has been performed the 10th November 2020 (the scraped data has been made available by third parties).

The *initial dataset* is composed by 35821 instances and 74 columns. In order not to confuse the reader with useless information, the attribute list is not here reported, however on the following chapter regarding preprocessing we justify in detail the actions performed on the data. For now, the most important things to know is that:

1. The dataset is composed by various mixed features regarding the estate, the host and he geographical position
2. The source file has been published without respecting exactly the *csv* format, thus some frameworks and *csv* file handlers are not able to parse it

[1] GitHub repository for the project: `https://github.com/edofazza/AirBnBPriceEstimator`

[2] In the case of the dataset form the website inside the *listing.csv* file, related to NYC, is updated, we stored the dataset used on Google Drive. It can be downloaded at: `https://drive.google.com/file/d/1KQ2yB6eJOrbSZoyL5fxWJjq72i4ONyQn/view?usp=sharing`

3. The initial data is very dirty: there are missing values, redundant attributes, lists of strings embedded in a single column, pointless features and so on. For all of these problems, a suitable solution has been provided and it is fully reported on the next chapter.

2 — Preprocessing

2.1 What can be preprocessed now?

Since we have to deal with classification problems, before preprocessing data we must be sure not to apply supervised filters on the whole dataset. If we need supervised filters, we must split the dataset in training set and test set before applying them. However, in our case there is no need for a supervised filters but attribute selection, so we performed all the unsupervised preprocessing operations at the beginning, postponing the attribute selection after the training/test split.

2.2 Data Cleaning and Reduction

Since the Weka framework was not able to parse correctly the source file, for the following operations we used *Microsoft Excel* and *Apple Numbers*.

2.2.1 Removing Noisy and Irrelevant attributes

The first preprocessing operation is the attribute reduction, indeed we decided to remove all the features that are not domain-specific, nor useful for classification purposes. Among them we deleted:

- **ID**s: *Bnb_ID, scrape_id, host_id*
- **URL**s: *listing_url, picture_url, host_url, host_thumbnail_url, host_picture_url*
- Scraping informations like *last_scraped_date, calendar_last_scraped*
- Useless information on the rent for classification purposes like *name, description, first_review_date*
- Useless information about the host like *host_name, host_location* (it is not the place in which the rent is, but where the host lives), *host_about, host_has_profile_pic*

2.2.2 Removing Redundant Attributes

The next step was the reduction of attributes explicitly redundant. For all of these we did not compute the χ^2 test because of the explicit correlation between the features

- *host_neighbourhood and neighbourhood* **w.r.t.** *neighbourhood_cleansed* are explicit redundancies; *neighbourhood_group_cleansed* has been demonstrated to be very highly correlated with *neighbourhood_cleansed* ($P\{independence\} < 0.05$)
- *host_total_listings_count* completely equal to *host_listing_count*
- *host verification* **w.r.t.** *host_identity_verified*
- *minimum_minimum_nights, minimum_maximum_nights, maximum_minimum_night, maximum_maximum_nights, minimum_nights_avg* and *maximum_nights_avg* are redundancies of the *attributes minimum_nights* and *maximum_nights*
- *review_scores_accuracy, review_scores_cleanliness, review_scores_checkin, review_scores_communications, review_scores_location, review_scores_value* are rounded values that have been more precisely combined in another already-existing attribute that is *review_scores_rating*
- *calculated_host_listings_count* (+ category) are redundant attributes of *listings_count* valid only for their respective category

2.2.3 Additional Removed Features

Eventually, the other columns that have been discarded are:

1. Attributes that strongly depend from the instant in which the scraping has been performed like *has_availability_now, availability_30, availability_60, availability_90, number_of_reviews_ltm, number_of_reviews_l30*
2. Empty attributes like *license*, *bathrooms*
3. Others, like *latitude* and *longitude* that are not more useful than the easier data on the neighborhood

2.3 Dealing with Missing Fields

Once the previous steps were performed, we achieved a **Weka-convertible CSV file**. Thus, the following steps have been implemented using the *Java Weka API*. The original dataset contains several missing values sparse in more than one attribute. The number of missing values is enough relevant to discourage the instance deletion, although they are pretty easy to handle, indeed:

- The majority of them are on numeric attributes characterized by low variance, like the *response_rate* ($\sigma = 26.17$ on a $0 - 100$ interval) or the *review_score_rating* ($\sigma = 9.52$ on a $0 - 100$ interval). In these cases, replacing the missing value with the mean does not introduces big error rates.
- Some missing values can be easily inferred with the simple analysis by the domain expert: for example, the attribute *bedrooms* contains missing values, but only when the corresponding *beds* value is 1: it's reasonable that the missing value for *bedrooms* is 1 as well.
- Supervised approaches for missing values could have guaranteed more precise results, but since we are exploiting a regression problem and we didn't split training and test sets yet, using a supervised filter here was an error.

2.4 Data Transformation

2.4.1 Attribute Formats Transformation

Some of the attributes persisted after the cleaning and reduction phases of the precedent sections were in a not suitable for the knowledge discovery operation that will be done in chapter 3. Thus, we decide to transform those attribute to make them adequate for classification. The features we transformed are: *amenities* (we will discuss them in chapter 2.4.2), *bathrooms*, and *price*.

The *bathrooms* in the scraped file were two columns, one empty and removed and the other one containing string values as "Shared bath", "1 shared bath", "1.5 bath". We manage to format it in two columns, one referring to the number of bathrooms inside the building and the other telling if the bathrooms are shared or not.

bathrooms	bathroomsShared
1	0
1	0
1	0
1	0
1	0
1	0
1	0
1	0
1	0
1	0
1	1

The *price* attribute in the original file was a string made of *$* plus the amount (e.g., *$170*). We transformed it simply by removing the dollar sign.

2.4.2 One Hot for the Amenities

Amenities, in the original file, were list of string inputed by the used that contain all the facility the specific BnB offers to the guests.

```
["Wifi", "Air conditioning", "Kitchen", "Cable TV", "TV", "Elevator", "Heating"]
```

```
["Cable TV", "Essentials", "Washer", "Heating", "Air conditioning", "Kitchen", "TV", "Dryer", "Wifi"]
```

Structured in this way they were difficult to process. Because many machine learning models need their input variables to be numeric, we need to transform these categorical variables using the *one-hot encoding*. One-hot encoding is a frequently used method to deal with categorical data, since many machine learning models need their input variables to be numeric, categorical variables need to be transformed in the pre-processing part.

"How did we proceed to create the one-hot table?" First, we collect all the distinct values from the entries in the csv amenitites' column and we created new columns using them, through a first scan of the column. Then we made a second scan to compute the values of each of the newly created attribute: if the attribute name is the same of one is contained in the amenities' list we are considering, then we put a value equal to 1, otherwise we put 0. Obtaining something like this:

Cleaning_before_checkout	Coffee_machine	Concierge	Crib
1	1	0	0
0	1	0	1
0	0	0	0
0	0	0	0

2.4.3 Amenities' One Hot refinements

The number of columns obtained was very huge, more than three hundreds, so we needed to reduced them in some ways. We notice that few of them were irrelevant and the most of

them were redundant, for all of these we did not compute the χ^2 test because of the explicit correlation between the features (*e.g.*, Keurig coffee machine, Coffee maker, Nespresso machine) but we simply merge them.

The irrelevant columns we removed through code, they were:

- *Limited housekeeping u2014 on request*: put just by one user and outdated nowadays;
- *Safe*: put just by one user and without a consistent meaning. Without knowing if safe stands that the building is safe or that the BnB is located in a safe location, we decide to remove it.

The columns that were similar to each other were merged forming a new columns made of the values of the grouped columns. Merging was done through code, summing they values in each position:

$$new\_column = \sum_{i}^{similar\ columns} columns_i$$

In the case of values inside the *new_column* greater than 1, we convert back to 1. The merged columns were (*new name for the column* →*columns merged*):

- **Toiletries** →*1802 Beekman toiletries, Diptyque toiletries, Gilchrist & Soames toiletries, Malin+Goetz toiletries, Natura toiletries, Toiletries, Neil George toiletries, C.O. Bigelow toiletries, comfort zone toiletries, Appelles toiletries, Cote Bastide Argan toiletries, Bio Beauty toiletries, Le Labo toiletries, Elemis toiletries, MOR toiletries*
- **Stove** →*Stainless steel gas stove, Wolf stainless steel gas stove, Viking stainless steel gas stove, Frigidaire stainless steel gas stove, Ge stove, We provide a portable gas stove in our kitchenette. gas stove, GAS COOK TOP ONLY NO OVEN gas stove, GE stove, Fridgedare 30 inches stainless steel gas stove, Magic Chef gas stove, Samsung stainless steel gas stove, Stove, Electric stove, LG Stove stainless steel electric stove, Fridgedare stainless steel gas stove, Stainless steel electric stove, Stainless steel stove, Gas stove, Single burner countertop range electric stove, Induction stove, 2 burner hot plate electric stove, Small portable induction stove electric stove, Two Burner Electric Cook-Top electric stove, GE electric stove, Stovetop works - Oven does not gas stove*
- **Refrigerator** →*Magic Chef refrigerator, LG refrigerator, Samsung refrigerator, small refrigerator, Stainless Steel Fridgedare refrigerator, Undercounter refrigerator, bloomberg refrigerator, Gaggenau refrigerator, Kenmore refrigerator, Undercounter Refrigerator refrigerator, Bosch refrigerator, Subzero refrigerator, Beko refrigerator, Whirlpool refrigerator, Americana refrigerator, Magic Chef refrigerator, Refrigerator, Fridgedare Stainless Steel refrigerator, Sub Zero refrigerator, LG smart Tech refrigerator, Inc refrigerator, Frigidaire refrigerator, GE refrigerator, Ge refrigerator*
- **Sound system** →*Built-in sound system in the apartment. sound system, Tivoli Audio Bluetooth sound system, Bluetooth sound system, Unknown - you can plug right into phone sound system with aux, Sonos over WiFi with built-in speakers throughout the house and backyard sound system, BOSE sound system with Bluetooth and aux, Marshall sound system with Bluetooth and aux, Marshall Bluetooth sound system, Bose Surround Speaker System in All Rooms sound system with Bluetooth and aux, Roku Bluetooth sound system, roku tv Bluetooth sound system, Echo Dot Bluetooth sound system, bose speaker Bluetooth sound system, Sound system, Bose sound system, Sound system with Bluetooth and aux, Yamaha Bluetooth sound system, Sonos sound system, Sonos Bluetooth sound system, Marshall sound system with Bluetooth and aux, Harman Kardon Bluetooth sound system, Cambridge Audio Bluetooth sound system, Sound system with aux, Samsung Bluetooth sound system, Bose sound system with Bluetooth and aux, Bose Bluetooth sound system, Yamaha sound system with Bluetooth and aux*

- **Linens** →*Supmia linens, Sferra linens, Bed linens, Frette linens, Sferra linens, linens*
- **Breakfast** →*Cooked-to-order breakfast available u2014 $30 per person per day, Breakfast buffet available u2014 $25 per person per day, Complimentary breakfast, Cooked-to-order breakfast available u2014 $25 per person per day, Complimentary continental breakfast, Hot breakfast available u2014 $20 per person per day, Complimentary hot breakfast, Cooked-to-order breakfast available for a fee, Breakfast, Cooked-to-order breakfast available u2014 $15 per person per day, Continental breakfast available u2014 $29 per person per day*
- **Air conditioning** →*ICE Air conditioner, Central air conditioning, Air conditioning, Portable air conditioning*
- **Dryer** →*Dryer, Hair dryer, Dryer u2013u00a0In unit, Dryer u2013 In building*
- **Extra pillows** →*Extra pillows and blankets, Bed sheets and pillows*
- **Parking** →*Free street parking, Paid parking lot on premises, Paid parking on premises u2013 1 space, Self-parking u2014 $35/day, Valet parking u2014 $65/day, Valet parking u2014 $75/day, Paid street parking off premises, Valet parking u2014 $80/day, Paid parking garage on premises u2013 1 space, Valet parking u2014 $45/day, Paid parking on premises, Paid parking off premises, Self-parking u2014 $19/day, Free driveway parking on premises, Paid parking lot on premises u2013 1 space, Free driveway parking on premises u2013 1 space, Self-parking u2014 $40/stay, Valet parking u2014 $85/day, Paid parking garage on premises, Valet parking u2014 $70/day, Free parking on premises, Paid valet parking on premises, Self-parking u2014 $50/day, Paid parking lot off premises, Paid parking garage off premises, Valet parking u2014 $40/day*
- **Wifi** →*Wifi u2013 500 Mbps, Pocket wifi, Wifi u2013 60 Mbps, Wifi u2013 200 Mbps, Wifi u2013 100 Mbps, Wifi u2013 24 Mbps, Free wifi, Wifi u2013 870 Mbps, Wifi, Wifi u2013 400 Mbps*
- **Oven** →*Frigedare stainless steel oven, Frigidaire stainless steel oven, Oven, GE oven, Stainless steel oven, Toaster oven oven, Samsung stainless steel oven, Fridgedare oven, Power Airfryer 360 stainless steel oven, Small portable oven oven, Wolf stainless steel oven, Frigidaire oven, Viking stainless steel oven, electric stainless steel oven, large toaster oven oven*
- **Garden** →*Onsite bar u2014 Clinton Hall & Rooftop Beer Garden, Garden, Onsite restaurant u2014 Clinton Hall & Rooftop Beer Garden, Garden or backyard*
- **Heating** →*Heating, Radiant heating, Central heating*
- **Kitchen** →*Kitchenette, Kitchen*
- **Onside bar** →*Onsite bar u2014 Gleason's Tavern, Onsite bar u2014 Crown Shy, Onsite bar u2014 Molyvos Restaurant - Bar, Onsite rooftop bar u2014 Make Believe, Onsite bar u2014 The National, Onsite bar, Minibar, face&body bar Bergman Kelly body soap, Onsite bar u2014 The Seville, Barbecue utensils*
- ***Onside restaurant*** →*Onsite restaurant u2014 Above SIXTY SoHo, Onsite restaurant u2014 Gleason's Tavern, Onsite restaurant u2014 Butter, Onsite restaurant u2014 Parker & Quinn, Onsite restaurant u2014 Blue Ribbon Sushi Izakaya, Onsite restaurant u2014 Caf u00e9 Hugo, Onsite restaurant u2014 Scarpetta, Onsite restaurant u2014 Park Cafe, Restaurant, Onsite restaurant u2014 Caf u00e9 Boulud, Onsite restaurant u2014 Broome Caf u00e9, Onsite restaurant u2014 The National, Onsite restaurant u2014 Blue Park Kitchen*

- ***Pool*** *→Pool, Outdoor pool*
- ***Washer*** *→Washer u2013 u00a0In building, Washer, Dishwasher, Washer u2013 u00a0In unit*
- ***Hot water*** *→Hot water, Hot tub*
- ***Gym*** *→24-hour fitness center, Gym, Fitness center*
- ***Coffee machine*** *→Keurig coffee machine, Coffee maker, Nespresso machine, Pour Over Coffee*
- ***Clothing storage*** *→Clothing storage, Clothing storage-closed*

2.5 Preprocessing Implementation

As already mentioned before, the preliminary operations needed to make the *csv* readable by **Weka** have been performed using spreadsheet software. The resulting cleansed file still needs some modifications, *i.e.*:

- As reported in par. 2.3, missing values need to be managed
- As widely discussed in par. 2.4, the *amenities* must be converted in a **One Hot**
- On the *price* attribute, we get rid of the $ sign and we convert it into numeric
- The *bathroom* attribute needs its format to be changed

After the loading of the *CSV* file, in order to speed up the operations we parallelized them using 4 different threads: the dataset is vertically split and every thread works only on its related partition, using *Java Weka API* or working directly on the text according to what was the fastest approach in every single scenario; every execution flow, before ending, writes results in a *CSV* file. The main thread spawns a thread for each of the tasks listed before, and then waits the termination of all of them before merging the results in a single file.

In addition to the benefits of parallelism, this approach promotes the separation of concerns of the tasks: the main method defines the preprocessing pipeline, but the actual operations on data are performed by separated and independent components. All the classes implemented at this point are collected in the *com.unipi.dmaml.airbnbpriceestimator.preprocessing* package.

3 — Classification

After the preprocessing phase, the dataset is ready to be used to learn regression models that will be used in the final application. In the following chapter all the chosen strategies are discussed, and the results are compared.

3.1 Strategies

3.1.1 Train and Test Splitting

All the classifiers have been evaluated with the same strategy: *10-fold cross validation*. This means that a model is learned from the folds of the training set (90%) and they are evaluated against the corresponding test fold (10%); then the operation is repeated 10 times, and at each iteration a different fold acts as test set.

Despite the fact that Weka provides the **Evaluation.crossValidateModel()** method, we wanted to save the results of every fold and to have access to the selected attributes in case of algorithm with feature selection.

For these reasons, instead of using a meta classifier, we manually split training set and test set through the *Weka* provided functions **Instances.trainCV(numFolds, currentFold)** and **Instances.testCV(numFolds, currentFold)** according to the *WekaWiki tutorial* on how to manually construct a *k-fold cross validation*; then the *attribute selection* algorithms (when present) have been built exclusively on the training set but applied both to training and to test set.

Finally, we saved the results of each fold, but since we need only a model of the classifier while this procedure generates 10 different models, we picked the one with the lowest *root mean squared error*, aware that this does not necessarily means that we chose the best one

3.1.2 Chosen Classifiers

According to our application domain, the classifiers needed are actually regression algorithms capable of handling our non-nominal class. The selected algorithms have been tested both on the full dimensionality of the dataset (138 features) and on the reduced subspaces identified by the supervised attribute selection methods. Once again, we underline the fact that those features selection algorithms have been modeled exclusively on the bases of the training set. They are *CfsSubsetEval+BestFirst* and *CfsSubsetEval+GreedyStepwise*.

We couldn't exploit the *InfoGain* evaluator since it is not capable of working with numerical classes; we discarded *PCA* since it works transforming dimensions, thus it would be difficult to understand if we could have afforded to ask only for a limited number of parameters to input from the user in the final application (**e.g.**, if *CfsSubsetEval+BestFirst* selects only 3 attributes, we can create an application that requires only 3 parameters as input. With *PCA* we don't know which original features has been chosen in order to generate the new space, so we are forced to ask the user to input alle the 138 parameters); eventually, we discard the use of a wrapped classifier in order not to deteriorate much the performance of the application.

To summarize, the tested classifiers are:

- *Linear Regression, Linear Regression with attribute selection*
- *Random Forest, Random Forest with attribute selection*
- *5-NN, 5-NN with attribute selection*
- *M5Rules with and without attribute selection*

3.2 Building Classification Models

On the following pictures the implementation of the classifiers is reported. The main method is not here shown, it simply loads data and triggers the algorithm. A multithreaded approach has been tested but not implemented since the huge amount of principal memory consumption due to the model building made the system fail more than once.

3.2.1 Data Loading

```
public Instances loadData() {
  Instances data=null;
        try {
            ConfigurationData configurationData = getConfigData();
            CSVLoader loader= new CSVLoader();
            loader.setDateAttributes(configurationData.dateAttributes);
            loader.setDateFormat(configurationData.dateFormat);
            loader.setEnclosureCharacters("\"");
            loader.setFieldSeparator(",");
            loader.setMissingValue("?");
            loader.setNominalAttributes(configurationData.nominalAttributes);
            loader.setNumericAttributes(configurationData.numericAttributes);
            loader.setSource(new File(datasetPath));
            data=loader.getDataSet();
        } catch (Exception e){
            e.printStackTrace();
        }
        return data;
}

private ConfigurationData getConfigData() throws JAXBException {
        JAXBContext jaxbContext = JAXBContext.newInstance(ConfigurationData.
    class);
        Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
        ConfigurationData conf = (ConfigurationData) jaxbUnmarshaller.unmarshal
    (new File(confDataPath));
        System.out.println(conf.dateAttributes + "\n" + conf.dateFormat + "\n"
    + conf.numericAttributes);
        return conf;
}
```

DatasetFromCsvLoader.java is the class in charge of loading the dataset stored as a *CSV file*. It simply uses the *Weka CSVLoader* but the options have been saved in a configuration file, so that to maximize the separation of concerns between the *Weka* internal representation of the data and our code.

3.2.2 Classifier Definition

```
private void buildLinearRegressionWithAttributeSelection() {
        try{
            Instances randData = new Instances(dataset);
            randData.randomize(new Random(1));

            for(int i=0; i<numFolds; i++){
                AttributeSelection filter = new AttributeSelection();
                filter.setEvaluator(new CfsSubsetEval());
                filter.setSearch(new BestFirst());
                executeCV(randData, i, filter, "CfsSubsetEval+BestFirst");
            }

            randData = new Instances(dataset);
```

```
            randData.randomize(new Random(1));

            for(int i=0; i<numFolds; i++){
                AttributeSelection filter = new AttributeSelection();
                filter.setEvaluator(new CfsSubsetEval());
                filter.setSearch(new GreedyStepwise());
                executeCV(randData, i, filter, "CfsSubsetEval+GreedyStepwise");
            }

            System.out.println("linear regression with attribute selection terminated");

        } catch (Exception e) {
            e.printStackTrace();
        }
}
```

Every algorithm is implemented in a separate class, which contains different methods based on the way we want to test it (*with* or *w/o* attribute selection). In the code shown the classifier definition is reported: after a randomization of the dataset, we call *numFolds* time the **executeCV()** that performs a single round of the *cross-validation*.

3.2.3 Classifier Implementation

```
private void executeCV(Instances dataset, int currentFold, AttributeSelection filter, String filterName) throws Exception {
        Instances train = dataset.trainCV(numFolds, currentFold);
        Instances test = dataset.testCV(numFolds, currentFold);
        List<Attribute> chosen=new ArrayList<>();
        if(filter!=null) {
            filter.setInputFormat(train);
            Instances trainReduced = Filter.useFilter(train, filter);
            Instances testReduced = Filter.useFilter(test, filter);
            train = new Instances(trainReduced);
            test = new Instances(testReduced);
            for(int i=0; i<train.numAttributes(); i++)
                chosen.add(train.attribute(i));
        }
        weka.classifiers.functions.LinearRegression classifier = new weka.classifiers.functions.LinearRegression();
        classifier.buildClassifier(train);
        Evaluation evaluation = new Evaluation(train);
        evaluation.evaluateModel(classifier, test);
        new FileSaver(evaluation, "LinearRegression", filterName, currentFold, chosen).save();
        if(currentFold==8)
            saveModel(filterName, classifier, test);
    }

private void saveModel(String filterName, weka.classifiers.functions.LinearRegression classifier, Instances dataFormat) throws Exception{
        SerializationHelper.write("models/LinearRegression_" + filterName + ".model", classifier);
        ArffSaver saver = new ArffSaver();
        saver.setInstances(dataFormat);
        saver.setFile(new File("models/LinearRegression_" + filterName + "_data.arff"));
        saver.writeBatch();
    }
```

In this method we perform a round of the **10-fold cross validation**, in the way suggested by the *WekaWiki tutorial*. At each iteration, the *trainCV* and *testCV* return non-overlapping

training sets and test sets; if an attribute selection method has been defined, it is built on top of the training set and applied to both of them and the list of the chosen attributes is stored. In the end, the model built on the **fold n.8** is stored in a *.model* file.

3.3 Performance Evaluation and Effects of Attribute Selection

Once the classifiers have been built, as anticipated in the paragraph 3.1 we tested them using a **10-fold cross validation**. In the following table we report the average values respectively of *correlation coefficient (CC), mean absolute error (mae), root mean squared error (rmse), relative absolute error (rae)* and *root relative squared error (rrse)* for each tested classifier (note that the chosen model, *i.e.* the one built in the 8^{th} fold, shows better measures than the average for every tested classifier). The green-colored values are the best ones in absolute.

ATTIBUTE SELECTION ⇒ REGRESSION ALGORITHM ⇓	**None**	**CfsSubsetEval + BestFirst**	**CfsSubsetEval + GreedyStepwise**
Linear Regression	Out of memory error	$\overline{CC} = 0,6793$ $\overline{Mae} = 40,0323$ $\overline{Rmse} = 57,8399$ $\overline{Rae} = 66,9399$ $\overline{Rrse} = 73,3966$	0,6794 40,0308 57,8339 66,9375 73,3891
Random Forest	0.7495 36,0881 52,8727 60,3423 67,0864	0.6988 37,9716 56,3332 63,4916 71,7302	0,7000 37,9451 56,5060 63,4470 71,6955
5-NN	0,5413 46,4003 67,0074 77,2522 84,3622	0,6772 39,0709 58,0759 65,3630 73,6845	0,6772 39,0709 58,0759 65,3630 73,6845
M5Rules	0,6589 567,0975 31.491,7374 942,2971 39.372,3750	0,6942 38,5873 56,7530 64,5234 72,0102	0,6942 38,5872 56,7498 64,5211 72,0100

As shown in the table, *Random Forest* without attribute selection is the algorithm that performs best **w.r.t.** all the parameters. Note that the attribute selection improved the outcomes of certain algorithms, anyway, deteriorating the ones of *Random Forest*. Now we will discuss the pros and cons of each of them, justifying the final choice for our application:

1. All the classifiers without attribute selection are, unfortunately, not very suitable for the usage of the application: they would require the user to input 288 parameters, which is a huge amount of work to do for him/her. On the contrary, all the attribute selected classifier needs less than 30 features.

2. *Linear Regression* shows good but not awesome results, it requires not much memory for the model loading and it is very fast at prediction time. However, it showed very long times and high memory consumption at training time.

3. *Random Forest* is the one with the best results in absolute, both with and without attribute selection. It is quite fast both to train and to use for prediction, anyway it needs a lot of time and memory for the model to be saved and loaded. Since the considerations at the point 0), we discarded the pure *Random Forest* and we chose *Random Forest with CfsSubsetEval+GreedyStepwise* as the default regression model for our application.

4. *K-NN* is definitely inadequate, although it requires very little time and memory for the model to be built (and then also to be loaded), the computational effort is concentrated at prediction time (thus increasing the application response time), and it also shows quite poor results.

5. *M5Rules* has very good performances and it needs very little time and memory to save and load the model. The prediction is not so fast, the time required to build the model is terribly long (up to 3 or 4 times more than Random Forest). Anyway, since the majority of the effort is at training time, it is suitable for our application, thus *M5Rules with CfsSubsetEval+GreedyStepwise* has been chosen as backup regression model.

3.4 Evaluation of Significance of the Classifiers' Results

Once evaluated the performance of the algorithms, the main question is if we can consider those differences statistically significant or not. In order to make such a computation, we used the paired **Student's t-test** on the *RMSE* considering all the outcomes of all the folds (of every algorithm we chose only the *CfsSubsetEval+ GreedyStepwise* version). The results are the following ($sig\_rate = 5\%, deg = 9$):

- *Random Forest* **w.r.t.** *Linear Regression*: t=5.5342; p=0.004 $\Rightarrow$ stat. sign.
- *Random Forest* against *M5Rules*: t=0.805; p=0.4413 $\Rightarrow$ non stat. sign.
- *M5Rules* against *Linear Regression*: t=5.5104; p=0.032 $\Rightarrow$ stat. sign.
- *Linear Regression* against *KNN*: t=0.6601; p=0.5257 $\Rightarrow$ non stat. sign

Random Forest and *M5Rules* are hence comparable, so they are equally valid for our application. On the contrary, the **t-test** shows that *Linear Regression* 's results are much more similar to *KNN* ones rather than to the outcomes of the other two algorithms, thus it has been discarded.

4 — AirBnb Price Estimator App

4.1 Introduction

AirBnB Price Estimator is a real estate cost estimation application in which the *Users* insert some informations related to the B&B they want to compute the price.

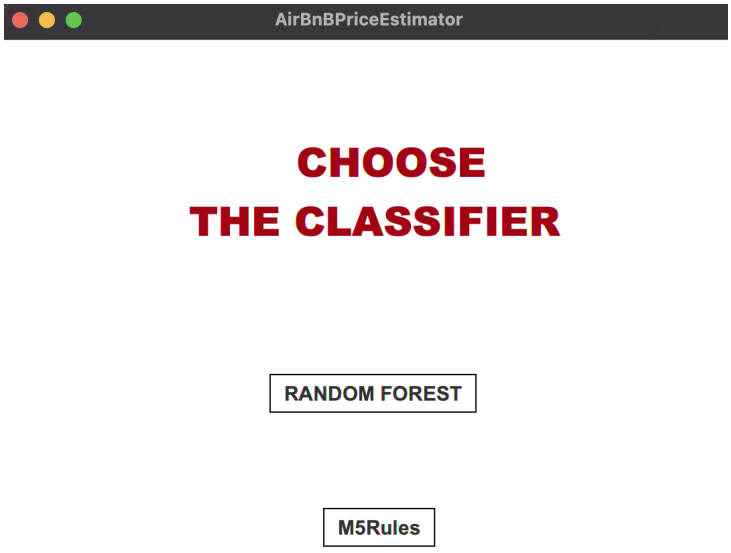
4.2 Use Cases List

There is no need for registration, every user can operate with the *applicative* as they open it. The **use cases** are:

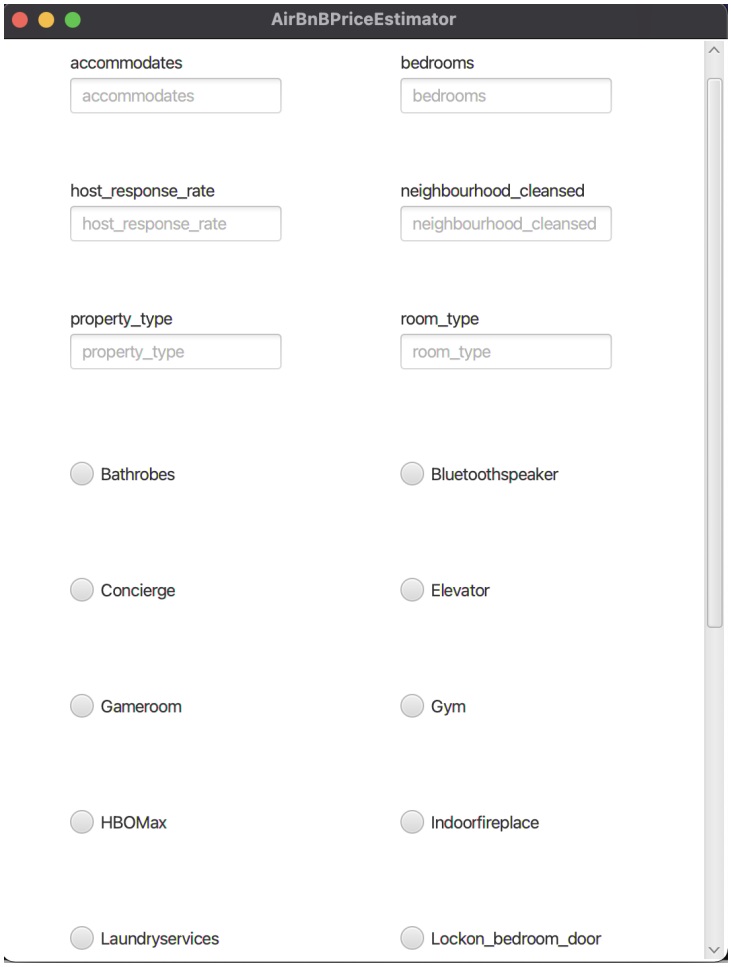
- Select the classifier to use
  - *Random Forest*
  - *M5Rules*
- Insert values for requested B&B related characteristics
- Compute the price using the algorithm previously selected

4.3 Application Guide

When the application is launched a window is opened telling the *User* to choose between the two classifier previously discussed. The *User* cannot undo the choose, if he/she clicks the incorrect classifier he/she has to reopen the application.



Selected the classifier, a new scene is loaded. This scene displays some *TextField*s and *RadioButton*s, those refer to a subset of the overall features in out dataset that is evaluated in the *Attribute Selection* process we talked about in chapter 3.3.



If a *RadioButton* is not selected the system will understand that as the absence of the particular amenity the *RadioButton* refers to.

After the *User* imputes all the values a *Button* saying *"PREDICT PRICE"* is located at the button of the window (scroll down if you don't see it). After clicked on it, a label displaying the predicted price is shown below the *Button*.