Recommendation system for restaurants

Project of the Machine Learning course
Academic year 2018-2019

Giovanni Ficarra, Leonardo Picchiami

September 9, 2019

Contents

List of Tables

List of Figures

Listings

1 Introduction

1.1 Problem description

We have devised a recommendation system for restaurants based on the Yelp Dataset Challenge.

Our aim is to predict if a certain user will like a certain restaurant, depending on characteristics of the restaurant, user's taste (derived from his previous reviews), opinion of similar users (who gave similar votes to similar restaurants), trustworthiness of the reviews.

In order to do so, we studied three papers, that we used as the basis of our work:

1. Restaurant Recommendation System, Ashish Gandhe [3];
2. Machine Learning and Visualization with Yelp Dataset, Zhiwei Zhang (with her repo) [4];
3. Recommendation for yelp users itself, Wenqi Hou, Gauravi Saha, Manying Tsang [5].

We started from the data cleaning performed by Hou, Saha and Tsang (3), then we applied the SVM model proposed by Zhang (2) to identify fake reviews, and assigned to each review a "truth score" (an indicator of the trustworthiness of the review), so that we could use this score as a weight in the computation of the historical features described by Gandhe (1).

After some further preprocessing we applied three machine learning models to the obtained dataset and got our predictions.

1.2 Datasets

The Yelp dataset we used is a available here and described in detail here.

It is composed of five JSON files:

1. *business.json* contains data about businesses (we've narrowed our search to restaurants only, but there were hotels and shops too) including location data, attributes and categories;
2. *review.json* contains full review text, the ids of the user who wrote it and of the restaurant it was about, and the number of stars given as a vote (we used the texts only for the detection of deceptive reviews);
3. *user.json* contains data about users, such as popularity, friends and name;
4. *checkin.json* contains the checkins on a business (we discarded this dataset);
5. *tip.json* contains tips written by a user on a business (we dropped the text and kept only the "compliment count", as a sign of the reliability of a user).

The most significant features are written in or obtained from the review file (2), since the label we want to predict, likes/dislikes, is calculated from the number of stars assigned by a user to a restaurant, and so are some of the historical features presented in Ghande [3] and discussed in a separate section [2.3].

The review dataset is also the biggest one, with more than 1 million reviews, spanning from 2014 to 2018, and more than 5GB in size.

1.3 Tools used

We summarize and motivate the tools used for this project:

- **Python 3**: the most widely used programming language for machine learning tasks, we used version 3.6 for compatibility with Tensorflow GPU;

- **Jupyter Notebook**: we decided to write our main scripts in a notebook since it's more readable and comments are clearer, even if it gives some problems, with the multiprocessing library for example;
- **Pandas**: we used this library to read and manage the datasets, but it appeared to be inherently inefficient, since it doesn't do anything to overcome the Python GIL, so we tried to parallelize it using Modin and Numba, but also those attempts have failed, so we used the previously mentioned multiprocessing module of the standard library;
- **Scikit-Learn**: it provides ready-to-use implementations of many ML non-neural models;
- **TensorFlow GPU**: it provides the implementations of neural networks.

Thus, most of our code is in the file `/src/main_notebook-multiprocess.ipynb`, the functions that are explicitly run on many processes are in `src/multiproc_utils.py`, the portion of preprocessing customized from [5] is in `/src/recommendation_system_preprocessing.ipynb` and the model for finding deceptive reviews from [4] is in `/Yelp_Sentiment_Analysis/Scripts/fake_reviews.ipynb`.

The machine used for running most of the code is an Asus notebook with Intel Core i7 processor with 6 physical cores (12 virtual cores), 16GB of RAM, Nvidia GeForce GTX 1050 GPU.

2 Preprocessing

2.1 Data cleaning

As the first thing we did with our datasets, we just applied the data cleaning by Hou, Saha and Tsang [5] with some adaptation to our purpose, e.g., we want to consider restaurants in all the available cities and not only Las Vegas.

What we do based on their kernel is:

1. Transfer json into Pandas dataframe with proper indexing;
2. Extract data that includes all restaurants;
3. Replace garbage data which includes incorrect states and postal codes;
4. Date transformations and standardization;
5. Create new explanatory features based on initial features;
6. Delete consequently unnecessary columns which could add ambiguity based on new features;
7. Delete duplicate restaurants entries and combine their reviews;
8. Save obtained datasets in pickle format so that they occupy less space in memory.

The main features they work on are `categories`, `attributes` and `hours` in the *business* dataset, renamed *restaurants* since other kind of businesses are not considered.
The feature `categories` contains a list of kinds of businesses, ranging from shops to sport centers, so only the items that contains `"restaurant"` in that list are kept; then the most frequent and significant categories are extracted and saved in a new feature called `cuisine`, that we used to infer the users' preferences and to compute the historical features [2.3].
The feature `attributes` contains a dictionary of characteristics of the restaurant, such as if it offers alcohol or if it has wifi connection, but these dictionaries are a bit messy, so the authors create a new feature for each attribute and make the values homogeneous, for example `wifi` has values such as `"No"`, ``"u`no'"``, ``"`no'"``, `` `None' `` and only `"No"` is kept.
The feature `hours` contains a dictionary of the form `{"day_of_the_week": "opening_hour-closing_hour"}`, so it's not very usable, thus the authors divide this features in two features per day of the week: one for the opening and one for the closing.

2.2 Fake Review Detection

With our data clean and tidy, we used some models described by Zhang in [4] to detect deceptive reviews and associate a *trustworthiness score* to each of them.

We trained all of their models: a Convolutional Neural Network, fed with sequences obtained by a tokenizer from the Keras preprocessing library, and a Support Vector Machine, fitted on vectors computer by Scikit Learn's TfidfVectorizer.
It turned out that the performances of the Convolutional Neural Network were worse with respect to the Support Vector Machine, as we can see in the tables [1] and [2].

Thus, we used the SVM to compute predictions and establish which reviews are reliable and which are deceptive, in a binary fashion, so we added to the review dataset a feature `bin_truth_score` that holds a 1 for true reviews and a -1 for fake ones.

In order to have a probability distribution instead of a binary result, the SVM is passed as a parameter to a Scikit Learn's CalibratedClassifierCV, whose predictions are the probabilities that a review is authentic, instead of binary results, and we used them to add a feature `real_truth_score` to the review dataset.

Once this was done, we removed the texts of the reviews.

In the figures [1] and [2] we can see the distribution of the two kinds of labels.

	precision	recall	f1-score	support
-1	0.18	0.18	0.18	24070
+1	0.87	0.88	0.88	158468
micro avg	0.78	0.78	0.78	182538
macro avg	0.53	0.53	0.53	182538
weighted avg	0.78	0.78	0.78	182538
accuracy	78.3%			

Table 1: Report for CNN model for fake review detection

	precision	recall	f1-score	support
-1	0.84	0.95	0.89	161016
+1	0.96	0.86	0.91	211105
micro avg	0.90	0.90	0.90	372121
macro avg	0.90	0.91	0.90	372121
weighted avg	0.91	0.90	0.90	372121
accuracy	89.99%			

Table 2: Report for SVM model for fake review detection

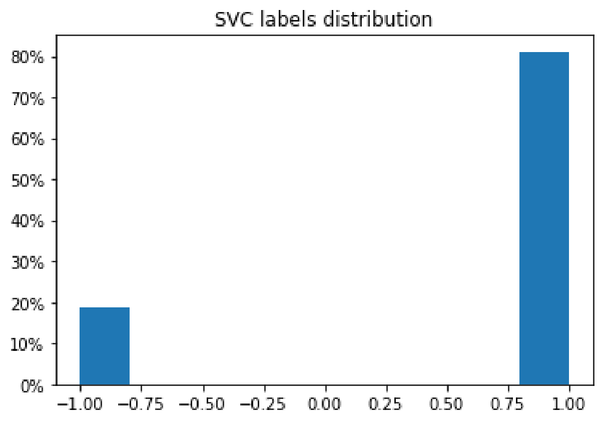

Figure 1: SVC truth labels

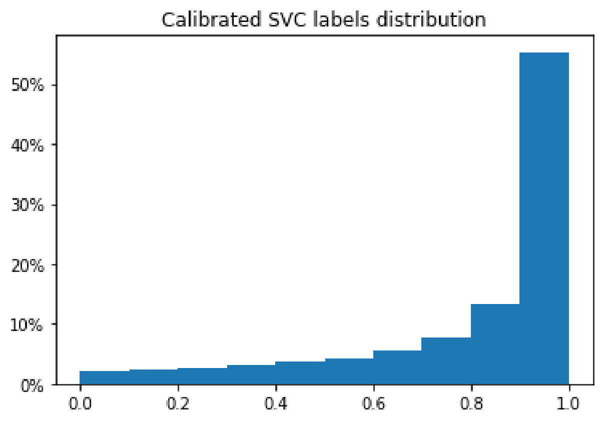

Figure 2: Calibrated truth labels

2.3 Historical features

Based on Gandhe's paper [3], we have added some *historical features* to our dataset:

1. User-level features:
   1.1. average of the ratings given by a certain user,
   1.2. number of reviews written by a certain user;
2. Business-level features:
   2.1. average of the ratings given to a certain restaurant,
   2.2. number of reviews written about a certain restaurant;
3. User-business-level features:
   3.1. average rating given by a certain user to each cuisine (feature defined in section Data cleaning), i.e., for each cuisine, the average of all the votes given by a user to all the restaurants that contains that cuisine in their list,
   3.2. average of the ratings given by a certain user to the cuisines of a certain restaurant, i.e., the average of the votes given by a user to the cuisines present in the list of a restaurant, as computed in the feature (3.1.).

In order to do this, we had to split the review dataset in three parts:

- *Test set*, from the last day considered in the dataset, to the previous m months;
- *Training set*, from the day before the beginning of the test set, up to n months before;
- *History*, the remaining part of the dataset, used to compute historical features.

We have picked $m = 3$ and $n = 8$, so the test set goes from 9/1/2018 to 11/30/2018, the training set goes from 1/1/2018 to 8/31/2018, the history contains the remaining data, from 10/12/2004 to 12/31/2017.

While the other features were quite straightforward to implement and fast to compute, we had to optimize the code of the feature (3.1.) in order to prevent its computation from taking more days, so we tried two libraries that are supposed to automatically speed up a python program running it on more processors, with a minimal intervention by the programmer. The first, Modin, is specifically meant for Pandas, and the second, Numba, more generically for Python math operations, but none of them actually worked, the former because of some incompatibility with Windows OS, and the latter because it "likes NumPy" (as written in the docs) but not Pandas. So, we decided to manually split the datasets and pass portions of them to a function launched in parallel on multiple processors by a process pool, and this reduced the computation time to just few hours.

We decided to apply our two estimators of the trustworthiness of the reviews (described in Fake Review Detection) to compute two additional version of the just described features, so we have three flavors for each of these features:

1. *standard*: each review is treated in the same way (every review has weight 1),
2. *binary*: the authentic reviews have weight 1 and the deceptive ones have weight 0, accordingly to their `bin_truth_score`, so fake reviews aren't counted in total or average,
3. *real*: each review is weighted with its `real_truth_score`, both for the total and for the average.

2.4 User-based collaborative approach

Then, we decided to add a new feature based on *collaborative filtering*: the idea is that we have lots of reviews written by lots of users, so we can use the score given by similar user to similar restaurants to predict what a certain user will think about a restaurant he never tried before. So we applied the following formula:

$$pred(u, r) = a_u + \frac{\sum_{u_i \in U} sim(u, u_i) * (a_{u_i,r} - a_r)}{\sum_{u_i \in U} sim(u, u_i)} \tag{1}$$

where

- a_u is the average of the votes given by the user u to all the restaurants that share some cuisines with r (i.e., the feature 3.2. between user u and restaurant r defined in section Historical features),
- U is the set of all the users in the dataset, u excluded,
- $a_{u_i,r}$ is the same as a_u, but for user u_i, so it is the value of the feature 3.2. for user u_i and restaurant r,
- a_r is the average rating received by r (i.e., the feature 2.1.),
- $sim(u, u_i)$ is computed as the cosine similarity between two vectors that represent the two users u and u_i, composed by the values for the features 3.1. (one for each cuisine and one for each version, as explained in Historical features).

We have three versions for this feature too, with the same meaning described in the previous section.

2.5 Dimensionality reduction and further preprocessing

At this point we had all the features we needed, so, that was the moment of preparing the data so that they were ready to be fed to the models. The next steps we performed were the following:

1. We joined all the interesting datasets in only one, to have all the needed features together and one review per row (the *checkin* dataset was discarded because not useful);
2. We added the label `likes` that holds a 1 if that user gave 4 or 5 stars in that review or 0 if he/she gave 1, 2 or 3 stars;
3. We removed unnecessary columns sush as `review date, restaurant name, restaurant address, user name`;
4. We applied some dimensionality reduction (more details on this later);
5. We filled missing values with the mode of the feature for the categorical values and with the mean for the numerical ones;
6. We converted categorical features into numerical features, readable by the models:
   - for the features `OutdoorSeating, BusinessAcceptsCreditCards, RestaurantsDelivery, RestaurantsReservations, WiFi, Alcohol, city` we used Pandas' `get_dummies()` function, that applies one hot encoding on the input features,
   - for the features `Monday_Open, Tuesday_Open, Wednesday_Open, Thursday_Open, Friday_Open, Saturday_Open, Sunday_Open, Monday_Close, Tuesday_Close, Wednesday_Close, Thursday_Close,Friday_Close, Saturday_Close, Sunday_Close, postal_code` we applied Scikit learn's `OrdinalEncoder.fit_transform()`, that encodes categorical features as an integer array: it could bias the models but it is less memory consuming with respect to one hot encoding, so we preferred this method for features that we consider less determinant and that have many possible values.

Since our dataset was huge (about 7GB), we decided to apply some dimensionality reduction in order to make it fit in RAM, especially when applying grid search (it turned out that to make parallelism more efficient the Scikit Learn's method for grid search tries to make a copy of the dataset for each parallel job that it runs).
We had two features that seem to be quite important and that could assume hundreds of values: `city` and `categories`. First of all, we plotted the distribution of the values of these features, as shown in the

figures [3] and [4]: it appears that there are few very frequent values (such as `'Las Vegas', 'Phoenix', 'Toronto'` for the cities and `'Food', 'Nightlife', 'Bars'` for the categories) and lots of values that are pretty rare, thus we decided to cut the less frequent values before proceeding with one hot encoding, so we choose a threshold $\theta = 100$ and replaced all the values that occur less than θ times with `'other'`. Let's note that, since after one hot encoding we have one column for each category, the previously created `cuisine` feature (see Data cleaning section) has become useless, so we removed it.
In the tables [3] and [4] are listed the ten most frequent and ten least frequent cities/categories.

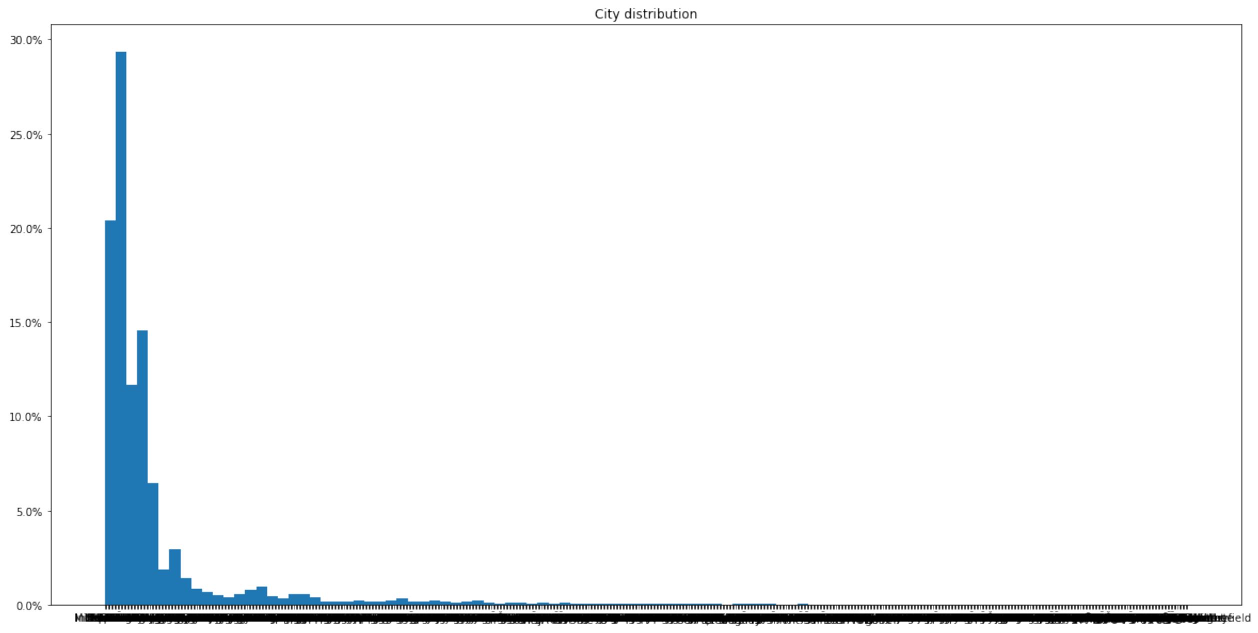

Figure 3: City distribution

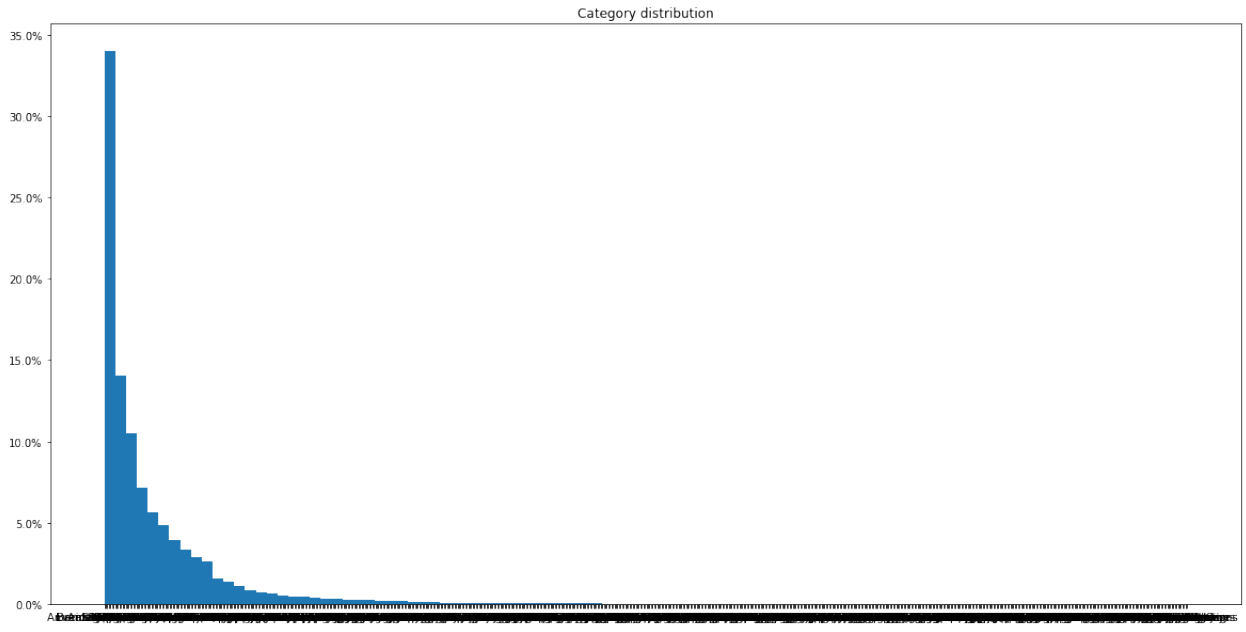

Figure 4: Category distribution

city	# occurrences
Las Vegas	208437
Phoenix	71126
Toronto	57047
Charlotte	40190
Scottsdale	36579
Pittsburgh	26891
Henderson	21518
Montreal	18531
Tempe	18354
Mesa	17235
...	...
Paw Creek	1
Tottenham	1
springdale	1
Coteau-du-Lac	1
Huntingdon	1
Fabreville	1
De Winton	1
Napierville	1
Laval, Ste Dorothee	1
Les Coteaux	1

Table 3: City frequencies

category	# occurrences
Food	192841
Nightlife	170259
Bars	165959
American (Traditional)	123975
Breakfast & Brunch	120310
American (New)	117437
Sandwiches	82264
Mexican	73726
Burgers	71598
Pizza	67538
...	...
Beer Hall	1
Banks & Credit Unions	1
University Housing	1
Pet Groomers	1
Gardeners	1
Home Health Care	1
Campgrounds	1
Holiday Decorations	1
Roofing	1
Senegalese	1

Table 4: Category frequencies

When parameter tuning was finished and grid search wasn't needed anymore, we tried to fit the obtained models with the original dataset, without applying dimensionality reduction and restricting the potentially dangerous use of `OrdinalEncoder` `postal_code` only, in order to see if we could get some improvement in performances of the models.
The procedure we followed and the results we obtained are explained in more detail in section Experiments and evaluation.

3 Models

For this task we have chosen to use three models: Linear SVM, Random Forest and a deep learning approach with a Feedforward Neural Network. We chose Linear SVM to have a direct comparison with the paper from which we started [3], Random Forest to be able to observe the performances of an ensemble method on this task and a Deep Learning approach to be able to observe the behavior of neural networks on this task.

3.1 Linear Support Vector Machine

As a first approach to predict whether a given user u likes a given restaurant r, we chose Linear Support Vector Machine classification method. The Support Vector Machine (SVM) is a supervised learning model, with associated learning algorithms, that analyze data used also for classification.

Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.

An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on the side of the gap on which they fall.

SVM can perform both linear classification and non-linear classification using different kernels that map samples in a different space. In this case, the kernel we decided to use is the linear classification kernel.

3.2 Random Forest

The second approach we used to make our predictions is Random Forest method. Random forests is an ensemble learning method for classification (in this case) that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes of the individual trees.
Each tree is grown using a bootstrap sample (sampling with replacement) of training data by choosing n times from all N available training cases, and uses the rest of the examples to estimate its error.
At each decision node, best attribute to test is chosen from a random sample of m attributes (where $m < M$ and M is the number of features), rather than from all attributes.

3.3 Deep Learning Approach

As a third approach to make our predictions, we choose a deep learning approach through a *feedforward neural network* with multiple hidden layers.
This kind of network is composed of a single input layer, a single output layer but multiple hidden layers. Through matrix calculation and activation function it calculates the network output comparing it with the "ground truth", minimizing the error using gradient descent and performing backpropagation.

Since there are many hidden layers, attention must be paid to the choice of the activation function for the various hidden layers so as not to run into the *vanishing gradient descent* problem:

- **Input layer**: there is no activation in this layer, so it is not necessary to choose any activation function;
- **Hidden layers**: in the various hidden layers there is an activation function but it is not possible to use the sigmoid for its instability, so it is necessary to adopt the ReLu function;
- **Output layer**: in the output layer we used the sigmoid function in order to have the probability that an event or the other occurs, resulting in a probability vector.

4 Experiments and evaluation

Let's describe each test we made for each model.

4.1 Linear Support Vector Machine

4.1.1 Implementation

To implement the linear SVM method we used Scikit Learn's LinearSVC.

The main decision to make on the linear SVM is how to configure the parameters of the model, so we decided to perform a grid search (always using a Scikit Learn module) to find the best configuration of the main parameter, C.

We created an instance of the Linear SVM model to be used in grid search, then performed a grid search training phase to get the best estimator, i.e. the linear SVM instance with the parameter C set to the best value among those proposed among the grid parameters.

For the grid search, only a portion of the training set was used, in order to improve the parallelizability of the procedure, since each parallel instance require its own copy of the dataset: `sub_train_set = train_set[:round(train_set.shape[0]/x)]` where `x` was tested with values `3,2,1`.

The code used for the instance of linear SVM and for grid search is the following:

```
#Linear SVM model
svc_classifier = LinearSVC(random_state = 0, max_iter = 50000)

#Grid Search model
param_grid = {'C':[0.001,0.01,0.1,0.25,0.5,0.75,1,10,100,1000]}

grid = GridSearchCV(estimator = svc_classifier,
                    param_grid = param_grid,
                    refit = True,
                    verbose = 2,
                    cv = 3,
                    error_score = np.nan,
                    n_jobs = -1,
                    pre_dispatch = 6)


grid.fit(sub_train_set.drop(columns=['likes',  'stars_review', 'review_id',
                                     'user_id', 'business_id']),
         sub_train_set['likes'])
```

Listing 1: Linear SVM model

Finally, we trained the obtained estimator on the entire dataset in order to learn about the target label and we performed the prediction on the test set. The relevant code is the following:

```
#Estimator to train and predict the label
best_model = grid.best_estimator_

#Best estimator training
best_model.fit(train_set.drop(columns=['likes', 'stars_review',
                                       'review_id', 'user_id', 'business_id']),
               train_set['likes'])

#Prediction of the target label
predic = best_model.predict(test_set.drop(columns=['likes', 'stars_review',
                                                    'review_id', 'user_id',
                                                    'business_id']))
```

Listing 2: Linear SVM training and predictions

4.1.2 Results

In order to find a good result within a reasonable amount of time, we made several attempts with different settings:

- training on 1/2 of the training set, with 5000 iterations, before applying dimensionality reduction: best training score 0.732, test score 0.701;
- training on 1/3 of the training set, with 10000 iterations, before applying dimensionality reduction: best training score 0.725, test score 0.736;
- training on 1/3 of the training set, with 10000 iterations, threshold for `city` $\theta_1 = 100$, threshold for `categories` $\theta_2 = 200$: best training score 0.722, test score 0.706;
- training on the entire training set, with 50000 iterations, threshold for `city` and `categories` $\theta = 100$ best training score 0.743, test score 0.737;

but we never achieved to reach convergence, neither with the last 30 hours long training.

Our best results are shown in table [5].

	precision	**recall**	**f1-score**	**support**
0	0.70	0.36	0.48	50930
1	0.74	0.92	0.82	103063
macro avg	0.72	0.64	0.65	153993
weighted avg	0.73	0.74	0.71	153993
accuracy	73.727%			

Table 5: Report for SVM model with dimensionality reduction

Even if the results aren't extraordinary good, it has to be noticed that we registered a 4.79% improvement in accuracy with respect to the best score obtained by Gandhe in [3] with the same model, and we also have less overfitting, since the difference between our train and test score is 0.6% while the difference between his scores is 3.37%.
However, the improvement can't be completely ascribed to our richer preprocessing, since our dataset was bigger than the one used by Gandhe, because his paper was referred to the 2014 edition of the Yelp Dataset Challenge, while our work is based on the 2019 edition.

4.2 Random Forest

4.2.1 Implementation

To implement the ensemble random forest method we used Scikit Learn's RandomForestClassifier.

The main decision to take about the random forest is how to configure the parameters of the model, so we decided to perform a grid search (always using Scikit Learn's GridSearchCV) to find the best configuration of the main parameters. To create the parameter grid we were partially inspired by this kernel.

We used two instances of RandomForestClassifier: one to make the grid search on half of the train set and one to train the model on the whole train set and make the predictions.

The code used for the first instance of random forest and for grid search is the following:

```
#First Random Forest model
random_forest = RandomForestClassifier(n_jobs = -1, random_state = 0)

#Grid Search model
param_grid = {'bootstrap': [True, False],
              'max_depth': [10, 30, 50],
              'min_samples_leaf': [1, 2, 4],
              'min_samples_split': [2, 5, 10],
              'n_estimators': [200, 500, 1000],
              'criterion': ['gini', 'entropy']}

grid = GridSearchCV(estimator = random_forest,
                    param_grid = param_grid,
                    refit = False,
                    verbose = 5,
                    cv = 3,
                    error_score = _np.nan,
                    n_jobs = -1,
                    pre_dispatch = 6)

sub_train_set = train_set[:round(train_set.shape[0]/2)]

grid.fit(sub_train_set.drop(columns=['likes', 'stars_review', 'review_id',
                                     'user_id', 'business_id']),
                            sub_train_set['likes'])
```

Listing 3: Random Forest model

Once the grid search is completed, we have a dictionary whose keys are the parameters used in the grid and whose values are the best values found for the corresponding parameters.
So we re-instantiated the random forest classifier by setting its parameters with the best values obtained. Then the model is trained on the whole train set and makes predictions on the whole test set.

The relevant code is as follows:

```
#Second Random Forest instances with the best value of the params
params = grid.best_params_
params['n_jobs'] = -1
params['verbose'] = 5
best_model = RandomForestClassifier(**params)

#Random Forest training
best_model.fit(train_set.drop(columns=['likes',
                                       'stars_review',
                                       'review_id',
                                       'user_id',
                                       'business_id']),
               train_set['likes'])
```

```
#Random Forest prediction
predic = best_model.predict(test_set.drop(columns=['likes', 'stars_review',
                                                   'review_id', 'user_id',
                                                   'business_id']))
```

Listing 4: Random Forest training and predictions

4.2.2 Results

Since there were many parameters to choose, and therefore many tests to be executed by the `GridSearchCV`, we needed all the parallelism achievable with our machine, so we fed the grid search method with only half of the dataset, after observing that we got many memory errors trying with the whole train set.

After more than three days of execution, the best train score we obtained was 0.745, then we trained the best estimator returned by `GridSearchCV` with the whole dataset, and, when used for computing predictions from the test set, it gave us the corresponding test score of 0.741.

The details of those results are shown in table [6].

	precision	**recall**	**f1-score**	**support**
0	0.69	0.40	0.51	50930
1	0.75	0.91	0.83	103063
macro avg	0.72	0.66	0.67	153993
weighted avg	0.73	0.74	0.72	153993
accuracy	74.168%			

Table 6: Report for Random Forest model with dimensionality reduction

From 73.727% to 74.168% the improvement is modest, but not negligible.

4.3 Feedforward Neural Network

4.3.1 Implementation

To implement the neural network we used Tensorflow exploiting Keras' Sequential model and Dense layer, and developed the model based on this article: *Building Neural Network using Keras for Classification*, Renu Khandelwal [6].

We implemented the neural network using the following code:

```
classifier = Sequential()

#First Hidden Layer
classifier.add(Dense(number_hidden_neurons,
                     activation = 'relu',
                     kernel_initializer = 'random_normal',
                     input_dim = number_features))

#Second Hidden Layer
classifier.add(Dense(number_hidden_neurons,
                     activation = 'relu',
                     kernel_initializer = 'random_normal'))

#Third Hidden Layer
classifier.add(Dense(number_hidden_neurons,
                     activation = 'relu',
                     kernel_initializer = 'random_normal'))

#Output Layer
classifier.add(Dense(1,
                     activation = 'sigmoid',
                     kernel_initializer = 'random_normal'))

#Compiling the neural network
classifier.compile(optimizer = 'adam',
                   loss = 'binary_crossentropy',
                   metrics = ['accuracy'])
```

Listing 5: Neural Network model

The first hidden layer is added to the model with the parameter `input_dim=n_features`, that represents the number of neurons per input layer needed (one per feature). The second and third hidden layers are added to the model without the previous parameter.

The main decision to take for each hidden layer is the number of neurons that make it up; we decided to use the following formula (from this thread on StackExchange):

$$N_h = \frac{N_s}{(\alpha * (N_i + N_o))} \tag{2}$$

where:

- N_h is the number of hidden neurons.
- N_i is the number of input neurons.
- N_o is the number of output neurons.
- N_s is the number of train samples.
- α an arbitrary scaling factor, usually between 5 and 10.

In this way, the number of hidden neurons is between the number of neurons in the input layer and the number of neurons in the output layer.

For the output layer we configured only one neuron: the task requires binary classification (yes/no) and therefore we have the probability that it is yes: $P(yes) = 1 - P(no)$.
We could use two neurons in the output layer but it would still represent the same information.

For the training part, Keras was always used, with the following code:

```
#Fitting the data to the training dataset
classifier.fit(train_set.drop(columns = ['likes', 'stars_review',
                                         'review_id', 'user_id',
                                         'business_id']),
               train_set['likes'],
               validation_split = 0.3,
               batch_size = 100,
               epochs = 100)
```

Listing 6: Neural Network training

In training, the model performs 100 iterations with one size for each large batch 100. A part of the training test will be used as a validation test, in order to have a 70 - 30 ratio.

In the prediction part, therefore we use Keras once again, with the following code:

```
prediction = classifier.predict(test_set.drop(columns = ['likes',
                                                         'stars_review',
                                                         'review_id',
                                                         'user_id',
                                                         'business_id']))

#Result binarization
binary_prediction = binarize(prediction, threshold = 0.5)
```

Listing 7: Neural Network predictions

With the `predict` method the target label is predicted resulting in a probability vector formed by one element. We therefore used the Scikit Learns' `binarize` function to transform the probabilistic result into a binary result using a threshold of 0.5: if the result is below or equal to threshold it is replaced with 0 (no), otherwise with 1 (yes).

4.3.2 Results

For this model, we started with the architecture proposed in [6] and a hyperparameter $\alpha = 6$, we saw that decreasing the value of α (also increasing the number of hidden layers) worsened the results, while using $\alpha = 7$ and 5 hidden layers instead of 3 gives slightly better results.

Our first scores were 0.753 in training and 0.742 in testing, in the latter experiment we obtained 0.757 and 0.747, respectively.

The details of those results are shown in table [6].

	precision	**recall**	**f1-score**	**support**
0	0.69	0.40	0.51	50930
1	0.75	0.91	0.83	103063
macro avg	0.72	0.66	0.67	153993
weighted avg	0.73	0.74	0.72	153993
accuracy	74.204%			

Table 7: Report for Neural Network model with dimensionality reduction

Again we obtained a slight improvement from 74.168% to 74.204%.

4.4 Linear SVM without dimensionality reduction

At this point, we decided to train our best SVM model on the whole train set, without dimensionality reduction, i.e., without filtering `city` and `categories` features and applying `OrdinalEncoder` only to the `postal_code` feature, that has too many possible values to apply one hot encoding (the resulting dataset size was about 11GB).

So we just loaded the previously obtained model and trained it with the new dataset:

```
best_model = jl.load("../models/best_SVM.joblib")
best_model.set_params(verbose=10)
best_model.get_params()

""" Out:
    {'C':                      0.001,
    'class_weight':           None,
    'dual':                   True,
    'fit_intercept':          True,
    'intercept_scaling':      1,
    'loss':                   'squared_hinge',
    'max_iter':               50000,
    'multi_class':            'ovr',
    'penalty':                'l2',
    'random_state':           0,
    'tol':                    0.0001,
    'verbose':                10}
"""

best_model.fit(train_set.drop(columns=['likes', 'stars_review', 'review_id',
                                       'user_id', 'business_id']),
               train_set['likes'])
```

Listing 8: SVM without dimensionality reduction

The final test score was 0.737, as shown in table [8].

	precision	recall	f1-score	support
0	0.72	0.34	0.46	50930
1	0.74	0.93	0.83	103063
macro avg	0.73	0.64	0.64	153993
weighted avg	0.73	0.74	0.71	153993
accuracy	73.74%			

Table 8: Report for SVM model without dimensionality reduction

The results we got weren't much better than those obtained with the SVM in [4.1], probably because it would be necessary to pass through a new grid search phase to take full advantage the richer dataset, but we couldn't do it since it would have taken too long, because we didn't have enough RAM to exploit parallelism in `GridSearchCV`.

4.5 Random Forest without dimensionality reduction

We followed a similar criterion for the Random Forest model, so we loaded the previously obtained model and instantiated a new one with the same parameters, customizing the two not set by `GridSearchCV`, i.e., `n_jobs` and `verbose`.

```
params = jl.load("../models/best_Random_Forest_2.joblib").get_params()
params['n_jobs'] = -1
params['verbose'] = 10
best_model = RandomForestClassifier(**params)
best_model.get_params()

""" Out:
    {'bootstrap':                False,
    'class_weight':              None,
    'criterion':                 'entropy',
    'max_depth':                 50,
    'max_features':              'auto',
    'max_leaf_nodes':            None,
    'min_impurity_decrease':     0.0,
    'min_impurity_split':        None,
    'min_samples_leaf':          2,
    'min_samples_split':         10,
    'min_weight_fraction_leaf':  0.0,
    'n_estimators':              1000,
    'n_jobs':                    -1,
    'oob_score':                 False,
    'random_state':              None,
    'verbose':                   10,
    'warm_start':                False}
"""

best_model.fit(train_set.drop(columns=['likes', 'stars_review', 'review_id',
                                       'user_id', 'business_id']),
               train_set['likes'])
```

Listing 9: Random Forest without dimensionality reduction

In this case the result is even worse than the one obtained in [4.2], since we got a test score of 0.739, versus the previous score of 0.741.
More details in table [9].

	precision	**recall**	**f1-score**	**support**
0	0.69	0.38	0.49	50930
1	0.75	0.92	0.82	103063
macro avg	0.72	0.65	0.66	153993
weighted avg	0.73	0.74	0.71	153993
accuracy	73.74%			

Table 9: Report for Random Forest model without dimensionality reduction

The reason of the worsening in the accuracy could be the same as for the previous experiment [4.4], i.e., the absence of the grid search phase (skipped for the same reasons formerly illustrated), that in this case is even more relevant, since there are many hyperparameters to tune.

4.6 Feedforward NN without dimensionality reduction

For the Neural Network, we used a different approach: since there isn't any grid search to do in this case, we manually tested some different configurations and trained completely new models.
These models are similar to the one used in [4.3] except for the dataset used, the number of hidden layers, the number of nodes in each hidden layer (that depends on α in the formula defined in [4.3], that we present again here), and the batch size.

Number of nodes per hidden layer:

$$N_h = \frac{N_s}{\alpha * (N_i + N_o)}$$

In table [10] we summarize all the configurations of the tests we made (see the notebook for more details and code).

#	num. hidden layers	alpha	batch size	train accuracy	test accuracy
1	3	6	100	0.756	0.743
2	3	2	100	0.755	0.738
3	3	7	100	0.753	0.742
4	5	7	100	0.757	0.746
5	5	7	500	0.754	0.74
6	5	6	100	0.76	0.745
7	5	6	500	0.758	0.742

Table 10: Report for Random Forest model without dimensionality reduction

So we achieved to improve the previous results quite significantly in test number 4, for which we present the code in listing [10] and some details in table [11].

```
classifier = Sequential()

classifier.add(Dense(number_hidden_neurons,
                     activation = 'relu',
                     kernel_initializer = 'random_normal',
                     input_dim = number_features))

#Second Hidden Layer
classifier.add(Dense(number_hidden_neurons,
                     activation = 'relu',
                     kernel_initializer = 'random_normal'))

#Third Hidden Layer
classifier.add(Dense(number_hidden_neurons,
                     activation = 'relu',
                     kernel_initializer = 'random_normal'))

#Fourth Hidden Layer
classifier.add(Dense(number_hidden_neurons,
                     activation = 'relu',
                     kernel_initializer = 'random_normal'))

#Fifth Hidden Layer
classifier.add(Dense(number_hidden_neurons,
                     activation = 'relu',
                     kernel_initializer = 'random_normal'))

#Output Layer
classifier.add(Dense(1,
                     activation = 'sigmoid',
                     kernel_initializer = 'random_normal'))
```

```
    # Compiling the neural network
    classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy',
                       metrics = ['accuracy'])

    # Fitting the data to the training dataset
    classifier.fit(train_set.drop(columns = ['likes', 'stars_review',
                                            'review_id', 'user_id',
                                            'business_id']),
                   train_set['likes'], validation_split = 0.3, batch_size = 100,
                   epochs = 100)
```
Listing 10: Neural Network without dimensionality reduction

	precision	**recall**	**f1-score**	**support**
0	0.69	0.43	0.53	50930
1	0.76	0.90	0.83	103063
macro avg	0.72	0.67	0.68	153993
weighted avg	0.74	0.75	0.73	153993
accuracy	74.625%			

Table 11: Report for Neural Network model without dimensionality reduction

4.7 Removing fake reviews

In this step we tried to make predictions using only trustworthy features, to see if fake reviews could bias the models, deviating their predictions.

To achieve this, we just applied the previously obtained best models to a different dataset; this is the only difference with respect to the code used in [4.4], [4.5] and [4.6]:

```
    train_set = train_set[train_set['bin_truth_score']!=-1]
```

Listing 11: Removing fake reviews

The score obtained with SVM is 0.723:

	precision	**recall**	**f1-score**	**support**
0	0.65	0.35	0.45	50930
1	0.74	0.91	0.81	103063
macro avg	0.70	0.63	0.63	153993
weighted avg	0.72	0.73	0.70	153993
accuracy	72.289%			

Table 12: Report for SVM model without fake reviews

The score obtained with Random Forest is 0.741:

	precision	**recall**	**f1-score**	**support**
0	0.68	0.41	0.51	50930
1	0.76	0.90	0.82	103063
macro avg	0.72	0.66	0.67	153993
weighted avg	0.73	0.74	0.72	153993
accuracy	74.149%			

Table 13: Report for Random Forest model without fake reviews

The score obtained with Neural Network is 0.74:

	precision	**recall**	**f1-score**	**support**
0	0.74	0.33	0.46	50930
1	0.74	0.94	0.83	103063
macro avg	0.74	0.64	0.64	153993
weighted avg	0.74	0.74	0.71	153993
accuracy	73.991%			

Table 14: Report for Neural Network model without fake reviews

Contrary to our expectations, the results have worsened rather than improved.
Our hypotheses for explaining this behavior are the following:

1. Also the fake reviews are useful to define a model of the user or a type of review, for example, if a user tends to give fake reviews, in this case the ML algorithms have less examples to understand and predict his/her votes;
2. Despite the model we used to detect fake reviews had very high accuracy on its test set (almost 99%, as shown in section Fake Review Detection), it was trained on a specific dataset, where each review was labeled with `true/false`, that could be too much different from our dataset to produce a significant model for our task.

5 Conclusions

TODO

6 References

1. Yelp dataset on Kaggle https://www.kaggle.com/yelp-dataset/yelp-dataset;
2. Yelp Dataset presentation https://www.yelp.com/dataset;
3. *Restaurant Recommendation System*, Ashish Gandhe
   https://www.semanticscholar.org/paper/Restaurant-Recommendation-System-Gandhe/093cecc3e
   53f2ba4c0c466ad3d8294ba64962050;
4. *Machine Learning and Visualization with Yelp Dataset*, Zhiwei Zhang
   https://medium.com/@zhiwei_zhang/final-blog-642fb9c7e781
   (with her repo https://github.com/zzhang83/Yelp_Sentiment_Analysis);
5. *Recommendation for yelp users itself*, Wenqi Hou, Gauravi Saha, Manying Tsang
   https://www.kaggle.com/wenqihou828/recommendation-for-yelp-users-itself;
6. *Building Neural Network using Keras for Classification*, Renu Khandelwal
   https://medium.com/datadriveninvestor/building-neural-network-using-keras-
   for-classification-3a3656c726c1.