Main

December 15, 2019

1 Prediction of restaurant ratings: Main Notebook

by Team Tissot: Laux David, Randin Samuel, Solonin Maksim, Vivarié Romin.

The business problem we chose to study is the prediction of restaurants ratings. Having notice that reviews being often biased, we wanted to adress a data mining and machine learning problem to analyse the impact of the different variables on restaurant ratings. Our aim is to study the potential correlations of features for a restaurant's rating and the comparison of different technics.

Our first challenge was to select a Data set with enough features and variables to adress the problem. After evaluating various options we chose a Yelp dataset. Yelp gives access to their dataset (https://www.yelp.com/dataset/challenge) for the USA and Canada. The website offers ten awards of $5'000 to the best solutions proposed by students for the following challenges:
- Photo classification
- Natural language processing
- Sentiment Analysis and graph mining

This set includes information about local businesses in 10 metropolitan areas across 2 countries. Round 13 of the challenge was launched on January 15, 2019 and will run through December 31, 2019.

After downloading the data, we had to convert it to a csv file. The cleaning taks are compile in a file dedicated, Cleaning Notebook.
The pre-processing tasks are compiled in a notebook dedicated, the Pre-processing Notebook.

Here is the link to a little video compiling the notebook.

[1]:
```
!pip install imblearn -q
!pip install keras -q
!pip install tensorflow -q
!pip install folium -q
!pip install seaborn -q
```

2 Get the data

Same as in EDA notbook

[2]:
```
import pandas as pd
import folium
import numpy as np
```

```
import matplotlib
import matplotlib.pyplot as plt
from pandas.io.json import json_normalize
import re
import seaborn as sns
sns.set_style('white')
sns.set_color_codes("dark")
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

[3]:
```
df = pd.read_csv("../data/data_clean_new.csv",
                            encoding='utf_8',
                            dtype = 'unicode',
                            parse_dates = True,
                            infer_datetime_format = True,
                            low_memory=False)
df = df.drop("Unnamed: 0", axis = 1)
```

3 Preprocess data

[4]:
```
df.head(3)
```

[4]:
```
                     address            business_id         city is_open  \
0        30 Eglinton Avenue W  QXAEGFB4oINsVuTFxEYKFQ  Mississauga       1
1   10110 Johnston Rd, Ste 15  gnKjwL_1w79qoiV3IC_xQQ    Charlotte       1
2     2450 E Indian School Rd  1Dfx3zM-rW4n-31KeC8sJg      Phoenix       1

        latitude       longitude                        name postal_code  \
0  43.6054989743  -79.652288909   Emerald Chinese Restaurant     L5R 3E7
1      35.092564      -80.859132  Musashi Japanese Restaurant       28210
2     33.4951941    -112.0285876                    Taco Bell       85016

            review_count stars  ... Sushi Bars Tex-Mex Thai Vegan Vegetarian  \
0   0.014979029358897545   2.5  ...          0       0    0     0          0
1   0.020011983223487118   4.0  ...          1       0    0     0          0
2  0.0017974835230677054   3.0  ...          0       1    0     0          0

  Vietnamese Wine & Spirits Wine Bars Anymusic name_length
0          0              0         0    False          26
1          0              0         0    False          27
2          0              0         0    False           9

[3 rows x 78 columns]
```

[5]:
```
df.stars.value_counts()
```

[5]:
```
3.5    12833
4.0    12816
3.0     9421
4.5     5881
2.5     5080
2.0     2771
1.5      936
5.0      931
1.0      217
Name: stars, dtype: int64
```

[6]:
```
#convert columms to the boolean datatype when we find the vlaue "True" or␣
↪"False" in the column
for column in df.columns :
    if df[column][0] == 'True' or df[column][0] =='False':
        df[column] = df[column]=='True'
    #Often we find a True or False in the first line
    #This tests if the column is a boolean by using the first row for efficiency␣
↪

    #otherwise we test if we find a True or False value in the whole column
    elif "True" in df[column].values :
        df[column] = df[column]=='True'
    elif "False" in df[column].values :
        df[column] = df[column]=='True'
```

[7]:
```
cuisine_type = ["American (New)","American (Traditional)","Arts &␣
↪Entertainment","Asian Fusion","Bakeries","Barbeque","Bars",
"Beer","Breakfast & Brunch","Buffets","Burgers","Cafes","Canadian␣
↪(New)","Caribbean","Caterers","Chicken Wings",
"Chinese","Cocktail Bars","Coffee & Tea","Comfort␣
↪Food","Delis","Desserts","Diners","Ethnic Food",
"Event Planning & Services","Fast Food","Food","Food Delivery␣
↪Services","French","Gastropubs","Gluten-Free",
"Greek","Grocery","Halal","Hot Dogs","Ice Cream & Frozen␣
↪Yogurt","Indian","Italian","Japanese","Juice Bars & Smoothies",
"Korean","Latin American","Lounges","Mediterranean","Mexican","Middle␣
↪Eastern","Nightlife","Pizza","Pubs",
"Salad","Sandwiches","Seafood","Soup","Specialty Food","Sports␣
↪Bars","Steakhouses","Sushi Bars","Tex-Mex",
"Thai","Vegan","Vegetarian","Vietnamese","Wine & Spirits","Wine Bars"]
```

[8]:
```
#The cuisine types have 1 or 0 instead of True/False
for column in df[cuisine_type] :
    df[column] = df[column]=="1"
```

[9]:
```
df.columns
```

[9]:
```
Index(['address', 'business_id', 'city', 'is_open', 'latitude', 'longitude',
       'name', 'postal_code', 'review_count', 'stars', 'state', 'Price',
       'American (New)', 'American (Traditional)', 'Arts & Entertainment',
       'Asian Fusion', 'Bakeries', 'Barbeque', 'Bars', 'Beer',
       'Breakfast & Brunch', 'Buffets', 'Burgers', 'Cafes', 'Canadian (New)',
       'Caribbean', 'Caterers', 'Chicken Wings', 'Chinese', 'Cocktail Bars',
       'Coffee & Tea', 'Comfort Food', 'Delis', 'Desserts', 'Diners',
       'Ethnic Food', 'Event Planning & Services', 'Fast Food', 'Food',
       'Food Delivery Services', 'French', 'Gastropubs', 'Gluten-Free',
       'Greek', 'Grocery', 'Halal', 'Hot Dogs', 'Ice Cream & Frozen Yogurt',
       'Indian', 'Italian', 'Japanese', 'Juice Bars & Smoothies', 'Korean',
       'Latin American', 'Lounges', 'Mediterranean', 'Mexican',
       'Middle Eastern', 'Nightlife', 'Pizza', 'Pubs', 'Salad', 'Sandwiches',
       'Seafood', 'Soup', 'Specialty Food', 'Sports Bars', 'Steakhouses',
       'Sushi Bars', 'Tex-Mex', 'Thai', 'Vegan', 'Vegetarian', 'Vietnamese',
       'Wine & Spirits', 'Wine Bars', 'Anymusic', 'name_length'],
      dtype='object')
```

[10]:
```
df['stars']=df['stars'].astype('float')
df.Price = pd.to_numeric(df.Price, errors='coerce')
df = df[np.isfinite(df['Price'])]
#df["Price"]= df["Price"].astype(int)

df["review_count"]= df["review_count"].astype(float)
df["name_length"]= df["name_length"].astype(int)
df["name"]= df["name"].astype(str)
df["address"]= df["address"].astype(str)

df['latitude'] = df['latitude'].astype(float)
df['longitude'] = df['longitude'].astype(float)
```

[11]:
```
df.review_count.dtypes
```

[11]:
```
dtype('float64')
```

4 EDA

4.0.1 Heat map

Let's start by visualizing where the restaurants in our dataset are located

[12]:
```
from folium import plugins
from folium.plugins import HeatMap
```

```
# Make an empty map
m = folium.Map(location=[28,-90], tiles="OpenStreetMap", zoom_start=4)

# Filter the DF for rows, then columns, then remove NaNs
heat_df = df[['latitude', 'longitude']]
heat_df = heat_df.dropna(axis=0, subset=['latitude','longitude'])

# List comprehension to make out list of lists
heat_data = [[row['latitude'],row['longitude']] for index, row in heat_df.
 ↪iterrows()]

# Plot it on the map
HeatMap(heat_data).add_to(m)

# show the map
m
```

[12]:
```
<folium.folium.Map at 0x1da6b03f188>
```

We can see that our data is only restaurants from North America.

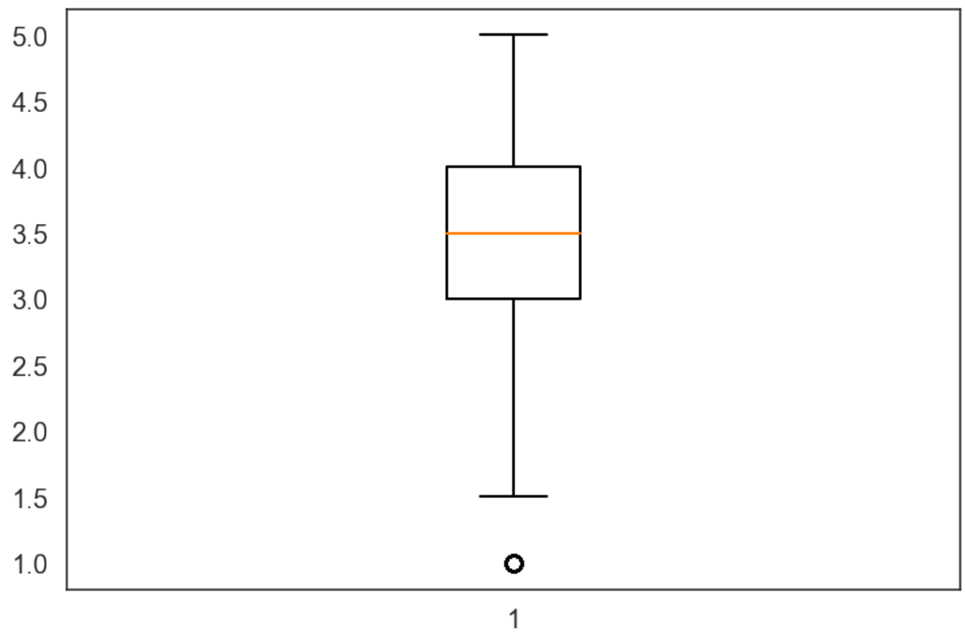
4.0.2 Stars on yelp

This will be the value that we want to predict, let's take a look.

[13]:
```
df.stars.describe()
```

[13]:
```
count    50839.000000
mean         3.447717
std          0.765024
min          1.000000
25%          3.000000
50%          3.500000
75%          4.000000
max          5.000000
Name: stars, dtype: float64
```

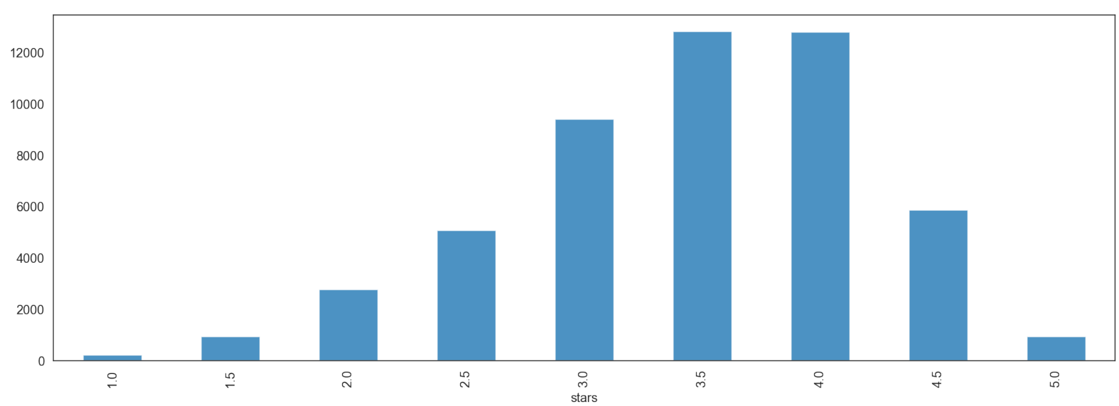
[14]:
```
box_plot_data = df['stars'].astype("float")
plt.boxplot(box_plot_data)
plt.show()
```



We can see that we have an outlier, the 1 star ratings.

[15]:
```
r1 = df[["business_id","stars"]].groupby(["stars"]).count()
r1.plot.bar(x=None,
            y=None,
            figsize=(15,5),
            alpha = 0.8, # make the plot 20% transparent
            legend = None,
           )
```

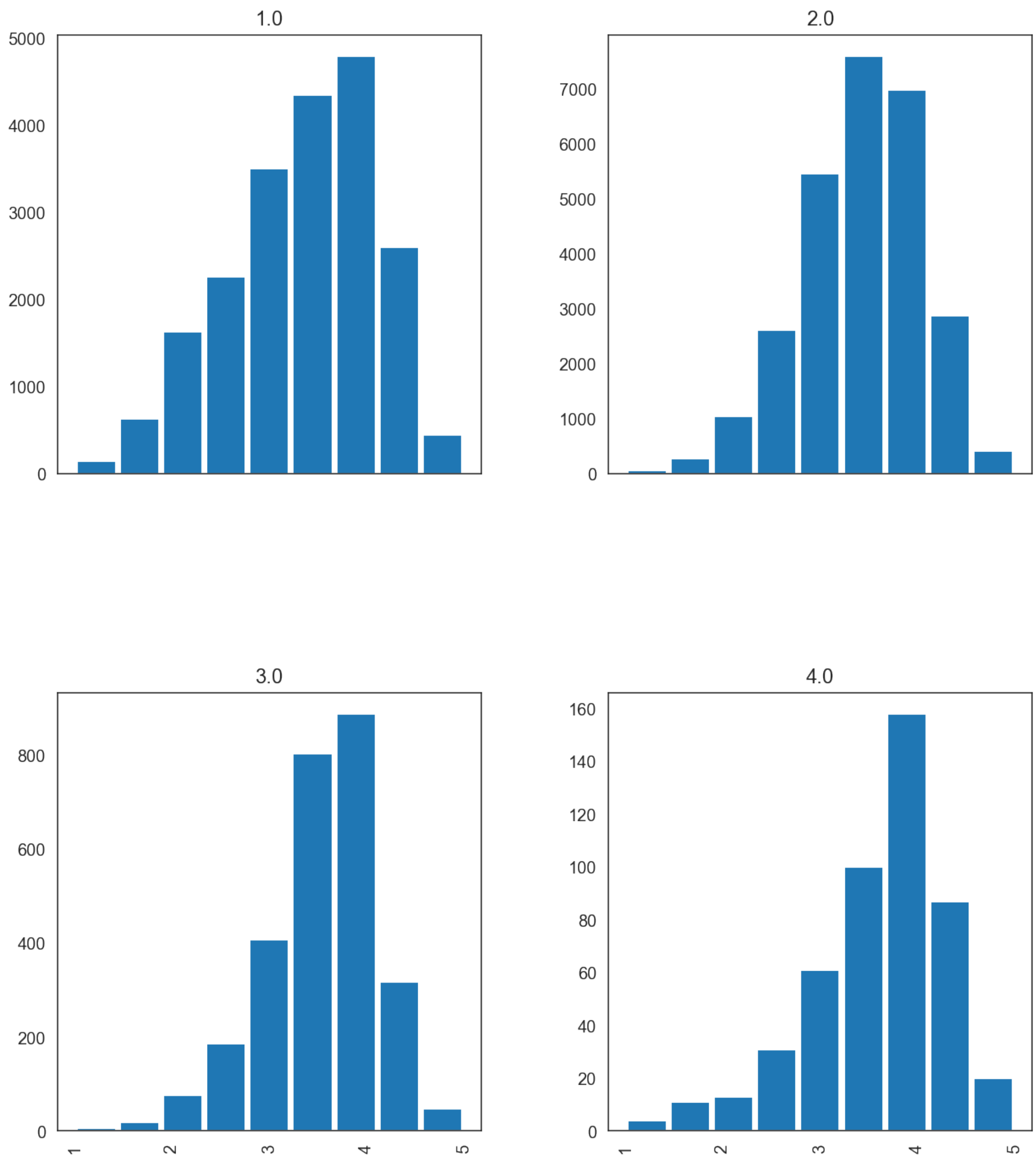
[15]: <matplotlib.axes._subplots.AxesSubplot at 0x1da72c3a388>



4.0.3 Price

We think that this will be an important feature so let's get som insights

[16]:
```
ax = df.hist(column="stars", by='Price',bins=9, grid=False, figsize=(10,12),␣
↪layout=(2,2), sharex=True, zorder=2, rwidth=0.9)
```

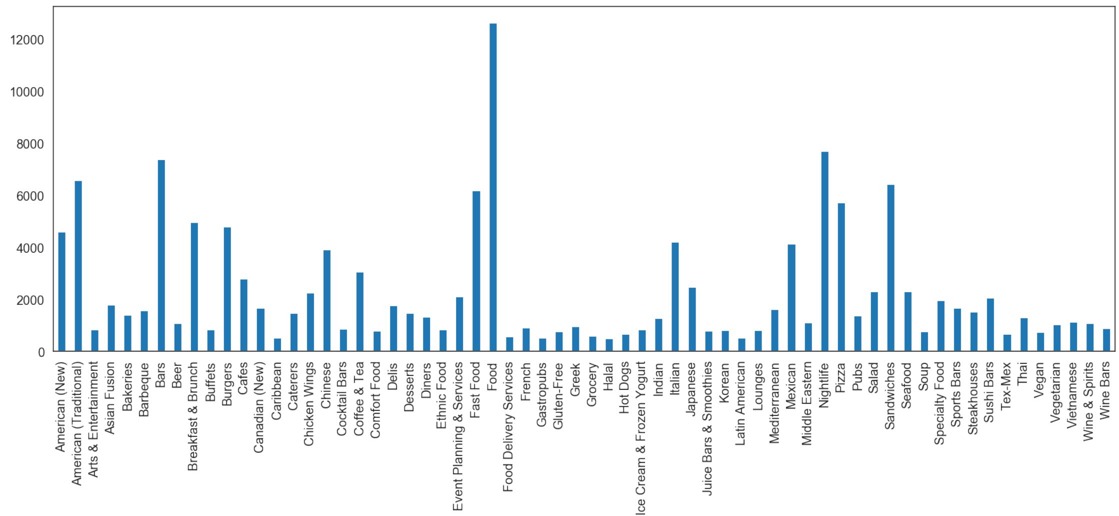


On this graph we ploted the restaurants price classes (from 1 to 4) by stars. We can see the different distributions of ratings depending on the price category.

4.0.4 Cuisine types

[17]:
```
r2 = df[cuisine_type].sum()
r2.plot.bar(x=None, y=None, figsize = (15, 5))
```

[17]: `<matplotlib.axes._subplots.AxesSubplot at 0x1da72e259c8>`



Since we have a lot of cuisine type we will analyze the top 20 We also remove the first 3 cuisine types (Food, Bars and Nightlife) since we don't

[18]:
```
top20_cuisines = list(df[cuisine_type].sum().sort_values(ascending=False).
  ↪index[0:19])
top20_cuisines
```

[18]:
```
['Food',
 'Nightlife',
 'Bars',
 'American (Traditional)',
 'Sandwiches',
 'Fast Food',
 'Pizza',
 'Breakfast & Brunch',
 'Burgers',
 'American (New)',
 'Italian',
 'Mexican',
 'Chinese',
 'Coffee & Tea',
 'Cafes',
 'Japanese',
```

```
'Seafood',
'Salad',
'Chicken Wings']
```

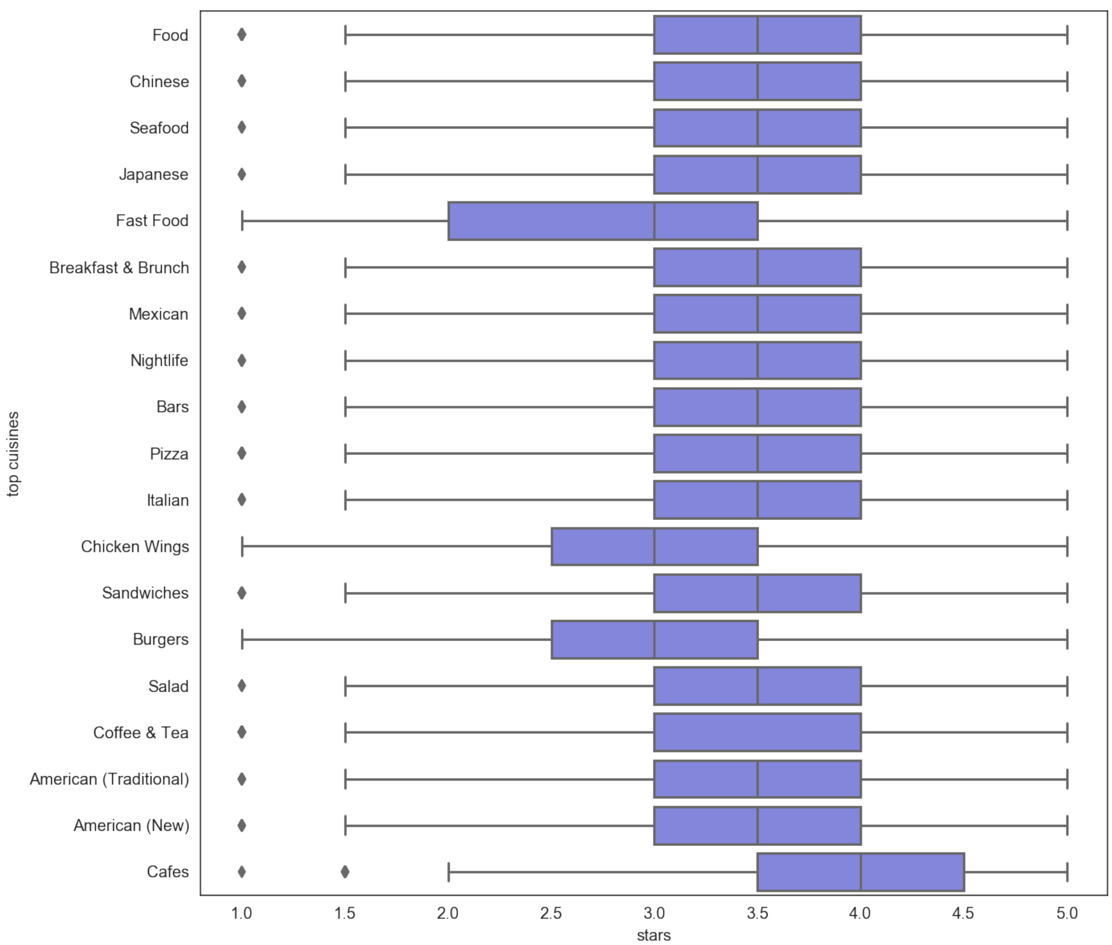
[19]:
```
df2 = df.copy() #create new DataFrame
st = df2.loc[:, top20_cuisines].stack()

all_ids = pd.Series(st.index.get_level_values(1),
                    st.index.get_level_values(0),
                    name='top cuisines')[st.values]

df2 = df2.join(all_ids, how='left').dropna()
```

[20]:
```
plt.figure(figsize = (10, 10))
sns.boxplot(data = df2[["top cuisines", "stars"]], x= "stars", y="top cuisines",␣
↪color=matplotlib.colors.to_hex('#7479e8'))
```

[20]: <matplotlib.axes._subplots.AxesSubplot at 0x1da72e0b8c8>



We can see that Fast Food, Chicken Wings and Burgers have a lower star ratings median. Cafes have have an above average median.

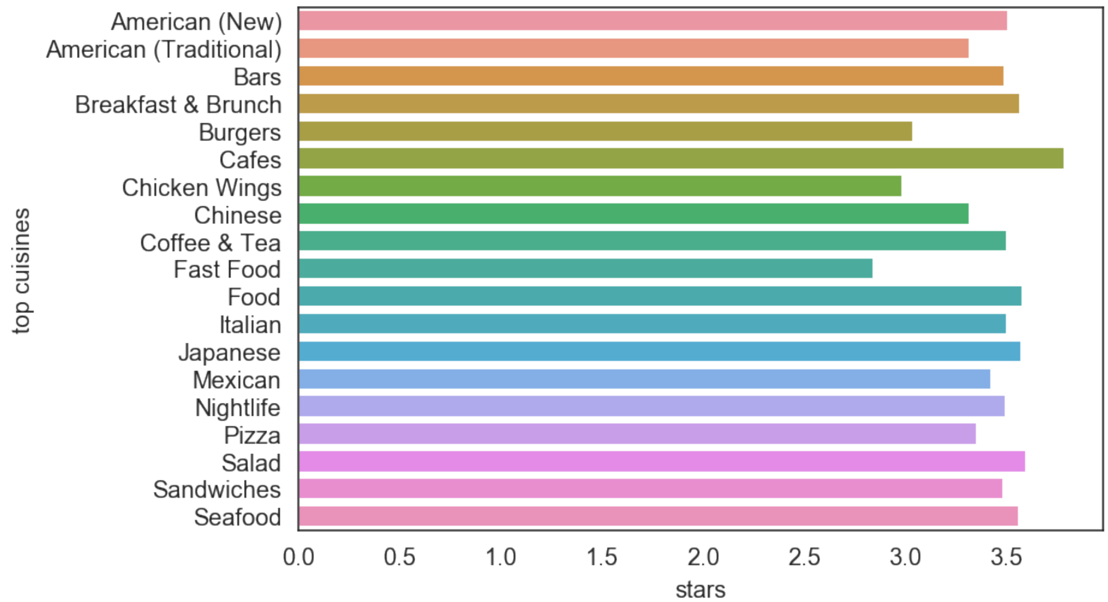
[21]:
```
df2.stars = df2.stars.astype(float)
avg_ratings_cuisine = pd.DataFrame(df2.groupby("top cuisines")["stars"].mean())
avg_ratings_cuisine
```

[21]:

	stars
top cuisines	
American (New)	3.510193
American (Traditional)	3.321602
Bars	3.495263
Breakfast & Brunch	3.572063
Burgers	3.038101
Cafes	3.790196
Chicken Wings	2.987556
Chinese	3.318252
Coffee & Tea	3.506048
Fast Food	2.846645
Food	3.582436
Italian	3.507121
Japanese	3.576039
Mexican	3.429625
Nightlife	3.495586
Pizza	3.356643
Salad	3.600738
Sandwiches	3.484052
Seafood	3.565415

[22]:
```
sns.barplot(data= avg_ratings_cuisine, x = "stars", y= avg_ratings_cuisine.index␣
↪)
```

[22]: <matplotlib.axes._subplots.AxesSubplot at 0x1da75783f88>



4.0.5 Base rate

The base rate is the size of the most common class divided by the size of the dataset. Our accuracy should be better than the default rate

[23]:
```
df.stars.dtypes
```

[23]:
```
dtype('float64')
```

[24]:
```
print("The most common class for the ratings is", df["stars"].mode()[0])

baseRate = df[df["stars"] == 3.5].count()["stars"] / df["stars"].count()
print("The baserate is :", baseRate)
```

```
The most common class for the ratings is 3.5
The baserate is : 0.25228663034284704
```

5 Preprocess data for models

The outlier we identified earlier, the 1 star rating, does not contain enough restaurants (only 216), thus we decided to drop it. We also drop string columns, which will not help in prediction (They are mainly related to geography).

[25]:
```
from sklearn import preprocessing
from sklearn import utils
```

[26]:
```
df.columns
```

[26]:
```
Index(['address', 'business_id', 'city', 'is_open', 'latitude', 'longitude',
       'name', 'postal_code', 'review_count', 'stars', 'state', 'Price',
       'American (New)', 'American (Traditional)', 'Arts & Entertainment',
       'Asian Fusion', 'Bakeries', 'Barbeque', 'Bars', 'Beer',
       'Breakfast & Brunch', 'Buffets', 'Burgers', 'Cafes', 'Canadian (New)',
       'Caribbean', 'Caterers', 'Chicken Wings', 'Chinese', 'Cocktail Bars',
       'Coffee & Tea', 'Comfort Food', 'Delis', 'Desserts', 'Diners',
       'Ethnic Food', 'Event Planning & Services', 'Fast Food', 'Food',
       'Food Delivery Services', 'French', 'Gastropubs', 'Gluten-Free',
       'Greek', 'Grocery', 'Halal', 'Hot Dogs', 'Ice Cream & Frozen Yogurt',
       'Indian', 'Italian', 'Japanese', 'Juice Bars & Smoothies', 'Korean',
       'Latin American', 'Lounges', 'Mediterranean', 'Mexican',
       'Middle Eastern', 'Nightlife', 'Pizza', 'Pubs', 'Salad', 'Sandwiches',
       'Seafood', 'Soup', 'Specialty Food', 'Sports Bars', 'Steakhouses',
       'Sushi Bars', 'Tex-Mex', 'Thai', 'Vegan', 'Vegetarian', 'Vietnamese',
       'Wine & Spirits', 'Wine Bars', 'Anymusic', 'name_length'],
      dtype='object')
```

[27]:
```
X = df[df['stars'] != 1.0]
X = X.drop(['stars', "name", "address", "business_id", "city", "state",␣
 ↪'postal_code', 'latitude', 'longitude'], axis = 1)
y = df[df["stars"] != 1.0]["stars"]
```

[28]:
```
y.value_counts()
```

[28]:
```
3.5    12826
4.0    12809
3.0     9414
4.5     5875
2.5     5075
2.0     2762
1.5      933
5.0      929
Name: stars, dtype: int64
```

[29]:
```
#SMOTE does not handle categorical data, we could also use SMOTE-NC
```

[30]:
```
X.Anymusic = X.Anymusic.astype(bool)
X.is_open = X.is_open.astype(bool)
```

[31]:
```
lab_enc = preprocessing.LabelEncoder()
encoded_y = lab_enc.fit_transform(y) #we label encode the star ratings
#X.Price = lab_enc.fit_transform(X.Price)
```

We divide our data into train and test.

[32]:
```
from sklearn.model_selection import train_test_split, GridSearchCV
from pprint import pprint
from time import time
```

[77]:
```
# split train/test
X_train, X_test, y_train, y_test = train_test_split(X, encoded_y, test_size=0.2,␣
↪random_state=72)
```

Now, we decided to try two methods to fight severe class imbalance. Just downsampling is not an option (then each class will have 900 observations), thus, first we tried upsampling our train data and the we wanted to try to combine the two methods by upsampling all the classes below the mean and downsampling the classes above it.

For upsampling we used a technique called SMOTE (**S**ynthetic **M**inority **O**ver-sampling **TE**chnique) that will synthesize new minority instances. So in our case we will basically generate "fake" restaurants based on other on the data set we have, for more details check out : http://rikunert.com/SMOTE_explained .

For downsampling we used the NearMiss methode

"first, the method calculates the distances between all instances of the majority class and the instances of the minority class. Then k instances of the majority class that have the smallest distances to those in the minority class are selected. If there are n instances in the minority class, the "nearest" method will result in k*n instances of the majority class."

source : https://towardsdatascience.com/sampling-techniques-for-extremely-imbalanced-data-part-i-under-sampling-a8dbc3d8d6d8

[34]:
```
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import NearMiss
```

```
Using TensorFlow backend.
```

[35]:
```
unique, counts = np.unique(encoded_y, return_counts=True)
dict(zip(unique, counts))
```

[35]:
```
{0: 933, 1: 2762, 2: 5075, 3: 9414, 4: 12826, 5: 12809, 6: 5875, 7: 929}
```

[36]:
```
#upsamples all the train data
X_resampled, y_resampled = SMOTE(random_state = 72).fit_resample(X_train,␣
↪y_train)
```

[37]:
```
med_cl_cnt= int(pd.Series(y_train).value_counts().median()) # median of obs over␣
↪classes
med_cl_cnt
```

[37]:
```
4379
```

[38]:
```
unique, counts = np.unique(y_train, return_counts=True)
dict(zip(unique, counts))
```

[38]: {0: 736, 1: 2202, 2: 4056, 3: 7539, 4: 10268, 5: 10266, 6: 4702, 7: 729}

[39]:
```
#downsamples and upsamples the train data (based on median)
X_resampled2, y_resampled2 = NearMiss(sampling_strategy = {3:med_cl_cnt, 4:
 ↪med_cl_cnt, 5:med_cl_cnt, 6:med_cl_cnt}).fit_resample(X_train, y_train)␣
 ↪#downsampling
X_resampled2, y_resampled2 = SMOTE(random_state = 72, sampling_strategy = {7:
 ↪med_cl_cnt, 0:med_cl_cnt, 1:med_cl_cnt, 2:med_cl_cnt}).
 ↪fit_resample(X_resampled2, y_resampled2) # upsampling
```

[40]:
```
unique, counts = np.unique(y_resampled2, return_counts=True)
dict(zip(unique, counts))
```

[40]: {0: 4379, 1: 4379, 2: 4379, 3: 4379, 4: 4379, 5: 4379, 6: 4379, 7: 4379}

Specify parameters values for grid search:

[41]:
```
parametersRF = {
    'n_estimators': (100,200,300),
    'max_depth': (10,20,30)
}
parametersLR = {
    'C': (0.1, 1,100),
    'solver': (['saga','lbfgs'])
}
parametersNN = {
    'epochs': ([10, 100]),
    'batch_size': ([20,30])
}
```

6 Logistic Regression

[42]:
```
from sklearn.linear_model import LogisticRegressionCV, LogisticRegression
```

We tried three different samples of data: normal, upsampled, upsampled+downsampled

We found that for logistic regression the best result is produced with the normal data

[78]:
```
# decomment, if need to run best model with other params
LR = LogisticRegression(solver='lbfgs',C=100, max_iter=2000, multi_class =␣
 ↪"auto")
LR.fit(X_train, y_train)
```

```
C:\Games\Python\lib\site-packages\sklearn\linear_model\_logistic.py:939:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html.
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

[78]:

```
LogisticRegression(C=100, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=2000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

[44]:

```python
if __name__ == "__main__":
    # multiprocessing requires the fork to happen in a __main__ protected
    # block

    # find the best parameters for both the feature extraction and the
    # classifier
    grid_search = GridSearchCV(LR, parametersLR, cv=2,
                               n_jobs=-1, verbose=1,scoring='accuracy')

    print("Performing grid search...")
    #print("pipeline:", [name for name, _ in pipeline2.steps])
    print("parameters:")
    pprint(parametersLR)
    t0 = time()
    grid_search.fit(X_train, y_train)
    print("done in %0.3fs" % (time() - t0))
    print()

    print("Best score for Logistic Regression: %0.3f" % grid_search.best_score_)
    print("Best parameters set:")
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parametersLR.keys()):
        print("\t%s: %r" % (param_name, best_parameters[param_name]))
```

```
Performing grid search...
parameters:
{'C': (0.1, 1, 100), 'solver': ['saga', 'lbfgs']}
Fitting 2 folds for each of 6 candidates, totalling 12 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 out of  12 | elapsed:  1.8min remaining:   22.1s
```

```
[Parallel(n_jobs=-1)]: Done  12 out of  12 | elapsed:  1.9min finished
```

```
done in 187.606s

Best score for Logistic Regression: 0.301
Best parameters set:
        C: 100
        solver: 'lbfgs'
```

```
C:\Games\Python\lib\site-packages\sklearn\linear_model\_logistic.py:939:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html.
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

The best model according to gridsearch is the one we tested at the beginning with lbfgs solver and C equal to 100.

[79]:
```
# train accuracy
LR.score(X_train, y_train)
```

[79]: 0.30994123166576126

[80]:
```
# test accuracy
LR.score(X_test, y_test)
```

[80]: 0.3008395061728395

[81]:
```
from sklearn.metrics import classification_report
target_names = ["1.5","2","2.5","3","3.5","4","4.5","5"]

print(classification_report(y_test, LR.predict(X_test), target_names=
 ↪target_names))
```

```
              precision    recall  f1-score   support

         1.5       0.00      0.00      0.00       197
           2       0.27      0.21      0.23       560
         2.5       0.20      0.04      0.07      1019
           3       0.27      0.15      0.19      1875
         3.5       0.29      0.46      0.36      2558
           4       0.32      0.54      0.40      2543
         4.5       0.42      0.05      0.09      1173
           5       0.00      0.00      0.00       200
```

```
    accuracy                           0.30     10125
   macro avg       0.22      0.18      0.17     10125
weighted avg       0.29      0.30      0.26     10125

C:\Games\Python\lib\site-packages\sklearn\metrics\_classification.py:1268:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Our test accuracy is above the baserate but it isn't really a good result since we cannot predict 1.5 and 5 stars.

7 Random Forest Classifier

Here we will try both data samples, but still we believe that sticking to one method of upsampling is better (at least more common).

[48]:
```
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import make_scorer
MAE = make_scorer(mean_absolute_error)
folds = 3
```

[49]:
```
clf = RandomForestClassifier(n_estimators = 200,max_depth = 30)
clf.fit(X_resampled, y_resampled)
```

[49]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=30, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=200,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[50]:
```
if __name__ == "__main__": # just for multiprocessing purposes
    grid_search = GridSearchCV(clf, parametersRF, cv=3,
                               n_jobs=-1, verbose=1,scoring='accuracy')

    print("Performing grid search...")
    #print("Random Forest:", [name for name, _ in clf.steps])
    print("parameters:")
    pprint(parametersRF)
    t0 = time()
```

```
    grid_search.fit(X_resampled2, y_resampled2)
    print("done in %0.3fs" % (time() - t0))
    print()

    print("Best score for Random Forest: %0.3f" % grid_search.best_score_)
    print("Best parameters set:")
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parametersRF.keys()):
        print("\t%s: %r" % (param_name, best_parameters[param_name]))
```

```
Performing grid search...
parameters:
{'max_depth': (10, 20, 30), 'n_estimators': (100, 200, 300)}
Fitting 3 folds for each of 9 candidates, totalling 27 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  27 out of  27 | elapsed:  1.4min finished
```

```
done in 93.464s

Best score for Random Forest: 0.417
Best parameters set:
        max_depth: 30
        n_estimators: 200
```

[51]:
```
clf = RandomForestClassifier(n_estimators = 300,max_depth = 30)
clf.fit(X_resampled, y_resampled)
```

[51]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=30, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=300,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[82]:
```
clf2 = RandomForestClassifier(n_estimators = 200,max_depth = 30)
clf2.fit(X_resampled2, y_resampled2)
```

[82]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=30, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=200,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[53]:
```
print("Train accuracy :", clf.score(X_resampled, y_resampled))
print("Test accuracy :", clf.score(X_test, y_test))
```

```
Train accuracy : 0.8877824308531359
Test accuracy : 0.268641975308642
```

[83]:
```
print("Train acc upsample + downsample :", clf2.score(X_resampled2,␣
↪y_resampled2))
print("Test acc upsample + downsample :", clf2.score(X_test, y_test))
```

```
Train acc upsample + downsample : 0.8382336149805891
Test acc upsample + downsample : 0.21955555555555556
```

Report for upsampling:

[55]:
```
print(classification_report(y_test, clf.predict(X_test), target_names=␣
↪target_names))
```

```
              precision    recall  f1-score   support

         1.5       0.15      0.31      0.20       197
           2       0.18      0.23      0.20       560
         2.5       0.19      0.19      0.19      1019
           3       0.25      0.24      0.24      1875
         3.5       0.32      0.30      0.31      2558
           4       0.36      0.31      0.34      2543
         4.5       0.25      0.25      0.25      1173
           5       0.09      0.21      0.13       200

    accuracy                           0.27     10125
   macro avg       0.22      0.26      0.23     10125
weighted avg       0.28      0.27      0.27     10125
```

Report for upsampling + downsampling:

[84]:
```
print(classification_report(y_test, clf2.predict(X_test), target_names=␣
↪target_names))
```

```
              precision    recall  f1-score   support

         1.5       0.14      0.29      0.19       197
           2       0.17      0.24      0.20       560
         2.5       0.16      0.25      0.19      1019
           3       0.25      0.17      0.20      1875
         3.5       0.30      0.20      0.24      2558
           4       0.31      0.18      0.23      2543
         4.5       0.20      0.40      0.26      1173
           5       0.07      0.18      0.10       200
```

```
    accuracy                           0.22     10125
   macro avg       0.20      0.24      0.20     10125
weighted avg       0.25      0.22      0.22     10125
```

For random forest upsampling data produces the best result

At the end, we would like to use mean absolute error as a performance metric for our models, because predicting rating of the restaurant is a classification with ordinal variable. Thus, misclassification of 0.5 star is better than 1.5 stars.

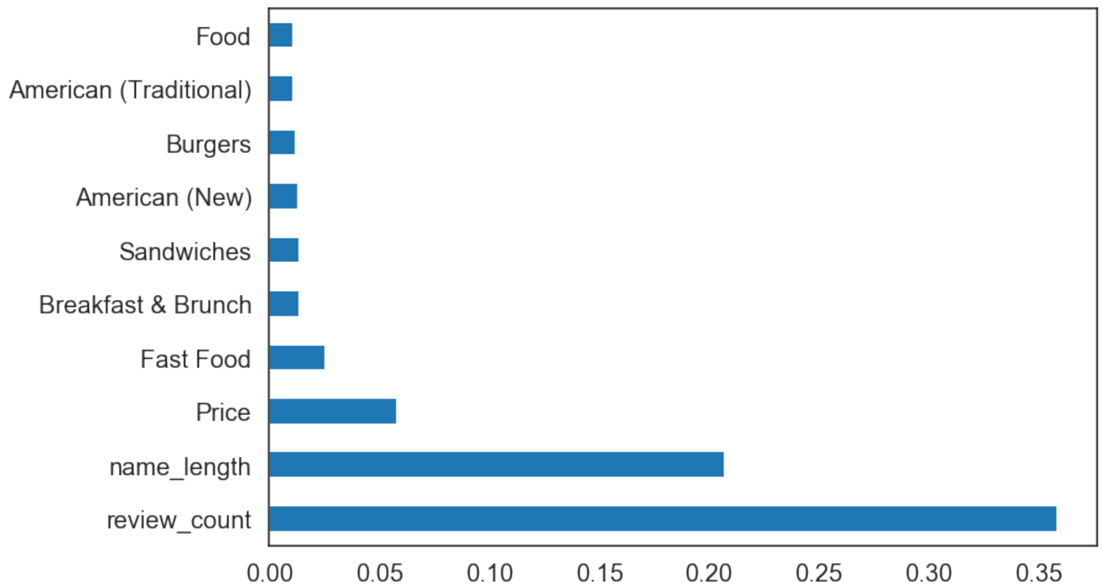
[57]:
```
MAE_RF = cross_val_score(clf,
    X_resampled,
    y_resampled,
    cv=folds,
    scoring=MAE)
```

[58]:
```
print("Random forest MAE :", np.mean(MAE_RF))
```

```
Random forest MAE : 0.8111601591532693
```

[59]:
```
model = ExtraTreesClassifier(n_estimators=100)
model.fit(X_resampled, y_resampled)
print(model.feature_importances_) #use inbuilt class feature_importances of tree␣
 ↪based classifiers
#plot graph of feature importances for better visualization
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
feat_importances.nlargest(10).plot(kind='barh')
plt.show()
```

```
[0.         0.35854874 0.05820922 0.01316088 0.01091862 0.0049285
 0.00664631 0.00464137 0.00658848 0.00357923 0.00213516 0.01385561
 0.00377376 0.01237477 0.00754995 0.0073732  0.00296826 0.0036767
 0.00810324 0.00549191 0.00230566 0.00674999 0.00484526 0.00640751
 0.00522001 0.00571557 0.00191725 0.00628447 0.0257023  0.01074152
 0.00302472 0.00399576 0.00271455 0.00454194 0.00465785 0.00269398
 0.00333989 0.00410748 0.00361991 0.00517233 0.01034663 0.0054067
 0.0033741  0.00359999 0.00325479 0.00295507 0.00520769 0.00880116
 0.00439719 0.00417155 0.00878083 0.00341773 0.00921033 0.01364807
 0.00846284 0.00396577 0.00558118 0.00268482 0.00555727 0.00463292
 0.00238094 0.00406075 0.00368334 0.00426701 0.0043957  0.00207615
 0.00206597 0.00387306 0.20745856]
```



8 Neural network

Here we are implementing neural network with basic architecture on not sampled data.

[60]:
```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras import optimizers
from keras.utils import np_utils
from keras.wrappers.scikit_learn import KerasClassifier
np.random.seed(1143)
```

[61]:
```
def model_NN():
    model = Sequential()
    model.add(Dense(512, input_shape=(69,)))
    model.add(Activation('relu')) # An "activation" is just a non-linear␣
 ↪function applied to the output
                              # of the layer above. Here, with a "rectified␣
 ↪linear unit"
                              # we clamp all values below 0 to 0.

    model.add(Dropout(0.2))   # Dropout helps protect the model from memorizing␣
 ↪or "overfitting" the training data
    model.add(Dense(8))
    model.add(Activation('softmax')) # This special "softmax" activation among␣
 ↪other things,
```

```
                                    # ensures the output is a valid probaility␣
↪distribution, that is
                                    # that its values are all non-negative and sum␣
↪to 1.
    #optimizer = optimizers.Adam(lr=0.01, decay=1e-6)
    optimizer = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer,␣
↪metrics=['accuracy'])
    return model
```

[62]:
```
model = KerasClassifier(build_fn= model_NN, epochs=100, batch_size=10, verbose=0)
```

[63]:
```
X = df[df["stars"] != 1]
X = X.drop(['stars', "name", "address", "business_id", "city", "state",␣
↪'postal_code', "longitude", "latitude"], axis = 1)
y = df[df["stars"] !=1]["stars"]
```

[64]:
```
#nb_classes = len(y.value_counts())

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=72)
```

[65]:
```
from keras.utils import to_categorical

y_train = to_categorical(y_train, num_classes=8)
y_test = to_categorical(y_test, num_classes=8)
```

[70]:
```
model_hist = model.fit(X_train, y_train,
                       batch_size=64, epochs=10,
                       verbose=1, validation_split=0.2)
```

```
Train on 32398 samples, validate on 8100 samples
Epoch 1/10
32398/32398 [==============================] - 2s 47us/step - loss: 1.1734 -
accuracy: 0.4488 - val_loss: 1.1633 - val_accuracy: 0.4860
Epoch 2/10
32398/32398 [==============================] - 1s 44us/step - loss: 1.1249 -
accuracy: 0.4643 - val_loss: 1.1618 - val_accuracy: 0.4377
Epoch 3/10
32398/32398 [==============================] - 2s 50us/step - loss: 1.1127 -
accuracy: 0.4752 - val_loss: 1.1111 - val_accuracy: 0.4763
Epoch 4/10
32398/32398 [==============================] - 2s 55us/step - loss: 1.1025 -
accuracy: 0.4787 - val_loss: 1.1367 - val_accuracy: 0.4822
Epoch 5/10
32398/32398 [==============================] - 2s 55us/step - loss: 1.1005 -
accuracy: 0.4810 - val_loss: 1.0953 - val_accuracy: 0.4878
```

```
Epoch 6/10
32398/32398 [==============================] - 2s 54us/step - loss: 1.0978 -
accuracy: 0.4828 - val_loss: 1.0910 - val_accuracy: 0.4978
Epoch 7/10
32398/32398 [==============================] - 2s 54us/step - loss: 1.0949 -
accuracy: 0.4833 - val_loss: 1.1093 - val_accuracy: 0.4795
Epoch 8/10
32398/32398 [==============================] - 2s 55us/step - loss: 1.0924 -
accuracy: 0.4850 - val_loss: 1.0964 - val_accuracy: 0.4936
Epoch 9/10
32398/32398 [==============================] - 2s 55us/step - loss: 1.0907 -
accuracy: 0.4857 - val_loss: 1.1086 - val_accuracy: 0.4753
Epoch 10/10
32398/32398 [==============================] - 2s 54us/step - loss: 1.0903 -
accuracy: 0.4849 - val_loss: 1.1169 - val_accuracy: 0.4627
```

[71]:
```
score = model.score(X_test, y_test, verbose=0)
score
```

[71]:
```
0.45550617575645447
```

8.1 Conclusion

We conducted good EDA for YELP restaurant data and tested three different models to predict the star rating.

The best model in terms of stars prediction turned out to be neural network with test accuracy equal to 0.45.

Thus, we suggest our friends use this model to understand, how their current restaurant performance will behave in America and estimate their rating among clients.