Linear Regression – Practical Assignment, Final Report:

Shahar oded, Nir Rahav

**See attached in the Moodle the complete code book

Part 1: Creation of Model's Classes:

As attached in the code book, we created 3 classes for 3 different models: Linear Regression, Logistic Regression and PCA.

Attached are examples of return values from the 3 custom models' proving that in 2 of them, their output is identical to the output of the same models from SK-Learn library:

Linear Regression:

```
Custom Model Coefficients: [ 0.00071039  0.05368674 -0.00210311]
Custom Model Intercept: 0.4788891477173974
Scikit-learn Model Coefficients: [ 0.00071039  0.05368674 -0.00210311]
Scikit-learn Model Intercept: 0.4788891477173981
Custom Model Predictions: [0.61044385 0.70787191 0.58629068 ... 0.58208446 0.55431926 0.75019245]
Scikit-learn Model Predictions: [0.61044385 0.70787191 0.58629068 ... 0.58208446 0.55431926 0.75019245]
```

PCA:

Please note that we implemented this class to automatically choose the ideal number of components based on the cumulative variance (can be set by the user).

```
Recommended number of components for total variance of at least 0.9 is 2, init function has been updated
Custom Model Components:
 [[ 0.99980613 -0.01338623 -0.01443995]
 [ 0.01448584  0.00333775  0.9998895 ]]
Scikit-learn Model Components:
 [[ 0.99980613 -0.01338623 -0.01443995]
 [ 0.01448584  0.00333775  0.9998895 ]]
Custom Model Transformed Data:
 [[ 14.47017239  -1.68463144]
 [  0.44611411  -1.88075777]
 [-19.52323601  -2.17715016]
 ...
 [-19.55211592  -0.17737115]
 [-16.29610457 116.89548101]
 [-15.5641702   -2.10919353]]
Scikit-learn Model Transformed Data:
 [[ 14.47017239  -1.68463144]
 [  0.44611411  -1.88075777]
 [-19.52323601  -2.17715016]
 ...
 [-19.55211592  -0.17737115]
 [-16.29610457 116.89548101]
 [-15.5641702   -2.10919353]]
```

We noticed that SK implementation only center the data points and not dividing it by STD. Out of belief in their experience better than ours, we chose to implement similarly, meaning that we are not fully standardizing the data (only centering it), and the COV-VAR matrix is calculated by dividing each result by N-1 and not N.

Logistic Regression:

```
Custom Model Coefficients: [-1138.01550562   357.03934068   638.23687639   -33.48384465]
Custom Model Intercept: -1138.0155056165743
Scikit-learn Model Coefficients: [[ 0.00644416  0.58471554 -0.01388088]]
Scikit-learn Model Intercept: [-1.15330221]
Custom Model Predictions: [1. 1. 1. ... 0. 1. 1.]
Scikit-learn Model Predictions: [1 1 1 ... 0 1 1]
Custom Model Accuracy: 0.7300295857988166
Scikit-learn Model Accuracy: 0.7662721893491125
<ipython-input-215-a9e44a697c86>:29: RuntimeWarning:
```

The difference in the Logistic Regression model might happen because:

Optimization Algorithm: The custom implementation uses gradient descent to optimize the coefficients and intercept, while scikit-learn's LogisticRegression uses various optimization .algorithms like liblinear, lbfgs, sag, or saga.

Regularization: The custom implementation does not include any regularization term, while scikit-learn's LogisticRegression allows you to choose different regularization types (L1 or L2) and specify the regularization strength.

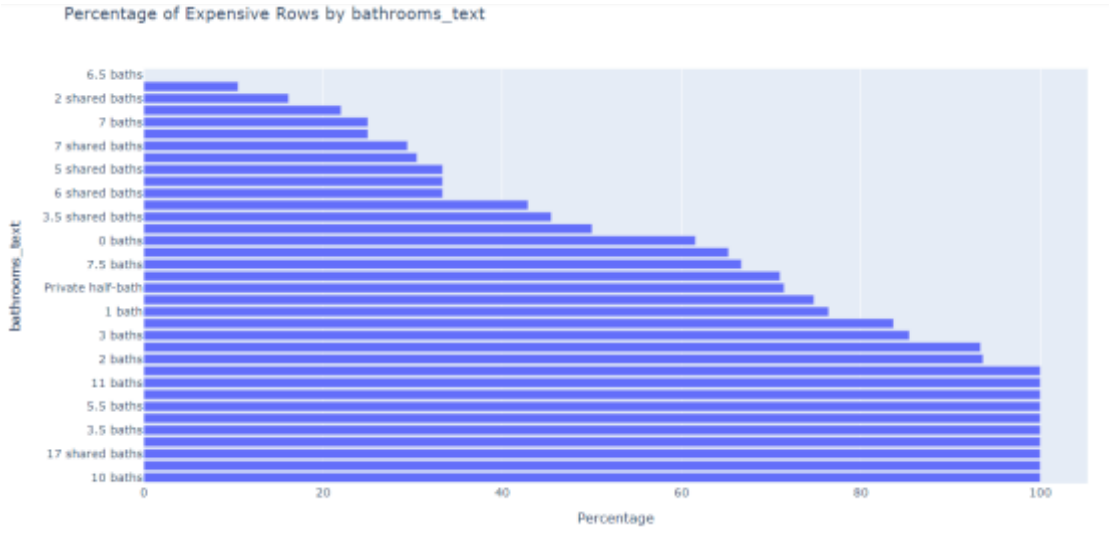
Part 2: EDA:

Boolean Features:

As a first practice we turned "propery_type" into a bunch of Boolean columns, to capture the "grade" of the asset the best we can (Motels, Cabins, Hotels, Apartments…). Full categories appear in the code book. We did a similar thing with baths – either shared or private.

Label Encoding:

Our main purpose was to create as many useful features as possible, so we started by looking into labeled data so we could numerically encode it. We wanted to use the hierarchy found naturally in each feature, so we created a function that checks for every category in the labeled what the percentage of expensive postings in it is. We used this function to hierarchy encode the labels in a reasonable order. Classes with the same ratio got the same label. Here is an example:



{'6.5 baths': 1, '3 shared baths': 2, '2 shared baths': 3, '1.5 shared baths': 4, 'Shared half-bath': 5, '7 baths': 6, '7 shared baths': 7,

In the end, some of the labels were encoded by hierarchy (int) while others were encoded with their "expensive ratio" in the train, depends on our assessment whether or not this classes value to the model can be quantified (and not only labeled).

Engineered Features:

- We used % string features as floats (host_response_rate, host_acceptance_rate)
- Number of months as a host.
- Time since last review.
- We also ranked the amenities by ratio of expensive listings they appeared in, than created these features:
    - N_expensive_amenities (80% of the listings that have them are expensive. About 65% of the unique amenities were on this list).
    - Count of amenities in total.
    - Ratio of expensive amenities for this listing. Might not be useful for amenities that appear very few times .

    We tried counting amenities as expensive only if the amenity appeared multiple times and that kept a certain expensive ratio, but given that the amenities are unique, and we didn't have time to bin them properly by meaning, we had to drop it and use the ratio as is.

- Distance from city center (by using Google Maps API and sending the listings coordinates). After extracting the closest city's coordinates, we calculated the distance between the listing coordinates and the city center's coordinates.
- In addition, by using the train file we directly targeted touristic locations in Tokyo (based on the coordinates range in the train file) and calculated the distance from them, which gave us a better assumption on the property location qualities.

    This feature is meant to be crossed with given location_score to provide another validation / feature for the model. We know that these 2 might have high correlation, so later we preformed feature selection.

The first 3 helped us by giving us data that might affect the host persona (which might affect the pricing). We then turned into floats to fit a regression model.

The fourth feature helped in expressing connections to listings qualities in the model.

The last feature helped in giving expression to location effects on the prediction.

We also removed a few features (most have been used to build calculated features):

```
['last_review','first_review','license','host_id','host_since','host_
verifications','latitude','longitude','amenities']
```

It's important to say that we've noticed a few features that are pretty much telling a similar story, so to stop the model from tilting we preformed dimensional reduction ('lasso') specifically on that group, than narrowed it down to this list, which was also removed:

```
['minimum_nights','maximum_nights',
'minimum_minimum_nights','maximum_minimum_nights',
'minimum_maximum_nights', 'maximum_maximum_nights',
              'availability_30', 'availability_60']
```

The final data set has 49 (excluding id) features which will be examined for importance and reduction based on their influence.

Another direction we wanted to explore was normalizing the avg review scores based on a t-test for example, but given that we don't have each avg variance, this direction is not possible.

Feature Analysis:

We then conducted an analysis on the final features, by creating this table:

	Feature	Unique Values	Null Count	Average	Standard Deviation	Min	Max
0	id	6759	0	3380.000	1.951300e+03	1.000	6.759000e+03
1	host_response_time	4	595	1.175	5.250000e-01	1.000	4.000000e+00
2	host_response_rate	33	595	0.978	1.000000e-01	0.000	1.000000e+00
3	host_acceptance_rate	57	483	0.951	1.200000e-01	0.000	1.000000e+00
4	host_is_superhost	2	1	0.264	4.410000e-01	0.000	1.000000e+00
5	host_listings_count	58	0	15.045	2.079700e+01	1.000	1.680000e+02
6	host_total_listings_count	83	0	22.584	3.322800e+01	1.000	4.620000e+02
7	host_has_profile_pic	2	0	1.000	1.700000e-02	0.000	1.000000e+00
8	host_identity_verified	2	0	0.930	2.550000e-01	0.000	1.000000e+00

(Which goes on… see full table in the code book)

Standardization:

We created a function to calculate the stats on the train, and a function to use these stats on the test.

Different features in the data are of different scales which might tilt a regression model, standardization of the data might balance these outliers so that every feature is examined compared to itself.

Replace Nulls in labels columns:

We tried 4 different methods for that:

MICE: using IterativeImputer object we created a function to fill nulls in every column based on a few (black box) models. We believe, and our results were accordingly, that this method gave the best performance to the model.

Replace nulls with 0: Was a little better, given that 0 should also be close to the avg of every column after Standardization. Still, was not as good as MICE.

Median: We tried filling the null value with the median, while keeping it in a dictionary for the test cases for every column, but that was not proven to be a better method than the avg.

Drop nulls: A basic method for that, decent one, but I believe that wasn't very helpful for our predictions. Also, limited us with our ability to predict on incomplete rows.

Dimensional Reduction:

Our next step was to reduce irrelevant columns based on their variance and influence on our model. We created a function to test 5 different methods (not learned in class) that returns a list of the meaningful features of the model, allowing us to reduce dimensionality. The methods implemented are Lasso, RFE, SFM, Elastic Net and Univariate. We also tried combining it with PCA as well.

In our submitted test, the most influential features were:

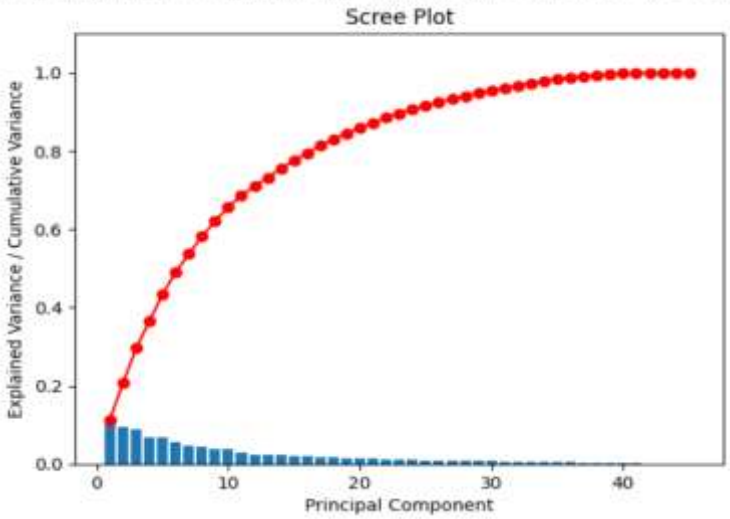
```
['host_listings_count', 'host_total_listings_count', 'property_type',
'accommodates', 'bathrooms_text', 'bedrooms', 'availability_90',
'availability_365', 'review_scores_value', 'calculated_host_listings_count',
'calculated_host_listings_count_entire_homes', 'reviews_per_month',
'shared_bath', 'distance_from_center', 'months_as_host',
'days_since_last_review', 'amenities_count', 'n_of_expensive_amenities',
'ratio_of_expensive_amenities']
```

PCA:

A final step was applying PCA to our data set as created in part 1. As stated, we created it to keep exactly the number of features helpful in charge of X% of the data's variance.

We also implemented scree plot to give a visual representation of the decision made in the __init__ method:

```
Recommended number of components for total variance of at least 0.95 is 30, init function has been updated
```



As shown, PCA does not reduce our dimensionality drastically, but we still tried it.

Important to say that we used these 2 methods to avoid manual correlation tests and to create a smarter model.

We found that combining the PCA with Univariate and RFE was useful, but with other methods such as Lasso it didn't improve the results of our testing.

Logistic Regression Coefficients:

Our final model's coefficients, which compose the logistic regression function, are:

```
array([ 0.67844649, -0.59042208,  1.4390858 ,  1.24419221, -0.13159323,
        0.51768731,  4.47732686, 10.12095918,  6.16597497,  0.46986912,
        0.42659143,  4.67683527,  1.23534994,  0.04046116,  1.61820012,
       -0.05364267,  1.31215926,  2.91098809,  2.69310033,  0.56108681,
        2.08875672,  0.12925444,  0.4321317 ,  5.52076142,  0.18398955,
       -1.20440356, -1.07571367,  4.97756082])
```

```
logistic_regression.intercept_

3.6365290623607525
```

- The complete logistic regression function won't show here because it's just looks bad with so many features…
- We chose logistic regression model since our target is a classification mission, thus the sigmoid function fits better than the regression line.

Part 3: Train:

To pick the best methods we created a recipe flow which was fed with different functions every time.

We trained the model using both regular train test split and cross validation. Important to say that even when using a thread pool, cross validation was a super complex action, so we couldn't really use it to test many different flows and had to settle for a regular train-test split out of our data.

Here are a few of our training results, which led us to decide on the final model (in the code):

Attempts	PCA	Lasso	rfe	sfm	elastic net	univariate	MICE	NULLS (0)	NULLS (remove)	Standartdization	Normalization	test - internal	cross validation	test - external
1	0.9						V			V		0.808	0.65	0.725
2	0.9						V			V		0.808		
3	1						V			V		0.828		0.745
4	1	V					V			V		0.844		0.624
5	0.9	V					V			V		0.83		0.7417
6		V					V			V		0.703		0.781
7	0.9	V						V		V		0.761		
8	0.9	V							V	V		0.71		
After EDA changes														
9	1	V							V	V				0.699
10		V					V			V		0.8429		0.788
11	0.9	V							V	V				0.65747
Added Geoinfo in EDA														
12		V					V			V		**0.8638**		0.7378
13			**V**				**V**			**V**		**0.8519**		**0.804**
14				V			V					**0.87311**		0.778
15					V		V			V		0.8087		
16						0.5	V			V		0.74759		
17	0.9		V				V			V		**0.8586**		0.719
18	0.9			V			V					0.8552		check this one
19						0.5	V			V		**0.8446**		0.5911

Part 4: Final model:

Our final model used, as shown in the table, dimensional reduction (RFE), Mice for nulls, Standardization, and of course all the procedures described in the EDA chapter.

A print of the predicted results for the supplied test is available in the final code block.