KU LEUVEN

Final Project

Advanced Analytics in Business [D0S07a] and Big Data Platforms & Technologies [D0S06a]

Group 3

by:
Jierong Wen - r0912240
Shivam Verma - r0959919
Riya Goyal - r0959390
Marco Chi Chung Fong - r0865521
Ye Liu - r0918311

May 2023

Contents

Assignment 1: Predictive Modeling on Tabular Data

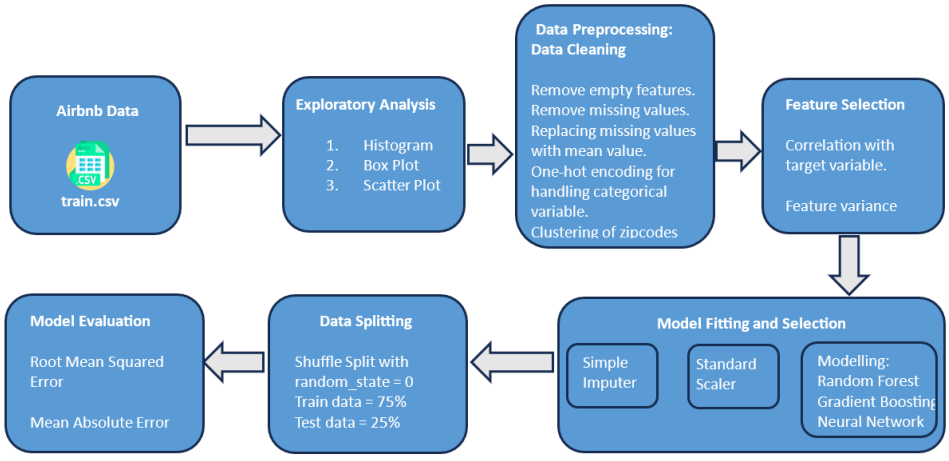
1.1 Introduction

In this assignment, the predictive model is built to predict the price of an Airbnb by training the model on the data set of around 9k Airbnb apartments in Belgium. Our main goal is to construct a prediction model with the least RMSE(Root mean square error) in predicting the Airbnb price. We will create the model with three algorithms: Random Forests, XGBoost and a Neural Network. Then select the model with the least RMSE score. The coding part of the assignment is coded on Jupyter using Python. All the codes of this project are publicly available on the GitHub repository. The Python 3.5 libraries used are the following:

- pandas
- matplotlib
- numpy
- collections
- sklearn
- xgboost
- random

1.2 Pipeline

The following block diagram shows the Data Pipeline with the operations involved in pre-processing and data splitting.



Reading Data: The first step is to load the Airbnb data onto the jupyter server for processing for this we read the train.csv file from the disk.

```
import pandas as pd
main_file_path = 'train.csv'
data = pd.read_csv(main_file_path, low_memory=False)
```

Exploratory Data Analysis: In this step, we try to understand the links between the variables and the data. To find patterns, relationships, and potential outliers, it uses data visualization and exploratory tools.

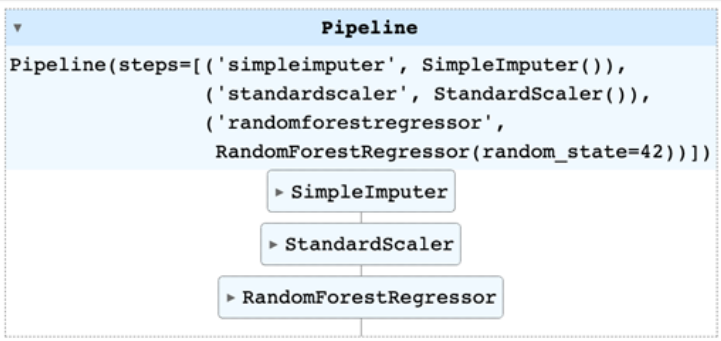
Data Prepocessing: To guarantee the quality and utility of the Data, it must be cleansed and preprocessed. Dealing with outliers, addressing missing values, normalizing, or transforming variables, and feature engineering are all involved in this.

Feature Engineering/selection: This action tries to increase the model's capacity for prediction. To choose the most pertinent features and lessen dimensionality, feature selection techniques like correlation analysis or regularization approaches can also be used.

Model Development: Different models (RF, GB and Neural) were built to predict target.

```
#Algorithm 1: Random Forest Regression with hyperparameters
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
import numpy as np
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
from sklearn.preprocessing import StandardScaler

# Create the pipeline (imputer + scaler + regressor)
my_pipeline_RF = make_pipeline(imputer, StandardScaler(),
                               RandomForestRegressor(random_state=42))
```

Out[33]:



Data Splitting: The data set is splitted into training and testing data by using `train_test_split` method of sklearn library.

```
# Training and Testing Sets
from sklearn.model_selection import train_test_split
train_X, test_X, train_y, test_y = train_test_split(X, y, random_state = 0)
```

Model Evaluation: RMSE(Root Mean Squared Error) is used to compare the 3 models and then the best performing model(with Least RMSE) is selected.

1.3 Exploratory Analysis

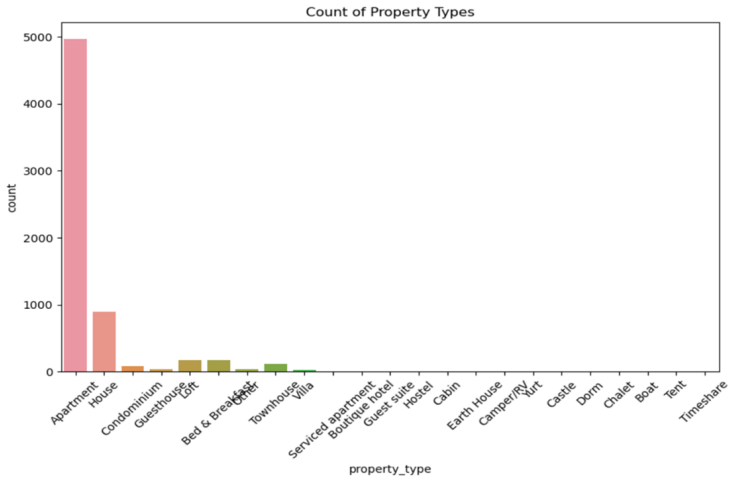
We explored the data to understand better the relationship between the observations and the relationship between each observation and predictor.

1. Check for collinearity among the features.
2. Visualise the relationship between each predictor and target(price)
3. Visualise the target variable to identify any skewness and any necessary transformations
4. Visualise the number of Airbnb properties by zip codes (density of Airbnb apartments/zip code)

Three libraries were used for data visualisation: Pandas, Matplotlib, and Seaborn.

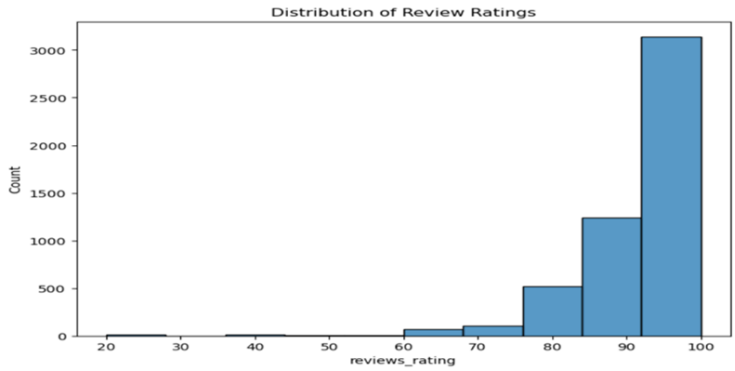
Exploration 1: Count of Property Types

Here we wanted to explore if only a few property types are present. We found that most of the buildings are apartments; hence this feature won't be a perfect feature selection for the model as it will bias because of the significant presence of one type of property.



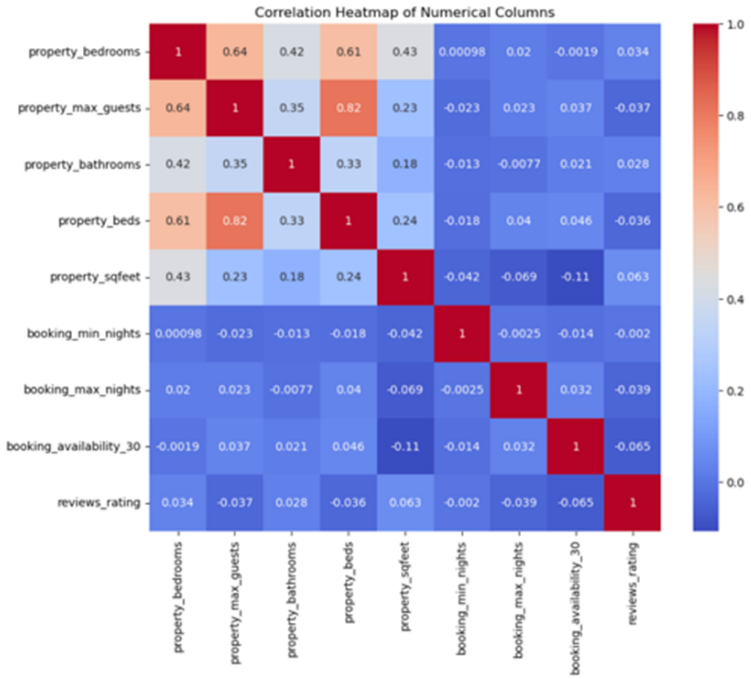
Exploration 2: Distribution of Review Rating

We built a histogram to see if the review rating provides some insight. We discovered that ratings are distributed and could be used as a predictor in the model.

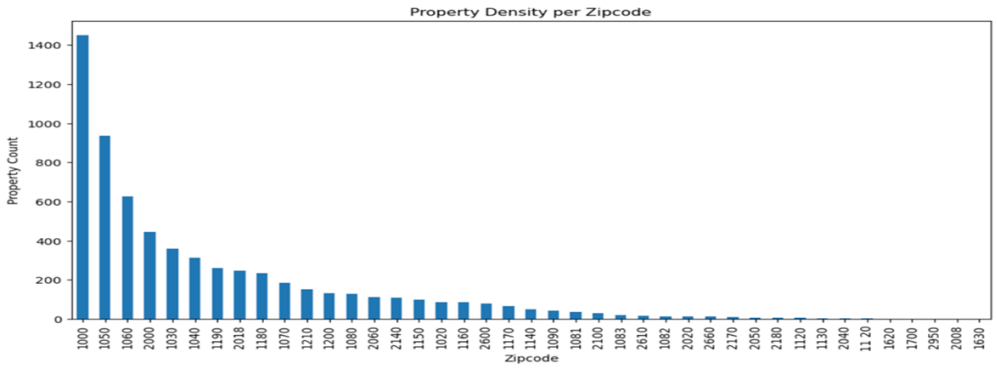


Exploration 3: Finding relation b/w Property type and Review Rating
Found out that service apartment, guest suite and Earth house have yet to receive any ratings. One possible reason could be the fewer properties available for this type.

Exploration 4: Finding Correlation important features The analysis showed that property_bedrooms and property_max_guests are related, similarly property_beds and property_max_guests and property_bathrooms and property_beds are related. One among all the correlated features can be removed to reduce the complexity of the model.

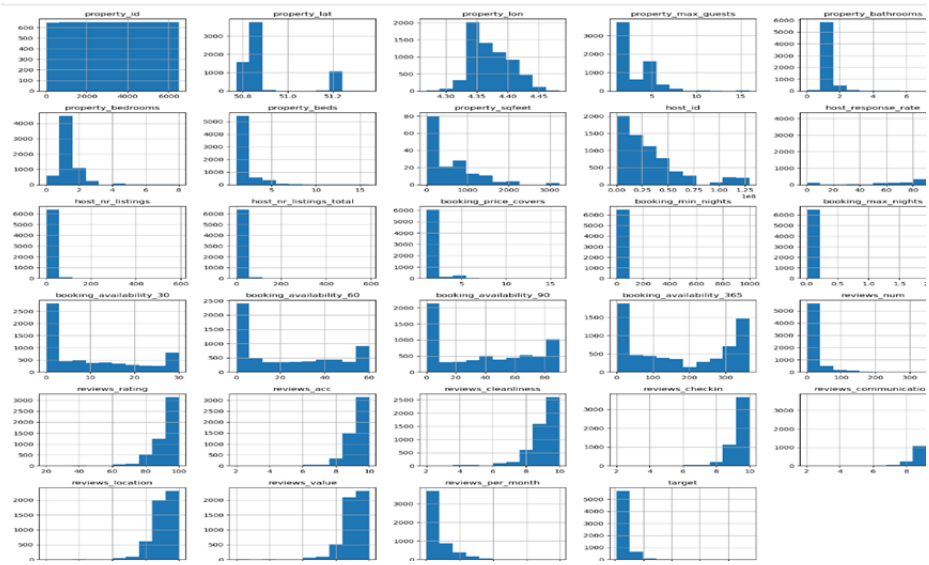


Exploration 5: Count of properties by zip code

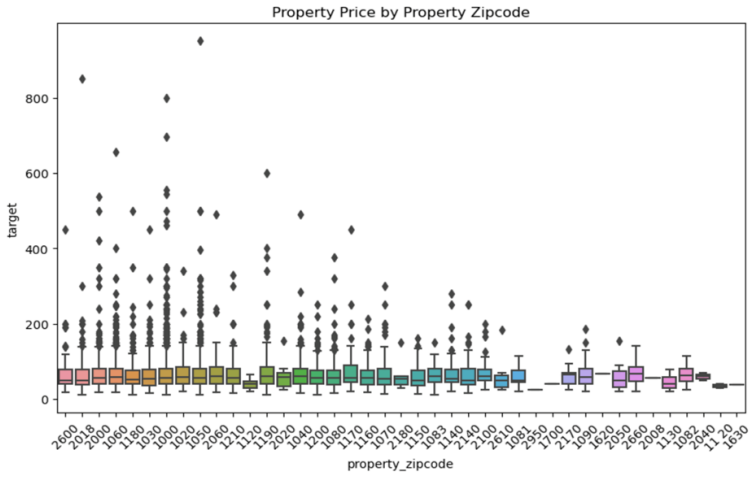


Exploration 6: Histogram of all the numerical(binning) and categorical variable

It is essential to check if features have only one or two prominent values. In that case, it is better to remove that feature as it only adds to the complexity of the model and does not have any predicting power.

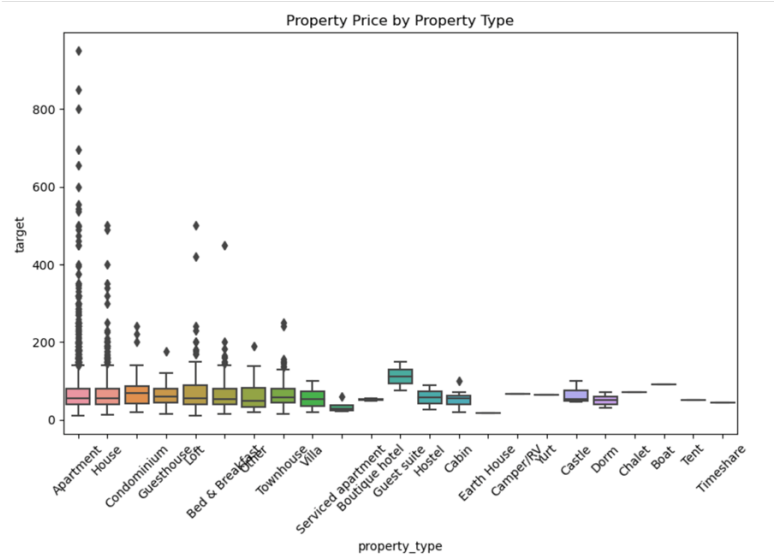


Exploration 7: Box Plot of Property Price by Property zip code



Exploration 8: Box Plot of Property Price by Property Type

Analysis revealed that property type apartment has many outliers, but when clubbing the pricing with the number of apartment properties available and in different zip codes it is available in, we found that properties in 6 zip codes were way costlier than average prices.



1.4 Pre-processing

In this step, we cleaned the data and selected the most valuable features for the models. The file train.csv contains detailed Airbnb apartment data, including various attributes (features) of each listing, such as location, number of bedrooms, bathrooms, type of bed, reviews, property location, property description etc. The raw data contains 6495 records and 55 columns. As all the features do not help predict the price(some are used for indexing and reference), we removed the useless features from the data set.

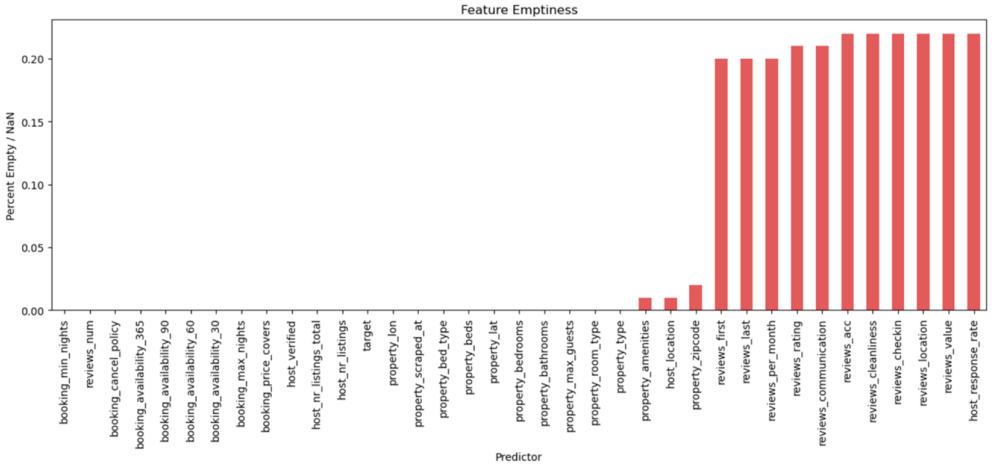
```
#Removing the unnecessary features from the data set
useless=['property_id', 'property_name', 'property_summary',
         'property_space', 'property_desc', 'property_neighborhood',
         'property_notes', 'property_transit', 'property_access',
         'property_interaction', 'property_rules', 'property_amenities',
         ↪  'property_last_updated','host_id', 'host_since',
         'host_about', 'host_response_time',
         'extra']
data.drop(useless,axis=1,inplace =True)
```

Next, we removed the features which have less than 600 valid entries. When we ran the code, property_sqfeet had less than 600 valid entries whereas total rows(listings) are 7000, so we cleaned that column.

```
count_data = data.describe().loc['count', ]<600
for i in range(len(count_data)):
```

```
        if count_data[i]:
            data.drop(count_data.index[i], axis=1, inplace=True)
```

Next, we evaluated all the categorical and numerical variables for feature emptiness. For this we used percent_empty method and plotted the result.



We found features like review_cleanliness, review_checkin, review_value, and host_response_rate to have an emptiness rate of around 25%, but as they can be essential features, we will use them in our model using an Imputer.

Modifying property_zipcde feature: Firstly, we found out number zipcodes in the data set.

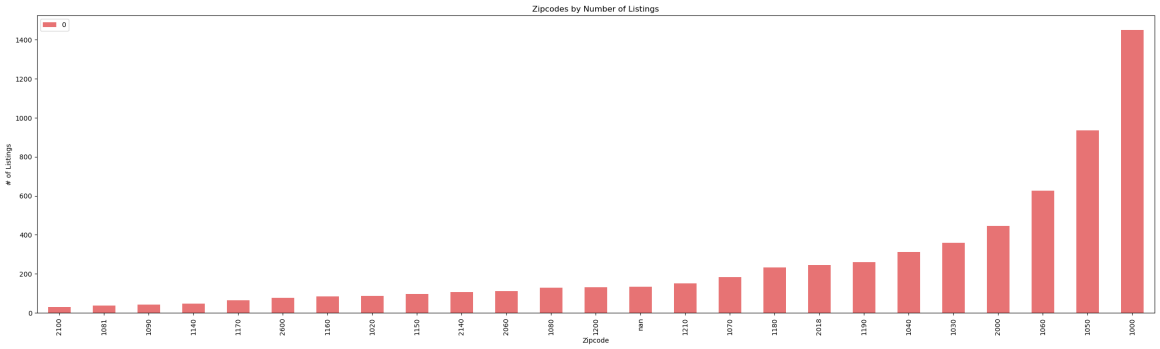
```
from collections import Counter
# Get number of zipcodes
nb_counts = Counter(data.property_zipcode)
print("Number of Zipcodes:", len(nb_counts))
```

```
Number of Zipcodes:  42
```

If we leave 42 zip codes as it is, then during one hot encoding of categorical variable, 42 new features will be added, which will cause the model to overfit, so first, we will see if we can remove less appearing zip codes or we can cluster them into the new feature and name it as property_neighbour.

```
import matplotlib.pyplot as plt
tdf = pd.DataFrame.from_dict(nb_counts, orient='index').sort_values(by=0)
ax = tdf.plot(kind='bar', figsize = (30,8), color = '#E35A5C', alpha = 0.85)
ax.set_title("Zipcodes by Number of Listings")
```

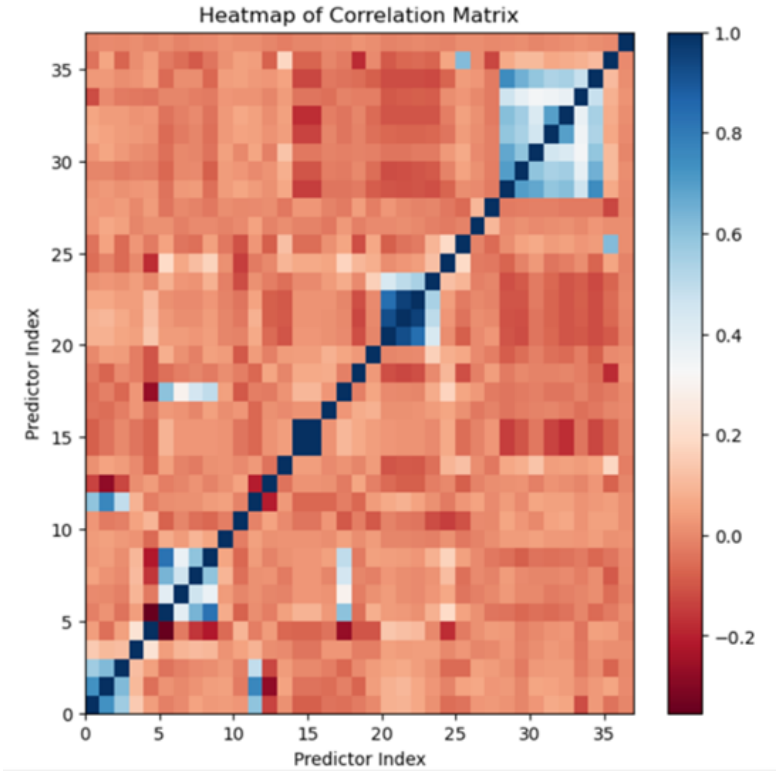
```
ax.set_xlabel("Zipcode")
ax.set_ylabel("# of Listings")
plt.show()
```



We then deleted the zipcodes with less than 5 entries as they can be an outlier and then clustered the remaining zipcodes into neighbour.

```
# Delete zipcodes with less than 5 entries
for i in list(nb_counts):
    if nb_counts[i] < 20:
        del nb_counts[i]
        data = data[data.property_zipcode != i]
dict ={'1000':'1000','1020':'1000','1030':'1000','1040':'1000','1050':'1000'
 ↪  ,'1060':'1100','1070':'1100','1080':'1100',
      '1090':'1100','1081':'1100','1082':'1100','1083':'1100','1120':'1150',
       ↪  '1130':'1150','1140':'1150','1150':'1150',
     '1160':'1200','1170':'1200','1180':'1200','1190':'1200','1200':'1200','
      ↪  1210':'1200','1630':'1600','1620':'1600',
     '1700':'1600','2000':'2000','2008':'2000','2018':'2000','2020':'2000','
      ↪  2040':'2000','2050':'2100','2060':'2100',
     '2100':'2100','2140':'2100','2170':'2100','2180':'2100','2600':'2600','
      ↪  2610':'2600','2660':'2600',
     '2950':'2900','11 20': '1150'  }
def convert_zips_codes(val):
     return dict[val]
df['property_zipcode'] = df['property_zipcode'].apply(convert_zips_codes)
```

With the clustering approach number of zipcodes were reduced to 9.
The next step we did was to create a visualisation of the correlation between features and remove one from pair of highly co-related features.




```
#dropping highly correlated variables
useless =['property_lat','property_beds','reviews_acc','reviews_value']
data.drop(useless,axis=1,inplace=True)
useless =['host_verified','reviews_first','reviews_last','host_location']
data.drop(useless,axis=1,inplace=True)
```

After the feature selection, we have a data frame with 69 features and 6361 rows. We kept 90% of the rows of the initial data frame and deleted 28 columns.

One-hot encoding: Categorical variables need to be One-hot Encoded to be converted into several numerical features and used in a Machine Learning model.

```
#one-hot encoding
data = pd.get_dummies(data)
```

1.5 Model Training and Evaluation

we decided to apply 3 different models -

- **Random Forest**, with the `RandomForestRegressor()` from the Scikit-learn library

- **Gradient Boosting method**, with the `XGBRegressor()` from the XGBoost library
- **Neural Network**, with the `MLPRegressor()` from the Scikit-learn library.

Each time, I applied the model with its default hyperparameters and I then tuned the model in order to get the best hyperparameters I could. The metrics we used to evaluate the models is the **median absolute error** due to the presence of extreme outliers and skewness in the data set.

1.5.1 Random Forest

a) With Default Hyperparameters: Created a pipeline such that it first imputes the missing values, then scales the data and Finally applies the model. We fitted the pipeline to the training data.

```
#Algorithm 1: Random Forest Regression with hyperparameters
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
from sklearn.preprocessing import StandardScaler
# Create the pipeline (imputer + scaler + regressor)
my_pipeline_RF = make_pipeline(imputer, StandardScaler(),
                               RandomForestRegressor(random_state=42))
# Fit the model
my_pipeline_RF.fit(train_X, train_y)
```

We evaluate this model on the testing set. We used both the median absolute error and root-mean-square error (RMSE) to measure the performance of the model. We created the `evaluate_model` function that will evaluate the performance.

```
#evaluating the model
def evaluate_model(model, predict_set, evaluate_set):
    predictions = model.predict(predict_set)
    print(predictions)
    from sklearn.metrics import median_absolute_error
    print("Median Absolute Error: " +
     ↪  str(round(median_absolute_error(predictions, evaluate_set), 2)))
    from sklearn.metrics import mean_squared_error
    from math import sqrt
    RMSE = round(sqrt(mean_squared_error(predictions, evaluate_set)), 2)
    print("RMSE: " + str(RMSE))
```

```
result_RF_test = evaluate_model(my_pipeline_RF, test_X, test_y)
result_RF_train = evaluate_model(my_pipeline_RF, train_X, train_y)
```

Ensuring that the feature selection is done right we performed feature importance available with Random Forrest Regressor.

```
# Get numerical feature importances
importances = list(my_pipeline_RF.steps[2][1].feature_importances_)

# List of tuples with variable and importance
feature_list = list(data.columns.drop("target"))
feature_importances = [(feature, round(importance, 2)) for feature,
↪  importance in zip(feature_list, importances)]

# Sort the feature importances by most important first
feature_importances = sorted(feature_importances, key = lambda x: x[1],
↪  reverse = True)

# Print out the feature and importances
#[print('Variable: {:20} Importance: {}'.format(*pair)) for pair in
↪  feature_importances[0:22]]

# List of features sorted from most to least important
sorted_importances = [importance[1] for importance in feature_importances]

# Cumulative importances
cumulative_importances = sum(sorted_importances)

print(cumulative_importances)
```

```
0.9900000000000004
```

b.) With Tuned Hyperparameters: There are two main methods available for this:

- Random search
- Grid search.

We started with a random search to roughly evaluate a good combination of parameters. Once this is done, I use the grid search to get more precise results.

```
import numpy as np

# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 10, stop = 1000, num =
↪  11)]
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 5)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'randomforestregressor__n_estimators': n_estimators,
               'randomforestregressor__max_depth': max_depth,
               'randomforestregressor__min_samples_split': min_samples_split,
               'randomforestregressor__min_samples_leaf': min_samples_leaf,
               'randomforestregressor__bootstrap': bootstrap}

from pprint import pprint
pprint(random_grid)

from sklearn.model_selection import RandomizedSearchCV

# Random search of parameters, using 2 fold cross validation,
# search across 100 different combinations, and use all available cores
rf_random = RandomizedSearchCV(estimator = my_pipeline_RF,
                               param_distributions = random_grid,
                               n_iter = 50, cv = 2, verbose=2,
                               random_state = 42, n_jobs = -1,
                               scoring = 'neg_median_absolute_error')

# Fit our model
rf_random.fit(train_X, train_y)
rf_random.best_params_
```

```
{'randomforestregressor__bootstrap': True,
'randomforestregressor__max_depth': 80,
'randomforestregressor__min_samples_leaf': 1,
'randomforestregressor__min_samples_split': 4,
'randomforestregressor__n_estimators': 800}
```

1.5.2 XGBoost Gradient Boosting Model

Similar steps to Random forest were applied to XGB model as well. First we applied the model with default hyperparameter and then applied Hyperparameter tuning.

```
#Algorithm 2: Gradient Boosting
from xgboost import XGBRegressor
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')

# Create the pipeline: Imputation + Scale + MLP regressor
my_pipeline_XGB = make_pipeline(imputer, StandardScaler(),
                                XGBRegressor(random_state = 42))

# Fit the model
my_pipeline_XGB.fit(train_X, train_y)

# Create the parameter grid based on the results of random search
param_grid = {'xgbregressor__learning_rate': [0.1, 0.05],
              'xgbregressor__max_depth': [5, 7, 9],
              'xgbregressor__n_estimators': [100, 500, 900]}

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = my_pipeline_XGB,
                           param_grid = param_grid,
                           cv = 3, n_jobs = -1, verbose = 2,
                           scoring = 'neg_median_absolute_error')

# Fitting Tuned XGB model
grid_search.fit(train_X, train_y)
```

1.5.3 Neural Network

a) With Default HyperParameters

```
from sklearn.neural_network import MLPRegressor
# Multi-layer Perceptron is sensitive to feature scaling, so it is highly
↪  recommended to scale your data
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
# Create the pipeline: Imputation + Scale + Feature Selection + MLP regressor
my_pipeline_NN = make_pipeline(imputer, StandardScaler(),
                               MLPRegressor(random_state = 42,
                                            max_iter = 3000))

# Fit the model
my_pipeline_NN.fit(train_X, train_y)
```

b) Hyperparameter Tuning

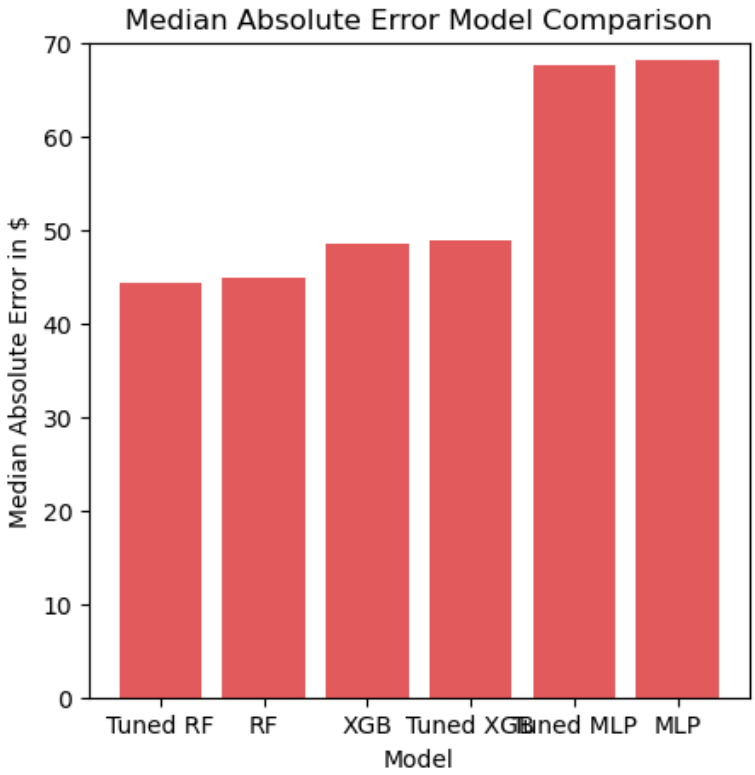
```
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
# Create the pipeline: imputation + MLP regressor
my_pipeline_NN_grid = make_pipeline(imputer, StandardScaler(),
                                    MLPRegressor(hidden_layer_sizes = (100,
                                    ↪  100, 100),
                                                 activation = 'logistic',
                                                 early_stopping = False,
                                                 learning_rate_init = 0.0001,
                                                 solver = 'sgd',
                                                 max_iter = 500,
                                                 random_state = 42))

# Fit the model
my_pipeline_NN_grid.fit(train_X, train_y)
```

1.5.4 Comparing All Models(Default and Tuned)

Below, the table compares the RMSE and MAE scores of all models.

	Model.Method	RMSE	MAE
1	Random Forest	40.78	21.73
2	Tuned Random Forest	36.38	25.09
3	Gradient Boosting(XGB)	44.49	24.59
4	Tuned Gradient Boosting(XGB)	44.88	25.56
5	Neural Network	64.28	33.43
6	Tuned Neural Network	63.52	23.58

When we compared all the models, we found that Radom Forest with tuned hyperparameters gives the best-predicting model(with the least RMSE score.) So, to predict the price of Airbnb in test data we opted for the Tuned RF model.



1.5.5 Predicting result for test data

```
# Writing the output in file
prediction = []
def predict_model(model, predict_set):
    predictions = model.predict(predict_set)
    print(test_data_dummy['property_id'])
    print(predictions)
```

```
    return predictions
result_on_test_data = predict_model(my_pipeline_RF_grid, test_z)
bins = test_data_dummy['property_id']

f = open("output_clustering.csv", "w")

for i in range(len(bins)):
    f.write("{},{}\n".format(bins[i], result_on_test_data[i]))

f.close()
os.getcwd()
```

1.6 Reflection on Public Leaderboard

Initial Rank: **17**
Final Rank: **18**

The shift in ranking on the public leaderboard after evaluating the models on the second half of the data may be because of the following reasons:

RMSE of each group has increased by a significant value which gives the impression that there may be some underlying hidden patterns in the unseen data that our models couldn't capture. There could be meaningful differences between the data we trained the model on and the testing data (especially the second half) used for evaluation.

We have used one-hot encoding for categorical variables. These variables need to be One-hot Encoded to be converted into several numerical features and used in a Machine Learning model. The values of these variables in the second half of the evaluation data set could be very different compared to the values used for training our model. Hence, our model can't perform better on this part of the data set.

We often get different results when we run the same algorithm on different data because of the variance of the machine learning algorithm. It measures how sensitive the algorithm is to the specific data used during training. A more sensitive algorithm has a more significant variance, which will result in more differences in the model, and in turn, the predictions made and evaluation of the model.

1.7 Conclusion

Overall, we have built three predictive models for predicting the prices of Airbnb properties in Belgium, then compared the three models on RMSE metric and selected Tuned Random Forest as the best among the three. The ability of this predictive machine learning model to correctly predict and forecast the target variable has been proven. We have successfully created a strong and trustworthy prediction model by thoroughly examining the dataset, feature engineering, and model selection.

Additionally, feature importance analysis improved the model's interpretability by giving insights into the significant variables affecting the target variable. Decision-making can be aided by this knowledge, which can also assist stakeholders in comprehending the underlying causes of the projections.

In conclusion, this predictive machine learning model has accurately predicted the target variable. Its use in real-world contexts is well supported by its reliable performance, interpretability, and validation outcomes. Stakeholders can make wise decisions and optimise their strategies to attain desired results by utilising the insights received from this approach. To maintain the model's continuous efficacy in changing data settings, it is essential to continue being vigilant in monitoring and developing the model.

Assignment 2: Deep learning on Images

2.1 Introduction

In the second assignment, deep learning on image classification via a pre-trained Convolutional Neural Network is explored on a dataset containing nearly 117K images from restaurants from the Michelin Guide. Our main goal is to construct a CNN prediction model to classify whether one image shows food or the restaurant's interior. The project is developed via Pytorch which is a well-known deep learning framework in Python due to its superior flexibility and accessibility, and all the codes of this project are public on Github.

2.2 Project Pipeline

The full pipeline of the project is presented in Figure 2.1 consisting of five major parts. First, considering the huge size of the given data set (over 20 GB) and the feasibility of manually labeling, only 998 images (679 food images and 319 interior images) are randomly selected from the full data pool, and they are manually labeled in the data preparation stage. Some common data preprocessing techniques such as label encoding, image augmentation, and normalization are implemented afterward to ensure the cleanliness of data and enhancement of the training efficiency. Then, CNN model based on Resnet architecture (short for Deep Residual Network, proposed by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in 2015) would be trained on preprocessed images. Finally, the performance of the tuned model is evaluated, and interpretability analysis is also carried out via grad-CAM (Gradient-weighted Class Activation Mapping). A more detailed discussion on those parts is in the following sections.

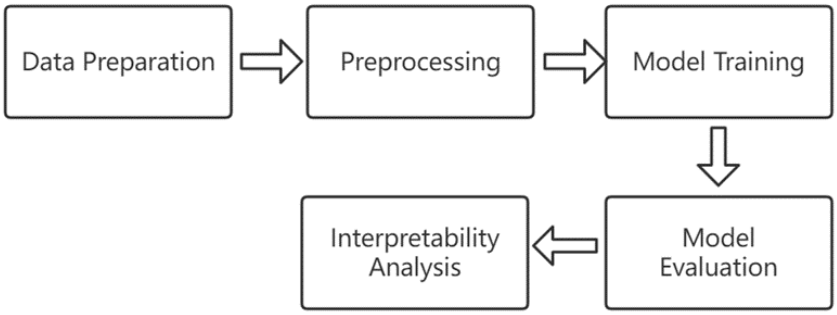


Figure 2.1: Project pipeline

2.3 Data Pre-processing

Text labels for each image created in the data preparation stage would be converted to 0-1 encoding, which can be expressed by

$$l_i = \begin{cases} 0, \text{ food} \\ 1, \text{ interior} \end{cases} \tag{2.1}$$

Furthermore, data augmentation technique has been implemented for preventing over-fitting and ensuring the robustness of model. Most common image augmentation functions are integrated into the package torch.vision.transforms which can be easily accessed. In this project, `train_tf` from the code snippet below is served as an image transformation pipeline for training data, image would be resized, randomly flipped horizontally, randomly flipped vertically, converted to a Pytorch tensor, and normalized sequentially. Note that random flipping is not included in the transformation for the validation set and test set.

```
        self.train_tf = transforms.Compose([
            transforms.Resize((self.img_size,self.img_size)),
            transforms.RandomHorizontalFlip(),
            transforms.RandomVerticalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
)
        ])
        self.val_tf = transforms.Compose([
            transforms.Resize((self.img_size,self.img_size)),
            transforms.ToTensor(),
            transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
)
        ])
```

2.4 Architecture of ResNet

One of the major features of ResNet is the utilization of Residual connections, which allows an output from an earlier convolutional layer to connect with input for the future convolutional layer through simple summation. By directly passing input from the earlier layer to the future layer, gradients vanishing in backpropagation could be alleviated since gradients can flow directly through the network via a "shortcut". The architecture of ResNet is mainly composed of several blocks consisting of a series of convolutional layers, moreover, batch normalization is also heavily used after each convolutional layer. From Figure 2.2, we can see the full architecture of ResNet. In this assignment, ResNet18 (ResNet with 18 layers) is utilized for later training process. ResNet 50 which carries a much deeper structure compared to ResNet18 is also tried in our project, but it fails to be preferred for its too complex architecture leading to a relatively long training time and no significant improvement in the later prediction performance.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 2.2: Architecture of ResNet[1]

2.5 Model Training

Since the model we used has already been pre-trained on ImageNet dataset, we only need to modify the last layer as our customized classifier and fine-tune this classifier based on our own data set, this process is usually called transfer learning. ResNet18 with pre-trained weights is provided from torch.nn module and we can easily employ the model via models.resnet18(pretrained=True).to(device), where `to(device)` refers that model would be passed to GPU and training process can be accelerated based on Nvidia CUDA platform.

From the code block below, we add an additional dropout layer with probability of 0.2 as a regularization technique for preventing overfitting, and the output size of ResNet is modified to the number of classes we have in our own data set.

```
#Choose what model are used in this experiment
model = models.resnet18(pretrained=True).to(device)
#Modify the last layer of model to match the size of output
classifier = nn.Sequential(
    nn.Dropout(0.2),
    nn.Linear(in_features=model.fc.in_features,out_features=2)
)
model.fc = classifier.cuda()
```

Before training, the data set has been randomly split into 3 subsets – training set, validation set and test set. The sizes of these three subsets are approximately 80%, 10% and 10% respectively. Validation set would be used in the training process to monitor the performance of model at each epoch (or batch) when tuning the hyperparameter, whereas test set is used to evaluate the final performance of model after training. Moreover, a few parameters like optimizer and loss function need also to be decide before training.

2.5.1 Optimizer

The optimizer implemented in the project is Adam optimizer. Adam optimizer (adaptive moment estimation) is an optimization method which combines with AdaGrad and RMSProp. It computes each individual learning rate by estimating the first and second moments of gradients. The advantage of Adam is its low memory requirement and few tuning on the hyperparameters, it also outperforms other optimization algorithms for its high computation efficiency.

```
optimizer = optim.Adam(model.fc.parameters(), lr=0.0001)
```

lr parameter in the above code sippet refers to the learning rate in Adam optimizer, which is a hyperparameter that need to be tuned. However, we can also set up a learning rate scheduler via function `torch.optim.lr_scheduler.StepLR()` embedded in pytorch package to allow learning rate to gradually decay during training. The learning rate is set to be reduced by a factor (gamma) 0.5 every 5 epochs in our case.

```
lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=5, gamma= 0.5)
```

2.5.2 Loss Function

Loss function used during training is cross entropy loss which is a commonly used metric to quantify the difference between two different probability distributions. It is also considered as

a measure for loss in classification problems with multi-classes, and it can be mathematically expressed as,

$$H(p,q) = -\sum_{x \in \text{labels}} p(x) \log q(x), \tag{2.2}$$

where $p(x)$ refers to probability distribution of true label and $q(x)$ stands for probability distribution for predicted label. Parameters in CNN model are estimated by optimizing (i.e., minimizing) the loss function, and the gradient of loss function is computed via chain rule backward from the last layer to the first layer in the back-propagation algorithm.

2.5.3 Training

Since it is extremely inefficient and cumbersome to do deep learning on CPU, model in our project is trained on cloud with high-performance GPU (Nvidia Tesla T4). From figure 2.3, training loss keeps slightly decreasing as expected and becomes generally stable when number of steps increases though some fluctuations still exist. As for the performance on validation set at each epoch from figure 2.4, classification accuracy gradually grows, and the curve becomes relatively flat after epoch 20. Prediction accuracy approaches highest (93.9%) at epoch 34 and the model of that epoch is saved as the optimal model we obtained from training stage.

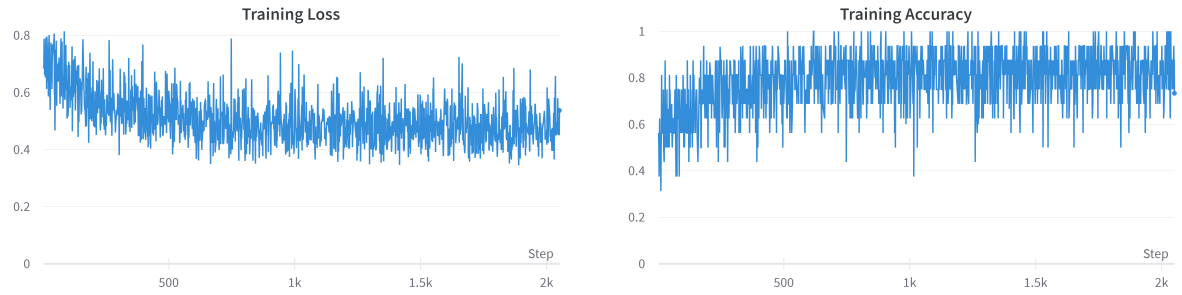


Figure 2.3: Training Loss Curve (left:cross entropy loss, right:classification accuracy, step means cumulative number of mini-batch iterated)

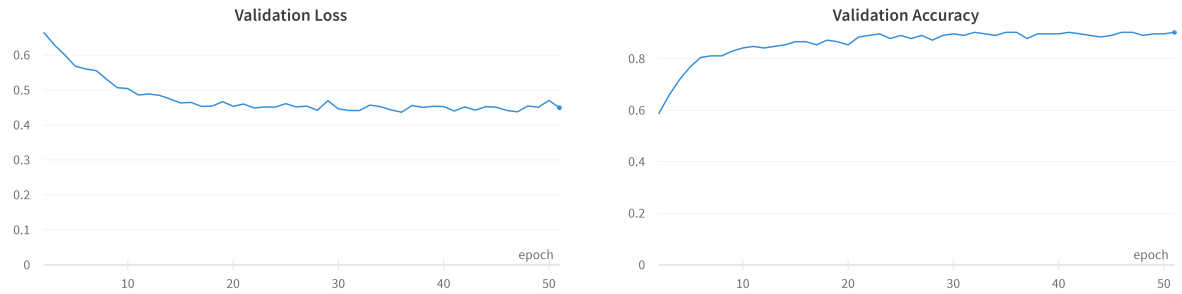


Figure 2.4: Validation Loss Curve (left:cross entropy loss, right:classification accuracy)

2.6 Model Evaluation

Next, the chosen model would be evaluated on the test set (82 images) through computing classification accuracy. The confusion matrix of the test result is plotted in Figure 2.5, which indicates the overall prediction accuracy is approximately 90.2% with 94% accuracy rate on food images and 84.4% accuracy rate on images of restaurant interior. In General, the prediction performance of the model is quite acceptable on both food and interior images.

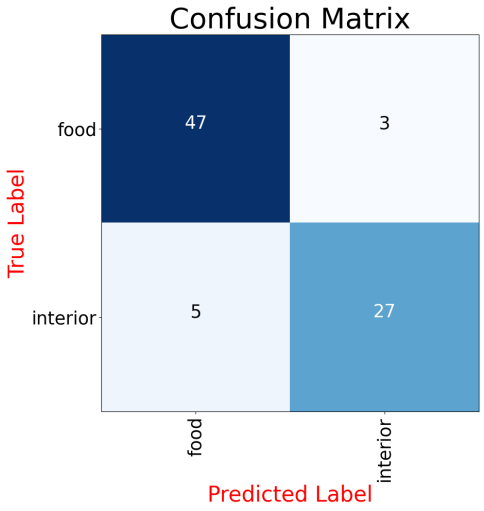


Figure 2.5: Confusion Matrix for Predicted Result

2.7 Interpretability Analysis

Although CNN based models possess superior learning performance under multiple computer vision tasks, their lack of interpretability and transparency is still a major concern in the field of artificial intelligence. In this section, Grad-CAM (Gradient-weighted Class Activation Mapping) technique is introduced to provide some "visual explanations" on the model we trained previously.

Grad-CAM utilizes gradients of the classification score with respect to the last convolutional layer to quantify the contributions of each parts of an input image to the final classification score[2]. Importance plot yielded by Grad-CAM via pytorch-grad-cam package is presented in Figure 2.6, where the red-color part represents having greater impact on the classification result. It can be observed that the model successfully to "focus on" the food part in both of the images, but some non-food information such as the floor of the restaurant is mistakenly treated as food from right image though both images are predicted accurately by the model.

Figure 2.6: Grad-CAM Saliency Map

2.8 Conclusion

Overall, the model trained in this assignment does well in terms of prediction on whether an image is food or interior of the restaurant and gains over 90% accuracy on test set. However, the model is not entirely unbiased since some predictions might be still based on wrong information. In order to further improve its performance, there are a few things we can do in the future.

- Increase the size of training data since 998 images are indeed too limited to do deep learning.
- Make the classes of data more balanced.
- Different optimizers can be tried.
- Different CNN architectures can be used (such as Vgg, AlexNet, GoogleNet...).

Assignment 3: Predicting on Streamed Textual Data

3.1 Introduction

This part of the assignment aims at constructing a predictive model using Spark (Structured) Streaming and textual data. The dataset is based on Steam which the model will predict the score (upvote or downvote) based on the game reviews of the newly released game by the user.

3.2 Project Pipeline

The structure mainly consists of 3 major parts which are the Data Streaming Part, Logistic Regression Model Training Part and the Prediction Part.

Figure 3.1: Flowchart of the Spark Structure

3.3 Data Collection

3.3.1 Data Streaming

We have utilized the sample which the professor has given to us to collect the data. The code is as follows:

```
# Construct the socket stream and then print out the JSON files that we have
 ↪  downloaded
lines = ssc.socketTextStream("seppe.net", 7778)
lines.pprint()

# Save the streaming files to a specific directory for storage
lines.saveAsTextFiles("file:///C:/Users/Spark/Data")

# Construct socketstream
ssc_t = StreamingThread(ssc)
ssc_t.start()
ssc_t.stop()
```

3.3.2 Data Structure

We are aware that the JSON consists of 4 attributes which are review id, app id, review text and label. The label showcases whether the review suggests the game while the 0 means downvote and 1 means upvote. The data structure can be seen as follows:

$$label_i = \begin{cases} 0, \text{ downvote} \\ 1, \text{ upvote} \end{cases} \tag{3.3}$$

```
{"review_id": "139038311", "app_id": "1372530", "review_text": "would love
↪  to recommend this game if it weren't for all the annoying UI bugs that
↪  make it almost impossible to manage anything if you progress far
↪  enough", "label": 0}
{"review_id": "139042017", "app_id": "754890", "review_text": "Gorgeous
↪  graphics, intriguing story line, puzzles a bit tricky.", "label": 1}
```

We can see the files that have been saved in the directory by each streaming epoch and the relevant data if data has been received for each epoch. If there is data received from the API, part-***** files will be downloaded in the directory as shown below.

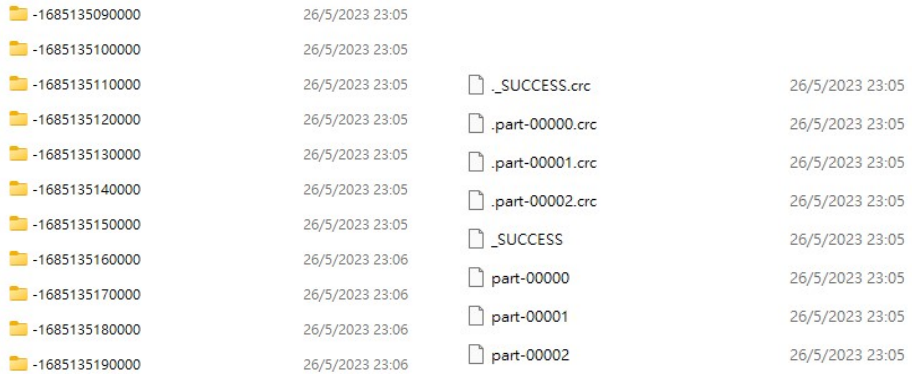


Figure 3.2: Streaming Data Collected from Seppe.net by Spark

3.3.3 Data Transformation

As we have thousands of files after the streaming, we have to consolidate all the data for the ease of further processing. We decided to consolidate all the files into JSON files as the data is JSON based. The code is as below:

```
from pyspark.sql import SparkSession
import os

# Load the directories
parent_directory = "C:/Users/chich/Desktop/spark/data"
subdirectories = [os.path.join(parent_directory, name) for name in
 ↪  os.listdir(parent_directory)
                  if os.path.isdir(os.path.join(parent_directory, name))]

df_list = []

# Create a SparkSession
spark = SparkSession.builder.getOrCreate()

for s in subdirectories:
    # Read the files from the directory as an RDD
    rdd = spark.sparkContext.textFile(s)

    # Convert the RDD to a DataFrame
    df = spark.read.json(rdd)

    # Check whether the file carries JSON data
    if not df.rdd.isEmpty():
        df_list.append(df)

# Combine the dataframes
combined_df = df_list[0]
for i in range(1, len(df_list)):
    combined_df = combined_df.unionAll(df_list[i])

# Save the combined DataFrame as JSON files
combined_df.write.json("C:/Users/chich/Desktop/spark/output")
```

3.4 Prediction Model

3.4.1 Choice of Prediction Model

As we have discussed above, our training dataset consists of 4 attributes that we reckon review text and label are useful for constructing our prediction mode. Therefore, this project can be regarded as a text classification task engineered by Spark. We understand that the transformer model is the State of the Art model for this task but we are currently focusing on training a model so we can come up with several model choices including Logistic Regression, Random Forest, and KNN Models. According to the comparative analysis done in 2020 [3], the logistic regression model performs the best in the entertainment category for text classification as shown as the chart below.

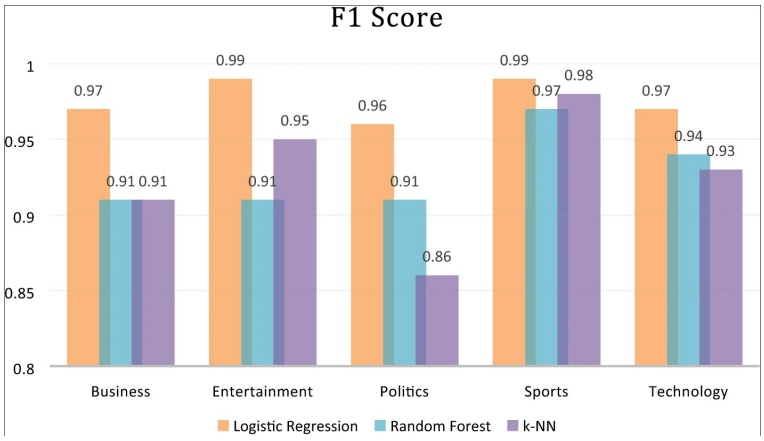


Figure 3.3: Performance of models in text classification by Shah (2020) [3]

3.4.2 Text Engineering

As we are now focusing on the review text, we decided to tokenize the review text for analysis.

```
# Tokenize the review text
tokenizer = Tokenizer(inputCol="review_text", outputCol="tokens")
df = tokenizer.transform(df)
```

After that, we have to remove stop words from the review text. This is because we have to remove the common words in the text which do not have an effect on our analysis such as "a", "an" and "the". The stop word remover is already embedded in the pyspark.ml package.

```
# Remove stop words from the tokens
stopwords_remover = StopWordsRemover(inputCol="tokens",
↪  outputCol="filtered_tokens")
```

```
df = stopwords_remover.transform(df)
```

We then convert the filtered tokens into feature vectors. The purpose of this step is to convert the textual data into a numerical representation that can be used as input for our learning model. We have used CountVectorizer as shown below which is a common technique used for converting text data into a feature vector representation.

```
# Convert the filtered tokens into a feature vector using CountVectorizer
vectorizer = CountVectorizer(inputCol="filtered_tokens",
↪  outputCol="features")
vectorizer_model = vectorizer.fit(df)
df = vectorizer_model.transform(df)
```

3.4.3 Logistic Regression Model

As we have discussed above, we would use logistic regression for the training algorithm. We have also utilized LogisticRegression from the pyspark.ml library. We then save the trained model for further prediction.

```
# Train a logistic regression model on the training data
lr = LogisticRegression(featuresCol="features", labelCol="label")
lr_model = lr.fit(df)
# Save the trained model for prediction use
lr_model.save("C:/Users/chich/Desktop/spark/model")
```

3.5 Prediction

After training our model, we have obtained a model for our prediction. We can simply load the model into the spark for streaming prediction.

```
# Load in the model if not yet loaded:
if not globals()['models_loaded']:
    globals()['my_model'] =
    ↪  LogisticRegressionModel.load("C:/Users/chich/Desktop/spark/model")
    globals()['models_loaded'] = True
```

The following picture shows how the output is shown on our end. Column pred is the prediction generated by our prediction model.

```
========= 2023-05-28 15:18:10 =========
+-------+-----+---------+--------------------+
| app_id|label|review_id|         review_text|
+-------+-----+---------+--------------------+
|1268750|    1|139149761|Good pacing. Fast...|
|1268750|    1|139149208|I hate is so much...|
|1268750|    1|139149100|  I'm Doing my part!|
+-------+-----+---------+--------------------+

+-------+-----+---------+--------------------+--------------------+
| app_id|label|review_id|         review_text|                pred|
+-------+-----+---------+--------------------+--------------------+
|1268750|    1|139149761|Good pacing. Fast...| 0.8453209633943649|
|1268750|    1|139149208|I hate is so much...|0.06571874479063111|
|1268750|    1|139149100|  I'm Doing my part!|0.29514153142889554|
+-------+-----+---------+--------------------+--------------------+

========= 2023-05-28 15:18:20 =========
+-------+-----+---------+--------------------+
| app_id|label|review_id|         review_text|
+-------+-----+---------+--------------------+
|1268750|    1|139149055|    This game is lit|
|1268750|    1|139148727|Feels like  the M...|
+-------+-----+---------+--------------------+

+-------+-----+---------+--------------------+------------------+
| app_id|label|review_id|         review_text|              pred|
+-------+-----+---------+--------------------+------------------+
|1268750|    1|139149055|    This game is lit|0.7357374073877095|
|1268750|    1|139148727|Feels like  the M...|0.8219390015604405|
+-------+-----+---------+--------------------+------------------+
```

Figure 3.4: Prediction Output

3.6 Conclusion

In this part of the assignment, we have covered constructing a predictive mode using Spark. We start by streaming the data and investigating how to read and process a JSON dataset, including tokenization, stop words removal, and feature vectorization. We then explore how to train a logistic regression model on the processed data and make predictions. This part of the assignment has showcased how to build and utilize streaming and prediction models efficiently within the Spark ecosystem.

Assignment 4: Graph Analytics

4.1 Introduction

In the fourth assignment, we conducted network analysis on a graphical dataset scrapped from Twitch, a popular live streaming service platform majorly focusing on video game live streaming. The main objective of this assignment is to investigate the underlying community structure-property of streamers on Twitch via community detection algorithm. More specifically, we aim to perform a community mining analysis on both regular streamers and blocked streamers based on games and tags associated with them, to analyze whether blocked streamers present any noticeable patterns in terms of games and tags they used. Furthermore, we could also identify some potential factors that contribute streamers to be blocked by Twitch from the above graphic analytics process.

4.2 Graph Schema and Used Tools

The network dataset we explored in this assignment includes 4 different types of nodes, namely streamer, game, tag, and squad, and each node is connected with directed edges. The relationships between these nodes are described in Figure 4.1. We query the data via Cypher on Memgraph and visualize the results using Gephi.

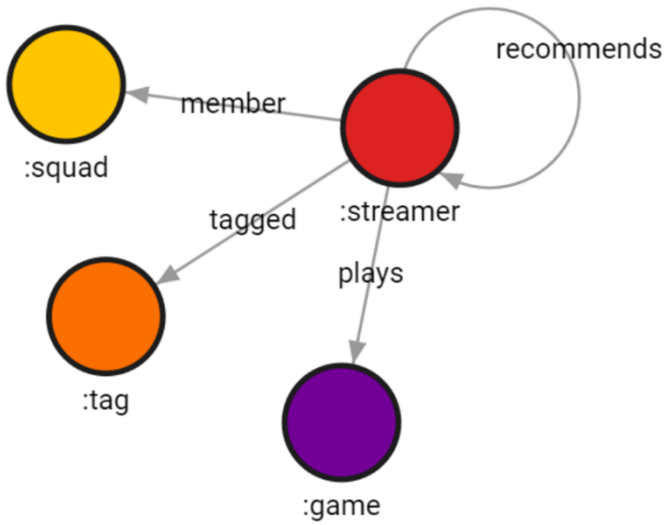


Figure 4.1: Graph Schema of Twitch Dataset

4.3 Community Mining on All Streamers

Initially, we do some preliminary explorations on the data in order to get a basic understanding of the general pattern within the network. We first query all the streamers together with their corresponding tags and games, then use Gephi to visualize the results after community mining analysis.

4.3.1 Tag

We could select all streamers along with their tags by using the query below,

```
match (s:streamer)-[ta:tagged]->(t:tag)
return *;
```

After importing the output data into Gephi, we applied Force Atlas 2 algorithm to adjust the layout of the graph and rank the size of each node by computing their degree centrality in the graph. In Gephi, communities are detected by running modularity in the graph which stands for a type of measurement on the strength of a division of a graph into communities. Higher modularity typically indicates there are denser internal connections within the same cluster and fewer connections to nodes in different clusters.

The visualization regarding connections between all streamers to tags is shown in Figure 4.2, and different colors are labeled on the node according to different modularity classes computed by Louvain method. It can be observed that several communities are detected in the figure and most of them represent language tags indicating which specific language the streamer would use in the live streaming. It is reasonable that language tags dominate the whole graph since it is a compulsory tag that would show on every streamer (it is probably categorized by language setting of the streamer). The most discernable cluster (colored in purple) gathered in the center of the graph is streamers tagged by "English", which should not be surprising since Twitch is an American company, therefore, most of the streamers should be English-speaking. Beyond that, there are no noticeable clues shown in the figure.

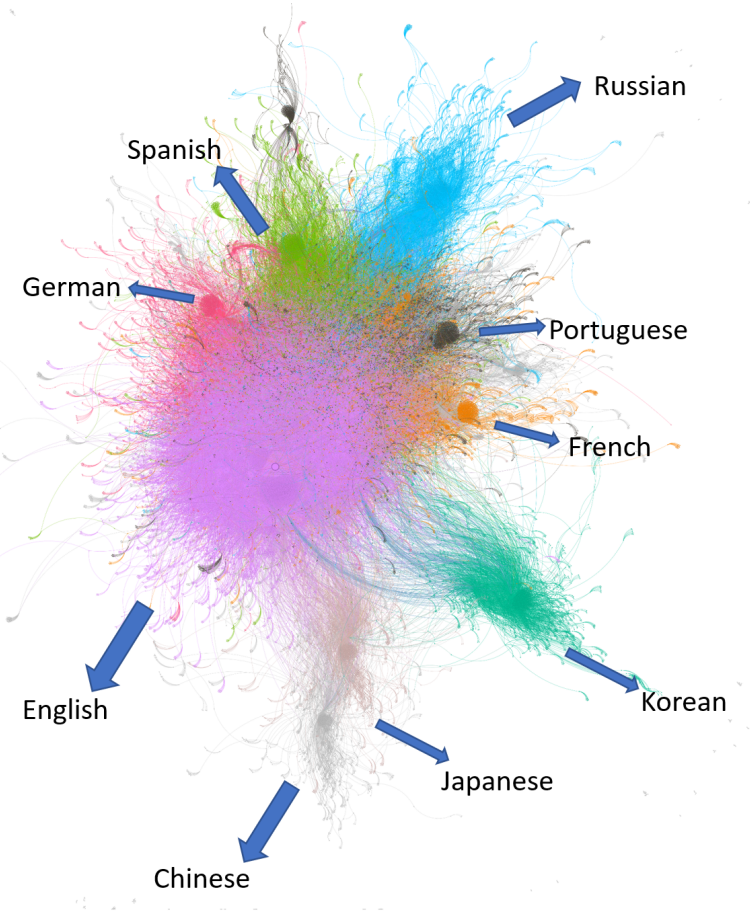


Figure 4.2: Visualization of All Streamers with Tags

4.3.2 Game

Next, we can use the same approach to have an inspection of the graph with all streamers and the games they play. From Figure 4.3, unlike the observation in the previous section, we cannot find any valuable clues rereading the relationship between streamers and games since no clear community structure can be detected and most of the nodes (streamer) are cluttered in the center of the graph. Guessing that since one streamer can play multiple games, it is highly possible that the games he/she plays are not exactly of the same category (for example, it is common to have PUBG: Battlegrounds and League of Legends in one's playlist, both of the two games are popular but they belong to completely different categories).

```
match (s:streamer)-[p:plays]->(g:game)
return *;
```

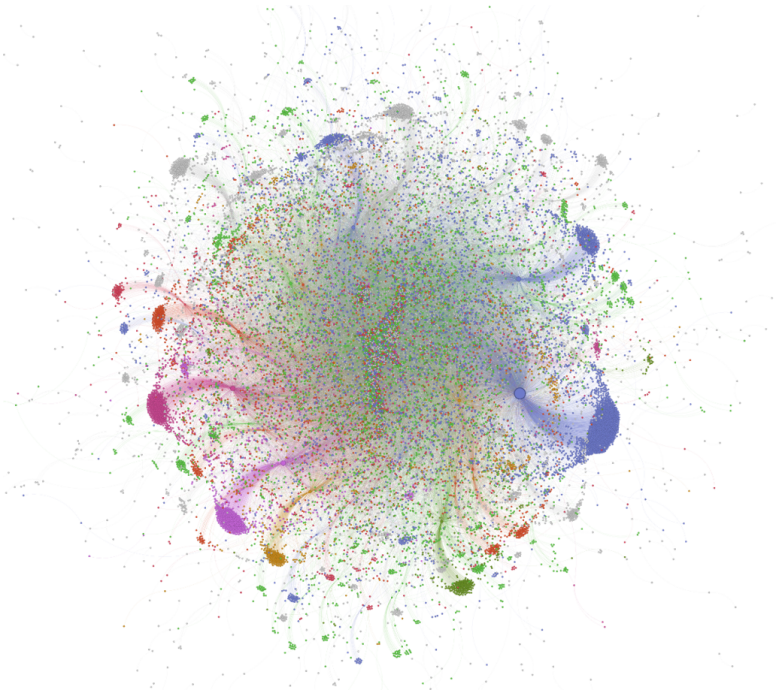


Figure 4.3: Visualization of All Streamers with Games

4.4 Community Mining on Blocked Streamers

4.4.1 Thought Process

In the second section for community mining, we narrow down the search range to the streamers blocked or banned by Twitch to examine whether they use particular tags or play particular games. Considering it is inaccurate if we only rank the tags or games by the number of blocked streamers because the sizes of different tags or games vary significantly, therefore we calculate the blocking rate for each specific tag or game which can be expressed as

$$\text{blocking rate} = \frac{\text{number of blocked streamers}}{\text{number of all streamers under that tag/game}} \tag{4.4}$$

Our assumption is simple, if there is a strong preference of blocked streamers on tags and games, it should be presented on the data which means the block rate should be abnormally high compared to others. These "preferred" tags or games of blocked streamers should also be reflected on the visualizations from community mining.

4.4.2 Tag

Numerical Comparison:

By using Cypher, block rates for each tag can be retrieved from the data set, note that only

when the size of the tag (number of streamers in the tag) is larger than 10 would be considered since the block rate would no longer be accurate if the size is too small. We set the cutoff value for block rate as 50%, that tag would be flagged as an anomaly if there are more than 50% streamers in that tag being blocked by Twitch.

```
match (s:streamer) -[ta:tagged] -> (t:tag)
with t, count(s) as numofstreamer
optional match (bs:streamer)-[:tagged] -> (t)
where bs.description is null
with t, numofstreamer,
     count(bs) as numofblockedstreamer
with t, numofstreamer,
     numofblockedstreamer,
     toFloat(numofblockedstreamer)/toFloat(numofstreamer) as percentage
where percentage < 1 and percentage > 0.5 and numofstreamer>10
return t.name as tagname,
       t.description as tagdescription,
       numofstreamer,
       numofblockedstreamer,
       percentage
order by percentage desc
limit 50;
```

The output of this query contains a data frame with 5 columns, and data is sorted in descending order by the calculated blocking rates as shown in Figure 4.4.

Data results Graph results unavailable Download Results Fullscreen

#	tagname	tagdescription	numofstreamer	numofblockedstr...	percentage
1	Bonus	null	30	29	0.967
2	switzerland	null	12	11	0.917
3	Gamble	null	30	27	0.9
4	Switzerland	For streams and conte...	19	17	0.895
5	Gambling	null	32	28	0.875
6	austria	null	15	13	0.867
7	Austria	For streams and conte...	21	18	0.857
8	онлайнказино	null	15	12	0.8
9	Slots	null	68	53	0.779
10	gamble	null	18	14	0.778

Figure 4.4: Tags with High Blocking Rate

It can be seen that the flagged tags are basically homogeneous since they are all related to gambling more or less. Although some "irrelevant" tags like "switzerland" and "austria" seem to be "mistakenly included", we still find that the majority of the live streaming under those

tags is related to gambling (from Figure 4.5). We can conclude that blocked streamers are strongly associated with tags that are related to gambling.

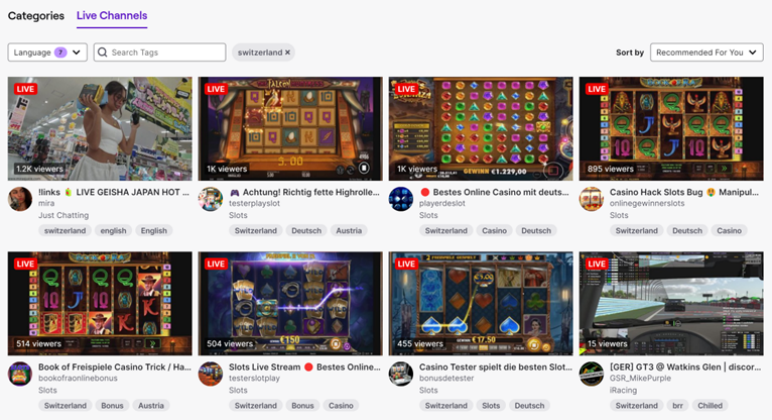


Figure 4.5: Live Streaming under "switzerland" tag

Visualization:

Visualization on blocked streamers and corresponding games can be performed via the same methodology as we discussed in the previous section. Compared with the graph of all streamers with games they play in Figure 4.7, the community structure is much clearer, and several clusters can be identified including "Virtual Casino" and "Slots" which have been flagged for their unusually high blocking rate. Some games with sexual content discussed in the previous part are not clearly presented in this figure since their sizes of them are too small to observe. Furthermore, notice that the most played "game" by blocked streamers is "Just Chatting" which is technically not a true game but a rather broad category intended for casual chatting with viewers. Due to its comparatively unrestricted content compared to other specific games, misconduct and even violation such as sexual harassment are more likely to happen during "chatting". Therefore, it is not strange that "Just Chatting" is most popular for blocked streamers.

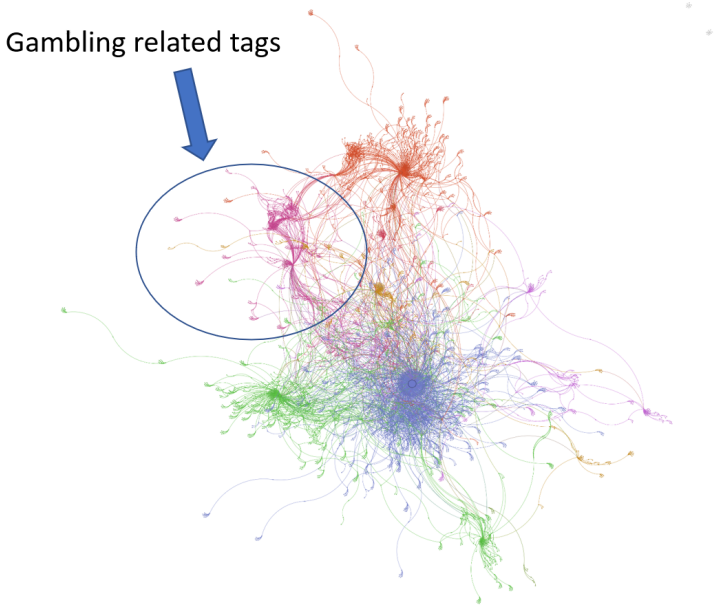


Figure 4.6: Visualizations of Blocked Streamers with Tags

4.4.3 Game

Numerical Comparison:

Using a similar approach, we can compute the corresponding block rate for each game and then filter the games with block rates larger than 25%.

```
match (s:streamer) -[p:plays] -> (g:game)
with g, count(s) as numofstreamer
optional match (bs:streamer)-[:plays] -> (g)
where bs.description is null
with g, numofstreamer,
     count(bs) as numofblockedstreamer
with g, numofstreamer,
     numofblockedstreamer,
     toFloat(numofblockedstreamer)/toFloat(numofstreamer) as percentage
where percentage < 1 and percentage > 0.25 and numofstreamer > 10
return g.id as gamename,
       numofstreamer,
       numofblockedstreamer,
       percentage
order by percentage desc
```

```
limit 50;
```

The output data is shown in Figure 4.7, and the resulting games are mainly consisting of two categories, that is games with potentially sexually explicit content (e.g. Sexy Sniper, Anime World) and games with stimulated gambling components (e.g. Slots, Virtual Casino, Always On). Significantly high blocking rates have been detected from these two types of games, which means streamers who play games containing sexual or gambling content might face a higher probability of being blocked by Twitch. As for "The Simpsons" and "Lineage", we do not have enough information to conclude why they also have extremely high blocking rates compared with other games.

Data results | Graph results unavailable | Download Results | Fullscreen

#	gamename	numofstreamer	numofblockedstreamer	percentage
1	Sexy Sniper	16	14	0.875
2	The Simpsons	15	13	0.867
3	Anime World	27	19	0.704
4	Lineage 2	25	12	0.48
5	Slots	243	82	0.337
6	Virtual Casino	366	114	0.311
7	Always On	13	4	0.308

Figure 4.7: Games with High Blocking Rates

Visualization:

Visualization on blocked streamers and corresponding games can be performed via the same methodology as we discussed in the previous section. Compared with the graph of all streamers with games they play in Figure 4.8, the community structure is much clearer, and several clusters can be identified including "Virtual Casino" and "Slots" which have been flagged for their unusually high blocking rate. Some games with sexual content discussed in the previous part are not clearly presented in this figure since their sizes of them are too small to observe. Furthermore, notice that the most played "game" by blocked streamers is "Just Chatting" which is technically not a true game but a rather broad category intended for casual chatting with viewers. Due to its comparatively unrestricted content compared to other specific games, misconduct and even violation such as sexual harassment are more likely to happen during "chatting". Therefore, it is not strange that "Just Chatting" is most popular for blocked streamers.

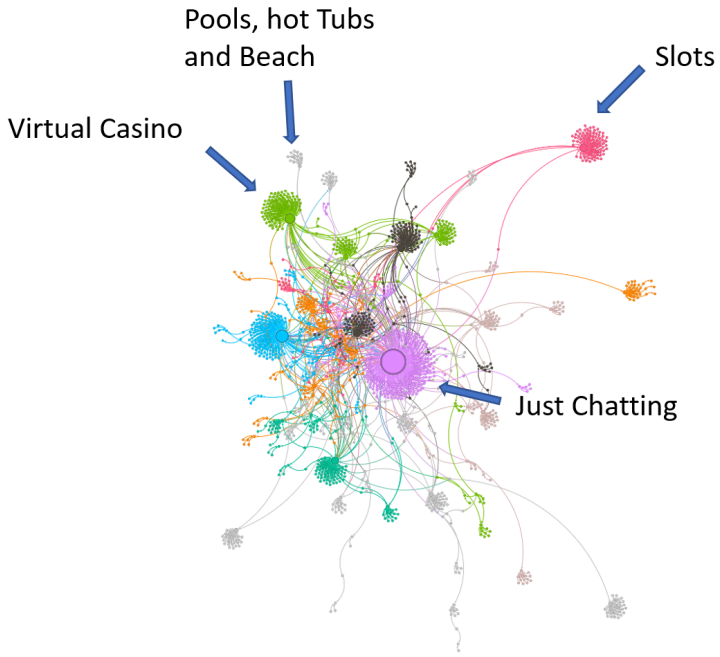


Figure 4.8: Visualization of Blocked Streamers With Games

4.5 Conclusion

We mainly conduct community mining and visualization on both regular streamers and blocked streamers in terms of the tags and games they play. From network analysis on all streamers, not many valuable clues regarding the underlying structure of data are revealed since the data size is too large.

Judging from the network analysis on blocked streamers, a major potential cause of streamers being blocked seems to be the appearance of gambling or sexual components appearing in their live streaming which should be strictly regulated under Twitch's community guidelines. In addition, streamers under "Just Chatting" channel also exhibit a higher probability of being blocked due to multiple causes.

Overall, stricter content moderation should be implemented especially for preventing illegal gambling activities and inappropriate sexual content in order to make the live-streaming platform safer and more enjoyable.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[2] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[3] Kanish Shah, Henil Patel, Devanshi Sanghvi, and Manan Shah. A comparative analysis of logistic regression, random forest and knn models for the text classification. *Augmented Human Research*, 5(1), 2020.