PH125.9x - HarvardX Capstone Project : Immigration in the European Union

HM

9/24/2020

Contents

Introduction

Description of the Dataset

The dataset we are going to study is the *First instance decisions on applications by citizenship, age and sex - annual aggregated data* dataset available on the Eurostat website. It consists of about one and a half million entries. We divide the data into two datasets: one for training and one for test, each of them containing the following features:

Variable	Description (Type)
`citizen`	Country of departure
`sex`	Sex category
`age`	Age of asylum applicants
`decision`	Decision taken by administrative bodies
`geo`	Country of arrival
`time`	Number of arrivals per year

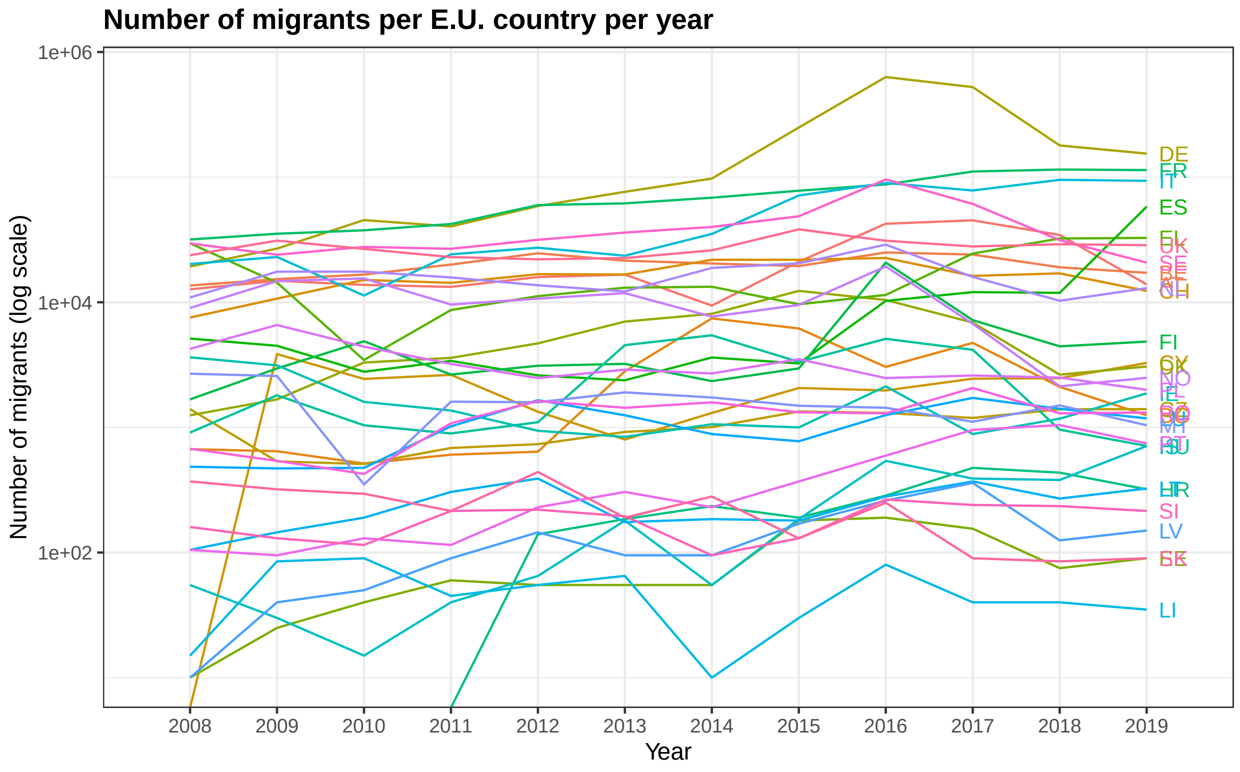
Lets visualize the first few rows of the training set `training` to get an idea of what we are dealing with:

	citizen	sex	age	decision	geo	2019	2018	2011	2010	2009	2008
1	AD	F	TOTAL	GENCONV	AT	0	0	0	0	0	0
2	AD	F	TOTAL	GENCONV	BE	0	0	0	0	0	0
3	AD	F	TOTAL	GENCONV	BG	0	0	0	0	0	0
4	AD	F	TOTAL	GENCONV	CH	0	0	0	0	0	0
5	AD	F	TOTAL	GENCONV	CY	0	0	0	0	0	0
7	AD	F	TOTAL	GENCONV	DE	0	0	0	0	0	0

There are 1429759 items in the training dataset `training` and 158863 items in the test dataset `validation`. In total, the full dataset contains 1588622 items. We choose this 90/10 ratio between the training and validation datasets as they contain many entries.

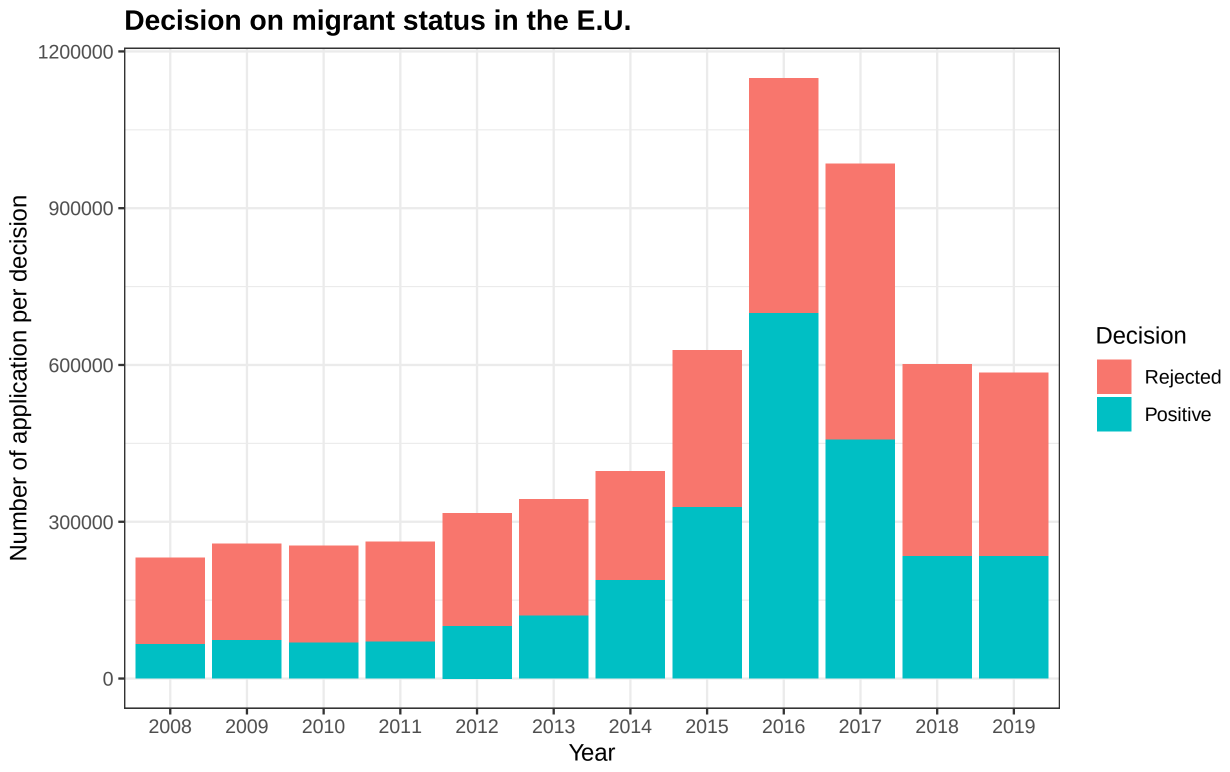
Data Analysis

There has been a recent surge in the number of migrants arriving in Europe. This dataset contains information on where migrants come from, where they are going, their age, sex and whether their application has been successful or not.



We notice that some countries have welcomed many more migrants than others. Furthermore, each immigrant must apply for a refugee or subsidiary protection status in the country of arrival.

This chart shows a sharp increase in the number of application in 2016 and 2017 with a stabilization the next two years.



Objective

Our goal is to correctly predict the number of migrants for each category mentioned above. This could be useful for immigration policies. We will be testing our methods on the `validation` dataset and report the RMSE (root-mean-square deviation) for each of the methods used.

$$RMSE = \sqrt{\frac{1}{N}\sum(y - \hat{y})^2}$$

With y being the predicted value, $\hat{y}$ being the corresponding expected value in the `validation` dataset, and N represents the number of items in total.

We will try to minimize the **RMSE**.

Methods

In order to accurately predict the number of migrants we will be conducting our analysis using different approaches. We will start with a basic prediction using only the overall mean, from there we will take into account the country of departure and the country of arrival. Finally we will implement a neural network approach to better take into account all the variables using the *tensorflow* library.

First Approach : overall mean

For this part we will suppose that every category accounts for the same number of migrants. We calculate the mean for every category between 2008 and 2018 in the `training` dataset

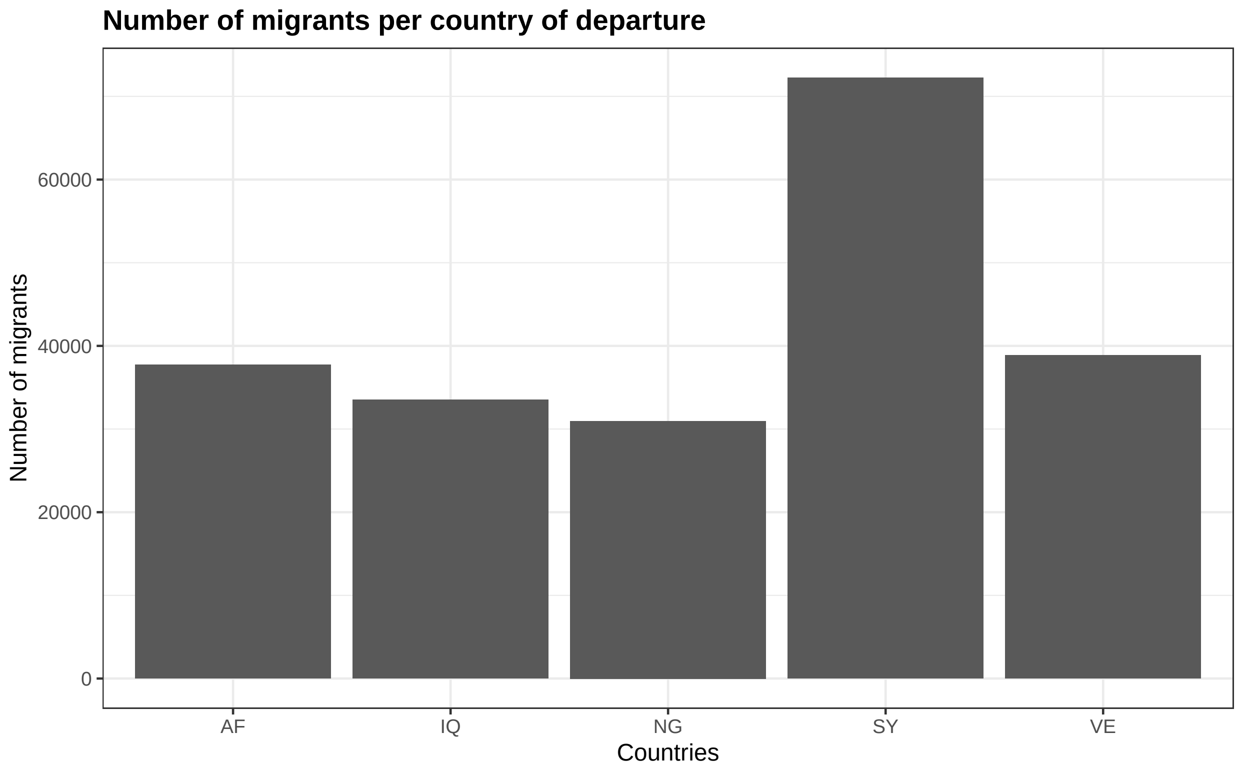
```
mu <- mean(colMeans(training[, 7:17]))
```

This very simple approach doesn't take into account the fact that some countries account for more migrants than others and that some E.U. countries tend to welcome more people. We will need to take into account those factors.

Second approach : country of departure and country of arrival

In this part we will study the impact of the countries of departure and arrival on the number of migrants.

We know from experience that migrants tend to come from a selected number of countries. Indeed, the data confirms this fact:



The top five countries for migrants to come from are : Syria, Venezuela, Afghanistan, Iraq and Nigeria.

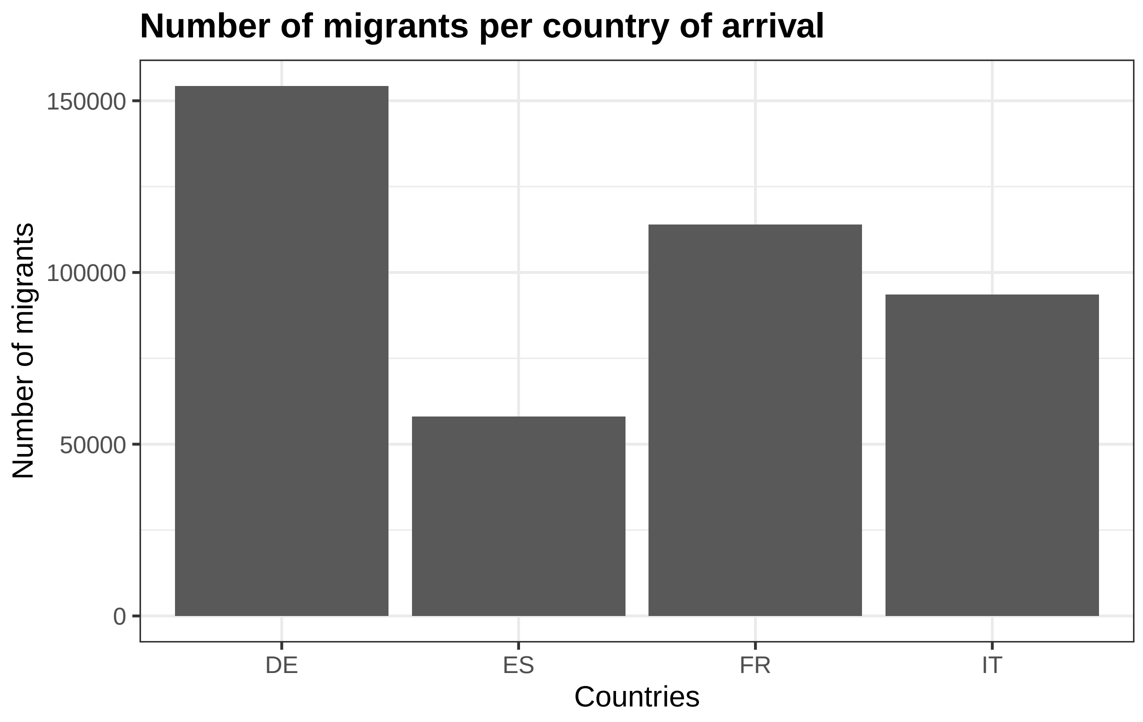
Using the following code we calculate the averages per country of departure :

```r
# Calculate mean per country per year
mean_per_category_peryear_percountry <- aggregate(training[,7:17],
                                                  list(training$citizen),
                                                  mean)

mean_percategory_percountry <- data.frame(
  citizen = mean_per_category_peryear_percountry[,1],
  meanc = rowMeans(mean_per_category_peryear_percountry[,-1])
  )

# Calculate mean per category
training_mean <- training %>%
  mutate(overmean = rowMeans(training[,7:17]))

# Delete all other columns
training_mean[6:17] <- list(NULL)

# Calculate bias per country of departure
cit_mean <- training_mean %>%
  group_by(citizen) %>%
  summarize(b_c = mean(overmean - mu))
```

After that we can compute the predicted number of migrants in the `validation` dataset.

```r
# Compute prediction
predicted_number <- mu + validation %>%
  left_join(cit_mean, by = "citizen") %>%
  pull(b_c)
```

However, this is still an imperfect model as the country of arrival is an important factor as seen in this chart:



We can run the same code as above with a few tweaks to include the impact of the country of arrival and calculated the predicted ratings:

```r
# Calculate mean per destination
mean_per_category_peryear_perdestination <- aggregate(training[,7:17],
                                                      list(training$geo),
                                                      mean)

mean_percategory_perdestination <- data.frame(
  geo = mean_per_category_peryear_perdestination[,1],
  meang = rowMeans(mean_per_category_peryear_perdestination[,-1])
  )

geo_mean <- training_mean %>%
  left_join(cit_mean, by = "citizen") %>%
  group_by(geo) %>%
  summarize(b_d = mean(overmean - mu - b_c))


# Compute prediction
predicted_number <- validation %>%
  left_join(geo_mean, by = "geo") %>%
  left_join(cit_mean, by = "citizen") %>%
  mutate(pred = mu + b_c + b_d) %>%
  pull(pred)
```

The result is a vector containing the predicted number of migrants in the `validation` dataset taking into account the country of departure and the country of arrival of migrants.

As the number of migrants can't be negative we will be imposing a limit to the predicted values:

```r
predicted_number_limit <- pmax(predicted_number, 0)
```

This method still doesn't take into account all of the labels, and implementing a linear regression for all of them would be too time consuming.

Third Approach: neural-network approach

For this final approach we will be using the *tensorflow* library to develop a neural-network in order to better predict the number of migrants arriving. Unfortunately, implementing this solution in R causes RStudio to crash so we will be implementing this solution using Python.

Lets start by importing the necessary libraries:

```python
##################################################
## IMPORT LIBRARIES
##################################################

import pathlib

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from pathlib import Path
print("Working directory", Path().absolute())
```

```
import tensorflow as tf


from tensorflow import keras
from tensorflow.keras import layers

print("Tensorflow version : ", tf.__version__)

import tensorflow_docs as tfdocs
import tensorflow_docs.plots
import tensorflow_docs.modeling
```

We then need to clean the data, the same way we did using R but this time we separate the labels and yearly data.

```
#################################################
## IMPORT AND CLEAN DATA
#################################################

##Import data table in tsv format
dl = pd.read_csv("migr_asydcfsta.tsv", sep='\t')

##Clean labels with csv encoding
dl_temp1 = dl[dl.columns[0]]
dl_temp1 = dl_temp1.str.split(",", expand = True)
dl_temp1.columns = ["unit", "citizen", "sex" , "age", "decision", "geo"]
dl_temp1 = dl_temp1.drop('unit', 1)

##Clean year number labels
dl_temp2 = dl[dl.columns[1:13]]
dl_temp2.columns = ["X2019", "X2018", "X2017", "X2016",
                    "X2015", "X2014", "X2013", "X2012",
                    "X2011", "X2010", "X2009", "X2008"]
dl_temp2 = dl_temp2.replace(":", "0", regex = True)
dl_temp2 = dl_temp2.replace(to_replace = {" b", " e", " p", " u", " z", " d", " o", " bd",
                                          " bde", " bdep", " bdp", " bdu", " bduo", " be",
                                          " bep", " bp", " bu", " buo", " bz"," de", " dep",
                                          " dp", " du", " duo"," ep","p u", " puo"
                                          },
                            value = "",
                            regex = True
                            )
dl_temp2 = dl_temp2.replace(" ", "", regex = True)
dl_temp2 = dl_temp2.astype(int)
```

The columns *citizen*, *sex* , *age*, *decision* and *geo* are categorical, not numerical. So we convert those to hot-ones. Using the following code:

```
####################
## ONE-HOT ENCODING
####################

##Encode each label
one_hot_geo = pd.get_dummies(dl_temp1["geo"], prefix = "geo")
```

```
one_hot_cit = pd.get_dummies(dl_temp1["citizen"], prefix = "cit")
one_hot_sex = pd.get_dummies(dl_temp1["sex"], prefix = "sex")
one_hot_dec = pd.get_dummies(dl_temp1["decision"], prefix = "dec")
one_hot_age = pd.get_dummies(dl_temp1["age"], prefix = "age")


##Join the arrays
one_hot_temp1 = one_hot_geo.join(one_hot_cit)
one_hot_temp2 = one_hot_temp1.join(one_hot_sex)
one_hot_temp3 = one_hot_temp2.join(one_hot_dec)
one_hot_temp4 = one_hot_temp3.join(one_hot_age)


dl_temp3 = one_hot_temp3.join(dl_temp2)
```

Now that the data is in the correct format we divide it into a train and a test dataset.

```
##################################################
## PREPARE DATA
##################################################

## Divide train/test
dataset = dl_temp3
print(dataset)

train_dataset = dataset.sample(frac=0.8,random_state=0)
test_dataset = dataset.drop(train_dataset.index)
```

Once this is done we can split the features from the labels.

```
## Split features from labels
train_labels = train_dataset.pop('X2019')
test_labels = test_dataset.pop('X2019')
```

We notice that the range of the features are very different. It is good practice to normalize the features that use different scales and ranges.

```
## Data stats
train_stats = train_dataset.describe()
train_stats = train_stats.transpose()
print(train_stats)

## Normalize the data
def norm(x):
  return (x - train_stats['mean']) / train_stats['std']
normed_train_data = norm(train_dataset)
normed_train_data.drop('X2019', axis=1, inplace=True)
normed_test_data = norm(test_dataset)
normed_test_data.drop('X2019', axis=1, inplace=True)
```

We will use this normalized data to train the model. We will use a *Sequential* model with two densely connected layers and one output layer.

```
##################################################
## NEURAL NETWORK APPROACH
##################################################

## Build the model
def build_model():
  model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=[len(train_dataset.keys())]),
    layers.Dense(64, activation='relu'),
    layers.Dense(1)
  ])

  optimizer = tf.keras.optimizers.RMSprop(0.001)

  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae', 'mse'])
  return model

model = build_model()
```

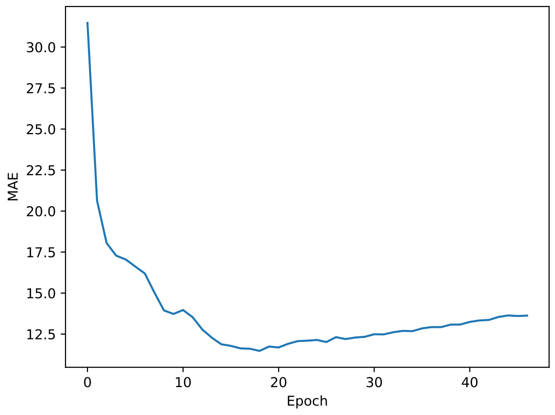
Once the model has been created we can view a description of it:

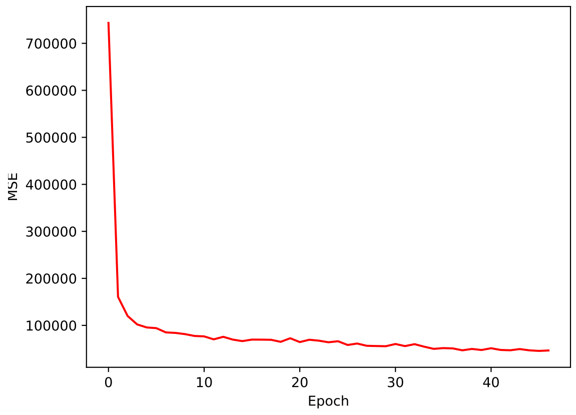
```
## Inspect the model
model.summary()

# Model: "sequential"
# _________________________________________________________________
# Layer (type)                 Output Shape              Param #
# =================================================================
# dense (Dense)                (None, 64)                17024
# _________________________________________________________________
# dense_1 (Dense)              (None, 64)                4160
# _________________________________________________________________
# dense_2 (Dense)              (None, 1)                 65
# =================================================================
# Total params: 21,249
# Trainable params: 21,249
# Non-trainable params: 0
# _________________________________________________________________
#
# Epoch: 0, loss:891718.3750,  mae:31.7619,  mse:891718.3750,  val_loss:173475.3594,
# val_mae:25.2753,  val_mse:173475.3594,
# ......................................................................
```

We train the model with 100 epochs but we use an *EarlyStopping callback* to stop the training when the model's validation error doesn't improve.

```
## Train the model
EPOCHS = 100

early_stop = keras.callbacks.EarlyStopping(monitor='val_loss',patience=10)

history = model.fit(normed_train_data,
                    train_labels,
                    epochs = EPOCHS,
                    validation_split = 0.2,
                    verbose = 0,
                    callbacks = [early_stop,
                                 tfdocs.modeling.EpochDots()]
                    )
```

We can visualize the model's training progress by plotting the different statistics stored in the *history* object.



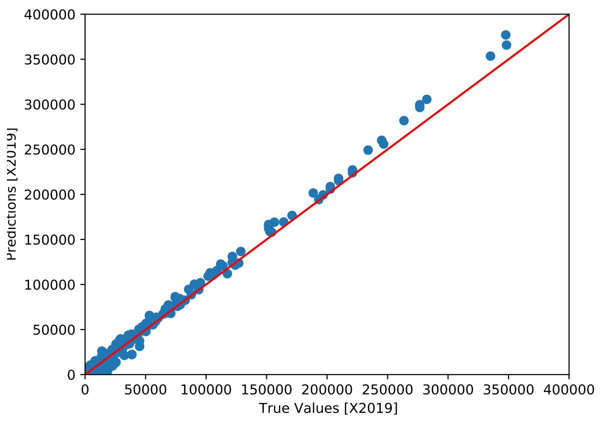


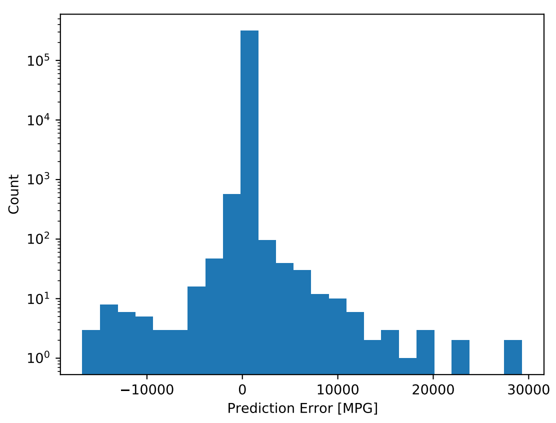
These plots show us that we differ from the correct value by 15, meaning that we are wrong by about 15 people.

Finally, we can compute the predicted values using the testing dataset:

```
test_predictions = model.predict(normed_test_data).flatten()
```

This method should yield better results as it takes into account every label. Furthermore, by plotting the predicted value against the true values we get a good idea of the acuracy of the model.





Results

We have presented the different methods we are going to evaluate. Each method will be used to predict the ratings in the validation dataset and we will compute the RMSE using this code in R:

```r
RMSE <- function(true_ratings, predicted_ratings){
  sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

And this one in Python:

```python
def rmse(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())
```

RMSE First Approach

This is the most basic estimate so we are not expecting to be very accurate. We get the following RMSE:

```r
RMSE_overallmean <- RMSE(validation[, 6], mu)
```

$$RMSE_{overallmean} = 2553$$

RMSE Second Approach

By computing the impact of the countries of departure and arrival we should be able to improve our RMSE. We get the following:

```r
# Compute RMSE
rmse_efcda <- RMSE(predicted_number, validation[, 6])

rmse_efcda_limit <- RMSE(predicted_number_limit, validation[, 6])
```

$$RMSE_{efcda} = 2526$$

$$RMSE_{efcdalimit} = 2522$$

RMSE Neural-Network

Using the Ml algorithm and the *tensorflow* library we get the following result :

```python
rmse_tensorflow = rmse(test_predictions, test_labels))
```

$$RMSE_{tensorflow} = 173$$

Conclusion

Summary

To conclude, here's a brief recap of our results:

Method	Results
Overall Mean	2553.39
Effect countries of departure/arrival	2522.63
Effect countries of departure/arrival with limit	2522.334
ML Using *tensorflow* library	173

Our final best RMSE is **173**, obtained with the tensorflow library using a neural-network algorithm. Furthermore, as seen above our mean absolute value is of 15 which means we differ from the correct value by 15 people on average which is sufficient.

Further Work

The Random Forest algorithm could be applied here but due to the size of the dataset and its complexity it would require too much computational power.

Another approach to improve our result would be to add more labels to our dataset. Adding information about the economic and political situation of the countries of departure could be interesting.