M3: Group Assignment

This assignment is based on a dataset from AirBnB in Singapore, the dataset was updated on 28 August 2019. There are 7907 samples.

We have made a problem statement, which will guide the analyses of our assignment. The problem statement is:

Does the accommodation features and titles have influence on the prices?

To answer the above problem statement, we will use Machine Learning techniques on six different features, which might have an influence on the prices to see how precise, we can predict whether the accommodation has a price above or below the average price. We will also use Deep Learning techniques on the accommodation titles to see how precise these predictions are compared to the actual prices and the first predictions.

Based on our prior knowledge, we were expecting that our deep learning models will be more accurate than the machine learning model. The reason for this assumption is, that we believe the title of the accommodations as a feature in the dataset contains more relevant information for the traveller than for example the availability and number of reviews. We believe this as this can vary in many different ways and as the reviews often are not present in the add. Yet, it is interesting to figure out whether or not this assumption is true.

To begin this assignment, we import some releveant libraries that we will use later on in the assignment. The rest of the used libraries will be imported through the assignemnt

In [0]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
```

In [2]:

```
from google_drive_downloader import GoogleDriveDownloader as gdd

gdd.download_file_from_google_drive(file_id='1A3xtXHco2uMClwIOib09wsWZ5BWCXh6g',
                                    dest_path='./data/my_output.file',
                                    unzip=False)
```

```
Downloading 1A3xtXHco2uMClwIOib09wsWZ5BWCXh6g into ./data/my_output.
file... Done.
```

In [0]:

```
dataset = pd.read_csv('./data/my_output.file')
```

In [0]:

```
dataset.name=dataset.name.astype(str)
```

Below, we use the .head() function to get an overview of your dataset. In the table, we can see a list of information about all of the different houses at this Airbnb site.

In [5]:

```
dataset.head()
```

Out[5]:

	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude
0	49091	COZICOMFORT LONG TERM STAY ROOM 2	266763	Francesca	North Region	Woodlands	1.44255
1	50646	Pleasant Room along Bukit Timah	227796	Sujatha	Central Region	Bukit Timah	1.33235
2	56334	COZICOMFORT	266763	Francesca	North Region	Woodlands	1.44246
3	71609	Ensuite Room (Room 1 & 2) near EXPO	367042	Belinda	East Region	Tampines	1.34541
4	71896	B&B Room 1 near Airport & EXPO	367042	Belinda	East Region	Tampines	1.34567

Using the .info(), we can see that we in the categories last_review and reviews_per_month, there are a lot of missing values. Therefore, we also use the .isnull().sum() to see the number of values with a value of zero. This gives us a better understanding of how many NaN values out dataset consists of.

In [6]:

```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7907 entries, 0 to 7906
Data columns (total 16 columns):
id                                7907 non-null int64
name                              7907 non-null object
host_id                           7907 non-null int64
host_name                         7907 non-null object
neighbourhood_group               7907 non-null object
neighbourhood                     7907 non-null object
latitude                          7907 non-null float64
longitude                         7907 non-null float64
room_type                         7907 non-null object
price                             7907 non-null int64
minimum_nights                    7907 non-null int64
number_of_reviews                 7907 non-null int64
last_review                       5149 non-null object
reviews_per_month                 5149 non-null float64
calculated_host_listings_count    7907 non-null int64
availability_365                  7907 non-null int64
dtypes: float64(3), int64(7), object(6)
memory usage: 988.5+ KB
```

In [7]:

```
dataset.isnull().sum()
```

Out[7]:

```
id                                    0
name                                  0
host_id                               0
host_name                             0
neighbourhood_group                   0
neighbourhood                         0
latitude                              0
longitude                             0
room_type                             0
price                                 0
minimum_nights                        0
number_of_reviews                     0
last_review                        2758
reviews_per_month                  2758
calculated_host_listings_count        0
availability_365                      0
dtype: int64
```

In the following lines of code, we create an array with our chosen features, as we will need the features in that format later in the assignment. For our target variable, we have chosen to use 'Price' and the feature variables consist of 'latitude','longitude','number_of_reviews','minimum_nights', 'availability_365', 'calculated_host_listings_count'.

We have chosen these variables as we would like to have a closer look at their connection to price.

In [0]:

```
# We create an array with the wanted feature for our model
data_features = ['latitude','longitude','number_of_reviews','minimum_nights', 'availability_365', 'calculated_host_listings_count']
```

Afterwards, we use the StandardScaler() function to scale our data. Scaling our data, makes it possible to compare the different chosen features. If one were to chose not to scale the data, there would be no guarantee that a comparison of the features would make sense or even be possible.

In [9]:

```
scaler = StandardScaler()
dataset[data_features]=scaler.fit_transform(dataset[data_features])
dataset.describe()
```

Out[9]:

	id	host_id	latitude	longitude	price	minimum_nig
count	7.907000e+03	7.907000e+03	7.907000e+03	7.907000e+03	7907.000000	7.907000e+
mean	2.338862e+07	9.114481e+07	-4.959806e-15	1.662078e-13	169.332996	1.135581e
std	1.016416e+07	8.190910e+07	1.000063e+00	1.000063e+00	340.187599	1.000063e+
min	4.909100e+04	2.366600e+04	-2.299960e+00	-4.630610e+00	0.000000	-3.922378e
25%	1.582180e+07	2.305808e+07	-6.017059e-01	-2.968147e-01	65.000000	-3.922378e
50%	2.470627e+07	6.344891e+07	-1.034313e-01	1.425499e-02	124.000000	-3.447228e
75%	3.234850e+07	1.553811e+08	2.589502e-01	5.437719e-01	199.000000	-1.784202e
max	3.811276e+07	2.885676e+08	4.591830e+00	2.853840e+00	10000.000000	2.334151e+

By using the .describe() function above, we can see the mean value of the prices for the houses in the Airbnb dataset. This mean value will be essential later on in the assignment, as we will categorise the houses in ‘price above mean’ and ‘price below mean’, respectively.

Exploratory data analysis

Below, we explore our dataset a little bit more. We look at the names of the most expensive houses, the names of the cheapest houses as well as find the minimum and maximum prices of the houses.

In [10]:

```
# Groupby 'name' and 'price'

dataset.groupby('name').price.mean().sort_values(ascending=False).index[:2]
```

Out[10]:

```
Index(['Testing', 'Comfortable & Quiet Master Bedroom'], dtype='obje
ct', name='name')
```

In [11]:

```
# Groupby 'name' and 'price'

dataset.groupby('name').price.mean().sort_values(ascending=True).index[:2]
```

Out[11]:

```
Index(['1 BR @ Little India & Farrer Park MRT', 'Central 1BR Apt in
Foodie Haven Hipster Paradise'], dtype='object', name='name')
```

Above results show which rooms/apartment are rented to respectively the lowest and highest price

Now we want to look at what the minimum and maximum price are to get a better understanding of our dataset.

In [12]:

```
dataset['price'].min()
```

Out[12]:

```
0
```

In [13]:

```
dataset['price'].max()
```

Out[13]:

```
10000
```

Linear Regression

For the last part of the explorative analysis, we create a linear regression for each of the features in comparison with the prices. A Linear regression is used for finding linear relationships between output and one or more inputs. With the linear regression we can get a more detailed understanding of the relationship between our variables.

In [0]:

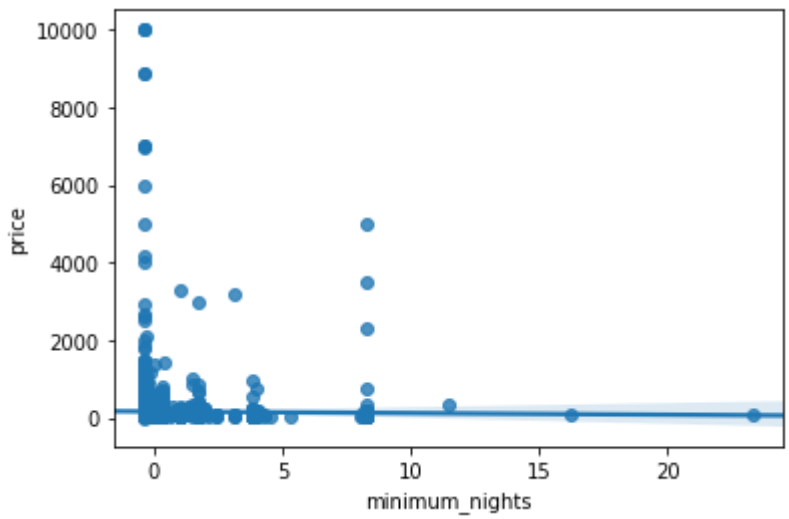
```
df=dataset[data_features]
```

In [15]:

```
sns.regplot(x='minimum_nights', y='price', data=dataset)
```

Out[15]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1be0588c50>
```

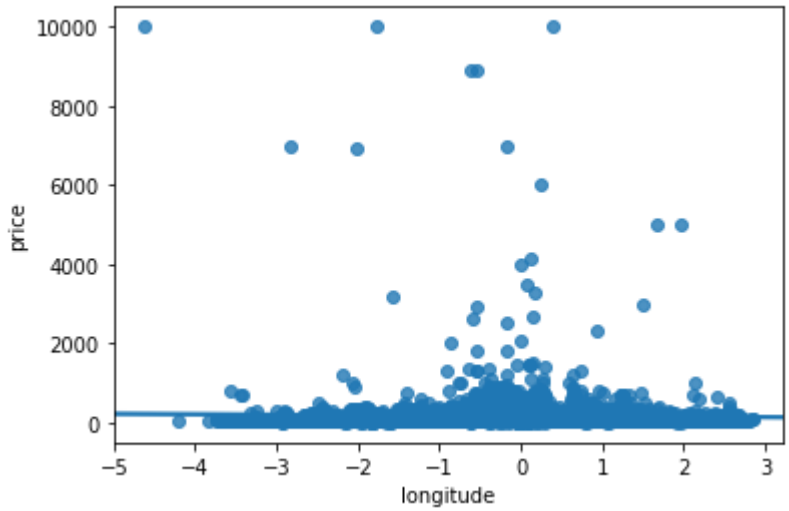


In [16]:

```
sns.regplot(x='longitude', y='price', data=dataset)
```

Out[16]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1be0525c50>
```

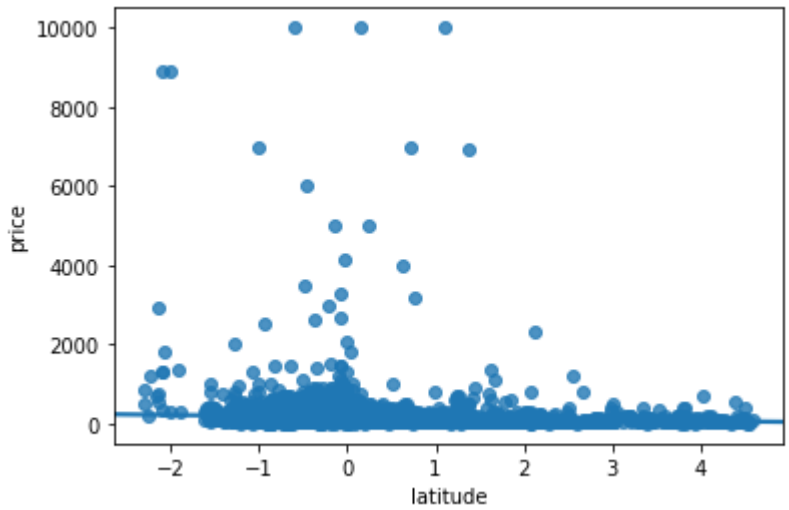


In [17]:

```
sns.regplot(x='latitude', y='price', data=dataset)
```

Out[17]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1bdd84be10>
```

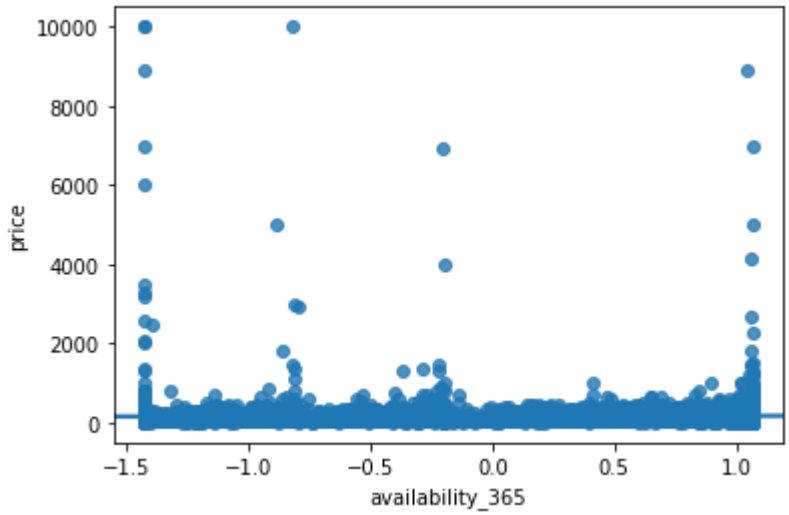


In [18]:

```
sns.regplot(x='availability_365', y='price', data=dataset)
```

Out[18]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1bdd7d58d0>
```

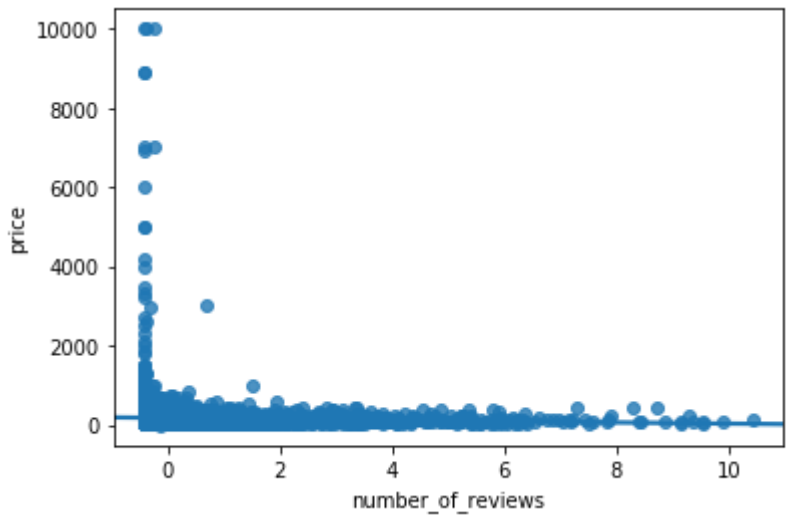


In [19]:

```
sns.regplot(x='number_of_reviews', y='price', data=dataset)
```

Out[19]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1bdd74eb70>
```



In the above linear regressions, we can interpret that the features doesnt have an big inpact on the price of the accomodation. As an example the regression with the number of reviews hasnt a big impact on the price. AS we can se in the plot there are a lot of expensive accomodations with no reviews. There could be some potential outliers here, it seems wierd that some of the accomodations costs 10000 singapore dollars, which is approx. 50.000 kroners.

Supervised Machine Learning

Supervised learning is where you have several input variables and an output variable and you use an algorithm to learn the mapping function from the input to the output. The goal is to approximate the mapping function so well that when you have a new input data that you can predict the output variables for that data.

For the next part of the assignment, we will try to predict whether the price for renting a hourse is above or below the mean value, which we in the EDA part found out was 169. Before creating our model, we import the libraries which we will need and define our X and y value. For this assignment, our X value is df and our y value is the 'Price'.

Defining X and y

In [0]:

```
# Creating a x

X = df
```

In [0]:

```
y = dataset['price']
```

Below, we have used an if statement to divide our data into the categories 'True' and 'False'. The if statement says that if the price(x) is less than 169, which is the mean value, the house should be in the category 'True' and else it should be 'False'.

In [0]:

```
mean_price = [True if x<169 else False for x in y ]
```

Below, you can see that instead of the actual numbers, the different variables are now showing wethter the variable is true or false according to the if statement above.

In [23]:

```
mean_price[0:10]
```

Out[23]:

```
[True, True, True, False, True, True, False, True, True, True]
```

As presented in list of above, only two out of ten variables are False, indicating that 20 % of the variables are higher than the mean value. This is of course only based on a really small samples, and is therefore, not statistically adequate. however, it presents a draft picture of the divided data.

Machine Learning models

To start of creating our machine learning model, we create the y that we will need. Thereafter, we import the libraries that we will need to perform our chosen baseline model. For this assignemnt, we have chosen to work with the decision tree model.

In [0]:

```
y = mean_price
```

After defining our X and y value, we divide our dataset into test set and a training set. The test set will be 25 % of the total data and the training set will be 75 % of the total data. Splitting the data this way, creates a risk in regards to overfitting the data. This can fx happen if the split for some reason is not random.

In [0]:

```
# Importing necessary library

from sklearn.model_selection import train_test_split
```

In [0]:

```
# Splitting the data by using train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.25, rando
m_state = 42)
```

In [0]:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV
```

In [0]:

```
from sklearn.metrics import classification_report
```

Below, we start of by creating the model and saving it as 'tree'. Thereafter, we fit the model on our training data. In order to predict wether the rent cost is above or below average, we use the .predict() function on our X_test.

In [0]:

```
tree = DecisionTreeClassifier()
```

In [0]:

```
legendary_tree = tree.fit(X_train, y_train)
```

In [0]:

```
legen_tree = legendary_tree.predict(X_test)
```

To evaluate our results, we print out our classification report.

In [32]:

```
print(classification_report(y_test, legen_tree))
```

```
              precision    recall  f1-score   support

       False       0.60      0.61      0.60       592
        True       0.83      0.82      0.83      1385

    accuracy                           0.76      1977
   macro avg       0.71      0.72      0.71      1977
weighted avg       0.76      0.76      0.76      1977
```

The recall rate of False/ True is the percentage of Accommodations correctly classified as above /below average from the. This is fairly low with 60/82%. The model is much better at predicting the prices below average (True) than the prices above average (False).

Precision tells how precise the model is at predicting the specific category based on the positive results. Our model is more precise in predicting True (83%) than False (58%).

The f1-score is the harmonic mean of precision and recall, and more or less summarize the two.

The difference in how good the model is at prediction True/False might be because the True category is overrepresented with 5383 accommodations and False is underrepresented with only 2524 accommodations.

Accuracy is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. It is this accuracy of 75% that we will use to compare our baseline model with our future deep learning model.

Deep Learning

As our results in the above Machine Learning model were fine, but for this next section, we will use Deep Learning models to see if we can get a more accurate prediction.

Firstly, we start with importing the necessary libraries for this part of the assignment. For the Deep Learning part of this assignemt we will use Keras which has been the most popular package for developers to use when working within the field of Deep Learning since the release in 2015. The reason for this is that Keras allows easy and fast prototyping (through user friendliness, modularity, and extensibility). Moreover, Keras supports both convolutional networks and recurrent networks, as well as combinations of the two.

In [0]:

```
pd.set_option('display.max_colwidth', 0)

from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing import sequence

from keras.models import Sequential

from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, LSTM, Dropout, Ac
tivation, Input, CuDNNLSTM, Embedding, SpatialDropout1D, GRU
from keras.layers.normalization import BatchNormalization

from collections import Counter
from sklearn.feature_extraction import DictVectorizer
```

In the following, we will use the tokenizer function as well as padding X. The tokenizer function turns strings in python into smaller tokens. The purpose of doing this, for this assignment, is that we wish to turn the names into smaller indices. Afterwards, we have chosen to set the max length to 120 as well as using the pad function. By using the pad function, we turn our list into a 2D numpy array and use the placeholder values to make sure all rows have the same length. This creates an easier workflow for us.

In [0]:

```
vocabulary_size = 5000
tokenizer = Tokenizer(num_words = vocabulary_size)
tokenizer.fit_on_texts(dataset['name'])
sequences = tokenizer.texts_to_sequences(dataset['name'])
```

In [35]:

```
MAXLEN = 120
X = pad_sequences(sequences, maxlen=MAXLEN)
X
```

Out[35]:

```
array([[   0,    0,    0, ...,   57,    1,   10],
       [   0,    0,    0, ...,  501,  293,  444],
       [   0,    0,    0, ...,    0,    0, 1307],
       ...,
       [   0,    0,    0, ...,   30,    7,    2],
       [   0,    0,    0, ...,   23,   36,   37],
       [   0,    0,    0, ...,   31,    7,   14]], dtype=int32)
```

Below, we create an index, which will be used in the train_test_split.

In [0]:

```
indices = range(len(X))
```

Below, we prepare for our data for the Deep Learning models by importing the neccessary library and dividing the data into a train set and a test set.

In [0]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test, index_train, index_test = train_test_split(X,
y, indices,  test_size=0.2)
```

Deep Learning Models

In this assignment, the overall architecture is Convolutional Neural Network (CNN). Then the LSTM, CuDNNLSTM and GRU models have been used to investigate the problem statement. The reason for using three different models are to get an overview of, which model predicts the best result.

LSTM

In the following we will create the model we will use to train the trainset, combining LSTM and Embedding Layers.

LSTM is short for Long short-term memory and is very useful, since it is cabable of learning long-term dependencies, which means that by using this model, it is able to remember information for a long period of time. This is very useful given that when working with neural networks, the model often forgets the first inputs from long sequences. Using this kind of model, allows for it to keep remembering, so if you are trying to process or predict on long paragraphs, it is important that the model remembers the first part of the input.

The embedding layers in the model can be used on text data in neural networks, which makes it possible to add words to the training data.

Lastly, we add BatchNormalization to our model to improve the speed, performance and stability of our neural network model as well as to normalize the input layers by scaling the activations.

In [38]:

```
embedding_vecor_length = 300

model = Sequential()
model.add(Embedding(vocabulary_size, embedding_vecor_length, input_length=MAXLEN
))
model.add(SpatialDropout1D(0.1))
model.add(LSTM(64))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(Dense(1, activation='sigmoid'))
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:66: The name tf.get_default_graph is
deprecated. Please use tf.compat.v1.get_default_graph instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:541: The name tf.placeholder is depr
ecated. Please use tf.compat.v1.placeholder instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:4432: The name tf.random_uniform is
deprecated. Please use tf.random.uniform instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:148: The name tf.placeholder_with_de
fault is deprecated. Please use tf.compat.v1.placeholder_with_defaul
t instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:3733: calling dropout (from tensorfl
ow.python.ops.nn_ops) with keep_prob is deprecated and will be remov
ed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rat
e = 1 - keep_prob`.
```

After creating the model with LSTM and Embedding layers, we compile the model using binary_crosentropy as loss, adam as optimizer and accuracy as metrics.

Binary Crossentropy Binary crossentropy is a loss function used on problems involving yes/no (binary) decisions. Moreover, binary crossentropy measures how far away from the true value (which is either 0 or 1) the prediction is. Therefore, it is a relevant parameter for our assignment.

Metrics A metric is a function that is used to judge the performance of your model. Metric functions are to be supplied in the metrics parameter when a model is compiled. In our example we are using the metric “accuracy”.

Optimizer Optimizing is, as its name implies, a way to optimize one’s model. The purpose is to obtain the best possible result by maximising and minimizing based in different criteria. For this assignment, we have chosen to use the optimizer ‘adam’, which is an algorithm that has been designed specifically for for training deep neural networks.

In [39]:

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'
])
print(model.summary())
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/optimizers.py:793: The name tf.train.Optimizer is deprecated. Plea
se use tf.compat.v1.train.Optimizer instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:3657: The name tf.log is deprecated.
Please use tf.math.log instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tenso
rflow_core/python/ops/nn_impl.py:183: where (from tensorflow.python.
ops.array_ops) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 120, 300)          1500000
_________________________________________________________________
spatial_dropout1d_1 (Spatial (None, 120, 300)          0
_________________________________________________________________
lstm_1 (LSTM)                (None, 64)                93440
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
batch_normalization_1 (Batch (None, 64)                256
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 1,593,761
Trainable params: 1,593,633
Non-trainable params: 128
_________________________________________________________________
None
```

In the above table, using summary function, we can see the different layers, their output shape, params for each layer and the total number of params. Params is an indication of (..) parameters constitute the learnt values (weights) obtained while optimizing the loss function.

In fitting the model, we use 1 epoch and a batch size of 64.

Epoch is the process in which the entire dataset is passed forward and backward through the neural network only once. Therefore, one can add more epochs to the model, to make the model pass the entire dataset forward and backward more times. Since one epoch is too big to feed to the computer at once, we divide it in several smaller ones.

batches by using batch size, which divide the number of training examples in one forward/backward pass.

In [40]:

```
model.fit(X_train, y_train, epochs=1, batch_size=64, validation_split=0.1)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:1033: The name tf.assign_add is depr
ecated. Please use tf.compat.v1.assign_add instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:1020: The name tf.assign is deprecat
ed. Please use tf.compat.v1.assign instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:3005: The name tf.Session is depreca
ted. Please use tf.compat.v1.Session instead.

Train on 5692 samples, validate on 633 samples
Epoch 1/1
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:190: The name tf.get_default_session
is deprecated. Please use tf.compat.v1.get_default_session instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:197: The name tf.ConfigProto is depr
ecated. Please use tf.compat.v1.ConfigProto instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:207: The name tf.global_variables is
deprecated. Please use tf.compat.v1.global_variables instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:216: The name tf.is_variable_initial
ized is deprecated. Please use tf.compat.v1.is_variable_initialized
instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
s/backend/tensorflow_backend.py:223: The name tf.variables_initializ
er is deprecated. Please use tf.compat.v1.variables_initializer inst
ead.

5692/5692 [==============================] - 28s 5ms/step - loss: 0.
5103 - acc: 0.7456 - val_loss: 0.3632 - val_acc: 0.8325
```

Out[40]:

```
<keras.callbacks.History at 0x7f1bda1c3860>
```

After creating and fitting our model, we are now going to predict whether the name has an influence on the renting prices. To do this, we first start by defining our starting and ending point as 0 and 8000.

In [0]:

```
start = 0
end = 8000
```

Below, we define our y_pred as a predictiong of the X_test based on that start and end defined above.

Then we create a list with the True / False values regarding the predictions to be able to compare this newly created list with the real prices of the accommodations

Lastly, we define the real label for the comparison.

In [0]:

```
y_pred = model.predict(X_test[start:end])

y_pred = np.where(y_pred > 0.5, True, False)
y_pred = [list(x) for x in y_pred]

real_price = dataset['price'][index_test[start:end]]
```

Below, we present the predicted labels and the real labels, which gives a nice looking overview.

In [43]:

```
pd.DataFrame({'prediction':y_pred, 'real': real_price})
```

Out[43]:

	prediction	real
7751	[True]	39
5352	[True]	50
7108	[False]	275
2761	[True]	125
741	[True]	150
...	...	...
472	[True]	119
3123	[False]	300
2711	[False]	36
5429	[False]	289
7405	[True]	215

1582 rows × 2 columns

Earlier, we defined the price of the accomodations in a boolean variable "real_price" with the mean price being 169. If the price is below 169 the values are True and if the price is above 169 the value are False. Therefore, we can see if the model has predicedted correctly in the above dataframe. As an example 4717 is correct as the predicted is False "above 169" and the real is 239. 1763 is True "under 169" and the real is 82. In General the predictions look pretty good, however, we want to evaluate how good the model really is.

Below, we define a new variable, which is the evalutation of the test set. The result shows that the predictions are between 78.34 and 84.89 %.

In [44]:

```
results = model.evaluate(X_test, y_test)
print("The Loss and Accuracy:", results)
```

```
1582/1582 [==============================] - 5s 3ms/step
The Loss and Accuracy: [0.3854507148341192, 0.8356510747398348]
```

As can be seen above, the LSTM model predicts between 78-34%-84.89% in accuracy.

In the following, we will try to tune the model using CuDNNLSTM instead of LSTM and by using 10 epochs instead of 1.

CuDNNLSTM

CuDNNLSTM is very similar to LSTM, as it is a fast LSTM implementation backend by CuDNN, which is a CuDNN library. This means that it has the same attributes as previously mentioned such as being able to remember information for a longer period of time. Therefore, it is argued that CuDNNLSTM is faster than the “regular” LSTM.

Using more or less the same approach as with the LSTM model above, we create our new model.

In [0]:

```
embedding_vecor_length = 300
model = Sequential()
model.add(Embedding(vocabulary_size, embedding_vecor_length, input_length=MAXLEN
))
model.add(SpatialDropout1D(0.1))
model.add(CuDNNLSTM(64))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(Dense(1, activation='sigmoid'))
```

Below, we compile the new model using binary_crossentropy as loss, adam as optimizer and accuracy as metrics as before. Like with the LSTM model, we can see the different layers, their output shape, params for each layer as well as the total number of params.

In [46]:

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'
])
print(model.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 120, 300)          1500000
_________________________________________________________________
spatial_dropout1d_2 (Spatial (None, 120, 300)          0
_________________________________________________________________
cu_dnnlstm_1 (CuDNNLSTM)     (None, 64)                93696
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
batch_normalization_2 (Batch (None, 64)                256
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 1,594,017
Trainable params: 1,593,889
Non-trainable params: 128
_________________________________________________________________
None
```

Below, we fit the model from the training set using 10 epochs instead of 1, as we did in the LSTM, a batch size of 64 and a validation split on 0.1.

In [47]:

```
model.fit(X_train, y_train, epochs=10, batch_size=64, validation_split=0.1)
```

```
Train on 5692 samples, validate on 633 samples
Epoch 1/10
5692/5692 [==============================] - 6s 1ms/step - loss: 0.5
129 - acc: 0.7454 - val_loss: 0.4201 - val_acc: 0.8167
Epoch 2/10
5692/5692 [==============================] - 3s 450us/step - loss:
0.3181 - acc: 0.8733 - val_loss: 0.4075 - val_acc: 0.8420
Epoch 3/10
5692/5692 [==============================] - 3s 443us/step - loss:
0.2458 - acc: 0.9050 - val_loss: 0.4558 - val_acc: 0.8294
Epoch 4/10
5692/5692 [==============================] - 3s 442us/step - loss:
0.2000 - acc: 0.9201 - val_loss: 0.4927 - val_acc: 0.8262
Epoch 5/10
5692/5692 [==============================] - 2s 430us/step - loss:
0.1841 - acc: 0.9294 - val_loss: 0.5188 - val_acc: 0.8357
Epoch 6/10
5692/5692 [==============================] - 2s 428us/step - loss:
0.1662 - acc: 0.9336 - val_loss: 0.5201 - val_acc: 0.8310
Epoch 7/10
5692/5692 [==============================] - 2s 424us/step - loss:
0.1456 - acc: 0.9450 - val_loss: 0.5111 - val_acc: 0.8341
Epoch 8/10
5692/5692 [==============================] - 2s 422us/step - loss:
0.1350 - acc: 0.9478 - val_loss: 0.5484 - val_acc: 0.8310
Epoch 9/10
5692/5692 [==============================] - 2s 422us/step - loss:
0.1352 - acc: 0.9489 - val_loss: 0.5461 - val_acc: 0.8262
Epoch 10/10
5692/5692 [==============================] - 2s 428us/step - loss:
0.1269 - acc: 0.9508 - val_loss: 0.6010 - val_acc: 0.8183
```

Out[47]:

```
<keras.callbacks.History at 0x7f1b44430d68>
```

In [48]:

```
results = model.evaluate(X_test, y_test)
print("The Loss and Accuracy:", results)
```

```
1582/1582 [==============================] - 0s 233us/step
The Loss and Accuracy: [0.5681619959896343, 0.8362831859160615]
```

Looking at our results, we can see that the accuracy of our prediction is between 80.43 and 83.82 %.

The accuracy of LSTM and CuDNNLSTM is pretty much the same. LSTM is sligtly more accurate, but the loss is much higher at CuDNNLSTM (38% vs 56%) As our results are still not satisfactory, we will try again using one last model.

GRU

Below, we are using GRU, which is a newer generation of Recurrent Neural networks and is pretty similar to an LSTM. GRU is very useful given that, when working with neural networks, the model often forgets the first inputs from long sequences and using this kind of model allows for it to keep remembering, so if you are trying to process or predict on long paragraphs, it is important that the model remembers the first part of the input.

For this model, we will use one epoch again and otherwise the same structure as presented in the above models and print the result.

In [0]:

```
embedding_vecor_length = 300

model = Sequential()
model.add(Embedding(vocabulary_size, embedding_vecor_length, input_length=MAXLEN
))
model.add(SpatialDropout1D(0.1))
model.add(GRU(64))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(Dense(1, activation='sigmoid'))
```

In [50]:

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'
])
print(model.summary())
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 120, 300)          1500000
_________________________________________________________________
spatial_dropout1d_3 (Spatial (None, 120, 300)          0
_________________________________________________________________
gru_1 (GRU)                  (None, 64)                70080
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
batch_normalization_3 (Batch (None, 64)                256
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 65
=================================================================
Total params: 1,570,401
Trainable params: 1,570,273
Non-trainable params: 128
_________________________________________________________________
None
```

In [51]:

```
model.fit(X_train, y_train, epochs=1, batch_size=64, validation_split=0.1)
```

```
Train on 5692 samples, validate on 633 samples
Epoch 1/1
5692/5692 [==============================] - 21s 4ms/step - loss: 0.
5471 - acc: 0.7196 - val_loss: 0.4277 - val_acc: 0.8073
```

Out[51]:

```
<keras.callbacks.History at 0x7f1ba11d7ef0>
```

In [52]:

```
results = model.evaluate(X_test, y_test)
print("The Loss and Accuracy:", results)
```

```
1582/1582 [==============================] - 4s 3ms/step
The Loss and Accuracy: [0.43380268123448273, 0.8109987356267898]
```

The accuracy of this model is between 74.9 and 84.95%.

Conclusion

This assignment shows that our deep learning predictions are higher than the supervised Machine Learning prediction (~ 83% vs. 75%). This might be because the Machine Learning model is based on features, which do not have a very high impact on the price of the accommodation. In the Deep Learning part, there has been used three different models to predict on AirBnB prices according to the title of the accommodation. The reason for choosing three different models to do the same predictions is that we wanted to investigate, which could deliver the best results by comparing the loss and accuracy of each model. The results show that our LSTM model predicted with the best accuracy and lowest loss score. However, the numbers were quite close and similar.

We believe that the reason that our Deep Learning models make better predictions is that the input in these models are based on the name which might include some more relevant information about the accomodation. For example location, number of rooms and description of the whether it is a luxury or studio accommodation, which were shown in previous examples.

Therefore, it can be concluded that with this dataset and our problem statement, the Deep Learning models are the best choice, since they have the highest accuracy.