Predicting Airbnb prices with machine learning and deep learning

Experimentation with XGBoost and tuning neural networks

Laura Lewis May 22, 2019 · 10 min read ★

Source: plumguide.com

Project aims and background

Airbnb is a home-sharing platform that allows home-owners and renters ('hosts') to put their properties ('listings') online, so that guests can pay to stay in them. Hosts are expected to set their own prices for their listings. Although Airbnb and other sites provide some general guidance, there are currently no free and accurate services which help hosts price their properties using a wide range of data points.

Paid third party pricing software is available, but generally you are required to put in your own expected average nightly price ('base price'), and the algorithm will vary the daily price around that base price on each day depending on day of the week, seasonality, how far away the date is, and other factors.

Airbnb pricing is important to get right, particularly in big cities like London where there is lots of competition and even small differences in prices can make a big difference. It is also a difficult thing to do correctly — price too high and no one will book. Price too low and you'll be missing out on a lot of potential income.

cleaning of Airbnb data and conducted some exploratory data analysis in previous posts. This post is all about the creation of models to predict Airbnb prices.

The dataset

The dataset used for this project comes from Insideairbnb.com, an anti-Airbnb lobby group that scrapes Airbnb listings, reviews and calendar data from multiple cities around the world. The dataset was scraped on 9 April 2019 and contains information on all London Airbnb listings that were live on the site on that date (about 80,000).

The data is quite messy, and has some limitations. The major one is that it only includes the advertised price (sometimes called the 'sticker' price). The sticker price is the overall nightly price that is advertised to potential guests, rather than the actual average amount paid per night by previous guests. The advertised prices can be set to any amount by the host.

Nevertheless, this dataset can still be used as a proof of concept. A more accurate version could be built using data on the actual average nightly rates paid by guests, e.g. from sites like AirDNA that scrape and sell higher quality Airbnb data.

After cleaning and dropping collinear columns, the features in the model were:

- The number of people the property accommodates
- The number of bathrooms
- Property type (e.g. apartment) and room type (e.g. entire home)
- Location of the property (on the level of borough (discussed further in a previous post), or in one model on the level of latitude and longitude — discussed further below)
- Security deposit, cleaning fee and extra person fee
- Minimum and maximum nights stay
- Number of days available to book in the next 90 days

- Review ratings for each category (accuracy, cleanliness, check-in, communication, location, value and overall total)
- Amount of time since the first and most recent reviews
- The type of cancellation policy
- Whether the property is instant bookable
- The presence or absence of a wide range of amenities (discussed in further depth in a previous post, but including items like TVs, coffee machines, balconies, internet and parking, whether or not the property is child-friendly, allows self check-in or allows pets, and many others)
- Host response times and rates
- Whether or not a host is a superhost (a mark of quality, requiring various conditions to be met) or has their identity verified (e.g. by verifying government ID, a phone number and an email address)
- How many listings the host is responsible for in total
- How many days the host has been listing on Airbnb

Building a machine learning model

In the interests of space I'll skip the data preparation stage here, but all the code for this project can be found in my GitHub repo if you're interested. To summarise, after cleaning the data, checking for multi-collinearity and removing collinear features, the data was standardised using sklearn's `StandardScaler()` unless otherwise stated. Categorical features were one-hot encoded using `pd.get_dummies()`. A train-test split was performed with a test size of 0.2.

Although I was keen to experiment with deep learning models for price prediction, I first built a vanilla (non-tuned) XGBoost machine learning model (specifically `xgb.XGBRegressor()`). This was in order to provide a baseline level of accuracy, and also to allow for the measuring of feature importance (something which is notoriously difficult once you enter the realm of deep learning). XGBoost is likely to provide the best

awesomeness as observed in Kaggle competitions.

Because this is a regression task, the evaluation metric chosen was mean squared error (MSE). I was also interested in accuracy, so I also had a look at the r squared value for each model produced.

Here's my code to fit and evaluate the model:

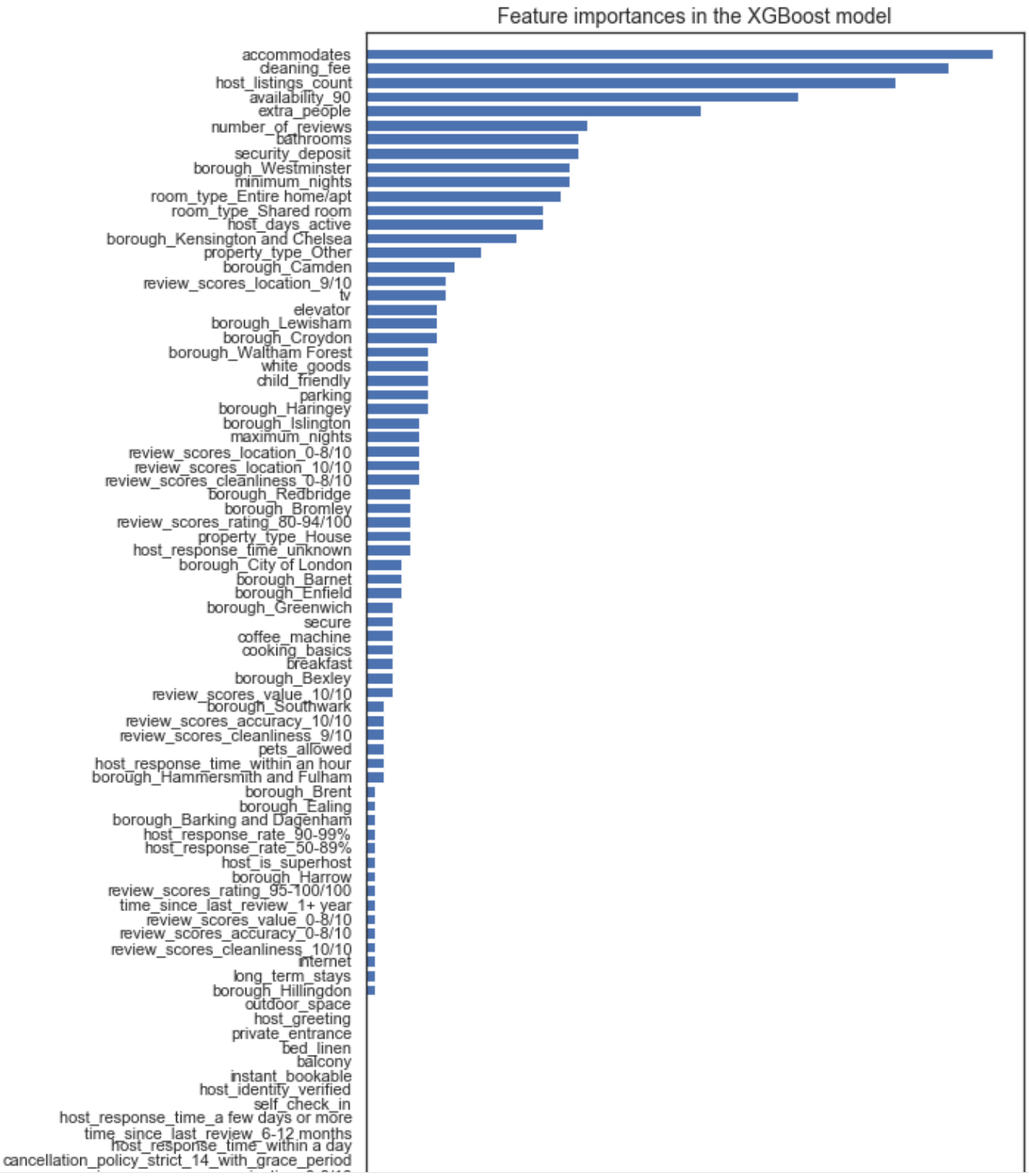
```
import xgboost as xgb

# Fitting the model
xgb_reg = xgb.XGBRegressor()
xgb_reg.fit(X_train, y_train)
training_preds_xgb_reg = xgb_reg.predict(X_train)
val_preds_xgb_reg = xgb_reg.predict(X_test)

# Printing the results
print(f"Time taken to run: {round((xgb_reg_end - xgb_reg_start)/60,1)} minutes")
print("\nTraining MSE:", round(mean_squared_error(y_train, training_preds_xgb_reg),4))
print("Validation MSE:", round(mean_squared_error(y_test, val_preds_xgb_reg),4))
print("\nTraining r2:", round(r2_score(y_train, training_preds_xgb_reg),4))
print("Validation r2:", round(r2_score(y_test, val_preds_xgb_reg),4))

# Producing a dataframe of feature importances
ft_weights_xgb_reg = pd.DataFrame(xgb_reg.feature_importances_, columns=['weight'], index=X_trai
ft_weights_xgb_reg.sort_values('weight', inplace=True)

# Plotting feature importances
plt.figure(figsize=(8,20))
plt.barh(ft_weights_xgb_reg.index, ft_weights_xgb_reg.weight, align='center')
plt.title("Feature importances in the XGBoost model", fontsize=14)
plt.xlabel("Feature importance")
plt.margins(y=0.01)
plt.show()
```

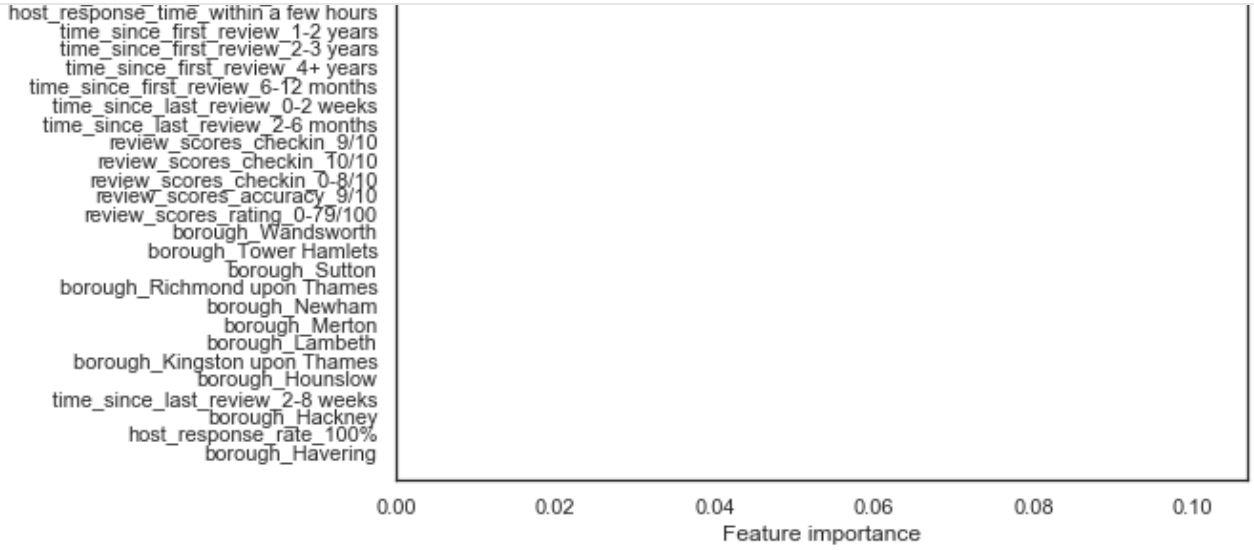
airbnb-xgboost.py hosted with ♡ by GitHub view raw

Results:

```
Training r2: 0.7321
Validation r2: 0.7274
```

Not bad for an un-tuned model. Now for the feature importances:





The top 10 most important features are:

- How many people the property accommodates
- The cleaning fee
- How many other listings the host has (and whether they are a multi-listing host)
- How many days are available to book out of the next 90
- The fee per extra person
- The number of reviews
- The number of bathrooms
- The security deposit
- If the property is in Westminster
- The minimum nights stay

It is not surprising that **the most important feature is how many people the property accommodates**, as that's one of the main things you would use to search for properties in the first place. It is also not surprising that **features related to location and reviews are in the top ten**.

However, this does not mean that a host that manages more properties will result in a listing gaining higher prices (although this is indeed the direction of the relationship). Firstly, the data appears to be somewhat skewed by a few very large property managers. Secondly, the relationship is with the advertised prices set, rather than actual prices achieved, suggesting that if anything more experienced hosts tend to set (rather than necessarily achieve) higher prices. And thirdly, we cannot necessarily imply a causative relationship — it could be that more experienced multi-listing hosts tend to take on more expensive properties (which is indeed the case for some, e.g. One Fine Stay).

It is also notable that three **other fee types — cleaning, security and extra people — all make the top 10 feature list**. It is likely that when a host sets a higher price for the nightly stay they are also likely to set other prices high, or vice versa.

Building a deep learning model

Next up, I decided to experiment with neural networks (NN), to see if I could improve upon the XGBoost model's score. I started off with a relatively shallow three layer NN with densely-connected layers, using a relu activation function for the hidden layers and a linear activation function for the output layer (as it is being used for a regression task). The loss function was mean squared error (again, because this is for regression).

Here's my code:

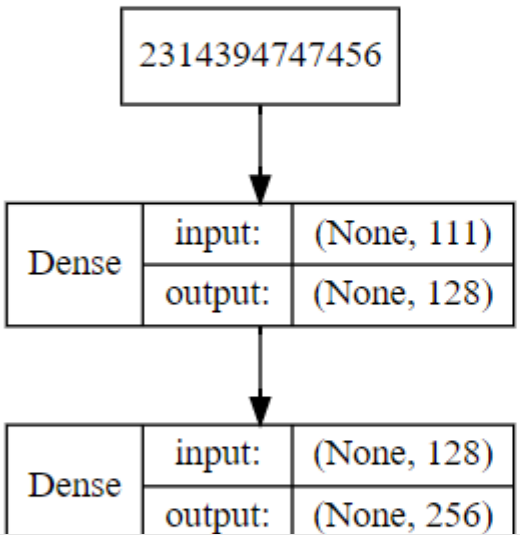
```
from keras import models, layers, optimizers, regularizers
from keras.utils.vis_utils import model_to_dot
from IPython.display import SVG

# Building the model
nn2 = models.Sequential()
nn2.add(layers.Dense(128, input_shape=(X_train.shape[1],), activation='relu'))
nn2.add(layers.Dense(256, activation='relu'))
nn2.add(layers.Dense(256, activation='relu'))
nn2.add(layers.Dense(1, activation='linear'))

# Compiling the model
nn2.compile(loss='mean_squared_error',
            optimizer='adam',
            metrics=['mean_squared_error'])
```

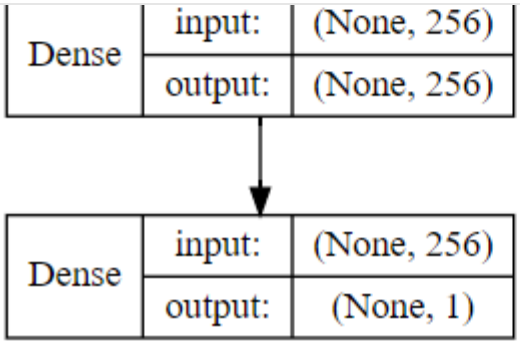
```
print(nn2.summary())

# Visualising the neural network
SVG(model_to_dot(nn2, show_layer_names=False, show_shapes=True).create(prog='dot', format='svg'))

# Training the model
nn2_history = nn2.fit(X_train,
                      y_train,
                      epochs=100,
                      batch_size=256,
                      validation_split = 0.1)
```

three-layer-nn.py hosted with ♡ by GitHub view raw

And here's the summary and visualisation:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               14336
_________________________________________________________________
dense_2 (Dense)              (None, 256)               33024
_________________________________________________________________
dense_3 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 257
=================================================================
Total params: 113,409
Trainable params: 113,409
Non-trainable params: 0
_________________________________________________________________
None
```





In order to save time when evaluating multiple methods, I built a handy function to print the MSE and r squared results for the test and train sets, as well as produce a line graph of the loss in each epoch for the test and train sets and a scatterplot of predicted vs. actual values:

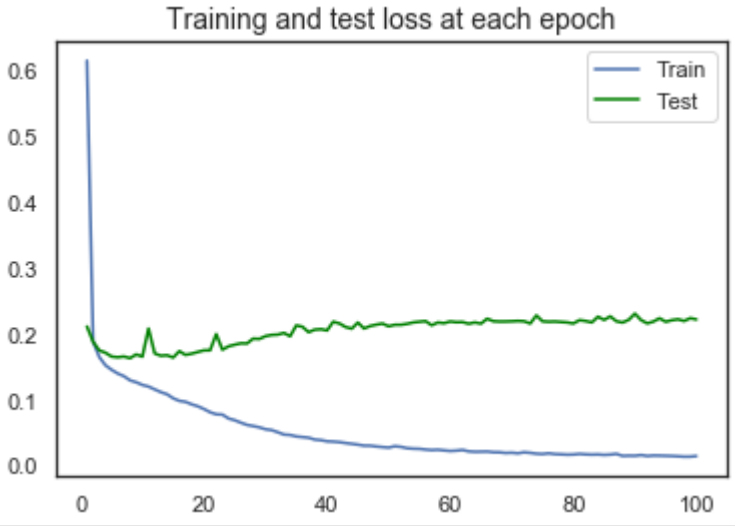
```
def nn_model_evaluation(model, skip_epochs=0, X_train=X_train, X_test=X_test, y_train=y_train, y
    """
    For a given neural network model that has already been fit, prints for the train and tests s
    values, a line graph of the loss in each epoch, and a scatterplot of predicted vs. actual va
    representing where predicted = actual values. Optionally, a value for skip_epoch can be prov
    number of epochs in the line graph of losses (useful in cases where the loss in the first ep
    larger than subsequent epochs). Training and test sets can also optionally be specified.
    """

    # MSE and r squared values
    y_test_pred = model.predict(X_test)
    y_train_pred = model.predict(X_train)
    print("Training MSE:", round(mean_squared_error(y_train, y_train_pred),4))
    print("Validation MSE:", round(mean_squared_error(y_test, y_test_pred),4))
    print("\nTraining r2:", round(r2_score(y_train, y_train_pred),4))
    print("Validation r2:", round(r2_score(y_test, y_test_pred),4))

    # Line graph of losses
    model_results = model.history.history
    plt.plot(list(range((skip_epochs+1),len(model_results['loss'])+1)), model_results['loss'][sk
    plt.plot(list(range((skip_epochs+1),len(model_results['val_loss'])+1)), model_results['val_l
    plt.legend()
    plt.title('Training and test loss at each epoch', fontsize=14)
    plt.show()

    # Scatterplot of predicted vs. actual values
```

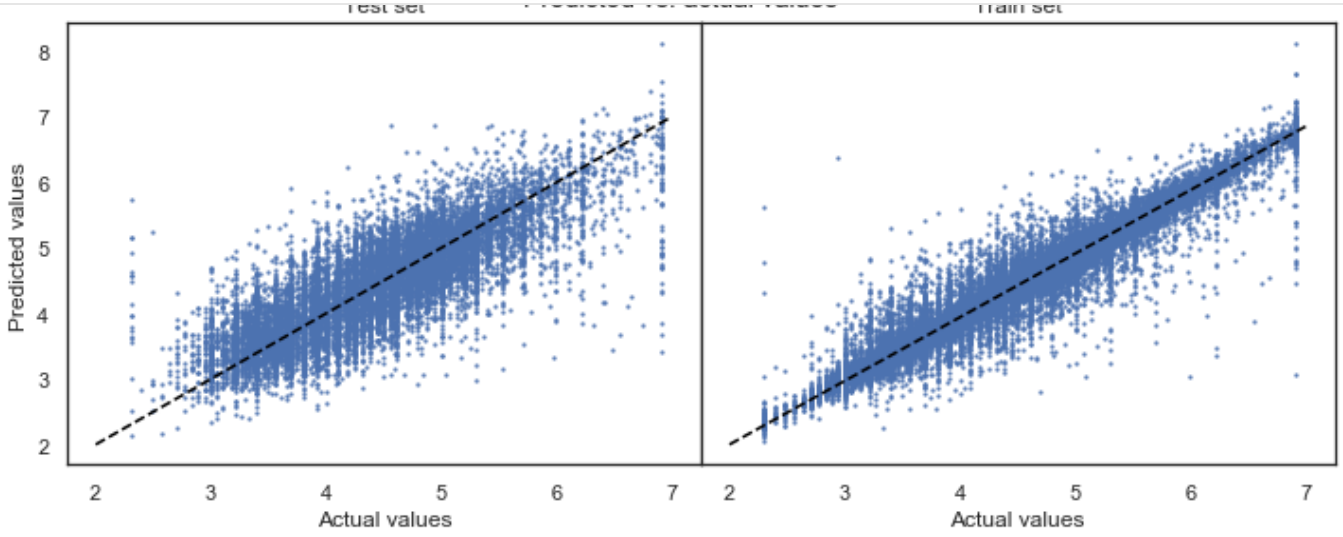
```
30
31      ax1.scatter(y_test, y_test_pred, s=2, alpha=0.7)
32      ax1.plot(list(range(2,8)), list(range(2,8)), color='black', linestyle='--')
33      ax1.set_title('Test set')
34      ax1.set_xlabel('Actual values')
35      ax1.set_ylabel('Predicted values')
36
37      ax2.scatter(y_train, y_train_pred, s=2, alpha=0.7)
38      ax2.plot(list(range(2,8)), list(range(2,8)), color='black', linestyle='--')
39      ax2.set_title('Train set')
40      ax2.set_xlabel('Actual values')
41      ax2.set_ylabel('')
42      ax2.set_yticklabels(labels='')
43
44      plt.show()
```

nn-evaluation.py hosted with ♡ by GitHub view raw

Here are the results:

```
Training MSE: 0.0331
Validation MSE: 0.2163

Training r2: 0.9438
Validation r2: 0.6292
```





Compared to the XGBoost model the neural network has performed worse. Overfitting also seems to be an issue, as seen from the difference between the train and test MSE and r squared results, as well as the difference between the train and test losses in the line graph, and the fact that the values cluster more closely to the line in the scatterplots.

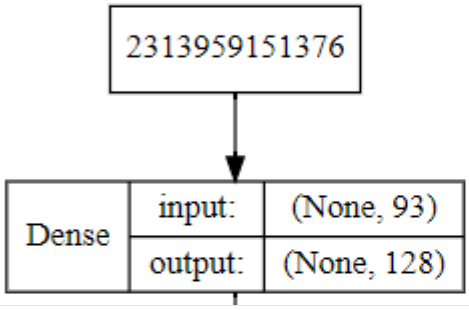
I then iterated through various other versions of the model in order to try and remove the overfitting and increase the accuracy. Overfitting was removed in each other version, although accuracy did vary. The adjustments that I experimented with were:

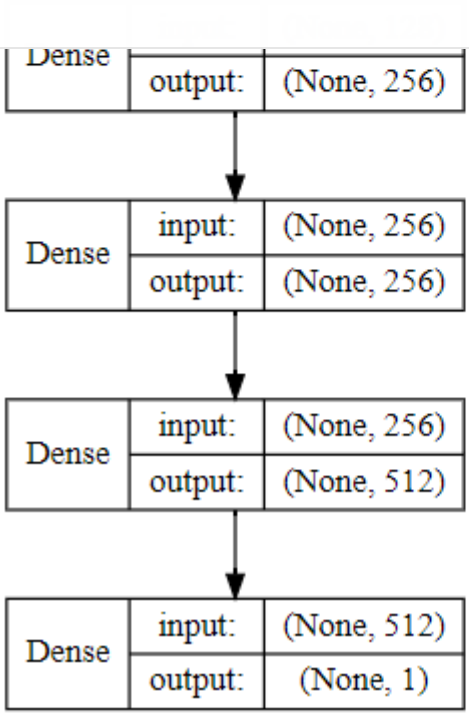
- Adding a fourth and fifth layer — a fourth layer improved the accuracy but a fifth layer didn't help.
- Using L1 regularization — this proved to be the biggest boost to accuracy.
- Using dropout regularization at 30% and 50% dropout rates — 50% turned out to be a terrible idea and significantly increased the MSE. 30% performed better, but not as well as L1 regularization.
- Using a stochastic gradient descent (SGD) optimiser instead of Adam — this performed slightly worse.
- Changing the batch size — this didn't make much difference.

- Removing most of the review rating columns — high review ratings in one category were fairly highly correlated with other categories, so I tried removing all except the overall rating. I then used this new truncated dataset to train the previously highest performing model architecture (with L1 regularization and an Adam optimizer). This performed essentially the same, but with 18 fewer columns, so would be the preferred model when choosing which model to put into production as it would require less data and be less computationally expensive.
- Using latitude and longitude instead of borough — again, this adjusted dataset was used with the best model architecture I had come up with so far. This performed slightly worse.
- Using `MinMaxScaler()` instead of `StandardScaler()`. This also performed slightly worse.

In the end, the best NN was the four-layer model with L1 regularization and an Adam optimizer, with the extra review columns removed:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 128)               12032
_________________________________________________________________
dense_24 (Dense)             (None, 256)               33024
_________________________________________________________________
dense_25 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_26 (Dense)             (None, 512)               131584
_________________________________________________________________
dense_27 (Dense)             (None, 1)                 513
=================================================================
Total params: 242,945
Trainable params: 242,945
Non-trainable params: 0
_________________________________________________________________
None
```

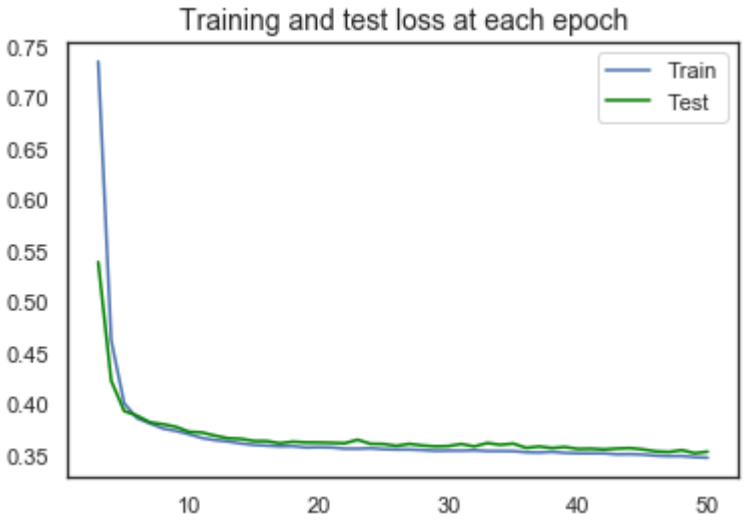


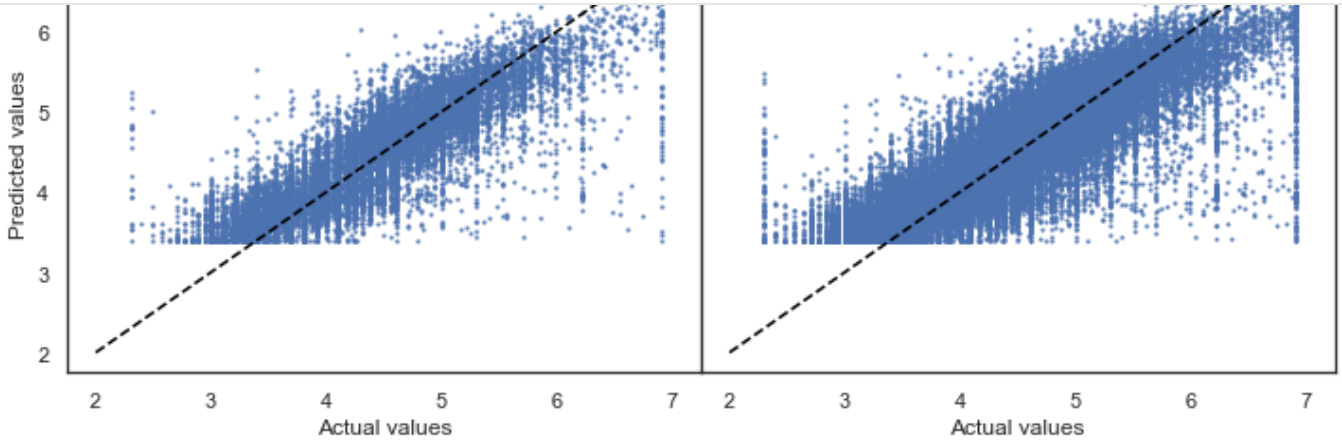


My best performing neural network architecture

Results:

```
Training MSE: 0.1708
Validation MSE: 0.1689

Training r2: 0.7096
Validation r2: 0.7105
```





I still haven't gotten to the bottom of why the NN is unable to predict values for the log-transformed price lower than about 3.1, but I'm pretty sure it's something to do with the use of regularization

However, even this model architecture did not perform quite as well as the XGBoost model. Overall, the XGBoost model is the preferred model, as it performs ever so slightly better than the best neural network and is less computationally expensive. It could possibly be improved even further with hyper-parameter tuning.

Conclusions

This is one of those situations where deep learning simply isn't necessary for prediction, and a machine learning model performs just as well.

However, even in the best performing model, the model was only able to explain 73% of the variation in price. The remaining 27% is probably made up of features that were not present in the data. It is likely that a significant proportion of this unexplained variance is due to variations in the listing photos. The photos of properties on Airbnb are very important in encouraging guests to book, and so can also be expected to have a significant impact on price — better photos (primarily better quality properties and furnishings, but also better quality photography) equal higher prices.

Potential directions for future work

- Find a way to incorporate image quality into the model, e.g. by using the output of a convolutional neural network to assess image quality as an input into the pricing model.

- Include a wider geographic area, e.g. the rest of the UK or other major cities around the world.
- Augment the model with natural language processing (NLP) of listing descriptions and/or reviews, e.g. for sentiment analysis or looking for keywords.
- In addition to predicting base prices, a sequence model could be created to calculate daily rates using data on seasonality and occupancy, which would allow the creation of actual pricing software.
- Tailor the model more specifically to new listings in order to help hosts set prices for new properties, by removing features that would not be known at the time — e.g. other fees, availability and reviews.

I hope you found this post interesting and/or helpful. Thanks for reading!

Machine Learning Deep Learning Data Science Airbnb Neural Networks