k-train Text-Regression: Final Report

Harman Ransi | 4/23/23 | Rowan University

I. Overview of ktrain

At the core, `k-train` is a python library which helps users develop machine learning models with significantly less lines of code. Serving as a wrapper class to `Keras` and other popular machine learning libraries, `k-train` helps simplify the process of creating, inspecting, and analyzing a model. It does so by automating preprocessing steps, providing modern models for users to select from, and predicting optimal learning rates for selected models. The library is also flexible, being able to handle text, vision, graph, and tabular data.

"ktrain: A Low-Code Library for Augmented Machine Learning", an article published in the Journal of Machine Learning Research (2022), details how `k-train` can be used with text data for both supervised and unsupervised Machine Learning tasks. For supervised tasks, it can handle preprocessing text, creating a BERT model, estimating the learning rate, and training the model using single line functions native to the ktrain library. For unsupervised tasks, such as creating a QA system, the library has in-built functions which can read and index the text. Generally speaking, this article touches on the basics of how to use the library for text-related tasks.

However, as I mentioned in my project proposal, the implementation of `k-train` in this article is rather simple. To expand on the article's ideas, I will be analyzing the linear text-regression libraries within `k-train` instead.

To explore this library, I created a linear text-regression model using `k-train` to help understand how the library functions. Then, to analyze `k-train`'s performance, I compared the model's accuracy to another similar text-regression model built using `Keras`. Because `k-train` is a wrapper class for `Keras`, comparing the performance between these models provided valuable insight.

II. Overview of Dataset, Models, & Project

In my initial report, the `k-train` and `Keras` regression models were built on a dataset of wine reviews. These models used the **`description`** variable of the wine records to predict their **`price`**. This research helped with becoming familiar with the libraries at hand. It also helped to identify key components of developing a robust model.

For this report, a `k-train` and `Keras` model were built using a Kaggle dataset of 2019 NYC AirBnB reports. These models used the **`name`** and **`neighbourhood`** of the posting records to predict their **`price`** per night.

There were two primary objectives of this project. The first was to compare the performance of similarly built `k-train` and `Keras` models. In particular, analyzing if the trade-off of `k-train`'s low code approach ids worthwhile compared to building a typical `Keras` model. The second objective was to minimize the mean absolute error of these models. This was to ensure the linear text-regression models could provide sufficient **`price`** predictions given the **`name`** and **`neighbourhood`** of a posting.

The first section of this report will provide a detailed insight into the AirBnB dataset. Continuing sections will review the process used to develop each model. The final sections will analyze the results of the models in an effort to uncover the trade-off, if any, of using `k-train`.

III. Exploration - Understanding and Cleaning Data

Before creating the text-regression models, the dataset was analyzed and cleaned. The AirBnB data consisted of approximately 50,000 records. As previously mentioned, the variables of interest were **`name`**, **`neighbourhood`**, and **`price`**. Although the variable names are rather self-explanatory, **`name`** is the title of the AirBnB posting, **`neighbourhood`** is the general location of the posting, and **`price`** is the rent per night of the posting. Below is a snapshot showing a few records of the data.

name	neighbourhood	price
by the park	Kensington	149
town Castle	Midtown	225
EW YORK !	Harlem	150
Brownstone	Clinton Hill	89
central park	East Harlem	80

`data.head()`

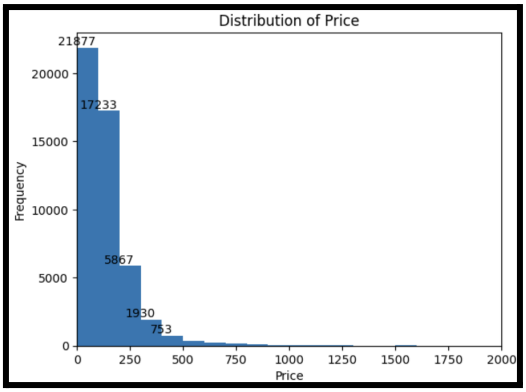
With the focus being to build a linear text-regression model, the **`name`** and **`neighbourhood`** inputs needed to be merged into one string. Using a lambda statement to combine these inputs, a new variable called **`name_neighbourhood`** was created. The code used is shown on the following page.

```
data.apply(lambda x: f"{x['name']} {x['neighbourhood']}"
```

data['name_neighbourhood']

Then, all null values for **name_neighbourhood** or **price** were dropped. Models cannot compile if there are missing values in either the training or testing data. Therefore, this was done to ensure that the models developed later would still compile.

To finish cleaning the data, all records with a **price** of $325 or above were dropped. The mean absolute error of any linear regression model is susceptible to outliers. A well-fitted linear model can still have a high mean absolute error if there are just a few outliers in the data. The linear text-regression models built in later steps were measured by their mean absolute error, making it essential to eliminate abnormal **price** records. To visualize the distribution of **price**, a histogram was created. From the histogram below, it is evident that a high majority of **price** records fall within $325.



np.histogram()

Before dropping these records, the dataset had a total of 48,895 rows. After dropping them, the dataset still consisted of 45,861 rows. Approximately 94% of the original record remained after dropping records with a **price** below $325. Therefore, the new dataset was still fairly representative of the initial data.

The data was then separated into a training and testing set with an 80/20 split. The `sci-kit-learn` library features a function called **train_test_split()**. The function is able to shuffle and split a dataset into testing and training sets according to user specified parameters. The code used is shown below.

```
train_test_split(X, y,
                 test_size=0.2,
                 random_state=42)
```

sci-kit-learn

IV. Exploration - Model Creation

Before exploring the process of developing the `k-train` and the `Keras` models, it is important to note that these models were not developed to be identical. The goal was to develop equivalent models in both libraries to allow for easy comparison. However, `k-train`'s API does not document the exact layers used in their text regression functions. Therefore, slightly different linear models were developed in `Keras` This issue will be addressed in the next section of this paper.

The first model created was made using the `k-train` library. The benefit of this wrapper class is that it simplifies the process of creating and training a neural network model. It handles several tasks such as data preprocessing, model creation, and identifying the most optimal learning rate. Although there is some room for customization of the model, such as the batch size of the data when it is being fitted, most tasks are handled by `k-train` using very few lines of code.

To build the model, the training and validation data had to be preprocessed for the `k-train` library. As mentioned previously, the library makes preprocessing text related data rather simple. The **texts_from_array()** method from the `k-train` library is able to correctly format the training and validation data. Parameters specified in the method include **ngram_range**, **maxlen**, and **max_features**. The **ngram_range** parameter specifies the length of phrases which the model will consider. This parameter was set to 3 to allow the model to process short phrases only. Processing longer phrases may cause inaccuracies or for the model to overfit. The **maxlen** variable specifies the maximum number of words in a record that the model will process before ignoring said record. Although unlikely to occur, this was set to 200 incase there were any descriptions that were too long. The **max_features** variable specifies the maximum number of words the vocabulary will include. Since a large amount of records were being handled, this parameter was set to 25,000. The final output of the **texts_from_array()** method is the formatted training data [**trn**], formatted validation data [**val**], and a preprocessing variable [**prepoc**] to store information for the model built in the next step. The code used is shown below.

```
texts_from_array(x_train=X_train, y_train=y_train,
                 x_test=X_test, y_test=y_test,
                 ngram_range=3,
                 maxlen=200,
                 max_features=25000)
```

k-train.text

Then, the model was built using one line of code and the **`text_regression_model()`** function. The method's parameters are **`name`**, **`train_data`**, and **`prepoc`**. The **`name`** parameter was set to 'linreg' to specify that the model would be a linear text-regression model. The **`train_data`** and **`prepoc`** parameters are self explanatory and are set to the outputs of the **`texts_from_array()`** method. The output variable [**`model`**] stores the model in the memory. The code used is shown below..

```
text.text_regression_model('linreg',
                           train_data=trn,
                           preproc=preproc)
```

`k-train.text`

Preprocessing the data using `Keras` requires many more steps compared to `k-train`. Firstly, a **`Tokenizer`** object was initialized with a parameter of **`num_words`**. This parameter was set to 25,000 because the parameter specifies the number of words in the vocabulary. Then, the **`Tokenizer`** object extracted the text data using the **`fits_on_texts()`** method. It extracted the data by creating a unique integer index for each word. After this, the **`text_to_sequences()`** method was used on the same **`Tokenizer`** object. This allowed for the numeric vocabulary created in the previous method to be converted to a sequence of integers. These integers are able to represent each record of text data in a numeric fashion. Similarly to `k-train`, this function has a parameter of **`maxlen.`** This parameter details the maximum number of words in a record that the model will process before ignoring said record. It was set to 200 in an effort to stay consistent with the `k-train` model. The code used is shown below.

```
# create variables
X = data['name_neighbourhood'].values
y = data['price'].values
# preprocess the text data
tokenizer = Tokenizer(num_words=25000)
tokenizer.fit_on_texts(X)
X = tokenizer.texts_to_sequences(X)
X = pad_sequences(X, maxlen=200)
```

`tensorflow.keras.preprocessing.text.Tokenizer`

As seen in the above code snippet, the preprocessing was done on the entire dataset and not specifically on the training and testing sets. This was in an effort to reduce code. The unseparated data was then split using the **`train_test_split()`** method with the same **`random_state`** parameter to ensure that the split was the same as it was in the `k-train` model.

The `k-train` library automatically standardizes data. However, the `Keras` library does not. Therefore, to standardize the data, a **`Scaler`** object was created. The **`fit_transform()`** and **`transform()`** methods were used on the data. The code used is shown below.

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

`sklearn.preprocessing.StandardScaler`

After preprocessing the data, a `Keras` model could be made and compiled. However, my initial `Keras` simple-regression model performed poorly while training. Therefore, an additional complex linear regression model was also developed. To make these two models, two **`Sequential`** objects were initialized using the `Keras` library.

On both models, an **`Embedding`** layer was added to help create a vectorized representation of the data. This allows each model to understand the data it receives. The parameters of this layer include **`input_dim`**, **`output_dim`**, and **`input_length`**. The **`input_dim`** parameter specifies the size of the data being inputted into the model. In our case, the vocabulary size was 35,000, and therefore the value was set to the vocabulary size. The **`output_dim`** parameter specifies the dimensionality of the input words after being embedded. The larger the parameter, the more complex of relationships the model is able to find. However, too high of a number may create too complex of a model. Therefore, a value of 100 was set for the parameter. The **`input_length`** parameter specifies the max length of the words processed per record. Because this was set to 200 previously using the **`Tokenizer`**, it was also set to 200 for this parameter.

On the initial linear regression model, a hidden **`Dense`** layer with a 'relu' **`activation`** was included. This helped to introduce some nonlinearity into the model, making it easier for the model to find complex relationships in the text input. Then, a hidden **`Dropout`** layer was added to prevent overfitting. Finally, an output layer with a 'linear' **`activation`** was placed at the end of the model.

In the second, better-performing model, the only hidden layer was an **`LSTM`** layer. **`LSTM`** layers are known to significantly increase the training run-time of a model. This is because these layers help handle data which may be sequential in nature, such as text. They are computationally expensive, but can lead to remarkable results during training and testing. Therefore, this layer was added to this model, helping to improve its accuracy and robustness. The **`units`** parameter was set to 64 and **`activation`** parameter was set to 'tanh' as those are the default options. An output layer with a 'linear' **`activation`** was placed at the end to complete the model.

The code of each model is shown below. The linear text regression model is shown first. Because the precise layers of `k-train`'s linear text regression model were not known, this model was not a perfect imitation of `k-train`'s model. However, this model was still fairly representative of a typical linear text regression model.

```
model = Sequential()
model.add(Embedding(input_dim=25000,
                    output_dim=100,
                    input_length=200))
model.add(Flatten())
model.add(Dense(units=64, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(units=1, activation='linear'))
```

Linear Text-Regression Model

The higher performing, complex linear regression model, is shown second. After experimenting with several linear models, one with an **`LSTM`** layer provided most robust and accurate model after training.

```
model = Sequential()
model.add(Embedding(input_dim=25000,
                    output_dim=100,
                    input_length=200))
model.add(LSTM(units=64,
               activation='tanh'))
model.add(Dense(units=1,
                activation='linear'))
```

Linear Text-Regression (w/ LSTM) Model

Finally, both `Keras` models were trained using the code below. The learning rates used in each model is discussed in the next section.

```
model.compile(loss='mean_squared_error',
              optimizer=optimizer_obj,
              metrics=['mae'])
```

At this point, the two models in `Keras` and the model in `k-train` were built. After this was completed, each model was trained and the results were analyzed.

V. Exploration - Model Fitting and Analysis

To analyze how well each model performs, the training data needs to be fitted to the models. However, a **`learner`** object needs to be initialized with `k-train` before doing so. This object serves as the main interface for training deep learning models while also providing additional functionality to ensure the model can learn optimally.

The **`learner`** object is created by using `k-train`'s **`get_learner()`** method. The method's parameters include the **`model`** object created earlier using `k-train`, the batch size for model fitting, and the training / validation data. For clarification, the batch size was set to 64 to allow for a moderate learning pace.

After creating the **`learner`** object, the **`lr_estimate()`** method was invoked to find the most optimal learning rate. This method allows the **`learner`** object to find the learning rate that would estimate the best loss. After running the code, the most optimal learning rate was ".006". The results of **`lr_estimate()`** are shown below.

```
0.0066474252
```

Results of **`lr_estimate()`**

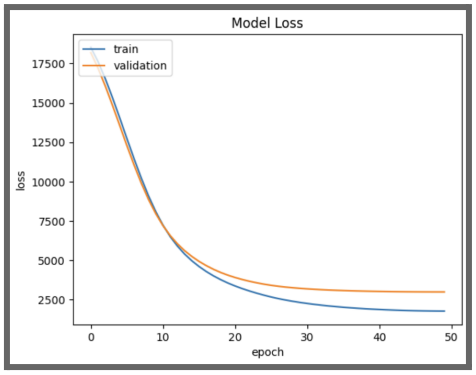
Using the results of the previous two methods, the model was trained by using the **`learner`** object's **`fit_onecycle()`** method. The method allowed the model to be trained using a one cycle policy. The method's first parameter is the learning rate, which was found to be ".006" from the **`lr_estimate()`** function. The second parameter specifies the number of epochs and was set to 50 to allow the model to learn for a significant amount of iterations. The **`learner`** object also features a plotting method which plots the change in training and validation loss over time. The output of the **`fit_onecycle()`** and **`plot()`** function are shown below.

```
loss: 1813.9111 - mae: 27.8974
loss: 1802.7715 - mae: 27.7750
loss: 1793.7261 - mae: 27.6695
loss: 1786.7543 - mae: 27.5894
loss: 1781.8260 - mae: 27.5335
loss: 1778.8915 - mae: 27.4979
```

`K-train`*'s Training Loss* (Last 6 Epochs)

```
val_loss: 3005.7278 - val_mae: 40.0505
val_loss: 3002.4155 - val_mae: 40.0330
val_loss: 2999.8691 - val_mae: 40.0176
val_loss: 2997.9727 - val_mae: 40.0061
val_loss: 2996.9387 - val_mae: 40.0002
val_loss: 2996.5505 - val_mae: 39.9981
```

`K-train`*'s Validation Loss* (Last 6 Epochs)



`K-train`'*s Loss Plot*

The gradual decline in both training and validation loss shown in the plot is a good sign of a decent model. The plot makes the model seem slightly overfitted, but it still has decent shape. Also, when looking at the change in the mean absolute error from **`fit_onecycle()`**'s results, we can see a gradual decrease in both the training and validation data. The final mean absolute error of the validation set was found to be $39.99.

After training the `k-train` model and calculating the final mean absolute error for the validation set, the same was done for the two `Keras` models.

Although it is possible to estimate the most optimal learning rate with `Keras`, it is a rather cumbersome process requiring hand-coded functions. There are no native functions which can estimate the learning rate in `Keras`. Therefore, the learning rate for each `Keras` model was different.

The lower-performing, simple linear regression model had a standard learning rate of "0.1" since a learning rate of ".006" was too slow for it. Although this model was developed to be similar to `k-train`'s model, it was not exactly the same, and therefore, using the same learning rate did not work well.

The higher-performing, **`LSTM`** linear regression model, had a learning rate of "0.006". This was done in an effort to keep the learning rate low and keep this model more consistent with the `k-train` model. **`LSTM`** layers typically benefit from having a lower learning rate. Also, setting the learning rate to be the same as the `k-train` model makes these models more alike, allowing for a fairer comparison of each library.

After determining the learning rates, the **`fit()`** function was called on each **`model`** object. The parameters of the function include the training / validation data, the number of epochs the model will be trained over, and the batch size for training.

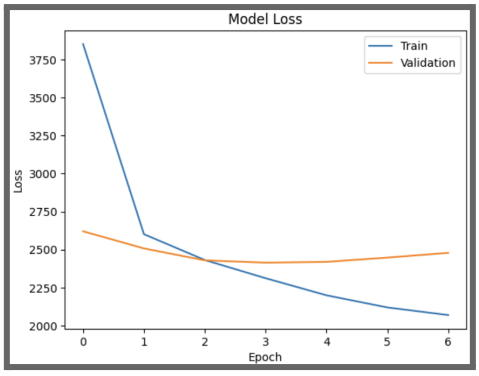
The simple linear text regression model was trained for 50 epochs, but due to the **`EarlyStopping`** parameter in the **`fit()`** function, the model only ran for 7 epochs. As hinted at previously, the model did not have the best performance. The batch size was set to 64 to stay consistent with the `k-train` model. The change in loss between the training and validation sets were also plotted using the **`matplotlib`** library. The output of the **`fit()`** and the plot functions are shown below.

```
loss: 3851.3677 - mae: 47.2882
loss: 2601.3376 - mae: 38.2052
loss: 2431.4397 - mae: 36.6503
loss: 2311.7102 - mae: 35.5267
loss: 2199.4094 - mae: 34.5824
loss: 2119.4551 - mae: 33.8460
loss: 2069.4138 - mae: 33.2782
```

`Keras [Simple Model]` - *Training Loss*

```
val_loss: 2620.2461 - val_mae: 38.3240
val_loss: 2507.8413 - val_mae: 38.3168
val_loss: 2429.3374 - val_mae: 36.1257
val_loss: 2414.0740 - val_mae: 36.1286
val_loss: 2419.2510 - val_mae: 36.3526
val_loss: 2447.3806 - val_mae: 36.6721
val_loss: 2478.6565 - val_mae: 36.2298
```

`Keras [Simple Model]` - *Validation Loss*



`Keras [Simple Model]` - *Loss Plot*

The fluctuation in the validation loss is a sign of a poor model. The model is not only overfit, but the mean absolute error increases in later epochs. As mentioned previously, the **`EarlyStopping`** parameter in the **`fit()`** function halts model training when the loss continues to increase. The final mean absolute error of the validation set was $36.22. Although this is almost $4 lower than the `k-train` model's mean absolute error, the loss plot indicates that the model is unreliable.

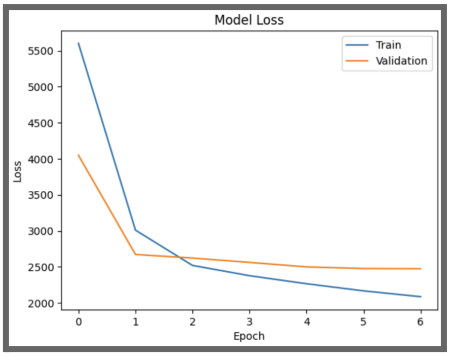
However, the **`LSTM`** linear regression model performed rather well. Due to RAM limitations, the epochs were reduced to 7. The batch size was set to 64 to stay consistent with the `k-train` model. The change in loss between the training and validation sets were also plotted using the **`matplotlib`** library. The output of the **`fit()`** and the plot functions are shown below.

```
loss: 5600.8735 - mae: 56.1118
loss: 3009.4097 - mae: 41.1998
loss: 2519.9451 - mae: 37.2816
loss: 2377.1572 - mae: 36.1676
loss: 2265.6555 - mae: 35.1546
loss: 2166.7202 - mae: 34.2696
loss: 2085.7676 - mae: 33.5661
```

`Keras [LSTM Model]` - *Training Loss*

```
val_loss: 4048.4705 - val_mae: 46.3560
val_loss: 2671.8740 - val_mae: 39.2692
val_loss: 2621.9548 - val_mae: 37.9529
val_loss: 2561.8303 - val_mae: 37.6831
val_loss: 2499.1177 - val_mae: 37.1538
val_loss: 2476.8086 - val_mae: 37.2412
val_loss: 2474.0535 - val_mae: 37.0087
```

`Keras [LSTM Model]` - *Validation Loss*



`Keras [LSTM Model]` - *Loss Plot*

The gradual decline in both training and validation loss is a good sign of a decent model. Although the model is slightly overfitted and the validation loss seemed to stagnate at later epochs, the shape of the loss plot looked decent. In the training and validation data, the mean absolute error decreases at each epoch. The final mean absolute error of the validation set was $37. Although the mean absolute error is a dollar higher than the simple `Keras` model, this model is significantly more robust. Based on the trajectory of the loss plot, this model may have produced even better results if it ran for more epochs on a more powerful machine.

VI. Exploration - Analysis of Model Results

The initial goal of this project was to build similar linear text regression models in `k-train` and `Keras` in an effort to fairly compare the libraries. In particular, the goal was to analyze if `k-train`'s low-code approach to text regression was a worthwhile trade-off relative to `Keras`. Thus, each model trained in the previous section will be analyzed.

Only one model was made in the `k-train` library. This model performed very well given its low development time. The model trained through 50 epochs in approximately 8 minutes. Furthermore, the model's mean absolute error was about $39.99. This is rather impressive for a model solely using titles to predict AirBnB prices.

Unfortunately, the layers of `k-train`'s text regression function were unknown. This made it rather difficult to rebuild an identical model using `Keras`. To combat this challenge, two `Keras` models were developed and tested in the hopes of creating a linear regression model with a low mean absolute error.

The simple linear regression model had poor performance. Although it had the lowest mean absolute error at $32.66, the loss plots did not have good shape. The runtime was only 5 minutes since the model stopped training at the 7th epoch. The model did not fairly replicate `k-train`'s text regression model. Other factors such as learning rate and batch size may have had an influence on the error.

However, the **`LSTM`** linear regression model had promising performance. Although this model did not fairly replicate `k-train`'s text regression model, it provided accurate results. The mean absolute error was $37 and its loss plot had a good shape. The **`LSTM`** layer was computationally heavy, causing the model to take 25 minutes to train over 7 epochs.

Given the performance of the `k-train` model and the two `Keras` models, it is difficult to confidently proclaim that `k-train`'s low-code approach is a worthwhile tradeoff.

On one hand, `k-train` was able to develop a linear regression model that produced a mean absolute error of $39.99 in about 5 lines of code. This highlights the library's ease of use and its decent accuracy.

On the other hand, the **`LSTM`** linear regression model developed in `Keras` resulted in a mean absolute error of $37. This model's error was almost $4 better than the `k-train` model. Even though it took much longer to train, this demonstrates the advantage to being able to customize layers in a `Keras` model.

These findings suggest that the tradeoff between `k-train` and `Keras` models depends on how low of error a developer wants. `k-train`'s text regression models perform rather well despite being low-code. However, if a developer wants more accurate models, `Keras` allows for layer modification which may lead to better accuracy.

VII. Conclusion

As extensively discussed in the previous section, the findings of this project suggest that the tradeoff between `k-train` and `Keras` is dependent on the error a developer is willing to tolerate. `k-train`'s text-regression models provide an easy way to build and train a model. However, `Keras` allows for more model flexibility, which may help increase accuracy by a significant degree.

The secondary goal of this project was to create models with a low mean absolute error. The average AirBnB price per night in the cleaned dataset was $152. The `k-train` model had a mean absolute error of $39.99, making the model approximately 74% accurate. The `Keras` model with the **`LSTM`** layer had a mean absolute error of $37, making the model 76% accurate. Given that these models used text-descriptions of AirBnB postings as a predictor of their price, each model's performance is rather decent. As previously mentioned, the loss plots of these models also suggest that the models are robust.

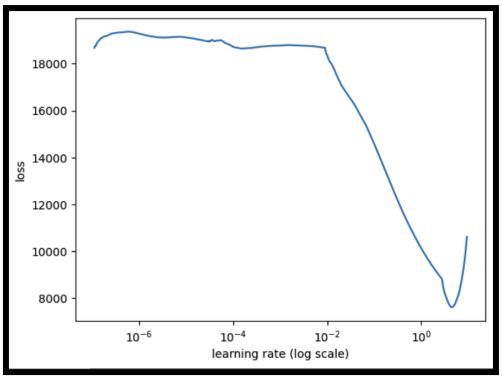
If further research were to be done on this project, investigating the exact layers in `k-train`'s text regression model may be of benefit. This would allow for fairer comparison of `k-train` and `Keras`. To further reduce the mean absolute error, experimenting with additional `Keras` layers may be beneficial.

VIII. Appendix

Some results found during this research were not mentioned in this report. `k-train` functions which were not useful in developing the models mentioned earlier were excluded. Older, less accurate, models were also omitted from this report. This section will analyze these results to provide additional context and information.

As previously mentioned, a text-regression model was made on a wine dataset for the first report. In the report, the k-train function **`lr_plot()`** was used to visualize the most optimal learning rate for the data. It did so by plotting the change in loss as a function of the learning rate. The function used the results of **`lr_find()`** to do so. This function was only included in the initial report because it was heavily mentioned on `k-train`'s official GitHub.

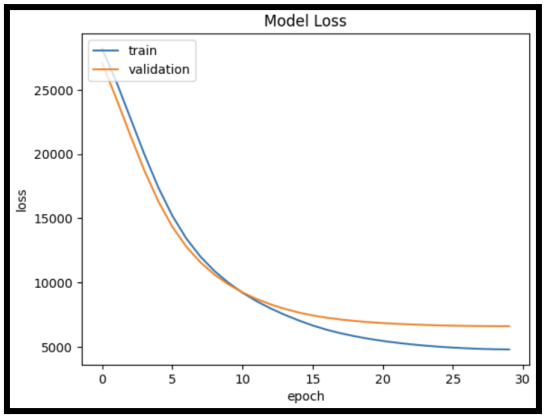
However, for this report, the function was not mentioned. This is because the x-axis of the plot was in a log scale, making it difficult to identify the exact optimal learning rate. Furthermore, **`lr_estimate()`** was able to provide the most optimal learning rate as an exact numerical value. **`lr_plot()`** was an interesting function to explore, but due to its poor visibility, it was rather unhelpful for this report. The results of using the function on the AirBnB dataset can be seen below.



Results of **`lr_plot()`**

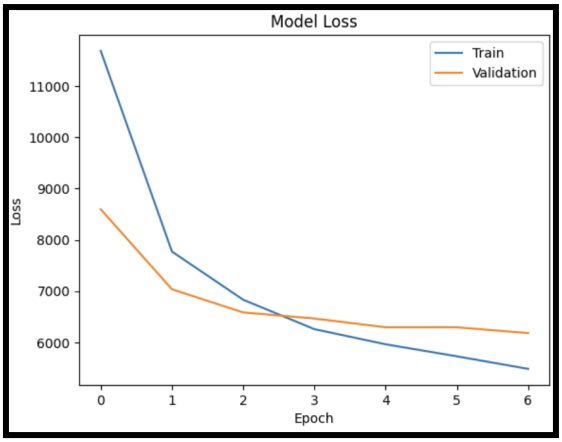
Older, less accurate models, were also omitted from this report. They were omitted due to their higher mean absolute error. Their mean average errors were approximately $10 higher than the models spotlighted earlier. However, the loss plots of these models had an excellent shape. Furthermore, they were trained on more data. Initially, records with a price of $750 or above were dropped. This is a much larger threshold compared to $325. However, in an effort to minimize the mean absolute error by removing more outliers, more records were dropped.

The older models were built very similarly to the ones previously mentioned in this report. The `k-train` and `Keras` model were built using the exact code detailed earlier. The only difference is that a slightly higher learning rate was used. Since the earlier models were trained on a dataset of prices below $750 as opposed to $325, the estimated optimal learning rate was different. This learning rate was found to be **`0.012`**.




```
val_mae: 52.1948
```

K-train's *Loss Plot*




```
val_mae: 51.8019
```

Kera's *Loss Plot*

The loss plots of the `k-train` and the `Keras` models are shown above. The mean absolute errors of these models were also shown. The `k-train` model ran for 30 epochs and the `Keras` model ran for 7 epochs. The `k-train` model's loss looked smoother, having a slight over-fit near the end. The `Keras` model's loss was not as smooth since it only ran for 7 epochs. However, relative to the `Keras` model mentioned earlier, it does have better shape.

These models were omitted from this report due to their lower mean absolute errors. But the shape of the loss plots, especially for the `Keras` model, show how these models may still be robust. However, the distribution histogram of **`price`** shown earlier suggests that there are not many AirBnB's that stretch beyond $325 in NYC. Therefore, a model was created on a smaller dataset, excluding many outlier **`price`** records.

VIII. References

ktrain: A Low-Code Library for Augmented Machine Learning
https://jmlr.org/papers/volume23/21-1124/21-1124.pdf

New York City AirBnB Open Data
https://www.kaggle.com/datasets/dgomonov/new-york-city-airbnb-open-data

Predicting Wine Prices from Textual Descriptions
https://nbviewer.org/github/amaiya/ktrain/blob/master/examples/text/text_regression_example.ipynb

Predicting Wine Prices from Textual Descriptions
https://www.kaggle.com/datasets/zynicide/wine-reviews