Predicting Resale Value of Knives from a Texas Government Surplus Store

Using Machine Learning to Support an Ebay Store's Financial Success

Model and Intepret Notebook

Author: Dylan Dey

Overview

[Texas State Surplus Store (https://www.tfc.texas.gov/divisions/supportserv/prog/statesurplus/)](https://www.tfc.texas.gov/divisions/supportserv/prog/statesurplus/)

[What happens to all those items that get confiscated by the TSA? Some end up in a Texas store. (https://www.wfaa.com/article/news/local/what-happens-to-all-those-items-that-get-confiscated-by-the-tsa-some-end-up-in-a-texas-store/287-ba80dac3-d91a-4b28-952a-0aaf4f69ff95)](https://www.wfaa.com/article/news/local/what-happens-to-all-those-items-that-get-confiscated-by-the-tsa-some-end-up-in-a-texas-store/287-ba80dac3-d91a-4b28-952a-0aaf4f69ff95)

[Texas Surplus Store PDF (https://www.tfc.texas.gov/divisions/supportserv/prog/statesurplus/State%20Surplus%20Brochure-one%20bar_rev%201-10-2022.pdf)](https://www.tfc.texas.gov/divisions/supportserv/prog/statesurplus/State%20Surplus%20Brochure-one%20bar_rev%201-10-2022.pdf)

Thousands of people make a living selling pre-owned items on sites like EBay. A good place to locate items for sale is the Texas Facilities Commission collects left behind possessions, salvage, and surplus from Texas state agencies such as DPS, TXDOT, TCEQ, and Texas Parks & Wildlife. Examples of commonly available items include vehicles, furniture, office equipment and supplies, small electronics, and heavy equipment. The goal of this project is to create a predictive model in order to determine the resale value of knives from the Texas State Surplus Store on eBay. Descriptive analysis of over 70K sold knives on eBay in the last 2 years will also be used to examine the profitability of investing in knives from the surplus store.

BUSINESS PROBLEM

[Texas Dave's Knives (https://www.ebay.com/str/texasdave3/Knives/_i.html?store_cat=3393246519)](https://www.ebay.com/str/texasdave3/Knives/_i.html?store_cat=3393246519)

My family has been running a resale shop and selling on Ebay and other sites for years and lately the business has picked up. We are interested in exploring if the most common item sold at the Texas Surplus Store, pocket knives, would be a safe investment. On the surface they seem great for reselling, as they are oftentimes collectible and small enough to be easily shipped.

I have been experimenting with low cost used knives for resale but have not risked a large capital investment in the higher end items. Analyzing past listings on eBay for the top brands available at the Surplus Store could prove useful for gaining insight on whether a larger investment would pay off. Understanding the risks involved in investing capital into different brands of knives and their potential returns will help narrow down what brands to invest in and help reduce excess inventory.

It has been very time consuming and inaccurate trying to find the correct value to list an item for on eBay. Currently when listing we try to identify the specific knife by Google search, and then try to find the same or similar items sold on Ebay or other sites. This “guess and check” method often results in inventory not moving due to overpricing or being sold at a price lower than its true potential profit. Building a model that predicts the value of a pocket knife on eBay could help to easily determine the correct value of the item before a listing is live on the website.

Data Understanding

> There are eight buckets of presorted brand knives that I was interested in exploring from the Texas Surplus Store. The Eight Pocketknife brands and their associated cost at the Texas Surplus Store:

- Benchmade: $45.00
- Buck: $20.00
- Case/Casexx: $20.00
- CRKT: $15.00
- Kershaw: $15.00
- SOG: $15.00
- Spyderco: $30.00
- Victorinox: $20.00

Domain Understading: Cost Breakdown

- padded envelopes: $0.50 per knife

- padded envelopes: $0.50 per knife
- flatrate shipping: $4.45 per knife
- brand knife at surplus store: 15, 20, 30, or 45 dollars per knife
- overhead expenses (gas, cleaning suplies, sharpening supplies, etc): $3.00
- Ebay's comission, with 13% being a reasonable approximation

> A majority of the data was scraped from eBays proprietary Terapeak webapp, as this data goes back 2 years as compared to the API listed data that only goes back 90 days. It is assumed a large enough amount of listed data should approximate sold data well enough to prove useful for this project.

> The target feature for the model to predict is the total price (shipping included) that a knife should be listed on eBay. One model will be using titles and images in order to find potential listings that are undervalued and could be worth investing in. Another model will accept only images as input, as this is an input that can easily be obtained in person at the store. This model will use past sold data of knives on eBay in order to determine within an acceptable amount of error the price it will resale for on eBay (shipping included) using only an image

In [1]:

```
from sklearn.model_selection import train_test_split
import os
from collections import Counter

import pandas as pd
import  json
import requests
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import ast
import re

import nltk
from nltk.corpus import stopwords
import string
from nltk import word_tokenize, FreqDist
from sklearn.feature_extraction.text import TfidfVectorizer


from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Dense, Input, GlobalMaxPooling1D
from tensorflow.keras.layers import LSTM, Embedding, Flatten, GRU
from tensorflow.keras.layers import Conv1D, MaxPooling1D, GlobalMaxPooling2D
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, BatchNormalization
from tensorflow.keras.layers import SimpleRNN
from tensorflow.keras.models import Model
from keras import models
from keras import layers
import tensorflow as tf
from keras.utils import plot_model
from sklearn.metrics import mean_absolute_error
from keras_preprocessing.image import ImageDataGenerator
```

In [2]:

```
#helps see plots in readme
plt.style.use('dark_background')
```

Function Definition

Define functions to import and clean data for modeling.

In [3]:
```
def apply_iqr_filter(df):

    price_Q1 = df['converted_price'].quantile(0.25)
    price_Q3 = df['converted_price'].quantile(0.75)
    price_iqr = price_Q3 - price_Q1

    profit_Q1 = df['profit'].quantile(0.25)
    profit_Q3 = df['profit'].quantile(0.75)
    profit_iqr = profit_Q3 - profit_Q1

    ROI_Q1 = df['ROI'].quantile(0.25)
    ROI_Q3 = df['ROI'].quantile(0.75)
    ROI_iqr = ROI_Q3 - ROI_Q1

    price_upper_limit = price_Q3 + (1.5 * price_iqr)
    price_lower_limit = price_Q1 - (1.5 * price_iqr)

    profit_upper_limit = profit_Q3 + (1.5 * profit_iqr)
    profit_lower_limit = profit_Q1 - (1.5 * profit_iqr)

    ROI_upper_limit = ROI_Q3 + (1.5 * ROI_iqr)
    ROI_lower_limit = ROI_Q1 - (1.5 * ROI_iqr)

#     print(f'Brand: {df.brand[0]}')
#     print(f'price upper limit: ${np.round(price_upper_limit,2)}')
#     print(f'price lower limit: ${np.round(price_lower_limit,2)}')
#     print('-----------------------------------')
#     print(f'profit upper limit: ${np.round(profit_upper_limit,2)}')
#     print(f'profit lower limit: ${np.round(profit_lower_limit,2)}')
#     print('-----------------------------------')
#     print(f'ROI upper limit: {np.round(ROI_upper_limit,2)}%')
#     print(f'ROI lower limit: {np.round(ROI_lower_limit,2)}%')
#     print('-----------------------------------')


    new_df = df[(df['converted_price'] < price_upper_limit) &
                (df['converted_price'] > price_lower_limit) &
                (df['profit'] < profit_upper_limit) &
                (df['ROI'] > profit_lower_limit) &
                (df['profit'] < ROI_upper_limit) &
                (df['ROI'] > ROI_lower_limit)]

    return new_df
#download jpg urls from dataFrame
def download(row):
    filename = os.path.join(root_folder, str(row.name) + im_extension)

# create folder if it doesn't exist
    os.makedirs(os.path.dirname(filename), exist_ok=True)

    url = row.Image
#     print(f"Downloading {url} to {filename}")

    try:
        r = requests.get(url, allow_redirects=True)
        with open(filename, 'wb') as f:
            f.write(r.content)
    except:
        print(f'{filename} error')



# This function removes noisy data
#lots/sets/groups of knives can
#confuse the model from predicting
#the appropriate value of individual knives
def data_cleaner(df):
    lot = re.compile('(?<!-\S)lot(?![^\s.,:?!])')
    group = re.compile('(group)')
    is_set = re.compile('(?<!-\S)set(?![^\s.,?!])')
    df['title'] = df['title'].str.lower()
    trim_list = [lot,group,is_set]
    for item in trim_list:
        df.loc[df['title'].apply(lambda x: re.search(item, x)).notnull(), 'trim'] = 1
    to_drop = df.loc[df['trim'] == 1].index
    df.drop(to_drop, inplace=True)
    df.drop('trim', axis=1, inplace=True)

    return df


#take raw data and prepare it for modeling
def prepare_listed(listed_data_df):
    listed_used_knives = listed_data_df.loc[listed_data_df['condition'] != 1000.0]
    listed_used_knives = data_cleaner(listed_used_knives.copy())
    listed_used_knives.reset_index(drop=True, inplace=True)
```

```
    return listed_used_knives

#take raw data and prepare it for modeling
def prepare_tera_df(df, x, overhead_cost=3):
    df['price_in_US'] = df['price_in_US'].str.replace("$", "")
    df['price_in_US'] = df['price_in_US'].str.replace(",", "")
    df['price_in_US'] = df['price_in_US'].apply(float)

    df['shipping_cost'] = df['shipping_cost'].str.replace("$", "")
    df['shipping_cost'] = df['shipping_cost'].str.replace(",", "")
    df['shipping_cost'] = df['shipping_cost'].apply(float)

    df['brand'] = list(bucket_dict.keys())[x]
    df['converted_price'] = (df['price_in_US'] + df['shipping_cost'])
    df['cost'] = list(bucket_dict.values())[x] + overhead_cost + 4.95
    df['profit'] = ((df['converted_price']*.87) -  df['cost'])
    df['ROI'] = (df['profit']/ df['cost'])*100.0

    return df


def avg_word_len(x):
    words = x.split()
    word_len = 0
    for word in words:
        word_len += len(word)

    return word_len / len(words)
```

Load Data

In [4]:
```
cd ..
```

```
/Users/dylandey/Documents/GitHub/Neural_Network_Predicting_Reseller_Success_Ebay
```

In [5]:
```python
#load Finding API data
df_bench = pd.read_csv("listed_data/df_bench.csv")
df_buck = pd.read_csv("listed_data/df_buck.csv")
df_case = pd.read_csv("listed_data/df_case.csv")
df_caseXX = pd.read_csv("listed_data/df_CaseXX.csv")
df_crkt = pd.read_csv("listed_data/df_crkt.csv")
df_kersh = pd.read_csv("listed_data/df_kershaw.csv")
df_sog = pd.read_csv("listed_data/df_sog.csv")
df_spyd = pd.read_csv("listed_data/df_spyderco.csv")
df_vict = pd.read_csv("listed_data/df_victorinox.csv")


#Load scraped terapeak sold data
sold_bench = pd.read_csv("terapeak_data/bench_scraped2.csv")
sold_buck1 = pd.read_csv("terapeak_data/buck_scraped2.csv")
sold_buck2 = pd.read_csv("terapeak_data/buck_scraped2_reversed.csv")
sold_case = pd.read_csv("terapeak_data/case_scraped2.csv")
sold_caseXX1 = pd.read_csv("terapeak_data/caseXX_scraped2.csv")
sold_caseXX2 = pd.read_csv("terapeak_data/caseXX2_reversed.csv")
sold_crkt = pd.read_csv("terapeak_data/crkt_scraped.csv")
sold_kershaw1 = pd.read_csv("terapeak_data/kershaw_scraped2.csv")
sold_kershaw2 = pd.read_csv("terapeak_data/kershaw_scraped2_reversed.csv")
sold_sog = pd.read_csv("terapeak_data/SOG_scraped2.csv")
sold_spyd = pd.read_csv("terapeak_data/spyd_scraped2.csv")
sold_vict1 = pd.read_csv("terapeak_data/vict_scraped.csv")
sold_vict2 = pd.read_csv("terapeak_data/vict_reversed.csv")

sold_list = [sold_bench,sold_buck1,
             sold_buck2,sold_case,
             sold_caseXX1,sold_caseXX2,
             sold_crkt,sold_kershaw1,
             sold_kershaw2,sold_sog,
             sold_spyd, sold_vict1,
             sold_vict2]


listed_df = pd.concat([df_bench,df_buck,
                       df_case,df_caseXX,
                       df_crkt,df_kersh,
                       df_sog,df_spyd,
                       df_vict])

used_listed = prepare_listed(listed_df)

bucket_dict = {'benchmade': 45.0,
               'buck': 20.0,
               'case': 20.0,
               'crkt': 15.0,
               'kershaw': 15.0,
               'sog': 15.0,
               'spyderco': 30.0,
               'victorinox': 20.0
               }
```

Prepare Data

In [6]:
```python
for dataframe in sold_list:
    dataframe.rename({'Text': 'title',
                      'shipping_': 'shipping_cost'},
                     axis=1, inplace=True)

    dataframe['date_sold'] = pd.to_datetime(dataframe['date_sold'])

#limited out at 10K columns while scraping. Combine dataframes that went over 10K.
sold_buck = pd.concat([sold_buck1,sold_buck2])
sold_caseXX = pd.concat([sold_caseXX1,sold_caseXX2])
sold_kershaw = pd.concat([sold_kershaw1,sold_kershaw2])
sold_vict = pd.concat([sold_vict1,sold_vict2])

#apply function to remove characters from price
#and create profit/ROI features
sold_bench = prepare_tera_df(sold_bench, 0)
sold_buck = prepare_tera_df(sold_buck, 1)
sold_case = prepare_tera_df(sold_case, 2)
sold_caseXX = prepare_tera_df(sold_caseXX, 2)
sold_crkt = prepare_tera_df(sold_crkt, 3)
sold_kershaw = prepare_tera_df(sold_kershaw, 4)
sold_sog = prepare_tera_df(sold_sog, 5)
sold_spyd = prepare_tera_df(sold_spyd, 6)
sold_vict = prepare_tera_df(sold_vict, 7)
```

In [7]:
```
#lowercase and strip titles and remove duplicates
for dataframe in sold_list:
    dataframe['title'] = dataframe['title'].str.lower()
    dataframe['title'] = dataframe['title'].str.strip()
    dataframe.drop_duplicates(
        subset = ['date_sold','price_in_US',
                  'shipping_cost'],
        keep = 'last', inplace=True)
```

In [8]:
```
sold_df = pd.concat([sold_bench, sold_buck,
                     sold_case, sold_caseXX,
                     sold_crkt, sold_kershaw,
                     sold_sog, sold_spyd,
                     sold_vict])
#remove lots
sold_knives = data_cleaner(sold_df).copy()

#combine data
df = pd.concat([sold_knives,used_listed]).copy()
df['Image'].fillna(df['pictureURLLarge'], inplace=True)

#apply IQR filtering
df = apply_iqr_filter(df).copy()
df.reset_index(drop=True, inplace=True)
```

Text Preprocessing

In [9]:
```
#load stopwords
from nltk.corpus import stopwords
stop = stopwords.words('english')
#remove any special characters
def remove_special_char(x):
    pattern = r'[^a-zA-z0-9\s]'
    text = re.sub(pattern, '', x)
    return text

def remove_punctuations(x):
    x.translate(str.maketrans('', '', string.punctuation))
    return x
#apply above functions to dataframe
def apply_text_prep(df):

    df['title'] = df['title'].apply(remove_punctuations)
    df['title'] = df['title'].apply(remove_special_char)
    #A lot of the strings had duplicate phrases
    #create a set on split strings in order to
    #only get unique words in each title
    df['title'] = df['title'].apply(lambda s: ' '.join(list(set(s.split()))))


    df['title_len'] = df['title'].apply(lambda x: len(x))
    df['word_count'] = df['title'].apply(lambda x: len(x.split()))
    df['avg_word_len'] = df['title'].apply(lambda x: avg_word_len(x))

    stop = stopwords.words('english')

    df['title_nostop'] = df['title'].apply(lambda x: ' '.join([word for word in x.split() if word not in stop]))

    return df
df = apply_text_prep(df)
```

Model

Neural network with "title" column as input

A title is something that is essential when posting an item for sale on eBay. A lot of sellers use the same format for titles depending on the product. Titles on eBay are essentially a list of keywords in order to promote it higher in the algorathim. Each knife brand and model combination have features that are desirable to collectors that sellers want to put right up at the top of their listing.

Pricing a certain pocketknife correctly when listing it, however, is time consuming and requires scrolling through webpages trying to find the most similar knives and guessing what would be competitive while hoping that none of your filters reset or you didnt miss a cruicial one. Pricing the item correctly is very important. If you post the item for sale too low, it might sell quickly and avoid excess inventory piling up, but it is also leaving revenue on the table. Listing the knife too high would mean it may not move off the shelf at all.

Creating a model to predict the price to list a knife for sale and accurately determing it's true resale value can not only save time and make listing items more efficient, it can also optimize for an equilibrium between excess inventory and lost revenue.

A number of different vectorization methods and modeling was tested for this project. The appendix has some brief examples of training a Random Forest model with feature importances and TfIDF vectorization. This type of vectorization, like one-hot tend to be very sparse for NLP. An advantage over this type of reperesentation is Word-embeddings: a learned represntation for text where words that have similar meaning have similar representation. Word embeddings are dense, lower-dimensional and learned from the data.

Recurrent Neural Networks are a type of Neural Network in which the output from the previous step is fed as input to the current step, making it well suited for handling sequence data.

RNNs are particularly useful if the prediction has to be at word-level, as it stores the information for current feature as well neighboring features for prediction. A RNN maintains a memory based on history information. A simple RNN, however, has a "short" memory and can often lead into problems with vanishing gradients. LSTM were created to use gates in order to filter for feature importance in order to combat this problem. This makes them better at finding and exposing long range dependencies in data which is imperative for sentence structures.

LSTMs are a bit more complex than GRU, with 3 gates compared to 2 for the GRU. GRUs are relatively new compared to LSTMs and their performance is on par with them, but computationally more efficient (as pointed out, they have a less complex structure).

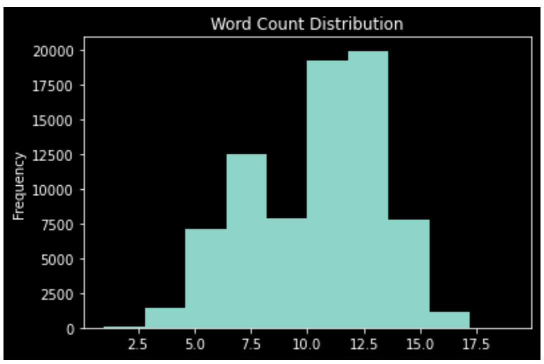
See below for model creation and hyperparameter tuning for different RNN architectures, inluding LSTMs and GRUs.

In [10]:
```
from tensorboard.plugins.hparams import api as hp
```

In [ ]:
```

```

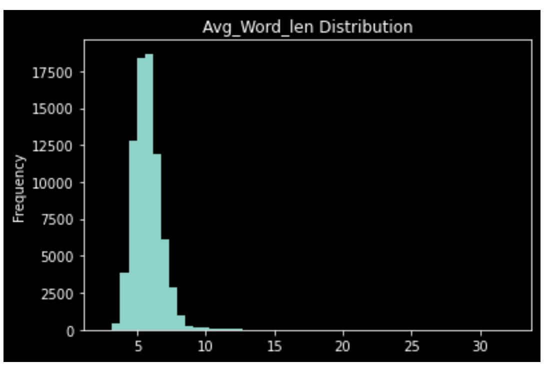
In [11]:
```
df['word_count'].plot(kind = 'hist', title = 'Word Count Distribution')
```

Out[11]: <AxesSubplot:title={'center':'Word Count Distribution'}, ylabel='Frequency'>

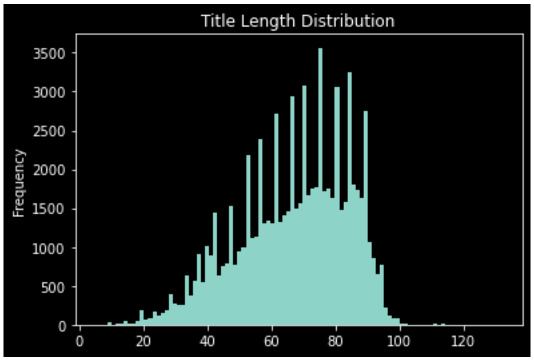


In [12]:
```
df['avg_word_len'].plot(kind='hist', bins = 50, title = 'Avg_Word_len Distribution')
```

Out[12]: <AxesSubplot:title={'center':'Avg_Word_len Distribution'}, ylabel='Frequency'>



In [13]:
```
df['title_len'].plot(kind='hist', bins= 100,title = 'Title Length Distribution');
```



simpleRNN

The mean max sequence for is not that long for eBay titles, thus a simpleRNN was tested to see if architecture with gates could be avoided to increase efficiency. Hyperparameter tuning is shown below. The best performing model had 100.00 units and dropout=0.4 in the RNN layer with a test MAE of 14.869. Different variations of max sequence length, vocab size, and number of embedding features was tested as well. Performed well but not as well as GRU model.

In [30]:
```
df_title = df.loc[:, ['title_nostop', 'converted_price']]


df_title.rename({'title_nostop': 'data',
                 'converted_price': 'labels'},
                axis=1, inplace=True)
```

In [31]:
```
df_title.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 76914 entries, 0 to 76913
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   data    76914 non-null  object 
 1   labels  76914 non-null  float64
dtypes: float64(1), object(1)
memory usage: 1.2+ MB
```

Split data into training and testing sets and the split the testing set to create equal size val and train sizes.

In [32]:
```
# df_title['labels'] = (df_title['labels']/mean_price)
y = df_title['labels'].values
X_train, X_test, y_train, y_test = train_test_split(df_title['data'],
                                                    y,
                                                    test_size=0.3,
                                                    random_state=42)
```

In [33]:
```
X_val, X_test, y_val, y_test = train_test_split(X_test,
                                                y_test,
                                                test_size=0.5,
                                                random_state=42)
```

In [ ]:
```

```

In [34]:
```
#Vectorize vocab
voc_size = 30000
max_len = 11
embedding_features = 100
tokenizer = Tokenizer(num_words=voc_size, oov_token = '<OOV>')
tokenizer.fit_on_texts(X_train)
sequences_train = tokenizer.texts_to_sequences(X_train)
sequences_val = tokenizer.texts_to_sequences(X_val)
sequences_test = tokenizer.texts_to_sequences(X_test)
```

In [35]:
```
#add padding to ensure all inputs are the same size
data_train = pad_sequences(sequences_train, maxlen=max_len, padding= 'post', truncating = 'post')
data_val = pad_sequences(sequences_val, maxlen=max_len, padding= 'post', truncating = 'post')
data_test = pad_sequences(sequences_test, maxlen=max_len, padding= 'post', truncating = 'post')
```

In [36]:
```
data_train.shape
```

Out[36]:
```
(53839, 11)
```

In [37]:
```
#set values for hyperparamter testing of simpleRNN model
HP_NUM_UNITS = hp.HParam('units', hp.Discrete([64,100,300,600]))
HP_DROPOUT = hp.HParam('dropout', hp.RealInterval(0.2,0.4))


METRIC_MAE = 'MAE'
#set metrics
with tf.summary.create_file_writer('logs/hparam_tuning').as_default():
    hp.hparams_config(
        hparams=[HP_NUM_UNITS, HP_DROPOUT],
        metrics=[hp.Metric(METRIC_MAE, display_name='MAE')],
    )
```

In [38]:
```
#create model for hyperparameter testing
def train_test_model(hparams):
    model = models.Sequential()
    model.add(Embedding(voc_size, embedding_features, input_length = max_len))
    model.add(SimpleRNN(hparams[HP_NUM_UNITS],dropout=(hparams[HP_DROPOUT])))
    model.add(Dense(1, activation = 'linear'))
    model.compile(
      optimizer='Adam',
      loss='MSE',
      metrics=['MAE'],
    )
    model.fit(data_train,
              y_train,
              epochs=5,
              validation_data=(data_val, y_val)
             )
    _, MSE = model.evaluate(data_test, y_test)
    return MSE
```

In [39]:
```
#write files
def run(run_dir, hparams):
    with tf.summary.create_file_writer(run_dir).as_default():
        hp.hparams(hparams)  # record the values used in this trial
        MSE = train_test_model(hparams)
        tf.summary.scalar(METRIC_MAE, MSE, step=1)
```

In [40]:
```
# %load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

In [41]:
```
# rm -rf ./logs/
```

In [42]:
```
#run hyperparamter testing for simpleRNN Model
session_num = 0

for num_units in HP_NUM_UNITS.domain.values:
    for dropout_rate in (HP_DROPOUT.domain.min_value, HP_DROPOUT.domain.max_value):
        hparams = {
          HP_NUM_UNITS: num_units,
          HP_DROPOUT: dropout_rate,
        }
        run_name = "run-%d" % session_num
        print('--- Starting trial: %s' % run_name)
        print({h.name: hparams[h] for h in hparams})
        run('logs/hparam_tuning/' + run_name, hparams)
        session_num += 1
```

```
--- Starting trial: run-0
{'units': 64, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 27s 16ms/step - loss: 1545.4021 - MAE: 28.9707 - val_loss: 1283.7516 - v
al_MAE: 28.4177
Epoch 2/5
1683/1683 [==============================] - 28s 16ms/step - loss: 1214.4720 - MAE: 26.8614 - val_loss: 924.2294 - va
l_MAE: 22.4981
Epoch 3/5
1683/1683 [==============================] - 27s 16ms/step - loss: 721.0527 - MAE: 18.8697 - val_loss: 596.2415 - val
_MAE: 18.1847
Epoch 4/5
1683/1683 [==============================] - 27s 16ms/step - loss: 459.7035 - MAE: 14.7469 - val_loss: 499.7278 - val
_MAE: 15.6992
Epoch 5/5
1683/1683 [==============================] - 27s 16ms/step - loss: 408.5789 - MAE: 13.8503 - val_loss: 733.4108 - val
_MAE: 20.3749
361/361 [==============================] - 0s 891us/step - loss: 708.7606 - MAE: 20.0188
--- Starting trial: run-1
{'units': 64, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 27s 16ms/step - loss: 1574.5758 - MAE: 29.1562 - val_loss: 1286.9081 - v
al_MAE: 28.3603
Epoch 2/5
1683/1683 [==============================] - 27s 16ms/step - loss: 1373.3777 - MAE: 28.6362 - val_loss: 1184.4480 - v
al_MAE: 26.5988
Epoch 3/5
1683/1683 [==============================] - 27s 16ms/step - loss: 803.9194 - MAE: 20.3736 - val_loss: 633.8552 - val
_MAE: 18.2170
Epoch 4/5
1683/1683 [==============================] - 27s 16ms/step - loss: 524.3430 - MAE: 16.1071 - val_loss: 561.8765 - val
_MAE: 17.3872
Epoch 5/5
1683/1683 [==============================] - 27s 16ms/step - loss: 432.9792 - MAE: 14.5322 - val_loss: 495.1504 - val
_MAE: 15.7714
361/361 [==============================] - 0s 886us/step - loss: 496.8597 - MAE: 15.7403
--- Starting trial: run-2
{'units': 100, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 28s 16ms/step - loss: 1462.1542 - MAE: 28.8304 - val_loss: 1327.4998 - v
al_MAE: 28.5528
Epoch 2/5
1683/1683 [==============================] - 28s 16ms/step - loss: 977.8948 - MAE: 22.7316 - val_loss: 657.8311 - val
_MAE: 19.3510
Epoch 3/5
1683/1683 [==============================] - 28s 17ms/step - loss: 522.1215 - MAE: 15.9999 - val_loss: 515.9554 - val
_MAE: 16.2598
Epoch 4/5
1683/1683 [==============================] - 30s 18ms/step - loss: 399.6503 - MAE: 13.7620 - val_loss: 494.2186 - val
_MAE: 15.4508
Epoch 5/5
1683/1683 [==============================] - 28s 17ms/step - loss: 341.4431 - MAE: 12.5825 - val_loss: 510.8235 - val
_MAE: 14.9544
361/361 [==============================] - 0s 1ms/step - loss: 515.0785 - MAE: 15.0194
--- Starting trial: run-3
{'units': 100, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 28s 17ms/step - loss: 1471.0468 - MAE: 28.8489 - val_loss: 1286.2722 - v
al_MAE: 28.4754
Epoch 2/5
1683/1683 [==============================] - 28s 16ms/step - loss: 932.9797 - MAE: 22.3484 - val_loss: 733.8779 - val
_MAE: 21.0845
Epoch 3/5
1683/1683 [==============================] - 27s 16ms/step - loss: 535.9272 - MAE: 16.2933 - val_loss: 502.5784 - val
_MAE: 15.6724
Epoch 4/5
1683/1683 [==============================] - 27s 16ms/step - loss: 416.0043 - MAE: 14.1715 - val_loss: 543.1646 - val
_MAE: 15.5629
Epoch 5/5
1683/1683 [==============================] - 27s 16ms/step - loss: 356.2544 - MAE: 12.9445 - val_loss: 504.3145 - val
_MAE: 14.9497
361/361 [==============================] - 0s 995us/step - loss: 501.2480 - MAE: 14.8687
--- Starting trial: run-4
{'units': 300, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 53s 32ms/step - loss: 2610.0273 - MAE: 37.4391 - val_loss: 1588.3590 - v
al_MAE: 28.3926
Epoch 2/5
1683/1683 [==============================] - 55s 32ms/step - loss: 1153.3793 - MAE: 25.6337 - val_loss: 856.4802 - va
l_MAE: 22.7115
Epoch 3/5
1683/1683 [==============================] - 55s 32ms/step - loss: 707.8936 - MAE: 19.3850 - val_loss: 624.0355 - val
_MAE: 18.0612
Epoch 4/5
1683/1683 [==============================] - 57s 34ms/step - loss: 541.5103 - MAE: 16.6095 - val_loss: 638.9877 - val
_MAE: 18.7663
```

```
Epoch 5/5
1683/1683 [==============================] - 56s 33ms/step - loss: 473.9166 - MAE: 15.3908 - val_loss: 572.2099 - val
_MAE: 17.9207
361/361 [==============================] - 1s 2ms/step - loss: 561.5010 - MAE: 17.7754
--- Starting trial: run-5
{'units': 300, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 56s 33ms/step - loss: 1335.2867 - MAE: 28.5574 - val_loss: 1294.5747 - v
al_MAE: 28.6047
Epoch 2/5
1683/1683 [==============================] - 56s 34ms/step - loss: 1460.6228 - MAE: 28.8012 - val_loss: 1137.7601 - v
al_MAE: 26.2833
Epoch 3/5
1683/1683 [==============================] - 57s 34ms/step - loss: 935.4930 - MAE: 22.9057 - val_loss: 808.2964 - val
_MAE: 21.2044
Epoch 4/5
1683/1683 [==============================] - 56s 33ms/step - loss: 709.0465 - MAE: 19.4407 - val_loss: 726.0751 - val
_MAE: 20.5629
Epoch 5/5
1683/1683 [==============================] - 56s 34ms/step - loss: 570.8647 - MAE: 17.1962 - val_loss: 594.7704 - val
_MAE: 17.7948
361/361 [==============================] - 1s 2ms/step - loss: 605.8956 - MAE: 17.8754
--- Starting trial: run-6
{'units': 600, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 65s 39ms/step - loss: 1304.3243 - MAE: 28.5447 - val_loss: 1300.5840 - v
al_MAE: 27.8199
Epoch 2/5
1683/1683 [==============================] - 65s 39ms/step - loss: 1458.1661 - MAE: 29.0395 - val_loss: 1396.2083 - v
al_MAE: 31.5006
Epoch 3/5
1683/1683 [==============================] - 65s 39ms/step - loss: 776.5053 - MAE: 20.4605 - val_loss: 621.6205 - val
_MAE: 17.7440
Epoch 4/5
1683/1683 [==============================] - 65s 39ms/step - loss: 527.6366 - MAE: 16.3058 - val_loss: 549.9774 - val
_MAE: 16.4058
Epoch 5/5
1683/1683 [==============================] - 65s 39ms/step - loss: 448.2758 - MAE: 14.8539 - val_loss: 525.6780 - val
_MAE: 15.9323
361/361 [==============================] - 1s 3ms/step - loss: 515.2919 - MAE: 15.8030
--- Starting trial: run-7
{'units': 600, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 65s 39ms/step - loss: 1326.9526 - MAE: 28.6390 - val_loss: 1374.0564 - v
al_MAE: 27.3407
Epoch 2/5
1683/1683 [==============================] - 66s 39ms/step - loss: 1445.6423 - MAE: 28.4284 - val_loss: 1115.3411 - v
al_MAE: 25.2061
Epoch 3/5
1683/1683 [==============================] - 66s 39ms/step - loss: 1016.0386 - MAE: 24.1690 - val_loss: 843.2446 - va
l_MAE: 22.1157
Epoch 4/5
1683/1683 [==============================] - 66s 39ms/step - loss: 728.7784 - MAE: 19.7069 - val_loss: 669.1804 - val
_MAE: 18.1211
Epoch 5/5
1683/1683 [==============================] - 66s 39ms/step - loss: 590.9900 - MAE: 17.4051 - val_loss: 610.2494 - val
_MAE: 16.9858
361/361 [==============================] - 1s 4ms/step - loss: 604.2512 - MAE: 16.9022
```

[simpleRNN Hyperparamter Optimization (https://tensorboard.dev/experiment/sngivEMGR2KSgnBDe3ncLQ)](https://tensorboard.dev/experiment/sngivEMGR2KSgnBDe3ncLQ)

GRU

An alternative to creating a simpleRNN layer after embedding is to use a GRU layer. The architecture of this layer is more efficient than an LSTM, but it is similar in its ability to filter features by importance before continuing in the network. This ability helps to fight short term memory and vanishing gradients. This usually works well with smaller sample size than more robust LSTM models. However, for this project the GRU model performed the best with a test MAE of 14.28.

In [19]:

```
#Vectorize vocab
voc_size = 30000
max_len = 11
embedding_features = 100
tokenizer = Tokenizer(num_words=voc_size, oov_token = '<OOV>')
tokenizer.fit_on_texts(X_train)
sequences_train = tokenizer.texts_to_sequences(X_train)
sequences_val = tokenizer.texts_to_sequences(X_val)
sequences_test = tokenizer.texts_to_sequences(X_test)
```

In [20]:
```
#add padding to ensure all inputs are the same size
data_train = pad_sequences(sequences_train, maxlen=max_len, padding= 'post', truncating = 'post')
data_val = pad_sequences(sequences_val, maxlen=max_len, padding= 'post', truncating = 'post')
data_test = pad_sequences(sequences_test, maxlen=max_len, padding= 'post', truncating = 'post')
```

In [21]:
```
data_train.shape
```

Out[21]: (53839, 11)

In [22]:
```
#setup units for hyperparameter testing
HP_NUM_UNITS = hp.HParam('units', hp.Discrete([64,100,300,600]))
HP_DROPOUT = hp.HParam('dropout', hp.RealInterval(0.2,0.4))


METRIC_MAE = 'MAE'
#log performance
with tf.summary.create_file_writer('logs/hparam_tuning').as_default():
    hp.hparams_config(
        hparams=[HP_NUM_UNITS, HP_DROPOUT],
        metrics=[hp.Metric(METRIC_MAE, display_name='MAE')],
    )
```

In [23]:
```
#define model for hyperparameter testing
def train_test_model(hparams):
    model = models.Sequential()
    model.add(Embedding(voc_size, embedding_features, input_length = max_len))
    model.add(GRU(hparams[HP_NUM_UNITS],dropout=(hparams[HP_DROPOUT])))
    model.add(Dense(1, activation = 'linear'))
    model.compile(
      optimizer='Adam',
      loss='MSE',
      metrics=['MAE'],
    )
    model.fit(data_train,
              y_train,
              epochs=5,
              validation_data=(data_val, y_val)
             )
    _, MSE = model.evaluate(data_test, y_test)
    return MSE
```

In [24]:
```
#write files
def run(run_dir, hparams):
    with tf.summary.create_file_writer(run_dir).as_default():
        hp.hparams(hparams)  # record the values used in this trial
        MSE = train_test_model(hparams)
        tf.summary.scalar(METRIC_MAE, MSE, step=1)
```

In [25]:
```
# %load_ext tensorboard
```

In [26]:
```
# rm -rf ./logs/
```

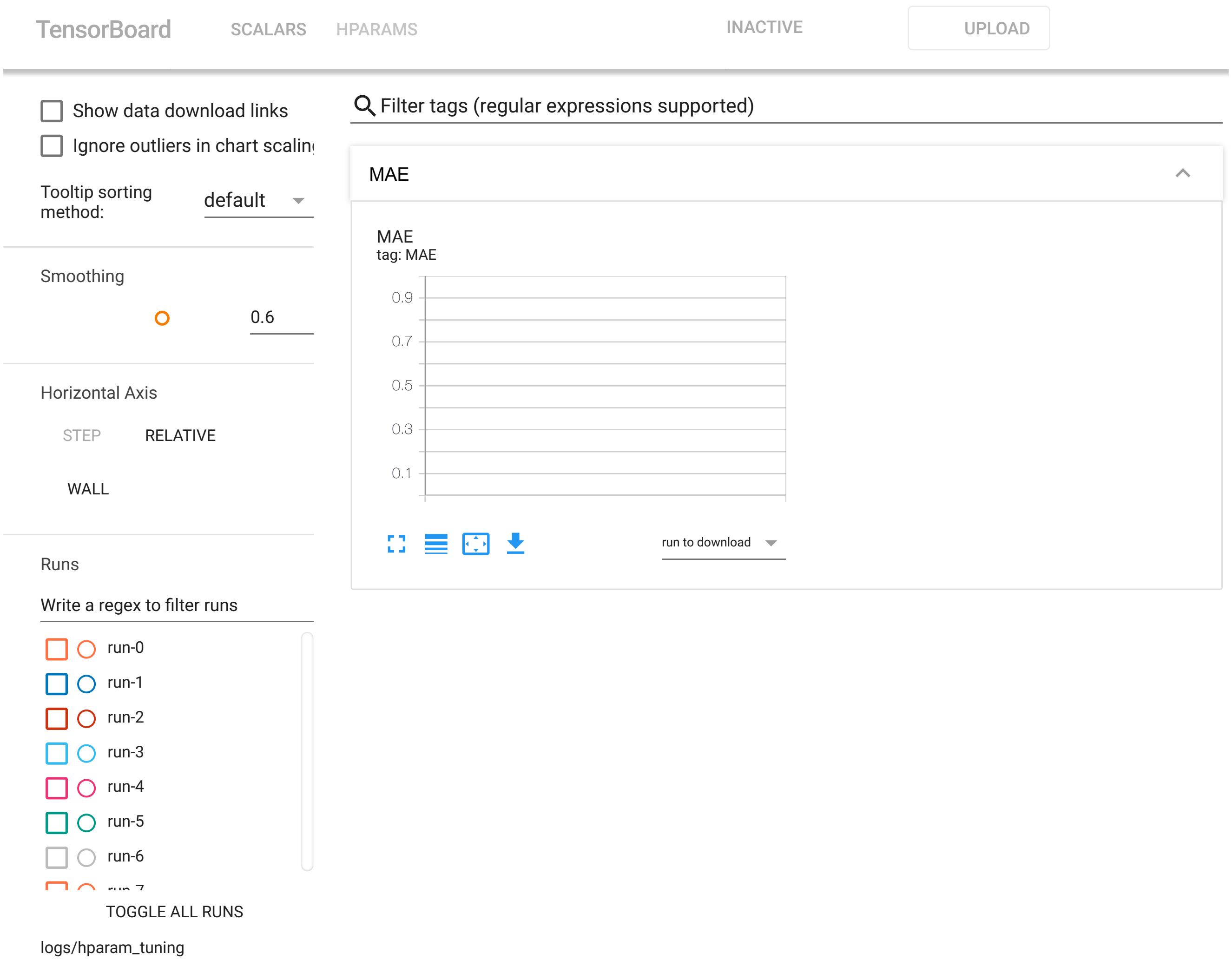
In [27]:
```
#run hyperparamter testing for GRU Model
session_num = 0

for num_units in HP_NUM_UNITS.domain.values:
    for dropout_rate in (HP_DROPOUT.domain.min_value, HP_DROPOUT.domain.max_value):
        hparams = {
          HP_NUM_UNITS: num_units,
          HP_DROPOUT: dropout_rate,
        }
        run_name = "run-%d" % session_num
        print('--- Starting trial: %s' % run_name)
        print({h.name: hparams[h] for h in hparams})
        run('logs/hparam_tuning/' + run_name, hparams)
        session_num += 1
```

```
--- Starting trial: run-0
{'units': 64, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 33s 20ms/step - loss: 1328.4364 - MAE: 25.5363 - val_loss: 696.2332 - va
l_MAE: 18.0849
Epoch 2/5
1683/1683 [==============================] - 32s 19ms/step - loss: 536.1657 - MAE: 15.7684 - val_loss: 488.1271 - val
_MAE: 15.4033
Epoch 3/5
1683/1683 [==============================] - 32s 19ms/step - loss: 372.3362 - MAE: 13.0475 - val_loss: 467.1312 - val
_MAE: 15.2950
Epoch 4/5
1683/1683 [==============================] - 32s 19ms/step - loss: 300.6385 - MAE: 11.5707 - val_loss: 452.5726 - val
_MAE: 14.5839
Epoch 5/5
1683/1683 [==============================] - 33s 20ms/step - loss: 258.0138 - MAE: 10.5685 - val_loss: 461.5382 - val
_MAE: 14.3966
361/361 [==============================] - 0s 1ms/step - loss: 461.8700 - MAE: 14.3679
--- Starting trial: run-1
{'units': 64, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 33s 19ms/step - loss: 1372.8239 - MAE: 26.2185 - val_loss: 725.1318 - va
l_MAE: 18.2312
Epoch 2/5
1683/1683 [==============================] - 33s 20ms/step - loss: 555.3496 - MAE: 16.0475 - val_loss: 492.6044 - val
_MAE: 15.3518
Epoch 3/5
1683/1683 [==============================] - 33s 19ms/step - loss: 392.1455 - MAE: 13.4332 - val_loss: 460.1981 - val
_MAE: 14.6972
Epoch 4/5
1683/1683 [==============================] - 33s 20ms/step - loss: 320.9182 - MAE: 11.9877 - val_loss: 452.2062 - val
_MAE: 14.3584
Epoch 5/5
1683/1683 [==============================] - 34s 20ms/step - loss: 278.1917 - MAE: 11.0076 - val_loss: 456.2365 - val
_MAE: 14.4011
361/361 [==============================] - 0s 1ms/step - loss: 454.7992 - MAE: 14.4289
--- Starting trial: run-2
{'units': 100, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 40s 24ms/step - loss: 1179.7050 - MAE: 24.3715 - val_loss: 591.7100 - va
l_MAE: 16.9928
Epoch 2/5
1683/1683 [==============================] - 40s 24ms/step - loss: 475.1254 - MAE: 14.9670 - val_loss: 480.8693 - val
_MAE: 14.6651
Epoch 3/5
1683/1683 [==============================] - 41s 24ms/step - loss: 347.6246 - MAE: 12.6015 - val_loss: 456.0936 - val
_MAE: 14.7398
Epoch 4/5
1683/1683 [==============================] - 41s 25ms/step - loss: 286.3542 - MAE: 11.2127 - val_loss: 456.3931 - val
_MAE: 14.5948
Epoch 5/5
1683/1683 [==============================] - 39s 23ms/step - loss: 248.6496 - MAE: 10.3452 - val_loss: 454.9020 - val
_MAE: 14.4236
361/361 [==============================] - 1s 1ms/step - loss: 450.9122 - MAE: 14.3617
--- Starting trial: run-3
{'units': 100, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 40s 24ms/step - loss: 1210.3173 - MAE: 24.7316 - val_loss: 596.8922 - va
l_MAE: 16.8955
Epoch 2/5
1683/1683 [==============================] - 40s 24ms/step - loss: 489.5698 - MAE: 15.2196 - val_loss: 471.4067 - val
_MAE: 15.0673
Epoch 3/5
1683/1683 [==============================] - 40s 24ms/step - loss: 368.0865 - MAE: 12.9956 - val_loss: 450.6420 - val
_MAE: 14.5353
Epoch 4/5
1683/1683 [==============================] - 40s 24ms/step - loss: 309.7200 - MAE: 11.7717 - val_loss: 445.8606 - val
_MAE: 14.3562
Epoch 5/5
1683/1683 [==============================] - 39s 23ms/step - loss: 273.9073 - MAE: 10.9282 - val_loss: 458.9141 - val
_MAE: 14.2353
361/361 [==============================] - 1s 1ms/step - loss: 459.7435 - MAE: 14.2825
--- Starting trial: run-4
{'units': 300, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 75s 45ms/step - loss: 1074.4812 - MAE: 24.5460 - val_loss: 535.6713 - va
l_MAE: 16.5163
Epoch 2/5
1683/1683 [==============================] - 74s 44ms/step - loss: 459.8755 - MAE: 14.8774 - val_loss: 468.3544 - val
_MAE: 15.3909
Epoch 3/5
1683/1683 [==============================] - 73s 43ms/step - loss: 354.5280 - MAE: 12.7766 - val_loss: 463.1013 - val
_MAE: 14.4731
Epoch 4/5
1683/1683 [==============================] - 74s 44ms/step - loss: 298.2746 - MAE: 11.5296 - val_loss: 475.8996 - val
_MAE: 14.8959
```

```
Epoch 5/5
1683/1683 [==============================] - 74s 44ms/step - loss: 259.9752 - MAE: 10.6780 - val_loss: 458.3766 - val
_MAE: 14.2217
361/361 [==============================] - 2s 5ms/step - loss: 465.5924 - MAE: 14.3413
--- Starting trial: run-5
{'units': 300, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 74s 44ms/step - loss: 1072.6168 - MAE: 24.6027 - val_loss: 557.0251 - va
l_MAE: 16.4730
Epoch 2/5
1683/1683 [==============================] - 73s 44ms/step - loss: 464.9824 - MAE: 14.9727 - val_loss: 466.4098 - val
_MAE: 14.9963
Epoch 3/5
1683/1683 [==============================] - 75s 45ms/step - loss: 363.4417 - MAE: 12.9132 - val_loss: 463.4724 - val
_MAE: 15.0004
Epoch 4/5
1683/1683 [==============================] - 76s 45ms/step - loss: 309.6641 - MAE: 11.7731 - val_loss: 455.5704 - val
_MAE: 14.3623
Epoch 5/5
1683/1683 [==============================] - 75s 44ms/step - loss: 271.5936 - MAE: 10.9213 - val_loss: 465.6620 - val
_MAE: 15.0551
361/361 [==============================] - 2s 6ms/step - loss: 467.6682 - MAE: 15.0540
--- Starting trial: run-6
{'units': 600, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 124s 73ms/step - loss: 906.4629 - MAE: 22.2624 - val_loss: 544.9484 - va
l_MAE: 16.5200
Epoch 2/5
1683/1683 [==============================] - 123s 73ms/step - loss: 473.9754 - MAE: 15.1554 - val_loss: 489.7672 - va
l_MAE: 15.9014
Epoch 3/5
1683/1683 [==============================] - 124s 73ms/step - loss: 389.7813 - MAE: 13.5291 - val_loss: 471.4116 - va
l_MAE: 15.2101
Epoch 4/5
1683/1683 [==============================] - 124s 74ms/step - loss: 329.8672 - MAE: 12.2954 - val_loss: 468.9655 - va
l_MAE: 15.2950
Epoch 5/5
1683/1683 [==============================] - 124s 74ms/step - loss: 294.2294 - MAE: 11.5079 - val_loss: 468.7699 - va
l_MAE: 14.4805
361/361 [==============================] - 6s 16ms/step - loss: 474.7774 - MAE: 14.6248
--- Starting trial: run-7
{'units': 600, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 123s 73ms/step - loss: 902.2408 - MAE: 22.2192 - val_loss: 527.5868 - va
l_MAE: 16.4546
Epoch 2/5
1683/1683 [==============================] - 124s 74ms/step - loss: 460.4124 - MAE: 14.9230 - val_loss: 487.9597 - va
l_MAE: 15.2907
Epoch 3/5
1683/1683 [==============================] - 125s 75ms/step - loss: 376.6473 - MAE: 13.2347 - val_loss: 473.4496 - va
l_MAE: 14.5197
Epoch 4/5
1683/1683 [==============================] - 127s 75ms/step - loss: 323.1026 - MAE: 12.0794 - val_loss: 461.3187 - va
l_MAE: 14.6928
Epoch 5/5
1683/1683 [==============================] - 126s 75ms/step - loss: 286.7183 - MAE: 11.3440 - val_loss: 468.0188 - va
l_MAE: 14.9206
361/361 [==============================] - 6s 16ms/step - loss: 466.9315 - MAE: 14.8696
```

In [28]:
```
%tensorboard --logdir logs/hparam_tuning
```

```
Reusing TensorBoard on port 6006 (pid 9777), started 13:11:10 ago. (Use '!kill 9777' to kill it.)
```



[GRU Hyperparameter tuning (https://tensorboard.dev/experiment/k0ny5jh6Tvm8OALWMwBkWQ/#hparams)](https://tensorboard.dev/experiment/k0ny5jh6Tvm8OALWMwBkWQ/#hparams)

In [ ]:
```
#Vectorize vocab
voc_size = 30000
max_len = 11
embedding_features = 100
tokenizer = Tokenizer(num_words=voc_size, oov_token = '<OOV>')
tokenizer.fit_on_texts(X_train)
sequences_train = tokenizer.texts_to_sequences(X_train)
sequences_val = tokenizer.texts_to_sequences(X_val)
sequences_test = tokenizer.texts_to_sequences(X_test)

#add padding to ensure all inputs are the same size
data_train = pad_sequences(sequences_train, maxlen=max_len, padding= 'post', truncating = 'post')
data_val = pad_sequences(sequences_val, maxlen=max_len, padding= 'post', truncating = 'post')
data_test = pad_sequences(sequences_test, maxlen=max_len, padding= 'post', truncating = 'post')
```

In [46]:

```
model = models.Sequential()
model.add(Embedding(voc_size, embedding_features, input_length = max_len))
model.add(GRU(100,dropout=0.4))
model.add(Dense(1, activation = 'linear'))
model.summary()
```

```
Model: "sequential_16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_16 (Embedding)     (None, 11, 100)           3000000
_________________________________________________________________
gru_8 (GRU)                  (None, 100)               60600
_________________________________________________________________
dense_16 (Dense)             (None, 1)                 101
=================================================================
Total params: 3,060,701
Trainable params: 3,060,701
Non-trainable params: 0
_________________________________________________________________
```

In [48]:

```
# Compile and fit
model.compile(
  loss='MSE',
  optimizer='adam',
  metrics=['mae']
)


print('Training model...')
r = model.fit(
  data_train,
  y_train,
  epochs=5,
  validation_data=(data_val, y_val)
)
```

```
Training model...
Epoch 1/5
1683/1683 [==============================] - 40s 24ms/step - loss: 1219.0217 - mae: 25.0416 - val_loss: 606.3350 - va
l_mae: 16.9413
Epoch 2/5
1683/1683 [==============================] - 40s 24ms/step - loss: 494.9228 - mae: 15.3000 - val_loss: 470.5830 - val
_mae: 15.1494
Epoch 3/5
1683/1683 [==============================] - 40s 24ms/step - loss: 369.6068 - mae: 13.0574 - val_loss: 452.6128 - val
_mae: 14.6191
Epoch 4/5
1683/1683 [==============================] - 40s 24ms/step - loss: 308.9924 - mae: 11.7638 - val_loss: 449.0191 - val
_mae: 14.3290
Epoch 5/5
1683/1683 [==============================] - 40s 24ms/step - loss: 272.4880 - mae: 10.9233 - val_loss: 457.1555 - val
_mae: 14.2373
```

In [49]:

```
s1 = "Spyderco Mantra 3 Liner Lock Knife Black Carbon Fiber & G-10 S30V Steel C233CFP"
s1_p = 136.1
s2 = "Benchmade 556 Green 154cm Combo Blade Pardue Design"
s2_p = 71.95
s3 = "Case XX 6207 SS Mini Trapper Brown Peachseed Bone Pocket Knife Made in Usa"
s3_p = 51.45
```

In [50]:

```
def test_single_string(s):
    s = remove_special_char(s.lower())
    s = remove_punctuations(s)
    s = ' '.join(list(set(s.split())))
    test = tokenizer.texts_to_sequences([s])
    test2 = pad_sequences(test, maxlen=max_len, padding= 'post', truncating = 'post')
    pred=model.predict(test2)
    return pred
```

In [51]:

```
pred1 = test_single_string(s1)[0][0]
pred2 = test_single_string(s2)[0][0]
pred3 = test_single_string(s3)[0][0]
```

In [52]:

```

```

```
LICENSE.md            cnn_grayscale_relu1.h5   listed_data/
Notebooks/            ebay.yaml                logs/
README.md             images/                  terapeak_data/
```

sample1

sample2

![sample3]

In [53]:
```
print(f'True value: ${s1_p}, Predicted Value: ${pred1:.2f}, difference: ${pred1 - s1_p:.2f}')
print(f'True value: ${s2_p}, Predicted Value: ${pred2:.2f} difference: ${pred2 - s2_p:.2f}')
print(f'True value: ${s3_p}, Predicted Value: ${pred3:.2f} difference: ${pred3 - s3_p:.2f}')
```

```
True value: $136.1, Predicted Value: $124.79, difference: $-11.31
True value: $71.95, Predicted Value: $85.27 difference: $13.32
True value: $51.45, Predicted Value: $50.85 difference: $-0.60
```

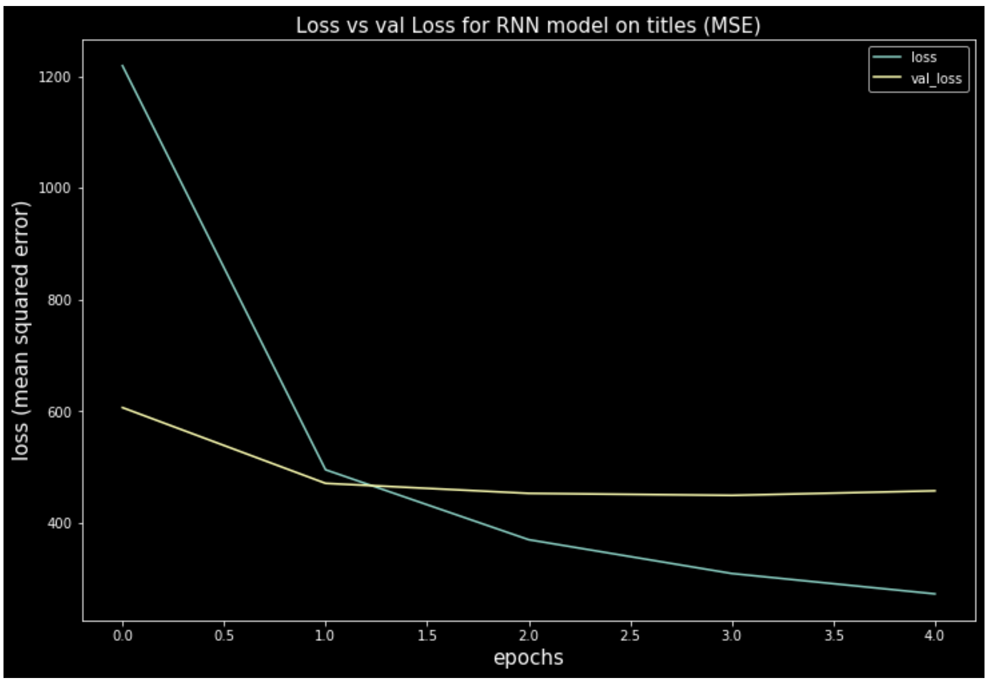
In [54]:
```
preds =model.predict(data_test)
```

In [55]:
```
preds = preds.reshape(len(preds))
```

In [57]:
```
test_results = model.evaluate(data_test, y_test)
```

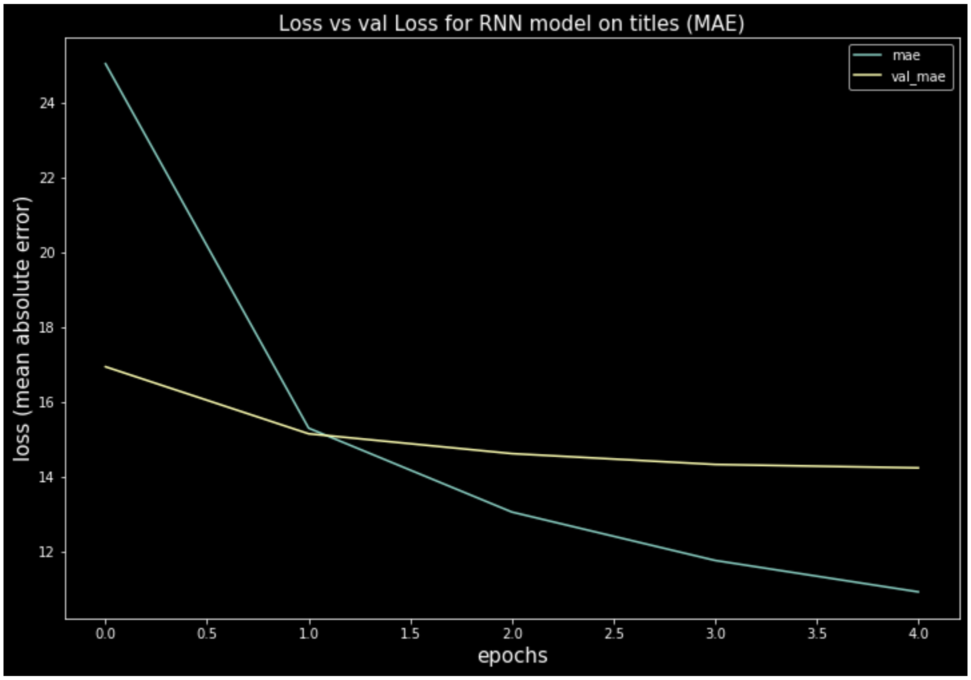
```
361/361 [==============================] - 1s 2ms/step - loss: 458.1150 - mae: 14.2777
```

In [58]:
```
fig = plt.subplots(figsize=(12,8))
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.title("Loss vs val Loss for RNN model on titles (MSE)", fontsize=15)
plt.xlabel("epochs", fontsize=15)
plt.ylabel("loss (mean squared error)", fontsize=15)
plt.legend();
plt.savefig('images/RNN_GRU_MSE1.png')
```



In [59]:

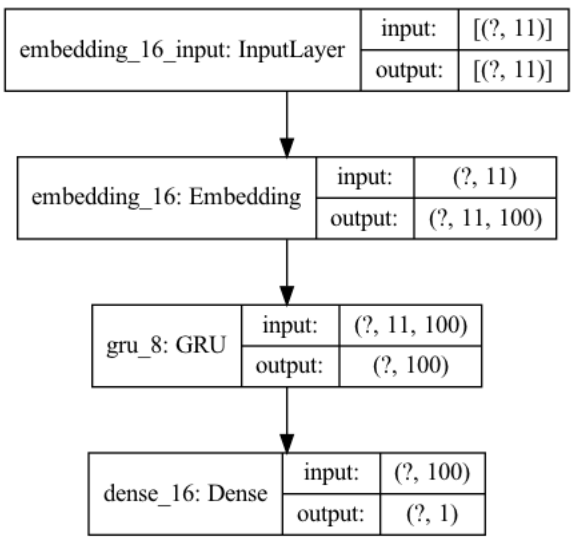
```
fig = plt.subplots(figsize=(12,8))
plt.plot(r.history['mae'], label='mae')
plt.plot(r.history['val_mae'], label='val_mae')
plt.title("Loss vs val Loss for RNN model on titles (MAE)", fontsize=15)
plt.xlabel("epochs", fontsize=15)
plt.ylabel("loss (mean absolute error)", fontsize=15)
plt.legend();
plt.savefig('images/RNN_GRU_MAE1.png')
```



In [60]:

```
plot_model(model,show_shapes=True, to_file='images/RNN_GRU1_arc.png')
```

Out[60]:



In [62]:

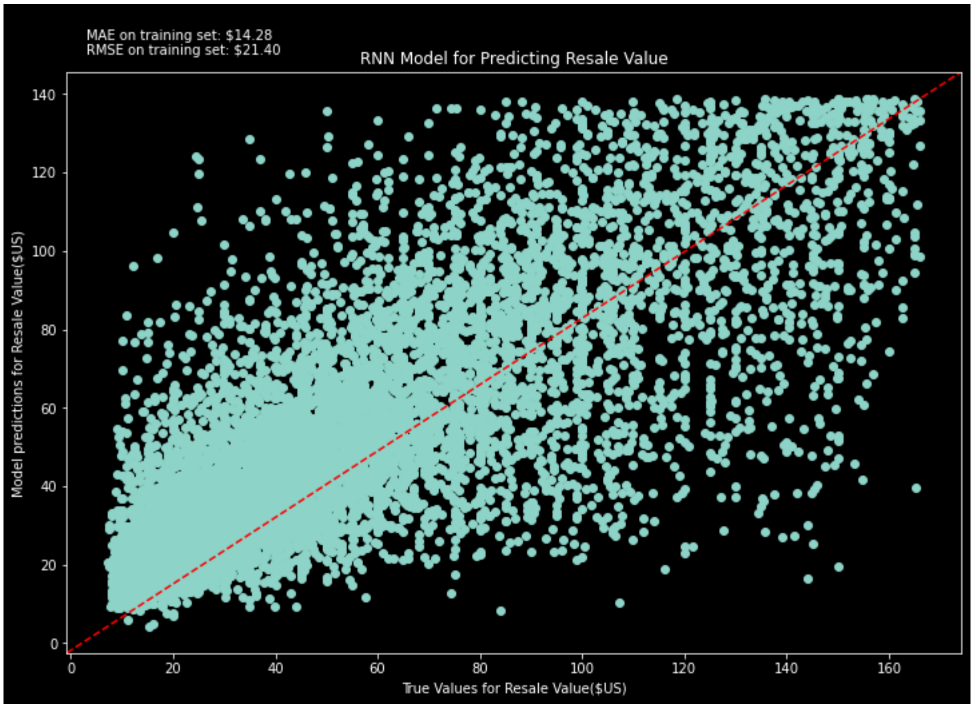
```
test_mae = mean_absolute_error(y_test, preds)
```

In [63]:

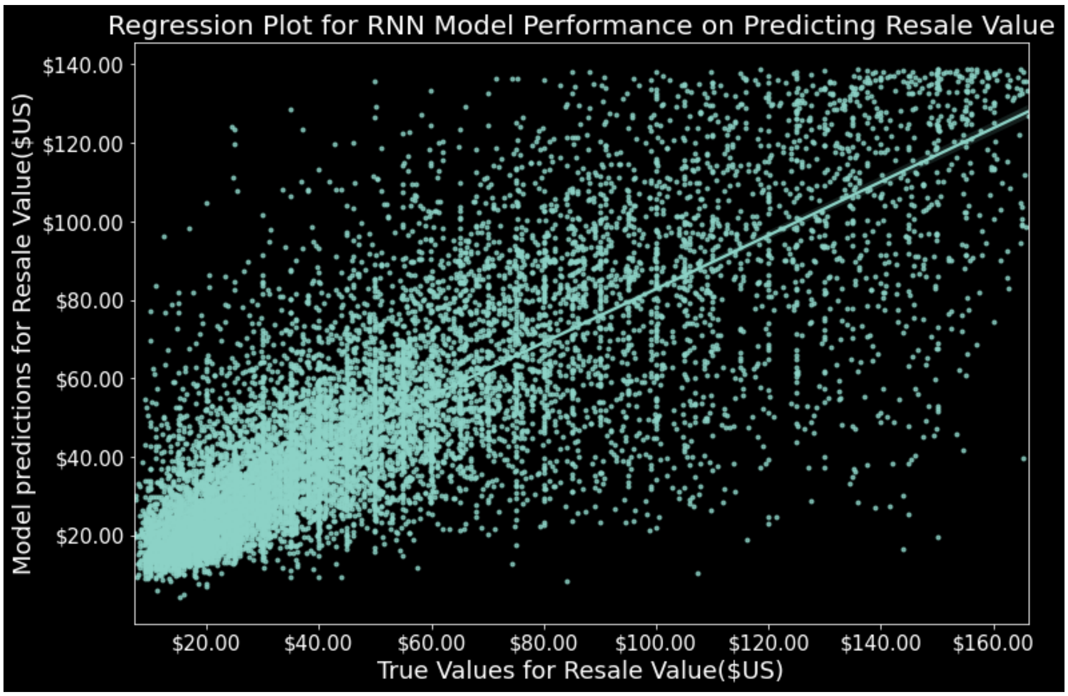
```
RMSE = np.sqrt(test_results[0])
```

In [65]:

```
string_score = f'\nMAE on training set: ${test_mae:.2f}'
string_score += f'\nRMSE on training set: ${RMSE:.2f}'
fig, ax = plt.subplots(figsize=(12, 8))
plt.scatter(y_test, preds)
ax.plot([0, 1], [0, 1], transform=ax.transAxes, ls="--", c="red")
plt.text(3, 150, string_score)
plt.title('RNN Model for Predicting Resale Value')
plt.ylabel('Model predictions for Resale Value($US)')
plt.xlabel('True Values for Resale Value($US)')
plt.savefig('images/regression_GRU_relu1.png');
```



In [93]:
```python
plt.figure(figsize=(12,8))

ax = sns.regplot(x=y_test,y=preds,marker='.')
plt.title('Regression Plot for RNN Model Performance on Predicting Resale Value',
         fontsize=20)
plt.ylabel('Model predictions for Resale Value($US)',
          fontsize=18)
plt.xlabel('True Values for Resale Value($US)',
          fontsize=18)
plt.xticks([20,40,
            60,80,
            100,120,
            140,160],
           ['$20.00','$40.00',
            '$60.00','$80.00',
            '$100.00','$120.00',
            '$140.00','$160.00'],
          fontsize=15)
plt.yticks([20,40,
            60,80,
            100,120,
            140],
           ['$20.00','$40.00',
            '$60.00','$80.00',
            '$100.00','$120.00',
            '$140.00'],
          fontsize=15)
plt.savefig('images/regPlot_GRU_performance.png');
```



In [107]:
```python
test_mae
```

Out[107]: 14.277658506004293

LSTM

In [94]:
```python
df_title = df.loc[:, ['title_nostop', 'converted_price']]


df_title.rename({'title_nostop': 'data',
                 'converted_price': 'labels'},
                axis=1, inplace=True)
```

In [95]:
```python
# df_title['labels'] = (df_title['labels']/mean_price)
Y = df_title['labels'].values
```

In [96]:
```
df_train, df_test, Ytrain, Ytest = train_test_split(df_title['data'],
                                                    Y,
                                                    test_size=0.3,
                                                    random_state=42)
```

In [97]:
```
X_val, X_test, Y_val, Y_test = train_test_split(df_test,
                                                Ytest,
                                                test_size=0.5,
                                                random_state=42)
```

In [98]:
```
# Convert sentences to sequences
MAX_VOCAB_SIZE = 30000
tokenizer = Tokenizer(num_words=MAX_VOCAB_SIZE)
tokenizer.fit_on_texts(df_train)
sequences_train = tokenizer.texts_to_sequences(df_train)
sequences_val = tokenizer.texts_to_sequences(X_val)
sequences_test = tokenizer.texts_to_sequences(X_test)
```

In [99]:
```
# get word -> integer mapping
word2idx = tokenizer.word_index
V = len(word2idx)
print('Found %s unique tokens.' % V)
```

```
Found 31847 unique tokens.
```

In [100]:
```
# pad sequences so that we get a N x T matrix
data_train = pad_sequences(sequences_train)
print('Shape of data train tensor:', data_train.shape)

# get sequence length
T = data_train.shape[1]
```

```
Shape of data train tensor: (53839, 19)
```

In [101]:
```
data_val = pad_sequences(sequences_val, maxlen=T)
print('Shape of data test tensor:', X_val.shape)
```

```
Shape of data test tensor: (11537,)
```

In [102]:
```
data_test = pad_sequences(sequences_test, maxlen=T)
print('Shape of data test tensor:', X_test.shape)
```

```
Shape of data test tensor: (11538,)
```

In [103]:
```
HP_NUM_UNITS = hp.HParam('units', hp.Discrete([16,32,64,100,300]))
HP_DROPOUT = hp.HParam('dropout', hp.RealInterval(0.2,0.4))


METRIC_MAE = 'MAE'

with tf.summary.create_file_writer('logs/hparam_tuning2').as_default():
    hp.hparams_config(
        hparams=[HP_NUM_UNITS, HP_DROPOUT],
        metrics=[hp.Metric(METRIC_MAE, display_name='MAE')],
    )
```

In [104]:
```
D = 11

def train_test_model(hparams):
    i = Input(shape=(T,))
    x = Embedding(V + 1, D)(i)
    x = LSTM(hparams[HP_NUM_UNITS],dropout=(hparams[HP_DROPOUT]),return_sequences=True)(x)
    x = GlobalMaxPooling1D()(x)
    x = Dense(1, activation='linear')(x)
    model = Model(i, x)
    model.compile(
      optimizer='Adam',
      loss='MSE',
      metrics=['MAE'],
    )
    model.fit(data_train,
              y_train,
              epochs=5,
              validation_data=(data_val, y_val)
             )
    _, MSE = model.evaluate(data_test, y_test)
    return MSE
```

In [105]:
```
def run(run_dir, hparams):
    with tf.summary.create_file_writer(run_dir).as_default():
        hp.hparams(hparams)  # record the values used in this trial
        MSE = train_test_model(hparams)
        tf.summary.scalar(METRIC_MAE, MSE, step=1)
```

In [ ]:
```
# %reload_ext tensorboard
```

In [ ]:
```
# rm -rf ./logs/
```

In [106]:
```
session_num = 0

for num_units in HP_NUM_UNITS.domain.values:
    for dropout_rate in (HP_DROPOUT.domain.min_value, HP_DROPOUT.domain.max_value):
        hparams = {
          HP_NUM_UNITS: num_units,
          HP_DROPOUT: dropout_rate,
        }
        run_name = "run-%d" % session_num
        print('--- Starting trial: %s' % run_name)
        print({h.name: hparams[h] for h in hparams})
        run('logs/hparam_tuning2/' + run_name, hparams)
        session_num += 1
```

```
--- Starting trial: run-0
{'units': 16, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 10s 6ms/step - loss: 2384.7729 - MAE: 34.5309 - val_loss: 1689.3279 - va
l_MAE: 27.8578
Epoch 2/5
1683/1683 [==============================] - 10s 6ms/step - loss: 1456.0245 - MAE: 27.1578 - val_loss: 1314.8094 - va
l_MAE: 27.2002
Epoch 3/5
1683/1683 [==============================] - 10s 6ms/step - loss: 1294.1053 - MAE: 27.8820 - val_loss: 1279.8707 - va
l_MAE: 28.1936
Epoch 4/5
1683/1683 [==============================] - 10s 6ms/step - loss: 1284.3530 - MAE: 28.3858 - val_loss: 1279.0942 - va
l_MAE: 28.3846
Epoch 5/5
1683/1683 [==============================] - 10s 6ms/step - loss: 1132.9052 - MAE: 25.6102 - val_loss: 925.0986 - val
_MAE: 21.9572
361/361 [==============================] - 0s 1ms/step - loss: 935.4721 - MAE: 21.9338
--- Starting trial: run-1
{'units': 16, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 10s 6ms/step - loss: 2454.6436 - MAE: 35.2228 - val_loss: 1741.5359 - va
l_MAE: 28.2011
Epoch 2/5
1683/1683 [==============================] - 10s 6ms/step - loss: 1483.2782 - MAE: 27.2002 - val_loss: 1323.5765 - va
l_MAE: 27.1177
Epoch 3/5
1683/1683 [==============================] - 10s 6ms/step - loss: 1296.8954 - MAE: 27.7967 - val_loss: 1279.9734 - va
l_MAE: 28.1704
Epoch 4/5
1683/1683 [==============================] - 10s 6ms/step - loss: 1284.4531 - MAE: 28.3950 - val_loss: 1279.4208 - va
l_MAE: 28.3440
Epoch 5/5
1683/1683 [==============================] - 10s 6ms/step - loss: 1111.0282 - MAE: 25.3697 - val_loss: 977.5331 - val
_MAE: 23.9133
361/361 [==============================] - 0s 1ms/step - loss: 987.4767 - MAE: 23.9082
--- Starting trial: run-2
{'units': 32, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 12s 7ms/step - loss: 1887.6387 - MAE: 30.7045 - val_loss: 1310.9747 - va
l_MAE: 27.2416
Epoch 2/5
1683/1683 [==============================] - 12s 7ms/step - loss: 1289.2524 - MAE: 28.0750 - val_loss: 1279.4521 - va
l_MAE: 28.3845
Epoch 3/5
1683/1683 [==============================] - 11s 7ms/step - loss: 1042.3108 - MAE: 24.4210 - val_loss: 891.2529 - val
_MAE: 22.5861
Epoch 4/5
1683/1683 [==============================] - 11s 7ms/step - loss: 811.8270 - MAE: 21.5575 - val_loss: 745.6796 - val_
MAE: 20.3276
Epoch 5/5
1683/1683 [==============================] - 11s 7ms/step - loss: 615.9078 - MAE: 17.7572 - val_loss: 559.6870 - val_
MAE: 16.5392
361/361 [==============================] - 1s 1ms/step - loss: 574.6607 - MAE: 16.6531
--- Starting trial: run-3
{'units': 32, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 12s 7ms/step - loss: 1878.6677 - MAE: 30.7487 - val_loss: 1307.0060 - va
l_MAE: 27.2892
Epoch 2/5
1683/1683 [==============================] - 11s 7ms/step - loss: 1289.2784 - MAE: 28.1167 - val_loss: 1279.4363 - va
l_MAE: 28.3628
Epoch 3/5
1683/1683 [==============================] - 12s 7ms/step - loss: 1284.4142 - MAE: 28.4498 - val_loss: 1279.4270 - va
l_MAE: 28.3553
Epoch 4/5
1683/1683 [==============================] - 12s 7ms/step - loss: 1284.5468 - MAE: 28.4475 - val_loss: 1279.4736 - va
l_MAE: 28.4069
Epoch 5/5
1683/1683 [==============================] - 12s 7ms/step - loss: 1284.4000 - MAE: 28.4544 - val_loss: 1279.4331 - va
l_MAE: 28.3764
361/361 [==============================] - 1s 2ms/step - loss: 1291.7626 - MAE: 28.4359
--- Starting trial: run-4
{'units': 64, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 13s 8ms/step - loss: 1597.9799 - MAE: 29.8745 - val_loss: 1279.4419 - va
l_MAE: 28.3215
Epoch 2/5
1683/1683 [==============================] - 13s 8ms/step - loss: 1250.3071 - MAE: 27.7324 - val_loss: 960.6080 - val
_MAE: 22.7962
Epoch 3/5
1683/1683 [==============================] - 14s 8ms/step - loss: 717.1694 - MAE: 19.3034 - val_loss: 597.9357 - val_
MAE: 18.1678
Epoch 4/5
1683/1683 [==============================] - 13s 8ms/step - loss: 486.6227 - MAE: 15.4496 - val_loss: 594.2352 - val_
MAE: 17.9450
```

```
Epoch 5/5
1683/1683 [==============================] - 13s 8ms/step - loss: 403.7130 - MAE: 13.8546 - val_loss: 485.2753 - val_
MAE: 15.1339
361/361 [==============================] - 1s 2ms/step - loss: 487.9741 - MAE: 15.0833
--- Starting trial: run-5
{'units': 64, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 13s 8ms/step - loss: 1595.2715 - MAE: 29.7548 - val_loss: 1279.6921 - va
l_MAE: 28.2267
Epoch 2/5
1683/1683 [==============================] - 13s 8ms/step - loss: 1217.2194 - MAE: 27.2196 - val_loss: 923.0527 - val
_MAE: 22.6759
Epoch 3/5
1683/1683 [==============================] - 13s 8ms/step - loss: 716.4354 - MAE: 19.5309 - val_loss: 559.7228 - val_
MAE: 17.0262
Epoch 4/5
1683/1683 [==============================] - 14s 8ms/step - loss: 498.1365 - MAE: 15.8132 - val_loss: 511.8537 - val_
MAE: 16.0226
Epoch 5/5
1683/1683 [==============================] - 13s 8ms/step - loss: 419.8455 - MAE: 14.3152 - val_loss: 482.3185 - val_
MAE: 15.1683
361/361 [==============================] - 1s 2ms/step - loss: 485.5522 - MAE: 15.1781
--- Starting trial: run-6
{'units': 100, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 20s 12ms/step - loss: 1493.6321 - MAE: 29.5059 - val_loss: 1279.8732 - v
al_MAE: 28.5279
Epoch 2/5
1683/1683 [==============================] - 19s 11ms/step - loss: 1094.8258 - MAE: 25.5558 - val_loss: 751.9514 - va
l_MAE: 20.8193
Epoch 3/5
1683/1683 [==============================] - 20s 12ms/step - loss: 637.5126 - MAE: 18.7021 - val_loss: 560.5487 - val
_MAE: 17.3457
Epoch 4/5
1683/1683 [==============================] - 20s 12ms/step - loss: 487.4558 - MAE: 15.7401 - val_loss: 505.0208 - val
_MAE: 16.0326
Epoch 5/5
1683/1683 [==============================] - 20s 12ms/step - loss: 421.6601 - MAE: 14.3280 - val_loss: 490.4229 - val
_MAE: 15.1483
361/361 [==============================] - 1s 3ms/step - loss: 493.7679 - MAE: 15.1880
--- Starting trial: run-7
{'units': 100, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 20s 12ms/step - loss: 1453.3962 - MAE: 28.5984 - val_loss: 888.5798 - va
l_MAE: 21.1965
Epoch 2/5
1683/1683 [==============================] - 20s 12ms/step - loss: 713.0633 - MAE: 19.6428 - val_loss: 613.6986 - val
_MAE: 18.5484
Epoch 3/5
1683/1683 [==============================] - 20s 12ms/step - loss: 525.8854 - MAE: 16.4260 - val_loss: 518.4414 - val
_MAE: 15.7141
Epoch 4/5
1683/1683 [==============================] - 20s 12ms/step - loss: 443.1071 - MAE: 14.7577 - val_loss: 497.2313 - val
_MAE: 16.1789
Epoch 5/5
1683/1683 [==============================] - 20s 12ms/step - loss: 398.6964 - MAE: 13.8659 - val_loss: 480.4944 - val
_MAE: 15.2288
361/361 [==============================] - 1s 3ms/step - loss: 486.1743 - MAE: 15.1957
--- Starting trial: run-8
{'units': 300, 'dropout': 0.2}
Epoch 1/5
1683/1683 [==============================] - 58s 35ms/step - loss: 1391.8549 - MAE: 28.8880 - val_loss: 1280.1849 - v
al_MAE: 28.5831
Epoch 2/5
1683/1683 [==============================] - 60s 35ms/step - loss: 828.7883 - MAE: 21.1159 - val_loss: 525.9389 - val
_MAE: 16.0498
Epoch 3/5
1683/1683 [==============================] - 59s 35ms/step - loss: 476.0586 - MAE: 15.2143 - val_loss: 476.8436 - val
_MAE: 15.0946
Epoch 4/5
1683/1683 [==============================] - 60s 35ms/step - loss: 406.2524 - MAE: 13.8185 - val_loss: 458.2765 - val
_MAE: 14.8149
Epoch 5/5
1683/1683 [==============================] - 60s 36ms/step - loss: 358.9321 - MAE: 12.8761 - val_loss: 449.5041 - val
_MAE: 14.4746
361/361 [==============================] - 4s 10ms/step - loss: 448.6512 - MAE: 14.3814
--- Starting trial: run-9
{'units': 300, 'dropout': 0.4}
Epoch 1/5
1683/1683 [==============================] - 56s 33ms/step - loss: 1383.2441 - MAE: 28.7544 - val_loss: 1279.6378 - v
al_MAE: 28.2320
Epoch 2/5
1683/1683 [==============================] - 56s 33ms/step - loss: 820.2665 - MAE: 21.0451 - val_loss: 538.5583 - val
_MAE: 16.5418
Epoch 3/5
1683/1683 [==============================] - 56s 33ms/step - loss: 513.9174 - MAE: 15.9021 - val_loss: 520.2703 - val
```

```
_MAE: 15.3313
Epoch 4/5
1683/1683 [==============================] - 54s 32ms/step - loss: 443.1652 - MAE: 14.5825 - val_loss: 469.6456 - val
_MAE: 14.8430
Epoch 5/5
1683/1683 [==============================] - 54s 32ms/step - loss: 399.0405 - MAE: 13.7069 - val_loss: 459.9512 - val
_MAE: 14.8769
361/361 [==============================] - 3s 9ms/step - loss: 462.2657 - MAE: 14.8595
```

LSTM Hyperparameter Tuning (https://tensorboard.dev/experiment/TmUjuPT7RGCVp0dZ2pwUPQ/) The Tensorboard above summarizes some hyperaparmeter optimization for the LSTM Network.

Business Recommendations

Deploying use of the GRU Price Predictive Model while listing a knife for resale can not only save time for the lister but can also help optimize for the correct price to list the knife which can balance between listing the knife too low and losing potential revenue or pricing the knife to high and creating inventory that stagnates on the shelf. for to balance excess inventory costs for too high of prices vs loss revenue for Pricing the item correctly is very important.

The performance metrics for the GRU Price Predictive Model was the Mean Squared Error and Mean Absolute Error for the model when predicting for the correct price to list the knife (an approximation for the true value of the knife). The best performing Model exhibited a Mean Absolute Error of $14.28. I believe an error range of plus or minus $14.28 outperforms the current process of scrolling through webpages and having the lister try to guess themselves. Even given missing the correct value by about 14 dollars and a quarter for each knife listed on average, the time saved trying to figure out a correct price within an acceptable limit without the model has implicit cost that is hard to value on a spreadsheet.

Summary: I reccomend deploying the Price Predicting Model before posting a pocket knife for sale on eBay.

CNN Titles

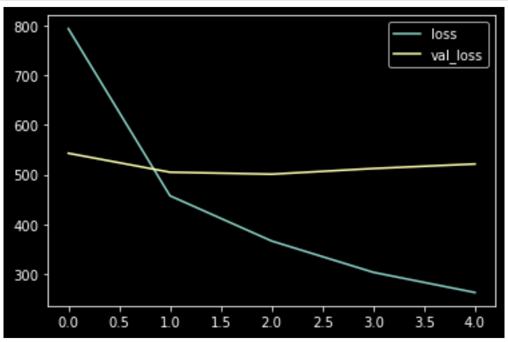
In [115]:

```
# Create the CNN model

# We get to choose embedding dimensionality
D = 100



i = Input(shape=(T,))
x = Embedding(V + 1, D)(i)
x = Conv1D(32, 3, activation='relu')(x)
x = MaxPooling1D(3)(x)
x = Conv1D(64, 3, activation='relu')(x)
x = MaxPooling1D(3)(x)
x = Dense(1, activation='linear')(x)

model = Model(i, x)
```

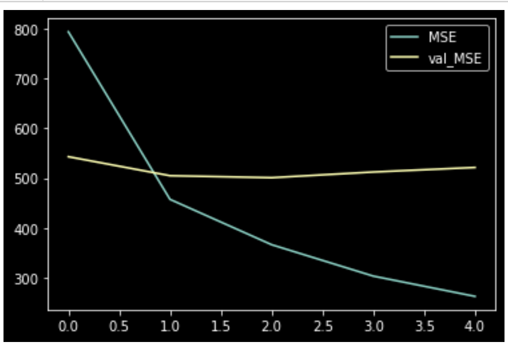
In [116]:
```
# Compile and fit
model.compile(
  loss='MSE',
  optimizer='adam',
  metrics=['mae']
)


print('Training model...')
r = model.fit(
  data_train,
  Ytrain,
  epochs=5,
  validation_data=(data_val, y_val)
)

```

```
Training model...
Epoch 1/5
1683/1683 [==============================] - 32s 19ms/step - loss: 793.7827 - mae: 20.0749 - val_loss: 542.9297 - val
_mae: 16.2382
Epoch 2/5
1683/1683 [==============================] - 33s 20ms/step - loss: 457.2762 - mae: 14.8287 - val_loss: 504.7583 - val
_mae: 15.5034
Epoch 3/5
1683/1683 [==============================] - 34s 20ms/step - loss: 366.2173 - mae: 13.0755 - val_loss: 500.7782 - val
_mae: 15.2826
Epoch 4/5
1683/1683 [==============================] - 33s 20ms/step - loss: 303.1649 - mae: 11.7271 - val_loss: 512.1823 - val
_mae: 15.8209
Epoch 5/5
1683/1683 [==============================] - 34s 20ms/step - loss: 262.4875 - mae: 10.8078 - val_loss: 521.3126 - val
_mae: 15.4711
```

In [117]:
```
# Plot loss per iteration
import matplotlib.pyplot as plt
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.legend();
```



In [118]:
```
# Plot accuracy per iteration
plt.plot(r.history['loss'], label='MSE')
plt.plot(r.history['val_loss'], label='val_MSE')
plt.legend();
```



CNN using images as input

In [119]:
```
df_imgs = df.drop(['title', 'url',
                   'date_sold', 'profit',
                   'ROI', 'brand', 'cost',
                   'pictureURLLarge'],
                     axis=1).copy()
```

In [120]:
```
df_imgs.dropna(subset=['Image'], inplace=True)
```

In [121]:
```
df_imgs.reset_index(drop=True, inplace=True)
```

In [122]:
```
df_imgs['file_index'] = df_imgs.index.values
df_imgs['file_index'] = df_imgs['file_index'].astype(str)
```

In [123]:
```
df_imgs['filename'] = df_imgs['file_index'] + '.jpg'
```

In [124]:
```
def download(row):
    filename = row.filepath

# create folder if it doesn't exist
#     os.makedirs(os.path.dirname(filename), exist_ok=True)

    url = row.Image
#     print(f"Downloading {url} to {filename}")

    try:
        r = requests.get(url, allow_redirects=True)
        with open(filename, 'wb') as f:
            f.write(r.content)
    except:
        print(f'{filename} error')
```

In [125]:
```
root_folder = 'C:/Users/12108/Documents/GitHub/Neural_Network_Predicting_Reseller_Success_Ebay/nn_images/'
df_imgs['filepath'] = root_folder + df_imgs['filename']
```

In [126]:
```
df_imgs['filepath'].sample(2).apply(print)
```

```
C:/Users/12108/Documents/GitHub/Neural_Network_Predicting_Reseller_Success_Ebay/nn_images/28310.jpg
C:/Users/12108/Documents/GitHub/Neural_Network_Predicting_Reseller_Success_Ebay/nn_images/39672.jpg
```

Out[126]:
```
28310    None
39672    None
Name: filepath, dtype: object
```

In [ ]:
```
# df_imgs.apply(download, axis=1)
```

All image files are stored locally for this project. The below markdown code is for reference.

```
img_list = os.listdir('C:/Users/12108/Documents/GitHub/Neural_Network_Predicting_Reseller_Success_Ebay/nn_image
s/')

img_df = df_imgs.loc[df_imgs['filename'].isin(img_list)].copy()

img_df.reset_index(drop=True, inplace=True)



img_df.rename({'Image': 'data',
               'converted_price': 'labels'},
                axis=1, inplace=True)
```

```
df_train, df_test, Ytrain, Ytest = train_test_split(img_df, Y, test_size=0.20)
datagen=ImageDataGenerator(rescale=1./255.,validation_split=0.20)

train_generator=datagen.flow_from_dataframe(
dataframe=df_train,
directory= None,
x_col="filepath",
y_col="labels",
subset="training",
batch_size=100,
seed=55,
shuffle=True,
class_mode="raw")

valid_generator=datagen.flow_from_dataframe(
dataframe=df_train,
directory=None,
x_col="filepath",
y_col="labels",
subset="validation",
batch_size=100,
seed=55,
shuffle=True,
class_mode="raw")

test_datagen=ImageDataGenerator(rescale=1./255.)
test_generator=test_datagen.flow_from_dataframe(
dataframe=df_test,
directory=None,
x_col="filepath",
y_col="labels",
batch_size=100,
seed=55,
shuffle=False,
class_mode="raw")
```

In [ ]:
```
# model = models.Sequential()

# model.add(layers.Conv2D(16, (3, 3), padding='same', activation='relu',
#                          input_shape=(256 ,256,  3)))
# model.add(layers.BatchNormalization())
# model.add(layers.Conv2D(16, (3, 3), activation='relu', padding='same'))
# model.add(layers.BatchNormalization())
# model.add(layers.MaxPooling2D((2, 2)))

# model.add(layers.Conv2D(32, (3, 3), padding='same', activation='relu',
#                          input_shape=(256 ,256,  3)))
# model.add(layers.BatchNormalization())
# model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same'))
# model.add(layers.BatchNormalization())
# model.add(layers.MaxPooling2D((2, 2)))

# model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
# model.add(layers.BatchNormalization())
# model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
# model.add(layers.BatchNormalization())
# model.add(layers.MaxPooling2D((2, 2)))

# model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
# model.add(layers.BatchNormalization())
# model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
# model.add(layers.BatchNormalization())
# model.add(layers.MaxPooling2D((2, 2)))

# model.add(layers.Flatten())

# model.add(Dense(512, activation='relu'))
# model.add(Dropout(0.1))
# model.add(Dense(256, activation='relu'))
# model.add(Dropout(0.1))
# model.add(Dense(128, activation='relu'))
# model.add(Dense(1, activation='linear'))

# model.compile(loss='MSE',
#               optimizer='Adam',
#                metrics=['mae', 'mse'])

# summary = model.fit(train_generator, epochs=3, validation_data=valid_generator)
```

In [127]:
```
model = tf.keras.models.load_model('cnn_grayscale_relu1.h5',  compile=False)
```

In [ ]:
```
plot_model(model, show_shapes=True, to_file="images/CNN_architecture.png")
```

In [128]:
```
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_24 (Conv2D)           (None, 500, 500, 16)      160       
_________________________________________________________________
batch_normalization_24 (Batc (None, 500, 500, 16)      64        
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 250, 250, 16)      0         
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 250, 250, 32)      4640      
_________________________________________________________________
batch_normalization_25 (Batc (None, 250, 250, 32)      128       
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 125, 125, 32)      0         
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 125, 125, 64)      18496     
_________________________________________________________________
batch_normalization_26 (Batc (None, 125, 125, 64)      256       
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 62, 62, 64)        0         
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 62, 62, 128)       73856     
_________________________________________________________________
batch_normalization_27 (Batc (None, 62, 62, 128)       512       
_________________________________________________________________
max_pooling2d_19 (MaxPooling (None, 31, 31, 128)       0         
_________________________________________________________________
flatten_4 (Flatten)          (None, 123008)            0         
_________________________________________________________________
dense_12 (Dense)             (None, 512)               62980608  
_________________________________________________________________
dense_13 (Dense)             (None, 128)               65664     
_________________________________________________________________
dense_14 (Dense)             (None, 1)                 129       
=================================================================
Total params: 63,144,513
Trainable params: 63,144,033
Non-trainable params: 480
_________________________________________________________________
```

In [ ]:
```

```

In [ ]:
```

```

Results

Recurrent Neural Network (GRU)

This model is recommend for use when listing a pocket knife on sale to help list it appropriately.

Convoluted Neural Network on Grayscale Images

- The MAE when testing the CNN was roughly $25.00. That is an error of plus or minus about 50% of the mean price of knives sold. Not acceptable yet as compared to the RNN with titles. Will address in future work.

Future Work

- Expand data to include other products readily purchasable at the Surplus Store.
- Attempt data augmentation on the CNN image network
- Attempt to obtain more aspect data for sold knives. Some important aspect data is limited access to sellers who average a certain amount of money per month.

Appendix

Random Forest with TFIDF vectorization and feature importance

In [ ]:
```
# df_title['labels'] = (df_title['labels']/mean_price)
Y = df_title['labels'].values
```

In [ ]:
```
df_title['data'].sample(10).apply(print)
```

In [ ]:
```
df_train, df_test, Ytrain, Ytest = train_test_split(df_title['data'],
                                                    Y,
                                                    test_size=0.3,
                                                    random_state=51)



```

In [ ]:
```
# X_val, X_test, Y_val, Y_test = train_test_split(df_test,
#                                                  Ytest,
#                                                  test_size=0.5,
#                                                  random_state=51)
```

In [ ]:
```
tfidf_vectorizer = TfidfVectorizer()
tfidf_vectorizer.fit(df_train)
X_train_vec = tfidf_vectorizer.transform(df_train)
x_test_vec = tfidf_vectorizer.transform(df_test)
```

In [ ]:
```
X_train_vec.get_shape()
```

In [ ]:
```
tfidf_vectorizer.get_feature_names()
```

In [ ]:
```
from sklearn.ensemble import RandomForestRegressor
rf_model = RandomForestRegressor(verbose=3, n_jobs=-1, random_state=42)
```

In [ ]:
```
rf_model.fit(X_train_vec,Ytrain)
```

In [ ]:
```
from sklearn import metrics

y_true = Ytest
y_pred = rf_model.predict(x_test_vec)

print('Mean Absolute Error (MAE):', metrics.mean_absolute_error(y_true, y_pred))
print('Mean Squared Error (MSE):', metrics.mean_squared_error(y_true, y_pred))
print('Root Mean Squared Error (RMSE):', metrics.mean_squared_error(y_true, y_pred, squared=False))
print('Explained Variance Score:', metrics.explained_variance_score(y_true, y_pred))
print('Max Error:', metrics.max_error(y_true, y_pred))
print('Mean Squared Log Error:', metrics.mean_squared_log_error(y_true, y_pred))
print('Median Absolute Error:', metrics.median_absolute_error(y_true, y_pred))
print('R^2:', metrics.r2_score(y_true, y_pred))
print('Mean Poisson Deviance:', metrics.mean_poisson_deviance(y_true, y_pred))
print('Mean Gamma Deviance:', metrics.mean_gamma_deviance(y_true, y_pred))
```

In [ ]:
```
features = tfidf_vectorizer.get_feature_names()
fi = rf_model.feature_importances_
importance = [(features[i], fi[i]) for i in range(0,2000)]
```

In [ ]:
```
importance[:50]
```

In [ ]:

```
# df_title['labels'] = (df_title['labels']/mean_price)
Y = df_title['labels'].values

df_title['data'].sample(10).apply(print)

df_train, df_test, Ytrain, Ytest = train_test_split(df_title['data'],
                                                    Y,
                                                    test_size=0.3,
                                                    random_state=51)




# X_val, X_test, Y_val, Y_test = train_test_split(df_test,
#                                                  Ytest,
#                                                  test_size=0.5,
#                                                  random_state=51)

tfidf_vectorizer = TfidfVectorizer()
tfidf_vectorizer.fit(df_train)
X_train_vec = tfidf_vectorizer.transform(df_train)
x_test_vec = tfidf_vectorizer.transform(df_test)

X_train_vec.get_shape()

tfidf_vectorizer.get_feature_names()

from sklearn.ensemble import RandomForestRegressor
rf_model = RandomForestRegressor(verbose=3, n_jobs=-1, random_state=42)

rf_model.fit(X_train_vec,Ytrain)

from sklearn import metrics

y_true = Ytest
y_pred = rf_model.predict(x_test_vec)

print('Mean Absolute Error (MAE):', metrics.mean_absolute_error(y_true, y_pred))
print('Mean Squared Error (MSE):', metrics.mean_squared_error(y_true, y_pred))
print('Root Mean Squared Error (RMSE):', metrics.mean_squared_error(y_true, y_pred, squared=False))
print('Explained Variance Score:', metrics.explained_variance_score(y_true, y_pred))
print('Max Error:', metrics.max_error(y_true, y_pred))
print('Mean Squared Log Error:', metrics.mean_squared_log_error(y_true, y_pred))
print('Median Absolute Error:', metrics.median_absolute_error(y_true, y_pred))
print('R^2:', metrics.r2_score(y_true, y_pred))
print('Mean Poisson Deviance:', metrics.mean_poisson_deviance(y_true, y_pred))
print('Mean Gamma Deviance:', metrics.mean_gamma_deviance(y_true, y_pred))

features = tfidf_vectorizer.get_feature_names()
fi = rf_model.feature_importances_
importance = [(features[i], fi[i]) for i in range(0,2000)]

importance[:50]
```