Mercari-III

May 24, 2020

[0]:
```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import os
import time
import datetime
import math
from contextlib import contextmanager
import scipy
from scipy.sparse import hstack
from sklearn.preprocessing import StandardScaler
from nltk.corpus import stopwords
from tqdm import tqdm
import re

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import KFold

from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge

import tensorflow as tf
from tensorflow.keras.layers import Dense,Input
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import EarlyStopping
```

[0]:
```
os.chdir("/content/drive/My Drive/Case Study I")
```

[0]:
```
# Reference : https://www.kaggle.com/lopuhin/
 ↪mercari-golf-0-3875-cv-in-75-loc-1900-s
# The central Idea of pre-processing and MLP architecture has been adopted from␣
 ↪the
```

```
# above reference.
# This Function concatenates text features from␣
↪item-name,brand,category,description to create
# the concatenated text feature

def preprocess(df):
  df['name'] = df['name'].fillna('') + ' ' + df['brand_name'].fillna('')
  df['text'] = (df['item_description'].fillna('') + ' ' + df['name'] + ' ' +␣
↪df['category_name'].fillna(''))

  return df[['name', 'text', 'shipping', 'item_condition_id']]
```

[0]:

```
# Reference : https://www.kaggle.com/lopuhin/
↪mercari-golf-0-3875-cv-in-75-loc-1900-s
@contextmanager
def timer(name):
    t0 = time.time()
    yield
    print(f'[{name}] done in {time.time() - t0:.0f} s')
```

[5]:

```
# Reference : https://www.kaggle.com/valkling/
↪mercari-rnn-2ridge-models-with-notes-0-42755
data = pd.read_csv('train.tsv',sep='\t')
data = data[(data.price >= 3) & (data.price <= 2000)].reset_index(drop=True)

cv = KFold(n_splits=20, shuffle=True, random_state=42)
train_ids, test_ids = next(cv.split(data))
# The above two line of code does K Fold Train Test Splitting where the Entire␣
↪data
# is divided into 20 folds each of size = len(data) // fold_size.
# Here data is roughly 1.4M and fold is 20 so each fold would be approx 74K

train, test = data.iloc[train_ids], data.iloc[test_ids]
# The Id's are saved to train and test respectively

# As we have seen that taking log values of the Price Column and
# standardizing them gives good result we will do the same

scaler = StandardScaler()
train_price = train['price'].values.reshape(-1,1)
test_price = test['price'].values.reshape(-1,1)

y_train = scaler.fit_transform(np.log1p(train_price))
y_test = scaler.transform(np.log1p(test_price))

print("X_Train Data Shape : ",train.shape)
```

```
print("y_train Shape : ",y_train.shape)
print("X_Test Data Shape : ",test.shape)
print("y_test Shape : ",y_test.shape)
```

```
X_Train Data Shape :  (1407575, 8)
y_train Shape :  (1407575, 1)
X_Test Data Shape :  (74083, 8)
y_test Shape :  (74083, 1)
```

1 Part I:

Here we will be using ensemble of 2 MLP's

[6]:
```
X_train = preprocess(train)
X_test = preprocess(test)
print(X_train.shape)
print(X_test.shape)
```

```
(1407575, 4)
(74083, 4)
```

[0]:
```
# Here we will create two Vectorized Copies of Train and Test
# One would be a normal TF-IDF Vectorized Copy
# Other being a Binarized Copy

Vectorizer = TfidfVectorizer(max_features=100000,token_pattern='\w+', dtype=np.
 ↪float32)
Vectorizer.fit(X_train['name'].values)

X_train_name = Vectorizer.transform(X_train['name'].values)
X_test_name = Vectorizer.transform(X_test['name'].values)

Vectorizer = TfidfVectorizer(max_features=100000,ngram_range =␣
 ↪(1,2),token_pattern='\w+', dtype=np.float32)

Vectorizer.fit(X_train['text'].values)

X_train_text = Vectorizer.transform(X_train['text'].values)
X_test_text = Vectorizer.transform(X_test['text'].values)


Vectorizer = OneHotEncoder(dtype=np.float32)

X_train_ship = Vectorizer.fit_transform(X_train['shipping'].values.
 ↪reshape(-1,1))
X_test_ship = Vectorizer.transform(X_test['shipping'].values.reshape(-1,1))
```

```
Vectorizer = OneHotEncoder(dtype=np.float32)

X_train_item = Vectorizer.fit_transform(X_train['item_condition_id'].values.
 ↪reshape(-1,1))
X_test_item = Vectorizer.transform(X_test['item_condition_id'].values.
 ↪reshape(-1,1))

X_train_tfidf = hstack((X_train_name,X_train_text,X_train_ship,X_train_item)).
 ↪tocsr()
X_test_tfidf = hstack((X_test_name,X_test_text,X_test_ship,X_test_item)).tocsr()

# Creating binary version of the Dataset, it means after we get a sparse␣
 ↪matrix,
# we will clip all non-zero values to 1. This is almost the same as using a
# CountVectorizer with binary=True but works much faster than that as
# we don't need to re-process the data.

X_train_binary, X_test_binary = [x.astype(np.bool).astype(np.float32)
                                    for x in [X_train_tfidf, X_test_tfidf]]
```

[8]:
```
print("X_train TFIDF Shape : ",X_train_tfidf.shape)
print("X_train Binarized Shape : ",X_train_binary.shape)
print("X_test TFIDF Shape : ",X_test_tfidf.shape)
print("X_test Binarized Shape : ",X_test_binary.shape)
```

```
X_train TFIDF Shape :  (1407575, 200007)
X_train Binarized Shape :  (1407575, 200007)
X_test TFIDF Shape :  (74083, 200007)
X_test Binarized Shape :  (74083, 200007)
```

[0]:
```
# Reference : https://www.kaggle.com/c/ashrae-energy-prediction/discussion/
 ↪113064

def rmsle_score(y, y_pred):
    assert len(y) == len(y_pred)
    to_sum = [(math.log(y_pred[i] + 1) - math.log(y[i] + 1)) ** 2.0 for i,pred␣
 ↪in enumerate(y_pred)]
    return (sum(to_sum) * (1.0/len(y))) ** 0.5
```

MLP Model 1

[10]:
```
input_layer = Input(shape=(X_train_tfidf.shape[1],), dtype='float32',␣
 ↪sparse=True)

layer1 = Dense(256, activation = "relu",kernel_initializer=tf.keras.
 ↪initializers.he_uniform(seed = 42))(input_layer)
```

```
layer2 = Dense(64, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer1)

layer3 = Dense(64, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer2)

layer4 = Dense(32, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer3)

output_layer = Dense(1,kernel_initializer=tf.keras.initializers.he_uniform(seed␣
 ↪= 42))(layer4)

model = Model(inputs = input_layer, outputs = output_layer)

model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 200007)]          0
_________________________________________________________________
dense (Dense)                (None, 256)               51202048
_________________________________________________________________
dense_1 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_3 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 33
=================================================================
Total params: 51,224,769
Trainable params: 51,224,769
Non-trainable params: 0
_________________________________________________________________
```

[11]:

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.003), loss =␣
 ↪"mean_squared_error")

for i in range(2):
  with timer (f'epoch {i + 1}'):
    model.fit(X_train_tfidf,y_train, batch_size= 2**(9 + i), epochs = 1,␣
 ↪verbose = 1,
              validation_data = (X_test_tfidf,y_test))
```

```
2750/2750 [==============================] - 21s 8ms/step - loss: 0.3429 -
val_loss: 0.3037
[epoch 1] done in 27 s
1375/1375 [==============================] - 14s 10ms/step - loss: 0.2022 -
val_loss: 0.2905
[epoch 2] done in 19 s
```

[12]:

```
model.save('model_part_1.h5')
y_pred = model.predict(X_test_tfidf)[:,0]
y_pred = np.expm1(scaler.inverse_transform(y_pred.reshape(-1, 1))[:, 0])
print("RMSLE from 1st MLP : ", rmsle_score(test_price,y_pred))
```

```
RMSLE from 1st MLP :  0.4019691273919564
```

MLP Model 2

[13]:

```
input_layer = Input(shape=(X_train_binary.shape[1],), dtype='float32',␣
 ↪sparse=True)

layer1 = Dense(256, activation = "relu",kernel_initializer=tf.keras.
 ↪initializers.he_uniform(seed = 42))(input_layer)

layer2 = Dense(64, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer1)

layer3 = Dense(64, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer2)

layer4 = Dense(32, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer3)

output_layer = Dense(1,kernel_initializer=tf.keras.initializers.he_uniform(seed␣
 ↪= 42))(layer4)

model1 = Model(inputs = input_layer, outputs = output_layer)

model1.summary()
```

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 200007)]          0
_________________________________________________________________
dense_5 (Dense)              (None, 256)               51202048
_________________________________________________________________
dense_6 (Dense)              (None, 64)                16448
_________________________________________________________________
```

```
dense_7 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_8 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_9 (Dense)              (None, 1)                 33
=================================================================
Total params: 51,224,769
Trainable params: 51,224,769
Non-trainable params: 0
_________________________________________________________________
```

[14]:

```
model1.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.003), loss =␣
 ↪"mean_squared_error")

for i in range(2):
  with timer (f'epoch {i + 1}'):
    model1.fit(X_train_binary,y_train, batch_size= 2**(9 + i), epochs = 1,␣
 ↪verbose = 1,
               validation_data = (X_test_binary,y_test))
```

```
2750/2750 [==============================] - 22s 8ms/step - loss: 0.3506 -
val_loss: 0.3113
[epoch 1] done in 29 s
1375/1375 [==============================] - 15s 11ms/step - loss: 0.2082 -
val_loss: 0.2953
[epoch 2] done in 19 s
```

[15]:

```
model1.save('model_part_2.h5')
y_pred_1 = model1.predict(X_test_binary)[:,0]
y_pred_1 = np.expm1(scaler.inverse_transform(y_pred_1.reshape(-1, 1))[:, 0])
print("RMSLE from 2nd MLP : ", rmsle_score(test_price,y_pred_1))
```

```
RMSLE from 2nd MLP :  0.40532368925506335
```

[0]:

```

```

Trying to find correct Weightage Proportion among the two models

[0]:

```
weights = np.linspace(0,1,30)
weight_value = []
predicted_value = []
for weight in weights:
  prediction = weight * y_pred + (1 - weight) * y_pred_1
  predicted_value.append(rmsle_score(test_price,prediction))
  weight_value.append(weight)
```

[24]:

```
index = np.argmin(predicted_value)
print("Min RMSLE Score : ", predicted_value[index])
```

```
print("Weights Proportion : ", weight_value[index])
```

```
Min RMSLE Score :  0.39149972214304307
Weights Proportion :  0.5517241379310345
```

From the weightage proportion it can be seen that the Ratio between MLP1 and MLP2 should be of 0.55 : 0.45

Ensemble the above two models

[25]:
```
print('Ensemble (weighted average of predictions from 2 models/runs')
y_prediction = np.average([y_pred, y_pred_1], weights=[0.55, 0.45], axis=0)
print("RMSLE from the Ensemble : ", rmsle_score(test_price,y_prediction))
```

```
Ensemble (weighted average of predictions from 2 models/runs
RMSLE from the Ensemble :  0.3914980708482419
```

Hence Using a Weighted Emsemble of two MLP's result in a Test RMSLE of 0.3914

2 Part II

Here we will create an emsemble of 2 MLP's with a Ridge Model

[26]:
```
print("X_Train Data Shape : ",train.shape)
print("y_train Shape : ",y_train.shape)
print("X_Test Data Shape : ",test.shape)
print("y_test Shape : ",y_test.shape)
```

```
X_Train Data Shape :  (1407575, 9)
y_train Shape :  (1407575, 1)
X_Test Data Shape :  (74083, 9)
y_test Shape :  (74083, 1)
```

[0]:
```
# Reference : Applied AI Course
def decontracted(phrase):
  # specific
  phrase = re.sub(r"won't", "will not", phrase)
  phrase = re.sub(r"can\'t", "can not", phrase)
  # general
  phrase = re.sub(r"n\'t", " not", phrase)
  phrase = re.sub(r"\'re", " are", phrase)
  phrase = re.sub(r"\'s", " is", phrase)
  phrase = re.sub(r"\'d", " would", phrase)
  phrase = re.sub(r"\'ll", " will", phrase)
  phrase = re.sub(r"\'t", " not", phrase)
  phrase = re.sub(r"\'ve", " have", phrase)
  phrase = re.sub(r"\'m", " am", phrase)
```

```
  return phrase
```

[28]:

```
import nltk
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Unzipping corpora/stopwords.zip.
```

[28]:

```
True
```

[0]:

```
stop_words = stopwords.words('english')
def preprocessing_text(text):
  preprocessed_text = []
  for sentence in tqdm(text.values):
    sentence = decontracted(sentence)
    sent = sentence.replace('\\r', ' ')
    sent = sent.replace('\\"', ' ')
    sent = sent.replace('\\n', ' ')
    sent = re.sub('[^A-Za-z0-9]+', ' ', sent)
    sent = ' '.join(e for e in sent.split() if e not in stop_words)
    preprocessed_text.append(sent.lower().strip())
  return preprocessed_text
```

[0]:

```
def new_preprocess(df):
  df['name'] = df['preprocess_name'].fillna('') + ' ' + df['brand_name'].
↪fillna('')
  df['text'] = (df['preprocess_desc'].fillna('') + ' ' + df['preprocess_name']␣
↪+ ' ' + df['category_name'].fillna(''))

  return df[['name', 'text', 'shipping', 'item_condition_id']]
```

[0]:

```
train['name'] = train['name'].replace([np.nan], '')
test['name'] = test['name'].replace([np.nan], '')

train['item_description'] = train['item_description'].replace([np.nan], '')
test['item_description'] = test['item_description'].replace([np.nan], '')
```

[32]:

```
train['preprocess_name'] = preprocessing_text(train['name'])
test['preprocess_name'] = preprocessing_text(test['name'])

train['preprocess_desc'] = preprocessing_text(train['item_description'])
test['preprocess_desc'] = preprocessing_text(test['item_description'])
```

```
100%|      | 1407575/1407575 [00:29<00:00, 47127.04it/s]
100%|      | 74083/74083 [00:01<00:00, 47422.96it/s]
100%|      | 1407575/1407575 [01:27<00:00, 16018.82it/s]
100%|      | 74083/74083 [00:04<00:00, 15790.97it/s]
```

[33]:
```
X_train = new_preprocess(train)
X_test = new_preprocess(test)
print(X_train.shape)
print(X_test.shape)
```

```
(1407575, 4)
(74083, 4)
```

[0]:
```
Vectorizer = TfidfVectorizer(max_features=100000,token_pattern='\w+', dtype=np.
 ↪float32)
Vectorizer.fit(X_train['name'].values)

X_train_name = Vectorizer.transform(X_train['name'].values)
X_test_name = Vectorizer.transform(X_test['name'].values)

Vectorizer = TfidfVectorizer(max_features=100000,ngram_range =␣
 ↪(1,2),token_pattern='\w+', dtype=np.float32)

Vectorizer.fit(X_train['text'].values)

X_train_text = Vectorizer.transform(X_train['text'].values)
X_test_text = Vectorizer.transform(X_test['text'].values)

Vectorizer = CountVectorizer(vocabulary= list(X_train['shipping'].
 ↪unique()),lowercase=False, binary = True)

X_train_ship = Vectorizer.fit_transform(X_train['shipping'].values.astype(str))
X_test_ship = Vectorizer.transform(X_test['shipping'].values.astype(str))

Vectorizer = CountVectorizer(vocabulary= list(X_train['item_condition_id'].
 ↪unique()),lowercase=False, binary = True)

X_train_item = Vectorizer.fit_transform(X_train['item_condition_id'].values.
 ↪astype(str))
X_test_item = Vectorizer.transform(X_test['item_condition_id'].values.
 ↪astype(str))

X_train_tfidf = hstack((X_train_name,X_train_text,X_train_ship,X_train_item)).
 ↪tocsr()
X_test_tfidf = hstack((X_test_name,X_test_text,X_test_ship,X_test_item)).tocsr()
```

[35]:
```
print("X_train TFIDF Shape : ",X_train_tfidf.shape)
print("X_test TFIDF Shape : ",X_test_tfidf.shape)
```

```
X_train TFIDF Shape :  (1407575, 200007)
X_test TFIDF Shape :  (74083, 200007)
```

[0]:

```
ridge_model = Ridge(solver = "lsqr", fit_intercept=False, alpha=10)
ridge_model.fit(X_train_tfidf, y_train)

ridge_pred = ridge_model.predict(X_test_tfidf)[:,0]
ridge_pred = np.expm1(scaler.inverse_transform(ridge_pred.reshape(-1, 1))[:, 0])
```

[37]:

```
input_layer = Input(shape=(X_train_tfidf.shape[1],), dtype='float32',␣
 ↪sparse=True)

layer1 = Dense(256, activation = "relu",kernel_initializer=tf.keras.
 ↪initializers.he_uniform(seed = 42))(input_layer)

layer2 = Dense(64, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer1)

layer3 = Dense(64, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer2)

layer4 = Dense(32, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer3)

output_layer = Dense(1,kernel_initializer=tf.keras.initializers.he_uniform(seed␣
 ↪= 42))(layer4)

model3 = Model(inputs = input_layer, outputs = output_layer)

model3.summary()
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 200007)]          0
_________________________________________________________________
dense_10 (Dense)             (None, 256)               51202048
_________________________________________________________________
dense_11 (Dense)             (None, 64)                16448
_________________________________________________________________
dense_12 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_13 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_14 (Dense)             (None, 1)                 33
=================================================================
Total params: 51,224,769
Trainable params: 51,224,769
Non-trainable params: 0
```

```
-----------------------------------------------------------------
```

[38]:

```
model3.compile(optimizer="adam", loss = "mean_squared_error")

for i in range(2):
  with timer (f'epoch {i + 1}'):
    model3.fit(X_train_tfidf,y_train, batch_size= 2**(9 + i), epochs = 1,␣
 ↪verbose = 1,
              validation_data = (X_test_tfidf,y_test))
```

```
2750/2750 [==============================] - 22s 8ms/step - loss: 0.3692 -
val_loss: 0.3246
[epoch 1] done in 27 s
1375/1375 [==============================] - 14s 10ms/step - loss: 0.2246 -
val_loss: 0.3127
[epoch 2] done in 18 s
```

[0]:

```
mlp1_pred = model3.predict(X_test_tfidf)[:,0]
mlp1_pred = np.expm1(scaler.inverse_transform(mlp1_pred.reshape(-1, 1))[:, 0])
```

[40]:

```
input_layer = Input(shape=(X_train_tfidf.shape[1],), dtype='float32',␣
 ↪sparse=True)

layer1 = Dense(512, activation = "relu",kernel_initializer=tf.keras.
 ↪initializers.he_uniform(seed = 42))(input_layer)

layer2 = Dense(256, activation = "relu",kernel_initializer=tf.keras.
 ↪initializers.he_uniform(seed = 42))(layer1)

layer3 = Dense(64, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer2)

layer4 = Dense(64, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer3)

layer5 = Dense(32, activation = "relu",kernel_initializer=tf.keras.initializers.
 ↪he_uniform(seed = 42))(layer4)

output_layer = Dense(1,kernel_initializer=tf.keras.initializers.he_uniform(seed␣
 ↪= 42))(layer5)

model4 = Model(inputs = input_layer, outputs = output_layer)

model4.summary()
```

```
Model: "model_3"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 200007)]          0

_________________________________________________________________
dense_15 (Dense)             (None, 512)               102404096

_________________________________________________________________
dense_16 (Dense)             (None, 256)               131328

_________________________________________________________________
dense_17 (Dense)             (None, 64)                16448

_________________________________________________________________
dense_18 (Dense)             (None, 64)                4160

_________________________________________________________________
dense_19 (Dense)             (None, 32)                2080

_________________________________________________________________
dense_20 (Dense)             (None, 1)                 33
=================================================================
Total params: 102,558,145
Trainable params: 102,558,145
Non-trainable params: 0

_________________________________________________________________
```

[41]:
```
model4.compile(optimizer="adam", loss = "mean_squared_error")

for i in range(2):
  with timer (f'epoch {i + 1}'):
    model4.fit(X_train_tfidf,y_train, batch_size= 2**(9 + i), epochs = 1,␣
 ↪verbose = 1,
              validation_data = (X_test_tfidf,y_test))
```

```
2750/2750 [==============================] - 35s 13ms/step - loss: 0.3677 -
val_loss: 0.3228
[epoch 1] done in 39 s
1375/1375 [==============================] - 21s 15ms/step - loss: 0.2140 -
val_loss: 0.3095
[epoch 2] done in 25 s
```

[0]:
```
mlp2_pred = model4.predict(X_test_tfidf)[:,0]
mlp2_pred = np.expm1(scaler.inverse_transform(mlp2_pred.reshape(-1, 1))[:, 0])
```

[0]:
```
model3.save('model_part_3.h5')
model4.save('model_part_4.h5')
```

Trying to find the Weight Proportion for Minimum RMSLE Score

[0]:
```
# Initially between two MLP's we need to find Ratio
weights = np.linspace(0,1,30)
weight_value = []
predicted_value = []
```

```
for weight in weights:
  prediction = weight * mlp1_pred + (1 - weight) * mlp2_pred
  predicted_value.append(rmsle_score(test_price,prediction))
  weight_value.append(weight)
```

[45]:

```
index = np.argmin(predicted_value)
print("Min RMSLE Score : ", predicted_value[index])
print("Weights Proportion : ", weight_value[index])
```

```
Min RMSLE Score :  0.4071754717586537
Weights Proportion :  0.4482758620689655
```

So the Weight Propoartion between MLP1 and MLP2 should be 0.45 : 0.55

Now we need to find Weight proportion between the MLP's and Ridge Model

[0]:

```
mlp_final_prediction = 0.45 * mlp1_pred + 0.55 * mlp2_pred
```

[0]:

```
weights = np.linspace(0,1,30)
weight_value = []
predicted_value = []
for weight in weights:
  prediction = weight * mlp_final_prediction + (1 - weight) * ridge_pred
  predicted_value.append(rmsle_score(test_price,prediction))
  weight_value.append(weight)
```

[49]:

```
index = np.argmin(predicted_value)
print("Min RMSLE Score : ", predicted_value[index])
print("Weights Proportion : ", weight_value[index])
```

```
Min RMSLE Score :  0.406350793936344
Weights Proportion :  0.896551724137931
```

We can see that the MLP's ratio to the Ridge Ratio should be 90 : 10. Hence the final distribution of Weights would be : * MLP 1 = 0.9 * 0.45 = 0.405 * MLP 2 = 0.9 * 0.55 = 0.495 * Ridge Model = 0.1

[50]:

```
print('Ensemble (weighted average of predictions from 3 models/runs')
y_prediction = np.average([ridge_pred, mlp1_pred,mlp2_pred], weights=[0.1, 0.
↪405,0.495], axis=0)
print("RMSLE from the Ensemble : ", rmsle_score(test_price,y_prediction))
```

```
Ensemble (weighted average of predictions from 3 models/runs
RMSLE from the Ensemble :  0.406353176306763
```

Ensemble of 2 MLP's and Ridge Model leads to Test RMSLE 0.406353176306763

One could defintely try other things like BatchNormalization, Dropouts,changing Activation Functions, but cannot increase the model's parameters as it would start overfitting and will also increase the time to train and test the Model.

[51]:

```
from prettytable import PrettyTable
x = PrettyTable()
x.field_names = ["Model", "Test RMSLE"]

x.add_row(["Weighted Ensemble of 2 Sparse MLP's",0.3914])

x.add_row(["Weighted Ensemble of 2 Sparse MLP's with Ridge Regressor ",0.4063])


print(x)
```

```
+-----------------------------------------------------------+------------+
|                           Model                           | Test RMSLE |
+-----------------------------------------------------------+------------+
|            Weighted Ensemble of 2 Sparse MLP's            |   0.3914   |
| Weighted Ensemble of 2 Sparse MLP's with Ridge Regressor  |   0.4063   |
+-----------------------------------------------------------+------------+
```

Both the Models performs better than other Models that we have tried so far and the advantage of them over others is they require lesser time to train and test as compared to other models.

[0]: