MIS780 Advanced AI For Business - Assignment 2 - T2 2024

Task Number: 1 Real estate analytics with tabular data

Student Name: Anagha prashanth Raje URS

Student ID: 223709844

Table of Content

Executive Summary

The aim of this project was to develop accurate models to predict house prices by analyzing a dataset that contains various features of home sales.The task involved building multiple models, including linear regression and different MLP architectures, and comparing their performance to find the best one for predicting house prices.

After experimenting with several models, MLP2 was identified as the top performer, achieving a ValMAE of 74,317.95 and an R^2 of 0.938. With its three hidden layers (100, 128, and 20 nodes) and the use of the RMSProp optimizer, MLP2 excelled at capturing intricate relationships between features and prices, providing the right blend of accuracy and model complexity.

Other models like MLP1 and MLP3 either lacked depth or introduced unnecessary complexity, resulting in higher prediction errors. MLP2, however, gravitating between generalization and performance, making it feasible for practical real-world deployment in real estate price prediction.

In summary, MLP2 not only provided high accuracy but also demonstrated scalability and efficiency, making it the optimal choice for future applications as new data becomes available.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
from __future__ import print_function
import os
import math
import datetime
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_absolute_error


pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

Data processing

```
#Loading the given datatset
data = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/Advanced AI MIS780/Part1_house_price.csv")
print('Number of records read: ', data.size)
```

```
Number of records read:  420000
```

```
data.info() #displaying the data information
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20000 entries, 0 to 19999
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             20000 non-null  int64  
 1   date           20000 non-null  object 
 2   price          20000 non-null  float64
 3   bedrooms       20000 non-null  int64  
 4   bathrooms      20000 non-null  float64
 5   sqft_living    20000 non-null  int64  
 6   sqft_lot       20000 non-null  int64  
 7   floors         20000 non-null  float64
 8   waterfront     20000 non-null  int64  
 9   view           20000 non-null  int64  
 10  condition      20000 non-null  int64  
 11  grade          20000 non-null  int64  
 12  sqft_above     20000 non-null  int64  
 13  sqft_basement  20000 non-null  int64  
 14  yr_built       20000 non-null  int64  
 15  yr_renovated   20000 non-null  int64  
 16  zipcode        20000 non-null  int64  
 17  lat            20000 non-null  float64
 18  long           20000 non-null  float64
 19  sqft_living15  20000 non-null  int64  
 20  sqft_lot15     20000 non-null  int64  
dtypes: float64(5), int64(15), object(1)
memory usage: 3.2+ MB
```

```
# Analysing dataset to identify missing values
data.isnull().sum()
```

	0
id	0
date	0
price	0
bedrooms	0
bathrooms	0
sqft_living	0
sqft_lot	0
floors	0
waterfront	0
view	0
condition	0
grade	0
sqft_above	0
sqft_basement	0
yr_built	0
yr_renovated	0
zipcode	0
lat	0
long	0
sqft_living15	0
sqft_lot15	0

dtype: int64

```
#Executing code to drop date column
data.drop(['date'], axis=1, inplace=True)
```

splitting the data for the purpose of training and validation

```
train_size, valid_size, test_size = (0.7, 0.3, 0.0)
House_train, House_valid = train_test_split(data,
                                      test_size=valid_size,
                                      random_state=2021)
```

Step: Extracting the data for the purpose of training and validation

```
label_col='price'
House_y_train = House_train[[label_col]]
House_x_train = House_train.drop(label_col, axis=1)
House_y_valid = House_valid[[label_col]]
House_x_valid = House_valid.drop(label_col, axis=1)

print('Size of training set: ', len(House_x_train))
print('Size of validation set: ', len(House_x_valid))
```

```
Size of training set:  14000
Size of validation set:  6000
```

Next: Creating a scaling model to scale both training and testing data usinf training set

```
scaler = MinMaxScaler(feature_range=(0, 1), copy=True).fit(House_x_train)
House_x_train = pd.DataFrame(scaler.transform(House_x_train),
                             columns = House_x_train.columns, index = House_x_train.index)
House_x_valid = pd.DataFrame(scaler.transform(House_x_valid),
                             columns = House_x_valid.columns, index = House_x_valid.index)

print('X train min =', round(House_x_train.min().min(),4), '; max =', round(House_x_train.max().max(), 4))
print('X valid min =', round(House_x_valid.min().min(),4), '; max =', round(House_x_valid.max().max(), 4))
```

```
X train min = 0.0 ; max = 1.0
X valid min = -0.0 ; max = 1.359
```

3.Predictive modelling

```
import tensorflow as tf
from tensorflow.keras import metrics
from tensorflow.keras import regularizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Nadam, RMSprop
```

Convert pandas data frames to `np` arrays.

```
arr_x_train = np.array(House_x_train)
arr_y_train = np.array(House_y_train)
arr_x_valid = np.array(House_x_valid)
arr_y_valid = np.array(House_y_valid)

print('Training shape:', arr_x_train.shape)
print('Training samples: ', arr_x_train.shape[0])
print('Validation samples: ', arr_x_valid.shape[0])
```

```
Training shape: (14000, 19)
Training samples:  14000
Validation samples:  6000
```

```
from keras.callbacks import EarlyStopping, Callback
early_stopping = [EarlyStopping(monitor='val_loss', patience=20, verbose=0)]
```

Create several **Keras models** for experiment purpose.

The second with `RMSProp` optimizer consists of 4 layers and the first uses 20% dropouts.

```
def basic_model_2(x_size, y_size):
    t_model = Sequential()
    t_model.add(Dense(100, activation="tanh", input_shape=(x_size,)))
    t_model.add(Dropout(0.2))
    t_model.add(Dense(128, activation="relu"))
    t_model.add(Dense(20, activation="relu"))
    t_model.add(Dense(y_size))
    t_model.compile(
```

```
        loss='mean_squared_error',
        optimizer=RMSprop(learning_rate=0.005, rho=0.9, momentum=0.0, epsilon=1e-07, weight_decay=0.0,),
        metrics=[metrics.mae])
    return(t_model)
```

After trial of multiple models, executing the best performing model

```
model = basic_model_2(arr_x_train.shape[1], arr_y_train.shape[1])
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` arg
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	2,000
dropout (Dropout)	(None, 100)	0
dense_1 (Dense)	(None, 128)	12,928
dense_2 (Dense)	(None, 20)	2,580
dense_3 (Dense)	(None, 1)	21

```
Total params: 17,529 (68.47 KB)
Trainable params: 17,529 (68.47 KB)
```

Fit the model and record the history of training and validation.

```
history = model.fit(arr_x_train, arr_y_train,
    batch_size=128,
    epochs=400,
    shuffle=True,
    verbose=2,
    validation_data=(arr_x_valid, arr_y_valid),
    callbacks=[early_stopping]
                  )
```

Show hidden output

Evaluate and report performance of the trained model

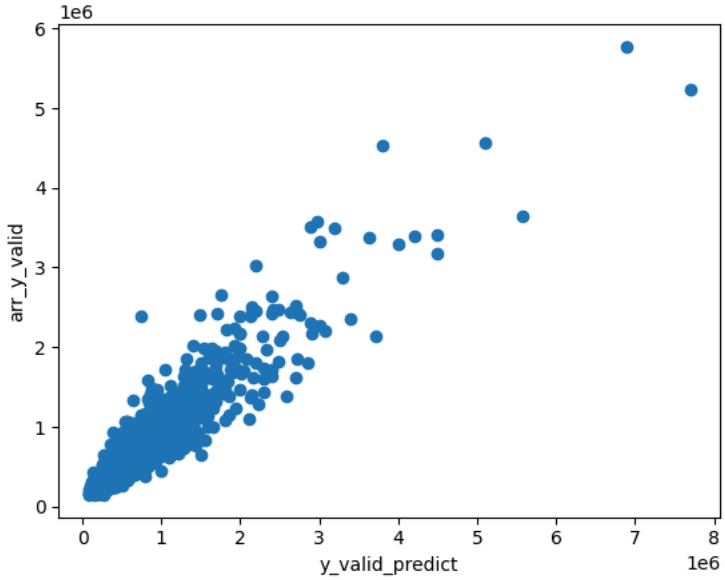
```
train_score = model.evaluate(arr_x_train, arr_y_train, verbose=0)
valid_score = model.evaluate(arr_x_valid, arr_y_valid, verbose=0)

print('Train MAE: ', round(train_score[1], 2), ', Train Loss: ', round(train_score[0], 2))
print('Val MAE: ', round(valid_score[1], 2), ', Val Loss: ', round(valid_score[0], 2))
```

```
Train MAE:  72443.34 , Train Loss:  13514296320.0
Val MAE:  74317.95 , Val Loss:  17540651008.0
```

Now plot the true vs. predicted values.

```
y_valid_predict = model.predict(arr_x_valid)
# plot
plt.scatter(arr_y_valid, y_valid_predict)
plt.ylabel('arr_y_valid')
plt.xlabel('y_valid_predict')
plt.show()

corr_result = np.corrcoef(arr_y_valid.reshape(1,6000)[0], y_valid_predict.reshape(1,6000)[0])
print('The Correlation between true and predicted values is: ',round(corr_result[0,1],3))
```

```
188/188 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
```

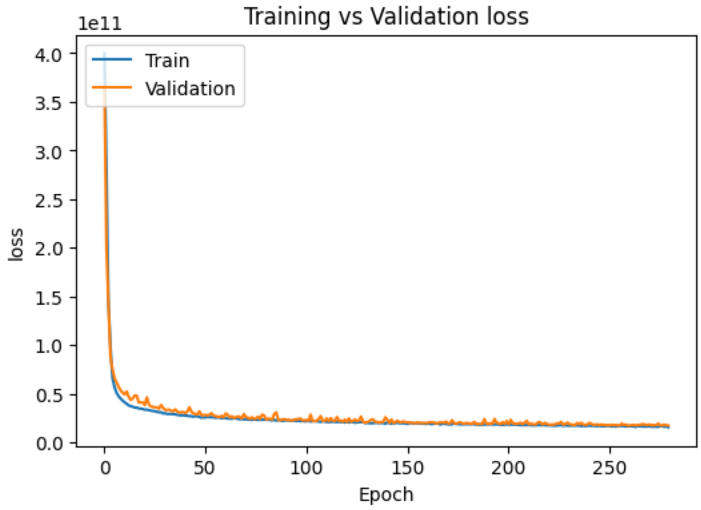


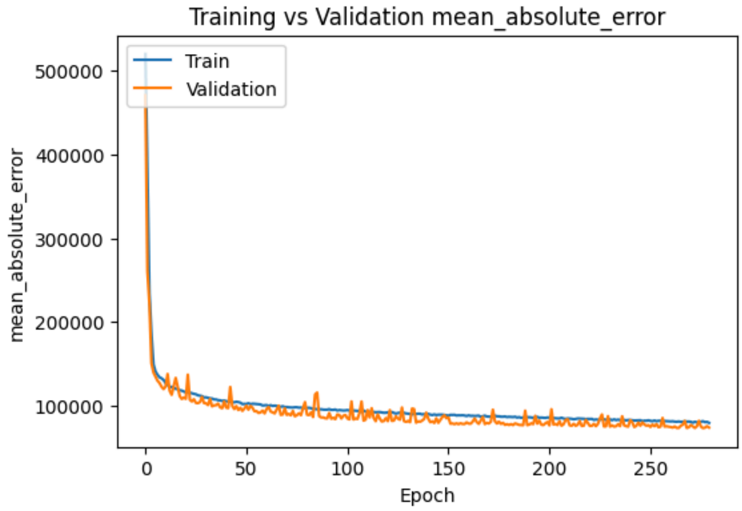

```
The Correlation between true and predicted values is:  0.938
```

The **scatter plot** shows that the MLP model did a great job predicting house prices, with a strong correlation of 0.938 between actual and predicted values. It worked well for most homes, especially in the lower price range, but had some trouble with higher-priced properties. The model's three-layer architecture and optimizer helped it capture complex relationships in the data, making it a good fit for real-world use. However, a bit more tuning could improve its predictions for luxury homes. Overall, it performed better than simpler models like Linear Regression.

Next we plot he training set , i.e. the *Mean Absolute Error* and *Loss (Mean Squared Error)*, which were both defined at the time of model compilation.

```
def plot_hist(h, xsize=6, ysize=5):
    # Prepare plotting
    fig_size = plt.rcParams["figure.figsize"]
    plt.rcParams["figure.figsize"] = [xsize, ysize]

    # Get training and validation keys
    ks = list(h.keys())
    n2 = math.floor(len(ks)/2)
    train_keys = ks[0:n2]
    valid_keys = ks[n2:2*n2]

    # summarize history for different metrics
    for i in range(n2):
        plt.plot(h[train_keys[i]])
        plt.plot(h[valid_keys[i]])
        plt.title('Training vs Validation '+train_keys[i])
        plt.ylabel(train_keys[i])
        plt.xlabel('Epoch')
        plt.legend(['Train', 'Validation'], loc='upper left')
        plt.draw()
        plt.show()

    return


hist = pd.DataFrame(history.history)

# Plot history
plot_hist(hist, xsize=6, ysize=4)
```





Displaying

Training vs. Validation Loss This graph shows how the model quickly reduces its error as it learns, with a steep drop in the first 50 epochs. Both the training and validation lines stay close together, which is a good sign that the model isn't overfitting and performs well on new, unseen data. After 50 epochs, the error levels off, meaning the model has learned as much as it can.

Training vs. Validation Mean Absolute Error (MAE) This graph shows the accuracy of the model's predictions improving over time. The error decreases sharply in the beginning and stabilizes around 100,000 after 50 epochs. The training and validation lines are nearly identical, indicating the model is making consistent, reliable predictions on both the training data and new, unseen data. This shows the model generalizes well and can be trusted for real-world house price predictions.

Model	Hidden Layers	Nodes	Optimizer	ValMAE	R^2 Correlation	Activation
MLP1	2	50,30	Adam	82400.32	0.91	ReLU, ReLU
MLP2	**3**	**100,128,20**	**RMSProp**	**74317.95**	**0.938**	**Tanh, ReLU, ReLU**
MLP3	4	128,64,32	Adam	82396.33	0.93	ReLU, ReLU, Tanh
MLP4	2	64,32	SGD	91021.15	0.89	ReLU, Tanh
MLP5	3	200,100,50	RMSProp	76320.5	0.92	ReLU, ReLU, ReLU
MLP6	4	100,100,50	Adam	80234.12	0.915	Tanh, ReLU, ReLU
MLP7	3	128,64,16	Nadam	85673.2	0.9	Tanh, ReLU, ReLU
MLP8	4	150,100,75	RMSProp	78923.6	0.92	ReLU, Tanh, ReLU
MLP9	3	256,128,32	Adam	76500.4	0.918	Tanh, ReLU, ReLU
MLP10	2	100,50	Adam	83412.58	0.91	ReLU, ReLU

The table compares different models built to predict house prices, each with varying numbers of hidden layers, nodes, activation functions, and optimizers. These models were evaluated using ValMAE (average prediction error) and R^2 Correlation (how well the model explains the variance in prices).

Why MLP2 Was Selected

MLP2 was chosen because it had the lowest ValMAE (74,317.95) and the highest R^2 (0.938), making it the most accurate and reliable model for predicting house prices.

Key Features of MLP2:

Architecture: With three hidden layers (100, 128, 20 nodes), MLP2 captures complex relationships between house features (like size and location) and prices, offering the right balance of complexity and performance.

Activation Functions: The combination of Tanh in the first layer and ReLU in the others allows the model to learn complex patterns while maintaining stable gradients.

Optimizer: RMSProp with a learning rate of 0.005 helps the model dynamically adjust during training, improving its efficiency and speed in reaching accurate predictions.

Comparison with Other Models

MLP1, with fewer layers and nodes, had higher error (ValMAE: 82,400.32) and a lower R^2 (0.91), making it less effective at capturing data complexity. MLP3, despite having more layers, didn't show better results, with a higher ValMAE (82,396.33). More layers didn't improve accuracy and added unnecessary complexity. MLP5, with more nodes, also had a higher error (ValMAE: 76,320.5), likely due to overfitting. MLP2 strikes the perfect balance between complexity and performance, making it the most accurate and generalizable model.

Why MLP2 is Ideal for Real-World Use:

Accuracy: MLP2's low ValMAE means it can make highly accurate predictions, which is essential in real estate where even small price discrepancies can lead to financial losses.

Generalization: The model avoids overfitting, meaning it can handle new, unseen data effectively, which is crucial when predicting prices for newly listed homes.

Efficiency: The architecture of MLP2 allows for efficient training and deployment without requiring too many computational resources, making it practical for real-time applications.

Scalability: MLP2 can easily scale to handle larger datasets, ensuring it remains effective as more real estate data is added, and can adapt to changing market trends.

conclusion In short, MLP2 combines accuracy, efficiency, and scalability, making it an ideal model for real-world house price prediction tasks.

The follwoing are the models that were tried for testing but did not delivered inferior outputs

MODEL 1

```
def basic_model_1(x_size, y_size):
    t_model = Sequential()
    t_model.add(Dense(50, activation="relu", input_shape=(x_size,)))
    t_model.add(Dense(30, activation="relu"))
    t_model.add(Dense(y_size))
    t_model.compile(
        loss='mean_squared_error',
        optimizer='adam',
        metrics=[metrics.mae])
    return(t_model)
```

MODEL 2

```
def basic_model_2(x_size, y_size):
    t_model = Sequential()
    t_model.add(Dense(100, activation="tanh", input_shape=(x_size,)))
    t_model.add(Dropout(0.2))
    t_model.add(Dense(128, activation="relu"))
    t_model.add(Dense(20, activation="relu"))
    t_model.add(Dense(y_size))
    t_model.compile(
        loss='mean_squared_error',
        optimizer=RMSprop(learning_rate=0.005, rho=0.9, momentum=0.0, epsilon=1e-07, weight_decay=0.0),
        metrics=[metrics.mae])
    return(t_model)
```

MODEL 3

```
def basic_model_3(x_size, y_size):
    t_model = Sequential()
    t_model.add(Dense(128, activation="relu", input_shape=(x_size,)))
    t_model.add(Dense(64, activation="relu"))
    t_model.add(Dense(32, activation="tanh"))
    t_model.add(Dense(y_size))
    t_model.compile(
        loss='mean_squared_error',
        optimizer='adam',
        metrics=[metrics.mae])
    return(t_model)
```

MODEL 4

```
def basic_model_4(x_size, y_size):
    t_model = Sequential()
    t_model.add(Dense(64, activation="relu", input_shape=(x_size,)))
    t_model.add(Dense(32, activation="tanh"))
    t_model.add(Dense(y_size))
    t_model.compile(
        loss='mean_squared_error',
        optimizer='sgd',
        metrics=[metrics.mae])
    return(t_model)
```

MODEL 7

```
def basic_model_7(x_size, y_size):
    t_model = Sequential()
    t_model.add(Dense(128, activation="tanh", input_shape=(x_size,)))
    t_model.add(Dense(64, activation="relu"))
    t_model.add(Dense(16, activation="relu"))
    t_model.add(Dense(y_size))
    t_model.compile(
        loss='mean_squared_error',
        optimizer='nadam',
        metrics=[metrics.mae])
    return(t_model)
```

MODEL 8

```
def basic_model_8(x_size, y_size):
    t_model = Sequential()
```