Application: Model

Case study: Houses for sale

Setup

In [9]:

```
%matplotlib inline

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing

print(tf.__version__)

sns.set_theme(style="ticks", color_codes=True)
```

```
2.4.1
```

Data preparation

See notebook `10a-application-model-data-exploration.ipynb` for details about data preprocessing and data exploration.

In [10]:

```python
ROOT = "https://raw.githubusercontent.com/kirenz/modern-statistics/main/data/"
DATA = "duke-forest.csv"
df = pd.read_csv(ROOT + DATA)

# Drop irrelevant features
df = df.drop(['url', 'address', 'type'], axis=1)

# Convert data types
df['heating'] = df['heating'].astype("category")
df['cooling'] = df['cooling'].astype("category")
df['parking'] = df['parking'].astype("category")

# drop column with too many missing values
df = df.drop(['hoa'], axis=1)

df.columns.tolist()
```

Out[10]:

```
['price',
 'bed',
 'bath',
 'area',
 'year_built',
 'heating',
 'cooling',
```

```
'parking',
'lot']
```

Simple regression

In [11]:

```python
# Select features for simple regression
features = ['area']
X = df[features]

X.info()
print("Missing values:",X.isnull().any(axis = 1).sum())

# Create response
y = df["price"]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 98 entries, 0 to 97
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   area    98 non-null     int64
dtypes: int64(1)
memory usage: 912.0 bytes
Missing values: 0
```

Data splitting

In [12]:

```
from sklearn.model_selection import train_test_split

# Train Test Split
# Use random_state to make this notebook's output identical at every run
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Linear regression

Start with a single-variable linear regression, to predict price from area.
Training a model with tf.keras typically starts by defining the model architecture.
In this case use a keras.Sequential model. This model represents a sequence of steps. In this case there is only one step:

- Apply a linear transformation to produce 1 output using layers.Dense.

The number of inputs can either be set by the input_shape argument, or automatically when the model is run for the first time.

In [13]:

```python
lm = tf.keras.Sequential([
    layers.Dense(units=1, input_shape=(1,))
])

lm.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
```

```
Non-trainable params: 0
_________________________________________________________________
___________________________
```

This model will predict price from area.
Run the untrained model on the first 10 area values. The output won't be good, but you'll see that it has the expected shape, (10,1):

In [14]:

```
lm.predict(X_train[:10])
```

Out[14]:

```
array([[-2667.3628],
       [-2423.789 ],
       [-1526.7021],
       [-2527.6526],
       [-2145.2878],
       [-2129.6624],
       [-1997.3052],
       [-3606.7305],
       [-2290.513 ],
       [-2918.2898]], dtype=float32)
```

In [15]:

```
lm.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.1),
    loss='mean_absolute_error')
```

In [16]:

```
%%time
history = lm.fit(
    X_train, y_train,
    epochs=400,
    # suppress logging
    verbose=0,
    # Calculate validation results on 20% of the training data
    validation_split = 0.1)
```

```
CPU times: user 14.5 s, sys: 747 ms, total: 1
5.3 s
Wall time: 14.9 s
```

In [17]:

```
y_train
```

Out[17]:

```
49    525000
70    520000
68    412500
15    610000
39    535000
       ...
60    509620
71    540000
14    631500
92    590000
51    725000
Name: price, Length: 78, dtype: int64
```

In [18]:

```
# Calculate R squared
from sklearn.metrics import r2_score

y_pred = lm.predict(X_train).astype(np.int64)
y_true = y_train.astype(np.int64)

r2_score(y_train, y_pred)
```

Out[18]:

```
-0.8072585066424944
```

In [19]:

```
# slope coefficient
lm.layers[0].kernel
```

Out[19]:

```
<tf.Variable 'dense_1/kernel:0' shape=(1, 1) dtype=float32, numpy=array([[118.29804]], dtype=float32)>
```

In [20]:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

Out[20]:

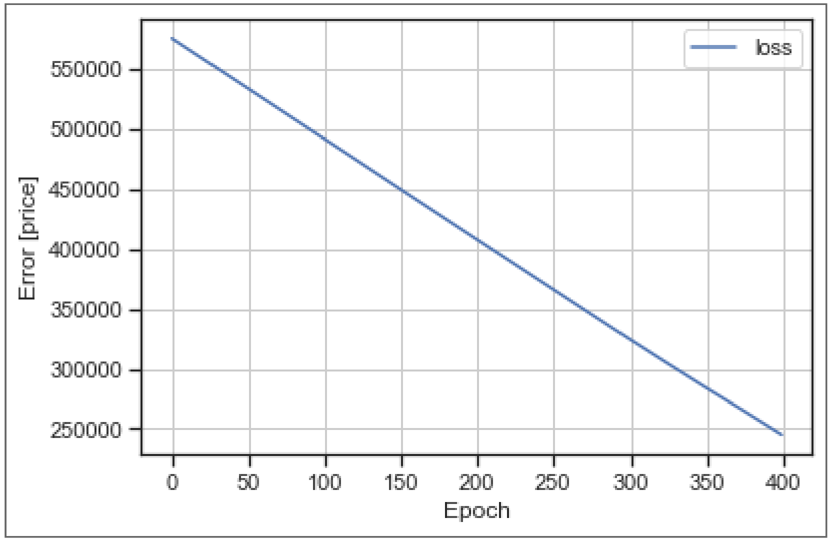
	loss	val_loss	epoch
395	248125.968750	194706.562500	395
396	247328.968750	193920.875000	396
397	246563.859375	193134.828125	397
398	245783.453125	192343.437500	398
399	244997.031250	191549.718750	399

In [21]:

```
def plot_loss(history):
  plt.plot(history.history['loss'], label='loss')
  plt.xlabel('Epoch')
  plt.ylabel('Error [price]')
  plt.legend()
  plt.grid(True)
```

In [22]:

```
plot_loss(history)
```



Collect the results (mean squared error) on the test set, for later:

In [23]:

```
test_results = {}

test_results['lm'] = lm.evaluate(
    X_test,
    y_test, verbose=0)

test_results
```

Out[23]:

```
{'lm': 276413.75}
```

Since this is a single variable regression it's easy to look at the model's predictions as a function of the input:

In [26]:

```
x = tf.linspace(0.0, 6200, 6201)
y = lm.predict(x)

y
```

Out[26]:

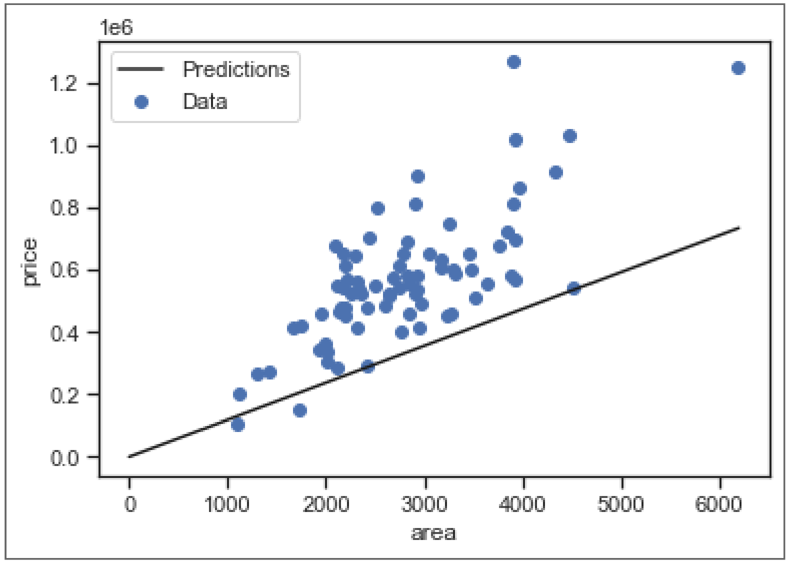
```
array([[1.1863766e+02],
       [2.3693570e+02],
       [3.5523373e+02],
       ...,
       [7.3332988e+05],
       [7.3344819e+05],
       [7.3356650e+05]], dtype=float32)
```

In [27]:

```
def plot_area(x, y):
  plt.scatter(X_train['area'], y_train, label='Data')
  plt.plot(x, y, color='k', label='Predictions')
  plt.xlabel('area')
  plt.ylabel('price')
  plt.legend()
```

In [28]:

```
plot_area(x,y)
```



Multiple Regression

In [29]:

```
# Select all relevant features
features= [
 'bed',
 'bath',
 'area',
 'year_built',
 'cooling',
 'lot'
  ]
X = df[features]

# Convert categorical to numeric
X = pd.get_dummies(X, columns=['cooling'], prefix='cooling', prefix_sep='_')

X.info()
print("Missing values:",X.isnull().any(axis = 1).sum())

# Create response
y = df["price"]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 98 entries, 0 to 97
Data columns (total 7 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   bed               98 non-null     int64  
 1   bath              98 non-null     float64
```

```
 2   area              98 non-null      int64
 3   year_built        98 non-null      int64
 4   lot               97 non-null      float64
 5   cooling_central   98 non-null      uint8
 6   cooling_other     98 non-null      uint8
dtypes: float64(2), int64(3), uint8(2)
memory usage: 4.1 KB
Missing values: 1
```

In [30]:

```
from sklearn.model_selection import train_test_split

# Train Test Split
# Use random_state to make this notebook's output identical at every run
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [31]:

```
lm_2 = tf.keras.Sequential([
    layers.Dense(units=1, input_shape=(7,))
])

lm_2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 1)                 8
=================================================================
Total params: 8
Trainable params: 8
```

```
Non-trainable params: 0
_________________________________________________________________
_____________________________
```

In [32]:

```
lm_2.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.1),
    loss='mean_absolute_error')
```

In [33]:

```
%%time
history = lm_2.fit(
    X_train, y_train,
    epochs=400,
    # suppress logging
    verbose=0,
    # Calculate validation results on 20% of the training data
    validation_split = 0.1)
```

```
CPU times: user 14.5 s, sys: 739 ms, total: 1
5.2 s
Wall time: 14.7 s
```

In [34]:

```
# Calculate R squared
from sklearn.metrics import r2_score

y_pred = lm_2.predict(X_train).astype(np.int64)
y_true = y_train.astype(np.int64)

r2_score(y_train, y_pred)
```

Out[34]:

```
0.2877238455082994
```

In [35]:

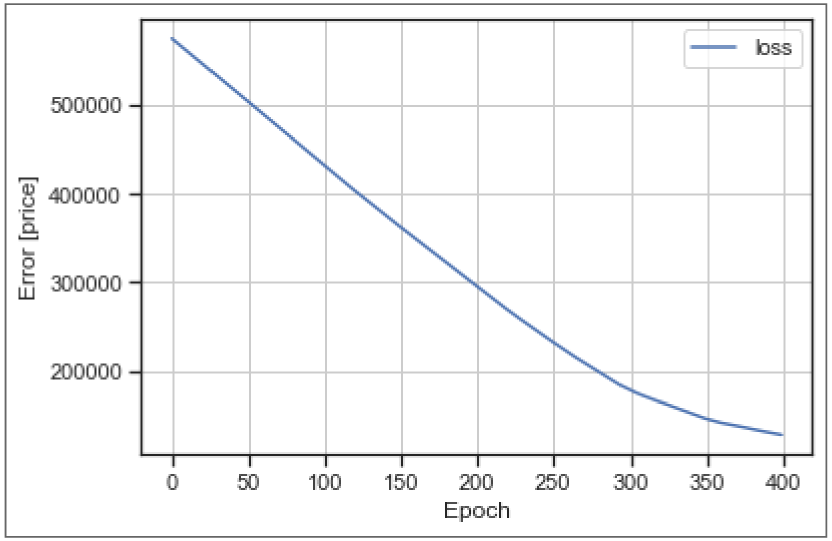
```
# slope coefficients
lm_2.layers[0].kernel
```

Out[35]:

```
<tf.Variable 'dense_2/kernel:0' shape=(7, 1)
dtype=float32, numpy=
array([[105.53154],
       [108.65832],
       [105.82798],
       [104.10174],
       [108.07097],
       [104.49152],
       [ 97.46925]], dtype=float32)>
```

In [36]:

```
plot_loss(history)
```



In [37]:

```
test_results['lm_2'] = lm_2.evaluate(
    X_test, y_test, verbose=0)
```

DNN regression

The previous section implemented linear models for single and multiple inputs. This section implements a multiple-input DNN models. The code is basically the same except the model is expanded to include some "hidden" non-linear layers. The name "hidden" here just means not directly connected to the inputs or outputs.

These models will contain a few more layers than the linear model:

- Two hidden, nonlinear, Dense layers using the relu nonlinearity.
- A linear single-output layer.

In [38]:

```
dnn_model = keras.Sequential([
      layers.Dense(units=1, input_shape=(7,)),
      layers.Dense(64, activation='relu'),
      layers.Dense(64, activation='relu'),
      layers.Dense(1)
  ])

dnn_model.compile(loss='mean_absolute_error',
                optimizer=tf.keras.optimizers.Adam(0.001))
```

In [39]:

```
%%time
history = dnn_model.fit(
    X_train, y_train,
    epochs=100,
    # suppress logging
    verbose=0,
    # Calculate validation results on 20% of the training data
    validation_split = 0.1)
```

```
CPU times: user 4.05 s, sys: 212 ms, total: 4.26 s
Wall time: 4.09 s
```

In [40]:

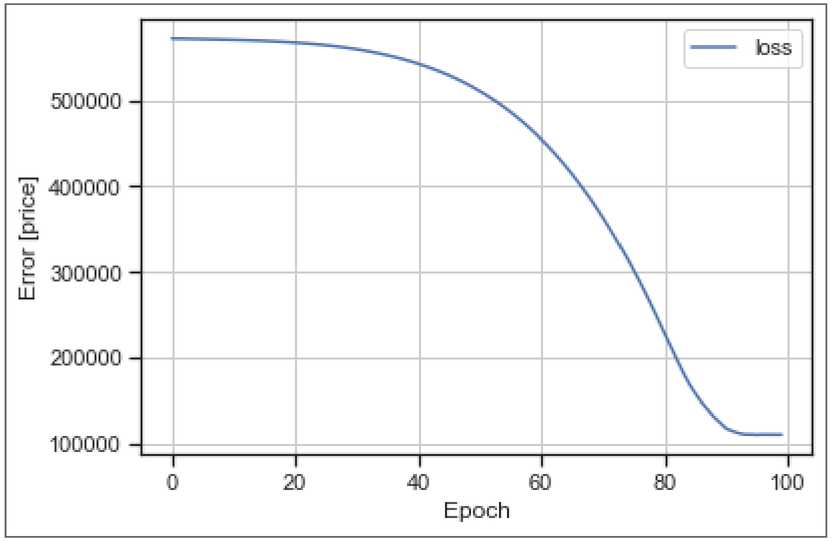
```
# Calculate R squared
from sklearn.metrics import r2_score

y_pred = dnn_model.predict(X_train).astype(np.int64)
y_true = y_train.astype(np.int64)

r2_score(y_train, y_pred)
```

Out[40]:

```
0.4847519024408956
```

In [41]:

```
plot_loss(history)
```



In [42]:

```
test_results['dnn_model'] = dnn_model.evaluate(
    X_test, y_test, verbose=0)
```

Performance comparison

In [43]:

```
pd.DataFrame(test_results, index=['Mean absolute error [price]']).T
```

Out[43]:

	Mean absolute error [price]
lm	276413.750000
lm_2	157647.671875
dnn_model	143017.937500