# Homework 7-8

November 14, 2023

# 1 Homework 7-8

## 1.1 Regression

Load the data.

[ ]:
```
import pandas as pd

df = pd.read_csv('Advertising.csv')
df = df.drop('Unnamed: 0', axis=1)
df.head(5)
```

[ ]:

| | TV | radio | newspaper | sales |
|---|---|---|---|---|
| 0 | 230.1 | 37.8 | 69.2 | 22.1 |
| 1 | 44.5 | 39.3 | 45.1 | 10.4 |
| 2 | 17.2 | 45.9 | 69.3 | 9.3 |
| 3 | 151.5 | 41.3 | 58.5 | 18.5 |
| 4 | 180.8 | 10.8 | 58.4 | 12.9 |

Scale the features.

[ ]:
```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

x = np.array(df[['TV', 'radio', 'newspaper']])
y = np.array(df[['sales']])

x_scaled = MinMaxScaler().fit_transform(x)
```

Split scaled data into training and testing splits.

[ ]:
```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(
    x_scaled, y, test_size=.2, random_state=42
)
```

### 1.1.1 Exercise 1

Fit linear regression to training data.

[ ]:
```
from sklearn.linear_model import LinearRegression

ex1_model = LinearRegression().fit(x_train, y_train)
```

Read off the learned model parameters.

[ ]:
```
print(f'Coefficients: {ex1_model.coef_}')
print(f'Intercept: {ex1_model.intercept_}')
```

```
Coefficients: [[13.22651832  9.38407469  0.3139387 ]]
Intercept: [3.01120633]
```

Compare $R^2$ scores of training and test data.

[ ]:
```
print(f'Training R^2: {ex1_model.score(x_train, y_train)}')
print(f'Test R^2: {ex1_model.score(x_test, y_test)}')
```

```
Training R^2: 0.8957008271017817
Test R^2: 0.8994380241009119
```

The training and testing $R^2$ are both fairly close to 1 and similar in value (though for different values of `random_state`, that is, different training and testing splits, the test $R^2$ tends to be a bit lower than the training $R^2$).

### 1.1.2 Exercise 2

Fit $K$-nearest neighbors model to the training data with $K = 1$.

[ ]:
```
from sklearn.neighbors import KNeighborsRegressor

ex2_model = KNeighborsRegressor(n_neighbors=1).fit(x_train, y_train)
```

Compare $R^2$ scores for training and testing data.

[ ]:
```
print(f'Training R^2: {ex2_model.score(x_train, y_train)}')
print(f'Test R^2: {ex2_model.score(x_test, y_test)}')
```

```
Training R^2: 1.0
Test R^2: 0.9025380308603167
```

$K$-nearest neighbors is perfect ($R^2 = 1$) on the training data when $K = 1$ by construction. On the test data, however, its performance is similar to that of the linear regression in Exercise 1.

Repeat the above for $K \in \{2, 3, \ldots, 10\}$.

[ ]:
```
ex2_models = [ex2_model]
for k in range(2, 11):
  ex2_models.append(KNeighborsRegressor(k).fit(x_train, y_train))
```

Print $R^2$ scores from each model.

```
for k in range(10):
  print(f'K = {k + 1}')
  print(f'Training R^2: {ex2_models[k].score(x_train, y_train)}')
  print(f'Test R^2: {ex2_models[k].score(x_test, y_test)}')
  print()
```

```
K = 1
Training R^2: 1.0
Test R^2: 0.9025380308603167

K = 2
Training R^2: 0.9812430719552749
Test R^2: 0.9324894681867459

K = 3
Training R^2: 0.9718620977717801
Test R^2: 0.9364836092038614

K = 4
Training R^2: 0.9671987204202612
Test R^2: 0.9299073794472469

K = 5
Training R^2: 0.9665527977251915
Test R^2: 0.9337192077502264

K = 6
Training R^2: 0.9632464215142899
Test R^2: 0.9498780845328242

K = 7
Training R^2: 0.9597421955914276
Test R^2: 0.9503836147120786

K = 8
Training R^2: 0.9570375379536363
Test R^2: 0.9461417178612622

K = 9
Training R^2: 0.9512721124742588
Test R^2: 0.9420403901172719

K = 10
Training R^2: 0.9477586389705528
Test R^2: 0.9382764359650904
```

For $K > 1$, the $K$-nearest neighbors regression is no longer perfect on the training data. Indeed, as

$K$ increases the training $R^2$ decreases, but the test $R^2$ tends to increase (though it is still always lower than the training $R^2$).

### 1.1.3 Exercise 3

Create linear model using `keras`.

[ ]:
```
import tensorflow as tf

ex3_model = tf.keras.Sequential(layers=[
    tf.keras.layers.Input(3), tf.keras.layers.Dense(1)
], name='Exercise3Model')

optim = tf.keras.optimizers.Adam(learning_rate=.15)
ex3_model.compile(loss='mse', optimizer=optim)
```

Train with 5000 epochs.

[ ]:
```
x_train_t = tf.convert_to_tensor(x_train)
y_train_t = tf.convert_to_tensor(y_train)

ex3_model.fit(
    x_train_t, y_train_t,
    epochs=5000,
    batch_size=len(x_train),
    verbose=0
)
```

[ ]:
```
<keras.src.callbacks.History at 0x2b15fef9610>
```

Get the trained parameters.

[ ]:
```
print(f'Weights: {ex3_model.layers[0].weights[0].value()}')
print(f'Bias: {ex3_model.layers[0].bias.value()}')
```

```
Weights: [[13.226515  ]
 [ 9.384072  ]
 [ 0.31393996]]
Bias: [3.0112088]
```

These parameters are comparable to those obtained using ordinary least squares (Exercise 1).

[ ]:
```
from sklearn.metrics import r2_score

print(f'Training R^2: {r2_score(y_train, ex3_model.predict(x_train))}')
print(f'Test R^2: {r2_score(y_test, ex3_model.predict(x_test))}')
```

```
5/5 [==============================] - 0s 499us/step
5/5 [==============================] - 0s 499us/step
Training R^2: 0.8957008421597463
```

```
2/2 [==============================] - 0s 1ms/step
Test R^2: 0.8994379998671596
```

The $R^2$ values are similar for the training and test splits and similar to the $R^2$ values obtained in Exercise 1.

Overall, this model is nearly the same as that obtained in Exercise 1.

#### 1.1.4 Exercise 4

Create the model.

```
[ ]: ex4_model = tf.keras.Sequential([
         tf.keras.layers.Input(3),
         tf.keras.layers.Dense(128, activation='relu'),
         tf.keras.layers.Dense(64, activation='relu'),
         tf.keras.layers.Dense(32, activation='relu'),
         tf.keras.layers.Dense(1)
     ], name='Exercise4Model')

     optim = tf.keras.optimizers.Adam(learning_rate=.05)
     ex4_model.compile(loss='mse', optimizer=optim)
```

Train for 5000 epochs.

```
[ ]: ex4_model.fit(
         x_train_t, y_train_t,
         batch_size=len(x_train), verbose=0,
         epochs=5000
     )
```

```
[ ]: <keras.src.callbacks.History at 0x2b161490590>
```

Get $R^2$ scores for training and testing data.

```
[ ]: print(f'Training R^2: {r2_score(y_train_t, ex4_model.predict(x_train_t,␣
      ↪verbose=0))}')
     print(f'Test R^2: {r2_score(y_test, ex4_model.predict(x_test, verbose=0))}')
```

```
Training R^2: 0.9969724869682755
Test R^2: 0.988450162299337
```

This model performs much better on both training and testing data than any of the previous ones.

### 1.2 Classification

Load the data, scale features, and split into training and testing data.

```
[ ]: df = pd.read_csv('diagnosis.csv')
     df.head(5)
```

```
[ ]:          id diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
     0    842302         M        17.99         10.38          122.80     1001.0
     1    842517         M        20.57         17.77          132.90     1326.0
     2  84300903         M        19.69         21.25          130.00     1203.0
     3  84348301         M        11.42         20.38           77.58      386.1
     4  84358402         M        20.29         14.34          135.10     1297.0

        smoothness_mean  compactness_mean  concavity_mean  concave points_mean  \
     0          0.11840           0.27760          0.3001              0.14710
     1          0.08474           0.07864          0.0869              0.07017
     2          0.10960           0.15990          0.1974              0.12790
     3          0.14250           0.28390          0.2414              0.10520
     4          0.10030           0.13280          0.1980              0.10430

        …  texture_worst  perimeter_worst  area_worst  smoothness_worst  \
     0  …          17.33           184.60      2019.0            0.1622
     1  …          23.41           158.80      1956.0            0.1238
     2  …          25.53           152.50      1709.0            0.1444
     3  …          26.50            98.87       567.7            0.2098
     4  …          16.67           152.20      1575.0            0.1374

        compactness_worst  concavity_worst  concave points_worst  symmetry_worst  \
     0             0.6656           0.7119                0.2654          0.4601
     1             0.1866           0.2416                0.1860          0.2750
     2             0.4245           0.4504                0.2430          0.3613
     3             0.8663           0.6869                0.2575          0.6638
     4             0.2050           0.4000                0.1625          0.2364

        fractal_dimension_worst  Unnamed: 32
     0                  0.11890          NaN
     1                  0.08902          NaN
     2                  0.08758          NaN
     3                  0.17300          NaN
     4                  0.07678          NaN

     [5 rows x 33 columns]
```

Get relevant features.

```
[ ]: # output class is 1 if diagnoisis is malignant, 0, if benign
     y = np.where(df['diagnosis'] == 'M', 1., 0.)
     x = np.array(df[['radius_mean', 'texture_mean', 'smoothness_mean']])
```

Scale features using `MinMaxScaler`.

```
[ ]: x_scaled = MinMaxScaler().fit_transform(x)
```

Split data into train and test splits.

```
[ ]: x_train, x_test, y_train, y_test = train_test_split(x_scaled, y, test_size=.2,
      random_state=42)
```

### 1.2.1 Exercise 5

Perform logistic regression on the training data.

```
[ ]: from sklearn.linear_model import LogisticRegression

     ex5_model = LogisticRegression(penalty=None).fit(x_train, y_train)
```

Print learned parameters.

```
[ ]: print(f'Coefficients: {ex5_model.coef_}')
     print(f'Intercept: {ex5_model.intercept_}')
```

```
Coefficients: [[27.33747378 10.76746566 15.43661982]]
Intercept: [-19.74212166]
```

Compute accuracy on training and testing data.

```
[ ]: print(f'Training accuracy: {ex5_model.score(x_train, y_train)*100 : .02f}%')
     print(f'Test accuracy: {ex5_model.score(x_test, y_test)*100 : .02f}%')
```

```
Training accuracy:  92.75%
Test accuracy:  94.74%
```

The training and test accuracies are both pretty good (>90%) and are similar. By chance, we got slightly better testing accuracy than training accuracy.

### 1.2.2 Exercise 6

Create and train a $K$-nearest neighbors model with $K = 1$.

```
[ ]: from sklearn.neighbors import KNeighborsClassifier

     ex6_model = KNeighborsClassifier(1).fit(x_train, y_train)
```

Compute training and test accuracy.

```
[ ]: print(f'Training accuracy: {ex6_model.score(x_train, y_train)*100 : .02f}%')
     print(f'Test accuracy: {ex6_model.score(x_test, y_test)*100 : .02f}%')
```

```
Training accuracy:  100.00%
Test accuracy:  92.98%
```

As we saw in the case of KNN regression, the KNN method is perfect (100% accuracy) on training data with $K = 1$ by construction. It still achieves high accuracy (but not 100%!) on the test data. The accuracy is comparable to that achieved by the logistic regression.

Repeat for $K \in \{2, 3, \ldots, 9\}$.

[ ]:
```python
ex6_models = []
for k in range(1, 12):
  ex6_models.append(KNeighborsClassifier(k).fit(x_train, y_train))

for k in range(len(ex6_models)):
  print(f'K = {k + 1}')
  print(f'Training accuracy: {ex6_models[k].score(x_train, y_train)*100 : .
 ↪02f}%')
  print(f'Test accuracy: {ex6_models[k].score(x_test, y_test)*100 : .02f}%')
  print()
```

```
K = 1
Training accuracy:  100.00%
Test accuracy:  92.98%

K = 2
Training accuracy:  94.29%
Test accuracy:  86.84%

K = 3
Training accuracy:  93.85%
Test accuracy:  90.35%

K = 4
Training accuracy:  92.53%
Test accuracy:  91.23%

K = 5
Training accuracy:  92.75%
Test accuracy:  92.98%

K = 6
Training accuracy:  91.87%
Test accuracy:  92.98%

K = 7
Training accuracy:  92.75%
Test accuracy:  94.74%

K = 8
Training accuracy:  91.65%
Test accuracy:  95.61%

K = 9
Training accuracy:  92.97%
Test accuracy:  95.61%

K = 10
```

```
Training accuracy:  91.87%
Test accuracy:  94.74%

K = 11
Training accuracy:  92.97%
Test accuracy:  95.61%
```

A similar trend to what occurred with KNN regression is observable here: as $K$ increases, training accuracy tends to decrease while test accuracy tends to increase. The training accuracy seems to level out at around 92% for $K \geq 4$, and the test accuracy seems to top out at around 95% for $K \geq 8$.

### 1.2.3 Exercise 7

Create one-layer neural network with sigmoid activation function and binary cross-entropy loss function.

[ ]:
```
ex7_model = tf.keras.models.Sequential([
    tf.keras.layers.Input(3),
    tf.keras.layers.Dense(1, activation='sigmoid'),
], name='Exercise7Model')

optim = tf.keras.optimizers.Adam(learning_rate=.1)
ex7_model.compile(loss='bce', optimizer=optim, metrics=['accuracy'])
```

Train for 5000 epochs on the training data.

[ ]:
```
x_train_t = tf.convert_to_tensor(x_train)
y_train_t = tf.convert_to_tensor(y_train)

ex7_model.fit(
    x_train_t, y_train_t,
    batch_size=len(x_train),
    epochs=5000,
    verbose=0
)
```

[ ]:
```
<keras.src.callbacks.History at 0x2b162857010>
```

Read the network parameters.

[ ]:
```
print(f'Weights: {ex7_model.layers[0].weights[0].value()}')
print(f'Bias: {ex7_model.layers[0].bias.value()}')
```

```
Weights: [[27.33742 ]
 [10.767428]
 [15.436617]]
Bias: [-19.742088]
```

These parameters are nearly the same as those obtained using `LogisticRegression`.

[ ]:
```
print(f'Training accuracy: {ex7_model.evaluate(x_train_t, y_train_t,␣
↪verbose=0)[1]*100 : .02f}%')
print(f'Test accuracy: {ex7_model.evaluate(x_test, y_test, verbose=0)[1]*100 : .
↪02f}%')
```

```
Training accuracy:  92.75%
Test accuracy:  94.74%
```

The training and test accuracies are, of course, the same as those obtained from `LogisticRegression` in Exercise 5, as this model is virtually the same model – the parameters were optimized by a different method, but we have already seen that the end result was nearly the same.

Train with 12000 epochs.

[ ]:
```
ex7_model_long = tf.keras.models.Sequential([
    tf.keras.layers.Input(3),
    tf.keras.layers.Dense(1, activation='sigmoid'),
], name='Exercise7Model_long')

optim = tf.keras.optimizers.Adam(learning_rate=.1)
ex7_model_long.compile(loss='bce', optimizer=optim, metrics=['accuracy'])

ex7_model_long.fit(
    x_train_t, y_train_t,
    batch_size=len(x_train),
    epochs=12000,
    verbose=0
)
```

[ ]:
```
<keras.src.callbacks.History at 0x2b166bb6850>
```

Show learned parameters and training and test accuracies.

[ ]:
```
print(f'Weights: {ex7_model_long.layers[0].weights[0].value()}')
print(f'Bias: {ex7_model_long.layers[0].bias.value()}')

print(f'Training accuracy: {ex7_model_long.evaluate(x_train_t, y_train_t,␣
↪verbose=0)[1]*100 : .02f}%')
print(f'Test accuracy: {ex7_model_long.evaluate(x_test, y_test,␣
↪verbose=0)[1]*100 : .02f}%')
```

```
Weights: [[27.33744 ]
 [10.767424]
 [15.436618]]
Bias: [-19.74212]
Training accuracy:  92.75%
Test accuracy:  94.74%
```

The parameters and corresponding training and test accuracies are the same as those obtained when training for only 5000 epochs; evidently, the optimization converges before 5000 epochs have passed.

### 1.2.4 Exercise 8

Create model with three hidden layers of size 128, 64, and 32, and one output layer.

```
[ ]: ex8_model = tf.keras.models.Sequential([
         tf.keras.layers.Input(3),
         tf.keras.layers.Dense(128, activation='relu'),
         tf.keras.layers.Dense(64, activation='relu'),
         tf.keras.layers.Dense(32, activation='relu'),
         tf.keras.layers.Dense(1, activation='sigmoid')
     ], 'Exercise8Model')


     optim = tf.keras.optimizers.Adam(learning_rate=.1)
     ex8_model.compile(loss='bce', optimizer=optim, metrics=['accuracy'])
```

Train for 5000 epochs on the training data.

```
[ ]: ex8_model.fit(
         x_train_t, y_train_t,
         batch_size=len(x_train),
         epochs=5000,
         verbose=0
     )
```

```
[ ]: <keras.src.callbacks.History at 0x2b16896c050>
```

Compute training and test accuracies.

```
[ ]: print(f'Training accuracy: {ex8_model.evaluate(x_train_t, y_train_t,␣
      ↪verbose=0)[1]*100 : .02f}%')
     print(f'Test accuracy: {ex8_model.evaluate(x_test, y_test, verbose=0)[1]*100 : .
      ↪02f}%')
```

```
Training accuracy:  94.07%
Test accuracy:  94.74%
```

This model achieves slightly higher training and test accuracy than the model from Exercise 7. It has a lot more parameters, though:

```
[ ]: ex7_model.summary()
```

```
Model: "Exercise7Model"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 1)                 4
```

```
=================================================================
Total params: 4 (16.00 Byte)
Trainable params: 4 (16.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 1)                 4

=================================================================
Total params: 4 (16.00 Byte)
Trainable params: 4 (16.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

[ ]:
```
ex8_model.summary()
```

```
Model: "Exercise8Model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_7 (Dense)             (None, 128)               512

 dense_8 (Dense)             (None, 64)                8256

 dense_9 (Dense)             (None, 32)                2080

 dense_10 (Dense)            (None, 1)                 33

=================================================================
Total params: 10881 (42.50 KB)
Trainable params: 10881 (42.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_7 (Dense)             (None, 128)               512

 dense_8 (Dense)             (None, 64)                8256

 dense_9 (Dense)             (None, 32)                2080

 dense_10 (Dense)            (None, 1)                 33

=================================================================
Total params: 10881 (42.50 KB)
```

```
Trainable params: 10881 (42.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```