Department of Electronic & Telecommunication Engineering,
University of Moratuwa, Sri Lanka.

Learning from Data and Related Challenges and Linear Models for Regression

Abeysinghe D.U. 210011X

Submitted in partial fulfillment of the requirements for the module
EN 3150 Pattern Recognition

2nd September 2024

1 Data pre-processing

1.1 Scaling Methods

1.1.1 Standard Scaling

Standard scaling is used to transform data to have a zero mean and a standard deviation of one. This method is useful for algorithms which assume normally distributed data.

$$X_{\text{standardized}} = \frac{X - \mu}{\sigma}$$

1.1.2 Min-Max Scaling

Min-max scaling is used to transform features to a fixed range, usually [0,1]. However, this method is sensitive to outliers since they can distort the scaling significantly.

$$X_{\text{normalized}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

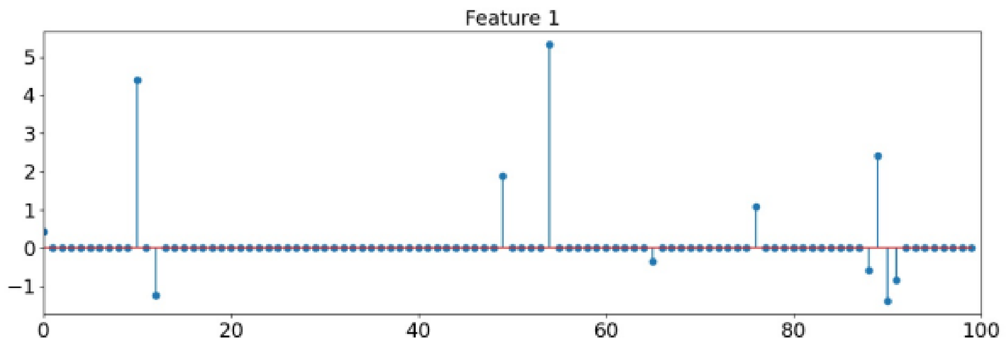
1.1.3 Max-Abs Scaling

In this method, each feature is divided by its maximum absolute value. This method preserves data sparsity.

$$X_{\text{max abs}} = \frac{X}{|X_{\text{max}}|}$$

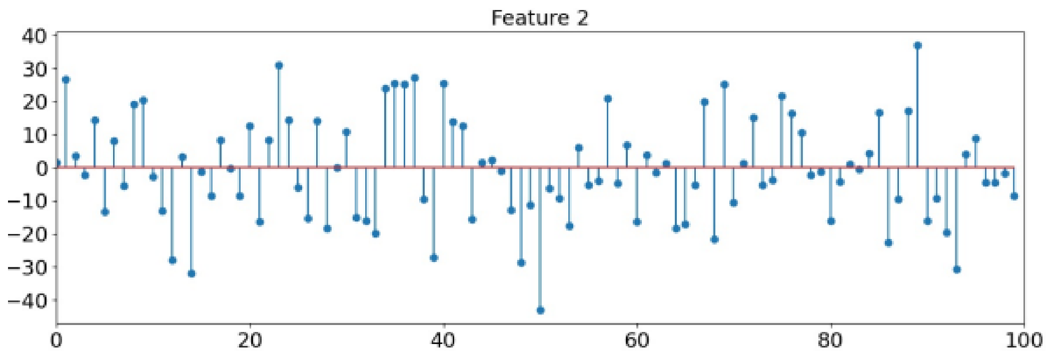
1.2 Selecting a scaling method with feature values

1.2.1 feature 1



In this feature, it can be observed that most of the feature values are 0. Hence this dataset can be considered sparse. Max-abs scaling is the most suitable method to preserve the sparsity of this dataset.

1.2.2 feature 2



This feature displays a normally distributed behavior. Therefore this feature can be scaled using the standard scaler.

2 Learning from data

2.1 Data Generation

The following code is used to generate data.

```
import numpy as np
import matplotlib . pyplot as plt
from sklearn . model_selection import train_test_split
from sklearn . linear_model import LinearRegression
# Generate 100 samples
n_samples = 100
# Generate X values ( uniformly distributed between 0
and 10)
X = 10 * np . random . rand ( n_samples , 1)
# Generate epsilon values ( normally distributed with
mean 0 and standard deviation 15)
epsilon = np . random . normal (0 , 15 , n_samples )
# Generate Y values using the model Y = 3 + 3X +
epsilon
Y = 3 + 2 * X + epsilon [: , np . newaxis ]
```

2.2 Data Visualization

The data can be visualized using the following code.

```
    #Data visualization

r = np.random.randint(104)

X_train, X_test, Y_train, Y_test = train_test_split(X,Y,
   test_size=0.2, random_state=r)

#plot the data points

plt.figure(figsize = (10,6))

plt.scatter( X_train , Y_train , alpha =1 , marker ='o', color =
   'red', label ='Training␣Data␣')
plt.scatter( X_test , Y_test , alpha =1 , marker ='s', color ='
   blue', label ='Testing␣Data␣')
plt.show ()
```

We can run the code multiple times and observe the behavior of the scatter plot.

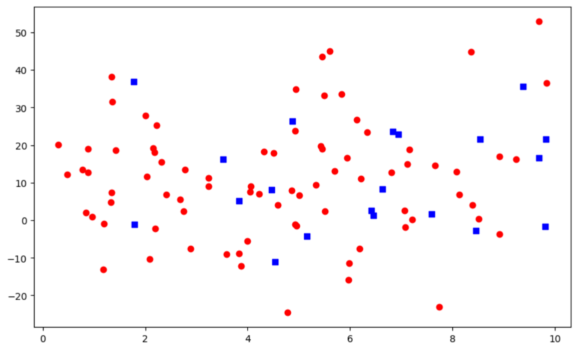


Figure 1: Visualized data from the first iteration

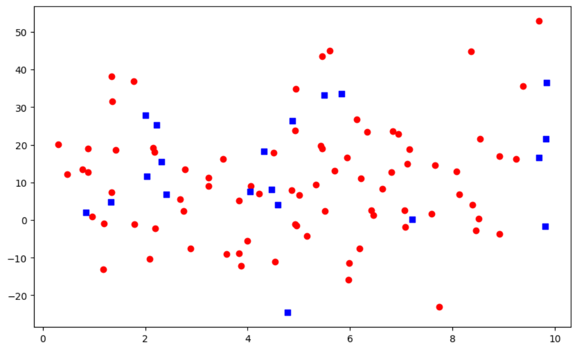


Figure 2: Visualized data from the second iteration

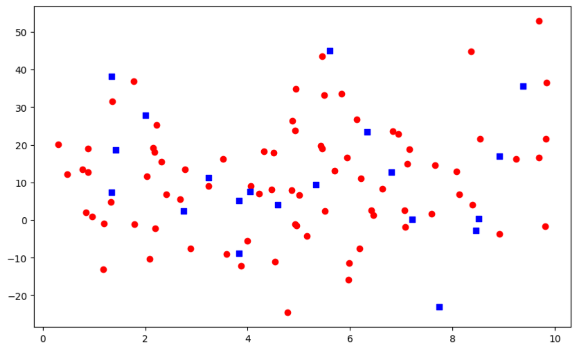


Figure 3: Visualized data from the third iteration

It can be observed that the scatter plots are different for each of the three iterations. This is due to the line `X = 10 * np.random.rand(n_samples,1)` which generates random values for X uniformly distributed between 0 and 10 for each iteration. Also the line `epsilon = np.random.normal(0,15,n_samples)` also generates random noise thus leading to differences in the three scatter plots.

2.3 Linear Regression

The following code is used to fit a linear regression model to the synthetic dataset.

```python
for i in range(10):
  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
     test_size=0.2, random_state = np.random.randint(104))
  model = LinearRegression()
  model.fit(X_train, Y_train)
  Y_pred_train = model.predict(X_train)
  plt.plot(X_train, Y_pred_train, label = f'LR␣{i+1}')

plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```

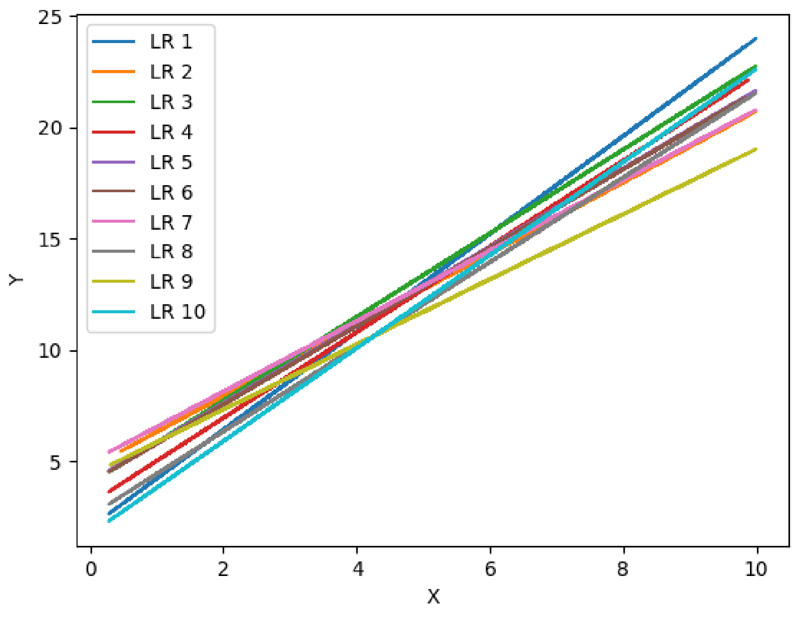


Figure 4: LR model fitting for the ten iterations

It can be observed that for each iteration the linear regression model is different. This is due to the fact that we are using a different value for the `random_state` variable. This leads to different test and training sets for the model in each iteration. Thus giving us the above output.

2.4 Increasing the number of Data Samples

The code in listing 1 is altered such that `n_samples` is set to 10000.

```
import numpy as np
import matplotlib . pyplot as plt
from sklearn . model_selection import train_test_split
from sklearn . linear_model import LinearRegression
# Generate 100 samples
n_samples = 1000
# Generate X values ( uniformly distributed between 0
and 10)
X = 10 * np . random . rand ( n_samples , 1)
# Generate epsilon values ( normally distributed with
mean 0 and standard deviation 15)
epsilon = np . random . normal (0 , 15 , n_samples )
# Generate Y values using the model Y = 3 + 3X +
epsilon
Y = 3 + 2 * X + epsilon [: , np . newaxis ]
```

After running the code in listing 3 again, the following output is obtained.

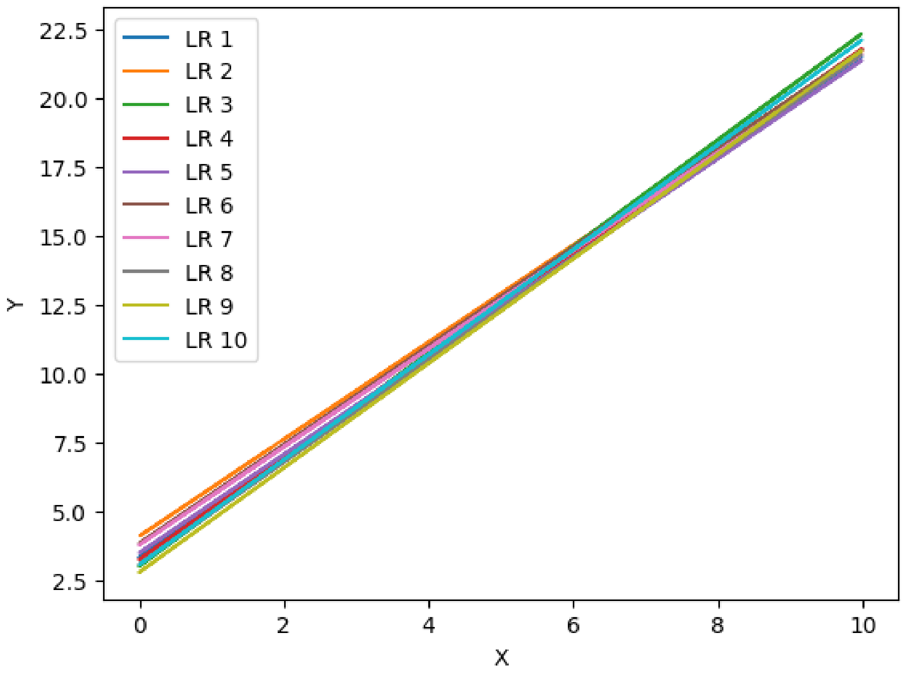


Figure 5: LR model fitting for `n_samples = 10000`

It can be seen that with a larger dataset, the model shows higher accuracy and even with the randomness in the iterations, shows similar regression lines. This is because the larger dataset allows the model to learn more accurately while reducing the risk of overfitting.

3 Linear Regression on Real World Data

3.1 Load the Dataset

We can load the dataset by using the following code.

```python
from ucimlrepo import fetch_ucirepo

# fetch dataset
infrared_thermography_temperature = fetch_ucirepo(id=925)

# data (as pandas dataframes)
X = infrared_thermography_temperature.data.features
y = infrared_thermography_temperature.data.targets

# metadata
print(infrared_thermography_temperature.metadata)

# variable information
print(infrared_thermography_temperature.variables)
```

3.2 Independent and Dependent Variables

The data set has 33 features.

Dependent Variables = 2
Independent Variables = Total Features - Dependent Variables

$$\text{Independent Variables} = 33 - 2 = 31$$

3.3 Is it possible to apply Linear Regression?

It is not possible to apply linear regression to the entire dataset since there are non-numerical data such as **gender** and **ethnicity**.
We can use encoding techniques to convert such data to numerical data before proceeding to linear regression. For example one-hot encoding can be used to convert the **"gender"** data into binary data of 1s and 0s.

3.4 Correct approach to drop missing values

The code given in **listing 5** uses `X = x.dropna()` and `y = y.dropna()` to drop the NaN values.

This can lead to the NaN values being dropped separately for X and y thus making misalignments in the columns.

To fix this, we must concatenate X and y into a single dataframe, drop the NaN values and separate them again.

The following block of code can be used to do this.

```python
import pandas as pd

data = pd.concat([X, y], axis = 1)

data = data.dropna()

X_clean = data.iloc[:, :-1]
y_clean = data.iloc[:, -1]
```

3.5 Selection of the Dependent feature and Independent features

"aveOralM" was selected as the dependent variable.

'Age', 'Humidity', 'T_FHC_Max1', 'T_FH_Max1', 'T_OR_Max1' were selected as the independent features.

Since "Age" is a categorical feature, we must convert it to a numerical value before using it in training our model. This was done using the following code.

```python
df = pd.DataFrame(data)

# Function to convert age range to mean
def age_range_to_mean(age_range):
    if pd.isna(age_range):  # Check for NaN values
        return None  # Return None for NaN
    if age_range.startswith('>'):
        # Handle case for ">60"
        return 65
    elif age_range.startswith('<'):
        # Handle case for "<21"
        return 18
    else:
        # Split the string into lower and upper bounds
        lower, upper = map(int, age_range.split('-'))
        # Calculate the mean
        return (lower + upper) / 2

# Apply the function to the Age column
df['Age'] = df['Age'].apply(age_range_to_mean)

df = df.dropna(subset=['Age', 'aveOralM'])

X = df
y = df[['aveOralM','aveOralM']]
```

The following code was used to select the dependent variable and the independent variables.

```python
# Select dependent feature as 'aveOralM'
y = y['aveOralM']

# Select independent features
X = X[['Age', 'Humidity', 'T_FHC_Max1', 'T_FH_Max1', 'T_OR_Max1'
    ]]
```

3.6 Splitting of data points

The data was split as 80% for training and 20% for testing.

```python
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=32)
```

The dataset is then scaled using the following code.

```python
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Fit the scaler on the training data and transform both the
    training and testing data
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

3.7 Training a linear Regression Model

A linear regression model was trained and the coefficients corresponding to the independent variables were estimated.

```python
# linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

coefficients = model.coef_[0]

# Print the estimated coefficients
for feature, coef in zip(X.columns, coefficients):
    print(f"Coefficient␣for␣{feature}:␣{coef}")
```

- Coefficient for Age: -0.00041355365991 38827
- Coefficient for Humidity: 0.0007692648278966596
- Coefficient for T_FHC_Max1: -0.037029890496185366
- Coefficient for T_FH_Max1: 0.1687616396235495
- Coefficient for T_OR_Max1: 0.2980422493799061

3.8 Which independent variable contributes highly for the dependent feature?

The dependent feature is mostly affected by the **T_OR_Max1** feature. Compared to the other independent variables it has the greatest absolute coefficient value.

3.9 Training a new regression model

A new model was trained for the given set of independent variables.

```python
df = pd.DataFrame(data)

#Training new model
y = df['aveOralM']

# Select independent features
X = df[['T_OR1', 'T_OR_Max1', 'T_FHC_Max1', 'T_FH_Max1']]


# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=32)

from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Fit the scaler on the training data and transform both the
    training and testing data
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# linear regression model
model = LinearRegression()
model.fit(X_train_scaled, y_train)

coefficients = model.coef_

# Print the estimated coefficients
for feature, coef in zip(X.columns, coefficients):
    print(f"Coefficient␣for␣{feature}:␣{coef}")
```

The following were obtained as the corresponding coefficients.

- Coefficient for T_OR1: 0.0943972549176098
- Coefficient for T_OR_Max1: 0.2039501045965917
- Coefficient for T_FHC_Max1: -0.03731875656457051
- Coefficient for T_FH_Max1: 0.16876142557362245

3.10 Calculating key parameters for the new model

- Residual sum of squares (RSS): 47907.28507533431
- Residual Standard Error (RSE): 15.515802460933338
- Mean Squared Error (MSE): 234.839632722227
- R squared statistic: -717.443004118583
- t-statistics for each feature:
  - const 9.893784
  - T_OR1 0.201601
  - T_OR_Max1 0.436282
  - T_FHC_Max1 -1.537039
  - T_FH_Max1 6.970662
- p-values for each feature:
  - const 7.275325e-22
  - T_OR1 8.402795e-01
  - T_OR_Max1 6.627488e-01
  - T_FHC_Max1 1.246747e-01
  - T_FH_Max1 6.549471e-12

  dtype: float64

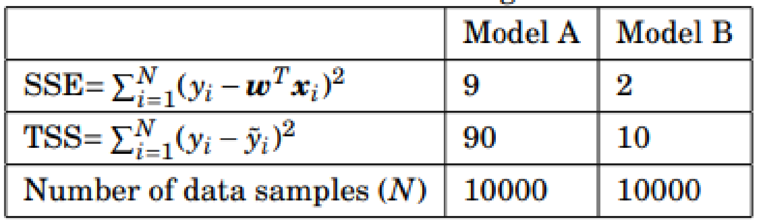
3.11 Will you be able to discard any features based on p-value ?

Since the p-values of T_OR1, T_OR_Max1, and T_FHC_Max1 exceed the significance threshold of 0.05, we can discard these independent variables.

4 Performance Evaluation of Linear Regression

4.1 Performance of the following Models will be evaluated.

Table 1: SSE and TSS of linear regression models.



	Model A	Model B
SSE= $\sum_{i=1}^{N}(y_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2$	9	2
TSS= $\sum_{i=1}^{N}(y_i - \bar{y}_i)^2$	90	10
Number of data samples (N)	10000	10000

4.2 Residual Standard Error of the Models

$$RSE_{\text{A}} = \sqrt{\frac{9}{10000-2-1}} = \sqrt{\frac{9}{9997}} \approx \sqrt{0.0009003} \approx 0.03$$

$$RSE_{\text{B}} = \sqrt{\frac{2}{10000-4-1}} = \sqrt{\frac{2}{9995}} \approx \sqrt{0.0002001} \approx 0.01414$$

Since Model B has a lower RSE, we can conclude it performs better than A.

4.3 R-squared for models A and B

$$R^2 = 1 - \frac{\text{SSE}}{\text{TSS}}$$

$$R_A^2 = 1 - \frac{9}{90} = 1 - 0.1 = 0.9$$

$$R_B^2 = 1 - \frac{2}{10} = 1 - 0.2 = 0.8$$

Since model A has a higher R-squared value, we can say it performs better than model B.

4.4 Comparison between RSE and R-squared

The R^2 value shows how much of the variation in the data is explained by the model, while the Residual Standard Error (RSE) measures the model's accuracy in the same units as the outcome variable. Since R^2 is dimensionless and accounts for the variability explained by the model, it is more suitable for comparing different models.

5 Linear regression impact on outliers

5.1 $L_1(w)$ and $L_2(w)$ as $a \to 0$

For $L_1(w)$ as $a \to 0$,

$$L_1(w) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\frac{r_i^2}{r_i^2}\right) = \frac{1}{N}\sum_{i=1}^{N} 1 = 1$$

In the case of large residuals ($a \ll r_i$), $L_1(w)$ becomes insensitive to the magnitude of the residuals, treating large and small residuals equally. Consequently, the loss function is no longer effective at differentiating between them.

For $L_2(w)$ as $a \to 0$,

$$L_2(w) \approx \frac{1}{N}\sum_{i=1}^{N}(1 - 0) = 1$$

Similarly, when residuals are large ($a \ll r_i$), $L_2(w)$ also loses sensitivity to the residuals' magnitude. This results in both large and small residuals being treated similarly, making it ineffective at distinguishing between them.

5.2 Reducing the Influence of Outliers

As $L_2(w)$ becomes less sensitive to large residuals when a increases, it is important to minimize the impact of outliers. Compared to $L_1(w)$, the $L_2(w)$ function with $a = 25$ gives a better balance by reducing the penalty for large residuals while still maintaining sensitivity to smaller ones. This makes $L_2(w)$ more effective in handling the influence of outliers on the model.