Data Science: Principles and Practice

Lecture 2: Linear Regression

Ekaterina Kochmar

1 Based on slides from Marek Rei
Two Sigma: Using News to Predict Stock Movements
Use news analytics to predict stock price performance
$100,000
1,349 teams

Airbus Ship Detection Challenge
Find ships on satellite images as quickly as possible
$60,000
681 teams

Google Analytics Customer Revenue Prediction
Predict how much GStore customers will spend
$45,000
3,338 teams

Human Protein Atlas Image Classification
$37,000

kaggle.com
drivendata.org
Practical Data Science

- Kaggle datasets (https://www.kaggle.com/datasets)
- Data Science competitions (https://www.drivendata.org)
- UC Irvine Machine Learning Repository (https://archive.ics.uci.edu/ml/)
- Registry of Open Data on AWS (https://registry.opendata.aws)
- A Comprehensive List of Open Data Portals from Around the World (http://dataportals.org)
- Financial and economic datasets (https://www.quandl.com)
- Datasets subreddit (https://www.reddit.com/r/datasets/)

Finally, your own data and projects
Data Science: Principles and Practice

01  Linear Regression
02  Optimization with Gradient Descent
03  Multiple Linear Regression and Polynomial Features
04  Overfitting
05  The First Practical
Linear regression

- **Linear regression** helps modelling how changes in one or more input variables (independent variables) affect the output (dependent variable)

- **Widely used algorithm** in machine learning and data science. **Application areas**: healthcare, social sciences, economics, environmental science, prediction of planetary movements

- Linear regression is an example of **supervised learning algorithms**

Supervised Learning

Dataset: \( \{ < x_1, y_1 >, < x_2, y_2 >, < x_3, y_3 >, \ldots, < x_n, y_n > \} \)

Input instances: \( x_1, x_2, x_3, x_4, \ldots, x_n \)

Known (desired) outputs: \( y_1, y_2, y_3, y_4, \ldots, y_n \)

Our goal: Learn the mapping \( f : X \rightarrow Y \)

such that \( y_i = f(x_i) \) for all \( i = 1, 2, 3, \ldots, n \)
Continuous vs Discrete Problems

**Regression**: the desired labels are continuous

- Company earnings, revenue $\rightarrow$ company stock price
- House size and age $\rightarrow$ price

**Classification**: the desired labels are discrete

- Handwritten digits $\rightarrow$ digit label
- User tweets $\rightarrow$ detect positive/negative sentiment

**Regression or classification?**

- Model the salary of baseball players based on their game statistics
Continuous vs Discrete Problems

**Regression**: the desired labels are continuous

- Company earnings, revenue $\rightarrow$ company stock price
- House size and age $\rightarrow$ price

**Classification**: the desired labels are discrete

- Handwritten digits $\rightarrow$ digit label
- User tweets $\rightarrow$ detect positive/negative sentiment

**Regression or classification?**

- Model the salary of baseball players based on their game statistics $\rightarrow$ regression
- Find what object is on a photo
Continuous vs Discrete Problems

**Regression**: the desired labels are continuous

- Company earnings, revenue → company stock price
- House size and age → price

**Classification**: the desired labels are discrete

- Handwritten digits → digit label
- User tweets → detect positive/negative sentiment

**Regression or classification?**

- Model the salary of baseball players based on their game statistics → **regression**
- Identify what object is on a photo → **classification**
- Predict election results
Continuous vs Discrete Problems

**Regression**: the desired labels are continuous

- Company earnings, revenue → company stock price
- House size and age → price

**Classification**: the desired labels are discrete

- Handwritten digits → digit label
- User tweets → detect positive/negative sentiment

Regression or classification?

- Model the salary of baseball players based on their game statistics → **regression**
- Identify what object is on a photo → **classification**
- Predict election results → **regression** (%) / **classification** (winner)
Simplest Possible Linear Model

What is the simplest possible model for $f : X \rightarrow Y$?

$$y = x$$
Simplest Possible Linear Model

What is the simplest possible model for \( f : X \rightarrow Y \)?

\[ y = x \]
Simplest Possible Linear Model

What is the simplest possible model for $f : X \rightarrow Y$?

$$y = x$$

**Estimated Corruption vs Government Effectiveness**

**Estimated Corruption vs Urban Population**
(Still Too Simple) Linear Models

\[ y = ax \]

\[ y = b \]
Linear Regression

A better linear model:

\[ y = ax + b \]
Linear Regression

A better linear model:  \[ y = ax + b \]
Linear Regression

\[ x : \text{GDP per Capita} \]
\[ y : \text{Enrolment Rate} \]
\[ \hat{y} = ax + b \]

How do we find the best values for \( a \) and \( b \)?
Loss Function

First, let’s define what “best” actually means for us.

\[ E = \frac{1}{2} \sum_{i=1}^{M} (\hat{y}_i - y_i)^2 \]
Loss Function

First, let’s define what “best” actually means for us.

\[ E = \frac{1}{2} \sum_{i=1}^{M} (\hat{y}_i - y_i)^2 \]

\[ E = \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2 \]
Loss Function

First, let’s define what “best” actually means for us.

\[ E = \frac{1}{2} \sum_{i=1}^{M} (\hat{y}_i - y_i)^2 \]

\[ E = \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2 \]

- Smaller value of \( E \) means our predictions are close to the real values
- Individual large errors incur a large exponential penalty
- Many small errors are acceptable and get a very small loss
- Easily differentiable function
Loss Function

First, let’s define what “best” actually means for us.

\[ E = \frac{1}{2} \sum_{i=1}^{M} (\hat{y}_i - y_i)^2 \]

\[ E = \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2 \quad RMSE = \sqrt{\frac{\sum_{i=1}^{M} (\hat{y}_i - y_i)^2}{M}} \]

- Smaller value of E means our predictions are close to the real values
- Individual large errors incur a large exponential penalty
- Many small errors are acceptable and get a very small loss
- Easily differentiable function
Gradient Descent

We can update \(a\) and \(b\) using the training data and the loss function.

The partial derivative of a function shows the direction of the slope.

\[
\frac{\partial E}{\partial a} = \frac{\partial}{\partial a} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2
\]
Gradient Descent

We can update $a$ and $b$ using the training data and the loss function.

The partial derivative of a function shows the direction of the slope.

\[
\frac{\partial E}{\partial a} = \frac{\partial}{\partial a} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2 \\
= \frac{1}{2} \sum_{i=1}^{M} \frac{\partial}{\partial a} (ax_i + b - y_i)^2
\]
Gradient Descent

We can update \( a \) and \( b \) using the training data and the loss function.

The partial derivative of a function shows the direction of the slope.

\[
\frac{\partial E}{\partial a} = \frac{\partial}{\partial a} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2
\]

\[
= \frac{1}{2} \sum_{i=1}^{M} \frac{\partial}{\partial a} (ax_i + b - y_i)^2
\]

\[
= \sum_{i=1}^{M} (ax_i + b - y_i)x_i = \sum_{i=1}^{M} (\hat{y}_i - y_i)x_i
\]
Gradient Descent

We can update $a$ and $b$ using the training data and the loss function.

The partial derivative of a function shows the direction of the slope.

$$\frac{\partial E}{\partial a} = \frac{\partial}{\partial a} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2$$

$$= \frac{1}{2} \sum_{i=1}^{M} \frac{\partial}{\partial a} (ax_i + b - y_i)^2$$

$$= \sum_{i=1}^{M} (ax_i + b - y_i)x_i = \sum_{i=1}^{M} (\hat{y}_i - y_i)x_i$$

$$\frac{\partial E}{\partial b} = \frac{\partial}{\partial b} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2$$
Gradient Descent

We can update $a$ and $b$ using the training data and the loss function.

The partial derivative of a function shows the direction of the slope.

\[
\frac{\partial E}{\partial a} = \frac{\partial}{\partial a} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2 = \frac{1}{2} \sum_{i=1}^{M} \frac{\partial}{\partial a} (ax_i + b - y_i)^2 = \sum_{i=1}^{M} (ax_i + b - y_i)x_i = \sum_{i=1}^{M} (\hat{y}_i - y_i)x_i
\]

\[
\frac{\partial E}{\partial b} = \frac{\partial}{\partial b} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2 = \sum_{i=1}^{M} (ax_i + b - y_i)
\]
Gradient Descent

We can update $a$ and $b$ using the training data and the loss function.

The partial derivative of a function shows the direction of the slope.

\[
\frac{\partial E}{\partial a} = \frac{\partial}{\partial a} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2 \\
= \frac{1}{2} \sum_{i=1}^{M} \frac{\partial}{\partial a} (ax_i + b - y_i)^2 \\
= \sum_{i=1}^{M} (ax_i + b - y_i)x_i = \sum_{i=1}^{M} (\hat{y}_i - y_i)x_i
\]

\[
\frac{\partial E}{\partial b} = \frac{\partial}{\partial b} \frac{1}{2} \sum_{i=1}^{M} (ax_i + b - y_i)^2 \\
= \sum_{i=1}^{M} (ax_i + b - y_i) \\
= \sum_{i=1}^{M} (\hat{y}_i - y_i)
\]
Gradient Descent

**Gradient descent**: Repeatedly update parameters $a$ and $b$ by taking small steps in the direction of the partial derivative.

\[
a := a - \alpha \frac{\partial E}{\partial a}
\]

\[
b := b - \alpha \frac{\partial E}{\partial b}
\]

$\alpha$ : learning rate / step size
Gradient Descent

**Gradient descent**: Repeatedly update parameters $a$ and $b$ by taking small steps in the direction of the partial derivative.

\[
a := a - \alpha \frac{\partial E}{\partial a} \quad \quad b := b - \alpha \frac{\partial E}{\partial b}
\]

$\alpha$ : learning rate / step size

\[
a := a - \alpha \sum_{i=1}^{M} (ax_i + b - y_i)x_i
\]

\[
b := b - \alpha \sum_{i=1}^{M} (ax_i + b - y_i)
\]
Gradient Descent

**Gradient descent**: Repeatedly update parameters \( a \) and \( b \) by taking small steps in the direction of the partial derivative.

\[
a := a - \alpha \frac{\partial E}{\partial a} \quad b := b - \alpha \frac{\partial E}{\partial b}
\]

\( \alpha \) : learning rate / step size

\[
a := a - \alpha \sum_{i=1}^{M} (ax_i + b - y_i)x_i \\
b := b - \alpha \sum_{i=1}^{M} (ax_i + b - y_i)
\]

This same algorithm drives nearly all of the modern neural network models.
Gradient Descent

Implementing gradient descent by hand:

```python
In [8]:
X = data["GDP per Capita (PPP USD)"].values
Y = data["Enrolment Rate, Tertiary (%)"].values
a = 0.0
b = 0.0
learning_rate = 1e-11

for epoch in range(10):
    update_a = 0.0
    update_b = 0.0
    error = 0.0
    for i in range(len(Y)):
        y_predicted = a * X[i] + b
        update_a += (y_predicted - Y[i]) * X[i]
        update_b += (y_predicted - Y[i])
        error += np.square(y_predicted - Y[i])
    a = a - learning_rate * update_a
    b = b - learning_rate * update_b
    rmse = np.sqrt(error / len(Y))
    print("RMSE: " + str(rmse))
plot_simple_linear_regression(X, Y, a, b)
```

RMSE: 43.60705215347086
RMSE: 27.764974739091667
RMSE: 27.121980238646962
RMSE: 27.101664900304836
RMSE: 27.101030544822766
RMSE: 27.101010112785858
RMSE: 27.101010079074783
RMSE: 27.10101007214674

See lecture2.ipynb at https://github.com/ekochmar/cl-datasci-pnp-2021
Gradient Descent

Implementing gradient descent by hand:

```python
X = data["GDP per Capita (PPP USD)"].values
Y = data["Enrolment Rate, Tertiary (%)"].values
a = 0.0
b = 0.0
learning_rate = 1e-11

for epoch in range(10):
    update_a = 0.0
    update_b = 0.0
    error = 0.0
    for i in range(len(Y)):
        y_predicted = a * X[i] + b
        update_a += (y_predicted - Y[i]) * X[i]
        update_b += (y_predicted - Y[i])
        error += np.square(y_predicted - Y[i])
    a = a - learning_rate * update_a
    b = b - learning_rate * update_b
    rmse = np.sqrt(error / len(Y))
print("RMSE: " + str(rmse))

plot_simple_linear_regression(X, Y, a, b)
```
Gradient Descent

A more compact version, operating over all the datapoints at once:

```
In [9]:
X = data["GDP per Capita (PPP USD)"].values
Y = data["Enrolment Rate, Tertiary (%)"].values
a = 0.0
b = 0.0
learning_rate = 1e-11

for epoch in range(10):
    y_predicted = a * X + b
    a = a - learning_rate * ((y_predicted - Y)*X).sum()
    b = b - learning_rate * (y_predicted - Y).sum()
    rmse = np.sqrt(np.square(y_predicted - Y).mean())
    print("RMSE: " + str(rmse))
plot_simple_linear_regression(X, Y, a, b)
```
The Gradient

It may be more convenient to work with vector notation.

The gradient is a vector of all partial derivatives.

For a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the gradient is

$$\nabla_{\theta} f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \frac{\partial f(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_n} \end{bmatrix}$$
Gradient Descent Pitfalls

- Starting with a “bad” random initialization point may cause the algorithm being stuck in a local (rather than global) minimum, or stop too early on a plateau.
- Luckily, cost function for Linear Regression is convex.
The Analytical Solution

Solving the single-variable linear regression with the analytical solution (normal equation)

\[ X = \begin{bmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_M & 1.0 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix}, \quad \theta = \begin{bmatrix} a \\ b \end{bmatrix} \]

\[ \nabla_\theta E(\theta) = X^T (X \theta - y) = 0 \]

\[ \implies \theta^* = (X^T X)^{-1} X^T y \]
The Analytical Solution

Solving the single-variable linear regression with the analytical solution (normal equation)

\[
X = \begin{bmatrix}
    x_1 & 1.0 \\
    x_2 & 1.0 \\
    \vdots & \vdots \\
    x_M & 1.0 \\
\end{bmatrix} \quad y = \begin{bmatrix}
    y_1 \\
    y_2 \\
    \vdots \\
    y_M \\
\end{bmatrix} \quad \theta = \begin{bmatrix}
    a \\
    b \\
\end{bmatrix}
\]

\[
\nabla_\theta E(\theta) = X^T(X\theta - y) = 0
\]

\[
\implies \theta^* = (X^TX)^{-1}X^Ty
\]

Great for directly finding the optimal parameter values. Not so great for large problems: matrix inversion has cubic complexity \(O(n^3)\).
Analytical Solution with Scikit-Learn

```python
from sklearn.linear_model import LinearRegression

model = LinearRegression(fit_intercept=True)
X = data["GDP per Capita (PPP USD)"].values.reshape(-1,1)
Y = data["Enrolment Rate, Tertiary (%)"]
model.fit(X, Y)

mse = np.square(Y - model.predict(X)).mean()
print("RMSE: " + str(np.sqrt(mse)))
```

RMSE: 22.630490998345973
Multiple Linear Regression

We normally use more than 1 input feature in our model

\[ y^{(i)} = \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \theta_3 x_3^{(i)} + \cdots + \theta_N x_N^{(i)} + \theta_{N+1} \]
Multiple Linear Regression

```python
model = LinearRegression(fit_intercept=True)
X = data.copy().drop(["Country Name", "Enrolment Rate, Tertiary (%)", axis=1)
Y = data["Enrolment Rate, Tertiary (%)"]
model.fit(X, Y)
mse = np.square(Y - model.predict(X)).mean()
print("RMSE: " + str(np.sqrt(mse)))
```

RMSE: 14.40196
Exploring the Parameters

- `model.coef_` now contains optimized coefficients for each of the input features.
- `model.intercept_` contains the intercept.

```python
headers = list(X)
coefficients = []
for i in range(len(headers):
    coefficients.append({"Property": headers[i],
                         "coefficient": model.coef_[i]})
pd.DataFrame(coefficients)
```

<table>
<thead>
<tr>
<th>Property</th>
<th>coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDP per Capita (PPP USD)</td>
<td>0.000236</td>
</tr>
<tr>
<td>Population Density (persons per sq km)</td>
<td>-0.012085</td>
</tr>
<tr>
<td>Population Growth Rate (%)</td>
<td>-12.605788</td>
</tr>
<tr>
<td>Urban Population (%)</td>
<td>0.361150</td>
</tr>
<tr>
<td>Life Expectancy at Birth (avg years)</td>
<td>0.584344</td>
</tr>
<tr>
<td>Fertility Rate (births per woman)</td>
<td>5.795337</td>
</tr>
<tr>
<td>Infant Mortality (deaths per 1000 births)</td>
<td>-0.092305</td>
</tr>
<tr>
<td>Unemployment, Total (%)</td>
<td>-0.312737</td>
</tr>
<tr>
<td>Estimated Control of Corruption (scale -2.5 to...)</td>
<td>-5.153427</td>
</tr>
<tr>
<td>Estimated Government Effectiveness (scale -2.5...)</td>
<td>4.035069</td>
</tr>
<tr>
<td>Internet Users (%)</td>
<td>0.149982</td>
</tr>
</tbody>
</table>
Exploring the Parameters

The coefficients are only comparable if we standardize the input features first.

```
Z = pd.DataFrame(data, columns=['GDP per Capita (PPP USD)'])
Z_scaled = preprocessing.scale(Z)
```

<table>
<thead>
<tr>
<th>Property</th>
<th>coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDP per Capita (PPP USD)</td>
<td>3.865747</td>
</tr>
<tr>
<td>Population Density (persons per sq km)</td>
<td>-2.748875</td>
</tr>
<tr>
<td>Population Growth Rate (%)</td>
<td>-14.487085</td>
</tr>
<tr>
<td>Urban Population (%)</td>
<td>8.359783</td>
</tr>
<tr>
<td>Life Expectancy at Birth (avg years)</td>
<td>5.126343</td>
</tr>
<tr>
<td>Fertility Rate (births per woman)</td>
<td>8.122616</td>
</tr>
<tr>
<td>Infant Mortality (deaths per 1000 births)</td>
<td>-2.126688</td>
</tr>
<tr>
<td>Unemployment, Total (%)</td>
<td>-2.385280</td>
</tr>
<tr>
<td>Estimated Control of Corruption (scale -2.5 to -1)</td>
<td>-5.023631</td>
</tr>
<tr>
<td>Estimated Government Effectiveness (scale -2.5 to -0.5)</td>
<td>3.714866</td>
</tr>
<tr>
<td>Internet Users (%)</td>
<td>4.329112</td>
</tr>
</tbody>
</table>
Polynomial Features

Polynomial combinations of the features.

With degree 2, features \([z_1, z_2]\)

would become \([1, z_1, z_2, z_1^2, z_1 z_2, z_2^2]\)

```python
from sklearn.preprocessing import PolynomialFeatures
model = LinearRegression(fit_intercept=True)
X = data.copy().drop(["Country Name", "Enrolment Rate, Tertiary (%)"], axis=1)
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)
Y = data["Enrolment Rate, Tertiary (%)"]
model.fit(X_poly, Y)
```

RMSE: 13.6692
Polynomial Features

With 3rd degree polynomial features, the linear regression model now has 364 input features:

\[
\frac{(n + d)!}{n!d!} = \frac{(11 + 3)!}{11!3!} = 364
\]

```python
model = LinearRegression(fit_intercept=True)
X = data.copy().drop(['Country Name', 'Enrolment Rate, Tertiary (%)'], axis=1)
poly = PolynomialFeatures(degree=3)
X_poly = poly.fit_transform(X)
Y = data['Enrolment Rate, Tertiary (%)']
model.fit(X_poly, Y)

mse = np.square(Y - model.predict(X_poly)).mean()
print("RMSE: " + str(np.sqrt(mse)))
```

RMSE: 0.00018
Overfitting

There are twice as many features/parameters as there are datapoints in the whole dataset.

This can easily lead to overfitting:
Dataset Splits

- **Training Set**
  - For training your models, fitting the parameters

- **Development Set**
  - For continuous evaluation and hyperparameter selection

- **Test Set**
  - For realistic evaluation once the training and tuning is done
Stratified Sampling

Making sure the proportion of classes is kept the same in the splits

- **Training Set**: For training your models, fitting the parameters
- **Development Set**: For continuous evaluation and hyperparameter selection
- **Test Set**: For realistic evaluation once the training and tuning is done
Overfitting

**Training set**
3rd degree polynomial features

**Development / Validation set**
3rd degree polynomial features

RMSE: 1.1422e-07

RMSE: 133.4137
Overfitting

**Development set**
2nd degree polynomial features

GDP vs Enrollment Rate

RMSE: 68.4123

**Development set**
1st degree polynomial features

GDP vs Enrollment Rate

RMSE: 16.1414
Overfitting

**Training set**
- 1 input feature (GDP)
- 3rd degree polynomial features

**Development set**
- 1 input feature (GDP)
- 3rd degree polynomial features

RMSE: 19.8130  
RMSE: 15.9834
How to Spot Overfitting

Learning curves with 1st degree polynomial features

Learning curves with 10th degree polynomial features

https://github.com/ageron/handson-ml/
Ways to Prevent Overfitting

Regularize (constrain) the model, so that it has fewer degrees of freedom. E.g. reduce the number of polynomial degrees, or *constrain the weights by adding a regularization term to the cost function*:

- **Ridge Regression** cost function:  
  \[ J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2 \]
  - i.e., adding \( l_2 \)-norm of the weight vector as the regularization term
  - *alpha* controls the amount of regularization: \( \alpha=0 \rightarrow \) Linear Regression

- **Lasso Regression** cost function:  
  \[ J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i| \]
  - i.e., \( l_1 \)-norm

- **Elastic Net** cost function:  
  \[ J(\theta) = MSE(\theta) + r \alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^{n} \theta_i^2 \]
  - i.e. a mix of Ridge and Lasso controlled by ratio \( r \)
Practical 1
Data

- **California House Prices Dataset** containing information on a number of independent variables about the block groups in California from 1990 Census

- **Dependent variable:** house price

<table>
<thead>
<tr>
<th>longitude</th>
<th>latitude</th>
<th>housing_median_age</th>
<th>total_rooms</th>
<th>total_bedrooms</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>20640.00000</td>
<td>20640.00000</td>
<td>20640.00000</td>
<td>20640.00000</td>
</tr>
<tr>
<td>mean</td>
<td>-119.569704</td>
<td>35.631861</td>
<td>28.639486</td>
<td>2635.763081</td>
</tr>
<tr>
<td>std</td>
<td>2.003532</td>
<td>2.135952</td>
<td>12.585558</td>
<td>2181.615252</td>
</tr>
<tr>
<td>min</td>
<td>-124.350000</td>
<td>32.540000</td>
<td>1.000000</td>
<td>2.000000</td>
</tr>
<tr>
<td>25%</td>
<td>-121.800000</td>
<td>33.930000</td>
<td>18.000000</td>
<td>1447.750000</td>
</tr>
<tr>
<td>50%</td>
<td>-118.490000</td>
<td>34.260000</td>
<td>29.000000</td>
<td>2127.000000</td>
</tr>
<tr>
<td>75%</td>
<td>-118.010000</td>
<td>37.710000</td>
<td>37.000000</td>
<td>3148.000000</td>
</tr>
<tr>
<td>max</td>
<td>-114.310000</td>
<td>41.950000</td>
<td>52.000000</td>
<td>39320.000000</td>
</tr>
</tbody>
</table>
Your task: Learning objectives

- Load the dataset
- Understand the data, the attributes and their correlations
- Split the data into training and test sets
- Apply normalisation, scaling and other transformations to the attributes if needed
- Build a machine learning model
- Evaluate the model and investigate the errors
- Tune your model to improve performance
Practical 1 Logistics

- Data and code for Practical 1 can be found on: Github (https://github.com/ekochmar/cl-datasci-pnp-2021/tree/master/DSPNP_practical1)

- Practical session is on Tuesday 10 November, 3-4pm, over Zoom

- At the practical, be prepared to discuss the task and answer the questions about the code to get a ‘pass’

- After the practical, upload your solutions (Jupyter notebook or Python code) to Moodle