Ordinary Least Squares Regression

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]:
%load_ext autoreload
%autoreload 2
%matplotlib inline

In [2]:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

In [3]:
import matplotlib.gridspec as gridspec
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import add_dummy_feature
import numpy.linalg as npla

Acknowledgement

- I based 5 of the diagrams on ones to be found in A. Géron: *Hands-On Machine Learning with Scikit-Learn & TensorFlow*, O'Reilly, 2017
Ordinary Least Squares Regression

- We’ve been focusing on topics such as Data Preparation and Error Estimation, treating the learning itself as a black box.
  - How does linear regression find the best model?

- Recap:
  - The learner is given a labeled training set \((m \times n)\) matrix \(X\) and \(m\)-dimensional vector \(y\) and it inserts an extra element \(x_0^{(i)}\) in each row \(i\), all of which will be 1.
  - An \(n\)-dimensional column vector \(\beta = \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_n \end{bmatrix}\) gives us a hypothesis \(h_\beta\) where \(\hat{y} = h_\beta(x) = x\beta\)
  - It must find the vector \(\beta\) that minimizes the following loss function
    \[
    J(X, y, \beta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2
    \]
  - In Machine Learning, the variables whose values we are trying to find (in this case, \(\beta\)) are called the parameters.

In [4]: # Loss function for OLS regression (assumes X contains all 1s in its first column)
```
def J(X, y, beta):
    return np.mean((X.dot(beta) - y)**2) / 2.0
```

2D Visualization of \(J\)

- Let’s visualize \(J\) using the Cork Property Prices Dataset as the training set.
- For a 2D visualization, we’ll assume that \(\beta_j = 0\) for all \(j\) except \(j = 1\). In other words, we are pretending that floor area is the only relevant feature.
  - Then we can plot \(J\) on the vertical axis against different values of \(\beta_1\) on the horizontal axis.
In [5]:

# Use pandas to read the CSV file
df = pd.read_csv("datasets/dataset_corkA.csv")

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df["flarea"].values
y = df["price"].values

# Set up the two subplots
fig = plt.figure(figsize=(8, 4))
gs = gridspec.GridSpec(1, 2, width_ratios=[2, 1])
# Lefthand diagram
ax0 = plt.subplot(gs[0])
plt.title("Training set and hypothesis")
plt.xlabel("Floor area (sq metres")
plt.xlim(-500, 500)
plt.ylabel("Price (000 euros")
plt.ylim(0, 1000)
# Righthand diagram
ax1 = plt.subplot(gs[1])
plt.title("Loss function")
plt.xlabel("$beta1$")
plt.xlim(-10, 10)
plt.ylabel("$J$")
plt.ylim(0, 1500000)
fig.tight_layout()

# Hypothesis (which assumes beta_i = 0 for all i except i = 1)
def h(x, beta1):
    return x * beta1

# Here's the statement you should vary: try different values of beta1
# To see anything on these axes, try beta1 between -10 and +10
beta1 = 8

# Scatter plot of the training set (lefthand diagram)
ax0.scatter(X, y, color = "green")

# Straight line plot of the hypothesis for this value of beta1 (lefthand diagram)
xvals = np.linspace(-500, 500, 3)
ax0.plot(xvals, h(xvals, beta1), color = "blue")

# Show the value of the loss function for this value of beta1 (righthand diagram)
ax1.scatter(beta1, J(X, y, beta1), color = "red")
plt.show()
Another 2D Visualization of $J$

- Instead of making manual adjustments, let's use a loop to try several values for $\beta_1$

```python
In [6]:
# Set up the two subplots
fig = plt.figure(figsize=(8, 4))
gs = gridspec.GridSpec(1, 2, width_ratios=[2, 1])

# Lefthand diagram
ax0 = plt.subplot(gs[0])
plt.title("Training set and hypothesis")
plt.xlabel("Floor area (sq metres)")
plt.xlim(-500, 500)
plt.ylabel("Price (000 euros)")
plt.ylim(0, 1000)

# Righthand diagram
ax1 = plt.subplot(gs[1])
plt.title("Loss function")
plt.xlabel("$\beta_1$")
plt.xlim(-10, 10)
plt.ylabel("$J$")
plt.ylim(0, 1500000)
fig.tight_layout()

# Different values of beta1
betas = np.linspace(-10, 10, 21)

# Scatter plot of the training set (lefthand diagram)
ax0.scatter(X, y, color = "green")

# Straight line plot of the hypothesis for this value of beta1 (lefthand diagram)
xvals = np.linspace(-500, 500, 3)
for beta1 in betas:
    ax0.plot(xvals, h(xvals, beta1), color = "blue")

# Show the value of the loss function for this value of beta1 (righthand diagram)
ax1.scatter(betas, [J(X, y, beta1) for beta1 in betas], color = "red")
plt.show()
```
The loss function is **convex**

Informally, this means:
- In 2D, it is u-shaped
- It has a unique minimum

This is no accident: it follows from the way the loss function has been defined.

### 3D Visualization of $J$

- Let's visualize $J$ again using the Cork Property Prices Dataset as the training set.
- This time, we'll assume that $\beta_j = 0$ for all $j$ except $j = 2$ and $j = 3$.
  - In other words, we are pretending that the number of bedrooms and bathrooms are the only relevant features.
  
This will be a 3D plot with $J$ on the vertical axis against different values of $\beta_2$ and $\beta_3$ on the horizontal axes.

```python
In [7]: # Get the feature-values and the target values into separate numpy arrays of numbers
   X = df["bdrms", "bthrms"].values
   y = df["price"].values
   fig = plt.figure()
   ax = Axes3D(fig)
   ax.set_title("Loss function")
   ax.set_xlabel("$\beta_2$")
   ax.set_ylabel("$\beta_3$")
   ax.set_zlabel("$J$")
   xvals = np.linspace(-100, 200, 301)
   yvals = np.linspace(-100, 200, 301)
   xxvals, yyvals = np.meshgrid(xvals, yvals)
   zs = np.array([J(X, y, [beta2, beta3]) for beta2, beta3 in zip(xxvals.flatten(), yyvals.flatten())])
   zvals = zs.reshape(xxvals.shape)
   ax.plot_surface(xxvals, yyvals, zvals)
   plt.show()
```

![3D Visualization of Loss Function](image-url)
Another Visualization of $J$

- Here is the same data that we had in 3D but on a different kind of plot, called a **contour plot**
- In effect, it flattens the diagram above
- The lines connect points that have the same value for $J$

```python
In [9]: fig = plt.figure(figsize = (6, 6))
plt.title("Loss function")
plt.xlabel("$\beta_2$")
plt.ylabel("$\beta_3$")
xvals = np.linspace(-100, 200, 301)
yvals = np.linspace(-100, 200, 301)
xxvals, yyvals = np.meshgrid(xvals, yvals)
zs = np.array([J(X, y, [beta2, beta3]) for beta2, beta3 in zip(xxvals.flatten(), yyvals.flatten())])
zvals = zs.reshape(xxvals.shape)
C = plt.contour(xxvals, yyvals, zvals, 15, colors = "black")
plt.clabel(C, inline=1, fontsize=10)
plt.show()
```

- The 3D visualization and the contour plot show that this too is convex
- Informally, this means:
  - In 3D it is bowl-shaped
  - again there is a unique minimum

### Ordinary Least Squares Linear Regression

- How to find the best values for $\beta$
- Two methods:
  - The Normal Equation
  - Gradient Descent
The Normal Equation

- The normal equation solves for $\beta$:
  $$\beta = (X^T X)^{-1} X^T y$$
- In other words, the normal equation gives us the parameters that minimize the loss function
- Where does it come from?
  - Take the gradient of the loss function: $\frac{1}{m} X^T (X\beta - y)$ (see next slide)
  - Set it to zero: $\frac{1}{m} X^T (X\beta - y) = 0$ (in fact, a $(n + 1)$-dimensional vector of zeros)
  - Then do some algebraic manipulation to get $\beta$ on the left-hand side: $\beta = (X^T X)^{-1} X^T y$

Partial Derivatives

- We need the gradient of the loss function with regards to each $\beta_j$
  - In other words, how much the loss will change if we change $\beta_j$ a little
  - With respect to a particular $\beta_j$, it is called the partial derivative
- Without doing the calculus, the partial derivatives of $J(X, y, \beta)$ with respect to $\beta_j$ are
  $$\frac{\partial J(X, y, \beta)}{\partial \beta_j} = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} \beta_j - y^{(i)}) x_j^{(i)}$$
- The gradient vector, $\nabla_{\beta} J(X, y, \beta)$, is a vector of each partial derivative:
  $$\nabla_{\beta} J(X, y, \beta) = \left[ \begin{array}{c} \frac{\partial J(X, y, \beta)}{\partial \beta_0} \\ \frac{\partial J(X, y, \beta)}{\partial \beta_1} \\ \vdots \\ \frac{\partial J(X, y, \beta)}{\partial \beta_n} \end{array} \right] = \left[ \begin{array}{c} \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} \beta_0 - y^{(i)}) x_0^{(i)} \\ \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} \beta_1 - y^{(i)}) x_1^{(i)} \\ \vdots \\ \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} \beta_n - y^{(i)}) x_n^{(i)} \end{array} \right]$$
- And there is a vectorized way to compute it:
  $$\nabla_{\beta} J(X, y, \beta) = \frac{1}{m} X^T (X\beta - y)$$

The Normal Equation in scikit-learn

- The fit method of scikit-learn's LinearRegression class does what we have described:
  - It inserts the extra column of 1s
  - It calculates $\beta$ using the normal equation

The Normal Equation in numpy

- For the hell of it, let's see how to implement it ourselves
  - (We'll be naughty: we'll train on the whole dataset)
In [10]:
# Use pandas to read the CSV file
df = pd.read_csv("datasets/dataset_corkA.csv")

# Get the feature-values and the target values
X_without_dummy = df["flarea", "bdrms", "bthrms"].values
y = df["price"].values

# Add the extra column to X
X = add_dummy_feature(X_without_dummy)

# Solve the normal equation
beta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)

# Display beta
beta

Out[10]: array([38.17496491, 2.49923248, -0.32659509, 0.78080337])

• Essentially, one line of code!
• But there's a problem:
  ■ The normal equation requires that $X^T X$ has an inverse
  ■ But it might not (see earlier lecture about matrices)
When we discussed this before, we mentioned that, in some cases, we can use the pseudo-inverse instead
  ■ This is one of those cases
• So the more robust way of writing this program is:

In [11]:
# Solve the normal equation - but using the pseudo-inverse
beta = np.linalg.pinv(X.T.dot(X)).dot(X.T).dot(y)

# Display beta
beta

Out[11]: array([38.17496491, 2.49923248, -0.32659509, 0.78080337])

**Gradient Descent**

• **Gradient Descent** is a generic method for finding optimal solutions to problems that involve minimizing a loss function
• It is a search in the model's parameter space for values of the parameters that minimize the loss function
• Conceptually:
  ■ It starts with an initial guess for the values of the parameters
  ■ Then repeatedly:
    ○ It updates the parameter values — hopefully to reduce the loss

• Ideally, it keeps doing this until convergence — changes to the parameter values do not result in lower loss
• The key to this algorithm is how to update the parameter values
Baby steps

- We'll use an example with a single feature/single parameter $\beta_1$ in order to visualize
- We update $\beta_1$ gradually, one baby step at a time, until the algorithm converges on minimum loss.

The size of the steps is determined by the **learning rate**

- If the learning rate is too small, it will take many updates until convergence:

- If the learning rate is too big, the algorithm might jump across the valley — it may even end up with higher loss than before, making the next step bigger.
  - This might make the algorithm **diverge**
Non-Convex Functions

- The loss function for OLS regression is convex
  - a single minimum
  - and it has a slope that never changes abruptly
  - The algorithm is guaranteed to approach arbitrarily close to the minimum (if it runs long enough and if the learning rate isn’t too high)
- But not all loss functions are convex, which can cause problems for Gradient Descent

Scaling

- When features have very different ranges of values, we may need to scale:
  - Whenever we use Euclidean distance, e.g. in a clustering algorithm
  - For PCA (although scikit-learn’s PCA class will do it for us)
- For Gradient Descent, we should also scale the features
- We do not need to scale if we are doing OLS regression using the Normal Equation
  - It is not ‘wrong-footed’ by features of different scales
Why we need to scale for Gradient Descent

- If features have different ranges, it affects the shape of the 'bowl'
- E.g. features 1 and 2 have similar ranges of values — a 'bowl'

- The algorithm goes straight towards the minimum
- E.g. feature 1 has a smaller values than feature 2 — an elongated 'bowl'

- Since feature 1 has smaller values, it takes a larger change in $\beta_1$ to affect the loss function, which is why it is elongated
- It takes more steps to get to the minimum — steeply down but not really towards the goal, followed by a long march down a nearly flat valley

In [ ]: