CS4619: Artificial Intelligence 2

OLS Linear Regression using Gradient Descent

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

OLS Regression using Gradient Descent

A reminder: we have a training set containing \( m \) examples, each having \( n \) features. We have represented the training set as an \( m \times (n + 1) \) matrix \( X \): each row is one of the examples; each column is one of the features; but all the values in the first column (which we designate column 0) are set to 1. Each hypothesis is a linear equation, \( h_\beta(x) = \beta_0 x_0 + \beta_1 x_1 + \ldots + \beta_n x_n \). We are trying to find a \( (n + 1) \)-dimensional vector \( \beta \) of parameters that minimise the loss function \( J(\beta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2 \).

In the previous lecture, we saw that one approach is to set the gradient of the loss function to zero and to solve for \( \beta \), giving what is called the normal equation. In this lecture, we take a different approach: we search for the \( \beta \) that minimises the loss function.

Conceptually, this approach works as follows:

- It starts with an initial guess — e.g. choose each \( \beta_j \) randomly or set each to zero
- It computes \( J(\beta) \) — the total error for this hypothesis on the training set
- And it repeats: it chooses another \( \beta \) — one for which \( J(\beta) \) is smaller
- It keeps doing this until \( J(\beta) \) converges — changes to \( \beta \) do not result in smaller \( J(\beta) \)

So the key to this algorithm is how it comes up with new parameter values, \( \beta \).

Gradient Descent
Gradient descent is a general method for finding parameters $\beta$ that minimize some function $J_\beta$. The idea is to repeatedly make small changes to the values of the parameters that lead to the greatest immediate decrease in the value of $J(\beta)$. The algorithm is as follows:

repeat
  - simultaneously update all $\beta_j$ as follows:
    \[ \beta_j \leftarrow \beta_j - \alpha \frac{\partial J(\beta)}{\partial \beta_j} \]
until convergence

$\alpha$ is called the learning rate and it controls the size of the changes that we make.

The Importance of Simultaneous Update

It's important that the updates to the $\beta_j$ are done simultaneously, not one after the other. We will see how to achieve this using vectorization. The alternative, if you want to stick with a more conventional programming style, is to use the following statements:

- $\delta_0 \leftarrow \frac{\partial J(\beta)}{\partial \beta_0}$
- $\delta_1 \leftarrow \frac{\partial J(\beta)}{\partial \beta_1}$
- \[ \vdots \]
- $\delta_n \leftarrow \frac{\partial J(\beta)}{\partial \beta_n}$

- $\beta_0 \leftarrow \beta_0 - \alpha \delta_0$
- $\beta_1 \leftarrow \beta_1 - \alpha \delta_1$
- \[ \vdots \]
- $\beta_n \leftarrow \beta_n - \alpha \delta_n$

A student reads the above, takes no notice, and simply implements the algorithm with the following sequence of statements:

- $\beta_0 \leftarrow \beta_0 - \alpha \frac{\partial J(\beta)}{\partial \beta_0}$
- $\beta_1 \leftarrow \beta_1 - \alpha \frac{\partial J(\beta)}{\partial \beta_1}$
- \[ \vdots \]
- $\beta_n \leftarrow \beta_n - \alpha \frac{\partial J(\beta)}{\partial \beta_n}$

What's the difference?

Further Questions
Some further questions that we will discuss in the lecture:

- Why do these update rules, using derivatives, make intuitive sense. (We will explain this with a diagram in which we plot $J_{\beta}$ on the vertical axis and just one of the parameters on the horizontal axis. We will consider separately the case where the gradient is positive and where it is negative and we will see how the update rules move us towards a minimum.)
- In practice, how will we define convergence?
- What is the role of the learning rate, $\alpha$?
  - What if the value of $\alpha$ is 'too small'?
  - What if the value of $\alpha$ is 'too large'?
  - Some people suggest a variant of the algorithm in which the value of $\alpha$ is decreased over time, i.e. its value in later iterations is smaller. Why do they suggest this? And why isn't it necessary?
- What happens if $J_{\beta}$ isn't convex?

### Gradient Descent for OLS Linear Regression

All that we need to apply this general algorithm to our specific scenario is to plug in the partial derivative of our definition of $J_{\beta}$. We want this:

$$
\frac{\partial}{\partial \beta_j} \frac{1}{2m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)})^2
$$

which is

$$
\frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)}) \times x_j^{(i)}
$$

So we now get gradient descent for OLS linear regression:

repeat
- simultaneously update $\beta_j$ as follows:
  \[
  \beta_j \leftarrow \beta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\beta(x^{(i)}) - y^{(i)}) \times x_j^{(i)}
  \]
  until convergence

### Gradient Descent for OLS Regression in Python

Watch out for the vectorized implementation of the simultaneous update rules!
In [3]: 

```python
def J(beta, X, y):
    """
    Loss function for OLS regression
    """
    h = X.dot(beta)
differences = h.flatten() - y
sq_differences = differences ** 2
return (1.0 / (2 * y.size)) * sq_differences.sum()

def gds_for_ols_linear_regression(X, y, alpha, num_iterations):
    """
    Gradient descent search for OLS linear regression.
    alpha is the learning rate.
    num_iterations is the number of updates - instead of a better definition of convergence.
    It returns parameters beta and also a numpy array of size num_iterations, containing
    the value of the loss function, J, after each iteration - so you can plot it.
    """

    Jvals = np.zeros(num_iterations)
m, n = X.shape
beta = np.zeros(n)

for iter in range(num_iterations):
    beta -= (1.0 * alpha / m) * (X.dot(beta) - y).dot(X)
    Jvals[iter] = J(beta, X, y)

return beta, Jvals

Running the Code

We'll run it on the Cork property dataset but, for reasons that we will discuss in a later lecture, I'm going to leave out the floor area feature.
In [4]: # Use pandas to read the CSV file
def = pd.read_csv("dataset-corkA.csv")

# Insert the extra feature (all ones)
df.insert(loc=0, column='ones', value=1)

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df[['ones', 'bdrms', 'bthrms']].values
y = df['price'].values

# Run the GDS
beta, Jvals = gds_for_ols_linear_regression(X, y, alpha = 0.03, num_iterations = 4000)

# Display beta
beta

Out[4]: array([-57.66816632,  49.06247762,  91.69181718])

So, using just those two features ($x_2$ and $x_3$, the number of bedrooms and bathrooms resp.) and a learning rate of 0.03, after 4000 iterations, the model we have learned is

$$ y = -57.67 + 49.06x_2 + 91.69x_3 $$

**Sanity Check**

It's a good idea to plot the values of the loss function against the number of iterations. If its value ever increases, then

- the code might be incorrect (I think it's OK!)
- the value of $\alpha$ is too big and is causing divergence

So let's do that:
Stochastic Gradient Descent
The algorithm we have presented is sometimes called **batch gradient descent**. On every iteration, for every parameter, it computes the loss function over all the examples in the training set. Even with a highly optimized vectorized implementation, this may not scale well to very, very large training sets. If there are hundreds of millions of examples, say, then they may not even fit in main memory, in which case they have to be repeatedly read in from disk.

An alternative is **stochastic gradient descent** (or 'incremental gradient descent'). It looks at each example in turn, and modifies the parameters $\beta$ on the basis of that individual example. Here's the pseudocode:

```
repeat
  for $i \leftarrow 1$ to $m$
    simultaneously update $\beta_j$ as follows:
    $$
    \beta_j \leftarrow \beta_j - \alpha(h_{\beta}(x^{(i)}) - y^{(i)}) \times x_j^{(i)}
    $$
  until convergence
```

Because it is not guided by the *global* minimum, some of the individual parameter updates taken in stochastic gradient descent may not reduce the loss function: its ‘journey’ towards convergence may be less direct. On the other hand, if you use it only when you have massive training sets, stochastic gradient descent might actually take fewer iterations (of the outermost loop) than you require for typical uses of batch gradient descent.

### OLS Linear Regression: Normal Equation vs Gradient Descent

Gradient Descent is such an important algorithm in machine learning and AI in general, that it was worth studying it. Numerous machine learning algorithms make use of it.

But, for OLS linear regression, you probably wouldn't use it. The normal equation gives a much more direct method. Here's a comparison of the two:

- Gradient Descent requires that we choose a learning rate, $\alpha$; the normal equation does not.
- Gradient Descent requires a definition of convergence; the normal equation does not.
- Gradient Descent can have problems with convergence when different features have very different ranges of values: see the discussion of feature scaling in an upcoming lecture. These problems do not arise with the normal equation.
- But Gradient Descent can handle large numbers of features; you might prefer it if the number of features $n$ is very large (several thousand, say). The normal equation does not scale so well to very large numbers of features: $(X^T X)$ is $n \times n$ (where $n$ is the number of features) and the typical algorithms for computing matrix inverses (or pseudo-inverses) are cubic in $n$ (and even the algorithms with better complexity are all more than quadratic). Inverting a matrix with tens of thousands of features can be too slow.
- Gradient Descent is a general method for minimizing a loss function: it crops up across a wide range of learning algorithms. (This is why we studied it!) By contrast, the normal equation is specific to linear regression.
There's No Need To Roll Your Own!

Let's make one final point by way of conclusion: of course, you don't need to implement either of these linear regression methods yourself! A good library such as Python's scikit-learn algorithm will give you a method for doing it. You've already seen it done. But here it is again (and I'll use the same two features as above, for comparison):

```python
In [6]: from sklearn.linear_model import LinearRegression

# Use pandas to read the CSV file
df = pd.read_csv("dataset-corkA.csv")

# Get the feature-values and the target values into separate numpy arrays of numbers
X = df[['bdrms', 'bthrms']].values
y = df['price'].values

# Create linear regression object
estimator = LinearRegression()

# Train the model using the data
estimator.fit(X, y)

# Print the parameters that it learns
print('Intercept: ', estimator.intercept_)
print('Coefficients: ', estimator.coef_)

('Intercept: ', -57.668442969719365)
('Coefficients: ', array([ 49.06255205, 91.69181314]))
```