Introduction to Machine Learning (by Implementation)

Lecture 4: Multiple Regression

Ian J. Watson

University of Seoul

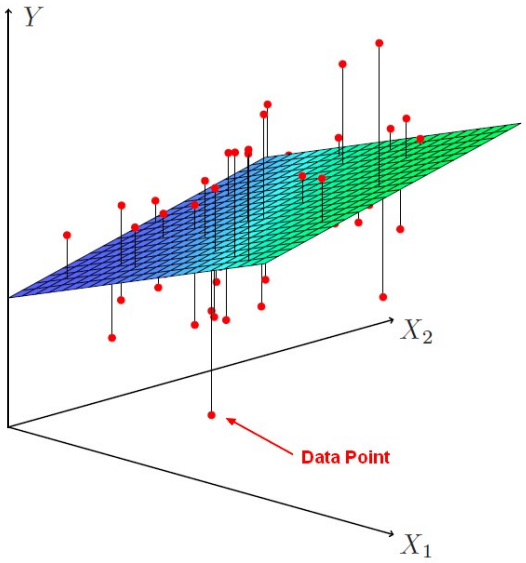
University of Seoul Graduate Course 2019

KRF KOREA RESEARCH FELLOWSHIP
해외우수신진연구자유치사업

Regression

- Regression is one of the major tasks in machine learning
- Idea: given some pieces of data, can you predict some dependent variables
- We have some variables $\mathbf{x}$ that represent our *measurement*, and we want to predict some y based on that
- In parametric regression, we build a *model*, a function $f(\mathbf{x}; \boldsymbol{\theta})$ which depends on the measurement variables and a set of *parameters* $\boldsymbol{\theta}$, which will *fit* or *train* to some known data sample
  - I.e. we have some known sample of $\mathbf{x}_i \rightarrow y_i$ which we will use to fix the parameters of the model, e.g. intercept and slope of a line
  - This is a *supervised learning* problem
- Example, we may have a sample $x_i \rightarrow y_i$, which we think can be modeled by a Gaussian, $f(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma}}$
  - The best estimate for μ will be the sample mean
  - The best estimate for σ^2 will be the sample variance
  - The above can be easily derived (we did in our stats class last semester)

Multiple regression



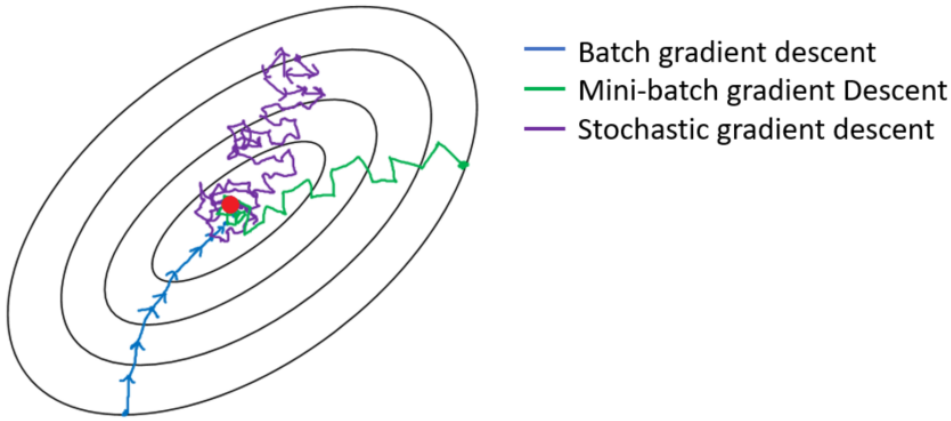
- Last week, we looked at simple linear regression: $y = \beta x + \alpha$
  - We solved for α and β exactly
- Today, we will extend to several potential explanatory variables: multiple regression

- If we have several variables, we can label them $x^{(j)}$, then extend the linear regression to include each variable
  - $y_i = \beta_0 + \beta_1 x_i^{(1)} + \ldots + \beta_k x_i^{(k)} + \epsilon_i$
  - Our model function is linear: $f(\vec{x}_i|\vec{\beta}) = \beta_0 + \beta_1 x_i^{(1)} + \ldots + \beta_k x_i^{(k)}$
  - As before, ϵ_i is our residual, the variance not captured in the model
- Multiple *linear* regression finds the best hyperplane which fits the data
- We can again try to minimize the loss, $L = \sum_i (f(\vec{x}_i|\vec{\beta}) - y_i)^2 = \sum_i \epsilon_i^2$
  - This week, we won't solve exactly but use our gradient descent code

Stochastic Gradient Descent

- When we did gradient descent, we assumed that we used the full function on each update
- Recall, our loss function is $L = \sum_{i=1}^{n} (f(\vec{x}_i; \vec{\theta}) - y_i)^2$
- This requires evaluating our regression function f at each point in the dataset every time we update the parameters
- For large datasets and/or complex functions f, this can be very expensive, and so very slow to converge
- Instead, our loss functions can be separated into *mini-batches*, and the updates done per batch:
  - Take a subset of the data U,
    we will take $U = \{\vec{x}_i\}$, i.e. a single data point, for simplicity
  - Evaluate and do the gradient descent parameter update on the loss function of the subset $L = \sum_{x_i \in U} (f(\vec{x}_i; \vec{\theta}) - y_i)^2$
  - Repeat until convergence
  - The stopping condition is subtle, since we only take part of the data, we can circle around for a long time, so we will shrink the learning rate
  - **Carefully read the algorithm description at the back**
- This general procedure is called *stochastic gradient descent*

SGD Illustration



- If we did normal gradient descent, we calculate the loss and gradient over the entire dataset (the whole "batch")
  - This can be very expensive, and very slow (especially the way we wrote it, with several test η)
- Instead, we can split up into mini-batches (or individual data samples, shown as SGD in the diagram), and take many small steps
  - Since each data point can have different prefered directions (due to the overall residuals), it will be less stable
- We will run over the whole dataset, then check convergence
  - We will require 100 such runs through the data with no improvement, decreasing out learning parameter each iteration

SGD Extensions

- Stochastic gradient descent should converge to the same point, and since it requires less evaluations per update, should converge faster
- But, since you only use part of the data each time, it can also wander more around the parameter space
- There are several extensions to SGD in common use to control updates and improve convergence, the main ones being
  - Momentum: keep track of the gradients and average over them
    - As batches go through, the average should remove fast varying components from residuals, and increase in the direction to the true minimum
    - $\Delta w_{i+1} := \alpha \Delta w_i - \eta \nabla f$, $w_{i+1} = w_i + \Delta w_{i+1}$
    - η is the learning rate as before, α controls how fast the momentum builds up
  - Adaptive gradients: generally, the learning rates for the parameters are set independently and updated as the gradient descent progresses
    - E.g. Adagrad keeps track of the size of the updates, and dampens fast-varying components: $g_{t+1} = g_t + \nabla L$, $w_{t+1} = w_t - \alpha \nabla L / \sqrt{g_{t+1}}$
    - RMSProp works similarly, but dampens with an exponential decay parameter γ: $g_{t+1} = \gamma g_t + (1-\gamma) \nabla L$, $w_{t+1} = w_t - \alpha \nabla L / \sqrt{g_{t+1}}$

R^2 in multiple regression

- The coefficient of determination naturally extends to multiple regression
- $R^2 = 1 - SSR^2/SST^2$,
  $SSR^2 = \sum_i (f(\vec{x}_i) - y_i)^2$, $SST^2 = \sum_i (y_i - \langle y \rangle)^2$
  - Describes what fraction of the variance in y is explained by the model
- We see though, that every variable you add is guaranteed to increase R^2
  - $y_i = \beta_0 + \beta_1 x_i^{(1)} + \ldots + \beta_k x_i^{(k)} + \epsilon_i$
  - If some β_j doesn't help reduce the residual, can set to 0 and get the same $\sum_i \epsilon_i^2$ as without it
- Need to be carefully interpreting the results
- A common procedure to understand the uncertainty when dealing with unknown datasets is the *bootstrap*, we may go through this later

Summary for Today

- Multiple linear regression extends simple linear regression to multiple independent variables $\vec{x}$ to explain a single dependent variable y by the linear function $f(\vec{x}|\vec{\beta}) = \beta_0 + \beta_1 x^{(1)} + \ldots \beta_k x^{(k)}$
  - In the code, you may find it simpler to introduce a dummy variable $x^{(0)}$, which is always 1, then you have simply $f(\vec{x}|\vec{\beta}) = \vec{\beta} \cdot \vec{x}$
  - I will accept either version
- We will use stochastic gradient descent (SGD) to find these $\vec{\beta}$
  - In stochastic gradient descent, instead of trying to update the parameters from the full loss function, $L = \sum_i (f(\vec{x}_i|\vec{\beta}) - y_i)^2$, we use a subset of the data, or a single data point for each parameter update
  - By updating on *mini-batches* the parameters should converge faster, with the caveat that different data points could pull the model in different directions
  - SGD is the heart of deep learning, every deep model you see has used an extension of SGD to find the parameter values

Exercises

- Implement stochastic gradient descent as a function `stochastic_minimize(f, df, x, y, theta0, alpha)`
  - Finds the parameters θ giving the minimum
  - The alpha parameter will reduce as we continue. (see next page)
- Run SGD to find alpha, beta (from last week) of the boston dataset for individual variables
  - Do you find the same values?
- Run SGD to find the best fit values for the full multiple regression of the boston dataset
  - You will need a `loss` function and ∇loss, `dloss`
- `loss(x_i, y_i, beta)` returns the loss for a single datapoint $(\beta_0 + \beta_1 x_i^{(1)} + \ldots + \beta_k x_i^{(k)} - y_i)^2$
- `dloss(x_i, y_i, beta)` returns the gradient of the loss $(2 \cdot (\beta_0 + \beta_1 x_i^{(1)} + \ldots + \beta_k x_i^{(k)} - y_i), 2x_i^{(1)} \cdot (\beta_0 + \beta_1 x_i^{(1)} + \ldots + \beta_k x_i^{(k)} - y_i), \ldots, 2x_i^{(k)} \cdot (\beta_0 + \beta_1 x_i^{(1)} + \ldots + \beta_k x_i^{(k)} - y_i))$

Exercises: Detailed Algorithm

- `stochastic_minimize(f, df, x, y, theta0, alpha0=0.001, iterations=50)`
  - `f`, $f = f(\vec{x}, y|\theta)$ will be the loss function for a single datapoint
    - So, $f(\vec{x}, y|\vec{\beta}) = (\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k - y)^2$ for today
  - `df` is the gradient of f w.r.t. the parameters, $2x_i f(x, y|\beta)$ for multiple regression
- Set `min_theta` to None, `min_value` to `float('inf')`, `alpha` to `alpha0`, `theta` to `theta0` and `iterations_without_improvement` to 0
- `while iterations_without_improvement < iterations:`
  - calculate `value`, the full loss function $\sum_i f(x_i, y)^2$
  - if `value` is less than `min_value`, reset `iterations_without_improvement` to 0, `alpha` to `alpha0` and set the new values of `min_value` to `value` and `min_theta` to `theta`
    - otherwise, add 1 to `iterations_without_improvement` and set `alpha` to `0.9*alpha`
  - `for x_i, y_i in in_random_order(data):`
    - calculate the gradient `gradient_i` or ∇f_i with `df`
    - set `theta` to $\vec{\theta} - \mathit{alpha} \cdot \nabla f_i$ (θ is a list, so you'll have to do component-wise subtraction!)

Tests

- I will test your `stochastic_minimize` against the boston dataset
  - The way to call `stochastic_minimize` is shown in `multiple.py`
  - Write your parameters in a file `results.txt`, one per line, starting with β_0, ending with $beta_1 3$
- Write some tests yourself! Make of some test data and a model with known minimum, and check you can find it
- E.g. try `x = [0], y = [0], f(x,y,theta) = t[0]**2`
  - The x, y are ignored by the model, so it should just find the minimum of `t[0]**2`
  - Example in `test_multiple.py`, don't rely on this alone though!!!
- You should find that the convergence is much, much better
  - We effectively have an adaptive gradient parameter in our α, c.f. our η list from gradient descent
- In many dimensions though, its easy to get into a false minima. Take starting parameters nearby the parameters from the individual fits. Play around with the parameters, what gives the best fit?
  - Bonus points to whoever finds the parameters with the smallest loss
  - Changing the iterations, and the original intercept plays a big part
  - You'll have to write a `full_loss` function to compare