UC Berkeley Self-Study/Review Notes: Machine Learning

Dun-Ming Huang

Contents

Chapter 1

DATA C100: Introduction to Modeling

1.1 Motivation

A **model** is an idealized representation of a system, which are mostly mathematical. Their mathematical properties lend us the computational opportunity to abstract a system in a computational space!
And in general, the machine learning works we perform in DATA C100 lend strength from constructions of models that allow us to predict new values from old data.
Outside the context of machine learning, there are a few categories of models that human history has used:

- **Deterministic Physical (Mechanical) Models**, such as kinematics equations.
- **Probabilistic Models**, which models how random processes can evolve.
- **Statistical Models**, which associates variables via statistical analysis.
- **Informal Models**, which are essentially stories or human-understandable descriptions of a complex phenomenon. Many pictographics might be an informal model.

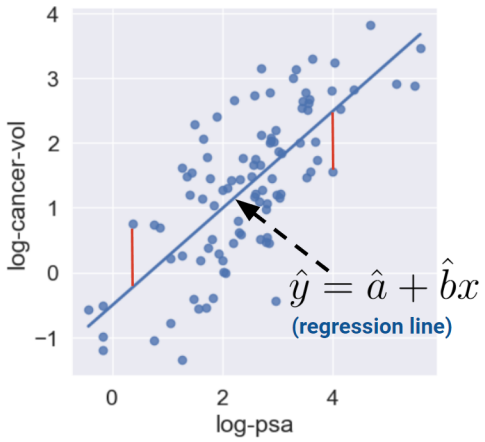
The introduction towards several categories of models only reinforce the idea that it is used for understanding the world as a complex phenomenon as well as providing predictions towards unseen cases.
Quite frequently, we would like to create models that are simple and interpretable, when we attempt to understand the association between different variables. But when we attempt to make extremely accurate preditcions, we would also risk providing an uninterpretable model whose complexity supports its performance.
Both directions of modeling in terms of interpretability have been used, and made their successes in each of their mission. Notably, complex models occur a lot in deep learning.

1.2 A Demonstration: Simple Linear Regression

1.2.1 What is a Regression Line?

Suppose we have a crowd of data points spread across a 2D plot, which we call a scatterplot, and we want to predict one dimension of the data point from another, then we may use a regression line:

Definition 1.2.1. Regression Line



A **Regression Line** is a linear model that attempts to predict a feature of a specific data-point via a linear combination of other features.

For now, let us simplify our description: "We would like to predict y by x".
The way we predict is by assuming there is a coefficient a and b such that:

$$y = a + bx$$

would accurately predict y for any provided x.
The most accurate parameter of this line would be denoted as $\hat{a}$ and $\hat{b}$, and the prediction following these best parameters would be denoted as $\hat{y}$, hence the regression line equation:

$$\hat{y} = \hat{a} + \hat{b}x$$

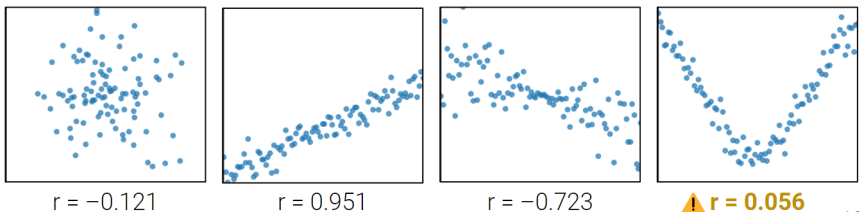
Each dataset that we attempt to apply a regression line onto has a specific statistic called **correlation**:

Definition 1.2.2. Pearson's Correlation Coeffcient

A **correlation** (denoted as r), formally known as the "Pearson's Correlation Coeffcient", quantifies how linearly associated two variables x and y may be via the following formula:

$$r = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{\sigma_x}\right)\left(\frac{y_i - \bar{y}}{\sigma_y}\right)$$

In other words, this is the average of the product of x and y, both measured in standard units.



At this point, you will probably have the following concerns:

- What does it mean when a parameter is "best" or "most accurate"?
- How do we obtain such parameters?

And here is where we deal with the mathematics of regression lines.

1.2.2 Model Selection: Simple Linear Regression

The **Simple Linear Regression** model is essentially a model using the regression line to predict the y of a datapoint for the x of a datapoint. In most definition, it's bound to only produce a two-dimensional regression line.
Such a model's prediction relies entirely on the value of its parameters. We recognize this by saying that Simple Linear Regression is a **parametric model**, described by its parameters a and b.
You might have noticed from the few graphs above that, the regression line doesn't accurately predict every single point. It is born from a set of points that don't necessarily form a line. Therefore, the best parameters that we provide it (again, called $\hat{a}$ and $\hat{b}$), are what we call the sample-based estimate of the inexistent, completely correct regression line.
Following that same logic, we noted the result of prediction as $\hat{y}$: our best estimate of the actual y of a new, unseen datapoint.

1.2.3 Quantifying Errors: Loss Functions

To quantify how "good" a prediction is, we would like to use a loss function:

Definition 1.2.3. Loss Function

A **loss function** is a function that characterizes the cost in predictions for choosing a set of parameters.
It quantifies how bad a prediction is for a single observation. The closer our prediction is to the actual observed value, the better the prediction, therefore, the lower the loss. Vice versa.

The choice of loss function affcets how we customize our model to adapt to mistakes. While it affects the accuracy of estimations, it would also decide the computational cost of estimation, as some loss functions are very costly to calculate for computers.
For Simple Linear Regression, we usually bother with two choices of loss functions:

Definition 1.2.4. Loss Functions of SLR

Squared Loss (L2 Loss)

$$L(y,\hat{y}) = (y-\hat{y})^2$$

This is a reasonable choice because there would be no loss when prediction is equal to observed value, and provide a lot of loss for predictions that are far from the observed values.

Absolute Loss (L1 Loss)

$$L(y,\hat{y}) = |y-\hat{y}|$$

This is a reasonable choice because there would be no loss when prediction is equal to observed value, and provide a fair amount of loss at a uniform sign (positive) for predictions that are far from the observed values (benefited from the absolute value).

Since we concern how costly our model's predictions are for the entire data set, the natural measure of how good a model is would be the average loss of model across all data points. This is also known as **empirical risk**:

Definition 1.2.5. Empirical Risk

The empirical risk is the average loss of a model across all data points. Minimizing empirical risk provides the optimal estimated parameters of a model for the corresponding loss function.
Mathematically expressed,

$$\hat{R}(\theta) = \frac{1}{n}\sum_i L(y_i,\hat{y}_i)$$

Here, θ represents the parameters of the model.

Let us inspect the case where we attempt to minimize the empirical risk with our loss being L2 loss function. In this case, we call our empirical risk **Mean Squared Error**.

Derivation 1.2.1: Estimated Parameters of Simple Linear Regression under MSE

To recall:

$$MSE(a,b) = \frac{1}{n}\sum_i (y_i - a - bx_i)^2$$

To minimize this function, we will find the conditions where the partial derivative of MSE with respect to every parameter is 0.

Minimizing a:

$$\begin{aligned}
\frac{\partial MSE}{\partial a} &= \frac{1}{n} \times \sum_i 2\frac{\delta}{\delta a}(y_i - a - bx_i)(y_i - a - bx_i) \\
&= \frac{1}{n} \times \sum_i -2(y_i - a - bx_i) \\
&= -\frac{2}{n}\sum_i (y_i - a - bx_i) \\
\frac{1}{n}\sum_i (y_i - \hat{y}_i) &= 0 \\
\sum_i (y_i - a - bx_i) &= \sum_i (y_i) - a - b\sum_i (x_i) \\
&= \bar{y} - a - b\bar{x} = 0 \\
a &= \bar{y} - b\bar{x}
\end{aligned}$$

Minimizing b:

$$\begin{aligned}
\frac{\partial MSE}{\partial b} &= \frac{1}{n} \times \sum_i 2\frac{\delta}{\delta b}(y_i - a - bx_i)(y_i - a - bx_i) \\
&= \frac{1}{n} \times \sum_i -2x_i(y_i - a - bx_i) \\
&= -\frac{2}{n}\sum_i x_i(y_i - a - bx_i) \\
\frac{1}{n}\sum_i x_i(y_i - \hat{y}_i) &= 0 \\
\frac{1}{n}\sum_i x_i(y_i - \hat{y}_i) - \frac{1}{n}\sum_i \bar{x}(y_i - \hat{y}_i) &= \frac{1}{n}\sum_i (x_i - \bar{x})(y_i - \hat{y}_i) \\
&= \frac{1}{n}\sum_i (x_i - \bar{x})(y_i - \bar{y} - b(x_i - \bar{x})) = 0 \\
\frac{1}{n}\sum_i (x_i - \bar{x})(y_i - \bar{y}) &= b\frac{1}{n}\sum_i (x_i - \bar{x})(x_i - \bar{x}) \\
r_{x,y}\sigma_x\sigma_y &= b\sigma_x^2 \\
b &= r\frac{\sigma_y}{\sigma_x}
\end{aligned}$$

Conclusion:

$$\begin{cases} \hat{a} = \bar{y} - \hat{b}\bar{x} \\ \hat{b} = r\frac{\sigma_y}{\sigma_x} \end{cases}$$

So at last, we also find that the optimizing condition of SLR model would be:

$$\begin{cases} \frac{1}{n}\sum_i (y_i - \hat{y}_i) & = 0 \\ \frac{1}{n}\sum_i x_i (y_i - \hat{y}_i) & = 0 \end{cases}$$

Which can be interpreted respectively as that:

- The residuals average to 0.
- The residuals are orthogonal to the predictor variable.

Chapter 2

DATA C100: Constant Model

2.1 Defining the Constant Model

Now that we have finished reading about simple linear regression from the previous lecture, let us discuss a slightly simpler model.

Definition 2.1.1. Constant Model

Constant Model, also known as a summary statistic, summarizes the sample data by always predicting the same number for any data point.
Mathematically expressed,

$$\hat{y} = \theta$$

In other words, the estimated y is always a constant parameter θ, whatever the input might be.

2.2 Experimenting Different Loss Functions

A model may have several options for its loss functions, which comes with different advantages and disadvantages. For this constant model, let us explore the L2 and L1 loss functions, see how each brings a different condition of optimization and robustness.

2.2.1 Exploring L2 Loss: MSE

In the L2 case, let us recall that our empirical risk is defined as follows:

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Therefore, for our constant model:

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta)^2$$

Let us proceed onto its optimization by attempting to find the critical point of empirical risk:

Derivation 2.2.1: Optimization of L2 Empirical Risk for Constant Model

$$\begin{aligned}\frac{\partial R}{\partial \theta} &= \frac{1}{n}\sum_{i=1}^{n} 2\frac{\delta}{\delta\theta}(y_i-\theta)(y_i-\theta)\\ &= -\frac{2}{n}\sum_{i=1}^{n}(y_i-\theta)\end{aligned}$$

And now, to optimize R,

$$\begin{aligned}\sum_{i=1}^{n}(y_i-\theta) &= 0\\ \bar{y}-\theta &= 0\\ \theta &= \bar{y}\end{aligned}$$

Therefore,

$$\hat{\theta} = \bar{y}$$

Let us enjoy some observations here.
First of all, the minimum MSE is thus:

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i-\bar{y})^2 = \sigma_y^2$$

, the variance of y.
Second of all, the estimated value of parameter θ is thus the mean of y. This means, provided an extreme outlier, the model will misbehave due to the mean being heavily influenced by some extreme outlier(s). Therefore, L2 Loss is not very robust (adaptative) towards the appearance of outliers.

2.2.2 Exploring L1 Loss: MAE

The L1 Loss Empirical Risk is also known as **Mean Absolute Difference**, which followed a similar naming logic to MSE.
Mathematically expressed,

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}|(y_i-\theta)|$$

The optimization of such an empirical risk becomes interesting (matheamtically), due to the appearance of absolute value.
However, we may always characterize the absolute value as a piecewise function:

$$f(x) = |x-\theta| \rightarrow f(x) = \begin{cases} x-\theta, & x > \theta \\ 0, & x = \theta \\ \theta - x, & x < \theta \end{cases}$$

Let us exploit this in the following toil:

Derivation 2.2.2: Optimization of L2 Empirical Risk for Constant Model

$$|y_i - \theta| = \begin{cases} y_i - \theta, & y_i > \theta \\ 0, & y_i = \theta \\ \theta - y_i, & y_i < \theta \end{cases}$$

$$\frac{\delta}{\delta\theta}|y_i - \theta| = \begin{cases} 1, & y_i > \theta \\ 0, & y_i = \theta \\ -1, & y_i < \theta \end{cases}$$

$$\frac{\partial R}{\partial \theta} = \frac{1}{n}\left(\sum_{\theta<y_i}(-1) + \sum_{\theta>y_i}(1)\right)$$

$$\sum_{\theta<y_i}(-1) + \sum_{\theta>y_i}(1) = 0$$

$$\sum_{\theta>y_i}(1) = \sum_{\theta<y_i}(1)$$

Therefore, $\hat{\theta}$ must be the median of y.

The estimated value of parameter θ is thus the mean of y. This means, provided an extreme outlier, the model will not misbehave due to the median not easily influenced by some extreme outlier(s). Therefore, L1 Loss is more robust (adaptative) towards the appearance of outliers.

2.2.3 Summary of Loss Function Optimization

In summary, the process of finding optimization conditions (which we also call **estimating equation**) would follow:

1. Differentiate the empirical risk with respect to parameters.
2. Attempt to find the critical point of empirical risk for per parameter.
3. Perform the derivative test (which requires multivariable calculus in most occassions, and is not performed for the span of DATA C100 for that reason) to confirm that the critical point is a minima.

The multivariable perspective offers a much computationally heavy test for confirming whether a critical point is a minima, which would involve calculating plural higher order partial derivatives. For those who are interested, this is in-scope for MATH 53.

2.3 Transformation and Model Linearity

In some cases, we face how the predictor variable x is not linearly correlated with y, but a transformation of x might be.
For example, the trajectory of a baseball is mostly quadratic. If I'd like to predict its motion, it is best that I present a model whose shape is not linear, but rather, quadratic:

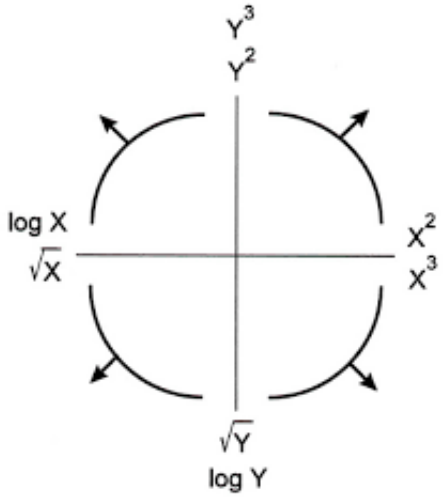
$$\hat{y} = \hat{a} + \hat{b}x^2$$

This happens frequently across datasets, but the question is: is the model still linear in this case?
The answer is yes. Linear Regression requires a regression line, decided as a linear combination of parameters. In this

case, the predictor variable is being a non-linear term, but the regression equation is still linear with respect to each of the parameters.

As previously mentioned, models sometimes require transformations to behave better, fitting to the behaviour of the dataset more closely.
To decide what transformations seem optimal, we can employ the following figure:

Figure 2.3.1. Turkey-Mosteller Bulging Diagram



Each of these transformations suggest how to transform x and y via suggesting that, for the direction that data's shape currently bulges towards, we should transform x and/or y in that direction.
In this case, we are usually then given the choice to either transform x or y, or as priorly mentioned, perhaps both.

Chapter 3

DATA C100: Ordinary Least Squares

3.1 Multiple Linear Regression Model

We have seen Simple Linear Regression Model, which uses one parameter for a constant term and another parameter to introduce the predictor variable.
The model currently has one predictor variable.
What if we can use more? What if we need to use more? What if our model would benefit greatly because what we attempt to predict in nature requires two or more variables for a good prediction?
If so, such a model would have some regression equation whose shape is to a huge degree similarly shaped as below:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p$$

Let us attempt to vectorize the above equation. Suppose, we rewrite the above equation as follows:

$$\hat{y}^{(i)} = \begin{bmatrix} 1 & x_1^{(i)} & \cdots & x_p^{(i)} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_p \end{bmatrix}$$

Then, we have successfully written the above equation in a matrix-vector multiplication form.
The anatomy of each term in the above row vector, $x_k^{(i)}$, is as follows:

- $x^{(i)}$ stands for the i^{th} data point inside the dataset.
- x_k stands for the k^{th} feature inside the dataset.
- Therefore, $x_k^{(i)}$ is the k^{th} feature of i^{th} data point inside the dataset.
- Among all columns, we appended another column of ones on the left of row vector to account for the need of intercept in regression line.

To vectorize this operation across numerous datapoints in the dataset, we may then formulate this model in terms of matrix-vector multiplication as follows:

$$\begin{bmatrix} \hat{y}_1 \\ \vdots \\ \hat{y}_k \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_p^{(1)} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_1^{(k)} & \cdots & x_p^{(k)} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_p \end{bmatrix}$$

And each term can then be respectively abbreviated into what is shown below:

$$\mathbb{Y} = \mathbb{X}\theta$$

Where we specifically name the matrix $\mathbb{X}$ containing the datapoints as the **design matrix**.

3.2 Optimization of Model: Least Squares Algorithm

3.2.1 Loss Function

The Loss function most frequently applied for such a model has to deal with a linear algebraic property called L2 Norm:

Definition 3.2.1. L2 Norm

The L2 Norm of a vector $\vec{x} \in \mathbb{R}^n$ is mathematically expressed as:

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

which occurs to be the magnitude of such vector $\vec{x}$.
We thus notate the distance of vectors $\vec{a}, \vec{b} \in \mathbb{R}^n$ as:

$$||a-b||_2$$

Our L2 loss function would be the distance between estimation and observed values, squared:

$$||\mathbb{Y} - \hat{\mathbb{Y}}||_2^2$$

Therefore yielding the empirical risk:

$$R(\theta) = \frac{1}{n}||\mathbb{Y} - \mathbb{X}\theta||_2^2$$

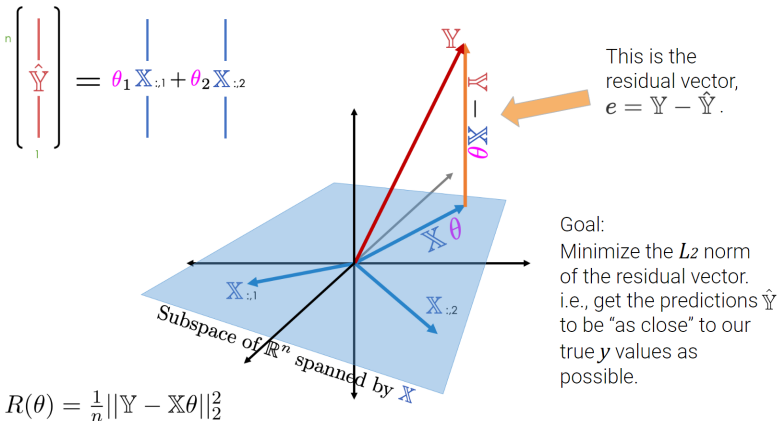
3.2.2 Optimization via Geometric Interpretation

We should first observe with our prior knowledge from EECS 16A (or alternatively MATH 54, just any college linear algebra introductory course), that since $\hat{Y}$ is a linear combination of the columns of $\mathbb{X}$ (as noted $\hat{Y} = \mathbb{X}\theta$),

$$\hat{\mathbb{Y}} \in span(\mathbb{X}) \subseteq \mathbb{R}^k$$

However, it is not necessary that $\mathbb{Y}$ belongs to the span of $\mathbb{X}$. Therefore, we see more clearly that our task is to minimize the distance between $\mathbb{Y} \notin span(\mathbb{X})$ and $\hat{\mathbb{Y}} \in span(\mathbb{X})$.
Let us observe a visualization from the DATA C100 Lecture Slides (since mine are underqualified and old):

Figure 3.2.1. Geometric Image of Residual in Multiple Linear Regression



This residual vector is essentially the shortest possible (minimized) when it is orthogonal to $\mathbb{X}\theta$ (which, in a 2D view, has to do with a property of right triangles called the Pythagorean Theorem).
There is a better intuition than Pythagorean Theorem, which is that the vector in $span(\mathbb{X})$ closest to $\mathbb{Y}$ must be its projection onto $span(\mathbb{X})$.
Either way, we will be introduced to a simplified situation:

To minimize

$$R(\theta) = \frac{1}{n}||\mathbb{Y} - \mathbb{X}\theta||_2^2$$

We require that

$$\mathbb{X}^T(\mathbb{Y} - \mathbb{X}\hat{\theta}) = 0$$

Such that the residual is orthogonal to $span(\mathbb{X})$.

Let us review its solution from EECS 16AB (or MATH 54, MATH 110):

Derivation 3.2.1: Least Squares Algorithm

$$\begin{aligned}\mathbb{X}^T(\mathbb{Y} - \mathbb{X}\hat{\theta}) &= 0\\ \mathbb{X}^T\mathbb{Y} &= \mathbb{X}^T\mathbb{X}\hat{\theta}\\ \hat{\theta} &= (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{Y}\end{aligned}$$

The equation above that shows the optimal parameters is known as the **Normal Equation**.
This normal equation would only be useful when $\mathbb{X}^T\mathbb{X}$ is invertible. Determining whether $\mathbb{X}^T\mathbb{X}$ is invertible is not as difficult as it sounds, since $N(A^TA) = N(A)$. I will not showcase this proof here.
Beyond the scope of DATA C100, we should also see some remedy for such situations when attempting to solve this exact optimization problem and working with a non-invertible design matrix. It is also not in the scope of this note yet.

3.3 Performance Factors

Just like in previous models, the residuals should be uncorrelated with the predicted values $\hat{y}$.
To determine the correlation of variables in a multivariable perspective like in Multiple Linear Regression, we work with a new coefficient:

Definition 3.3.1. Coefficient of Determination

The coefficient of determination characterizes the correlation of variables in a Multiple Linear Regression model:

$$R^2 = \frac{\text{Variance of predicted values}}{\text{Variance of observed values}} = \frac{\sigma_{\hat{y}}^2}{\sigma_y^2}$$

Just like the Pearson's Correlation Coefficient (r in Simple Linear Regression), R^2 spans between 0 and 1 (except it is the absolute value of r that spans between 0 and 1, not r itself).

Chapter 4

DATA C100: Gradient Descent

4.1 Computational Minimization of Loss Function

We have been able to minimize the loss functions provided for prior linear regression models. But we may always encounter harder loss functions to analyze, either via calculus or linear algebra-geometry.
This is where we fall back to the power of computation. Numerical computing. Such subfield of computer science focuses on solving complex mathematical problems using arithmetic operations. Very fortunately, Python is well supported by such libraries:

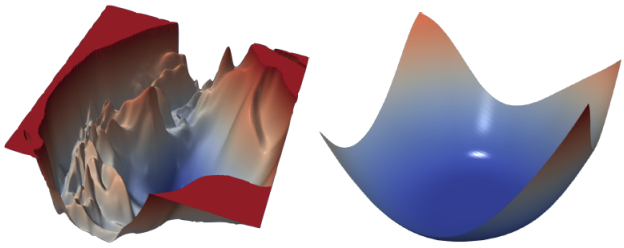
Code 4.1.1. Using scipy to Minimize Functions

```
scipy.optimize.minimize
Args:
    fun: The objective function to be minimized, where $x$ is a 1D
    array.
    x0: Initial guess, with the same size as input x from fun.
Returns:
    The result of optimization with a solution array for optimal
    parameters.
Example:
    >>> from scipy.optimize import minimize
    >>> minimize(arbitrary, x0 = [6, 3])
    OptimizationResult(x = [5.5, 2])
```

4.2 Gradient Descent Algorithm

When in a multivariable context, the loss function might take plural inputs and thus not be plottable in the two-dimensional space anymore. In that case, each point (x_1, x_2) correspond to the cost of choosing them as model parameters, and we end up with a loss surface.

Figure 4.2.1. Loss Function as a Surface



(a) ResNet-110, no skip connections (b) DenseNet, 121 layers

As you may see, each point (x_1, x_2) has a respective height for their respective loss value. Therefore, the lowest point on the surface is the point of optimization, where by choosing that point as the set of parameters, we arrive at the lowest possible loss value.
We will also take notice on why the right-side surface is much preferrable than the left-side surface.

Now, our task is focused on "how to descend on the loss surface". The entire algorithm that breifly leads us to descend off the loss surface is therefore known as the Gradient Descent Algorithm.

4.2.1 Interpretation of Gradients

This section is noted from MATH 53. It is not in scope for DATA C100. However, I think the reader can benefit from knowing what gradients are.
Let us begin with the notion of Directional Derivatives:

Definition 4.2.1. Directional Derivatives

We know how to compute the partial derivatives for a function $f(x,y)$ with respect to x and y, which are respectively the rate of change of f as we solely vary x and solely vary y.
The **directional derivative**, then, is the rate of change of f when we let both variables x and y change. Mathematically, we would be able to characterize the direction of changing as a vector.
The rate of change of $f(x,y)$ along a unit vector $\vec{u} =< a,b >$ can thus be denoted as:

$$D_{\vec{u}} f(x,y) = \lim_{h \to 0} \frac{f(x+ah, y+bh) - f(x,y)}{h}$$

In practical form, after a series of derivations:

$$D_{\vec{u}} f(x,y) =< f_x(x,y), f_y(x,y) > \cdot \vec{u}$$

Recall that:

$$\cos\left(\theta_{\vec{u},\vec{v}}\right) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}$$

Therefore, the directional derivative is maximized when the direction at which we change x and y is exactly the vector $< f_x(x,y), f_y(x,y) >$.
This means the direction provides the most positive change (provides the maximal rate of change) towards f, if we were to travel in that direction. For convenience and mathematical property, let us name this great observation as follows:

Definition 4.2.2. Gradient

The **gradient** of a function $f(x_1, \cdots, x_n)$ is the vector:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \cdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

Where $\nabla f(x_1, \cdots, x_n)$ notes the direction of greatest ascent (greatest rate of value increase) along the function f from $(x_1, \cdots, x_n)$

In other words, the opposite direction of the gradient is the direction of greatest descent. We can thus come up with the following iterative procedure for descending along a loss surface of $f(x, y)$:

Code 4.2.1. Sketch of Gradient Descent Algorithm

```
initial_guess = ? #an array of parameters for initial guess
for _ in range(epoch):
    grad = gradient(f, x, y)
    initial_guess -= grad
return initial_guess
```

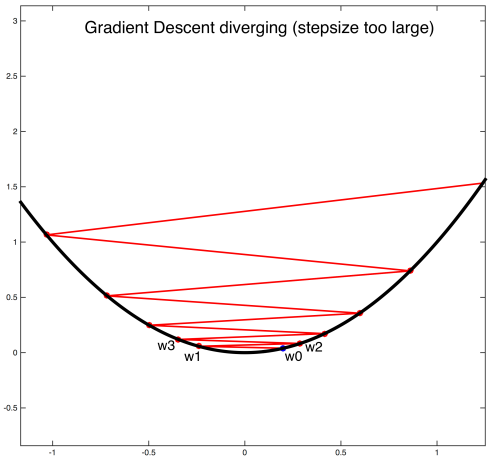
4.2.2 Improving the Gradient Descent Sketch

We have a few problems left.
First of all:

What if the direction of greatest descent doesn't lead us down anymore?

For example:

Figure 4.2.2. Divergence of Gradient Descent



As you may see, because the gradient's magnitude is too big, we ended up going away from the local minima.
To correct this situation, we will now improve the sketch of gradient algorithm to involve a parameter that controls what portion of the gradient do we travel. We call this the **learning step** (α).

However, notice that if the learning step is small, then the distance at which we descent along the loss surface would also be smaller per iteration. This makes it take too long for gradient descent to converge at the minimum. Therefore,

we would usually need to try learning steps by increasing or decreasing it tenfold until we find the appropriate learning step, and perhaps make it smaller as we go onto further iterations:

Code 4.2.2. Gradient Descent Algorithm

```
alpha = ? $customized learning step
initial_guess = ? #an array of parameters for initial guess
for _ in range(epoch):
    grad = gradient(f, x, y)
    initial_guess -= alpha * grad
return initial_guess
```

Mathematically

Expressed as a state-transition system (borrowing the work of EECS 16A):

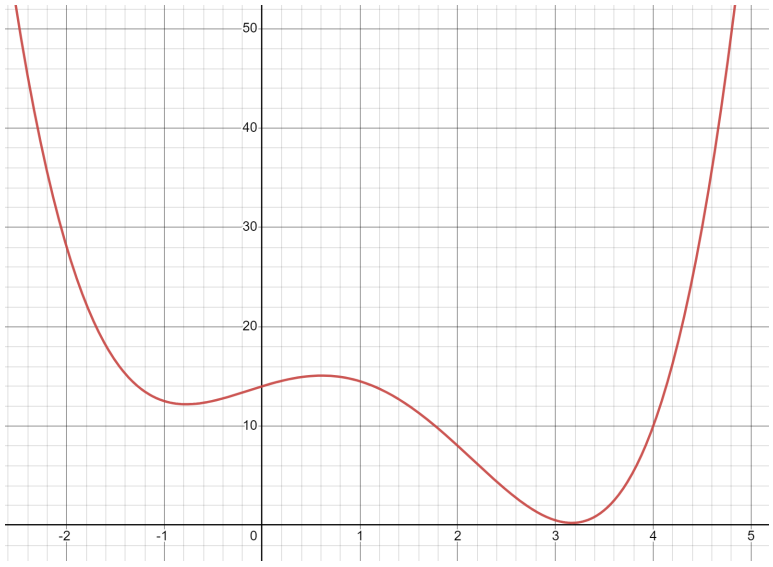
$$\vec{\theta}[t+1] = \vec{\theta}[t] - a\nabla L(\vec{\theta}, x_1, \cdots, x_n)$$

4.2.3 Convexity

We still have not decided how to make an initial guess of parameters, as well as how does gradient descent behave in real life.

First of all, let's perform a thought experiment. Suppose our loss function is a curve along which we gradient descent, what happens when we set the initial guess to the left side of the curve, and what happens when we guess from the right side of the curve instead?

Figure 4.2.3. Loss Functions with Local Minima



In this case, guessing from different directions would lead us to different local minima. Therefore, it is not exactly guaranteed that gradient descent leads us to the global minimum of our loss surface.

Then, if we manage to compromise with that (which we would have to, for now), how do we initialize our guess? This is where the idea of random initialization comes in for more complex models, where, we initialize each parameter to a random number as our initial guess.

For the scope of DATA C100, we would stick with zero initialization, where the initial guess of parameters would all be 0. This would lead to a problem. If the gradient descent updates unfortunately lead to the parameters keep being equal, then we will realistically make no improvements on the model since every predictor variable receives the same weight in a regression equation. But, for now, let us not worry about this until a long time later.

But some functions guarantee us to find a global minimum when using gradient descent. These functions are called **convex functions**:

Definition 4.2.3. Convexity

Function f is convex iff:

$$\forall a,b \in D_f, t \in [0,1](tf(a)+(1-t)f(b) \geq f(ta+(1-t)b))$$

Or in English: if a line is drawn between two points on the curve, all values on the curve must be on or below the line.

4.2.4 Stochastic Gradient Descent

Sometimes, our dataset is too big. We would therefore need to perform gradient descent by batches:

Definition 4.2.4. Mini-Batch Gradient Descent

In mini-batch gradient descent, we use a subset of data to calculate the gradient.
An approximate sketch of procedure follows:

- Decide a batch size (The amount of data used to compute gradient). A popular choice is 32 (according to professor, for no particular reason it seems).
- For all batches:

  Compute gradient on current batch of the data
- Repeat the process until we arrive at a stopping condition where we decide we have optimized the parameters sufficiently.

Via mini-batch gradient descent, we avoid heavy computational costs for large datasets and in turn are provided an approximation of the best way down an approximately true loss surface. It works well enough in practice.

We can also choose a batch size of 1, which would be a process called **Stochastic Gradient Descent** (SGD in some libraries).
In this case, we compute gradient based on one single data point. This technique is used on datasets whose data entries involve many parameters, as via one datapoint, we may have updated millions of parameters for that epoch. In the long run, the effect of SGD is very similar to computing the true gradient based on the entire dataset. Therefore, it is also a practice-able choice for real-life machine learning models.

Chapter 5

DATA C100: Feature Engineering

5.1 Motivation and Generalization

Sometimes, we would like to extract or build new features of the dataset off previous ones. Such an engineering process provides us more resources in modeling, and for its practicality we generally named it:

Definition 5.1.1. Feature Engineering

Feature Engineering is the process of transforming raw features into more informative features for modeling. Such a process allows the data scientist to capture, incorporate domain knowledge into the dataset, express non-linear relationships and predictor variables, as well as transform non-numeric features into numeric features.

Generally, throughout the process of feature engineering, we have a function that takes the original design matrix and transforms it into a richer design matrix via providing more features to work with from old ones.
This function is known as a **Feature Function**, which is customizable by data scientists' needs:

$$f : \mathbb{X} \in \mathbb{R}^{n \times d} \rightarrow \Phi \in \mathbb{R}^{n \times p}$$

The transformed data matrix is, as you see, now denoted as Φ.
By introducing new features that have a wide range of values, our model can become more sensitive to such data. This sensitivity is **variance**.

5.2 One Hot Encoding

To describe this process very generally:

Definition 5.2.1. One Hot Encoding

For every column of a categorical feature involving k categories, construct k new columns off it, with the elements being 0 and 1 respectively expressing whether the data record of corresponding row has a specific label as its value for the categorical feature.
For example:

	total_bill	size	day
193	15.48	2	Thur
90	28.97	2	Fri
25	17.81	4	Sat
26	13.37	2	Sat
190	15.69	2	Sun

	Thur	Fri	Sat	Sun
193	1	0	0	0
90	0	1	0	0
25	0	0	1	0
26	0	0	1	0
190	0	0	0	1

Multiple Linear Regression, however, can suffer from this.
In our design matrix, we always have a column of 1s: $\vec{1}$.
Using the exact example above, we can in fact find that:

$$\vec{Sunday} = \vec{1} - \vec{Thursday} - \vec{Friday} - \vec{Saturday}$$

because if a guest appeared on neither Thursday, Friday, nor Saturday, then the guest must appear on Sunday. In this case, we have created a linearly dependent design matrix.
Fortunately, the Python library for one hot encoding has captured this deficiency, and provided remedy with optional arguments:

Code 5.2.1. One Hot Encoding via Pandas in Python

```
pd.get_dummies
Args:
    data: A DataFrame, Series, or Array to One Hot Encode.
    drop_first: A boolean to indicate whether to drop the first
    category of each categorical variable to encode, so to
    prevent linear dependence.
Returns:
    The result of One Hot Encoding data.
```

Chapter 6

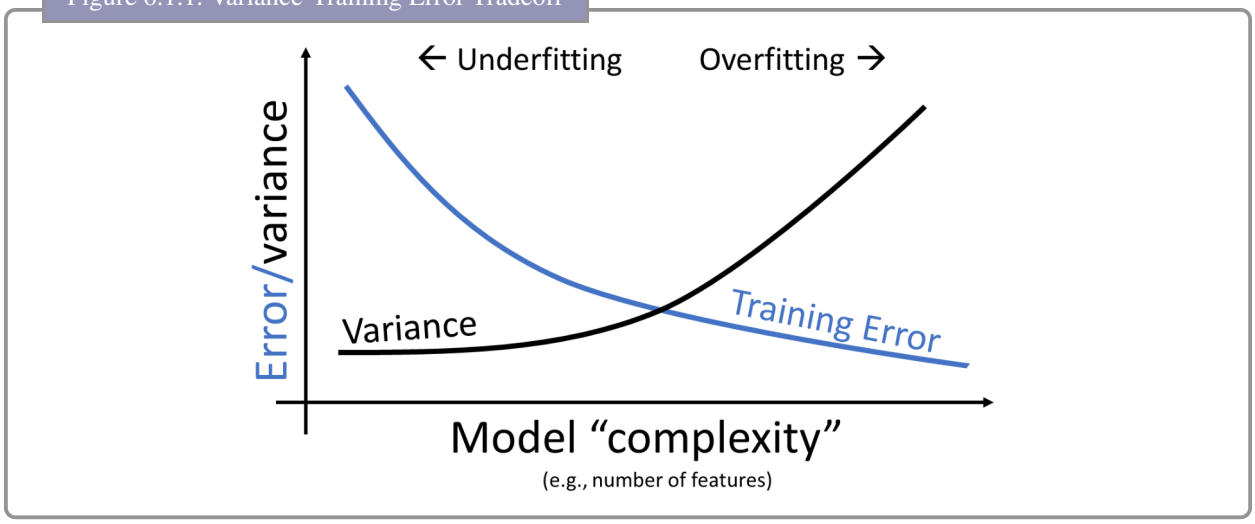
DATA C100: Cross Validation

6.1 Variance and Training Error

When attempting to form a model for predicting new datapoints, say linear regression, we often have some way to find optimal parameters.
The process of finding these parameters that lead to the best predictions possible is known as **training**. However, even the model itself would then have some errors when compared to the original data. This error is known as the **training error**.
To reduce training error, we might sometimes make the model more complex, so it covers more ground by using more information from a data entry and can end up making better predictions. The sensitivity of our model towards its fata is known as **variance**.
Of course, we can try to make a model entirely correct when applied onto the data we train it on. For an n-point dataset, we can use a $n-1$ degree polynomial that passes through all n points of the dataset. At that point, the model has high variance, high complexity, and virtually 0 training error.
But then, is this model use-able on other real-world data whose behavior deviates from the dataset?

Machine Learning scientists have developed the following famous diagram to discuss such phenomenon:

Figure 6.1.1. Variance-Training Error Tradeoff



When a model **overfits**, it refers to the phenomenon discussed above: the model is aggressively tailored towards the dataset and fails to predict data outside the dataset it's trained on.
In this case, while we do attain a zero MSE, the model is useless outside the dataset we have trained on.

6.2 Validation

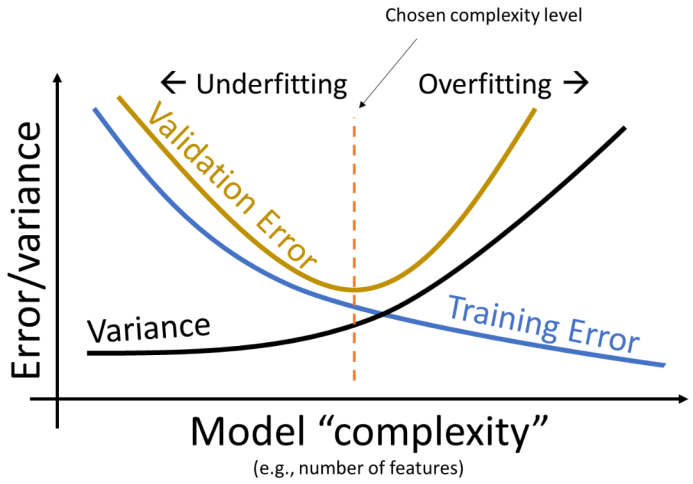
Scientists then came up with an idea: what if we attempt to validate our model using extra data point?
The first brief method that came into mind is a "holdout method", where for a datset of 100 points, we use 70 of 100 on training the data, which we call the **training set** (as in the dataset on which we train the model), and hold 30 of it out.
After training the dataset on the 70 data points, we test our model on the rest 30 which we have not trained on. This set of held out data is often called the **validation set** (or alternatively, development set, "dev set").
This allows us to validate our model by seeing how it behaves when predicting data outside the dataset we trained the model on. For the technology in DATA C100, its computation is as follows:

Code 6.2.1. Hold-Out Dataset in Python

```
>>> from sklearn.utils import shuffle
>>> training_set, dev_set = np.split(shuffle(df), [70])
```

Ideally, we then arrive at what the following diagram describes:

Figure 6.2.1. Variance-Erroe Tradeoff



where we attempt to use a model whose complexity allows the minimum training error and validation error.

In this case, the degree of predictor variables, which form a polynomial for the regression line, is what we can blame and adjust for the deficiency of our model in overfitting. In machine learning, these variables that influence the learning process as outlined by the above tradeoff diagram is called **hyperparameter**. This can also involve, for example, the ratio of number of data in training set to that in validation set.

There is also an inherent problem with this hold-out idea, which we will address in the following section.

6.3 K-Fold Cross Validation

If we continue to use the same validation set for every single time we test our data, then our data would also be tailored towards the validation set. This causes a similar mistake to overfitting.
To address this issue, let us consider another similar approach of valiation, called **K-Fold Cross Validation**.

Imagine that we have divided our entire dataset into k equal partition:

Figure 6.3.1. K-Fold Cross Validation Sets

Then, for each $i \in \{1, \cdots, k\}$, let us use the partition of dataset D_i as our validation set to train the model. Denote the resulting validation error as VE_i.
Record such value from VE_1 to VE_k, and note the average of these validation errors as the overall validation error of training this model.
This way, we managed to validate our model without being stuck with one same validation set, and can provide a much more objective review towards the quality of current model complexity.

Popular choices of such value k would be 5, 10, and N (the length of the entire dataset).
In the case we have chosen to perform N-fold cross validation, which is also known as "leave one out cross validation", each point becomes a validation set and will usually yield the greatest result. However, it is computationally expensive. On the other hand, using $k = 5$ offers a less ideal result in tradeoff with a smaller computational cost.

Code 6.3.1. Hold-Out Dataset in Python

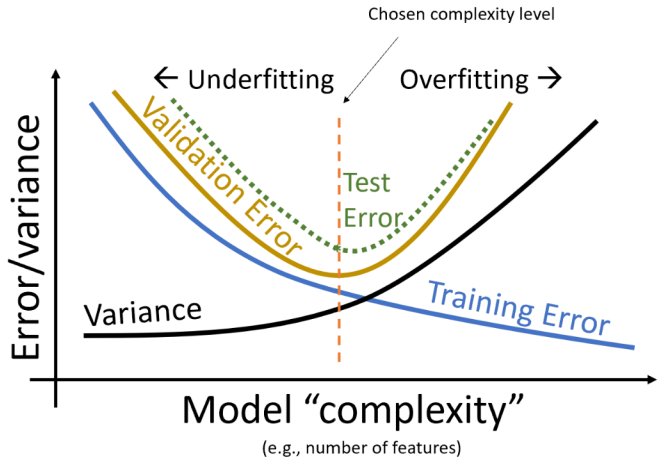
```
GridSearchCV
Args:
    estimator: The model for finding optimal parameters.
    param_grid: Hyperparameters stored in a dictionary.
    scoring: The loss function by which we assess performance
    of hyperparameters, usually "neg_mean_squared_error"
    cv: number of folds, or a 2D list containing indices
    for training set and dev set.
Returns:
    The optimal parameters for estimator specified.
```

6.4 Test Set

While we may find the model with the best validation set loss as the optimal one we will use in real life, when reporting the model on a public occassion (say paper, or actual incorporation with some procedure), we would like to assess our model using a special test set that we have never seen or used for any purpose.
This also relates back to how we preferred cross validation over hold-out on most occassions: avoiding biases towards the validation set used.
A test set can be easily produced via partitioning our original dataset into the **Training Set**, **Validation Set**, and **Testing Set**. We usually have some habits in how to partition a dataset into these subsets, such as producing a 70-15-15 partition.

To incorporate testing set into the big-picture of learning process with error-variance tradeoff, let's referrence this visual from DATA C100:

Figure 6.4.1. Variance-Training Error Tradeoff



We may see that the only difference between testing error and validation error is essentially how restrictive the computation of those error is, as we are, for testing purposes, neither allowed nor supposed to access the error curve for testing set.

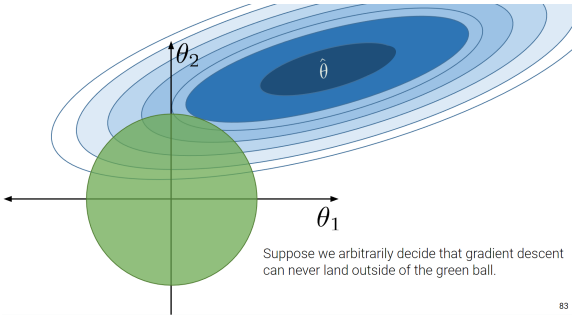
Chapter 7

DATA C100: Regularization

7.1 General Introduction to Regularization

There are other ways to prevent overfitting, and regularization is one of them.
The brief idea of regularization is to restrain the development on size of parameters.
For now, let us borrow a view from some two-parameter model under the gradient descent scheme for optimization:

Figure 7.1.1. Regularization on Graphical Representation, DATA C100



Now we may ask, how does the radius of this green circle tie in with reducing the complexity of a model.
When the green circle is small, we are limited into a less complex model, because the values of each parameters are more restrained. Under the extreme circumstance our green circle is essentially a point, the model would then output 0 for any possible predictor variable, as the output prediction of model is bound within the green circle.
Here, we may also notice that θ_0 is not involved in the model despite being involved in the multiple linear regression line. This would imply that reducing the green circle into a point produces a constant model that merely returns θ_0 as its prediction, which as discussed before, is optimally the mean of every observed value.
On the other hand, the larger the green circle's radius is the closer our model behaves to the original model. This allows for more complexity than the previously mentioned constant model.
Noticeably then, overly small and overly large radius for the restraining circle each lead to unideal consequences.

- Small radius leads to a constant model with miniscule variance, where validation and training error are both high.
- Large radius leads to the original model of regression, which allows for large variance and in exchange brings high validation error with low training error.

However, the nature of variables can lead to their parameters being naturally large. For example, variables that span between $1e-5$ and $1e-3$, perhaps due to unit or measurement, will naturally have larger parameters to help signify their significance in the regression model.
To prevent such issues, we should attempt to standardize each data by replacing every data entry with its z-score within

its variable category:

$$z_k = \frac{x_k - \mu_k}{\sigma_k}$$

(This is also how the course COMPSCI 70 evaluate each students' performance on an exam, due to its average frequently being around 50%).

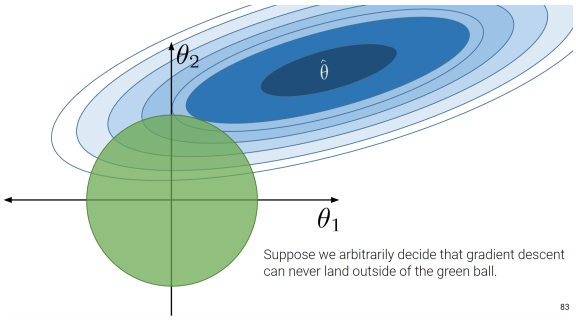
7.2 Choices of Regularization

In other choices of regularization scheme, the circle can be substituted by other shapes. The green shape on the visual at the beginning of prior section purposes to restrain the parameter size. Below, we will epxlore shapes other than circles that can also fulfill such duty.

7.2.1 L2 Regularization (Ridge)

Let us review the visualization of this scheme once again:

Figure 7.2.1. L2 Regularization



L2 Regularization is the scheme where we choose the green shape to be a circle around the origin. In this case, we attempt to minimize the empirical risk, paying attention to this constraint:

$$\sum_i \theta_i^2 \leq r^2$$

For an arbitrary, user-chosen r as the radius of the green circle.

We can run ordinary least squares in the coursework's technology using a "Ridge" class as follows:

Code 7.2.1. L2 Ridge Regularization in Python

```
>>> from sklearn.linear_model import Ridge
>>> ridge_model = Ridge(alpha = 10000)
>>> #alpha is proportional to inverse of radius of
    regularization circle
>>> ridge_model.fit(design_matrix, y)
>>> ridge_model.coef_
an array of optimal model parameters
```

The reason for which this class is named "Ridge" is because the OLS model with an L2 Regularization term is also called **Ridge Regression** in machine learning:

Definition 7.2.1. Ridge Regression Model

In a ridge regression model, we find parameters that minimize the following empirical risk as optimal:

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\theta_0 + \sum_{j=1}^{d}\theta_j\Phi_{i,j}))^2 + \alpha\sum_{k=1}^{d}\theta_k^2$$

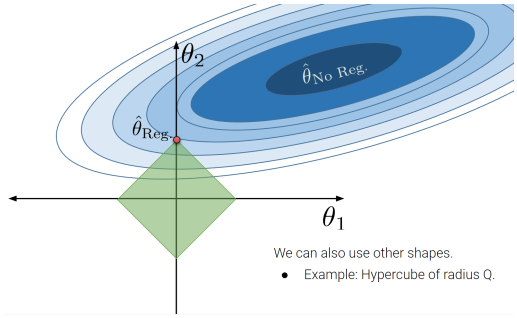
The optimal solution is, for linear algebra that we will not discuss until EECS 127:

$$\hat{\theta}_{ridge} = (\mathbb{X}^T\mathbb{X} + n\alpha I)^{-1}\mathbb{X}^T\mathbb{Y}$$

7.2.2 L1 Regularization (LASSO)

In this scheme, we replace the circle with a cube:

Figure 7.2.2. L1 Regularization



Our empirical risk thus becomes:

$$R(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \left(\theta_0 + \sum_{j=1}^{d}\theta_j\Phi_{i,j}\right)\right)^2 + \alpha\sum_{k=1}^{d}|\theta_k|$$

In the coursework technology, we may perform L1 regularization on OLS, otherwise known as LASSO Regularization, as noted below:

Code 7.2.2. L1 LASSO Regularization in Python

```
>>> from sklearn.linear_model import Lasso
>>> lasso_model = Lasso(alpha = 10)
>>> lasso_model.fit(design_matrix, y)
>>> lasso_model.coef_
an array of optimal model parameters
```

LASSO Regularization, or "Least Absolute Shrinkage and Selection Operator", tend to involve many zeros in its optimal parameters. In other words, it only chooses a subset of features to use in its regression model.

7.2.3 Summary of Regularization Choices

Name	Model	Loss	Reg.	Empirical Risk	Solution
OLS	$\hat{\mathbb{Y}} = \mathbb{X}\theta$	L2	None	$\frac{1}{n}\|\|\mathbb{Y} - \mathbb{X}\theta\|\|_2^2$	$\hat{\theta}_{OLS} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{Y}$
Ridge	$\hat{\mathbb{Y}} = \mathbb{X}\theta$	L2	L2	$\frac{1}{n}\|\|\mathbb{Y} - \mathbb{X}\theta\|\|_2^2 + \lambda\sum_{j=1}^{d}\theta_j^2$	$\hat{\theta}_{OLS} = (\mathbb{X}^T\mathbb{X} + n\lambda I)^{-1}\mathbb{X}^T\mathbb{Y}$
LASSO	$\hat{\mathbb{Y}} = \mathbb{X}\theta$	L2	L1	$\frac{1}{n}\|\|\mathbb{Y} - \mathbb{X}\theta\|\|_2^2 + \lambda\sum_{j=1}^{d}\|\theta_j\|$	no closed form