Intuition Behind Gradient Descent

The bivariate linear regression model can be expressed as $\mathbf{Y} = \theta_0 + \theta_1\mathbf{x}$. You can estimate the value of the regression parameters (θ_0 and θ_1) using the Ordinary Least Squares(OLS) method of optimization. However, there is another method that is widely used for estimating these parameters – Gradient Descent. But before getting into the nuts and bolts of this method let's understand why we need gradient descent in the first place.

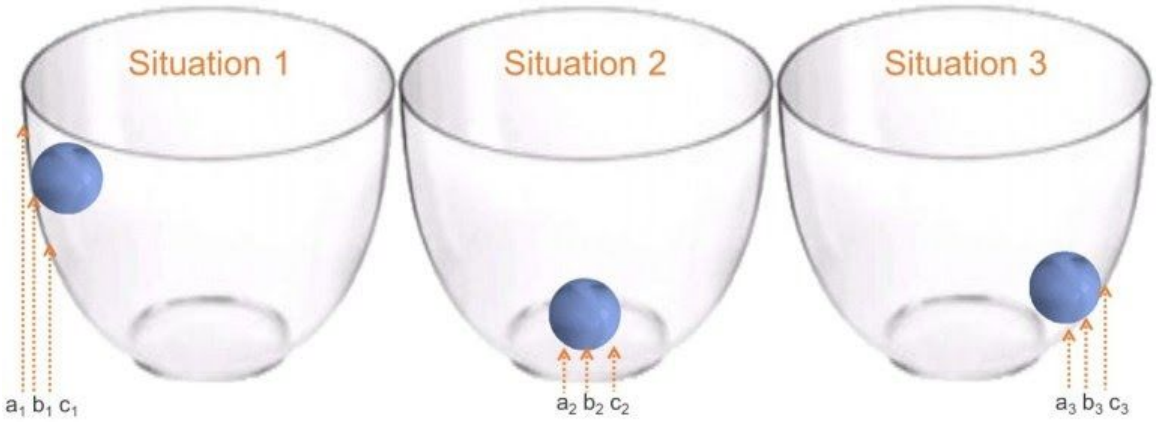
Why use Gradient Descent when we already have OLS?

There are a few reasons why algorithms like Gradient Descent are preferred over OLS for parameter estimation.

- The OLS method is an expensive process as it involves matrix inversion. As the amount of data increases, this operation becomes more complex (algorithmic complexity is
- $O(n^3)$ and hence it scales poorly.
- Most non-linear trends do not have a closed-form solution where OLS will fail to find parameters.

The intuition of Gradient Descent

Consider the following image:



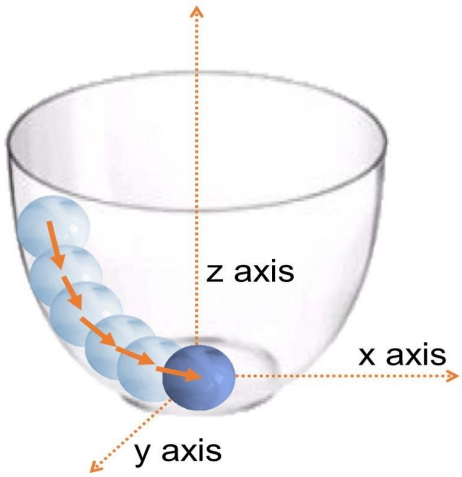
These three pictures represent three different situations where a ball is placed in a glass bowl at three separate positions. It is a known fact that the force of gravity will pull the ball towards the bottom of the bowl.

In situation 1, the ball will move from position b1 to c1 and not a1. This ball will continue to travel until it reaches the bottom of the bowl. In situation 2, since the static ball is already at the bottom it will stay at b2 and it won't move at all. In general, the ball always tries to move from a higher potential energy state to a lower state. Gravity is responsible for these different potential energy states of the ball.

But how is this similar to gradient descent optimization? Actually, this is exactly how gradient descent optimization works. The only major difference is that gradient descent optimization tries to minimize a loss

function instead of potential energy. (If you remember, the loss function in OLS was minimizing the sum of squared residuals.)

Now, let's understand how the ball will roll down when it is placed in situation 1 and why does it do so. In terms of the coordinate system, the potential energy reduces as the ball rolls down the Z-axis (i.e., the vertical axis). The ball tries to modify its position on the X and Y-axes to determine the lowest possible potential energy or the minimum possible value on the z-axis.



The gradient descent optimization can be thought of as a process similar to the rolling example in the following manner:

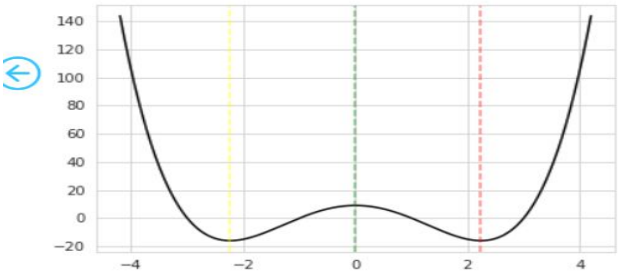
- The Z-axis can be considered as the loss function
- The X and Y-axes are the coefficients of the model
- The loss function is equivalent to the potential energy of the ball in the bowl
- The idea is to minimize the loss function by adjusting the X and Y-axes (i.e., the coefficients of the model)

Gradient Descent: An Iterative Approach to Find Minimum

Remember how you used differential calculus to find the minimum value of a function? In case you've forgotten or don't know about differential calculus do not worry, we are providing a simple example below to help you come to terms with it.

Let's solve for the minimum value of the $f(x) : x^4 - 10x^2 + 9$ function.

The plot for the function is shown in the following graph, where the yellow and green dotted vertical lines represent the values of x for which the function has the least value.



1. Differentiate with respect to `x` and set this value to `0`. Solve for `x` in this step

$$\frac{d}{dx} f(x) = 4x^3 - 20x = 0$$

2. Solving gives values of `x` for which the function `f(x)` has minimum values. The value of `x` thus found is

$$x = \pm\sqrt{5} = \pm 2.24$$

The method that we have shown just now is the **logical method**. While this method is pretty simple and straightforward, but unfortunately for a lot of complicated functions, it is not possible to solve equations this way. So we need to identify some numeric methods to solve such equations that require iterative calculations.

ITERATIVE METHOD

Let's start with initial value of `x = 5` for the function

$$f(x) : x^4 - 10x^2 + 9$$

with a learning rate

$$\alpha = 0.001$$

. **The learning rate controls how much we adjust the weights with respect to the loss gradient. The lower the value, the slower we travel along the downward slope and vice-versa.**

The gradient of the function for any value of x is:

$$\frac{d}{dx}(f(x)) = 4x^3 - 20x$$

FIRST ITERATION

At `x = 5`, the gradient is

$$4.5^3 - 20.5 = 500 - 100 = 400$$

Now, we will decrease `x` by an amount equal to the product of the learning rate i.e. α and the gradient at that point.

So, the new value of x is:

$$x = 5 - 0.001 * 400 = 5 - 0.4 = 4.6$$

This is the new value of `x` at the end of the first iteration

SECOND ITERATION

Following the same procedure as the first one we have,

New value of `x` as:

$$x = 4.6 - 0.001 * (4 * 4.6^3 - 20 * 4.6) = 4.302$$

If we continue doing iterations this way, it is possible to arrive at the minimum value, i.e., `2.236`.

Python implementation

A Pythonic implementation is shown as follows:

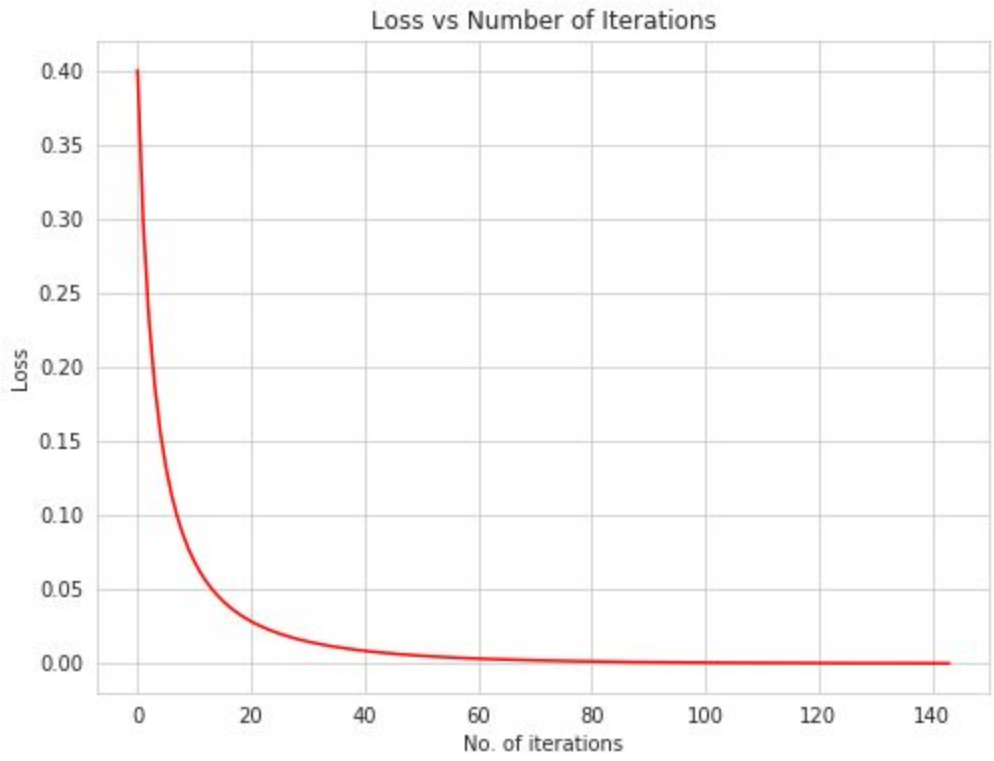
```
# Algorithm starts at x = 5
x_0 = 5
```

```
# Learning Rate
rate = 0.001

# Stopping criteria for gradient descent
precision = 0.0001

# Difference between current x and new x
diff = 1

# Maximum number of iterations
max_iters = 10000

# Number of iterations we performed
iters = 0

# Lambda function for calculating gradient
df = lambda x: 4*(x)**3 - (20*x)

'''Loop till difference between previous and new x-value is greater than precision
    and the number of iterations performed is less than the maximum number of iterations'''

while diff > precision and iters < max_iters:
    # Old x-value
    prev_x = x_0
    # New x-value
    x_0 = x_0 - rate*df(prev_x)
    # Absolute difference between old and new x-values
    diff = abs(x_0 - prev_x)
    # Increment number of iterations performed by 1
    iters += 1
    if not iters%12:
        print("Iteration {}: x = {}".format(iters, x_0))

print('='*50)

print("Global minima is at x = {}".format(x_0))
```

The output is:

```
Iteration 12: x = 3.0859858700832468
Iteration 24: x = 2.650430113743793
Iteration 36: x = 2.463011882474194
Iteration 48: x = 2.3668266660505166
Iteration 60: x = 2.3133970293306554
Iteration 72: x = 2.2824663690748417
Iteration 84: x = 2.264141835908863
Iteration 96: x = 2.2531388781091835
Iteration 108: x = 2.2464792668057707
Iteration 120: x = 2.2424291258463547
Iteration 132: x = 2.2399588108854416
Iteration 144: x = 2.2384494200813188
==================================================
Global minima is at x = 2.2384494200813188
```

The loss function also decreases as the number of iterations increases and ultimately, it stops after finding the optimum value of x at which the loss is minimum, as depicted in the following image:



The analogy of the Iterative Method with a Rolling Ball

The iterative method is similar to rolling a ball downwards, and measuring its position with each calculation until it reaches the bottom. The idea is to measure the speed of the ball based on the gradient or the slope of the curve.

Gradient Descent: A Mathematical Approach

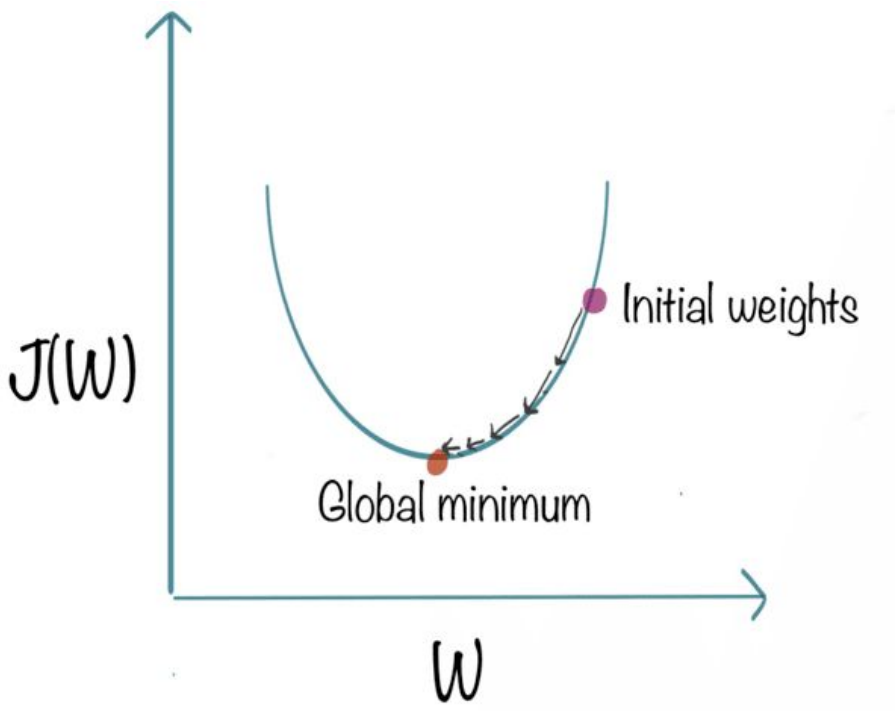
Four steps to simplify the entire process intuitively and mathematically, which is discussed as follows:

Steps of Gradient Descent

Given a machine learning model with parameters (weights i.e. θ) and a cost function ($J(\theta)$, find a good set of weights for our model which minimizes the cost function. The step-wise procedure would be as follows:

- Start with some initial set of values for our model parameters (weights and bias) where our goal is to find the best set of model parameters by iterations
- Calculate the gradient(G) of the cost function at the current set of model parameters as well as the cost function to find the direction of the decreasing cost function. This is the ideal direction that gives us the best model parameters.
- Update the weights by an amount proportional to the calculated gradient (G) at the current set of model parameters, i.e., $\theta=\theta-\eta G$.
- Repeat the second and third steps until the cost function stops reducing or, in other words, you have arrived at the global minima.

The following image describes exactly what gradient descent aspires to do:



Calculating Gradient Descent

In linear regression, let's assume we have an output variable $\mathbf{y}$ that depends linearly on the input vector $\mathbf{x}$ with m instances and n features. Matrix $\mathbf{x}$ is also the matrix that consists of all the feature vectors. We will approximate $\mathbf{y}$ by

$$\mathbf{y} = h_\theta(\mathbf{x}) = \theta^T \mathbf{x} = \sum_{i=1}^{n} \theta_i x_i$$

The cost function $J(\theta)$ is:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m} (X\theta - y)^T (X\theta - y)$$

The four steps described in the previous topic are expanded as follows:

1. **Initialize model parameters (θ)**: Let's initialize a set of model parameters as θ. We will be continually making improvements to the set to arrive at the best combination.

2. **Calculate gradients**: You will calculate the gradient of the cost function ($J(\theta)$) to find the direction of the minimizing cost function. Since there are a lot of model parameters ($\theta_i = 1$ to $\theta_i = n$), we will calculate the gradients for every model parameter (θ_i) by partial differentiation of the cost function($J(\theta)$) with respect to a particular model parameter (θ_i). Let's look at how to do that mathematically for a single training example:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial}{\partial \theta_j} (h_\theta(x) - y)^2$$

$$\frac{\partial}{\partial \theta_j} J(\theta) = 2.\frac{1}{2}.(h_\theta(x) - y)).\frac{\partial}{\partial \theta_j} (h_\theta(x) - y)$$

$$\frac{\partial}{\partial \theta_j} J(\theta) = (h_\theta(x) - y)).\frac{\partial}{\partial \theta_j} (\sum_{i=1}^{n} \theta_i x_i)$$

$$\frac{\partial}{\partial\theta_j}J(\theta) = (h_\theta(x) - y)).x_j$$

In vector form, the gradient is denoted by $\nabla(J(\theta)$and is given by,

$$\nabla(J(\theta) = \frac{1}{m}X^T(X\theta - y)$$

The right-hand side is the gradient and you calculate the value for this gradient with the current model parameters. Since this is the gradient for only a single training example, in order to find the overall gradient of the batch simply the gradients of all the training examples.

3. **Update weights**: Now, it's time to update weights with an amount proportional to the gradient. The proportionality constant is defined by the parameterαalso called the learning rate. Let's look at how to do this with a single training example:

$$\theta_j = \theta_j - \alpha.(h_\theta(x) - y)).x_j$$

In vector form:

$$\theta_j = \theta_j - \alpha.\nabla(J(\theta) = \theta_j - \alpha(\frac{1}{m}X^T(X\theta - y))$$

For m training examples, we have:

$$\text{Repeat until convergence: } \theta_j = \theta_j - \alpha.\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})).x_j^{(i)} \text{ for every j}$$

This weight update rule is intuitive because more correct is our prediction, lesser is the term $(h_\theta(x) - y))$ and vice-versa.

4. **Repetition of steps 2 and 3**: Repeat steps 2 and 3 until convergence occurs.

Since this approach updates weights by taking all the training examples into account, it is also called BATCH GRADIENT DESCENT.

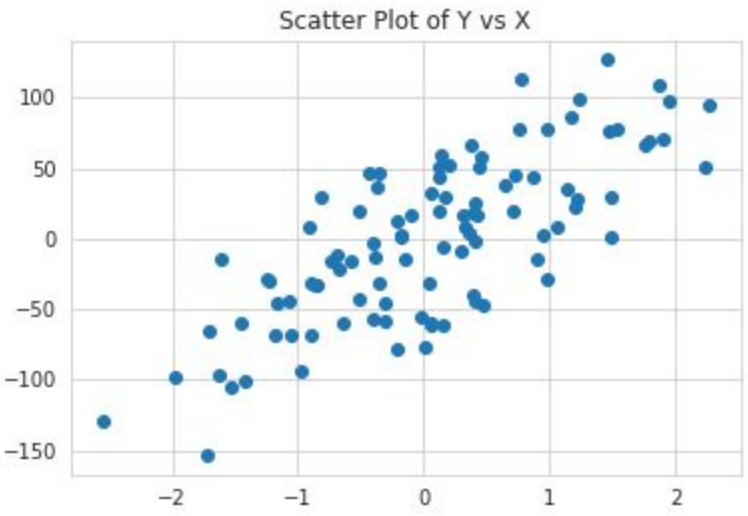
NOTE: Calculate the gradients in the second step by going through the entire batch of training examples and then use it to update weights.

Batch Gradient Descent with a Toy Dataset

A toy dataset with100instances is given consisting of a single feature and a continuous target. The first five rows look somewhat like this:

	Feature	Target
0	-0.359553	-31.745994
1	0.976639	-28.799142
2	0.402342	-2.184503
3	-0.813146	28.966797
4	-0.887786	-30.879117

Now let's take a look at the scatter-plot for distributions of 'Target' and 'Feature'. The scatter plot looks like this:



Python implementation with NumPy for calculating Gradient Descent is shown as follows:

```
def gradient_descent(alpha, x, y, numIterations):
    '''
    Arguments:
        alpha---> learning rate
        x---> features
        y---> target
        numIterations---> number of iterations
    Returns:
        theta---> model parameters
    '''
    # number of samples
    m = x.shape[0]
    # initialize model parameters
    theta = np.ones(2)
    # transpose design matrix
    x_transpose = x.transpose()
    # Iterate over the entire batch
    for iters in range(1, numIterations+1):
        # Predicted target
```

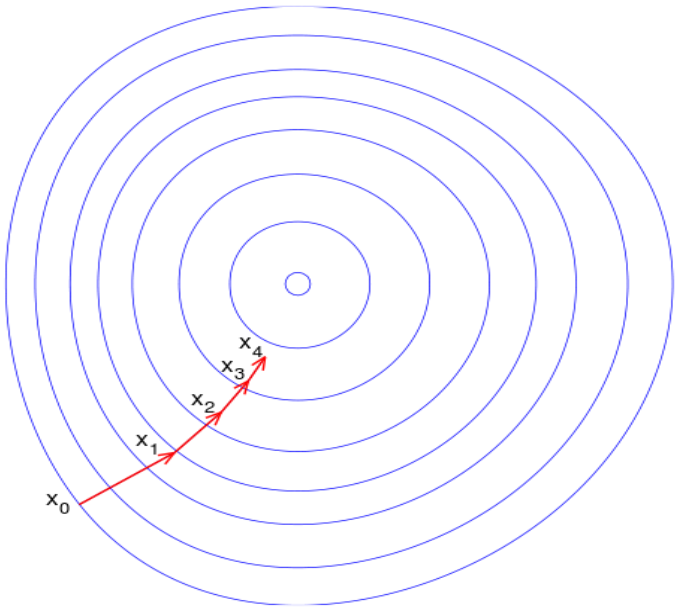
```
        hypothesis = np.dot(x, theta)
        # Loss
        loss = hypothesis - y
        # Cost function
        J = np.sum(loss ** 2) / (2 * m)
        # Calculate gradient
        gradient = np.dot(x_transpose, loss) / m
        # Update model parameters
        theta = theta - alpha * gradient
        if not iters%200:
            print("Iteration {} | J: {}".format(iters, J))
    print('='*50)
    return "Final thetas are {} and {}".format(theta[0], theta[1])
```

After running the above code on the toy dataset with 5000 iterations and 0.01 learning rate, the output is:

```
Iteration 0 | J: 1604.873080714912
Iteration 200 | J: 715.2500524101275
Iteration 400 | J: 699.5977423555787
Iteration 600 | J: 699.3061763021875
Iteration 800 | J: 699.3004712673605
Iteration 1000 | J: 699.3003552586653
Iteration 1200 | J: 699.3003528330281
Iteration 1400 | J: 699.3003527813338
Iteration 1600 | J: 699.3003527802181
Iteration 1800 | J: 699.3003527801936
Iteration 2000 | J: 699.3003527801934
Iteration 2200 | J: 699.3003527801932
Iteration 2400 | J: 699.3003527801935
Iteration 2600 | J: 699.3003527801934
Iteration 2800 | J: 699.3003527801934
```

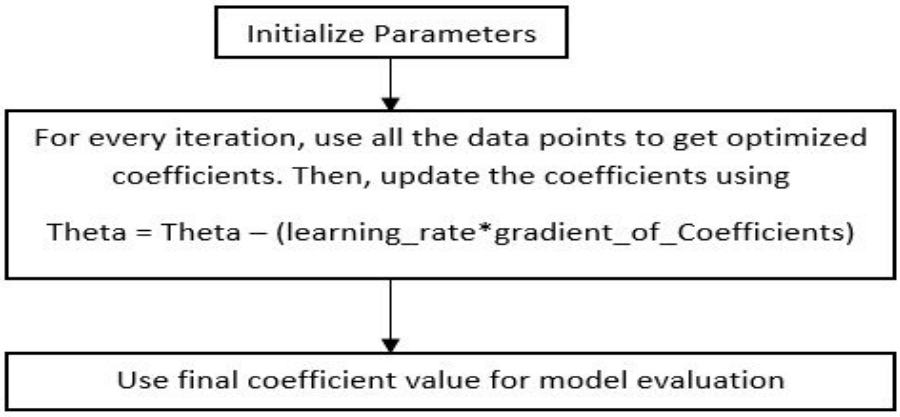
```
Iteration 3000 | J: 699.3003527801932

Iteration 3200 | J: 699.3003527801934

Iteration 3400 | J: 699.3003527801934

Iteration 3600 | J: 699.3003527801934

Iteration 3800 | J: 699.3003527801934

Iteration 4000 | J: 699.3003527801934

Iteration 4200 | J: 699.3003527801934

Iteration 4400 | J: 699.3003527801934

Iteration 4600 | J: 699.3003527801934

Iteration 4800 | J: 699.3003527801934

==================================================

'Final thetas are -2.84963639 and 43.20424388'
```

You can also visualize gradient descent somewhat like this where you are going in the direction of decreasing the cost with every ellipse denoting regions with a similar cost.



Stochastic Gradient Descent

The flow of the Batch Gradient Descent approach is:

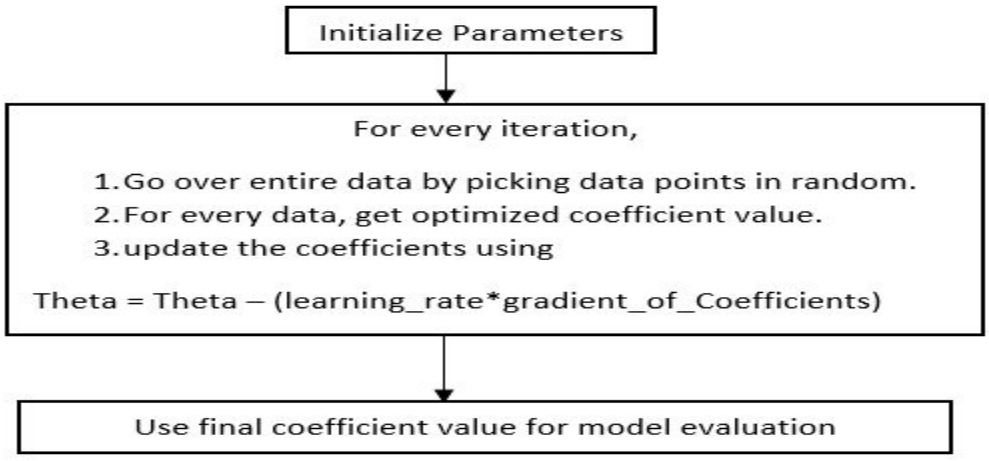


This approach of using the entire batch of training examples to update weights every time makes complete sense, however, with an increasing number of training instances it will become more cumbersome to update model parameters. This will result in exploding training time which should be avoided using some techniques.

A simple workaround is; *"Instead of using the entire batch for training, use a single randomly selected instance for calculating the gradient and update the model parameters."* This approach is called Stochastic Gradient Descent (SGD) because the gradient calculated this way is a stochastic approximation to the gradient calculated using the entire training data. Each update is now much faster to calculate than in batch gradient descent, and over many updates, we will head in the same general direction.

In SGD, with every iteration, you need to shuffle the training set and pick a random training example from that. Since you'll be using one training example, your path to the local minima will be very noisy like a drunk man after having one too many drinks.

The flow of SGD can be summarized as follows:



Pythonic Implementation of SGD

Now let's look at the Pythonic implementation of Stochastic Gradient Descent

```
def sgd(alpha, x, y, numIterations):
    '''
    Arguments:
        alpha ---> Learning rate
        x ---> Features
        y ---> Target
        numIterations ---> Number of Iterations
    Returns:
        '''
    # Number of training instances
    m = x.shape[0]
    # Initial thetas
    theta = np.ones(2)
    # Iterate till a desired number
    for iters in range(numIterations):
        # Pick a random number
        index = np.random.randint(m)
        # Random training instance (both feature and target)
        x_random = x[index].transpose()
        y_random = y[index]
        # Prediction for random instance
        y_pred = np.dot(x_random, theta)
        # Loss for random instance
        loss = y_pred - y_random
        # Cost for random instance
        J = loss**2 / (2*m)
        # Gradient for random instance
        gradient = np.dot(x_random, loss) / m
```

```
        # Update theta based on gradient
        theta = theta - alpha*gradient
        if not iters%1000:
            print("Iteration {} | J : {}".format(iters, J))
    print('='*50)
    return "Final thetas are {} and {}".format(theta[0], theta[1])
```

The function outputs the following when running with the toy dataset with the same

50000

50000and

0.01

0.01learning rat:

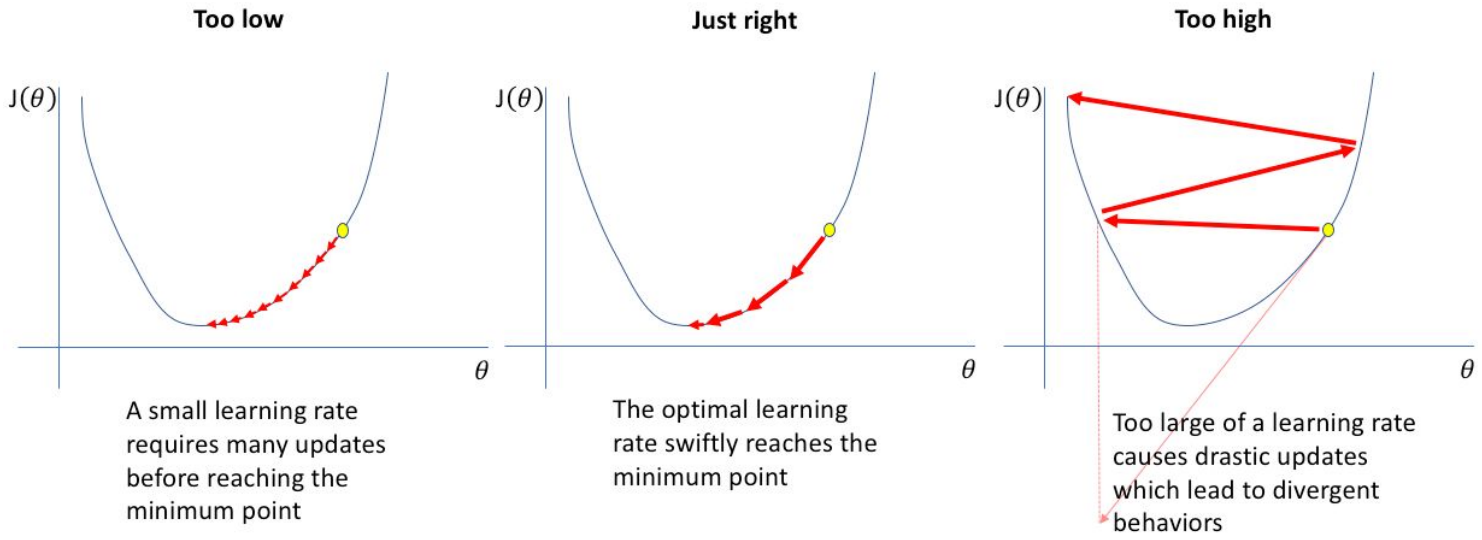
```
Iteration 0 | J : 21.26804415031862
Iteration 1000 | J : 9.23721020080173
Iteration 2000 | J : 0.053554520449303464
Iteration 3000 | J : 15.312069883348792
Iteration 4000 | J : 0.705699298171491
..............................
..............................
Iteration 45000 | J : 1.2920779852293338
Iteration 46000 | J : 11.957135340910568
Iteration 47000 | J : 0.12814469793457733
Iteration 48000 | J : 0.0006467831314213575
Iteration 49000 | J : 10.091530655370994
==================================================
'Final thetas are -2.757638863381235 and 42.76226358622706'
```

As you can see, the thetas obtained by SGD are almost similar to that by Batch Gradient Descent.

Learning Rate, Feature Scaling, and Mini-Batch Gradient Descent

Impact of Learning Rate

The learning rate is the amount by which you move down in the direction of decreasing cost denoted usually by α. If α is too big, then there may be a possibility to overshoot the minima. On the other hand, on choosing a smaller learning rate, the learning algorithm may take too much time to learn from the data, which will make our model slower. So, choosing a good learning rate that reaches the lowest cost swiftly is of prime importance with an iterative approach like gradient descent.

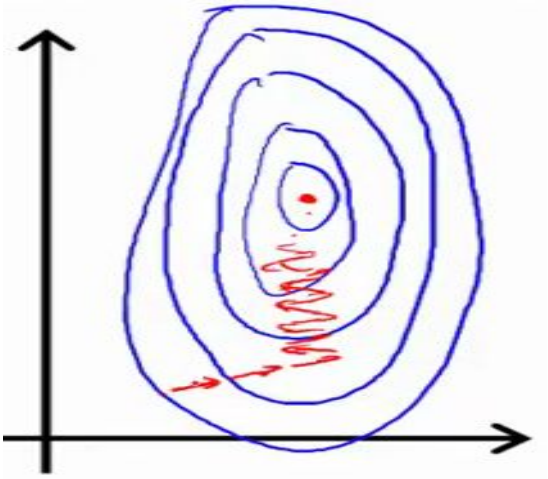


Feature Scaling in Gradient Descent

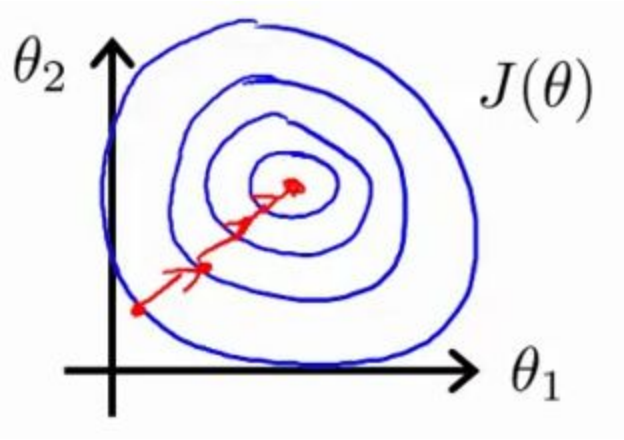
You must be wondering what is the role does of feature scaling in gradient descent? From the previous sections, we have the weight update rule for batch gradient descent as follows: $\boldsymbol{\theta j = \theta_j - \alpha . (h_\theta (x) - y)) . x_j}$

Observe that the updates are dependent on the x_j feature magnitude. If all the features are not on the same scale, this may lead to some weights getting updated faster than the others, which, in turn, slows down the entire process of gradient descent. The way to tackle this obstacle is to normalize the features that give us an even contour to navigate and find the best set of model parameters quickly.

Let's consider a situation with two features having $\theta 1$ and $\theta 2$ coefficients and $J(\theta)$ cost. In the case of unscaled features, we have the contour with an elliptical/oval shape. This shape makes it very difficult for the model to arrive at the optimum θs and takes a long time to do so.



Upon normalizing the features, we will obtain a more even contour and the model can quickly arrive at the best set of model parameters (θs) in lesser time as compared to the situation with non-normalized features.



Initializing Model Parameters

Weight/parameter initialization is a key step in gradient descent. This is because a close initial guess can lead to faster convergence and a bad guess can lead to a much longer time. It is almost like shooting a target in the dark. Only with domain expertise one can have an approximation for the model parameters.

Mini-Batch Gradient Descent

Until now you have come across two variants of gradient descent: Batch and Stochastic. There is another useful variant that falls somewhere between them, known as the Mini-Batch variant. Here, instead of iterating the overall training examples or a single training example, we process n training examples at once. This is a good choice for large data sets. For example, we can take 16, 32, 128, etc as the sample size of the mini-batch for training.

Questions:
Gradient Descent always finds global minima. True or false?

Ans: FALSE

Explanation:
When there exist functions with multiple local minima the gradient descent may or may not converge to the global minimum.

Polynomial Features

Dataset: During the entire concept, we will use the same housing dataset as we did for Linear Regression. In total, we have 114 features (you can read about them in the text file provided) from which you have to predict the `SalePrice` of the houses.

A lot of preprocessing is done so that you can focus more on the algorithm since this is the first time you will learn about regularization. Later on, you will learn later the techniques to preprocess data and prepare it for machine learning.

Why Polynomial Features?

The hypothesis in linear regression is that the features at our disposal have a linear relationship with the target. But what if during a visual inspection of relationships (take a scatter plot) you observe that the relationship is not actually linear but of some other type? Apart from that, multicollinearity may also exist between predictor variables. In such a case the model becomes highly unstable and small changes to the data can cause large changes to the model.

Also, during the inspection of residuals, if you try to fit a linear model to curved data, a scatter plot of residuals (Y-axis) on the predictor (X-axis) has patches of many positive residuals in the middle, and patches of negative residuals at either end (or vice versa). This is a good sign that a linear model is not appropriate, and a polynomial may do better. Here rises the need for polynomial features.

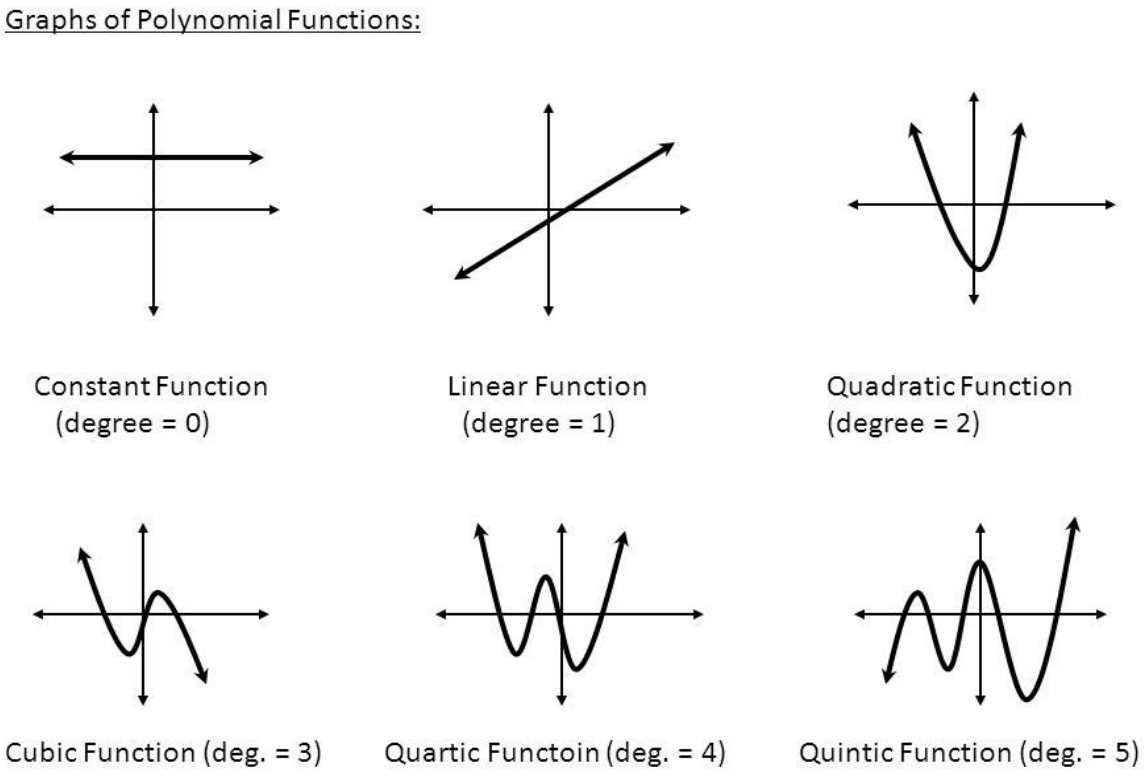
Polynomial Function

In statistics, polynomial regression is a form of regression analysis in which the relationship between the independent variable `x` and the dependent variable `y` is modeled as an `nth` degree polynomial in `x`.

The equation is $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + ... + \beta_n x^n$

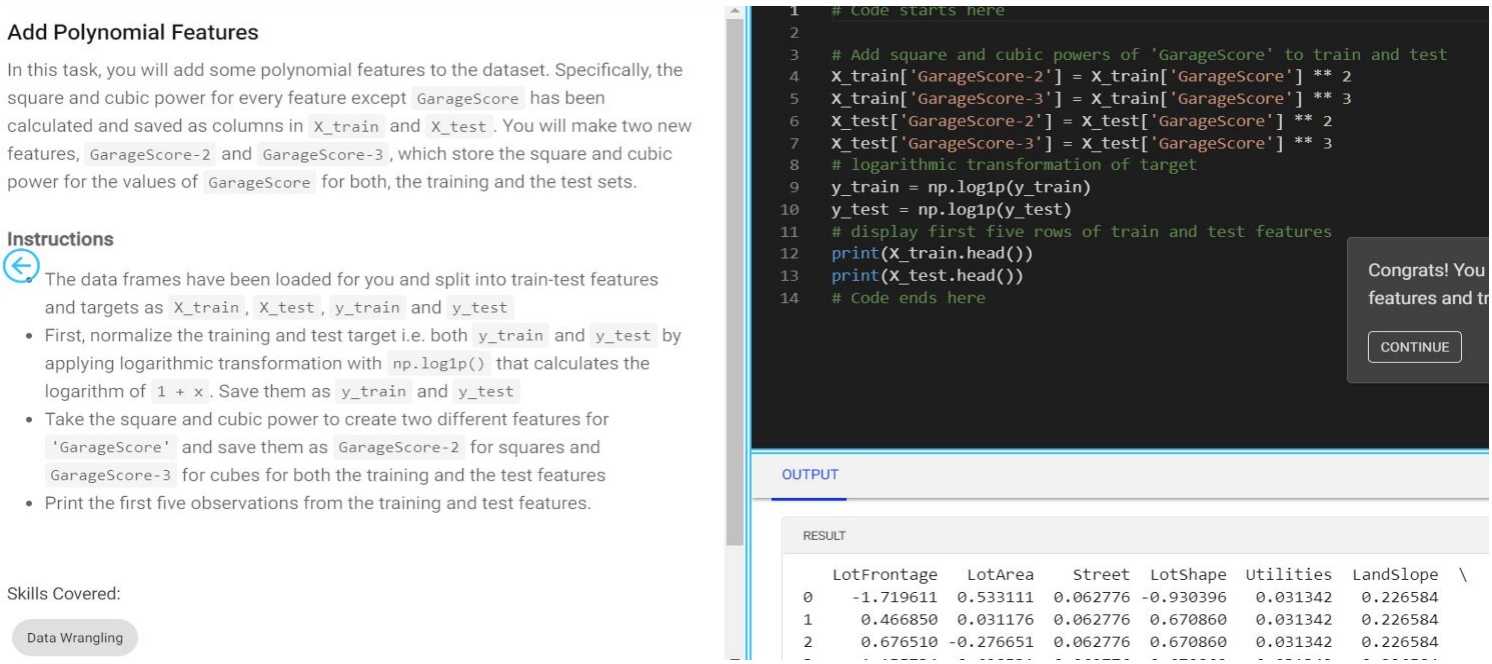
Here, we consider a single independent variable `x`. We can also have multiple independent features $(x_1, x_2, ..., x_n)$

Examples of Polynomial Functions



Don't Consider Polynomial Functions as Non-Linear Functions

Now, you may wonder as the relationship between independent and dependent variables is no longer linear this new type of function is a non-linear function. Wrong! This is a linear model as the function itself is still a linear combination of weights.



Bias-Variance Trade-off

By now, you already have a brief idea about predictive models - you preprocess the data (clean, remove outliers, skewness, etc.), fit a model, and make a prediction on it. Then you will calculate the error metric and try to find the best model that minimizes this error. This error is made up of three parts:

- Error due to squared bias
- Error due to variance
- Irreducible error

Mathematically,

$$\text{Error}(x) = \{\text{Bias}\}^2 + \text{Variance} + \text{Irreducible Error}$$

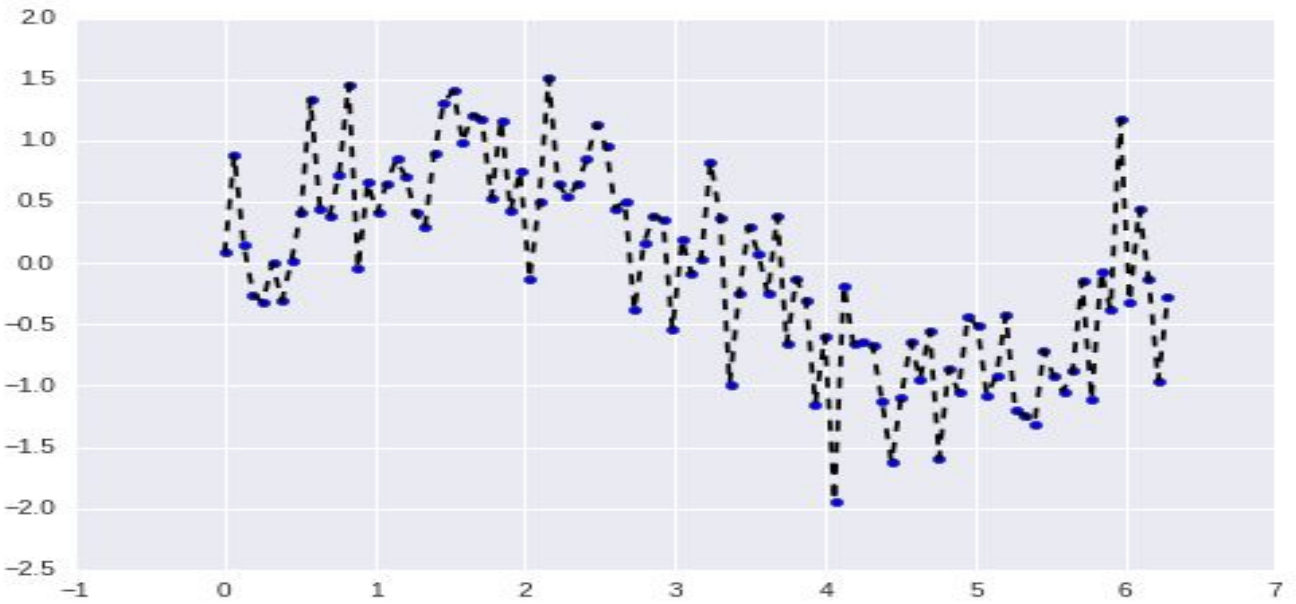
Irreducible error is also known as noise, and it can't be reduced by your choice in the algorithm. It typically comes from inherent randomness, a misframed problem, or an incomplete feature set. So, we will discuss what is bias and variance and also the trade-off required in order to find a good model.

Error due to squared bias: Simply put, bias is the difference between your model's expected predictions and true values. Expected predictions mean that you are averaging the predictions of the model (if you repeat the entire model building process on new random data multiple times). Due to the inherent randomness in the data, there is bound to be a range for predictions. Bias measures how far off are the predictions from the true values. Bias error arises due to incorrect assumptions in our learning algorithm which makes it hard to learn true relationships between independent and dependent variables.

A low bias model is one that can be highly flexible and has the capacity to fit a variety of different shapes and patterns. A high bias model would be unable to estimate values close to their true values. Models with high bias always lead to a high error on training and test data.

In the preceding image, no matter how many more data points we add, we simply won't be able to fit a straight line because it is unable to capture non-linear trends. In other words, we can say that the model is underfitting the data.

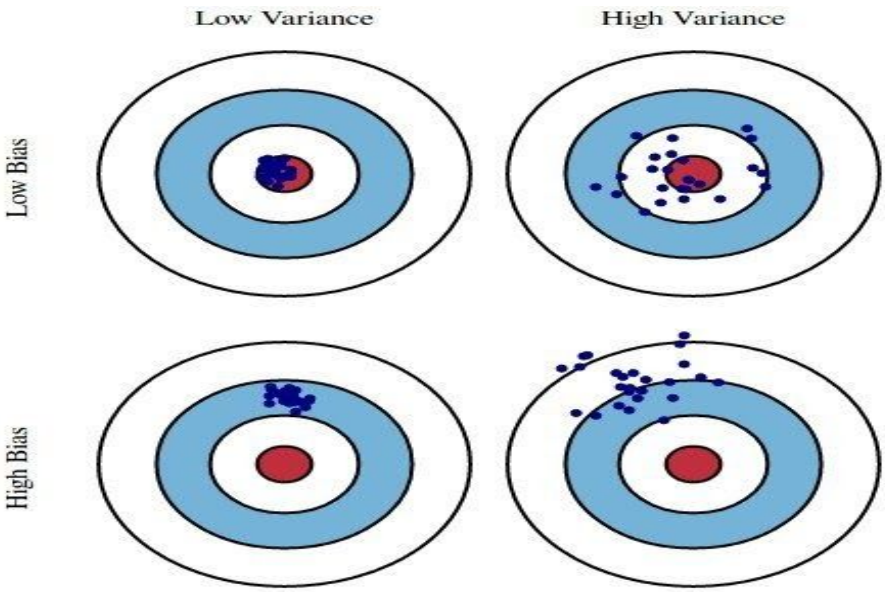
Error due to variance: Variance is the degree of fluctuation of a data point for different model predictions (assuming we repeat the entire model building process on new random data multiple times). A model with high variance pays a lot of attention to training data and does not generalize on the data it hasn't seen before. As a result, such models perform very well on training data but have high error rates on test data.



In the image above the model is capturing almost every data point and learns too much; however, when we test it on unseen data, it will not generalize well and will produce a high error. This phenomenon is also called overfitting.

Linear models, in general, have a high bias because of its inability to capture non-linear patterns.

Graphical Representation of Bias-variance



The preceding image is a graphical visualization of bias and variance using a bulls-eye diagram. Here, the center of the target is a model that perfectly predicts the correct values. As we move away from the bulls-eye, our predictions get worse and worse. Now, imagine that we can repeat our entire model building process to get a number of separate hits on the target. Each hit represents an individual realization of our model, given the chance variability in the training data we gather. Sometimes, we will get a good distribution of training data so we will predict very well and we are close to the bulls-eye, while sometimes our training data may be full of outliers or non-standard values that result in poorer predictions. These different realizations result in a scatter of hits on the target.

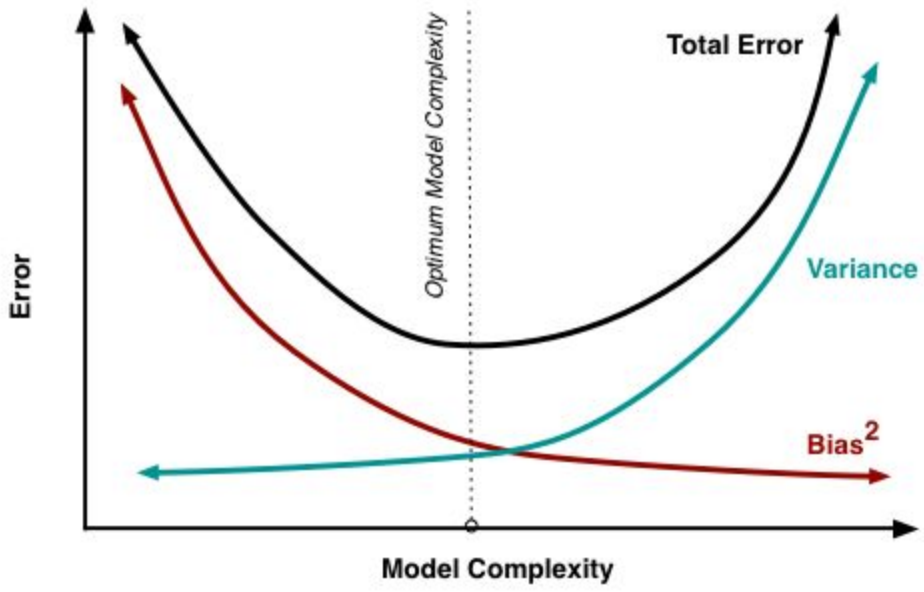
Bias-Variance trade-off is an important concept that get's asked frequently in machine learning roles.

Model complexity

Definition: Model complexity can be defined as a function of a number of free parameters; the more free parameters a model has, the more complex the model is. In a dataset, the free parameters are the independent variables (features), and so model complexity increases with the increasing number of features and vice-versa.

Relationship Between Model Complexity and Bias-variance

Understand it this way: the more features you have, the more the model will remember the patterns and hence more will be the variance. On the other hand, if you decrease the number of features at hand, you are reducing the memory power of the model (you are reducing variance); however, if you do it too much, it may ultimately lead to a generalization where the variance is very low but the bias is very high. The bias increases because you are having too many restrictions by reducing the number of features. So, while we try to reduce the variance, bias increases, and vice-versa. There is a trade-off between the two, which is shown in the graph below:



The above image captures the total error on the test data. Observe that by increasing model complexity, the bias decreases while variance increases. At the same time, the total error continues to go down until a point

and from there onwards it shoots upwards. Sometimes, this point is also called Sweet-Spot, a point or condition which we always try to achieve by balancing bias and variance.

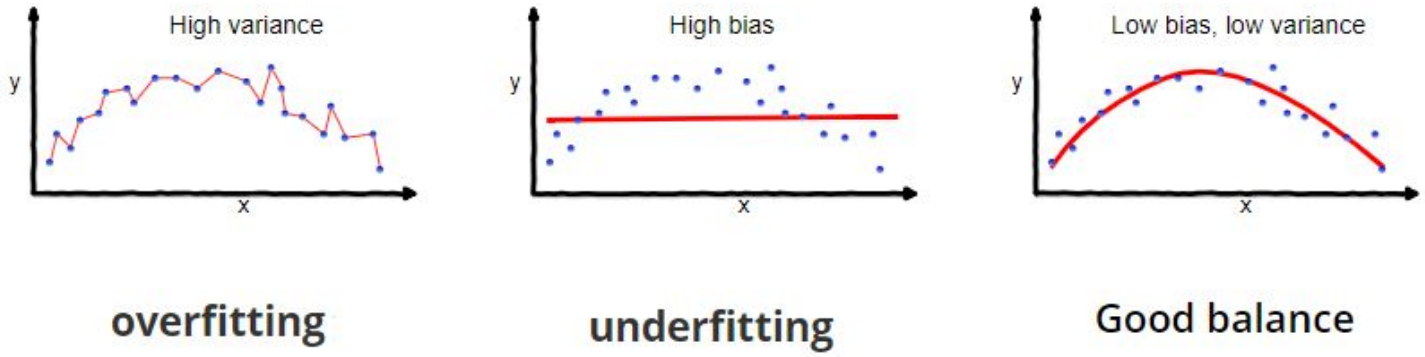
Footnote

- Overfitting: Too much reliance on the training data
- Underfitting: A failure to learn the relationships in the training data
- High Variance: Model changes significantly based on training data
- High Bias: Assumptions about model lead to ignoring training data
- Overfitting and underfitting cause poor generalization on the test set

Effect of Training Set Size

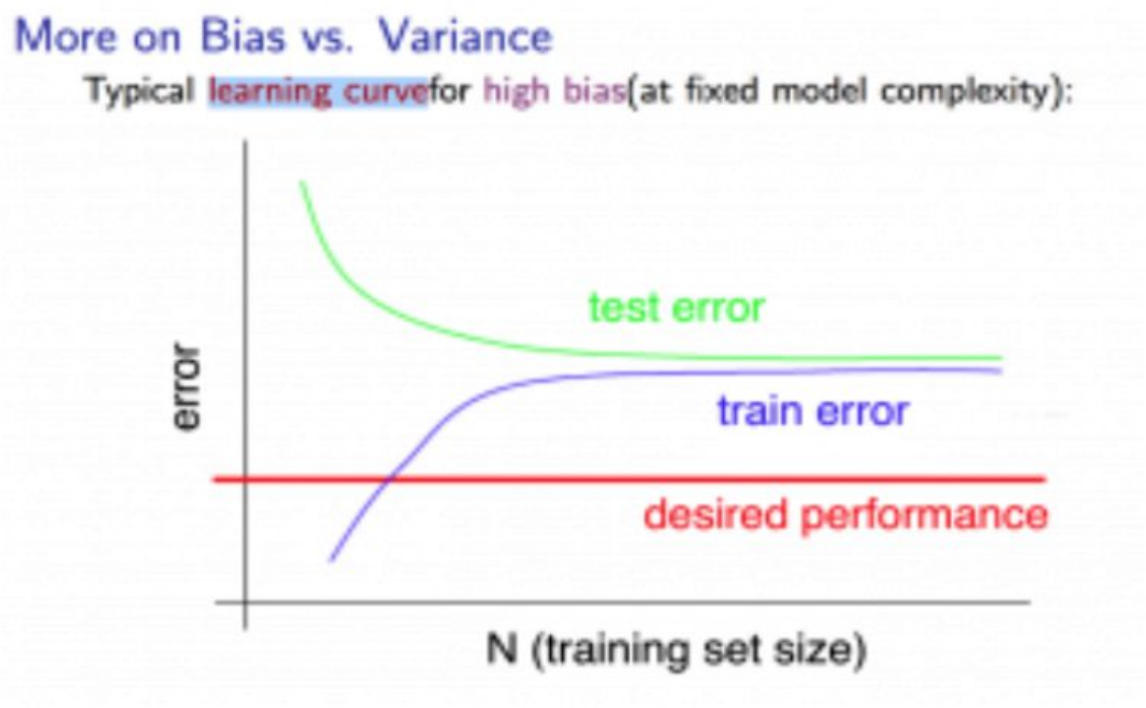
As you have already covered in the previous topic, the increase of bias leads to a decrease in variance and vice-versa. You also learned about the *sweet spot*, which our model should always try to achieve so that it has a good amount of generalization power on unseen data. This relationship between bias and variance is thus called the Bias-variance trade-off.

The ideal model should strive to avoid high bias (underfitting) and high variance (overfitting) and achieve a good balance as shown in the following diagram:

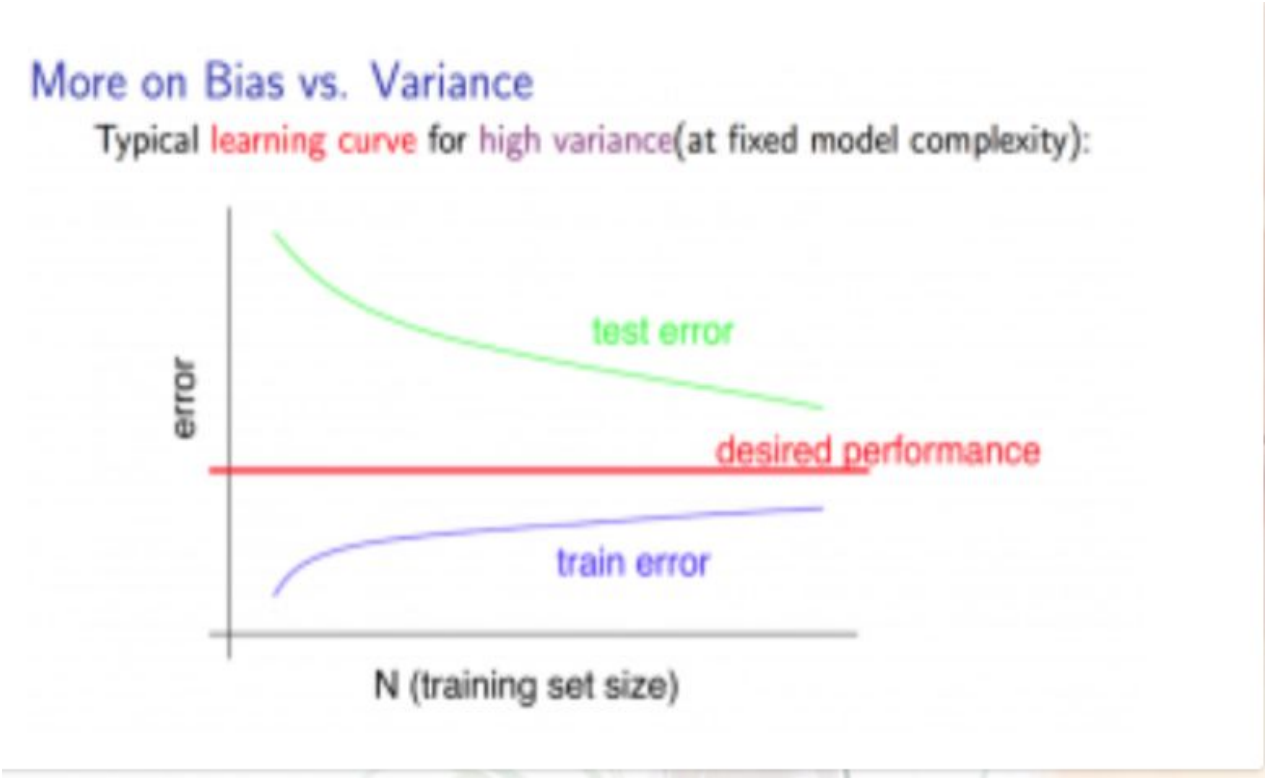


Now let's observe the behavior of two models; one that has a high bias and the other with a high variance on similar model complexity:

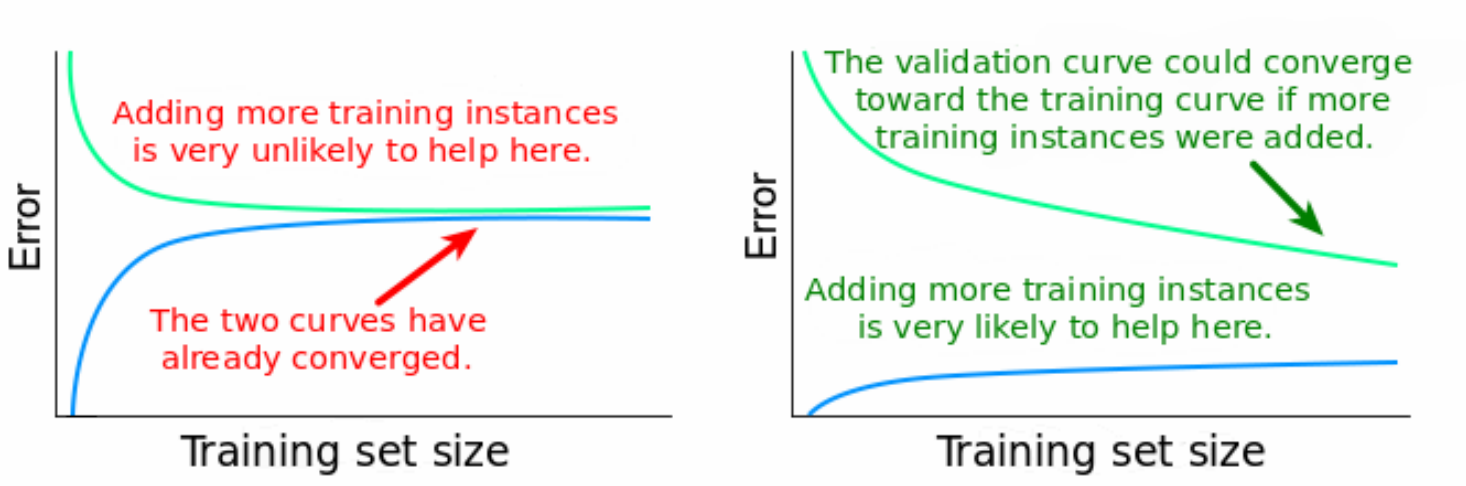
1. Model with high bias: A model with high bias will have a high training set error because the average value of predictions is far off the true values. Now, if we increase the size of the training set, the training error will not decrease too much due to incorrect assumptions of the model. As a result, it will fare poorly during the prediction phase as well. And, obviously, the test set error will be bad since the model hasn't learned properly. It will decrease initially until a point where the decrease becomes almost non-existent.



Model with high variance: Now, a learning algorithm high variance will have a low training set error because it learns almost everything from data points. As we increase the training set size, it will try to incorporate learnings from the new data points and find some general settings, due to which the training set error increases slightly. The new learnings improve the generalization capability of the algorithm, due to which the test set error decreases.



To summarize our discussions so far: Adding more training instances to a high bias model won't help too much but it can help in the case of a high variance model.



Motivation to Use Regularization

Regularization is the method by which you can address the problem of overfitting. To reduce overfitting you can either try reducing the model complexity or increasing the training set size. Regularization achieves exactly that by reducing the model complexity.

Questions:
Adding polynomial features increase variance. True or false?

Ans: TRUE

Explanation:
The more polynomial features you add, the more the model tries to learn the training data perfectly; therefore, and when this model is fit on the *test* data, it performs poorly since it has learned the *training set* so well that it fails to generalize on unseen data points.

This lack of generalization is nothing but an error due to variance.

Regularization

What is it?

Regularization is a technique used to prevent overfitting in datasets. It does so by reducing the magnitude of the coefficients of the attributes. The more the magnitude of the coefficient, the more its predictive power. But a very high magnitude of coefficients means that the learning algorithm has modeled the pattern as well as the noise. This noise can be eliminated by regularization.

How Does Regularization Reduce Noise?

As written above, regularization reduces noise. But how? By simply penalizing the coefficients according to some function along with the loss function. Let's see how.

The cost function for linear regression is as follows:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$$

You penalize your loss function by adding a multiple of the norm of your weights/coefficients vector θ (it is the vector of the learned parameters in your linear regression). You get the following equation:

$$J(\theta) = L(x, y) + \lambda N(\theta_i)$$

In this equation,$L(x, y)$ is the cost function, and $N(w_i)$ is the norm of the coefficients, and λ is a constant called the regularization parameter.

Intuition Behind Regularization

The more the cost function, the more it is penalized, i.e., the more the is the magnitude of the coefficients, the more its function is penalized. In this way, it reduces the importance given to the predictor variables which results in decreased variance. This ensures that the algorithm doesn't learn too much from the training data and generalizes well on unseen data.

Another factor that we have not yet talked about is that the computational resources used up are directly proportional to the number of features. If it becomes too huge, then the calculation could become cumbersome and regularization at that point can be an effective method to counter the issue.

Types of Regularization

Now that you know what regularization is, lets proceed to understanding the variants of regularization. There are three main types and they all vary in the $N(\theta)$function:

1. **Lasso (L1):** It stands for *Least Absolute Shrinkage and Selection Operator* and adds **absolute value of magnitude of coefficient** as a penalty term to the loss function. Mathematically, the new regularized cost function becomes:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\sum_{i=1}^{n}|\theta_i|$$

2. **Ridge (L2):** Ridge regression adds **squared magnitude of coefficient** as a penalty term to the loss function. The new cost function becomes:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\sum_{i=1}^{n}{\theta_i}^2$$

3. **Elastic Net :** It combines both L1 and L2 to overcome the individual challenges and penalizes **both absolute and squared magnitude of coefficient.** The cost function for Elastic net is:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda_1\sum_{i=1}^{n}{\theta_i}^2 + \lambda_2\sum_{i=1}^{n}|\theta_i|$$

Lasso (L1) Regularization

So, Lasso regression penalizes the absolute magnitude of coefficient in the loss function or mathematically

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\sum_{i=1}^{n}|\theta_i|$$

Variable Selection Property of Lasso

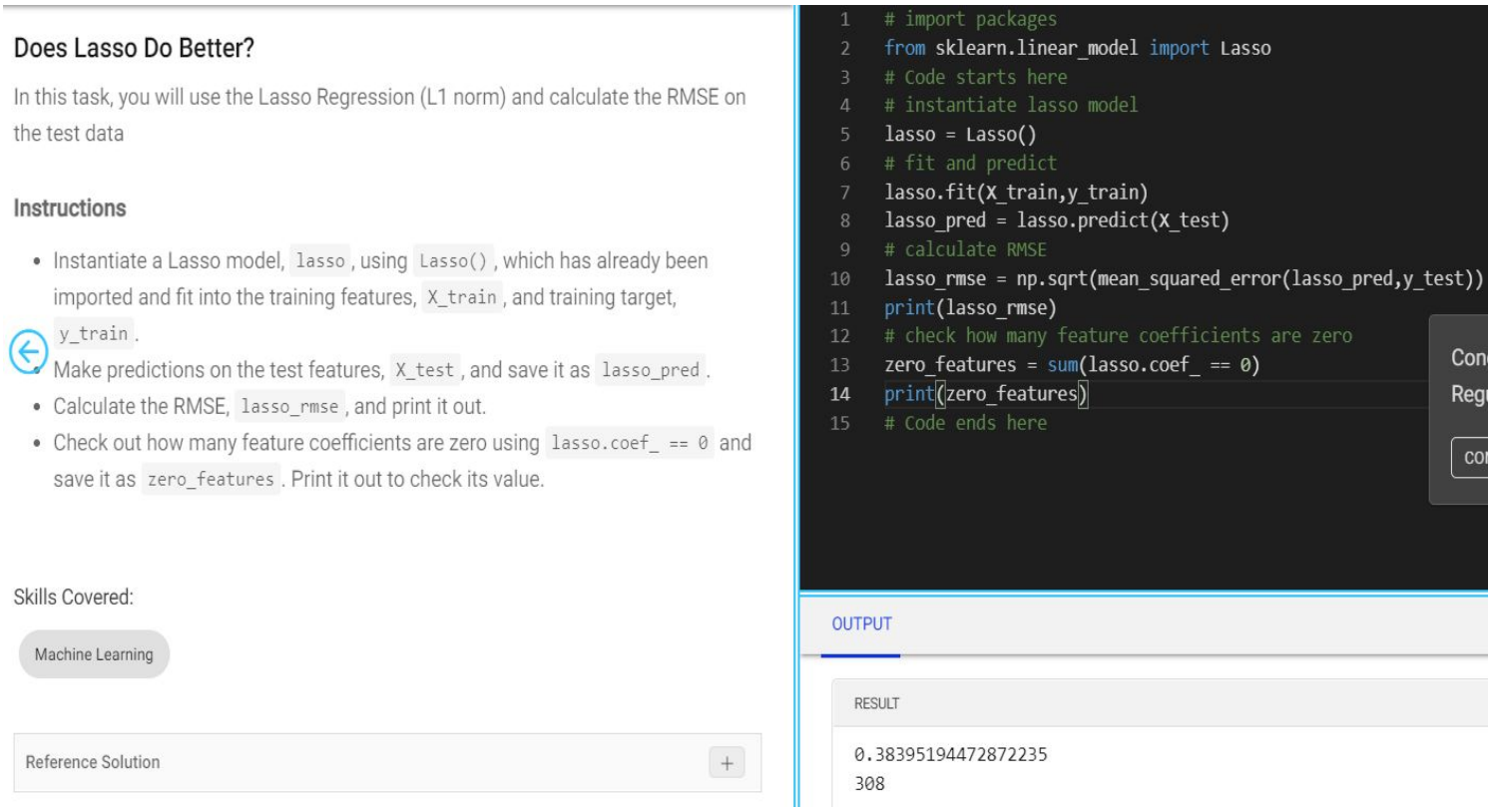
The two keywords in its full form *Least Absolute Shrinkage and Selection Operator* are Absolute and Selection. Here, absolute means penalizing the magnitude of the coefficients of predictors and selection refers to the ability of the L1 method to reduce some of the coefficients to `0` . In this way, the number of features gets reduced and we now have only a subset of features.

This property of selection is particularly important if you have a large number of features at disposal. Multicollinearity and non-linear trends can exist and it's not feasible to check for each and every feature. Lasso provides a way around this conundrum by reducing redundant features and keeping only those features that are relevant. In case of correlated features it arbitrarily selects any one feature among the highly correlated ones and reduces the coefficients of the rest to zero. Also, the chosen variable changes randomly with a change in model parameters.

Role of Regularization Parameter (λ)

Here, λ (lambda) provides a trade-off between balancing the cost function and the magnitude of coefficients.

- Very low values of λ may not do anything.
- $\lambda=0$: Same coefficients as simple linear regression and the same cost function.
- $\lambda=\infty$: All coefficients are zero as there will be a high penalty causing underfitting.
- $0<\lambda<\infty$: Coefficients between `0` and that of a simple linear regression including `0`.



Ridge (L2) Regularization

Ridge penalizes the squared magnitude of coefficient in the cost function. Mathematically,

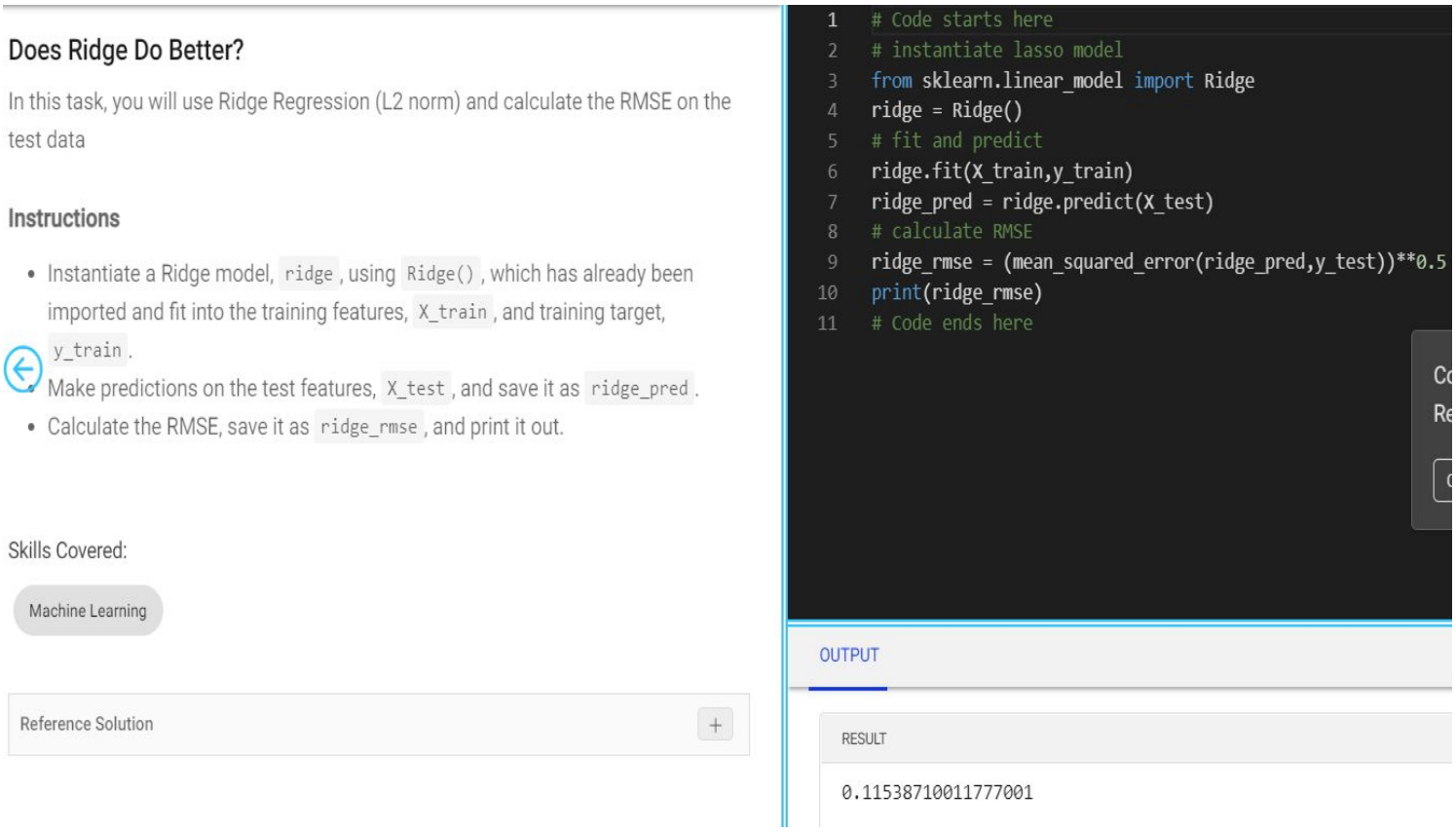
$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\sum_{i=1}^{n}{\theta_i}^2$$

But, unlike Lasso, it doesn't reduce the coefficients to 0. Rather, it only performs shrinkage of all the coefficients. Note that coefficients that are further from zero pull stronger towards zero. This makes it more stable around zero because the regularization changes gradually around zero.

Since the coefficients are squared in the penalty expression, it has a different effect from the L1 norm, namely, it forces the coefficient values to be spread out more equally. For correlated features, it means that they tend to get similar coefficients.

Role of Regularization Parameter (λ)

- $\lambda=0$: The objective becomes the same as the simple linear regression and we get the same coefficients as the simple linear regression.
- $\lambda=\infty$: The coefficients will be zero. Why? Because of the infinite weight on the square of coefficients, anything less than zero will make the objective infinite.
- $0<\lambda<\infty$: The coefficients will be somewhere between `0` and ones for simple linear regression.



L1 vs. L2 Regularization

- **Computational Difficulty**

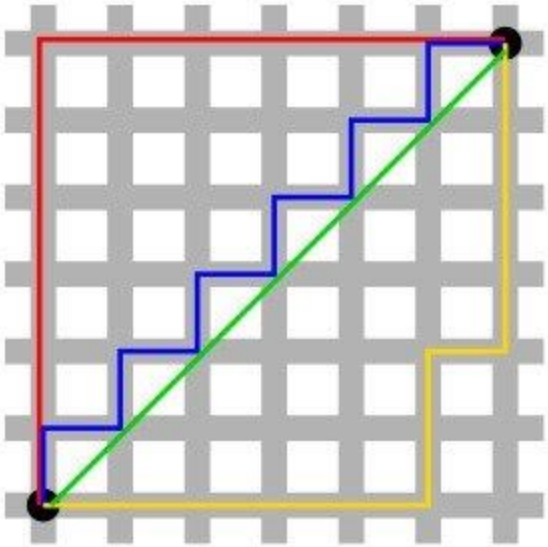
Ridge has a closed-form solution because it's a square of something but Lasso does not because it is a non-differentiable piecewise function that involves an absolute value. For this reason, Lasso is computationally more expensive, as we can't solve it in terms of matrix math, and mostly relies on approximations (in the lasso case, coordinate descent).

- **Sparsity**

Sparsity means that only a very few entries in a matrix (or vector) are non-zero. Lasso has the property of producing many coefficients with zero values or very small values with few large coefficients, and so it often tends to have a great sparsity as compared to Ridge.

- **Solution Numeracy**

As L2 is the Euclidean distance, there is always one right answer as to how to get between two points. While L1 is Manhattan distance (taxicab distance), there are as many solutions to getting between two points as there are ways of driving between two points in Manhattan! This is best illustrated by the following image:



The green line represents L2 distance while the blue, yellow, and red lines represent the L1 distance.

- **Built-in Feature Selection**

Lasso has the ability to perform feature selection by eliminating redundant features, but Ridge does not. This is actually a result of the L1-norm, which tends to produce a sparse coefficient. Suppose the model has 100 coefficients but only 10 of them have non-zero coefficients, this is effectively saying that “the other 90 predictors are useless in predicting the target values”. The L2-norm produces non-sparse coefficients, so it does not have this property.

1.	Which type of regularization can help in feature selection?	✓	Lasso	Lasso

Explanation:

Both Lasso and Ridge shrink the coefficient estimates to 0.

However the *l1* penalty in Lasso which is => $\lambda \sum_{i=1}^{N} |\beta_i|$ forces some of the coefficient estimates to be exactly equal to zero when the tuning parameter λ is sufficiently large.

This can help in variable selection as some variables are eliminated.

In Ridge, the coefficients will be as low as possible but will not become zero. This results in a situation where features are never eliminated or conversely selected.

2.	Which type of regularization penalizes the square of the feature coefficients?	✓	Ridge	Ridge

Explanation:

The *l2* penalty in ridge regression is given by -

$$\lambda \sum_{i=1}^{N} (\beta_i)^2$$

The above penalty penalizes the square of the feature coefficients.

HyperParameters

Before understanding about cross-validation and hyperparameter tuning and why do you need to do it in the first place, let's understand what is a hyperparameter and how does it differ from a parameter.

What is a Parameter?

Imagine you have built a linear regression model that has feature coefficients $\theta 1, \theta_2, ..., \theta_n$. This model assumes that the relationship between predictors and the target variable is linear. The θ_i variable is a weight vector that represents the normal vector for the line; it specifies the slope of the line. This is known as a model parameter, which is being learned during the training phase. So, any property of the training data that is being learned by the learning algorithm in the training phase is a parameter. In our example, the coefficients are the parameters as they are found after minimizing the total training error.

What is a Hyperparameter?

There is another set of parameters whose values must be specified outside of the training procedure; these are known as hyperparameters. Remember the learning rate in gradient descent! The value of the learning rate had to be specified before we estimate the value of θ (which is a parameter). While linear regression itself does not have any hyperparameters, variants of linear regression like ridge regression and lasso both add a regularization term to linear regression; the penalizing coefficient for the regularization term is the hyperparameter. Hyperparameters can be thought of as knobs with which we can control the performance of our learning algorithm. Remember that they are specified externally and cannot be learned by the model.

Key Differences

- Parameters are estimated by the learning algorithm during the training phase whereas hyperparameters are not.
- Hyperparameter values can be manually provided while parameter values cannot be provided
- The number of parameters can sometimes be impossible to determine (in case of non-parametric models) but the number of hyperparameters is fixed

Importance of cross-validation and hyperparameter tuning

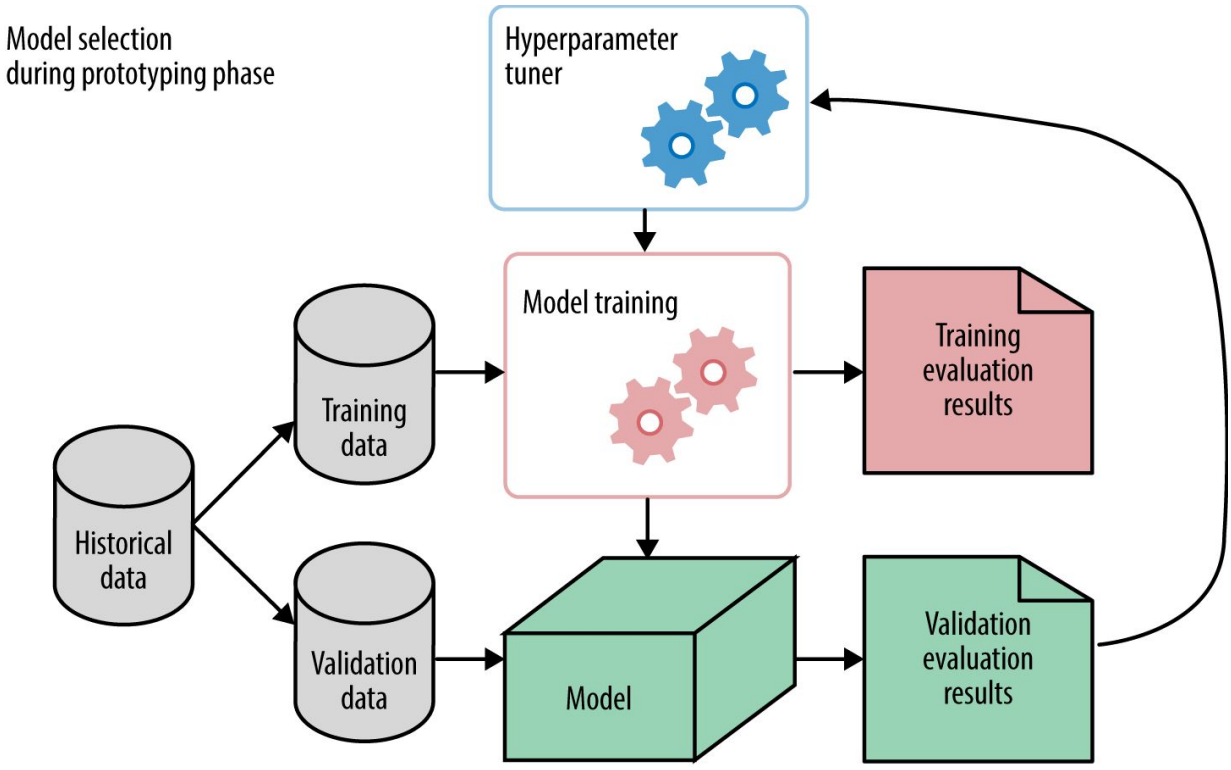
Let's say you have built two different models, A and B. The next step, logically, should be to validate them on unseen data in order to determine their stability. You need to make sure that they generalize well on unseen data and capture only relevant patterns and ignore the noise. At this point, you will face two key issues:

- How to select the best model?
- Once the model is selected, how to find its best set of hyperparameters?

Naive Approach

You may be tempted to use the entire training data to train your learning algorithm and cross your fingers that it performs well on unseen data and then find the right set of hyperparameters. Well, this is a naive approach because it may lead to overfitting.

How to Avoid It?



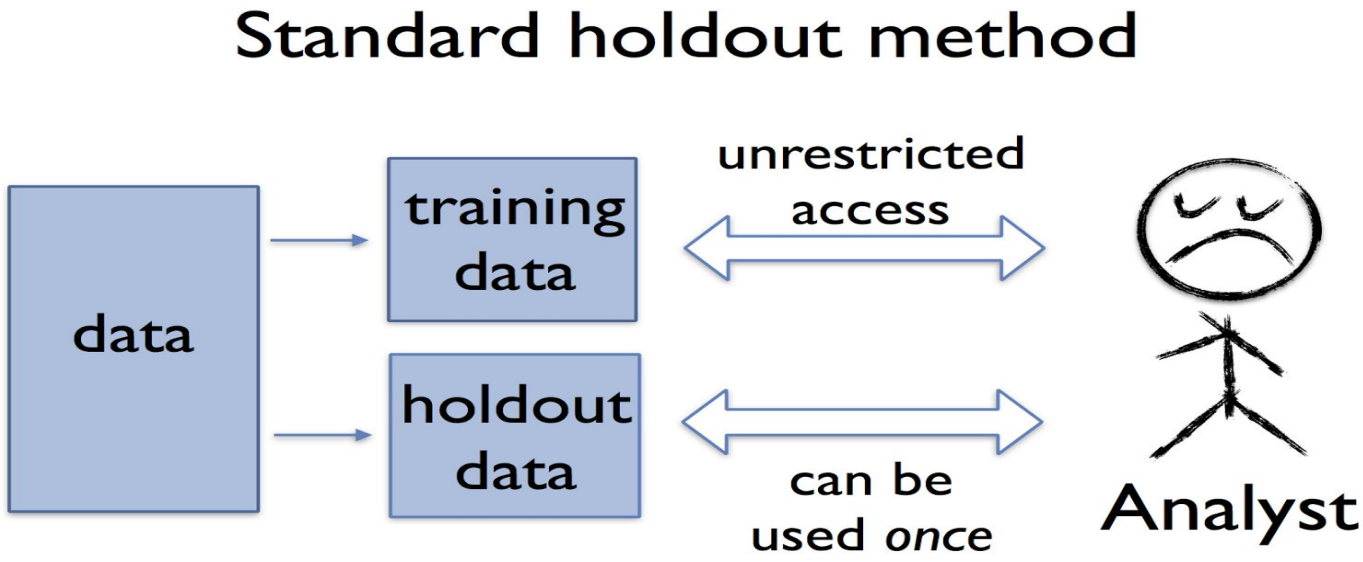
In the above figure, the historical data (training data) is divided into two halves, namely, training and validation data. Note that here hyperparameter tuning is illustrated as a meta process that controls the training process. In the coming topics, you will learn how to get the best set of hyperparameters. Then, the model training

process receives the training data and produces a model, which is evaluated on validation data. The results from validation are passed back to the hyperparameter tuner, which tweaks some knobs and trains the model again.

In this way, you can ensure that your unseen data which is generally assumed to be independent of training data gives us an estimate of the generalization error and takes a call on the learning ability of the algorithm.

Selecting the validation set - Holdout Method

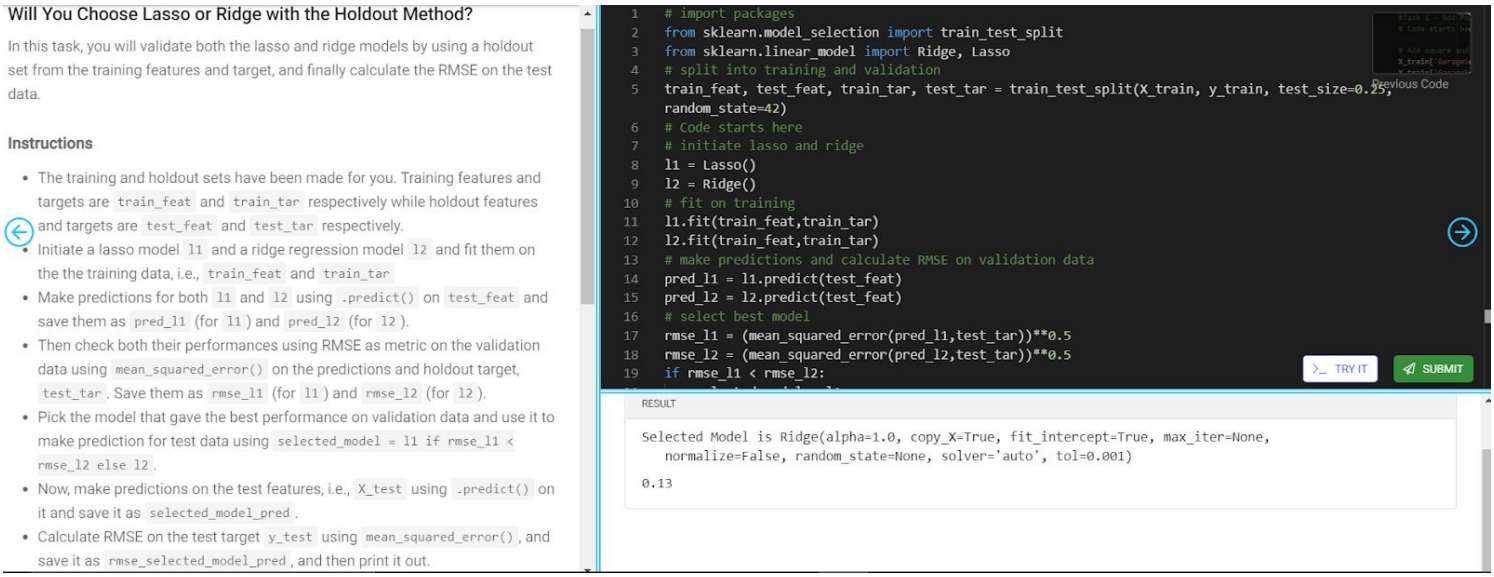
Let's discuss the holdout validation strategy, which will help you chose the best model.



The most simple strategy involves removing a part of the training data and using it to get predictions from the model trained on the rest of the data. Then, the error will tell you how the model is doing on unseen data which is the validation set in this scenario. It is also known as the holdout method.

This is because it is not certain which data points will end up in the validation set and the result may be entirely different for different sets. If you happen to get an unfortunate split, then you may end up rejecting that model and choosing a poor one, even though the former one has a better generalization power. Another drawback is that with sparse datasets, you may not even have the flexibility to set aside a portion of the training data for validation purposes. Its only advantage is that it is computationally much cheaper than other validation strategies.

You can overcome the drawbacks of the holdout method by using cross-validation strategies.

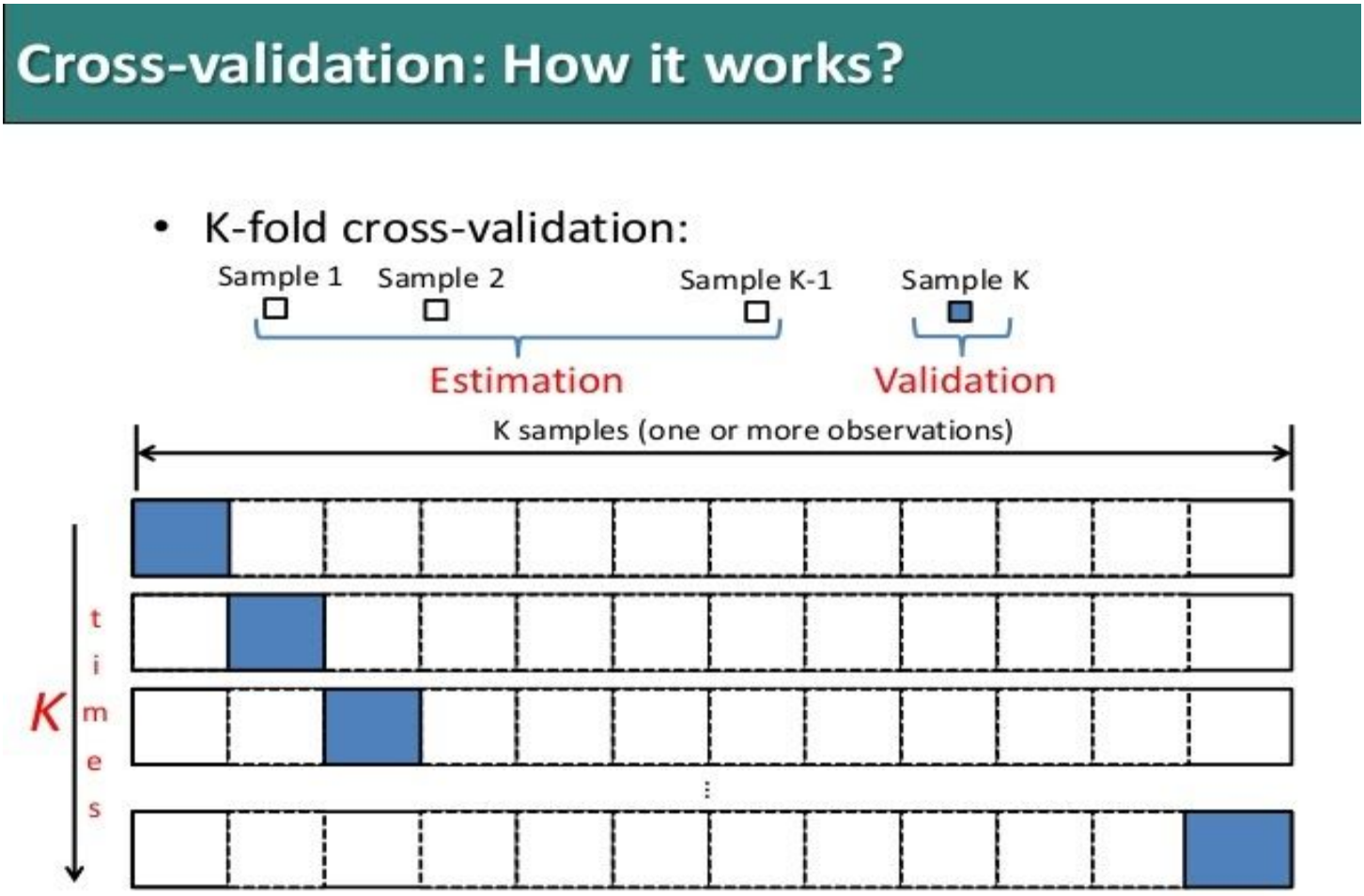


Code:

```
# import packages
from sklearn.model_selection import train_test_split

# split into training and validation
train_feat, test_feat, train_tar, test_tar = train_test_split(X_train, y_train,
test_size=0.25, random_state=42)

# Code starts here
# initiate lasso and ridge
l1 = Lasso()
l2 = Ridge()

# fit on training
l1.fit(train_feat, train_tar)
l2.fit(train_feat, train_tar)
# make predictions and calculate RMSE on validation data
pred_l1 = l1.predict(test_feat)
pred_l2 = l2.predict(test_feat)

rmse_l1 = np.sqrt(mean_squared_error(pred_l1, test_tar))
rmse_l2 = np.sqrt(mean_squared_error(pred_l2, test_tar))

# select best model
selected_model = l1 if rmse_l1 < rmse_l2 else l2
print(selected_model)

# make prediction on test data and calculate RMSE
selected_model_pred = selected_model.predict(X_test)
rmse_selected_model_pred = np.sqrt(mean_squared_error(selected_model_pred, y_test))
print(rmse_selected_model_pred)
```

Cross-Validation Strategies

Cross-validation is simply a way of generating training and validation sets. There is never enough data to train your model, removing a part of it for validation poses a problem of underfitting while selecting all data for training can cause overfitting. Therefore, you require an appropriate method that provides ample data for training the model and also leaves ample data for validation. Cross-validation does exactly that.



There are many variants of cross-validation and the most widely used is K-fold cross-validation. The image above demonstrates a K-fold strategy, where we first divide the training dataset into K folds.

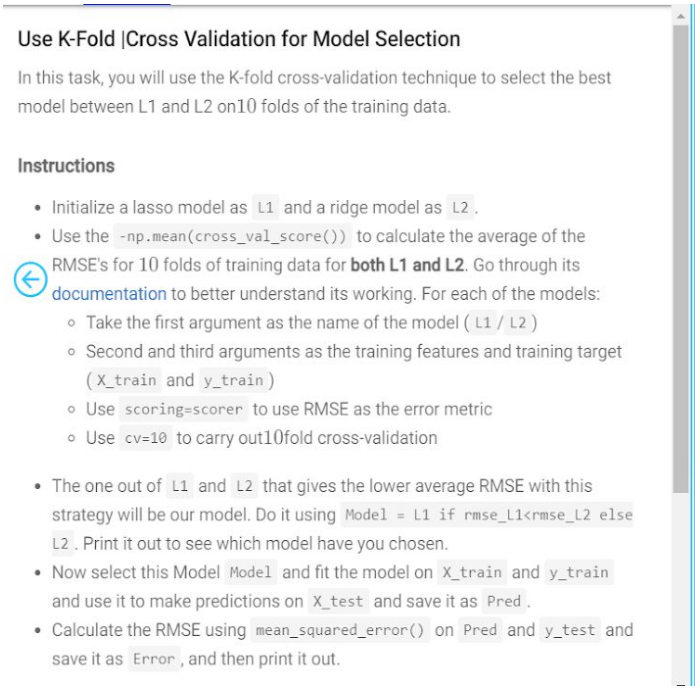
For a given hyperparameter setting, each of the K folds takes turns being the holdout validation set; a model is trained on the rest of the K – 1 folds and measured on the held-out fold. The overall performance is taken to be the average of the performance on all K folds. This procedure is then repeated for all of the hyperparameter settings that need to be evaluated.

A special variant of the K-fold cross-validation is leave-p-out cross-validation. This is essentially the same as k-fold cross-validation, where p is equal to the total number of data points in the dataset. If p is equal to `1`, then this method is referred to as leave one out cross validation.

Documentation link:
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

Code:

```
# import packages
from sklearn.metrics import make_scorer
from sklearn.model_selection import cross_val_score
scorer = make_scorer(mean_squared_error, greater_is_better = False)
# Code starts here
# instantiate L1 and L2
L1 = Lasso()
L2 = Ridge()
# cross validation with L1
rmse_L1 = -np.mean(cross_val_score(L1, X_train, y_train, scoring=scorer, cv=10))
# cross validation with L2
rmse_L2 = -np.mean(cross_val_score(L2, X_train, y_train, scoring=scorer, cv=10))
print(rmse_L1, rmse_L2)
# select best model
Model = L1 if rmse_L1<rmse_L2 else L2
print(Model)
# calculate RMSE on test data
Model.fit(X_train, y_train)
Pred = Model.predict(X_test)
Error = np.sqrt(mean_squared_error(Pred, y_test))
print(Error,rmse_L1,rmse_L2)
# Code ends here
```

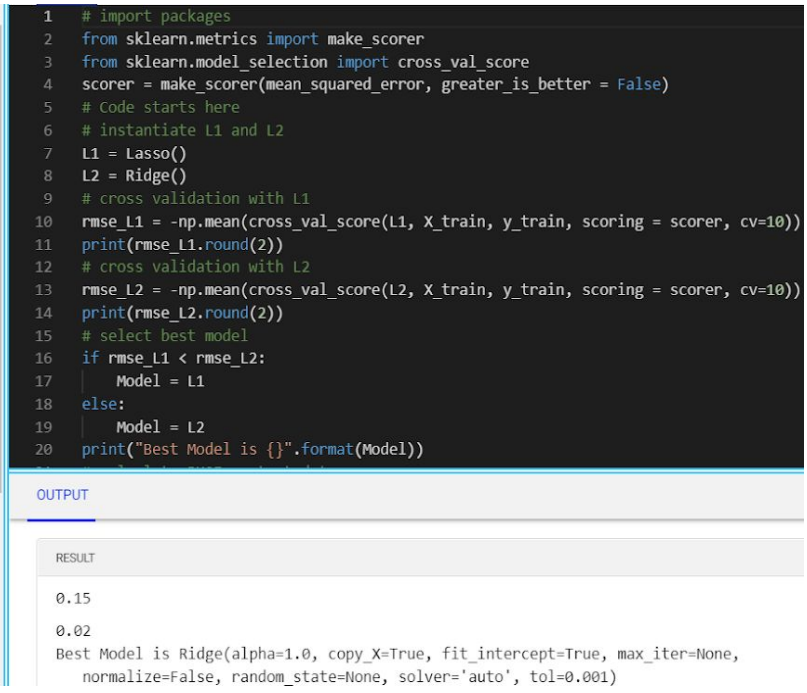


Use K-Fold |Cross Validation for Model Selection

In this task, you will use the K-fold cross-validation technique to select the best model between L1 and L2 on10 folds of the training data.

Instructions

- Initialize a lasso model as `L1` and a ridge model as `L2`.
- Use the `-np.mean(cross_val_score())` to calculate the average of the RMSE's for 10 folds of training data for **both L1 and L2**. Go through its documentation to better understand its working. For each of the models:
  - Take the first argument as the name of the model (`L1` / `L2`)
  - Second and third arguments as the training features and training target (`X_train` and `y_train`)
  - Use `scoring=scorer` to use RMSE as the error metric
  - Use `cv=10` to carry out10fold cross-validation
- The one out of `L1` and `L2` that gives the lower average RMSE with this strategy will be our model. Do it using `Model = L1 if rmse_L1<rmse_L2 else L2`. Print it out to see which model have you chosen.
- Now select this Model `Model` and fit the model on `X_train` and `y_train` and use it to make predictions on `X_test` and save it as `Pred`.
- Calculate the RMSE using `mean_squared_error()` on `Pred` and `y_test` and save it as `Error`, and then print it out.



```
# import packages
from sklearn.metrics import make_scorer
from sklearn.model_selection import cross_val_score
scorer = make_scorer(mean_squared_error, greater_is_better = False)
# Code starts here
# instantiate L1 and L2
L1 = Lasso()
L2 = Ridge()
# cross validation with L1
rmse_L1 = -np.mean(cross_val_score(L1, X_train, y_train, scoring = scorer, cv=10))
print(rmse_L1.round(2))
# cross validation with L2
rmse_L2 = -np.mean(cross_val_score(L2, X_train, y_train, scoring = scorer, cv=10))
print(rmse_L2.round(2))
# select best model
if rmse_L1 < rmse_L2:
    Model = L1
else:
    Model = L2
print("Best Model is {}".format(Model))
```

OUTPUT

RESULT

```
0.15
0.02
Best Model is Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

Hyperparamter Tuning

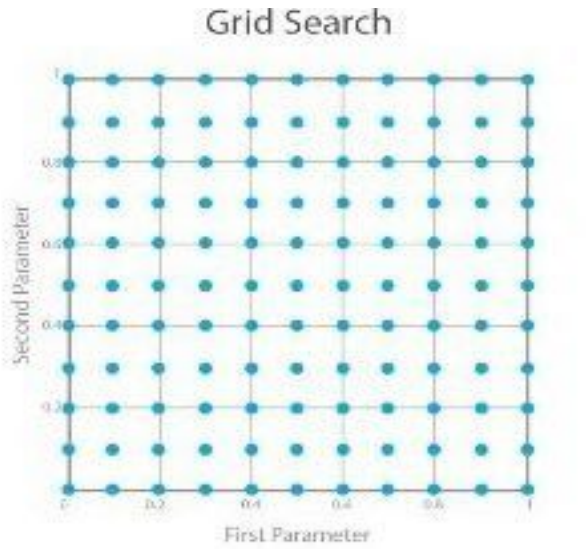
What is It?

After model selection comes the part of selecting the best possible set of hyperparameters. This is the process of hyperparameter tuning where we find the combination of hyperparameter values for a machine learning model that performs the best on a validation dataset. Hyperparameters can be thought of as model settings that need to be tuned for each problem because the best model hyperparameters for one particular dataset will not be the best across all datasets.

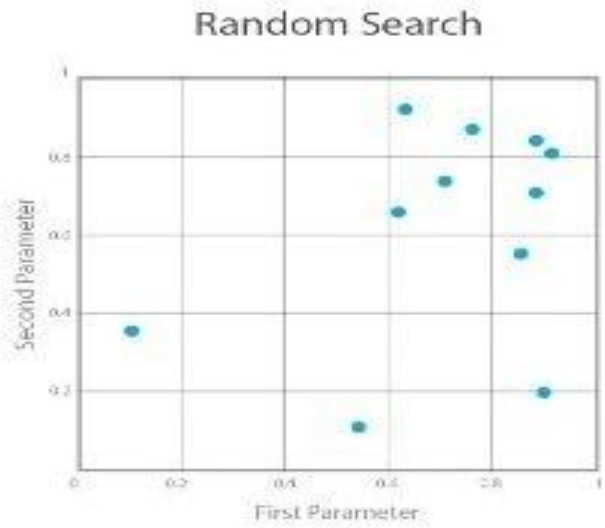
Mechanism of Tuning Hyperparameters

Let's take an example where you have selected an L1 model and you want to find out the best value of λ. You can select the values of λ according to the following ways:

- **Grid search:** As the name itself suggests, it evaluates every possible combination of hyperparameters from a grid and picks up the best combination. In our example, it means we will select a bunch of values for λ, i.e., $\lambda_1, \lambda_2, \lambda_3$, ..., λ_n, and evaluate our model on the validation set for each of these values and later return the best λ.
  Grid search is dead simple to set up but is the most expensive method in terms of total computation time.



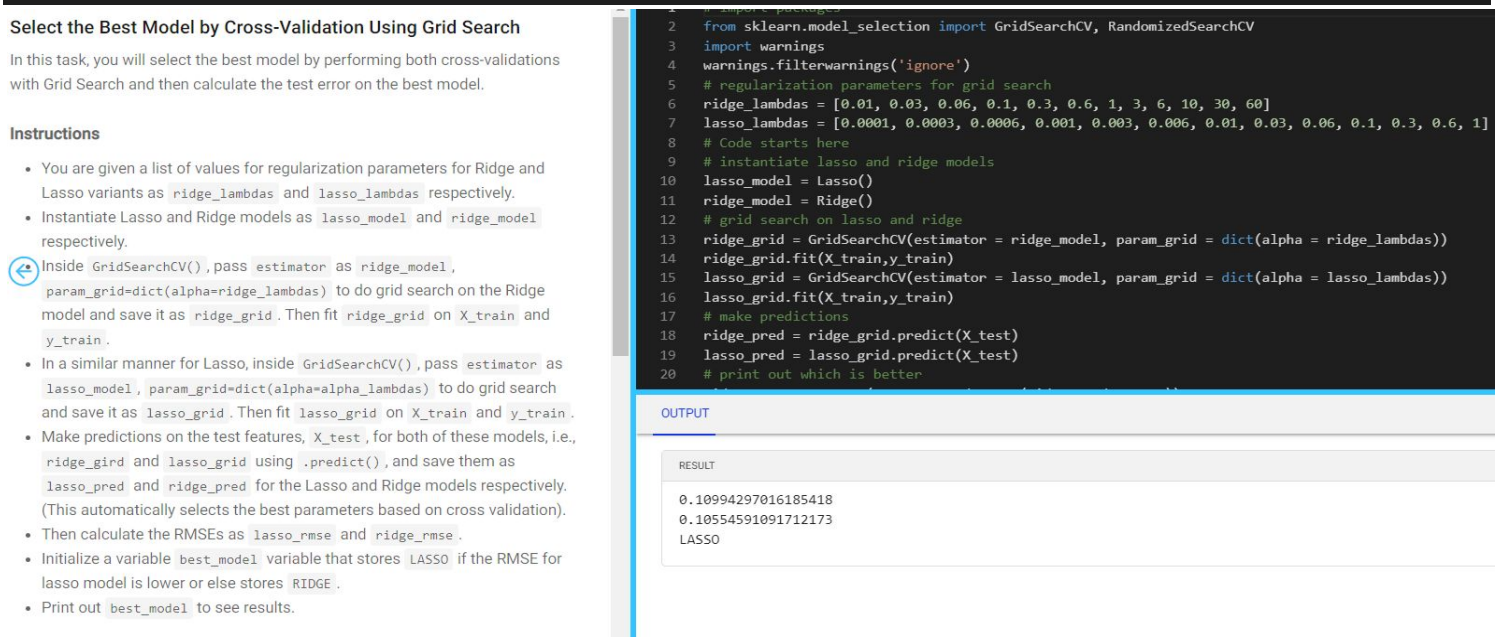
- **Random search:** Random search is a slight variation on grid search. Instead of searching over the entire grid, it only evaluates a random sample of points on the grid. This makes it a lot cheaper than a grid search. With our example of finding the best λ, we can simply define a range $[\lambda_1, \lambda_2]$ and leave it to a random search where it will find randomly select values for λ within this range and return the one with the best performance on the validation set.



Code:

```
# import packages

from sklearn.model_selection import GridSearchCV, RandomizedSearchCV

import warnings

warnings.filterwarnings('ignore')

# regularization parameters for grid search

ridge_lambdas = [0.01, 0.03, 0.06, 0.1, 0.3, 0.6, 1, 3, 6, 10, 30, 60]

lasso_lambdas = [0.0001, 0.0003, 0.0006, 0.001, 0.003, 0.006, 0.01, 0.03, 0.06, 0.1,
0.3, 0.6, 1]

# Code starts here

# instantiate lasso and ridge models

ridge_model = Ridge()

lasso_model = Lasso()

# grid search on lasso and ridge

ridge_grid = GridSearchCV(estimator=ridge_model, param_grid=dict(alpha=ridge_lambdas))

ridge_grid.fit(X_train, y_train)

lasso_grid = GridSearchCV(estimator=lasso_model, param_grid=dict(alpha=lasso_lambdas))

lasso_grid.fit(X_train, y_train)

# make predictions
```

```
ridge_pred = ridge_grid.predict(X_test)

ridge_rmse = np.sqrt(mean_squared_error(ridge_pred, y_test))

lasso_pred = lasso_grid.predict(X_test)

lasso_rmse = np.sqrt(mean_squared_error(lasso_pred, y_test))

# print out which is better

best_model = "LASSO" if lasso_rmse < ridge_rmse else "RIDGE"

print(best_model)

# Code ends here
```



Select the Best Model by Cross-Validation Using Grid Search

In this task, you will select the best model by performing both cross-validations with Grid Search and then calculate the test error on the best model.

Instructions

- You are given a list of values for regularization parameters for Ridge and Lasso variants as `ridge_lambdas` and `lasso_lambdas` respectively.
- Instantiate Lasso and Ridge models as `lasso_model` and `ridge_model` respectively.
- Inside `GridSearchCV()`, pass `estimator` as `ridge_model`, `param_grid=dict(alpha=ridge_lambdas)` to do grid search on the Ridge model and save it as `ridge_grid`. Then fit `ridge_grid` on `X_train` and `y_train`.
- In a similar manner for Lasso, inside `GridSearchCV()`, pass `estimator` as `lasso_model`, `param_grid=dict(alpha=alpha_lambdas)` to do grid search and save it as `lasso_grid`. Then fit `lasso_grid` on `X_train` and `y_train`.
- Make predictions on the test features, `X_test`, for both of these models, i.e., `ridge_gird` and `lasso_grid` using `.predict()`, and save them as `lasso_pred` and `ridge_pred` for the Lasso and Ridge models respectively. (This automatically selects the best parameters based on cross validation).
- Then calculate the RMSEs as `lasso_rmse` and `ridge_rmse`.
- Initialize a variable `best_model` variable that stores `LASSO` if the RMSE for lasso model is lower or else stores `RIDGE`.
- Print out `best_model` to see results.

```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
import warnings
warnings.filterwarnings('ignore')
# regularization parameters for grid search
ridge_lambdas = [0.01, 0.03, 0.06, 0.1, 0.3, 0.6, 1, 3, 6, 10, 30, 60]
lasso_lambdas = [0.0001, 0.0003, 0.0006, 0.001, 0.003, 0.006, 0.01, 0.03, 0.06, 0.1, 0.3, 0.6, 1]
# Code starts here
# instantiate lasso and ridge models
lasso_model = Lasso()
ridge_model = Ridge()
# grid search on lasso and ridge
ridge_grid = GridSearchCV(estimator = ridge_model, param_grid = dict(alpha = ridge_lambdas))
ridge_grid.fit(X_train,y_train)
lasso_grid = GridSearchCV(estimator = lasso_model, param_grid = dict(alpha = lasso_lambdas))
lasso_grid.fit(X_train,y_train)
# make predictions
ridge_pred = ridge_grid.predict(X_test)
lasso_pred = lasso_grid.predict(X_test)
# print out which is better
```

OUTPUT

RESULT

```
0.10994297016185418
0.10554591091712173
LASSO
```

1. **In leave one out cross validation, how many samples are there in the validation set?**

Ans:1

Explanation:

As the name suggests leave-one-out keeps only `one` sample in the validation set and uses all the other samples to train the model.

2. **`Model Parameters` are learned during the training phase of an algorithm. True or false?**

Ans: TRUE

Explanation:

A `model parameter` is a configuration variable that is internal to the model, and its value is estimated from the data. They are often not set manually by the practitioner.

Not to be confused with a `model hyperparameter` which is a configuration that is external to the model, and one whose value cannot be estimated from data. These have to be set by the practitioners manually.