In this chapter we will build our first machine learning model formally , we will also learn about things like bias-variance trade-off and hyper parameter tuning which is going to be relevant for other algorithms down the line. We will start with a regression problem and later will touch upon a classification problem.

# Simple Linear Regression

We saw the general model for regression as $\widehat{y_i} = f(X_i)$, in case of linear regression, $f(X_i)$ is a linear combination of features in the data :

$$\hat{y_i} = f(X_i) = w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip}$$

$x_1, x_2, \cdots, x_p$ are features in the data and $w_0, w_1, \cdots, w_p$ are associated weights for the features in the linear combination. This can also be written in matrix format like this :

$$\begin{aligned}
&\widehat{Y} = XW \\
&where \\
&Y = \begin{bmatrix} \widehat{y_1} \\ \widehat{y_2} \\ . \\ . \\ \widehat{y_n} \end{bmatrix} \\
&and \\
&X = \begin{bmatrix} 1 & x_{11} & x_{12} & . & . & x_{1p} \\ 1 & x_{12} & x_{22} & . & . & x_{2p} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 1 & x_{1n} & x_{n2} & . & . & x_{np} \end{bmatrix} \\
&and \\
&W = \begin{bmatrix} w_0 \\ w_1 \\ . \\ . \\ w_p \end{bmatrix}
\end{aligned}$$

https://github.com/lalitsachan/text-courses

cost function for regression was as mentioned below in the beginning and then we rewrite these using the matrices shown here [you will recognize these many of these discussion done earlier in parts and pieces].

$$\begin{aligned}
&\widehat{y_i} = w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip} \\
&e_i = y_i - \widehat{y_i} \\
&C = \sum_{i=1}^{n} {e_i}^2
\end{aligned}$$

using the matrices mentioned above, we can rewrite the above as follows .

$$\widehat{Y} = \begin{bmatrix} \widehat{y_1} \\ \widehat{y_2} \\ \vdots \\ \widehat{y_n} \end{bmatrix} = \begin{bmatrix} w_0 + w_1 x_{11} + w_2 x_{12} + \cdots + w_p x_{1p} \\ w_0 + w_1 x_{21} + w_2 x_{22} + \cdots + w_p x_{2p} \\ \vdots \\ w_0 + w_1 x_{n1} + w_2 x_{n2} + \cdots + w_p x_{np} \end{bmatrix}$$

$$\widehat{Y} = \begin{bmatrix} 1 & x_{11} & x_{12} & . & . & x_{1p} \\ 1 & x_{21} & x_{22} & . & . & x_{2p} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 1 & x_{n1} & x_{n2} & . & . & x_{np} \end{bmatrix} * \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_p \end{bmatrix} = XW$$

$$e = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_p \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{bmatrix} - \begin{bmatrix} \widehat{y_1} \\ \widehat{y_2} \\ \vdots \\ \widehat{y_n} \end{bmatrix} = Y - XW$$

$$C = \sum_{i=1}^{n} {e_i}^2 = [e_1 \quad e_2 \quad \cdots \quad e_p] * \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_p \end{bmatrix} = e^T e$$

$$C = (Y - XW)^T (Y - XW)$$

https://github.com/lalitsachan/text-courses

we can optimize this cost in two ways.Using linear algebra we can find a closed form solution for optimal $W$, or we can use gradient descent to find optimal values of $W$. For both the scenarios we are going to need the gradient . We have already derived gradient of this term towards the end of chapter on calculus for ML . we will use the result here :

$$\nabla C = 2(X^T XW - Y^T X)$$

equating the above to zero and solving for $W$ gives us :

$$W = (X^T X)^{-1} Y^T X$$

this is the closed form solution we will use , we will also be using gradient descent with the gradient formula mentioned here to arrive at the optimal values of $W$ as another way to solve the problem.
outline

The dataset that we are going to work on comes from a peer to peer lending firm . The usual process at the firm is that people apply for a loan and then firm presents their application to the peers who collectively decide what fraction of the loan will be funded and at what interest rate. Firm is looking to streamline this process by proposing an indicative interest rate to the peers for each application, they want this proposed interest rate to be similar to what peers would have come up with on their own. To do that they want to model interest rate as function of the loan application received . We have been given historical data for the same. One caveat there , which should be a lesson for you for other business problems too. Usually the historical data that you receive for modeling , also contains features which might not be actually available at the process point where the eventual model will be deployed . For example in our data we have been given `Amount Funded By Investors` as a column, clearly this information will not be available before the peers have had a chance to look at the application, this certainly will not be part of the loan application. You need to be cautious and make sure that you do not use such information in the modeling process.

We have processed this data before in our data processing chapter [Referring to Book: Python for ML], we will use the same , I have also included the same in the python notebook for this chapter , you can follow along there.

Post data prep we have `x_train` containing our processed features and `y_train` with our target values. As mentioned earlier we will use couple of ways to estimate $W$ and then compare those with estimates from `sklearn` . Lets get those first .

```
from sklearn.linear_model import LinearRegression
sk_lr=LinearRegression()
sk_lr.fit(x_train,y_train)
w_sk=[sk_lr.intercept_]+list(sk_lr.coef_)
```

`sklearn` outputs $w_0$ to attribute `intercept_` and rest of the weights to `coef_` , we are combining them and storing in the list `w_sk`.We will use `w_sk` as the benchmark that we are trying to arrive at with our own estimation methods . We are doing this cement our understanding of how parameter estimation works, eventually we will be using `sklearn` for most of our tasks.

# Estimating $W$ with Closed Form Solution

Note that `x_train` does not contain the constant `1` column at the beginning which we need in order to use the closed form solution formula that we saw earlier in this chapter . We will create a copy of `x_train` instead of making changes in that directly.

```
x_la=x_train.copy()
x_la.insert(0,'constant',1)
```

Lines of code below calculate $(X^TX)^{-1}$, using symbol `@` and `numpy` functions. Symbol `@` is used for matrix multiplication in python. function `inv` from linear algebra sub-package of numpy `np.linalg` is used for computing inverse of a matrix .

```
x_t_x=x_la.T@x_la
x_t_x_inv=np.linalg.inv(x_t_x)
```

Lines of code below, take this process further , calculate $Y^TX$ and then use the formula $W=(X^TX)^{-1}Y^TX$ to estimate optimal $W$.

```
y_t_x=np.dot(y_train.T,x_la)
w_la=np.dot(x_t_x_inv,y_t_x)
```

You can use the code below to compare this with estimates from sklearn :

```
print(w_la[0],w_wk[0])
list(zip(x_train.columns,w_sk[1:],w_la[1:]))
```

# Estimating $W$ with Stochastic Gradient Descent [SGD]

https://github.com/lalitsachan/text-courses

We did have closed form solution for the weights , but for many cost functions , especially when the model architecture becomes complex [like deep neural network], closed form solution is either too difficult to obtain or it simply does not exist. Thats when we can use gradient descent to arrive at the optimal solution .

Right now here , in the context of linear regression , we are doing this for two reasons, one to show you mechanics of things and second to show you that it works beyond toy data sets.

Gradients how ever are sensitive to scale of the features , so we will standardize the data before using gradient descent to estimate the parameters.

```
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler()
scaler.fit(x_train)
x_sd=pd.DataFrame(scaler.transform(x_train),
                  columns=x_train.columns)
x_sd.insert(0,'constant',1)
```

Following functions implement , calculations of predictions [$XW$], error [$Y - XW$], cost $[e^T e]$ and gradient $[-X^T(Y - XW) = -X^T e]$, $Y, X$ and $W$.

```
def mypred(x,w):

    y_hat=x@w
    return(y_hat)


def myerror(y,x,w):

    y_hat=mypred(x,w)
    errors=y-y_hat
    return(errors)


def mycost(y,x,w):

    errors=myerror(y,x,w)
    cost=errors.T@errors
    return(cost)


def gradient(y,x,w):

    errors=myerror(y,x,w)
    grad=-x.T@errors/x.shape[0]
    return(grad)
```

https://github.com/lalitsachan/text-courses

Note that we have scaled down the gradient by number of obs in the data so that we do not have to keep the learning rate too small to keep the effective change in parameters small as required .

We are going to use stochastic gradient descent with which we discussed in the chapter of optimisation with some ideas which did not discuss there . here is the full code for the function which implements SGD and other functionalities .

```python
def my_lr_sgd(y,x,learning_rate,num_steps):

    weights=np.zeros(x.shape[1])
    decay_rate = 0.9
    prev_cost = float('inf')

    for i in np.arange(num_steps):
        rand_ind=np.random.choice(range(x.shape[0]),100)
        y_sub=y[rand_ind]
        x_sub=x.iloc[rand_ind,:]

        gd=gradient(y_sub,x_sub,weights)
        weights -= learning_rate*gd
        curr_cost = mycost(y, x, weights)

        # Stop if converged
        if np.abs(prev_cost - curr_cost) < 1e-6:
            print(prev_cost,curr_cost)
            print(f"Converged at iteration {i}")
            break

        if i%20000==0:
            learning_rate*=decay_rate

        prev_cost = curr_cost

    return weights
```

Apart from what we have already learned about gradient descent in the optimizers chapters , there are two new things that this functions does

**Using Learning Rate Decay**: After some iterations, we reach close to optimal values, risk for taking a large step and missing the optimal point becomes higher, in order to avoid that we can come up with a way to decay the learning rate and keep the change small as we move closer to optima. In the function above we have used a simple step decay , but there are other strategies like exponential decay or inverse decay or some other type of decay schedule which you can experiment with.

**Stopping Early in Case of Convergence** : Its possible that beyond a certain number of iterations further improvement in cost function is not worth to keep on going any further .

this bit of the code within the function above, implements both the ideas :

```python
if np.abs(prev_cost - curr_cost) < 1e-6:
    print(f"Converged At Iteration {i}")
    break

if i%20000==0:
    learning_rate*=decay_rate
```

In my run the convergence happened around 96000 iterations , this can be different when you run because of random sampling.

https://github.com/lalitsachan/text-courses

```
w_sgd=my_lr_sgd(y_train,x_sd,.01,500000)
```

```
     0 405452.9881012124
 20000 9002.899175585753
 40000 8980.418135388343
 60000 8986.361829555015
 80000 8979.906650441255
Converged at iteration 96593
```

when you compare this with results from closed form solution [shown in the accompanying python notebook], you will find these to be a little far from optimal value that you were able to reach to with closed form solution. You can always do better by using other optimizers that we discussed , relaxing the convergence criterion and let the process go on for longer . I will leave those experiments for you to try out .

Do note that before comparing the estimates you need to realize that we estimated $W$ for non standardized data using closed form solution and here we are estimating $W$ with standardized data so they are not comparable as is. You can use following function to convert weights for standardized data to weights for non-standardized data. This function implements following formulas .

$$w_j = \frac{{w_j}'}{\sigma_j} \quad \forall j \in \{1, 2, \cdots, p\}$$

$$w_0 = {w_0}' - \sum_{j=1}^{p} \frac{\mu_j {w_j}'}{\sigma_j}$$

where ${w_j}'$ are the estimates for standardized data, $\mu_j$ and $\sigma_j$ are mean and standard deviation used to standardize $j^{th}$ feature in the data.

https://github.com/lalitsachan/text-courses

```
def convert_to_non_standardized_weights(scaler, standardized_weights):
    # Extract mean and scale (standard deviation) from the StandardScaler object
    means = scaler.mean_
    scales = scaler.scale_

    # Initialize the array to store the non-standardized weights
    non_standardized_weights = np.zeros_like(standardized_weights)

    # Compute the weights for the non-standardized data
    # w_j_non_standardized = w_j_standardized / scale_j for j > 0
    non_standardized_weights[1:] = standardized_weights[1:] / scales

    # Adjust the intercept term (w_0)
    w0_adjustment = np.sum((means * standardized_weights[1:]) / scales)
    non_standardized_weights[0] = standardized_weights[0] - w0_adjustment

    return non_standardized_weights
```

```
w_sgd=convert_to_non_standardized_weights(scaler,w_sgd)
```

# Bias Variance Trade-Off

Simple linear regression is the first machine learning model that we are trying here. We will slowly venture into building more complex machine learning models. From intro to ML you'd know that eventual goal of building these models is to use them in context of new data and performance there is what matters . An ideal modeling algorithm will have two desirable properties

- It is able to extract good meaningful patterns from training set
- Extracted patterns make good consistent predictions on test/evaluation set

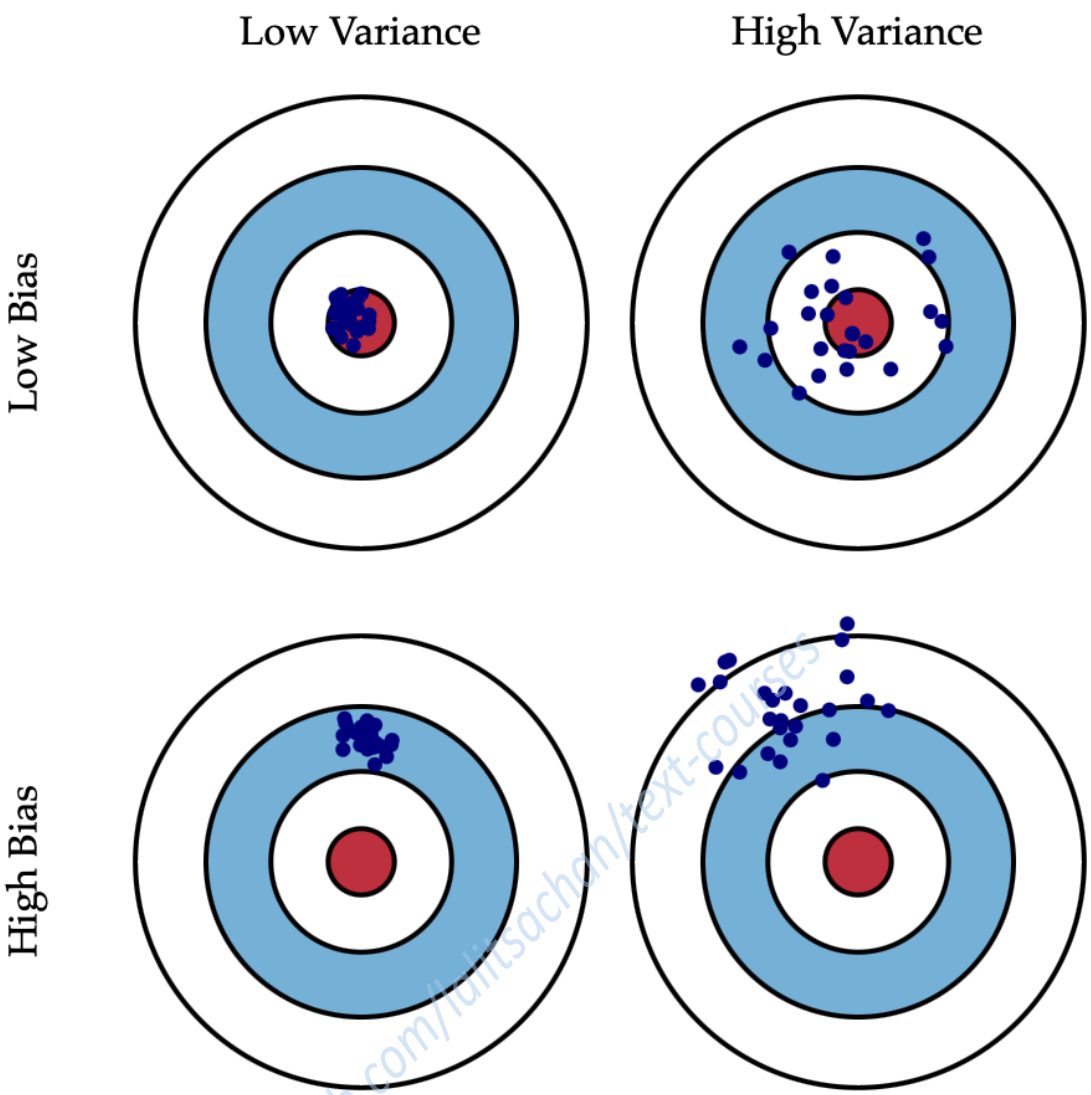
Its not easy to achieve both due to real life data almost always having noisy patterns which do not generalise outside the training data. If a modeling algorithm is too good at extracting all patterns from training data, it will perform very well on training data; but in the process if it has included noisy patterns as well, it will perform not so good and consistent on the test/valuation set. This is called **over-fitting** to the training data or simply over-fitting .

This is often discussed formally in terms of bias and variance . To avoid confusion, know beforehand that both the terms are variances; but of different kinds. Also these are usually discussed in context of modeling algorithms in general, not in context of a specific instance of the modeling algorithm. For example it makes to say that simple linear regression is low bias , high variance modeling algorithm; but it wouldn't really make sense to say that the simple linear regression model that we built for our specific single training data of interest rate has high or low variance .

**Bias** : Bias is a measure of how far off the predictions of a model are from the true value in the data. A low complexity model like linear regression will tend to have high bias as its not able to capture complex non-linear patterns present in the data and hence is not able to make good predictions.

**Variance** : Now imagine a fixed set of observations. Consider that a modeling algorithm is fed many different training sets and after each such training set, it is made to make a prediction for this test set observation . An algorithm which is able to extract complex niche patterns from the training data might naturally be susceptible to capturing noisy patterns, so it might get good predictions on training set, but since the noise will change from one training set to another , the predictions on that test observation will not be consistent , different instances of models from the algorithm will vary a lot across different training sets [for the same business problem], such an algorithm is called a high variance algorithm.[`Variance` measures variance of these predictions within themselves].
In comparison a simple algorithm like simple linear regression; unable to capture niche patterns will tend to have high bias. However its predictions on test set will be more consistent , leading to low variance.

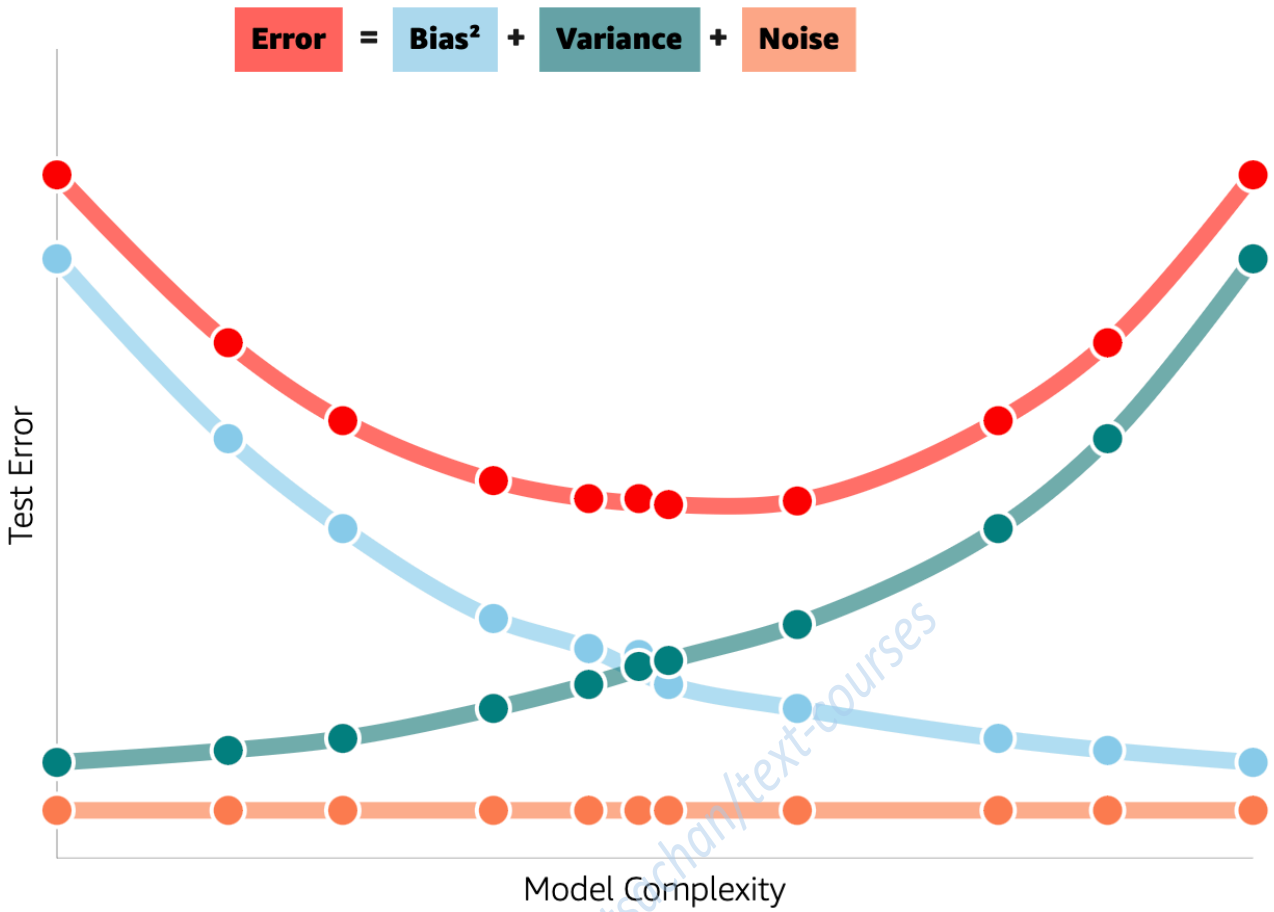
https://github.com/lalitsachan/text-courses




source:[http://scott.fortmann-roe.com/docs/BiasVariance.html](http://scott.fortmann-roe.com/docs/BiasVariance.html)

An ideal modeling algorithm will be the one with low bias and low variance which is like a unicorn . Usual path that we will go through will be something like this : imagine that we started with having no model, we simply took average of the target in the training set and used that as our prediction for new incoming data. Empirically for any real life data, this will have pretty high bias and possibly low variance. From here we go to simple linear regression, which with better pattern extraction capabilities than averaging will have reduced bias than earlier but will lead to possibly increased variance in comparison. This we call **over-fitting**. In our path of learning about machine learning algorithms we will go through cycles of the process where an algorithm improves on the capability of extracting patterns from the training set thus reducing bias but increasing variance and next we attempt to decrease the variance , that part is known as **regularisation**.

https://github.com/lalitsachan/text-courses




source : https://mlu-explain.github.io/bias-variance/

from the source :

> When a model is too simple (i.e. small values along the x-axis), it ignores useful information, and the error is composed mostly of that from bias.

> When a model is too complex (i.e. large values along the x-axis), it memorizes non-general patterns, and the error is composed mostly of that from variance.

> The ideal model aims to minimize both bias and variance. It lays in the sweet spot - not too simple, nor too complex. Achieving such a balance will yield the minimum error.

Each algorithm is going to have its own reasons for susceptibility to noise in the training set thus high variance , regularization methods will also differ as per the reason for over-fitting for each algorithm. We will see next how to regularize linear models . These methods will also be relevant in regularizing other parametric models like neural networks. Some version of this will also show up in regularizing non-parametric models like GBM. But first lets focus on parametric models. Lets see how we regularize simple linear models.

# Regularizing Simple Linear Models

**why do linear models overfit** : Whether we are using closed form solutions or gradient descent to estimate weights, we almost always get non-zero weights ; even if we include a random column in the data. These seemingly random or unrelated columns/features in the data end up having small value for their corresponding $w$ . These weights arising from very weak correlation between target and feature do not generalize very well. Reason being that weak correlation is usually due to noise which does not persist through

https://github.com/lalitsachan/text-courses

other samples of the data or new data. We'll ideally want these small weights to become exactly zero or very close to zero so that their contribution to model becomes negligible thus reducing over-fitting and variance .

How can we force smaller weights to go closer to zero or even exactly zero? We can do that by adding an explicit term to our cost function which penalizes size of $w$s themselves .

# L1 and L2 Penalties on simple linear regression[Lasso and Ridge]

$l_p$ norm of vector $W$ is written as $\sum \left(|w_j|^p\right)^{\frac{1}{p}}$. In practice the penalties added to cost are not exactly the same , but derive from these ideas, hence the name $l_1$ and $l_2$ penalties .

cost with $l_1$ penalty :

$$C = (Y - XW)^T(Y - XW) + \alpha||W||_1$$
$$\text{where}$$
$$||W||_1 = \sum |w_j|$$

cost with $l_2$ penalty :

$$C = (Y - XW)^T(Y - XW) + \alpha W^T W$$
$$\text{where}$$
$$W^T W = \sum {w_j}^2$$

**How do penalties work against weak correlations only** :

In order to understand that you need to realize that total contribution by a feature to the overall cost function is not just the penalty; its summation of the penalty plus sum or squared errors considering its inclusion in the model [consider $l_2$ penalty here, you can reason out about the $l_1$ penalty on your own on similar lines ] :

$$C = \sum_{i=1}^{n}(y_i - \sum_{k=0}^{p} w_k x_{ik})^2 + \alpha \sum_{t=0}^{p} {w_t}^2 \quad where \quad x_{i0} = 1 \quad \forall i$$

$$\text{now consider contribution from } j^{th} \text{ feature}$$

$$C = \sum_{i=1}^{n}\left[(y_i - w_j x_{ij} - \sum_{k\neq j, k=0}^{p} w_k x_{ik})^2\right] + \alpha {w_j}^2 + \alpha \sum_{t\neq j, t=0}^{p} {w_t}^2$$

$$C = \sum_{i=1}^{n}\left[(y_i - w_j x_{ij})^2 + (\sum_{k\neq j, k=0}^{p} w_k x_{ik})^2 - 2(y_i - w_j x_{ij})(\sum_{k\neq j, k=0}^{p} w_k x_{ik})\right] + \alpha {w_j}^2 + \alpha \sum_{t\neq j, t=0}^{p} {w_t}^2$$

$$C_j = \alpha {w_j}^2 + \sum_{i=1}^{n}\left[(y_i - w_j x_{ij})^2 - 2(y_i - w_j x_{ij})(\sum_{k\neq j, k=0}^{p} w_k x_{ik})\right]$$

features which are very well correlated with the target , for them the term $\sum_{i=1}^{n}\left[(y_i - w_j x_{ij})^2 - 2(y_i - w_j x_{ij})(\sum_{k\neq j, k=0}^{p} w_k x_{ik})\right]$ will decrease much more than the amount with which term $\alpha {w_j}^2$ goes up when we increase $w_j$ in absolute value. So using gradient descent increasing absolute value of $w_j$ might still be in the direction of decreasing cost function. Whereas for very weak correlated feature, the first term which relies on how well they are correlated with response doesn't go down as much as $\alpha {w_j}^2$ goes up, so the direction of decreasing cost function which the gradient descent will take leads to $w_j$ for these kind of feature getting suppressed towards zero.

https://github.com/lalitsachan/text-courses

Exact effect of both the penalties on eventual estimates for weights is slightly different from each other , lets discuss them one by one.

Lasso Cost [with $l_1$ penalty], because of absolute term appearing in cost , its not traditionally differentiable in matrix format as is. We need to work with conditional gradients for each parameter individually . Here is how it is defined :

$$\nabla C = -2X^T(Y - XW) + \alpha \cdot \text{sign}(\mathbf{W})$$
$$\text{Where the } sign \text{ function is defined element-wise as :}$$
$$\text{sign}(w_j) = \begin{cases} +1 & \text{if } w_j > 0 \\ -1 & \text{if } w_j < 0 \text{ and } \alpha{>}0 \\ 0 & \text{if } w_j = 0 \end{cases}$$

A model with more than one parameter has little more mathematically complex explanation for what I am going to demonstrate now, but idea and conclusion remains more or less the same. consider a single weight $w$ in the model which is $w > 0$, now in that light if you equate the gradient given above to zero you get :

$$w = (X^TX)^{-1}(X^TY - \alpha)/2$$

this expression tells us that there can exist a $\alpha$ , the extent of penalty such that $w = 0$, implying that $l_1$ penalty can lead to weights being exactly zero thus reducing the model size by eliminating features. However it also requires that $X^TX$ be invertible; which might not be the case if there is some multicollinearity in the data. So in summary , Lasso model can result in model reduction but doesn't work with data with multicollinearity .

Now lets look at linear regression with $l_2$ penalty , also known as Ridge Regression:

$$C = (Y - XW)^T(Y - XW) + \alpha W^TW$$
$$\nabla C = -2X^T(Y - XW) + 2\alpha W = 0$$
$$W = (X^TX + \alpha I)^{-1}X^TY$$

So for $l_2$ penalty here , there exists a closed form solution which makes it far more easier to implement. However note that $\alpha$ here turns in the inverse part of the formula . By increase $\alpha$ you can scale the weights , but you cant make them exactly zero. Which implies that $l_2$ penalty in context of linear model will suppress weights close to 0 for weakly correlated features but it can not make them exactly zero, thus not resulting in model reduction as was the case in Lasso . Another thing to note here is that even if $X^TX$ is not invertible the formula doesn't break down, because we are inverting $(X^TX + \alpha I)$ here; so Ridge regression can work with data with multicollinearity as well . A quick summary table on differences between ridge and lasso:

| | **Ridge $l_2$** | **Lasso $l_1$** |
|---|---|---|
| Closed Form Solution | Yes | No |
| Model Reduction | No | Yes |
| Works Well with Multicollinearity | Yes | No |

I have included functions for both Ridge and Lasso gradients in the accompanying python notebook. You can use them if you want to experiment with implementing your own estimators for your models instead of using sklearn implementation.

**What Value of $\alpha$ to select**

https://github.com/lalitsachan/text-courses

There is no mathematical formula to give you best value of $\alpha$ in general. In context of your data it can be anything in the interval $[0, \infty)$. What you can do is to try out multiple values of $\alpha$, see how the model performance is using cross validation and then finalize the value of alpha where the cross-validated performance is the best. You can implement that grid search with cross-validation across all the values of $\alpha$ that you want to try out on your own if you want to , but thats more of a programming assignment than implementing an ML algorithm. We will skip that detail here and use sklearn's implementation of grid search for the same.

```
from sklearn.linear_model import Ridge,Lasso
from sklearn.model_selection import GridSearchCV
```

we will start with creating a dictionary containing values of $\alpha$ that we want to try out and the model that will receive this values in the internal iterations of grid search.

```
params={'alphas'=np.linspace(1,100,100)}
model=Ridge()
```

`cv=10` indicates `10` fold cross validation, since its a regression problem we use `scoring='neg_mean_absolute_error'`. All values of `alphas` are scored on this with 10 fold cross validation and then ranked according to whatever value received higher score [thats why the use of negative with mean absolute error, as such in regression we want the errors to be as low as possible, with the negative sign it becomes higher the better]

```
gs=GridSearchCV(model,
                param_grid=params,
                cv=10,
                scoring='neg_mean_absolute_error'
                )
gs.fit(x_train,y_train)
```

once the `gs` object gets fitted, we can pass on the extensive dictionary it creates to report function to extract top performing `alpha` values.

```
mt.report(gs.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: -1.604376 (std: 0.118215)
> Parameters: {'alpha': 50.0}
> Model with rank: 2
> Mean validation score: -1.604376 (std: 0.118165)
> Parameters: {'alpha': 49.0}
> Model with rank: 3
> Mean validation score: -1.604380 (std: 0.118267)
> Parameters: {'alpha': 51.0}

You can further fine tune this by trying values of `alpha` around `50` with smaller intervals to get some more improvement.

https://github.com/lalitsachan/text-courses

Accompanying notebook talks more about where to go from here in terms of syntax for Lasso, looking at the estimates , making predictions etc; please follow along there.

Here we will move on to classification with linear models [Logistic Regression]

# Linear Model for Classification [Logistic Regression]

Revising a bit from intro to ML about this will help you in this section. Linear model for classification looks like this

$$p_i = \frac{1}{1 + e^{\left(-w_0 - \sum_{j=1}^{p} w_j x_{ij}\right)}}$$

to estimate $W$s we need to minimise following cost as we discussed in intro to ML.

$$\begin{aligned} C &= -\sum_{i=1}^{n} \left[ y_i log(p_i) + (1 - y_i) log(1 - p_i) \right] \\ C &= -[Y^T log(P) + (I - Y)^T log(1 - P)] \\ C &= -[Y^T XW - log(1 + e^{XW})] \\ \text{where} P &= \frac{1}{1 + e^{-XW}} \\ \nabla C &= -X^T (Y - P) \end{aligned}$$

you can use the gradient above to implement your own version of parameter estimation with gradient descent . I have included all the helper functions you might in the python notebook. Here we will discuss sklearn implementation.

In `sklearn` simple logistic regression is not implemented . Implementation includes penalty by default .

Business problem that we will be solving for this is to predict which revenue category the customer might fall in given their demographic and financial information features . Data prep for the given dataset has already been covered in the data prep chapter and has been included in the python notebook as is. There are three hyper-parameters to tune for `Logistic Regression`:

**penalty**: takes one of the two values : `['l1','l2']`, implying model either uses $l_1$ penalty or $l_2$ .

**C** : this is akin to $\alpha$ in Lasso and Ridge implementation however `it works inversely`. High values of C would imply lower penalty and low values lead to high penalty. So if you do want to do something close to simple logistic regression, you could get their to some extend by simply choosing a very large value for this hyper-parameter .

**class_weight** : It takes two values `[None,'balanced']` lets understand this with an example as to what do these two options entail. Consider that we have 10 observations with following target values : $[0, 0, 0, 0, 1, 0, 1, 0, 0, 0]$ , for this discussion we will denote the classes as $s \in \{0, 1\}$ . Also lets say individual contribution of obs to the total cost is written as $C_i$ . When `class_weight=None` is used , it implies that total cost will be a simple sum of individual contribution towards the cost from all observations

$$\text{for class_weight=None :}$$
$$\text{total cost} = \sum C_i$$

https://github.com/lalitsachan/text-courses

However this might not be the best thing to do when dataset is imbalanced and contains far more occurrences of one class outcome in comparison to another . This will lead to model optimisation and eventual result to be favoring the majority class instead of focusing on separating the two classes which is the eventual goal of the process. To stop this from happening we artificially increase the contribution of the minority class. this is achieved by multiplying cost of the individual observation with an adjustment factor which is calculated as :

$$a_s = \frac{\text{total number of observations}}{\text{number of classes} \times \text{frequency of class } s}$$

so if the target data was like this : $[0, 0, 0, 0, 1, 0, 1, 0, 0, 0]$ and we used `class_weight='balanced'`:

$$a_1 = \frac{10}{2 \times 2} = 2.5$$
$$a_0 = \frac{10}{2 \times 8} = 0.625$$
$$\text{total cost} = a_1 \left( \sum_{k=1,i} C_{ik} \right) + a_0 \left( \sum_{k=0,i} C_{ik} \right)$$

Note that this scales down the contribution from the majority class and inflates the contribution from the minority in proportion and thus makes the total cost being optimized, balanced. Which drives the focus of the optimization process towards separating the two classes .

In the code below we define the hyper-parameter choices that we want to experiment with :

```
params={'class_weight':['balanced',None],
        'penalty':['l1','l2'],
        'C':[.0001,.0005,.001,.005,.01,.05,.1,1,2,5,10]}
model=LogisticRegression(solver='liblinear')
```

In the version of `sklearn` when this document was being written, the default `solver` did not support `l1` penalty, hence the explicit option `solver='liblinear` is being used which does support both the $l_1$ and $l_2$ penalties .

**What if the best value of the parameter being tuned comes out to be at the edge of the range being tried**:

This concern is not applicable for the hyper-parameters with fixed number of choices like `penalty` and `class_weight` here . This issue can happen for hyper-parameters which take potentially open ranges as input; like `C` or `alpha` that we saw earlier in Lasso and Ridge.

If the best value for these parameters comes out at the edge of the range being tried , you simply do the experiment again by expanding on that side of the range. [However do keep in mind that you can not keep on doing it forever if the best value keeps on coming at the edge, you will need to have some rationale to stop at some point, and this will differ from parameter to parameter]. For example the hyper-parameters pertaining to extent of penalty such as `C` or `alpha`; if they keep on moving in the direction which implies higher and higher penalty, it might mean all the features are junk and you should get new data or check data integrity and problem context. On the other hand if they keep on moving in the direction of lower and lower penalty it might imply that all features are very well related to response/target and you do not need penalty after all.

https://github.com/lalitsachan/text-courses

Once you have built the model with identified best parameters , one more task remains that is of finding the threshold for converting the probabilities to hard classes , i have included the function for the same demonstrated in the accompanying notebook.

https://github.com/lalitsachan/text-courses