Project 2: Learning to Rank using Linear Regression

Name - Abhav Luthra
Person no. - 50288904

1. Description of Model and Procedure followed

I was able to break the model into following parts:

- **Dataset**
  In this project we are applying the concept of linear regression on a learning to rank (LeToR) dataset. In the LeToR dataset the input vector is derived from a query-URL pair and the target value is human value assignment about how well the URL corresponds to the query.
  The first column of the dataset is the relevance label of the row. It takes one of the discrete values 0, 1 or 2. The larger the relevance label, the better is the match between query and document.
  It is followed by 46 columns representing features. They are the 46-dimensional input vector x for our linear regression model. All the features are normalized to fall in the interval of [0, 1].

- **Splitting and cleaning the Data Set**
  Given 69,623 data is cleaned i.e. the column with zero variance are removed and then the data is split into 80% for data training 10% for validation and 10% for testing. Zero Variance means that whole column has the same value therefore it makes no contribution in linear regression. It also doesn't allow us to calculate inverse in Phi matrix.

- **Regression Model**
  Linear regression is a **linear** approach to modelling the relationship between a scalar response (or dependent variable) and one or more explanatory variables (or independent variables) that are continuous in nature. We have a list of 46 independent variable also called features above and a target variable that is computed using basis function. This problem categorically falls in classification as it gives output in form of class 0,1,2. But we convert our linear output to class by rounding it to nearest integer. The problem here is solved in 2 ways : closed form and gradient descent but we observe that they give the similar accuracy therefore, any way can be chosen.

  Linear Regression function y(x,w) has the form,

$$T=W^T\Phi(x)$$

where w = (w0;w1;wM-1) is a weight vector to be learnt from training samples and $\Phi = (\Phi1, \Phi2, \ldots)$ is a vector of M basis functions

In this project we use Gaussian radial basis function

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_j)^\top \Sigma_j^{-1}(\mathbf{x}-\boldsymbol{\mu}_j)\right)$$

where μ_j = Centroid of each cluster
$\sum^{-1}{}_j$ = Variance - Covariance Matrix

- **Closed Form Solution**
  In classification problem, we use equation *y=wx+b* where w stands for weights, x for independent variable, y for target value and b for bias. But in our Linear Regression, as the output is not a straight line but a line with multiple curves (increasing and decreasing) i.e. is not linear so we represent such curve with basis functions represented by Φ(x) which is a set of curves.

  To calculate the basis function, we need to calculate μ and ∑. We have 3 possible ways:
  1. Centers for Gaussian radial basis functions μj: A simple way is to randomly pick up M data points as the centers.
  2. Spread for Gaussian radial basis functions ∑ j: A first try would be to use uniform spread for all basis functions ∑ j = ∑.
  3. k-means clustering: A more advanced method for choosing basis functions is to first use k -means clustering to partition the observations into M clusters. Then fit each cluster with a Gaussian radial basis function.

  We use the k-means clustering to calculate μ as it is the simplest and gives the highest accuracy as we take centroid over a cluster rather than guessing.

  To calculate, ∑ we need to find variance in the training data that can be easily found using numpy library.

  Once we get Φ(x), we need to invert it to find W as,

$$W=\Phi^{-1}(x)\ T$$

  As we can't invert Φ directly, we use Moore Penrose pseudo-inverse of the matrix Φ which is show as,

$$\Phi = (\Phi^{T}\Phi)^{-1}\Phi^{T}$$

  It is not an exact inverse because of dimensions, but is very close to real inverse.

  The closed form solution can therefore be shown as,

$$W= (\Phi^{T}\Phi)^{-1}\Phi^{T}\ T$$

  The closed-form solution with least-squared regularization,

$$W= (\ \lambda I + \Phi^{T}\Phi)^{-1}\Phi^{T}\ T$$

  Regularization is done to avoid the problem of over-fitting in this model. Regularization tries to remove the over dependency created in the training data set.

Now that the model is trained, we can find the error in the model. We also need to add a regularization term to the error function,

$$E(w) = E_D(w) + \lambda E_w(w)$$

Where λ is the relative importance of the regularization term and

$$E_w(w) = 0.5\ w^T w$$

And E is the root mean square of form,

$$E_{RMS} = \sqrt{2E(w^*)/N_V}$$

Where E(w) is of form,

$$E_D = 0.5 \sum (t_n - w^T \Phi(x_n))^2$$

This is the root mean square, it is reduced to optimize our model.

- **Stochastic Gradient Descent Solution**

  In Gradient Descent optimization, we compute the cost gradient based on the complete training set; hence, we sometimes also call it *batch Gradient Descent*. In case of very large datasets, using Gradient Descent can be quite costly since we are only taking a single step for one pass over the training set -- thus, the larger the training set, the slower our algorithm updates the weights and the longer it may take until it converges to the global cost minimum.

  The term "stochastic" comes from the fact that the gradient based on a single training sample is a "stochastic approximation" of the "true" cost gradient. Due to its stochastic nature, the path towards the global cost minimum is not "direct" as in GD, but may go "zig-zag" if we are visualizing the cost surface in a 2D space. However, it has been shown that SGD almost surely converges to the global cost minimum if the cost function is convex

  Weights are updated in this form:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta \mathbf{w}^{(\tau)}$$

  W is calculated by first order linear differentiation of weights given by:

$$\nabla E = \nabla E_D + \lambda \nabla E_W$$

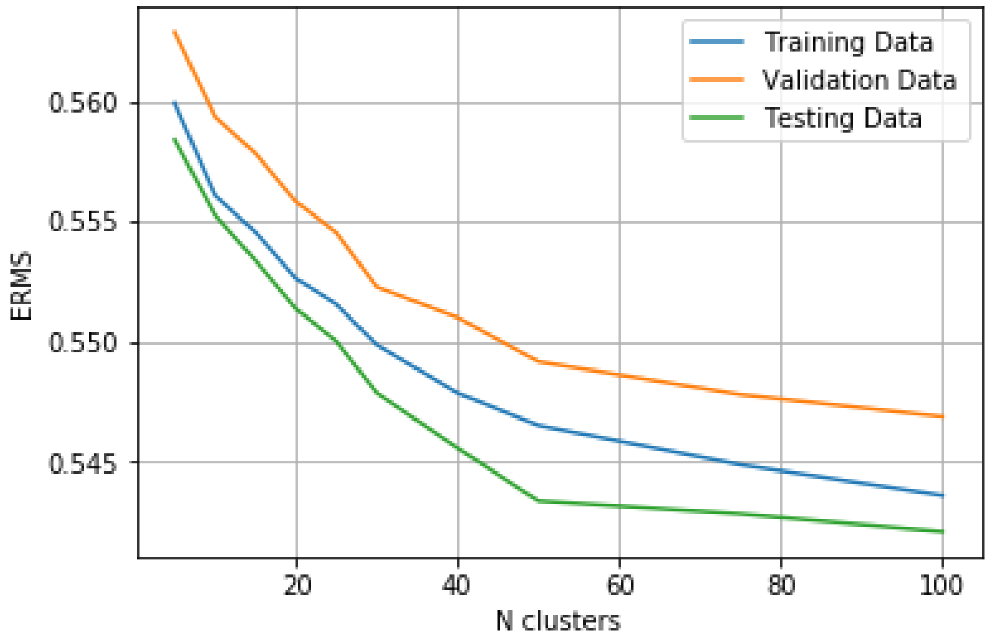
$$\nabla E_D = -(t_n - \mathbf{w}^{(\tau)\top} \phi(\mathbf{x}_n))\phi(\mathbf{x}_n)$$

$$\nabla E_W = \mathbf{w}^{(\tau)}$$

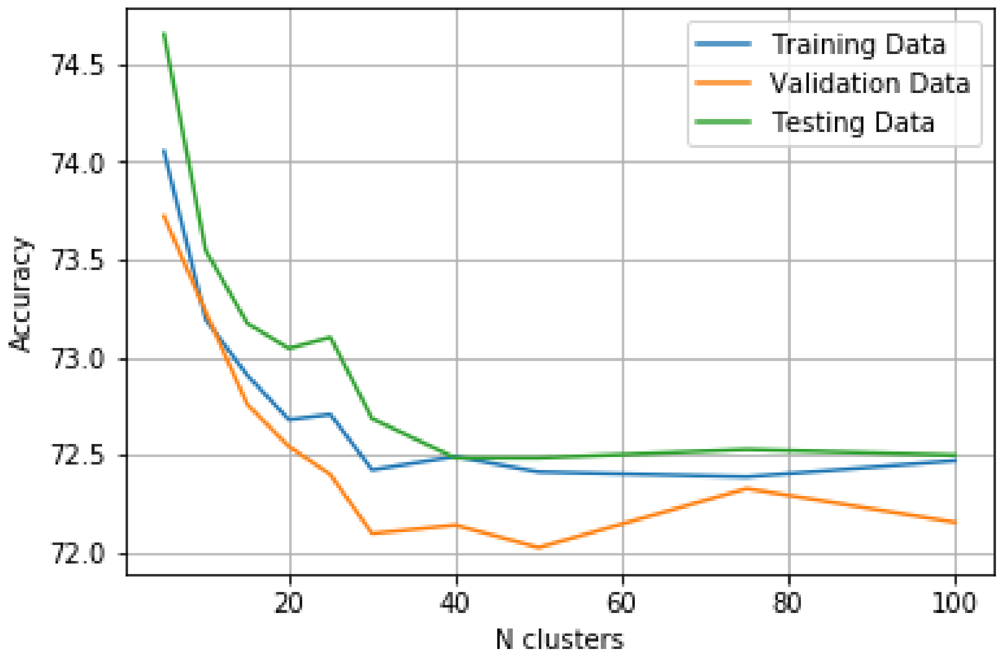
Where lambda is the learning rate for the optimization function

2. Experiment

Closed Form Solution:
Here we increased no of cluster from 5 to 100 and observed ERMS and Accuracy as:

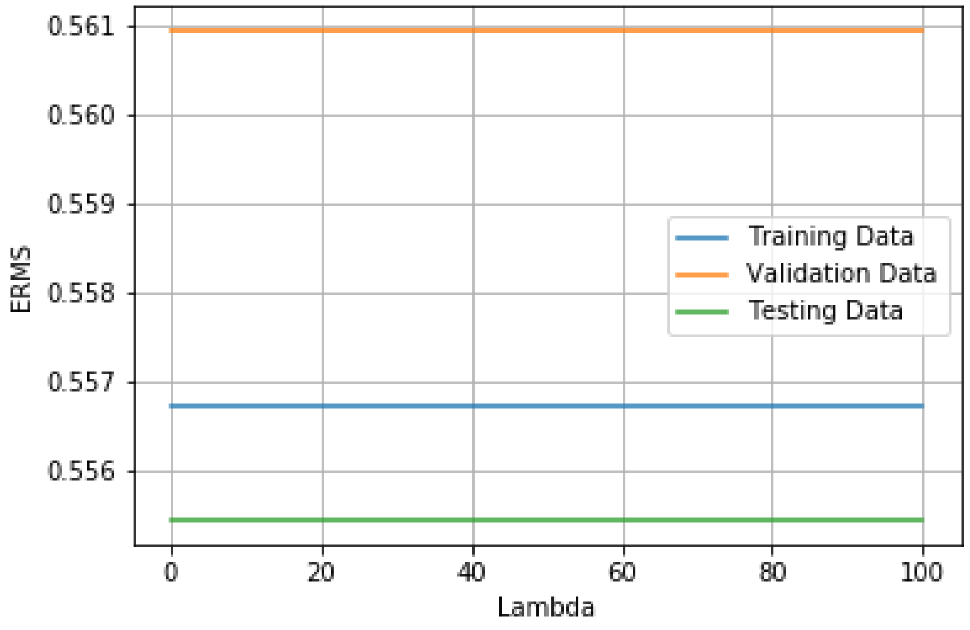


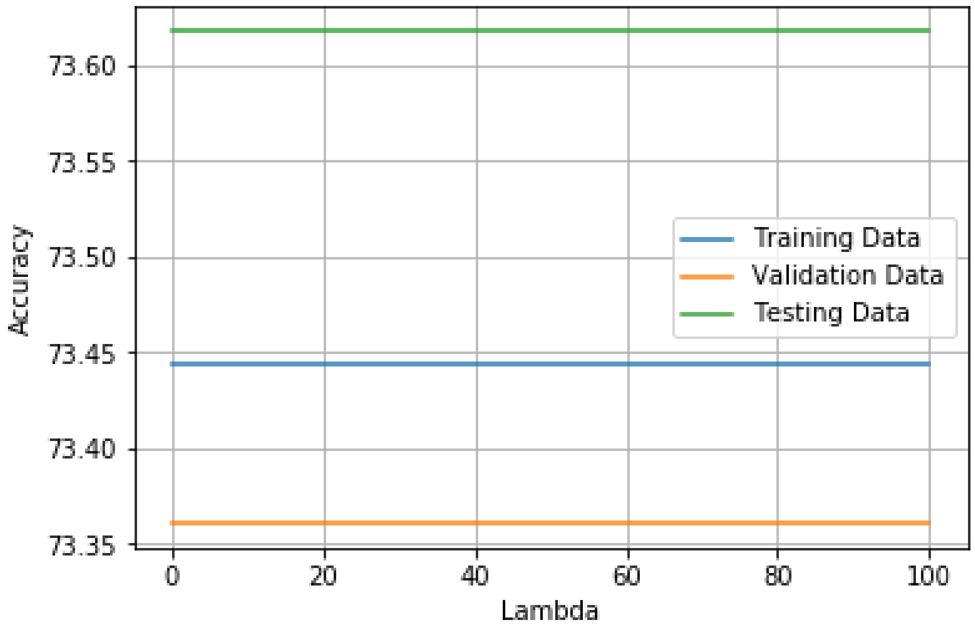
ERMS decreases as no of cluster increases.



Accuracy also show a general decreasing trend when no of cluster are increased. The more the number of clusters, the lesser would be the variance in the clusters and it will overfit the data which gives lower ERMS but also lower accuracy.

Here we increased Lamda from 0.1 to 100 and observed ERMS and Accuracy as:

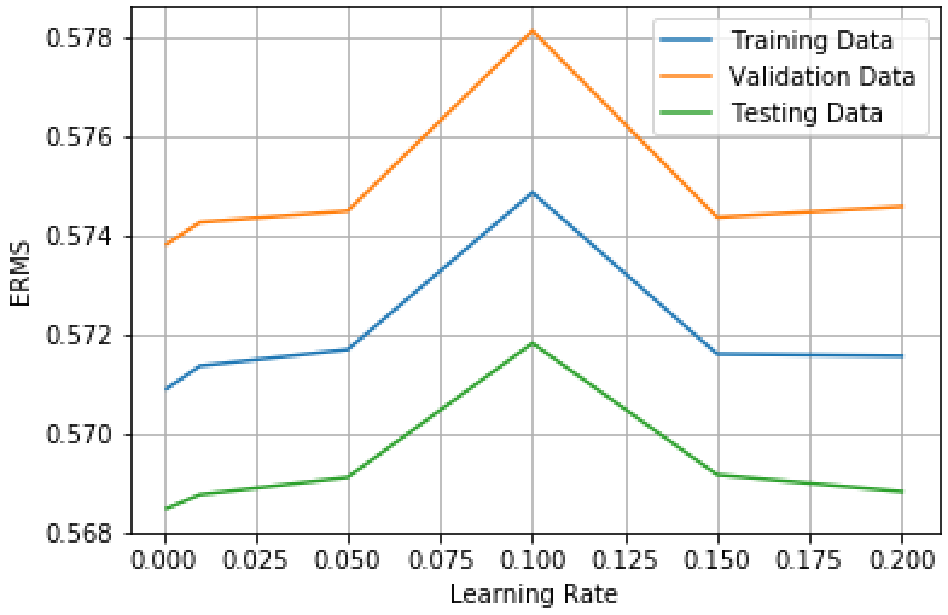


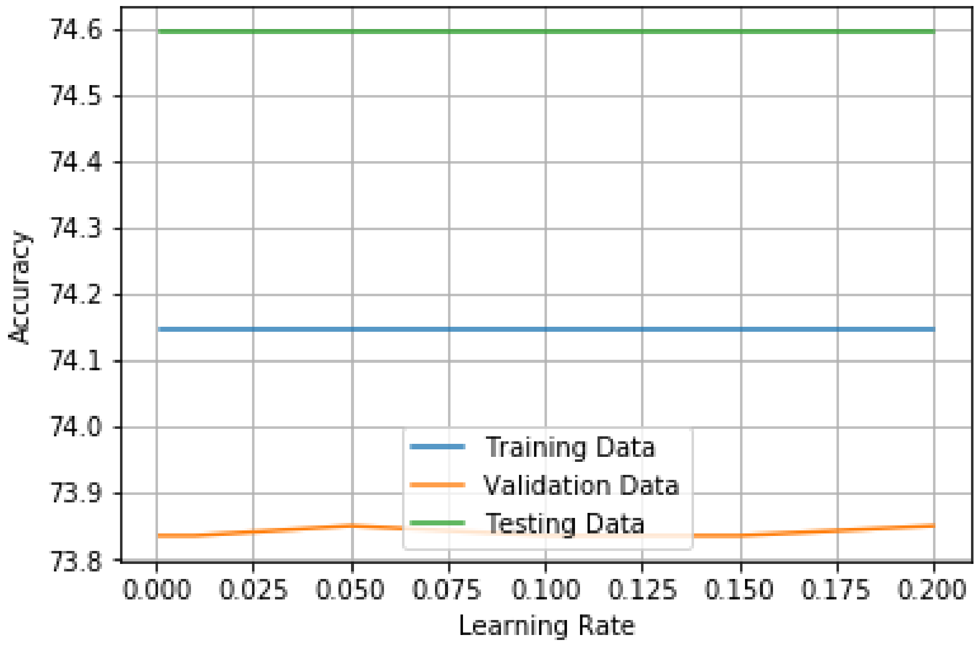


I have observed no overall impact of Lambda on the output values as Accuracy and ERMS remain similar even for extreme values of Lambda. Therefore, it’s just used to avoid regularization and its value does matter at all.

Gradient Descent

Here we increased Learning rate from 0.01 to 0.3 and observed ERMS and Accuracy as:





At 0.3, the erms value is 3.0 and the accuracy hits 0. So, we can't increase it more than that. Even though the accuracy more or less remain same for all learning rate, ERMS hits it max value at 0.1 and almost similar at other ends of the graph like a normal distribution.