Assignment #1

A REPORT ON

problems related to Curse of Dimensionality (CoD) & Gradient Descent and their corresponding solutions

BY

1. Akshit Khanna 2017A7PS0023P
2. Vitthal Bhandari 2017A7PS0136P

Prepared in partial fulfilment of the course
Foundations of Data Science - CS F320

SUBMITTED TO

Dr. Navneet Goyal

Professor

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

Task #1 Curse of Dimensionality (CoD)

We identified the problems in high dimensional data due to Curse of Dimensionality (CoD). They were as follows :

- *Problem #1* → **Overfitting of data (with limited samples)**

  If we keep adding features to a limited number of samples, the dimensionality of the feature space grows, and it becomes sparser and sparser. For d dimensions and v values to be distinguished along each axis, we seem to need $O(v^d)$ regions and examples.
  The ability to generalize correctly becomes exponentially harder as the dimensionality of the training dataset grows, as the training set covers a decreasing fraction of the input space.
  As a result, the classifier learns the appearance of specific instances and exceptions of the training dataset. Because of this, the resulting classifier would fail on real-world data, consisting of an infinite amount of unseen data points that often do not adhere to these exceptions.
  Using too many features results in overfitting and it is a direct result of CoD.

  Applying SVM on a small subset of Sonar dataset consisting of 50 samples with 60 dimensions we get overfitting with accuracy on training set being **77 %** and test set being **55%**. But after applying PCA and reducing dimensions from **60 to 19** to capture 95 % variance and then applying SVM we are able to get **70 %** training accuracy and much improved **66 %** test accuracy.

```
Training Accuracy by SVM on data : 0.7722222
Test Accuracy by SVM on data : 0.5555556

$train
[1] 0.7722222

$test
 Accuracy
0.5555556

Number of Dimensions reduced from  60  to  19 by PCA.
Training Accuracy by SVM on data : 0.7016667
Test Accuracy by SVM on data : 0.6666667

$train
[1] 0.7016667
```

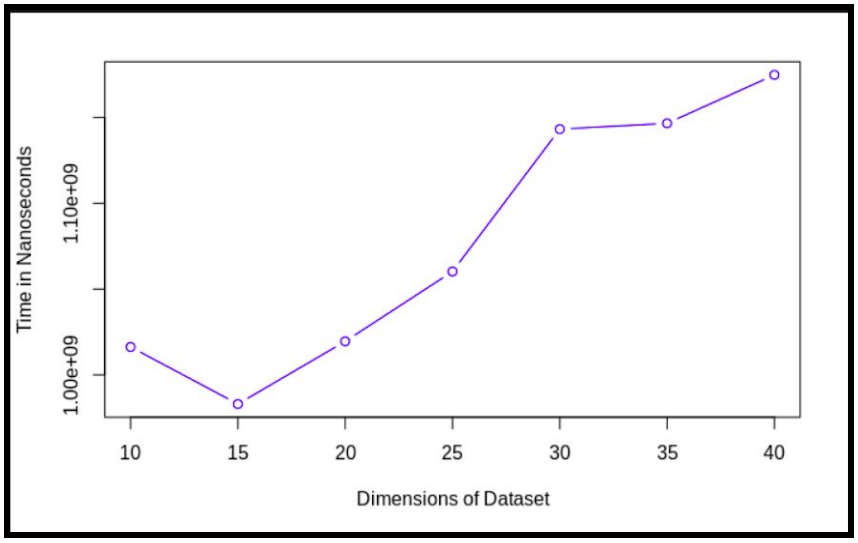
Overfitting Example of Sonar Dataset

- *Problem #2* → **Increased Computations on Data**

The more dimensions we add to a data set, the more difficult it becomes to predict certain quantities. When we add more dimensions, it makes sense that the computational burden also increases.

The statistical curse of dimensionality refers to a related fact : a required sample size n will grow exponentially with data that has d dimensions. In simple terms, adding more dimensions could mean that the sample size we need quickly becomes unmanageable.

Here we have demonstrated the above problem by showing a comparison of time for SVM classification on different numbers of dimensions on the Sonar Dataset.



Time Taken by SVM on different dimension of Sonar Dataset

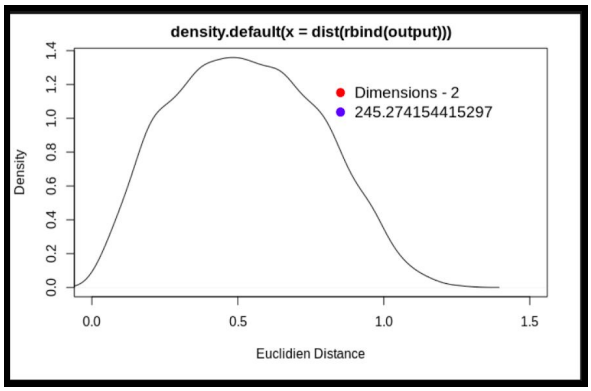
❖ *Problem #3* → **Concentration effect of Lp-Norms**

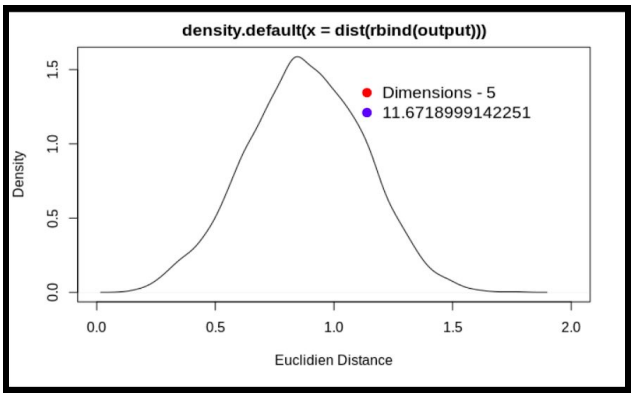
In high dimensional spaces, most of the training data resides in the corners of the hypercube defining the feature space. Instances in the corners of the feature space are much more difficult to classify than instances around the centroid of the hypersphere.

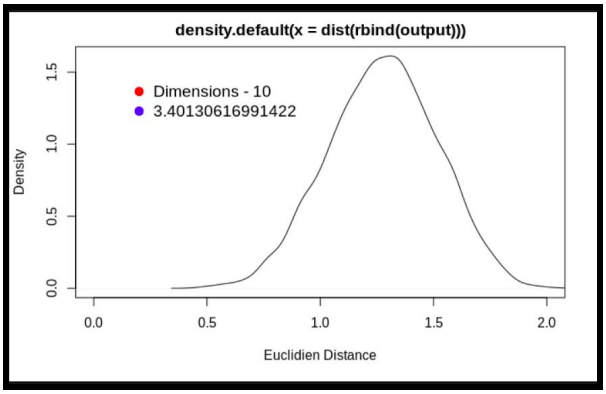
As a result, when the dimensionality of the feature space goes to infinity, the ratio of the difference in minimum and maximum Euclidean distance from sample point to the centroid, and the minimum distance itself, tends to zero.

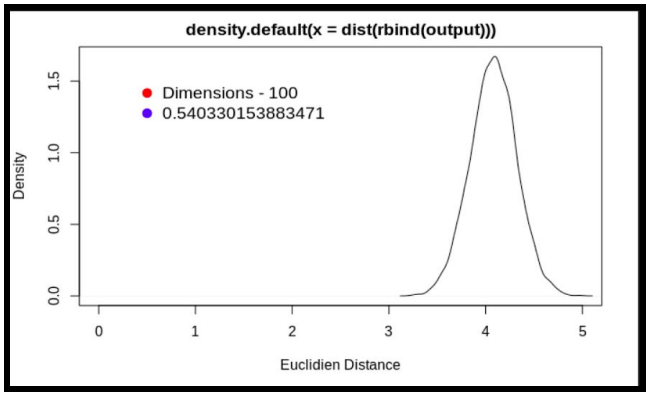
Therefore, distance measures start losing their effectiveness to measure dissimilarity in highly dimensional spaces. Since classifiers depend on these distance measures (e.g. Euclidean distance, Mahalanobis distance, Manhattan distance), classification is often easier in lower-dimensional spaces where less features are used to describe the object of interest.

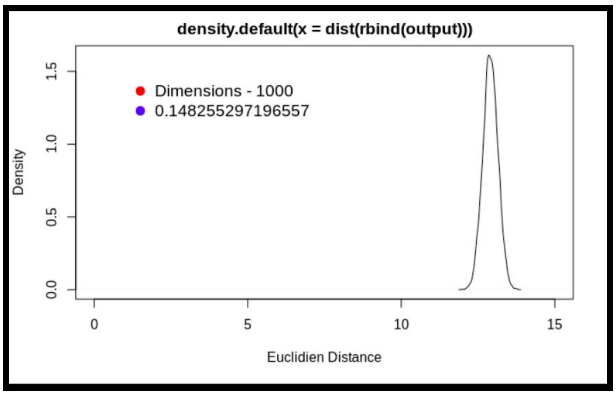
Here we have demonstrated the concentration effect of Lp-Norms using 5 different plots of euclidean distance between among the samples for different numbers of dimensions (2, 5, 10, 100, 1000) for 100 samples randomly and uniformly distributed in these dimensions. The value in blue written on the graphs shows the ratio of (max-min) / min distance in these dimensions. As expected the ratio keeps on decreasing as the number of dimensions is increased and tends to zero.











Density Plots for uniformly distributed samples in different number of dimensions

❖ *Problem #4* → **Relevant and Irrelevant features**

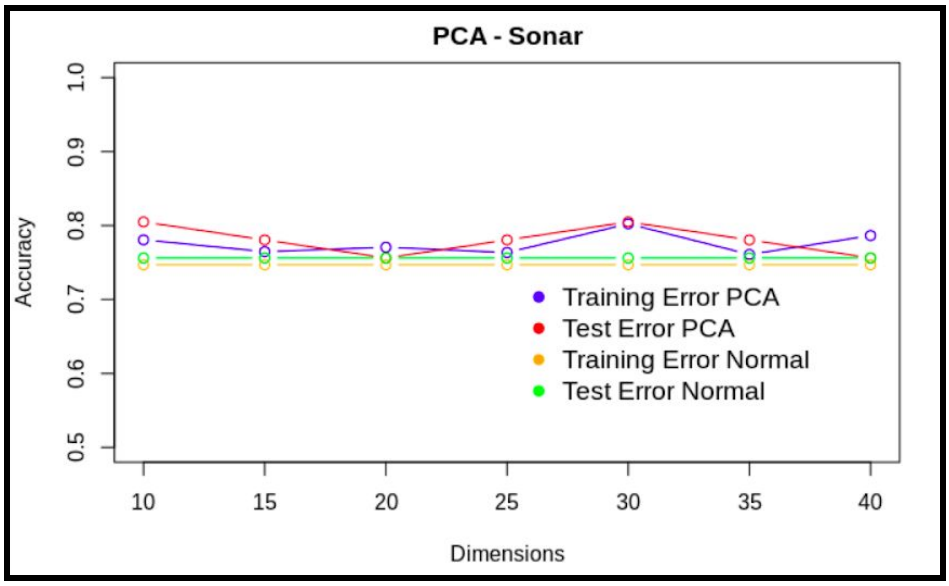
in high dimensional data, a significant number of attributes may be irrelevant to the classification or regression problem at hand. It may happen, for example, that for a sample training set with 22 attributes (say), a few attributes have the same value for all points. In such a case these attributes provide no additional differentiation to help classify points. Thus, such irrelevant attributes need to be removed.

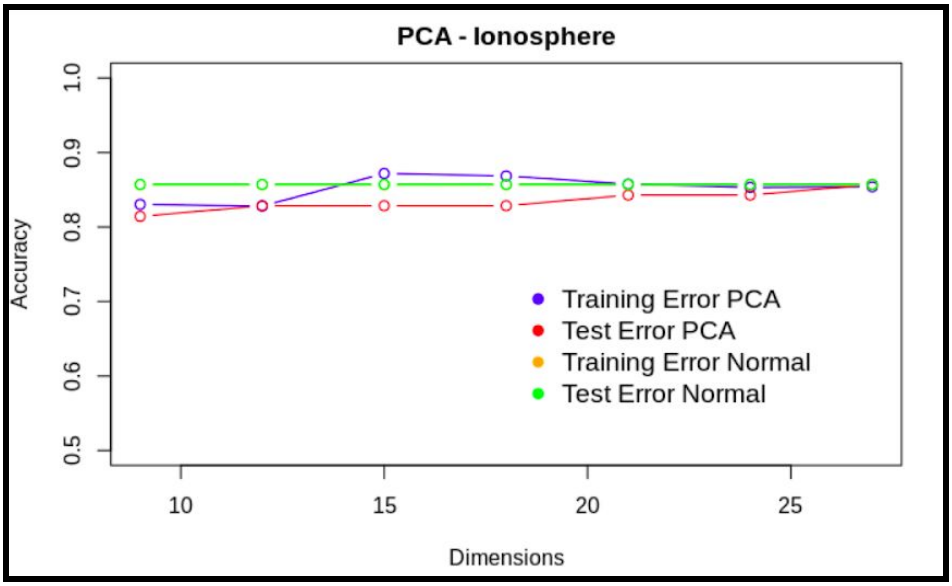
In other cases, many attributes contribute little or no amount of variance to the overall population. It makes sense to compute on only the attributes with high variability to reduce computation cost assuming no loss of generality.

- ❖ *Solutions to above Problems* → Successful existing algorithms are :
- ➢ Principal Component Analysis (**PCA**)
- ➢ **Kernel PCA**
- ➢ Singular Value Decomposition (**SVD**)

PCA is an unsupervised, non-parametric statistical technique primarily used for dimensionality reduction of **linearly separable datasets**.
We have used standard datasets in R that will demonstrate the advantage of dimensionality reduction.
For implementing PCA we have used the **Connectionist Bench (Sonar, Mines vs. Rocks) Data Set**. It contains 111 patterns obtained by bouncing sonar signals off a metal cylinder at various angles and under various conditions and 97 patterns obtained from rocks under similar conditions.
Each pattern is a set of 60 numbers in the range 0.0 to 1.0. Each number represents the energy within a particular frequency band, integrated over a certain period of time. The label associated with each record contains the letter "R" if the object is a rock and "M" if it is a mine (metal cylinder).
The dataset comprises **208** number of Instances and **61** Attributes.

We have also used the **Johns Hopkins University Ionosphere Data set.**
It consists of a phased array of 16 high-frequency antennas with a total transmitted power on the order of 6.4 kilowatts. "Good" radar returns are those showing evidence of some type of structure in the ionosphere. "Bad" returns are those that do not; their signals pass through the ionosphere.
There were 17 pulse numbers for the Goose Bay system. Instances in this database are described by 2 attributes per pulse number, corresponding to the complex values returned by the function resulting from the complex electromagnetic signal.
The dataset contains **351** instances and **35** attributes.

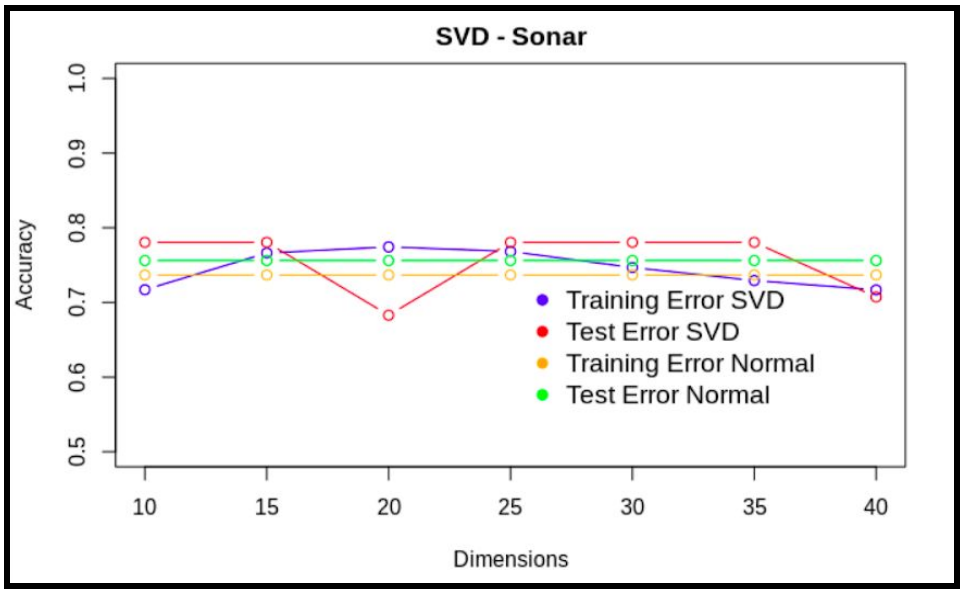


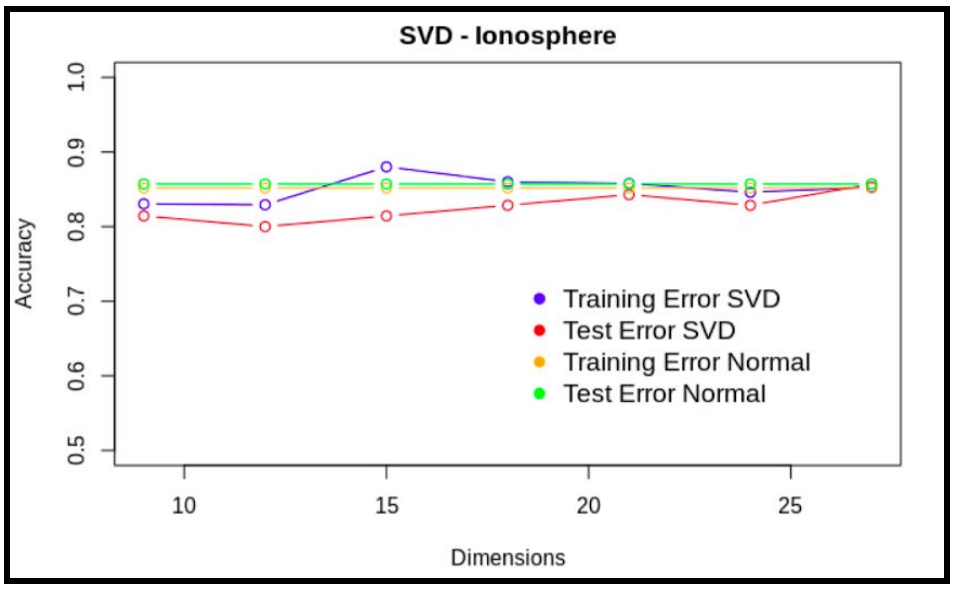


PCA Results on both datasets

Singular Value Decomposition (SVD) is one of the most widely used Unsupervised learning algorithms, that is at the core of many recommendation and dimensionality reduction systems .

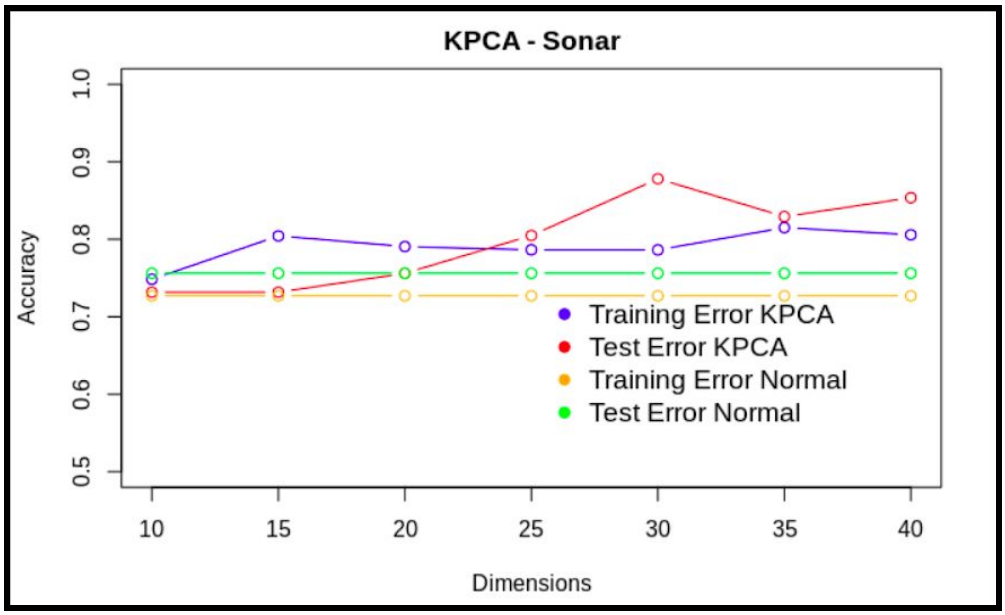
In simple terms, SVD is the factorization of a matrix into 3 matrices ($A=U\Sigma V^T$).

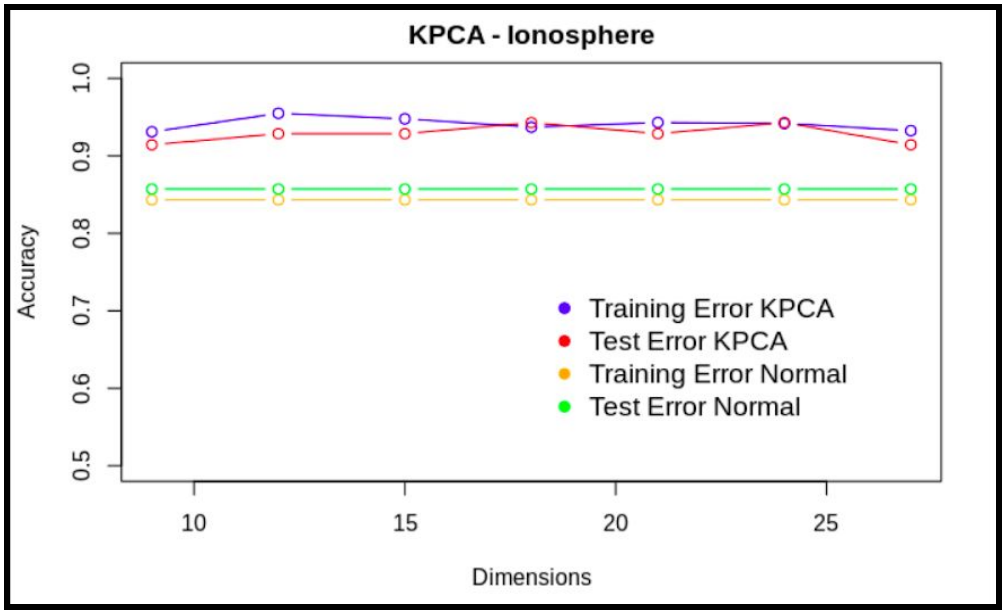




SVD Results on both datasets

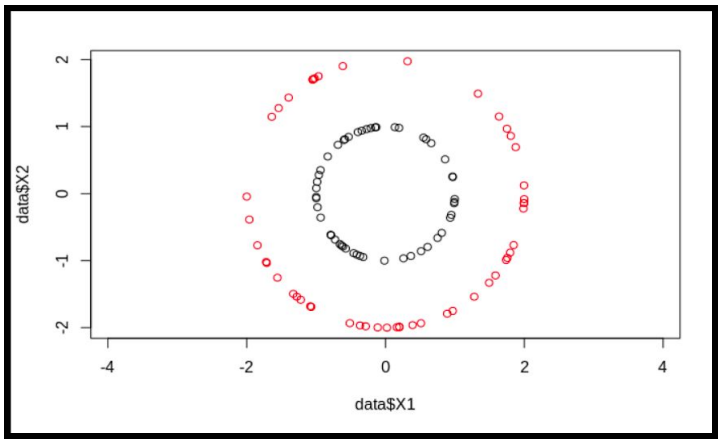
Another technique called Kernel PCA uses a kernel function to project a dataset onto a higher dimensional feature space, where it is linearly separable. It is primarily used for dimensionality reduction of **non-linearly separable datasets**.
We have also shown its results on both Sonar and Ionosphere datasets.



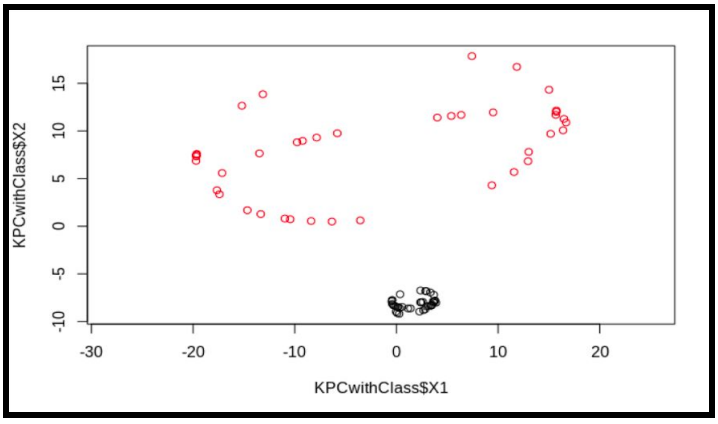


Kernel PCA Results on both datasets

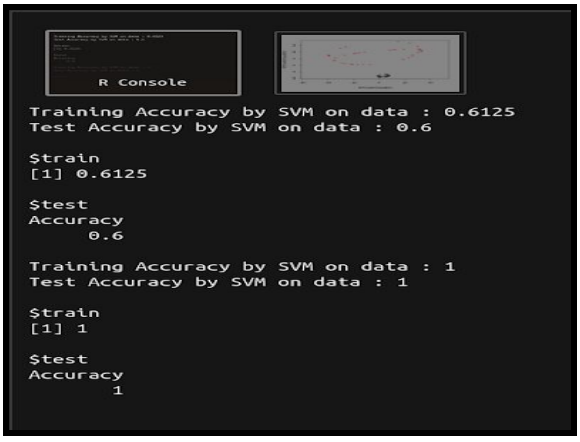
For specific application of Kernel PCA we have shown its results on the concentric circles dataset which is a nonlinear dataset with 2 dimensions and 2 classes.



Original Dataset Plot



Dataset Plot after KPCA application



Accuracy using SVM before and after using KPCA

❖ *Interpretation of Results obtained from various Dimensionality Reduction Techniques*

From the above plots of the results from dimensionality reduction techniques such as PCA,SVD and KCPA, we observe that in both the datasets the results from KPCA consistently are better than both PCA and SVD across all the different dimensions. KPCA results are also better than the baseline results ,that is the results without dimensionality reduction. This leads us to conclude that there must be some non-linearity in both the datasets.

Both PCA and SVD are able to maintain similar performance to the baseline accuracies after reducing dimensions as is expected because they capture the most of the variance into principal components and utilise them for classification.

Task #2 Gradient Descent (GD)

Perhaps the simplest algorithm for unconstrained optimization is gradient descent. This can be written as follows :

$$\theta_{k+1} = \theta_k - \eta_k g_k$$

Where η_k is the step size or **learning rate**.
We identified various problems in GD. They were as follows :

❖ *Problem #1* → **It can lead to local minima for a non-convex loss function**

Neural networks are complicated functions, with lots of non-linear transformations thrown in the hypothesis function. The resultant loss function is hardly convex in real life.

Gradient descent is driven by the gradient, which will be zero at the base of any minima. Local minima are called so since the value of the loss function is minimum at that point in a local region. Whereas, a global minima is called so since the value of the loss function is minimum there, globally across the entire domain of the loss function.

With gradient descent the problem is that once if we converge to a local minima, there is no way we can get out of it.

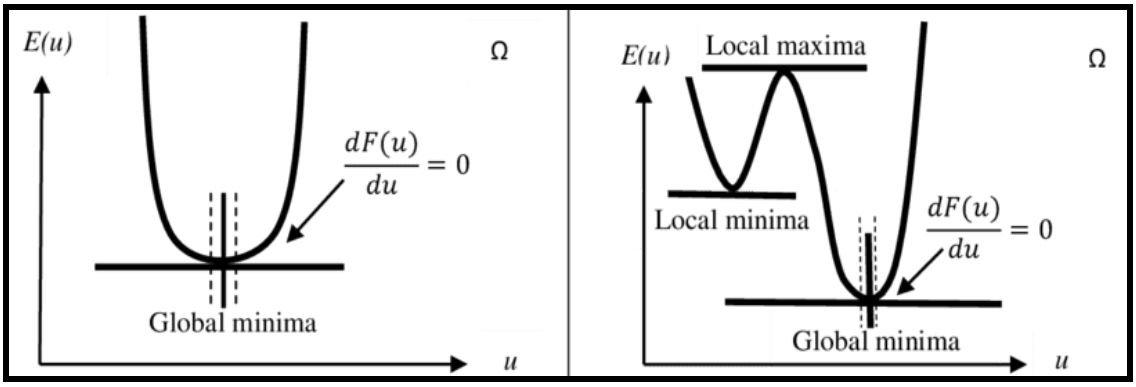


Figure depicting local minima for a convex loss-function (which becomes global minima) and global minima for a non-convex loss function

❖ *Problem #2* → **Choice of a suitable learning rate**

The main issue in gradient descent is: how should we set the step size?

The learning rate may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time. If it is **too large**, the learning curve will show **violent oscillations**, with the cost function often increasing significantly. Gentle oscillations are fine, especially if training with a stochastic cost function such as the cost function arising from the use of dropout. If the learning rate is **too low**, learning **proceeds slowly**, and if the initial learning rate is too low, learning may become stuck with a high cost value.

The following figures show the loss function of gradient descent on linear regression with different learning rates (0.01, 0.1, 0.3, 0.5). As seen in the figures a lower learning rate leads to increase in the number of iterations taken to minimize the loss function, while a higher learning rate reaches the minima faster. But care must be taken in setting the learning rate too high as it could lead to divergence as a larger step size could skip the minima curve altogether as is shown in the 0.5 learning rate plot.

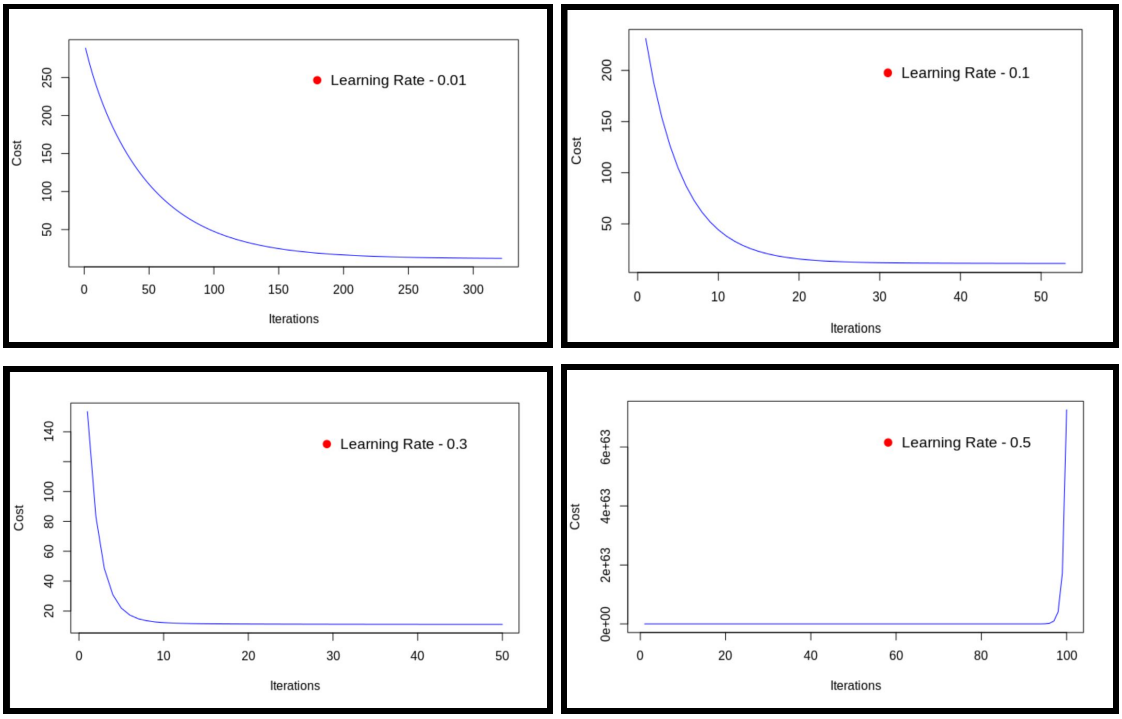


Figure depicting the loss function of gradient descent on linear regression with different learning rates (0.01, 0.1, 0.3, 0.5)

❖ *Problem #3* → **Slow for large datasets**

Gradient descent is not suitable for huge datasets. As we need to calculate the gradient on the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. After initializing the parameter with arbitrary values we calculate gradient of cost function using following relation :

Repeat until convergence

{

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

}

where 'm' is the number of training examples.

If we have a large number of records we need to read all the records into memory from disk because we can't store them all in memory.

After calculating sigma for one iteration, we move one step. Then **repeat for every step**. This means it takes a long time to converge.
Especially because disk I/O is typically a system bottleneck anyway, and this will inevitably require a huge number of reads. Thus :

1. Too many gradient descent updates are required.

2. Each gradient descent step is too expensive.

❖ *Solutions to above Problems* → Various algorithms exist for gradient descent. All are variants of gradient descent only. These are :

➢ **Batch Gradient Descent**

➢ **Stochastic Gradient Descent**

➢ **Mini-batch Gradient Descent**

We have used the **Boston Housing Dataset**. This dataset contains information collected by the U.S Census Service concerning housing in the area of Boston Mass. The objective is to predict the value of prices of the house using the given features. It comprises **506** instances and **14** attributes.

Batch Gradient Descent is the first basic type of gradient descent in which we use the complete dataset available to compute the gradient of cost function.

Batch Gradient Descent turns out to be a slower algorithm. So, for faster computation, we prefer to use stochastic gradient descent.

The first step of the algorithm is to randomize the whole training set. Then, for updation of every parameter we use only one training example in every iteration to compute the gradient of cost function. As it uses one training example in every iteration this algorithm is faster for larger data sets. In SGD, one might not achieve accuracy, but the computation of results are faster.

SGD Never actually converges like batch gradient descent does,but ends up wandering around some region close to the global minimum.SGD also fluctuates more than the other methods due to a single random sample taken at every iteration.

Mini batch algorithm is the most favorable and widely used algorithm that makes precise and faster results using a batch of ‘m‘ training examples. In mini batch algorithm, rather than using the complete data set, in every iteration we use a set of ‘m’ training examples called batch to compute the gradient of the cost function. Common mini-batch sizes range between 50 and 256, but can vary for different applications.

In this way, the algorithm :

- reduces the variance of the parameter updates, which can lead to more stable convergence.
- can make use of a highly optimized matrix, that makes computation of gradients very efficient.

While implementing the algorithm, updation of parameters should be done simultaneously. This means, during updation values of parameters should be stored first in some temporary variable and then assigned to the parameters.

Comparing time taken by Batch, Stochastic and Mini Batch Gradient Descent it is clear that Stochastic is the fastest algorithm since it only calculates gradient for a single sample. Then Mini-Batch takes some time as is expected while Batch gradient descent takes more than double the time of the other methods.

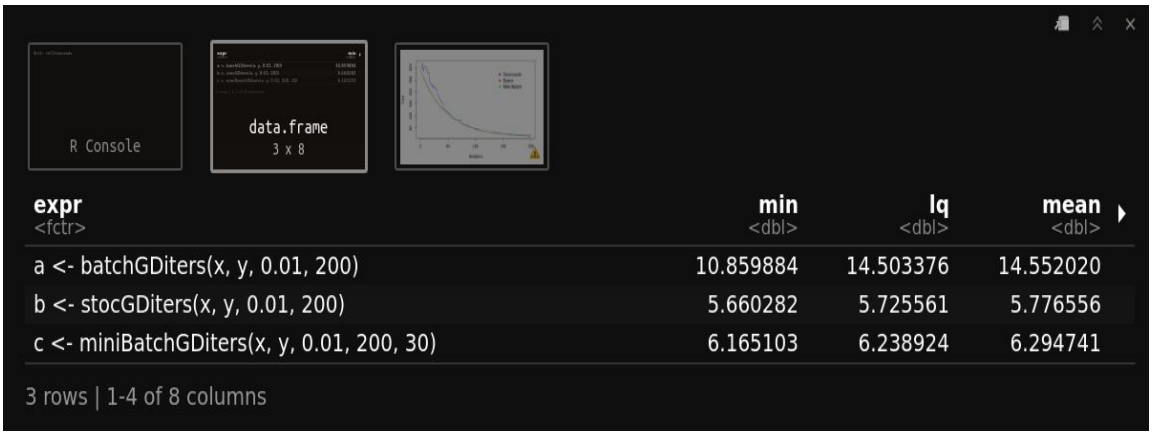


Figure depicting comparison between time taken by different variants of Gradient Descent

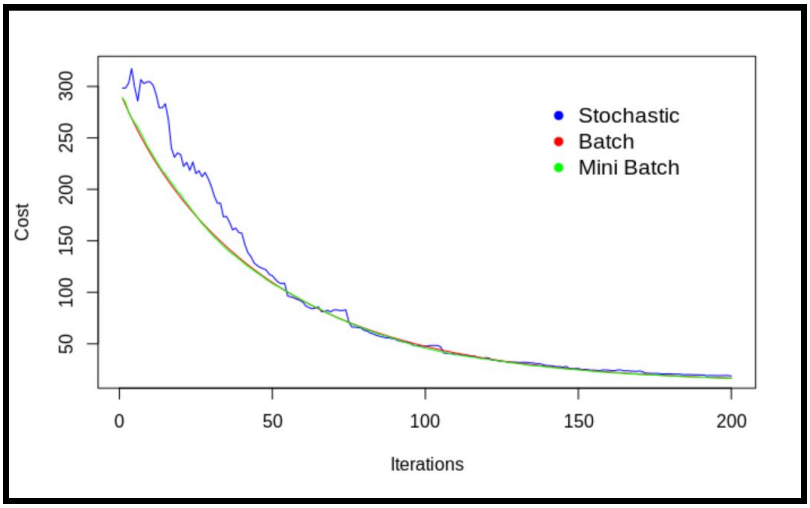


Figure depicting comparison between cost vs iterations for different variants of Gradient Descent

Both Batch and Mini-Batch gradient Descent have similar plots while Stochastic gradient Descent has a fluctuating plot due to taking only a single sample each iteration to calculate gradient.

❑ Mini-Batch with Momentum

The method of momentum is designed to **accelerate learning**, especially in the face of high curvature, small but consistent gradients.The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Momentum aims primarily to solve variance in the mini-batch gradient.

Previously, the size of the step was simply the norm of the gradient multiplied by the learning rate. Now, the size of the step depends on how large and how aligned a sequence of gradients are. The step size is largest when many successive gradients point in exactly the same direction.

Formally, the momentum algorithm introduces a variable v that plays the role of velocity. The velocity is set to an exponentially decaying average of the negative gradient. A hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay.

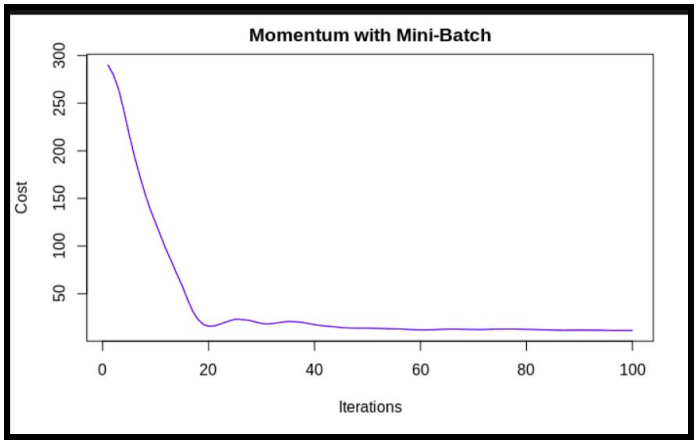


Figure depicting comparison between cost vs iterations for Momentum with mini-batch

❑ Nesterov Momentum

It is a variant of the momentum algorithm. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied. Thus, instead of using the current position's gradient, it uses the next approximated position's gradient with the hope that it will give us better information when we're taking the next step.

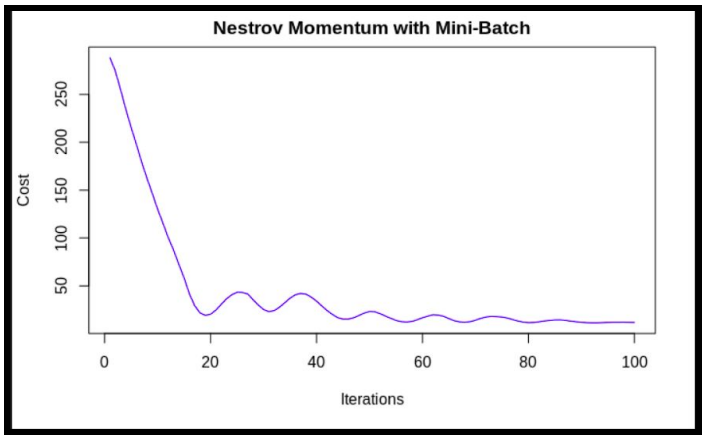


Figure depicting comparison between cost vs iterations for Nestrov Momentum with mini-batch

❑ AdaGrad

The AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values. The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial

derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space. It makes the learning rate dynamic and helps eliminate the importance of learning rate hyperparameter.

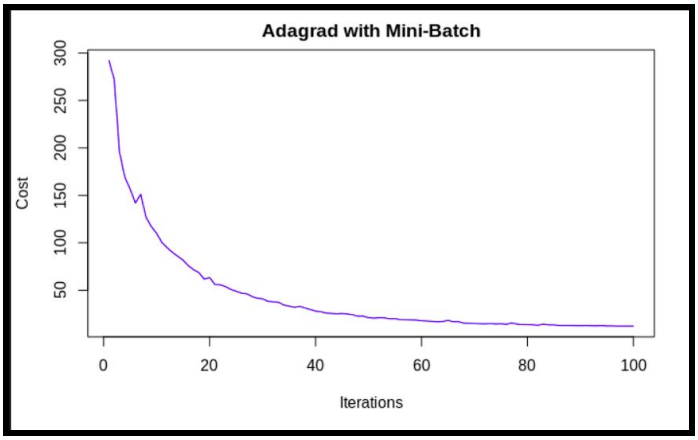


Figure depicting comparison between cost vs iterations for AdaGrad with mini-batch

❑ Adam

Adam is yet another adaptive learning rate optimization algorithm. The name “Adam” derives from the phrase “adaptive moments”.

1. In Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient.
2. Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin. Adam is generally regarded as being fairly robust to the choice of hyperparameters.

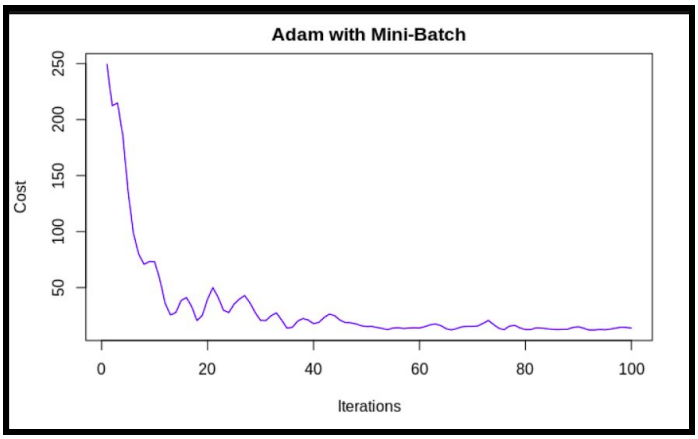


Figure depicting comparison between cost vs iterations for Adam with mini-batch

❖ *Interpretation of Results obtained from various Gradient Descent Techniques*

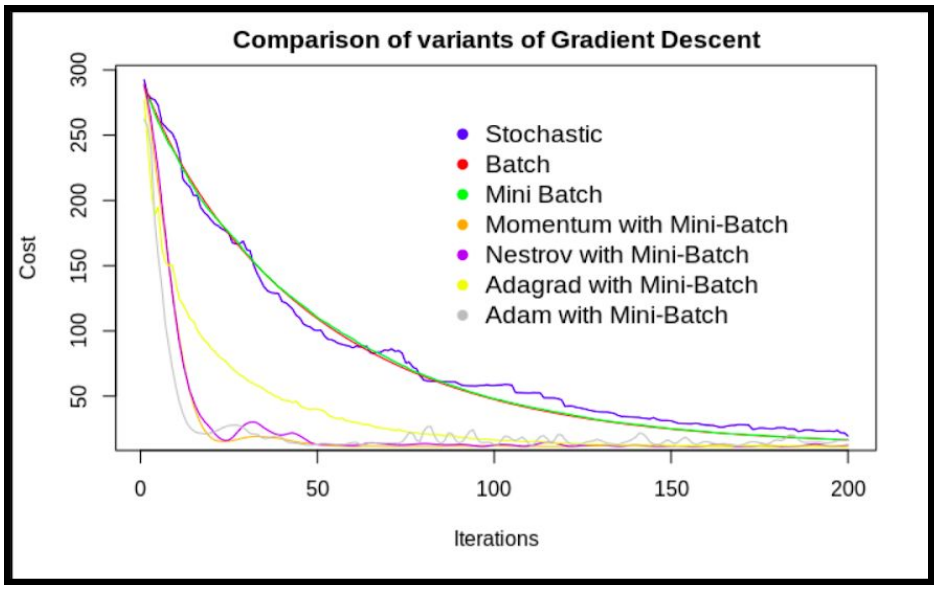


Figure depicting comparison between cost vs iterations for different variants of Gradient descent along with Adaptive learning rate Methods

expr <fctr>	min <dbl>	lq <dbl>	mean <dbl>
a <- batchGDiters(x, y, 0.01, 200)	11.126079	11.630210	13.377883
b <- stocGDiters(x, y, 0.01, 200)	5.768593	6.166519	6.173004
c <- miniBatchGDiters(x, y, 0.01, 200, 20)	6.559953	6.571839	6.915220
d <- momentumMiniGDiters(x, y, 0.01, 200, 20)	6.532727	6.847045	6.922575
e <- nestrovMiniGDiters(x, y, 0.01, 200, 20)	6.583830	7.171625	8.878027
f <- adaMiniGDiters(x, y, 1.5, 200, 20)	7.045609	7.135313	7.179251
g <- adamMiniGDiters(x, y, 1.5, 200, 20)	7.139991	7.529898	7.711808

Table depicting comparison between time taken by different variants of Gradient descent along with Adaptive learning rate methods

The times taken by each variant apart from batch gradient descent are in a similar neighbourhood.

Adam is the fastest to get to the minima of the dataset. Nesterov and Momentum are similar in performance. Adagrad is slow in convergence since it depends on the starting learning rate. If a sufficiently large learning rate is used it can converge even faster. Then the performance of normal methods is worse than these improvements as is expected.

Conclusion

In this report we first looked at various problems related to high dimensional data which includes sparsity of data, overfitting, irrelevant features and diminishing effectiveness of distance measures. The proposed solutions included Principal Component Analysis (PCA), Kernel PCA and Singular Value Decomposition (SVD). These solutions were implemented in R and the results obtained clearly indicate their efficacy in overcoming all the problems.

We then shifted our focus to Gradient Descent and realized that we face problems where the objective function gets stuck at a local minima for a non-convex loss function or that GD runs slow for larger datasets. We also observed that the learning rate should be chosen suitably.

A few successful variations of GD exist to overcome these issues such as Batch Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent. We saw that the cost is often highly sensitive to some directions in parameter space and insensitive to others. The momentum algorithm can mitigate these issues somewhat, but does so at the expense of introducing another hyperparameter.
We then discussed a number of incremental (or mini-batch-based) methods that adapt the learning rates of model parameters. These include Nesterov Momentum, AdaGrad and Adam.

Having implemented these solutions in R, the results suggest that the family of algorithms with adaptive learning rates performed fairly robustly and emerged as clear winners.
The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning).