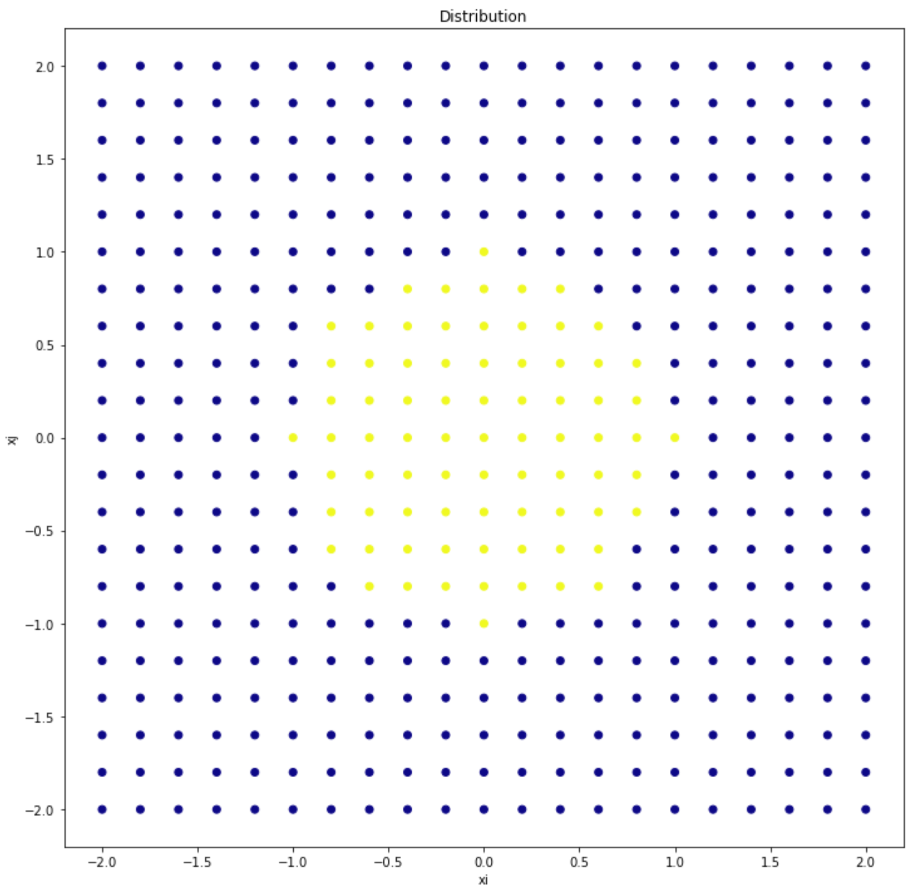
PROBLEM 2: RBF NEURAL NETWORK

SOLUTION:

We create our input and mapping of size 441 according to the directions and formulae given in the question. When plotted, the following distribution is observed (yellow dots have an output of 1 and blue dots have an output of -1)



We then split the dataset using the *train_test_split()* function from the Scikit-Learn library into 2 parts with a ratio 8:2, one for training and one for testing. The training set has 352 observations while the test set has 89.

The following sections will detail the design of the network followed by the solution to all 3 sub questions. Each solution will contain a table showing the mean squared errors for every spread value chosen followed by a graph showing a visual representation of the same. The lowest test MSE and the spread at which it is obtained as well as average training time will be mentioned at the end of each section. Finally, comparison graphs are shown and a conclusion will be drawn with based on all the results obtained.

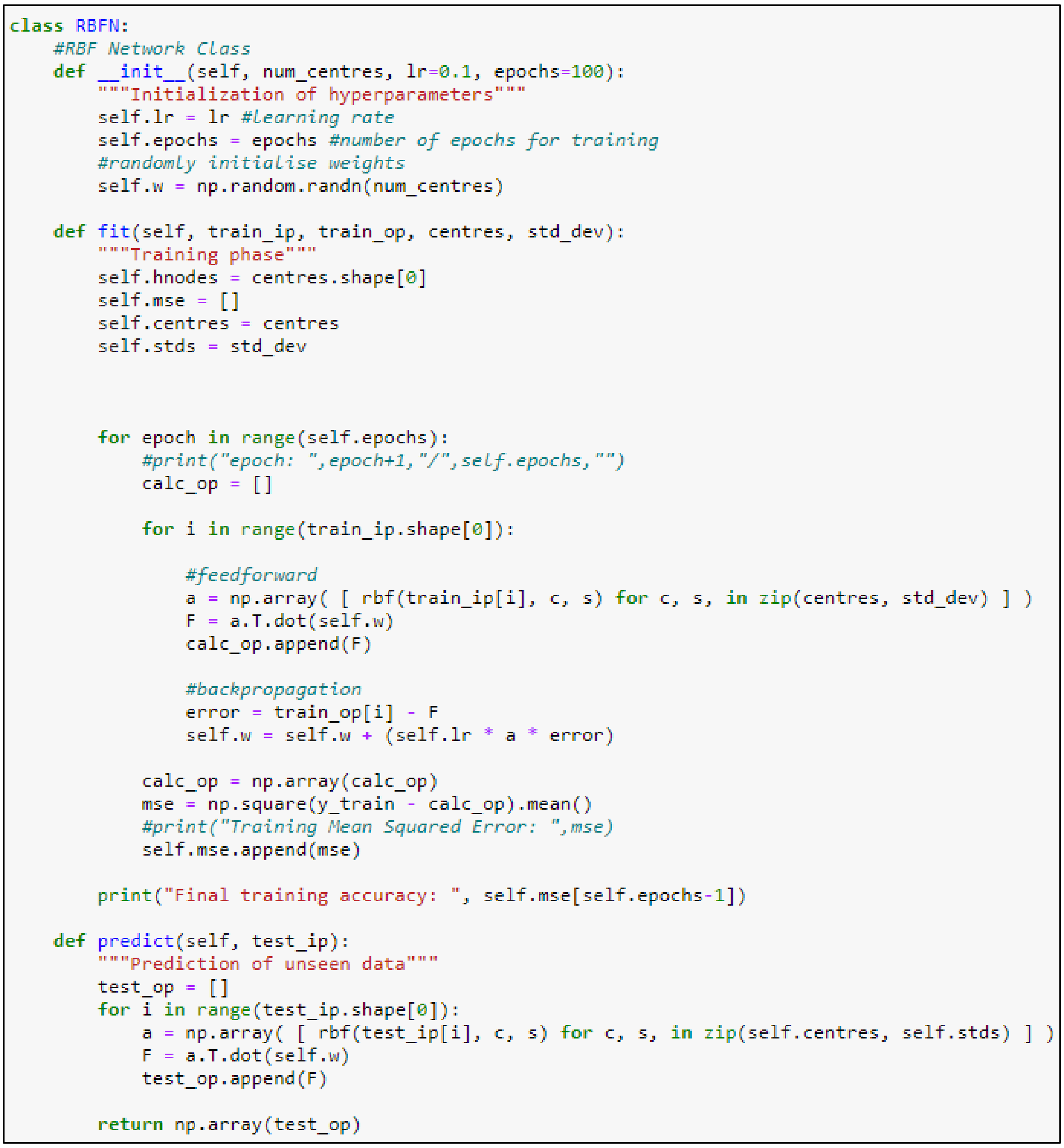
Everything discussed here is present in the Jupyter notebook containing the code and brief descriptions as Markdown text.

Design of the RBF NN –

In the code, we first defined a separate *rbf()* function that implements the Gaussian activation function shown below

$$G(x) = \exp\left(-\frac{\|x - c\|^2}{2\sigma^2}\right)$$

The neural network itself is designed as a Python Class so it can be reused easily, as shown below. We use gradient descent for learning and weight updating. A random seed is set at the start of the notebook to ensure consistent results.

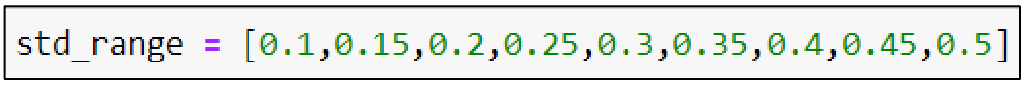


```python
class RBFN:
    #RBF Network Class
    def __init__(self, num_centres, lr=0.1, epochs=100):
        """Initialization of hyperparameters"""
        self.lr = lr #learning rate
        self.epochs = epochs #number of epochs for training
        #randomly initialise weights
        self.w = np.random.randn(num_centres)

    def fit(self, train_ip, train_op, centres, std_dev):
        """Training phase"""
        self.hnodes = centres.shape[0]
        self.mse = []
        self.centres = centres
        self.stds = std_dev



        for epoch in range(self.epochs):
            #print("epoch: ",epoch+1,"/",self.epochs,"")
            calc_op = []

            for i in range(train_ip.shape[0]):

                #feedforward
                a = np.array( [ rbf(train_ip[i], c, s) for c, s, in zip(centres, std_dev) ] )
                F = a.T.dot(self.w)
                calc_op.append(F)

                #backpropagation
                error = train_op[i] - F
                self.w = self.w + (self.lr * a * error)

            calc_op = np.array(calc_op)
            mse = np.square(y_train - calc_op).mean()
            #print("Training Mean Squared Error: ",mse)
            self.mse.append(mse)

        print("Final training accuracy: ", self.mse[self.epochs-1])

    def predict(self, test_ip):
        """Prediction of unseen data"""
        test_op = []
        for i in range(test_ip.shape[0]):
            a = np.array( [ rbf(test_ip[i], c, s) for c, s, in zip(self.centres, self.stds) ] )
            F = a.T.dot(self.w)
            test_op.append(F)

        return np.array(test_op)
```

(Above, removing the two commented lines in the *for* block allows us to observe the training MSE at every epoch. Since we train our network with 10 spread parameters and 3 centres each for 100 epoch, this would cause our notebook to have 3000 lines of results. Hence, they were commented out and not reported)

We set the epoch value at 100 and the learning rate at 0.1. This hyperparameters were set after some experimentation based on probability as well as computation time required for finding a local minimum in the mean squared error plots.

As required in the question, we keep the spread parameter i.e., the standard deviation of the kernel function same for all the nodes in the hidden layer of the network. Also, we vary this parameter between each run to compare their effect on the mean squared error of the model. We chose a range of 0.1 to 0.5 with steps of 0.05, giving us a total of 10 spread values for each type of centre method chosen.



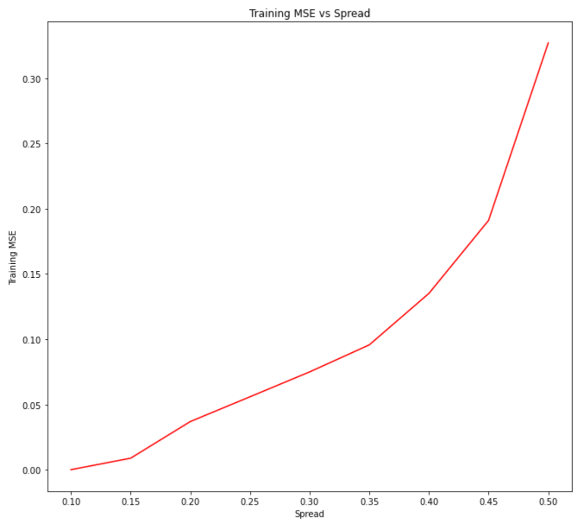
```
std_range = [0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5]
```

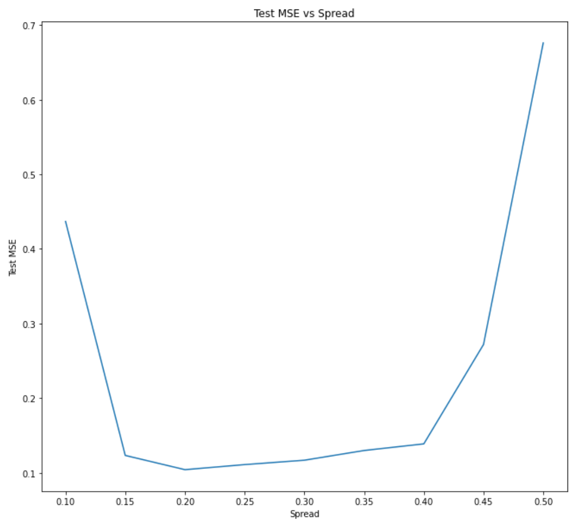
The above range is chosen because values smaller than 0.1 showed no significant characteristics that could impact our end results. Values larger than 0.6 suffered from the exploding gradient problem even at lower learning rates. Hence, a range of [0.1,0.5] proved to be ideal.

Part 1 –

The training datapoints themselves are chosen as the centres for the kernel function, meaning that there will be 352 nodes in the hidden layer of our neuron. We obtain the following mean squared errors across the chosen spread values.

Spread	Training MSE	Test MSE
0.1	0.00017	0.43679
0.15	0.00879	0.12341
0.2	0.03687	0.10434
0.25	0.05584	0.11113
0.3	0.07492	0.11700
0.35	0.09562	0.13001
0.4	0.13509	0.13891
0.45	0.19105	0.27198
0.5	0.32699	0.67586





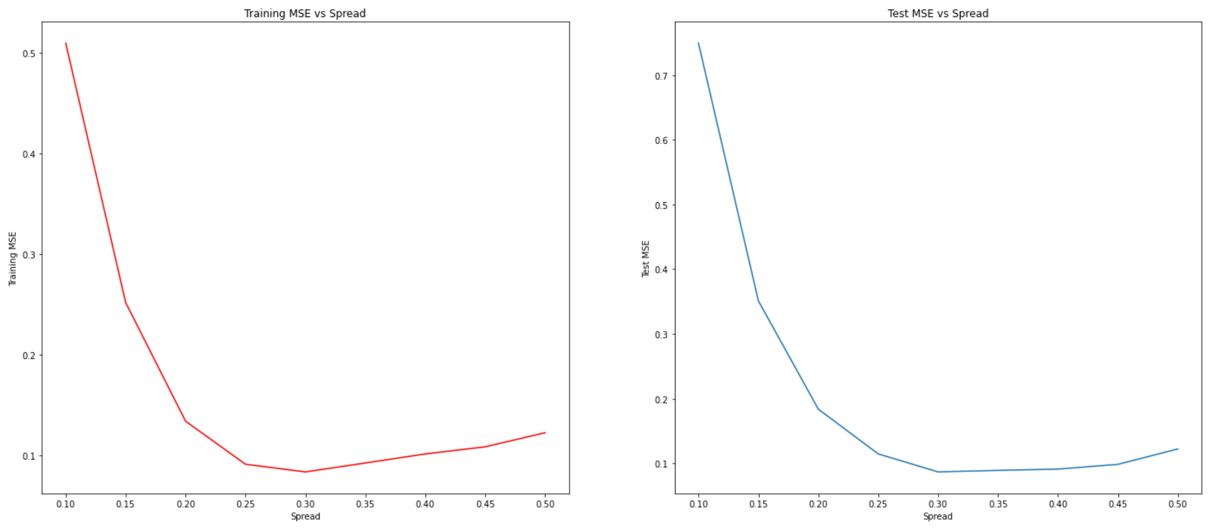
The left red curve is the MSE on the training set while the right blue curve is MSE on the test set, both plotted against spread. As expected, the training MSE overall is quite lower when compared to the test MSE because the network is being trained and the weights being updated according to the error in output for the training set. The test set contains unseen data for the model and hence provides a more real world generalized look at performance.

Lowest test MSE	= 0.10434
Spread at lowest MSE	= 0.2
Average training time	= 136.6 seconds

Part 2.a) –

We randomly chose 150 datapoints in our training data as our centres, meaning we now have 150 nodes in our hidden layer. We obtain the following mean squared errors across the chosen spread values.

Spread	Training MSE	Test MSE
0.1	0.50989	0.74942
0.15	0.25172	0.35096
0.2	0.13364	0.18362
0.25	0.09080	0.11441
0.3	0.08316	0.08656
0.35	0.09204	0.08891
0.4	0.10104	0.09091
0.45	0.10815	0.09825
0.5	0.12216	0.12204

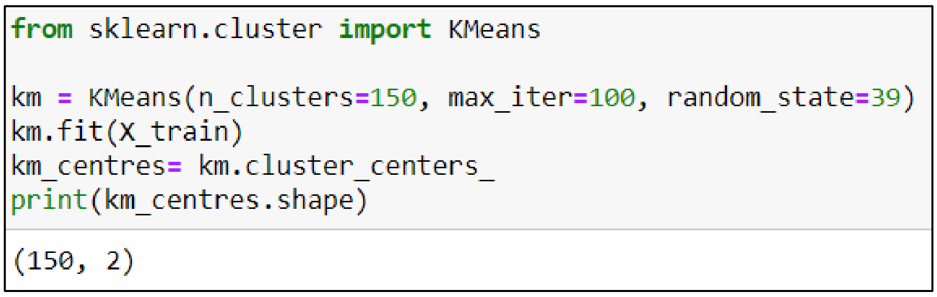


The left red curve is the MSE on the training set while the right blue curve is MSE on the test set, both plotted against spread. Here, the differences in range of MSE between the two is not large indicating that our model is generalizing much better for data. The curves obtained here are very similar in characteristics, showing a somewhat rectangular hyperbolic structure. The most surprising part is that both the training and test MSE’s are lowest at the same spread value.

Lowest test MSE	= 0.08656
Spread at lowest MSE	= 0.3
Average training time	= 59.9 seconds

Part 2.b) –

Here, we find 150 centres for our kernel functions by using K-Means Clustering, an unsupervised machine learning algorithm. We use the implementation of the same from the Scikit-Learn library, shown below.

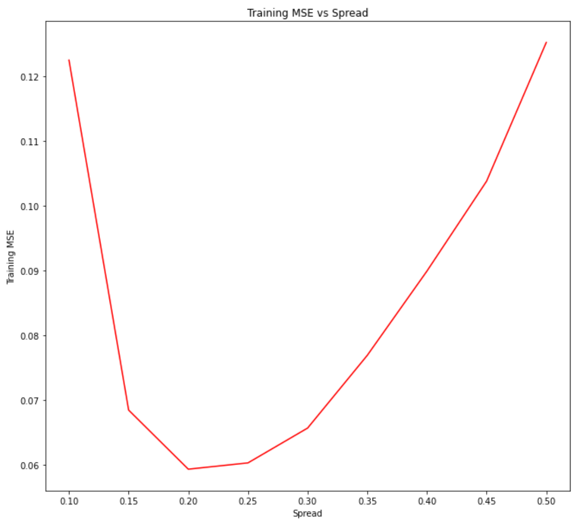


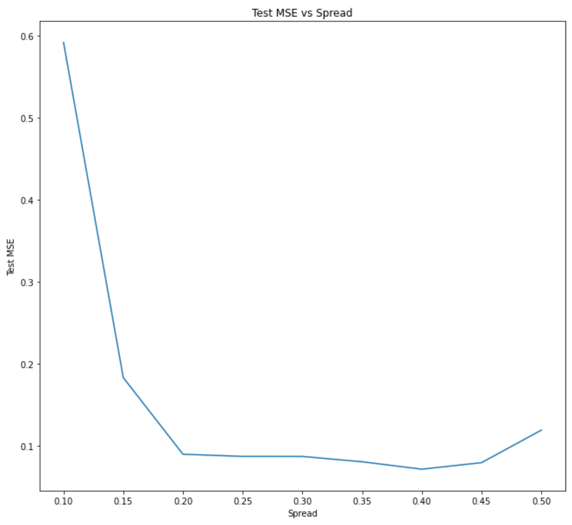
```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=150, max_iter=100, random_state=39)
km.fit(X_train)
km_centres= km.cluster_centers_
print(km_centres.shape)
```

```
(150, 2)
```

After the centres are found, the usual procedure is followed and we obtain the following mean squared errors across the chosen spread values.

Spread	Training MSE	Test MSE
0.1	0.12252	0.59130
0.15	0.06845	0.18347
0.2	0.05932	0.09016
0.25	0.06029	0.08753
0.3	0.06567	0.08750
0.35	0.07691	0.08100
0.4	0.08992	0.07198
0.45	0.10382	0.07991
0.5	0.12525	0.11944

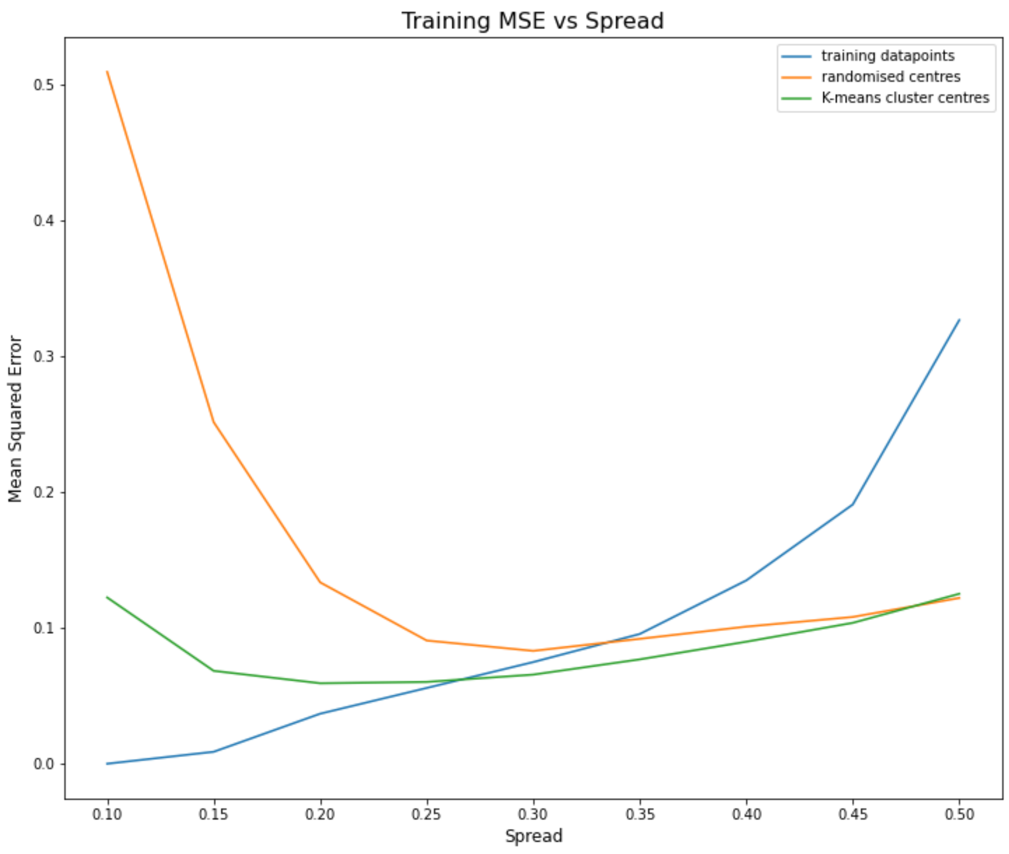


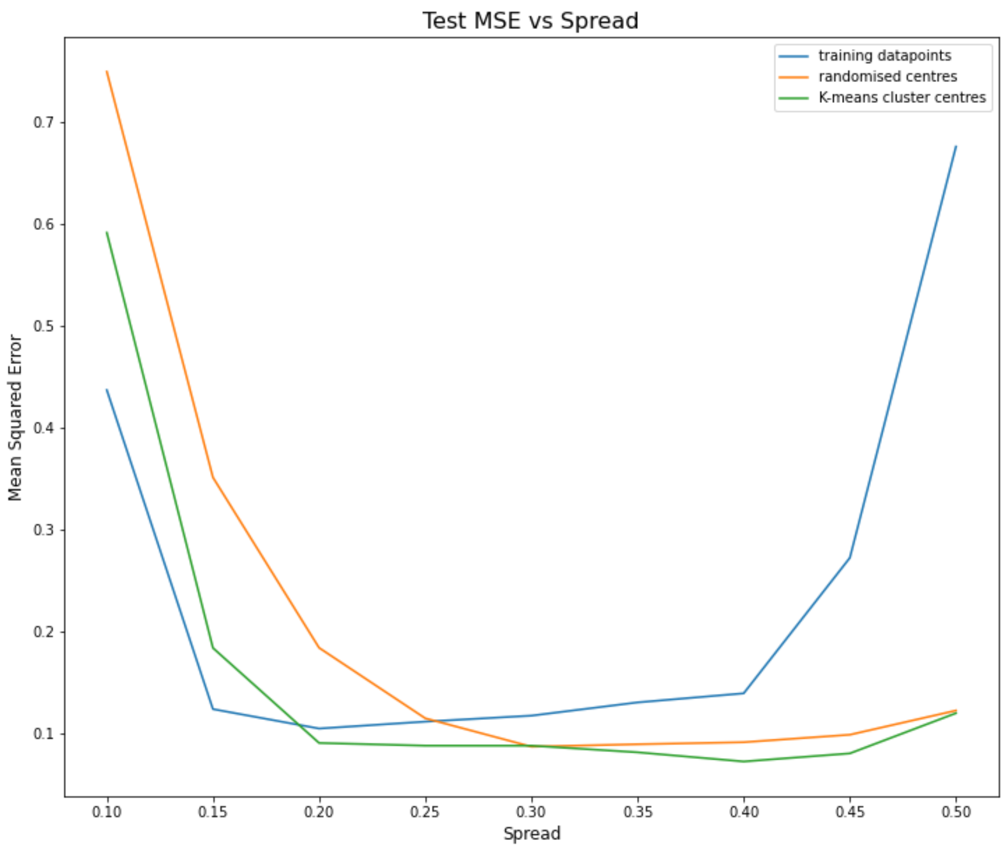


The left red curve is the MSE on the training set while the right blue curve is MSE on the test set, both plotted against spread. Unlike the randomized centres, the differences in range of MSE between the two is larger. While the training MSE curve follows a parabola and stays at a local minimum for a very short time, the test MSE is more of a rectangular hyperbola and stabilizes a bit around a local minimum before showing signs of increase.

Lowest test MSE	= 0.07098
Spread at lowest MSE	= 0.4
Average training time	= 58.4 seconds

Comparison Graphs –





Conclusion–

- The lowest mean squared error obtained is 0.07098 at a constant spread of 0.4 across all 150 kernel functions and the centres of the kernel functions found by using **K-Means Clustering** algorithm. This can be the procedure of choice, but some room for improvements like reducing overfitting by hyperparameter tuning can be made if needed.
- Choosing **randomized centres** gave us an MSE very close to that obtained by K-Means, while also providing better generalization in the model as the training error is much closer in value to the test error. However, this result is heavily dependent on the choice of centres and rerunning with a different randomization seed might give worse results making this method easily fallible.
- The first method of choosing all the **training datapoints as centres** not only gave the worst test MSE, it also suffers from overfitting and requires twice the computation time as the rest of the methods due to inherently having more nodes in the hidden layers.