INTRODUCTION

A common artificial neural network type for regression and classification issues is the Radial Basis Function Network (RBFN). The Radial Basis Function Network (RBFN) is composed of three layers: an input layer, a hidden layer of RBFs, and an output layer.

Usually, supervised learning and clustering methods are combined to train the RBFN. The supervised learning algorithm is used to learn the weights of the output layer, and the clustering algorithm is utilized to locate the centroids of the RBFs.

In this study, we will cover an implementation of the RBFN that initializes the RBFs using k-means clustering and learns the weights using the Least Mean Squares (LMS) technique. We will give a general description of the code and go through a detailed example of how to utilize it to resolve a binary classification issue.

An implementation of a machine learning technique that uses a neural network with radial basis functions to learn patterns from data is the RBFN (Radial Basis Function Network) class

The methods that we used in our implementation of the Radial Basis Function Network are:

1. __init__(self, k, lr=0.1, epochs=1000): This is the constructor of the RBFN class, which initializes the instance variables k, lr, and epochs. The k parameter specifies the number of centroids to use for the RBFN, while lr specifies the learning rate for the LMS algorithm and epochs specifies the number of epochs to train the LMS algorithm.

2. k_means_clustering(self, X): This method performs the K-means clustering algorithm to find the centroids of the RBFN. Given a matrix X of input data, the algorithm first initializes k centroids randomly, and then iteratively updates the centroids until convergence. During each iteration, the algorithm assigns each data point in X to its nearest centroid and then updates the centroids to be the mean of the data points assigned to each centroid. The final centroids are stored in the self.centroids variable, while the variances of each RBF are stored in the self.variances variable.

   And we set the spreads as the variance of the centroids.

3. lms(self, X, y): This method trains the RBFN using the LMS algorithm. Given a matrix X of input data and a vector y of corresponding output labels, the algorithm first initializes the weights randomly. It then iteratively updates the weights using the LMS algorithm, which calculates the RBFN output for each input in X and compares it to the corresponding label in y. The difference between the predicted and actual output is used to update the weights, with the learning rate lr controlling the rate of convergence. The final weights are stored in the self.weights variable.
   The RBF we use here is the Gaussian RBF

   Which can be written by the formula:

$$Phi(x) = exp((x - centroid)^2)/variance))$$

4. fit(self, X, y): This method is used to fit the RBFN to the input data X and corresponding labels y. It first calls the k_means_clustering method to find the centroids and variances of the RBFN, and then calls the lms method to train the RBFN.

5. predict(self, X): This method predicts the output labels for a given matrix X of input data, using the weights and centroids that were learned during the training phase. It first calculates the RBFN output for each input in X and then rounds the output to the nearest integer before returning it as a numpy array of predicted labels.

Worked Example

Data:

popular dataset for classification in machine learning is the Iris dataset. It includes 150 iris flower samples with the four characteristics of sepal length, sepal width, petal length, and petal width. The iris flower's species, which has three possible values (setosa, versicolor, or virginica), is the target variable.

methodology :

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create an RBFN with 10 centroids
rbfn = RBFN(k=10)

# Train the RBFN on the training data
rbfn.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = rbfn.predict(X_test)

# Calculate the accuracy of the predictions
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

✓ 2.4s

```
Accuracy: 0.9111111111111111
```

We utilized the predict() method to make predictions on the testing data after training the RBFN. Using the accuracy score() method from the scikit-learn library, we determined the forecasts' accuracy.

Results

We have achieved an accuracy of 91.1% on the testing data using the RBFN. This is a good accuracy considering the simplicity of the RBFN and the small size of the dataset.
But it might differ everytime because the selection of the centroids in the k means clustuing is random and it wont always give us the best result.

Conclusion

We have provided a Python implementation of a Radial Basis Function Network (RBFN) for categorization in this documentation. It is frequently utilized in applications like image identification, speech recognition, and financial forecasting because the RBFN is a potent tool for data classification.

Using hyperparameters for the number of centroids, learning rate, and epochs, our version of the RBFN is made to be adaptable and simple to use. The RBFN can be trained using the fit method on fresh data, and predictions can be made using the predict method.

The RBFN approach's capacity to learn non-linear decision boundaries is one of its main advantages, making it suitable for a variety of classification applications. However, it has limitations and might be sensitive to the type and volume of training data, much like any machine learning method.