Binary classifier using Radial Basis Function Network

Nikolaos Karapatsias 10661 (Department of Electrical and Computing Engineering AUTH)

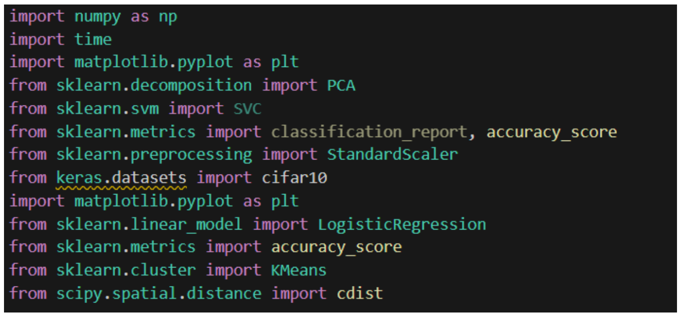
This document provides an implementation of a binary classifier using Radial Basis Network (RBF) which was implemented nearly from scratch. This model is trained and tested with two out of 10 classes of the dataset of Cifar-10 which is consisted of 50.000 training samples and 10.000 testing samples all cassified in 10 groups. More details can be found on cifar website

I. ANALYZING THE CODE

a) *Preperation code:*

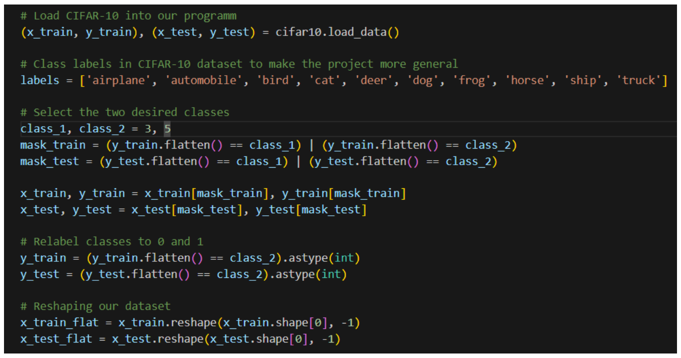
The preperation code is nearly the same with the one used on the previous project with Support Vector Machines (SVM) model as shown below:

- Firstly we are importing all of the required libraries such as scikit learn, Numpy etc:



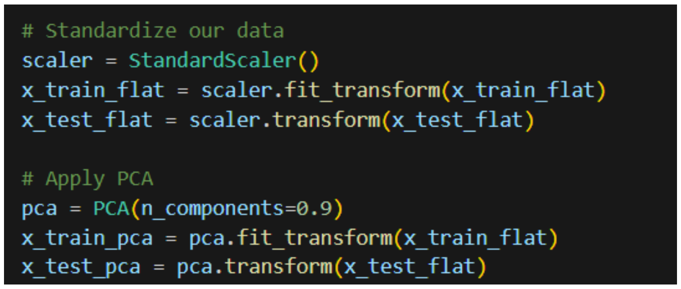
```python
import numpy as np
import time
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score
from sklearn.preprocessing import StandardScaler
from keras.datasets import cifar10
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
```

- From there, we are ready to load Cifar10 dataset from Tensorflow framework and filtering out the two desired classes which are for this implementation "Dog" and "Cat" with 4th and 5th class respectively. Finally we are relabeling the two y arrays for simplicity reasons by setting 0 for "Cat" class and 1 for "Dog" classification and also reshaping our dataset to fit the next step.



```python
# Load CIFAR-10 into our programm
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Class labels in CIFAR-10 dataset to make the project more general
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# Select the two desired classes
class_1, class_2 = 3, 5
mask_train = (y_train.flatten() == class_1) | (y_train.flatten() == class_2)
mask_test = (y_test.flatten() == class_1) | (y_test.flatten() == class_2)

x_train, y_train = x_train[mask_train], y_train[mask_train]
x_test, y_test = x_test[mask_test], y_test[mask_test]

# Relabel classes to 0 and 1
y_train = (y_train.flatten() == class_2).astype(int)
y_test = (y_test.flatten() == class_2).astype(int)

# Reshaping our dataset
x_train_flat = x_train.reshape(x_train.shape[0], -1)
x_test_flat = x_test.reshape(x_test.shape[0], -1)
```

- Before we proceed to RBFNN creation, we are scaling both x_test and x_train arrays in order to have a mean of 0 and a standard deviation of 1. Also, after that, we are applying Principal component analysis (PCA) aiming to lower our dataset's dimension without loosing more than 10% of information:

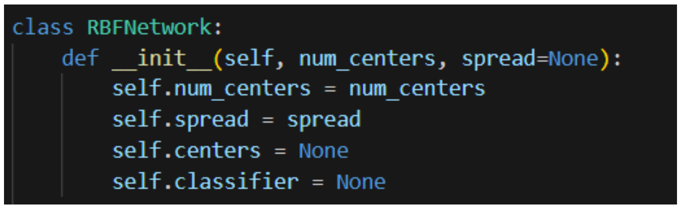


```python
# Standardize our data
scaler = StandardScaler()
x_train_flat = scaler.fit_transform(x_train_flat)
x_test_flat = scaler.transform(x_test_flat)

# Apply PCA
pca = PCA(n_components=0.9)
x_train_pca = pca.fit_transform(x_train_flat)
x_test_pca = pca.transform(x_test_flat)
```

b) *Building the RBF Network:*

Now that we got all data in the format we need them, we are ready to start creating the functions that will implement the Radial Basis Function Network.

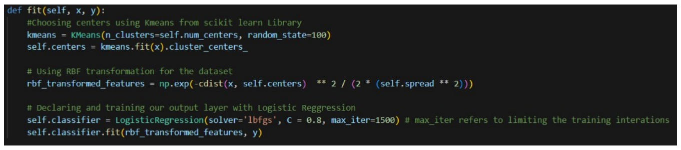
-Firstly we are creating a class named RBFNN and its initializer (constructor) where we are initializing some variables we'll need to use later:



```python
class RBFNetwork:
    def __init__(self, num_centers, spread=None):
        self.num_centers = num_centers
        self.spread = spread
        self.centers = None
        self.classifier = None
```

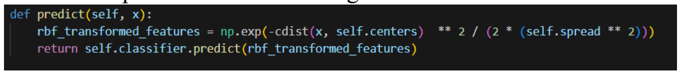
-Firstly we are implementing a function which performs an RBF transformation to our inputed dataset using the Gaussian function. Note that the cdist() function calculates the distance between all the data points and all the centers of our network.

-From there, we are ready to implement the function that will train our model using K-Means algorithm to produce the centroids, use .fit(x).cluster_centers_ to fit/train these centroids to our dataset. In the next step, we are applying an RBF transformation to our dataset using the predefined centroids. Note that the RBF transformation is done with a Gaussian function where x' is each centroid and σ^2 is the square of spread. We are calculating the distance between datapoint's and centroids by using the function `cdist()`. Last but not least we are producing an output layer which implements a logistic regression and training it by using "Lbfgs " optimizer as this was considered the best option for this type of dataset.



```python
def fit(self, x, y):
    #Choosing centers using Kmeans from scikit learn Library
    kmeans = KMeans(n_clusters=self.num_centers, random_state=100)
    self.centers = kmeans.fit(x).cluster_centers_

    # Using RBF transformation for the dataset
    rbf_transformed_features = np.exp(-cdist(x, self.centers)  ** 2 / (2 * (self.spread ** 2)))

    # Declaring and training our output layer with Logistic Reggression
    self.classifier = LogisticRegression(solver='lbfgs', C = 0.8, max_iter=1500) # max_iter refers to limiting the training interations
    self.classifier.fit(rbf_transformed_features, y)
```

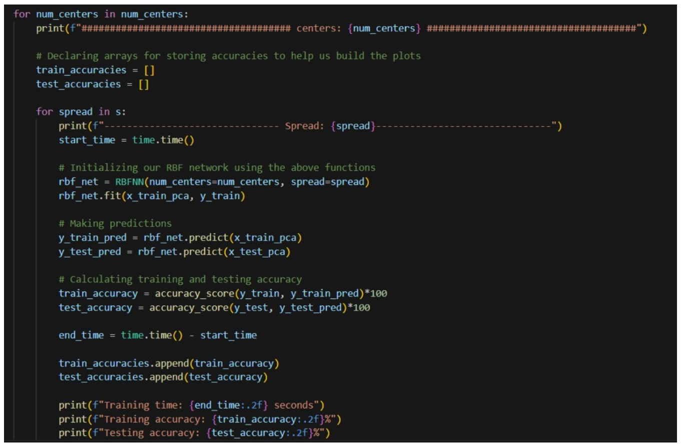
- Finally, once our RBF network is ready, we are declaring a function to make the predictions to the testing set:



```python
def predict(self, x):
    rbf_transformed_features = np.exp(-cdist(x, self.centers)  ** 2 / (2 * (self.spread ** 2)))
    return self.classifier.predict(rbf_transformed_features)
```

c) Training the RBF network and making predictions:

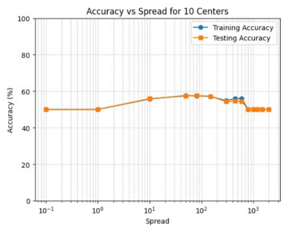
Once everything is set up, we are ready to use all these functions to create our Radial Basis Function network and calculate the training and testing accuracy for various values:

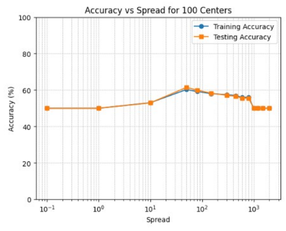


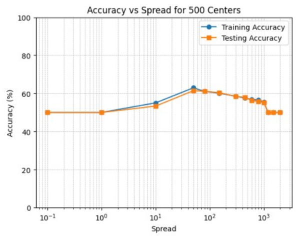
```python
for num_centers in num_centers:
    print(f"##################################### centers: {num_centers} #####################################")

    # Declaring arrays for storing accuracies to help us build the plots
    train_accuracies = []
    test_accuracies = []

    for spread in s:
        print(f"------------------------------ Spread: {spread}------------------------------")
        start_time = time.time()

        # Initializing our RBF network using the above functions
        rbf_net = RBFNN(num_centers=num_centers, spread=spread)
        rbf_net.fit(x_train_pca, y_train)

        # Making predictions
        y_train_pred = rbf_net.predict(x_train_pca)
        y_test_pred = rbf_net.predict(x_test_pca)

        # Calculating training and testing accuracy
        train_accuracy = accuracy_score(y_train, y_train_pred)*100
        test_accuracy = accuracy_score(y_test, y_test_pred)*100

        end_time = time.time() - start_time

        train_accuracies.append(train_accuracy)
        test_accuracies.append(test_accuracy)

        print(f"Training time: {end_time:.2f} seconds")
        print(f"Training accuracy: {train_accuracy:.2f}%")
        print(f"Testing accuracy: {test_accuracy:.2f}%")
```

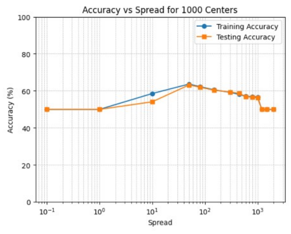
II. RESULTS

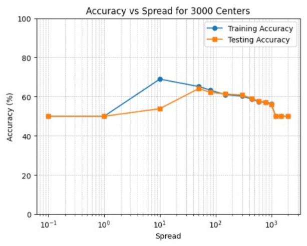
a) Plots: On this section, we are going to represent and analyze the results of our RBF model. We trained our model for a variety values both for centroids, from 10 till 10.000, and spread, from 0.1 till 2000. Note that in the below charts the blue curve represent the accuracy of our model in the training set, while the orange one represent the accuracy in the testing set.

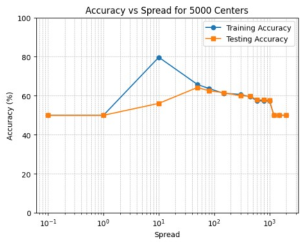


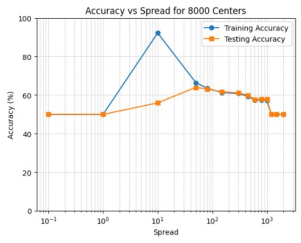


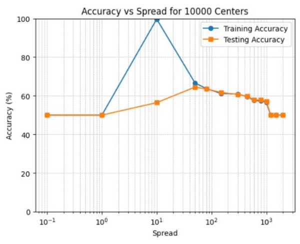












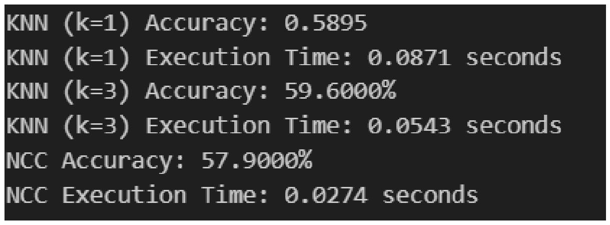
b)Observations: In the above plots we are noticing that our model stucks on nearly 50-55% of testing accuracy for a set of spread values no matter the centroids number which means that the model is mostly guessing. This is done for spread lower than 1 and higher than 600. Also we can clearly see that for spread = 10 our model is driven to overfitting as the training accuracy approaches 100% at a high rate, when the number of centers is getting bigger. Lastly we have to notice that in most of the cases the training and testing accuracies are really close, which is typical for RBF networks.

c) Best performance: The best testinc accuracy we achieved was 64,25% without overfiting as the training accuracy is 65.77% for spread = 50 and number_of_centroids = 5000 in 76 seconds.

c) Execution time: The execution time varied for each situation but for the values that the model didn't overfit, underfit or guessing it took briefly around from 70 till 120 seconds to execute the results.

III. VS KNN AND NCC

We’ve implemented Nearest Centroid Classifier (NCC) and K-Nearest Neighbors (KNN) for this binary classification problem in a previous project, and the results are the below:

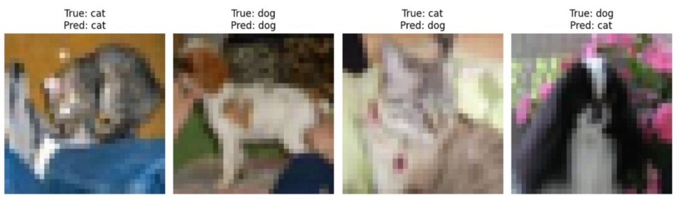


```
KNN (k=1) Accuracy: 0.5895
KNN (k=1) Execution Time: 0.0871 seconds
KNN (k=3) Accuracy: 59.6000%
KNN (k=3) Execution Time: 0.0543 seconds
NCC Accuracy: 57.9000%
NCC Execution Time: 0.0274 seconds
```

In comparison with our RBF Network with spread = 50 and number_of_centroids = 5000 the RBF has a slightly better testing accuracy than these two algorithms of about 6-8% but with a way worse execution time of 76 seconds while both KNN and NCC have less than a second.

IV. CORRECT AND WRONG CLASSIFICATION

In this section, we are providing some image examples that were correctly and wrongly classified from our RBF network model with parameters as: sigma = 50, centroids = 5000



V. GENERATING CENTROIDS RANDOMLY

In this part, we are going to follow a different path to generate the centroids. Instead of using K-means clustering algorithm, we are going to produce them randomly.

-Code: The code is nearly the same with the one we used previously but with one small difference. In the .fit() function inside the RBFNN class we are replacing:

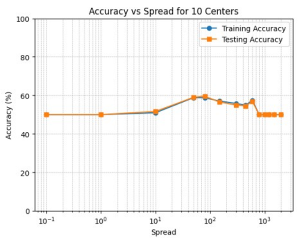
```
kmeans = KMeans(n_clusters=self.num_centers, random_state=100)

self.centers = kmeans.fit(x).cluster_centers_
```

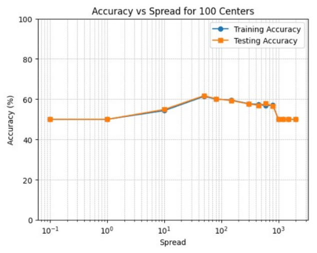
with

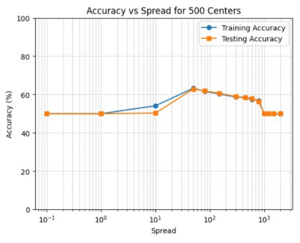
```
random_indices=np.random.choice(X.shape[0], self.num_centers, replace=False)

self.centers = X[random_indices]
```

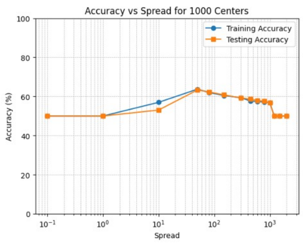
In that way, we are randomly setting some dataset points as our model’s centroids.

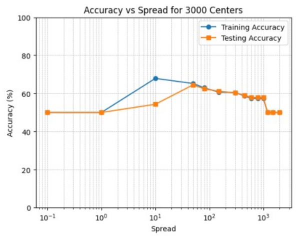
-Results: We used the same exact values for sigma and number of centroids as we used on the previous model. The plots are shown below:

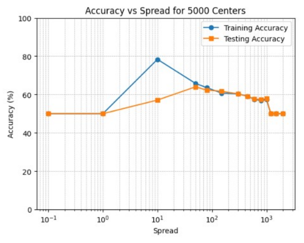


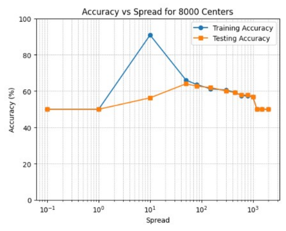


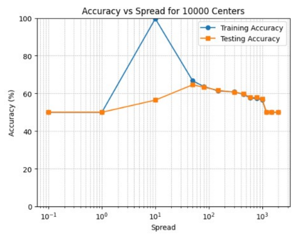












-Observations: As we can see, the results are really close to the previous model with the K-Means centroid generation. Again it stucks on 50-55% of testing accuracy for a set of spread values no matter the centroids number which means that the model is guessing. The best training accuracy we achieved was again for spread = 50 and number_of_centroids = 3000 at 64.5%. The training accuracy for that is 65.21% which means that the model is not close to overfiting.