Machine Learning Coursework Report

Peter David Fazekas, ID:U1705902, University of Warwick 04/12/2020

Introduction

This coursework focuses on designing and implementing a multi-class perceptron algorithm and applying it to the IRIS dataset to classify each types of iris plants. The initial data is not linearly separable so when running the perceptron algorithm it should not be able to achieve a 100% accuracy. I will be using Principal Component Analysis or (PCA) to project the data along the 2 and 4 top principal components and investigate if classifying the data in 2-D space improves the accuracy over using all 4 projected components. To further improve the accuracy of the model, I use a kernel to define a non-linear manifold learning technique. It is called the Radial Basis Function (RBF) Kernel which projects the data into a a higher dimensional subspace to attempt to further separate the data which would allow it to be separated by a hyper-plane.

Part A: Data visualization

First, the dataset is imported using the Python library Pandas, this allows to easily manipulate the data and gain some insight into how the values are distributed. A code snippet is provided to show how it is done.

Listing 1: Importing the dataset

```
import pandas as pd
from numpy.linalg import svd
import matplotlib.pyplot as plt
import numpy as np
df = pd.read_csv("iris.csv")
df.columns =["sepal length","sepal width","petal length","petal width","↩
    classes"]

X=df.iloc[:,(df.columns!="classes")]
y=df.loc[:,(df.columns=="classes")]
#use pandas to turn the classes into integers
y=pd.factorize(y['classes'])[0]
print("X has shape:",X.shape)
print("y has shape:",y.shape)
```

The full code output for this part of the assignment is shown in Figure 1 below after centering

the data around the mean. This is needed for the RBF Kernel. The minimum and maximum variance of the original dataset can be used to find the range of gammas that the gridsearch function will use to find the best gamma and the number of top principal components to use to achieve the highest accuracy. The kernel that will be used later on has the form $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||^2)$. The reason for calculating the minimum and maximum variance is because the range of gammas are given by $[\gamma_{min} = \frac{1}{2(\sigma^2_{max}+\varepsilon)}, \gamma_{max} = \frac{1}{2(\sigma^2_{min}+\varepsilon)}]$ where ε is a small number. As stated in the coursework specification, ε is set to 0.05.

```
X has shape: (149, 4)
y has shape: (149,)
minimum variance: 0.18792127698167987
maximum variance: 3.096372211137314
training data is:
   sepal length  sepal width  petal length  petal width
0           4.9          3.0           1.4          0.2
1           4.7          3.2           1.3          0.2
2           4.6          3.1           1.5          0.2
3           5.0          3.6           1.4          0.2
4           5.4          3.9           1.7          0.4


information about the initial centered training data is:
       sepal length   sepal width  petal length   petal width
count  1.490000e+02  1.490000e+02  1.490000e+02  1.490000e+02
mean  -1.704826e-15 -8.464519e-16 -2.187661e-15 -8.643347e-16
std    8.285941e-01  4.334989e-01  1.759651e+00  7.612920e-01
min   -1.548322e+00 -1.051007e+00 -2.774497e+00 -1.105369e+00
25%   -7.483221e-01 -2.510067e-01 -2.174497e+00 -9.053691e-01
50%   -4.832215e-02 -5.100671e-02  6.255034e-01  9.463087e-02
75%    5.516779e-01  2.489933e-01  1.325503e+00  5.946309e-01
max    2.051678e+00  1.348993e+00  3.125503e+00  1.294631e+00
```

Figure 1: Code output for the information about the dataset itself after centering

I now turn back to PCA which allows the visualization of high dimensional data in 2-D along the top 2 Principal components. A code snippet will be given to show how this can be implemented.

Listing 2: Using PCA to visualise the data

```
#when using PCA we need to subtract the mean
mean = np.mean(X,axis=0)
def PCA(X,n,mean):
    #X is training data, n is the number of top PC's to return
    X_t = X.copy()
    #subtract the mean for SVD
    X_t-=mean
    #get matrices using SVD
    U,S,V = svd(X_t,full_matrices=True)

    #only return the top n principal components
    print("V has shape",V.shape)
    return V[:n]
print("Top 2 PC's are:",PCA(X,2,mean))
```

```
PC_2 = PCA(X,2,mean)
#project the data into 2-d space
X_trans=np.dot(X,PC_2.T)
```

By running the PCA algorithm and projecting along the top 2 PC's, it can be seen that the output on Figure 2 shows which classes of the Iris data set are linearly separable and which are not. From looking at this plot we see that a linear classifier should not be able to achieve a 100% accuracy, no matter how long it is trained for. Classes 2 and 3 are not linearly separable whereas class 1 can be separated by a hyper-plane from all the other classes. In 2-D it is just a straight line.

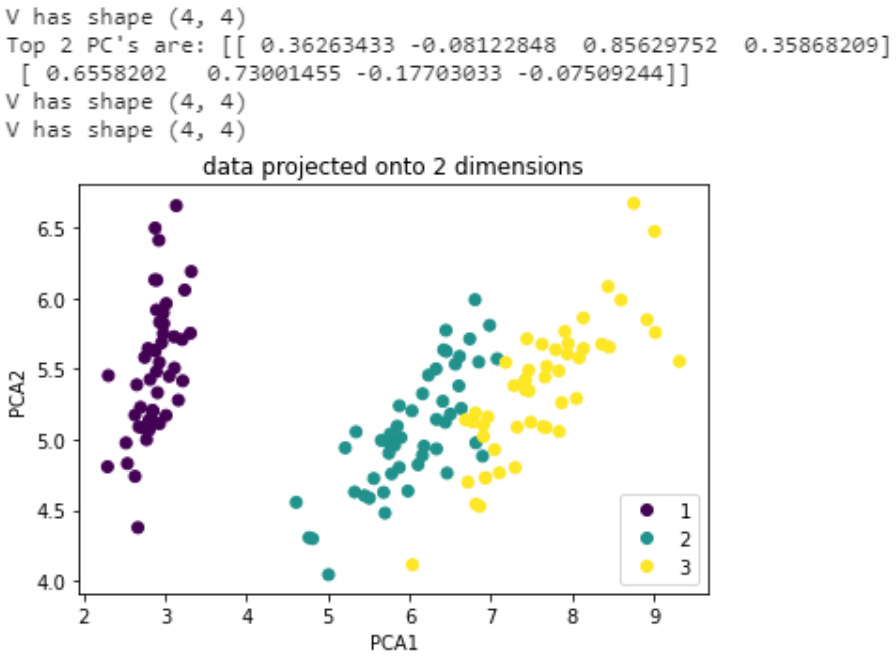

Figure 2: Code output for the output of the PCA algorithm

Part B: Perceptron algorithm

This part of the assignment focuses on designing and implementing the Multi-Class Perceptron algorithm. Below I will provide the pseudo code from the assignment specification. The implemented algorithm can be found in the Jupyter Notebook.

Listing 3: Pseudo code for perceptron algorithm

```
for it=1 to last iteration
    W_it=solution at iteration it
        for all training examples in random order
            classify training example i
            if example i misclassified
                refine W_it
            end if
```

```
        end for
    use refined W_it for next iteration
end for
```

The code snippet below is a utility function to set up the weight and bias matrices. It makes sure they have the correct shape which should be $(2, 3)$ for the weight matrices and $(3, 1)$ for the bias vector when using the top 2-PC's. Similarly when using all 4 top PC's the weight and bias will have shapes $(4, 3)$ and $(3, 1)$ respectively.

Listing 4: helper function to initialize the weights for the algorithm

```
#initialize the weight arrays as 0,each row corresponds to 1 class
uniqueC=len(np.unique(y))
print("number of unique classes are:",uniqueC)
def initialize(s1,s2):
    w = np.zeros((s1,s2))
    b= np.zeros((s2,1))
    return w,b
```

To run the perceptron algorithm and compare how it performs when top 2/top 4 PC's are used we use the code below. The print statements are used as a sanity check to make sure the dimensions add up, as well as displaying the final accuracy after 50 iterations. For more details and the rest of the code, please see the Jupyter Notebook.

Listing 5: Code to run the algorithm

```
w,b=initialize(X_trans.shape[1],uniqueC)
print("initial w is",w," with shape: ",w.shape)
print("initial b is",b," with shape: ",b.shape)
weight_PC=MC_perceptron(50,X_trans,y,w,b,Print =True)[0]
print("\n")
#get final accuracy from err_arr[-1]
print("Final accuracy for projecting data along top 2 PC's: ",1-err_arr↩
    [-1])
print("\n")
w,b=initialize(X.shape[1],uniqueC)
print("initial w is",w," with shape: ",w.shape)
print("initial b is",b," with shape: ",b.shape)
MC_perceptron(50,X_4,y,w,b,Print =True)
print("Final accuracy after projecting to top 4 PC's: ",1-err_arr[-1])
```

The output of the algorithm is shown on Figure 3. We can see that after 50 iterations projecting along the top 2 PC's achieves an accuracy of 91%, on the other hand projecting along all 4 PC's achieves a 95% accuracy which is higher. During training the data is fed into the algorithm randomly when we update the weights and biases therefore each time it is run the results will be

different. Please note that sometimes projecting along the top 2 PC's will achieve a higher accuracy than projecting along all 4 PC's. To be more precise, the algorithm sometimes achieves a 93% accuracy when projecting along 2 PC's and only a 91% accuracy when using all 4 components. In both cases the accuracy on the training set is fairly high, we can see that the algorithm is doing well at separating the data after only 50 iterations. I have experimented with training the classifier for longer but there is a clear drop off in the improvement of the accuracy. After training for 500 iterations, the algorithm achieves around 93% and 95% when projecting along the top 2 and 4 PC's respectively. This indicates the fact that no matter how long the algorithm is run for, it is still unable to fully separate the data when using PCA.

```
initial w is [[0. 0. 0.]
 [0. 0. 0.]]  with shape:  (2, 3)
initial b is [[0.]
 [0.]
 [0.]]  with shape:  (3, 1)


Final accuracy for projecting data along top 2 PC's:  0.9194630872483222


initial w is [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]  with shape:  (4, 3)
initial b is [[0.]
 [0.]
 [0.]]  with shape:  (3, 1)
Final accuracy after projecting to top 4 PC's:  0.9530201342281879
```

Figure 3: Code output for the perceptron algorithm after 50 iterations

Figure 4 shows how the error decreases each iteration when the algorithm is running.

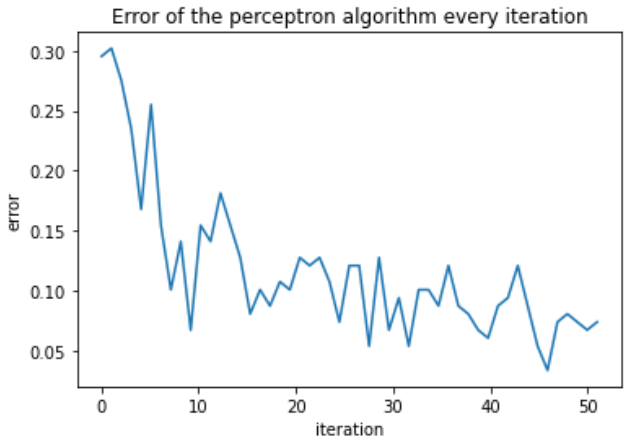

Figure 4: Error output every iteration

I have made a function which allows the visualization of the decision boundary in 2-D when the Iris data set is projected along the top 2 PC's as shown on Figure 5. The idea is to create a mesh along the top 2 PC's and predict every value and assign a color to each predicted class. When combined with the scatter plot of the projected data we can see where the algorithm thinks the "ground truth" decision boundary is. When a datapoint is in one of the 3 regoins, it will be assigned one of the 3 classes for the Iris flowers, "Iris Setosa," "Iris Versicolour," "Iris Virginica".

To obtain this image I have trained the perceptron algorithm for 500 epochs, and I believe this is a good sanity check to make sure the code is working as intended. Using PCA or taking 2-D slices of decision boundaries gives an intuition about how these algorithms "learn" and assign the classes to the data, especially when working with high dimensional data. But can we do better?

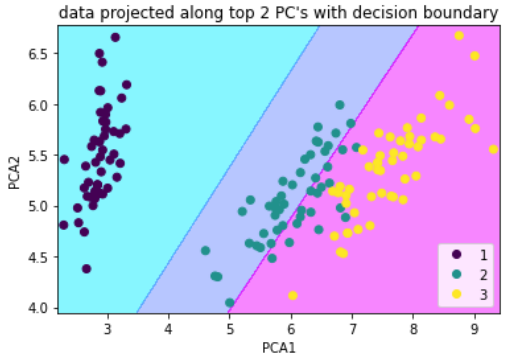

Figure 5: Output of the decision boundary function

Part C: Projection onto a high-dimensional subspace

Part C of the coursework focuses on implementing the Radial Basis Function Kernel. As stated in the introduction, this is a non-linear manifold learning technique which creates the non-linear version of PCA. We must compute a normalized Kernel Matrix $\tilde{\boldsymbol{K}}$ where $\tilde{\boldsymbol{K}} = \boldsymbol{K} - \boldsymbol{A}\boldsymbol{K} - \boldsymbol{K}\boldsymbol{A} + (\boldsymbol{A}\boldsymbol{K})\boldsymbol{A}$. Further note that the matrix $\boldsymbol{K}$ is computed from $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{X}^T$ where the entries are the dot product between all the training examples. More specifically the entries have the form $k_{i,j} = \boldsymbol{x}_i\boldsymbol{x}_j^T$ and i,j are the i-th and j-th training example respectively. When using the Kernel of the form $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||^2)$ the entries of matrix $\boldsymbol{K}$ are given by $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Please see the Jupyter Notebook for the implementation of the kernel algorithm. The kernel function is used to project the original data set into a high dimensional subspace and then a a grid search function will test out the range of gammas and top PC's to use. We can find the range of gammas to test from the equation $[\gamma_{min} = \frac{1}{2(\sigma^2_{max}+\varepsilon)}, \gamma_{max} = \frac{1}{2(\sigma^2_{min}+\varepsilon)}]$. From Figure 1, the minimum and maximum variance is obtained, which are 0.19 and 3.10 respectively. This gives a range of gammas approximately in the range $(0.16, 3.6)$. The purpose of using the RBF kernel is to project the data into a higher dimensional subspace than the dimensions of the data set. This means that when searching for the best Hyper-Parameters, the number of top PC's should be $(> d)$ where d is the dimensions of the data. When running the grid search function I have searched over 15 top PC's to use, starting from the top 53 PC's in increments of 1 each iteration. I have picked the starting top PC's to be 53 because it seems to be the value above which the classifier is able to achieve a 100% accuracy and and this can be used to compare to the performance of other hyper-parameter combinations. The gamma hyper-parameter is increased by 0.1 each iteration. The perceptron algorithm is run for 50 iterations each time and the error is stored in a matrix which has dimensions $(15, int(\frac{\gamma_{max}-\gamma_{min}}{0.1}) + 1)$. The output of the grid search function is visualised by plotting a heat-map, where each entry of the matrix has the

number of errors the algorithm made using a specific gamma and top PC's.

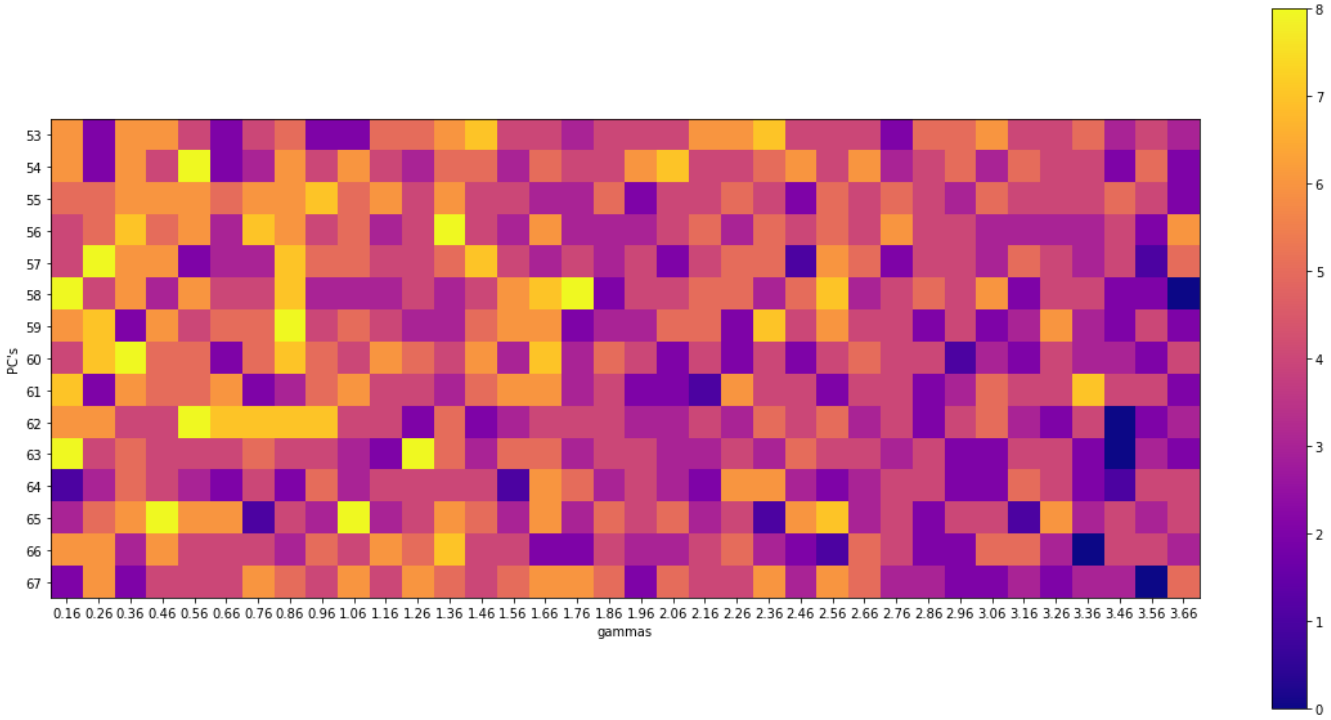

Figure 6: heat map of the gridsearch function

It can be seen from Figure 6 that generally, more top PC's used does not always increase the accuracy of the perceptron algorithm. Similarly using a larger value of gamma does not necessarily improves the performance of the classifier either, when projected into higher dimensions. The heat map is the darkest where the algorithm achieves a 100% classification accuracy. The yellow color indicates the combinations of gammas and top PC's used where the algorithm is having a hard time to fully separate the data w.r.t. other values. It can be seen that even when using more top PC's there are still values where the algorithm performs worse. When searching from 53 top PCs the worst error the algorithm achieves is 8 out of 149 training examples. When converted into a percentage, this corresponds to an accuracy of about 95%. Comparing this with the normal PCA from Part B, this combination of gammas and top PC's used perform in a similar way to projecting along the 4 PC's without using an RBF Kernel.

The conclusion is that specific combinations of Hyper-parameters for the RBF Kernel can greatly increase the performance of the perceptron algorithm when the data is not linearly separable, but obtaining a better classification accuracy will not necessarily happen by just guessing values of gammas and top PC's to use. A grid search function should be implemented when projecting data into higher dimension and the best hyper parameters should be found by running the machine learning algorithm over a range of gammas and top PC's to find the best one for the classification task at hand. This can get computationally expensive when dealing with more complex models but when implementing a linear multi-class perceptron algorithm, it does not take a long time to find the best parameters.

References

[1] https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs342

[2] coursework_cs342_2020-2021.pdf