Graduate Project
ID3 and PCA

Submitted By

Gargi Mrunal Kulkarni

(CWID: 893210922)

CPSC 483

Data Mining and Pattern Recognition

Fall, 2016

Prof: Kenytt Avery

Department of Computer Science

California State University, Fullerton

December 8, 2016

Table of Contents

Introduction 3
Algorithm and Its Implementation 3
  Iterative Dichotomiser 3 (ID3) 3
    About the Algorithm 3
    Programming Environment and Dataset 5
    Installation or Setup Steps 6
    Code in Action 6
  Principal Component Analysis 10
    About the Algorithm 10
    Programming Environment and Dataset 11
    Installation or Setup Steps 12
    Code in Action 13
Conclusion 19
References 19

Introduction

Data Mining is a developing technology which performs analysis of structured or unstructured data, to identify patterns and extract useful information from the same for decision making. There are several data mining algorithms invented to mine this big data to obtain valuable knowledge from it. Some of the data mining techniques are Association, Classification, Clustering, Prediction, Sequential patterns, decision trees, Combination, Long term processing etc. The data mining tools implement these algorithms for analysis and decision making. The different data mining tools available are Weka, Orange, Rapid Miner, R-Programming, Apache Mahout, Pyhton Scikit/Sklearn etc. (data mining, n.d.)

In this project, the decision tree algorithm ID3 (Iterative Dichotomiser 3) in R-Programming and PCA (Principal Component Analysis) in Python is implemented.

Algorithm and Its Implementation

In this project following algorithms were implemented:

1. Iterative Dichotomiser 3 (ID3) decision tree algorithm
    - Programming language: R-Programming
    - Dataset: Iris Dataset
2. Principal Component Analysis (PCA) algorithm
    - Programing language: Python
    - Dataset: Play tennis Dataset

Iterative Dichotomiser 3 (ID3)

About the Algorithm

In decision tree learning, ID3 (Iterative Dichotomiser 3) is an algorithm invented by Ross Quinlan used to generate a decision tree from a dataset. ID3 is the precursor to the C4.5 algorithm, and is typically used in the machine learning and natural language processing domains.
The ID3 algorithm begins with the original set ***S*** as the root node. On each iteration of the algorithm, it iterates through every unused attribute of the set ***S*** and calculates the entropy ***H(S)*** (or information gain ***IG(S)***) of that attribute. It then selects the attribute which has the smallest entropy (or largest information gain) value. The set ***S*** is then split by the selected attribute to produce subsets of the data. The algorithm continues to recurse on each subset, considering only attributes never selected before.
Recursion on a subset may stop in one of these cases:

- every element in the subset belongs to the same class (+ or -), then the node is turned into a leaf and labelled with the class of the examples
- there are no more attributes to be selected, but the examples still do not belong to the same class (some are + and some are -), then the node is turned into a leaf and labelled with the most common class of the examples in the subset
- there are no examples in the subset, this happens when no example in the parent set was found to be matching a specific value of the selected attribute.Then a leaf is created, and labelled with the most common class of the examples in the parent set.

Throughout the algorithm, the decision tree is constructed with each non-terminal node representing the selected attribute on which the data was split, and terminal nodes representing the class label of the final subset of this branch. (ID3_algorithm, n.d.)

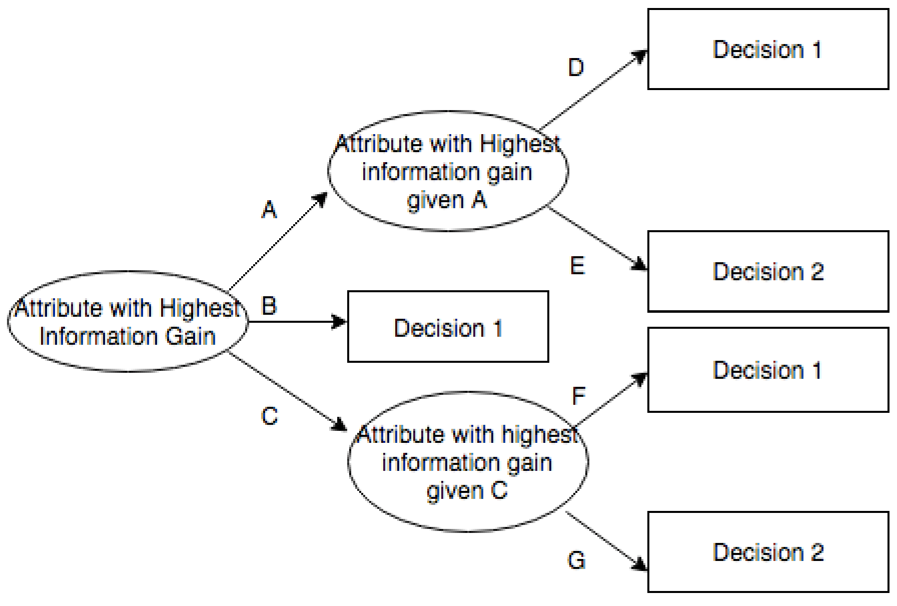

Attribute with Highest Information Gain
A
B
C
Attribute with Highest information gain given A
Decision 1
Attribute with highest information gain given C
D
E
F
G
Decision 1
Decision 2
Decision 1
Decision 2

Figure 1: ID3 algorithm

The summary of steps of the algorithm is as follows:

1. Calculate the entropy or information gain of every attribute using the data set ***S***.
2. Split the set ***S*** into subsets using the attribute for which entropy is minimum (or, equivalently, information gain is maximum)
3. Make a decision tree node containing that attribute
4. Recurse on subsets using remaining attributes.

Terminologies used – Entropy and Information Gain

Entropy

Entropy ***H(S)*** is a measure of the amount of uncertainty in the (data) set ***S*** (i.e. entropy characterizes the (data) set ***S***). (ID3_algorithm, n.d.)

$$H(S) = -\sum_{x \in X} p(x) \log_2 p(x)$$

Where,

S - The current (data) set for which entropy is being calculated
X - Set of classes in ***S***
p(x) - The proportion of the number of elements in class ***x*** to the number of elements in set ***S***

When ***H(S)***=0, the set ***S*** is perfectly classified (i.e. all elements in ***S*** are of the same class).In ID3, entropy is calculated for each remaining attribute. The attribute with the smallest entropy is used to split the set ***S*** on this iteration.

Information Gain

Information gain ***IG(A)*** is the measure of the difference in entropy from before to after the set ***S*** is split on an attribute ***A***. In other words, how much uncertainty in ***S*** was reduced after splitting set ***S*** on attribute ***A***. (ID3_algorithm, n.d.)

$$IG(A,S) = H(S) - \sum_{t\in T} p(t)H(t)$$

Where,

H(S) - Entropy of set ***S***
T - The subsets created from splitting set ***S*** by attribute ***A*** such that $S = \bigcup_{t\in T} t$
p(t) - The proportion of the number of elements in ***t*** to the number of elements in set ***S***
H(t) - Entropy of subset ***t***

In ID3, information gain can be calculated (instead of entropy) for each remaining attribute. The attribute with the largest information gain is used to split the set ***S*** on this iteration.

Programming Environment and Dataset

Programming Environment

The ID3 algorithm is coded in R-programming language. R provides the `data.tree` package which simplifies the operations related to decision tree. The `data.tree` package lets you create hierarchies, called `data.tree` structures. The building block of these structures are Node objects. The package provides basic traversal, search, and sort operations, and an infrastructure for recursive tree programming. You can decorate `Nodes` with your own fields and methods, so as to extend the package to your needs. The package also provides convenience methods for neatly printing and plotting trees. It supports conversion from and to `data.frames`, `lists`, and other tree structures such as `dendrogram`, `phylo` objects from the ape package, `igraph`, and other packages.
Technically, `data.tree` structures are bi-directional, ordered trees. Bi-directional means that you can navigate from parent to chidren and vice versa. Ordered means that the sort order of the children of a parent node is well-defined. (Glur, Introduction to data.tree, 2016)

About the dataset

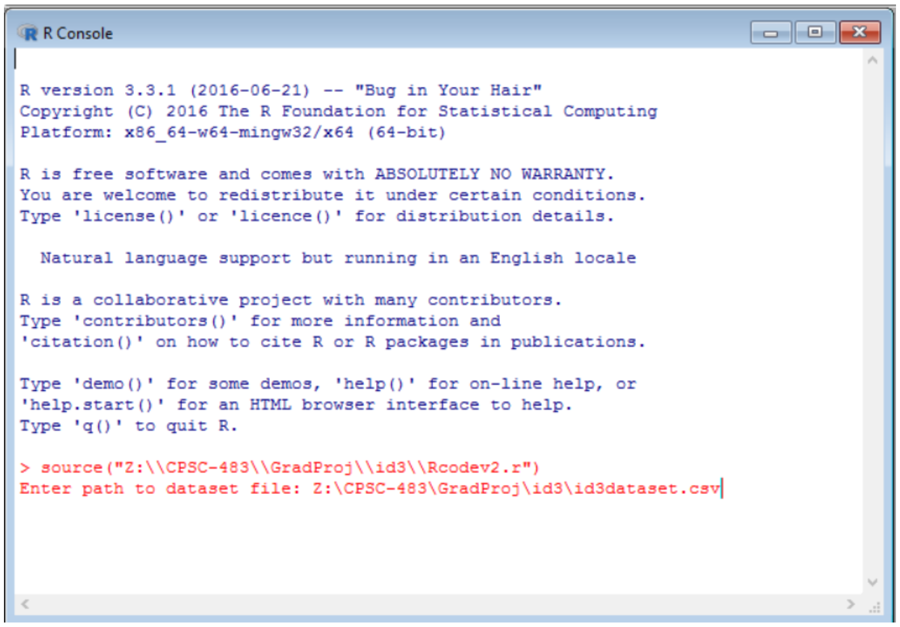
The dataset used for this simple and small play tennis dataset. The dataset is easily available online along with its ID3 tree calculation and tree generation. To understand the algorithm and verify the implementation in R, this dataset is used. The data is read in form of CSV from the program. [Separator is “;”]

The dataset information is as follows:

1. Number of Observations: 14
2. Number of Attributes: 5
   outlook; temperature; humidity; wind; play-tennis
3. Attribute information:
   - Input Variables: outlook; temperature; humidity; wind
   - Output Variable: play-tennis
4. Missing Attribute Values: None

Installation or Setup Steps

1. Download and install R from https://cran.r-project.org/
2. Open R and install package `data.tree` using command - `install.packages("data.tree ")`
3. Save the source code R file and Dataset CSV file on local machine.
4. Load the source R code File → Source R Code
5. Enter the path to dataset file.



```
R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> source("Z:\\CPSC-483\\GradProj\\id3\\Rcodev2.r")
Enter path to dataset file: Z:\CPSC-483\GradProj\id3\id3dataset.csv
```

Figure 2: Execution of the code

6. Call function `plot(tree)` to visualize the tree

Code in Action

Implementation

The implementation of the algorithm in R is as follows. (Glur, ID3 Classification using data.tree, 2015)

1. Load the library and data set.

```
library("data.tree")
file.path <- readline(prompt="Enter path to dataset file: ")
dataset = read.csv(file=file.path,header=TRUE,sep=';')
```

2. Define the helper functions to calculate information gain and entropy of set and subsets.

- To check whether node contains only one class or not function `IsPure()` is defined.

```
IsPure <- function(data) {
 length(unique(data[,ncol(data)])) == 1
}
```

- To calculate the information gain the `InformationGain()` is defined. It implements the information gain formula.

```
InformationGain <- function( tble ) {
  tble <- as.data.frame.matrix(tble)
  entropyBefore <- Entropy(colSums(tble))
  s <- rowSums(tble)
  entropyAfter <- sum (s / sum(s) * apply(tble, MARGIN = 1, FUN =
Entropy ))
  informationGain <- entropyBefore - entropyAfter
  return (informationGain)
}
```

- To calculate the entropy the `Entropy()` function is defined. It implements the entropy formula.

```
Entropy <- function( vls ) {
  res <- vls/sum(vls) * log2(vls/sum(vls))
  res[vls == 0] <- 0
  -sum(res)
}
```

- The `Predict()`function predicts the output for test data.

```
Predict <- function(tree, features) {
if (tree$children[[1]]$isLeaf) return (tree$children[[1]]$name)
child <- tree$children[[features[[tree$feature]]]]
return ( Predict(child, features))
}
```

3. Write the function that builds the ID3 tree recursively. The function `buildID3Tree()`is defined to generate tree. The function is called recursively to generate the subtrees. It implements the algorithm stated in above. It takes in root node and data, calculates the information gain for each attribute and assign node to attribute having highest information gain. Then it splits the data in subtrees, removing the node and repeats the process, calling function recursively, till it reaches the leaf nodes. It trains the tree with training dataset.

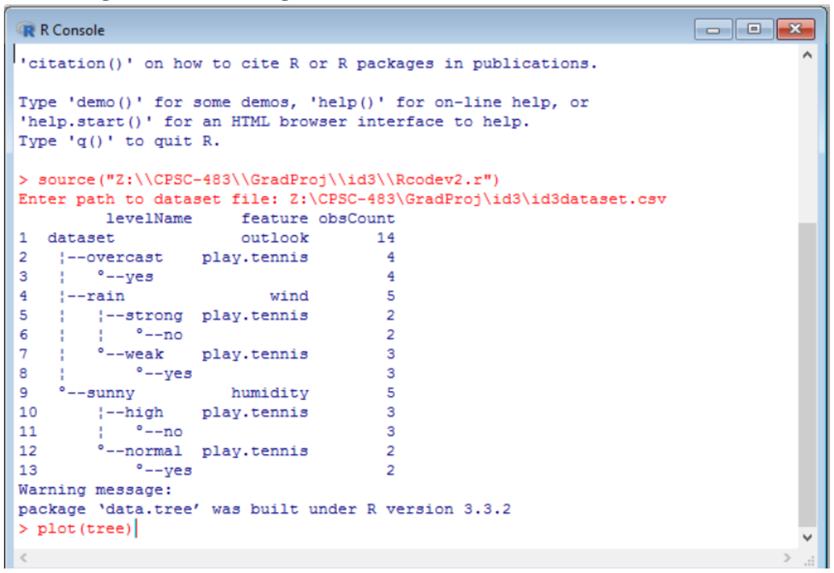
```
    buildID3Tree <- function(node, data) {

      node$obsCount <- nrow(data)

      #if the data-set is pure (e.g. all toxic), then
      if (IsPure(data)) {
        #construct a leaf having the name of the pure feature (e.g.
    'toxic')
       child <- node$AddChild(unique(data[,ncol(data)]))
        node$feature <- tail(names(data), 1)
        child$obsCount <- nrow(data)
        child$feature <- ''
      } else {
        #chose the feature with the highest information gain (e.g.
    'color')
        ig <- sapply(colnames(data)[-ncol(data)],
                function(x) InformationGain(
                  table(data[,x], data[,ncol(data)])
                  )
                )
        feature <- names(ig)[ig == max(ig)][1]
        node$feature <- feature

        #take the subset of the data-set having that feature value
        childObs <- split(data[,!(names(data) %in% feature)],
    data[,feature], drop = TRUE)

        for(i in 1:length(childObs)) {
          #construct a child having the name of that feature value (e.g.
    'red')

          child <- node$AddChild(names(childObs)[i])
          #call the algorithm recursively on the child and the subset
          buildID3Tree(child, childObs[[i]])
        }

      }

    }
```

4. Following code calls to the method to generate tree and display the output.

```
tree <- Node$new("dataset")
buildID3Tree(tree, dataset)
print(tree, "feature", "obsCount")
SetGraphStyle(tree, rankdir = "TB")
SetEdgeStyle(tree, arrowhead = "vee", color = "grey35", penwidth = 2)
SetNodeStyle(tree, style = "filled,rounded", shape = "box", fillcolor =
"lightblue",
            fontname = "helvetica", tooltip = GetDefaultTooltip)
Do(tree$leaves, function(node) SetNodeStyle(node, shape = "egg",fillcolor
= "cyan"))
plot(tree)
```

Output

The output of the code is give in below figures.



```
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> source("Z:\\CPSC-483\\GradProj\\id3\\Rcodev2.r")
Enter path to dataset file: Z:\CPSC-483\GradProj\id3\id3dataset.csv
         levelName      feature obsCount
1  dataset              outlook       14
2   ¦--overcast     play.tennis        4
3   ¦   °--yes                         4
4   ¦--rain                wind        5
5   ¦   ¦--strong   play.tennis        2
6   ¦   ¦   °--no                      2
7   ¦   °--weak     play.tennis        3
8   ¦       °--yes                     3
9   °--sunny           humidity        5
10      ¦--high     play.tennis        3
11      ¦   °--no                      3
12      °--normal   play.tennis        2
13          °--yes                     2
Warning message:
package ‘data.tree’ was built under R version 3.3.2
> plot(tree)
```

Figure 3: Code output – generated tree on R console

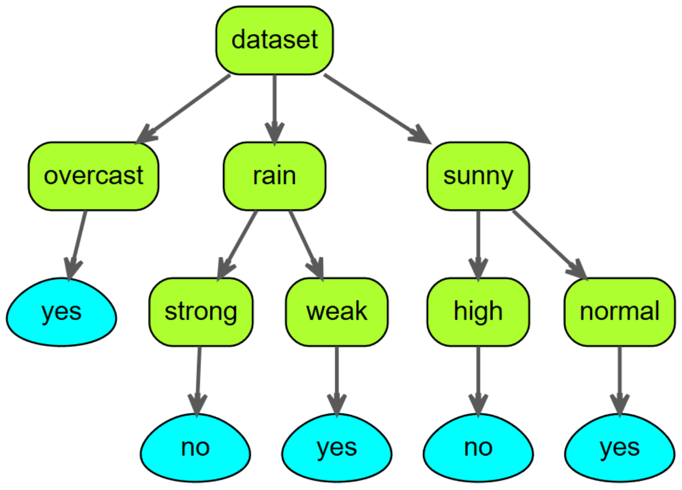
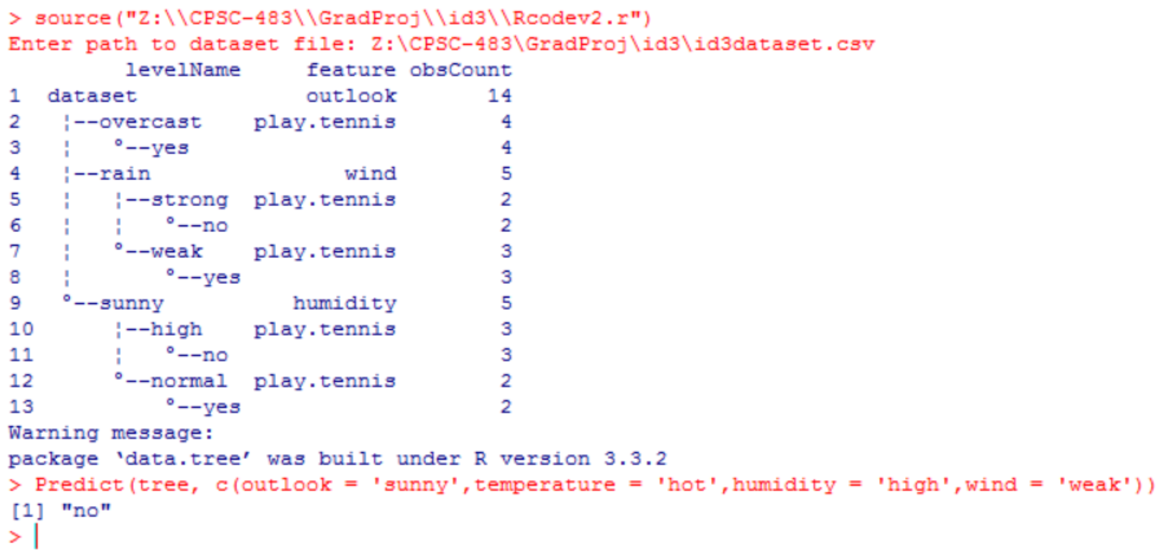


Figure 4: Output plot of generated tree

Since *Outlook* had highest information gain, thus it forms the root node. The subsequent nodes are generated depending on information gain in split data. The play.tennis the class attribute forming the leaf node. The *Temperature* attribute does not appear in the tree as its information gain is zero indicating that it does not contribute any information in predicting whether tennis will be played or not. The attributes are not displayed in the tree.

To predict the test data, call the predict function. Below is the example of test data prediction.



```
> source("Z:\\CPSC-483\\GradProj\\id3\\Rcodev2.r")
Enter path to dataset file: Z:\CPSC-483\GradProj\id3\id3dataset.csv
         levelName      feature obsCount
1  dataset              outlook       14
2   ¦--overcast     play.tennis        4
3   ¦   °--yes                         4
4   ¦--rain                wind        5
5   ¦   ¦--strong   play.tennis        2
6   ¦   ¦   °--no                      2
7   ¦   °--weak     play.tennis        3
8   ¦       °--yes                     3
9   °--sunny           humidity        5
10      ¦--high     play.tennis        3
11      ¦   °--no                      3
12      °--normal   play.tennis        2
13          °--yes                     2
Warning message:
package ‘data.tree’ was built under R version 3.3.2
> Predict(tree, c(outlook = 'sunny',temperature = 'hot',humidity = 'high',wind = 'weak'))
[1] "no"
> |
```

Figure 5: Predicting the test observation

Thus the ID3 is the basic decision tree used for predicting and classifying data. It is visually easy to interpret. Next is the Principal Component analysis in which we reduce the dimension of the input feature vector.

Principal Component Analysis

About the Algorithm

Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables. The desired goal is to reduce the dimensions of a d-dimensional dataset by projecting it onto a k-dimensional subspace (where k<d) in order to increase the computational efficiency while retaining most of the information. This transformation is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components. The resulting vectors are an uncorrelated orthogonal basis set. PCA is sensitive to the relative scaling of the original variables. PCA is mostly used as a tool in exploratory data analysis and for making predictive models. PCA can be done by eigenvalue decomposition of a data covariance (or correlation) matrix or singular value decomposition of a data matrix, usually after mean centering (and normalizing or using Z-scores) the data matrix for each attribute. (PCA algorithm, n.d.)

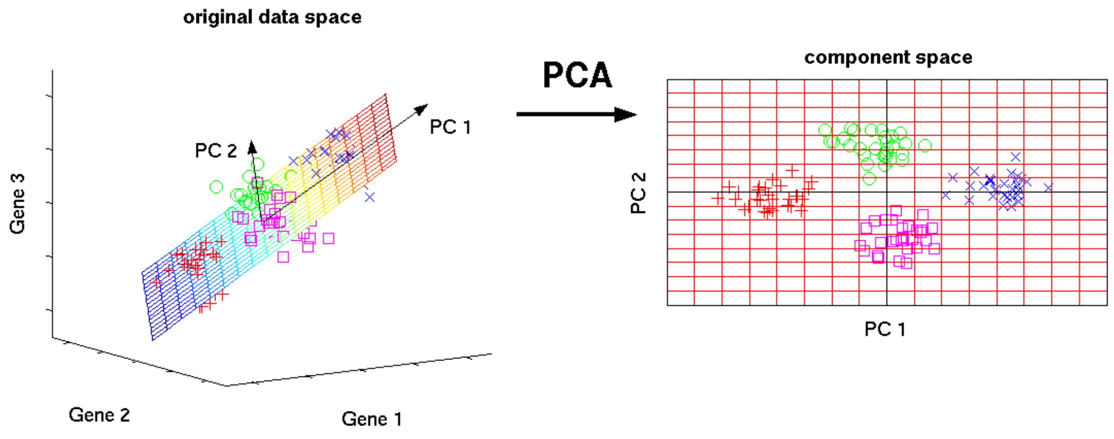


Figure 6: PCA concept

The summary of steps of the algorithm is as follows:

1. Standardize the data.
2. Obtain the Eigenvectors and Eigenvalues from the covariance matrix or correlation matrix, or perform Singular Vector Decomposition.
3. Sort eigenvalues in descending order and choose the k eigenvectors that correspond to the k largest eigenvalues where k is the number of dimensions of the new feature subspace (k≤d).
4. Construct the projection matrix W from the selected k eigenvectors.
5. Transform the original dataset X via W to obtain a k-dimensional feature subspace Y.

Terminologies used – Entropy and Information Gain

Eigen Vector and Eigen Values

In linear algebra, an eigenvector or characteristic vector of a linear transformation is a non-zero vector that does not change its direction when that linear transformation is applied to it. More formally, if T is a linear transformation from a vector space V over a field F into itself and v is a vector in V that is not the zero vector, then v is an eigenvector of T if T(v) is a scalar multiple of v. This condition can be written as the equation

$$T(v) = \lambda v$$

where λ is a scalar in the field F, known as the eigenvalue, characteristic value, or characteristic root associated with the eigenvector v. (PCA algorithm, n.d.)

Covariance Matrix

Covariance matrix Σ is a d×d matrix where each element represents the covariance between two features. The covariance between two features is calculated as follows:

$$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^{N} \left(x_{ij} - \bar{x}_j\right)\left(x_{ik} - \bar{x}_k\right).$$

We can summarize the calculation of the covariance matrix via the following matrix equation:

$$\Sigma = \frac{1}{n-1}\left((\mathbf{X} - \bar{\mathbf{x}})^T \, (\mathbf{X} - \bar{\mathbf{x}})\right)$$

where $\bar{x}$ is the mean vector.
The mean vector is a d-dimensional vector where each value in this vector represents the sample mean of a feature column in the dataset. (PCA algorithm, n.d.)

Singular Value Decomposition

In linear algebra, the singular value decomposition (SVD) is a factorization of a real or complex matrix. It is the generalization of the eigendecomposition of a positive semidefinite normal matrix (for example, a symmetric matrix with positive eigenvalues) to any *m by n* matrix via an extension of polar decomposition. It has many useful applications in signal processing and statistics. While the eigendecomposition of the covariance or correlation matrix may be more intuitiuve, most PCA implementations perform a Singular Vector Decomposition (SVD) to improve the computational efficiency. (Raschka, 2015)

Explained Variance

The explained variance tells us how much information (variance) can be attributed to each of the principal components. (Raschka, 2015)

Programming Environment and Dataset

Programming Environment

The programing language used is Python. Python provides inbuilt package *sklearn.decomposition.PCA* For PCA calculation given as below:

class `sklearn.decomposition.`**`PCA`**(*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*)

The Linear dimensionality reduction is done using Singular Value Decomposition of the data to project it to a lower dimensional space. It uses the LAPACK implementation of the full SVD or a randomized truncated SVD, depending on the shape of the input data and the number of components to extract. Python also provides different packages like numpy, matplotlib, scipy etc to perform different calculations. It eases the matrix operation, have inbuilt function to calculate Eigen values and vectors etc. (sklearn.decomposition.PCA, n.d.)

About the dataset

The dataset used for this is Iris dataset available in UCI machine learning repository.
https://archive.ics.uci.edu/ml/datasets/Iris
Since the example given on sklearn site uses this dataset for PCA example, it is easy to compare and verify visual results. Thus this dataset is used.

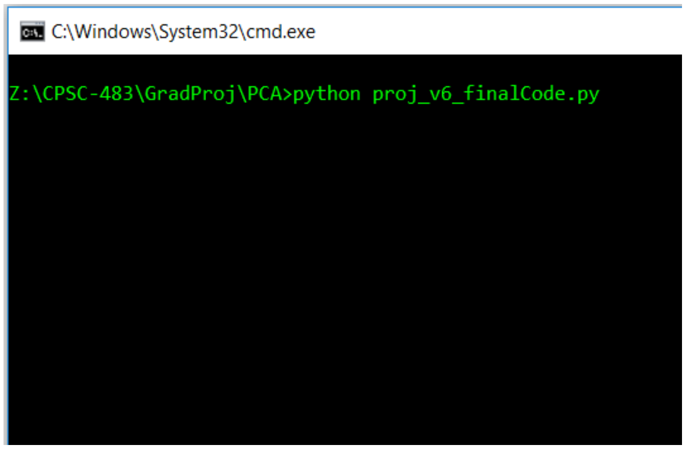
The dataset information is as follows:

1. Number of Instances: 150 (50 in each of three classes)
2. Number of Attributes: 4 numeric, predictive attributes and the class
3. Attribute Information:
    1. sepal length in cm
    2. sepal width in cm
    3. petal length in cm
    4. petal width in cm
    5. class:
        -- Iris Setosa
        -- Iris Versicolour
        -- Iris Virginica
4. Missing Attribute Values: None

Installation or Setup Steps

1. Download and install Python 3.5
2. Install the required libraries numpy, scipy, matplotlib, sklearn for Python 3.5. Instruction on http://scikit-learn.org/stable/install.html
3. Save the source code python file.
4. Open command window and navigate to path where the source code is saved.
5. Run the code using *python file_name* command.

Note: The Python version required is 3.5. If version is different and libraries installed are for different version, code will fail to execute. For correct installation please check online instructions.



```
C:\Windows\System32\cmd.exe

Z:\CPSC-483\GradProj\PCA>python proj_v6_finalCode.py
```

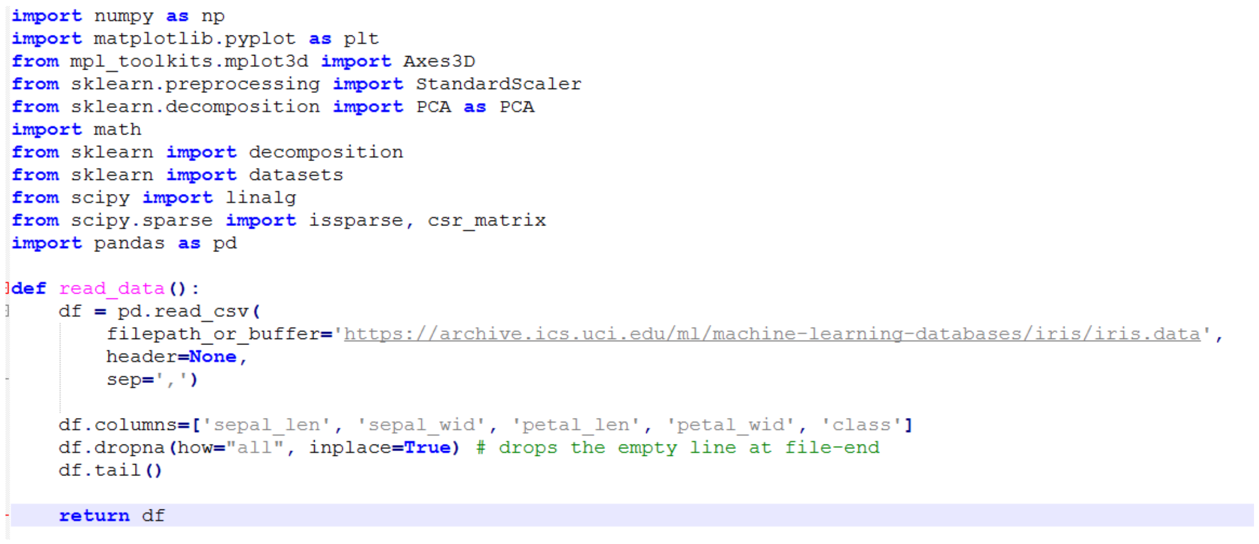
Figure 7: Execution of the code

Code in Action

Implementation

The implementation of the algorithm in python is as follows. (Raschka, 2015) (PCA example with Iris Data-set, n.d.)

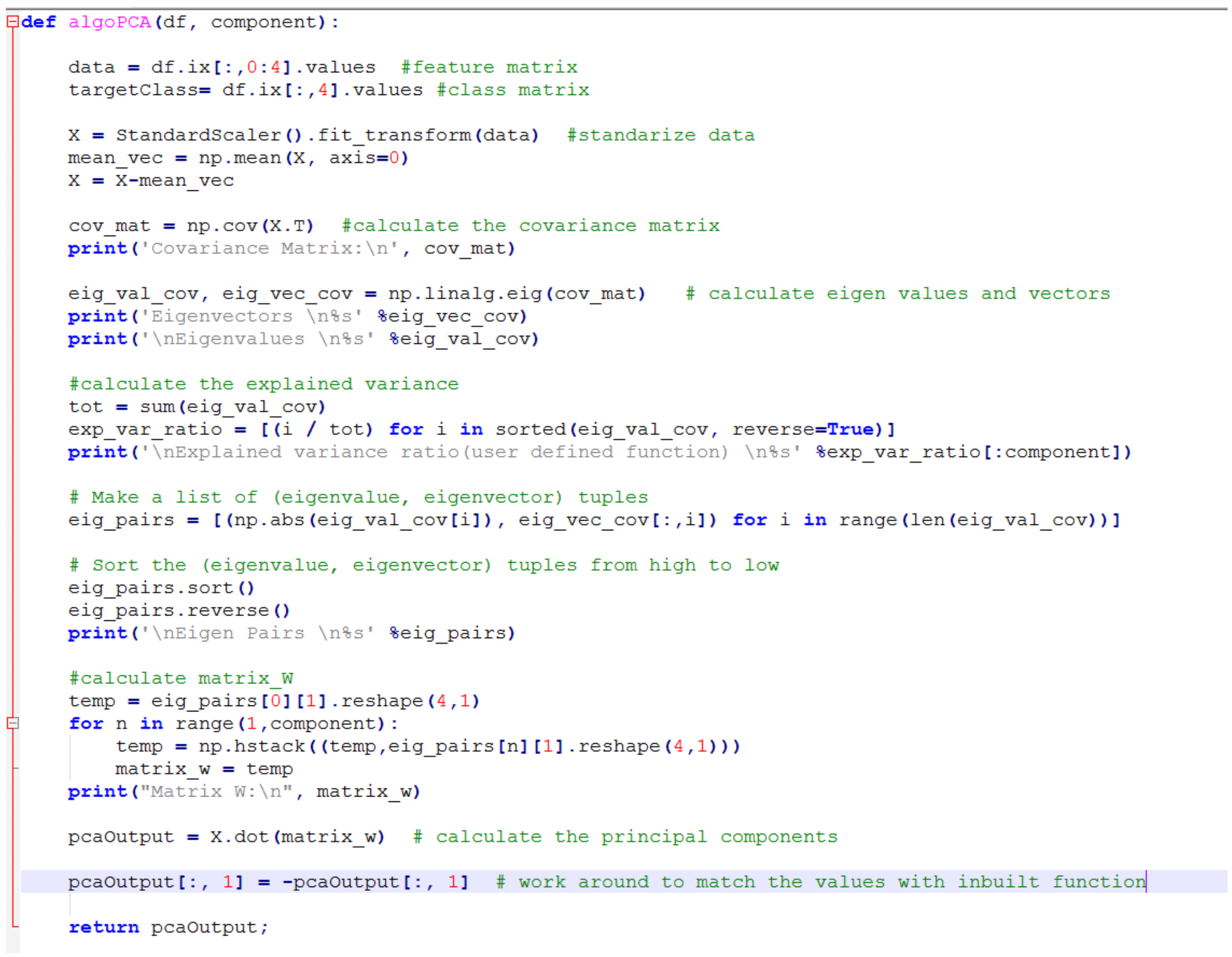
1. Load the library and data set.



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA as PCA
import math
from sklearn import decomposition
from sklearn import datasets
from scipy import linalg
from scipy.sparse import issparse, csr_matrix
import pandas as pd

def read_data():
    df = pd.read_csv(
        filepath_or_buffer='https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data',
        header=None,
        sep=',')

    df.columns=['sepal_len', 'sepal_wid', 'petal_len', 'petal_wid', 'class']
    df.dropna(how="all", inplace=True) # drops the empty line at file-end
    df.tail()

    return df
```

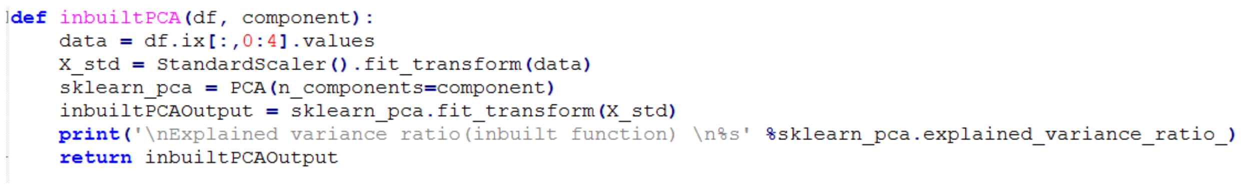
2. Define the functions to calculate PCA - user defined and inbuilt. The user defined function implements the steps explained above.

   - User defined function to calculate PCA



```
def algoPCA(df, component):

    data = df.ix[:,0:4].values  #feature matrix
    targetClass= df.ix[:,4].values #class matrix

    X = StandardScaler().fit_transform(data)  #standarize data
    mean_vec = np.mean(X, axis=0)
    X = X-mean_vec

    cov_mat = np.cov(X.T)  #calculate the covariance matrix
    print('Covariance Matrix:\n', cov_mat)

    eig_val_cov, eig_vec_cov = np.linalg.eig(cov_mat)   # calculate eigen values and vectors
    print('Eigenvectors \n%s' %eig_vec_cov)
    print('\nEigenvalues \n%s' %eig_val_cov)

    #calculate the explained variance
    tot = sum(eig_val_cov)
    exp_var_ratio = [(i / tot) for i in sorted(eig_val_cov, reverse=True)]
    print('\nExplained variance ratio(user defined function) \n%s' %exp_var_ratio[:component])

    # Make a list of (eigenvalue, eigenvector) tuples
    eig_pairs = [(np.abs(eig_val_cov[i]), eig_vec_cov[:,i]) for i in range(len(eig_val_cov))]

    # Sort the (eigenvalue, eigenvector) tuples from high to low
    eig_pairs.sort()
    eig_pairs.reverse()
    print('\nEigen Pairs \n%s' %eig_pairs)

    #calculate matrix_W
    temp = eig_pairs[0][1].reshape(4,1)
    for n in range(1,component):
        temp = np.hstack((temp,eig_pairs[n][1].reshape(4,1)))
        matrix_w = temp
    print("Matrix W:\n", matrix_w)

    pcaOutput = X.dot(matrix_w)  # calculate the principal components

    pcaOutput[:, 1] = -pcaOutput[:, 1]  # work around to match the values with inbuilt function

    return pcaOutput;
```

- Inbuilt function to calculate PCA



```
def inbuiltPCA(df, component):
    data = df.ix[:,0:4].values
    X_std = StandardScaler().fit_transform(data)
    sklearn_pca = PCA(n_components=component)
    inbuiltPCAOutput = sklearn_pca.fit_transform(X_std)
    print('\nExplained variance ratio(inbuilt function) \n%s' %sklearn_pca.explained_variance_ratio_)
    return inbuiltPCAOutput
```

3. Since we have only 4 features in data set, the code calculates principal component =2 and principal component =3. To display the results visually 2D and 3D plots are generated (specific to dataset) using following functions.

    - 2D plot function

```
def plot2PC(PCAOutput,y,msg,i):

    with plt.style.context('seaborn-whitegrid'):
        plt.figure(i,figsize=(6, 4))
        for lab, col in zip(('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'),
                            ('blue', 'red', 'green')):
            plt.scatter(PCAOutput[y==lab, 0],
                        PCAOutput[y==lab, 1],
                        label=lab,
                        c=col)
        plt.title(msg)
        plt.xlabel('Principal Component 1')
        plt.ylabel('Principal Component 2')
        plt.legend(loc='lower center')
        plt.tight_layout()
        plt.show()
```

- 3D plot function

```
def plot3PC(PCAOutput,y,msg,i):
    centers = [[1, 1], [-1, -1], [1, -1]]
    fig = plt.figure(i, figsize=(8, 8))
    plt.clf()
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
    plt.cla()

    for index, item in enumerate(y):
        if (item == 'Iris-setosa'):
            y[index] = 0
        if(item == 'Iris-versicolor'):
            y[index] = 1
        if(item == 'Iris-virginica'):
            y[index] = 2
    y=y.astype(np.int32)
    for name, label in [('setosa', 0), ('versicolor', 1), ('virginica', 2)]:
        ax.text3D(PCAOutput[y == label, 0].mean(),
                  PCAOutput[y == label, 1].mean() + 1.5,
                  PCAOutput[y == label, 2].mean(), name,
                  horizontalalignment='center',
                  bbox=dict(alpha=.5, edgecolor='w', facecolor='w'))


    # Reorder the labels to have colors matching the cluster results
    y = np.choose(y, [1, 2, 0]).astype(np.float)

    ax.scatter(PCAOutput[:, 0], PCAOutput[:, 1], PCAOutput[:, 2], c=y, cmap=plt.cm.spectral)
    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])
    ax.set_title(msg)
    ax.set_xlabel('Principal Component 1')
    ax.set_ylabel('Principal Component 2')
    ax.set_zlabel('Principal Component 3')
    plt.show()
```

4. Following code calls to the method to generate Principal Components and display the output.

```
# read data
df_implemented = read_data()
y= df_implemented.ix[:,4].values
# user defined PCA with PC=2
Y_userDef_PC2 = algoPCA(df_implemented,2)
#plot the 2D graph
plot2PC(Y_userDef_PC2,y,'Results of user defined function for PC=2',1)
# user defined PCA with PC=3
Y_userDef_PC3 = algoPCA(df_implemented,3)
#plot the 3D graph
plot3PC(Y_userDef_PC3,y,'Results of user defined function for PC=3',2)

# read data
df_inbuilt = read_data()
y=df_inbuilt.ix[:,4].values
# inbuilt PCA for PC=2
Y_sklearn_PC2=inbuiltPCA(df_inbuilt,2)
#plot the 2D graph
plot2PC(Y_sklearn_PC2,y,'Results of inbuilt function for PC=2',3)
# inbuilt PCA for PC=3
Y_sklearn_PC3=inbuiltPCA(df_inbuilt,3)
#plot the 3D graph
plot3PC(Y_sklearn_PC3,y,'Results of inbuilt function for PC=3',4)
```

Output

The output of the code is give in below figures.

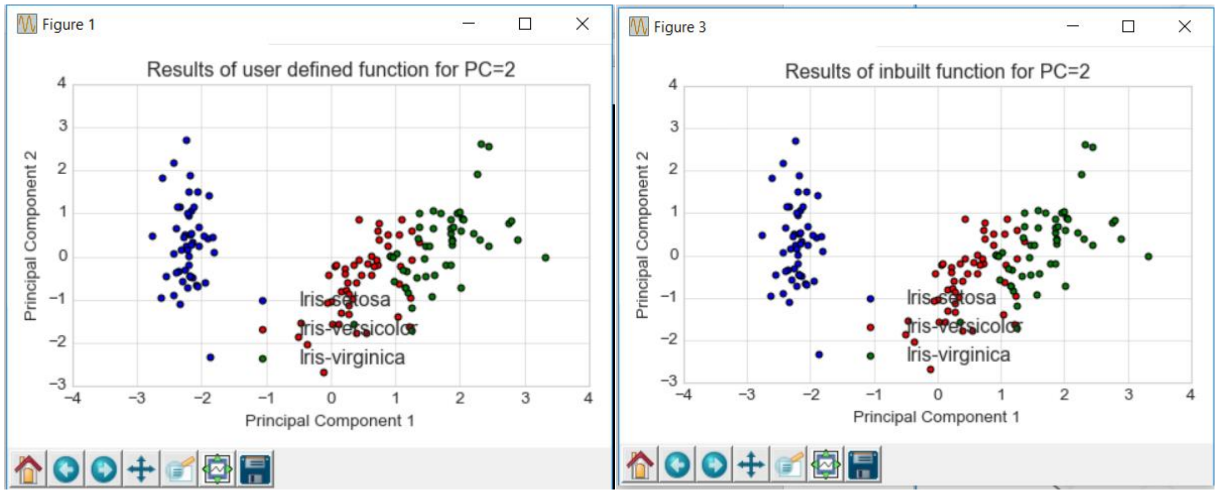


Figure 8: The plots for PC=2 using user defined and inbuilt functions

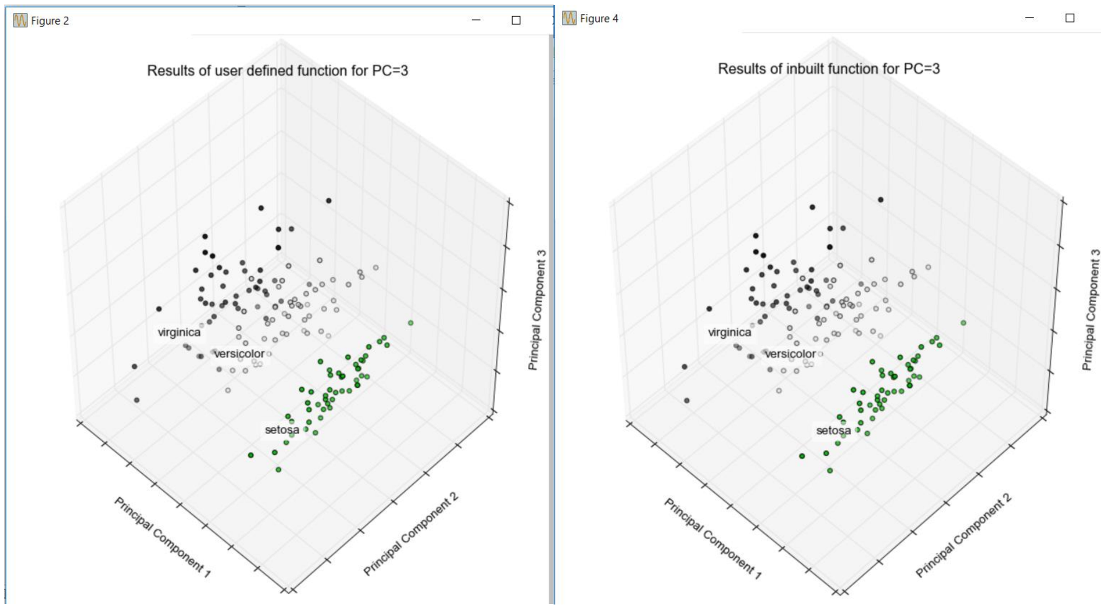


Figure 9: The plots for PC=3 using user defined and inbuilt functions

The console output is as given below:

```
Z:\CPSC-483\GradProj\PCA>python proj_v6_finalCode.py
Covariance Matrix:
 [[ 1.00671141 -0.11010327  0.87760486  0.82344326]
 [-0.11010327  1.00671141 -0.42333835 -0.358937  ]
 [ 0.87760486 -0.42333835  1.00671141  0.96921855]
 [ 0.82344326 -0.358937    0.96921855  1.00671141]]
Eigenvectors
[[ 0.52237162 -0.37231836 -0.72101681  0.26199559]
 [-0.26335492      -0.92555649           0.24203288      -0.12413481]
[      0.58125401      -0.02109478           0.14089226      -0.80115427]
[ 0.56561105 -0.06541577  0.6338014   0.52354627]]

Eigenvalues
[ 2.93035378  0.92740362  0.14834223  0.02074601]

Explained variance ratio(user defined function)
[0.72770452093801319, 0.23030523267680636]

Eigen Pairs
[(2.930353775589317, array([ 0.52237162, -0.26335492,  0.58125401,
0.56561105])), (0.92740362151734179, array([-0.37231836, -0.92555649,
-0.02109478,    -0.06541577])),    (0.14834222648163969,    array([-
0.72101681,      0.24203288,      0.14089226,      0.6338014   ])),
(0.020746013995596203, array([ 0.26199559, -0.12413481, -0.80115427,
0.52354627]))]                                            Matrix  W:
[[ 0.52237162 -0.37231836]
 [-0.26335492 -0.92555649]
 [ 0.58125401 -0.02109478]
 [ 0.56561105 -0.06541577]]
Covariance Matrix:
 [[ 1.00671141 -0.11010327  0.87760486  0.82344326]
 [-0.11010327  1.00671141 -0.42333835 -0.358937  ]
 [ 0.87760486 -0.42333835  1.00671141  0.96921855]
 [ 0.82344326 -0.358937    0.96921855  1.00671141]]
Eigenvectors
[[ 0.52237162 -0.37231836 -0.72101681  0.26199559]
 [-0.26335492 -0.92555649  0.24203288 -0.12413481]
 [ 0.58125401 -0.02109478  0.14089226 -0.80115427]
 [ 0.56561105 -0.06541577  0.6338014   0.52354627]]

Eigenvalues
[ 2.93035378  0.92740362  0.14834223  0.02074601]

Explained variance ratio(user defined function)
[0.7277045209380131 9, 0.23030523267680636, 0.036838319576273829]

Eigen Pairs
[(2.930353775589317, array([ 0.52237162, -0.26335492,  0.58125401,
0.56561105])), (0.9274036215173417 9, array([-0.37231836, -0.92555649,
-0.02109478,    -0.06541577])),    (0.14834222648163969,    array([-
0.72101681,      0.24203288,      0.14089226,      0.6338014   ])),
(0.020746013995596203, array([ 0.26199559, -0.12413481, -0.80115427,
0.52354627]))]
Matrix W:
 [[ 0.52237162 -0.37231836 -0.72101681]
 [-0.26335492 -0.92555649  0.24203288]
 [ 0.58125401 -0.02109478  0.14089226]
 [ 0.56561105 -0.06541577  0.6338014 ]]

Explained variance ratio(inbuilt function)
[ 0.72770452  0.23030523]

Explained variance ratio(inbuilt function)
[ 0.72770452  0.23030523  0.03683832]
```

As seen from the plots and explained variance ratio (highlighted in yellow) the values of user defined function and inbuilt function are same. The difference between the inbuilt and user defined function is that, inbuilt function implements SVD, which is computationally efficient, to calculate PCA while user defined implements Eigendecomposition of the covariance matrix method to calculate PCA.

Conclusion

The project implements the ID3 and PCA algorithm. Following changes were faced during the implementation:

1. Initially JAVA was proposed to implement ID3.However JAVA have many limitations for matrix calculation and is computationally extensive. Also it does not have any support for tree generation. Thus the language was switched to R which eased the matrix calculation and tree generation.
2. For PCA the values generated form inbuilt function and user defined function have sign inversion for some components. This is due to inbuilt function flips the *U* vector signs to enforce deterministic output. Following is the code snippet form inbuilt sklearn PCA implementation. (sklearn Principal Component Analysis, n.d.)

```
# flip eigenvectors' sign to enforce deterministic output
        U, V = svd_flip(U, V)
```

```
def svd_flip(u, v, u_based_decision=True):
    """Sign correction to ensure deterministic output from SVD.
    Adjusts the columns of u and the rows of v such that the loadings in the
    columns in u that are largest in absolute value are always positive.
```

The user defined code is not very generalized, and thus the output gives sign error for some columns. To fix this work around is implemented to match the values. This inadequacy can be improved by generalizing the code.

```
pcaOutput[:, 1] = -pcaOutput[:, 1]  # work around to match the values with inbuilt function
```

References

data mining. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Data_mining

Glur, C. (2015). *ID3 Classification using data.tree*. Retrieved from http://cran.bic.nus.edu.sg/web/packages/data.tree/vignettes/ID3.html

Glur, C. (2016). *Introduction to data.tree*. Retrieved from https://cran.r-project.org/web/packages/data.tree/vignettes/data.tree.html

ID3_algorithm. (n.d.). Retrieved from https://en.wikipedia.org/wiki/ID3_algorithm

PCA algorithm. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Principal_component_analysis

PCA example with Iris Data-set. (n.d.). Retrieved from http://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_iris.html

Raschka, S. (2015). *Principal Component Analysis in 3 Simple Steps*. Retrieved from http://sebastianraschka.com/Articles/2015_pca_in_3_steps.html

sklearn Principal Component Analysis. (n.d.). Retrieved from https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/decomposition/pca.py

sklearn.decomposition.PCA. (n.d.). Retrieved from http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html