DATA MINING

Dr. Jason Wang

Option: 1

SUPERVISED DATA MINING

[CLASSIFICATION]

Final Term Project

SUMMER 2018

By:

Himanshu Hunge | hh292 | hh292@njit.edu

LinkedIn:www.linkedin.com/in/himanshu-hunge-njit

PROJECT PROPOSAL

Name: Himanshu Hunge

UCID: hh292

Project option number: Option 1

Project option name: Supervised data mining - Classification

Algorithms to be used in the project:

- Category 3 – Decision Tree [Optimized version of CART or C 4.5]
- Category 1 – Support Vector Machine [LIBSVM radial basis function (RBF) kernel]

Tools: Jupyter Notebook 5.5.0

Package: Anaconda, Inc.

Programming languages to be used in the project: Python 3.6.5

Data Set: Absenteeism at work

Data Source Url: https://archive.ics.uci.edu/ml/datasets/Absenteeism+at+work#

Dataset Description: https://archive.ics.uci.edu/ml/machine-learning-databases/00445/

My Git hub Url:
https://github.com/hh292/Data_Mining/blob/master/Data_Mining_Absenteeism_at_work.ipynb

Table of Content

Introduction:

The Project involves detailed analysis of different classification algorithm on a dataset with numerous attributes and instances. The data mining algorithms which prevails us to predict the instances based on the model's accuracy. This project comprises model of two different data mining algorithms based on supervised classification.

Abstract:

There are numerous amount of reasons or causes that are produced all over the case when you don't make it for the work. We tend to give many reasons that are genuine and orderly faced. Notwithstanding, we never inquired about the genuine reasons and how often this scenario we faced while we cannot make it for the work. Identifying the most frequent issue that may become useful strategy and to understand the causes behind this is the main motive for this analysis. For example, the most likely cause of not going to work or being absent could be because of Health issue or the transportation on that given day. The other aspect of the cause could be on what frequent day the most employees are not making it to the work. The purpose of this paper is to make a prediction of the absenteeism in time and to predict what is the main cause of most frequent absenteeism.

Business Understanding:

The aim of the project is to analyze the reason for Absenteeism at work data set and create a predictive model based on supervised learning. The model is used to help the consultant to identify an employee will be absent of his/ her work. For instance, examining the effect of components like Social Drinker, Employees children, Work load at office, Transportation expenses can help the consultant to predict the reason of absenteeism.

Dataset Description:

Title: Absenteeism at work Data Set

- Updated on 2018-04-05 by Martiniano, A., Ferreira, R. P., Sassi, R. J., & Affonso, C. (2012)

Sources: Creators original owner and donors: Andrea Martiniano, Ricardo Pinto Ferreira, and Renato Jose Sassi.

Relevant Information:

The data set allows several new combinations of attributes and attribute exclusions, or the modification of the attribute type (categorical, integer, or real) depending on the purpose of the research. The data set (Absenteeism at work - Part I) was used in academic research at the Universidad Nove de Julho - Postgraduate Program in Informatics and Knowledge Management.

Number of Instances: 740

Attributes: There are total of 21 attributes as which are as follows:

1. Individual identification (ID)
2. Reason for absence (ICD).

   Absences attested by the International Code of Diseases (ICD) stratified into 21 categories (I to XXI) as per the diseases.

3. Month of absence
4. Day of the week - Monday (2), Tuesday (3), Wednesday (4), Thursday (5), Friday (6)
5. Seasons (summer (1), autumn (2), winter (3), spring (4))
6. Transportation expense
7. Distance from Residence to Work (kilometers)
8. Service time

9. Age
10. Work load Average/day
11. Hit target
12. Disciplinary failure (yes=1; no=0)
13. Education (high school, graduate, postgraduate, master and doctor)
14. Son (number of children)
15. Social drinker (yes=1; no=0)
16. Social smoker (yes=1; no=0)
17. Pet (number of pet)
18. Weight
19. Height
20. Body mass index
21. Absenteeism time in hours.

Target Attribute: Absenteeism time in hours

Missing Attribute Values: None

Data Type of Attributes: All are integer except one attribute [Work load Average/day].

About Tool

Jupyter Notebook:

The Jupyter Notebook is an interactive computing environment that enables users to author notebook documents that include: - Live code - Interactive widgets - Plots - Narrative text - Equations - Images - Video

The Jupyter Notebook combines three components:

- **The notebook web application**: An interactive web application for writing and running code interactively and authoring notebook documents.
- **Kernels:** Separate processes started by the notebook web application that runs users' code in a given language and returns output back to the notebook web application. The kernel also handles things like computations for interactive widgets, tab completion and introspection.
- **Notebook documents:** Self-contained documents that contain a representation of all content visible in the notebook web application, including inputs and outputs of the computations, narrative text, equations, images, and rich media representations of objects. Each notebook document has its own kernel.

Anaconda:

Anaconda is a free and open source distribution of the **Python** and **R** programming languages for data science and machine learning related applications (large-scale data processing, predictive analytics, scientific computing), that aims to simplify package management and deployment. Package versions are managed by the package management system conda. The Anaconda distribution is used by over 6 million users, and it includes more than 250 popular data science packages suitable for Windows, Linux, and MacOS.

Data Mining Process

Initialization of libraries:

Importing all basic files in Jupyter like pandas, sklearn, matlablib.

```
In [1]: #All the basic libraries- pandas, sklearn, matplotlib required for the analysis of the
        #dataset are loaded into the notebook.

        import pandas as pd       #Pandas software library for data manipulation and analysis
        import numpy as np        #numpy package for scientific computing

        #Using all basic libraries for mining , statistics and visulization.

        from sklearn.metrics import accuracy_score
        from sklearn.model_selection import train_test_split
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.feature_selection import RFE
        from sklearn.linear_model import LinearRegression
        from sklearn.metrics import confusion_matrix

        import matplotlib.pyplot as plt
        from sklearn import cross_validation
        from sklearn.metrics import roc_curve, auc
        from sklearn import tree
        from sklearn.metrics import f1_score

        /anaconda3/lib/python3.6/site-packages/sklearn/cross_validation.py:41: DeprecationWarning: This module was deprecated
        in version 0.18 in favor of the model_selection module into which all the refactored classes and functions are moved.
        Also note that the interface of the new CV iterators are different from that of this module. This module will be remo
        ved in 0.20.
          "This module will be removed in 0.20.", DeprecationWarning)
```

Data Cleansing:

Data Understanding and preparation: (Input [2] to Input [11])

Data Understanding and Data Preparation

In [2]:
```
#Reading the data from csv file "Asenteesim at work".

data = pd.read_csv("Absenteeism_at_work.csv")
```

In [3]:
```
data.head(5)
```

Out[3]:

	ID	Reason for absence	Month of absence	Day of the week	Seasons	Transportation expense	Distance from Residence to Work	Service time	Age	Work load Average/day	...	Disciplinary failure	Education
0	11	26	7	3	1	289	36	13	33	239,554	...	0	1
1	36	0	7	3	1	118	13	18	50	239,554	...	1	1
2	3	23	7	4	1	179	51	18	38	239,554	...	0	1
3	7	7	7	5	1	279	5	14	39	239,554	...	0	1
4	11	23	7	5	1	289	36	13	33	239,554	...	0	1

5 rows × 21 columns

In [4]:
```
data.columns
```

Out[4]:
```
Index(['ID', 'Reason for absence', 'Month of absence', 'Day of the week',
       'Seasons', 'Transportation expense', 'Distance from Residence to Work',
       'Service time', 'Age', 'Work load Average/day ', 'Hit target',
       'Disciplinary failure', 'Education', 'Son', 'Social drinker',
       'Social smoker', 'Pet', 'Weight', 'Height', 'Body mass index',
       'Absenteeism time in hours'],
      dtype='object')
```

In [5]:
```
#Removing special character of column name "Work load Average/day".

data = data.rename(columns = {'Work load Average/day ':'Work_Load_Avg_per_day'})
```

In [6]:
```
data.dtypes
```

Out[6]:
```
ID                                 int64
Reason for absence                 int64
Month of absence                   int64
Day of the week                    int64
Seasons                            int64
Transportation expense             int64
Distance from Residence to Work    int64
Service time                       int64
Age                                int64
Work_Load_Avg_per_day             object
Hit target                         int64
Disciplinary failure               int64
Education                          int64
Son                                int64
Social drinker                     int64
Social smoker                      int64
Pet                                int64
Weight                             int64
Height                             int64
Body mass index                    int64
Absenteeism time in hours          int64
dtype: object
```

In [7]:
```
#Removing special chacater "," (comma) in Work_Load_Avg_per_day column and
#converting its type to float.

data['Work_Load_Avg_per_day'] = data.Work_Load_Avg_per_day.str.replace(',', '').astype(float)
```

In [8]:
```
data.head(3)
```

Out[8]:

	ID	Reason for absence	Month of absence	Day of the week	Seasons	Transportation expense	Distance from Residence to Work	Service time	Age	Work_Load_Avg_per_day	...	Disciplinary failure
0	11	26	7	3	1	289	36	13	33	239554.0	...	0
1	36	0	7	3	1	118	13	18	50	239554.0	...	1
2	3	23	7	4	1	179	51	18	38	239554.0	...	0

3 rows × 21 columns

```
In [9]: data.describe()
```

Out[9]:

	ID	Reason for absence	Month of absence	Day of the week	Seasons	Transportation expense	Distance from Residence to Work	Service time	Age	Work_
count	740.000000	740.000000	740.000000	740.000000	740.000000	740.000000	740.000000	740.000000	740.000000	
mean	18.017568	19.216216	6.324324	3.914865	2.544595	221.329730	29.631081	12.554054	36.450000	
std	11.021247	8.433406	3.436287	1.421675	1.111831	66.952223	14.836788	4.384873	6.478772	
min	1.000000	0.000000	0.000000	2.000000	1.000000	118.000000	5.000000	1.000000	27.000000	
25%	9.000000	13.000000	3.000000	3.000000	2.000000	179.000000	16.000000	9.000000	31.000000	
50%	18.000000	23.000000	6.000000	4.000000	3.000000	225.000000	26.000000	13.000000	37.000000	
75%	28.000000	26.000000	9.000000	5.000000	4.000000	260.000000	50.000000	16.000000	40.000000	
max	36.000000	28.000000	12.000000	6.000000	4.000000	388.000000	52.000000	29.000000	58.000000	

8 rows × 21 columns

```
In [10]: data.shape
Out[10]: (740, 21)
```


```
In [10]: data.shape
Out[10]: (740, 21)
```


```
In [11]: data.isnull().any()
Out[11]: ID                                 False
         Reason for absence                 False
         Month of absence                   False
         Day of the week                    False
         Seasons                            False
         Transportation expense             False
         Distance from Residence to Work    False
         Service time                       False
         Age                                False
         Work_Load_Avg_per_day              False
         Hit target                         False
         Disciplinary failure               False
         Education                          False
         Son                                False
         Social drinker                     False
         Social smoker                      False
         Pet                                False
         Weight                             False
         Height                             False
         Body mass index                    False
         Absenteeism time in hours          False
         dtype: bool
```

Data Training:

1. Training the Data (Input [12] to Input [15]):

- Creating the X columns to train the data.
- From the available 21 attribute it has been observe that following attribute can be eliminated since they do not contribute towards Target attribute that is Absenteeism time in hours.
- ID -> Since every ID represent its unique feature of disease which I am not considering here.
- From following attribute, I am trying to understand the reason behind the absenteeism time hence I have included all possible columns for the analysis.

```
In [12]: x=data.drop(['Absenteeism time in hours','ID'], axis=1)

In [13]: data.columns

Out[13]: Index(['ID', 'Reason for absence', 'Month of absence', 'Day of the week',
                'Seasons', 'Transportation expense', 'Distance from Residence to Work',
                'Service time', 'Age', 'Work_Load_Avg_per_day', 'Hit target',
                'Disciplinary failure', 'Education', 'Son', 'Social drinker',
                'Social smoker', 'Pet', 'Weight', 'Height', 'Body mass index',
                'Absenteeism time in hours'],
               dtype='object')

In [14]: x.shape

Out[14]: (740, 19)

In [15]: x.head(3)

Out[15]:
```

	Reason for absence	Month of absence	Day of the week	Seasons	Transportation expense	Distance from Residence to Work	Service time	Age	Work_Load_Avg_per_day	Hit target	Disciplinary failure	Education	Son	Social drinker	Social smoker
0	26	7	3	1	289	36	13	33	239554.0	97	0	1	2	1	0
1	0	7	3	1	118	13	18	50	239554.0	97	1	1	1	1	0
2	23	7	4	1	179	51	18	38	239554.0	97	0	1	0	1	0

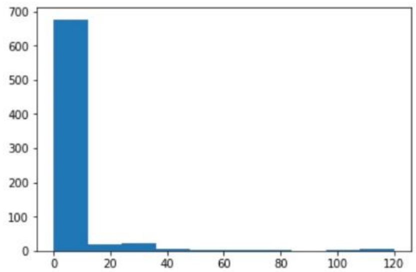
2. Declare the Y that is Target (Input [16] to Input [17])

Using attribute “Absenteeism time in hours” as the target I am Analyzing the “Absenteeism time in hours” attributes.

Declare the Y that is Target.

- Using attribute Absenteeism time in hours as the target. Analysing the Absenteeism time in hours attributes

In [16]:
```
plt.hist(data['Absenteeism time in hours'])
plt.show()
```



Understanding the Absenteeism data is to interpret into two sections

In [17]:
```
data['Absenteeism time in hours'].describe()
```

Out[17]:
```
count     740.000000
mean        6.924324
std        13.330998
min         0.000000
25%         2.000000
50%         3.000000
75%         8.000000
max       120.000000
Name: Absenteeism time in hours, dtype: float64
```

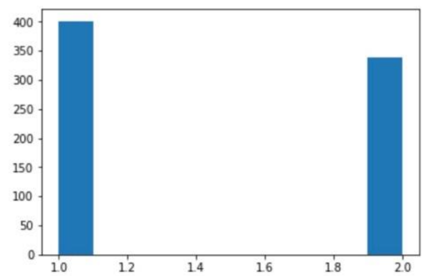
3. Creating a new attribute as Category to be used as Target attribute (Input [18] - Input [21])

- The Columns are identified as either 1 or 2 on following basis:
  - **Category 1** - The "absenteeism time in work" that are shared less than 50% that is less than value 3.00 -> The issue isn't that serious for an Employee.
  - **Category 2** - The "absenteeism time in work" that are shared more than 50% that is more than value 3.00 -> The Employee is suffering with some issue.

```
In [18]: data_class = data['Absenteeism time in hours']
         data['category'] = (data_class<=3)*1 | (data_class>3)*2
```

Ploting the category data

```
In [19]: plt.hist(data['category'])
         plt.show()
```




```
In [20]: data.head(3)
```

Out[20]:

	ID	Reason for absence	Month of absence	Day of the week	Seasons	Transportation expense	Distance from Residence to Work	Service time	Age	Work_Load_Avg_per_day	...	Education	Son	Social drinker	Social smoker	Pet	Weigl
0	11	26	7	3	1	289	36	13	33	239554.0	...	1	2	1	0	1	9
1	36	0	7	3	1	118	13	18	50	239554.0	...	1	1	1	0	0	9
2	3	23	7	4	1	179	51	18	38	239554.0	...	1	0	1	0	0	8

3 rows × 22 columns

```
In [21]: data.shape
Out[21]: (740, 22)
```

4. Category is added to the Data as 22 attribute and will be used as target (Input [22] – Input [24])

Category is added to the Data as 22 attribute and will be used as target

```
In [22]: y = data['category']

In [23]: # Checking the X and Y
         x.shape
Out[23]: (740, 19)

In [24]: y.shape
Out[24]: (740,)
```

Splitting Data:

1. Splitting the data into training and test data (Input [25] – Input [29])

Splitting the data into training and test data taking 20% from the overall data as test data and 80% as training data

```
In [25]: X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.20, random_state=324)
```

Verifying the Data

```
In [26]: X_train.shape
Out[26]: (592, 19)

In [27]: y_train.shape
Out[27]: (592,)

In [28]: X_test.shape
Out[28]: (148, 19)

In [29]: y_test.shape
Out[29]: (148,)
```

Training Data Analysis:

1. Training Data Analysis and Identifying the most relevant attributes (Input [30] – Input [31])

2. Using Decision Tree Classifier for finding accuracy of training data.

Training Data Analysis and Identifying the most relevant attributes

Using Decision Tree Classifier for finding accuracy of training data.

```
In [30]: rank_classifier = DecisionTreeClassifier(max_leaf_nodes=10, random_state=0)
         rank_classifier.fit(X_train,y_train)
         print(type(rank_classifier))
         Predictions_train = rank_classifier.predict(X_train)

         <class 'sklearn.tree.tree.DecisionTreeClassifier'>

In [31]: # Verifying the accuracy

         Training_Accuracy = accuracy_score(y_true = y_train, y_pred = Predictions_train)

         print("Training Data Acuracy-")
         print(Training_Accuracy)

         Training Data Acuracy-
         0.7956081081081081
```

3. Using Recursive feature elimination for identifying the most relevant attributes

Recursive Feature Elimination or RFE uses a model (e.g. linear Regression or SVM) to select either the best or worst-performing feature, and then excludes that feature. The whole process is then iterated until all features in the dataset are used up (or up to a user-defined limit). Sklearn conveniently possesses a RFE function via the sklearn.feature_selection call and I use this along with a simple linear regression model and logistic regression model for ranking the features and to decide on the attribute for the model building.

```
In [32]: from sklearn.feature_selection import RFE
         from sklearn.linear_model import LinearRegression
         #use linear regression as the model
         lr = LinearRegression()
         rfe = RFE(lr, n_features_to_select=1)
         rfe.fit(x,y)
         print(rfe.support_)
         print(rfe.ranking_)

         [False False False False False False False False False False  True False
          False False False False False False False]
         [11 14  6 10 15 17 13 16 19 18  1  5  3  2  4 12  8  9  7]

In [33]: from sklearn.feature_selection import RFE
         from sklearn.linear_model import LogisticRegression
         #use logistic regression as the model
         lr = LogisticRegression()
         rfe = RFE(lr, n_features_to_select=1)
         rfe.fit(x,y)
         print(rfe.support_)
         print(rfe.ranking_)

         [False False False False False False False False False False  True False
          False False False False False False False]
         [ 7  8  4  6 17 18 11 14 19 13  1  5  2  3  9 10 15 16 12]
```

```
In [34]: list(x)

Out[34]: ['Reason for absence',
          'Month of absence',
          'Day of the week',
          'Seasons',
          'Transportation expense',
          'Distance from Residence to Work',
          'Service time',
          'Age',
          'Work_Load_Avg_per_day',
          'Hit target',
          'Disciplinary failure',
          'Education',
          'Son',
          'Social drinker',
          'Social smoker',
          'Pet',
          'Weight',
          'Height',
          'Body mass index']

In [35]: x_New = data[['Disciplinary failure','Social drinker','Son','Social smoker','Education','Day of the week','Height','Wei

In [36]: x_New.shape

Out[36]: (740, 19)
```

Implementing Data Mining Algorithms

Data Mining Algorithms

- **Classification Analysis:**

Classification is a data mining technique that assigns categories to a collection of data in order to aid in more accurate predictions and analysis. This method is utilized to recover critical and important information about a data set. It is utilized to organize information in various classes. Very large databases are becoming the norm in today's world of "big data." The primary challenge of big data is how to make sense of it. And sheer volume is not the only problem: big data also tends to be diverse, unstructured and fast-changing. Consider audio and video data, social media posts, 3D data or geospatial data. This kind of data is not easily categorized or organized. To meet this challenge, a range of automatic methods for extracting useful information has been developed, among them classification.

- **Applications:**

Classification, and other data mining techniques, is behind much of our day-to-day experience as consumers. Weather predictions might make use of classification to report whether the day will be rainy, sunny or cloudy. The medical profession might analyze health conditions to predict medical outcomes. From fraud detection to product offers, classification is behind the scenes every day analyzing data and producing predictions.

1. Category 3: Decision tree Implementation

1. Decision Tree Classifier - CART / C4.5 (Classification and Regression Trees)(Input [37] – Input [49])

A decision tree is a guide of the conceivable results of a progression of related choices. Decision tree is one of the most used techniques in data mining because of its simplicity to explain the results. Besides, there are decision tree algorithms that work with parallel and incremental techniques, which help to process large databases for classifying new objects faster than traditional algorithms. A decision tree ordinarily begins with a single node, which branches into conceivable results.

- ## Applications:

Decision trees are very “user-friendly” because they are easy to understand by practically everyone and provide reasonably accurate results. They provide an easy to analyze breakdown of the data and can be used practically in every business area that requires decision-making, including, but not limited to marketing, pharmacology, financial analysis, manufacturing, production, etc.

```
In [37]: #splitting the data
         X_train, X_test, y_train, y_test = train_test_split(x_New, y, test_size=0.20, random_state=250)

In [38]: rank_classifier = DecisionTreeClassifier(max_leaf_nodes=10, random_state=0)
         rank_classifier.fit(X_train,y_train)
         print(type(rank_classifier))
         Predictions = rank_classifier.predict(X_test)

         <class 'sklearn.tree.tree.DecisionTreeClassifier'>

In [39]: rank_classifier.fit(X_train,y_train)

Out[39]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                     max_features=None, max_leaf_nodes=10,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, presort=False, random_state=0,
                     splitter='best')

In [40]: #calculating the accuracy score
         Accuracy_DT = accuracy_score(y_true = y_test, y_pred = Predictions)
         Accuracy_DT

Out[40]: 0.8243243243243243

In [41]: from sklearn import metrics
         confusion=metrics.confusion_matrix(y_test, Predictions)
         print(confusion)
         TP_DT = confusion[1, 1]
         TN_DT = confusion[0, 0]
         FP_DT = confusion[0, 1]
         FN_DT = confusion[1, 0]

         [[65 16]
          [10 57]]
```

Calculating Misclassification Rate, Precision, Recall, F_1 -Score, Sensitivity, Specificity for Decision Tree:

```
In [42]: # Calculating "Misclassification Rate"which represent how often is the classifier incorrect?
         Mis_Rate_DT = (1 - metrics.accuracy_score(y_test, Predictions))
         print("Misclassification Rate: ",Mis_Rate_DT)

         #Precision: When a positive value is predicted, how often is the prediction correct?
         precision_DT = metrics.precision_score(y_test, Predictions)
         print("Precision: ",precision_DT)

         #Recall:Recall is the fraction of the relevant results that are successfully retrieved.
         recall_DT = metrics.recall_score(y_test, Predictions)
         print("recall: ",recall_DT)

         #F1 Score: balanced F-score or F-measure which measure's test accuracy
         f1_DT=f1_score(y_test,Predictions)
         print("F1_Score:", f1_DT)


         #Sensitivity: When the actual value is positive, how often is the prediction correct?
         #Also known as "True Positive Rate" or "Recall"
         sensitivity_DT = TP_DT / float(FN_DT + TP_DT)
         print("Sensitivity: ",sensitivity_DT)

         #Specificity: When the actual value is negative.
         specificity_DT = TN_DT / (TN_DT + FP_DT)
         print("Specificity: ",specificity_DT)

         Misclassification Rate:  0.17567567567567566
         Precision:  0.8666666666666667
         recall:  0.8024691358024691
         F1_Score: 0.8333333333333334
         Sensitivity:  0.8507462686567164
         Specificity:  0.8024691358024691

In [43]: #Cross Validation scoreDEPLOYMENT
         scores_DT = cross_validation.cross_val_score(rank_classifier, x_New, y, cv=10).mean()
         print (scores_DT)

         0.7579901271134147
```

Receiver Operating Characteristic (ROC) for Decision Tree:

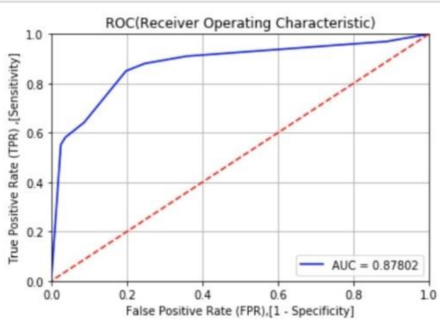
```
In [44]: #from sklearn.preprocessing import label_binarize.Calculating fpr,tpr for ROC.

         y_pred_prob = rank_classifier.predict_proba(X_test)[:, 1]

         fpr, tpr, thresholds = metrics.roc_curve(y_test,y_pred_prob,pos_label=2)

In [45]: roc_auc = auc(fpr, tpr)

In [46]: plt.title('ROC(Receiver Operating Characteristic)')
         plt.plot(fpr, tpr, 'b', label='AUC = %0.5f'% roc_auc)
         plt.legend(loc='lower right')
         plt.rcParams['font.size'] = 10
         plt.plot([1,0],[1,0],'r--')
         plt.xlim([0.0,1.0])
         plt.ylim([0.0,1.0])
         plt.ylabel('True Positive Rate (TPR) ,[Sensitivity]')
         plt.xlabel('False Positive Rate (FPR),[1 - Specificity]')
         plt.grid(True)
         plt.show()
```



```
In [47]: def evaluate_threshold(threshold):
             print('Sensitivity:', tpr[thresholds > threshold][-1])
             print('Specificity:', 1 - fpr[thresholds > threshold][-1])

In [48]: evaluate_threshold(0.5)

         Sensitivity: 0.8507462686567164
         Specificity: 0.8024691358024691

In [49]: #loss function
         from sklearn.metrics import log_loss
         Loss_DT = log_loss(y_test,y_pred_prob)
         Loss_DT

Out[49]: 0.8462093123888312
```

2. Category 1: Support Vector Machine (SVM)

1. LIBSVM radial basis function (RBF) kernel (Input [50] – Input [55])

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labelled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In two-dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.

Here I am using **RBF kernel** (radial basis function) for SVM algorithm.

2. Applications

As we have seen, SVMs depends on supervised learning algorithms. The aim of using SVM is to correctly classify unseen data. SVMs have several applications in several fields. Some common applications of SVM are- Face detection, Text and hypertext categorization, Classification of images, Bioinformatics, Protein fold and remote homology detection, Handwriting recognition, Generalized predictive control(GPC)

```
In [50]: from sklearn import svm

In [51]: X_train, X_test, y_train, y_test = train_test_split(x_New, y, test_size=0.20, random_state=250)

In [52]: svm_classifier = svm.SVC(probability=True)

In [53]: svm_classifier.fit(X_train,y_train)

Out[53]: SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
           max_iter=-1, probability=True, random_state=None, shrinking=True,
           tol=0.001, verbose=False)

In [54]: Predictions_svm = svm_classifier.predict(X_test)
         Accuracy_SVM = accuracy_score(y_true = y_test, y_pred = Predictions_svm)
         Accuracy_SVM

Out[54]: 0.7094594594594594

In [55]: confusion_matrix(y_test,Predictions_svm)
         from sklearn import metrics
         confusion=metrics.confusion_matrix(y_test, Predictions_svm)
         print(confusion)
         TP_SVM = confusion[1, 1]
         TN_SVM = confusion[0, 0]
         FP_SVM = confusion[0, 1]
         FN_SVM = confusion[1, 0]

         [[43 38]
          [ 5 62]]
```

Calculating Misclassification Rate, Precision, Recall, F1 -Score, Sensitivity, Specificity for SVM:

```
In [56]: # Calculating "Misclassification Rate"how often is the classifier incorrect?
         Mis_Rate_SVM = (1 - metrics.accuracy_score(y_test, Predictions_svm))
         print("Misclassification Rate: ",Mis_Rate_SVM)

         #Precision: When a positive value is predicted, how often is the prediction correct?
         precision_SVM = metrics.precision_score(y_test, Predictions_svm)
         print("Precision: ",precision_SVM)

         #Recall:Recall is the fraction of the relevant results that are successfully retrieved.
         recall_SVM = metrics.recall_score(y_test, Predictions_svm)
         print("Recall: ",recall_SVM)

         #F1 Score: balanced F-score or F-measure which measure's test accuracy.
         f1_SVM=f1_score(y_test,Predictions_svm)
         print("F1_Score:", f1_SVM)

         #Sensitivity: When the actual value is positive, how often is the prediction correct?
         #Also known as "True Positive Rate" or "Recall"
         sensitivity_SVM = TP_SVM / float(FN_SVM + TP_SVM)
         print("Sensitivity: ",sensitivity_SVM)

         #Specificity: When the actual value is negative, how often is the prediction correct?
         #This specifies how "specific" (or "selective") is the classifier in predicting positive instances?
         specificity_SVM = TN_SVM / (TN_SVM + FP_SVM)
         print("Specificity: ",specificity_SVM)

         Misclassification Rate:  0.29054054054054057
         Precision:  0.8958333333333334
         Recall:  0.5308641975308642
         F1_Score: 0.6666666666666666
         Sensitivity:  0.9253731343283582
         Specificity:  0.5308641975308642
```

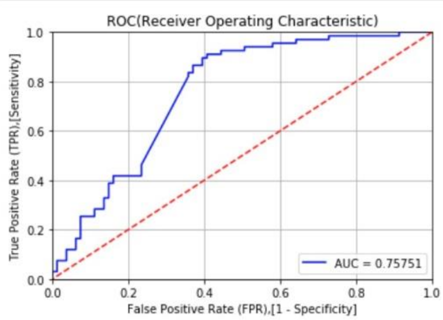
Receiver Operating Characteristic (ROC) for SVM:

```
In [57]: y_pred_prob = svm_classifier.predict_proba(X_test)[:, 1]

         fpr, tpr, thresholds = metrics.roc_curve(y_test,y_pred_prob,pos_label=2)

In [58]: roc_auc = auc(fpr, tpr)

In [59]: plt.title('ROC(Receiver Operating Characteristic)')
         plt.plot(fpr, tpr, 'b', label='AUC = %0.5f'% roc_auc)
         plt.legend(loc='lower right')
         plt.rcParams['font.size'] = 10
         plt.plot([1,0],[1,0],'r--')
         plt.xlim([0,1.0])
         plt.ylim([0,1.0])
         plt.ylabel('True Positive Rate (TPR),[Sensitivity]')
         plt.xlabel('False Positive Rate (FPR),[1 - Specificity]')
         plt.grid(True)
         plt.show()
```



```
In [60]: def evaluate_threshold(threshold):
             print('Sensitivity:', tpr[thresholds > threshold][-1])
             print('Specificity:', 1 - fpr[thresholds > threshold][-1])
```

```
In [61]: evaluate_threshold(0.5)

         Sensitivity: 0.9253731343283582
         Specificity: 0.5555555555555556
```

```
In [62]: #loss function
         from sklearn.metrics import log_loss
         Loss_SVM = log_loss(y_test,y_pred_prob)
         Loss_SVM

Out[62]: 0.5899355772607078
```

RESULTS

Comparing the models based on following factors:

Accuracy, Loss, True Positive, False Positive, True Negative, False Negative, Sensitivity, Precision, Recall and F_1 Score:

In [63]:
```
resultdf = pd.DataFrame(columns=["Model","Accuracy","Loss","TP","FP","TN","FN","Sensitivity","Specificity","Precision",
resultdf.loc[0] = ["Decision Tree", Accuracy_DT,Loss_DT,TP_DT,FP_DT,TN_DT,FN_DT,sensitivity_DT, specificity_DT,precisio
resultdf.loc[1] = ["SVM", Accuracy_SVM, Loss_SVM, TP_SVM,FP_SVM,TN_SVM,FN_SVM,sensitivity_SVM, specificity_SVM,precisio

resultdf
```

Out[63]:

	Model	Accuracy	Loss	TP	FP	TN	FN	Sensitivity	Specificity	Precision	Recall	F1 score
0	Decision Tree	0.824324	0.846209	57	16	65	10	0.850746	0.802469	0.866667	0.802469	0.833333
1	SVM	0.709459	0.589936	62	38	43	5	0.925373	0.530864	0.895833	0.530864	0.666667

In [64]:
```
arraycuracy =[]
arraycuracy.append(Accuracy_DT)
arraycuracy.append(Accuracy_SVM)
```

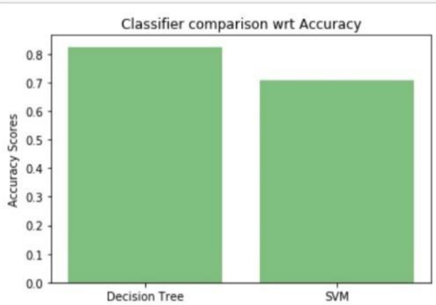
In [65]:
```
precisionray =[]
precisionray.append(precision_DT)
precisionray.append(precision_SVM)
```

In [66]:
```
recallray =[]
recallray.append(recall_DT)
recallray.append(recall_SVM)
```

In [67]:
```
f1ray=[]
f1ray.append(f1_DT)
f1ray.append(f1_SVM)
```

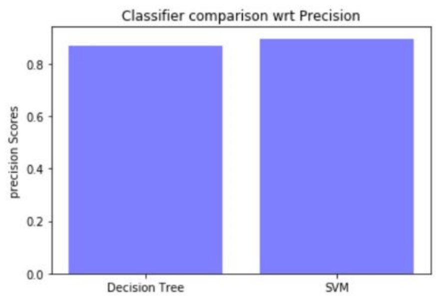
Comparison with respect to Accuracy:

In [68]:
```
import  matplotlib.pyplot  as plt
classifier_names=('Decision Tree','SVM')
y_axis=np.arange(len(classifier_names))
plt.bar(y_axis, arraycuracy, alpha=0.5,color='Green')
plt.xticks(y_axis, classifier_names)
plt.ylabel('Accuracy Scores')
plt.title('Classifier comparison wrt Accuracy')
plt.show()
```



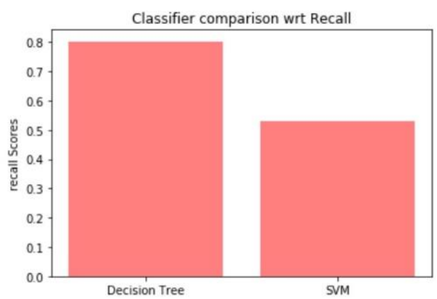
Comparison with respect to Precision:

```
In [69]: y_axis=np.arange(len(classifier_names))
         plt.bar(y_axis,precisionray, alpha=0.5,color='Blue',ecolor='black')
         plt.xticks(y_axis, classifier_names)
         plt.ylabel('precision Scores')
         plt.title('Classifier comparison wrt Precision')
         plt.show()
```



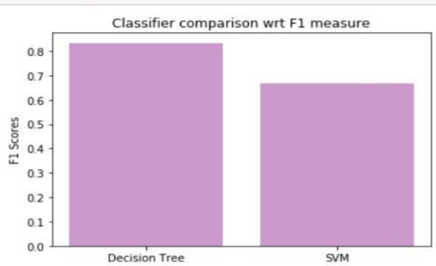
Comparison with respect to Recall:

```
In [70]: y_axis=np.arange(len(classifier_names))
         plt.bar(y_axis,recallray, alpha=0.5,color='Red',ecolor='black')
         plt.xticks(y_axis, classifier_names)
         plt.ylabel('recall Scores')
         plt.title('Classifier comparison wrt Recall')
         plt.show()
```



Comparison with respect to F1 Score:

```
In [71]: y_axis=np.arange(len(classifier_names))
         plt.bar(y_axis,f1ray, alpha=0.4,color='Purple',ecolor='black')
         plt.xticks(y_axis, classifier_names)
         plt.ylabel('F1 Scores')
         plt.title('Classifier comparison wrt F1 measure')
         plt.show()
```



Conclusion

With the aim to predict the absenteeism in work on the various factors we build models based on two different supervised classification techniques

- Decision Tree
- Support Vector Machine

As we can observe through the results, The Decision tree has performed better as compared to SVM in terms of accuracy, recall and F_1 Score.

From the available 22 attributes, I identified 19 based on domain knowledge and the recursive feature elimination (RFE) model.

Some of the key factors that contribute towards Absenteeism in work are

- 'Disciplinary failure' is the biggest issue of absenteeism.
- How many children does an employee has?
- Does drinking alcohol is an issue of absenteeism - 'Social Drinker'?

The Decision Tree model that I have built gives **82 %** accuracy on the testing data. Consequently, I have built a model that can predict whether an Employee will be absent on for the work given that the features provided for his/her absence.

Source Code

Code for Decision Tree [Optimized Version of CART or C4.5]:

This module gathers decision tree method both Single and multi-output problems are both handled.

```
# Authors: Gilles Louppe <g.louppe@gmail.com>

#        Peter Prettenhofer <peter.prettenhofer@gmail.com>

#        Brian Holt <bdholt1@gmail.com>

#        Noel Dawe <noel@dawe.me>

#        Satrajit Gosh <satrajit.ghosh@gmail.com>
```

```
#       Joly Arnaud <arnaud.v.joly@gmail.com>
#       Fares Hedayati <fares.hedayati@gmail.com>
#       Nelson Liu <nelson@nelsonliu.me>
#
# License: BSD 3 clause
from __future__ import division
import numbers
import warnings
from abc import ABCMeta
from abc import abstractmethod
from math import ceil
import numpy as np
from scipy.sparse import issparse
from ..base import BaseEstimator
from ..base import ClassifierMixin
from ..base import RegressorMixin
from ..base import is_classifier
from ..externals import six
from ..utils import check_array
from ..utils import check_random_state
from ..utils import compute_sample_weight
from ..utils.multiclass import check_classification_targets
from ..utils.validation import check_is_fitted
from ._criterion import Criterion
from ._splitter import Splitter
from ._tree import DepthFirstTreeBuilder
```

```
from ._tree import BestFirstTreeBuilder
from ._tree import Tree
from . import _tree, _splitter, _criterion



__all__ = ["DecisionTreeClassifier",
        "DecisionTreeRegressor",
        "ExtraTreeClassifier",
        "ExtraTreeRegressor"]

# =============================================================================
# Types and constants
# =============================================================================

DTYPE = _tree.DTYPE
DOUBLE = _tree.DOUBLE
CRITERIA_CLF = {"gini": _criterion.Gini, "entropy": _criterion.Entropy}
CRITERIA_REG = {"mse": _criterion.MSE, "friedman_mse": _criterion.FriedmanMSE,
          "mae": _criterion.MAE}
DENSE_SPLITTERS = {"best": _splitter.BestSplitter,
            "random": _splitter.RandomSplitter}



SPARSE_SPLITTERS = {"best": _splitter.BestSparseSplitter,
            "random": _splitter.RandomSparseSplitter}

# =============================================================================
```

```
# Base decision tree
# =============================================================================


class BaseDecisionTree(six.with_metaclass(ABCMeta, BaseEstimator)):
    """Base class for decision trees.

    Warning: This class should not be used directly.
    Use derived classes instead.
    """

    @abstractmethod
    def __init__(self,
                 criterion,
                 splitter,
                 max_depth,
                 min_samples_split,
                 min_samples_leaf,
                 min_weight_fraction_leaf,
                 max_features,
                 max_leaf_nodes,
                 random_state,
                 min_impurity_decrease,
                 min_impurity_split,
                 class_weight=None,
                 presort=False):
        self.criterion = criterion
        self.splitter = splitter
```

```
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.min_weight_fraction_leaf = min_weight_fraction_leaf
        self.max_features = max_features
        self.random_state = random_state
        self.max_leaf_nodes = max_leaf_nodes
        self.min_impurity_decrease = min_impurity_decrease
        self.min_impurity_split = min_impurity_split
        self.class_weight = class_weight
        self.presort = presort



    def fit(self, X, y, sample_weight=None, check_input=True,
            X_idx_sorted=None):



        random_state = check_random_state(self.random_state)
        if check_input:
            X = check_array(X, dtype=DTYPE, accept_sparse="csc")
            y = check_array(y, ensure_2d=False, dtype=None)
            if issparse(X):
                X.sort_indices()



                if X.indices.dtype != np.intc or X.indptr.dtype != np.intc:
                    raise ValueError("No support for np.int64 index based "
```

```
                      "sparse matrices")




        # Determine output settings
        n_samples, self.n_features_ = X.shape
        is_classification = is_classifier(self)
        y = np.atleast_1d(y)
        expanded_class_weight = None
        if y.ndim == 1:
            # reshape is necessary to preserve the data contiguity against vs
            # [:, np.newaxis] that does not.
            y = np.reshape(y, (-1, 1))
        self.n_outputs_ = y.shape[1]
        if is_classification:
            check_classification_targets(y)
            y = np.copy(y)
            self.classes_ = []
            self.n_classes_ = []
            if self.class_weight is not None:
                y_original = np.copy(y)
            y_encoded = np.zeros(y.shape, dtype=np.int)
            for k in range(self.n_outputs_):
                classes_k, y_encoded[:, k] = np.unique(y[:, k],
                                                       return_inverse=True)
                self.classes_.append(classes_k)
                self.n_classes_.append(classes_k.shape[0])
```

```
    y = y_encoded



    if self.class_weight is not None:
        expanded_class_weight = compute_sample_weight(
            self.class_weight, y_original)
else:
    self.classes_ = [None] * self.n_outputs_
    self.n_classes_ = [1] * self.n_outputs_
self.n_classes_ = np.array(self.n_classes_, dtype=np.intp)
if getattr(y, "dtype", None) != DOUBLE or not y.flags.contiguous:
    y = np.ascontiguousarray(y, dtype=DOUBLE)
# Check parameters
max_depth = ((2 ** 31) - 1 if self.max_depth is None
             else self.max_depth)
max_leaf_nodes = (-1 if self.max_leaf_nodes is None
                  else self.max_leaf_nodes)
if isinstance(self.min_samples_leaf, (numbers.Integral, np.integer)):
    if not 1 <= self.min_samples_leaf:
        raise ValueError("min_samples_leaf must be at least 1 "
                         "or in (0, 0.5], got %s"
                         % self.min_samples_leaf)
    min_samples_leaf = self.min_samples_leaf
else:  # float
    if not 0. < self.min_samples_leaf <= 0.5:
        raise ValueError("min_samples_leaf must be at least 1 "
                         "or in (0, 0.5], got %s"
```

```
                    % self.min_samples_leaf)
        min_samples_leaf = int(ceil(self.min_samples_leaf * n_samples))
    if isinstance(self.min_samples_split, (numbers.Integral, np.integer)):
        if not 2 <= self.min_samples_split:
            raise ValueError("min_samples_split must be an integer "
                             "greater than 1 or a float in (0.0, 1.0]; "
                             "got the integer %s"
                             % self.min_samples_split)
        min_samples_split = self.min_samples_split
    else:  # float
        if not 0. < self.min_samples_split <= 1.:
            raise ValueError("min_samples_split must be an integer "
                             "greater than 1 or a float in (0.0, 1.0]; "
                             "got the float %s"
                             % self.min_samples_split)
        min_samples_split = int(ceil(self.min_samples_split * n_samples))
        min_samples_split = max(2, min_samples_split)
    min_samples_split = max(min_samples_split, 2 * min_samples_leaf)

    if isinstance(self.max_features, six.string_types):
        if self.max_features == "auto":
            if is_classification:
                max_features = max(1, int(np.sqrt(self.n_features_)))
            else:
                max_features = self.n_features_
        elif self.max_features == "sqrt":
            max_features = max(1, int(np.sqrt(self.n_features_)))
```

```
    elif self.max_features == "log2":
        max_features = max(1, int(np.log2(self.n_features_)))
    else:
        raise ValueError(
            'Invalid value for max_features. Allowed string '
            'values are "auto", "sqrt" or "log2".')
elif self.max_features is None:
    max_features = self.n_features_
elif isinstance(self.max_features, (numbers.Integral, np.integer)):
    max_features = self.max_features
else:  # float
    if self.max_features > 0.0:
        max_features = max(1,
                           int(self.max_features * self.n_features_))
    else:
        max_features = 0

self.max_features_ = max_features



if len(y) != n_samples:
    raise ValueError("Number of labels=%d does not match "
                     "number of samples=%d" % (len(y), n_samples))
if not 0 <= self.min_weight_fraction_leaf <= 0.5:
    raise ValueError("min_weight_fraction_leaf must in [0, 0.5]")
if max_depth <= 0:
    raise ValueError("max_depth must be greater than zero. ")
```

```
if not (0 < max_features <= self.n_features_):
    raise ValueError("max_features must be in (0, n_features]")
if not isinstance(max_leaf_nodes, (numbers.Integral, np.integer)):
    raise ValueError("max_leaf_nodes must be integral number but was "
                     "%r" % max_leaf_nodes)
if -1 < max_leaf_nodes < 2:
    raise ValueError(("max_leaf_nodes {0} must be either None "
                      "or larger than 1").format(max_leaf_nodes))
if sample_weight is not None:
    if (getattr(sample_weight, "dtype", None) != DOUBLE or
            not sample_weight.flags.contiguous):
        sample_weight = np.ascontiguousarray(
            sample_weight, dtype=DOUBLE)
    if len(sample_weight.shape) > 1:
        raise ValueError("Sample weights array has more "
                         "than one dimension: %d" %
                         len(sample_weight.shape))
    if len(sample_weight) != n_samples:
        raise ValueError("Number of weights=%d does not match "
                         "number of samples=%d" %
                         (len(sample_weight), n_samples))



if expanded_class_weight is not None:
    if sample_weight is not None:
        sample_weight = sample_weight * expanded_class_weight
    else:
```

```
        sample_weight = expanded_class_weight

    # Set min_weight_leaf from min_weight_fraction_leaf
    if sample_weight is None:
        min_weight_leaf = (self.min_weight_fraction_leaf *
                           n_samples)
    else:
        min_weight_leaf = (self.min_weight_fraction_leaf *
                           np.sum(sample_weight))



    if self.min_impurity_split is not None:
        warnings.warn("The min_impurity_split parameter is deprecated and"
                      " will be removed in version 0.21. "
                      "Use the min_impurity_decrease parameter instead.",
                      DeprecationWarning)
        min_impurity_split = self.min_impurity_split
    else:
        min_impurity_split = 1e-7
    if min_impurity_split < 0.:
        raise ValueError("min_impurity_split must be greater than "
                         "or equal to 0")

    if self.min_impurity_decrease < 0.:
        raise ValueError("min_impurity_decrease must be greater than "
                         "or equal to 0")
    allowed_presort = ('auto', True, False)
```

```
    if self.presort not in allowed_presort:
        raise ValueError("'presort' should be in {}. Got {!r} instead."
                         .format(allowed_presort, self.presort))
    if self.presort is True and issparse(X):
        raise ValueError("Presorting is not supported for sparse "
                         "matrices.")



    presort = self.presort
    # Allow presort to be 'auto', which means True if the dataset is dense,
    # otherwise it will be False.
    if self.presort == 'auto':
        presort = not issparse(X)



    # If multiple trees are built on the same dataset, we only want to
    # presort once. Splitters now can accept presorted indices if desired,
    # but do not handle any presorting themselves. Ensemble algorithms
    # which desire presorting must do presorting themselves and pass that
    # matrix into each tree.
    if X_idx_sorted is None and presort:
        X_idx_sorted = np.asfortranarray(np.argsort(X, axis=0),
                                         dtype=np.int32)
if presort and X_idx_sorted.shape != X.shape:
        raise ValueError("The shape of X (X.shape = {}) doesn't match "
                         "the shape of X_idx_sorted (X_idx_sorted"
                         ".shape = {})".format(X.shape,
```

```
                              X_idx_sorted.shape))

# Build tree
criterion = self.criterion
if not isinstance(criterion, Criterion):
    if is_classification:
        criterion = CRITERIA_CLF[self.criterion](self.n_outputs_,
                                                 self.n_classes_)
    else:
        criterion = CRITERIA_REG[self.criterion](self.n_outputs_,
                                                 n_samples)


SPLITTERS = SPARSE_SPLITTERS if issparse(X) else DENSE_SPLITTERS

splitter = self.splitter
if not isinstance(self.splitter, Splitter):
    splitter = SPLITTERS[self.splitter](criterion,
                                        self.max_features_,
                                        min_samples_leaf,
                                        min_weight_leaf,
                                        random_state,
                                        self.presort)

self.tree_ = Tree(self.n_features_, self.n_classes_, self.n_outputs_)

# Use BestFirst if max_leaf_nodes given; use DepthFirst otherwise
```

```
        if max_leaf_nodes < 0:
            builder = DepthFirstTreeBuilder(splitter, min_samples_split,
                                            min_samples_leaf,
                                            min_weight_leaf,
                                            max_depth,
                                            self.min_impurity_decrease,
                                            min_impurity_split)
        else:
            builder = BestFirstTreeBuilder(splitter, min_samples_split,
                                           min_samples_leaf,
                                           min_weight_leaf,
                                           max_depth,
                                           max_leaf_nodes,
                                           self.min_impurity_decrease,
                                           min_impurity_split)

        builder.build(self.tree_, X, y, sample_weight, X_idx_sorted)

        if self.n_outputs_ == 1:
            self.n_classes_ = self.n_classes_[0]
            self.classes_ = self.classes_[0]

        return self

    def _validate_X_predict(self, X, check_input):
        """Validate X whenever one tries to predict, apply, predict_proba"""
        if check_input:
```

```
            X = check_array(X, dtype=DTYPE, accept_sparse="csr")
            if issparse(X) and (X.indices.dtype != np.intc or
                                X.indptr.dtype != np.intc):
                raise ValueError("No support for np.int64 index based "
                                 "sparse matrices")

        n_features = X.shape[1]
        if self.n_features_ != n_features:
            raise ValueError("Number of features of the model must "
                             "match the input. Model n_features is %s and "
                             "input n_features is %s "
                             % (self.n_features_, n_features))



        return X
    def predict(self, X, check_input=True):
        check_is_fitted(self, 'tree_')
        X = self._validate_X_predict(X, check_input)
        proba = self.tree_.predict(X)
        n_samples = X.shape[0]
        # Classification
        if is_classifier(self):
            if self.n_outputs_ == 1:
                return self.classes_.take(np.argmax(proba, axis=1), axis=0)
            else:
                predictions = np.zeros((n_samples, self.n_outputs_))
```

```
            for k in range(self.n_outputs_):
                predictions[:, k] = self.classes_[k].take(
                    np.argmax(proba[:, k], axis=1),
                    axis=0)
            return predictions
    # Regression
    else:
        if self.n_outputs_ == 1:
            return proba[:, 0]



        else:
            return proba[:, :, 0]



def apply(self, X, check_input=True):
    """
    Returns the index of the leaf that each sample is predicted as.

    .. versionadded:: 0.17

    Parameters
    ----------
    X : array_like or sparse matrix, shape = [n_samples, n_features]
        The input samples. Internally, it will be converted to
        ``dtype=np.float32`` and if a sparse matrix is provided
```

```
        to a sparse ``csr_matrix``.

    check_input : boolean, (default=True)
        Allow to bypass several input checking.
        Don't use this parameter unless you know what you do.

    Returns
    -------
    X_leaves : array_like, shape = [n_samples,]
        For each datapoint x in X, return the index of the leaf x
        ends up in. Leaves are numbered within
        ``[0; self.tree_.node_count)``, possibly with gaps in the
        numbering.
    """
    check_is_fitted(self, 'tree_')
    X = self._validate_X_predict(X, check_input)
    return self.tree_.apply(X)

def decision_path(self, X, check_input=True):
    """Return the decision path in the tree

    Parameters
    ----------
    X : array_like or sparse matrix, shape = [n_samples, n_features]
        The input samples. Internally, it will be converted to
        ``dtype=np.float32`` and if a sparse matrix is provided
        to a sparse ``csr_matrix``.
```

```
        check_input : boolean, (default=True)
            Allow to bypass several input checking.
            Don't use this parameter unless you know what you do.


        Returns
        -------
        indicator : sparse csr array, shape = [n_samples, n_nodes]
            Return a node indicator matrix where non zero elements
            indicates that the samples goes through the nodes.


        """
        X = self._validate_X_predict(X, check_input)
        return self.tree_.decision_path(X)
    @property
    def feature_importances_(self):
        """Return the feature importances.
        The importance of a feature is computed as the (normalized) total
        reduction of the criterion brought by that feature.
        It is also known as the Gini importance.
        Returns
        -------
        feature_importances_ : array, shape = [n_features]
        """
        check_is_fitted(self, 'tree_')
        return self.tree_.compute_feature_importances()
```

For SVM – [libsvm RBF]:

```
"""
Binding for libsvm_skl
----------------------
These are the bindings for libsvm_skl, which is a fork of libsvm[1]
that adds to libsvm some capabilities, like index of support vectors
and efficient representation of dense matrices.
These are low-level routines, but can be used for flexibility or
performance reasons. See sklearn.svm for a higher-level API.
Low-level memory management is done in libsvm_helper.c. If we happen
to run out of memory a MemoryError will be raised. In practice this is
not very helpful since hight changes are malloc fails inside svm.cpp,
where no sort of memory checks are done.
[1] http://www.csie.ntu.edu.tw/~cjlin/libsvm/
Notes
-----
Maybe we could speed it a bit further by decorating functions with
@cython.boundscheck(False), but probably it is not worth since all
work is done in lisvm_helper.c
Also, the signature mode='c' is somewhat superficial, since we already
check that arrays are C-contiguous in svm.py
Authors
-------
2010: Fabian Pedregosa <fabian.pedregosa@inria.fr>
    Gael Varoquaux <gael.varoquaux@normalesup.org>
"""
```

```
import warnings

import  numpy as np

cimport numpy as np

cimport libsvm

from libc.stdlib cimport free

cdef extern from *:

    ctypedef struct svm_parameter:

        pass

np.import_array()

######################################################################################

# Internal variables

LIBSVM_KERNEL_TYPES = ['linear', 'poly', 'rbf', 'sigmoid', 'precomputed']

######################################################################################

# Wrapper functions

def fit(

    np.ndarray[np.float64_t, ndim=2, mode='c'] X,

    np.ndarray[np.float64_t, ndim=1, mode='c'] Y,

    int svm_type=0, kernel='rbf', int degree=3,

    double gamma=0.1, double coef0=0., double tol=1e-3,

    double C=1., double nu=0.5, double epsilon=0.1,

    np.ndarray[np.float64_t, ndim=1, mode='c']

        class_weight=np.empty(0),

    np.ndarray[np.float64_t, ndim=1, mode='c']

        sample_weight=np.empty(0),

    int shrinking=1, int probability=0,

    double cache_size=100.,

    int max_iter=-1,
```

```
int random_seed=0):
cdef svm_parameter param
cdef svm_problem problem
cdef svm_model *model
cdef const char *error_msg
cdef np.npy_intp SV_len
cdef np.npy_intp nr
if len(sample_weight) == 0:
   sample_weight = np.ones(X.shape[0], dtype=np.float64)
else:
   assert sample_weight.shape[0] == X.shape[0], \
         "sample_weight and X have incompatible shapes: " + \
         "sample_weight has %s samples while X has %s" % \
         (sample_weight.shape[0], X.shape[0])
kernel_index = LIBSVM_KERNEL_TYPES.index(kernel)
set_problem(
   &problem, X.data, Y.data, sample_weight.data, X.shape, kernel_index)
if problem.x == NULL:
   raise MemoryError("Seems we've run out of memory")
cdef np.ndarray[np.int32_t, ndim=1, mode='c'] \
   class_weight_label = np.arange(class_weight.shape[0], dtype=np.int32)
set_parameter(
   &param, svm_type, kernel_index, degree, gamma, coef0, nu, cache_size,
   C, tol, epsilon, shrinking, probability, <int> class_weight.shape[0],
   class_weight_label.data, class_weight.data, max_iter, random_seed)
error_msg = svm_check_parameter(&problem, &param)
if error_msg:
```

```
    # for SVR: epsilon is called p in libsvm
    error_repl = error_msg.decode('utf-8').replace("p < 0", "epsilon < 0")
    raise ValueError(error_repl)
# this does the real work
cdef int fit_status = 0
with nogil:
    model = svm_train(&problem, &param, &fit_status)

# from here until the end, we just copy the data returned by
# svm_train
SV_len  = get_l(model)
n_class = get_nr(model)

cdef np.ndarray[np.float64_t, ndim=2, mode='c'] sv_coef
sv_coef = np.empty((n_class-1, SV_len), dtype=np.float64)
copy_sv_coef (sv_coef.data, model)

# the intercept is just model.rho but with sign changed
cdef np.ndarray[np.float64_t, ndim=1, mode='c'] intercept
intercept = np.empty(int((n_class*(n_class-1))/2), dtype=np.float64)
copy_intercept (intercept.data, model, intercept.shape)

cdef np.ndarray[np.int32_t, ndim=1, mode='c'] support
support = np.empty (SV_len, dtype=np.int32)
copy_support (support.data, model)

# copy model.SV
```

```
cdef np.ndarray[np.float64_t, ndim=2, mode='c'] support_vectors
if kernel_index == 4:
    # precomputed kernel
    support_vectors = np.empty((0, 0), dtype=np.float64)
else:
    support_vectors = np.empty((SV_len, X.shape[1]), dtype=np.float64)
    copy_SV(support_vectors.data, model, support_vectors.shape)

# TODO: do only in classification
cdef np.ndarray[np.int32_t, ndim=1, mode='c'] n_class_SV
n_class_SV = np.empty(n_class, dtype=np.int32)
copy_nSV(n_class_SV.data, model)

cdef np.ndarray[np.float64_t, ndim=1, mode='c'] probA
cdef np.ndarray[np.float64_t, ndim=1, mode='c'] probB
if probability != 0:
    if svm_type < 2: # SVC and NuSVC
        probA = np.empty(int(n_class*(n_class-1)/2), dtype=np.float64)
        probB = np.empty(int(n_class*(n_class-1)/2), dtype=np.float64)
        copy_probB(probB.data, model, probB.shape)
    else:
        probA = np.empty(1, dtype=np.float64)
        probB = np.empty(0, dtype=np.float64)
    copy_probA(probA.data, model, probA.shape)
else:
    probA = np.empty(0, dtype=np.float64)
    probB = np.empty(0, dtype=np.float64)
```

```
    svm_free_and_destroy_model(&model)
    free(problem.x)

    return (support, support_vectors, n_class_SV, sv_coef, intercept,
        probA, probB, fit_status)
cdef void set_predict_params(
    svm_parameter *param, int svm_type, kernel, int degree, double gamma,
    double coef0, double cache_size, int probability, int nr_weight,
    char *weight_label, char *weight) except *:
    """Fill param with prediction time-only parameters."""
    # training-time only parameters
    cdef double C = .0
    cdef double epsilon = .1
    cdef int max_iter = 0
    cdef double nu = .5
    cdef int shrinking = 0
    cdef double tol = .1
    cdef int random_seed = -1

    kernel_index = LIBSVM_KERNEL_TYPES.index(kernel)

    set_parameter(param, svm_type, kernel_index, degree, gamma, coef0, nu,
                  cache_size, C, tol, epsilon, shrinking, probability,
                  nr_weight, weight_label, weight, max_iter, random_seed)
def predict(np.ndarray[np.float64_t, ndim=2, mode='c'] X,
        np.ndarray[np.int32_t, ndim=1, mode='c'] support,
```

```
        np.ndarray[np.float64_t, ndim=2, mode='c'] SV,
        np.ndarray[np.int32_t, ndim=1, mode='c'] nSV,
        np.ndarray[np.float64_t, ndim=2, mode='c'] sv_coef,
        np.ndarray[np.float64_t, ndim=1, mode='c'] intercept,
        np.ndarray[np.float64_t, ndim=1, mode='c'] probA=np.empty(0),
        np.ndarray[np.float64_t, ndim=1, mode='c'] probB=np.empty(0),
        int svm_type=0, kernel='rbf', int degree=3,
        double gamma=0.1, double coef0=0.,
        np.ndarray[np.float64_t, ndim=1, mode='c']
          class_weight=np.empty(0),
        np.ndarray[np.float64_t, ndim=1, mode='c']
          sample_weight=np.empty(0),
        double cache_size=100.):

    cdef np.ndarray[np.float64_t, ndim=1, mode='c'] dec_values
    cdef svm_parameter param
    cdef svm_model *model
    cdef int rv

    cdef np.ndarray[np.int32_t, ndim=1, mode='c'] \
        class_weight_label = np.arange(class_weight.shape[0], dtype=np.int32)

    set_predict_params(&param, svm_type, kernel, degree, gamma, coef0,
                       cache_size, 0, <int>class_weight.shape[0],
                       class_weight_label.data, class_weight.data)
    model = set_model(&param, <int> nSV.shape[0], SV.data, SV.shape,
                      support.data, support.shape, sv_coef.strides,
```

```
                sv_coef.data, intercept.data, nSV.data, probA.data, probB.data)


    #TODO: use check_model
    try:
        dec_values = np.empty(X.shape[0])
        with nogil:
            rv = copy_predict(X.data, model, X.shape, dec_values.data)
        if rv < 0:
            raise MemoryError("We've run out of memory")
    finally:
        free_model(model)


    return dec_values
def predict_proba(
    np.ndarray[np.float64_t, ndim=2, mode='c'] X,
    np.ndarray[np.int32_t, ndim=1, mode='c'] support,
    np.ndarray[np.float64_t, ndim=2, mode='c'] SV,
    np.ndarray[np.int32_t, ndim=1, mode='c'] nSV,
    np.ndarray[np.float64_t, ndim=2, mode='c'] sv_coef,
    np.ndarray[np.float64_t, ndim=1, mode='c'] intercept,
    np.ndarray[np.float64_t, ndim=1, mode='c'] probA=np.empty(0),
    np.ndarray[np.float64_t, ndim=1, mode='c'] probB=np.empty(0),
    int svm_type=0, kernel='rbf', int degree=3,
    double gamma=0.1, double coef0=0.,
    np.ndarray[np.float64_t, ndim=1, mode='c']
        class_weight=np.empty(0),
    np.ndarray[np.float64_t, ndim=1, mode='c']
```

```
    sample_weight=np.empty(0),
double cache_size=100.):

cdef np.ndarray[np.float64_t, ndim=2, mode='c'] dec_values
cdef svm_parameter param
cdef svm_model *model
cdef np.ndarray[np.int32_t, ndim=1, mode='c'] \
    class_weight_label = np.arange(class_weight.shape[0], dtype=np.int32)
cdef int rv

set_predict_params(&param, svm_type, kernel, degree, gamma, coef0,
                   cache_size, 1, <int>class_weight.shape[0],
                   class_weight_label.data, class_weight.data)
model = set_model(&param, <int> nSV.shape[0], SV.data, SV.shape,
                  support.data, support.shape, sv_coef.strides,
                  sv_coef.data, intercept.data, nSV.data,
                  probA.data, probB.data)

cdef np.npy_intp n_class = get_nr(model)
try:
    dec_values = np.empty((X.shape[0], n_class), dtype=np.float64)
    with nogil:
        rv = copy_predict_proba(X.data, model, X.shape, dec_values.data)
    if rv < 0:
        raise MemoryError("We've run out of memory")
finally:
    free_model(model)
```

```
    return dec_values
def decision_function(
    np.ndarray[np.float64_t, ndim=2, mode='c'] X,
    np.ndarray[np.int32_t, ndim=1, mode='c'] support,
    np.ndarray[np.float64_t, ndim=2, mode='c'] SV,
    np.ndarray[np.int32_t, ndim=1, mode='c'] nSV,
    np.ndarray[np.float64_t, ndim=2, mode='c'] sv_coef,
    np.ndarray[np.float64_t, ndim=1, mode='c'] intercept,
    np.ndarray[np.float64_t, ndim=1, mode='c'] probA=np.empty(0),
    np.ndarray[np.float64_t, ndim=1, mode='c'] probB=np.empty(0),
    int svm_type=0, kernel='rbf', int degree=3,
    double gamma=0.1, double coef0=0.,
    np.ndarray[np.float64_t, ndim=1, mode='c']
        class_weight=np.empty(0),
    np.ndarray[np.float64_t, ndim=1, mode='c']
         sample_weight=np.empty(0),
    double cache_size=100.):
    """
    Predict margin (libsvm name for this is predict_values)
    We have to reconstruct model and parameters to make sure we stay
    in sync with the python object.
    """
    cdef np.ndarray[np.float64_t, ndim=2, mode='c'] dec_values
    cdef svm_parameter param
    cdef svm_model *model
    cdef np.npy_intp n_class
```

```
cdef np.ndarray[np.int32_t, ndim=1, mode='c'] \
    class_weight_label = np.arange(class_weight.shape[0], dtype=np.int32)

cdef int rv

set_predict_params(&param, svm_type, kernel, degree, gamma, coef0,
                   cache_size, 0, <int>class_weight.shape[0],
                   class_weight_label.data, class_weight.data)
model = set_model(&param, <int> nSV.shape[0], SV.data, SV.shape,
                  support.data, support.shape, sv_coef.strides,
                  sv_coef.data, intercept.data, nSV.data,
                  probA.data, probB.data)

if svm_type > 1:
    n_class = 1
else:
    n_class = get_nr(model)
    n_class = n_class * (n_class - 1) / 2

try:
    dec_values = np.empty((X.shape[0], n_class), dtype=np.float64)
    with nogil:
        rv = copy_predict_values(X.data, model, X.shape, dec_values.data, n_class)
    if rv < 0:
        raise MemoryError("We've run out of memory")
finally:
```

```
        free_model(model)


    return dec_values
def cross_validation(
    np.ndarray[np.float64_t, ndim=2, mode='c'] X,
    np.ndarray[np.float64_t, ndim=1, mode='c'] Y,
    int n_fold, svm_type=0, kernel='rbf', int degree=3,
    double gamma=0.1, double coef0=0., double tol=1e-3,
    double C=1., double nu=0.5, double epsilon=0.1,
    np.ndarray[np.float64_t, ndim=1, mode='c']
        class_weight=np.empty(0),
    np.ndarray[np.float64_t, ndim=1, mode='c']
        sample_weight=np.empty(0),
    int shrinking=0, int probability=0, double cache_size=100.,
    int max_iter=-1,
    int random_seed=0):
    """
    Binding of the cross-validation routine (low-level routine)
    Parameters
    ----------
    X : array-like, dtype=float, size=[n_samples, n_features]
    Y : array, dtype=float, size=[n_samples]
        target vector
    svm_type : {0, 1, 2, 3, 4}
        Type of SVM: C SVC, nu SVC, one class, epsilon SVR, nu SVR
    kernel : {'linear', 'rbf', 'poly', 'sigmoid', 'precomputed'}
        Kernel to use in the model: linear, polynomial, RBF, sigmoid
```

```
    or precomputed.
degree : int
    Degree of the polynomial kernel (only relevant if kernel is
    set to polynomial)
gamma : float
    Gamma parameter in rbf, poly and sigmoid kernels. Ignored by other
    kernels. 0.1 by default.
coef0 : float
    Independent parameter in poly/sigmoid kernel.
tol : float
    Stopping criteria.
C : float
    C parameter in C-Support Vector Classification
nu : float
cache_size : float
random_seed : int, optional
    Seed for the random number generator used for probability estimates.
    0 by default.
Returns
-------
target : array, float
"""

cdef svm_parameter param
cdef svm_problem problem
cdef svm_model *model
cdef const char *error_msg
```

```
cdef np.npy_intp SV_len
cdef np.npy_intp nr

if len(sample_weight) == 0:
   sample_weight = np.ones(X.shape[0], dtype=np.float64)
else:
   assert sample_weight.shape[0] == X.shape[0], \
         "sample_weight and X have incompatible shapes: " + \
         "sample_weight has %s samples while X has %s" % \
         (sample_weight.shape[0], X.shape[0])

if X.shape[0] < n_fold:
   raise ValueError("Number of samples is less than number of folds")

# set problem
kernel_index = LIBSVM_KERNEL_TYPES.index(kernel)
set_problem(
   &problem, X.data, Y.data, sample_weight.data, X.shape, kernel_index)
if problem.x == NULL:
   raise MemoryError("Seems we've run out of memory")
cdef np.ndarray[np.int32_t, ndim=1, mode='c'] \
   class_weight_label = np.arange(class_weight.shape[0], dtype=np.int32)

# set parameters
set_parameter(
   &param, svm_type, kernel_index, degree, gamma, coef0, nu, cache_size,
   C, tol, tol, shrinking, probability, <int>
```

```
        class_weight.shape[0], class_weight_label.data,
        class_weight.data, max_iter, random_seed)

    error_msg = svm_check_parameter(&problem, &param);
    if error_msg:
        raise ValueError(error_msg)

    cdef np.ndarray[np.float64_t, ndim=1, mode='c'] target
    try:
        target = np.empty((X.shape[0]), dtype=np.float64)
        with nogil:
            svm_cross_validation(&problem, &param, n_fold, <double *> target.data)
    finally:
        free(problem.x)

    return target
def set_verbosity_wrap(int verbosity):
    """
    Control verbosity of libsvm library
    """
    set_verbosity(verbosity)
```

Extra Analysis

For curiosity in results, I am building Logistic regression for comparing the accuracy and f_1 score.

Category 12: Logistic Regression Algorithm [Regression Analysis]

Logistic regression predicts the probability of an outcome that can only have two values (i.e. a dichotomy). The prediction is based on the use of one or several predictors (numerical and categorical). A linear regression is not appropriate for predicting the value of a binary variable for two reasons:

- A linear regression will predict values outside the acceptable range (e.g. predicting probabilities outside the range 0 to 1)

- Since the dichotomous experiments can only have one of two possible values for each experiment, the residuals will not be normally distributed about the predicted line.

On the other hand, a logistic regression produces a logistic curve, which is limited to values between 0 and 1. Logistic regression is similar to a linear regression, but the curve is constructed using the natural logarithm of the “odds” of the target variable, rather than the probability. Moreover, the predictors do not have to be normally distributed or have equal variance in each group.

Category 12: LogisticRegression

Note: For Curosity in finding high accuracy I have implemented Logistic Regression algorithm for comparing accuracy.

```
In [72]: from sklearn.linear_model import LogisticRegression

In [73]: LR_Classifier = LogisticRegression(random_state=250)

In [74]: LR_Classifier.fit(X_train,y_train)

Out[74]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                  intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                  penalty='l2', random_state=250, solver='liblinear', tol=0.0001,
                  verbose=0, warm_start=False)

In [75]: Predictions_Logistic = LR_Classifier.predict(X_test)

In [76]: Accuracy_Logistic = accuracy_score(y_true = y_test, y_pred = Predictions_Logistic)

In [77]: Accuracy_Logistic

Out[77]: 0.7432432432432432

In [78]: from sklearn import metrics
         confusion=metrics.confusion_matrix(y_test, Predictions_Logistic)
         print(confusion)
         TP_LR = confusion[1, 1]
         TN_LR = confusion[0, 0]
         FP_LR = confusion[0, 1]
         FN_LR = confusion[1, 0]

         [[68 13]
          [25 42]]
```

Calculating Misclassification Rate, Precision, Recall, F1 -Score, Sensitivity, Specificity for Logistic Regression:

```
In [79]: # Calculating "Misclassification Rate"how often is the classifier incorrect?
         Mis_Rate_LR = (1 - metrics.accuracy_score(y_test, Predictions_Logistic))
         print("Misclassification Rate: ",Mis_Rate_LR)

         #Precision: When a positive value is predicted, how often is the prediction correct?
         precision_LR = metrics.precision_score(y_test, Predictions_Logistic)
         print("Precision: ",precision_LR)

         #Recall:Recall is the fraction of the relevant results that are successfully retrieved.
         recall_LR = metrics.recall_score(y_test, Predictions_Logistic)
         print("Recall: ",recall_LR)

         #F1 Score: balanced F-score or F-measure which measure's test accuracy.
         f1_LR=f1_score(y_test,Predictions_Logistic)
         print("F1_Score:", f1_LR)

         #Sensitivity: When the actual value is positive, how often is the prediction correct?
         #Also known as "True Positive Rate" or "Recall"
         sensitivity_LR = TP_LR / float(FN_LR + TP_LR)
         print("Sensitivity: ",sensitivity_LR)

         #Specificity: When the actual value is negative, how often is the prediction correct?
         #How "specific" (or "selective") is the classifier in predicting positive instances?
         specificity_LR = TN_LR / (TN_LR + FP_LR)
         print("Specificity: ",specificity_LR)

         Misclassification Rate:  0.2567567567567568
         Precision:  0.7311827956989247
         Recall:  0.8395061728395061
         F1_Score: 0.7816091954022988
         Sensitivity:  0.6268656716417911
         Specificity:  0.8395061728395061
```

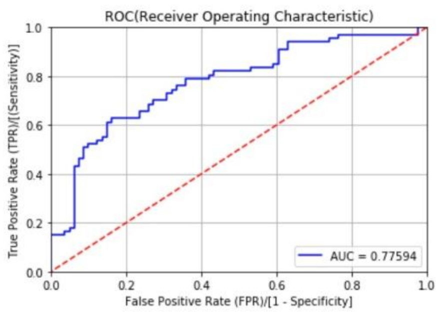
Receiver Operating Characteristic (ROC) for Logistic Regression:

In [80]:
```
y_pred_prob = LR_Classifier.predict_proba(X_test)[:, 1]

fpr, tpr, thresholds = metrics.roc_curve(y_test,y_pred_prob,pos_label=2)
```

In [81]:
```
roc_auc = auc(fpr, tpr)
```

In [82]:
```
plt.title('ROC(Receiver Operating Characteristic)')
plt.plot(fpr, tpr, 'b', label='AUC = %0.5f'% roc_auc)
plt.legend(loc='lower right')
plt.rcParams['font.size'] = 10
plt.plot([1,0],[1,0],'r--')
plt.xlim([0,1.0])
plt.ylim([0,1.0])
plt.ylabel('True Positive Rate (TPR)/[(Sensitivity)]')
plt.xlabel('False Positive Rate (FPR)/[1 - Specificity]')
plt.grid(True)
plt.show()
```



In [83]:
```
def evaluate_threshold(threshold):
    print('Sensitivity:', tpr[thresholds > threshold][-1])
    print('Specificity:', 1 - fpr[thresholds > threshold][-1])
```

In [84]:
```
evaluate_threshold(0.5)
```

```
Sensitivity: 0.6268656716417911
Specificity: 0.8395061728395061
```

In [85]:
```
#loss function
from sklearn.metrics import log_loss
Loss_Logistic = log_loss(y_test,y_pred_prob)
Loss_Logistic
```

Out[85]:
```
0.6022166534726436
```

RESULT - Comparing Decision Tree, SVM and Logistic Regression

RESULT - Comparing Decision Tree , SVM and Logistic Regression.

In [86]:
```
resultdf = pd.DataFrame(columns=["Model","Accuracy","Loss","TP","FP","TN","FN","Sensitivity","Specificity","Precision",
resultdf.loc[0] = ["Decision Tree", Accuracy_DT,Loss_DT,TP_DT,FP_DT,TN_DT,FN_DT,sensitivity_DT, specificity_DT,precisio
resultdf.loc[1] = ["SVM", Accuracy_SVM, Loss_SVM, TP_SVM,FP_SVM,TN_SVM,FN_SVM,sensitivity_SVM, specificity_SVM,precisio
resultdf.loc[2] = ["LogisticRegression", Accuracy_Logistic, Loss_Logistic, TP_LR,FP_LR,TN_LR,FN_LR,sensitivity_LR, spec
resultdf
```

Out[86]:

	Model	Accuracy	Loss	TP	FP	TN	FN	Sensitivity	Specificity	Precision	Recall	F1 Score
0	Decision Tree	0.824324	0.846209	57	16	65	10	0.850746	0.802469	0.866667	0.802469	0.833333
1	SVM	0.709459	0.589936	62	38	43	5	0.925373	0.530864	0.895833	0.530864	0.666667
2	LogisticRegression	0.743243	0.602217	42	13	68	25	0.626866	0.839506	0.731183	0.839506	0.781609

In [87]:
```
arraycuracy =[]
arraycuracy.append(Accuracy_DT)
arraycuracy.append(Accuracy_SVM)
arraycuracy.append(Accuracy_Logistic)
```

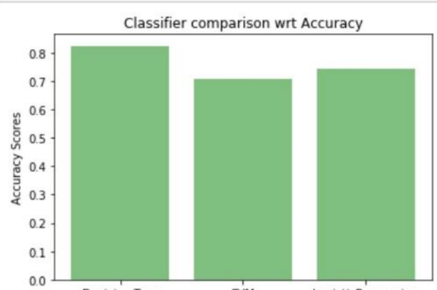
In [88]:
```
precisionray =[]
precisionray.append(precision_DT)
precisionray.append(precision_SVM)
precisionray.append(precision_LR)
```

In [89]:
```
recallray =[]
recallray.append(recall_DT)
recallray.append(recall_SVM)
recallray.append(recall_LR)
```

In [90]:
```
f1ray=[]
f1ray.append(f1_DT)
f1ray.append(f1_SVM)
f1ray.append(f1_LR)
```

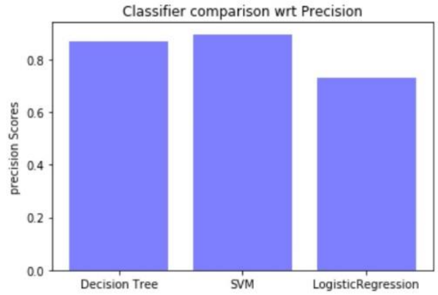
Comparison with respect to Accuracy:

In [91]:
```
import matplotlib.pyplot as plt
classifier_names=('Decision Tree','SVM','LogisticRegression')
y_axis=np.arange(len(classifier_names))
plt.bar(y_axis, arraycuracy, alpha=0.5,color='Green')
plt.xticks(y_axis, classifier_names)
plt.ylabel('Accuracy Scores')
plt.title('Classifier comparison wrt Accuracy')
plt.show()
```



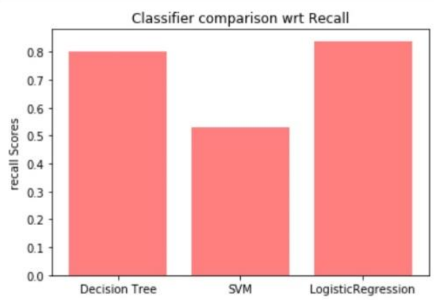
Comparison with respect to Precision:

In [92]:
```
y_axis=np.arange(len(classifier_names))
plt.bar(y_axis,precisionray, alpha=0.5,color='Blue',ecolor='black')
plt.xticks(y_axis, classifier_names)
plt.ylabel('precision Scores')
plt.title('Classifier comparison wrt Precision')
plt.show()
```



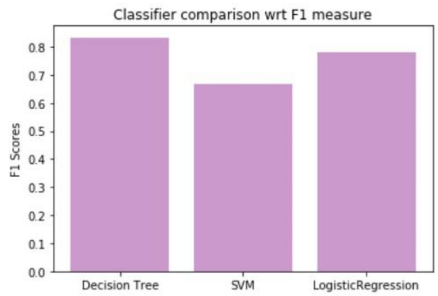
Comparison with respect to Recall:

In [93]:
```
y_axis=np.arange(len(classifier_names))
plt.bar(y_axis,recallray, alpha=0.5,color='Red',ecolor='black')
plt.xticks(y_axis, classifier_names)
plt.ylabel('recall Scores')
plt.title('Classifier comparison wrt Recall')
plt.show()
```



Comparison with respect to F1 Score:

```
In [94]: y_axis=np.arange(len(classifier_names))
         plt.bar(y_axis,f1ray, alpha=0.4,color='Purple',ecolor='black')
         plt.xticks(y_axis, classifier_names)
         plt.ylabel('F1 Scores')
         plt.title('Classifier comparison wrt F1 measure')
         plt.show()
```



Conclusion

After implementing the Logistic regression algorithm on the same data set it performed better than the SVM but not as compared with the Decision tree algorithm. The parameters like Accuracy, Precision, Specificity, Sensitivity and F1 Score clearly interprets that the Decision tree model's accuracy is much better than the others. While, if we consider Precision then SVM performed well as represented on result.

Hence, The Decision Tree model that I have built gives **82 %** accuracy on the testing data.

Links and References

1. Jupyter Notebook and Anaconda

http://jupyter.org/install

https://www.anaconda.com/download/#macos

2. Dataset Primary:

https://archive.ics.uci.edu/ml/machine-learning-databases/00445/ OR

https://archive.ics.uci.edu/ml/datasets/Absenteeism+at+work#

3. Source Code:

https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/tree/tree.py

https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/svm/libsvm.pyx

4. Scikit-learn:

- Decision Tree:

  http://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart

  http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

- SVM

  http://scikit-learn.org/stable/modules/svm.html

  http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[End]