# Course: CS634 - 001

# Professor: Jason Wang

# Student Name: Amey Pai Raiker

# UCID: ap2225

# Email: ap2225@njit,.edu

# Wine quality Prediction

# Table of Contents

|  |  |  |
| --- | --- | --- |
| Section No | Name | Page no |
| 1 | Project Topic Overview | 3 |
| 2 | Dataset | 4 |
| 3 | Data Information | 5 |
| 4 | Data Visualization | 6 |
| 5 | Label Encoder | 8 |
| 6 | Train-Test split | 9 |
| 7 | Model Build | 10 |
| 8 | Confusion matrix and accuracy check | 11 |
| 9 | Project Source code | 12 |
| 10 | Support Vector Machine Source code | 16 |
| 11 | Decision Tree source code | 28 |

# Project Overview:

**Problem**: Based on 11 features such as fixed acidity and chlorides etc predict the quality of wine.

**Data details:**

1. Number of Instances: red wine - 1599

2. Number of Attributes: 11 + output attribute

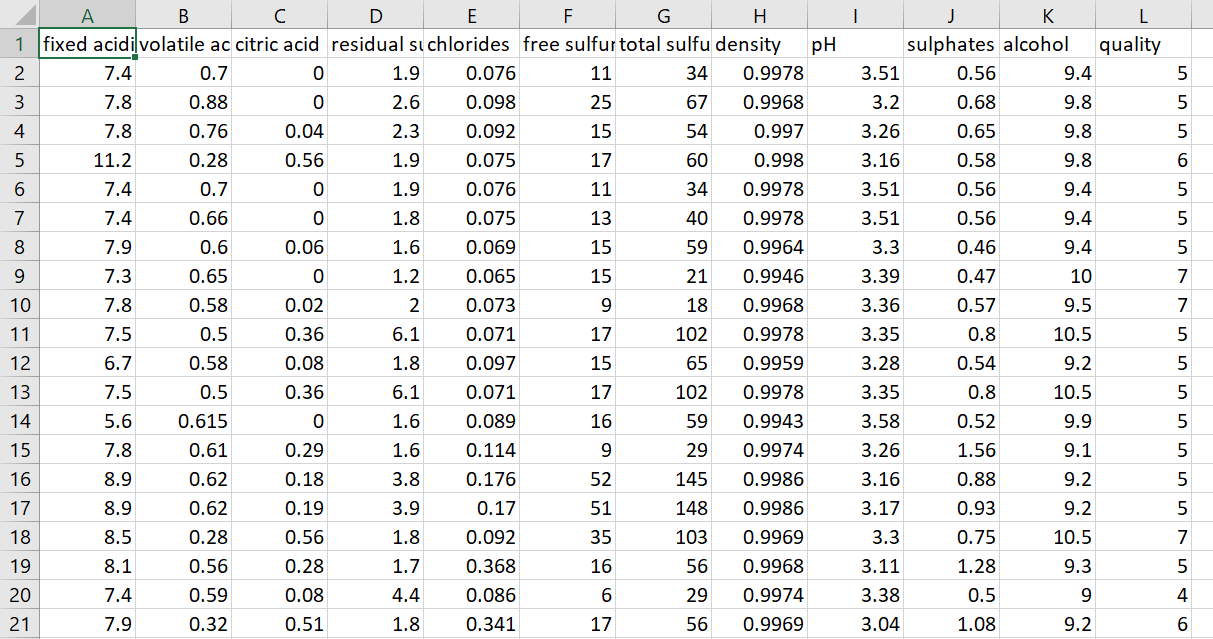
3. Attribute information:

Input variables (based on physicochemical tests):

* fixed acidity
* volatile acidity
* citric acid
* residual sugar
* chlorides
* free sulfur dioxide
* total sulfur dioxide
* density
* pH
* sulphates
* alcohol
* Output variable (based on sensory data):
  + quality (score between 0 and 10)

4. Missing Attribute Values: None

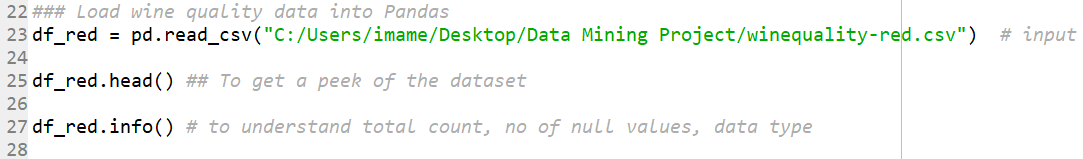
## Dataset



# Data Information

To get a brief idea about the dataset and how to go about it.

# Code and Output:

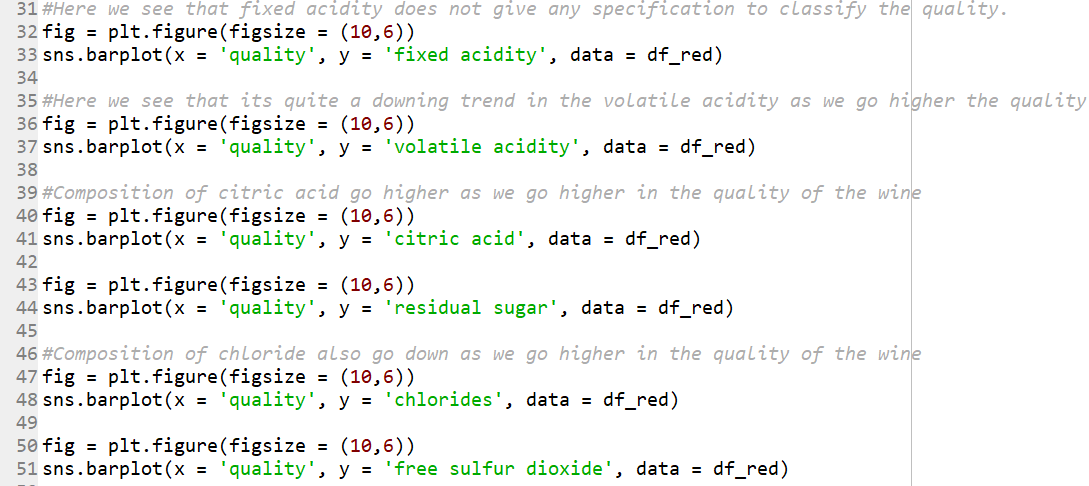


# 

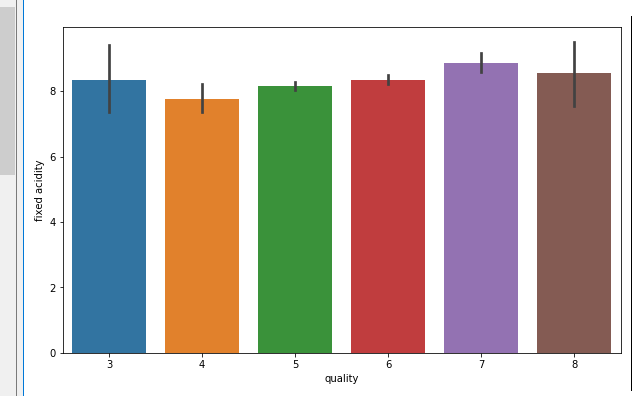
## Data Pre-processing

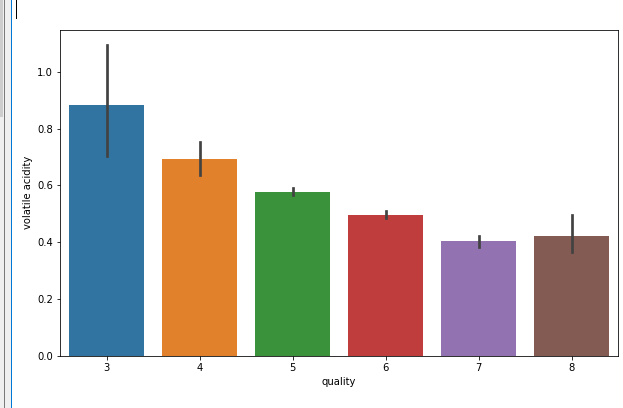
## Data Visualization

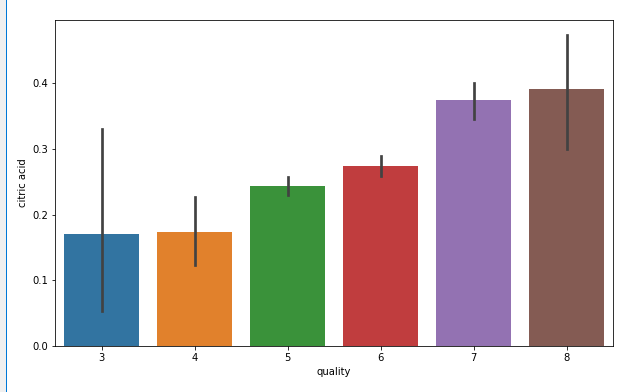
**Input code:**

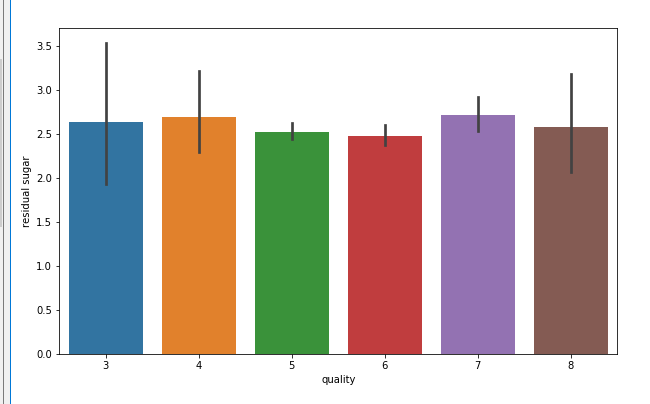


**Output:**





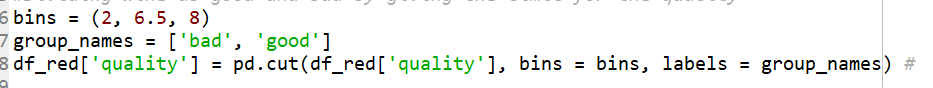




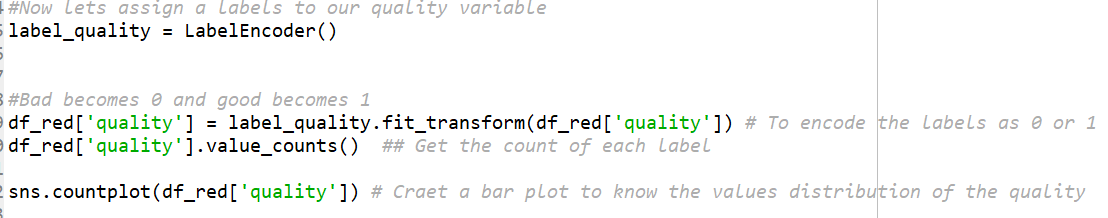
### Data Pre-processing Label Encoder

**Label Encoder to divide the output data into 2 types instead of 10 different floating values.**

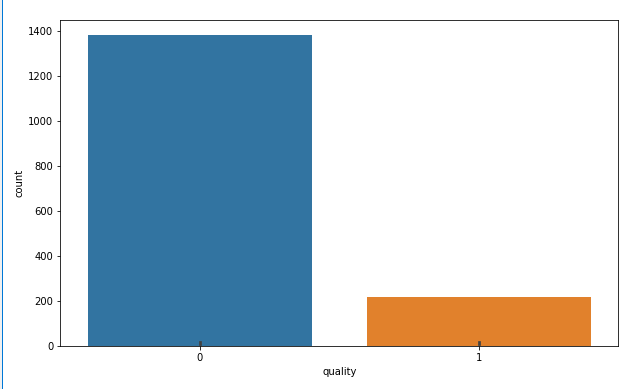
#### Input: To use label encoder library



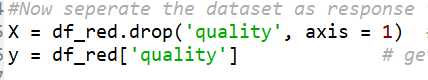
To separate values into labels:



**Output:**

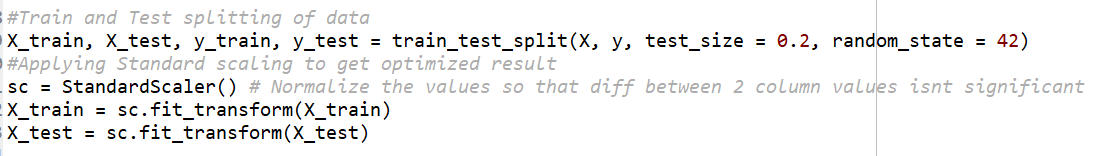


Separate the dataset into X and Y variable.



## Train-Test split

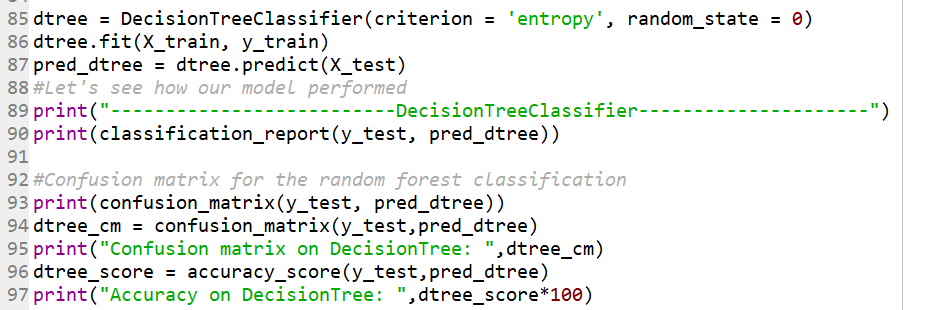
**Splitting the Data into Training and Testing with 80% training data and 20% testing data.**

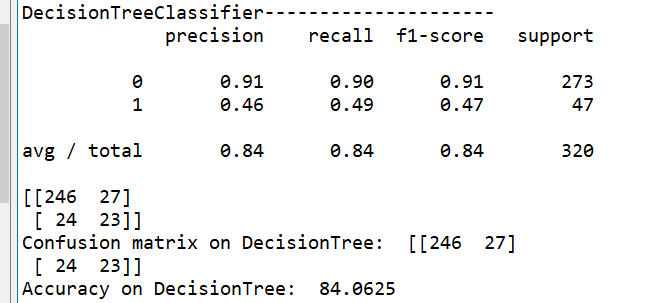


## Model Build

# Algorithm 1: Decision Trees

I have used the DecisionTreeClassifier() to implement the Decision tree algorithm. The source code f which has been attached.

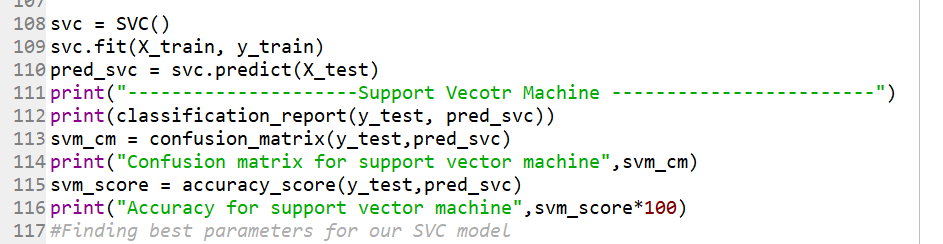


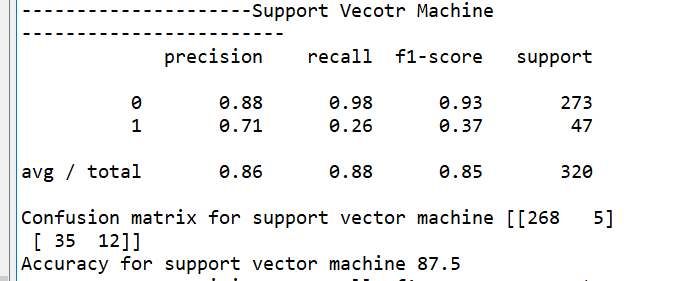


## Build a Model

# Algorithm 2: Support Vector Machine

I have used the SVC library from sklearn to implement the SVM algorithm for the given dataset. The source code of the decision tree algorithm has been attached.





# Actual Code for the term project

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

#from sklearn.ensemble import RandomForestClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.svm import SVC

from sklearn.linear\_model import SGDClassifier

from sklearn.metrics import confusion\_matrix, classification\_report

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.model\_selection import train\_test\_split, GridSearchCV, cross\_val\_score

from sklearn.metrics import accuracy\_score

### Load wine quality data into Pandas

df\_red = pd.read\_csv("C:/Users/imame/Desktop/Data Mining Project/winequality-red.csv") # input the red wine dataset

df\_red.head() ## To get a peek of the dataset

df\_red.info() # to understand total count, no of null values, data type

# Let's do some plotting to know how the data columns are distributed in the dataset

#Here we see that fixed acidity does not give any specification to classify the quality.

fig = plt.figure(figsize = (10,6))

sns.barplot(x = 'quality', y = 'fixed acidity', data = df\_red)

#Here we see that its quite a downing trend in the volatile acidity as we go higher the quality

fig = plt.figure(figsize = (10,6))

sns.barplot(x = 'quality', y = 'volatile acidity', data = df\_red)

#Composition of citric acid go higher as we go higher in the quality of the wine

fig = plt.figure(figsize = (10,6))

sns.barplot(x = 'quality', y = 'citric acid', data = df\_red)

fig = plt.figure(figsize = (10,6))

sns.barplot(x = 'quality', y = 'residual sugar', data = df\_red)

#Composition of chloride also go down as we go higher in the quality of the wine

fig = plt.figure(figsize = (10,6))

sns.barplot(x = 'quality', y = 'chlorides', data = df\_red)

fig = plt.figure(figsize = (10,6))

sns.barplot(x = 'quality', y = 'free sulfur dioxide', data = df\_red)

# Data Pre-processing

#Making binary classificaion for the response variable.

#Dividing wine as good and bad by giving the limit for the quality

bins = (2, 6.5, 8)

group\_names = ['bad', 'good']

df\_red['quality'] = pd.cut(df\_red['quality'], bins = bins, labels = group\_names) #

""" pd.cut divides the quality 2-6.5 as bad

6.5 - 8 as good """

#Now lets assign a labels to our quality variable

label\_quality = LabelEncoder()

#Bad becomes 0 and good becomes 1

df\_red['quality'] = label\_quality.fit\_transform(df\_red['quality']) # To encode the labels as 0 or 1

df\_red['quality'].value\_counts() ## Get the count of each label

sns.countplot(df\_red['quality']) # Craet a bar plot to know the values distribution of the quality

#Now seperate the dataset as response variable and feature variabes

X = df\_red.drop('quality', axis = 1) # Get all the columns except the last one

y = df\_red['quality'] # get the last column as the label set

#Train and Test splitting of data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 42)

#Applying Standard scaling to get optimized result

sc = StandardScaler() # Normalize the values so that diff between 2 column values isnt significant

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.fit\_transform(X\_test)

dtree = DecisionTreeClassifier(criterion = 'entropy', random\_state = 0)

dtree.fit(X\_train, y\_train)

pred\_dtree = dtree.predict(X\_test)

#Let's see how our model performed

print("--------------------------DecisionTreeClassifier---------------------")

print(classification\_report(y\_test, pred\_dtree))

#Confusion matrix for the random forest classification

print(confusion\_matrix(y\_test, pred\_dtree))

dtree\_cm = confusion\_matrix(y\_test,pred\_dtree)

print("Confusion matrix on DecisionTree: ",dtree\_cm)

dtree\_score = accuracy\_score(y\_test,pred\_dtree)

print("Accuracy on DecisionTree: ",dtree\_score\*100)

"""

sgd = SGDClassifier(penalty=None)

sgd.fit(X\_train, y\_train)

pred\_sgd = sgd.predict(X\_test)

print(classification\_report(y\_test, pred\_sgd))

print(confusion\_matrix(y\_test, pred\_sgd))

"""

svc = SVC()

svc.fit(X\_train, y\_train)

pred\_svc = svc.predict(X\_test)

print("---------------------Support Vecotr Machine ------------------------")

print(classification\_report(y\_test, pred\_svc))

svm\_cm = confusion\_matrix(y\_test,pred\_svc)

print("Confusion matrix for support vector machine",svm\_cm)

svm\_score = accuracy\_score(y\_test,pred\_svc)

print("Accuracy for support vector machine",svm\_score\*100)

# Code to implement SVM

|  |  |
| --- | --- |
|  |  |
|  | 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): |
|  | """ |
|  | Train the model using libsvm (low-level method) |
|  |  |
|  | Parameters |
|  | ---------- |
|  | X : array-like, dtype=float64, size=[n\_samples, n\_features] |
|  |  |
|  | Y : array, dtype=float64, size=[n\_samples] |
|  | target vector |
|  |  |
|  | svm\_type : {0, 1, 2, 3, 4}, optional |
|  | Type of SVM: C\_SVC, NuSVC, OneClassSVM, EpsilonSVR or NuSVR |
|  | respectively. 0 by default. |
|  |  |
|  | kernel : {'linear', 'rbf', 'poly', 'sigmoid', 'precomputed'}, optional |
|  | Kernel to use in the model: linear, polynomial, RBF, sigmoid |
|  | or precomputed. 'rbf' by default. |
|  |  |
|  | degree : int32, optional |
|  | Degree of the polynomial kernel (only relevant if kernel is |
|  | set to polynomial), 3 by default. |
|  |  |
|  | gamma : float64, optional |
|  | Gamma parameter in rbf, poly and sigmoid kernels. Ignored by other |
|  | kernels. 0.1 by default. |
|  |  |
|  | coef0 : float64, optional |
|  | Independent parameter in poly/sigmoid kernel. 0 by default. |
|  |  |
|  | tol : float64, optional |
|  | Numeric stopping criterion (WRITEME). 1e-3 by default. |
|  |  |
|  | C : float64, optional |
|  | C parameter in C-Support Vector Classification. 1 by default. |
|  |  |
|  | nu : float64, optional |
|  | 0.5 by default. |
|  |  |
|  | epsilon : double, optional |
|  | 0.1 by default. |
|  |  |
|  | class\_weight : array, dtype float64, shape (n\_classes,), optional |
|  | np.empty(0) by default. |
|  |  |
|  | sample\_weight : array, dtype float64, shape (n\_samples,), optional |
|  | np.empty(0) by default. |
|  |  |
|  | shrinking : int, optional |
|  | 1 by default. |
|  |  |
|  | probability : int, optional |
|  | 0 by default. |
|  |  |
|  | cache\_size : float64, optional |
|  | Cache size for gram matrix columns (in megabytes). 100 by default. |
|  |  |
|  | max\_iter : int (-1 for no limit), optional. |
|  | Stop solver after this many iterations regardless of accuracy |
|  | (XXX Currently there is no API to know whether this kicked in.) |
|  | -1 by default. |
|  |  |
|  | random\_seed : int, optional |
|  | Seed for the random number generator used for probability estimates. |
|  | 0 by default. |
|  |  |
|  | Returns |
|  | ------- |
|  | support : array, shape=[n\_support] |
|  | index of support vectors |
|  |  |
|  | support\_vectors : array, shape=[n\_support, n\_features] |
|  | support vectors (equivalent to X[support]). Will return an |
|  | empty array in the case of precomputed kernel. |
|  |  |
|  | n\_class\_SV : array |
|  | number of support vectors in each class. |
|  |  |
|  | sv\_coef : array |
|  | coefficients of support vectors in decision function. |
|  |  |
|  | intercept : array |
|  | intercept in decision function |
|  |  |
|  | probA, probB : array |
|  | probability estimates, empty array for probability=False |
|  | """ |
|  |  |
|  | 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.): |
|  | """ |
|  | Predict target values of X given a model (low-level method) |
|  |  |
|  | Parameters |
|  | ---------- |
|  | X : array-like, dtype=float, size=[n\_samples, n\_features] |
|  | 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'} |
|  | Type of kernel. |
|  | degree : int |
|  | Degree of the polynomial kernel. |
|  | 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. |
|  |  |
|  | Returns |
|  | ------- |
|  | dec\_values : array |
|  | predicted values. |
|  | """ |
|  | 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.): |
|  | """ |
|  | Predict probabilities |
|  |  |
|  | svm\_model stores all parameters needed to predict a given value. |
|  |  |
|  | For speed, all real work is done at the C level in function |
|  | copy\_predict (libsvm\_helper.c). |
|  |  |
|  | We have to reconstruct model and parameters to make sure we stay |
|  | in sync with the python object. |
|  |  |
|  | See sklearn.svm.predict for a complete list of parameters. |
|  |  |
|  | Parameters |
|  | ---------- |
|  | X : array-like, dtype=float |
|  | kernel : {'linear', 'rbf', 'poly', 'sigmoid', 'precomputed'} |
|  |  |
|  | Returns |
|  | ------- |
|  | dec\_values : array |
|  | predicted values. |
|  | """ |
|  | 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) |

##### Source code for Decision Tree

|  |
| --- |
| import sys |
|  | import math |
|  | import pandas as pd |
|  |  |
|  | class Node(object): |
|  | def \_\_init\_\_(self, attribute, threshold): |
|  | self.attr = attribute |
|  | self.thres = threshold |
|  | self.left = None |
|  | self.right = None |
|  | self.leaf = False |
|  | self.predict = None |
|  |  |
|  | # First select the threshold of the attribute to split set of test data on |
|  | # The threshold chosen splits the test data such that information gain is maximized |
|  | def select\_threshold(df, attribute, predict\_attr): |
|  | # Convert dataframe column to a list and round each value |
|  | values = df[attribute].tolist() |
|  | values = [ float(x) for x in values] |
|  | # Remove duplicate values by converting the list to a set, then sort the set |
|  | values = set(values) |
|  | values = list(values) |
|  | values.sort() |
|  | max\_ig = float("-inf") |
|  | thres\_val = 0 |
|  | # try all threshold values that are half-way between successive values in this sorted list |
|  | for i in range(0, len(values) - 1): |
|  | thres = (values[i] + values[i+1])/2 |
|  | ig = info\_gain(df, attribute, predict\_attr, thres) |
|  | if ig > max\_ig: |
|  | max\_ig = ig |
|  | thres\_val = thres |
|  | # Return the threshold value that maximizes information gained |
|  | return thres\_val |
|  |  |
|  | # Calculate info content (entropy) of the test data |
|  | def info\_entropy(df, predict\_attr): |
|  | # Dataframe and number of positive/negatives examples in the data |
|  | p\_df = df[df[predict\_attr] == 1] |
|  | n\_df = df[df[predict\_attr] == 0] |
|  | p = float(p\_df.shape[0]) |
|  | n = float(n\_df.shape[0]) |
|  | # Calculate entropy |
|  | if p == 0 or n == 0: |
|  | I = 0 |
|  | else: |
|  | I = ((-1\*p)/(p + n))\*math.log(p/(p+n), 2) + ((-1\*n)/(p + n))\*math.log(n/(p+n), 2) |
|  | return I |
|  |  |
|  | # Calculates the weighted average of the entropy after an attribute test |
|  | def remainder(df, df\_subsets, predict\_attr): |
|  | # number of test data |
|  | num\_data = df.shape[0] |
|  | remainder = float(0) |
|  | for df\_sub in df\_subsets: |
|  | if df\_sub.shape[0] > 1: |
|  | remainder += float(df\_sub.shape[0]/num\_data)\*info\_entropy(df\_sub, predict\_attr) |
|  | return remainder |
|  |  |
|  | # Calculates the information gain from the attribute test based on a given threshold |
|  | # Note: thresholds can change for the same attribute over time |
|  | def info\_gain(df, attribute, predict\_attr, threshold): |
|  | sub\_1 = df[df[attribute] < threshold] |
|  | sub\_2 = df[df[attribute] > threshold] |
|  | # Determine information content, and subract remainder of attributes from it |
|  | ig = info\_entropy(df, predict\_attr) - remainder(df, [sub\_1, sub\_2], predict\_attr) |
|  | return ig |
|  |  |
|  | # Returns the number of positive and negative data |
|  | def num\_class(df, predict\_attr): |
|  | p\_df = df[df[predict\_attr] == 1] |
|  | n\_df = df[df[predict\_attr] == 0] |
|  | return p\_df.shape[0], n\_df.shape[0] |
|  |  |
|  | # Chooses the attribute and its threshold with the highest info gain |
|  | # from the set of attributes |
|  | def choose\_attr(df, attributes, predict\_attr): |
|  | max\_info\_gain = float("-inf") |
|  | best\_attr = None |
|  | threshold = 0 |
|  | # Test each attribute (note attributes maybe be chosen more than once) |
|  | for attr in attributes: |
|  | thres = select\_threshold(df, attr, predict\_attr) |
|  | ig = info\_gain(df, attr, predict\_attr, thres) |
|  | if ig > max\_info\_gain: |
|  | max\_info\_gain = ig |
|  | best\_attr = attr |
|  | threshold = thres |
|  | return best\_attr, threshold |
|  |  |
|  | # Builds the Decision Tree based on training data, attributes to train on, |
|  | # and a prediction attribute |
|  | def build\_tree(df, cols, predict\_attr): |
|  | # Get the number of positive and negative examples in the training data |
|  | p, n = num\_class(df, predict\_attr) |
|  | # If train data has all positive or all negative values |
|  | # then we have reached the end of our tree |
|  | if p == 0 or n == 0: |
|  | # Create a leaf node indicating it's prediction |
|  | leaf = Node(None,None) |
|  | leaf.leaf = True |
|  | if p > n: |
|  | leaf.predict = 1 |
|  | else: |
|  | leaf.predict = 0 |
|  | return leaf |
|  | else: |
|  | # Determine attribute and its threshold value with the highest |
|  | # information gain |
|  | best\_attr, threshold = choose\_attr(df, cols, predict\_attr) |
|  | # Create internal tree node based on attribute and it's threshold |
|  | tree = Node(best\_attr, threshold) |
|  | sub\_1 = df[df[best\_attr] < threshold] |
|  | sub\_2 = df[df[best\_attr] > threshold] |
|  | # Recursively build left and right subtree |
|  | tree.left = build\_tree(sub\_1, cols, predict\_attr) |
|  | tree.right = build\_tree(sub\_2, cols, predict\_attr) |
|  | return tree |
|  |  |
|  | # Given a instance of a training data, make a prediction of healthy or colic |
|  | # based on the Decision Tree |
|  | # Assumes all data has been cleaned (i.e. no NULL data) |
|  | def predict(node, row\_df): |
|  | # If we are at a leaf node, return the prediction of the leaf node |
|  | if node.leaf: |
|  | return node.predict |
|  | # Traverse left or right subtree based on instance's data |
|  | if row\_df[node.attr] <= node.thres: |
|  | return predict(node.left, row\_df) |
|  | elif row\_df[node.attr] > node.thres: |
|  | return predict(node.right, row\_df) |
|  |  |
|  | # Given a set of data, make a prediction for each instance using the Decision Tree |
|  | def test\_predictions(root, df): |
|  | num\_data = df.shape[0] |
|  | num\_correct = 0 |
|  | for index,row in df.iterrows(): |
|  | prediction = predict(root, row) |
|  | if prediction == row['Outcome']: |
|  | num\_correct += 1 |
|  | return round(num\_correct/num\_data, 2) |
|  |  |
|  | # Prints the tree level starting at given level |
|  | def print\_tree(root, level): |
|  | print(counter\*" ", end="") |
|  | if root.leaf: |
|  | print(root.predict) |
|  | else: |
|  | print(root.attr) |
|  | if root.left: |
|  | print\_tree(root.left, level + 1) |
|  | if root.right: |
|  | print\_tree(root.right, level + 1) |
|  |  |
|  | # Cleans the input data, removes 'Diagnosis' column and adds 'Outcome' column |
|  | # where 0 means healthy and 1 means colic |
|  | def clean(csv\_file\_name): |
|  | df = pd.read\_csv(csv\_file\_name, header=None) |
|  | df.columns = ['K', 'Na', 'CL', 'HCO', 'Endotoxin', 'Anioingap', 'PLA2', 'SDH', 'GLDH', 'TPP', 'Breath rate', 'PCV', 'Pulse rate', 'Fibrinogen', 'Dimer', 'FibPerDim', 'Diagnosis'] |
|  | # Create new column 'Outcome' that assigns healthy horses a value of 0 (negative case) and |
|  | # horses with colic a value of 1 (positive case), this makes creating our decision tree easier |
|  | df['Outcome'] = 0 |
|  | df.loc[df['Diagnosis'] == 'colic.', 'Outcome'] = 1 |
|  | df.drop(['Diagnosis'], axis=1 ) |
|  | cols = df.columns |
|  | df[cols] = df[cols].apply(pd.to\_numeric, errors='coerce') |
|  | return df |
|  |  |
|  | def main(): |
|  | # An example use of 'build\_tree' and 'predict' |
|  | df\_train = clean('horseTrain.txt') |
|  | attributes = ['K', 'Na', 'CL', 'HCO', 'Endotoxin', 'Anioingap', 'PLA2', 'SDH', 'GLDH', 'TPP', 'Breath rate', 'PCV', 'Pulse rate', 'Fibrinogen', 'Dimer', 'FibPerDim'] |
|  | root = build\_tree(df\_train, attributes, 'Outcome') |
|  |  |
|  | print("Accuracy of test data") |
|  | df\_test = clean('horseTest.txt') |
|  | print(str(test\_predictions(root, df\_test)\*100.0) + '%') |
|  |  |
|  | if \_\_name\_\_ == '\_\_main\_\_': |
|  | main() |