Course: CS634 - 001

Professor: Jason Wang

Student Name: Amey Pai Raiker

UCID: ap2225

Email: ap2225@njit,.edu

Wine quality Prediction

Table of Contents

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

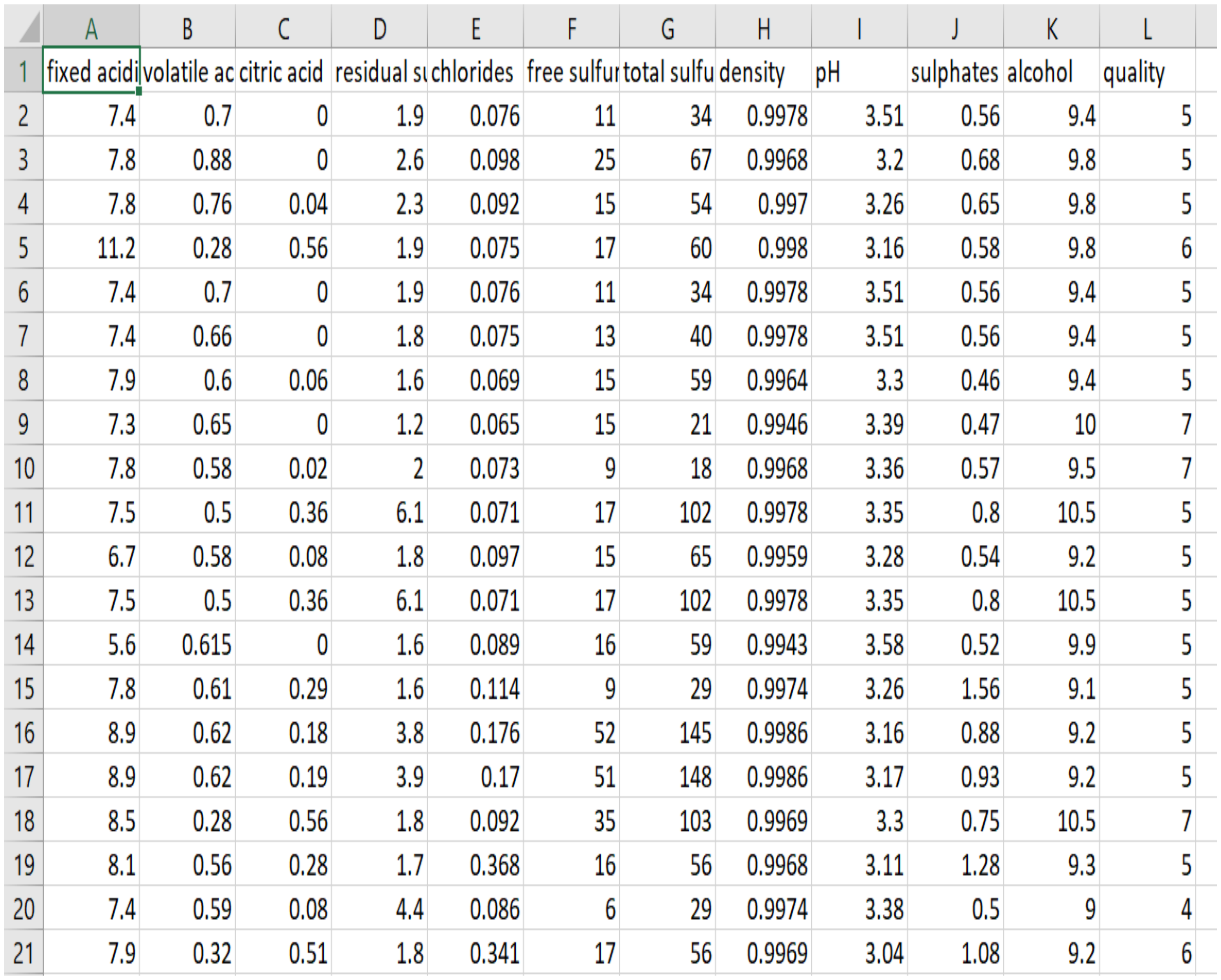
3. Attribute information:

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

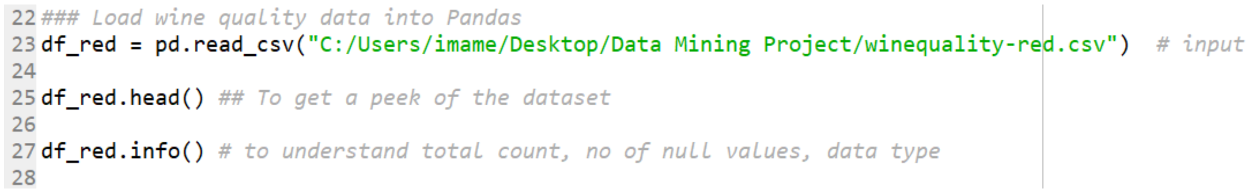


	A	B	C	D	E	F	G	H	I	J	K	L
1	fixed acidi	volatile ac	citric acid	residual su	chlorides	free sulfur	total sulfu	density	pH	sulphates	alcohol	quality
2	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
3	7.8	0.88	0	2.6	0.098	25	67	0.9968	3.2	0.68	9.8	5
4	7.8	0.76	0.04	2.3	0.092	15	54	0.997	3.26	0.65	9.8	5
5	11.2	0.28	0.56	1.9	0.075	17	60	0.998	3.16	0.58	9.8	6
6	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
7	7.4	0.66	0	1.8	0.075	13	40	0.9978	3.51	0.56	9.4	5
8	7.9	0.6	0.06	1.6	0.069	15	59	0.9964	3.3	0.46	9.4	5
9	7.3	0.65	0	1.2	0.065	15	21	0.9946	3.39	0.47	10	7
10	7.8	0.58	0.02	2	0.073	9	18	0.9968	3.36	0.57	9.5	7
11	7.5	0.5	0.36	6.1	0.071	17	102	0.9978	3.35	0.8	10.5	5
12	6.7	0.58	0.08	1.8	0.097	15	65	0.9959	3.28	0.54	9.2	5
13	7.5	0.5	0.36	6.1	0.071	17	102	0.9978	3.35	0.8	10.5	5
14	5.6	0.615	0	1.6	0.089	16	59	0.9943	3.58	0.52	9.9	5
15	7.8	0.61	0.29	1.6	0.114	9	29	0.9974	3.26	1.56	9.1	5
16	8.9	0.62	0.18	3.8	0.176	52	145	0.9986	3.16	0.88	9.2	5
17	8.9	0.62	0.19	3.9	0.17	51	148	0.9986	3.17	0.93	9.2	5
18	8.5	0.28	0.56	1.8	0.092	35	103	0.9969	3.3	0.75	10.5	7
19	8.1	0.56	0.28	1.7	0.368	16	56	0.9968	3.11	1.28	9.3	5
20	7.4	0.59	0.08	4.4	0.086	6	29	0.9974	3.38	0.5	9	4
21	7.9	0.32	0.51	1.8	0.341	17	56	0.9969	3.04	1.08	9.2	6

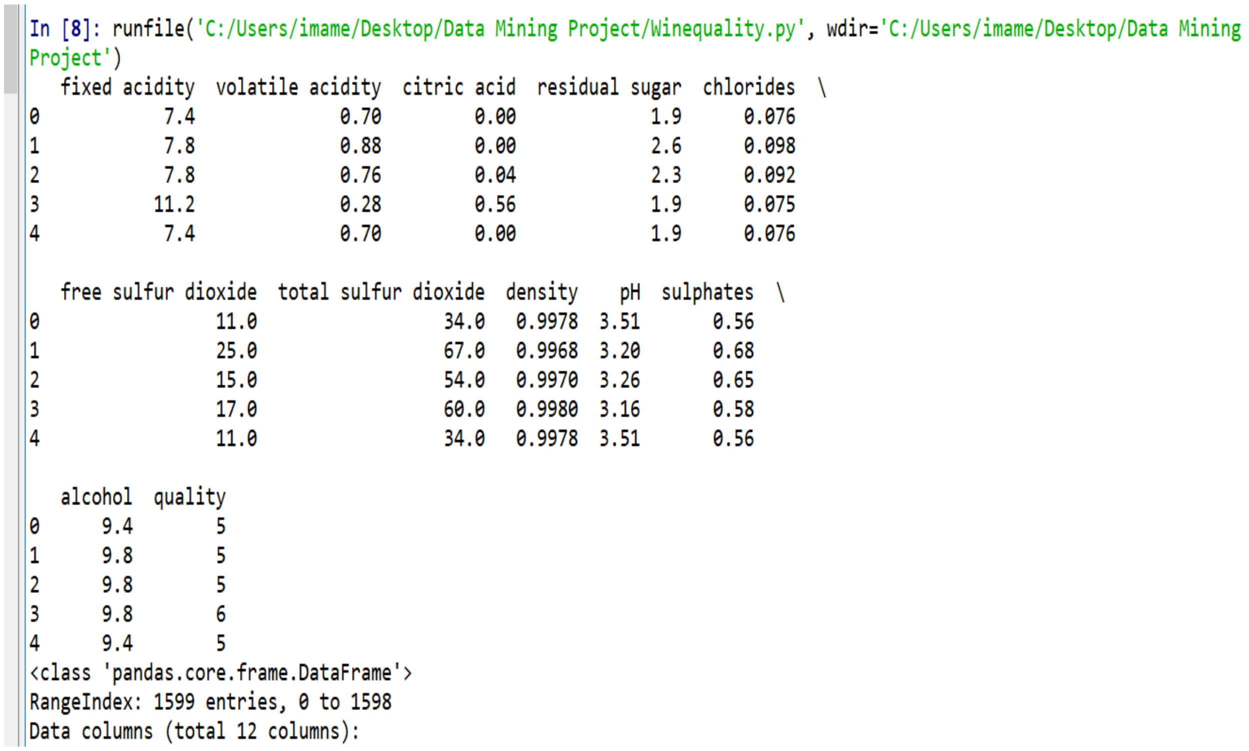
Data Information

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

Code and Output:



```
### Load wine quality data into Pandas
df_red = pd.read_csv("C:/Users/imame/Desktop/Data Mining Project/winequality-red.csv")  # input

df_red.head() ## To get a peek of the dataset

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

```



```
In [8]: runfile('C:/Users/imame/Desktop/Data Mining Project/Winequality.py', wdir='C:/Users/imame/Desktop/Data Mining
Project')
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
```

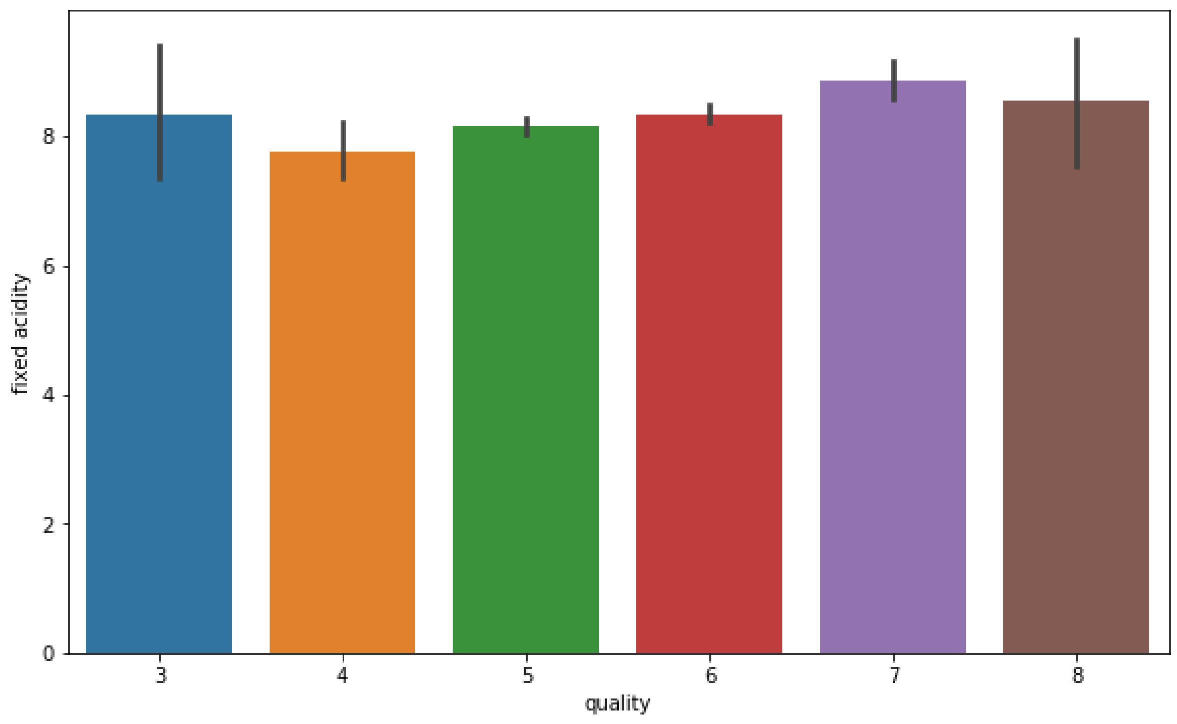
Data Pre-processing

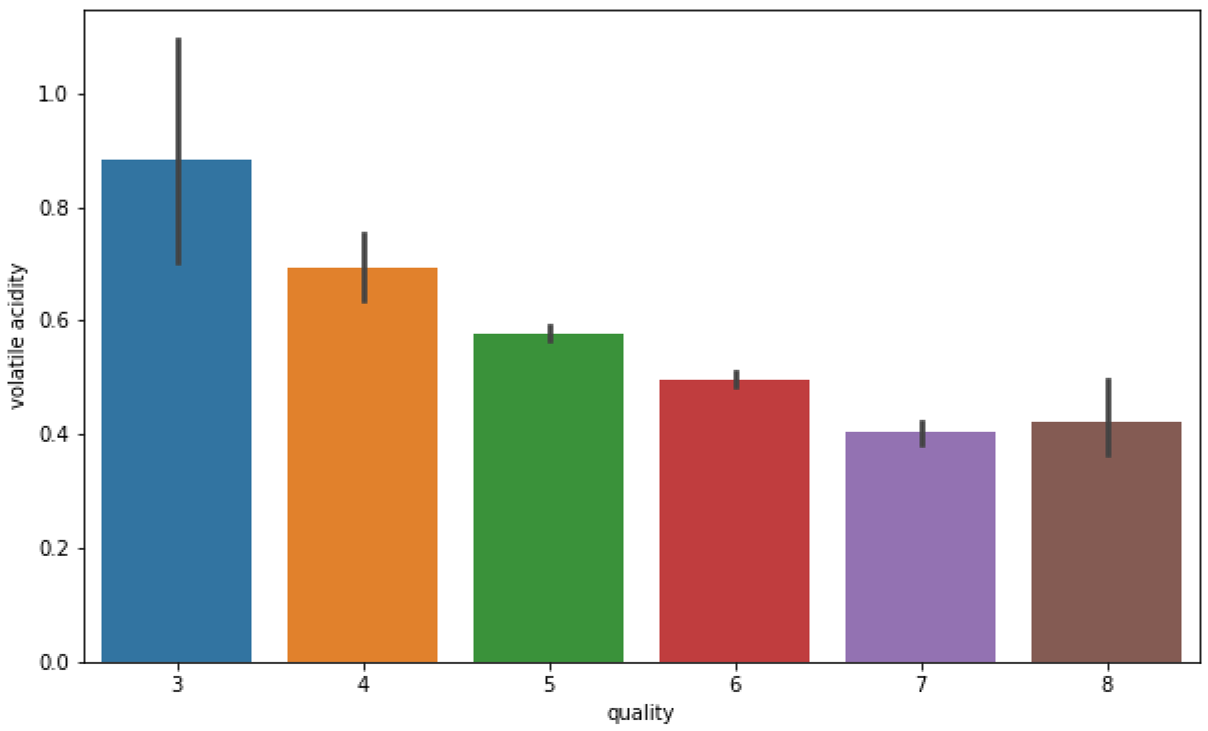
Data Visualization

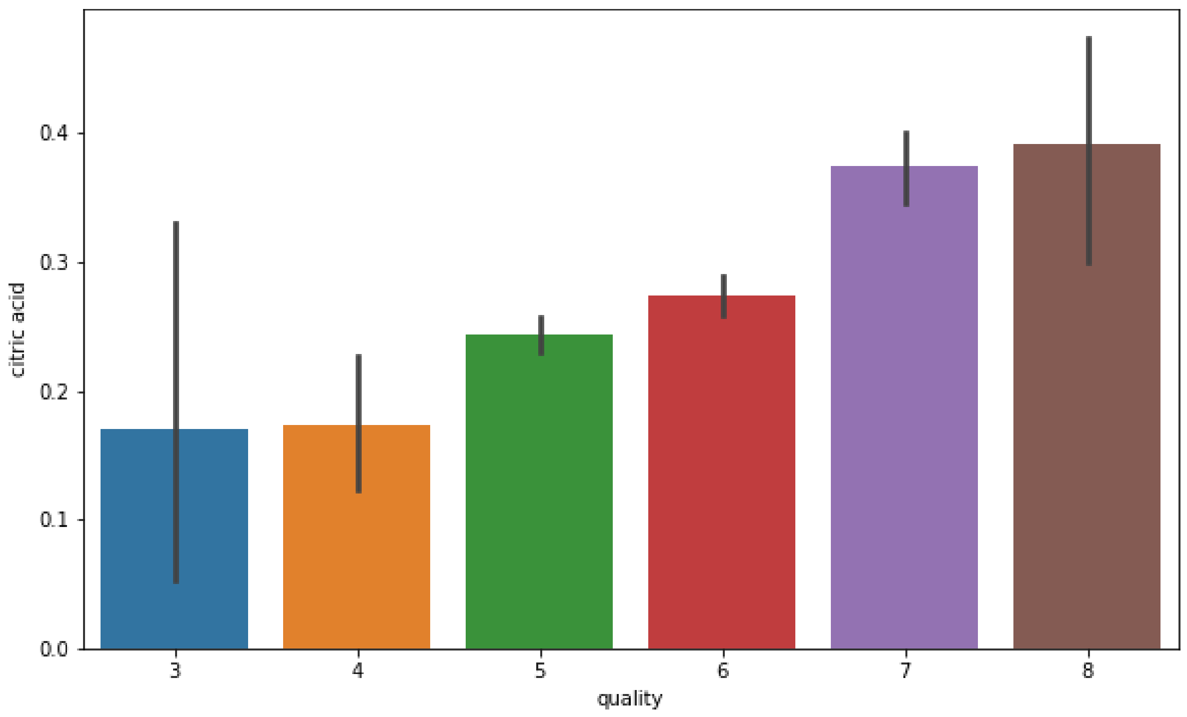
Input code:

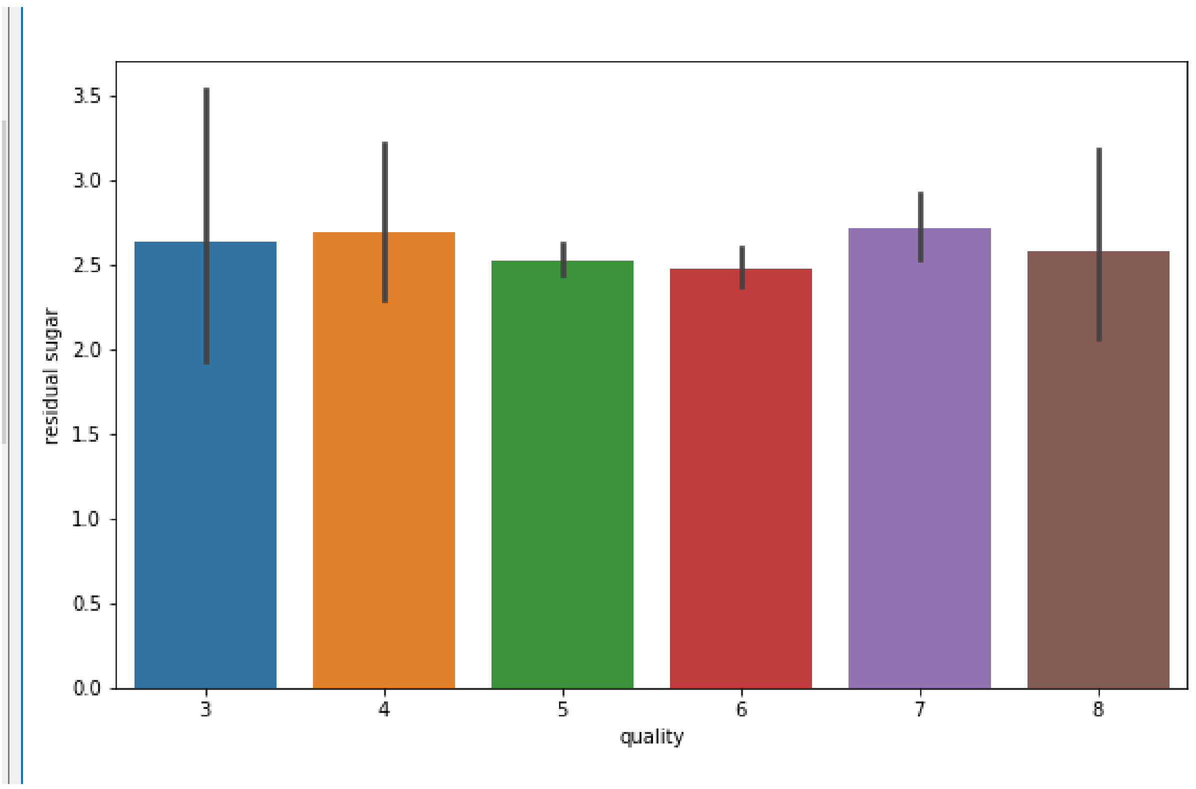
```
#Here we see that fixed acidity does not give any specification to classify the quality.
fig = plt.figure(figsize = (10,6))
sns.barplot(x = 'quality', y = 'fixed acidity', data = df_red)

#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)
```

Output:




volatile acidity
1.0
0.8
0.6
0.4
0.2
0.0
3
4
5
6
7
8
quality

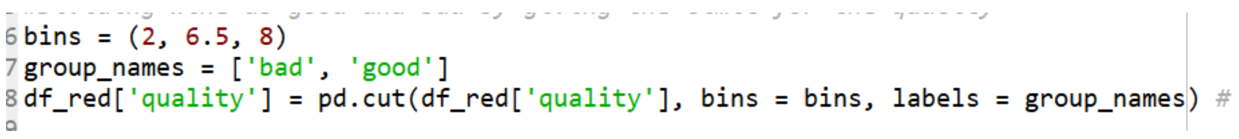

citric acid
0.4
0.3
0.2
0.1
0.0
3
4
5
6
7
8
quality



Data Pre-processing Label Encoder

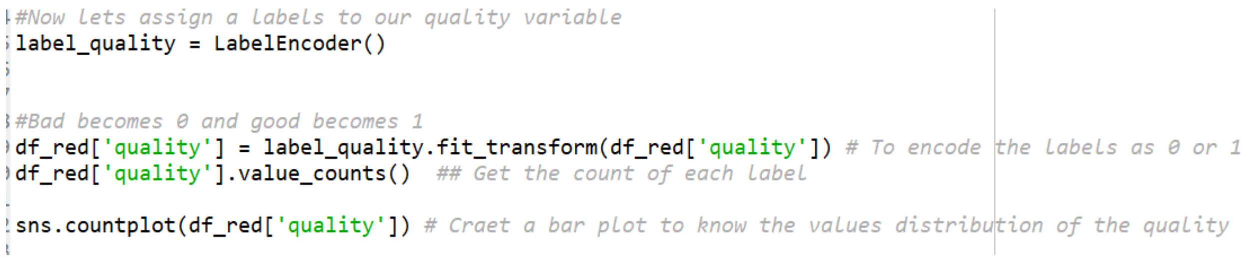
Label Encoder to divide the output data into 2 types instead of 10 different floating values.

Input: To use label encoder library



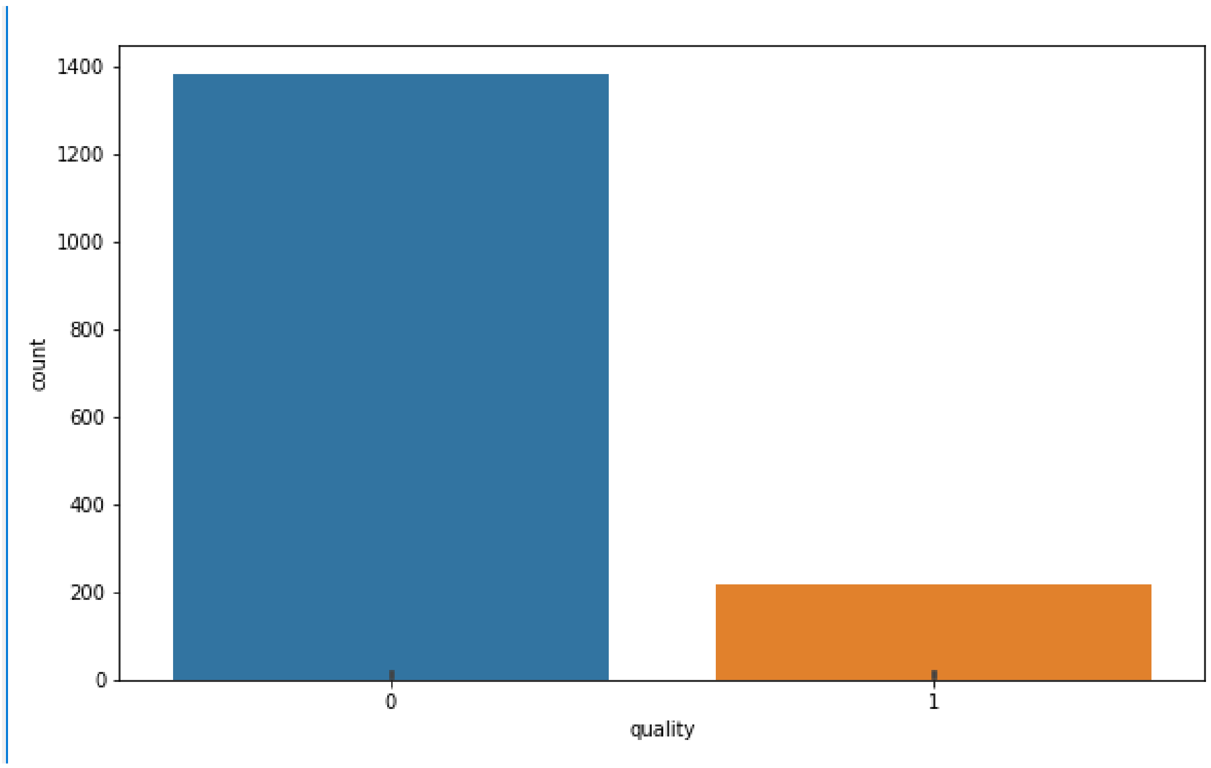
```
bins = (2, 6.5, 8)
group_names = ['bad', 'good']
df_red['quality'] = pd.cut(df_red['quality'], bins = bins, labels = group_names) #
```

To separate values into labels:

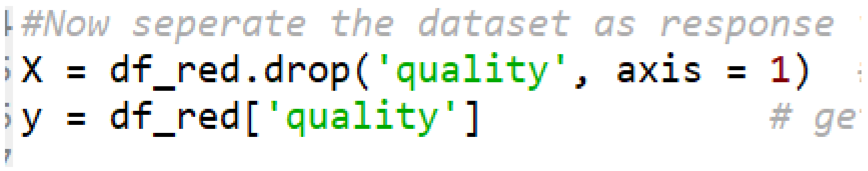


```
#Now lets assign a labels to our quality variable
label_quality = LabelEncoder()


#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
```

Output:



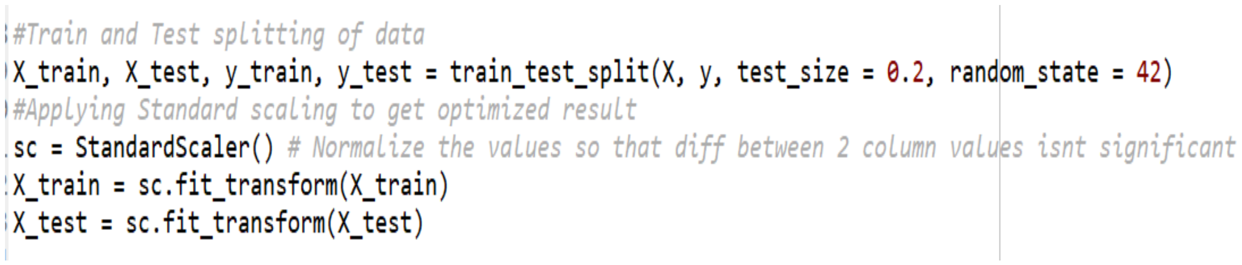
Separate the dataset into X and Y variable.



```
#Now seperate the dataset as response
X = df_red.drop('quality', axis = 1)
y = df_red['quality']            # ge
```

Train-Test split

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

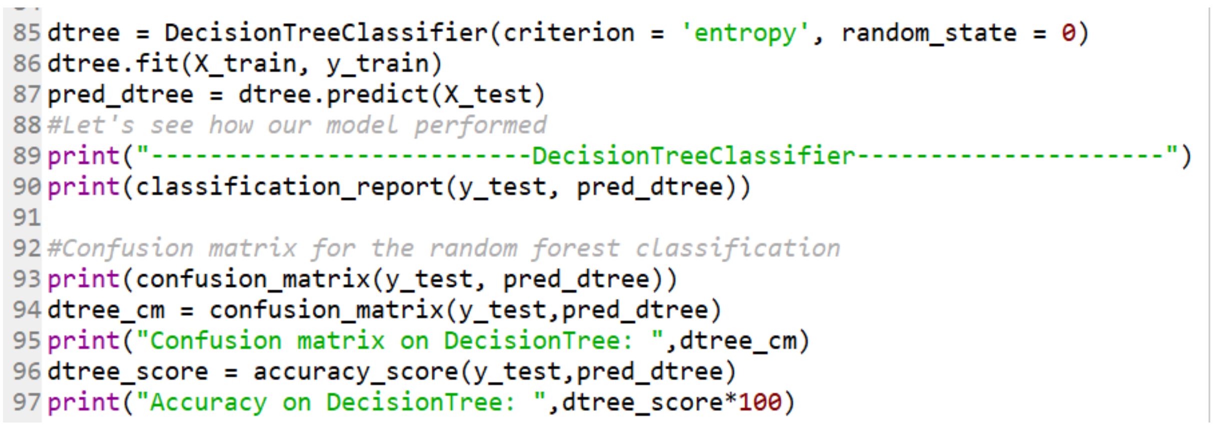


```
#Train and Test splitting of data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
#Applying Standard scaling to get optimized result
sc = StandardScaler() # Normalize the values so that diff between 2 column values isnt significant
X_train = sc.fit_transform(X_train)
X_test = sc.fit_transform(X_test)
```

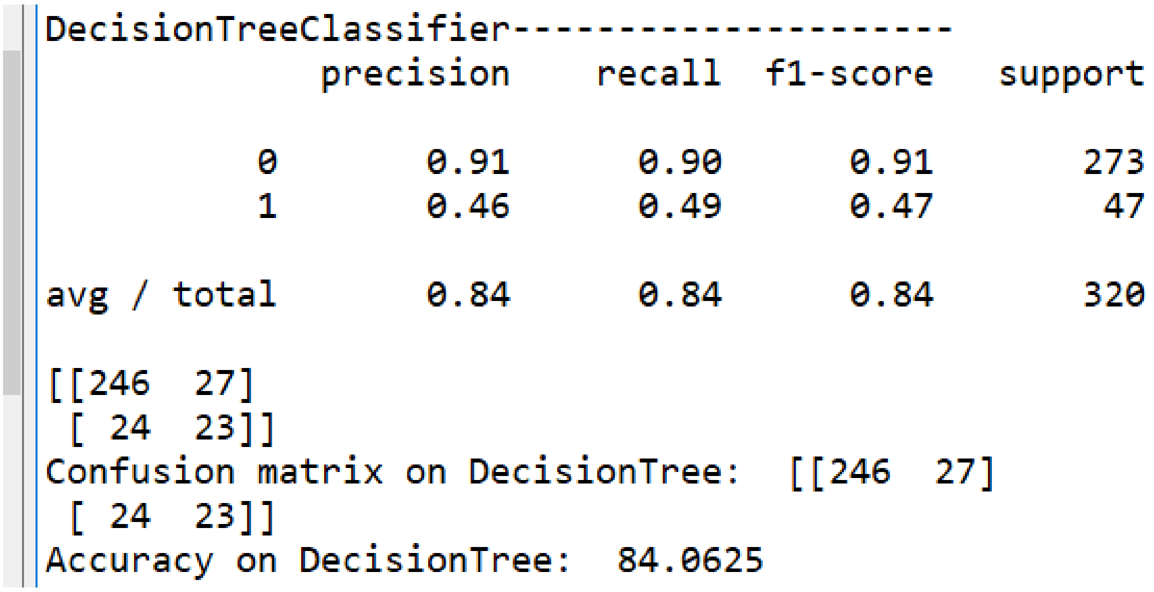
Model Build

Algorithm 1: Decision Trees

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



```python
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)
```

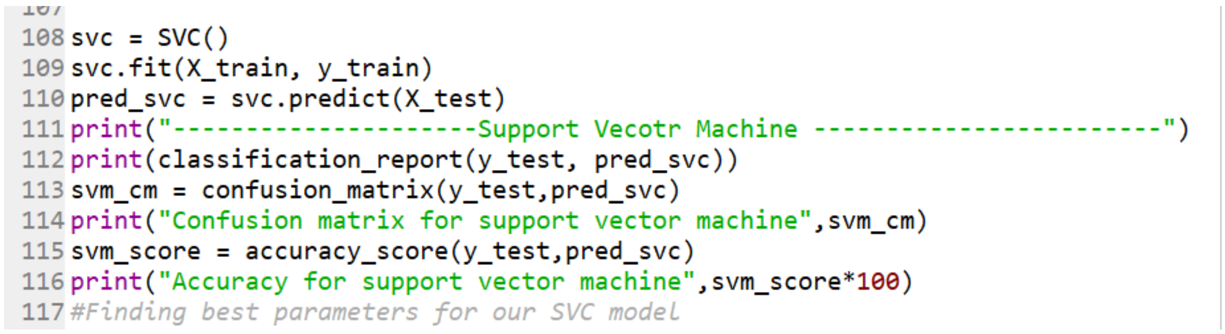


```
DecisionTreeClassifier--------------------
             precision    recall  f1-score   support

          0       0.91      0.90      0.91       273
          1       0.46      0.49      0.47        47

avg / total       0.84      0.84      0.84       320

[[246  27]
 [ 24  23]]
Confusion matrix on DecisionTree:  [[246  27]
 [ 24  23]]
Accuracy on DecisionTree:  84.0625
```

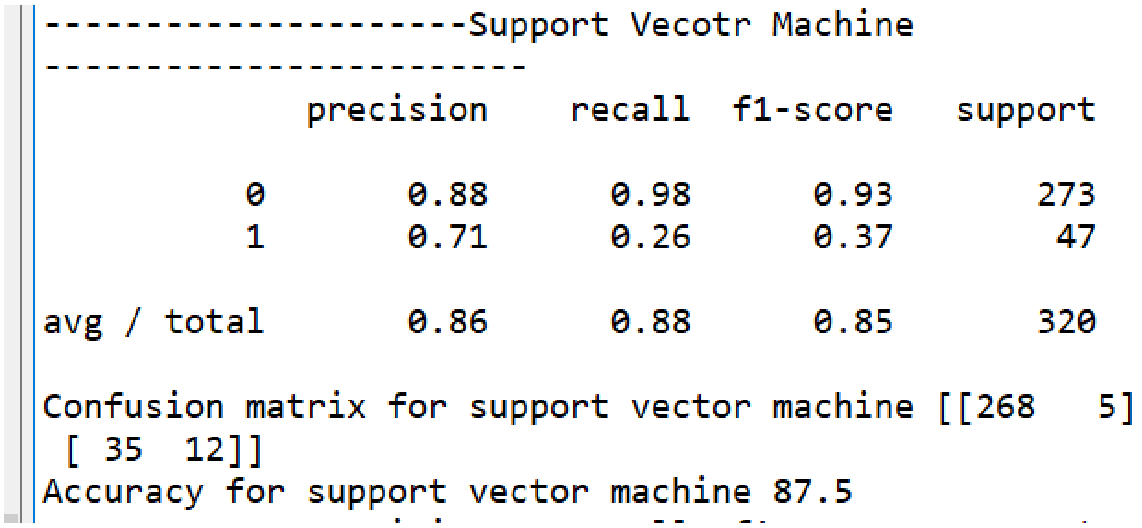
Build a Model

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.



```
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)
#Finding best parameters for our SVC model
```



```
---------------------Support Vecotr Machine
-----------------------
             precision    recall  f1-score   support

          0       0.88      0.98      0.93       273
          1       0.71      0.26      0.37        47

avg / total       0.86      0.88      0.85       320

Confusion matrix for support vector machine [[268   5]
 [ 35  12]]
Accuracy for support vector machine 87.5
```

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()
```