This notebook handles training and performance evaluation of the experimental datasets created in feature_discretization.ipynb. The models that are trained here are all classifiers, specifically including Support Vector Machine, Decision Tree, and Random Forest algorithms. The performance measures accuracy and area under the ROC curve are stored for comparison. Since deployement of unsupervised discretization was utilized to reduce the potential model complexity of large, continuous, and often highly correlated variables, the goal is to determine the efficacy of different unsupervised approaches to discretization in the building of classifier models. To that end, the experimental groups were devised by optimizing bin numbers per variable and per discretization technique experimentally using either inter-variable variance or intervariable-entropy. To further optimize the dicretization process, ensembles of discretization techniques were deployed where for each continuous predictive variable, a distinct sequence of tranformations were created using constant frequency, constant width, and kmeans discretization techniques. For instance the first continuous variable might have been binned (i.e., discretized) using kmeans, with its optimized number of bins for that given continuous variable. The next continuous variable might have been binned using width and so on. Each potential sequence's correlation was compared to that of the continuous data and the sequence with the greatest reduction of correlation was selected. This process was repeated using mutual information as well. Ultimately, 6 experimental datasets were created and were grouped based on whether variance or entropy were used to optimize bin number. In group 1, variance was used for finding bin numbers so that those potential transformations of continuous variables were assessed first with correlation analysis (1a) and then mutual information (1b). (1c) consisted of randomly selected bin numbers developed with randomly selected techniques. This is meant to determine if any improvements might be explained by random chance. The second group followed an identical procedure, but with entropy as the measure used to create bin sizes for potential transformations. (2a) and (2b) transformed variable sequences were also determined using correlation and mutual information, respectively, while the randomly derived dataset was the same one used in group one in order to allow performances between the two groups to be compared relative to it. For point of comparison, the models will also be trained on the the original continuous data offering a baseline for performance.

importing dependencies

In [4]:
```
from sklearn import tree
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold, cross
from sklearn.metrics import accuracy_score, roc_auc_score, make_scorer


import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
from glob import glob
from collections import OrderedDict
import os
from prettytable import PrettyTable
import pickle
```

imports control groups and seperates the class column (test group csvs dont have test column so 'y' defined here is added to each model training)

In [5]:
```
control_df = pd.read_csv("../Dataset/preprocessed_data.csv")

y = control_df['class'] # this is added to df_dict as 'y'
control_df.drop('class', axis=1, inplace= True)
control_df.head()
```

Out[5]:

	alpha	delta	u	g	r	i	z	redshift
0	135.689107	32.494632	23.87882	22.27530	20.39501	19.16573	18.79371	0.634794
1	144.826101	31.274185	24.77759	22.83188	22.58444	21.16812	21.61427	0.779136
2	142.188790	35.582444	25.26307	22.66389	20.60976	19.34857	18.94827	0.644195
3	338.741038	-0.402828	22.13682	23.77656	21.61162	20.50454	19.25010	0.932346
4	345.282593	21.183866	19.43718	17.58028	16.49747	15.97711	15.54461	0.116123

Imports all test group dfs

In [6]:
```
file_list = glob('../Dataset/*.csv')

print(file_list)

# removes file names in filse list that is not a test group
file_list = [f for f in file_list if "TestGroup_" in f]

print(file_list)

# visual aid (i can't keep the syntax right)
# general comprehension: [expression for item in iterable if condition]
```

```
['../Dataset/TestGroup_2c.csv', '../Dataset/TestGroup_2b.csv', '../Dataset/T
estGroup1.csv', '../Dataset/Control_Group.csv', '../Dataset/TestGroup2.csv',
'../Dataset/TestGroup_1c.csv', '../Dataset/TestGroup_2a.csv', '../Dataset/Te
stGroup_1a.csv', '../Dataset/preprocessed_data.csv', '../Dataset/TestGroup_1
b.csv', '../Dataset/star_classification.csv', '../Dataset/test_performanceRe
sults.csv']
['../Dataset/TestGroup_2c.csv', '../Dataset/TestGroup_2b.csv', '../Dataset/T
estGroup_1c.csv', '../Dataset/TestGroup_2a.csv', '../Dataset/TestGroup_1a.cs
v', '../Dataset/TestGroup_1b.csv']
```

Creates dictionary to organize test groups with models and performance measures. Adds subdictionary for each test group with initial df predict and class (y) values

In [7]:
```
df_dict = {}
for file in file_list:
    file_name = os.path.splitext(os.path.basename(file))[0]
    df = pd.read_csv(file)
    predict = df.iloc[:,0:len(df)+1]

    df_dict[f"{file_name}_df"] = {'df' : df,
                                  'x' : predict,
                                  'y' : y} # y defined above as class columr

# orders df_dict objects by key name
df_dict = OrderedDict(sorted(df_dict.items()))
```

adds control group to df_dict

In [8]:
```
df_dict["control_df"] = {'df' : control_df,
                         'x' : control_df,
                         'y' : y}
```

In [ ]:
```
print(df_dict.keys())

# saves key names for iterating through or grab subdictionaries without reme
key_names = list(df_dict.keys())
```

```
odict_keys(['TestGroup_1a_df', 'TestGroup_1b_df', 'TestGroup_1c_df', 'TestGr
oup_2a_df', 'TestGroup_2b_df', 'TestGroup_2c_df', 'control_df'])
```

Creates training/testying groups for each model so that they are constant for all tests. The standard 80%/20% training to testing allocation is used. Because some of the binning techniques have variable bin sizes, stratified "sampling" is used meaning that 20% of each bin was used for testing and 80% of each bin for training to ensure that the overall ditributions of the training/testing groups are consistent with the full set of source data for the model.

In [10]:
```
for group_name, values in df_dict.items():
    X = values['x']
    y = values['y']

    X_train, X_test, y_train, y_test = train_test_split(
        X,y,test_size=0.2, random_state = 33, stratify= y
    )

    values.update(
        {'X_train': X_train,
         'X_test': X_test,
         'y_train': y_train,
         "y_test": y_test}
    )
```

---

decision tree model training

tests purity measures on one test group to determine if there accuracy varies. If accuracy is consistent only gini index used in training on all groups.

In [11]:
```
test_object1 = df_dict[key_names[1]]

criterion = ['gini', 'entropy','log_loss']
max_depth = [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]
score = {
    'gini' : [],
    'entropy' : [],
    'log_loss' : []
}
for criteria in criterion:
    for depth in max_depth:
        X = test_object1['X_train']
        Y =  test_object1['y_train']
        clf = tree.DecisionTreeClassifier(criterion=criteria, max_depth=dept
        score[criteria].append(np.mean(cross_val_score(clf, X, Y, cv=10)))

print(f"gini: {score['gini']}")
print(f"entropy: {score['entropy']}")
print(f"log_loss: {score['log_loss']}")
```

```
gini: [np.float64(0.9362691723908754), np.float64(0.953508593048975), np.flo
at64(0.9600516227958087), np.float64(0.9645244237893023), np.float64(0.96488
22708608724), np.float64(0.9623263897110395), np.float64(0.959668278671325
1), np.float64(0.9566395557039735), np.float64(0.9541220050474826), np.float
64(0.9522945535035587), np.float64(0.9509527135321028), np.float64(0.9502242
855591543), np.float64(0.9502243116865303), np.float64(0.950416001711343), n
p.float64(0.9508888304651733)]
entropy: [np.float64(0.9362691723908754), np.float64(0.9537769473263937), n
p.float64(0.9627864160507), np.float64(0.9653039536435033), np.float64(0.966
057935823), np.float64(0.9637832178965992), np.float64(0.9613167968816978),
np.float64(0.9577257703289355), np.float64(0.955553274127611), np.float64(0.
9539686308193627), np.float64(0.952652380979597), np.float64(0.9524351383811
969), np.float64(0.9521667645082461), np.float64(0.9520261812635689), np.flo
at64(0.9519878605679928)]
log_loss: [np.float64(0.9362691723908754), np.float64(0.9537769473263937), n
p.float64(0.9627736364979844), np.float64(0.9652656166183176), np.float64(0.
9658662392663432), np.float64(0.9637065356814224), np.float64(0.961176239764
3963), np.float64(0.9581858293278159), np.float64(0.9560516750505604), np.fl
oat64(0.9537897448416803), np.float64(0.9531124334466335), np.float64(0.9524
35141647119), np.float64(0.951911173453933), np.float64(0.9521412152005805),
np.float64(0.9517194899609642)]
```

because the scores for all purity measures are comparable for each criterion gini will be used

Creates and fits Decision Tree claissifier for all test groups and appends to

In [12]:
```
for group_name, values in df_dict.items():

    dt_model = tree.DecisionTreeClassifier(criterion = 'gini',random_state =
```

```
    dt_model.fit(values['X_train'], values['y_train'])

    values.update(
        {'dt_model': dt_model}
    )
```

accuracy, AUC, and complexity is evaluated and appended for each test data set. This includes the distribution of AUC scores and their means

In [13]:
```
for group_name, values in df_dict.items():
    # Get the trained decision tree model and test data
    dt_model = values['dt_model']
    X_test = values['X_test']
    y_test = values['y_test']

    # Predict the labels and probabilities for the test set
    y_pred = dt_model.predict(X_test)
    y_proba = dt_model.predict_proba(X_test)

    # Evaluate accuracy and AUC for multi-class
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_proba, multi_class='ovr', average='macro')

    # Model Complexity Measures for Decision Trees
    tree_depth = dt_model.get_depth()  # Depth of the decision tree
    num_nodes = dt_model.tree_.node_count  # Number of nodes in the tree
    num_features_used = len(dt_model.feature_importances_[dt_model.feature_i


    # Create a DataFrame for complexity measures
    complexity_df = pd.DataFrame({
        'tree_depth': [tree_depth],
        'num_nodes': [num_nodes],
        'dt_features_used': [num_features_used]
    })

    # Add performance and complexity data to the values dictionary
    values.update({
        'dt_y_pred': y_pred,
        'dt_y_proba': y_proba,
        'dt_accuracy': acc,
        'dt_auc': auc,
        'dt_complexity_metrics': complexity_df
    })
```

viewer friendly check on whats produced

In [14]:
```
table = PrettyTable()
table.field_names = ["Group", "Tree Depth", "Node Count", "Features Used"]

for group_name, values in df_dict.items():
    dt_model = values['dt_model']
    table.add_row([
        group_name,
```

```
        dt_model.get_depth(),
        dt_model.tree_.node_count,
        len(dt_model.feature_importances_[dt_model.feature_importances_ > 0]
    ])

print(table)
```

```
+----------------+------------+------------+---------------+
|      Group     | Tree Depth | Node Count | Features Used |
+----------------+------------+------------+---------------+
| TestGroup_1a_df |     33     |   17299    |       8       |
| TestGroup_1b_df |     30     |    8585    |       8       |
| TestGroup_1c_df |     29     |   16045    |       8       |
| TestGroup_2a_df |     25     |    9691    |       8       |
| TestGroup_2b_df |     28     |    9247    |       8       |
| TestGroup_2c_df |     30     |   16467    |       8       |
|    control_df   |     33     |    3807    |       8       |
+----------------+------------+------------+---------------+
```

Random Forest model training

tests purity measures for random forrests. also defers to just gini if comparable vaues in each purity measure

In [15]:
```
test_object1 = df_dict[key_names[1]]

criterion = ['gini', 'entropy','log_loss']
estimators = [20,50,100]
max_depth = [2,4,6,8,10,12,14,16,18,20]
score = {
    'gini' : [],
    'entropy' : [],
    'log_loss' : []
}
for criteria in criterion:
    for estimator in estimators:
        for depth in max_depth:
            X = test_object1['X_train']
            Y =  test_object1['y_train']
            rf_model = RandomForestClassifier(n_estimators=estimator, criter
            score[criteria].append(np.mean(cross_val_score(rf_model, X, Y, c

print(f"gini: {score['gini']}")
print(f"entropy: {score['entropy']}")
print(f"log_loss: {score['log_loss']}")
```

```
gini: [np.float64(0.8165384313286832), np.float64(0.9455469676322636), np.fl
oat64(0.9593104136371838), np.float64(0.964946127800426), np.float64(0.96759
14658192687), np.float64(0.9686777131034505), np.float64(0.968409321267929),
np.float64(0.9688566399051577), np.float64(0.9680259738775231), np.float64
(0.9675020203809861), np.float64(0.8437464485139646), np.float64(0.945598107
0716192), np.float64(0.9590803833212631), np.float64(0.9658534646125105), n
p.float64(0.968140962091627), np.float64(0.9691377590386432), np.float64(0.9
692911136712311), np.float64(0.9692527864438111), np.float64(0.9690483397277
365), np.float64(0.9687543969515884), np.float64(0.869893184755656), np.floa
t64(0.9446141174376557), np.float64(0.9603966772509753), np.float64(0.966160
1657128814), np.float64(0.9685371314917346), np.float64(0.969367790987525),
np.float64(0.9696617272318289), np.float64(0.9693038981228297), np.float64
(0.9691505598198518), np.float64(0.9688566252085087)]
entropy: [np.float64(0.8602188363683274), np.float64(0.9395533623075046), n
p.float64(0.9584286293986866), np.float64(0.9648311069271023), np.float64(0.
9678598331603755), np.float64(0.9687799413603709), np.float64(0.968971647714
7936), np.float64(0.9683709956734698), np.float64(0.9682048957803471), np.fl
oat64(0.9677065095540465), np.float64(0.8578803627459546), np.float64(0.9443
458415423642), np.float64(0.9595660014255749), np.float64(0.965700113245844
7), np.float64(0.9681281907037164), np.float64(0.9691888968450378), np.float
64(0.9694316903840642), np.float64(0.9689972019213418), np.float64(0.9691505
516550467), np.float64(0.9687160566604804), np.float64(0.8583404103141081),
np.float64(0.9451380366420116), np.float64(0.9580963446984695), np.float64
(0.9654573001112862), np.float64(0.9684476762556858), np.float64(0.969278347
1822033), np.float64(0.9692400085240565), np.float64(0.9694700274092503), n
p.float64(0.9688438472887541), np.float64(0.9689333041577637)]
log_loss: [np.float64(0.8267109193652027), np.float64(0.9430677409372705), n
p.float64(0.9577896484969818), np.float64(0.9645755632286578), np.float64(0.
9678981620207565), np.float64(0.9684221269480202), np.float64(0.968422117150
2545), np.float64(0.9687160468627145), np.float64(0.9680770920883857), np.fl
oat64(0.9677959321308753), np.float64(0.8416122632104504), np.float64(0.9442
818164078288), np.float64(0.9578663699032225), np.float64(0.965444514026726
7), np.float64(0.9680387289358239), np.float64(0.9691249974484984), np.float
64(0.9691888919461548), np.float64(0.9693166891062723), np.float64(0.9688055
135294901), np.float64(0.9689972051872638), np.float64(0.8400019497554233),
np.float64(0.947387216691474), np.float64(0.9585053018160977), np.float64(0.
965585082574755), np.float64(0.9686904844913613), np.float64(0.9692016649670
263), np.float64(0.9695850466496131), np.float64(0.9692655627306047), np.flo
at64(0.9692016665999874), np.float64(0.968958846933585)]
```

again, the measures are comprable but worth checking since this is not guarenteed. Gini used again for simplicity in training.

In [16]:
```
for group_name, values in df_dict.items():
    rf_model = RandomForestClassifier(
        n_estimators=100,
        criterion='gini',
        max_depth=None,
        random_state=33,
        n_jobs=24  # use up to 24 cores; yay memory update!
    )
    rf_model.fit(values['X_train'], values['y_train'])

    values.update({'rf_model': rf_model})
```

In [17]:
```
df_dict[key_names[3]]['rf_model']
```

Out[17]:
```
RandomForestClassifier
RandomForestClassifier(n_jobs=24, random_state=33)
```

accuracy, AUC, and complexity is evaluated and appended for each test data set. This includes the distribution of AUC scores and their means

In [18]:
```
for group_name, values in df_dict.items():
    # Get the trained random forest model and test data
    rf_model = values['rf_model']
    X_test = values['X_test']
    y_test = values['y_test']

    # Predict the labels and probabilities for the test set
    y_pred = rf_model.predict(X_test)
    y_proba = rf_model.predict_proba(X_test)

    # Evaluate accuracy and AUC for multi-class
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_proba, multi_class='ovr', average='macro')

    # Model Complexity Measures for Random Forests
    tree_depths = [estimator.tree_.max_depth for estimator in rf_model.estim
    num_nodes_list = [estimator.tree_.node_count for estimator in rf_model.e

    avg_depth = np.mean(tree_depths)
    avg_nodes = np.mean(num_nodes_list)
    num_features_used = len([i for i in rf_model.feature_importances_ if i >
    num_estimators = rf_model.n_estimators

    # Create a DataFrame for complexity measures
    complexity_df = pd.DataFrame({
        'avg_tree_depth': [avg_depth],
        'avg_num_nodes': [avg_nodes],
        'rf_features_used': [num_features_used],
        'num_estimators': [num_estimators]
    })

    # Add performance and complexity data to the values dictionary
    values.update({
        'rf_y_pred': y_pred,
        'rf_y_proba': y_proba,
        'rf_accuracy': acc,
        'rf_auc': auc,
        'rf_complexity_metrics': complexity_df
    })
```

In [19]:
```
table = PrettyTable()
table.field_names = ["Group", "Ave Tree Depth", " Ave Node Count", "Features

for group_name, values in df_dict.items():
```

```
    rf_compl_df = values['rf_complexity_metrics'].iloc[0]
    table.add_row([
        group_name,
        rf_compl_df['avg_tree_depth'],
        rf_compl_df['avg_num_nodes'],
        rf_compl_df['rf_features_used'],
        rf_compl_df['num_estimators']
    ])

print(table)
```

```
+-----------------+----------------+-----------------+---------------+------
------+
|      Group      | Ave Tree Depth |  Ave Node Count | Features Used | Estim
ators |
+-----------------+----------------+-----------------+---------------+------
------+
| TestGroup_1a_df |     31.73      |     15270.56    |      8.0      |   10
0.0    |
| TestGroup_1b_df |     29.69      |     8674.02     |      8.0      |   10
0.0    |
| TestGroup_1c_df |     28.32      |     13710.38    |      8.0      |   10
0.0    |
| TestGroup_2a_df |     23.95      |      8291.8     |      8.0      |   10
0.0    |
| TestGroup_2b_df |      27.7      |     8456.86     |      8.0      |   10
0.0    |
| TestGroup_2c_df |     28.17      |     14066.76    |      8.0      |   10
0.0    |
|    control_df   |     30.27      |      4343.3     |      8.0      |   10
0.0    |
+-----------------+----------------+-----------------+---------------+------
------+
```

Support Vector Machine Learning

---

fitting models

In [20]:
```
for group_name, values in df_dict.items():

    svm_model = svm.SVC(probability=True, kernel='rbf', random_state=33)

    svm_model.fit(values['X_train'], values['y_train'])

    values.update({'svm_model': svm_model})
```

evaluating models

In [21]:
```
for group_name, values in df_dict.items():
```

```
    svm_model = values['svm_model']
    X_test = values['X_test']
    y_test = values['y_test']


    # Predict on the test data
    y_pred = svm_model.predict(X_test)
    y_proba = svm_model.predict_proba(X_test)

    # Evaluate on the test set: accuracy and AUC
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_proba, multi_class='ovr', average='macro')
    # model complexity
    num_support_vecs = len(svm_model.support_vectors_)
    # Store the results
    values.update({
        'svm_y_pred': y_pred,
        'svm_y_proba': y_proba,
        'svm_accuracy': acc,
        'svm_auc': auc,
        'svm_complexity_metrics': num_support_vecs
    })
```

store results as pickle (so I don't have to repeat the building the nested dictionary again)

In [22]:
```
with open('../Dataset/Testgroup_results.pkl', 'wb') as f:
    pickle.dump(df_dict, f)
```

tests data dump file and ensures that content is there

In [23]:
```
with open('../Dataset/Testgroup_results.pkl', 'rb') as f:
    df_dict = pickle.load(f)
```

Checks content of pickle

In [24]:
```
for group_name, values in df_dict.items():
    print(f"{group_name}: {list(values.keys())}")
```

```
TestGroup_1a_df: ['df', 'x', 'y', 'X_train', 'X_test', 'y_train', 'y_test',
'dt_model', 'dt_y_pred', 'dt_y_proba', 'dt_accuracy', 'dt_auc', 'dt_complexi
ty_metrics', 'rf_model', 'rf_y_pred', 'rf_y_proba', 'rf_accuracy', 'rf_auc',
'rf_complexity_metrics', 'svm_model', 'svm_y_pred', 'svm_y_proba', 'svm_accu
racy', 'svm_auc', 'svm_complexity_metrics']
TestGroup_1b_df: ['df', 'x', 'y', 'X_train', 'X_test', 'y_train', 'y_test',
'dt_model', 'dt_y_pred', 'dt_y_proba', 'dt_accuracy', 'dt_auc', 'dt_complexi
ty_metrics', 'rf_model', 'rf_y_pred', 'rf_y_proba', 'rf_accuracy', 'rf_auc',
'rf_complexity_metrics', 'svm_model', 'svm_y_pred', 'svm_y_proba', 'svm_accu
racy', 'svm_auc', 'svm_complexity_metrics']
TestGroup_1c_df: ['df', 'x', 'y', 'X_train', 'X_test', 'y_train', 'y_test',
'dt_model', 'dt_y_pred', 'dt_y_proba', 'dt_accuracy', 'dt_auc', 'dt_complexi
ty_metrics', 'rf_model', 'rf_y_pred', 'rf_y_proba', 'rf_accuracy', 'rf_auc',
'rf_complexity_metrics', 'svm_model', 'svm_y_pred', 'svm_y_proba', 'svm_accu
racy', 'svm_auc', 'svm_complexity_metrics']
TestGroup_2a_df: ['df', 'x', 'y', 'X_train', 'X_test', 'y_train', 'y_test',
'dt_model', 'dt_y_pred', 'dt_y_proba', 'dt_accuracy', 'dt_auc', 'dt_complexi
ty_metrics', 'rf_model', 'rf_y_pred', 'rf_y_proba', 'rf_accuracy', 'rf_auc',
'rf_complexity_metrics', 'svm_model', 'svm_y_pred', 'svm_y_proba', 'svm_accu
racy', 'svm_auc', 'svm_complexity_metrics']
TestGroup_2b_df: ['df', 'x', 'y', 'X_train', 'X_test', 'y_train', 'y_test',
'dt_model', 'dt_y_pred', 'dt_y_proba', 'dt_accuracy', 'dt_auc', 'dt_complexi
ty_metrics', 'rf_model', 'rf_y_pred', 'rf_y_proba', 'rf_accuracy', 'rf_auc',
'rf_complexity_metrics', 'svm_model', 'svm_y_pred', 'svm_y_proba', 'svm_accu
racy', 'svm_auc', 'svm_complexity_metrics']
TestGroup_2c_df: ['df', 'x', 'y', 'X_train', 'X_test', 'y_train', 'y_test',
'dt_model', 'dt_y_pred', 'dt_y_proba', 'dt_accuracy', 'dt_auc', 'dt_complexi
ty_metrics', 'rf_model', 'rf_y_pred', 'rf_y_proba', 'rf_accuracy', 'rf_auc',
'rf_complexity_metrics', 'svm_model', 'svm_y_pred', 'svm_y_proba', 'svm_accu
racy', 'svm_auc', 'svm_complexity_metrics']
control_df: ['df', 'x', 'y', 'X_train', 'X_test', 'y_train', 'y_test', 'dt_m
odel', 'dt_y_pred', 'dt_y_proba', 'dt_accuracy', 'dt_auc', 'dt_complexity_me
trics', 'rf_model', 'rf_y_pred', 'rf_y_proba', 'rf_accuracy', 'rf_auc', 'rf_
complexity_metrics', 'svm_model', 'svm_y_pred', 'svm_y_proba', 'svm_accurac
y', 'svm_auc', 'svm_complexity_metrics']
```