a1

March 25, 2022

1 COMPSCI 762 Assignment 1 - Decision Tree Learning

Chase Robertson
crob873

[1]:
```
import os
import pandas as pd
import numpy as np
from pathlib import Path
from sklearn.model_selection import train_test_split

pd.set_option('mode.chained_assignment',None)
PATH_ROOT = Path(os.getcwd())

RANDOM_STATE = 123456
np.random.seed(RANDOM_STATE)
```

2 Task 1 - Data Preprocessing

2.1 Code and Results

2.1.1 Preprocessing - `arrhythmia.csv`

[2]:
```
heart = pd.read_csv(os.path.join(PATH_ROOT, 'arrhythmia.csv'),␣
 ↪skipinitialspace=True, na_values='?')
heart_nulls = heart.isnull().sum()

print('Raw arrhythmia data shape and # of nulls')
print(heart.shape)
print(heart_nulls[heart_nulls > 0])

heart2 = heart.drop('J', axis=1)

# --- separate training and test sets
heart_x = heart2.drop('class', axis=1)
heart_y = heart2.loc[:, 'class']
h_train_x, h_test_x, h_train_y, h_test_y = train_test_split(heart_x, heart_y,␣
 ↪test_size=0.2, random_state=RANDOM_STATE)
```

```
# Interpolate training and testing data separately
for df in [h_train_x, h_test_x]:
    df.interpolate(method='linear', inplace=True)
    df_nulls = df.isnull().sum()
    print('\n'+'Preprocessed data shape and # of nulls:')
    print(df.shape)
    print(df_nulls[df_nulls > 0])
```

```
Raw arrhythmia data shape and # of nulls
(452, 280)
T              8
P             22
QRST           1
J            376
heartrate      1
dtype: int64

Preprocessed data shape and # of nulls:
(361, 278)
Series([], dtype: int64)

Preprocessed data shape and # of nulls:
(91, 278)
Series([], dtype: int64)
```

2.1.2 Preprocessing - BCP.csv

[3]:
```
bcp = pd.read_csv(os.path.join(PATH_ROOT, 'BCP.csv'), dtype=np.int64)
bcp_nulls = bcp.isnull().sum()
print(bcp.shape)
print(bcp_nulls[bcp_nulls > 0])
print(bcp.columns)
```

```
(683, 11)
Series([], dtype: int64)
Index(['Sample code number', 'Clump Thickness', 'Uniformity of Cell Size',
       'Uniformity of Cell Shape', 'Marginal Adhesion',
       'Single Epithelial Cell Size', 'Bare Nuclei', 'Bland Chromatin',
       'Normal Nucleoli', 'Mitoses', 'Class'],
      dtype='object')
```

2.1.3 Preprocessing - website-phishing.csv

[4]:
```
website = pd.read_csv(os.path.join(PATH_ROOT, 'website-phishing.csv'), dtype=np.
 ↪int64)
website_nulls = website.isnull().sum()
print(website.shape)
print(website_nulls[website_nulls > 0])

website.columns = website.columns.str.strip()
print(website.columns)
```

```
(11055, 31)
Series([], dtype: int64)
Index(['having_IP_Address', 'URL_Length', 'Shortining_Service',
       'having_At_Symbol', 'double_slash_redirecting', 'Prefix_Suffix',
       'having_Sub_Domain', 'SSLfinal_State', 'Domain_registeration_length',
       'Favicon', 'port', 'HTTPS_token', 'Request_URL', 'URL_of_Anchor',
       'Links_in_tags', 'SFH', 'Submitting_to_email', 'Abnormal_URL',
       'Redirect', 'on_mouseover', 'RightClick', 'popUpWidnow', 'Iframe',
       'age_of_domain', 'DNSRecord', 'web_traffic', 'Page_Rank',
       'Google_Index', 'Links_pointing_to_page', 'Statistical_report',
       'Class'],
      dtype='object')
```

2.2 Discussion

`arrhythmia`
I opened the csv data source in a text editor to get an idea of the raw data before attempting an import. Each column looked to be numeric, but I noticed some `?` characters likely meant to denote missing values. I imported specifying the `na_values` argument to capture those question marks as missing values, and the `skipinitialspace` argument to eliminate the leading space in each column header. I then retrieved which attributes were missing, and how many values each of those attributes were missing. The `J` attribute was missing from the vast majority of rows, so it seemed best to simply drop that attribute entirely, as it most likely would not provide our models with any information. I judged it best to linear-interpolate the other missing values, as interpolated values would be more informative for the models than values from other imputation methods. This decision could be revisited after models are built, to confirm that interpolated values are indeed more informative than another imputation method. It did require, however, that I separate the data into testing and training sets before imputing, in order not to have information leak between training and testing data. Because the dataset is so small, it would actually be better to impute 5 separate folds of data and use those to cross-validate performance. I haven't done so in the interest of avoiding the refactoring time.

`BCP`
Visually scanning the raw CSV suggested that each attribute's values were integers, so missing values could be found by specifying data type on import and checking for type conversion failures. No nulls were found after conversion to np.int64, so there were no missing values in the BCP dataset.

`website-phishing`
On visual inspection, all values seemed to be integers in the range `-1,0,1`. Converting to `np.int64` and searching for nulls should flush out any missing values, assuming missing values were not represented by an integer in the range `-1,0,1`. No missing values were found in this dataset under that assumption. Some column names in this dataset included leading and trailing whitespace, so a simple strip and replace was conducted to eliminate that whitespace.

3 Task 2 - Model Implementation

3.1 Utility Functions

[5]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_validate
from sklearn.model_selection import GridSearchCV


def Tree(X_train, y_train, depth_range=[], max_depth=None):

    # if given a range of depths, cross-validate which is best
    if depth_range != []:
        temp_model = DecisionTreeClassifier(criterion='entropy',␣
 ↪random_state=RANDOM_STATE)
        parameters = [{'max_depth': depth_range}]
        cv = GridSearchCV(temp_model, parameters)
        cv.fit(X_train, y_train)

        best_depth = cv.best_params_['max_depth']
    else:
        best_depth = max_depth

    model = DecisionTreeClassifier(criterion='entropy', max_depth=best_depth,␣
 ↪random_state=RANDOM_STATE)
    model.fit(X_train, y_train)
    return model


def print_results(name, models, X_train, X_test, y_train, y_test):
    m = models[name]
    train_score = m.score(X_train, y_train) * 100
    test_score = m.score(X_test, y_test) * 100
    print(name, '(depth {:})\ttrain: {:.2f}%\ttest: {:.2f}%'.format(m.tree_.
 ↪max_depth, train_score, test_score))


models = {}
```

3.2 Code and Results

3.2.1 arrhythmia.csv

[6]:
```
models['Arr Stump'] = Tree(h_train_x, h_train_y, max_depth=1)
print_results('Arr Stump', models, h_train_x, h_test_x, h_train_y, h_test_y)

models['Arr Unpruned'] = Tree(h_train_x, h_train_y)
print_results('Arr Unpruned', models, h_train_x, h_test_x, h_train_y, h_test_y)
```

```
models['Arr Pruned'] = Tree(h_train_x, h_train_y, depth_range=range(1, 20))
print_results('Arr Pruned', models, h_train_x, h_test_x, h_train_y, h_test_y)
```

```
Arr Stump (depth 1)     train: 55.96%   test: 60.44%
Arr Unpruned (depth 10) train: 100.00%  test: 72.53%
```

```
/Users/chase/opt/anaconda3/lib/python3.9/site-
packages/sklearn/model_selection/_split.py:666: UserWarning: The least populated
class in y has only 2 members, which is less than n_splits=5.
  warnings.warn(("The least populated class in y has only %d"
```

```
Arr Pruned (depth 5)    train: 81.72%   test: 71.43%
```

3.2.2 BCP.csv

```
[7]: X = bcp.drop('Class', axis=1)
     y = bcp.loc[:, 'Class']
     bcp_X_train, bcp_X_test, bcp_y_train, bcp_y_test = train_test_split(X, y,␣
      ↪test_size=0.2, random_state=RANDOM_STATE)


     # Fit models and display accuracy scores
     models['BCP Stump'] = Tree(bcp_X_train, bcp_y_train, max_depth=1)
     print_results('BCP Stump', models, bcp_X_train, bcp_X_test, bcp_y_train,␣
      ↪bcp_y_test)

     models['BCP Unpruned'] = Tree(bcp_X_train, bcp_y_train)
     print_results('BCP Unpruned', models, bcp_X_train, bcp_X_test, bcp_y_train,␣
      ↪bcp_y_test)

     models['BCP Pruned'] = Tree(bcp_X_train, bcp_y_train, depth_range=range(1,10))
     print_results('BCP Pruned', models, bcp_X_train, bcp_X_test, bcp_y_train,␣
      ↪bcp_y_test)
```

```
BCP Stump (depth 1)     train: 92.12%   test: 94.89%
BCP Unpruned (depth 7)  train: 100.00%  test: 96.35%
BCP Pruned (depth 6)    train: 99.27%   test: 96.35%
```

3.2.3 website-phishing.csv

```
[8]: X = website.drop('Class', axis=1)
     y = website.loc[:, 'Class']
     web_X_train, web_X_test, web_y_train, web_y_test = train_test_split(X, y,␣
      ↪test_size=0.2, random_state=RANDOM_STATE)


     # Fit models and display accuracy scores
     models['Web Stump'] = Tree(web_X_train, web_y_train, max_depth=1)
```

```
print_results('Web Stump', models, web_X_train, web_X_test, web_y_train,␣
 ↪web_y_test)

models['Web Unpruned'] = Tree(web_X_train, web_y_train)
print_results('Web Unpruned', models, web_X_train, web_X_test, web_y_train,␣
 ↪web_y_test)

models['Web Pruned'] = Tree(web_X_train, web_y_train, depth_range=range(10,30))
print_results('Web Pruned', models, web_X_train, web_X_test, web_y_train,␣
 ↪web_y_test)
```

```
Web Stump (depth 1)     train: 88.67%   test: 89.78%
Web Unpruned (depth 24) train: 99.06%   test: 96.43%
Web Pruned (depth 19)   train: 98.69%   test: 96.65%
```

3.3 Discussion

I chose to use pre-pruning for the pruned model, because the decision stumps were already quite accurate, and it was very simple to implement a hard limit on the tree depth. However, pruning in this way did not greatly improve test performance over the unpruned models.

4 Task 3 - Hyperparameter Tuning

4.1 Discussion

I chose to use exhaustive grid search with cross-validation to select my pre-pruning hyperparameter. This method iterates through each combination of hyperparameters in the specified range of options. For each hyperparameter option, the training data is split into 5 different training/validation folds. The model is separately fit and tested on each training/validation set combination (fold), and the performance scores of each fold are averaged out to get a single performance score for the hyperparameter option being used. By repeating this process for each hyperparameter option, the "best" hyperparameter for the data can be selected with minimal bias. Grid search was selected over random search because only one hyperparameter needed tuning, so the two operations would likely be equivalent in runtime and result.

The `arrhythmia` dataset was optimised at a very shallow depth for the number of features included. I attributed this to an overabundance of features in the dataset relative to the number of examples. This assumption is strengthened by the very low test performance of all attempted models. There is simply not enough data to discover many relationships between the hundreds of features and the class.

The `bcp` dataset was optimised near its maximum possible depth: nearly all of the features were at least somewhat informative. However, the high test performance of the decision stump suggests that only one feature contributed the vast majority of information necessary to make an accurate prediction.

The `website` dataset turned out to be similar to `bcp`, though with many more features and many more examples. The decision stump scored quite well, with the unpruned and pre-pruned models using a high number of features to gain some small amount of new information.

5 Task 4 - Comparing Models

5.1 Code and Results

[9]:
```
from mlxtend.evaluate import paired_ttest_5x2cv

for m1 in ['Arr Stump', 'Arr Unpruned']:
    for m2 in ['Arr Unpruned', 'Arr Pruned']:
        if m1 == m2: # do not compare the unpruned model to itself
            continue
        t, p = paired_ttest_5x2cv(estimator1=models[m1], estimator2=models[m2],
                                  X=h_train_x, y=h_train_y,␣
 ↪random_seed=RANDOM_STATE)
        flag = "\t<--Significant difference" if p < 0.05 else ""
        print(m1, 'vs', m2, '\tp value: %.3f' % p, flag)
print()

for m1 in ['BCP Stump', 'BCP Unpruned']:
    for m2 in ['BCP Unpruned', 'BCP Pruned']:
        if m1 == m2: # do not compare the unpruned model to itself
            continue
        t, p = paired_ttest_5x2cv(estimator1=models[m1], estimator2=models[m2],
                                  X=bcp_X_train, y=bcp_y_train,␣
 ↪random_seed=RANDOM_STATE)
        flag = "\t<--Significant difference" if p < 0.05 else ""
        print(m1, 'vs', m2, '\tp value: %.3f' % p, flag)
print()

for m1 in ['Web Stump', 'Web Unpruned']:
    for m2 in ['Web Unpruned', 'Web Pruned']:
        if m1 == m2: # do not compare the unpruned model to itself
            continue
        t, p = paired_ttest_5x2cv(estimator1=models[m1], estimator2=models[m2],
                                  X=web_X_train, y=web_y_train,␣
 ↪random_seed=RANDOM_STATE)
        flag = "\t<--Significant difference" if p < 0.05 else ""
        print(m1, 'vs', m2, '\tp value: %.3f' % p, flag)
```

```
Arr Stump vs Arr Unpruned       p value: 0.199
Arr Stump vs Arr Pruned         p value: 0.231
Arr Unpruned vs Arr Pruned      p value: 0.707

BCP Stump vs BCP Unpruned       p value: 0.012  <--Significant difference
BCP Stump vs BCP Pruned         p value: 0.011  <--Significant difference
BCP Unpruned vs BCP Pruned      p value: 0.384

Web Stump vs Web Unpruned       p value: 0.000  <--Significant difference
Web Stump vs Web Pruned         p value: 0.000  <--Significant difference
Web Unpruned vs Web Pruned      p value: 0.597
```

5.2 Discussion

Significance tests were conducted using a *5x2cv* paired test. For 5 iterations, the data was split into two 50% halves, with one half used to fit the two specified models, and other half used to evaluate each of the models. The models were then fit and tested again, but with the two halves of data swapped. The difference in prediction accuracy between the two models for each of the two train/test phases was computed, and a mean and variance for the set of two differences is computed. When all 5 iterations finished, the very first accuracy difference computed was used, along with the variance of the distribution of mean differences in accuracy, to compute the t statistic. The t statistic given 5 degrees of freedom yielded the p value, illuminating the statistical significance of the differences in prediction accuracy between models. Any reported p values below the significance threshold of 0.05 indicate a statistically significant difference in prediction accuracy.

The differences in prediction accuracy between all three `arrhythmia` models were not statistically significant, because all three models performed so poorly. Likewise, a statistically significant difference between any unpruned vs pruned models trained on any dataset was not found, likely because the pre-pruning method was not able to remove any nodes without negatively affecting validation scores.

There was a statistically significant difference in prediction accuracy between the decision stump model and the other two models, for both the `BCP` and `website` datasets. The decision stump was able to capture the most important feature in each dataset, and perform quite well as a result, but continuing to build the tree with more features made a significant difference on prediction accuracy.

6 Task 5 - Different Pruning Strategy

6.1 Utility Functions

[10]:
```
from sklearn.model_selection import RandomizedSearchCV

def CostPrunedTree(X_train, y_train):
    temp_model = DecisionTreeClassifier(criterion='entropy',␣
↪random_state=RANDOM_STATE)

    path = temp_model.cost_complexity_pruning_path(X_train, y_train)
    alphas = path.ccp_alphas[:-1]
    parameters = [
        {'ccp_alpha': alphas}
    ]
    n_iter = 10 if len(alphas) < 100 else int(len(alphas)**0.5)

    clf = RandomizedSearchCV(temp_model, parameters, n_iter=n_iter,␣
↪random_state=RANDOM_STATE)
    clf.fit(X_train, y_train)
    best_alpha = clf.best_params_['ccp_alpha']
    model = DecisionTreeClassifier(criterion='entropy', ccp_alpha=best_alpha,␣
 ↪random_state=RANDOM_STATE)
    model.fit(X_train, y_train)
    return model
```

6.2 Code and Results

[11]:
```
models['Arr CCPruned'] = CostPrunedTree(h_train_x, h_train_y)
print_results('Arr CCPruned', models, h_train_x, h_test_x, h_train_y, h_test_y)

models['BCP CCPruned'] = CostPrunedTree(bcp_X_train, bcp_y_train)
print_results('BCP CCPruned', models, bcp_X_train, bcp_X_test, bcp_y_train,␣
 ↪bcp_y_test)

models['Web CCPruned'] = CostPrunedTree(web_X_train, web_y_train)
print_results('Web CCPruned', models, web_X_train, web_X_test, web_y_train,␣
 ↪web_y_test)
```

```
/Users/chase/opt/anaconda3/lib/python3.9/site-
packages/sklearn/model_selection/_split.py:666: UserWarning: The least populated
class in y has only 2 members, which is less than n_splits=5.
  warnings.warn(("The least populated class in y has only %d"
```

```
Arr CCPruned (depth 8)  train: 83.10%   test: 72.53%
BCP CCPruned (depth 7)  train: 99.45%   test: 97.08%
Web CCPruned (depth 23) train: 98.88%   test: 96.56%
```

[12]:
```
# Arrhythmia pre-pruning vs cost-complexity pruning
m1, m2 = 'Arr Pruned', 'Arr CCPruned'
t, p = paired_ttest_5x2cv(estimator1=models[m1], estimator2=models[m2],
                          X=h_train_x, y=h_train_y, random_seed=RANDOM_STATE)

flag = "\t<--Significant difference" if p < 0.05 else ""
print(m1, 'vs', m2, '\tp value: %.3f' % p, flag)


# BCP pre-pruning vs cost-complexity pruning
m1, m2 = 'BCP Pruned', 'BCP CCPruned'
t, p = paired_ttest_5x2cv(estimator1=models[m1], estimator2=models[m2],
                          X=bcp_X_train, y=bcp_y_train,␣
 ↪random_seed=RANDOM_STATE)

flag = "\t<--Significant difference" if p < 0.05 else ""
print(m1, 'vs', m2, '\tp value: %.3f' % p, flag)


# website-phishing pre-pruning vs cost-complexity pruning
m1, m2 = 'Web Pruned', 'Web CCPruned'
```

```
t, p = paired_ttest_5x2cv(estimator1=models[m1], estimator2=models[m2],
                          X=web_X_train, y=web_y_train,␣
 ↪random_seed=RANDOM_STATE)

flag = "\t<--Significant difference" if p < 0.05 else ""
print(m1, 'vs', m2, '\tp value: %.3f' % p, flag)
```

```
Arr Pruned vs Arr CCPruned      p value: 0.863
BCP Pruned vs BCP CCPruned      p value: 0.384
Web Pruned vs Web CCPruned      p value: 0.573
```

6.3 Discussion

The new pruning strategy executed was cost complexity pruning, with the complexity parameter (alpha) determined by random search cross-validation over at least ten possible values. Alpha acts like a misclassification threshold: nodes whose decisions cause misclassification at a rate higher than alpha are pruned, in order of severity of misclassification. This is done by calculating the cost complexity of each node, pruning the node with maximum cost complexity, and repeating the process until every node's cost complexity is greater than alpha. Random cross-validation is used to select alpha because exhaustive search would take too long, and a random sample of all possible alphas of the right size should find an alpha that is good enough.

Cost-complexity pruning did not yield significantly better results than simply restricting the depth of the decision tree. This is confirmed with paired t-test cv accuracy comparisons and their resulting high p-values. The insignificant improvement over pre-pruning likely comes from the datasets themselves: they are either too complex for a decision tree to be very accurate (`arrhythmia`), or they are so easily predictable by a decision tree that it does not matter much that the tree is pruned any certain way, or pruned at all (`BCP` and `website`).