Yunke Zhu and 1330327

HW03 Code

You will complete the following notebook, as described in the PDF for Homework 03 (included in the download with the starter code). You will submit:

1. This notebook file, along with your COLLABORATORS.txt file and the two tree images (PDFs generated using `graphviz` within the code), to the Gradescope link for code.
2. A PDF of this notebook and all of its output, once it is completed, to the Gradescope link for the PDF.

Please report any questions to the class Piazza page (https://piazza.com/tufts/spring2020/comp135).

Import required libraries.

In [213]:
```
import os
import numpy as np
import pandas as pd

import warnings

import sklearn.linear_model
import sklearn.tree
import sklearn.metrics

from matplotlib import pyplot as plt
import seaborn as sns
%matplotlib inline
plt.style.use('seaborn') # pretty matplotlib plots

import graphviz
```

Part One: Cancer-Risk Screening

1.1: Compute true/false positives/negatives.

Complete the following code.

In [214]:
```
def calc_TP_TN_FP_FN(ytrue_N, yhat_N):
    ''' Compute counts of four possible outcomes of a binary classifie
r for evaluation.

    Args
    ----
    ytrue_N : 1D array of floats
        Each entry represents the binary value (0 or 1) of 'true' labe
l of one example
        One entry per example in current dataset
    yhat_N : 1D array of floats
        Each entry represents a predicted binary value (either 0 or 1)
.
        One entry per example in current dataset.
        Needs to be same size as ytrue_N.

    Returns
    -------
    TP : float
        Number of true positives
    TN : float
        Number of true negatives
    FP : float
        Number of false positives
    FN : float
        Number of false negatives
    '''
    TP = 0.0
    TN = 0.0
    FP = 0.0
    FN = 0.0
    for i in range(len(ytrue_N)):
        real = ytrue_N[i]
        predict = yhat_N[i]
        if real == 1 and predict == 1:
            TP += 1
        if real == 0 and predict == 0:
            TN += 1
        if real == 1 and predict == 0:
            FN += 1
        if real == 0 and predict == 1:
            FP += 1
    return TP, TN, FP, FN
```

In [215]:
```
all0 = np.zeros(10)
all1 = np.ones(10)
calc_TP_TN_FP_FN(all0, all1)
```

Out[215]:
```
(0.0, 0.0, 10.0, 0.0)
```

In [216]:
```
calc_TP_TN_FP_FN(all1, all0)
```

Out[216]:
```
(0.0, 0.0, 0.0, 10.0)
```

In [217]:
```
calc_TP_TN_FP_FN(all1, all1)
```

Out[217]:
```
(10.0, 0.0, 0.0, 0.0)
```

In [218]:
```
calc_TP_TN_FP_FN(all0, all0)
```

Out[218]:
```
(0.0, 10.0, 0.0, 0.0)
```

Supplied functions for later use

Do not edit the following functions. They are already complete, and will be used in your later code.

In [219]:
```
def calc_perf_metrics_for_threshold(ytrue_N, yproba1_N, thresh):
    ''' Compute performance metrics for a given probabilistic classifier and threshold
    '''
    tp, tn, fp, fn = calc_TP_TN_FP_FN(ytrue_N, yproba1_N >= thresh)
    ## Compute ACC, TPR, TNR, etc.
    acc = (tp + tn) / float(tp + tn + fp + fn + 1e-10)
    tpr = tp / float(tp + fn + 1e-10)
    tnr = tn / float(fp + tn + 1e-10)
    ppv = tp / float(tp + fp + 1e-10)
    npv = tn / float(tn + fn + 1e-10)

    return acc, tpr, tnr, ppv, npv

def print_perf_metrics_for_threshold(ytrue_N, yproba1_N, thresh):
    ''' Pretty print perf. metrics for a given probabilistic classifier and threshold
    '''
    acc, tpr, tnr, ppv, npv = calc_perf_metrics_for_threshold(ytrue_N, yproba1_N, thresh)

    ## Pretty print the results
    print("%.3f ACC" % acc)
    print("%.3f TPR" % tpr)
    print("%.3f TNR" % tnr)
    print("%.3f PPV" % ppv)
    print("%.3f NPV" % npv)
```

In [220]:
```
def calc_confusion_matrix_for_threshold(ytrue_N, yproba1_N, thresh):
    ''' Compute the confusion matrix for a given probabilistic classifier and threshold

    Args
    ----
    ytrue_N : 1D array of floats
        Each entry represents the binary value (0 or 1) of 'true' label of one example
        One entry per example in current dataset
    yproba1_N : 1D array of floats
        Each entry represents a probability (between 0 and 1) that correct label is positive (1)
        One entry per example in current dataset
        Needs to be same size as ytrue_N
    thresh : float
        Scalar threshold for converting probabilities into hard decisions
        Calls an example "positive" if yproba1 >= thresh

    Returns
    -------
    cm_df : Pandas DataFrame
        Can be printed like print(cm_df) to easily display results
    '''
    cm = sklearn.metrics.confusion_matrix(ytrue_N, yproba1_N >= thresh)
    cm_df = pd.DataFrame(data=cm, columns=[0, 1], index=[0, 1])
    cm_df.columns.name = 'Predicted'
    cm_df.index.name = 'True'
    return cm_df
```

In [221]:
```
def compute_perf_metrics_across_thresholds(ytrue_N, yproba1_N, thresh_grid=None):
    ''' Compute common binary classifier performance metrics across many thresholds

    If no array of thresholds is provided, will use all 'unique' values
    in the yproba1_N array to define all possible thresholds with different performance.

    Args
    ----
    ytrue_N : 1D array of floats
        Each entry represents the binary value (0 or 1) of 'true' label of one example
        One entry per example in current dataset
    yproba1_N : 1D array of floats
```

```
        Each entry represents a probability (between 0 and 1) that cor
rect label is positive (1)
        One entry per example in current dataset

    Returns
    -------
    thresh_grid : 1D array of floats
        One entry for each possible threshold
    perf_dict : dict, with key, value pairs:
        * 'acc' : 1D array of accuracy values (one per threshold)
        * 'ppv' : 1D array of positive predictive values (one per thre
shold)
        * 'npv' : 1D array of negative predictive values (one per thre
shold)
        * 'tpr' : 1D array of true positive rates (one per threshold)
        * 'tnr' : 1D array of true negative rates (one per threshold)
    '''
    if thresh_grid is None:
        bin_edges = np.linspace(0, 1.001, 21)
        thresh_grid = np.sort(np.hstack([bin_edges, np.unique(yproba1_
N)]))
    tpr_grid = np.zeros_like(thresh_grid)
    tnr_grid = np.zeros_like(thresh_grid)
    ppv_grid = np.zeros_like(thresh_grid)
    npv_grid = np.zeros_like(thresh_grid)
    acc_grid = np.zeros_like(thresh_grid)
    for tt, thresh in enumerate(thresh_grid):
        # Apply specific threshold to convert probas into hard binary
values (0 or 1)
        # Then count number of true positives, true negatives, etc.
        # Then compute metrics like accuracy and true positive rate
        acc, tpr, tnr, ppv, npv = calc_perf_metrics_for_threshold(ytru
e_N, yproba1_N, thresh)
        acc_grid[tt] = acc
        tpr_grid[tt] = tpr
        tnr_grid[tt] = tnr
        ppv_grid[tt] = ppv
        npv_grid[tt] = npv
    return thresh_grid, dict(
        acc=acc_grid,
        tpr=tpr_grid,
        tnr=tnr_grid,
        ppv=ppv_grid,
        npv=npv_grid)

def make_plot_perf_vs_threshold(ytrue_N, yproba1_N, bin_edges=np.linsp
ace(0, 1, 21)):
    ''' Make pretty plot of binary classifier performance as threshold
increases
```

```
    Produces a plot with 3 rows:
    * top row: hist of predicted probabilities for negative examples (
shaded red)
    * middle row: hist of predicted probabilities for positive example
s (shaded blue)
    * bottom row: line plots of metrics that require hard decisions (A
CC, TPR, TNR, etc.)
    '''
    fig, axes = plt.subplots(nrows=3, ncols=1, figsize=(12, 8))
    sns.distplot(
        yproba1_N[ytrue_N == 0],
        color='r', bins=bin_edges, kde=False, rug=True, ax=axes[0]);
    sns.distplot(
        yproba1_N[ytrue_N == 1],
        color='b', bins=bin_edges, kde=False, rug=True, ax=axes[1]);

    thresh_grid, perf_grid = compute_perf_metrics_across_thresholds(yt
rue_N, yproba1_N)
    axes[2].plot(thresh_grid, perf_grid['acc'], 'k-', label='accuracy'
)
    axes[2].plot(thresh_grid, perf_grid['tpr'], 'b-', label='TPR (reca
ll/sensitivity)')
    axes[2].plot(thresh_grid, perf_grid['tnr'], 'g-', label='TNR (spec
ificity)')
    axes[2].plot(thresh_grid, perf_grid['ppv'], 'c-', label='PPV (prec
ision)')
    axes[2].plot(thresh_grid, perf_grid['npv'], 'm-', label='NPV')

    axes[2].legend()
    axes[2].set_ylim([0, 1])
```

Load the dataset.

The following should ***not*** be modified. After it runs, the various arrays it creates will contain the 2- or 3-feature input datasets.

In [222]:
```
# Load 3 feature version of x arrays
x_tr_M3 = np.loadtxt('./data_cancer/x_train.csv', delimiter=',', skipr
ows=1)
x_va_N3 = np.loadtxt('./data_cancer/x_valid.csv', delimiter=',', skipr
ows=1)
x_te_N3 = np.loadtxt('./data_cancer/x_test.csv', delimiter=',', skipro
ws=1)

# 2 feature version of x arrays
x_tr_M2 = x_tr_M3[:, :2].copy()
x_va_N2 = x_va_N3[:, :2].copy()
x_te_N2 = x_te_N3[:, :2].copy()
```

In [223]:
```
y_tr_M = np.loadtxt('./data_cancer/y_train.csv', delimiter=',', skipro
ws=1)
y_va_N = np.loadtxt('./data_cancer/y_valid.csv', delimiter=',', skipro
ws=1)
y_te_N = np.loadtxt('./data_cancer/y_test.csv', delimiter=',', skiprow
s=1)
```

1.2: Compute the fraction of patients with cancer.

Complete the following code. Your solution needs to ***compute*** these values from the training, validation, and testing sets (i.e., don't simply hand-count and print the values).

In [224]:
```
def computeFrac(data):
    return sum(data)/len(data)

train_frac = computeFrac(y_tr_M)
valid_frac = computeFrac(y_va_N)
test_frac = computeFrac(y_te_N)
```

In [225]:
```
print("Fraction with cancer in TRAIN: %.3f" % train_frac) #TODO: modif
y what is printed here.
print("Fraction with cancer in VALID: %.3f" % valid_frac)
print("Fraction with cancer in TEST : %.3f" % test_frac)
```

```
Fraction with cancer in TRAIN: 0.141
Fraction with cancer in VALID: 0.139
Fraction with cancer in TEST : 0.139
```

1.3: The predict-0-always baseline

(a) Compute the accuracy of the always-0 classifier.

Complete the code to compute and print the accuracy of the always-0 classifier on validation and test outputs.

In [226]:
```
def computeAccuracy(data=None,baseline=None):
    TP, TN, FP, FN = calc_TP_TN_FP_FN(data,baseline)
    return float((TP + TN)/(TP+TN+FP+FN))
```

In [227]:
```
valid_all_zero = np.zeros(len(y_va_N))
valid_accuracy = computeAccuracy(y_va_N,valid_all_zero)

test_all_zero = np.zeros(len(y_te_N))
test_accuracy = computeAccuracy(y_te_N,test_all_zero)
```

In [228]:
```
print("Always-0: accuracy on VALID: %.3f" % valid_accuracy) # TODO edi
t values!
print("Always-0: accuracy on TEST : %.3f" % test_accuracy)
```

```
Always-0: accuracy on VALID: 0.861
Always-0: accuracy on TEST : 0.861
```

(b) Print a confusion matrix for the always-0 classifier.

Add code below to generate a confusion matrix for the always-0 classifier on the validation set.

In [229]:
```
# TODO call print(calc_confusion_matrix_for_threshold(...))
calc_confusion_matrix_for_threshold(y_va_N,valid_all_zero,0.5)
#print_perf_metrics_for_threshold(valid_all_zero,y_va_N,0)
```

Out[229]:

Predicted	0	1
True		
0	155	0
1	25	0

(c) Reflect on the accuracy of the always-0 classifier.

Answer: The always-0 classifier has a relatively acceptable accuracy at 86% compared to some problem like flipping coin with only 50% accuracy. But we can only treat always-0 classifier as a baseline. In reality, when we do a investigation on cancer, it is possibly that most people don't have a cancer. So compared to the it's original low cancer occurance rate, 86% accuracy is not that reliable. What's more important is, we need to focus on the group that carry cancer instead of who doesn't, and for this always-0 classifier, we can get nothing useful about the predicted true cases, so we can not calculate the Positive Predictive Value via True Positive Value and False Positive Value. And in this case, Positive Predictive Value is all we care about. So the always-0 classifier can be only treated as a baseline instead of a reliable classifier.

(d) Analyze the various costs of using the always-0 classifier.

Answer: The weakness of this classifier is that it pays more attention on the people who doesn't have cancer but ignore all the patients who have cancer. In reality, hospitals care more on the people who has cancer. From another aspects, we need False Negative as small as possible and False Positive weighs not that much. This means that we will focus more on people who diagnosed with cancer even it is not accurate, but we don't want those who have cancer but not detected because a person's life is invaluable. But this classifier only focus on the average accuracy and in reality this classifier is worse than a always-1 classifier even always-1 classifier only got 14% accuracy but it will not ignore any single life.

1.4: Logistic Regression

(a) Create a set of `LogisticRegression` models.

Each model will use a different control parameter, `C`, and each will be fit to 2-feature data. Probabilistic predictions will be made on both training set and validation set inputs, and logistic-loss for each will be recorded.

In [230]:
```
from sklearn.linear_model import LogisticRegression
tr_loss_list_2 = list()
va_loss_list_2 = list()

# TODO fit, predict_proba, and evaluate logistic loss
# Record the best model here
def loss_detection(C,x_tr,y_tr,x_va,y_va):
    LRM = LogisticRegression(C=C,solver='liblinear').fit(x_tr, y_tr)
    pred_tr = LRM.predict_proba(x_tr)
    pred_va = LRM.predict_proba(x_va)
    return sklearn.metrics.log_loss(y_tr,pred_tr),sklearn.metrics.log_
loss(y_va,pred_va)
```

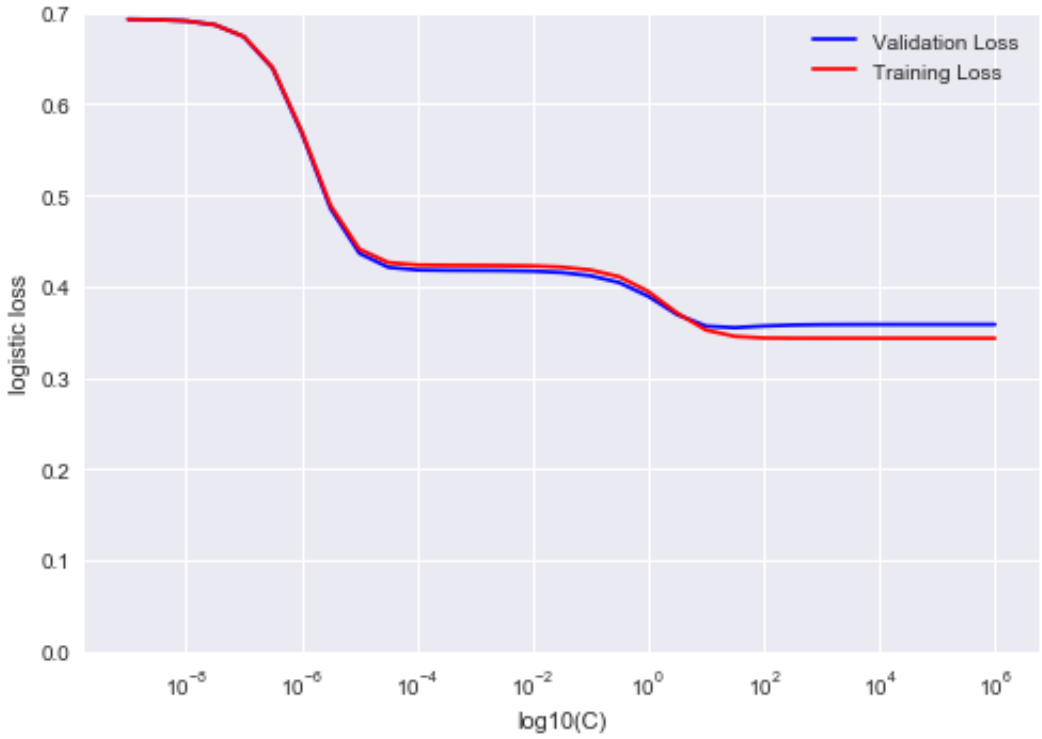
In [231]:
```
C_grid = np.logspace(-9, 6, 31)
for C in C_grid:
    tr_loss,va_loss = loss_detection(C,x_tr_M2,y_tr_M,x_va_N2,y_va_N)
    tr_loss_list_2.append(tr_loss)
    va_loss_list_2.append(va_loss)
```

Plot logistic loss (y-axis) vs. C (x-axis) on the training set and validation set.

The best values for `C` and the loss should be printed.

In [232]:
```
# TODO make plot
plt.xscale('log')
plt.xlabel('log10(C)')
plt.ylabel('logistic loss')
plt.plot(C_grid,va_loss_list_2, color='blue',label = "Validation Loss"
)
plt.plot(C_grid,tr_loss_list_2, color='red',label = "Training Loss")
plt.ylim([0.0, 0.7]);

# TODO add legend
# plt.legend(...);
plt.legend()
plt.show()
print("Best C-value for LR with 2-feature data: %.3f" % C_grid[np.argm
in(va_loss_list_2)]) # TODO
print("Validation set log-loss at best C-value: %.4f" % min(va_loss_li
st_2))
```




```
Best C-value for LR with 2-feature data: 31.623
Validation set log-loss at best C-value: 0.3549
```

(b) Plot the performance of the predictions made by the best classifier from step (a) on the validation set.

In [233]:
```
best_C_2 = C_grid[np.argmin(va_loss_list_2)]
LRM = LogisticRegression(C=best_C_2,solver='liblinear').fit(x_tr_M2, y
_tr_M)
yproba = LRM.predict_proba(x_va_N2)[:,1]
```

In [234]:
```
# TODO call make_plot_perf_vs_threshold(...)
make_plot_perf_vs_threshold(y_va_N, yproba)
```

(c) Model fitting with 3-feature data

Repeat the model generation from **1.4 (a)**, using the full 3-feature data.

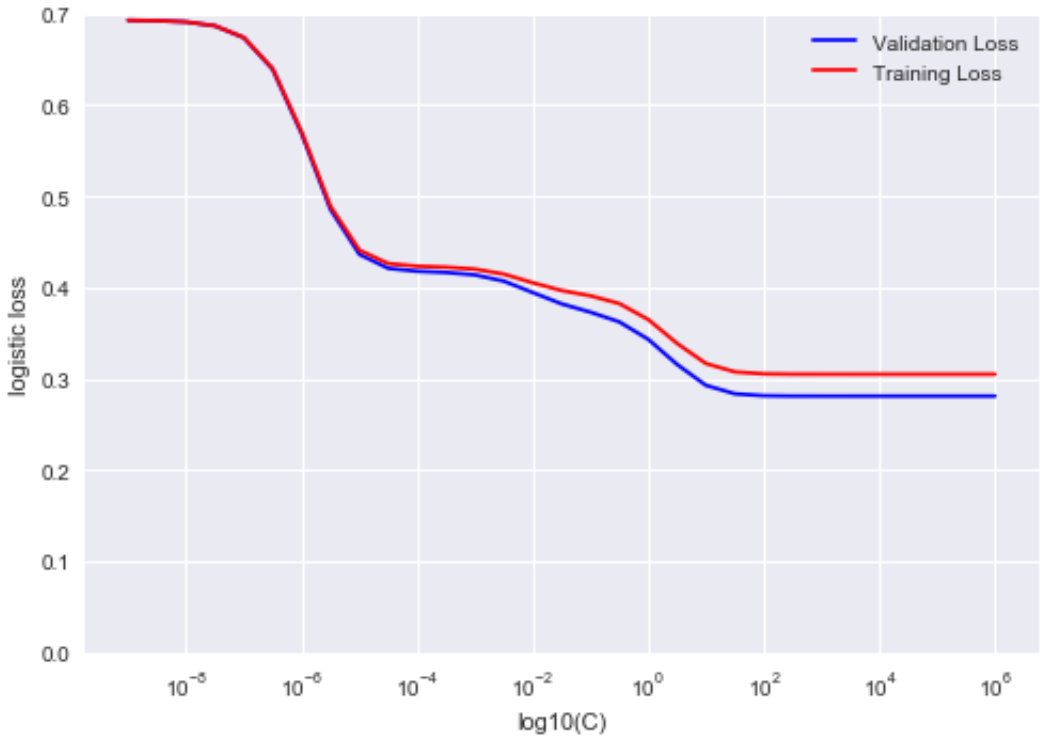
In [235]:
```
# TODO like 1.4 (a), but with 3 features
tr_loss_list_3 = list()
va_loss_list_3 = list()
C_grid = np.logspace(-9, 6, 31)
for C in C_grid:
    tr_loss,va_loss = loss_detection(C,x_tr_M3,y_tr_M,x_va_N3,y_va_N)
    tr_loss_list_3.append(tr_loss)
    va_loss_list_3.append(va_loss)
```

Plot logistic loss (y-axis) vs. C (x-axis) for the 3-feature classifiers on the training set and validation set.

Again, the best values for `C` and the loss should be printed.

In [236]:
```
# TODO make plot
plt.xscale('log')
plt.xlabel('log10(C)')
plt.ylabel('logistic loss')
plt.plot(C_grid,va_loss_list_3, color='blue',label = "Validation Loss"
)
plt.plot(C_grid,tr_loss_list_3, color='red',label = "Training Loss")
plt.ylim([0.0, 0.7]);

# TODO add legend
# plt.legend(...);
plt.legend()
plt.show()
print("Best C-value for LR with 2-feature data: %.3f" % C_grid[np.argm
in(va_loss_list_3)]) # TODO
print("Validation set log-loss at best C-value: %.4f" % min(va_loss_li
st_3))
```

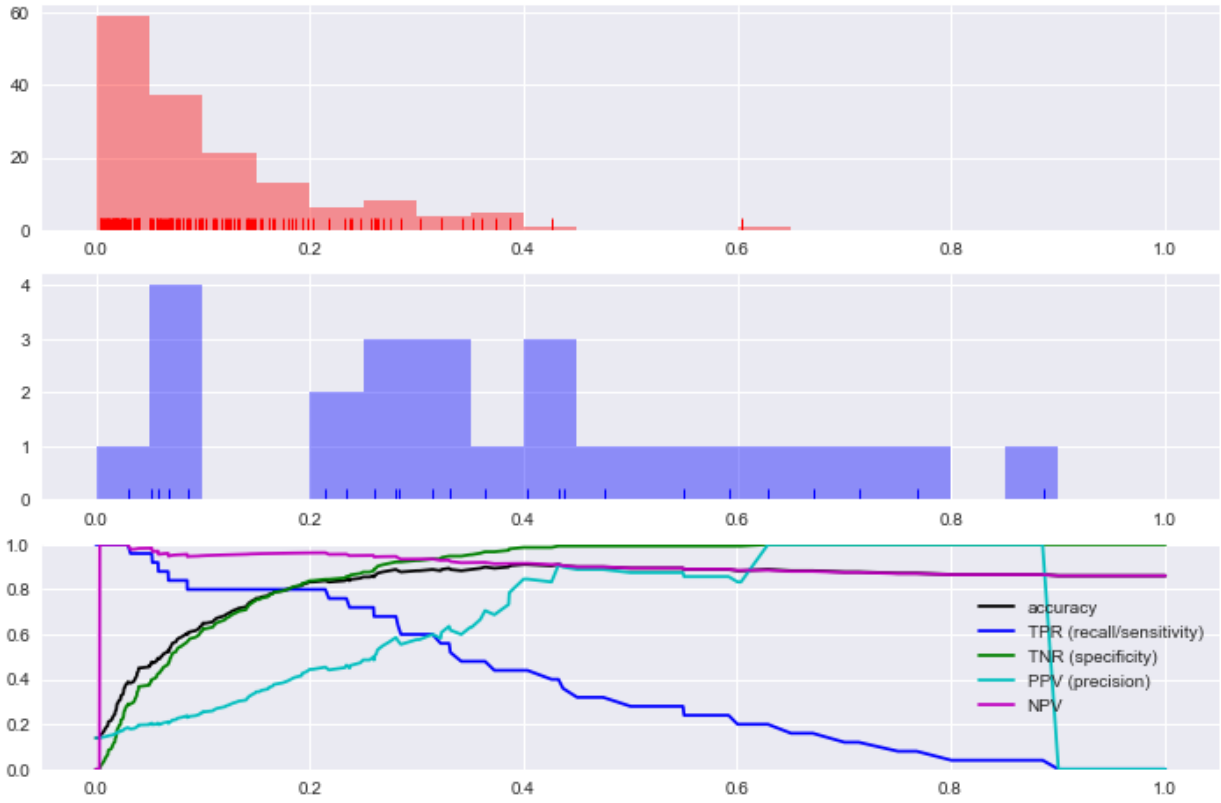



```
Best C-value for LR with 2-feature data: 1000000.000
Validation set log-loss at best C-value: 0.2810
```

Plot the performance of the predictions made by the best 3-valued classifier on the validation set.

In [237]:
```
# TODO call make_plot_perf_vs_threshold(...)
best_C_3 = C_grid[np.argmin(va_loss_list_3)]
LRM = LogisticRegression(C=best_C_3,solver='liblinear').fit(x_tr_M3, y
_tr_M)
yproba = LRM.predict_proba(x_va_N3)[:,1]

make_plot_perf_vs_threshold(y_va_N, yproba)
```



1.5: ROC Curves

These curves allow us to compare model performance in terms of trade-offs between false positive and true positive results.

(a) Plot ROC curves on the validation set.

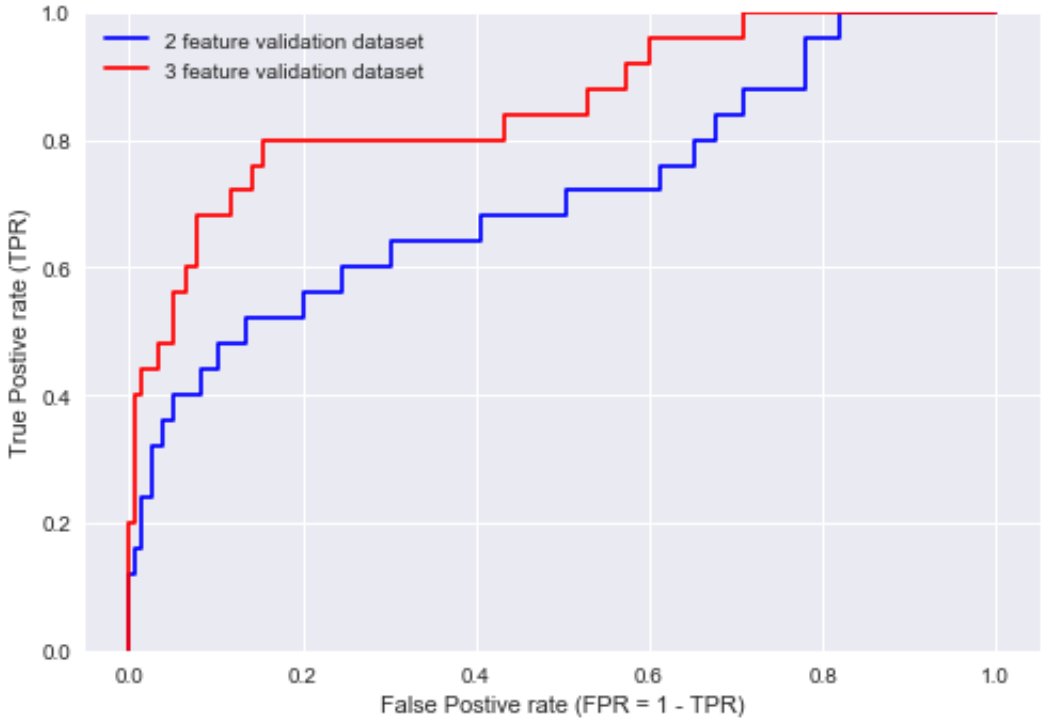
There should be two curves in the plot, one for each of the best two classifiers from prior steps.

In [238]:
```
LRM = LogisticRegression(C=best_C_2,solver='liblinear').fit(x_tr_M2, y
_tr_M)
yproba_va_2 = LRM.predict_proba(x_va_N2)[:,1]
LRM = LogisticRegression(C=best_C_3,solver='liblinear').fit(x_tr_M3, y
_tr_M)
yproba_va_3 = LRM.predict_proba(x_va_N3)[:,1]

fpr_va_2, tpr_va_2, thresholds_va_2 = sklearn.metrics.roc_curve(y_va_N
, yproba_va_2)
fpr_va_3, tpr_va_3, thresholds_va_3 = sklearn.metrics.roc_curve(y_va_N
, yproba_va_3)
```

In [239]:
```
# TODO something like: fpr, tpr, thr = sklearn.metrics.roc_curve(...)

plt.ylim([0, 1]);
plt.plot(fpr_va_2,tpr_va_2, color='blue',label = "2 feature validation
dataset")
plt.plot(fpr_va_3,tpr_va_3, color='red',label = "3 feature validation
dataset")
plt.xlabel("False Postive rate (FPR = 1 - TPR)");
plt.ylabel("True Postive rate (TPR)");
plt.legend()
plt.show()
```



(b) Plot ROC curves on the test set.

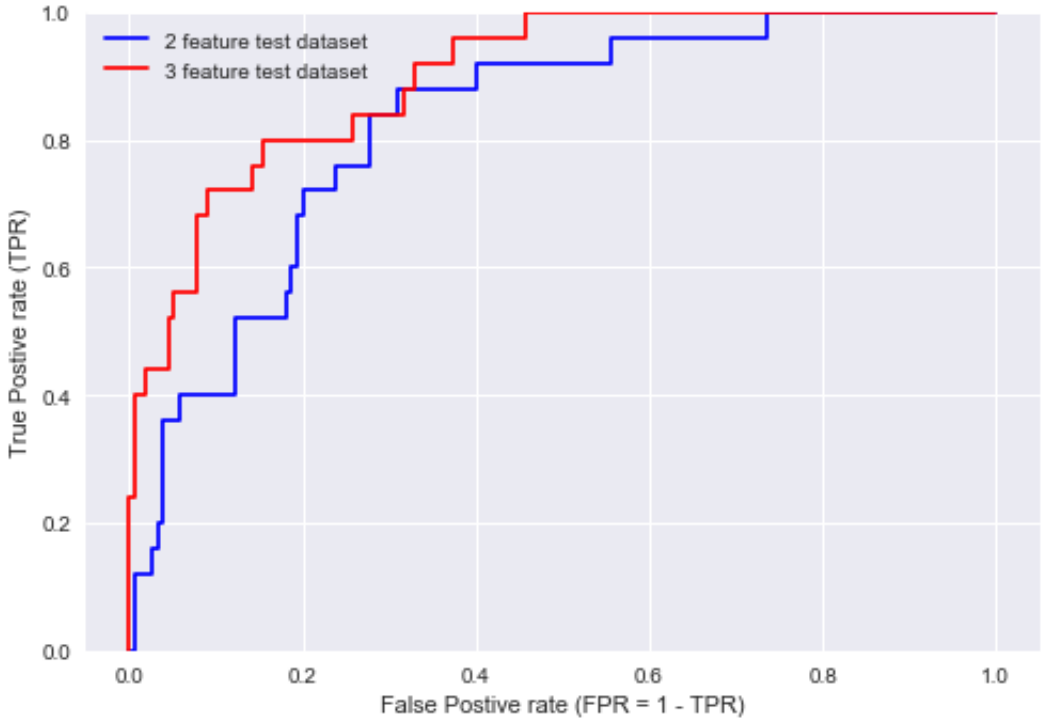
There should be two curves in the plot, one for each of the best two classifiers from prior steps.

In [240]:
```
LRM = LogisticRegression(C=best_C_2,solver='liblinear').fit(x_tr_M2, y
_tr_M)
yproba_te_2 = LRM.predict_proba(x_te_N2)[:,1]
LRM = LogisticRegression(C=best_C_3,solver='liblinear').fit(x_tr_M3, y
_tr_M)
yproba_te_3 = LRM.predict_proba(x_te_N3)[:,1]

fpr_te_2, tpr_te_2, thresholds_te_2 = sklearn.metrics.roc_curve(y_te_N
, yproba_te_2)
fpr_te_3, tpr_te_3, thresholds_te_3 = sklearn.metrics.roc_curve(y_te_N
, yproba_te_3)
```

In [241]:
```
# TODO something like: fpr, tpr, thr = sklearn.metrics.roc_curve(...)

plt.ylim([0, 1]);
plt.plot(fpr_te_2,tpr_te_2, color='blue',label = "2 feature test datas
et")
plt.plot(fpr_te_3,tpr_te_3, color='red',label = "3 feature test datase
t")
plt.xlabel("False Postive rate (FPR = 1 - TPR)");
plt.ylabel("True Postive rate (TPR)");
plt.legend()
plt.show()
```



(c) Analyze the results shown in both the above plots, to compare classifier performance.

Answer: For most situation 3 features classifier performs better than 2 features classifier in both testing dataset and validation dataset. The performance of a classifier in AUC depends on the area of its line intersects with line y = 0 and line x = 1, as we can see from two plots, red line's area stands for 3 features has a bigger area than blue line's area. So 3 features classifier performs better than 2 features classifer.

1.6: Selecting a decision threshold

(a) Using default 0.5 threshold.

Generate a confusion matrix for the best 3-feature logistic model on the test set, using threshold 0.5.

In [258]:
```
best_thr = 0.5

LRM = LogisticRegression(C=best_C_3,solver='liblinear').fit(x_tr_M3, y
_tr_M)
ypred_te_3 = LRM.predict(x_te_N3)
print("ON THE VALIDATION SET:")
print("Chosen best thr = %.4f" % best_thr)
print("")
print("ON THE TEST SET:")

print(calc_confusion_matrix_for_threshold(y_te_N,ypred_te_3,best_thr))
# TODO: print(calc_confusion_matrix_for_threshold(...))
print("")
print_perf_metrics_for_threshold(y_te_N, ypred_te_3, best_thr)
# TODO: print(print_perf_metrics_for_threshold(...))
```

```
ON THE VALIDATION SET:
Chosen best thr = 0.5000

ON THE TEST SET:
Predicted    0   1
True
0          152   3
1           15  10

0.900 ACC
0.400 TPR
0.981 TNR
0.769 PPV
0.910 NPV
```

(b) Pick a threshold to maximize TPR, while ensuring PPV >= 0.98.

After finding the best threshold on the validation set, plot its confusion matrix and print its various performance metrics, for the test set.

In [259]:
```
LRM = LogisticRegression(C=best_C_3,solver='liblinear').fit(x_tr_M3, y
_tr_M)
yproba_va_3 = LRM.predict_proba(x_va_N3)[:,1]
thresh_grid, perf_grid = compute_perf_metrics_across_thresholds(y_va_N
, yproba_va_3, thresh_grid=None)
yproba_te_3 = LRM.predict_proba(x_te_N3)[:,1]


def bestSatisfiedTPR(perf_grid,ppv):
    tmp = perf_grid['tpr'].copy()
    if (max(perf_grid['ppv']) < ppv):
        print ("no threshold satisfied!")
        return

    while True:
        index = np.argmax(tmp)
        if perf_grid['ppv'][index] >= ppv:
            return index
        else:
            tmp[index] = 0

best_thr_index = bestSatisfiedTPR(perf_grid, 0.98)
best_thr = thresh_grid[best_thr_index]
```

In [260]:
```
# TODO thresh_grid, perf_grid = compute_perf_metrics_across_thresholds
(...)

# TODO Find threshold that makes TPR as large as possible, while satis
fying PPV >= 0.98

print("ON THE VALIDATION SET:")
print("Chosen best thr = %.4f" % best_thr)

print("")
print("ON THE TEST SET:")
print(calc_confusion_matrix_for_threshold(y_te_N,yproba_te_3,best_thr)
)
# TODO: print(calc_confusion_matrix_for_threshold(...))
print("")
print_perf_metrics_for_threshold(y_te_N, yproba_te_3, best_thr)
# TODO: print(print_perf_metrics_for_threshold(...))
```

```
ON THE VALIDATION SET:
Chosen best thr = 0.6290

ON THE TEST SET:
Predicted    0  1
True
0          155  0
1           20  5

0.889 ACC
0.200 TPR
1.000 TNR
1.000 PPV
0.886 NPV
```

(c) Pick a threshold to maximize PPV, while ensuring TPR >= 0.98.

After finding the best threshold on the validation set, plot its confusion matrix and print its various performance metrics, for the test set.

In [261]:
```
LRM = LogisticRegression(C=best_C_3,solver='liblinear').fit(x_tr_M3, y
_tr_M)
yproba_va_3 = LRM.predict_proba(x_va_N3)[:,1]
thresh_grid, perf_grid = compute_perf_metrics_across_thresholds(y_va_N
, yproba_va_3)
yproba_te_3 = LRM.predict_proba(x_te_N3)[:,1]


def bestSatisfiedPPV(perf_grid,tpr):
    if (max(perf_grid['tpr']) < tpr):
        print ("no threshold satisfied!")
        return
    tmp = perf_grid['ppv'].copy()
    while True:
        index = np.argmax(tmp)
        if perf_grid['tpr'][index] >= tpr:
            return index
        else:
            tmp[index] = 0


best_thr_index = bestSatisfiedPPV(perf_grid, 0.98)
best_thr = thresh_grid[best_thr_index]
```

In [262]:
```
# TODO thresh_grid, perf_grid = compute_perf_metrics_across_thresholds
(...)

# TODO Find threshold that makes PPV as large as possible, while satis
fying TPR >= 0.98


print("ON THE VALIDATION SET:")
print("Chosen best thr = %.4f" % best_thr) # TODO
print("")
print("ON THE TEST SET:")
# TODO: print(calc_confusion_matrix_for_threshold(...))
print(calc_confusion_matrix_for_threshold(y_te_N,yproba_te_3,best_thr)
)
print("")
# TODO: print(print_perf_metrics_for_threshold(...))
print_perf_metrics_for_threshold(y_te_N, yproba_te_3, best_thr)
```

```
ON THE VALIDATION SET:
Chosen best thr = 0.0300

ON THE TEST SET:
Predicted   0   1
True
0          57  98
1           0  25

0.456 ACC
1.000 TPR
0.368 TNR
0.203 PPV
1.000 NPV
```

(d) Compare the confusion matrices from (a)–(c) to analyze the different thresholds.

Answer: From previous discussion, we conclude that we will pay more attention on reducint the False Negative Value since we cannot ignore those who have cancer. Among the TPR,TNR,PPV,NPV, we will focus more on TPR. Since $TPR = TP/(TP + FP)$, with greater TPR, we can guarantee less amount of False Negative Values. So let's analyze three different model with different thresholds.\ Thresholds = 0.5: This model has a relatively good accuracy as 90%, but the TPR with 40% is too small, which means among all the people have cancer in real life, most of them are not detected. So if the hospital pays more attention on the diagnosis's correctness of cancer, then this model is not suitable for use in real life.\ Thresholds = 0.629: Compared to the first model, this model has a similar accuracy but with less TPR, since the first model is no longer useful in real life, this model is much worse. The high PPV stands for the accuracy of those detected as cancer are correctly labeled. This is important as well in real life, but as an exchange, this model ignored the one who have cancer but not detected, which is not reliable. \ Thresholds = 0.03 This model has a perfect TPR as we required, which means that we won't ignore any single person who has a cancer, which is great. But the problem is the accuracy is not ideal. The 45.6% accuracy may seems not very reliable in detecting cancer, an always-0 classifier has 86.1% accuracy. But the advantage of this model is that it will not neglect any patients who could have cancer. And compared to a always-1 classifier with 13.9% accuracy, this one works better.

(e) How many biopsies can be avoided using the best threshold for the classifier?

Answer: Suppose we choose thresholds = 0.03 as our best thresholds, and we can guarantee that all the patients have cancer in real life won't be ignored, than we can avoid biopsies on all the patients with prediction value of False, which is 57. If we choose thresholds = 0.5 or 0.629, we can avoid 167 or 175 biopsies, but some patients with cancer can also be ignored as well.

Part Two: Decision Trees

You should start by computing the two heuristic values for the toy data described in the assignment handout. You should then load the two versions of the abalone data, compute the two heuristic values on features (for the simplified data), and then build decision trees for each set of data.

2.1 Compute both heuristics for toy data.

(a) Compute the counting-based heuristic, and order the features by it.

In [247]:
```
# TODO
attr_A = np.array([[1,1],[0,0,0,0,1,1]])
attr_B = np.array([[1,1,1,0],[1,0,0,0]])


def countingBased(feature, attr, labels=2):
    totalVal = 0.0
    mojority = 0.0

    for l in attr:
        d = dict() #dictionary is used to store the number of labels i
n each attribute, decide the majority
        for i in range(labels):
            d[i] = 0
        if labels == 2:
            mojority += max(sum(l),len(l)-sum(l))
        else:
            for label in l:
                d[label] += 1
            mojority += max(d[_] for _ in d)

        totalVal += len(l)

    print(feature + ": "+ str(int(mojority)) +"/" + str(int(totalVal))
)

countingBased("A", attr_A)
countingBased("B", attr_B)
```

```
A: 6/8
B: 6/8
```

(b) Compute the information-theoretic heuristic, and order the features by it.

In [248]:
```
# TODO
def calEntropy(num):
    if num == 0:
        return 0
    return (-1) * num * np.log2(num)

def infoTheory(feature, attr, labels=2):
    totalNumberOfVal = 0
    total_trueVal = 0
    reminder = 0
    d_total = {}
    for l in attr:
        totalNumberOfVal += len(l)
        for value in l:
            if value in d_total:
                d_total[value] += 1
            else:
                d_total[value] = 1



    TotalAverage = 0.0
    for key in d_total.keys():
        labelPossibility = d_total[key] / totalNumberOfVal
        TotalAverage += calEntropy(labelPossibility)

    for l in attr:
        posibility = len(l) / totalNumberOfVal
        sub_label = {}
        for value in l:
            if value in sub_label:
                sub_label[value] += 1
            else:
                sub_label[value] = 1
        for key in sub_label.keys():
            sub_label_posibility = sub_label[key] / len(l)
            reminder += posibility * calEntropy(sub_label_posibility)
    gain = TotalAverage - reminder
    print (feature + ": " + str(gain))

infoTheory("A", attr_A)
infoTheory("B", attr_B)
```

```
A: 0.31127812445913283
B: 0.18872187554086706
```

(c) Discussion of results.

Typically the counting-based heuristic is based on the mojority from each branch, in this method we will treat each leaf equally. As for feature A, when we apply with counting-based heurisitic, we can see that the left branch weighs can convey more information than right branch. But when computing the results, the useful information brought by left branch get balanced by the right branch.\ Information-theoretic heuristic, however, can help us detect the really useful branch that conveys more information. After the computation of feature A and B, we can see that A weighs more than B, which makes more sense since the left branch of A can help excluding some data while B cannot do in either its branch. \ In real life, counting-based heuristic is easier to implemented but not as accurate as information-theoretic heuristic. Basically information-theoretic heuristic stands for the entropy of the information, which exaclty satisfied our requirment for finding the most "useful" information in building a decision tree.

2.2 Compute both heuristics for simplified abalone data.

(a) Compute the counting-based heuristic, and order the features by it.

In [249]:
```
#TODO
small_tr_x = np.loadtxt('./data_abalone/small_binary_x_train.csv', del
imiter=',', skiprows=1)
small_te_x = np.loadtxt('./data_abalone/small_binary_x_test.csv', deli
miter=',', skiprows=1)
small_tr_y = np.loadtxt('./data_abalone/3class_y_train.csv', delimiter
=',', skiprows=1)
small_te_y = np.loadtxt('./data_abalone/3class_y_test.csv', delimiter=
',', skiprows=1)
```

In [250]:
```
import csv
small_reader = csv.reader(open('./data_abalone/small_binary_x_train.csv','rt'))
small_headers = next(small_reader)
all_reader = csv.reader(open('./data_abalone/x_train.csv','rt'))
all_headers = next(all_reader)
for i in range(len(small_headers)):
    attr = []
    trueList = []
    falseList = []
    for index, value in enumerate(small_tr_x[:,i].transpose()):
        if value == 0:
            falseList.append(small_tr_y[index])
        else:
            trueList.append(small_tr_y[index])
    attr.append(trueList)
    attr.append(falseList)
    feature = small_headers[i]
    countingBased(feature, attr,3)
```

```
is_male: 1864/3176
length_mm: 2230/3176
diam_mm: 2266/3176
height_mm: 2316/3176
```

(b) Compute the information-theoretic heuristic, and order the features by it.

In [251]:
```
# TODO
for i in range(len(headers)):
    attr = []
    trueList = []
    falseList = []
    for index, value in enumerate(small_tr_x[:,i].transpose()):
        if value == 0:
            falseList.append(small_tr_y[index])
        else:
            trueList.append(small_tr_y[index])
    attr.append(trueList)
    attr.append(falseList)
    feature = headers[i]
    infoTheory(feature,attr,3)
```

```
is_male: 0.02451648227175207
length_mm: 0.13543816377043694
diam_mm: 0.1500706886802703
height_mm: 0.17302867291002488
```

2.3 Generate decision trees for full- and restricted-feature data

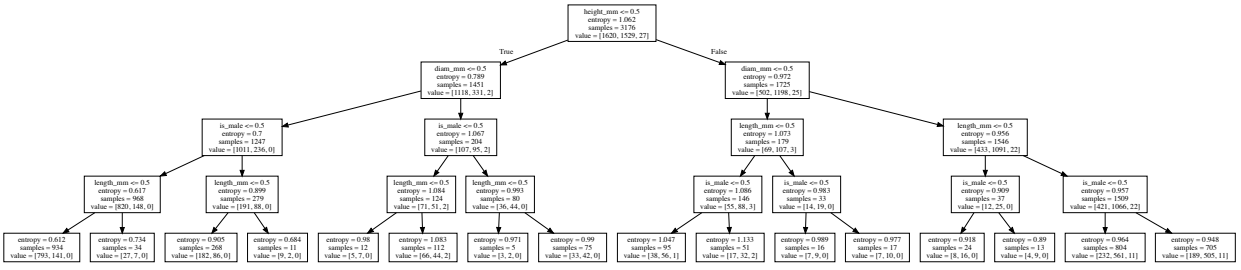
(a) Print accuracy values and generate tree images.

In [252]:
```
all_tr_x = np.loadtxt('./data_abalone/x_train.csv', delimiter=',', skiprows=1)
all_te_x = np.loadtxt('./data_abalone/x_test.csv', delimiter=',', skiprows=1)
all_tr_y = np.loadtxt('./data_abalone/y_train.csv', delimiter=',', skiprows=1)
all_te_y = np.loadtxt('./data_abalone/y_test.csv', delimiter=',', skiprows=1)
```

In [253]:
```
# TODO
DTM_Small = sklearn.tree.DecisionTreeClassifier(criterion = 'entropy')
DTM_Small = DTM_Small.fit(small_tr_x,small_tr_y)
score_te_small = DTM_Small.score(small_te_x,small_te_y)
score_tr_small = DTM_Small.score(small_tr_x,small_tr_y)
print ("Accuracy in testing data set is: %.4f"%score_te_small)
print ("Accuracy in training data set is: %.4f"%score_tr_small)
#sklearn.tree.plot_tree(DTM_Small)
dot_data = sklearn.tree.export_graphviz(DTM_Small,out_file=None,feature_names = small_headers)
graph = graphviz.Source(dot_data)
graph.render("simplified_dataset")
graph
```

```
Accuracy in testing data set is: 0.7220
Accuracy in training data set is: 0.7327
```

Out[253]:



In [254]:
```
# TODO
DTM_All = sklearn.tree.DecisionTreeClassifier(criterion = 'entropy')
DTM_All.fit(all_tr_x,all_tr_y)
score_te_all = DTM_All.score(all_te_x,all_te_y)
score_tr_all = DTM_All.score(all_tr_x,all_tr_y)
print ("Accuracy in testing data set is: %.4f"%score_te_all)
print ("Accuracy in training data set is: %.4f"%score_tr_all)
dot_data = sklearn.tree.export_graphviz(DTM_All,out_file=None,feature_
names = all_headers)
graph = graphviz.Source(dot_data)
graph.render("full_dataset")
```

```
Accuracy in testing data set is: 0.1900
Accuracy in training data set is: 1.0000
```

Out[254]:
```
'full_dataset.pdf'
```

In [255]:
```
graph
```

Out[255]:

(b) Discuss the results seen for the two trees :

Answer: As we can see from two decision trees, the former one with 4 features composed of simplified binary numbers is much smaller than the one with 8 features and all of float numbers. By giving more features with detailed data helps raise the accuracy of training dataset. But we can see that the testing accuracy is declined. This is called the overfitting problem, if we fill the model with all the features via all detailed data, then this will lead to an overfitting problem like we met before in linear regression model. \ So the first tree performs better than the second overfitting tree, and the cause of the overfitting is that there eixsts too much detailed features which leads to the number of leaves and depths of the tree much bigger than expected. When we have so much branches and leaves, the overffiting problem occurs, but we can avoid this by pruning some branches just like the first tree does.