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, 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/class/k0grsypt15j73g).

Import required libraries.

In [91]:

```
import os
import numpy as np
import pandas as pd

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

from matplotlib import pyplot as plt
import seaborn as sns
import sys
```

Part One: Cancer-Risk Screening

1.1: Compute true/false positives/negatives.

Complete the following code.

In [92]:

```
def calc_TP_TN_FP_FN(ytrue_N, yhat_N):
    ''' Compute counts of four possible outcomes of a binary classifier for eval
uation.

    Args
    ----
    ytrue_N : 1D array of floats
        Each entry represents the binary value (0 or 1) of 'true' label of one e
xample
        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

    '''

    if len(ytrue_N) != len(yhat_N):
        raise ValueError("Error: Input vector have different lengths")

    N = len(ytrue_N)

    TP = 0.0
    TN = 0.0
    FP = 0.0
    FN = 0.0

    for i in range(N):
        if (ytrue_N[i] == 1) and (yhat_N[i] == 1):
            TP += 1
        elif (ytrue_N[i] == 0) and (yhat_N[i] == 0):
            TN += 1
        elif (ytrue_N[i] == 0) and (yhat_N[i] == 1):
            FP += 1
        else:
            FN += 1

    return TP, TN, FP, FN
```

In [93]:

```
all0 = np.zeros(10)
all1 = np.ones(10)
calc_TP_TN_FP_FN(all0, all1)
```

Out[93]:

```
(0.0, 0.0, 10.0, 0.0)
```

In [94]:

```
calc_TP_TN_FP_FN(all1, all0)
```

Out[94]:

```
(0.0, 0.0, 0.0, 10.0)
```

In [95]:

```
calc_TP_TN_FP_FN(all1, all1)
```

Out[95]:

```
(10.0, 0.0, 0.0, 0.0)
```

In [96]:

```
calc_TP_TN_FP_FN(all0, all0)
```

Out[96]:

```
(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 [97]:

```
def calc_perf_metrics_for_threshold(ytrue_N, yproba1_N, thresh):
    ''' Compute performance metrics for a given probabilistic classifier and thr
eshold
    '''
    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 thre
shold
    '''
    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 [98]:

```
def calc_confusion_matrix_for_threshold(ytrue_N, yproba1_N, thresh):
    ''' Compute the confusion matrix for a given probabilistic classifier and th
reshold

    Args
    ----
    ytrue_N : 1D array of floats
        Each entry represents the binary value (0 or 1) of 'true' label of one e
xample
        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 [99]:

```
def compute_perf_metrics_across_thresholds(ytrue_N, yproba1_N, thresh_grid=None
):
    ''' Compute common binary classifier performance metrics across many thresho
lds

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

    Args
    ----
    ytrue_N : 1D array of floats
        Each entry represents the binary value (0 or 1) of 'true' label of one e
xample
        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

    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 threshold)
        * 'npv' : 1D array of negative predictive values (one per threshold)
        * '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(ytrue_N, yprob
a1_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.linspace(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 re
d)
    * middle row: hist of predicted probabilities for positive examples (shaded
 blue)
    * bottom row: line plots of metrics that require hard decisions (ACC, TPR, T
NR, 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(ytrue_N, ypr
oba1_N)
    axes[2].plot(thresh_grid, perf_grid['acc'], 'k-', label='accuracy')
    axes[2].plot(thresh_grid, perf_grid['tpr'], 'b-', label='TPR (recall/sensiti
vity)')
    axes[2].plot(thresh_grid, perf_grid['tnr'], 'g-', label='TNR (specificity)')
    axes[2].plot(thresh_grid, perf_grid['ppv'], 'c-', label='PPV (precision)')
    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 [100]:

```
# Load 3 feature version of x arrays
x_tr_M3 = np.loadtxt('./data_cancer/x_train.csv', delimiter=',', skiprows=1)
x_va_N3 = np.loadtxt('./data_cancer/x_valid.csv', delimiter=',', skiprows=1)
x_te_N3 = np.loadtxt('./data_cancer/x_test.csv', delimiter=',', skiprows=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 [101]:

```
y_tr_M = np.loadtxt('./data_cancer/y_train.csv', delimiter=',', skiprows=1)
y_va_N = np.loadtxt('./data_cancer/y_valid.csv', delimiter=',', skiprows=1)
y_te_N = np.loadtxt('./data_cancer/y_test.csv', delimiter=',', skiprows=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 [102]:

```
def numCorrect(vec):
    n = len(vec)
    num_0 = 0
    num_1 = 0

    for i in vec:
        if i == 1.0:
            num_1 += 1
        else:
            num_0 += 1

    return num_1


frac_train = numCorrect(y_tr_M) / len(y_tr_M)
frac_valid = numCorrect(y_va_N) / len(y_va_N)
frac_test = numCorrect(y_te_N) / len(y_te_N)


print("Fraction with cancer in TRAIN: %.3f" % frac_train)
print("Fraction with cancer in VALID: %.3f" % frac_valid)
print("Fraction with cancer in TEST : %.3f" % frac_test)
```

```
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 [103]:

```
print("Always-0: accuracy on VALID: %.3f" % (1.0 - frac_valid))
print("Always-0: accuracy on TEST : %.3f" % (1.0 - frac_test))
```

```
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 [104]:

```
#For the always-0 classifier, I just pass in zeros as the probability vector and
1 as a threshold.
print(calc_confusion_matrix_for_threshold(y_va_N, np.zeros(len(y_va_N)), 1))
```

```
Predicted    0  1
True
0          155  0
1           25  0
```

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

Answer: The accuracy of the always-0 classifier (for example, 0.861 for the valid dataset) makes sense because of the nature of the dataset in question, which contains medical patient entries as well as whether they have cancer or not(0,1). Because having cancer is less probable than not having cancer, it is entirely believable that the majority of the entries in this dataset will have a value of 0 which means no cancer, hence a high accuracy for this classifier is expected behavior. Having the accuracy of the always-0 classifier in mind is useful because it reflects a minimum bound for any future classifiers. That is, all classifiers that we will try to build should have a higher accuracy than the always-0 classifier.

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

Answer: The always-0 classifier is good to be used as a preliminary measure, because it is simple and fast, but for the task in hand it will not fair well and should not be employed because it will not lead to any advancements (waste of time). It will always miss all relevant entries that have a positive label (i.e. it will have only True negatives and False negatives) so it will have a True-positive rate of zero. It also does not make classifications by considering training dataset features. Because we need to minimize the need of a biopsy for cancer detection, a good classifier should be built based on other input features of dataset entries and should have a non-zero true-positive rate.

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 [105]:

```
from sklearn.linear_model import LogisticRegression
import math

tr_loss_list = list()
va_loss_list = list()

C_grid = np.logspace(-9, 6, 31)
log_C = []

LR_models = {}

for c in C_grid:
    log_C.append(math.log10(c))
    model = LogisticRegression(solver='liblinear', C=c)
    #fit training model
    model.fit(x_tr_M2, y_tr_M)
    prob_tr = model.predict_proba(x_tr_M2)
    loss_tr = sklearn.metrics.log_loss(y_tr_M, prob_tr)
    tr_loss_list.append(loss_tr)


    prob_va = model.predict_proba(x_va_N2)
    loss_va = sklearn.metrics.log_loss(y_va_N, prob_va)
    va_loss_list.append(loss_va)

    LR_models[c] = model

# TODO fit, predict_proba, and evaluate logistic loss
# Record the best model here
```

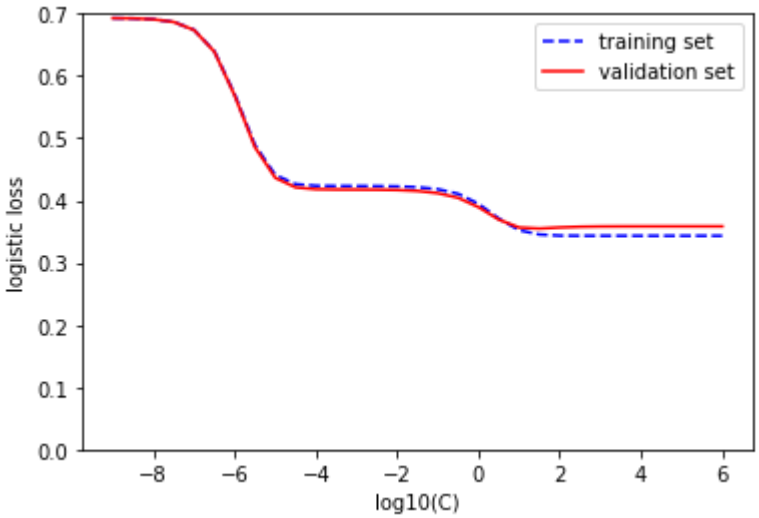
Plot logistic loss (y-axis) vs. C (x-axis) on the training set and validation set.

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

In [107]:

```
# TODO make plot
plt.xlabel('log10(C)');
plt.ylabel('logistic loss');
plt.ylim([0.0, 0.7]);
plt.plot(log_C, tr_loss_list, '-b', label="training set", linestyle="--")
plt.plot(log_C, va_loss_list, '-r', label="validation set")

# TODO add legend
plt.legend();

#log loss minimized for validation:
va_min_loss = min(va_loss_list)
idx = va_loss_list.index(va_min_loss)
min_c_two = C_grid[idx]

print("Best C-value for LR with 2-feature data: %.3f" % min_c_two) # TODO
print("Validation set log-loss at best C-value: %.4f" % va_min_loss)
```

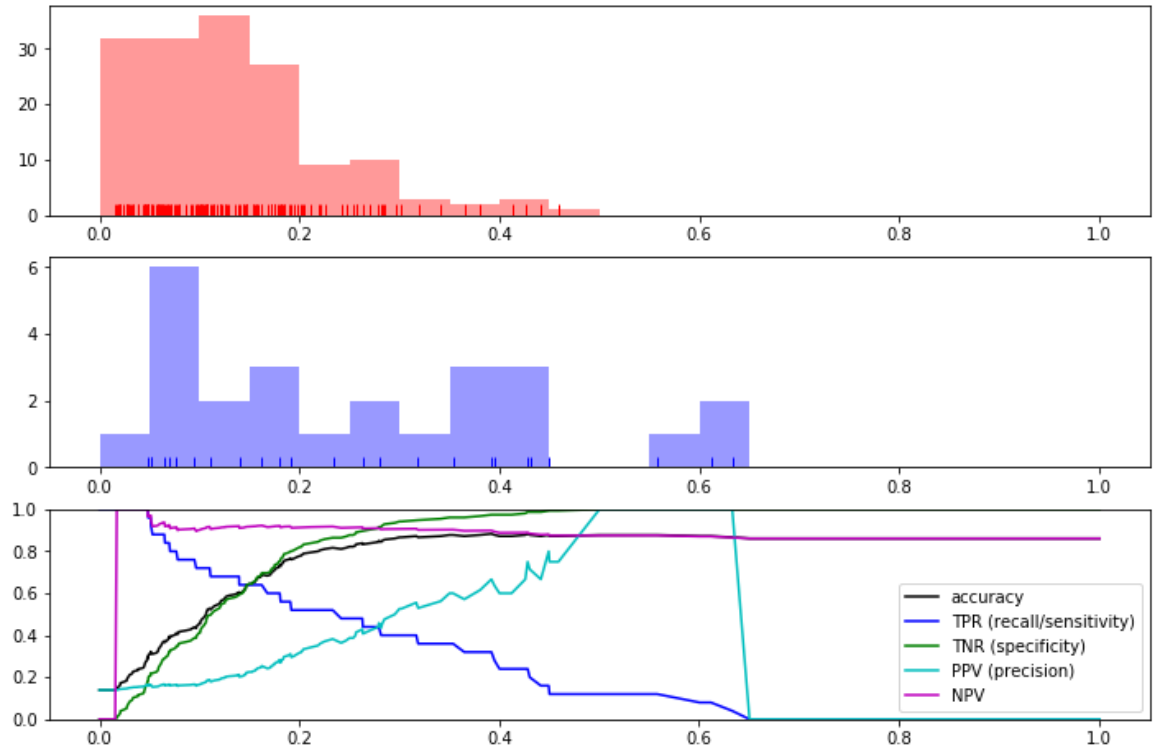
```
Best C-value for LR with 2-feature data: 31.623
Validation set log-loss at best C-value: 0.3549
```



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

In [127]:

```
def getProbEstimatesPositive(model, x):
    prob_val = model.predict_proba(x)
    y_prob = np.empty(len(prob_val))
    j = 0
    for i in prob_val:
        y_prob[j] = i[1]
        j += 1

    return y_prob


make_plot_perf_vs_threshold(y_va_N, getProbEstimatesPositive(LR_models[min_c_two
], x_va_N2))
```



(c) Model fitting with 3-feature data

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

In [126]:

```
tr_loss = list()
va_loss = list()

LR_three_models = {}


for c in C_grid:
    model = LogisticRegression(solver='liblinear', C=c)
    model.fit(x_tr_M3, y_tr_M)
    prob_tr = model.predict_proba(x_tr_M3)
    loss = sklearn.metrics.log_loss(y_tr_M, prob_tr)
    tr_loss.append(loss)

    prob_va = model.predict_proba(x_va_N3)
    loss = sklearn.metrics.log_loss(y_va_N, prob_va)
    va_loss.append(loss)

    LR_three_models[c] = model
```

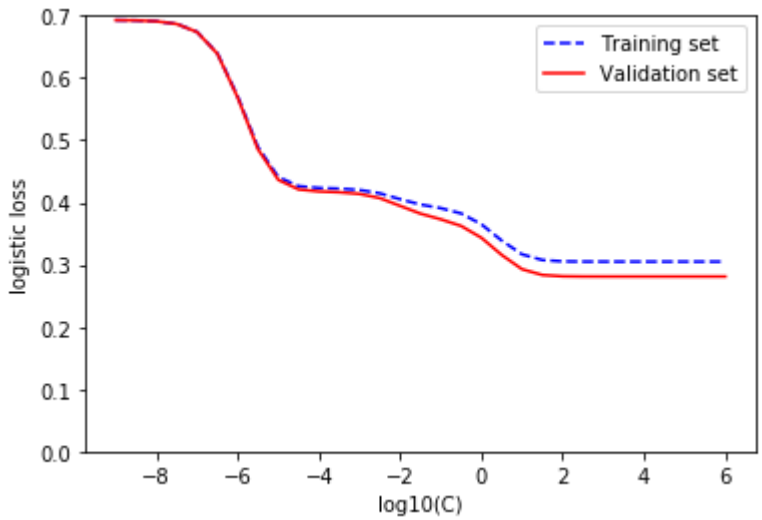
Plot logistic loss (y-axis) vs. C (x-axis) for the 3-feature classifiers on the training set and validation set.

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

In [125]:

```
plt.xlabel('log10(C)');
plt.ylabel('logistic loss');
plt.ylim([0.0, 0.7]);
plt.plot(log_C, tr_loss, '-b', label="Training set", linestyle = "--")
plt.plot(log_C, va_loss, '-r', label="Validation set")

plt.legend()

#log loss minimized for validation:
validation_loss = min(va_loss)
idx = va_loss.index(validation_loss)
min_c_three = C_grid[idx]

print("Best C-value for LR with 3-feature data: %.3f" % min_c_three) # TODO
print("Validation set log-loss at best C-value: %.4f" % validation_loss)
```

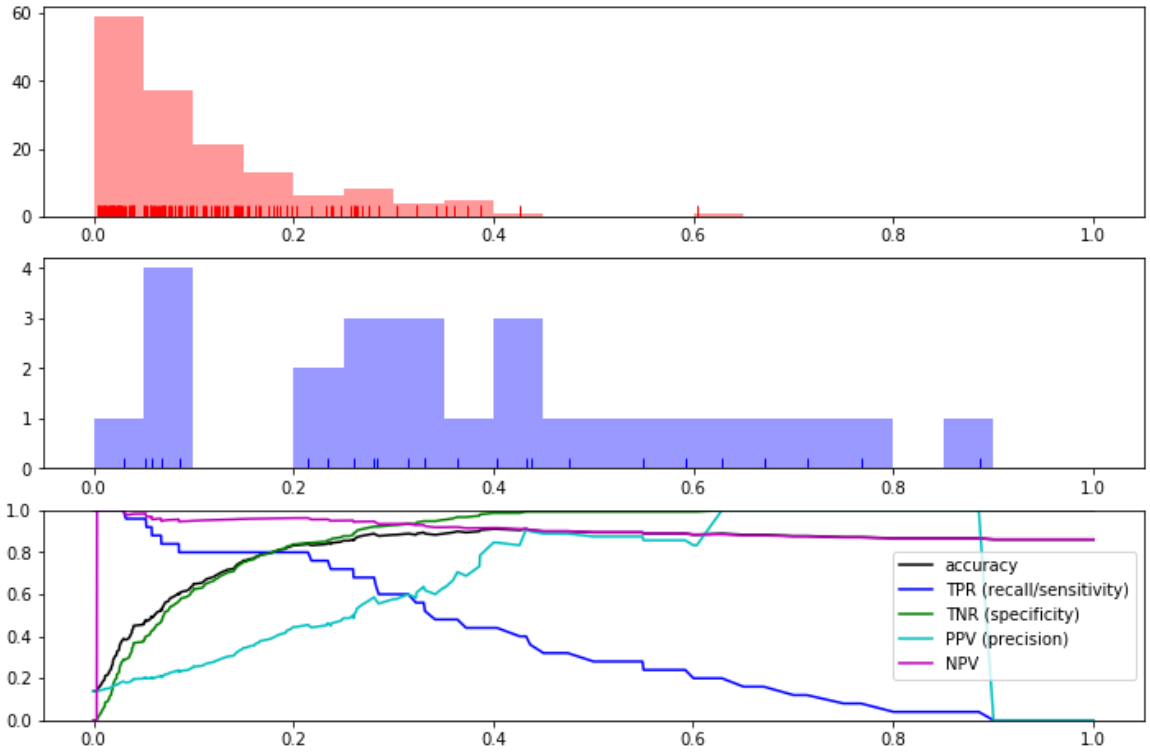
```
Best C-value for LR with 3-feature data: 1000000.000
Validation set log-loss at best C-value: 0.2810
```



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

In [111]:

```
yprob_N_three = getProbEstimatesPositive(LR_three_models[min_c_three], x_va_N3)
make_plot_perf_vs_threshold(y_va_N, yprob_N_three)
```



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.

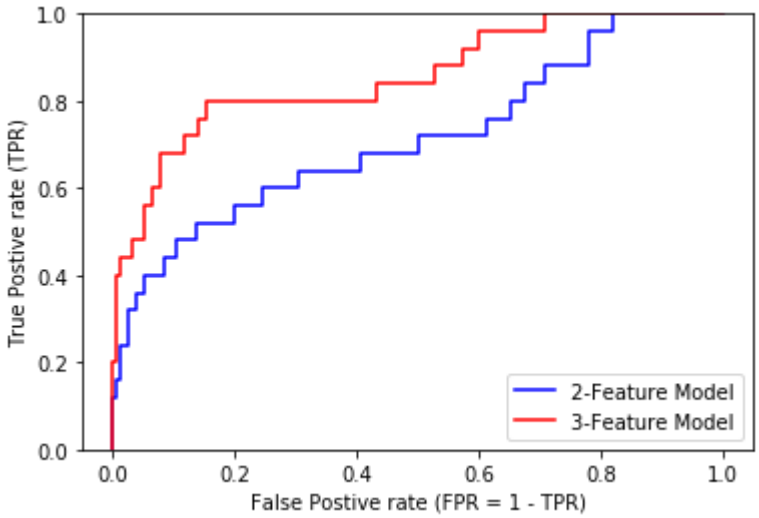
There should be two curves in the plot, one for each of the best two classifiers from prior steps.

In [124]:

```
y_score_two = getProbEstimatesPositive(LR_models[min_c_two], x_va_N2)
y_score_three = getProbEstimatesPositive(LR_three_models[min_c_three], x_va_N3)


# for 2-feature model:
fpr, tpr, thr = sklearn.metrics.roc_curve(y_va_N, y_score_two)

#for 3-feature model:
fpr_three, tpr_three, thr_three = sklearn.metrics.roc_curve(y_va_N, y_score_thre
e)

plt.ylim([0, 1]);
plt.xlabel("False Postive rate (FPR = 1 - TPR)")
plt.ylabel("True Postive rate (TPR)")
plt.plot(fpr, tpr, '-b', label="2-Feature Model")
plt.plot(fpr_three, tpr_three, '-r', label="3-Feature Model")
plt.legend(loc = "lower right")
```

Out[124]:

```
<matplotlib.legend.Legend at 0x1a17e45cd0>
```



(b) Plot ROC curves on the test set.

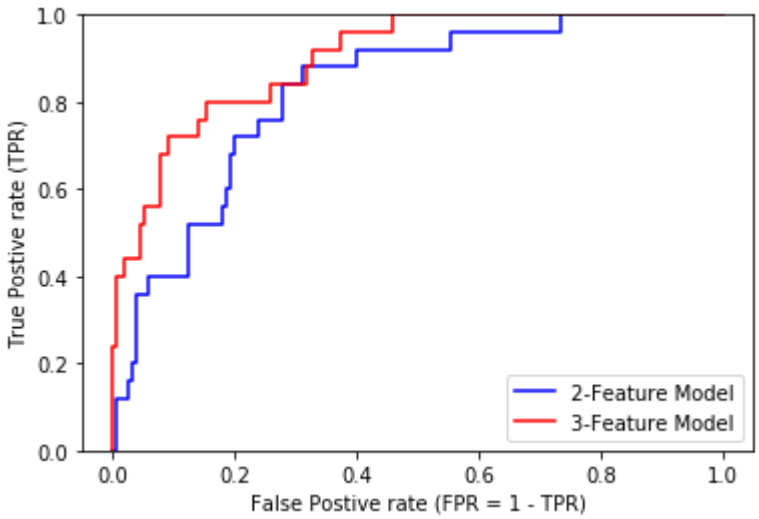
There should be two curves in the plot, one for each of the best two classifiers from prior steps.

In [114]:

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

y_score_two = getProbEstimatesPositive(LR_models[min_c_two], x_te_N2)
y_score_three = getProbEstimatesPositive(LR_three_models[min_c_three], x_te_N3)

fpr, tpr, thr = sklearn.metrics.roc_curve(y_te_N, y_score_two)
fpr_three, tpr_three, thr_three = sklearn.metrics.roc_curve(y_te_N, y_score_thre
e)


plt.ylim([0, 1]);
plt.xlabel("False Postive rate (FPR = 1 - TPR)");
plt.ylabel("True Postive rate (TPR)");
plt.plot(fpr, tpr, '-b', label="2-Feature Model")
plt.plot(fpr_three, tpr_three, '-r', label="3-Feature Model")
plt.legend(loc = "lower right")
```

Out[114]:

```
<matplotlib.legend.Legend at 0x1a17f5f5d0>
```



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

Answer: In the validation set, it is clear that the area under the ROC curve for the 3-feature model is greater than the area for the 2-feature model. This indicates that the 3-feature model will reach a higher true positive success rate faster than the 2-feature model and that generally the 3-feature model will hava a better overal performance. The same occurs in the testing set as the area under the 3-feature model curve is geater. Therefore, the 3-feature classifier is superior performance-wise.

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 [115]:

```
best_thr = 0.5

print("ON THE VALIDATION SET:")
print("Chosen best thr = %.4f" % best_thr)
print("")
print("ON THE TEST SET:")
y_probal_te = getProbEstimatesPositive(LR_three_models[min_c_three], x_te_N3)
print(calc_confusion_matrix_for_threshold(y_te_N, y_probal_te, best_thr)) # calc
confusion matrix
print("")
print_perf_metrics_for_threshold(y_te_N, y_probal_te, best_thr) # printing of va
rious metrics
```

```
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 [123]:

```
y_probal_va = getProbEstimatesPositive(LR_three_models[min_c_three], x_va_N3)
thresh_grid, perf_dict = compute_perf_metrics_across_thresholds(y_va_N, y_probal
_va)

# TODO Find threshold that makes TPR as large as possible, while satisfying PPV
 >= 0.98
# get all elems with ppv >= 0.98 -> add them to a list of tuples (i0, i1) where
 i0 is index in thresh_grid
max_ppv = []
c = 0
for ppv in perf_dict['ppv']:
    if ppv >= 0.98:
        max_ppv.append((c, ppv))
    c += 1

max_tpr = 0
best_thr = thresh_grid[0]

for i in max_ppv:
    val = perf_dict['tpr'][i[0]]
    if val >= max_tpr:
        max_tpr = val
        best_thr = thresh_grid[i[0]]

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, y_probal_te, best_thr))
print("")
print_perf_metrics_for_threshold(y_te_N, y_probal_te, best_thr)
```

```
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 [122]:

```
y_probal_va = getProbEstimatesPositive(LR_three_models[min_c_three], x_va_N3)
thresh_grid, perf_dict = compute_perf_metrics_across_thresholds(y_va_N, y_probal
_va)

# TODO Find threshold that makes PPV as large as possible, while satisfying TPR
 >= 0.98
tpr_vals = []
idx = 0
for tpr in perf_dict['tpr']:
    if tpr >= 0.98:
        tpr_vals.append((idx, tpr))
    idx += 1
max_ppv = 0
best_thr = thresh_grid[0]

for i in tpr_vals:
    val = perf_dict['ppv'][i[0]]
    if val >= max_ppv:
        max_ppv = val
        best_thr = thresh_grid[i[0]]


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, y_probal_te, best_thr))
print("")
print_perf_metrics_for_threshold(y_te_N, y_probal_te, 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: For the task in hand, the idea of wanting to avoid life-threatening mistakes at all costs is closely related to maximizing the classifier's True Positive Rate (because we want to be sure that the classifier does its best in detecting cancer early), as well as maximizing the classifier's True Negative Rate because being able to specify the absence of cancer with high accuracy is the most important factor in minimizing the need for a biopsy. In addition, we saw from the accuracy of the always-0 classifier that there is a high class imbalance in our dataset (a lot more examples don't have cancer than do). For the purposes of our classifier therefore, False Positives are more tolerable than False Negatives, so more emphasis has to be placed on minimizing false negatives. A good comparison for our three confusion matrices is to calculate the F-2 measure of each classifier. Note that F-2 is good for our case because it places more emphasis on false negative. The F-2 results are 0.442, 0.238 and 0.56 for classifiers a,b and c respectively. A good place to start is classifier C with the highest F-2 measure.

Starting with a simple analysis of the confusion matrices yields an error rate of 0.1, 0.11 and 0.54 for classifiers A,B and C respectively. The error rate of the third classifier is high due to the high number of False positives (98) with respect to the other two classifiers. Although this classifier correctly labels all positive examples (0 False negatives and high F-2 score), its high number of false positives (low ppv) does not help in eliminating biopsies as all examples that were classified wrongly as relevant would have to be biopsied, so even though it has the highest F-2 measure it is not appropriate for our task. On the other hand, classifiers A and B have a similar and low error rate. As far as classifier a and b are concerned, they have similar and high TNR so a low number of false positives (much lower than classifier c). They have similar non-zero false negatives (c has zero false negatives), which means that relevant information is missed, and similar non-zero false positives. To minimize the number of biopsies, the number of false positives should be zero. In classifier B, a prediction of 1 was never made when truth was 0 as the number of FP is zero but a higher number of relevant information is missed as it has more false negatives than A. Classifier A has non-zero but very low number of False positives, and lower number of False Negatives. Using classifier A for this dataset, keeps the total negative rate reasonably high (0.981 compared to 1 for b) which means that biopsies are minimized and also has a higher F-2 score and double the TPR of b. Therefore, for our task I believe that classifier A is better all-around. Therefore, the default threshold of 0.5 should be used.

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

Answer: Using a threshold of 0.5 the total number of biopsies that can be avoided is TP + TN for the confusion matrix of that threshold. TP + TN = 152 + 10 = 162 biopsies would be avoided. In other words, 90% of biopsies could have been avoided if this version of the classifier was used.

Part Two: Abalone Age

2.1: Build k-neighor regression models.

Your code should load the abalone data, and build a sequence of nearest neighbor models for it. For each model, the MSE on the various input sets should be determined.

In [121]:

```
# Load in data
x_tr_data = np.loadtxt('./data_abalone/x_train.csv', delimiter=',', skiprows=1)
x_va_data = np.loadtxt('./data_abalone/x_valid.csv', delimiter=',', skiprows=1)
x_te_data = np.loadtxt('./data_abalone/x_test.csv', delimiter=',', skiprows=1)

y_tr_data = np.loadtxt('./data_abalone/y_train.csv', delimiter=',', skiprows=1)
y_va_data = np.loadtxt('./data_abalone/y_valid.csv', delimiter=',', skiprows=1)
y_te_data = np.loadtxt('./data_abalone/y_test.csv', delimiter=',', skiprows=1)


# variables for computation
N = len(y_tr_data)
k_values = [1, 3, 5, 7, 9, 11, 21, 41, 61, 81, 101, 201, 401, 801, N]

MSE_tr = list()
MSE_va = list()
MSE_te = list()

best_model_idx = 0 # Index of best k for validation model (mninimum MSE) will be
stored here
min_mse_error = float('inf')

for k in k_values:
    knn_regr = sklearn.neighbors.KNeighborsRegressor(n_neighbors=k, algorithm='b
rute', metric='euclidean')
    #fit on training data
    knn_regr.fit(x_tr_data, y_tr_data)

    #predict validation and testing data and record MSE
    y_tr_pred = knn_regr.predict(x_tr_data)
    y_val_pred = knn_regr.predict(x_va_data)
    y_te_pred = knn_regr.predict(x_te_data)

    val_error = sklearn.metrics.mean_squared_error(y_va_data, y_val_pred)
    MSE_va.append(val_error)


    #compare with maximum
    if val_error < min_mse_error:
        min_mse_error = val_error
        best_model_idx = len(MSE_va) - 1

    MSE_tr.append(sklearn.metrics.mean_squared_error(y_tr_data, y_tr_pred))
    MSE_te.append(sklearn.metrics.mean_squared_error(y_te_data, y_te_pred))
```

2.2: Plot the MSE.

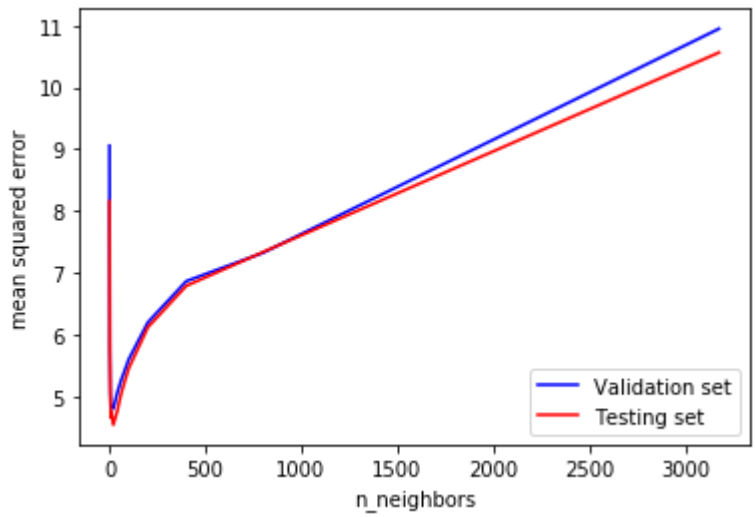
Your code should plot the mean squared error on the test and validation sets, for each of the settings of the number of neighbors.

In [119]:

```
plt.xlabel('n_neighbors');
plt.ylabel('mean squared error');

plt.plot(k_values, MSE_va, '-b', label="Validation set")
plt.plot(k_values, MSE_te, '-r', label="Testing set")
plt.legend(loc = "lower right")
```

Out[119]:

```
<matplotlib.legend.Legend at 0x1a17e47e90>
```



2.3: Determine best overall model.

In [120]:

```
print("The minimal error model on the validation set has n_neighbors = %d" % k_v
alues[best_model_idx]) # TODO: fix
print("")
print("MSE for Training set: %f" % MSE_tr[best_model_idx])
print("MSE for Validation set: %f" % MSE_va[best_model_idx])
print("MSE for Testing set: %f" % MSE_te[best_model_idx])
```

```
The minimal error model on the validation set has n_neighbors = 21

MSE for Training set: 4.384303
MSE for Validation set: 4.804739
MSE for Testing set: 4.542798
```

In [ ]:

```

```