HW03 Notebook

Complete the following notebook, as described in the PDF for Homework 03 (included in the download with the starter code). Submit the following:

1. This notebook file and `hw3.py`, 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.

NOTE: The purpose of this notebook is to demonstrate the functionality implemented in `hw3.py`. As part of this demo, all analysis (i.e., questions that prompt for a short answer) are to be added to the notebook. Keep the order of the problems as listed in the assignment description. Furthermore, cells are provided as placeholders for each response; however, cells can be added as needed.

Please report any questions to the class Piazza page.

Import required libraries.

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

import warnings

import sklearn.linear_model
import sklearn.metrics
from hw3 import calc_confusion_matrix_for_threshold
from hw3 import calc_percent_cancer
from hw3 import calc_binary_metrics
from hw3 import predict_0_always_classifier
from hw3 import calc_accuracy
from hw3 import print_perf_metrics_for_threshold
from hw3 import calc_perf_metrics_for_threshold

from matplotlib import pyplot as plt
import seaborn as sns

# load .py changes
%load_ext autoreload
%autoreload 2

%matplotlib inline
plt.style.use('seaborn') # pretty matplotlib plots
```

```
/var/folders/sp/njx9g_354jz0zcs8bj_s_2cm0000gn/T/
ipykernel_43054/1998611089.py:25: MatplotlibDeprecationWarning: The
seaborn styles shipped by Matplotlib are deprecated since 3.6, as they
no longer correspond to the styles shipped by seaborn. However, they
```

```
will remain available as 'seaborn-v0_8-<style>'. Alternatively,
directly use the seaborn API instead.
  plt.style.use('seaborn') # pretty matplotlib plots
```

1) Function to calculate TP, TN, FP, and FN.

The following four calls to the function `calc_binary_metrics` to test it. This way, the function can be tested for several edge cases. ***Don't modify this.***

```
all0 = np.zeros(10)
all1 = np.ones(10)
TP, TN, FP, FN = calc_binary_metrics(all0, all1)
print(f"0 vs 1\n========\nTP: {TP}\nTN: {TN}\nFP: {FP}\nFN: {FN}")
```

```
0 vs 1
========
TP: 0.0
TN: 0.0
FP: 10.0
FN: 0.0
```

```
TP, TN, FP, FN = calc_binary_metrics(all1, all0)
print(f"1 vs 0\n========\nTP: {TP}\nTN: {TN}\nFP: {FP}\nFN: {FN}")
```

```
1 vs 0
========
TP: 0.0
TN: 0.0
FP: 0.0
FN: 10.0
```

```
TP, TN, FP, FN = calc_binary_metrics(all1, all1)
print(f"1 vs 1\n========\nTP: {TP}\nTN: {TN}\nFP: {FP}\nFN: {FN}")
```

```
1 vs 1
========
TP: 10.0
TN: 0.0
FP: 0.0
FN: 0.0
```

```
TP, TN, FP, FN = calc_binary_metrics(all0, all0)
print(f"0 vs 0\n========\nTP: {TP}\nTN: {TN}\nFP: {FP}\nFN: {FN}")
```

```
0 vs 0
========
TP: 0.0
TN: 10.0
FP: 0.0
FN: 0.0
```

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.

```
# Load the x-data and y-class arrays
x_train = np.loadtxt('./data/x_train.csv', delimiter=',', skiprows=1)
x_test = np.loadtxt('./data/x_test.csv', delimiter=',', skiprows=1)

y_train = np.loadtxt('./data/y_train.csv', delimiter=',', skiprows=1)
y_test = np.loadtxt('./data/y_test.csv', delimiter=',', skiprows=1)
```

Inspect Data. The following should ***not*** be modified.

```
feat_names = np.loadtxt(f'data/x_train.csv', delimiter=',', dtype=str,
max_rows=1)
print(f"features: {feat_names}\n")
target_name = np.loadtxt(f'data/x_test.csv', delimiter=',', dtype=str,
max_rows=1)
df_sampled_data = pd.DataFrame(x_test, columns=feat_names)
df_sampled_data[str(target_name)] = y_test
df_sampled_data.sample(15)
```

```
features: ['age' 'famhistory' 'marker']
```

	age	famhistory	marker	['age' 'famhistory' 'marker']
115	66.86597	0.0	0.065931	0.0
14	68.27528	0.0	0.316553	0.0
88	76.69211	0.0	0.373651	0.0
98	62.80098	0.0	0.061921	0.0
60	63.59347	0.0	2.948919	0.0
137	59.93010	0.0	0.042513	0.0
172	66.20989	1.0	0.812278	0.0
3	66.91557	0.0	2.612115	0.0
113	69.00063	1.0	0.375517	0.0
41	58.86290	0.0	0.234737	0.0
22	61.34507	0.0	0.535074	0.0
47	63.99250	0.0	0.070910	0.0
107	62.48188	1.0	0.237959	0.0
148	64.68880	0.0	0.042153	0.0
100	71.86153	0.0	0.879809	0.0

2) Compute the fraction of patients with cancer.

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

```
#TODO: modify these prints
tr_percent = calc_percent_cancer(y_train)
te_percent = calc_percent_cancer(y_test)
```

```
print("Percent of data that has_cancer on TRAIN: %.3f" % tr_percent)
print("Percent of data that has_cancer on TEST : %.3f" % te_percent)
```

```
Percent of data that has_cancer on TRAIN: 14.035
Percent of data that has_cancer on TEST : 13.889
```

3) The predict-0-always baseline

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

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

```
#TODO: implement predict_0_always_classifer()
y_train_pred = predict_0_always_classifier(x_train)
y_test_pred = predict_0_always_classifier(x_test)

acc_train = calc_accuracy(*calc_binary_metrics(y_train, y_train_pred))
acc_test = calc_accuracy(*calc_binary_metrics(y_test, y_test_pred))
print("acc on TRAIN: %.3f" % acc_train) #TODO: modify these values
print("acc on TEST : %.3f" % acc_test)
```

```
acc on TRAIN: 0.860
acc on TEST : 0.861
```

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

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

```
# TODO: call print(calc_confusion_matrix_for_threshold(...))
print(calc_confusion_matrix_for_threshold(y_true_N=y_test,
y_proba1_N=y_test_pred)) # args : y_true_N, y_proba1_N, thresh=0.5
```

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

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

Answer: TODO You will see reasonable accuracy for the simple baseline classifier. Why not just use it for this task? Your answer, written into the notebook as text, should detail the pluses and minuses of using this simple classifier

This is pretty good accuracy at roughly 86%. Before going into the dangers of using it for the true model, we can definitely choose it for a baseline reference. So we will seek a higher accuracy rating for our chosen model - since what is the point of using machine learning if you can't do better than guessing no one has cancer. So we will need to see above 86% accuracy ideally. Accuracy doesn't tell the whole story though and we need to have a high

True Positive rate in our model, not just a > 86% accuracy rating. The False Negatives are good, however.

Since we are always assuming there is no cancer; however, that is very dangerous if 14% of people have cancer, and we choose not to biopsy them. That is unacceptable. If this was a social media algorithm like a recommender, 86% is plenty good, but life and death needs far better accuracy. It is safer to biopsy all patients to catch the 14%. You don't want to biopsy an unnecessary 86% on the other hand, but that is a lower concern since it is not taking the risk of misdiagnosis. That is why we can't use only a always-0 or always-1 predictor since surgery and people's lives are the consequences.

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

Answer: TODO Given the task of this classification experiment—determining if patients have cancer without doing a biopsy—what are the errors the always-0 classifier can make? For each such type of mistake, what would be the cost of that mistake? (Possible costs might be, for example, lost time or money, among other things.) What would you recommend about using this classifier, given these possibilities? Be sure to speak about the binary metrics (i.e., TP, FP, TN, FN).

TN FP FN TP

TP: will never occur - not a concern with this classifier, and really bad that no one with cancer knows they have it

FP: will never occur - not a concern with this classifier, this would cause biopsies which are an inconvenience but not a lost life Results in wasted time and money and potentially going through chemo to remediate a problem that they do not have

TN: will always be perfect, not a concern here

FN: no one will be diagnosed with cancer that has cancer - the worst case - so people will likely die, there families will sue the hopsital and doctors and develops of the model, the hospital will go out of business, the doctors will be barred from practicing medicine and lose their licenses, doctors and devs could face manslaughter charges or worse, and so on.

4: Basic Perceptron Models

(i) Normalize data

```
from hw3 import standardize_data
# print(x_test)
#TODO
X_train, X_test = standardize_data(X_train=x_train, X_test=x_test)

print(X_train)
```

```
[[0.53637473 0.         0.10816263]
 [0.43257395 0.         0.05418465]
 [0.72202778 1.         0.18904397]
 ...
```

```
 [0.45372565 0.         0.57842732]
 [0.33260801 0.         0.1189701 ]
 [0.62875077 0.         0.03734379]]
```

(ii) Create a basic `Perceptron` classifier

Fit a perceptron to the training data. Print out accuracy on this data, as well as on testing data. Print out a confusion matrix on the testing data.

```
#TODO: train a basic perceptron model using default parameter values,
and modify these accuracies below
# Perceptron model
# # fit to training data
# # make prediction on test data

from hw3 import perceptron_classifier
# inputs => {normalized X_train, X_test; standard y_train, y_test}
pred_train, pred_test = perceptron_classifier(x_train=X_train,
y_train=y_train, x_test=X_test, y_test=y_test)
# [opt args] default vals: penalty="l2", alpha=0, random_state=42)

from sklearn.metrics import accuracy_score
print("acc on TRAIN: %.3f" % accuracy_score(y_train, pred_train))
print("acc on TEST : %.3f" % accuracy_score(y_test, pred_test))
# print("acc on TRAIN: %.3f" % 0)
# print("acc on TEST : %.3f" % 0)

print("\nConfusion matrix for TEST:")
# TODO: call print(calc_confusion_matrix_for_threshold(...))
print(calc_confusion_matrix_for_threshold(y_test, pred_test))
# print(calc_confusion_matrix_for_threshold(...))
# print(calc_confusion_matrix_for_threshold(y_true_N, y_proba1_N,
thresh=0.5))

acc on TRAIN: 0.249
acc on TEST : 0.272

Confusion matrix for TEST:
Predicted   0    1
True
0          24  131
1           0   25
```

(iii) Compare the `Perceptron` to the always-0 classifier.

Answer: TODO TN FP FN TP

The perceptron classifier is much worse than the always-0 classifier for accuracy. However, it is more helpful to look at the individual categories. The FN count is 0 which means no one with cancer is told they don't have cancer. This is super important. There are fewer people told they don't have cancer who don't but that is not a bad thing. This classifier is more

careful in a sense. The TP detection is clearly better with 25 people knowing they have cancer for sure. The weight is heavily towards the FP group which means a lot of people think they have cancer when they don't so that will result in angst and biopsies, time and money, but that is still a far better thing than patients losing their lives. I would highly prefer this model and just tell the patients that although it says they probably have cancer, the accuracy is not high, and they should get a biopsy to make sure. I would be comfortable having this model in a hospital as long as this was thoroughly told to the patients.

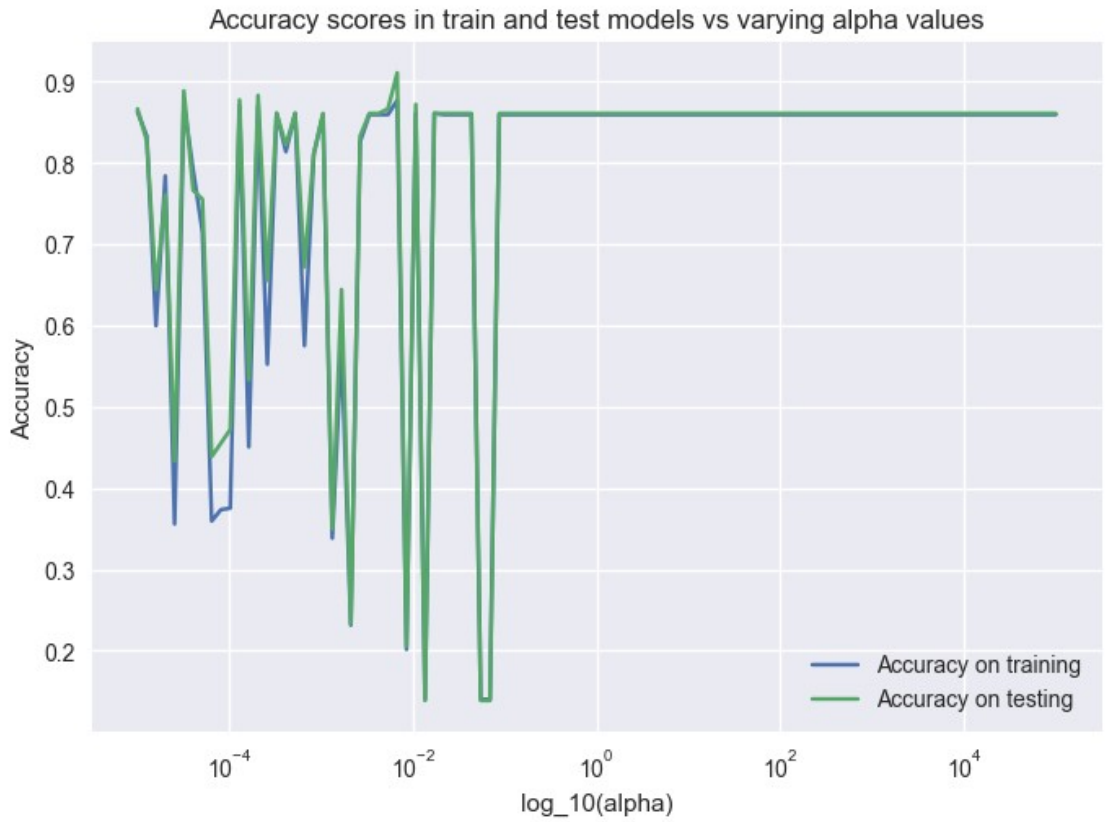
(iv) Generate a series of regularized perceptron models

Each model will use a different `alpha` value, multiplying that by the L2 penalty. You will record and plot the accuracy of each model on both training and test data.

```
# TODO: create, fit models here and record accuracy of each (Implement
functions needed)
from hw3 import series_of_preceptrons

alphas = np.logspace(-5, 5, base=10, num=100)
# print(alphas)

train_accuracy_list, test_accuracy_list =
series_of_preceptrons(x_train=X_train, y_train=y_train, x_test=X_test,
y_test=y_test, alphas=alphas)
```

Plot accuracy on train/test data across the different alpha values plotted on a logarithmic scale. Make sure to show title, legends, and axis labels.

```
# TODO make plot
plt.plot(alphas, train_accuracy_list, label='Accuracy on training')
plt.plot(alphas, test_accuracy_list, label='Accuracy on testing')

# TODO add legend, titles, etc. set x-scale appropriately
plt.title('Accuracy scores in train and test models vs varying alpha
values')
plt.legend(loc='lower right')
plt.xscale('log')
plt.xlabel('log_10(alpha)')
plt.ylabel('Accuracy')
plt.show()
```



(iv) Discuss what the plot is showing you.

Answer: TODO What does the performance plot tell you? It will look quite different from the ones you would have seen; think about these differences and address them in your answer (hints: Is there a specific trend? Does it look normal? How does the performance look compared to the performance of the always-0 classifier? Is regularization helpful?).

The plot shows the accuracy score for each alpha level (totaling 100) for the training and testing data. As the log scale moves to the right we can see that accuracy on training and testing merges and is identical from roughly 10^-3 and larger alpha values. The initial training and testing data follows each other closely for alphas up to roughly 10^-3. The training accuracy dips more than the test data in each case where they diverge - likely because there is more data and outliers are more common likely given more data. They can be higher in the sum of the squared residuals in L2 Regularization because at these points the alpha resultant penalty is small and doesn't overshadow the features. The regularization appears to be helpful because you can't compare age and chemical markers and family history without standardizing the data to a common range (of 0 to 1). The accuracy swinging wildly from around 10% to around 90% at error rates below 10^-1 seems like a case of overfit but the training data and testing data is consistent. The final accuracy rate of approximately 86% on alpha values above 10^-1 suggests underfit as the penalty is growing at this point and can cancel out the sum of squared residuals - which are based on the features - which are the basis of our cancer prediction. A property of L2 Regularization aka Ridge Regression is that the penalty increases to the square of the

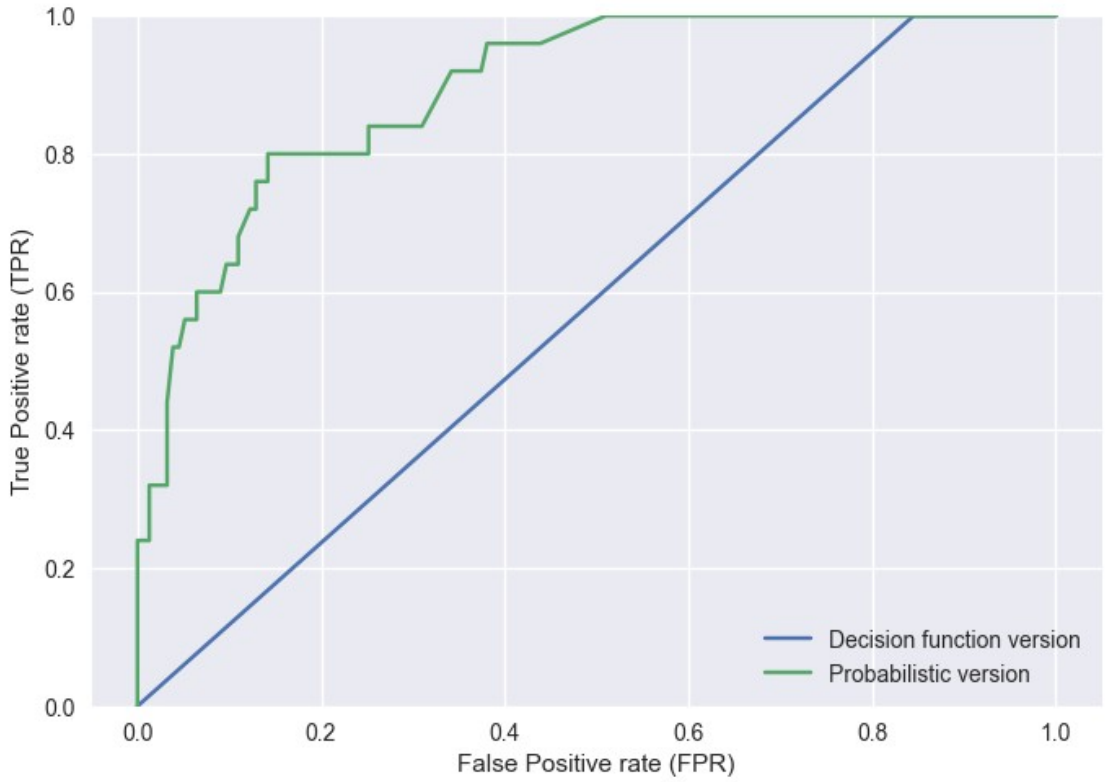
magnitudes of the coefficients. L1 will remove features but L2 is not used to remove features since in this study age, family history, and the chemical marker are highly correlated. The plateau of accuracy at ~86% with alphas above 10^-1 matches close to the always-0 classifier's accuracy score and if the binary metrics are equivalent to predicting always-0, that is not helpful. If we have 86% accuracy with no FN values and a high distribution of TP values, that is a much more helpful model.

5: Decision functions and probabilistic predictions

(a) Create two new sets of predictions

Fit `Perceptron` and `CalibratedClassifierCV` models to the data. Use their predictions to generate ROC curves.

```
# TODO: fit a Perceptron and generate its decision_function() over the
test data.
# _, decisions_test = perceptron_classifier()  # pred_train, pred_test
# _, pred_prob_test = calibrated_perceptron_classifier()  #
pred_train, pred_test
from hw3 import perceptron_classifier,
calibrated_perceptron_classifier

decisions_train, decisions_test = perceptron_classifier(X_train,
y_train, X_test, y_test)

# TODO: Build a CalibratedClassifierCV, using a Perceptron as its
base_estimator,
#       and generate its probabilistic predictions over the test data.
# pred_prob_train, pred_prob_test =
calibrated_perceptron_classifier(decisions_train, y_train,
decisions_test, y_test)
pred_prob_train, pred_prob_test =
calibrated_perceptron_classifier(X_train, y_train, X_test, y_test)

# TODO something like: fpr, tpr, thr = sklearn.metrics.roc_curve(...)
from hw3 import find_best_thresholds
from sklearn.metrics import roc_curve

fpr, tpr, thr = roc_curve(y_test, decisions_test)
plt.plot(fpr, tpr, label='Decision function version')

fpr2, tpr2, thr2 = roc_curve(y_test, pred_prob_test)
plt.plot(fpr2, tpr2, label='Probabilistic version')

plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.xlabel("False Positive rate (FPR)")
plt.ylabel("True Positive rate (TPR)")

Text(0, 0.5, 'True Positive rate (TPR)')
```




```
from sklearn.metrics import roc_auc_score

# print("AUC on TEST for Perceptron: %.3f" %
sklearn.metrics.roc_auc_score(
#     y_test, decisions_test))
# print("AUC on TEST for probabilistic model: %.3f" %
sklearn.metrics.roc_auc_score(
#         y_test, pred_prob_test))

# print("AUC on TEST for Perceptron: %.3f" % 0) #TODO: modify these
values
# print("AUC on TEST for probabilistic model: %.3f" % 0)
test_ppn_ras = roc_auc_score(y_test, decisions_test)
test_pm_ras = roc_auc_score(y_test, pred_prob_test)

print("AUC on TEST for Perceptron: %.3f" % test_ppn_ras) #TODO: modify
these values
print("AUC on TEST for probabilistic model: %.3f" % test_pm_ras)
```

```
AUC on TEST for Perceptron: 0.577
AUC on TEST for probabilistic model: 0.894
```

(b) Discuss the results above

Answer: TODO Discuss the results in the plots and AUC values. How do the two versions of the model differ? How are they similar? Which might you prefer, and why?

Above a False Positive Rate of approx 0.83, the decision function and probabilistic function converge at an ideal case of a 1.0 True Positive Rate. The linear trend of the decision function is predictable and stable, so I might choose this for a baseline or for general cases. For sheer performance this dataset shows the probabilistic version climbs to higher TPR rates much more quickly and hits 1.0 TPR as we pass roughly a 0.5 False Positive Rate. The roughly 0.15 to 0.25 FPR range shows the probabilistic version stall out on its improvement of TPR which stays at 0.80 temporarily. If I was looking at that specific FPR range I might want to look deeper as to why this is happening or use the decision function instead for predictable improvements of TPR. The rapid spike of the Probabilistic version at FPR levels 0.0, and roughly 0.03 are excellent for progress across a dismal False Positive Rate. There could be a unique use case for this. The area captured by the Probabilistic version is nearly 90% which shows great performance but the perceptron/decision function is just over half at about 58%. So I can't justify choosing the decision function for this instance.

(c) Compute model metrics for different probabilistic thresholds

Complete `calc_perf_metrics_for_threshold` that takes in a set of correct outputs, a matching set of probabilities generated by a classifier, and a threshold at which to set the positive decision probability, and returns a set of metrics if we use that threshold.

(d) Compare the probabilistic classifier across multiple decision thresholds

Try a range of thresholds for classifying data into the positive class (1). For each threshold, compute the true positive rate (TPR) and positive predictive value (PPV). Record the best value of each metric, along with the threshold that achieves it, and the *other* metric at that threshold.

```
# TODO: test different thresholds to compute these values
from hw3 import find_best_thresholds

best_TPR, best_PPV_for_best_TPR, best_TPR_threshold, best_PPV,
best_TPR_for_best_PPV, best_PPV_threshold =
find_best_thresholds(y_test=y_test, pred_prob_test=pred_prob_test) #
y_true_N, y_proba1_N, thresh=0.5

print("Best TPR threshold: %.4f => TPR: %.4f; PPV: %.4f" %
(best_TPR_threshold, best_TPR, best_PPV_for_best_TPR))
print("Best PPV threshold: %.4f => PPV: %.4f; TPR: %.4f" %
(best_PPV_threshold, best_PPV, best_TPR_for_best_PPV))
```

```
Best TPR threshold: 0.0400 => TPR: 1.0000; PPV: 0.2294
Best PPV threshold: 0.6406 => PPV: 1.0000; TPR: 0.2400
```

(e) Exploring different thresholds

(i) Using default 0.5 threshold.

Generate confusion matrix and metrics for probabilistic classifier, using threshold 0.5.

```
best_thr = 0.5
print("ON THE TEST SET:")
print("Chosen best threshold = %.4f" % best_thr)
print("")
# TODO: print(calc_confusion_matrix_for_threshold(...))
# print(calc_confusion_matrix_for_threshold(...))
print(calc_confusion_matrix_for_threshold(y_test, pred_prob_test,
best_thr))

print("")
# TODO: print_perf_metrics_for_threshold(...)
# print_perf_metrics_for_threshold(...)
print_perf_metrics_for_threshold(y_test, pred_prob_test, best_thr)
```

```
ON THE TEST SET:
Chosen best threshold = 0.5000

Predicted    0   1
True
0          150   5
1           15  10

0.889 ACC
0.400 TPR
0.968 TNR
0.667 PPV
0.909 NPV
```

(ii) Using threshold with highest TPR.

Generate confusion matrix and metrics for probabilistic classifier, using threshold that maximizes TPR.

```
best_thr = best_TPR_threshold
print("ON THE TEST SET:")
print("Chosen best threshold = %.4f" % best_thr)
print("")

# TODO: print(calc_confusion_matrix_for_threshold(...))
print("")
# print(calc_confusion_matrix_for_threshold(...))
print(calc_confusion_matrix_for_threshold(y_test, pred_prob_test,
best_thr))

# TODO: print_perf_metrics_for_threshold(...)
# print_perf_metrics_for_threshold(...)
print_perf_metrics_for_threshold(y_test, pred_prob_test, best_thr)
```

```
ON THE TEST SET:
Chosen best threshold = 0.0400
```

```
Predicted   0   1
True
0          71  84
1           0  25
0.533 ACC
1.000 TPR
0.458 TNR
0.229 PPV
1.000 NPV
```

(iii) Using threshold with highest PPV.

Generate confusion matrix and metrics for probabilistic classifier, using threshold that maximizes PPV.

```
best_thr = best_PPV_threshold
print("ON THE TEST SET:")
print("Chosen best threshold = %.4f" % best_thr)
print("")

# TODO: print(calc_confusion_matrix_for_threshold(...))
# print(calc_confusion_matrix_for_threshold(...))
print(calc_confusion_matrix_for_threshold(y_test, pred_prob_test,
best_thr))

# TODO: print_perf_metrics_for_threshold(...)
# print_perf_metrics_for_threshold(...)
print_perf_metrics_for_threshold(y_test, pred_prob_test, best_thr)
```

```
ON THE TEST SET:
Chosen best threshold = 0.6406

Predicted    0  1
True
0          155  0
1           19  6
0.894 ACC
0.240 TPR
1.000 TNR
1.000 PPV
0.891 NPV
```

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

Answer: TODO Discuss these results. What would the effects be if we decided to apply each of these thresholds to our data and then used that process to classify patients and decide whether to do biopsies?

TN FP FN TP

A) default threshold (0.5) Predicted 0 1 True 0 150 5 1 15 10

B) highest TPR threshold (0.0400) Predicted 0 1 True 0 71 84 1 0 25

C) highest PPV threshold (0.6406) Predicted 0 1 True 0 155 0 1 19 6

TN FP FN TP

Conceptually, I care about detecting cancer whenever it is there above all else, and then doing further checks on unclear data, so the TPR being high is great but a high FNR is most important of all to avoid deaths. If I was to look at overall accuracy it would be between the default threshold and probabilistic model, but in the case of cancer, I care most about having a perfect FN category so no one who has cancer is told that they don't. For safety, I would choose the highest TPR threshold as a model since there are no False Negatives. This is easily overlooked if studying just accuracy, but the binary metrics are much more important for cancer. The default model and highest PPV threshold are very similar in accuracy and bucketization. Although the highest PPV one doesn't have any False Positives, there are more False Negatives than the default, and I would rather have plenty of False Positives than even 1 more False Negative in a model for cancer diagnosis.

A) most people without cancer will be accurately told they don't have cancer and some people with cancer will be told they have it. There will be a few unnecessary surgeries due to 5 FP values, but the dangerous part is the 15 False Negative values which means those people will likely die of cancer because they were incorrectly told they didn't have it. Or it could be caught at a later time and the chemo etc. will be very rough.

B) This model is not great at telling people they don't have cancer accurately but there are no False Negatives, so it is the only safe model. No patients are told they don't have cancer when they actually do. It is fairly good at correctly guessing a patient has cancer. It's biggest flaw is False Positives so there will be a lot of unnecessary surgeries and people freaking out, but it is still better than people dying. The key is to tell people that a diagnosis does not have a high accuracy rate, and they should get a biopsy to make sure it is not cancer. I choose this one.

C) This is the best at accurately predicting no cancer. It doesn't produce False Positives, so no surgeries will take place incorrectly. It is pretty bad at predicting cancer with only 6 detections correct. There are the most False Negative values so there will likely be the most death as a result of this hospital's negligence. And lots of collateral damage. Don't be tricked by the best accuracy score alone!

A) default threshold (0.5) 0.889 ACC 0.400 TPR 0.968 TNR 0.667 PPV 0.909 NPV

B) highest TPR threshold (0.0400) 0.533 ACC 1.000 TPR 0.458 TNR 0.229 PPV 1.000 NPV

C) highest PPV threshold (0.6406) 0.894 ACC 0.240 TPR 1.000 TNR 1.000 PPV 0.891 NPV

A) Here we have a solid accuracy score of 88.9%. We have no perfect scores here but TNR is close, so you can tell patients they don't have cancer, well. The NPV score (90.9%) is very good. The TPR (40.0%) and PPV (66.7%) are lacking scores which indicates low ability to diagnose cancer.

B) We have a low accuracy score of 53.3%, which is worse than our always-0 baseline. However, the content of what really matters in our binary metrics is favorable. The perfect scores in TPR and NPV, show cancer detection is high and many people know if they don't have cancer accurately, so they can avoid surgery. There are poor scores in TNR (45.8%) and PPV (22.9%) as well. These being low doesn't matter too much, its just not ideal at diagnosing cancer like the perceptron model; causing additional testing to take place.

C) This yields the best accuracy score at 89.4% by a small margin over the 0.5 threshold. The perfect scores in TNR and PPV makes this model a good no cancer predictor and cancer detector of the very low TPR score, so that part is almost canceled out in effectiveness due to high FN values which are dangerous. The NPV (89.1%) is very good so people can know if they don't have cancer most times. Less surgery.

TPR = Predict cancer and has cancer # goal TNR = Predict no cancer and has no cancer # goal FPR = Predict cancer and has no cancer # extra surgery - inconvenient FNR = Predict no cancer and has cancer # dangerous - people die / hospital gets sued

sensitivity

tpr = tp / (tp + fn) # proportion of correctly predicting cancer, of all the people with cancer

specificity

tnr = tn / (tn + fp) # proportion of correctly predicting no cancer, of all the people with no cancer

precision

ppv = tp / (tp + fp) # proportion of correctly predicting cancer, of all predictions of cancer

npv

npv = tn / (tn + fn) # proportion of correctly predicting no cancer, of all predictions of no cancer

---

Initial Notes:

If I was a patient I would care most about detecting cancer (at whatever cost). If I had control of a study I would categorize very certain TN and TP values so those people know for sure if they have cancer or not. With any gray area data I would suggest a biopsy to prevent patients leaving with cancer and not knowing or thinking they don't have it. The basic perceptron model was actually very good in this respect. There weren't false negatives which is the major binary metric to avoid for patient safety. Very similar to the highest TPR threshold model.