lab01

January 29, 2020

1 Lab 1: Fairness and Machine Learning

Welcome to the first DS102 lab!

The goals of this lab are to get familiar with concepts in decision theory (true positive rates, false positive rates, etc.) through the lens of fairness in machine learning. This lab is adapted from Chapter 2 of the book Fairness and Machine Learning: Limitations and Opportunities.

The code you need to write is commented out with a message "TODO: fill in". There is additional documentation for each part as you go along.

1.1 Course Policies

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the labs, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** in the cell below.

Submission: to submit this assignment, rerun the notebook from scratch (by selecting Kernel > Restart & Run all), and then print as a pdf (File > download as > pdf) and submit it to Gradescope.

This assignment should be completed and submitted before Thursday, January 30, 2020 at 11:59 PM.

2 Collaborators

Write the names of your collaborators in this cell.

3 Introduction

Risk assessment is an important component of the criminal justice system. In the United States, judges set bail and decide pre-trial detention based on their assessment of the risk that a released defendant would fail to appear at trial or cause harm to the public. While actuarial risk assessment is not new in this domain, there is increasing support for the use of learned risk scores to guide human judges in their decisions. Proponents argue that machine learning could lead to greater efficiency and less biased decisions compared with human judgment. Critical voices raise the concern that such scores can perpetuate inequalities found in historical data, and systematically harm historically disadvantaged groups.

In this lab, we'll begin to scratch at the surface of the complex criminal justice domain. Our starting point is an investigation carried out by ProPublica of a proprietary risk score, called the

COMPAS score. These scores are intended to assess the risk that a defendant will re-offend, a task often called recidivism prediction. Within the academic community, the ProPublica article drew attention to issues with the use of machine learning for decision making.

4 Setup

Let's begin by importing the libraries we will use. * matplotlib * numpy * pandas

[30]:
```
import matplotlib.pyplot as plt
import matplotlib.ticker
import numpy as np
import pandas as pd

%matplotlib inline

# Imports and helper functions used for tests.
import hashlib
import sys
def get_hash(num):
    return hashlib.md5(str(num).encode()).hexdigest()
```

5 Dataset setup

We'll use data obtained and released by ProPublica as a result of a public records request in Broward Country, Florida, concerning the COMPAS recidivism prediction system. The data is available at https://raw.githubusercontent.com/propublica/compas-analysis/master/compas-scores-two-years.csv. Following ProPublica's analysis, we'll filter out rows where `days_b_screening_arrest` is over 30 or under 30, leaving us with 6,172 rows.

Here, we will download the data from the link above and apply this filter.

[31]:
```
data_url = "https://raw.githubusercontent.com/propublica/compas-analysis/master/
 ↪compas-scores-two-years.csv"
df = pd.read_csv(data_url)
df = df.query('days_b_screening_arrest <= 30 & days_b_screening_arrest >= -30')
len(df)
```

[31]:
```
6172
```

[32]:
```
# We will also filter the data for only two races.
races = ['African-American', 'Caucasian']
df = df[df['race'].isin(races)]
```

6 1. From scores to classifiers

We will observe what happens when we use COMPAS risk scores to create a classifier to predict whether an individual will re-offend (recidivate). We are not given the original COMPAS risk score, but we are given a column which contains the decile of the COMPAS risk score (similar to

a percentile, but out of 10). We will refer to this decile value (a number between 1 and 10) as our COMPAS "decile score". Our classifier will take the form of a threshold on the COMPAS decile score.

6.0.1 Group the dataset by `race` and `decile_score`.

[42]:
```
# Create a separate dataframe for each race and decile_score pair.
groups = df.groupby(['race', 'decile_score'], as_index=False)
```

6.1 1a) For each `race`/`decile_score` pair, compute:

1. the number of cases where recidivism occurred within two years,
2. the number of cases where recidivism did not occur, and
3. the total number of examples in the dataframe.

Note: the column `two_year_recid` is a column that takes value 1 if recidivism occurred within two years, and 0 if recidivism did not occur.

[43]:
```
# TODO: compute the total number of examples in which recidivism occured within␣
 ↪two years.
def recid_count_fn(df_recid_column):
    """Computes the total number of examples in which recidivism occurred.

    Args:
      df_recid_column: dataframe column where each row takes value 1 if
        recidivism occured, and 0 if recidivism did not occur.

    Returns:
      The total number of rows in which recidivism occurred.
    """
    recid_count = df_recid_column.tolist().count(1)
    return recid_count

print("Total number of examples of recidivism in the dataset:",␣
 ↪recid_count_fn(df['two_year_recid']))

# Test for correctness of this function.
assert(get_hash(recid_count_fn(df['two_year_recid'])) ==␣
 ↪'2c6ae45a3e88aee548c0714fad7f8269')
print("Test passed!")
```

```
Total number of examples of recidivism in the dataset: 2483
Test passed!
```

[44]:
```
# TODO: compute the total number of examples in which recidivism did not occur␣
 ↪within two years.
def non_recid_count_fn(df_recid_column):
    """Computes the total number of examples in which recidivism did not occur.
```

```
    Args:
      df_recid_column: dataframe column where each row takes value 1 if
        recidivism occured, and 0 if recidivism did not occur.

    Returns:
      The total number of rows in which recidivism did not occur.
    """
    non_recid_count = df_recid_column.tolist().count(0)
    return non_recid_count


print("Total number of examples of non-recidivism in the dataset:",␣
 ↪non_recid_count_fn(df['two_year_recid']))

# Test for correctness of this function.
assert(get_hash(non_recid_count_fn(df['two_year_recid'])) ==␣
 ↪'a7f592cef8b130a6967a90617db5681b')
print("Test passed!")
```

```
Total number of examples of non-recidivism in the dataset: 2795
Test passed!
```

[45]:
```
# TODO: Compute the total number of examples in the dataset.
def total_count_fn(df_recid_column):
    """Computes the total number of examples in the dataset.

    Args:
      df_recid_column: dataframe column where each row takes value 1 if
        recidivism occured, and 0 if recidivism did not occur.

    Returns:
      The total number of rows in the dataset.
    """
    total_count = len(df_recid_column.tolist())
    return total_count

print("Total number of examples in the dataset:",␣
 ↪total_count_fn(df['two_year_recid']))

# Test for correctness of this function.
assert(get_hash(total_count_fn(df['two_year_recid'])) ==␣
 ↪'82f292a22966b857d968fb578ccbead9')
print("Test passed!")
```

```
Total number of examples in the dataset: 5278
Test passed!
```

6.1.1 Create the `summary` dataframe

We now create a dataframe called `summary`, where each row contains summary statistics for each `race/decile_score` pair, including the total number of examples with that race and decile_score (`total_count`), the number of examples with that race and decile_score where recidivism occurred (`recid_count`), and the number of examples with that race and decile_score where recidivism did not occur (`non_recid_count`).

The `.agg` function below applies the functions you just wrote over a column of the dataframe corresponding to each `race/decile_score` pair. Each function will be computed on the column `two_year_recid` for each `race/decile_score` pair.

[48]:
```
# Each function will be computed on the column 'two_year_recid' for each group.
# Note: no TODOs in this cell, just run the cell and understand what it's doing.
summary = groups['two_year_recid'].agg({'recid_count': recid_count_fn,␣
 ↪'non_recid_count': non_recid_count_fn, 'total_count': total_count_fn})
summary
```

[48]:

	race	decile_score	recid_count	non_recid_count	total_count
0	African-American	1	85	280	365
1	African-American	2	105	241	346
2	African-American	3	125	173	298
3	African-American	4	158	179	337
4	African-American	5	158	165	323
5	African-American	6	187	131	318
6	African-American	7	209	134	343
7	African-American	8	215	86	301
8	African-American	9	229	88	317
9	African-American	10	190	37	227
10	Caucasian	1	128	477	605
11	Caucasian	2	100	221	321
12	Caucasian	3	82	156	238
13	Caucasian	4	98	145	243
14	Caucasian	5	91	109	200
15	Caucasian	6	93	67	160
16	Caucasian	7	68	45	113
17	Caucasian	8	72	24	96
18	Caucasian	9	55	22	77
19	Caucasian	10	35	15	50

6.2 1b) Compute the classifier outcomes for different decision thresholds.

For each race in the `summary` dataframe, we will now observe the outcomes of the classifier when the decision threshold occurs at each `decile_score`.

Specifically, we will iterate through the `decile_scores` in the `summary` dataframe, and for each `decile_score`, we will compute the number of true positives, true negatives, false positives, and false negatives under the assumption that the decision threshold occurs just below this `decile_score`. For example, in the row of the `summary` dataframe corresponding to a `decile_score` of 5, we will compute the number of true positives under the assumption that every example receiving a `decile_score` of 5 or above is classified as positive.

[54]:
```
# TODO: compute the number of true positives assuming the decision threshold
# occurs just below each decile_score.
# If you'd like, you may also restructure the loop inside this function
# (but do not change the function definition).
def get_TP_column(summary_df):
    """Returns an array of the number of true positives for each decile_score␣
 ↪threshold.

    Args:
      summary_df: dataframe containing columns for 'decile_score',␣
 ↪'recid_count',
        'non_recid_count', and 'total_count' for a single race.

    Returns:
      An array of the number of true positives for each decile_score (under the␣
 ↪assumption that every example
      receiving that row's decile_score or above is classified as positive --␣
 ↪aka, the decision threshold occurs
      just below the row's decile score.)
    """
    TPs = []
    for threshold in summary_df['decile_score']:
        positive_predict = 0
        for i in summary_df['decile_score']:
            if threshold <= i:
                positive_predict = positive_predict + summary_df['recid_count'].
 ↪tolist()[i - 1]
        true_positives = positive_predict
        # TODO: compute the number of true positives for this threshold.
        # Hint: iterate through the summary_df, compare the threshold to␣
 ↪'decile_score', and add up the 'recid_count' column.
        TPs.append(true_positives)
    return np.array(TPs, dtype=np.int32)

print("TP column for Caucasian:",␣
 ↪get_TP_column(summary[summary['race']=='Caucasian']))

assert(get_hash(get_TP_column(summary[summary['race']=='Caucasian'])) ==␣
 ↪'fdca79d31fe11760d9c6a06a4f8cb660')
print("Test passed!")
```

```
TP column for Caucasian: [822 694 594 512 414 323 230 162  90  35]
Test passed!
```

[55]:
```
# TODO: compute the number of true negatives assuming the decision threshold
# occurs just below each decile_score.
def get_TN_column(summary_df):
```

```
    """Returns an array of the number of true negatives for each decile_score␣
 ↪threshold.

    Args:
      summary_df: dataframe containing columns for 'decile_score',␣
 ↪'recid_count',
        'non_recid_count', and 'total_count' for a given race.

    Returns:
      An array of the number of true negatives for each decile_score (under the␣
 ↪assumption that every example
      receiving that row's decile_score or above is classified as positive --␣
 ↪aka, the decision threshold occurs
      just below the row's decile score.)
    """
    TNs = []
    for threshold in summary_df['decile_score']:
        negative_predict = 0
        for i in summary_df['decile_score']:
            if threshold > i:
                negative_predict = negative_predict +␣
 ↪summary_df['non_recid_count'].tolist()[i - 1]
        true_negatives = negative_predict
        # TODO: compute the number of true negatives for this threshold.
        TNs.append(true_negatives)
    return np.array(TNs, dtype=np.int32)

print("TN column for Caucasian:",␣
 ↪get_TN_column(summary[summary['race']=='Caucasian']))

assert(get_hash(get_TN_column(summary[summary['race']=='Caucasian']))  ==␣
 ↪'8175ee4854079441b234ca97e7f9a1c5')
print("Test passed!")
```

```
TN column for Caucasian: [   0  477  698  854  999 1108 1175 1220 1244 1266]
Test passed!
```

[57]:
```
# TODO: compute the number of false positives assuming the decision threshold
# occurs just below each decile_score.
def get_FP_column(summary_df):
    """Returns an array of the number of false positives for each decile_score␣
 ↪threshold.

    Args:
      summary_df: dataframe containing columns for 'decile_score',␣
 ↪'recid_count',
        'non_recid_count', and 'total_count' for a given race.
```

```
    Returns:
      An array of the number of false positives for each decile_score (under␣
 ↪the assumption that every example
      receiving that row's decile_score or above is classified as positive --␣
 ↪aka, the decision threshold occurs
      just below the row's decile score.)
    """
    FPs = []
    for threshold in summary_df['decile_score']:
        false_predict = 0
        for i in summary_df['decile_score']:
            if threshold <= i:
                false_predict = false_predict + summary_df['non_recid_count'].
 ↪tolist()[i - 1]
        false_positives = false_predict
        # TODO: compute the number of false positives for this threshold.      ␣
 ↪
        FPs.append(false_positives)
    return np.array(FPs, dtype=np.int32)

print("FP column for Caucasian:",␣
 ↪get_FP_column(summary[summary['race']=='Caucasian']))

assert(get_hash(get_FP_column(summary[summary['race']=='Caucasian']))  ==␣
 ↪'bc995fd0c02ad77eb6924ea48482f9ed')
print("Test passed!")
```

```
FP column for Caucasian: [1281  804  583  427  282  173  106   61   37   15]
Test passed!
```

[58]:
```
# TODO: compute the number of false negatives assuming the decision threshold
# occurs just below each decile_score.
def get_FN_column(summary_df):
    """Returns an array of the number of false negatives for each decile_score␣
 ↪threshold.

    Args:
      summary_df: dataframe containing columns for 'decile_score',␣
 ↪'recid_count',
        'non_recid_count', and 'total_count' for a given race.

    Returns:
      An array of the number of false negatives for each decile_score (under␣
 ↪the assumption that every example
      receiving that row's decile_score or above is classified as positive --␣
 ↪aka, the decision threshold occurs
```

```
      just below the row's decile score.)
    """
    FNs = []
    for threshold in summary_df['decile_score']:
        wrong_predict = 0
        for i in summary_df['decile_score']:
            if threshold > i:
                wrong_predict = wrong_predict + summary_df['recid_count'].
 ↪tolist()[i - 1]
        false_negatives = wrong_predict
        # TODO: compute the number of false negatives for this threshold.
        FNs.append(false_negatives)
    return np.array(FNs, dtype=np.int32)

print("FN column for Caucasian:",␣
 ↪get_FN_column(summary[summary['race']=='Caucasian']))

assert(get_hash(get_FN_column(summary[summary['race']=='Caucasian']))  ==␣
 ↪'f680125060a0bfb18a1447c6658fdfb8')
print("Test passed!")
```

```
FN column for Caucasian: [  0 128 228 310 408 499 592 660 732 787]
Test passed!
```

[61]:
```
# Note: no TODOs in this cell, just run it to assign the columns you created to␣
 ↪the summary dataframe.
# Fill in the TP, TN, FP, FN for each race.
for race in races:
    rows = summary['race'] == race
    summary.loc[rows, 'TP'] = get_TP_column(summary[rows])
    summary.loc[rows, 'TN'] = get_TN_column(summary[rows])
    summary.loc[rows, 'FP'] = get_FP_column(summary[rows])
    summary.loc[rows, 'FN'] = get_FN_column(summary[rows])
summary.fillna(0 , inplace=True)
```

[62]:
```
summary
```

[62]:

	race	decile_score	recid_count	non_recid_count	total_count	\
0	African-American	1	85	280	365	
1	African-American	2	105	241	346	
2	African-American	3	125	173	298	
3	African-American	4	158	179	337	
4	African-American	5	158	165	323	
5	African-American	6	187	131	318	
6	African-American	7	209	134	343	
7	African-American	8	215	86	301	
8	African-American	9	229	88	317	
9	African-American	10	190	37	227	

10	Caucasian	1	128	477	605
11	Caucasian	2	100	221	321
12	Caucasian	3	82	156	238
13	Caucasian	4	98	145	243
14	Caucasian	5	91	109	200
15	Caucasian	6	93	67	160
16	Caucasian	7	68	45	113
17	Caucasian	8	72	24	96
18	Caucasian	9	55	22	77
19	Caucasian	10	35	15	50

	TP	TN	FP	FN
0	1661.0	0.0	1514.0	0.0
1	1576.0	280.0	1234.0	85.0
2	1471.0	521.0	993.0	190.0
3	1346.0	694.0	820.0	315.0
4	1188.0	873.0	641.0	473.0
5	1030.0	1038.0	476.0	631.0
6	843.0	1169.0	345.0	818.0
7	634.0	1303.0	211.0	1027.0
8	419.0	1389.0	125.0	1242.0
9	190.0	1477.0	37.0	1471.0
10	822.0	0.0	1281.0	0.0
11	694.0	477.0	804.0	128.0
12	594.0	698.0	583.0	228.0
13	512.0	854.0	427.0	310.0
14	414.0	999.0	282.0	408.0
15	323.0	1108.0	173.0	499.0
16	230.0	1175.0	106.0	592.0
17	162.0	1220.0	61.0	660.0
18	90.0	1244.0	37.0	732.0
19	35.0	1266.0	15.0	787.0

6.3 1c) Plot the ROC curve for each race.

Here, we will compute the the true positive rate (TPR), false positive rate (FPR), and positive predictive value (PPV) metrics. PPV is also known as precision, and is defined as the number of true positives divided by the number of examples classified as positive.

Then, we will plot the ROC curve for each race.

[67]:
```
# TODO: compute the TPR for each race using other columns in summary.
summary['TPR'] = summary['TP'] / (summary['TP'] + summary['FN']) # TODO: fill␣
 ↪in the columns to use.

assert(get_hash(np.array(summary['TPR'].round(1), dtype=np.float32)) ==␣
 ↪'4e6755717caed5f4d6a62fc6fe8abbee')
print("Test passed!")
```

```
Test passed!
```

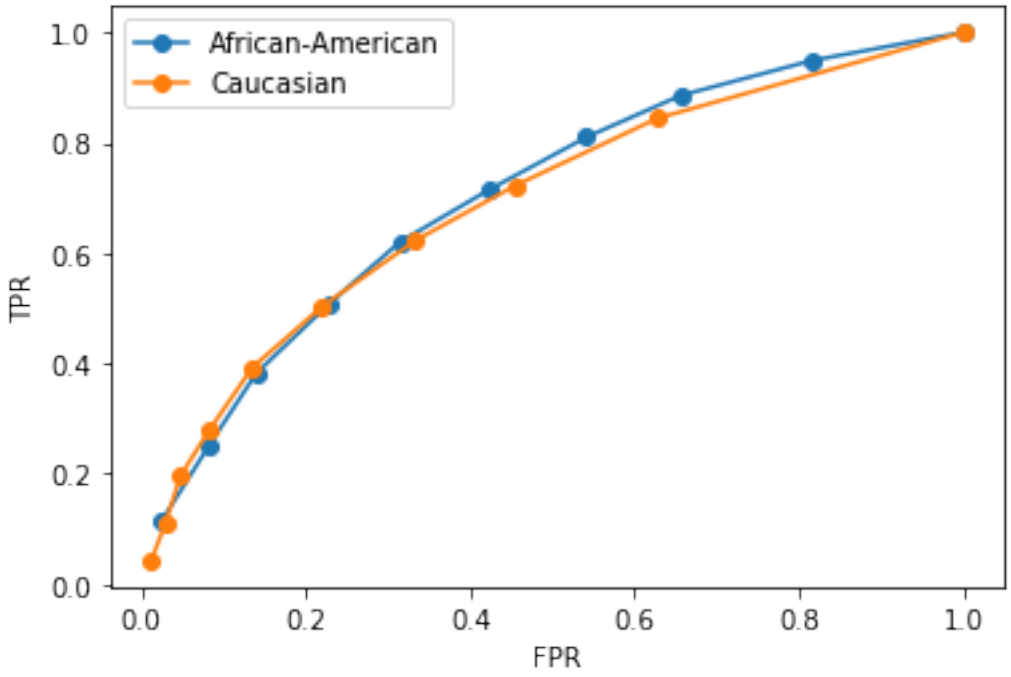
[69]:
```
# TODO: compute the FPR for each race using other columns in summary.
summary['FPR'] = summary['FP'] / (summary['TN'] + summary['FP']) # TODO: fill␣
 ↪in the columns to use.

assert(get_hash(np.array(summary['FPR'].round(1), dtype=np.float32)) ==␣
 ↪'e0eba6216d5fada1d6eabbfe568c527b')
print("Test passed!")
```

```
Test passed!
```

[70]:
```
# TODO: compute the PPV for each race using other columns in summary.
summary['PPV'] = summary['TP'] / (summary['TP'] + summary['FP']) # TODO: fill␣
 ↪in the columns to use.

assert(get_hash(np.array(summary['PPV'].round(1), dtype=np.float32))  ==␣
 ↪'6db5b24b7cd46af910f6a781eee40134')
print("Test passed!")
```

```
Test passed!
```

[72]:
```
# TODO: plot the ROC curve for each race.
plt.figure()

for race in races:
    rows = summary[summary['race']==race]
    plt.plot(rows['FPR'], rows['TPR'], '-o', label=race) # TODO: fill in the␣
 ↪correct columns to use.

plt.legend()
plt.xlabel('FPR') # TODO: fill in the correct labels for the x axis on an ROC␣
 ↪curve.
plt.ylabel('TPR') # TODO: fill in the correct labels for the y axis on an ROC␣
 ↪curve.
plt.show()
```



6.4 1d) Equalizing TPR and FPR.

Can you find two thresholds (one for black defendants, one for white defendants) such that FPR and TPR are roughly equal for the two groups (say, within 1% of each other)? Note: trivial thresholds of 0 or 11 don't count. Hint: it may be helpful to look at the ROC curves for each race.

[80]:
```
# TODO: choose a decision threshold for each race corresponding to a
# decile_score between 1 and 10 such that the FPR and TPR are
# roughly equal for the two races.

caucasian_threshold = 4
# TODO: choose a decision threshold to use for the subset of data with race =␣
 ↪Caucasian.
african_american_threshold = 6
# TODO: choose a decision threshold to use for the subset of data with race =␣
 ↪African-American.

thresh = {'Caucasian': caucasian_threshold, 'African-American':␣
 ↪african_american_threshold}
```

[81]:
```
# Measure the TPR and FPRs for the thresholds you chose.
# Note: no TODOs in this cell, just run it and observe the result.
for race in races:
    fpr = float(summary[(summary['race'] == race) & (summary['decile_score'] ==␣
 ↪thresh[race])]['FPR'])
```

```
    tpr = float(summary[(summary['race'] == race) & (summary['decile_score'] ==␣
 ↪thresh[race])]['TPR'])
    print("for {}, fpr={} and tpr={}".format(race, fpr, tpr))
```

```
for African-American, fpr=0.3143989431968296 and tpr=0.6201083684527393
for Caucasian, fpr=0.3333333333333333 and tpr=0.6228710462287105
```

[82]:
```
# Compute the PPV for the thresholds that you chose. Does equalizing TPR and␣
 ↪FPR also equalize PPV?
# Note: no TODOs in this cell, just run it and observe the result.
for race in races:
    ppv = float(summary[(summary['race'] == race) & (summary['decile_score'] ==␣
 ↪thresh[race])]['PPV'])
    print("for {}, ppv={}".format(race, ppv))
```

```
for African-American, ppv=0.6839309428950863
for Caucasian, ppv=0.5452609158679447
```

1e) Conclusions and implications In this lab, we studied the consequences of creating a binary classifier by applying a decision threshold to a numerical score (COMPAS's `decile_score`). For different decision thresholds, we measured the TPR, FPR, and the PPV. Now, we will take a closer look at what those measurements mean in the context of fairness.

Chapter 2 of Fairness and Machine Learning: Limitations and Opportunities introduces the concepts of **Sufficiency** and **Separation** as possible non-discrimination criteria.

1. Separation says that the classifier decisions are independent of the sensitive attribute (race) conditioned on the label (whether or not recidivism occurred). In other words, for all of the examples where recidivism actually occurred, the probability that the classifier outputs a positive decision should not differ between the races. Which of the metrics we measured today (TPR, FPR, and PPV) best signals whether the classifier achieves separation? Does the classifier we chose in 1d) achieve separation?

   TODO: fill in your answer.

2. Sufficiency says that the label (whether or not recidivism occurred) is independent of the sensitive attribute (race) conditioned on the classifier decisions. In other words, for all of the examples where the classifier outputs a positive decision, the probability of recidivism actually having occurred for those examples should not differ between the races. Which of the metrics we measured today (TPR, FPR, and PPV) best signals whether the classifier achieves sufficiency? Does the classifier we chose in 1d) achieve sufficiency?

   TODO: fill in your answer.

[86]:
```
"TPR & Yes, the classifier we chose in 1d did achieve seperation."
```

[86]:
```
'TPR & Yes, the classifier we chose in 1d achieve seperation.'
```

[87]:
```
"PPV & No, the classifier we chose in 1d did not achieve sufficiency."
```

[87]:
```
'PPV & No, the classifier we chose in 1d did not achieve sufficiency.'
```

[ ]: