Notes for ISL Chapter 2: Statistical Learning

Justin Burruss

2025-08-06

Background

The Python code and notes below are for the ISL study group. This is to try out some of the concepts from Chapter 2 on the `Heart` data set. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

Loading the data set

The Pandas library ("Panda" as in "**Pan**el **da**ta") makes it easy to load the CSV and offers some functionality similar to `data.frame` or `data.table` in R. There are 13 independent variables for only about 300 observations, so let's choose just three independent variables to use in the analysis. For now we treat each as a categorical.

```
import pandas as pd
import numpy as np

# read CSV and tidy up datatypes for the desired Y and Xs

c = pd.read_csv(r"E:\docs\Classes\ISL\heart.csv") \
    .rename(columns={"Unnamed: 0": "Id"}) \
    .dropna() \
    .set_index('Id')

c['AHD'] = c['AHD'].map({"Yes":1, "No":0})
c['Ca'] = c['Ca'].astype('category')
c['Thal'] = c['Thal'].astype('category')
c['ChestPain'] = c['ChestPain'].astype('category')
```

Comparing X and Y

Our dependent variable Y comes from the `AHD` column in the CSV file. For our independent X we selected `Thal`, `Ca`, and `ChestPain`.

First off, if we had no X at all, what's our baseline error rate?

```
print("Error rate with no info is",
      np.round(c['AHD'].value_counts(normalize=True)
               .nsmallest(1).values[0], 3))
```

```
## Error rate with no info is 0.461
```

Now let's find our Bayes error rate per equation 2.11 on page 36. If we had those three variables we selected above, what's the lowest error rate possible?

$$Bayes\ Error\ Rate = 1 - E[max_j Pr(Y = j|X)]$$

An easy way to do this is using `groupby` and a quick `lambda` function. The `groupby` makes it straightforward to construct a table of probabilities `Pr` then apply the Bayes error rate formula.

```
# find our probability of Y given X
Pr = c.groupby(['Thal', 'Ca', 'ChestPain'], observed=False) \
    .agg(AHDCnt=('AHD', 'sum'), Cnt=('AHD', 'count'))
Pr['Pr'] = (Pr['AHDCnt'] / Pr['Cnt']).fillna(0)

# find our "max over j probability" and Bayes error rate
Pr['Mj'] = Pr['Pr'].apply(lambda x: 1-x if x < 0.5 else x)
BER = 1 - ((Pr['Mj'] * Pr['Cnt']).sum() / Pr['Cnt'].sum())

print('Bayes error rate is', np.round(BER, 3))
```

```
## Bayes error rate is 0.148
```

Training and Testing a Classifier

Next, we train and test a classifier. Once again the Pandas library makes it easy: we can use the built-in random sample functionality to do a 70/30 train/test split into training `Tr` and testing `Te`. Then we set aside separate classifier training & testing dataframes.

```
#  do a train/test split
Tr = c.sample(frac=0.7, random_state=2025).index
Te = c.drop(Tr).index

# create separate classifier Training & Testing dataframes
C_Tr = c.loc[Tr, ['Thal', 'Ca', 'ChestPain', 'AHD']]
C_Te = c.loc[Te, ['Thal', 'Ca', 'ChestPain', 'AHD']] \
    .rename(columns={'AHD': 'Actual'})
```

We construct our training table of probabilities `C_Pr_Tr` similar to how we did so with `Pr` previously.

```
C_Pr_Tr = C_Tr \
    .groupby(['Thal', 'Ca', 'ChestPain'], observed=False) \
    .agg(AHDCnt=('AHD', 'sum'), Cnt=('AHD', 'count'))
C_Pr_Tr['P'] = (C_Pr_Tr['AHDCnt'] / C_Pr_Tr['Cnt']).fillna(0)
```

Then we plug in the training probabilities for our observed test X using `merge`.

```
C_Te = C_Te.merge(C_Pr_Tr['P'], left_on=['Thal', 'Ca', 'ChestPain'],
                  right_index=True)
```

It's a binary response, so for our predicted $\hat{Y}$ we just check for probabilities $> 50\%$.

```
C_Te['Predicted'] = C_Te['P'].apply(lambda x: 1 if x > 0.5 else 0)
```

Our error rate is $1 - \frac{1}{n}\sum_{i=1}^{n} I(\hat{y}_i = y_i)$, which with a little luck should fall in between our "no information" error rate and the Bayes error rate (which we recalculate for the test sample).

```
ER = 1 - \
    (C_Te['Predicted'] == C_Te['Actual']).sum() / C_Te['Predicted'].count()

BER_Te = 1 - C_Te.groupby(['Thal','Ca','ChestPain'], observed=True) \
    .agg(Wt=('Actual', 'count'), Mean=('Actual', 'mean')) \
    .apply(lambda x: x.Wt / C_Te['Actual'].count()
           * (1-x.Mean if x.Mean < 0.5 else x.Mean), axis=1) \
    .sum()

print("Test error rates (best possible vs. tested) are:",
      np.round(BER_Te, 3), "vs.", np.round(ER, 3))
```

```
## Test error rates (best possible vs. tested) are: 0.124 vs. 0.202
```

Naive Bayes Classifier

Chapter 4 introduces the Naive Bayes Classifier. Under Naive Bayes we assume that within each class the predictors are independent. If they are independent, we can chain them together and apply Bayes' Theorem to model the probability of seeing dependent Y given independent X as

$$Pr(Y|X) = \frac{Pr(X|Y)Pr(Y)}{Pr(X)}$$

So we must at a minimum calculate $Pr(Y)$, $Pr(X)$, and $Pr(X|Y)$. We want to try out all the concepts from Chapter 2, so let's also find $Pr(\neg Y)$ and $Pr(X|\neg Y)$.

Above formulation is correct, but to reduce the risk of an underflow later we will rewrite using logarithms: "the log of the product is the sum of the logs" and "the log of the quotient is the difference of the logs". We apply log() to both sides to rewrite as

$$\begin{aligned}
\log(Pr(Y|X)) &= \log\left(\frac{P(X|Y)P(Y)}{P(X)}\right) && \text{take log of both sides} \\
&= \log(Pr(X|Y)Pr(Y)) - \log(Pr(X)) && \text{quotient rule of logarithms} \\
&= \log(Pr(X|Y)) + \log(Pr(Y)) - \log(Pr(X)) && \text{product rule of logarithms}
\end{aligned}$$

We'll reuse the same train/test split as before: `C_Tr` has our classifier training data.

First we start with $Pr(X)$; the Pandas library makes it easy to find $Pr(X_1)$, $Pr(X_2)$, and $Pr(X_3)$ using `value_counts(normalize=True)`. We use `np.log()` to calculate the natural log of the probabilities.

```
Thal = C_Tr['Thal'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('Thal')
Thal['LP'] = np.log(Thal['P'])
```

```
Ca = C_Tr['Ca'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('Ca')
Ca['LP'] = np.log(Ca['P'])

ChestPain = C_Tr['ChestPain'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('ChestPain')
ChestPain['LP'] = np.log(ChestPain['P'])
```

Next up are $\log(Pr(X_1|Y))$, $\log(Pr(X_2|Y))$, and $\log(Pr(X_3|Y))$. Same as above but filtering on `AHD==1`.

```
ThalY = C_Tr.loc[C_Tr['AHD']==1, 'Thal'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('Thal')
ThalY['LP'] = np.log(ThalY['P'])

CaY = C_Tr.loc[C_Tr['AHD']==1, 'Ca'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('Ca')
CaY['LP'] = np.log(CaY['P'])

ChestPainY = C_Tr.loc[C_Tr['AHD']==1, 'ChestPain'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('ChestPain')
ChestPainY['LP'] = np.log(ChestPainY['P'])
```

We find $\log(Pr(Y))$ using `value_counts(normalize=True)[1]` to filter on `AHD==1`.

```
LP_Y = np.log(C_Tr['AHD'].value_counts(normalize=True)[1]) # ln(P(Y))
```

Let's build another training table of probabilities `B_Pr_Tr` so we can compare with the `C_Pr_Tr` we constructed earlier. First we create the empty structure using `index=C_Pr_Tr.index`, then we merge the individual $\log(Pr(X_1|Y))$, $\log(Pr(X_2|Y))$, and $\log(Pr(X_3|Y))$ values, and finally sum the logs ("the log of the product equals the sum of the logs") & save that total joint log probability $\log(Pr(X|Y))$ result in our table. Let's display the first few rows of as well just to make sure it looks right.

```
B_Pr_Tr = pd.DataFrame(index=C_Pr_Tr.index) \
    .merge(ThalY['LP'], left_index=True, right_index=True) \
    .merge(CaY['LP'], left_index=True, right_index=True) \
    .merge(ChestPainY['LP'], left_index=True, right_index=True) \
    .assign(LP_X_Y=lambda x: x['LP_x'] + x['LP_y'] + x['LP']) \
    .drop(['LP', 'LP_x', 'LP_y'], axis=1) \
    .rename(columns={'LP_X_Y': 'LP(X|Y)'}) # ln(P(X|Y))

B_Pr_Tr.head(n=3)
```

```
##                           LP(X|Y)
## Thal  Ca  ChestPain
## fixed 0.0 asymptomatic -3.668299
##           nonanginal   -5.337456
##           nontypical   -6.292967
```

Likewise we load the individual $\log(Pr(X_1))$, $\log(Pr(X_2)$, and $\log(Pr(X_3))$ values and sum up & save the total joint log probability $\log(Pr(X))$.

```
B_Pr_Tr = B_Pr_Tr \
    .merge(Thal['LP'], left_index=True, right_index=True) \
    .merge(Ca['LP'], left_index=True, right_index=True) \
    .merge(ChestPain['LP'], left_index=True, right_index=True) \
    .assign(LP_X=lambda x: x['LP_x'] + x['LP_y'] + x['LP']) \
    .drop(['LP', 'LP_x', 'LP_y'], axis=1) \
    .rename(columns={'LP_X': 'LP(X)'}) # ln(P(X))
```

We apply our formula $\log(Pr(Y|X)) = \log(Pr(X|Y)) + \log(Pr(Y)) - \log(Pr(X))$ and check how our table is looking.

```
B_Pr_Tr['LP(Y|X)'] = B_Pr_Tr['LP(X|Y)'] + LP_Y - B_Pr_Tr['LP(X)'] # ln(Pr(Y|X))

B_Pr_Tr.head(n=3)
```

```
##                           LP(X|Y)     LP(X)   LP(Y|X)
## Thal  Ca  ChestPain
## fixed 0.0 asymptomatic -3.668299 -4.040361 -0.443688
##           nonanginal   -5.337456 -4.557943 -1.595262
##           nontypical   -6.292967 -5.051962 -2.056755
```

Let's also calculate and record $\log(Pr(X|\neg Y))$ and $\log(Pr(\neg Y))$. With Y we filtered on `AHD==1`, and now for $\neg Y$ we filter on `AHD==0`. To get $\log(Pr(X|\neg Y))$ we find $\log(Pr(X_1|\neg Y))$, $\log(Pr(X_2|\neg Y))$, and $\log(Pr(X_3|\neg Y))$, then sum them up to get the total joint log probability. For $\log(Pr(\neg Y))$ we use `value_counts(normalize=True)[0]` to filter on `AHD==0`.

```
ThalY0 = C_Tr.loc[C_Tr['AHD']==0, 'Thal'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('Thal')
ThalY0['LP'] = np.log(ThalY0['P'])

CaY0 = C_Tr.loc[C_Tr['AHD']==0, 'Ca'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('Ca')
CaY0['LP'] = np.log(CaY0['P'])

ChestPainY0 = C_Tr.loc[C_Tr['AHD']==0, 'ChestPain'] \
    .value_counts(normalize=True) \
    .reset_index(name='P') \
    .set_index('ChestPain')
ChestPainY0['LP'] = np.log(ChestPainY0['P'])
```

```
LP_Y0 = np.log(C_Tr['AHD'].value_counts(normalize=True)[0]) # ln(P(¬Y))

B_Pr_Tr = B_Pr_Tr \
    .merge(ThalY0['LP'], left_index=True, right_index=True) \
    .merge(CaY0['LP'], left_index=True, right_index=True) \
    .merge(ChestPainY0['LP'], left_index=True, right_index=True) \
    .assign(LP_X_Y0=lambda x: x['LP_x'] + x['LP_y'] + x['LP']) \
    .drop(['LP', 'LP_x', 'LP_y'], axis=1) \
    .rename(columns={'LP_X_Y0': 'LP(X|¬Y)'}) # ln(P(X|¬Y))
```

Bayes Formula works the same with $\neg Y$ as it did above with Y; we use $\log(Pr(\neg Y|X)) = \log(Pr(X|\neg Y)) + \log(Pr(\neg Y)) - \log(Pr(X))$ and save our result.

```
B_Pr_Tr['LP(¬Y|X)'] = \
    B_Pr_Tr['LP(X|¬Y)'] + LP_Y0 - B_Pr_Tr['LP(X)'] # ln(P(¬Y|X))
```

This time let's save our prediction in the table of probabilities. A quick `lambda` sets the prediction based on which outcome is most probable: we check $\log(Pr(Y|X)) \stackrel{?}{>} \log(Pr(\neg Y|X))$ and set the prediction to 1 when $\log(Pr(Y|X))$ is greater, 0 otherwise.

Let's also print our table just to make sure it looks as expected.

```
B_Pr_Tr['Predicted'] = B_Pr_Tr[['LP(Y|X)', 'LP(¬Y|X)']] \
    .apply(lambda x: 1 if x['LP(Y|X)'] > x['LP(¬Y|X)'] else 0, axis=1)

B_Pr_Tr.head(n=3)
```

```
##                          LP(X|Y)     LP(X)  ...  LP(¬Y|X)  Predicted
## Thal  Ca  ChestPain                          ...
## fixed 0.0 asymptomatic -3.668299 -4.040361  ... -1.506007          1
##           nonanginal   -5.337456 -4.557943  ... -0.560980          0
##           nontypical   -6.292967 -5.051962  ... -0.461616          0
##
## [3 rows x 6 columns]
```

Above works just fine, and it's useful because we want to illustrate all the concepts from Chapter 2, but we're doing more work than we need to if all we wanted was to run the above test: we do not need to find $\log(Pr(X))$ if all we wanted to do was check $\log(Pr(Y|X)) \stackrel{?}{>} \log(Pr(\neg Y|X))$. To see why, notice that since $\log(Pr(Y|X)) = \log(Pr(X|Y)) + \log(Pr(Y)) - \log(Pr(X))$ and $\log(Pr(\neg Y|X)) = \log(Pr(X|\neg Y)) + \log(Pr(\neg Y)) - \log(Pr(X))$ we can rewrite our comparison as:

$$\log(Pr(X|Y)) + \log(Pr(Y)) - \log(Pr(X)) \stackrel{?}{>} \log(Pr(X|\neg Y)) + \log(Pr(\neg Y)) - \log(Pr(X))$$

We have $-\log(Pr(X))$ on both sides of our comparison, so really would have just as easily used:

$$\log(Pr(X|Y)) + \log(Pr(Y)) \stackrel{?}{>} \log(Pr(X|\neg Y)) + \log(Pr(\neg Y))$$

Our last step is to plug in the training predictions for our observed test X using `merge` and print the resulting error rate.

```
B_Te = c.loc[Te, ['Thal', 'Ca', 'ChestPain', 'AHD']] \
    .rename(columns={'AHD': 'Actual'}) \
    .merge(B_Pr_Tr[['LP(Y|X)', 'Predicted']],
```

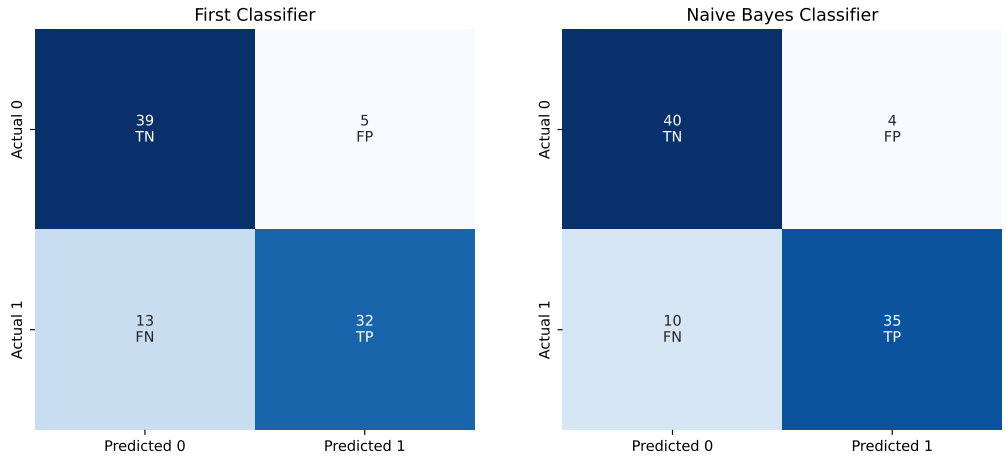
```
          left_on=['Thal', 'Ca', 'ChestPain'],
          right_index=True)

ERB = 1 \
    - (B_Te['Predicted'] == B_Te['Actual']).sum() / B_Te['Predicted'].count()

print("Test error rates (best possible vs. tested NB) are:",
      np.round(BER_Te, 3), "vs.", np.round(ERB, 3))
```

```
## Test error rates (best possible vs. tested NB) are: 0.124 vs. 0.157
```

Comparing Results

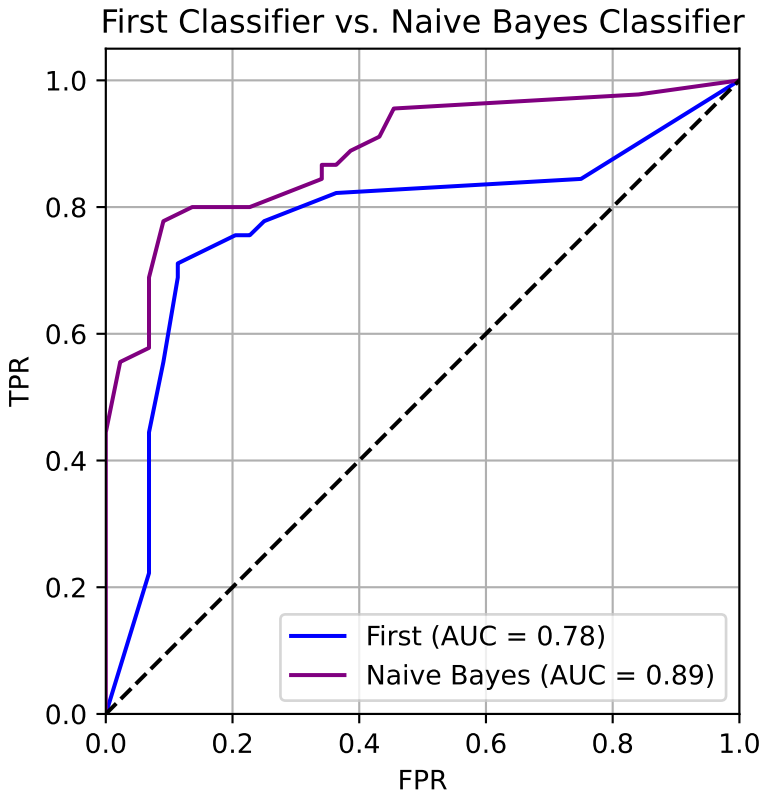
A `seaborn` heatmap makes for a nice comparison of the True Positives (TP), False Positives (FP), and so forth.

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

cm1 = confusion_matrix(C_Te['Actual'], C_Te['Predicted'])
cm2 = confusion_matrix(B_Te['Actual'], B_Te['Predicted'])

def annotate_cm(cm):
    """Retun annotated TN, FP, etc., for use in a heatmap"""
    annotations = np.empty(cm.shape, dtype=object)
    annotations[0, 0] = f'{cm[0, 0]}\nTN'
    annotations[0, 1] = f'{cm[0, 1]}\nFP'
    annotations[1, 0] = f'{cm[1, 0]}\nFN'
    annotations[1, 1] = f'{cm[1, 1]}\nTP'
    return annotations

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

sns.heatmap(cm1, annot=annotate_cm(cm1), fmt='', ax=axes[0],
            cmap='Blues', cbar=False,
            xticklabels=['Predicted 0', 'Predicted 1'],
            yticklabels=['Actual 0', 'Actual 1'])
axes[0].set_title('First Classifier')

sns.heatmap(cm2, annot=annotate_cm(cm2), fmt='', ax=axes[1],
            cmap='Blues', cbar=False,
            xticklabels=['Predicted 0', 'Predicted 1'],
            yticklabels=['Actual 0', 'Actual 1'])
axes[1].set_title('Naive Bayes Classifier')

plt.show()
```



The ROC functionality from `sklearn.metrics` makes for a quick ROC curve and AUC calculation.

```
from sklearn.metrics import roc_curve, auc

FPR, TPR, thresholds = roc_curve(C_Te['Actual'], C_Te['P'])
FPR_b, TPR_b, thresholds_b = roc_curve(B_Te['Actual'], B_Te['LP(Y|X)'])

plt.figure()
plt.plot(FPR, TPR, color='blue',
         label=f'First (AUC = {auc(FPR, TPR):.2f})')
plt.plot(FPR_b, TPR_b, color='purple',
         label=f'Naive Bayes (AUC = {auc(FPR_b, TPR_b):.2f})')
plt.plot([0,1], [0,1], color='black', linestyle='--')
plt.xlim([0.0, 1.0]);
plt.ylim([0.0, 1.05]);
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.legend(loc='lower right')
plt.title('First Classifier vs. Naive Bayes Classifier')
plt.grid()
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```



K-Nearest Neighbors & Our Variables Reexamined

Chapter 2 also includes the method of K-Nearest Neighbors (KNN) when introducing the concept of unsupervised learning. Let's implement KNN; first though we'll need to reexamine our independent variables.

A Second Look at `Thal`, `Ca`, and `ChestPain`

For the supervised learning examples above we cavalierly chose to treat each as a categorical, but for KNN we will need to decide on distance metrics (for the "N" in "KNN"), so we ought to examine what each variable really represents.

`Thal` Short for thallium. Some of the top search engine hits for the heart dataset claim that `Thal` is for "thalassemia": this is completely wrong. `Thal` refers to an imaging procedure done using thallium-201 ("exercise myocardial scintigraphy"). The category `fixed` means thallium uptake is reduced both at rest *and* during exercise, meaning the defect is likely some kind of permanent damage. The `reversible` category means normal thallium uptake at rest but restricted ("ischemic") during exercise. The `normal` category means normal update both at rest and exercise.

`Ca` Short for Calcium. This is actually a count of the number of major vessels (0-3) colored by flourosopy, i.e., that have coronary calcium. Lower is better from a heart health standpoint: we don't want to see a buildup of calcium in our heart vessels.

`ChestPain` The `typical` category means the pain meets all three criteria for *angina pectoris*, `nontypical` means two out of three criteria are met, `nonanginal` means there is still some pain but only one or zero criteria are met, and `asymptomatic` means no chest pain.

For `Thal` it's not clear that `reversible` and `fixed` would really belong in the same vector from a distance point of view, so let's decompose it into two binary variables: `ThalShowsReversible` and `ThalShowsFixed` where a `normal` result will zero out both variables.

The `Ca` really does appear to be one vector, but how should we scale it? Let's not use equal spacing: probably zero ought to be farther from the other values than they are to each other. We might use `np.emath.logn(4, Ca+1)` so that 1 is a little farther from 2 than 2 is from 3, or we could use `.4*np.sign(Ca)+Ca*(.2)` so that the distances from 1 to 3 are equal. Let's go with `np.emath.logn(4, Ca+1)`. Normalization is baked into that formula: we'll get values from 0-1.

Our `ChestPain` variable does look to be one vector with an ordinal relation. Similar to `Ca` it looks as if our zero (no symptoms) ought to be farther away. Let's again go with `np.emath.logn(4, ChestPain+1)` with values `typical=3`, `nontypical=2`, `nonanginal=1`, `asymptomatic=0`.

K-Nearest Neighbors

We first copy and map the variables over into a new DataFrame `D` and print the first few rows to make sure it looks OK.

```
D = c[['Ca', 'ChestPain', 'AHD']].assign(
    ThalShowsReversible=lambda x: (c['Thal'] == 'reversable').astype(float),
    ThalShowsFixed=lambda x: (c['Thal'] == 'fixed').astype(float),
    Ca=lambda x: np.emath.logn(4, x['Ca'].astype(float) + 1),
    ChestPain=lambda x: np.emath.logn(4, x['ChestPain'] \
                                      .map({"typical": 3,
                                            "nontypical": 2,
                                            "nonanginal": 1,
                                            "asymptomatic": 0}) \
                                      .astype(float) + 1))

D.head(n=3)
```

```
##           Ca  ChestPain  AHD  ThalShowsReversible  ThalShowsFixed
## Id
## 1   0.000000        1.0    0                  0.0             1.0
## 2   1.000000        0.0    1                  0.0             0.0
## 3   0.792481        0.0    1                  1.0             0.0
```

We reconfigured out independent variables somewhat, so let's measure our new Bayes Error Rate $1 - E[max_j Pr(Y = j|X)]$. It *should* be unchanged assuming we did not inadvertently drop information with this configuration.

```
PrD = D.groupby(['ThalShowsReversible', 'ThalShowsFixed', 'Ca', 'ChestPain'],
                observed=False).agg(AHDCnt=('AHD', 'sum'),
                                    Cnt=('AHD', 'count'))

PrD['Pr'] = (PrD['AHDCnt'] / PrD['Cnt']).fillna(0)
PrD['Mj'] = PrD['Pr'].apply(lambda x: 1-x if x < 0.5 else x)
BERD = 1 - ((PrD['Mj'] * PrD['Cnt']).sum() / PrD['Cnt'].sum())

print("Bayes Error Rate before & after is", np.round(BER, 3),
      "&", np.round(BERD, 3))
```

```
## Bayes Error Rate before & after is 0.148 & 0.148
```

On to the algorithm. Chapter 2 gives us the following equation.

$$Pr(Y = j|X = x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

For some test observation x_0, we look at the K closest neighbors $\mathcal{N}_0$. To find the closest neighbors we need to define some kind of distance metric for the heart data. For example, we might choose Euclidean, or we could simply add up the "city block" style distance. Let's go with Euclidean. Keeping our ground rules in mind, we build our distance matrix `DM` with just basic Python + Pandas + NumPy. We may as well make it a DataFrame.

```
DM = pd.DataFrame(((
    D.to_numpy()[:, np.newaxis] - D.to_numpy()[np.newaxis, :]) ** 2)
    .sum(axis=2) ** 0.5, index=c.index, columns=c.index)
```

Note that `**` is the power operator in Python. The built-in function `pow()` can also be used for exponentiation.

Let's retain the same train/test split as above. We build a table `DM_Te` of distances *from* our testing samples (rows) *to* our training samples (columns).

```
DM_Te = DM.loc[Te, Tr]
```

Now we find the K closest neighbors $\mathcal{N}_0$ for each x test observation. For K We can pick anything from 1 up to the number of training samples minus one (207). Let's pick an odd number for K just to avoid ties. We'll make it $K = 51$. Then we pick the most common occurrence via `mode()`.

We save to a new table of test data `D_Te`.

```
D_Te = pd.DataFrame(DM_Te
                    .apply(lambda x: x.nsmallest(51).index, axis=1)
                    .apply(lambda x: c.loc[x, 'AHD'].mode())) \
                    .rename(columns={0: 'Predicted'}) \
                    .merge(c['AHD'], left_index=True, right_index=True) \
                    .rename(columns={'AHD':'Actual'})
```

Calculate and display the error rate.

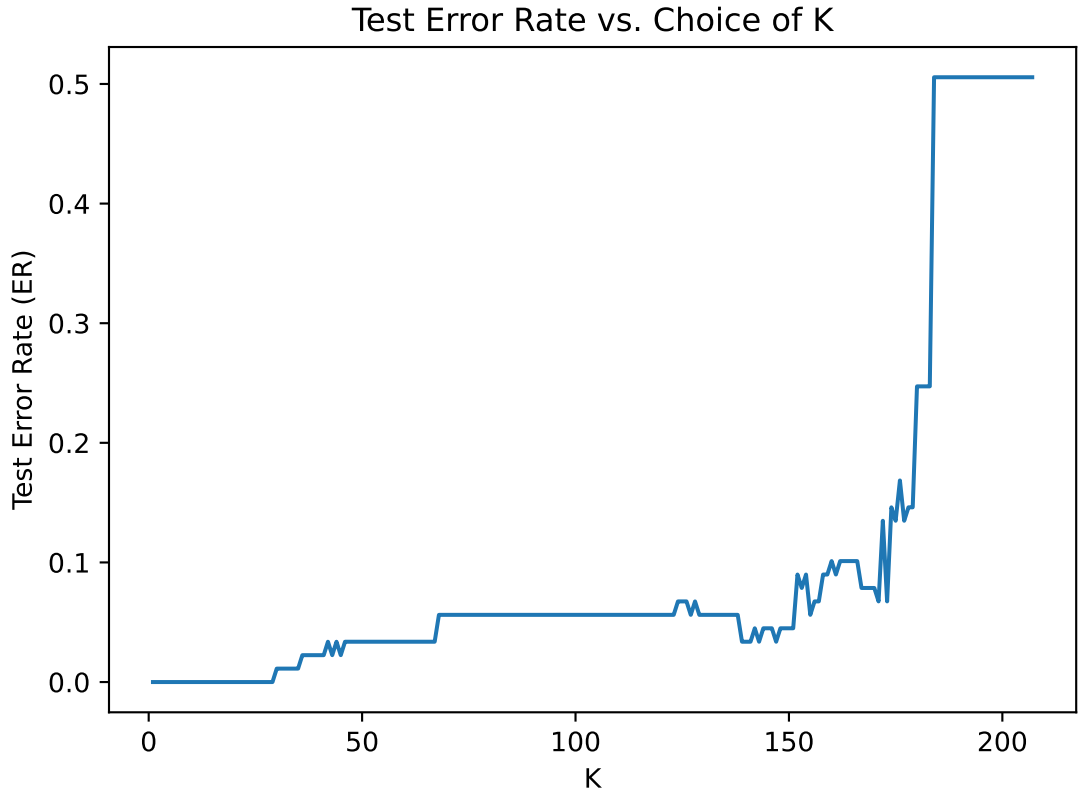
```
ERK = 1 \
    - (D_Te['Predicted'] == D_Te['Actual']).sum() / D_Te['Predicted'].count()

print("Test error rate for K=51 is:", np.round(ERK, 3))
```

```
## Test error rate for K=51 is: 0.034
```

What would have been our best possible choice for K for the test sample? We can apply the above logic to all the values from 1 to 207 and save to another DataFrame `TeKn`. Print the best choice and plot all choices.

```
def te_k_pred(k):
    """Calculate and return training KNN predictions given K"""
    return(DM_Te
           .apply(lambda x: x.nsmallest(k).index, axis=1)
           .apply(lambda x: c.loc[x, 'AHD'].mode())
```

```
            .rename(columns={0: 'Predicted'}))

TeKn = pd.DataFrame({'K': range(1, 208)})
TeKn['ER'] = TeKn['K'] \
    .apply(lambda k: abs(te_k_pred(k)['Predicted'] - D_Te['Actual']).sum()
           / D_Te['Actual'].count())

plt.figure()
plt.plot(TeKn['K'], TeKn['ER'])
plt.xlabel('K')
plt.ylabel('Test Error Rate (ER)')
plt.title('Test Error Rate vs. Choice of K')
plt.show()
```




```
print("The best choice for K and resulting test error rate would have been\n",
      TeKn.nsmallest(1, 'ER').to_string(index=False))
```

```
## The best choice for K and resulting test error rate would have been
##   K  ER
##  1 0.0
```

No surprise that our "just pick something" value of 51 was not the optimal choice.

Using Scikit-learn

Let's validate the "hand-coded" Naive Bayes & KNN classifiers above against the popular Scikit-learn implementations. We'll see if we get the same answer both ways. The relevant implementations are `CategoricalNB` and `KNeighborsClassifier`.

```
from sklearn.naive_bayes import CategoricalNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
```

Naive Bayes Classifier from Scikit-learn

First we need to format the data for `CategoricalNB`. We copy the relevant columns of our heart data just to make sure we don't clobber what we have going on above, then use `factorize` to tidy up the data types for `CategoricalNB`.

```
c2 = c[['Thal', 'Ca', 'ChestPain', 'AHD']].copy()
Th = c2['Thal'].factorize()
Ca = c2['Ca'].factorize()
Cp = c2['ChestPain'].factorize()
Hd = c2['AHD'].factorize()
c2['Thal'] = Th[0]
c2['Ca'] = Ca[0]
c2['ChestPain'] = Cp[0]
c2['AHD'] = Hd[0]
```

We'll fit the classifier using the same training sample as before.

```
slnb = CategoricalNB(alpha=0, force_alpha=True, fit_prior=True)
slnb.fit(c2[['Thal', 'Ca', 'ChestPain']].drop(Te),
         c2['AHD'].drop(Te));
```

We'll reuse the same testing sample as before.

```
SLB_Te= c2[['Thal', 'Ca', 'ChestPain']].drop(Tr)
```

Testing the model is a one-liner.

```
SLB_Te['Predicted'] = slnb.predict(SLB_Te[['Thal','Ca','ChestPain']])
```

Now let's compare our predictions from `CategoricalNB` in `SLB_Te` against our predictions from the hand-code Naive Bayes Classifier in `B_Te`.

```
print("For the NB Classifier, let's compare hand-coded with Scikit-learn\n",
      pd.merge(B_Te['Predicted'], SLB_Te['Predicted'],
               left_index=True, right_index=True, suffixes=('_hc', "_sl"))
      .groupby(['Predicted_hc', 'Predicted_sl'])
      .agg(Cnt=('Predicted_hc', 'count')))
```

```
## For the NB Classifier, let's compare hand-coded with Scikit-learn
##                              Cnt
## Predicted_hc Predicted_sl
## 0            0               50
## 1            1               39
```

No differences.

KNN Classifier from Scikit-learn

Presence of ties could lead to different outcomes. Per the `KNeighborsClassifier` documentation:

> Warning Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor k+1 and k, have identical distances but different labels, the results will depend on the ordering of the training data.

Our Bayes Error Rate is non-zero, so we must have some duplicates. How many in our population before applying the train/test split?

```
print("Percent duplicates is",
      np.round((D.shape[0] - D[['Ca', 'ChestPain', 'ThalShowsReversible',
                               'ThalShowsFixed']].drop_duplicates()
                .shape[0])/D.shape[0], 3))
```

```
## Percent duplicates is 0.889
```

We have lots of duplicates.

We can compare, but if we see differences it may be due to ties.

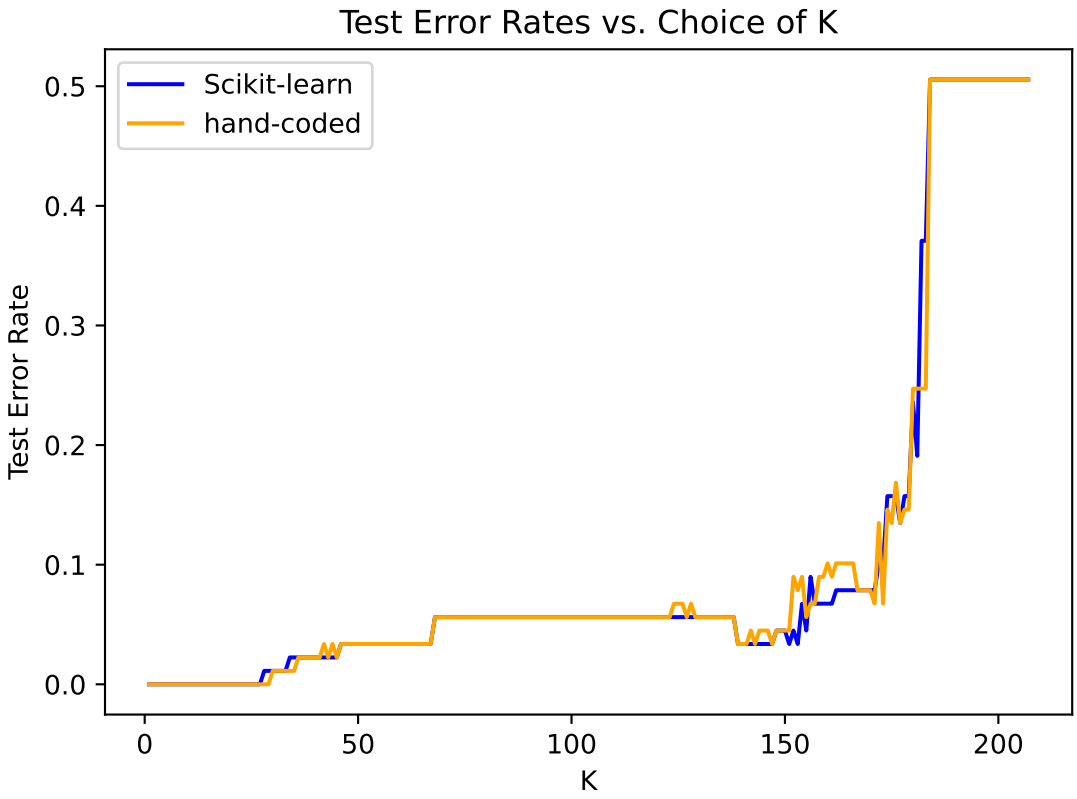
First thing is to set up the same training samples distances, taking care to reuse the same train/test split as before.

```
DM_Tr = DM.loc[Tr, Tr]
```

Then we build and fit the model using our already-computed distances. Let's go ahead and run it on all 207 choices for K and plot the two methods together. It the lines are close then *probably* the implementation is correct.

```
E_slkn = []

for k in range(1, 208):
    slkn = KNeighborsClassifier(n_neighbors=k, metric='precomputed')
    x = slkn.fit(DM_Tr, D.loc[Tr, 'AHD'].values)
    y_pred = slkn.predict(DM_Te)
    accuracy = accuracy_score(D.loc[Te, 'AHD'].values, y_pred)
    E_slkn.append(1-accuracy)

plt.figure()
plt.plot(range(1, 208), E_slkn, color='blue', label='Scikit-learn')
plt.plot(TeKn['K'], TeKn['ER'], color='orange', label='hand-coded')
plt.xlabel('K')
```

```
plt.ylabel('Test Error Rate')
plt.title('Test Error Rates vs. Choice of K')
plt.legend(loc='upper left')
plt.show()
```



Ultimately we cannot validate due to the presence of ties; however, the two models came out almost identical based on the plot above, so it's likely that the "hand-coded" implementation is correct.

Pandas DataFrame Methods & Attributes

A Pandas DataFrame is a 2-dimensional labeled data structure consisting of a dictionary of Series objects. Columns are implemented using one Series for each column. Each Series is a 1-dimensional labeled array. Rows are implemented via an index shared by the DataFrame and each Series. Pandas was built using NumPy, so it shares many of the same concepts and data types.

`agg` Do more than one aggregate, e.g., both a sum and a count; chain this after a `groupby`.

`apply` Apply a function.

`argsort` Method is only for for Index or Series, not DataFrame; returns the integer indicies that would sort the Index or Series. Use `kind='stable'` to make it more deterministic.

`assign` Return a copy of the DataFrame with new and changed columns.

`astype` Casts to one of the Pandas/NumPy dtypes in a wrapper sense; the actual data may remain numpy.float64, etc., even as the *column* dtype changes to CategoricalDtype.

`columns` Attribute: the columns of the DataFrame (which may be a `pandas.core.indexes.base.Index`, etc.).

`count` Counts the non-NA elements.

`dropna` Use `inplace=True` to drop NAs directly, omit if chaining.

`factorize` A top-level method or method for an Index or Series, not DataFrame; returns the factors plus the levels, e.g. `codes, uniques = pd.factorize(...)`.

`fillna` Fill both NA and NAN values.

`groupby` Like `group by` in SQL or `by` in `data.table`; chain with aggregate functions. Use `observed=False` to include the zero-count categoricals.

`iloc` Like `loc` but with integer indexing to select by position.

`index` Attribute: the index of the DataFrame (which may be a `pandas.core.indexes.base.Index`, `pandas.core.indexes.multi.MultiIndex`, etc.).

`isin` Returns True/False for whether each element is in the list of values.

`loc` For accessing just some of the row and columns of a DataFrame by label or Booleans.

`map` Pass this a dictionary or a function to map values elementwise.

`mean` Arithmetic mean. Default behavior is to skip NAs, use skipna=False to NA out when NAs are present.

`merge` Merge with a SQL style join on columns or indices.

`mode` Mode; default behavior is to skip NAs, use skipna=False to NA out when NAs are present

`nsmallest` Return the n smallest in ascending order.

`rename` Use `inplace=True` to rename directly, omit if chaining.

`read_csv` Read a Comma-Separated Values (CSV) file. Put an "r" in front of the filename string to keep the backslashes from turning into escape characters.

`reindex` Return a copy of the DataFrame with the index changed, e.g., to reorder the columns or add new columns. Match to the index of another DataFrame by passing it as an argument, e.g., `.reindex(myOtherDF.index)`.

`reset_index` Resets the index; turns a Series that came out of a `groupby` back into a DataFrame.

`sample` Randomly sample, e.g., `.sample(frac=0.7)` to sample 70%.

`sum` Default behavior is to skip NAs, use skipna=False to NA out when NAs are present.

`to_numpy` Return the DataFrame elements as `numpy.ndarray` values.

`to_string` Use when printing to adjust how the DataFrame is represented.

`value_counts` Return a Series with frequency counts; use `normalize=True` to return percentages instead.