INFO371 Problem Set: Distance Metric and k-NN

April 16, 2022

Introduction

This problem set revolves about distance and metrics, although most of your effort will probably go on coding. It is a brand now one, hope it is doable :-)

Please submit a) your code *and* b) the results in a final output form (html or pdf). You are free to choose either R or python for solving this problem set.

1 Feature normalization in k-NN (60pt)

In the first problem use use Wisconsin Breast Cancer Dataset (WBCD). The dataset orginates from UCI Machine Learning Repository. Better to use the version on canvas, *wdbc.csv.bz2*.

The dataset is downloaded from UCI ML Repository and it contains 569 cases from November 1995. The variables are

The data includes tumor diagnosis, with "M" meaning cancer (*malignant*) and "B" no cancer (*benign*), and 10 features, describing physical properties of cell nuclei from biopsy samples. Each feature is represented three times, once for mean, once for standard deviation, and once for the worst values. More specifically, the variables are

id case id

Diagnosis (M = malignant, B = benign). This is what you normally predict.

ten real-valued features computed for each cell nucleus. For each feature the mean, standard error, and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 3 is Mean Radius, field 13 is Radius SE, field 23 is Worst Radius.

It contains the following features:

radius mean of distances from center to points on the perimeter

texture standard deviation of gray-scale values

perimeter

area

smoothness local variation in radius lengths

compactness $perimeter^2/\texttt{area} - 1.0$

concavity severity of concave portions of the contour

concpoints number of concave portions of the contour

symmetry

fracdim fractal dimension, "coastline approximation"−1

Your task is to predict diagnosis based on this data, and our focus in to use k-NN and different metrics.

1.1 Explore data (6pt)

As the first step, explore the data.

1. (1pt) Load the data. Ignore *id*.
2. (2pt) Create a table where you show ranges (min and max) for each variable. You may include more statistics you consider useful.
3. (3pt) What would be the accuracy of the naive model, such model that predicts every case to the majority category?

1.2 Feature transformation (54pt)

Your next task is to compare k-NN with normalized/non-normalized features, and by using Mahalanobis metric (equivalent to Mahalanobis transformation). You will visualize the results by plotting the decision boundary and compare the goodness using cross-validation. If you are uncertain about what is decision boundary, I recommend to consult **?** book Section 2. For instance, Figure 2.13 on p38 depicts a 2D classification case where certain X_1, X_2 values are classified as orange and others as blue. Decision boundary is the dashed winding line that separates these two regions. Feature normalization and Mahalanobis distance are described in Lecture Notes, Ch 3.2 (page 118 for now).

You have first to fit your model. Thereafter you have to predict the classes (here cancer/no cancer) on a regular dense grid that covers the parameter space (these are the small blue/orange dots on figure 2.13). Afterwards you can plot your predicted values with a certain color code. So, in order to make a nice plot:

i) Fit your model using only two features (you cannot easily display more on a figure).

ii) Predict the diagnosis on a grid (at least 100×100) that covers the range of the explanatory variables. This gives you 100x100 predicted diagnoses.

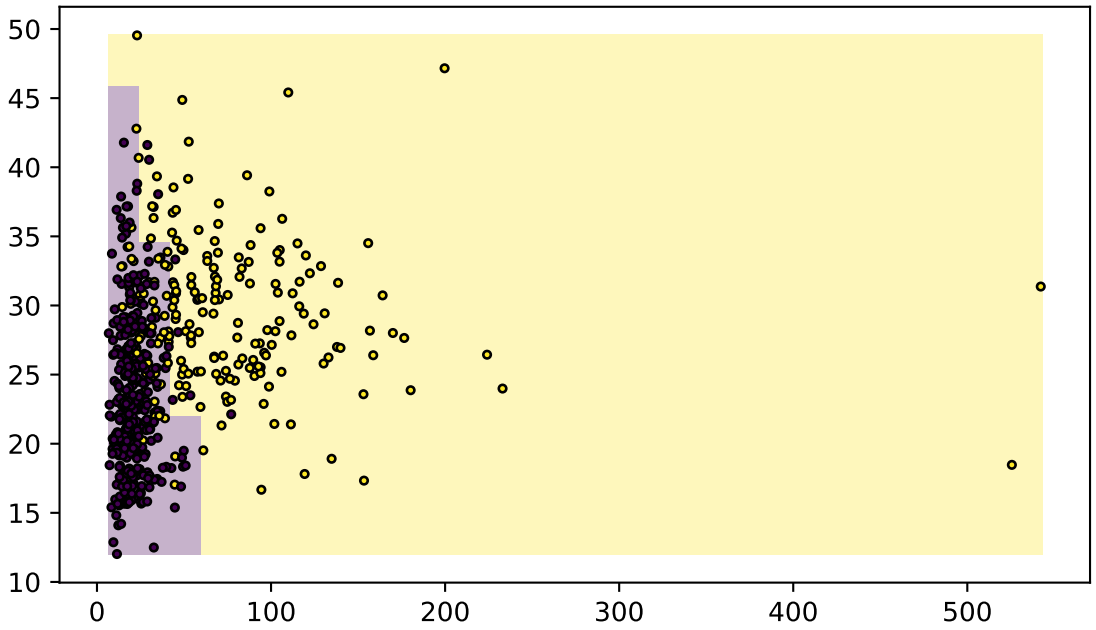
iii) Plot the predicted grid values and actual data on the same plot. Ensure that actual observations and predictions are clearly distinguishable, and that one can easily understand the color code.

Example code for 30×30 grid using logistic regression is below:

```
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt

x1 = wdbc["area.se"]
x2 = wdbc["texture.worst"]
y = wdbc.diagnosis == "M"
X = np.column_stack((x1, x2))
m = LogisticRegression(solver="lbfgs").fit(X, y)
# Now greate the regular grid
ex1 = np.linspace(x1.min(), x1.max(), 30)  # 30 elements
ex2 = np.linspace(x2.min(), x2.max(), 30)
xx1, xx2 = np.meshgrid(ex1, ex2)
# meshgrid creates two 30x30 matrices
g = np.column_stack((xx1.ravel(), xx2.ravel()))
# we create the design matrix by stacking the xx1, xx2
# after unraveling those into columns
# predict on the grid
hatY = m.predict(g).reshape(30, 30)
# imshow wants a matrix, so we reshape the predicted
```

```
# vector into one
_ = plt.imshow(hatY, extent=(x1.min(), x1.max(), x2.min(), x2.max()),
           aspect="auto",  # let the image fit to screen
           interpolation='none', origin='lower',
           # you need to specify that the image begins from below,
           # not above, otherwise it will be flipped around
           alpha=0.3)
_ = plt.scatter(x1, x2, c=y, edgecolor='k', s=8)
_ = plt.show()
```



This is not a good plot, but you can see that the boundary between purple and yellow predictions broadly agrees with the location of purple and yellow dots. No worries, you can make it better!

The tasks below work best if you write a function that takes in k and feature names, and creates the decision boundary plot and cross-validated accuracy.

1. (12pt) Pick features *texture.mean* and *concpoints.mean* and do the following:
   (a) Use these two to model diagnosis with k-NN where k = 7 using original features (i.e. not normalized features).
   (b) (6pt) Plot the corresponding decision boundary (use at least 100 × 100 matrix). Explain why do you see apparently arbitrary stripes on the image (vertical stripes in case you put texture on x-axis).
   (c) (2pt) Cross-validate your prediction accuracy.
2. (5pt) Now normalize these two features and repeat the previous steps.
3. (10pt) Explain why does the decision boundary plot look different now. Comment on the accuracy: was accuracy better with original features or normalized features?
4. (10pt) now use the original (non-normalized) features and repeat the process. However, this time use Mahalanobis distance.

5. (7pt) Repeat all these steps with a few other sets of two features and a few other k-s of your choice.

6. (10pt) Describe your observations. How well do original/normalized/Mahalanobis-transformed features work? Does it depend on the features? Does the visual impression from images align with the cross-validation results?

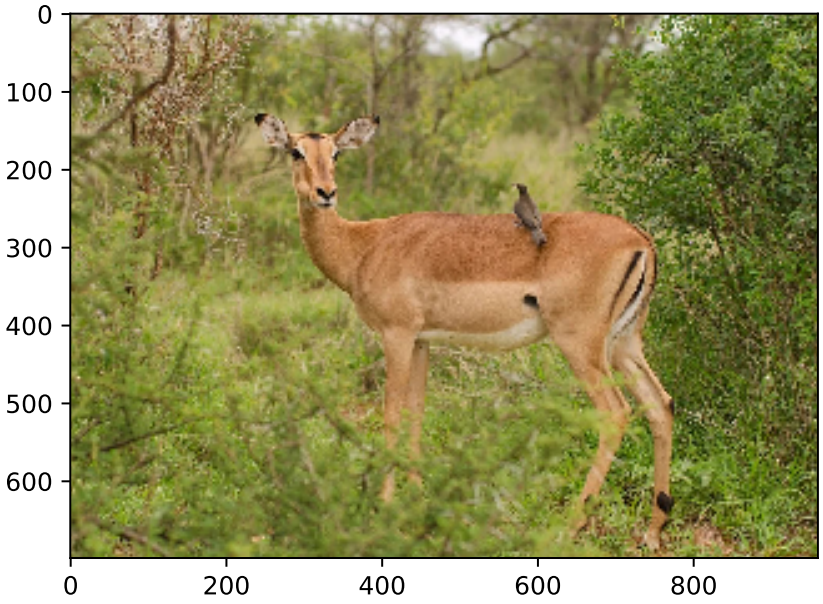
---

2 Images: Reduce number of colors (40pt)

1. (8pt) Get a color image you like, load it as matrix, and display it using matplotlib. You may use the code along these lines:

```
import matplotlib.pyplot as plt
from matplotlib.image import imread

pixels = imread("img/impala.jpg")
ax = plt.subplot(1,1,1)
ax.set_aspect("equal")
_ = plt.imshow(pixels)
_ = plt.show()
```



Note: imread works with *jpg* images, it may not work with certain other formats. Select a small image to speed up the computations (e.g. 200×300). You may replace it with a large version later when everything works, if you wish (not needed!).

Next, let's convert your image down to 16 colors only. We use clustering to group the colors and convert each pixel to its cluster center, i.e. one of the 16 colors.

See the code snippet here:

```
pixels.shape
# convert the 'pixels' matrix into Nx3 (or Nx4) matrix
# where N is the number of pixels, and 3 or 4 is the number
# of color channels
# so each row is a pixel, and the row data is its color value
```

```
## (699, 960, 3)
```

```
M = pixels.reshape((-1,pixels.shape[2]))
print(M.shape)
# 3 (or 4) columns, one for each color channel
```

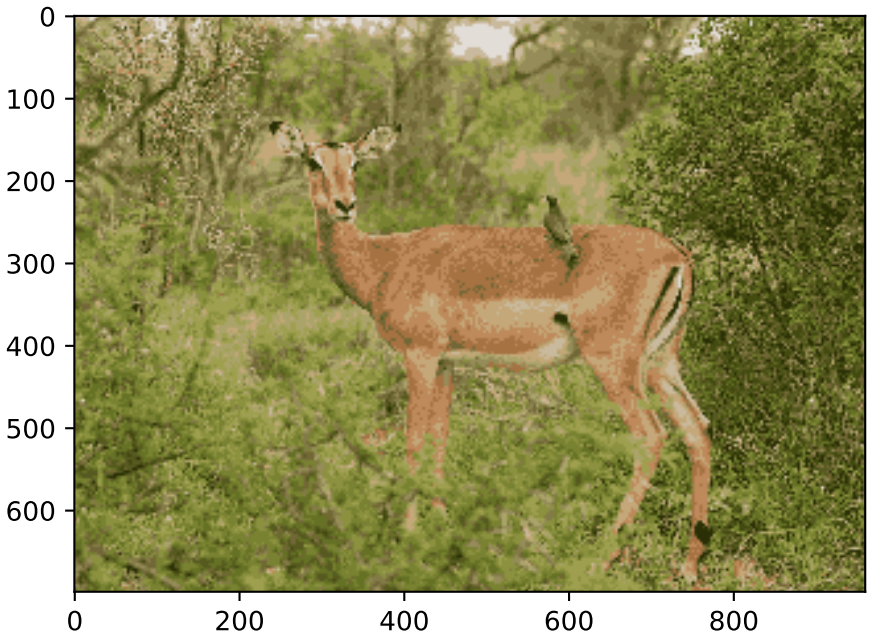
```
## (671040, 3)
```

```
from sklearn.cluster import KMeans

m = KMeans(16).fit(M)  # cluster the color values
cl = m.predict(M)  # find the cluster id for each pixel
centers = m.cluster_centers_  # find the cluster centers (i.e. colors)
print("center color values:\n", centers)
```

```
## center color values:
##  [[167.80836669 114.47089672  66.84755805]
##  [120.93002467 130.09952767  49.78606979]
##  [ 87.50027963  90.54179925  32.3785363 ]
##  [159.3271217  160.8291537   79.38992526]
##  [201.71628895 186.7368272  158.56565156]
##  [131.06205292 123.57764371  71.23068172]
##  [197.83701023 168.13350118 121.62341463]
##  [101.33542132 113.14335826  36.16974404]
##  [ 66.08828199  74.82737906  16.28670477]
##  [190.56134612 138.61481954  91.40713533]
##  [149.97296129 140.30899027  89.31257073]
##  [ 43.16752119  47.11355906   6.37296309]
##  [226.62492314 221.77208444 215.41975815]
##  [171.8453769  164.22874226 106.08018364]
##  [139.91184987 145.77570987  63.90087152]
##  [114.42115074 103.93644784  54.04729418]]
```

```
compressed = centers[cl]/255  # normalize to between 0 and 1
compressed = compressed.reshape(pixels.shape)
# make it back to NxM array
ax = plt.subplot(1,1,1)
ax.set_aspect("equal")
_ = plt.imshow(compressed)
_ = plt.show()
```



2. (24pt) Get this code to work with your image and understand what it does. Explain:

   (a) What does shape of *pixels* mean
   (b) What is matrix M? Why does it have 671040 rows and 3 columns in this example?
   (c) What do *cl* values tell us? E.g. if `cl[12345] = 2`, what does it mean?
   (d) What are the columns of "center color values" printed above? E.g. the first row, second column is value 163.37. What does that number mean?
   (e) How many "center color values" are there? Why do we have this number?

3. (8pt) Next repeat the above by reducing the number of colors to 2, and 4. Show the pictures!

   Hint: make a function that takes the number of colors as an argument, then you don't have to copy-paste your code that much.
