CSCI 141 - Spring 2019
Assignment 5: Cancer Classification using Machine Learning
Due Date: See Canvas

1 Overview

The goal of this project is to gain more practice with using functions, lists and dictionaries and gain some intuition for Machine Learning, the field of computer science concerned with writing algorithms that allow computes to "learn" from data.

The problem we'll be solving is as follows: Given a data file containing hundreds of patient records with values describing measurements of cancer tumors and whether or not each tumor is malignant or benign, develop a simple rule-based classifier that can be used to predict whether an as-yet-unseen tumor is malignant or benign.

The general idea is that malignant tumors are different than benign tumors. Malignant tumors tend to have larger radii, to be more smooth, to be more symmetric, etc. Measurements have been taken on many tumors whose class (malignant or benign) is known. The code you are going to write will get the average score across all the malignant tumors for an attribute (e.g. 'area') as well as the average score for that attribute for benign tumors. Let's say that the average area for malignant tumors is 100, and for benign tumors is 50. We can then use that information to try to predict whether a given tumor is malignant or benign.

Imagine you are presented with a new tumor and told the area was 99. All else being equal, we would have reason to think this tumor is more likely to be malignant than had its area been 51. Based on this intuition, we are going to create a simple classification scheme. We will calculate the midpoint between the malignant average and the benign average (75 in our hypothetical example), and simply say that for each new tumor, if its value for that attribute is greater than or equal to the midpoint value for that attribute, that is one vote for the tumor being malignant. Each attribute that we are using produces a vote, and at the end of counting votes for each attribute, if the malignant votes are greater than or equal to the benign votes, we predict that the tumor is malignant.

2 Machine Learning Framework

"Machine learning" is a popular buzzword that might evoke computer brain simulations, or robots walking among humans. In reality (for now, anyway), machine learning refers to something less fanciful: algorithms that use previously observed data to make predictions about new data. It may sound less glamorous than fully sentient robots, but that's exactly what was described above! You can get more sophisticated about the specifics of how you go about this, but that's the core of what machine learning really means.

If using data to make predictions on new data is our goal, you might think it makes sense to use all the data we have to learn from. But in fact, if we truly don't know the labels (e.g., malignant or benign) of the data we're testing our algorithm on, we won't have any idea whether it's doing a good job! For this reason, it makes sense to split the data we have labels for into a *training* *set*, which we'll use to "learn" from, and a *test set*, which we'll use to evaluate how well the algorithm does on new data (i.e., data it wasn't trained on). We will take about 80% of the data as our training set, and use the remaining 20% as our test set.

2.1 Training Phase

Here's how our classifier will work: In the training phase, we will "learn" (read: compute) the average value each attribute (e.g. area, smoothness, etc.) among the malignant tumors. We will also "learn" (again: compute) the average value of each attribute among benign tumors. Then we'll compute the midpoint for each attribute. This collection of midpoints, one for each attribute, is our classifier.

2.2 Testing Phase

Having trained our classifier, we can now use it to make an educated guess about the label of a new tumor if we have the measurements of all of its attributes. Our educated guess will be pretty simple:

- If the tumor's value for an attribute is greater than or equal to the midpoint value for that attribute, cast one vote for the tumor being malignant.
- If the tumor's attribute value is less than the midpoint, cast one vote for the tumor being benign.
- Tally up the votes cast according to these rules for each of the ten attributes. If the malignant votes are greater than or equal to the benign votes, we predict that the tumor is malignant.

If we want to use this classifier to diagnose people, we have an important question to answer: how good are our guesses? To answer this question, we'll run test our algorithm on the 20% of our data that we held out as the test set, which we *didn't* use to train the classifier, but we *do* know the correct labels. Our rate of accuracy on these data should be indicative of how well our classifier will do on new, unlabeled tumors.

3 Dataset Description

You have been provided with `cancertTrainingData.txt`, a text file containing the 80% of the data that we'll use as our training set.

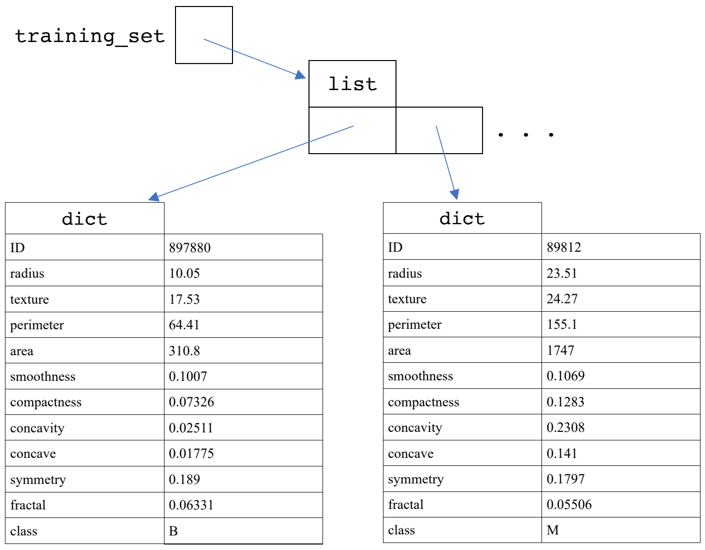
The file has many numbers per patient record, some of which refer to attributes of the tumor. The skeleton code includes the function `make_training_set()`, which reads in the important information from this file and produces a list of dictionaries. Each dictionary contains attributes for a single tumor as follows:

0. ID

1. radius
2. texture
3. perimeter
4. area
5. smoothness
6. compactness
7. concavity
8. concave
9. symmetry
10. fractal
11. class

The middle 10 attributes (numbered 1 through 10) are the numbers that describe the tumor. The first attribute is just the patient ID number, and the last attribute is the actual real life state of the tumor, namely, malignant (represented by “M”) or benign (represented by “B”).

We don’t need to know what these attributes mean: all we need to know is that they are measurements of the tumors, and that benign and malignant tumors tend to have different attribute values. For these 10 tumor attributes when comparing to the midpoint values, higher numbers indicate malignancy. Pictorially, the list of dictionaries looks like this (two are shown, but the list contains many more than that):



ID	897880
radius	10.05
texture	17.53
perimeter	64.41
area	310.8
smoothness	0.1007
compactness	0.07326
concavity	0.02511
concave	0.01775
symmetry	0.189
fractal	0.06331
class	B

ID	89812
radius	23.51
texture	24.27
perimeter	155.1
area	1747
smoothness	0.1069
compactness	0.1283
concavity	0.2308
concave	0.141
symmetry	0.1797
fractal	0.05506
class	M

Figure 1: Illustration of the data layout of the training set returned by `make_training_set`

The dictionary stored in the 0th spot in the list gives the attributes for the 0th tumor: `training_set[0]["class"]` gives the true class label (in this case, ”B” for benign) of the 0th tumor.

4 Getting Started

Download the skeleton code (`cancer_classifier.py`), training set (`cancerTrainingData.txt`), and the test set (`cancerTestingData.txt`). Make sure all three files are in the same directory, or the main program will not be able to load the data from the files.

5 Tasks

5.0 Overview

Training and evaluating our classifier involves several steps. The first task, which has been done for you, is to write code to load the training and test data sets from text files into lists of dictionaries representing patient records, as described in the previous section. The functions `make_training_set` and `make_test_set` are included in the skeleton code to complete these steps.

You will complete the following four tasks:

- TODO 1: Train the classifier
- TODO 2: Apply the classifier to the test set
- TODO 3: Calculate and report accuracy on the test set
- TODO 4: Provide classifier details on user-specified patients

The main program has been provided to you: you will be implementing functions that are called from the main program at the bottom of the skeleton code file. Take a moment to read through and understand the main program (notice that the parts of the program that use TODOs 1–4 are commented out).

Each of the above steps is described in detail in the remainder of this section. After you finish each TODO (2 and 3 are completed together), uncomment the corresponding block in the main program and run your code to make sure your output matches the sample output provided below.

5.1 TODO 1: Train the classifier

A classifier is simply some model of a problem that allows us to make predictions about new records. We use the training set to build up a simple model, as described in Section 2:

- For all malignant records, calculate the average value of each attribute.
- For all benign records, calculate the average value of each attribute.
- Calculate the midpoint between these averages for each attribute.

Our classifier is a single dictionary that stores this midpoint value for each attribute.

Implement this functionality in `train_classifier`. My solution for this part totals roughly 30 lines of code. As always, you may find it useful to write helper methods that perform smaller tasks: for example, you could create a helper function to initialize a dictionary with each of the attributes as keys and 0 as values.

When done, uncomment the block of code in the main program that calls `train_classifier` and debug your code until your attribute midpoints match the sample output.

5.2 TODO 2: Apply the classifier

After computing the classifier (namely, the dictionary of attribute midpoints), we can use these values to make predictions given the attribute values of a new patient. A record is classified as follows:

For each attribute, determine whether the record's value is less than or equal to the classifier's midpoint value. If so, cast one vote for Benign; otherwise, cast one vote for Malignant. If the votes for Malignant are greater than or equal to the votes for Benign, the record is classified as Malignant; otherwise, it is classified as Benign.

Implement this classification scheme in the `classify` function, applying it to each record in the test set. Notice that the prediction for a record is to be stored in the `"prediction"` field of the dictionary for that record.

5.3 TODO 3: Report accuracy

For each record in the test set, compare the predicted class to the actual class. Print out the percentage of records that were labeled correctly (i.e., the predicted class is the same as the true class).

5.4 TODO 4: Provide patient details

The final task is to provide a user the opportunity to examine the details of the predictions made for individual patients. Implement `check_patients`, which contains commented pseudocode describing its the exact behavior. You are **strongly** encouraged to write helper functions that are called from within this function: if a pseudocode step requires more than a few lines of code, consider making a helper function to accomplish that step.

If the user-specified patient ID is found in the test set, print a table with four columns:

- Attribute: the name of the attribute
- Patient: the patient's value for that attribute
- Classifier: the classifier's threshold (midpoint) for that attribute
- Vote: the vote cast by the classifier on for that attribute

See the sample output for specifics of what the table should look like.

Printing a table of results with nice even columns and uniformly formatted decimal numbers requires delving into the details of string formatting in Python. There are multiple ways to do this, but the following tips should be sufficient[1]:

- String objects have `rjust` and `ljust` methods, which return a copy of the string padded to the given width, justified either right or left. For example, `"abc".rjust(5)` returns `"  abc"`.
- Floating point numbers can be formatted nicely using the `format` method, which is called on strings containing special formatting specifiers. For example, `"{:.2f}".format(8.632)` formats the argument (8.632) as a float with 2 decimal places, resulting in the string `"8.63"`.

Your table does not need to match the sample output character for character, but your columns should be lined up, right justified, and floating-point values should be printed with the decimals aligned and a consistent number of digits following the decimal point.

6 Sample Output

A sample run of my solution program is shown below. User input is bolded.

```
Reading in training data...
Done reading training data.
Reading in test data...
Done reading test data.

Training classifier...
Classifier cutoffs:
    radius: 14.545393772893773
    texture: 19.279093406593404
    perimeter: 94.91928571428579
    area: 693.337728937729
    smoothness: 0.09783294871794869
    compactness: 0.1104729532967033
    concavity: 0.09963735815018318
    concave: 0.054678068681318664
    symmetry: 0.18456510989010982
    fractal: 0.06286657967032966
Done training classifier.

Making predictions and reporting accuracy
Classifier accuracy: 92.20779220779221
```

[1]For much more detail on string formatting, see the Python Tutorial entry: `https://docs.python.org/3/tutorial/inputoutput.html`

```
Done classifying.

Enter a patient ID to see classification details: 897880
       Attribute     Patient  Classifier        Vote
          radius     10.0500     14.5454      Benign
         texture     17.5300     19.2791      Benign
       perimeter     64.4100     94.9193      Benign
            area    310.8000    693.3377      Benign
      smoothness      0.1007      0.0978   Malignant
     compactness      0.0733      0.1105      Benign
       concavity      0.0251      0.0996      Benign
         concave      0.0177      0.0547      Benign
        symmetry      0.1890      0.1846   Malignant
         fractal      0.0633      0.0629   Malignant
Classifier's diagnosis: Benign
Enter a patient ID to see classification details: 89812
       Attribute     Patient  Classifier        Vote
          radius     23.5100     14.5454   Malignant
         texture     24.2700     19.2791   Malignant
       perimeter    155.1000     94.9193   Malignant
            area   1747.0000    693.3377   Malignant
      smoothness      0.1069      0.0978   Malignant
     compactness      0.1283      0.1105   Malignant
       concavity      0.2308      0.0996   Malignant
         concave      0.1410      0.0547   Malignant
        symmetry      0.1797      0.1846      Benign
         fractal      0.0551      0.0629      Benign
Classifier's diagnosis: Malignant
Enter a patient ID to see classification details: quit
```

7 Hints and Guidelines

- Start by reading through the skeleton code, and making sure you know what the main program does and how the functions you are tasked with implementing fit into the overall program.
- If your understanding of lists and dictionaries is shaky, you will have great difficulty making progress. Visit my office hours, TA office hours, or mentor hours **early** so you don't spend too much time struggling.
- The top of the skeleton file has a global variable called `ATTRS`, which is a list of the attribute names each patient record has. Using global variables with all-caps names is a common convention when you have variables that need to be referenced all over your program and (crucially) **never change value**. You may refer to `ATTRS` from anywhere in your program, including inside function definitions, without passing it in as a parameter.
- As in A4, all variables (other than `ATTRS`) referenced from within functions must be local variables - if you need access to information from outside the function, it must be passed into the function as a parameter.

- When iterating over patient record dictionaries, use loops over the keys stored in ATTRS rather than looping directly over the dictionary's keys. An example of this appears in the main program where the classifier cutoffs are printed.
- The functions provided in the skeleton code include headers and specifications. Make sure you follow the given specifications (and don't modify them!).
- Keep the length of each function short: if you're writing a function that takes more than about 30 lines of code (not including comments and whitespace), consider how you might break the task into smaller pieces and implement each piece using a helper function.
- All helper functions you write must have docstrings with precise, clearly written specifications.
- Test each function after you've written it by running the main program with the corresponding code block uncommented. Don't move on until the corresponding portion of the output matches the sample.

Submission

Upload `cancer_classifier.py` to Canvas and fill in the A5 Hours quiz with an estimate of how many hours you spent working on this assignment.

Rubric

Submission Mechanics (2 points)	
File called `cancer_classifier.py` is submitted to Canvas	2
Code Style and Clarity (28 points)	
Comment at the top with author/date/description	3
Comments throughout code clarify any nontrivial code sections	5
Variable and function names are descriptive	5
Helper functions are used to keep functions no longer than about 30 lines of code (not counting comments and blank lines)	5
ATTRS is used to iterate over dictionary attributes	5
No global variables except ATTRS are referenced from within functions	5
Correctness (70 points)	
The trained classifier has the correct midpoint values for each attribute	30
Prediction is performed as described using the midpoints computed in training	5
Accuracy is computed and reported correctly as shown in the demo output	10
User is repeatedly prompted for Patient ID	5
Message is printed if given ID is not in the test set.	5
If ID is in the test set, table is printed with all four columns and rows for all 10 attributes	10
Table columns are right-justified and aligned	3
Floating-point values in the table are lined up on the decimal point and have a fixed number of digits after the decimal.	2
Total	100 points

Acknowledgements

This assignment was adapted from a version used by Perry Fizzano, who adapted it from an original assignment developed at Michigan State University for their CSE 231 course.

8 Challenge Problem

The following challenge problem is worth up to 10 points of extra credit. As usual, this can be done using only material we've learned in class, but it's much more open-ended. If you are trying to tackle this, feel free to come talk to me about it in office hours.

In this assignment, you trained a simple classifier that used means over the entire training set to classify unseen examples. This simple classifier does quite well (92% accuracy) on the test set. There are many more sophisticated methods for learning classifiers from training data, some of which depend on some pretty heavy-duty mathematical derivations.

One type of classifier that doesn't require a lot of math but nonetheless performs pretty well on a lot of real-world problems is called the **Nearest Neighbor Classifier** or its more general cousin, the **K Nearest Neighbors Classifier**. The idea behind KNN is that records with similar attributes should have similar labels. So a reasonable way to guess the label of a previously-unseen record is to find the record from the training set that is most similar to it and guess that record's label.

To implement a nearest neighbor classifier, we need some definition of what it means to be "near". One of the the simplest choices for numerical data like ours is the Sum of Squared Differences metric. Given two records, compute the difference between the two records' values, square the difference, and add up all the squared differences over all 10 attributes. The smaller the SSD metric, the more similar the two records are.

(up to 5 points) Implement a nearest neighbor classifier using the SSD metric in a file called `KNN.py`. Feel free to copy and re-use the data loading functions, and any other functions that remain relevant, from `cancer_classifier.py`. Evaluate your classifier's accuracy like we did in the base assignment. Write a comment in `KNN.py` reporting your classifier's performance.

You might notice an issue with the SSD metric applied to our dataset: some attributes have huge values (e.g., in the hundreds) and others have tiny values. When computing SSD, the large-valued attributes will dominate the SSD score, even if they aren't the most important attributes. Come up with a way to modify the distance metric so that it weights attributes evenly. Describe your approach and compare the performance of your new metric with SSD in a comment in `KNN.py`.

(up to 5 points) The nearest neighbor classifier can be a bit fiddly, because the guess depends on a single datapoint in your training set, so a single unusual (or mislabeled!) training record could cause a wrong prediction. A more robust classifier looks not just at the single nearest neighbor, but each of K nearest neighbors, for some choice of K. Generalize your nearest neighbor classifier to a KNN classifier. Try out different values of K and include a comment discussing the classification accuracy for values of K. Do any of them beat the base assignment classifier's performance?