Machine Learning

Two broad categories of machine learning: supervised learning and unsupervised learning:

Supervised Learning: Supervised learning problems are problems for which we have some labeled data available to us that we can use to train our learning algorithm (fit our model). That is, we have labeled examples of data similar to what we will ultimately use to make predictions. The first headshot example above is an example of supervised learning since we have an initial set of labeled head shots to work from. Two common types of supervised learning problems are classification and regression. In classification problems the labels we are trying to predict are drawn from a finite set of classes (Example: Given the length and weight of an animal predict the type of animal from some finite set of categories like fish or horse). In regression problems we will try to predict a continuous number (Example: Given the age of a bear and the time of year, predict its weight).

Unsupervised Learning: In unsupervised learning problems we are not given any labeled training data to start with. One common type of unsupervised learning problem is clustering. Given some set of (unlabeled) data, we determine if the data separates into two or more clusters (clusters are groups of data that may be similar in some obvious or subtle way). Many problems in natural language processing are also unsupervised learning problems including the famous cocktail party problem where we try to isolate a single voice or conversation in a noisy environment.

K Nearest Neighbors: The first classifier we will use is called K-nearest neighbor (Knn) and the way it works is we plot the labeled data, which we will call training data, and then when given a new unlabeled observation we plot it and look at its nearest neighbors among the training data. That is we look at the nearest k neighbors and let their labels determine our predicted label. So we will label the new observation malignant if a majority of its k nearest neighbors are labeled malignant and we label it benign otherwise.

Cross Validation: More generally we would like to know how well we can expect our classifier to work in the future. To this end we will partition the data into five separate cells (mutually exclusive similar sized subsets of the data). We will use four of these cells as labeled training

data and for the other cell (so 20% of the data) we will pretend we don't actually know the labels and will use the classifier to predict them. We call this last cell the test set. Since we actually do know the labels of the test set we can measure our classifier's empirical performance on the test set. We don't just do this once, we do it five times using each of the cells as the test set once while using the other four cells as training data. So each observation will be used as training data four times and in the test set once. We call this process 5-fold cross-validation and more generally K-fold cross-validation. Remember, what we're doing here is very meta. We are estimating the performance of our classifier. The hope is that future data looks like the labeled data set we start with so if we observe our performance on that, we have evidence to suggest that future performance will be similar.

Numpy

Consider the following code:

```
import numpy as np
x = np.ones(3)
print(x.shape)
x = x.reshape((3,1))
print(x.shape)
```

One outputs (3,) and the other (3, 1) the difference is the first is a row vector and the second is column vector.

We can convert Python lists to numpy arrays using np.array:

```
l=[1,2,3]
a=np.array(l)
```

There are 4 important properties of numpy arrays: **dtype**, **ndim**, **shape**, **size**

Numpy arrays are more mathematical than python lists, 2 * x = [2. 2. 2.] rather than [1,1,1,1,1,1] which normal lists would do.

Pandas

We import pandas as follows:

import pandas as pd

We do not need to assign an alias we just usually do

A single column in pandas is a series and multiple series is known as a dataframe

tickets=pd.read_csv('tiny_parking.csv') This is how we read in csv files

tickets.head(x) or tickets.tail(x) provides the first or last x entries, if x is not provided the default is 5.

We can reassign the index: tickets=tickets.set_index('Plate')

We have two primary locate methods: .iloc[] and .loc the former using the normal 0 to n-1 indexing and the latter using the specified index.

There functions to be used on series like sum() some like describe() can be used on entire dataFrame objects.

New Python Stuff

Default Parameters: def g(a,b=12,c=0): b and c have default values in the event nothing is passed into it when calling g

Variable Length Positional Parameters: def f(a,b=0,*args): all additional arguments are packaged into a tuple called args, the * gives this away, we are still allowed to specify more arguments afterwards but these would have to be *keyword only* otherwise how would we know what is to be included in args

Keyword Only Parameters: def g(a,b=0,*args,c=1,d=2): c and d are keyword only due to the presence of *args

Variable Length Keyword Parameters: def g(a,b=0,*args,c=1,d=2, **kwargs): kwargs is a dictionary that will contain a map from the keyword to its value that it was passed in

Positional Only Parameters: def f(x,/,y,z=0): all declared parameters before the / are positional only. Any attempt to call f with x=... in it will fail

Probability and Call Options

Two key random variables to know about:

Bernoulli: Returns either 0 or 1 based solely on success probability p: expected value is p

Binomial: Based on two parameters: n and p, you can think of a binomial as running n bernoulli with success probability p. From this we get the expected value of np.

Old Python Stuff

Object References - always draw these out whenever you are asked a question about them consider the following example:

```
l = [1, [1,2], 3]
b = l[:]
b[1][0] = 7
```

What do you expect the values to be? Normal assumption may lead to the following:

```
l => [1, [1,2], 3]
b => [1, [7,2], 3]
```

But rather the truth is this:

```
l => [1, [7,2], 3]
b => [1, [7,2], 3]
```

This is because when you choose to have a list as an element the actual element being stored is a reference to it which gets copied over when doing l[:] so altering a value within one internal list will affect the other.

Looping Over Elements:

The Midterm question a lot of people seemed to hate:

```
def foo(a):
  b = []
  for item in a:
     c = 0
     for elem in a:
       if item == elem:
          c += 1
     b.append(c)
  return b
```

Then we execute the function with the following call: foo('Halloween')

Here is a break down of the call to the function:

1. We create an initially empty list called b
2. We begin iterating through each letter of a so item will take on the values within the following collection:
   item <- ['H','a','l','l','o','w','e','e','n']
3. We then start a variable c with the value 0
4. We begin iterating through each letter of a again so elem will take on the values within the following collection:
   elem <- ['H','a','l','l','o','w','e','e','n']
5. If item is the same as elem then add 1 to c
6. After the inner loop completes append c to b
7. Go back to 2 but start with the next letter until you run out.
8. Return b

From this we can figure out that every element will match at least once with itself. So we can basically count all duplicates of each element + 1 so: b = [1,1,2,2,1,1,2,2,1]

Old Definitions:

Algorithm: A well-ordered sequence of operations that produce a result and halt in a finite amount of time.

Computing: The activity of using or creating algorithmic processes to complete some task.

Computer Science: The study of algorithms including their mathematical properties, their linguistic realizations, their physical realizations, and their applications.