Applying K Nearest Neighbors to Data

Classification, which is a supervised form of machine learning, and explained the K Nearest Neighbors algorithm intuition. In this tutorial, we're actually going to apply a simple example of the algorithm using Scikit-Learn, and then in the subsquent lectures we'll build our own algorithm to learn more about how it works under the hood.

To exemplify classification, we're going to use a Breast Cancer Dataset, which is a dataset donated to the University of California, Irvine (UCI) collection from the University of Wisconsin-Madison. UCI has a large Machine Learning Repository. The datasets here are organized by types of machine learning often used for them, data types, attribute types, topic areas, and a few others. Very useful both for educational uses, as well as for machine learning algorithm development. I find myself coming back here frequently, it's definitely worth a bookmark. From the Breast Cancer Dataset page, choose the Data Folder link. From there, grab `breast-cancer-wisconsin.data` and `breast-cancer-wisconsin.names`. These may not download, but instead display in browser. Right click to save as if this is the case for you.

After downloading, go ahead and open the `breast-cancer-wisconsin.names` file. Looking at this file, scrolling down to just after line 100, we get the names of the attributes (columns). With this information, we're going to just manually add these labels to the `breast-cancer-wisconsin.data` file. Open that, and enter a new first line: `id,clump_thickness,uniform_cell_size, uniform_cell_shape,marginal_adhesion, single_epi_cell_size,bare_nuclei,bland_chromation, normal_nucleoli,mitoses,class`. Right out of the gate, you should be thinking what our features will be and what our label will be. We're attempting to classify things, so it should be obvious that the classes are going to be that the list of attributes leads to a benign or malignant tumor. Also, most of these columns appear to be of use, but are there any that are similar to the others or maybe useless? Absolutely, this ID column is not something we actually want to feed into the classifier.

Missing/bad data: This dataset also has some missing data in it, which we're going to need to clean! Let's start off with our imports, pulling in the data, and some cleaning:

```
import numpy as np
from sklearn import preprocessing, cross_validation, neighbors
import pandas as pd

df = pd.read_csv('breast-cancer-wisconsin.data.txt')
df.replace('?',-99999, inplace=True)
df.drop(['id'], 1, inplace=True)
```

After reading in the data, we take note that there are some columns with missing data. These columns have a "?" filled in. The `.names` file informed us of this, but we would have discovered this eventually via an error if we attempted to feed this

information through to a classifier. In this case, we're choosing to fill in a -99,999 value for any missing data. You can choose how you want to handle missing data, but, in the real world, you may find that 50% or more of your rows contain missing data in one of the columns, especially if you are collecting data with extensive attributes. -99999 isn't perfect, but it works well enough. Next, we're dropping the ID column. When we are done, we'll comment out the dropping of the id column just to see what sort of impact it might have to include it.

Next, we define our features (X) and labels (y):

```
X = np.array(df.drop(['class'], 1))
y = np.array(df['class'])
```

The features `X` are everything except for the class. Doing `df.drop` returns a new dataframe with our chosen column(s) dropped. The labels, `y`, are just the class column.

Now we create training and testing samples, using Scikit-Learn's `cross_validation.train_test_split`:

```
X_train, X_test, y_train, y_test =
cross_validation.train_test_split(X, y, test_size=0.2)
```

Define the classifier:

```
clf = neighbors.KNeighborsClassifier()
```

In this case, we're using the K Nearest Neighbors classifier from Sklearn.

Train the classifier:

```
clf.fit(X_train, y_train)
```

Test:

```
accuracy = clf.score(X_test, y_test)
print(accuracy)
```

The result should be about 95%, and that's out of the box without any tweaking. Very cool! Just for show, let's show what happens when we do indeed include truly meaningless and misleading data by commenting out the dropping of the id column:

```
import numpy as np
from sklearn import preprocessing, cross_validation, neighbors
import pandas as pd

df = pd.read_csv('breast-cancer-wisconsin.data.txt')
df.replace('?',-99999, inplace=True)
```

```
#df.drop(['id'], 1, inplace=True)

X = np.array(df.drop(['class'], 1))
y = np.array(df['class'])

X_train, X_test, y_train, y_test =
cross_validation.train_test_split(X, y, test_size=0.2)

clf = neighbors.KNeighborsClassifier()
clf.fit(X_train, y_train)
accuracy = clf.score(X_test, y_test)
print(accuracy)
```

The impact is staggering, where accuracy drops from ~95% to ~60% on average. In the future, when AI rules the planet, note that you just need to feed it meaningless attributes to outsmart it! Interestingly enough, adding noise can be a way to help or hurt your algorithm. When combatting your robot overlords, being able to distinguish between helpful noise and malicious noise may save your life!

Next, you can probably guess how we'll be predicting if you followed from the regression tutorial that used Scikit-Learn. First, we need some sample data. We can just make it up. For example, I will look at one of the lines in the sample file, and make something similar, merely shifting some of the values. You can also just add noise to do further testing, provided the standard deviation is not outrageous. Doing this is relatively safe as well, since you're not actually training on the falsified data, you're merely testing. I will just manually do this by making up a line:

```
example_measures = np.array([4,2,1,1,1,2,3,2,1])
```

Feel free to search the document for that list of features. It doesn't exist. Now you can do:

```
prediction = clf.predict(example_measures)
print(prediction)
```

...or depending on when you are watching this, you might not be able to! When doing that, I get a warning:

```
DeprecationWarning: Passing 1d arrays as data is deprecated in
0.17 and will raise ValueError in 0.19. Reshape your data
either using X.reshape(-1, 1) if your data has a single
feature or X.reshape(1, -1) if it contains a single sample.
```

Okay, no problem. Do we have a single feature? Nope. Do we have a single example? Yes! So we will use `X.reshape(1, -1)`:

```
example_measures = np.array([4,2,1,1,1,2,3,2,1])
example_measures = example_measures.reshape(1, -1)
prediction = clf.predict(example_measures)
```

```
print(prediction)
```

Output:

```
0.95
[2]
```

The output here is first the accuracy (95%) and then the prediction (2), which is what we modeled our fake data to be.

What if we had two samples?

```
example_measures =
np.array([[4,2,1,1,1,2,3,2,1],[4,2,1,1,1,2,3,2,1]])
example_measures = example_measures.reshape(2, -1)
prediction = clf.predict(example_measures)
print(prediction)
```

Darn this hard-coding. What if we don't know how many samples!?!

```
example_measures =
np.array([[4,2,1,1,1,2,3,2,1],[4,2,1,1,1,2,3,2,1]])
example_measures =
example_measures.reshape(len(example_measures), -1)
prediction = clf.predict(example_measures)
print(prediction)
```

As you can see, implementing K Nearest Neighbors is not only easy, it's extremely accurate in this case. In the next lecture, we're going to build our own K Nearest Neighbors algorithm from scratch, rather than using Scikit-Learn, in attempt to learn more about the algorithm, understanding how it works, and, most importantly, one of its pitfalls.