Overview of the CCDataSet data structure

Manuchehr Aminian

April 4, 2018

What would we like to do?

Broadly what we're trying to do is to identify biomarkers which play significant roles in the early stages of infectious disease, in one way or another. Studies related to this may or may not measure the same variables, be conducted exactly the same way, or have the same species of subjects. They may or may not have additional attached to them - when and where the study was done, the sex of the subjects, etc, etc.

Here's a list of some of things we would like to automate cleanly:

1. Control for batch effects across multiple studies.
2. Perform feature selection based on one or more attributes.
3. Perform various classification tasks based on one or more attributes (not necessarily the same ones as above).
4. Identify important biomarkers common across multiple studies.

Here, we use the word "attributes" broadly. These could be real-valued, binary, string, etc. If they're going to be used somewhere along the data analysis pipeline, they probably need to be finite and countable. If they're going to be added on as extra features, they just need to be able to be concatenated to the original features.

Now, here's how we've structured the data structure to handle these tasks so far. There are four basic classes which enable the functionality. These may be subject to change; there are advantages and disadvantages to what I've done. Anyway, they are:

1. `CCDataSet`: The parent class, holding all the data, metadata, and functions for slicing and handling the data however we wish;
2. `CCList`: A subclassed Python list. This is a minor modification of a usual list where the display is more compact.
3. `CCDataPoint`: A subclassed `numpy.ndarray`. This can be anything in principle, but it's assumed this is an array of experimentally measured values. For the moment it's assumed all `CCDataPoint()`s have the same dimensions but this isn't strictly necessary.
4. `CCDataAttr`: A class for each attribute for each data point. This keeps track of the value and type of the attribute, whether or not it's from the original data or derived from some kind of preprocessing, etc.

These are the building blocks. Now into detail:

The typical workflow for loading

To this point, the way I've been using this is on a per-dataset basis. The basic order of operations in the loading script is:

1. `ccd = CCDataSet()` (initialize an empty dataset);
2. Manually create a (python) list of attributes you know the data have. For example, `attrnames=['SubjectID','sex','StudyID'`
3. Manually create a second list with plaintext descriptions of these attributes. For example, `attrdescrs=['Subject ID for the mouse','Sex of the mouse (M or F)','The identifier for when/where the study was performed']`. This is non-essential, so you *could* use empty strings, but don't do that.
4. Do a call to `ccd.add_attrs(attrnames,attrdescrs)` to tell `ccd` what to expect below;

5. Create empty lists `dpoints=[]` and `dattrvals=[]` (or whatever you want to call them);
6. Do a for-loop, `.append()`ing data and the attributes for each sample/subject/etc; (this is the bulk of the work)
7. Do a call to `ccd.add_datapoints(dpoints,attrnames,dattrvals)` to populate the thing.

After this point, you can use the functions built in to `ccd` to slice up the data however you like, depending on the attributes given.

The data structure

CCDataSet

The parent class is a "CCDataSet" (Calcom dataset) has the following stuff in it.

(Python) Attributes:

- `self.data` : A `CCList()` (a subclassed list with a minor tweak) containing zero or more `CCDataPoint()`s
- `self.attrnames` : A Python list of strings of all attributes the data points in the `CCDataSet` have
- `self.attrs` : A Python dictionary which maps entries in `self.attrnames` to corresponding `CCDataAttr()`.

Functions:

- Data loading and i/o:
  - `self.generate_id()`: Generates a unique IDs for new datapoints.
  - `self.is_numeric()`: Checks if an input is of type int, float, or np.double.
  - `self.add_attrs()`: Initializes a set of new `CCDataAttr()`s in self.attrs given a list of names and descriptions.
  - `self.append_attr()`: Adds a single new `CCDataAttr()` and applies it to all the data points in the list (needs fixing).
  - `self.add_datapoints()`: Adds `CCDataPoint()`s to self.data with the given attributes and attribute values.
  - `self.save_to_disk()`: **DOESN'T WORK PROPERLY.** Intention is to save the entire structure to hard drive, data, attributes, and all.
- Data slicing and label generation:
  - `self.generate_labels()`: Generates a list of integer labels based on some attribute. Also outputs a dictionary mapping the labels to their original attribute values.
  - `self.labels_to_attrnames()`: maps the labels to their original attribute values (based on `self.labels_to_attrnames()`).
  - `self.generate_data_matrix()`: Takes data points in self.data and generates a raw data matrix (type numpy.ndarray). Capable of selecting a subset of the data and a subset of the features.
  - `self.find_attr_by_value()`: Gives a list of pointers to data points which have a given value of a given attribute.
  - `self.find_attrs_by_values()`: Gives a list of pointers to data points which match all the given criteria (attrname/attrvalue pairs).
  - `self.get_attr_values()`: Gives a list of attribute *values* associated with the given attribute.
  - `self.generate_relaxed_attr()`: Not fully implemented. Creates a new `CCDataAttr()` for each data point based on an existing `CCDataAttr()` which is expected to be real-valued.
  - `self.attr_value_equivclasses()`: Returns the equivalence classes of a given attribute; that is, a list of attribute values (similar to above) and a dictionary which maps these attribute values to lists of pointers associated with that attribute value.

CCDataPoint

The `CCDataPoint`s are intended to be just that - data points. What that means depends on context. Currently this is implemented as a subclassed `numpy.ndarray`, so it doesn't necessarily need to be a vector; it can be an array as well.

- Everything that `numpy.ndarray`s do, and...
- `self.set_attrs()`: Applies a new set of attributes with the given names and values on to the `CCDataPoint`. For example, `d` is a `CCDataPoint`, we might have `CCDataAttr`s inside it, such as `d.SubjectID`, `d.sex`, etc.

CCDataAttr

The `CCDataAttr`s are intended to be "smart" in the sense that they don't only store the value, but some extra data relevant to the kinds of operations we might want to do with that attribute. Why would we want this? Some attributes might be real-valued (time since exposure); others might be discrete strings (the CC line, the hospital of the study, etc). We might also have attributes that were *calculated* based on the data point (e.g., Fourier coefficients, coarsened labels, etc) and we want to keep in mind that they weren't part of the original data set. For these and a few other reasons, the `CCDataAttr` is a simple class set up as:

- `self.name` : A string indicating the attribute's name (possibly a little redundant)
- `self.value` : The value of the attribute itself. This can be anything - string, float, integer, list, array, etc.
- `self.is_derived` : Whether the attribute is "derived" from the original data in some way – that is, it's not a value, but it's come from the result of some transform/preprocessing/etc.
- `self.type` : Should correspond to `type(self.value)`.
- `self.is_numeric` : Whether the value for this kind of attribute can be embedded on the real line, with the implications about total ordering, intervals, etc. Important if (for example) we have an attribute with discrete values of 0, 1, 2, ... but we don't actually want to imply that ordering has any meaning.
- `self.long_description` : A string which describes the attribute in plain English.

Things that need to be implemented

These are colored by expected difficulty (red is hardest).

- Important:
  - ■ **!!!** Setting up as automated a pipeline as possible going from raw GEO datasets to `CCDataSet`s. Mmanu has some code and documentation in his SVN folder which talks about this.
  - ■ *Done.* Keeping track of variable names on load of a single dataset.
  - ■ *Done, using HDF – just needs to be maintained as more things are added to the data structure.* Saving the dataset to disk properly using pickle files and/or HDF. Both require some extra coding to do correctly save/load because of the multi-level data structure. Pickle files are easier to load in principle, but still require some unpacking when `pickle.load` is called because the default behavior of pickle doesn't work for complicated classes.
  - ■ *Done.* Ability to calculate and store feature subsets and access them by name when generating data matrices (probably done using a dictionary).
  - ■ *Done.* Functionality to do leave-one-out and other cross-validation techniques based on an attribute, e.g., `SubjectID`, `StudyID`, etc. Related to the end-to-end pipeline. These functions probably shouldn't be within the dataset, but could take the dataset as an input. Unclear if it would be a switch on the existing `Experiment` class for example, or if something new should be created.
- Non-essential/Wishlist/tidying up:
  - ■ *Done.* Restructuring how `CCDataAttr()`s work to optimize for storage. There's a lot of redundancy currently – for example, long descriptions of attributes don't need to be repeated for every data point. There are other examples along these lines as well.
  - ■ Cleaning up the functions in `CCDataSet`. There is currently a bit of redundancy in the "data slicing" functions. Maybe this is okay, however. Needs thinking about.

- Would like to make the format of inputs/outputs to functions, optional arguments, etc, uniform across everything in the `CCDataSet`.
- *Done.* Functions to sort by a single attribute; by a sequence of attributes. Would be nice if this only works on the list of pointers, where ccd.data[idx_set] (roughly) would give give you a sorted list if you were to ask for attribute values.
- *Done.* There are "one-off" conveniences built in to some of the functions in `CCDataSet`. For example, if you give a list of `numpy.ndarrays` in to `self.add_datapoints` but only give it a single list of attribute values, it will assume you want to apply those attributes to every element of the list when they're added to the dataset. These are nice in principle, but they can be annoying when they only exist in some places; people will expect a certain convenience they used in one function and get errors when they try it elsewhere. Someone needs to map out a plan of what sorts of conveniences should (or shouldn't) be given to the user when calling functions.
- Modify the `__str__` and `__repr__` definitions for `CCList` to make the list partially visible when displaying in an interactive Python session. **Or**, have a threshold number, beyond which only the total number is shown. **Or** make the head and tail of the list visible beyond a threshold.

- Long-term things, and things to think about:
  - *In progress with CCGroup().* Possibly adding another class layer above this to store multiple datasets
  - Tracking biomarkers/genes/pathways/etc across multiple species. Possibly tied in with the previous point.
  - Implementing a graph structure to keep track of the relationship between attributes to look at "relaxed" classification, towards the goal of looking at how one classification applies to another. For example, how time of exposure classification done in 8-hour intervals performs if relaxed to 24-hour intervals.
  - Following up from the previous point, a pipe dream would be to relate how biomarkers for infection relate across biological taxonomies; e.g., relating different types of monkeys, relating monkeys to humans, etc.
  - Something to think about – is there any reason that `CCDataPoints` have an elevated status? Meaning, when you ask for `ccd.data[0]`, the thing you get back is a `CCDataPoint`. Does this make the most sense? Can they be considered just another `CCDataAttr` that just happens to be vector-valued? If yes, and we have a good reason to do something about it, then this means we would need to do some fundamental restructuring. This might be as easy as doing what I said above, but a fair number of functions in `CCDataSet` would need to be modified.

Examples

Below are a couple examples of how I've used this so far. Typically this has been done by pulling information from one or more files which we already know information about, and values are populated semi-manually. We'd like to move away from this if possible.

TAMU mice telemetry data

(placeholder)

Human challenge 7-study dataset (derived from already preprocessed GEO73072)

(placeholder)

Parameter sweep in a numerical algorithm applied to a small collection of functions

This is in relation to code that Ma has for anomaly detection based on radial basis functions. The algorithm has a real-valued parameter we deal with. There are a few types of functions which each take a parameter δ which has essentially the same role of adding noise of scale δ to a smooth function. Given all this, there are three attributes;