Architecture for metadata inference

December 16, 2013

1 Overview and very rough pseudocode

Between us we've got at least two, maybe more, different classification tasks. I think a lot of the underlying data structures / classes can be reused, but there will also be significant differences. Language is a categorical variable, whereas date is an ordinal variable. Also, you'll be needing a dataset broken down to the page level, whereas I'll be working with volumes.

That kind of flexibility won't be *too* hard to design for. The more significant challenge may be scale. We're almost certainly working with datasets that are too large to fit into memory all at once. And yet we want to train classifiers using as much data as we can.

The elegant, painful way to handle this would be to work with something like the Mahout library, built on top of Hadoop. If you want to tackle that, I'm game! But I spent last summer trying to translate routines into Hadoop and found that the task was likely to take more time than it was actually worth to me.

I suspect we can probably handle the problem serially, with a bit of ingenuity. Perhaps the general logic of a solution is **a)** map metadata for the whole collection, so we know which subsets we need for which task. The metadata probably fits in memory. Then **b)** read in subsets of the collection as needed to train specific classifiers. In other words, we might read in all German texts, plus a random selection of non-German texts, in order to train a "German" classifier. Then we release that memory (or the garbage collector releases it for us) and we read in all Italian texts, etc. The classifiers themselves will fit in memory without difficulty. Finally **c)** we make a pass through the whole collection, reading in chunks of n volumes at a time (where n fits in memory) and applying all our classifiers to make predictions for each chunk.

A more detailed model of the procedure, in very rough pseudocode. I'm mainly concerned to identify the classes we're going to need and the parts of the logic that need to be abstracted for maximum flexibility.

1. Import metadata keyed to HathiTrust volume IDs.

2. Create *Volume* objects and organize these in a *Collection.* For page-level classification it may possibly be useful to know in advance how many pages are contained in each volume.

   > At this point we also *might* need to make a preliminary pass through volumes to identify most-common features. But possibly that should be a separate task, since it's something that can be done once for a given collection.

3. Create a *ClassMap* object. This will tell us a) what metadata field in the *Volumes* is relevant to this classification task, b) the specific classLabels we want to train and c) how to map the values in (a) into labels contained in (b). E.g., with date, this is going to be fairly complex, because we're going to create classes that represent "bins" of dates at (say) the decade level, plus we may have to deal with missing data or entries like "[18??]." We probably need to create some general protocol for class labels that are missing, suspect, or not relevant to this task.

4. Now we iterate through the list of classLabels in our *ClassMap,* building a *Classifier* for each one and storing those classifiers in an *Ensemble*. For each classLabel, do:

   - Identify the subset of *Volumes* or *VolumePages* that we intend to use as a training set for this classLabel. An important question here is how to balance the number of positive and negative examples for each class. We'd like to use as many data points as possible, but since we can't fit everything in memory, this conflicts with the goal of reproducing the original ratio between positive and negative examples in the larger Collection.
   - Identify the features to be used for classification. This might be either a global set of features used for all classifiers, or a set of features selected for this specific classLabel. In the latter case, we may need to make a pass through the *Volumes,* loading texts or recalculated word counts, in order to identify features that are useful for discrimination here. Possibly this task overlaps with the next one.
   - Create one or more *Corpus*es. A *Corpus* is an array of *Instance*s, which can represent either Volumes or Pages. The important thing about an *Instance* is that it has a classLabel and a set of features mapped to featureValues (i.e., words and wordcounts). So at this point (if not in the previous step) we're doing disk access to get feature counts.
   - Send the *Corpus* to a *Classifier.* There's probably an interface or superclass like *Classifier* that defines general methods all classifiers must have, but this is implemented as *NaiveBayesClassifier, kNNClassifier,* and so on. Construction of the classifier actually trains a model for predicting this classLabel, and creates parameters, coefficients, etc. that are stored in the *Classifier.* Note that some classifiers

are going to need to normalize feature counts – i.e., as relative rather than absolute frequencies. So information like "total number of words in a given volume" needs to be incorporated in the *Corpus* for the *Classifier* to use if needed.

5. Now that we have an *Ensemble* of *Classifier*s, we can test them or actually perform classification. For testing/evaluation of our method we would apply these classifiers to a held-out *TestCorpus*. For actual production, we would iterate through the whole *Collection,* chunk by chunk. For each chunk do:

   - Convert the chunk of Volumes into a *Corpus* by reading in the appropriate features (and segmenting by page if that's required for this classification task).
   - For each classifier in the ensemble, generate a prediction for all *Instances* in this *Corpus*.
   - Store all predictions in a *PredictionArray*.

6. For some tasks, we might just use the *PredictionArray* as-is. But if we're predicting date, or a similar continuous/ordinal variable, there's another step. We have a histogram of probabilities that each Volume belong in a particular decade (or whatever bin size we decide to use). Now to generate an actual date at the level of individual years, we need to do smoothing and interpolation. For instance, we might do loess smoothing on all twenty decade probabilities, and then find the peak of the curve. The x-axis value of the peak is the inferred date of the document.

   Another possibility here is to use a kNN classifier, which can predict a continuous value directly, without smoothing and interpolation.

7. For some tasks, we might end here. But in practice, it's likely that we'll want to identify outliers in the dataset—works where the inferred classLabel differs greatly from the metadata provided by Hathi. Since we can do a better job if we're training on a cleaner corpus, we could mark those Volumes as unreliable for training purposes and re-run the whole process.

Testing this process is tricky, because the whole point is that we don't trust the supposed "ground truth" represented by metadata in Hathi. We might need to manually confirm a small dataset for testing purposes.

2 Variants and ways this might eventually be extended.

I'm eventually also going to want to use a lot of these classes for a couple of other tasks, so I may want to set them up in a way that can be extended.

- I'm going to need to expand my page-level genre classification workflow to encompass datasets that don't fit in memory. Much of the code we write here—especially iterating through a collection too large to fit in memory—will be applicable to that task. Some relevant differences: the features I use in that task are different, including things like number-of-capitalized lines and position-of-page-within-volume. Also, after page-level classification is complete, there's a separate process of smoothing the sequence using a hidden Markov model. Also, in that task labeled training data is separate from the main collection, which is not labeled. So, on the whole a pretty different process. It may just be a question of reusing parts of this code.
- I might also use some of these classes for a co-training process where the hidden Markov model and page-level classifiers effectively keep each other honest.

3 Rough class definitions

We're sort of taking this from the top level down.

3.1 Main class

Called whatever we want to call this—I haven't decided. "MetadataPredictor" is a little clunky, maybe we can come up with something better. CollectionMapper? This class accepts string arguments so it can be called from the command line. Uses those arguments to select a *ClassificationProcess*. This allows us to encapsulate some of the variations involved in different applications, instead of having a lot of code for different variations confusingly piled up in the main class itself.

String arguments probably also include file paths to sources of data and metadata.

3.2 ClassificationProcess

An abstract superclass; we can define subclasses like LanguageClassificationProcess, DateClassificationProcess, GenreClassificationProcess, and so on. This class does guides the overall flow of the process and decides things like, whether the whole classification process runs once or twice, or more; whether evaluation is done, etc. If there are generalizable aspects of implementation we can move them to the superclass.

There are aspects of this class that won't be generalizable. But in general, this class accepts paths to data and metadata. It passes the metadata path to some kind of *MetadataReader,* which returns a *Collection* object and a *ClassMap.* Then it iterates through the classLabels in the *ClassMap*; in each case it

- Iterates across classLabels in the *ClassMap* and, for each one, gets back an ArrayList of *Volumes* that need to be read.

- It sends that array of *Volumes* to some kind of CorpusReader, which actually reads in the data for those volumes, breaks them down to pages if it's the kind of CorpusReader that produces page-level data, and returns a Corpus of Instances.

3.3 MetadataReader

Abstract superclass, will need to be instantiated as *HTRCMetadataReader* or as *TaubMetadataReader*, etc., because I work with metadata that is already in a specific tabular form.

It accepts path(s) and uses them to locate a) the metadata itself and b) some kind of file that defines a *ClassMap* by mapping specific metadata values to classes. Format to be defined.

3.4 Collection

Contains an ArrayList of *Volumes* and a *ClassMap*. Has a method that accepts a classLabel and returns all *Volumes* matching that label. Another method should accept a classLabel and an integer n and return n randomly distributed *Volumes* matching the classLabel. Possibly a third method returns n randomly distributed *Pages.*

3.5 Volume

Fields: a volume must have a String volume ID (normally the HathiTrust volume ID).

It can have a number of other metadata fields drawn from existing manual metadata (date, language, genre, possibly title and author).

It can also have tentative predictions about those fields produced by our own classification process. (This will become relevant if we need to repeat the classification process more than once, using past predictions to filter the training set.) Note that predictions can be a little tricky to represent, e.g. our prediction about language is not necessarily a string like "Greek." It could be a series of language categories, each connected to a probability (real number between 0 and 1) for that language. So I'm going to tentatively suggest that we create a *Prediction* class, and that the Volume should store *Prediction*s rather than values.

We shouldn't assume that the volume actually contains information for all possible metadata fields. For many projects, most fields will be irrelevant.

A volume may have integer fields that record the number of pages in the volume and number of words in the volumes. But it doesn't contain actual wordcount data for pages or the volume as a whole. That's the job of *Instances.* A Volume is a persistent object; Instances are created and destroyed each time we train a classifier or classify a chunk of the collection.

3.6 ClassMap

Abstract superclass instantiated as subclasses: OrdinalClassMap, NominalClassMap, etc.

Basically, it contains an ArrayList of classLabels, and maps specific metadata values to classLabels. An OrdinalClassMap probably also contains a field for binWidth.

Also contains a link to the Collection that it maps.

For the date problem, classLabels are integers representing the midpoint of a "bin," for many other problems, they are strings. We could handle this in one of two ways: 1) treat dates as String representations of the integer. or 2) let Ordinal classLabels be integers and try to encapsulate this problem within the ClassMap. E.g., define a nextSubset() method inside the ClassMap that allows the ClassificationProcess to iterate across classLabels without necessarily needing to know whether they are strings or integers. The process just tells the ClassMap "next!" and lets the ClassMap identify the next label and extract matching volumes from the Collection.

Things are going to get a bit sketchier here, because I'm running out of time to write this morning.

3.7 Classifier

Abstract class, instantiated as BayesianClassifier, KNNClassifier, etc.

3.8 Corpus

Basically an array of Instances.

3.9 Instance

Each instance has a class label, and a list of features mapped to feature counts. Not sure yet whether this is implemented as a pair of ArrayLists or as a HashMap.

3.10 CorpusReader

Abstract superclass instantiated as as *HTRCMetadataReader* or *TaubMetadataReader,* because it's likely that our data will be in different formats.

This class reads in word count data for the Volumes it is sent (or possibly Volumes + a list of page indexes if we want to read in specific pages).

It may either read this from word count files already produced by a separate tokenizing process, or it may actually do the tokenizing. I think in the case of my Taub data it's going to do the former. I already have a separate tokenizing/word counting workflow. But this could be different for the HTRC implementation.

3.11 Prediction

3.12 PredictionArray

4

A couple of sources on date prediction. No need to look at them, they're not necessarily relevant to our broader project. Just filing them here for future reference.

References

[1] Tilahun, Gelila, Andrey Feuerverger, and Michael Gervers. "Dating Medieval English charters." *The Annals of Applied Statistics* 6, no. 4 (2012): 1615-1640.

[2] Kumar, Abhimanu, Jason Baldridge, Matthew Lease, and Joydeep Ghosh. "Dating Texts without Explicit Temporal Cues." arXiv preprint arXiv:1211.2290 (2012).