Project 1 Sample Solution

Created by Eytan Adar for CSE454

This is a walkthrough for the sample solution. The actual compliable code is available separately from the course website. In all, we have ~150 lines of actual code. Though the documentation for both Hadoop and Mallet are somewhat lacking, I found the necessary information for everything here from the javadoc and wiki pages for Hadoop (I didn't know either before implementing this solution).

Let's start defining our class…

```
public class VectorBuilder {

    public static void main(String[] args) throws Exception {

      // Remove common prefix from all the input class directories
      // …A bunch of stuff we gave you…
```

Construct the pipeline

```
      Pipe instancePipe;
```

instancePipe is the pipeline to generate Instance objects. Each element in the pipeline acts on the incoming data in some way to process it. Recall from the documentation that an Instance consists of 4 parts: data, target, name, and source (see the javadoc for more info). The FileIterator which we use to read from the file system creates an initial Instance with:

- data = a pointer to the file
- target = the path of the file, this is also the label
- name = the URI to the file
- source = null

```
      instancePipe = new SerialPipes (new Pipe[] {
```

The system keeps a "label alphabet" of labels it knows about, so we convert the path to the label, but basically the system is just building a list of all the labels

```
            new Target2Label(),
```

The system now copies the source from the data field.

```
            new SaveDataInSource(),
```

The system looks at the data field, realizes it's a pointer to a file, loads up the text in the file and stuffs that back into the data field. Data is now just the text as a big long character array

```
            new Input2CharSequence(),
```

We now need to split the long character sequence into a token (i.e. word) sequence. So CharSequence2TokenSequence does exactly that. It replaces the data field of the Instance with a token sequence (a bunch of words)

```
            new CharSequence2TokenSequence(),
```

At this point you now have the opportunity to run the normalizations you want (lowercasing, removing stopwords, etc. etc.)

Next, transform the sequence of tokens into a sequence of features (e.g. each word gets assigned a number)

```
            new TokenSequence2FeatureSequence(),
```

Lastly, we take the sequence and transform it to a vector (counts of number of times a certain word appears). What we're left with in the Instance is

- data = the feature vector (specifically an object of type FeatureVector)
- target = the label
- name = the URI to the file
- source = the original path

```
            new FeatureSequence2AugmentableFeatureVector()

        });
```

To read in all the data from the file system we do:

```
InstanceList ilist = new InstanceList(instancePipe);
boolean removeCommonPrefix=false;
```

The instance list calls the file iterator which produces our starting instance (see above), which then gets passed through the pipeline. We're going to "cheat" a little and just process the full set of files. This is the way to go if we want to create one Instance per file to send over to Hadoop, but isn't ideal if we're going to just save the text for the "test" data. Ideally, we would randomly select a subset here and send 60% of it through the pipeline and keep 40% for later processing, but I'm being lazy so… I'm also being lazy by assuming the directory structure, and therefore the filenames, contain the correct class of the file we're trying to classify. This will make it easier for us to compare the "guessed" classification with the known one.

```
ilist.add (new FileIterator(directories,
                        FileIterator.STARTING_DIRECTORIES,
                        removeCommonPrefix));
```

Lets divide up the data into a train and test set

```
InstanceList[] ilists = ilist.split(new Random(),
                              new double[] {.6,.4});
```

```
InstanceList train = ilists[0];
InstanceList test = ilists[1];
```

Create the trainer (there are other trainers/classifiers that you could use):

```
NaiveBayesTrainer nbt = new NaiveBayesTrainer();
```

Train the classifier

```
Classifier classifier = nbt.train(train);
```

Save the classifier to disk

```
ObjectOutputStream oos =
    new ObjectOutputStream(new FileOutputStream("classifier"));
oos.writeObject(classifier);
oos.close();
```

Here's where things get a little tricky. We want to save out the test set somehow. We have some choices to make. First, is the type of object we're going to keep around for Hadoop.

Keeping around pure instances is generally a bad idea, but you can do that if you want (your file size will blow up). The other option is to make a "light" Instance format which contains the fields necessary to reconstruct the real Instance without all the extra cruft. This means saving, the target,name, and source (all three of which are small), and also saving the FeatureVector. We don't want the pure FeatureVector because that contains the alphabet. Instead we'd grab the indices and values (getIndices() and getValues() respectively).

An easier way to do this, by far, is to just save the text and pass it through the pipeline at each mapper node.

Once we've decided if we want to keep around an "Instance" or text we have to decide what file format to use. If we use an Instance, the right thing is probably to use a SequenceFile. If we use Text we can use either the SequenceFile or stick one file per line in a big text file. I personally think that a SequenceFile is more appropriate for either scheme so I'll walk you through that.

Create a new configuration object

```
Configuration conf = new Configuration();
```

We're going to use the local filesystem, and create a local file

```
FileSystem fs = FileSystem.getLocal(conf);
Path p = new Path("test.data");
```

Make a new sequence file writer, tell it where to save the file and the type of keys/values. If we want to use our "light" instance the value type should be ObjectWritable or a BytesWritable (the former means that system takes care of serializing for you, the latter means you have to do it). We're just going to use text/text with no compression (see the sequencefile documentation for information about compression)

```
SequenceFile.Writer seq =
    SequenceFile.createWriter(fs, conf, p,
                        Text.class, // the key type
                        Text.class // the value type
                        );
```

Iterate over the instances in the test data

```
for (int i = 0; i < test.size(); i++) {
    Instance inst = test.getInstance(i);

    Writable value = null;
```

If we were going to save an Instance in an ObjectWritable we might do (if we defined the "light instance" type):

```
        // LightInstance li = createLightInstance(inst);
        // value = new ObjectWritable(li);
```
or just
```
        // value = new ObjectWritable(inst)
```

If we wanted to use the ByteArrayWritable, we might do:

```
        // ByteArrayOutputStream baos = new ByteArrayOutputStream(...)
        // oos = new ObjectOutputStream(baos)

        // oos.writeObject(inst);
```
or
```
        // oos.writeObject(createLightInstance(inst);


        // oos.close();
        // value = new BytesWritable(baos.toByteArray());
```

But we're just reading text so...

```
        BufferedReader br =
          new BufferedReader(new FileReader(inst.getSource().toString()));
        StringBuffer content = new StringBuffer();
        while (br.ready()) {
          content.append(br.readLine());
          content.append('\n');
          // if we wanted to put everything on one line, we'd do
          // content.append(' ') instead
        }
        br.close();
```

Create the key

```
        Text key = new Text(inst.getSource().toString());
```

Create the value

```
            value = new Text(content.toString());
```

Append it to the sequence file

```
            seq.append(key, value);
        }
```

Close it up

```
        seq.close();
```

And we're done... we now have one file (test.data) that has all our test data, and one file (classifier) that has the trained classifier and pipeline. You load up these two files into the hadoop file system.

The next part of the assignment was to create a mapper that outputs id, classification pairs. So here we go:

```
public class Project1Mapper extends MapReduceBase implements Mapper {
```

Let's create a singleton for the classifier and pipeline

```
    static Classifier classifier = null;
    static Pipe instancePipe = null;
```

We have to load up the classifier from disk, and we'd prefer to do that once per mapper rather than every time map is called. A reasonable place to do that is in the "configure" method which gets called once per mapper. Here's the definition of this method.

```
    public void configure(JobConf job) {
      // if we had coded in our file locations, we would pull them out
      // of the configuration... I'm just going to use a hard coded
      // location
      try {

          // we could probably have stuck both the classifier and
          // the alphabet in one file

          if (classifier == null) {
            // singleton pattern, probably should be synchronized
            Path p = new Path("input/classifier");
            FileSystem fs = p.getFileSystem(job);
            FSDataInputStream fsdis = fs.open(p);

            // load it up from disk
            ObjectInputStream ois = new ObjectInputStream(fsdis);
            classifier = (Classifier)ois.readObject();
            ois.close();
```

```
            // grab the instance pipe from the classifier
            instancePipe = classifier.getInstancePipe();
        }
      } catch (Exception ex) {
          ex.printStackTrace();
      }
    }
```

Next comes the actual meat. We want to define map:

```
    public void map(WritableComparable key, Writable value,
                OutputCollector output, Reporter reporter)
      throws IOException {
```

Load the instance object, if we had saved it out in the writable we'd reconstitute it here, but we have text, so...

```
      Instance inst =
          new Instance(value.toString(),key.toString(),
                   key.toString(),instancePipe);
```

Only trick is this, we want to pipe the data so that the two pipelines match (the one used for the classifier and the one used for classification) otherwise we'll get an exception

```
      inst = inst.getPipedCopy(instancePipe);
```

Generate the classification

```
      Classification classification = classifier.classify(inst);
```

Pull out the best classification, and the file id (I'm cheating and actually sending the target string which is the correct classification, what I really should ship is the source):

```
      Text label =
          new Text(classification.getLabeling().getBestLabel().toString());

      Text target = new Text(key.toString());
```

Ship them off:

```
      output.collect(target,label);
```

That pretty much does it for the mapper class. The next step is to build a reducer. Remember that a reducer will get a key and a set of values. The thing we have to be careful about is that each call to reduce will only give us one ID with one associated classification and we want an aggregation. This means two things. First, we have to force all keys to one reducer. Second, we're going to keep around the a static counter variable (though an instance variable would probably do just fine). When the reducer is done, we're going to write out the value.

```
public class Project1Reducer extends MapReduceBase implements Reducer {

    static int correct = 0;
    static int total = 0;

    public void reduce(WritableComparable key, Iterator values,
                       OutputCollector output, Reporter reporter)
      throws IOException {
      while (values.hasNext()) {
          Text t = (Text)values.next();
          if (key.toString().contains(t.toString())) {
            correct++;
          }
          total++;
      }
    }
```

When we're done, Hadoop will call "close." At this point we can calculate the accuracy and save it out somewhere (in this case, a file called "accuracy")

```
    public void close() throws IOException {
      // write out the file with the accuracy to disk
      Path p = new Path("output/accuracy");
      FileSystem fs = p.getFileSystem(job);
      FSDataOutputStream fsdis = fs.create(p);
      PrintStream ps = new PrintStream(fsdis);
      ps.println("accuracy: " + ((double)correct/(double)total));
      ps.close();
    }
…
```

That's more or less it for the reducer. Last step is to write a main method that configures and runs everything. I'm cheating and hard-coding pretty much everything. The right way is to encode all the configuration stuff in various configuration variables. I'm also going to put the main loop inside the Project1Mapper class though it can be put somewhere else.

```
    public static void main(String[] args) throws Exception {

      …

      Path inDir = new Path(args[0]);
      Path outDir = new Path(args[1]);

      JobConf job = new JobConf(new Configuration());

      job.setJarByClass(Project1Mapper.class);
      job.setJobName("classifier");
      job.setOutputPath(outDir);
      job.setInputPath(inDir);
```

Tell Hadoop how the input is structured (good habit to get into):

```
job.setInputFormat(SequenceFileInputFormat.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

We're forcing everything through one reduce task:

```
job.setReducerClass(Project1Reducer.class);
job.setNumReduceTasks(1);
```

Set the mapper class:

```
job.setMapperClass(Project1Mapper.class);
job.setNumMapTasks(20);  // or pick your poison
```

Configure and run:

```
JobClient client = new JobClient(job);
ClusterStatus cluster = client.getClusterStatus();
JobClient.runJob(job);
```