Intro to Bioinformatics
Dartmouth CS 75/175, QBS 175
Fall 2019
Lab 3: Expression analysis

This homework explores unsupervised and supervised analysis of expression data. You are to implement your own code for the various function bodies, not just call code from machine learning packages. And just to remind you of the honor code principle, you are to do this without reading existing codes, though as always, you are welcome to read journal articles and textbooks and their descriptions of the algorithms, and talk in natural language about them.

As with the in-class assignments, expression data is represented as a list of ExpressionProfiles, each for a different sample, with an array of values for different genes. The same datasets are considered -- the fake “simple” dataset, the Golub et al. ALL/AML dataset (Science 1999) and the van't Veer et al. cancer recurrence dataset (Nature 2002). Feel free to find others; please describe and upload, and share on Piazza for others to try.

Submit on Canvas, before class, your solutions to the following. For the programming problems, submit a Python file with your code inserted in the provided template. As always, please don’t modify the command-line drivers that the TAs will use to test your code, but feel free to use them yourself. For the application parts submit a PDF, Word file, etc.

1. Unsupervised: hierarchical clustering (25 points)
   In hierarchical.py, implement hierarchical clustering for a given set of expression profiles, filling in the body of the ExpressionHierarchicalCluster.cluster method.
   - A scaffold for a suitable binary tree structure is provided, with the base class ExpressionHierarchicalCluster and subclasses for Leaf and Inner nodes. Each leaf holds a profile, while the inner nodes hold left and right children, themselves ExpressionHierarchicalClusters (could be inner or leaf). Helper functions are provided to pretty-print (pprint) the tree structure line-by-line (good for large trees) or with the usual python str() method (for small trees). There’s also a very simple plotting function in ExpressionProfile (revised from the in-class version) to show the heatmap, which you can do after reordering the profiles according to the tree (see the command-line driver for an example).
   - Start by creating leaves for all the profiles and computing Euclidean distances between each pair. (5 points)
   - Then repeatedly find the closest pair of clusters (could be leaves or inner nodes; doesn’t matter) and merge them into a new inner node that can be used in subsequent iterations. (9 points)
   - Compute cluster-profile and cluster-cluster distances, allowing the choice of average linkage (i.e., the average of all pairwise distances from members of one cluster to members of the other) or min or max linkage (i.e., the smallest or

largest distance from a member of one cluster to a member of the other). (6 points)

- Discuss the results of clustering the two real datasets, in terms of both visual patterns in the heatmap and correspondence with the known (to us, but unused by the unsupervised learning approach) class labels. How does the choice of linkage affect the results? (5 points)

2. Supervised: random forest (55 points)
   In rf.py, implement a random forest classifier for expression profiles, training on a set of given profiles and predicting on others.
   - A scaffold is provided for the forest itself, as well as its trees, with a Tree superclass and Leaf/Inner subclasses. Again there is an implementation to allow str() to generate a string representation for printing. I recommend that you implement and test a single tree first, before dealing with all the details of the whole forest.
   - The *random* module provides helpful functions for sampling with replacement, choosing a subset, and shuffling. Or if you want to get one level deeper insight, you can easily implement those yourself starting from the function to generate a random number.
   - Tree
     - Base class, as described above
     - *impurity*: compute the Gini impurity of a set of profiles (3 points)
     - *train*: to guide the construction of trees (each split), call this method and let it dispatch to either Inner or Leaf construction. If there are too few remaining profiles or all the allowed splits are used up, it makes a leaf; else it makes an inner. (provided for you)
   - Leaf (5 points*)*
     - A leaf just represents a single label. So *train* stores it away and *classify* uses it.
   - Inner (25 points)
     - An inner node decides which child to branch to, by testing the value of a certain feature. So *train* chooses a feature and value, and *classify* recurses accordingly.
     - *classify*: branch according to test (4 points)
     - *train*
       - Find the best split (below) and build child trees. (6 points)
       - In the constructed node, also keep track of the mean decrease in impurity (mdi), for use in calculating feature importances. This is the impurity of the node itself minus the weighted impurities of its children, with this decrease in turn weighted by the fraction of the total profiles that actually reach the node. (3 points)
     - *find_split*: determine the feature and value to split on for an inner node

        - Consider a random subset of features of the specified size. (For testing purposes, the size can be that of the whole set, but in practice, it's typically the sqrt of that.) (3 points)
        - For each feature under consideration, and each splitting value, calculate the impurity of each of the two subsets of profiles split accordingly. Sum these, weighted by the fraction of profiles in each subset. Identify the feature and splitting value that give the best (smallest) weighted impurity. For splitting values, consider the various values that show up in the profiles for the feature at hand, and take the midpoints between adjacent values. (10 points)
    - *add_mdis*: a helper function for computing feature importances over the whole forest. The argument is a "running total" mdis dictionary that's being passed around, mapping each feature to the sum, over the trees, of the mdi for that feature. So here, a node adds its saved mdi value (computed during *train*) for their feature, and recurses to its children. (3 points)
- Forest (15 points)
    - *classify*: majority vote of the trees (5 points)
    - *train*: train the specified number of trees, each on randomly resampled profiles (if the flag is true) or the original profiles (false) (5 points)
    - *importances*: compute feature importances by asking each tree to add_mdis to a "running total" dictionary, and then dividing by the number of trees to get the overall mean (5 points)
- To help you with testing, there's a function to *concoct* data according to a specification. I had too much fun, and we'll use it to test, so I'm providing it to you; however, feel free to ignore depending on how you choose to answer the application questions.
    - The *n_per_label* argument is a list indicating how many samples per class. For a 2-class setup with 10 samples each: [10,10], i.e., 10 of class label 0 and 10 of label 1.
    - The *feat_specs* argument is a list of feature specs, with one spec for each feature to be generated.
        - The feature values are conveniently the same as the class labels (with Gaussian noise if desired). So a two-class setup will have feature values of 0 and 1. Makes interpretation easier :).
        - A feature spec is a list, for each class, of how many of its members should get each label. So [5,5] would indicate that 5 of the members get value 0 and 5 get 1. (Again, plus noise if desired.)
        - Thus [[8,2],[3,7]] would be a feature spec indicating that for class label 0, 8 of the samples should have value 0 and 2 value 1, while for class label 1, 3 should have value 0 and 7 value 1.
    - The *sigma* argument controls how much noise to add

    - Finally, the *shuffle* argument indicates whether to randomize how the features values are dealt across the samples. If not shuffled, then in the above example the first 8 of class 0 would get value 0 and the remaining 2 would get value 1, and similarly first 3 / last 7 for class 1. Shuffling breaks that correlation.
- Application
    - Devise some artificial test cases to demonstrate your code. You can use my concoct function if you find it helpful, or come up with them some other way, or both. Test the construction of an individual tree, as well as a forest. Briefly summarize the test cases and why they give you confidence. (5 points)
    - Train forests for the two real datasets we've been using (leukemia_sm and recur). Artificially hold out some of the data, partitioning into training and testing as was done for the in-class assignment, and see how well the forest performs, over a few such splits. Examine the features and the consistency (or not) of the most important ones. Briefly summarize your findings. (5 points)

3. Supervised: cross-validation (20 points)
   Cross-validation is an important aspect of assessing generalization ability of a classifier. While random forest implementations often characterize this with their "out of bag" error, cross-validation is a standard mechanism that can be readily compared across different implementations. Augment rf.py with a cross-validation mechanism for your forest.
    - Implement repeated n-fold cross-validation, in which for each repetition (say 10 of them), the data is split into a given number (say 5) of "folds" (subsets). For each fold, use it as the test set (make predictions for its labels) based on a training set comprised of the members of the other folds. In this case, that means for each expression profile, predict its label according to a random forest constructed from the other folds. For each repetition, randomly partition into new folds, thereby varying the training data used for predicting each instance. Assess the quality of the classification in terms of how many of the test instances are assigned correct labels, keeping separate counts for each label. (15 points: 6 for folds, 6 for repetition, and 3 for accuracy calculation)
    - One implementation detail: so as not to have to pass all the random forest arguments into the cross-validation function, we instead pass a function that just needs the training data, and will call the random forest train method appropriately (with the additional previously specified parameters). There's an example in the command-line driver:

      ```
      lambda eps: ExpressionRandomForest.train(eps, ntrees,
      min_size, max_splits, nfeats_test)
      ```

      This specifies a function that takes the expression profiles and calls the RF train method with those eps and a bunch of other arguments.
    - Application

    - Now put the real datasets through cross-validation evaluation (10 repeats of 5-fold is fine, or vary if you like). Discuss performance over different choices for the random forest parameters. (5 points)

Extra credit ideas

- Find some interesting original data, with associated gene name, and dig deeper into the forests and feature importances.
- Beef up the hierarchical clustering -- support different distance metrics, be more efficient, and help choose where to chop the tree with a metric for quality.
- More fully implement and explore random forests, supporting evaluation of out-of-bag error, allowing for imbalanced datasets, etc.