Bioinformatics III

Prof. Dr. Volkhard Helms
Markus Hollander

Saarland University
Chair for Computational Biology

Summer Semester 2021

Exercise Sheet 3

Due: May 6, 2021 before 12:00

Submit your solutions to markus-hollander@web.de with two attachments: (1) A ZIP file containing all your source code files, potential result files and whatever else is needed to generate your solution, (2) a PDF file containing your answers and your properly formatted source. For detailed instructions see 'Submission Process and Rules' on the first assignment sheet.

Network Communities and Bayesian Classification

In this assignment you will implement a naive Bayes classifier in **bayes.py** and the the algorithm of *Radicchi et al.* in **network_communities.py** to identify the communities of a given network.

Exercise 3.1: Network communities (50 points)

(a) **Edge-clustering coefficient (5 points)**

The edge–clustering coefficient $\tilde{C}_{i,j}^{(3)}$ of an edge between nodes i and j is defined as the ratio of the number of triangles $z_{i,j}^{(3)}$ to which the edge between i and j contributes and the number of possible triangles, determined by the minimum of the degrees k_i and k_j of the two nodes i and j:

$$\tilde{C}_{i,j}^{(3)} = \frac{z_{i,j}^{(3)} + 1}{\min[k_i - 1, k_j - 1]}$$

If one of the nodes has a degree of 1, then $\tilde{C}_{i,j}^{(3)}$ is infinite. What is the maximal *finite* value that the edge–clustering coefficient can take and why? For which configuration does this occur? Give an example!

(b) **Network Decomposition (15 points)**

(1) Implement the function **triangles(**i**,** j**)** that computes the number of triangles to which the edge between i and j contributes.

(2) Implement the function **edge_clustering_coefficient(**i**,** j**)** that computes $\tilde{C}_{i,j}^{(3)}$ for the edge between i and j.

(3) Implement the function **decomposition(***network***)** that decomposes the input network as follows:

i. Calculate the edge–clustering coefficient $\tilde{C}_{i,j}^{(3)}$ for each edge.

ii. Find the edge with the smallest $\tilde{C}_{i,j}^{(3)}$, store it in a list and then delete it from the network. Print the step, name of the two nodes and their current $\tilde{C}_{i,j}^{(3)}$.

iii. Repeat i. and ii. until no edges are left in the network.

iv. Return the list of deleted edges.

(4) Read in the network from 'network.tsv' and decompose it. Do not forget to include the print outs from the decomposition in your PDF.

(c) **Building Communities (15 points)**

(1) Implement the function **classify(***community***)** that uses the following criteria (see *Radicchi et al., 2004*) to classify a community:

i. In a *community in a strong sense*, every single member of the subgraph V has more links to the inside of the community (k^{in}) than to the outside (k^{out}):

$$k_i^{\text{in}}(V) > k_i^{\text{out}}(V) \qquad \forall i \in V$$

ii. In a *community in a weak sense*, the total number of links inside the subgraph V is bigger than to the outside:

$$\sum_{i \in V} k_i^{\text{in}} > \sum_{i \in V} k_i^{\text{out}}$$

(2) Implement the function **rebuild(***edges***)** that iterates over the list of deleted edges from a network decomposition in reverse order to construct the communities. In each iteration:

i. If the edge is not connected to an existing community, create a new one.

ii. If the edge has a single node in common with an existing community, then add the other node to the community as well.

iii. If both nodes of the edge are already part of the same community, there is nothing to do. If they are, however, part of two different communities, merge the two communities into one. Print all current communities with their classification.

(3) Rebuild the network you decomposed in (b). Do not forget to include the print outs in your PDF.

(d) **Visualising Communities (15 points)**
Draw the network from (b) and then the corresponding dendrogram of the community structure from the results in (c), similar to the example given on slide 25 in lecture 5. To do so, whenever two communities are merged (case iii.), connect them with a bridge. If a single node is merged into a community (case ii.), it is added on the same level (see the rightmost sub–tree on the lecture slide).

Exercise 3.2: Naive Bayes Classifier (50 points)

A naive Bayes classifier is a simple classifier based on the application of Bayes' theorem and the (naive) assumption of independence among all features. This exercise is about implementing and evaluating such a classifier on the basis of two artificial data sets that relate 100 discrete features (for simplicity) to a boolean outcome.

Let's imagine that there are certain nucleotide positions in the genome that are associated with some disease. Each such position is a feature in our classifier and can be one of four variants: A, C, G, T. Let D denote an individual with the disease and H a healthy individual. V_i is the variant at position $1 \leq i \leq 100$ and $V = \{V_1, \ldots, V_{100}\}$ is the collection of all variants. Given V, the goal is to infer if an individual likely has the disease with the commonly used log–likelihood:

$$L = \log \frac{P(D \mid V)}{P(H \mid V)}$$

(a) Briefly explain what $P(D \mid V)$ and $P(H \mid V)$ are and how the value of L is used for classification.

(b) Before implementing the classifier, you need to rearrange the log–likelihood formula to derive an equivalent term that uses observable probabilities such as $P(V_i \mid D)$. State what you are doing in each step.

(c) Briefly explain two advantages of using the log–likelihood ratio for the classifier and two reasons why it may perform poorly on a real world data set.

(d) Next, implement the binary naive Bayes classifier in **bayes.py**.

(1) Implement the initialisation function that builds the classifier model from a tab–separated training data file. That means determining all necessary priors and observed probabilities from (b) and storing them in some manner. Each row of the training data file represents a sample and contains the group (healthy or diseased) in the first column and the variants (features) in the remaining 100 columns.

(2) Implement the function **report(**n**)** that prints the observed prior probabilities $P(D)$ and $P(H)$, as well as the n variants V_i with the highest absolute log–ratios:

$$\log \frac{P(V_i \mid D)}{P(V_i \mid H)}$$

(3) Implement the function **classify(***variants***)** that returns 'healthy' or 'disease' given a list of variants V based on the formula derived in (b).

(4) Implement the function **evaluate(***test_data_path***)** that classifies the health status for each sample in the test data file, compares the classification to the ground truth and reports the overall accuracy of the model, as well as the frequency of 'healthy' and 'diseased' in the test data.

(e) Build the model for 'traning_set_1.tsv' and note the results of the report function for $n = 10$. Which features seem the most useful? Evaluate the model with the data in 'test_set_1.tsv' and report the accuracy of the classifier.

(f) Repeat (e) for 'traning_set_2.tsv' and 'test_set_2.tsv'. Why is the performance inferior? Could this have been expected?

Have fun!