ModCompExam - GAUSSIAN TREE MODELS OF GENE LENGTH EVOLUTION

Magnus Ejegod (scz973)[1], Jonathan Wenshøj (jlv849)[2], Elias Rønnow(wkr770)[3], and Alexander Mittet (sxn123)[4]

[1,2,3,4]Department of Computer Science, University of Copenhagen

Marts 31, 2023

Contents

1 Part 1: A Generative Model **2**

2 Part 2: Inference of hidden nodes **6**
- 2.1 Implement inference algorithms for computing the conditional distribution of each of the variables Z0,...,Zn 2 given X1,...,Xn 6
- 2.2 Gaussian Belief propgation 6
  - 2.2.1 Canonical form 7
- 2.3 Test the inference algorithms using simulated data 9
- 2.4 Apply the inference algorithms on the data in the data file and present the results 10

3 Part 3: Learning of the parameters **11**
- 3.0.1 Linear Regression 11
- 3.0.2 EM algorithm 11

A Appendix - Simulation Functions **12**
- A.1 def simulation 12
- A.2 def n_simulations 14

B Appendix - LinReg **14**

1 Part 1: A Generative Model

For this task, *2. A Generative Model*, we had to construct a Gaussian tree model, which is a type of Bayesian network. This is relevant for our task because by using a Gaussian tree model, we can systematically trace the evolutionary history based on present day observations. In general, Bayesian networks are good at describing relationships because they are able to capture both the direct and indirect dependencies between variables in a probabilistic and intuitive way. This is the main argument for choosing such a model for this task.

Moving ahead, we began by encoding the relationships between the parents and children based on the data in *tree.csv*. Now, for our forward simulation of the Gaussian tree model, we had to mathematically describe the relationships. Suppose all the V and $\bar{V}$ variables are independent Gaussian variables. The CPD's given in part 1 can be rewritten to:

$$V_0 = \mathcal{N}(\alpha_0; \sigma_0^2)$$

$$\bar{V}_i = \mathcal{N}(a\bar{t}_i; \sigma^2\bar{t}_i)$$

$$V_j = \mathcal{N}(at_j; \sigma^2 t_j)$$

For $i = 1, ..., 202$ and $j = 1, ..., 204$, we have:

$$Z_0 = V_0$$

$$Z_i = \beta \cdot Pa_{Z_i} + \bar{V}_i$$

$$X_j = \beta \cdot Pa_{X_j} + V_j$$

We then did **the forward simulation** by doing a BFS (breadth-first-search) over our Bayesian network representation and applying our mathematical update equations shown above for each non-leaf node and leaf node. Our tree structure implies that each child only has one one parent, so with a BFS, we can optimally traverse the graph and the simulate the values.

Pseudo-code for forward simulation:

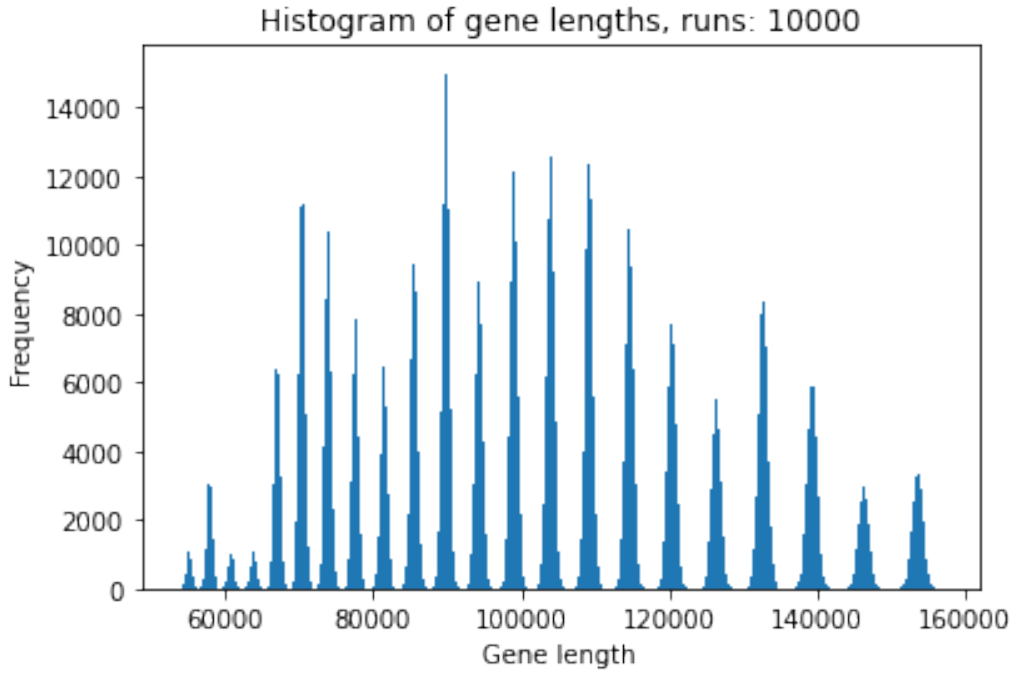
```
def simulation(root, graph, alpha0, variance0, alpha, beta, variance):
    # dictionary with gene lengths for results
    gene_lengths = {}

    # Init Z0
```

```
    z0 = cpd_z0(alpha0, variance0)

    # queue for BFS
    queue = [(root, z0)]

    while queue:
        # handle the next element in queue
        node, z = queue.pop(0)

        # set node in results to genelength z
        gene_lengths[node] = z

        # get all children for the given node
        children = graph.getChildren(node)

        if children:
            for child in children:
                id, t = child

                # mean and variance for child nodes
                mean = (alpha * t) + (beta * z)
                var = variance * t

                # cpd() is for both Xi and Zi
                queue.append((id, cpd(mean, var)))
        else:
                continue

    return gene_lengths

...

# generate n simulations
sim_arr = array(n_simulations * num_of_nodes)
for i in range(n_simulations):
    sim_arr[i] = simulation(...)
```

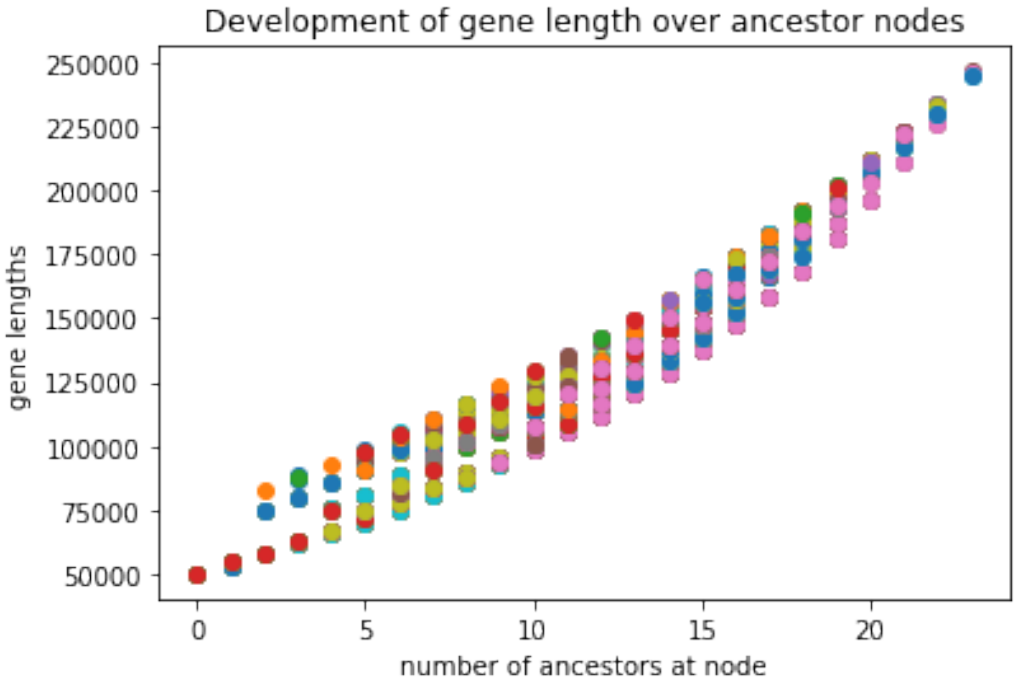
We now wanted to illustrate our implementation, so we **simulated data based on the following parameters:**

Parameters	
alpha_0	**50.000**
variance_0	**5.000**
alpha	**0.15**
beta	**1.05**
variance	**2.500**
Settings	
num_of_samples	**10.000**
root	**407**

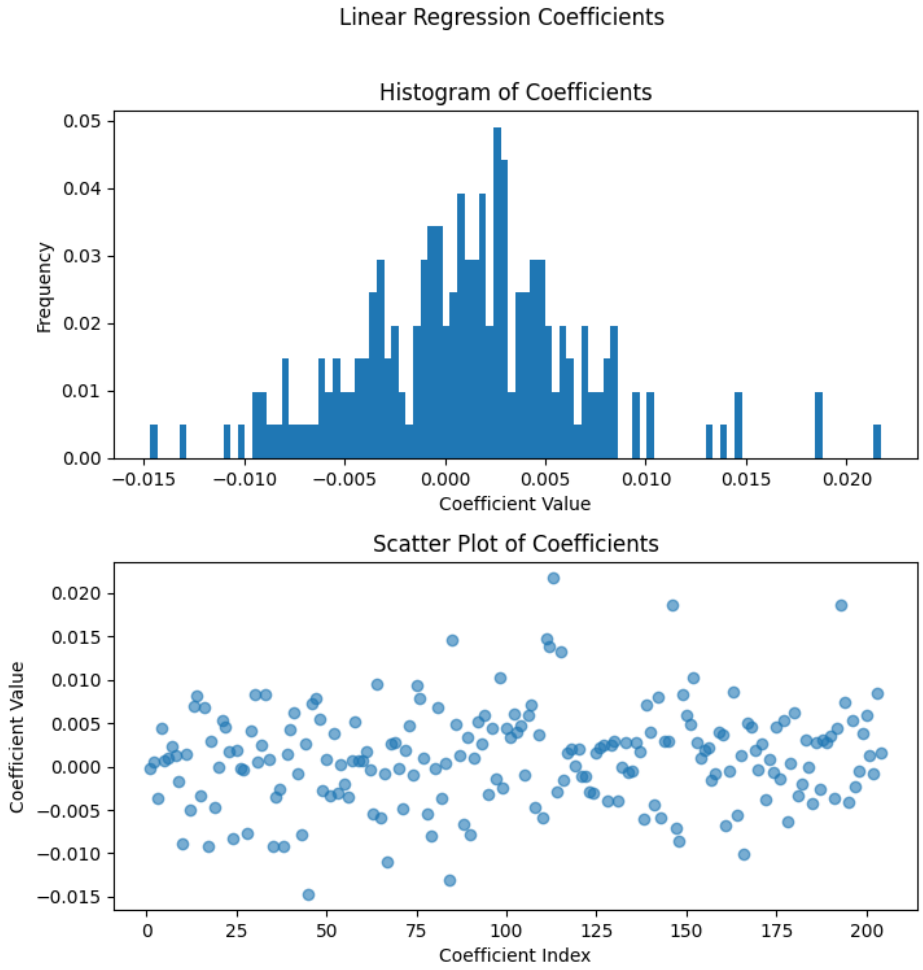
We plotted two plots; a histogram showing the frequency of different gene lengths, and a scatter plot of the gene lengths over the number of given ancestors. In the histogram we see that there are "steps" where the gene lengths fall around. It looks like small Gaussian distributions around some fixed gene lengths.



In the scatter plot, the colors represent an "extended family". Essentially the same color means a given gene has the same ancestor. We see that the lengths of the genes increase what looks like exponentially when the number of given ancestors increase. The variance also decreases when number of given ancestors increase.



Lastly, we made a linear regression on the simulated data. We plot the coefficients both with a histogram and a scatter plot. In the histogram, we see that the coefficient values are Gaussian distributed with a mean close to 0. In the scatter plot the coefficient values look random across the genes. There seems to be no trend across the different genes, as we traverse the coefficients (seen on the x-axis in the second plot as "coefficient index").



2 Part 2: Inference of hidden nodes

2.1 Implement inference algorithms for computing the conditional distribution of each of the variables Z0,...,Zn 2 given X1,...,Xn

2.2 Gaussian Belief propgation

Factor graphs are a graphical representation of factorization in probability distributions. They provide a unifying framework for a wide range of inference algorithms. In our case, the Gaussian Tree Model can be represented as a factor graph, which allows us to leverage efficient inference algorithms

such as Gaussian Belief Propagation (GaBP).

The Sum-Product Message Algorithm is an approach to message passing that computes the marginal distributions of variables in a factor graph. It is relevant to our Gaussian Tree Models as it can be applied to compute the marginal distributions of the hidden nodes given the observed leaf nodes.

GaBP is particularly useful as it can efficiently compute the marginal distributions of each hidden node given the observed leaf nodes. Given the tree structure, we are also guaranteed convergence, so it is exact inference. Additionally, having the correct scheduling of messages guarantees that in Gaussian Tree, we will converge after exactly one upwards and downwards pass.

GaBP is a specific instance of the sum–product message passing for Gaussian graphical models. It uses the properties of Gaussian distributions to derive a simpler and more efficient algorithm. Since our Bayesian Network can be represented as a binary Gaussian tree model, GaBP is a perfect choice for our case.

2.2.1 Canonical form

The following theory is based on chapter 14 of PGM:
Gaussian variables can be factorized through canonical form. Written as:

$$\mathcal{C}(\mathbf{X}, K, \mathbf{h}, g) = exp(-\frac{1}{2} \cdot \mathbf{X}^T K \mathbf{X} + \mathbf{h}^T X + g)$$

$$K = \Sigma^{-1}$$

$$\mathbf{h} = \Sigma^{-1} \cdot \mu$$

$$g = -\frac{1}{2}\mu^T \Sigma^{-1} \mu - log((2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}})$$

The product of two canonical forms is:

$$\mathcal{C}(K_1, \mathbf{h}_1, g_1) \cdot \mathcal{C}(K_2, \mathbf{h}_2, g_2) = \mathcal{C}(K_1 + K_2, \mathbf{h}_1 + \mathbf{h}_2, g_1 + g_2)$$

This makes multiplying factors in message passing quite easy since it is reduced to simple additions.

A canonical form can always be rewritten back into a normal distribution by isolating Σ and μ in the second and third equation right above. Generally when working with canonical forms in the context of message passing the parameter g can be ignored. Instead of dealing with a single value for messages they can be split up into a J and h message. Here J is equal to the K from the definition.

$$\hat{J}_{ij} = J_{ii} + \Sigma_{k \in \mathcal{N}b_i - \{j\}} J_{k->i}$$
$$\hat{h}_{ij} = h_{ii} + \Sigma_{k \in \mathcal{N}b_i - \{j\}} h_{k->i}$$

$$J_{i->j} = -J_{ji} \hat{J}_{ij}^{-1} J_{ji}$$
$$h_{i->j} = -h_{ji} \hat{J}_{ij}^{-1} \hat{h}_{ij}$$

$$\hat{J}_i = J_{ii} + \Sigma_{k \in \mathcal{N}b_i} J_{k->i}$$
$$\hat{h}_i = h_i + \Sigma_{k \in \mathcal{N}b_i} h_{k->i}$$

$$\hat{\mu}_i = (\hat{J}_i^{-1} \hat{h}_i$$
$$\hat{\sigma}_i^2 = (\hat{J}_i)^{-1}$$

By using a Bethe-structured cluster graph were each node has its own cluster and each edge has cluster a message can be sent from node i to j by using the above equations.
We wanted to implement our efficient algorithm using the above message passing algorithm, however we did not manage to do this. We did manage to implement an inefficient inference algorithm based on matrix algebra given in chapter 7 of PGM. Theorem 7.4: Let $\{\mathbf{X}, Y\}$ be a joint distribution defined as:

$$p(\mathbf{X}, \mathbf{Y}) = \mathcal{N}(\begin{pmatrix} \mu_{\mathbf{X}} & \mu_{\mathbf{Y}} \end{pmatrix}; \begin{bmatrix} \Sigma_{\mathbf{XX}} & \Sigma_{\mathbf{XY}} \\ \Sigma_{\mathbf{YX}} & \Sigma_{\mathbf{YY}} \end{bmatrix})$$

The conditional distribution of Y conditioned on the set of variables $\mathbf{X}$ can be calculated as:

$$p(Y|\mathbf{X}) = \mathcal{N}(\beta_0 + \beta^T \mathbf{X}; \sigma^2)$$

where

$$\beta_0 = \mu_Y - \Sigma_{YX} \Sigma_{XX}^{-1} \mu_X$$
$$\beta = \Sigma_{XX}^{-1} \Sigma_{YX}$$

$$\sigma^2 = \Sigma_{YY} - \Sigma_{YX}\Sigma_{XX}^{-1}\Sigma_{XY}$$

We want to condition all the Z nodes on the entire set of leaves, so therefore we need the mean vector and covariance matrix for the tree given a set of parameters $(\alpha, \beta, \sigma^2, \alpha_0, \sigma_0^2)$.

Theorem 7.3: Let X and Y be defined as before, but let now X be the set of parents for Y. The conditional distribution of Y given X is:

$$p(Y|X = x) = \mathcal{N}(\beta_0 + \beta^T x; \sigma^2)$$

The distribution of Y is

$$p(Y) = \mathcal{N}({}_Y; \sigma_Y^2)$$
$${}_Y = \beta_0 + \beta^T \mu$$
$$\sigma_Y^2 = \sigma^{\beta T}\Sigma\beta$$

In our implementation we start at the root, which has mean α_0 and variance σ_0^2. Therefor its children mean and variance can be calculated from the above two formulas and so forth all the way to the leaves. When this is done the above paragraphs about conditioning can be used, it is slow but the inference produces fine results.

2.3 Test the inference algorithms using simulated data

Predictions are very close to ground truth values. We see the mean difference between prediction and true are very low compared to the gene lengths themselves (z0)
These parameters are used in part 2 inference. The linear regression is also trained on simulated data on these parameter

Parameters	
alpha_0	**50.000**
variance_0	**500.000**
alpha	**20**
beta	**1.03**
variance	**5.000**
Settings	
num_of_samples	**10.000**
root	**407**

Table 1: Inference vs LinReg on gene data

maxDiff	minDiff	meanDiff	z0Pred	z0True
264	0	77	49980	49870
358	1	90	50043	50037
321	0	75	49977	49909
339	1	84	50000	49993
448	1	95	50062	50078
281	0	80	50054	50114
372	0	77	50059	50159
375	0	95	50029	50073
401	0	91	50048	50108
371	0	78	50041	49987

2.4 Apply the inference algorithms on the data in the data file and present the results

We used these parameters for the inference simulation:
Using these two different methods to guess something that nobody knows (the z-values), it is comforting that the two predictions return numbers very close to each other. Therefore we assume that our prediction methods are somewhat accurate.

Table 2: Inference on forward simulation

z0Inference	z0linReg
79319	79958
34268	32592
15819	15580
11326	10858
4214	4322
23658	24426
47796	48255
9300	8392
21849	20845
5201	5016

3 Part 3: Learning of the parameters

For this part, we had to learn the parameters. To begin with, we learn the parameters from all of the data. This is done using linear regression. We were not able to implement a functional EM algorithm.

3.0.1 Linear Regression

We are not going to explain how linear regression works, because it is trivial, we have 408 variables and 408 constants for each parameter we're trying to predict.

$$parameter_i = a_{i,0} + a_{i,1} \cdot x_1 + .. + a_{i,204} \cdot x_2 04 + a_{i,205} \cdot z_0 + .. + a_{i,407 \cdot z_2 03}$$

where $i = \{\alpha, \beta, \sigma^2, \alpha_0\}$

Here are the learned parameters on test data using linear regression:

R2 on Alpha:	**0.997**
R2 on Beta	**0.9998501327152961**
R2 on Sigma	**-0.06085210093429927**

As can be seen in the table above, the linear regression for alpha and beta is indeed very good. Sigma is equally good as the constant function equal the mean of the y values, which is not good. Sigma changes the variance of the nodes, it can therefor affect the gene length of a node to be both smaller and bigger. A change in alpha or beta only affects the genes to be greater or smaller.
Afterwards, we wanted to learn the parameters only based on the leaves. This was done based on the pseudo-code below, which is called the hard-assignment EM algorithm.

3.0.2 EM algorithm

```
1  def EM_algorithm(tree, gene_lengths, alpha_0, alpha, beta, var_0, var,
   ↪  tol):
2      # gene_lengths: [X1,...,Xn]
3      old_params = [alpha, beta, var]
4      new_params = []
5
6      while True:
```

```
        # E-Step
        Z_expectations = []
        # loop over all Z's (hidden variables)
        for Z in tree.hidden_variables():
            Zi_expected = compute_expectation(Z, tree, gene_lengths,
            ↪  alpha_0, alpha, beta, var_0, var)
            Z_expectations.append(Zi_expected)

        # M-Step
        alpha, beta, var = update_params(Z_expectations, gene_lengths,
        ↪  alpha_0, beta_0, var_0)

        # Check convergence
        new_params = [alpha, beta, var]
        if check_convergence(old_params, new_params, tol):
            break

        old_params = new_params

    return new_params

```

A Appendix - Simulation Functions

A.1 def simulation

```
    def simulation(root: int, graph: dict, alpha0, variance0, alpha,
    ↪  beta, variance) -> dict:
    """
    BFS
    returns z value for all 407 nodes
    """
    gene_lengths = {}

    z0 = max(np.random.normal(alpha0, np.sqrt(variance0)), 1)
    queue = [(root, z0)]

    while queue:
        # handle the next element in queue
```

```
        node, z = queue.pop(0)

        # set node in results in genelength z
        gene_lengths[node] = z

        # get all children for the node
        children = graph.get(node, [])

        if not children:
            continue
        else:
            # for all the children for this node calculate their z
            # since we know it's a tree, only the parent can have
            ↪ influence
            for child in children:
                id, t = child
                t = round(t, 3)

                # draw a sample
                mean = (alpha * t) + (beta * z)
                var = variance * t


                cpd_z = max(np.random.normal(mean, np.sqrt(var)), 1)
                #print(f"id:{id}, t:{t}, var:{var}, std:{np.sqrt(var)},
                ↪ mean:{mean}, cpd:{cpd_z}")
                #if id == 205:
                #    print(var)
                #    print(mean)
                #    print(cpd_z)
                # append child to queue
                queue.append((id, cpd_z))

    return gene_lengths
```

A.2 def n_simulations

```
def n_simulations(n, root, graph, alpha0, variance0, alpha, beta, variance) -> tuple:
    """
    Returns results for X=X1 ... X204 n times, and y=Z0 n times s
    """
    X = np.empty((n, 204))
    Z = np.empty((n, 203))
    y = np.empty(n)

    for i in range(n):
        if type(alpha0) == np.dtype('object_'):
            results = simulation(root=root, graph=graph, alpha0=alpha0[i], variance0=variance0[i], alpha=alpha[i], beta=beta[i], variance=variance[i])
        else:
            results = simulation(root=root, graph=graph, alpha0=alpha0, variance0=variance0, alpha=alpha, beta=beta, variance=variance)
        # extract the first 204 values from the dictionary and add them to X as a row
        row_X = [results[key] for key in range(1, 205)]
        row_Z = [results[key] for key in range(205, 408)]

        X[i] = np.array(row_X)
        Z[i] = np.array(row_Z)


    return X, Z
```

B Appendix - LinReg

```
regAlpha = LinearRegression().fit(V[:train_size,:], alpha[:train_size])
regBeta = LinearRegression().fit(V[:train_size,:], beta[:train_size])
regVariance = LinearRegression().fit(V[:train_size,:], variance[:train_size])

```

```
# Train results
yt_pred = regAlpha.predict(V[train_size:,:])
train_r2 = regAlpha.score(V[train_size:,:], alpha[train_size:])
train_mse = mean_squared_error(alpha[train_size:], yt_pred)

yt_pred1 = regBeta.predict(V[train_size:,:])
yt_pred2 = regVariance.predict(V[train_size:,:])
```