

笔记 II: fastSTRUCTURE 包

Jonathan Chow

2022-09-29

目录

1	fastSTRUCTURE 包	1
2	参数更新函数	2
2.1	admixprop.pyx 和 C_admixprop.c	2
2.2	allelefreq.pyx 和 C_allelefreq.c	2
3	损失函数	4
4	主函数	5
5	绘图函数	6

1 fastSTRUCTURE 包

fastSTRUCTURE 包通过 python 实现使用变分推断算法 (VI) 拟合 PSD 模型。

主函数: `fastStructure.pyx`。

参数更新函数: `admixprop.pyx`、`allelefreq.pyx` 以及 `C_admixprop.c`、`C_allelefreq.c`。

损失函数: `marglikehood.pyx`、`C_marglikehood.c`。

绘图函数: `distruct.py`。

2 参数更新函数

2.1 `admixprop.pyx` 和 `C_admixprop.c`

`admixprop.pyx` 和 `C_admixprop.c` 是更新个体的种群分布矩阵的函数, 在 `pyx` 文件中实现算法类的定义, 在 `c` 文件中实现核心的参数更新以加快运算速度。

我们用的是一个参数为 $1/K$ 的对称 Dirichlet 分布作为先验。

我们还使用了平方更新的方式以加快 VI 的收敛。

```
cdef class AdmixProp:
    def __cinit__(self, long N, long K):
    cdef copy(self):
    cdef require(self):
    cdef update(self, np.ndarray[np.uint8_t, ndim=2] G,
                  af.AlleleFreq pi):
    cdef square_update(self, np.ndarray[np.uint8_t, ndim=2] G,
                      af.AlleleFreq pi):

void Q_update(const uint8_t* G, const double* zetabeta,
              const double* zetagamma, const double* xi,
              double* new_var, long N, long L, long K)
```

2.2 `allelefreq.pyx` 和 `C_allelefreq.c`

`allelefreq.pyx` 和 `C_allelefreq.c` 是更新种群的基因分布的函数, 在 `pyx` 文件中实现算法类的定义, 在 `c` 文件中实现核心的参数更新以加快运算速度。

我们使用两种先验。第一是简单先验，即一个参数为 (1,1) 的 Beta 分布。事实上在此基础上调整参数可以实现 F 先验，即以 F 统计量的函数作为参数的 Beta 分布。第二是逻辑斯蒂先验。由于此时先验包含了超参数，我们在每一次迭代之后还应同步更新超参数。

我们还使用了平方更新的方式以加快 VI 的收敛。

```
cdef class AlleleFreq:
    def __cinit__(self, long L, long K, str prior):
    cdef copy(self):
    cdef require(self):
    cdef _update_simple(self, np.ndarray[np.uint8_t, ndim=2] G,
                        ap.AdmixProp psi):
    cdef _update_logistic(self, np.ndarray[np.uint8_t, ndim=2] G,
                        ap.AdmixProp psi):
    cdef _unconstrained_solver(self, np.ndarray[np.float64_t, ndim=2] Dvarbeta,
                        np.ndarray[np.float64_t, ndim=2] Dvargamma):
    cdef update(self, np.ndarray[np.uint8_t, ndim=2] G, ap.AdmixProp psi):
    cdef square_update(self, np.ndarray[np.uint8_t, ndim=2] G,
                        ap.AdmixProp psi):
    cdef update_hyperparam(self, bool nolambda):

void P_update_simple(const uint8_t* G, const double* zetabeta,
                    const double* zetagamma, const double* xi,
                    const double* beta, const double* gamma,
                    double* var_beta, double* var_gamma,
                    long N, long L, long K)

void P_update_logistic(const double* Dvarbeta, const double* Dvargamma,
                      const double* mu, const double* Lambda,
                      double* var_beta, double* var_gamma,
                      double mintol, long L, long K)
```

3 损失函数

`marglikehood.pyx` 和 `C_marglikehood.c` 是计算 ELBO 的函数, ELBO 可以分为三个部分, 分别记作 E1、E2 和 E3。在 `pyx` 文件中我们计算需要使用 Γ 函数同时不需要复杂循环的部分, 即 E2 和 E3, 这是因为我们可以使用 python 的 `numpy` 库方便地进行科学计算, 这是 C 所不具备的。在 `c` 文件中我们计算不需要使用 Γ 函数同时需要复杂循环的部分, 即 E1, 这是因为我们可以利用 C 循环的效率以加快运算速度。

```
def marginal_likelihood(np.ndarray[np.uint8_t, ndim=2] G,
                        ap.AdmixProp psi, af.AlleleFreq pi):

    cdef double E1, E2, E3, Etotal

    E1 = marglikehood(<np.uint8_t*> G.data,
                      <double*> pi.zetabeta.data, <double*> pi.zetagamma.data,
                      <double*> psi.xi.data,
                      psi.N, pi.L, pi.K)

    E2 = (utils.insum(gammaln(psi.var) - gammaln(psi.alpha)
                      - (psi.var-psi.alpha)
                      * np.nan_to_num(np.log(psi.xi)), [1])
          - gammaln(utils.insum(psi.var, [1]))
          + gammaln(utils.insum(psi.alpha, [1]))).sum()

    if pi.prior=='simple':
        E3 = (gammaln(pi.var_beta) - gammaln(pi.beta)
              - (pi.var_beta-pi.beta)
              * np.nan_to_num(np.log(pi.zetabeta))
              + gammaln(pi.var_gamma) - gammaln(pi.gamma)
              - (pi.var_gamma-pi.gamma)
              * np.nan_to_num(np.log(pi.zetagamma))
              - gammaln(pi.var_beta+pi.var_gamma)
              + gammaln(pi.beta+pi.gamma)).sum()
```

```

elif pi.prior=='logistic':
    diff = digamma(pi.var_beta)-digamma(pi.var_gamma)-pi.mu
    E3 = 0.5*pi.L*np.log(pi.Lambda).sum()
        - 0.5*(pi.Lambda*utils.outsum(diff**2)).sum()
        - 0.5*(pi.Lambda*utils.outsum(polygamma(1,pi.var_beta)
            + polygamma(1,pi.var_gamma))).sum()
        - np.sum(utils.nplog(pi.zetabeta)
            + utils.nplog(pi.zetagamma))
        + ((pi.var_beta>0)*gammaln(pi.var_beta)
        - (pi.var_beta-1)*utils.nplog(pi.zetabeta)
        + (pi.var_gamma>0)*gammaln(pi.var_gamma)
        - (pi.var_gamma-1)*utils.nplog(pi.zetagamma)
        - gammaln(pi.var_beta+pi.var_gamma)).sum()

    Etotal = (E1 + E2 + E3)/float(psi.N*pi.L)

return Etotal

```

```

double marglikelihood(const uint8_t* G,
                    const double* zetabeta, const double* zetagamma,
                    const double* xi,
                    long N, long L, long K)

```

4 主函数

`fastStructure.pyx` 是主函数，分为三个函数。第一是参数更新与拟合模型的函数，第二是根据拟合预测未知基因型的函数，第三是计算交叉验证误差的函数。

```

def infer_variational_parameters(np.ndarray[np.uint8_t, ndim=2] G,
                                int K, str outfile, double mintol,
                                str prior, int cv):

```

```
cdef double expected_genotype(ap.AdmixProp psi, af.AlleleFreq pi,
                              int n, int l):

cdef np.ndarray CV(np.ndarray[np.uint8_t, ndim=2] Gtrue,
                  ap.AdmixProp psi, af.AlleleFreq pi,
                  int cv, double mintol):
```

5 绘图函数

`distruct.py` 是绘制结构图的函数，主要分为制作数据集和作图两部分。

```
def plot_admixture(admixture,
                  population_indices, population_labels,
                  title):

def get_admixture_proportions(params):
```