「先進ゲノム支援」情報解析講習会受講報告

○加藤大和

生物・生体技術支援室　生物機能解析・実験実習技術グループ

はじめに

次世代シークエンサーの普及により、生命科学分野の技術支援においてバイオインフォマティクススキルの重要性がますます高まっているが、本学においてウェットとドライの両方のスキルを併せ持つ技術職員は少ない。全学技術センターのさらなるサービス機能強化のため、2019 年 10 月に国立遺伝学研究所で開催された「先進ゲノム支援」情報解析講習会に参加したので、その概要と習得技術の業務への活用例について報告する。

1　講習の概要

最初に講習の概要を簡単に紹介する。本講習は若手研究者や技術者を対象としてバイオインフォマティクスのリテラシーを向上させることを目的としたものである。講習で使用するソフトウェア等の動作を統一するため、参加者は事前に自分の PC に Oracle VM VirtualBox と仮想マシン OS として Ubuntu (ver. 18 以降)をインストールし、Ubuntu 上に Python の仮想環境として Anaconda を導入することが指示されていた。講習は 3 日間にわたり、自分の PC 上で Linux 端末と Jupyter Notebook を使って、1 日目に Python 基本文法の復習、2 日目に表形式ファイルの処理、データ補正と視覚化、統計的仮説検定など、最終日には多変量解析を扱った。自分でスクリプトを打ち込む場面が多々あり、非常に実践的な講習であった。幾分難解な内容も多かったが、すぐに業務に活用できそうなものもあり、研究者だけでなく生命科学関連の技術職員が受講するのにも適した内容であると感じた。バイオインフォマティクスに興味のある方には是非とも受講をお勧めする。

2　業務への活用例

ここでは今年度の奨励研究によるイネゲノムリシーケンシング解析から得た変異アリルのレコードが記録された VCF (Variant Call Format)ファイルを、Python スクリプトと Jupyter Notebook を使って対話的に解析した例を紹介する。VCF ファイルはテキスト情報がカンマで区切られた、いわゆる csv 形式になっている。扱う VCF ファイルは 2 つあり、それぞれ新規変異体とその背景系統である野生型コシヒカリのものである。目的は変異体アリルのレコードの中から、背景系統であるコシヒカリには存在しない変異体特異的アリルレコードを抽出することである。変異体の原因遺伝子は核に存在するものと想定している。次世代シーケンサーから得られる情報量は膨大であり、今回の VCF ファイルもそれぞれ 300 万以上のレコードが含まれているため、例えば Excel のスプレッドシートなどで簡便に取り扱うことができない。以下に実際の Python スクリプトとその Jupyter Notebook 上での実行結果について説明していく。なお紙面の都合上、スクリプトがすべて記載できていないスクリーンショットもあるがご了承頂きたい。

まず 2 つのファイルのファイルパスを指定し、次に表計算に必要なライブラリをインポートしている。それから変異体と野生型の VCF ファイルを pandas データフレームに読み込む。今回用いる VCF ファイルの第 0 列は数値と文字列が含まれた mixed types であるため、警告が出ないように low_memory=False を指定する。読み込み後、後でデータフレームを結合する際に変異体か野生型どちらのアリルか判別できるように、新し

い列”WorM” (値は‘MT または‘WT’) を追加しておく。そしてそれぞれのデータフレームの行数と列数を表示している。上述したようにどちらのファイルも 300 万以上のレコードがあり、ともに列数が 34+1 であると分かる。

In [2]:
```
mutant = "C:¥¥Users¥¥Hirokazu¥¥Desktop¥¥2020Gijyutu¥¥aal4_SNV.csv"
koshi = "C:¥¥Users¥¥Hirokazu¥¥Desktop¥¥2020Gijyutu¥¥Koshihikari_SNV.csv"

import pandas as pd
import numpy as np

df_mutant = pd.read_csv(mutant, low_memory=False)
df_koshi = pd.read_csv(koshi, low_memory=False)

df_mutant["WorM"] = 'MT'
df_koshi["WorM"] = 'WT'

print(df_mutant.shape, df_koshi.shape)
```

```
(3050193, 35) (3564240, 35)
```

変異体のデータフレームの内容を表示させてみる。このようにスクリプトを逐一実行しながら結果をすぐに目視で確認きるのでプログラミング初心者、初級者にとっては非常に便利だ。

In [3]:
```
df_mutant
```

Out[3]:

	Chromosome	Region	Type	Reference	Allele	Reference allele	Length	Linkage	Zygosity
0	1	1010	SNV	A	A	Yes	1	NaN	Heterozygous
1	1	1045	SNV	A	A	Yes	1	NaN	Heterozygous
2	1	1052	SNV	A	A	Yes	1	NaN	Heterozygous
3	1	1151	SNV	C	A	No	1	NaN	Homozygous
4	1	1327	SNV	G	G	Yes	1	NaN	Heterozygous
5	1	2223	SNV	T	T	Yes	1	NaN	Heterozygous
6	1	2374	SNV	T	A	No	1	NaN	Heterozygous
7	1	2374	SNV	T	T	Yes	1	NaN	Heterozygous

先にも述べたように、”Chromosome”列の要素は数値（染色体番号 1~12）または文字列 (‘chloroplast’/’mitochondrion’) になっているため、明示的に列の型を文字列 str に指定しておく。さらに、”Region”列の要素は”Type”列の要素に依存して数値の場合と文字列の場合があるので、この列の型も明示的に文字列 str に指定しておく。このようにしておくことで、この後の不必要なレコードの削除や染色体番号による抽出操作が意図したよう動いてくれる。この 2 つの準備が整ったら、必要のないレコードをデータフレームから削除する。イネの場合、日本晴のゲノムシークエンスがリファレンスとなっていて、その最新バージョンは IRGSP-1.0 である。VCF ファイルのレコードには変異アリルに加えて、Heterozygous なアリルの場合などはそれに対応する野生型日本晴のアリルも含まれているため、これらのレコードを削除する。クロロプラストとミトコンドリアのゲノムにコールされたレコードも削除する。さらに、特に 4 番染色体に多いのだが、日本晴リファレンスが’N’となっているためにコールされたアリルもレコードに含まれているため、これらも削除している。削除後のデータフレームのレコード数は約 1/3 程度になっている。

In [3]:
```
# ここで明示的に'Chromossome'列とRegion'列の型を'str'にしておく。
df_mutant['Chromosome'].astype('str')
df_koshi['Chromosome'].astype('str')
df_mutant['Region'].astype('str')
df_koshi['Region'].astype('str')
# これで後のdrop_duplicates()で重複レコードの削除が上手くいくようになった。
print(df_mutant['Chromosome'].dtypes, df_koshi['Chromosome'].dtypes)
print(df_mutant['Region'].dtypes, df_koshi['Region'].dtypes)
```

```
object object
int64 int64
```

In [4]:
```
df_mutant = df_mutant[(df_mutant.Chromosome != 'chloroplast') &
                      (df_mutant.Chromosome != 'mitochondrion') &
                      (df_mutant.Reference != 'N') &
                      (df_mutant["Reference allele"] == 'No')]
df_koshi = df_koshi[(df_koshi.Chromosome != 'chloroplast') &
                    (df_koshi.Chromosome != 'mitochondrion') &
                    (df_koshi.Reference != 'N') &
                    (df_koshi["Reference allele"] == 'No')]

print(df_mutant.shape, df_koshi.shape)
```

```
(1027839, 35) (1145536, 35)
```

必要なレコードのみを持った2つのデータフレームをappend()メソッドで結合する。そしてdrop_duplicates()メソッドで重複したレコードを削除する。”Chromosome”列、“Region”列、“Type”列、“Allele”列のすべてが同じ値のレコードを同一のアリルと判定する。変異体アリル、野生型アリル両方のレコードを削除するために、keep引数にFalseを指定している。

In [6]:
```
df_mutant_koshi_app = df_mutant.append(df_koshi, sort=True)[df_mutant.columns.tolist()]
print(df_mutant_koshi_app.shape)
```

```
(2173375, 35)
```

In [7]:
```
df_mutant_koshi_unique = df_mutant_koshi_app.drop_duplicates(subset=['Chromosome', 'Regio
                                                             keep = False)

df_mutant_koshi_unique.shape
```

Out[7]:
```
(1217551, 35)
```

この時点では、変異体特異的アリルと野生型特異的アリルの両方がデータフレームに残っている。変異体とコシヒカリそれぞれについて染色体ごとのデータフレームのリストを作りつつ、出来たデータフレームにレコードを分けていく。上述したように、”Region”列の型をstr型にしたため、染色体番号も文字列として比較している。変異体と野生型の判別には先ほど自分で追加した”WorM”列を使う。

In [8]:
```
mutant_df_list = list()
koshi_df_list = list()

for i in range(1, 13):
    mutant_df_list.append(pd.DataFrame())
    koshi_df_list.append(pd.DataFrame())

    mutant_df_list[i-1] = df_mutant_koshi_unique[(df_mutant_koshi_unique.Chromosome == st
                                                  (df_mutant_koshi_unique.WorM == 'MT')]
    koshi_df_list[i-1] = df_mutant_koshi_unique[(df_mutant_koshi_unique.Chromosome == str
                                                 (df_mutant_koshi_unique.WorM == 'WT')]
```

試しに変異体に特異的なアリルがどのくらい残ったかを計算してみる。

```
num_of_mutant_sp = 0
for i in range(1, 13):
    num_of_mutant_sp += len(mutant_df_list[i - 1])

print(num_of_mutant_sp)
```

```
549927
```

約 55 万カ所の変異体に特異的なアリルが抽出できたようだ。いよいよグラフの描画に入る。グラフはあくまでも染色体上でアリル頻度の高い領域を俯瞰するために使うので、アリル頻度"Frequency"列とカバレッジ"Coverage"列のカットオフ値を好きなように設定してプロット数を調整する。グラフの描画には Matplotlib ライブラリを利用する。イネは 12 本の染色体を持っているため、2 つの figure を作り、1 つの figure に 6 枚のグラフを 3 行×2 列で表示することにする。

In [10]:
```
allele_freq = 40 # 基準とするアリル頻度
coverage = 9 # 基準とするカバレッジ

mutant_df_list_plot = list()

for i in range(1, 13):

    mutant_df_list_plot.append(pd.DataFrame())
    mutant_df_list_plot[i - 1] = mutant_df_list[i - 1][(mutant_df_list[i - 1].Frequency >
                                                        (mutant_df_list[i - 1].Coverage >=

import matplotlib.pyplot as plt

# 描画の準備
row_num = 3 # グラフの行数
col_num = 2 # グラフの列数
graph_num = row_num * col_num # 1 figureに6枚のグラフを描画
axes1 = [] # 空のリスト、figureインスタンスを入れる
axes2 = []
```

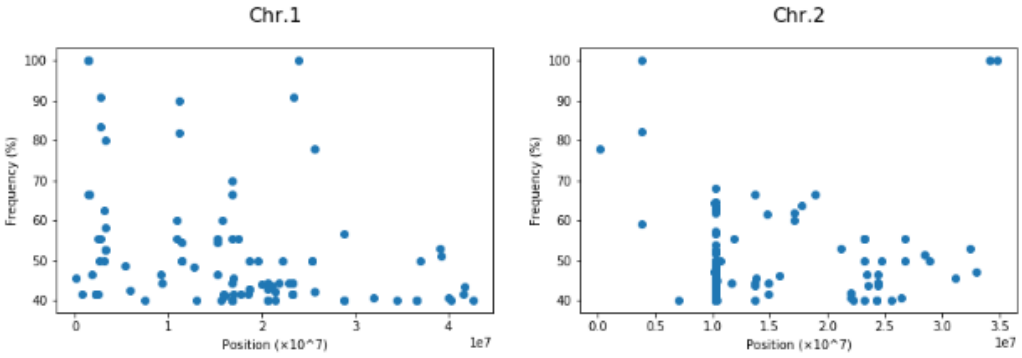
figure インスタンスを作成して、まず染色体 1 番から 6 番までの散布図を描く。x 軸に染色体上の位置を示す”Region”列、y 軸にアリル頻度”Frequency”列を指定している。

```
# figure（描画領域）の設定
fig = plt.figure(1, figsize = (14, 15)) # サイズを指定してfigureインスタンスを作成

for i in range(1, graph_num + 1):

    axes1.append(fig.add_subplot(row_num, col_num, i))

    axes1[i - 1].scatter(mutant_df_list_plot[i - 1].Region, mutant_df_list_plot[i - 1].Fr

    # グラフタイトルと軸の設定
    title = "Chr." + str(i)
    axes1[i - 1].annotate(title, xy=(0.5, 1.1), fontsize=16, xycoords='axes fraction',
                          horizontalalignment='center')
    axes1[i - 1].set_ylabel("Frequency (%)")
    axes1[i - 1].set_xlabel("Position (×10^7)")

# グラフ間の隙間調整
fig.subplots_adjust(wspace=0.2, hspace=0.4)
```

ブラウザにアリル頻度をプロットしたグラフを表示させる。描画したグラフは解像度などを指定して.png 形式のファイルとして保存することができる。

```
# ブラウザに表示
plt.show()
# 300 dpiの解像度でpng形式のファイルに保存
fig.savefig("300_dpi_Chr1-6.png", format="png", dpi=300)
```

さらに残りの 7 番染色体から 12 番染色体のグラフも同じように描く（スクリプトは省略）。実際に Jupyter Notebook 上に描かれた 12 枚のグラフのうち 1 番染色体と 2 番染色体の散布図を以下に示す。



グラフからそれとなくアリル頻度の高そうなゲノム領域が垣間見えてきた。この後は変異体の原因遺伝子の候補を探索する。

In [11]:
```
frequency = 80 # 基準とするアリル頻度
cover = 5 # 基準とするカバレッジ

mutant_df_pickup = list()

mutant_df_pickup_all = pd.DataFrame()

for i in range(1, 13):
    mutant_df_pickup.append(pd.DataFrame())

    mutant_df_pickup[i - 1] = mutant_df_list[i - 1][ ( (mutant_df_list[i - 1].Frequency >
                                                        (mutant_df_list[i - 1].Coverage >= co
                                                        (mutant_df_list[i - 1]["Non-synonymou
                                                        ( (mutant_df_list[i - 1].Frequency >=
                                                        (mutant_df_list[i - 1].Coverage >= co
                                                       (mutant_df_list[i - 1]["Splice effect"
                                                        "Possible splice site disruption") )

    mutant_df_pickup_all = mutant_df_pickup_all.append(mutant_df_pickup[i - 1])

    mutant_df_pickup[i - 1].to_csv('mutant_df_pickupChr%d.csv'%i, index = False)

mutant_df_pickup_all.to_csv('mutant_df_pickup_all.csv', index = False)

# 抽出されたアリルレコードの表示（1番染色体の例）
mutant_df_pickup[0]
```

ここではアリル頻度"Frequency"が設定値以上でカバレッジ"Coverage"が設定数以上、かつ遺伝子のコード領域内でアミノ酸置換やフレームシフトを伴うか、または推定上のスプライシングサイトに当たっているアリルレコードを新しいデータフレームに抽出している。各設定値は研究者の好みに合わせて決める。抽出されたレコードはブラウザ上に表示させることもできるし、染色体ごとに.csv ファイルとして保存することもできる。この後はアリル頻度が高そうな領域内で、アリル頻度が 100%に近いものを選んで、IGV などのゲノム

ブラウザで.bam ファイルを見ながら変異体の原因アリルの候補をさらに絞っていくことになる。当然のことながら、次世代シーケンサーを用いたゲノムリシーケンシング解析は生物学実験であるから、用いたサンプル DNA の純度、ライブラリーのクオリティ、シーケンサーやリードのマッピングに用いたソフトウェアなどの違いが結果に影響を与える。今回作成した Python スクリプトで抽出された変異体特異的アリルのうちの幾つかは、ただ単に変異体ではマップされたリードがある領域で野生型ではリードが一つもないというものであった。本来なら野生型でも複数のリードがあり、かつ変異が存在しない変異体アリルレコードを抽出すべきだが、野生型でリードがあって変異がない場所は VCF ファイルにコールされないため、野生型にリードがあるかないかを VCF ファイルのみから区別することはできない。

おわりに

これまで VCF ファイルからの変異体特異的アリルの抽出には自作の Perl スクリプトを使っていたが、本講習会受講後、Python と Jupyter Notobook を導入することにより、同様のことをより効率的に、視覚的に確認しながら確実に進めることができるようになった。今後の全学技術センターの技術支援サービスの効率化と高度化に向けて非常に有意義な講習会であった。本講習に参加できるようにご配慮をいただいた全学技術センター企画室と研修係の皆様に深く感謝致します。