Copy_Number_Project_2.1

November 14, 2017

1 This purpose of this notebook is to predict DLBCL subtype (ABC, GCB, or Unclassified) from patient chromosome copy number data

The overview will consist of two main ways to look at the data:

- Summing across every mutation within a chromosome (24 features).
- Treating every mutation as a predictor (>50,000 features).

We will predict DLBCL subtype using the following statistical methods:

- Logistic Regression
- K Nearest Neighbors
- Random Forests
- Naive Bayes

First, we will examine the copy number data according to 1.)

```
In [30]: import pandas as pd
         import os
         import re
         import xlsxwriter

         #Set up an empty data frame
         df1 = pd.DataFrame()

         #Read in all .txt patient chromosome copy number files into a data frame
         for filename in os.listdir('.'):
             if filename.endswith('.txt'):
                 filename_df = pd.read_table(filename, names = ['chr','start','stop',
                                             'copynumber'], index_col = 'chr')
                 filename_df.drop(['start','stop'], axis=1, inplace=True)
                 filename_df = filename_df.groupby('chr').sum()
                 filename_df = filename_df.T
                 filename_df['Sample'] = re.sub('\.txt$', '' , str(filename))
                 filename_df.set_index('Sample', inplace=True)
                 df1 = df1.append(filename_df)
```

```
        df1 = df1.iloc[:,0:24]

    #Merge another data frame to include the DLBCL Subtype in the same data frame
    df2 = pd.read_excel('DLBCL_Subtype.xlsx')
    df2.set_index('Sample', inplace = True)
    df1 = df1.join(df2, how='left')
    df1.reset_index(inplace=True)
    del df1['Sample']
    df1.dropna(inplace=True)
    df1['DLBCL Subtype Map'] = df1['DLBCL Subtype'].map({'ABC' : 1, 'GCB' : 0,
                                                         'Unclass' : 2})
    print(df1.head())
    print(df1.shape)
```

```
   chr1  chr10  chr11  chr12  chr13  chr14  chr15  chr16  chr17  chr18  \
0   2.0    0.0    0.0    0.0    0.0    1.0   -2.0   -2.0    2.0    0.0
1  -4.0    1.0    0.0   -2.0   -4.0   -2.0   -2.0    0.0    3.0    3.0
2  -1.0   -1.0   -3.0    2.0   -2.0   -5.0   -3.0   -2.0   -3.0    3.0
3  -6.0   -2.0    0.0   -2.0   -1.0    1.0   -2.0   -2.0   -2.0    6.0
4  -3.0   -1.0   -5.0    2.0   -2.0   -4.5   -6.0   -1.0    5.0   15.0

        ...          chr4  chr5  chr6  chr7  chr8  chr9  chrX  chrY  \
0       ...           0.0   7.0  -3.0   0.0   0.0  -2.0   4.0   0.0
1       ...          -6.0   0.0   1.0  -1.0   3.0  -3.0  -1.0   0.0
2       ...          -1.0   2.0 -10.0   1.0  -3.0  -2.0   5.0   1.0
3       ...          -5.0  -1.0  -5.0   6.0   0.0  -2.0   1.0  -1.0
4       ...          -2.0   2.0  -5.0  -6.0  -2.0  -4.0   6.0  -2.0

   DLBCL Subtype  DLBCL Subtype Map
0            ABC                  1
1            ABC                  1
2            ABC                  1
3            ABC                  1
4            ABC                  1

[5 rows x 26 columns]
(539, 26)
```

As we can see, this data frame now contains all of our predictors (chromosomal insertion/deletion sums) and our response (DLBCL Subtype).

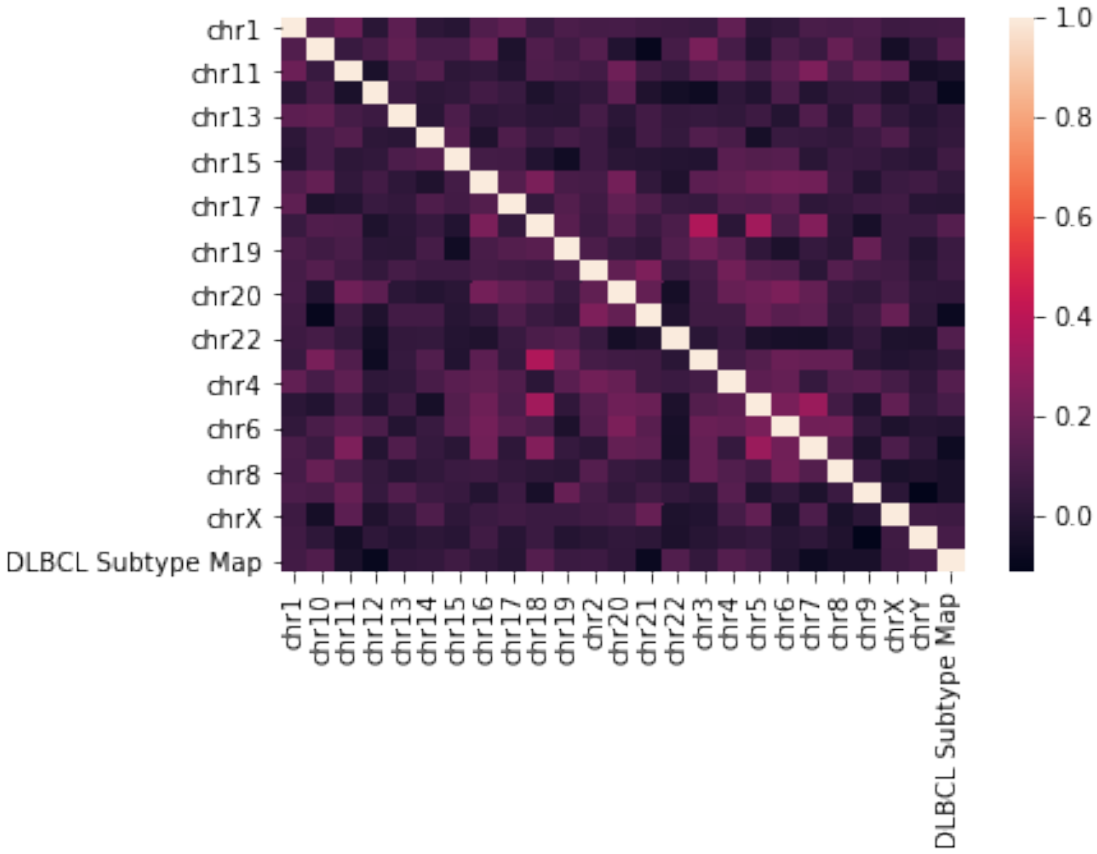
There a 25 columns and 539 rows. We are now in a good position to use ScikitLearn for make an attempt to find an accurate classifier of DLBCL Subtype.

First, let's examine the correlation of the data to see if any chromosome copy numbers correlate with each other.

```
In [31]: import matplotlib.pyplot as plt
         import seaborn as sns
         %matplotlib inline

         sns.heatmap(df1.corr())
```

```
Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x111c9d160>
```



There does not seem to be any strong correlations. Let's look at the null accuracy of guessing 'ABC' for every subtype, since this is the majority. If we can not beat this metric, then our model is useless.

```
In [32]: print(df1['DLBCL Subtype'].value_counts())

         print('\n' + 'The overall null accuracy rate for the whole data set is: '
               + str(round(278 / (278 + 154 + 107)*100)) + str('%'))
```

```
ABC        278
GCB        154
Unclass    107
```

```
Name: DLBCL Subtype, dtype: int64

The overall null accuracy rate for the whole data set is: 52%
```

We want to beat the null accuracy rate of 52%. Let's start with logistic regression. We will use train, test, split to check between the different models and later turn to cross-validation.

```
In [33]: from sklearn.linear_model import LogisticRegression
         from sklearn.model_selection import train_test_split
         from sklearn.metrics import accuracy_score

         X = df1[['chr1', 'chr10', 'chr11', 'chr12', 'chr13', 'chr14', 'chr15', 'chr16',
                'chr17', 'chr18', 'chr19', 'chr2', 'chr20', 'chr21', 'chr22', 'chr3',
                'chr4', 'chr5', 'chr6', 'chr7', 'chr8', 'chr9', 'chrX', 'chrY']]

         y = df1['DLBCL Subtype Map']

         logreg = LogisticRegression(random_state = 42)
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 42)
         logreg.fit(X_train, y_train)
         y_pred = logreg.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.614814814815
```

We can already see that running a logistic regression already beats our null accuracy guesses by 10%. Let's play around with adjusting some of the parameters, initially by changing the values for the l2 penalty (The 'C' paramteter). From comparing L1 and L2 penalties for several different values, the model does not seem to approve.

Let's move on to a new model: K Nearest Neighbors.

```
In [34]: from sklearn.neighbors import KNeighborsClassifier
         knn = KNeighborsClassifier()
         X_train, X_test, y_train, y_test = train_test_split(X,y, random_state = 42)
         knn.fit(X_train, y_train)
         y_pred = knn.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.481481481481
```

The initial run of K Nearest Neighbors performs worse than the null accuracy! Let's see if we can improve it by adjusting the number of neighbors parameter.

```
In [35]: def knn_accuracy(num):
             knn = KNeighborsClassifier(num)
             X_train, X_test, y_train, y_test = train_test_split(X,y, random_state = 42)
             knn.fit(X_train, y_train)
             y_pred = knn.predict(X_test)
             print(accuracy_score(y_test, y_pred))

         knn_accuracy(40)
```

```
0.585185185185
```

Changing the number of nearest neighbors to 40 improves the model over the null accuracy by ~4%.

Let's try a new model: random forests.

```
In [36]: from sklearn.ensemble import RandomForestClassifier
         treeclass = RandomForestClassifier(max_depth = 90, random_state = 42)
         X_train, X_test, y_train, y_test = train_test_split(X,y,random_state = 42)
         treeclass.fit(X_train, y_train)
         y_pred = treeclass.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.562962962963
```

Let's now move on to naive bayes.

Naive bayes requires non-negative numbers, so we need to add numbers to the copy number data frame so that there are no negative numbers. If we add a constant number to every row, it shouldn't change the data.

```
In [37]: X = X + 100
```

```
In [38]: from sklearn.naive_bayes import MultinomialNB
         X_train, X_test, y_train, y_test = train_test_split(X,y, random_state = 42)
         nb = MultinomialNB()
         nb.fit(X_train, y_train)
         y_pred = nb.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.533333333333
```

We can see that the default Naive Bayes classifer doesn't really beat the null accuracy. Let's try adjusting the alpha parameter to see if we can make our model any more accurate. The alpha parameter seems to have no effect.

Let's take the absolute value of X instead of adding 100 and see if that changes our accuracy score.

```
In [39]: X = X - 100
         X = abs(X)

In [40]: from sklearn.naive_bayes import MultinomialNB
         X_train, X_test, y_train, y_test = train_test_split(X,y, random_state = 42)
         nb = MultinomialNB()
         nb.fit(X_train, y_train)
         y_pred = nb.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.533333333333
```

Nothing happened. Let's see how else we can improve our prediction accuracy.

Alright, so now it is time to see if we can improve our model by changing our data. Instead of summing all the mutations for each chromosome, let's create a new column for every chromosomal mutation. We will save this data frame as df2, so that the other data frame is still maintained as df1.

Warning: This code takes ~5 minutes to run because the data frame it generates is so large.

```
In [41]: df2 = pd.DataFrame()

         for filename in os.listdir('.'):
             if filename.endswith('.txt'):
                 filename_df = pd.read_table(filename, names =
                                 ['chr','start','stop','copy_number'], header=1)

                 filename_df['start'] = filename_df['start'].astype(str)
                 filename_df['stop'] = filename_df['stop'].astype(str)

                 filename_df['copynumber_master'] = filename_df['chr'] + '_' + \
                                     filename_df['start'] + '_' + filename_df['stop']

                 filename_df.drop(['start','stop', 'chr'], axis=1, inplace=True)
                 filename_df = filename_df.groupby('copynumber_master').sum()
                 filename_df = filename_df.T
                 filename_df['Sample'] = re.sub('\.txt$', '' , str(filename))
                 filename_df.set_index('Sample', inplace = True)
                 df2 = df2.append(filename_df)
                 df2.fillna(0, inplace = True)

In [42]: df = pd.read_excel('DLBCL_Subtype.xlsx')
         df.set_index('Sample', inplace=True)
```

```
        df2 = df2.join(df, how = 'left')
        df2.reset_index(inplace=True)
        del df2['Sample']
        df2.dropna(inplace=True)
```

```
In [43]: print(df2.head())
```

```
   chr10_100573689_100581212  chr10_100581212_100710076  \
0                        0.0                        0.0
1                        0.0                        0.0
2                        0.0                        0.0
3                        0.0                        0.0
4                        0.0                        0.0

   chr10_100710076_100965696  chr10_100782024_135516692  \
0                        0.0                        0.0
1                        0.0                        0.0
2                        0.0                        0.0
3                        0.0                        0.0
4                        0.0                        0.0

   chr10_100965696_101060063  chr10_101009889_135516692  \
0                        0.0                        0.0
1                        0.0                        0.0
2                        0.0                        0.0
3                        0.0                        0.0
4                        0.0                        0.0

   chr10_101060063_126321496  chr10_101291168_101874591  \
0                        0.0                        0.0
1                        0.0                        0.0
2                        0.0                        0.0
3                        0.0                        0.0
4                        0.0                        0.0

   chr10_101292720_101349143  chr10_101310596_101840758      ...        \
0                        0.0                        0.0      ...
1                        0.0                        0.0      ...
2                        0.0                        0.0      ...
3                        0.0                        0.0      ...
4                        0.0                        0.0      ...

   chrY_9390879_28809923  chrY_9432191_9443799  chrY_9439059_9441690  \
0                    0.0                   0.0                   0.0
1                    0.0                   0.0                   0.0
2                    0.0                   0.0                   0.0
3                    0.0                   0.0                   0.0
4                    0.0                   0.0                   0.0
```

```
   chrY_9441690_22222983  chrY_9441690_28809923  chrY_9443799_9787623  \
0                    0.0                    0.0                   0.0
1                    0.0                    0.0                   0.0
2                    0.0                    0.0                   0.0
3                    0.0                    0.0                   0.0
4                    0.0                    0.0                   0.0

   chrY_9758611_13934470  chrY_9774900_22066345  chrY_9787623_28809923  \
0                    0.0                    0.0                    0.0
1                    0.0                    0.0                    0.0
2                    0.0                    0.0                    0.0
3                    0.0                    0.0                    0.0
4                    0.0                    0.0                    0.0

   DLBCL Subtype
0            ABC
1            ABC
2            ABC
3            ABC
4            ABC

[5 rows x 54620 columns]
```

We now have a new data frame (df2) that has every single mutation across all chromosomes as predictors.

```
In [44]: df2.shape
```

```
Out[44]: (539, 54620)
```

We have 539 rows and 54,620 columns.

Let's re-run the same models we did for df1 on df2, starting with logistic regression. We will first need to re-format X and y to match the new data frame.

```
In [45]: X = df2.iloc[:,0:54619]
         y = df2.iloc[:, 54619]
         print(X.shape)
         print(y.shape)
```

```
(539, 54619)
(539,)
```

```
In [46]: X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 42)
         logreg = LogisticRegression()
```

```
        logreg.fit(X_train, y_train)
        y_pred = logreg.predict(X_test)
        print(accuracy_score(y_test, y_pred))
```

```
0.666666666667
```

The default logistic regression model is predicting DLBCL subtype 11% better than the null accuracy. Let's try adjusting the punishment type and severity to see if we can get more accurate.

```
In [47]: X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 42)
         logreg = LogisticRegression(C=.1)
         logreg.fit(X_train, y_train)
         y_pred = logreg.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.681481481481
```

Let's move on to further models - we will do naive bayes last because that involves manipulating the data frame. We can do KNN next.

```
In [48]: knn = KNeighborsClassifier(400)
         X_train, X_test, y_train, y_test = train_test_split(X,y, random_state = 42)
         knn.fit(X_train, y_train)
         y_pred = knn.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.540740740741
```

The greatest accuracy score we can get with KNN doesn't really beat the null model. Let's move on to random forests.

```
In [49]: from sklearn.ensemble import RandomForestClassifier
         treeclass = RandomForestClassifier(max_depth = 60, random_state = 42)
         X_train, X_test, y_train, y_test = train_test_split(X,y,random_state = 42)
         treeclass.fit(X_train, y_train)
         y_pred = treeclass.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.651851851852
```

Let's move on to naive bayes. First, we'll need to make sure the copy number has no negatives. We'll add 100 to every row like we previously have done.

```
In [50]: X = X + 100
```

```
In [51]: nb = MultinomialNB()
         X_train, X_test, y_train, y_test = train_test_split(X,y,random_state =42)
         nb.fit(X_train, y_train)
         y_pred = nb.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.540740740741
```

```
In [52]: X = X - 100
         X = abs(X)
```

Let's see if just taking the absolute value of X makes the accuracy any better

```
In [53]: nb = MultinomialNB()
         X_train, X_test, y_train, y_test = train_test_split(X,y,random_state =42)
         nb.fit(X_train, y_train)
         y_pred = nb.predict(X_test)
         print(accuracy_score(y_test, y_pred))
```

```
0.637037037037
```

So far, the best analysis has been logistic regression on all of the data, producing an accuracy score of 68%. Let's see if we can improve this with dimensionality reduction.