Copy_Number_Project_2.1

November 15, 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

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

```
In [189]: import pandas as pd
          import os
          import re
          import xlsxwriter
          import numpy as np

          #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)
```

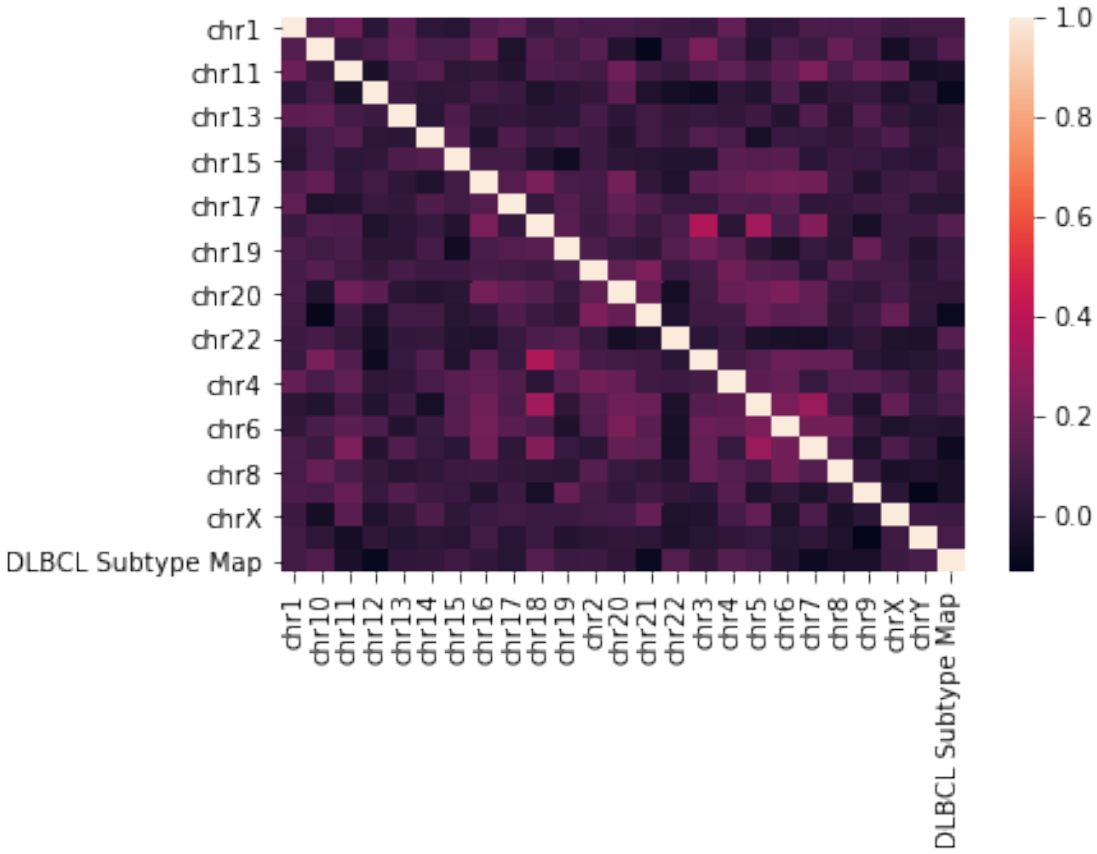
As we can see, this data frame now contains all of our predictors (chromosomal insertion/deletion sums) and our response (DLBCL Subtype).

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 [190]: import matplotlib.pyplot as plt
          import seaborn as sns
          %matplotlib inline

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

```
Out[190]: <matplotlib.axes._subplots.AxesSubplot at 0x114c84710>
```



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 [191]: 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 [192]: 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']

In [193]: 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%.

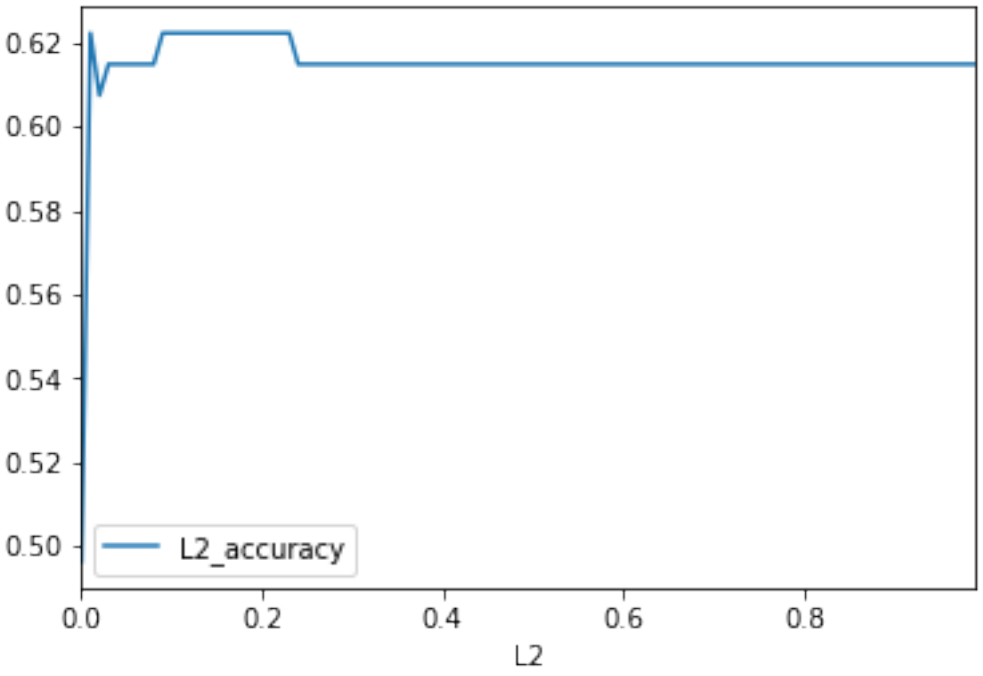
Let's play around with adjusting some of the parameters, initially by changing the values for the L2 penalty (The 'C' paramteter).

```
In [195]: L2_range = np.arange(.0001, 1, 0.01)

          L2_accuracy_list = []

          def L2_logreg_accuracy(L2_range):
              for i in L2_range:
                  logreg = LogisticRegression(C = i, random_state = 42)
                  logreg.fit(X_train, y_train)
                  y_pred = logreg.predict(X_test)
                  L2_accuracy = accuracy_score(y_test, y_pred)
                  L2_accuracy_list.append(L2_accuracy)

          L2_logreg_accuracy(L2_range)

          L2_dict = {'L2' : L2_range, 'L2_accuracy' : L2_accuracy_list}
```

```
L2_df = pd.DataFrame(L2_dict).set_index('L2')

L2_df.plot(y = 'L2_accuracy')
L2_df.sort_values('L2_accuracy', ascending = False).head()
```

Out[195]:

L2	L2_accuracy
0.1801	0.622222
0.1301	0.622222
0.2301	0.622222
0.2201	0.622222
0.2101	0.622222



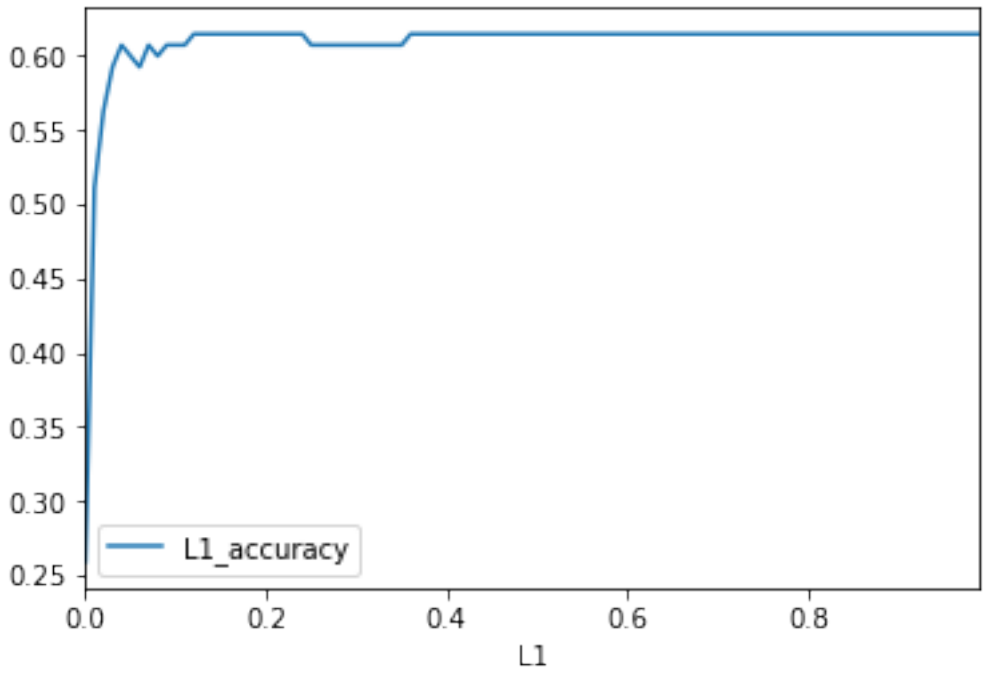
Let's now try using an 'L1' punishment and trying a range of values for 'C'.

```
In [196]: L1_range = np.arange(.0001, 1, 0.01)

          L1_accuracy_list = []

          def L1_logreg_accuracy(L1_range):
              for i in L1_range:
                  logreg = LogisticRegression(penalty = 'l1', C = i, random_state = 42)
                  logreg.fit(X_train, y_train)
                  y_pred = logreg.predict(X_test)
```

```
            L1_accuracy = accuracy_score(y_test, y_pred)
            L1_accuracy_list.append(L1_accuracy)

    L1_logreg_accuracy(L1_range)

    L1_dict = {'L1' : L1_range, 'L1_accuracy' : L1_accuracy_list}

    L1_df = pd.DataFrame(L1_dict).set_index('L1')

    L1_df.plot(y = 'L1_accuracy')

    L1_df.sort_values('L1_accuracy', ascending = False).head()
```

Out[196]:

L1	L1_accuracy
0.5001	0.614815
0.7501	0.614815
0.7301	0.614815
0.7201	0.614815
0.7101	0.614815



We see that we don't really change the accuracy of our logistic regression model for different values of 'C' for both 'L1' and 'L2' regularizations.

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

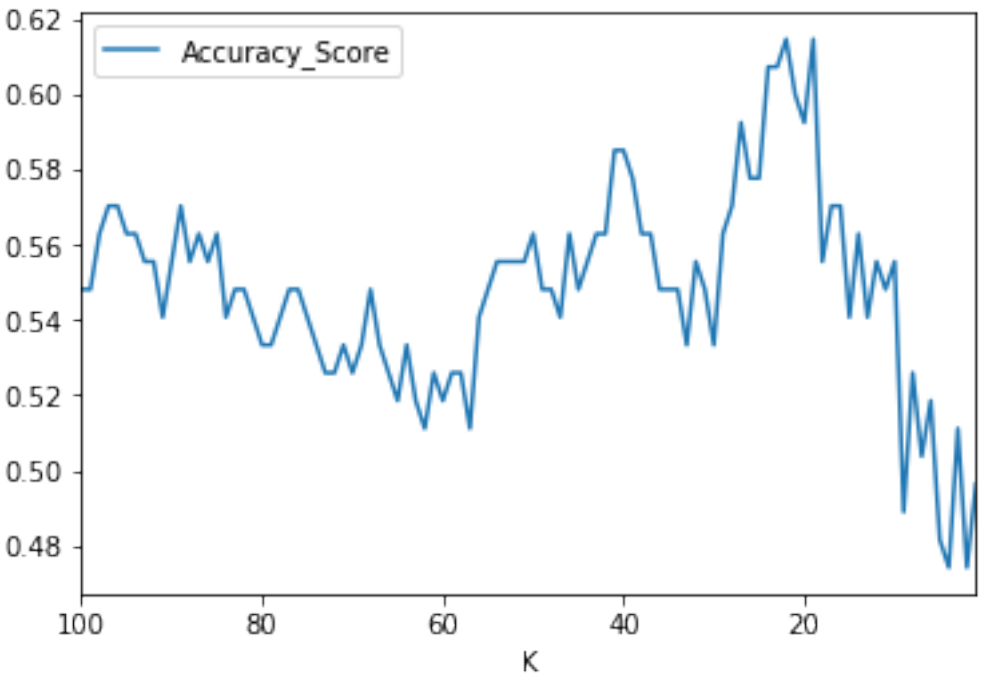
```
In [197]: from sklearn.neighbors import KNeighborsClassifier
          knn = KNeighborsClassifier()
          X_train, X_test, y_train, y_test = train_test_split(X,y, random_state = 42)
          knn.fit(X_train, y_train)
          y_pred = knn.predict(X_test)
          print(accuracy_score(y_test, 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 [198]: k_range = range(1, 101)
          knn_accuracy_list = []

          def knn_accuracy(k_range):
              for i in k_range:
                  knn = KNeighborsClassifier(i)
                  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)
                  knn_accuracy = accuracy_score(y_test, y_pred)
                  knn_accuracy_list.append(knn_accuracy)

          knn_accuracy(k_range)

          knn_dict = {'K': k_range, 'Accuracy_Score': knn_accuracy_list}
          df_knn = pd.DataFrame(knn_dict).set_index('K').sort_index(ascending=False)


          df_knn.plot(y='Accuracy_Score')

          df_knn.sort_values('Accuracy_Score', ascending = False).head()
```

```
Out[198]:     Accuracy_Score
          K
          19        0.614815
          22        0.614815
          24        0.607407
          23        0.607407
          21        0.600000
```



Changing the number of nearest neighbors (K) to 19 or 22 gives us an accuracy rate of 61%.

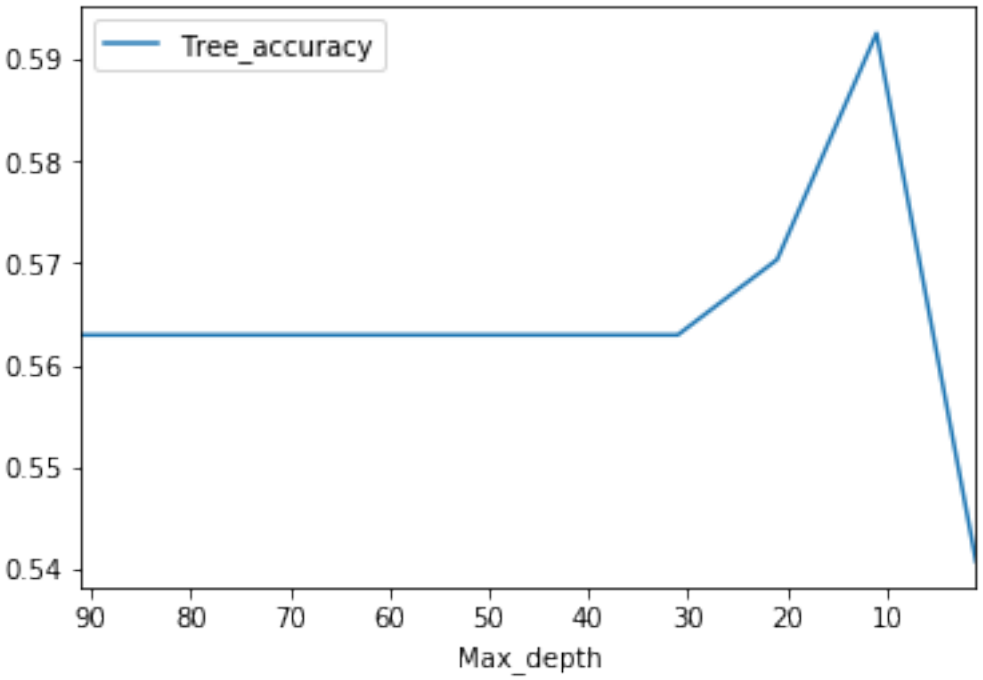
Let's try a new model: random forests.

```
In [199]: from sklearn.ensemble import RandomForestClassifier

          depth_range = range(1,100,10)

          tree_accuracy_list = []

          def tree_score(depth_range):
              for i in depth_range:
                  treeclass = RandomForestClassifier(max_depth = i, 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)
                  tree_accuracy = accuracy_score(y_test, y_pred)
                  tree_accuracy_list.append(tree_accuracy)

          tree_score(depth_range)
          tree_dict = {'Max_depth': depth_range, 'Tree_accuracy' : tree_accuracy_list}
          df_tree = pd.DataFrame(tree_dict).set_index('Max_depth').sort_index(ascending=False)

          df_tree.plot(y='Tree_accuracy')
```

```
df_tree.sort_values('Tree_accuracy', ascending = False).head()
```

Out[199]:

Max_depth	Tree_accuracy
11	0.592593
21	0.570370
91	0.562963
81	0.562963
71	0.562963



A max depth of 11 gives the highest accuracy score of 59%

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 [204]:
```
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 [205]: 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 [206]: 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 [207]: df2.shape
```

```
Out[207]: (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 [208]: X = df2.iloc[:,0:54619]
          y = df2.iloc[:, 54619]
          print(X.shape)
          print(y.shape)
```

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

```
In [209]: 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 at a ~66% accuracy rate.

Let's try adjusting the punishment type and severity to see if we can get more accurate. We'll start with 'L2'.

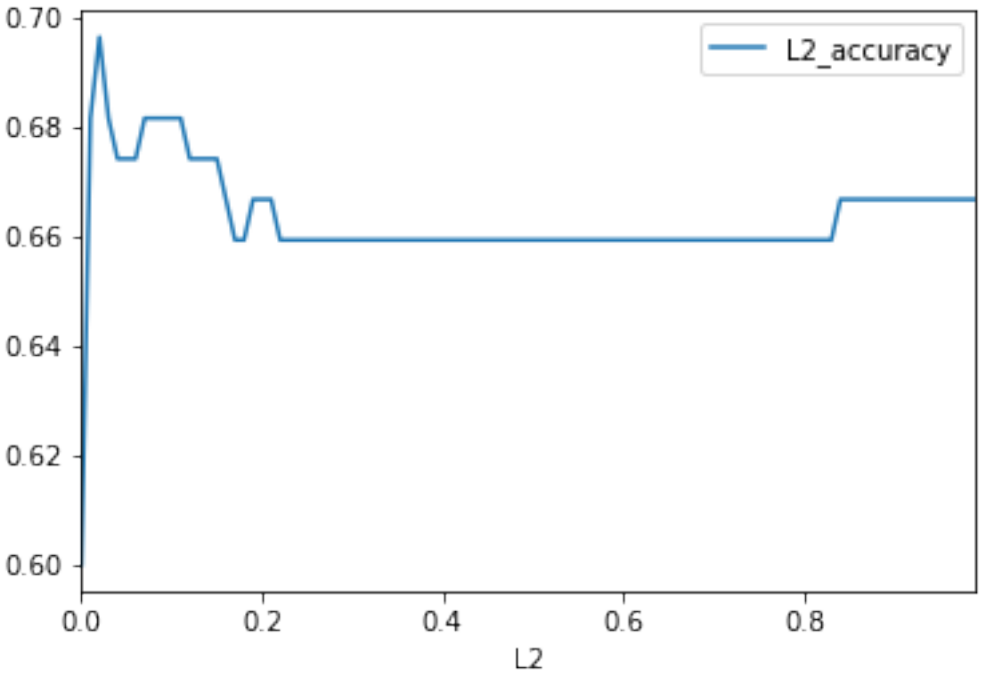
Warning: The following code block will take a while to run.

```
In [214]: L2_range = np.arange(.0001, 1, 0.01)

          L2_accuracy_list = []

          def L2_logreg_accuracy(L2_range):
              for i in L2_range:
                  logreg = LogisticRegression(C = i, random_state = 42)
                  logreg.fit(X_train, y_train)
                  y_pred = logreg.predict(X_test)
                  L2_accuracy = accuracy_score(y_test, y_pred)
                  L2_accuracy_list.append(L2_accuracy)

          L2_logreg_accuracy(L2_range)

          L2_dict = {'L2' : L2_range, 'L2_accuracy' : L2_accuracy_list}
```

```
L2_df = pd.DataFrame(L2_dict).set_index('L2')

L2_df.plot(y = 'L2_accuracy')
L2_df.sort_values('L2_accuracy', ascending = False).head()
```

Out[214]:

L2	L2_accuracy
0.0201	0.696296
0.0901	0.681481
0.1101	0.681481
0.0301	0.681481
0.0101	0.681481



Now, let's look at adjusting 'C' for 'L1'.

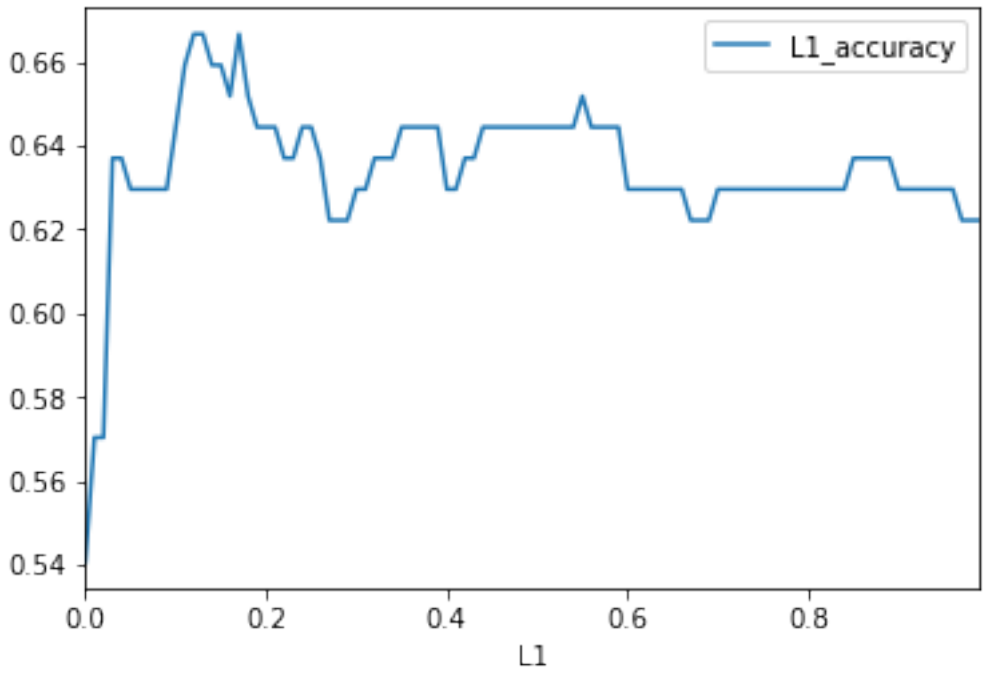
Warning: The following code block will take a while to run.

```
In [215]: L1_range = np.arange(.0001, 1, 0.01)

          L1_accuracy_list = []

          def L1_logreg_accuracy(L1_range):
              for i in L1_range:
                  logreg = LogisticRegression(penalty = 'l1', C = i, random_state = 42)
                  logreg.fit(X_train, y_train)
```

```
            y_pred = logreg.predict(X_test)
            L1_accuracy = accuracy_score(y_test, y_pred)
            L1_accuracy_list.append(L1_accuracy)

        L1_logreg_accuracy(L1_range)

        L1_dict = {'L1' : L1_range, 'L1_accuracy' : L1_accuracy_list}

        L1_df = pd.DataFrame(L1_dict).set_index('L1')

        L1_df.plot(y = 'L1_accuracy')
        L1_df.sort_values('L1_accuracy', ascending = False).head()
```

Out[215]:

L1	L1_accuracy
0.1701	0.666667
0.1301	0.666667
0.1201	0.666667
0.1401	0.659259
0.1501	0.659259



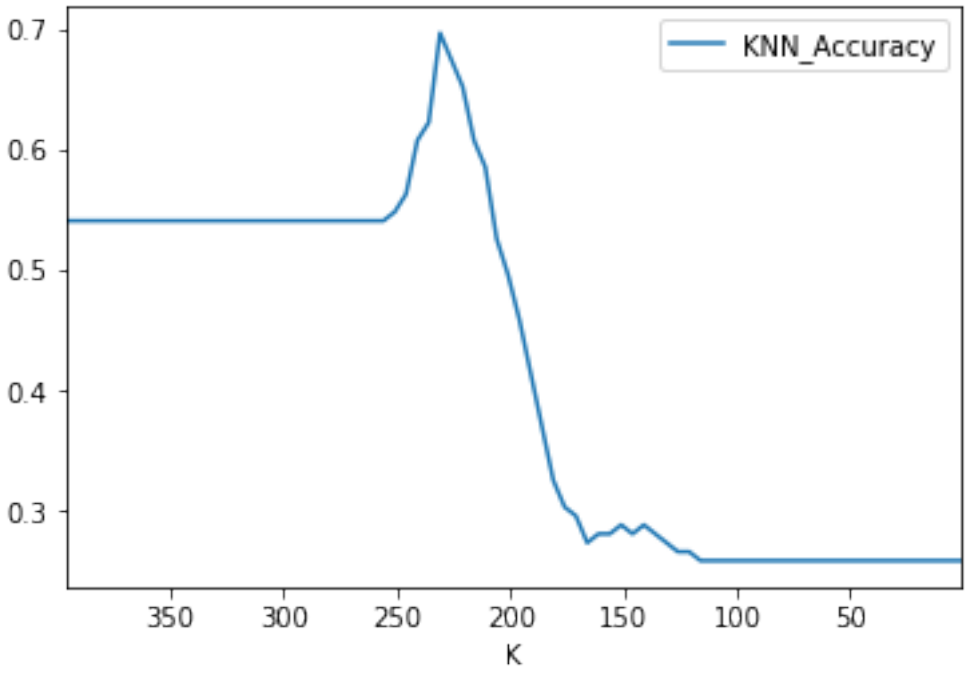
Let's move on to further models - we will do naive bayes last because that involves manipulating the data frame. We can do KNN next.

Warning: The next block of code takes a while to run.

In [216]:
```
k_range = range(1, 400, 5)
knn_accuracy_list = []

def knn_accuracy(k_range):
    for i in k_range:
        knn = KNeighborsClassifier(n_neighbors = i)
        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)
        accuracy = accuracy_score(y_test, y_pred)
        knn_accuracy_list.append(accuracy)

knn_accuracy(k_range)

knn_dict = {'K' : k_range, 'KNN_Accuracy' : knn_accuracy_list}
knn_df = pd.DataFrame(knn_dict).set_index('K').sort_index(ascending = False)

knn_df.plot(y='KNN_Accuracy')
knn_df.sort_values('KNN_Accuracy', ascending = False).head()
```

Out[216]:

K	KNN_Accuracy
231	0.696296
226	0.674074
221	0.651852
236	0.622222
216	0.607407



The greatest accuracy score we can get with KNN is ~69% when K = 231.

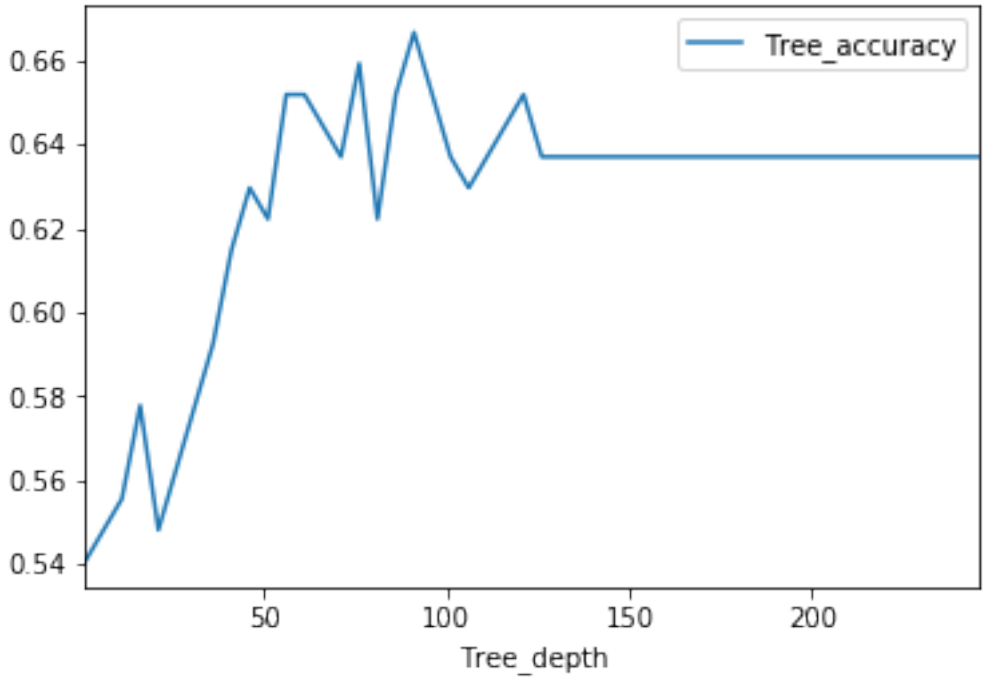
Let's move on to random forests.

Warning: The next block of code takes a while to run.

In [217]:
```
from sklearn.ensemble import RandomForestClassifier

depth_range = range(1, 250, 5)
tree_accuracy_list = []

def tree_accuracy(depth_range):
        for i in depth_range:
            treeclass = RandomForestClassifier(max_depth = i, \
                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)
            tree_accuracy = accuracy_score(y_test, y_pred)
            tree_accuracy_list.append(tree_accuracy)

tree_accuracy(depth_range)

tree_dict = {'Tree_depth' : depth_range, 'Tree_accuracy' : tree_accuracy_list}
tree_df = pd.DataFrame(tree_dict).set_index('Tree_depth')

tree_df.plot(y='Tree_accuracy')
tree_df.sort_values('Tree_accuracy', ascending = False).head()
```

Out[217]:

Tree_depth	Tree_accuracy
91	0.666667
76	0.659259
86	0.651852
121	0.651852
96	0.651852



The highest accuracy rate was ~67% when K = 91.

1.1 Conclusion:

The best model appears to be logistic regression with an L2 regularization on the data frame that uses every mutation as a feature. This produces a wide range of accuracy rates around 66%, which is an improvement over the null accuracy of 52%. We do not want to choose a model that gives a slightly better percentage at only one certain state, because this is likely a result of luck.