Lab2Code

September 17, 2019

```
In [1]: import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        import numpy as np
        from sklearn import datasets, linear_model
```

1 1. Correlations.

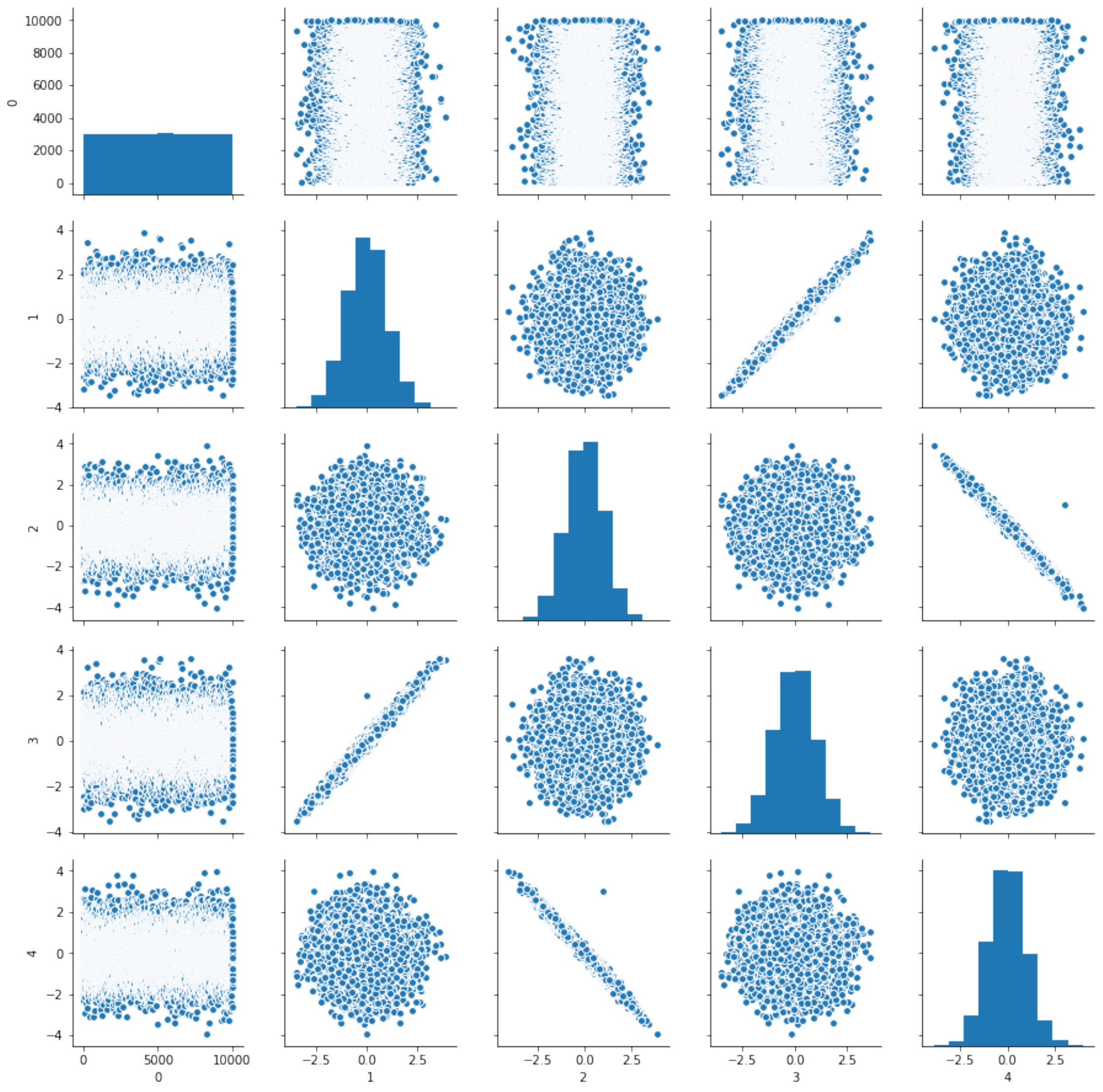
a. When given a data matrix, an easy way to tell if any two columns are correlated is to look at a scatter plot of each column against each other column. For a warm up, do this: Look at the data in DF1 in Lab2 Data.zip. Which columns are (pairwise) correlated? Figure out how to do this with Pandas, and also how to do this with Seaborn.

```
In [2]: df1 = pd.read_csv("Lab2_Data/DF1", header=None)
```

```
In [3]: df1.corr()
```

```
Out[3]:           0         1         2         3         4
        0  1.000000 -0.003991  0.008789 -0.004044 -0.007086
        1 -0.003991  1.000000 -0.003998  0.989869  0.004107
        2  0.008789 -0.003998  1.000000 -0.003887 -0.989445
        3 -0.004044  0.989869 -0.003887  1.000000  0.004662
        4 -0.007086  0.004107 -0.989445  0.004662  1.000000
```

```
In [4]: df1.fillna(df1.mean(), inplace=True)
        graph = sns.pairplot(df1)
```



The Seaborn graphs corroborate that graphs 1 and 3 are correlated and graphs 2 and 4 are correlated.

b. Compute the covariance matrix of the data. Write the explicit expression for what this is, and then use any command you like (e.g., np.cov) to compute the 4×4 matrix. Explain why the numbers that you get fit with the plots you got.

The covariance between any two columns x and y is (x1 - x_mean)(y1 - y_mean)/n + (x2 - x_mean)(y2 - y_mean)/n + ... + (xn - x_mean)(yn - y_mean)/n.

The numbers fit with the plots because the covariance of columns that are highly correlated are closer to 1.

```
In [5]: df1.cov()
```

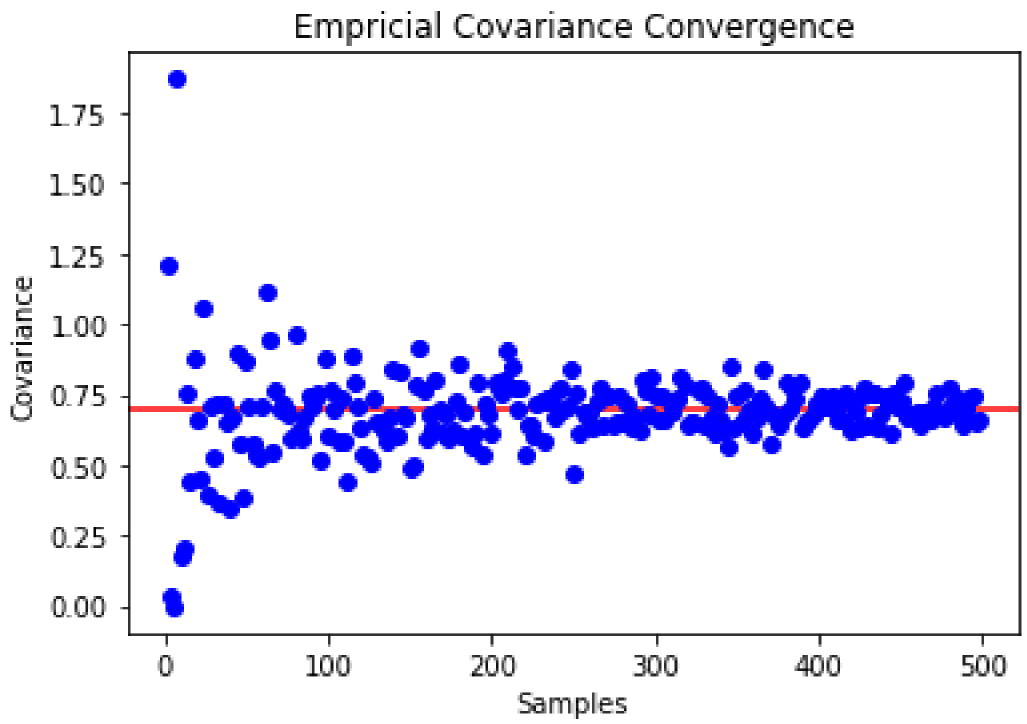
```
Out[5]:                  0          1          2          3          4
        0   8.333333e+06 -11.529682  25.437628 -11.681482 -20.508118
```

```
1 -1.152968e+01   1.001458  -0.004012   0.991523   0.004122
2  2.543763e+01  -0.004012   1.005376  -0.003901  -0.995059
3 -1.168148e+01   0.991523  -0.003901   1.001885   0.004680
4 -2.050812e+01   0.004122  -0.995059   0.004680   1.005973
```

c. The above problem in reverse. Generate a zero-mean multivariate Gaussian random variable in 3 dimensions, Z = (X1, X2, X3) so that (X1, X2) and (X1, X3) are uncorrelated, but (X2, X3) are correlated. Specifically: choose a covariance matrix that has the above correlations structure, and write this down. Then find a way to generate samples from this Gaussian. Choose one of the non-zero covariance terms (Cij, if C denotes your covariance matrix) and plot it vs the estimated covariance term, as the number of samples you use scales. The goal is to get a visual representation of how the empirical covariance converges to the true (or family) covariance.

```
In [6]: #function for calulating covariance of samples
        def estimateCovariance(mean, cov, samples, i, j):
            X = np.random.multivariate_normal(mean, cov, samples).T
            return np.cov(X)[i][j]



        mean = [0, 0, 0]
        cov = [[1, 0, 0],
               [0, 1, .7],
               [0, .7, 1]] #covariance matrix, X2 and X3 correlated

        plt.title("Empricial Covariance Convergence")
        plt.xlabel("Samples")
        plt.ylabel("Covariance")
        plt.axhline(.7, color='r')

        for n in range(0, 500, 2):
            plt.plot(n, estimateCovariance(mean, cov, n, 1, 2), marker='o', color='b')
```

```
C:\Users\jlu90\AppData\Roaming\Python\Python36\site-packages\numpy\lib\function_base.py:390: Run
  avg = a.mean(axis)
C:\Users\jlu90\AppData\Roaming\Python\Python36\site-packages\numpy\core\_methods.py:154: Runtime
  ret, rcount, out=ret, casting='unsafe', subok=False)
C:\Users\jlu90\Anaconda3\lib\site-packages\ipykernel_launcher.py:4: RuntimeWarning: Degrees of f
  after removing the cwd from sys.path.
C:\Users\jlu90\AppData\Roaming\Python\Python36\site-packages\numpy\lib\function_base.py:2455: Ru
  c *= np.true_divide(1, fact)
C:\Users\jlu90\AppData\Roaming\Python\Python36\site-packages\numpy\lib\function_base.py:2455: Ru
  c *= np.true_divide(1, fact)
```



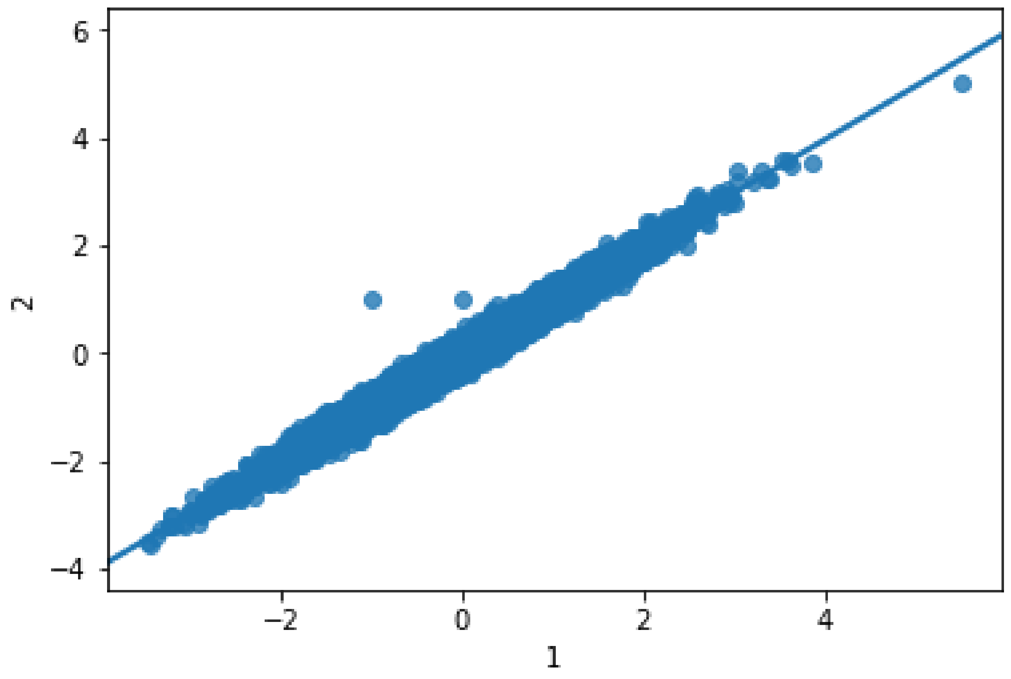
2 2. Outliers.

```
In [7]: df2 = pd.read_csv("Lab2_Data/DF2", header=None)
```

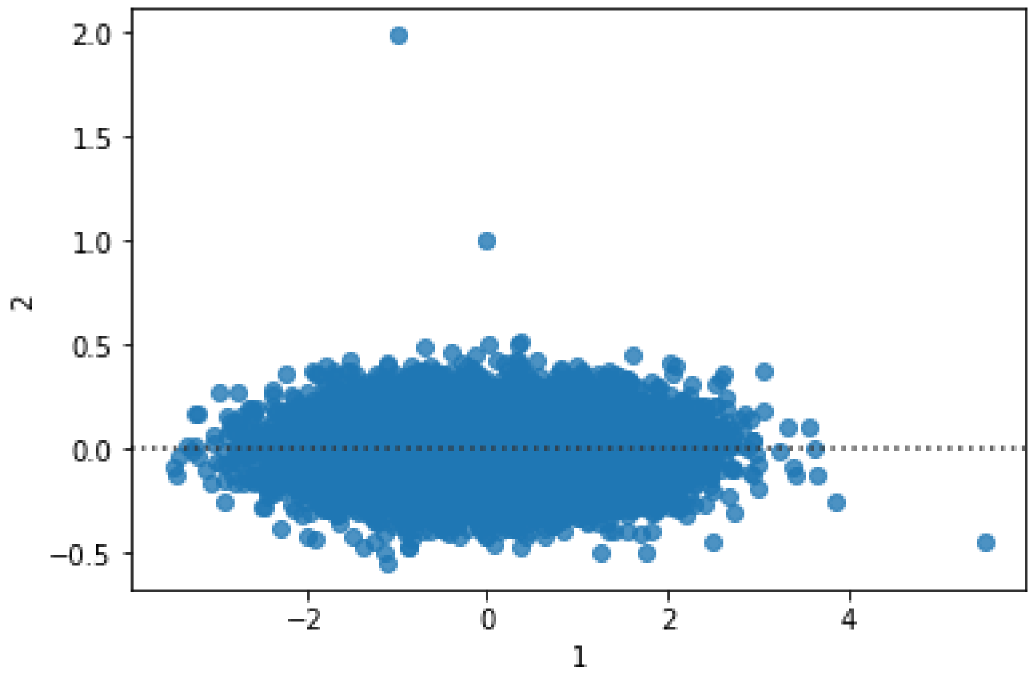
```
In [8]: sns.regplot(df2[1], df2[2])
```

```
C:\Users\jlu90\Anaconda3\lib\site-packages\scipy\stats\stats.py:1706: FutureWarning: Using a non
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

```
Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x1d403d21320>
```



```
In [9]: sns.residplot(df2[1], df2[2])

Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x1d40564c6d8>
```



By plotting the residual graph, we can see that the point (-1, 1) is much more of an outlier at residual value 2.0 than the point (5.5, 5) which has a residual value of -

3 3. Standard Error

a. Compute the empirical standard deviation of the error for n = 150 (the number we used in class). In class, in the exercise where I tried to find a linear regression of y vs. noise, we found $\hat{\beta} = -0.15$. Given your empirical computation of the standard deviation of the error, how significant is the value -0.15?

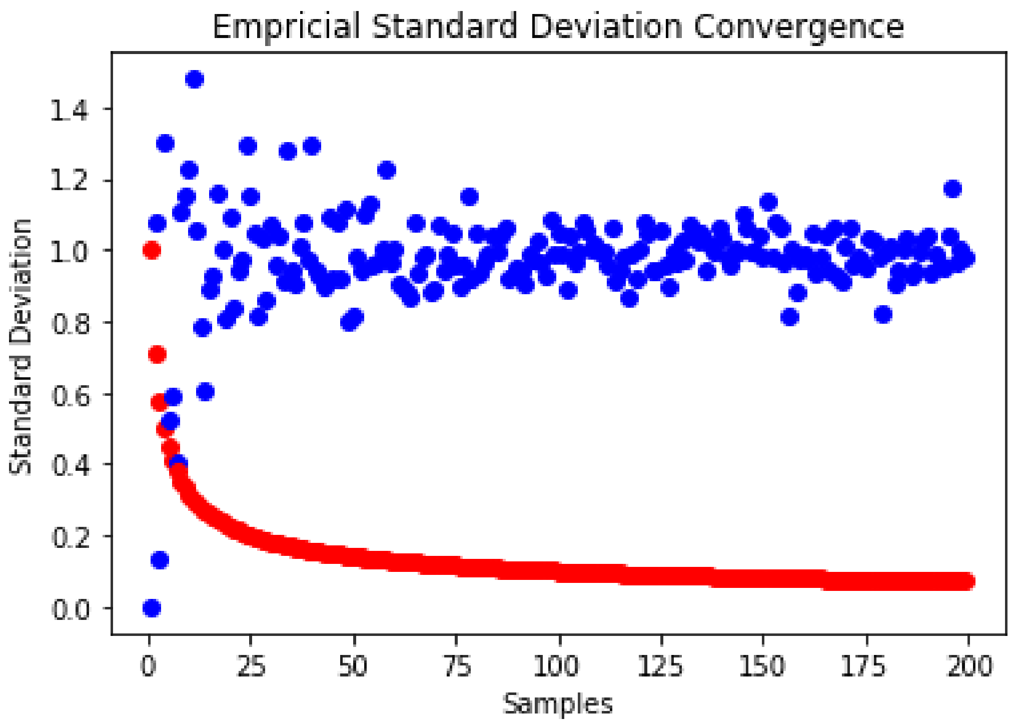
```
In [10]: mean = -1
         std = 1
         n = 150
         samples = np.random.normal(mean, std, n)
         print("std: " +  str(samples.std()))
```

```
std: 0.9747631284817826
```

The actual value of β is 0, and the predicted value of β is -0.15. The predicted value of -0.15 is not very significant because its less than one standard deviation away from the actual value.

b. Now repeat the above experiment for different values of n. Plot these values, and on the same plot, plot 1/sqrt(n). How is the fit?

```
In [11]: plt.title("Empricial Standard Deviation Convergence")
         plt.xlabel("Samples")
         plt.ylabel("Standard Deviation")

         for n in range(1,200):
             samples = np.random.normal(mean, std, n)
             plt.plot(n, samples.std(), marker='o', color="blue")
             plt.plot(n, 1/np.sqrt(n), marker='o', color="red")
```



The emperical standard deviation converges to 1 as n increases. It seems to have the same relationship as 1/sqrt(n) but reflected across y=0.5

4 4. Names and Frequencies

a. Write a program that on input k and XXXX, returns the top k names from year XXXX.

```
In [12]: def topnames(k, XXXX):
             names = pd.read_csv("Names/Names/yob" + str(XXXX) + ".txt", header=None)
             print(names[0:int(k)])

In [13]: topnames(5, 1995)

          0  1      2
0   Jessica  F  27935
1    Ashley  F  26603
2     Emily  F  24378
3  Samantha  F  21646
4     Sarah  F  21369
```

b. Write a program that on input Name returns the frequency for men and women of the name Name.

```
In [14]: def namefrequency(name):
             frequency = 0
             for i in range(1880, 2016):
                 names = pd.read_csv("Names/Names/yob" + str(i) + ".txt", header=None)
                 for j in range(0, len(names)):
                     if names[0][j] == str(name):
                         frequency += names[2][j]

             print(frequency)

In [15]: namefrequency("John")

5117331
```

c. It could be that names are more diverse now than they were in 1880, so that a name may be relatively the most popular, though its frequency may have been decreasing over the years. Modify the above to return the relative frequency. Note that in the next coming lectures we will learn how to quantify diversity using entropy

```
In [16]: def relativefrequency(name, startYear, endYear):
             totalOccurrences = 0  # find total no. occurences in time period
             totalNames = 0 # find total names recorded in time period
             for year in range(startYear, endYear):
                 names = pd.read_csv("Names/Names/yob" + str(year) + ".txt", header=None)
                 for j in range(0, len(names)):
                     totalNames += names[2][j]
                     if names[0][j] == str(name):
                         totalOccurrences += names[2][j]

             frequency = totalOccurrences/totalNames
             print(name + "'s frequency between " + str(startYear) + "-" + str(endYear) + ": " +

In [17]: relativefrequency("James", 1890, 1900)
         relativefrequency("James", 2010, 2016)

James's frequency between 1890-1900: 0.01516893078457864
James's frequency between 2010-2016: 0.003791415180079932
```

d. Find all the names that used to be more popular for one gender, but then became more popular for another gender.

```
In [18]: #get frequencies of men and womens names
         nameFreq_men = {}
         nameFreq_women = {}

         for year in range(1880, 2016):
             names = pd.read_csv("Names/Names/yob" + str(year) + ".txt", header=None)
```

```
            for j in range(0, len(names)):
                name = names[0][j]
                gender = names[1][j]
                freq = names[2][j]

                if gender=="M" and name not in nameFreq_men:
                    listFreq = [0]*136  # initialize new list of frequencies for name
                    listFreq[year-1880] = freq
                    nameFreq_men[name] = listFreq

                elif gender=="M" and name in nameFreq_men:
                    listFreq = nameFreq_men[name]  # update list of frequencies
                    listFreq[year-1880] = freq
                    nameFreq_men[name] = listFreq

                elif gender=="F" and name not in nameFreq_women:
                    listFreq = [0]*136  # initialize new list of frequencies for name
                    listFreq[year-1880] = freq
                    nameFreq_women[name] = listFreq

                elif gender=="F" and name in nameFreq_women:
                    listFreq = nameFreq_women[name]  # update list of frequencies
                    listFreq[year-1880] = freq
                    nameFreq_women[name] = listFreq
```

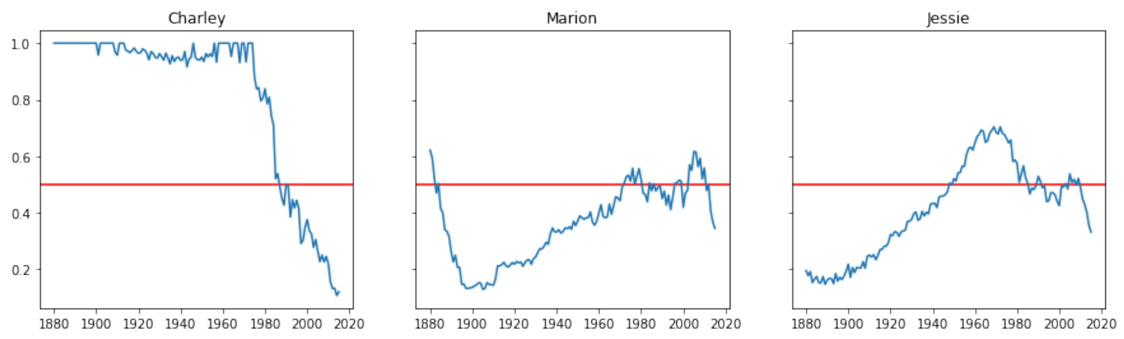
```
In [19]: m_w_ratio = {}  #find ratios of men to women's names
         for name in nameFreq_men:
             if name in nameFreq_women:
                 for year in range(1880, 2016):
                     menFreq = nameFreq_men[name][year - 1880]
                     womenFreq = nameFreq_women[name][year - 1880]

                     if name not in m_w_ratio:
                         ratio = [0]*136
                         if menFreq == 0 and womenFreq == 0:
                             ratio[year - 1880] = float(np.nan)
                         else:
                             ratio[year - 1880] = menFreq / (menFreq + womenFreq)
                         m_w_ratio[name] = ratio

                     else:
                         ratio = m_w_ratio[name]
                         if menFreq == 0 and womenFreq == 0:
                             ratio[year - 1880] = float(np.nan)
                         else:
                             ratio[year - 1880] = menFreq / (menFreq + womenFreq)
                         m_w_ratio[name] = ratio
```

```
In [20]: # drop names where ratio never crosses 0.5
```

```
droppedNames = []

for name in m_w_ratio:
    moreMen = False
    moreWomen = False
    for year in range(len(m_w_ratio[name])):
        if m_w_ratio[name][year] != float(np.nan) and m_w_ratio[name][year] > 0.5:
            moreMen = True
        if m_w_ratio[name][year] != float(np.nan) and m_w_ratio[name][year] < 0.5:
            moreWomen = True

    if moreMen != moreWomen:
        droppedNames.append(name)

for name in droppedNames:
    m_w_ratio.pop(name)
```

In [25]:
```
print(str(len(m_w_ratio)) + " Names became more popular for another gender at some poin
names = list(m_w_ratio.keys())
print(names)

fig, ax = plt.subplots(1,3, sharex=True, sharey=True)
fig.set_figwidth(15)

X = [i+1880 for i in range(0,136)]

#examples
ax[0].title.set_text(names[0])
ax[0].axhline(0.5, color="red")
ax[0].plot(X, m_w_ratio[names[0]])

ax[1].title.set_text(names[1])
ax[1].axhline(0.5, color="red")
ax[1].plot(X, m_w_ratio[names[1]])

ax[2].title.set_text(names[2])
ax[2].axhline(0.5, color="red")
ax[2].plot(X, m_w_ratio[names[2]])
```

```
4891 Names became more popular for another gender at some point
['Charley', 'Marion', 'Jessie', 'Sidney', 'Leslie', 'Alva', 'Harley', 'Ollie', 'Monroe', 'Emery'
```



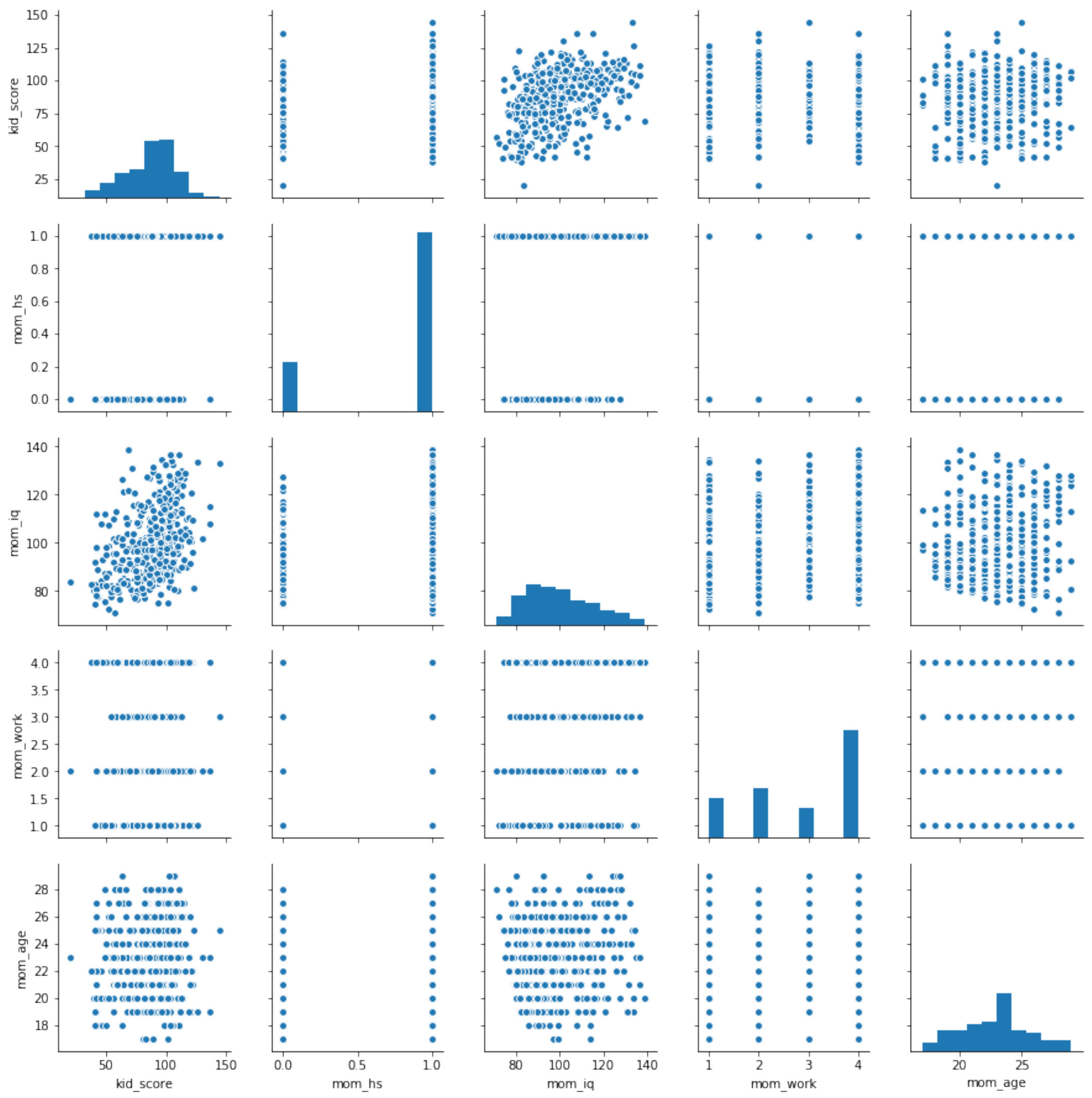
5 5. Regression and Interaction Terms.

Run through the Jupyter Notebook from Thursday's class. You will have to download the data set yourselves. Then do the exercise of adding an interaction term. Explain what you see, and how it relates to the graph you obtain before adding the interaction term. That is, use plots / visualization, to argue convincingly that the interaction term should or shouldn't be there, and then tell us what this means.

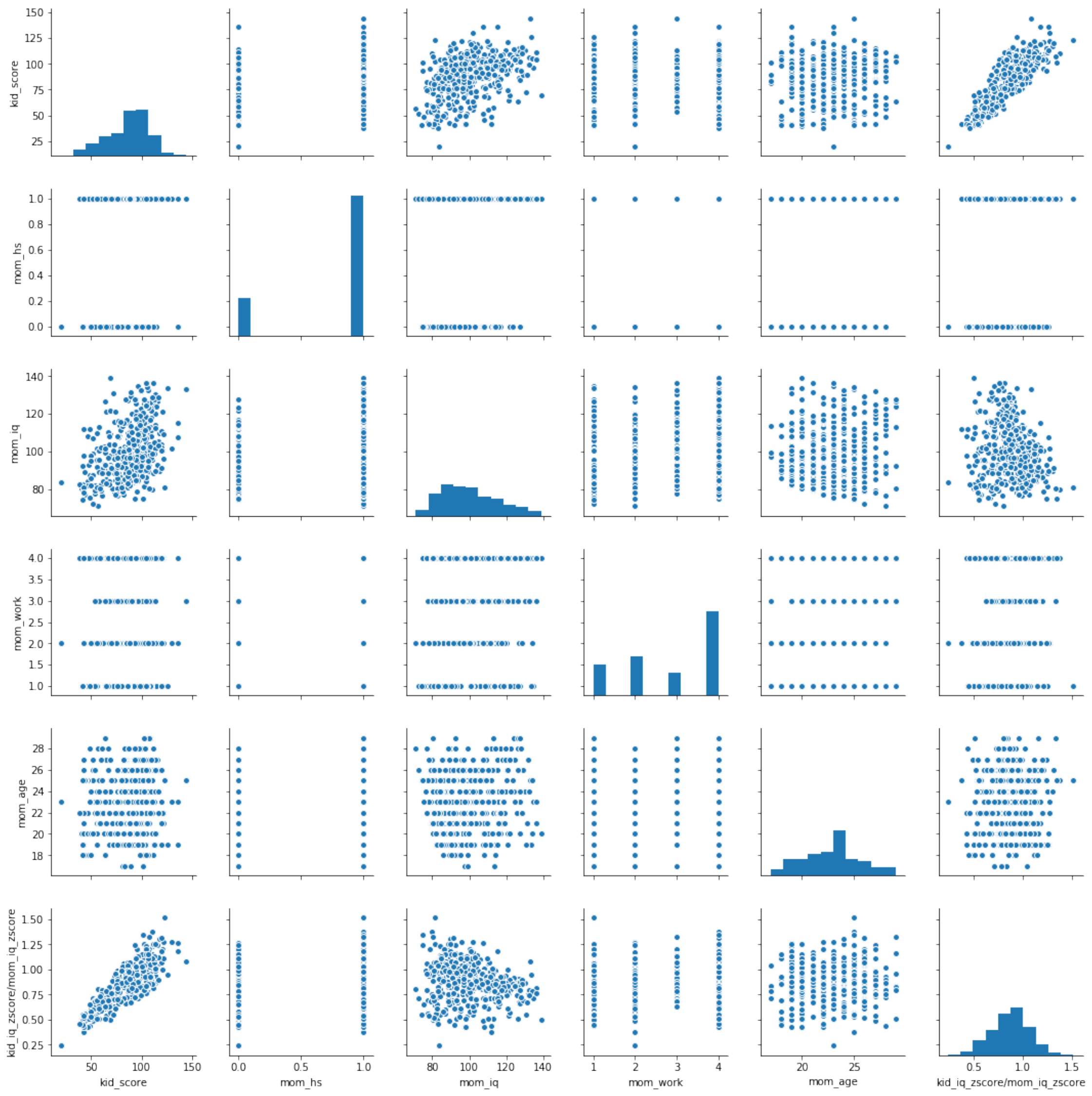
```
In [22]: kidiq = pd.read_stata('ARM_Data/child.iq/kidiq.dta')
         kidiq.head()
```

```
Out[22]:    kid_score  mom_hs      mom_iq  mom_work  mom_age
         0         65     1.0  121.117529         4       27
         1         98     1.0   89.361882         4       25
         2         85     1.0  115.443165         4       27
         3         83     1.0   99.449639         3       25
         4        115     1.0   92.745710         4       27
```

```
In [23]: sns.pairplot(kidiq)
         sns.despine()
```



```
In [24]: kidiq["kid_iq_zscore/mom_iq_zscore"] = abs(kidiq["kid_score"]/kidiq["mom_iq"])
         sns.pairplot(kidiq)
         sns.despine()
```



The interaction term that was added is a ratio of the kid's score and the mom's score. Before it was added, there was no easy way to compare between the categorical data and the score of the child relative to how the mother performed. For example, if we wanted to see if the mom's work was related to how the kid would do, we would want to condition on how the mom did in the first place. The interaction term gives us this flexibility.

Let us look at the mom_work vs kid_score graph (A) and the mom_work vs kid_iq_score/mom_iq_score graph (B). In B, there seems to be less variance in work 3 than there is in work 3 of A. This might indicate that for the moms in that line of work, there is less of a difference in mom-kid iq scores