Dataset Manipulation

In [1]:

```
#load libraries
import numpy as np
from sklearn import preprocessing
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from PIL import Image
```

In [2]:

```
#omit Pokémon Trainer and split into composite pokemon
characters='Cloud,Pokémon Trainer,Pikachu,Peach/Daisy,Zero Suit
Samus,Joker,Palutena,Wario,Lucina,Mr. Game &
Watch,Olimar,R.O.B.,Snake,Roy,Inkling,Mario,Wolf,Fox,Young Link,Mega Man,Ness'
character_col= characters
character_row= 'Pokémon Trainer,Joker,Palutena,Lucina,Mr. Game & Watch,Wolf,Peach/Daisy,Young Link
,Wolf,Roy'
charrow_list=character_row.split(',') #create list of top tiers that I play(rows)
charcol_list=character_col.split(',') #create list of top tiers that I play against(columns)
import pandas as pd
matrix= pd.read_csv('MatchupMatrix.csv') #load data
match_mat=matrix.fillna(value=0) #replace NaNs with 0
match_mat=match_mat.replace(to_replace='-',value=0) #replace '-' with 0 so we can do mathematical
operations
```

In [3]:

```
##extracting only relevant info from initial data
isin=match_mat['Unnamed: 0'].isin(charrow_list) #returns booleans for every entry of the data that
has value from char_list
indices=match_mat.index[isin== True].tolist() #gets indices that satisfies above requirement
mymatch=match_mat.iloc[indices,:] #creates new matrix from data of only top-tiers
mymatch.index=mymatch['Unnamed: 0']
```

In [4]:

```
mymatch
```

Out[4]:

Unnamed: 0	Unnamed: 0	Unnamed: 1	Palutena	Joker	Peach/Daisy	Pikachu	Zero Suit Samus	Wario	Lucina	Shulk	...	Isabelle	King Dedede	Jigg
Palutena	Palutena	0.0	0	-0.4	0.2	-1.1	-0.2	0.5	-0.1	0.7	...	1.6	2.3	
Joker	Joker	0.0	0.4	0	0.0	-1.0	0.8	0.6	0.5	0.0	...	1.1	1.6	
Peach/Daisy	Peach/Daisy	0.0	-0.2	0.0	0	0.3	-0.5	0.5	-0.2	-1.1	...	0.3	0.8	
Lucina	Lucina	0.0	0.1	-0.5	0.2	-0.6	-0.3	0.4	0	-0.4	...	1.4	1.2	
Pokémon Trainer	Pokémon Trainer	0.0	0.4	-0.3	0.0	-0.9	0.4	0.3	0.1	-0.4	...	1.4	1.3	
Wolf	Wolf	0.0	0.0	-0.5	-0.4	-1.2	0.9	0.3	0.0	-0.8	...	1.3	1.5	
Mr. Game & Watch	Mr. Game & Watch	0.0	-1.2	-0.3	0.9	1.1	-1.2	0.6	-0.6	-1.5	...	1.2	2.0	
Roy	Roy	0.0	0.4	-0.6	0.0	-1.0	0.7	-0.1	0.0	0.1	...	1.1	1.1	
Young Link	Young Link	0.0	-1.3	-0.9	0.1	-1.2	0.0	-0.3	-0.5	-1.6	...	0.8	2.0	

9 rows × 82 columns

In [5]:

```
##checking for errors
CharLstDf=pd.DataFrame(charrow_list) #list --> df
test=list(match_mat['Unnamed: 0'])   #df --> list
isin2=CharLstDf.isin(test)     #returns booleans for every entry in char_list that isn't in new ma
trix(checks for misses)
indices2=isin2.index[isin2[0]==False].tolist() #gets indices that satisifes above requirement
CharLstDf.iloc[indices2,:] #creates new rectified matrix
if mymatch.shape[0] == len(charrow_list): #check to see if anything is missing
    print('Nothing missing')
else:
    print("Something missing")
```

```
Something missing
```

In [6]:

```
##since payoff matrix has rows=cols, then this time we do for columns
empty=[] #loop over each column name, and if it isn't in char_list, add it to a list to drop it
for _ in mymatch.columns:
    if _ in charcol_list:
        continue
    else:
        empty.append(_)
mymatch=mymatch.drop(columns=empty)
mymatch=pd.DataFrame(mymatch,dtype=float)
labels=mymatch.index
reg_matchup=mymatch.set_index(labels) #reset index names(cols=rows)
```

In [7]:

```
MyMatchupMatrix=reg_matchup.to_csv(r'C:\Users\Michael\Documents\Python
Scripts\MyMatchupMatrix.csv') #save raw dataframe
```

In [8]:

```
##standardize and normalize the columns(variables)
stds=reg_matchup.std(axis=0) #get the std of each column
avgs=reg_matchup.mean(axis=0) #get the avg of each column
normalized=reg_matchup.sub(avgs,axis=1) #subtract avg from each column
standnorm=normalized.div(stds,axis=1) #divide std from each column
mymatch=pd.DataFrame(standnorm,index=labels)
mymatch=mymatch.fillna(value=0)
print(mymatch.std(axis=0))
mymatch.columns
mymatch.index
```

```
Palutena            1.0
Joker               1.0
Peach/Daisy         1.0
Pikachu             1.0
Zero Suit Samus     1.0
Wario               1.0
Lucina              1.0
Pokémon Trainer     1.0
Inkling             1.0
Wolf                1.0
Fox                 1.0
Snake               1.0
Mario               1.0
Mr. Game & Watch    1.0
R.O.B.              1.0
Roy                 1.0
Olimar              1.0
Mega Man            1.0
Young Link          1.0
Cloud               1.0
Ness                1.0
dtype: float64
```

Out[8]:

```
Index(['Palutena', 'Joker', 'Peach/Daisy', 'Lucina', 'Pokémon Trainer', 'Wolf',
```

```
       'Mr. Game & Watch', 'Roy', 'Young Link'],
      dtype='object', name='Unnamed: 0')
```

In [9]:

```
NormSSBU=mymatch.to_csv(r'C:\Users\Michael\Documents\MATLAB\NormSSBU.csv') #save normalized
version
```

Data Analysis

Sklearn-PCA

In [10]:

```
pca=PCA(n_components=min(mymatch.shape)) #initialize principal components from sklearn
fit_tran=pca.fit_transform(mymatch) #this is the data-matrix x pca loadings = scores for each char
acter
pcaDF=pd.DataFrame(fit_tran,index=labels) #(characters vs. concept matrix)
print('First 3 ratios of explained variance are')
pca.explained_variance_ratio_[0:4] #shows how much variance is explained by the first 3 principal
components
```

```
First 3 ratios of explained variance are
```

Out[10]:

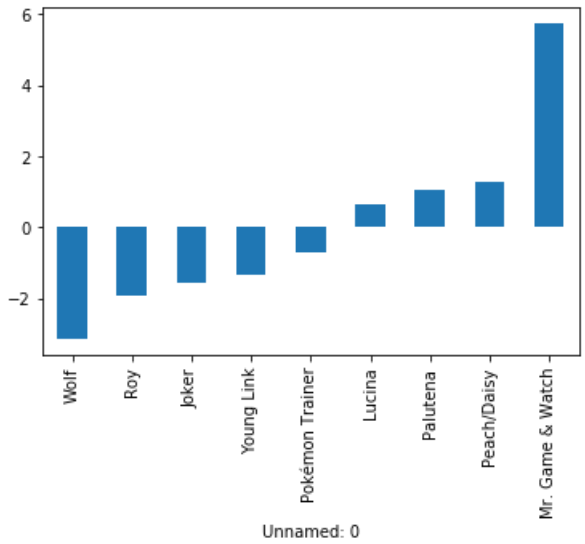
```
array([0.32694742, 0.27890271, 0.1522353 , 0.11277571])
```

Scores

In [11]:

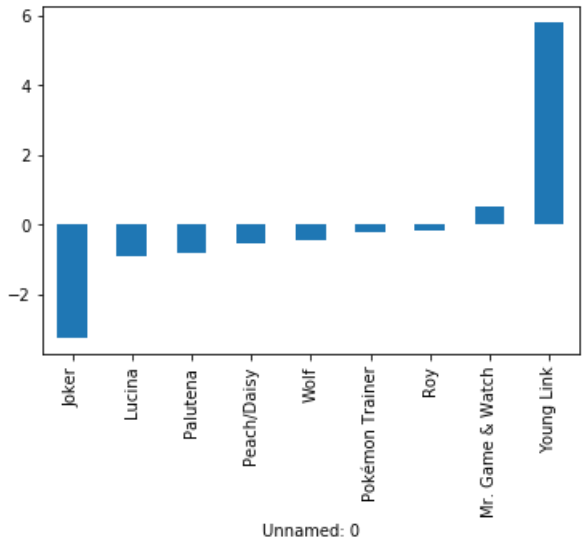
```
##for each principal component, rank from least to greatest that each sample(character) scores
for pc in [0,1,2]: #python indexing
    sorts=pcaDF[pc].sort_values(axis=0) #sort values of the series vertically
    print('Negative & Positive PC:'+ str(pc+1))
    print(sorts.index[0], ',',sorts.index[-1]) #show which variables contribute minimally and maxim
ally
    sorts.plot.bar() #graph sorted values
    plt.show()
```

```
Negative & Positive PC:1
Wolf , Mr. Game & Watch
```

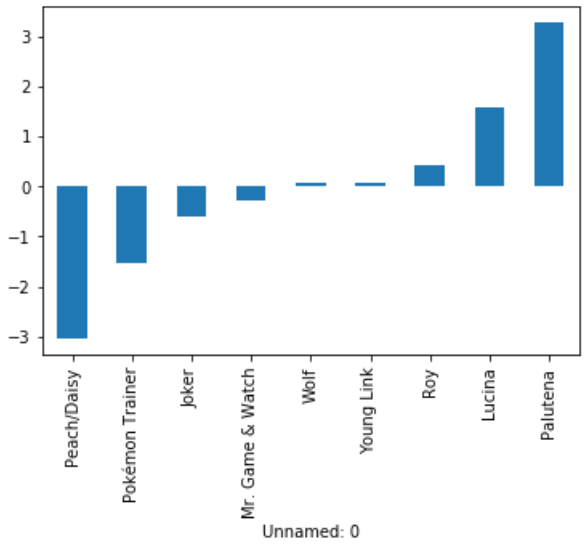



```
Negative & Positive PC:2
```

```
Joker , Young Link
```




```
Negative & Positive PC:3
Peach/Daisy , Palutena
```



Loadings

In [12]:

```
pca.components_.shape #shape is k-components x m variables, so right now it is V-transposed
loadings=pca.components_.T
print(loadings.shape) #shape is now m variable x k-components
```

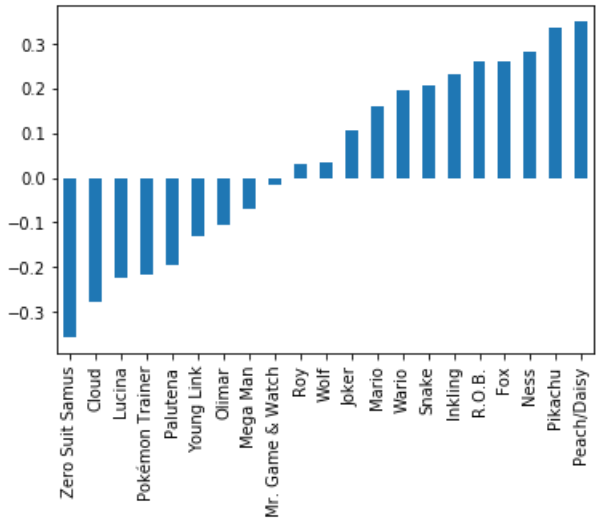
```
(21, 9)
```

In [13]:

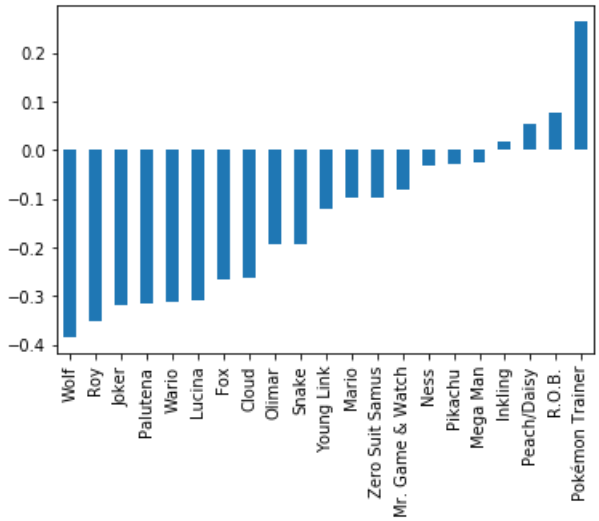
```
variables=mymatch.columns
first3loadings=pd.DataFrame(loadings[:,0:3],index=variables)
for pc in [0,1,2]: #python indexing
    sorts=first3loadings[pc].sort_values(axis=0) #sort values of the series vertically
    print('Negative & Positive PC Loading:'+ str(pc+1))
    print(sorts.index[0], ',',sorts.index[-1]) #show which variables contribute minimally and maxim
ally
    sorts.plot.bar() #graph sorted values
    plt.show()
```

```
Negative & Positive PC Loading:1
```

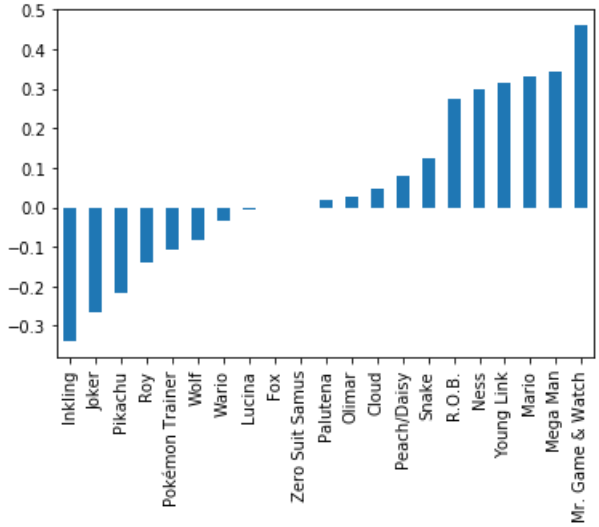
Negative & Positive PC Loading:1
Zero Suit Samus , Peach/Daisy



Negative & Positive PC Loading:2
Wolf , Pokémon Trainer



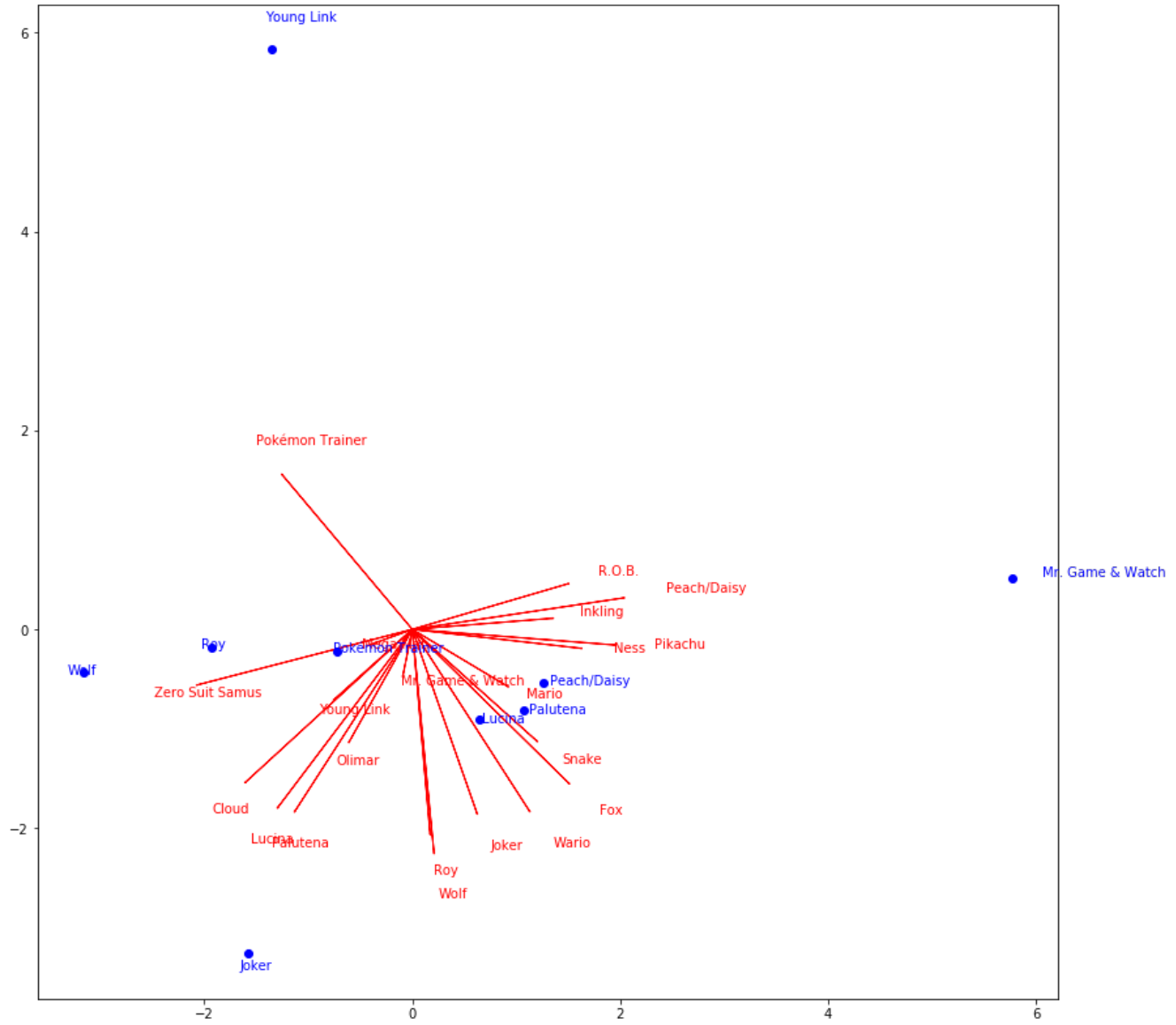
Negative & Positive PC Loading:3
Inkling , Mr. Game & Watch



Graphing

In [14]:

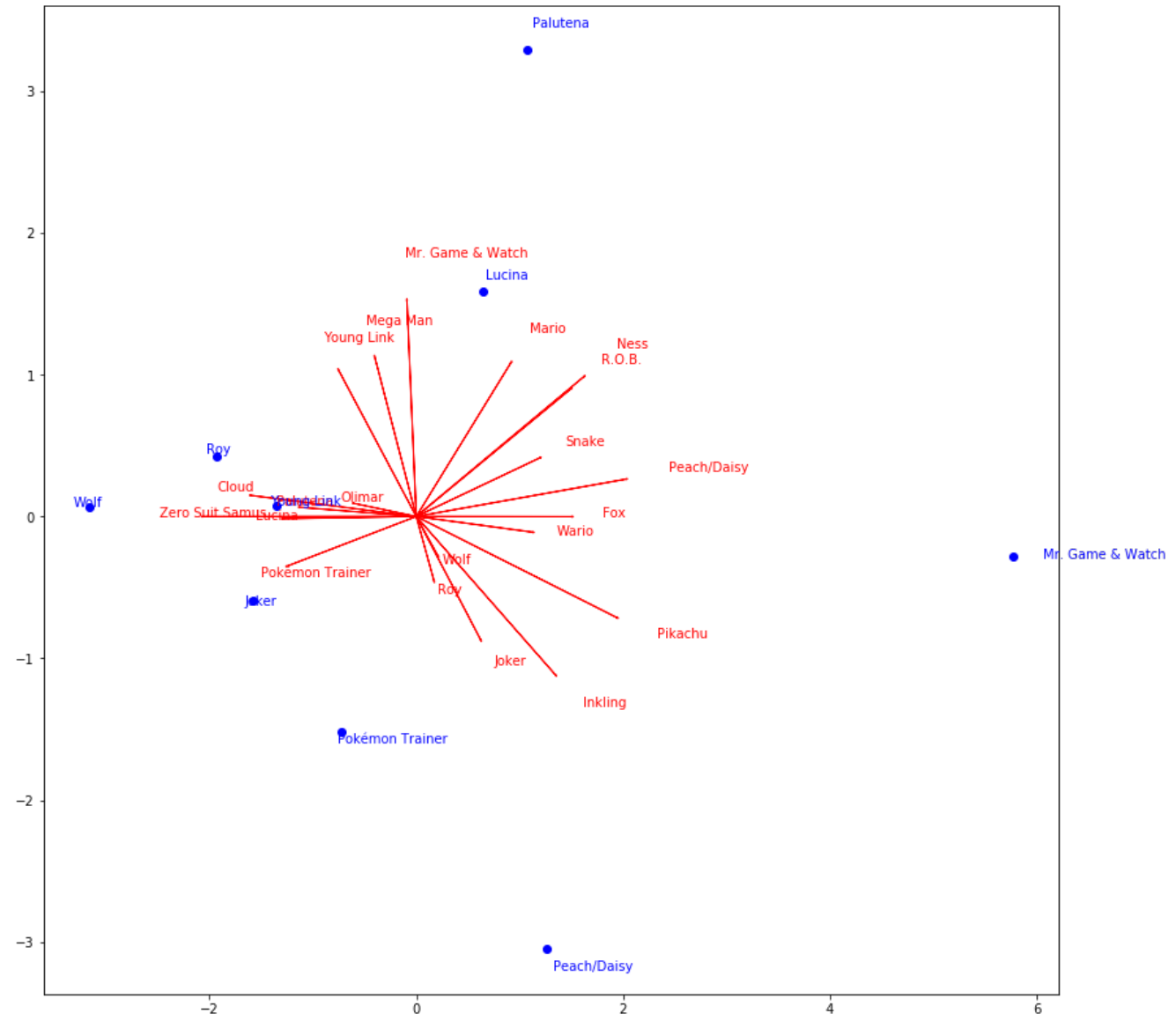
```
##pc1 vs pc2
def biplot12(pc,loadings,labels=None): #got this from stackexchange(only takes arrays and not DFs)
    loadingsT=loadings.T #shape should be k components x m variables(so now 9 x 19)
    xs = pc[:,0] #pc1 scores
    ys = pc[:,1] #pc2 scores
    scalex = 1.0/(xs.max() - xs.min())
    scaley = 1.0/(ys.max() - ys.min())
    plt.figure(figsize=(14,14))
    xvector = loadingsT[0] #pc1 loadings
    yvector = loadingsT[1] #pc2 loadings
    for i in range(len(xvector)):
    # arrows project features (ie columns from csv) as vectors onto PC axes
        plt.arrow(0, 0, xvector[i]*max(xs), yvector[i]*max(ys),
                  color='r', width=0.001, head_width=0.01)
        plt.text(xvector[i]*max(xs)*1.2, yvector[i]*max(ys)*1.2,
                 list(mymatch.columns.values)[i], color='r')

    for i in range(len(xs)):
    # circles project documents (ie rows from csv) as points onto PC axes
        plt.plot(xs[i], ys[i], 'bo')
        plt.text(xs[i]*1.05, ys[i]*1.05, list(mymatch.index)[i], color='b')
    plt.show()
biplot12(fit_tran,loadings,labels=labels)
```



In [15]:

```
##pc1 vs pc3
def biplot23(pc,loadings,labels=None): #got this from stackexchange(only takes arrays and not DFs)
    loadingsT=loadings.T #shape should be k components x m variables(so now 9 x 19)
```

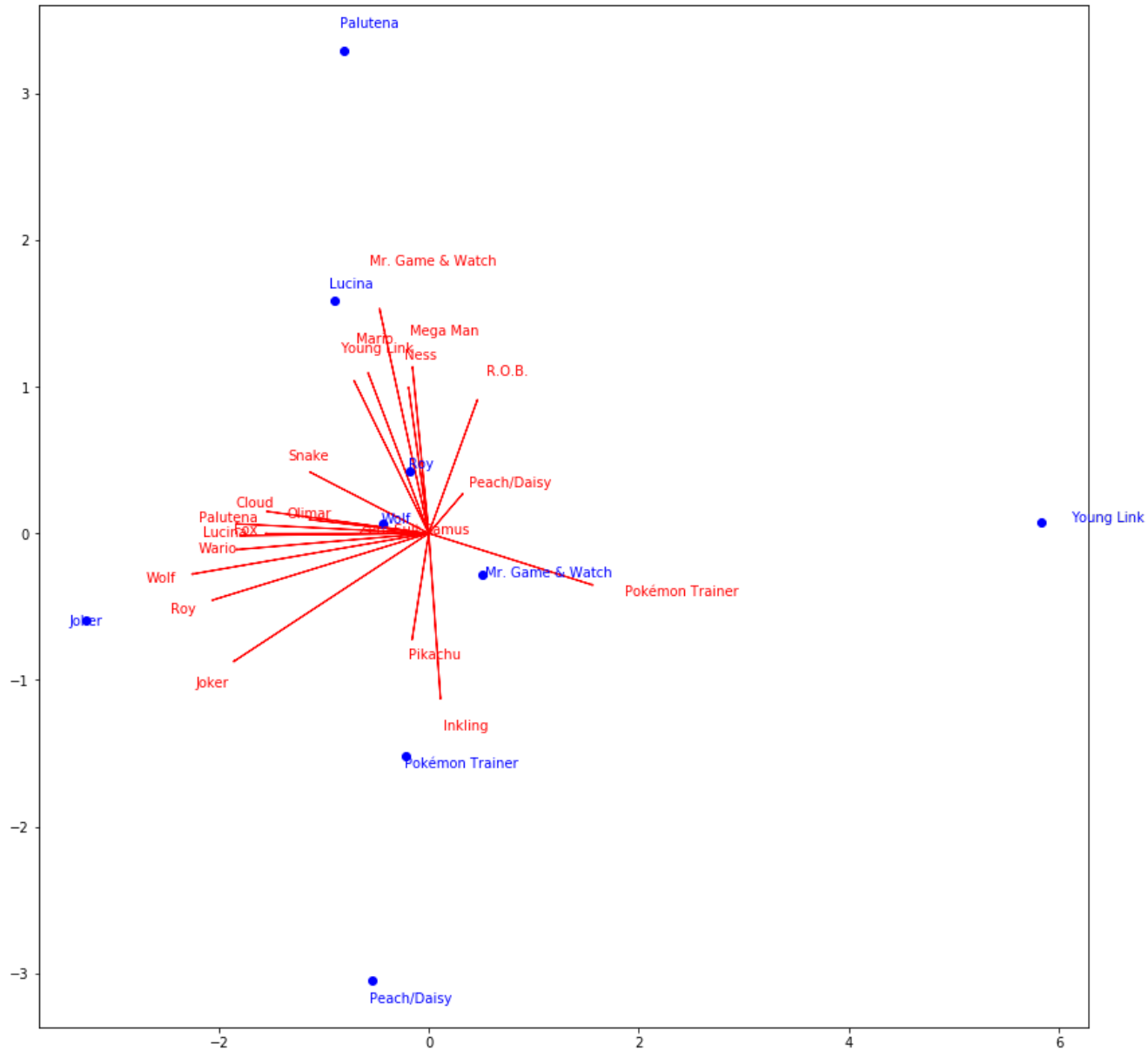
```
    loadingsT=loadings.T #shape should be k components x m variables(so now 9 x 19)
    xs = pc[:,0] #pc1 scores
    ys = pc[:,2] #pc2 scores
    scalex = 1.0/(xs.max() - xs.min())
    scaley = 1.0/(ys.max() - ys.min())
    plt.figure(figsize=(14,14))
    xvector = loadingsT[0] #pc1 loadings
    yvector = loadingsT[2] #pc2 loadings
    for i in range(len(xvector)):
    # arrows project features (ie columns from csv) as vectors onto PC axes
        plt.arrow(0, 0, xvector[i]*max(xs), yvector[i]*max(ys),
                  color='r', width=0.001, head_width=0.01)
        plt.text(xvector[i]*max(xs)*1.2, yvector[i]*max(ys)*1.2,
                 list(mymatch.columns.values)[i], color='r')

    for i in range(len(xs)):
    # circles project documents (ie rows from csv) as points onto PC axes
        plt.plot(xs[i], ys[i], 'bo')
        plt.text(xs[i]*1.05, ys[i]*1.05, list(mymatch.index)[i], color='b')
    plt.show()
biplot23(fit_tran,loadings,labels=labels)
```



In [16]:

```
##pc2 vs pc3
def biplot23(pc,loadings,labels=None): #got this from stackexchange(only takes arrays and not DFs)
    loadingsT=loadings.T #shape should be k components x m variables(so now 9 x 19)
    xs = pc[:,1] #pc1 scores
    ys = pc[:,2] #pc2 scores
    scalex = 1.0/(xs.max() - xs.min())
    scaley = 1.0/(ys.max() - ys.min())
    plt.figure(figsize=(14,14))
```

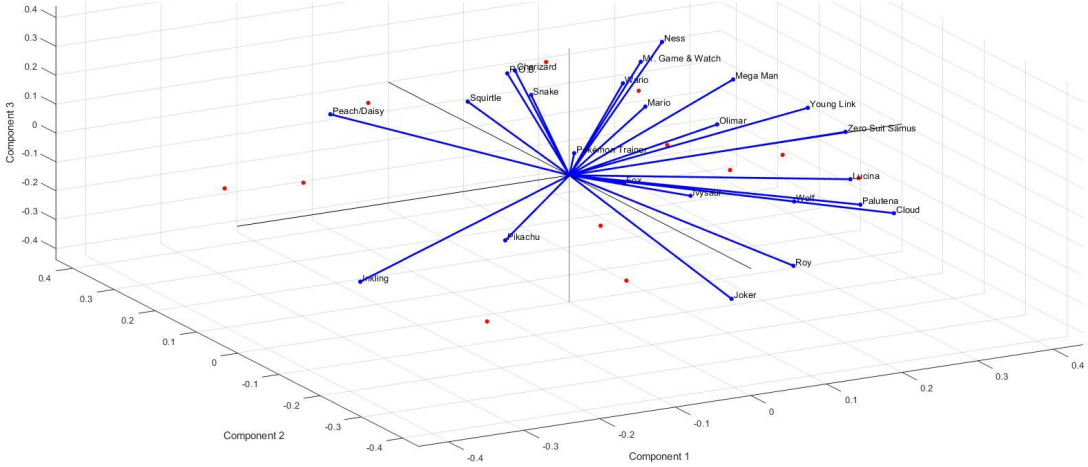
```
    xvector = loadingsT[1] #pc1 loadings
    yvector = loadingsT[2] #pc2 loadings
    for i in range(len(xvector)):
    # arrows project features (ie columns from csv) as vectors onto PC axes
        plt.arrow(0, 0, xvector[i]*max(xs), yvector[i]*max(ys),
                  color='r', width=0.001, head_width=0.01)
        plt.text(xvector[i]*max(xs)*1.2, yvector[i]*max(ys)*1.2,
                 list(mymatch.columns.values)[i], color='r')

    for i in range(len(xs)):
    # circles project documents (ie rows from csv) as points onto PC axes
        plt.plot(xs[i], ys[i], 'bo')
        plt.text(xs[i]*1.05, ys[i]*1.05, list(mymatch.index)[i], color='b')
    plt.show()
biplot23(fit_tran,loadings,labels=labels)
```



In [17]:

```
##pc1 vs. pc2 vs. pc3 done via MATLAB
Triplot=Image.open(r'C:\Users\Michael\Documents\Python Scripts\TriplotSSBU.jpg')
Triplot
```

Out[17]:



Notes

Given a matchup matrix, which is essentially just a payoff matrix, doing a principal component analysis should not only serve the purposes of feature extraction and dimensionality reduction, but also give us a more visual and efficient approach to finding characters to play in order to cover a wide range of matchups. The matrix has n-samples, being the characters that I play, and m-variables, being the characters that I'm concerned with playing against. The beauty of principal component analysis is that we need not even know or interpret the principal components in order to achieve our goal. Rather, all we need is a simple understanding of how it works.

With the way PCA works, and the nature of the data, the loadings and the scores establish a relationship between the samples, variables, and principal components. Since the samples, and variables are of the same medium(characters), then I will just simply denote them as "sample" vs. "variable." The loadings are give us how much each variable contributes to the variance of the data, in how it much contributes to the principal component. Some variables correlate positively, some negatively, and some not at all. In this way, in order to get the principal component scores, we relate this relationship betwen variables and principal components to the relationship between samples and variables. If the matchup matrix defines an entry, from the perspective of a row, as "how well the sample character does against the variable character," then we see that the PCA scores represent how well a sample character does against a principal component-defined type of variable character. This is just another way of expressing that the product of the transpose of the loadings matrix, and the data matrix yields the scores matrix.

Mathematically, if a sample character scores strongly positive in a principal component, then we can infer that the character does well against the variable characters that correlate strongly and positively to the principal component. Likewise, the same can be said in terms of a character whcih scores negatively. This is because the variable characters are seen as weights from the principal component they are related to, in the linear combinations that make up the scores. In the point of view of the biplots(and the triplot), as we graph both the scores as points, and the loadings as vectors, the scores which align in the direction of the loading vectors, with a greater magnitude in that direction, are the sample characters that do great against the variable characters(loading vectors). In essence, we've found a way to find which characters tend to naturally counter a wide range of other characters, and can represent this via correlational plots of the first 3 PCAs.

As an example, in principle component 1, we see that Ivysaur has the highest PCA score, and in the loadings, the variable characters that positively correlate with PC1 are Peach and Inkling. Hence, Ivysaur's pca score is a result of it doing well against both Inkling and Peach, which also passes a human judgement test, since those characters are pretty exploitable by Ivysaur. In this way, we can just use a distance function to measure the distance between the scores and the respective loadings as a metric of similarity, and ultimately, matchup viability, and if we construct a matrix out of these distances, we end up back with the matchup matrix.

In [ ]: