lab08

March 4, 2021

1 lab 08 - Distances and PCA

Name: Robb Alexander and Ryan Bailis Class: CSCI349 Semester: 2021SP Instructor: Brian King

[1]:
```
# Setting things up
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
from scipy.stats import zscore
from sklearn.metrics import pairwise_distances
from sklearn.decomposition import PCA
```

1) Set up a pandas data frame with the following 8 observations and 3 variables:

[2]:
```
data = [['A', 'excellent', 25],['C', 'fair', 32],['C', 'good', 60],['B',␣
↪'fair', 53],['A', 'poor', 23],['B', 'excellent', 37],['C', 'good', 45],['B',␣
↪'good', 49]]
index = ['A0','A1','A2','A3','A4','A5','A6','A7']
columns = ['test1','test2','test3']
df = pd.DataFrame(data=data, index=index, columns=columns)

df.test1 = pd.Categorical(values=df.test1, categories=['A', 'B', 'C'])
df.test2 = pd.Categorical(values=df.test2, ordered=True, categories=['poor',␣
↪'fair', 'good', 'excellent'])
df.test3 = pd.to_numeric(df.test3, downcast='integer')

df.info()
df
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8 entries, A0 to A7
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   test1   8 non-null      category
 1   test2   8 non-null      category
 2   test3   8 non-null      int8
dtypes: category(2), int8(1)
memory usage: 424.0+ bytes
```

[2]:
```
   test1      test2  test3
A0     A  excellent     25
A1     C       fair     32
A2     C       good     60
A3     B       fair     53
A4     A       poor     23
A5     B  excellent     37
A6     C       good     45
A7     B       good     49
```

2) Show the output of df.describe(include='all'). What does the include='all' parameter do?

[3]:
```
"""
include='all' essentially shows everything
 even if a column contains Nan values that
 would normally be excluded from the basic
 describe() function.
"""

df.describe(include='all')
```

[3]:
```
       test1 test2      test3
count      8     8   8.000000
unique     3     4        NaN
top        B  good        NaN
freq       3     3        NaN
mean     NaN   NaN  40.500000
std      NaN   NaN  13.416408
min      NaN   NaN  23.000000
25%      NaN   NaN  30.250000
50%      NaN   NaN  41.000000
75%      NaN   NaN  50.000000
max      NaN   NaN  60.000000
```

3) Show the output of df.test1.cat.categories, and df.test2.cat.categories. What is this showing? Does it work for df.test3.cat.categories? (If not, then comment this line out.)

[4]:
```
"""
The .cat variable is getting the categorical
 Accessor of the variable and then it shows
 the indexes with the .categories caller.
 This is the categories labels
```

```
"""
df.test1.cat.categories
```

```
[4]: Index(['A', 'B', 'C'], dtype='object')
```

```
[5]: df.test2.cat.categories
```

```
[5]: Index(['poor', 'fair', 'good', 'excellent'], dtype='object')
```

```
[6]: # df.test3.cat.categories
```

4) Show the output of df.test1.cat.codes, and df.test2.cat.codes. What is this showing?

```
[7]: """
This is showing the numbers that each one
 of the observations have according to the
 category numbers. This is the converted
 strings into numerics.
 This is the categories' int mappings

"""
df.test1.cat.codes
```

```
[7]: A0    0
A1    2
A2    2
A3    1
A4    0
A5    1
A6    2
A7    1
dtype: int8
```

```
[8]: df.test2.cat.codes
```

```
[8]: A0    3
A1    1
A2    2
A3    1
A4    0
A5    3
A6    2
A7    2
dtype: int8
```

5) Report the counts of each level of the categorical variables.

```
[9]: df.test1.cat.codes.value_counts()
```

```
[9]: 1    3
     2    3
     0    2
     dtype: int64
```

```
[10]: df.test2.cat.codes.value_counts()
```

```
[10]: 2    3
      1    2
      3    2
      0    1
      dtype: int64
```

6) Report a cross tabulation (i.e. contingency table) between test1 and test2. Include the margins (i.e. the sum of the rows and the columns) in your reported table (HINT: Look up pandas crosstab() function)

```
[11]: pd.crosstab(df.test1, df.test2, margins=True)
```

```
[11]: test2  poor  fair  good  excellent  All
      test1
      A         1     0     0          1    2
      B         0     1     1          1    3
      C         0     1     2          0    3
      All       1     2     3          2    8
```

7) From the previous table, store the contingency table without the margins in a variable called observed

```
[12]: observed = pd.crosstab(df.test1, df.test2)
```

8) Run a chi-squared test to determine whether test1 and test2 are dependent. Use the contingency table from the previous step. Clearly report the chi2 statistic, the p value, and the degrees of freedom, and then use the p-value to clearly state whether test1 and test2 are independent (assume p=0.05 threshold to test for independence)

```
[13]: c2, p, dof, expected = stats.chi2_contingency(observed)
      print("Chi Squared Value:", c2)
      print("Degrees of Freedom", dof)
      print("p-value:", p, "\nTherefore we can conclude that 0.37 > 0.05,\nrejecting␣
       ↪the null hypothesis that the two variables\nare independent; so they are␣
       ↪thus, dependent")
```

```
Chi Squared Value: 6.444444444444444
Degrees of Freedom 6
p-value: 0.37528525266160834
```

```
Therefore we can conclude that 0.37 > 0.05,
rejecting the null hypothesis that the two variables
are independent; so they are thus, dependent
```

9) Create a new data frame called df_num, that represents a numeric version of the above. Do NOT do any rescaling of your variables yet! NOTE: If you do this from a dataframe that has the categorical variables set up properly, then this step is simple to do. The two choices I generally follow are either: 1) use the cat member of your categorical data, which stores a CategoricalAccessor object (look it up), or use one of the encoders in the sklearn.preprocessing module. The first option is easier, and yet another reason why it's so important to take the time to preprocess your data as correctly and error-free as possible.

[14]:
```
import sklearn.preprocessing as pre

df_num = df.copy()
enc = pre.OrdinalEncoder(dtype=int, categories=[['A', 'B', 'C'], ['poor',␣
↪'fair', 'good', 'excellent']])
df_num[['test1', 'test2']] = enc.fit_transform(df_num[['test1', 'test2']])

df_num.info()
df_num
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8 entries, A0 to A7
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   test1   8 non-null      int64
 1   test2   8 non-null      int64
 2   test3   8 non-null      int8
dtypes: int64(2), int8(1)
memory usage: 500.0+ bytes
```

[14]:
```
    test1  test2  test3
A0      0      3     25
A1      2      1     32
A2      2      2     60
A3      1      1     53
A4      0      0     23
A5      1      3     37
A6      2      2     45
A7      1      2     49
```

10) As you learned in lecture, you must rescale your data to fall on a similar scale. There are different approaches to doing so. A standardized z-score is among the most common, but not necessarily always the best approach, especially when you are dealing with numeric representations of true categorical data. Rescaling your data to all fall

between 0 and 1 is also a common approach, particularly when you have categorical data.

[15]:
```
scaler = pre.MinMaxScaler()

df_num_zeroone = df_num.copy()
df_num_zeroone[:] = scaler.fit_transform(df_num)
df_num_zeroone
```

[15]:
```
    test1     test2     test3
A0    0.0  1.000000  0.054054
A1    1.0  0.333333  0.243243
A2    1.0  0.666667  1.000000
A3    0.5  0.333333  0.810811
A4    0.0  0.000000  0.000000
A5    0.5  1.000000  0.378378
A6    1.0  0.666667  0.594595
A7    0.5  0.666667  0.702703
```

11) Notice the value of test3. Quite often, when we have solid knowledge of what we expect our range to be, then we can rescale our data using that knowledge. In the case of test3, you learned that the data must fall between 0 and 100. Therefore, reassign test3 so that the min and max before rescaling are assuming to be between 0 and 100, respectively. (i.e. simply divide the original variable by 100)

[16]:
```
df_num_zeroone.loc[:,'test3'] = df_num.test3 / 100
df_num_zeroone
```

[16]:
```
    test1     test2  test3
A0    0.0  1.000000   0.25
A1    1.0  0.333333   0.32
A2    1.0  0.666667   0.60
A3    0.5  0.333333   0.53
A4    0.0  0.000000   0.23
A5    0.5  1.000000   0.37
A6    1.0  0.666667   0.45
A7    0.5  0.666667   0.49
```

12) Compute a single distance matrix called distmat_zeroone. Use a standard Euclidean distance measure. Your reported result should be an 8x8 matrix with appropriately labeled rows and columns. (HINT – study the output of the distance matrix functions! They do not output a square matrix. As discussed in lecture you can use either pairwise_distances from scikit-learn, or the pdist and squareform functions in scipy.spatial.distance). Be sure to label the rows and columns to be the index from the input dataframe. Round it to 3 significant digits.

[17]:
```
names = ['A0','A1','A2','A3','A4','A5','A6','A7']
```

```
distmat_zeroone = pd.DataFrame(pairwise_distances(X=df_num_zeroone).round(3),␣
↪index=names, columns=names)
distmat_zeroone
```

[17]:
```
       A0     A1     A2     A3     A4     A5     A6     A7
A0  0.000  1.204  1.111  0.879  1.000  0.514  1.073  0.647
A1  1.204  0.000  0.435  0.542  1.058  0.835  0.358  0.625
A2  1.111  0.435  0.000  0.605  1.258  0.643  0.150  0.512
A3  0.879  0.542  0.605  0.000  0.672  0.686  0.606  0.336
A4  1.000  1.058  1.258  0.672  0.000  1.127  1.222  0.873
A5  0.514  0.835  0.643  0.686  1.127  0.000  0.606  0.354
A6  1.073  0.358  0.150  0.606  1.222  0.606  0.000  0.502
A7  0.647  0.625  0.512  0.336  0.873  0.354  0.502  0.000
```

13) Output the top three closest pairs of observations. You MUST write Python code to report these results! Do not simply print out your distance matrix and tell me your answers! Consider that this may have been thousands of observations! Always generate reported answers in code whenever you can! For each pair, output the pair of observations from the original dataframe, and the distance between them. (HINT: I found this easier to do with the output of pdist.)

[18]:
```
import heapq, itertools
diag_df = distmat_zeroone.mask(np.tril(np.ones(distmat_zeroone.shape,␣
↪dtype=bool)))
pairs = list(itertools.product(names, names))
output = []

for i, v in enumerate(pairs):
    value = diag_df.loc[v[0], v[1]]
    if not np.isnan(value):
        output.append((value, v))

output[:5]
```

[18]:
```
[(1.204, ('A0', 'A1')),
 (1.111, ('A0', 'A2')),
 (0.879, ('A0', 'A3')),
 (1.0, ('A0', 'A4')),
 (0.514, ('A0', 'A5'))]
```

[19]:
```
small = heapq.nsmallest(3, output)
for i, (dist, (row, col)) in enumerate(small):
    print(f'Closest: #{i}: [\'{row}\', \'{col}\'] dist={dist}')
    print(pd.concat([df.loc[row], df.loc[col]], axis=1).T, "\n")
```

```
Closest: #0: ['A2', 'A6'] dist=0.15
   test1 test2 test3
```

```
A2      C  good     60
A6      C  good     45

Closest: #1: ['A3', 'A7'] dist=0.336
   test1 test2 test3
A3      B  fair     53
A7      B  good     49

Closest: #2: ['A5', 'A7'] dist=0.354
   test1      test2 test3
A5      B  excellent    37
A7      B       good    49
```

14) Now, output the three most distant (least similar) pairs of observations. Again, for each pair, output the two observations, and the distance between them

[20]:
```python
largest = heapq.nlargest(3, output)

for i, (dist, (row, col)) in enumerate(largest):
    print(f'Farthest: #{i}: [\'{row}\', \'{col}\'] dist={dist}')
    print(pd.concat([df.loc[row], df.loc[col]], axis=1).T, "\n")
```

```
Farthest: #0: ['A2', 'A4'] dist=1.258
   test1 test2 test3
A2      C  good     60
A4      A  poor     23

Farthest: #1: ['A4', 'A6'] dist=1.222
   test1 test2 test3
A4      A  poor     23
A6      C  good     45

Farthest: #2: ['A0', 'A1'] dist=1.204
   test1      test2 test3
A0      A  excellent    25
A1      C       fair    32
```

15) Create a new data frame, df_num_binarized, that stores the a binarized version for test1 and test2.

[21]:
```python
df_num_binarized = pd.get_dummies(data=df, prefix=['test1', 'test2'])

cols = list(df_num_binarized.columns)
df_num_binarized = df_num_binarized[cols[1:] + cols[0:1]]

df_num_binarized.loc[:,'test3'] = df_num_binarized.test3 / 100
df_num_binarized
```

[21]:
```
    test1_A  test1_B  test1_C  test2_poor  test2_fair  test2_good  \
A0        1        0        0           0           0           0
A1        0        0        1           0           1           0
A2        0        0        1           0           0           1
A3        0        1        0           0           1           0
A4        1        0        0           1           0           0
A5        0        1        0           0           0           0
A6        0        0        1           0           0           1
A7        0        1        0           0           0           1

    test2_excellent  test3
A0                1   0.25
A1                0   0.32
A2                0   0.60
A3                0   0.53
A4                0   0.23
A5                1   0.37
A6                0   0.45
A7                0   0.49
```

16) Now, compute distmat_binarized by computing the distance matrix for the df_binarized.

[22]:
```
names = ['A0','A1','A2','A3','A4','A5','A6','A7']
distmat_binarized = pd.DataFrame(pairwise_distances(X=df_num_binarized).
 ↪round(3), index=names, columns=names)
```

17) Report the three closest pairs, and the three most distant pairs from distmat_binarized

[23]:
```
diag_df = distmat_binarized.mask(np.tril(np.ones(distmat_binarized.shape,␣
 ↪dtype=bool)))
pairs = list(itertools.product(names, names))
output = []

for i, v in enumerate(pairs):
    value = diag_df.loc[v[0], v[1]]
    if not np.isnan(value):
        output.append((value, v))
```

[24]:
```
small = heapq.nsmallest(3, output)
for i, (dist, (row, col)) in enumerate(small):
    print(f'Closest: #{i}: [\'{row}\', \'{col}\'] dist={dist}')
    print(pd.concat([df.loc[row], df.loc[col]], axis=1).T, "\n")
```

```
Closest: #0: ['A2', 'A6'] dist=0.15
   test1 test2 test3
A2     C  good    60
```

```
A6     C  good    45

Closest: #1: ['A0', 'A4'] dist=1.414
   test1      test2 test3
A0     A  excellent    25
A4     A       poor    23

Closest: #2: ['A3', 'A7'] dist=1.415
   test1 test2 test3
A3     B  fair    53
A7     B  good    49
```

[25]:
```
largest = heapq.nlargest(3, output)

for i, (dist, (row, col)) in enumerate(largest):
    print(f'Farthest: #{i}: [\'{row}\', \'{col}\'] dist={dist}')
    print(pd.concat([df.loc[row], df.loc[col]], axis=1).T, "\n")
```

```
Farthest: #0: ['A2', 'A4'] dist=2.034
   test1 test2 test3
A2     C  good    60
A4     A  poor    23

Farthest: #1: ['A0', 'A2'] dist=2.03
   test1      test2 test3
A0     A  excellent    25
A2     C       good    60

Farthest: #2: ['A3', 'A4'] dist=2.022
   test1 test2 test3
A3     B  fair    53
A4     A  poor    23
```

18) Take a moment and compare and contrast your results. Which method do you think have the better results? Why? Which variable do you think was the distinguishing player in affecting the different outcomes between both of the above approaches to transforming your data to numeric results? Why? Summarize what would have been the best transformation to make for all three variables that would have given the most accurate results. The general difference between the two is that it seems that the binarization one judges distance in the amount of exact matches instead of the caring about the order as much. By looking at the closest distance for the binarized one shows that it thinks poor and excellent are close enough to be the lowest distance. The binarized one thus cares more for the numerical chances with less focus on the So the important variable we had to take care of was test3. In the first one (zeroone), it cared more for the test2 and less so for the values of test3, while the opposite for the second one. So the best choice for test1 is binarization, test2 is zeroone, and test3 is factor of 100

19) Load in your next dataset using the following: df_car_crashes = sns.load_dataset('car_crashes')

```
[26]: df_car_crashes = sns.load_dataset('car_crashes')
```

20) Preprocess your data. Minimally, you should move the state code to become the index for the dataframe, and then drop that column from your dataframe. Show the first five rows.

```
[27]: df_car_crashes.index = df_car_crashes.abbrev
      df_car_crashes.drop('abbrev', axis=1, inplace=True)
      df_car_crashes.head(5)
```

```
[27]:        total  speeding  alcohol  not_distracted  no_previous  ins_premium  \
      abbrev
      AL       18.8     7.332    5.640          18.048       15.040       784.55
      AK       18.1     7.421    4.525          16.290       17.014      1053.48
      AZ       18.6     6.510    5.208          15.624       17.856       899.47
      AR       22.4     4.032    5.824          21.056       21.280       827.34
      CA       12.0     4.200    3.360          10.920       10.680       878.41

             ins_losses
      abbrev
      AL         145.08
      AK         133.93
      AZ         110.35
      AR         142.39
      CA         165.63
```

21) Create a new dataframe called df_car_crashes_zscore that represents the zscore transformation for df_car_crashes. Again, show the first five rows.

```
[28]: df_car_crashes_zscore = df_car_crashes.apply(zscore)
      df_car_crashes_zscore.head(5)
```

```
[28]:           total  speeding   alcohol  not_distracted  no_previous  \
      abbrev
      AL      0.737446  1.168148  0.439938        1.002301     0.277692
      AK      0.565936  1.212695 -0.211311        0.608532     0.807258
      AZ      0.688443  0.756709  0.187615        0.459357     1.033141
      AR      1.619498 -0.483614  0.547408        1.676052     1.951700
      CA     -0.928653 -0.399524 -0.891763       -0.594276    -0.891968

             ins_premium  ins_losses
      abbrev
      AL       -0.580083    0.430514
      AK        0.943258   -0.022900
      AZ        0.070876   -0.981778
      AR       -0.337701    0.321125
```

```
CA         -0.048418    1.266178
```

22) Create a distance matrix called distmat_cars based on the df_car_crashes_zscore. Display the entire distance matrix.

[29]:
```
distmat_cars = pd.DataFrame(pairwise_distances(X=df_car_crashes_zscore),␣
↪index=df_car_crashes_zscore.index, columns=df_car_crashes_zscore.index)
distmat_cars
```

[29]:
```
abrev         AL        AK        AZ        AR        CA            CO  \
abrev
AL      0.000000  1.848559  1.875942  2.616265  3.450372  2.687193e+00
AK      1.848559  0.000000  1.461454  2.961700  3.503615  2.674699e+00
AZ      1.875942  1.461454  0.000000  2.592934  3.873068  2.690949e+00
AR      2.616265  2.961700  2.592934  0.000000  4.773783  4.082043e+00
CA      3.450372  3.503615  3.873068  4.773783  0.000000  1.382473e+00
CO      2.687193  2.674699  2.690949  4.082043  1.382473  2.107342e-08
CT      4.022977  3.719860  4.341409  5.511677  1.360401  2.171261e+00
DE      2.426745  1.305465  2.322674  3.375632  2.801377  2.348937e+00
DC      6.627070  5.923340  6.312294  7.648001  3.725440  4.294426e+00
FL      2.863217  2.014017  2.471886  2.722795  3.294284  2.884757e+00
GA      2.740439  2.609142  2.619278  3.132498  1.994627  1.592712e+00
HI      1.967575  2.309612  2.090333  3.887437  4.367894  3.364956e+00
ID      3.227303  3.462721  2.330031  4.160356  4.023485  2.745180e+00
IL      2.648799  2.872699  2.779181  4.036791  1.416124  5.753356e-01
IN      2.936806  3.210155  2.440773  3.699517  2.832514  1.789320e+00
IA      3.114833  3.587012  2.842219  3.547224  3.017301  2.214125e+00
KS      1.858608  2.173195  1.774256  2.804821  2.488592  1.447705e+00
KY      1.951130  2.154966  1.855779  1.793132  3.376552  2.635520e+00
LA      3.864433  3.260930  4.243512  4.014378  4.931876  4.790391e+00
ME      2.803998  3.213492  2.324265  4.160980  3.400465  2.191438e+00
MD      4.050022  3.809894  4.515303  5.093197  1.647736  2.550081e+00
MA      5.453420  5.160132  5.288766  6.535539  2.506906  3.019168e+00
MI      3.471272  2.994360  3.509289  4.352050  1.698430  2.010389e+00
MN      4.735183  4.795405  4.670325  5.819223  2.042084  2.280417e+00
MS      4.470256  4.270578  4.110102  4.692240  3.437689  3.078930e+00
MO      1.086939  2.000188  2.032453  3.294652  2.593112  1.781583e+00
MT      3.494754  3.874574  2.985649  3.963502  6.286038  5.169746e+00
NE      3.307722  3.668479  2.927505  3.634205  2.980809  2.250106e+00
NV      2.248853  1.565833  2.021377  3.406048  2.217568  1.521520e+00
NH      3.706577  3.863496  3.575319  5.069694  2.025549  1.486772e+00
NJ      5.352132  4.657829  5.230140  6.069845  2.768178  3.475273e+00
NM      2.710780  2.482601  1.747059  2.587646  3.284483  2.254951e+00
NY      4.241067  3.472039  4.115832  5.286360  2.134562  2.525314e+00
NC      1.240410  2.276668  1.781326  3.167884  3.005854  1.983954e+00
ND      3.890314  4.709646  3.884830  3.080659  6.754302  5.867464e+00
OH      2.571284  3.237271  2.846455  3.806845  2.062510  1.403170e+00
```

```
OK      1.862936  2.387338  2.898265  2.211916  3.817251  3.464228e+00
OR      3.810394  3.647019  3.202878  4.971264  2.587825  1.657845e+00
PA      1.220905  1.621449  2.302468  3.249592  3.815218  3.163196e+00
RI      4.196825  3.701274  4.172927  5.395331  1.860928  2.315362e+00
SC      3.510426  4.067596  3.673676  3.676638  6.732674  5.840733e+00
SD      2.272782  3.030764  1.805917  2.756086  4.729624  3.599348e+00
TN      1.760396  2.612893  2.413952  2.090210  2.984643  2.449727e+00
TX      1.768022  1.983290  2.496229  2.858256  4.199246  3.623915e+00
UT      4.073530  3.820444  3.638752  5.375719  2.525129  1.893114e+00
VT      2.984473  3.230915  2.564600  4.012612  2.637784  1.579660e+00
VA      3.691474  3.939532  3.910659  4.577666  1.211871  1.580230e+00
WA      4.099448  3.874399  3.716797  5.500790  2.322342  1.824522e+00
WV      2.658679  2.759540  3.094545  2.401244  5.589191  4.943274e+00
WI      3.882304  4.080096  3.452450  5.215076  3.094339  2.114426e+00
WY      1.345666  1.794331  1.165002  3.105415  3.392714  2.217199e+00

abbrev        CT        DE        DC        FL  …        SD        TN  \
abbrev                                          …
AL      4.022977  2.426745  6.627070  2.863217  …  2.272782  1.760396
AK      3.719860  1.305465  5.923340  2.014017  …  3.030764  2.612893
AZ      4.341409  2.322674  6.312294  2.471886  …  1.805917  2.413952
AR      5.511677  3.375632  7.648001  2.722795  …  2.756086  2.090210
CA      1.360401  2.801377  3.725440  3.294284  …  4.729624  2.984643
CO      2.171261  2.348937  4.294426  2.884757  …  3.599348  2.449727
CT      0.000000  2.792618  3.100788  3.490418  …  5.401468  3.816674
DE      2.792618  0.000000  5.106619  1.442845  …  3.827587  2.556844
DC      3.100788  5.106619  0.000000  5.343223  …  7.356393  6.230942
FL      3.490418  1.442845  5.343223  0.000000  …  3.696806  2.343908
GA      2.812662  2.197382  4.611878  1.878250  …  3.457118  1.848972
HI      4.523348  2.949539  6.947997  3.693017  …  2.599939  3.261208
ID      4.726892  4.006878  6.093580  4.130941  …  2.181769  3.440391
IL      2.256232  2.540638  4.370549  2.929698  …  3.451012  2.344501
IN      3.702964  3.339940  5.138253  3.256948  …  2.607172  2.552997
IA      4.043616  3.709853  5.452221  3.431343  …  2.760459  2.449758
KS      3.354811  2.326150  5.470177  2.440633  …  2.510956  1.426519
KY      4.080040  2.379312  6.165330  1.882937  …  2.495190  1.163669
LA      4.801947  2.755112  7.275689  3.124341  …  5.476430  3.904461
ME      4.083418  3.642062  5.661144  3.886650  …  2.282470  3.080945
MD      1.521528  2.716766  4.030791  3.302111  …  5.648360  3.474840
MA      2.347552  4.425745  1.789043  4.599359  …  6.061665  4.915531
MI      1.670508  2.110640  3.504421  2.157263  …  4.563139  2.958103
MN      2.658911  4.295743  3.168166  4.460124  …  5.187956  4.140718
MS      3.816565  3.612514  5.417047  3.701167  …  5.072581  3.417782
MO      3.141856  2.171346  5.764451  2.835552  …  2.640388  1.835812
MT      6.566770  4.646580  8.635002  4.679935  …  2.416251  4.295997
NE      3.813197  3.550278  5.170326  3.138947  …  2.936898  2.434622
NV      2.439824  1.037125  4.634382  1.637924  …  3.326173  2.265584
```

```
NH      2.706122  3.648531  3.929630  3.951091  …  4.023538  3.453310
NJ      2.134778  3.619032  2.179169  3.634153  …  6.362815  4.733901
NM      3.963412  2.525134  5.666833  2.150652  …  2.637339  1.922194
NY      1.402568  2.514226  2.725368  2.851930  …  5.338243  3.940502
NC      3.711126  2.742267  6.027567  3.128901  …  1.948967  1.937027
ND      7.307187  5.313225  9.499838  4.860228  …  2.870567  4.058068
OH      3.006352  3.111211  4.970356  3.207489  …  3.008980  2.113621
OK      4.358405  2.422355  7.070035  2.638959  …  3.614204  1.870577
OR      3.115736  3.552542  4.055564  3.820087  …  3.835613  3.599599
PA      4.088063  2.219743  6.799545  3.089770  …  3.160487  2.666025
RI      1.096656  2.774926  2.653771  3.125753  …  5.217107  3.864479
SC      7.019778  4.826271  9.420597  4.823476  …  3.252538  4.405594
SD      5.401468  3.827587  7.356393  3.696806  …  0.000000  2.763780
TN      3.816674  2.556844  6.230942  2.343908  …  2.763780  0.000000
TX      4.346224  2.171300  7.019399  2.622673  …  3.276498  2.529293
UT      3.070455  3.771261  3.805078  4.179509  …  4.352226  4.049046
VT      3.461110  3.324290  4.898792  3.396841  …  2.816778  2.746196
VA      2.371614  3.389056  3.999740  3.459389  …  4.470627  2.817993
WA      2.500124  3.571201  3.303523  3.962686  …  4.418741  3.991185
WV      5.938001  3.438088  8.418524  3.439624  …  3.570243  3.369988
WI      3.628320  4.037819  4.939871  4.436470  …  3.829554  3.706725
WY      3.899592  2.427114  6.196665  2.962916  …  1.926057  2.216252

abbrev        TX        UT        VT        VA        WA            WV  \
abbrev
AL      1.768022  4.073530  2.984473  3.691474  4.099448  2.658679e+00
AK      1.983290  3.820444  3.230915  3.939532  3.874399  2.759540e+00
AZ      2.496229  3.638752  2.564600  3.910659  3.716797  3.094545e+00
AR      2.858256  5.375719  4.012612  4.577666  5.500790  2.401244e+00
CA      4.199246  2.525129  2.637784  1.211871  2.322342  5.589191e+00
CO      3.623915  1.893114  1.579660  1.580230  1.824522  4.943274e+00
CT      4.346224  3.070455  3.461110  2.371614  2.500124  5.938001e+00
DE      2.171300  3.771261  3.324290  3.389056  3.571201  3.438088e+00
DC      7.019399  3.805078  4.898792  3.999740  3.303523  8.418524e+00
FL      2.622673  4.179509  3.396841  3.459389  3.962686  3.439624e+00
GA      3.484814  2.800175  1.991735  1.741679  2.777024  4.447891e+00
HI      2.117453  4.479970  3.571822  4.728124  4.265045  3.336821e+00
ID      4.456028  2.685512  1.551591  3.638644  2.946004  5.210840e+00
IL      3.624767  2.045405  1.378476  1.422768  1.823447  4.980795e+00
IN      4.143568  2.150286  0.431526  2.248464  2.271576  5.075878e+00
IA      4.408638  2.593952  1.109788  2.229273  2.815298  5.169456e+00
KS      3.018528  2.764565  1.635965  2.319119  2.900155  3.936542e+00
KY      2.642535  3.921383  2.720046  3.201370  3.962989  3.125623e+00
LA      2.495853  6.434672  5.818144  5.553106  6.195012  3.135338e+00
ME      4.117779  2.305619  1.128301  3.111323  2.438338  5.077069e+00
MD      4.164078  3.915989  3.993253  2.514904  3.571909  5.628262e+00
MA      6.086706  2.707846  3.469482  2.473351  2.036127  7.538069e+00
```

```
MI      3.794287  3.018837  2.907516  2.116173  2.590439  5.085073e+00
MN      5.710425  1.939448  2.442816  1.514164  1.649265  7.041151e+00
MS      4.594207  4.310691  3.829552  3.268082  3.985916  5.837810e+00
MO      2.236473  3.282298  2.329714  2.912129  3.171201  3.599080e+00
MT      3.393748  6.083821  4.738011  6.275800  5.859396  3.636479e+00
NE      4.264643  2.890587  1.339458  2.176171  2.679777  5.253256e+00
NV      2.595977  2.900213  2.336402  2.625465  2.666985  3.912213e+00
NH      4.822693  1.242724  1.341300  1.686579  0.999245  6.119078e+00
NJ      5.470085  3.905327  4.356134  3.154822  3.342637  6.801718e+00
NM      3.292759  3.350379  2.146540  2.939637  3.339743  4.062231e+00
NY      4.393399  3.154522  3.527207  2.774801  2.597656  5.784860e+00
NC      2.743189  3.102607  1.881427  3.026984  3.144146  3.763090e+00
ND      3.818408  7.030391  5.370539  6.521633  6.888195  3.403958e+00
OH      3.773061  2.433395  1.150318  1.617616  2.274646  4.972930e+00
OK      1.710763  5.142844  4.155705  4.132564  5.158602  2.214235e+00
OR      4.829134  1.091142  1.295080  2.273842  1.022188  6.031538e+00
PA      1.392841  4.549810  3.805005  4.364385  4.549805  2.429254e+00
RI      4.449817  2.970212  3.251505  2.441335  2.200508  5.981297e+00
SC      2.984889  7.044900  5.693158  6.856409  6.880602  2.545397e+00
SD      3.276498  4.352226  2.816778  4.470627  4.418741  3.570243e+00
TN      2.529293  4.049046  2.746196  2.817993  3.991185  3.369988e+00
TX      0.000000  5.209589  4.201931  4.635763  4.967709  1.982593e+00
UT      5.209589  0.000000  1.817711  2.422996  1.247430  6.312553e+00
VT      4.201931  1.817711  0.000000  2.148753  1.914058  5.241011e+00
VA      4.635763  2.422996  2.148753  0.000000  2.300740  5.870215e+00
WA      4.967709  1.247430  1.914058  2.300740  0.000000  6.353763e+00
WV      1.982593  6.312553  5.241011  5.870215  6.353763  5.960464e-08
WI      4.885596  2.160709  1.854834  2.782838  1.908689  6.204276e+00
WY      2.368477  3.370680  2.294087  3.532372  3.364606  3.428472e+00

abbrev        WI        WY
abbrev
AL      3.882304  1.345666
AK      4.080096  1.794331
AZ      3.452450  1.165002
AR      5.215076  3.105415
CA      3.094339  3.392714
CO      2.114426  2.217199
CT      3.628320  3.899592
DE      4.037819  2.427114
DC      4.939871  6.196665
FL      4.436470  2.962916
GA      3.237933  2.680223
HI      3.832770  1.485756
ID      2.220161  2.207048
IL      2.162329  2.263062
IN      2.111961  2.300292
```

```
IA      2.639014  2.753406
KS      2.642380  1.552764
KY      3.815375  2.165643
LA      6.342526  4.357395
ME      1.876843  1.896015
MD      4.223813  4.141530
MA      3.510709  5.017776
MI      3.699237  3.446214
MN      2.601384  4.286404
MS      3.374649  3.950141
MO      3.020349  1.151658
MT      5.189069  3.125564
NE      2.568873  2.865231
NV      3.207777  1.986116
NH      1.665251  3.105043
NJ      4.751999  5.207632
NM      2.980717  2.141581
NY      3.969970  4.042974
NC      2.798840  0.965639
ND      6.246906  4.132499
OH      2.279745  2.322495
OK      5.031478  2.787489
OR      1.345338  2.938413
PA      4.478598  1.877227
RI      3.583398  3.947220
SC      6.392267  3.814222
SD      3.829554  1.926057
TN      3.706725  2.216252
TX      4.885596  2.368477
UT      2.160709  3.370680
VT      1.854834  2.294087
VA      2.782838  3.532372
WA      1.908689  3.364606
WV      6.204276  3.428472
WI      0.000000  2.891556
WY      2.891556  0.000000

[51 rows x 51 columns]
```

23) An interesting way to suggest outliers is to take a distance matrix, aggregate the mean over each row or column, then sort the output in order. Why would this work? Because an observation that is an outlier should have a relatively high mean distance to all other observations! Do this, and output the entire ordered list in descending order. (HINT: DC should be your largest outlier.)

[30]:
```
"""
This works because the average distance is how
```

```
 closely the data connects with other datas in
 the list. If the average is high, then it means
 the data is further and less similar, which is
 the idea of an outlier.
"""

distmat_cars.mean().sort_values(ascending=False)
```

[30]:
```
abbrev
DC    5.265664
ND    5.257310
SC    5.214264
LA    4.767480
MT    4.759816
WV    4.512766
NJ    4.310802
MA    4.197513
MS    4.048305
AR    3.897635
MD    3.749939
MN    3.719807
OK    3.618059
HI    3.591755
SD    3.585247
TX    3.572109
NY    3.531806
CT    3.514075
WI    3.468777
ID    3.440899
PA    3.440610
UT    3.438146
RI    3.434056
WA    3.296993
FL    3.205449
VA    3.156358
OR    3.130498
CA    3.116879
ME    3.105251
NH    3.100905
IA    3.098437
AK    3.091304
MI    3.072811
NE    3.065206
AL    3.033363
DE    3.021971
AZ    2.999864
TN    2.965254
```

```
KY    2.964333
NM    2.914953
IN    2.841386
WY    2.839487
VT    2.820077
NC    2.777656
OH    2.746863
MO    2.734449
GA    2.726648
NV    2.627495
CO    2.617432
IL    2.602264
KS    2.573618
dtype: float64
```

24) From this analysis, which 4 states seem to be strongest outliers?

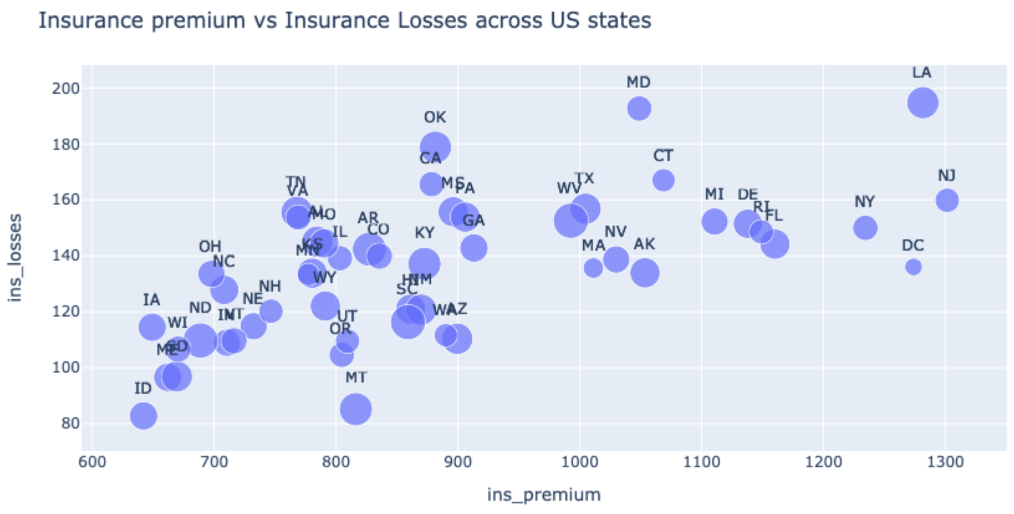
[31]:
```
distmat_cars.mean().sort_values(ascending=False)[:4].index.values.tolist()
```

[31]:
```
['DC', 'ND', 'SC', 'LA']
```

The outliers are DC, ND, SC, LA.

25) OK. Let's explore the data visually. First, using the original, unscaled data frame df_car_crashes, create a scatter plot of insurance premiums vs. insurance losses, with total number of accidents as the size of the point. Create a label near to every point representing the two letter state code.

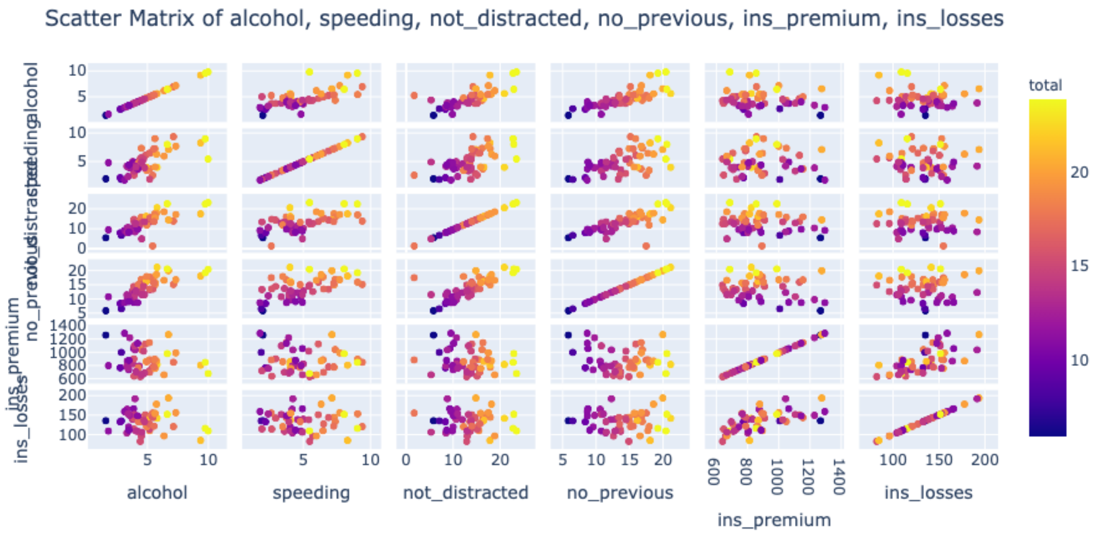
[32]:
```
import plotly.express as px
fig = px.scatter(df_car_crashes, x='ins_premium', y='ins_losses',size='total',␣
 ↪text=df_car_crashes.index, render_mode='webgl')
fig.update_traces(textposition='top center')
fig.update_layout(title_text='Insurance premium vs Insurance Losses across US␣
 ↪states')
fig.show()
```

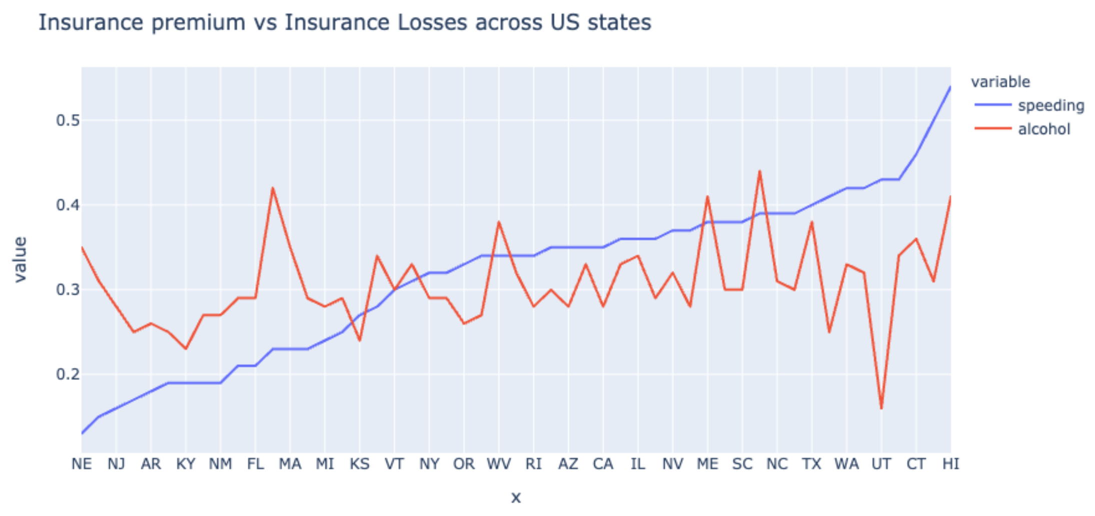


26) Next, generate two interesting plots that show some relationships between variables in the data. Try to use as many variables as you can without creating chaos! Don't just throw in multiple variables for the sake of showing them, only include them if it makes sense to do so. Your aim is to derive meaning from your data. Good visualizations tell a story. Strive to use at least one additional variable as size, color, or shape in your data, so you can show more than just 2 variables on a single plot. Add titles, legends and label your axes as appropriate.

[33]:
```
"""
The two plots I chose to show are the relationships between the variables
 in a scatter plot using total crashes as color. In the resulting matrix
 plot, it shows that this data is insufficient to draw a conclusion between
 alcohol, speeding, and not distracted. This is because they are not normalized
 with the total amount. The top lefts all have higher totals regardless of
 the type of crash.
 The matrix also shows that in general, total crashes tends to mean that
 the insurance will be higher. This should be expected, but these plots
 force us to use normalized data.
The second plot uses percentages of the total in relation with the alcohol
 vs speeding. This is then sorted to show the percent of speeding normalized
 with the total. This shows that most states have more speeding than alcohol
 crashes, which should be the norm. The states with higher alcohol than
 speeding also seem to have 'notorious drivers' or are more rural states.
 An outlier is also seen in utah where there is very littler drinking in general
 due to their demographic.
 Another outlier is MT, which has the law that allows for the consumpition of
 alcohol while driving if you're under a certain threshold.
"""
```

```
fig = px.scatter_matrix(df_car_crashes, dimensions=["alcohol", "speeding",␣
↪"not_distracted", "no_previous", "ins_premium", "ins_losses"], color="total")
fig.update_layout(title_text='Scatter Matrix of alcohol, speeding,␣
↪not_distracted, no_previous, ins_premium, ins_losses')
fig.show()

df_car_crashes_percent = df_car_crashes.copy()
df_car_crashes_percent.alcohol /= df_car_crashes_percent.total
df_car_crashes_percent.speeding /= df_car_crashes_percent.total
df_car_crashes_percent.not_distracted /= df_car_crashes_percent.total
df_car_crashes_percent = df_car_crashes_percent.sort_values(by=['speeding'])
fig = px.line(df_car_crashes_percent, x=df_car_crashes_percent.
↪sort_values(by=['speeding']).index, y=['speeding','alcohol'],␣
↪render_mode='webgl')
fig.update_layout(title_text='Insurance premium vs Insurance Losses across US␣
↪states')

fig.show()
```





27) Run a full PCA on the z_score transformed data. Set n_components to be the same number of columns as the data. Be sure to fit the data to your PCA model, and then output the components, explained variance, and the explained variance ratio.

```
[34]: pca = PCA(n_components=len(df_car_crashes_zscore.columns))
      pca.fit(df_car_crashes_zscore)
      print("Components:\n", pca.components_, "\n")
      print("Explained Variance:\n", pca.explained_variance_, "\n")
      print("Explained Variance Ratio:\n", pca.explained_variance_ratio_)
```

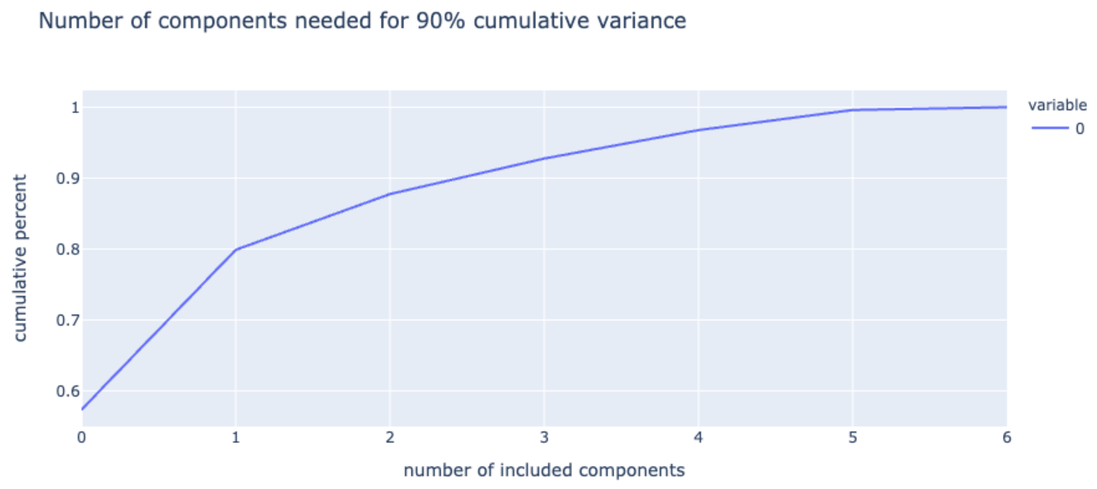
```
Components:
 [[ 0.47947078  0.37534719  0.45437635  0.4380328   0.45703414 -0.1308319
  -0.06996048]
 [ 0.06893769  0.0765846   0.03345835  0.04237473  0.0961294   0.6852266
   0.71252436]
 [-0.26908802  0.81826935  0.08293253 -0.12616845 -0.31798812  0.25614247
  -0.26173503]
 [ 0.0389558  -0.36374887  0.14834351  0.1712655   0.03948141  0.65639617
  -0.61839859]
 [ 0.14597659 -0.02282818  0.35479821 -0.85646854  0.33611019  0.04214531
  -0.06327152]
 [-0.16970508 -0.22479246  0.7837677   0.08510479 -0.50401185 -0.11577348
   0.17805184]
 [ 0.80082487  0.01784783 -0.15285774 -0.14247844 -0.55875371  0.04126619
  -0.02804966]]

Explained Variance:
 [4.0942308  1.6095732  0.56161403 0.35753958 0.2863854  0.20263316
 0.02802383]
```

```
Explained Variance Ratio:
 [0.57342168 0.22543042 0.07865743 0.05007557 0.04011    0.02837999
 0.00392491]
```

28) Use your intuition – what do the weights of the first couple of components suggest where most of the variance in the data is coming from? It looks like the weights tend to be higher for the first two components. It is much lower for the latter few. This can also be seen in the fact that the first two comps have the highest variance ratio.

29) Create a plot of the cumulative sum of the explained variance. How many components will get you to 90% of the explained variance?

[35]:
```
# 3 components
fig = px.line(pca.explained_variance_ratio_.cumsum(), labels={'index':'number␣
↪of included components','value':'cumulative percent'}, title='Number of␣
↪components needed for 90% cumulative variance', render_mode='webgl')
fig.show()
```

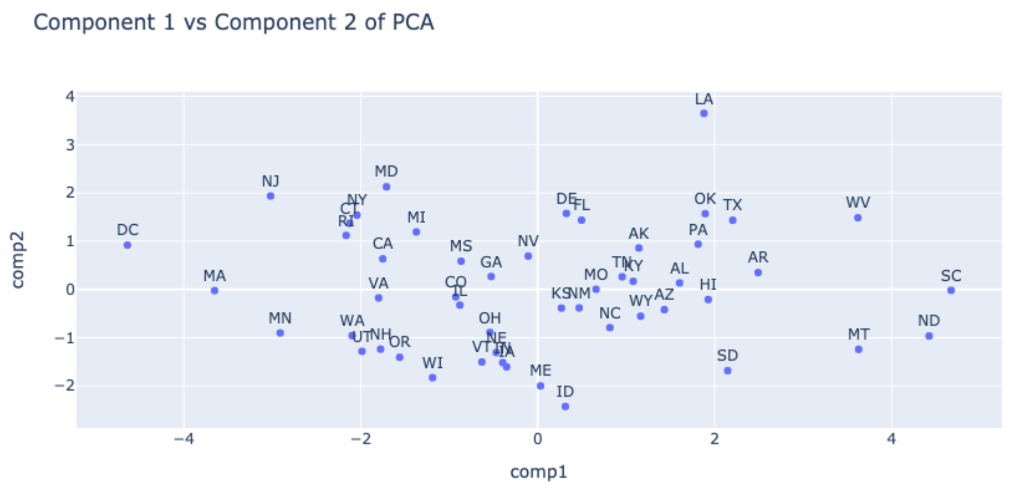


30) Transform the z_score transformed data using your PCA model (i.e. using the transform function of the pca object.) (NOTE: I often just store the transformed data temporarily as some arbitrary variable, X, to make it easier to manipulate the data for plotting.)

[36]:
```
X = pca.transform(df_car_crashes_zscore)
df_car_crashes_comps = df_car_crashes.copy()
df_car_crashes_comps['comp1'] = X[:,0]
df_car_crashes_comps['comp2'] = X[:,1]
df_car_crashes_comps['comp3'] = X[:,2]
```

31) Generate a 2D plot using the first two principal components as your x and y coordinates. Be sure to label each point, and label your axes as component 1 and component 2, respectively.

[37]:
```
fig = px.scatter(df_car_crashes_comps, x='comp1', y='comp2',␣
 ↪text=df_car_crashes.index, labels={'x':'component 1', 'y':'component 2'},␣
 ↪title="Component 1 vs Component 2 of PCA")
fig.update_traces(textposition='top center')

fig.show()
```



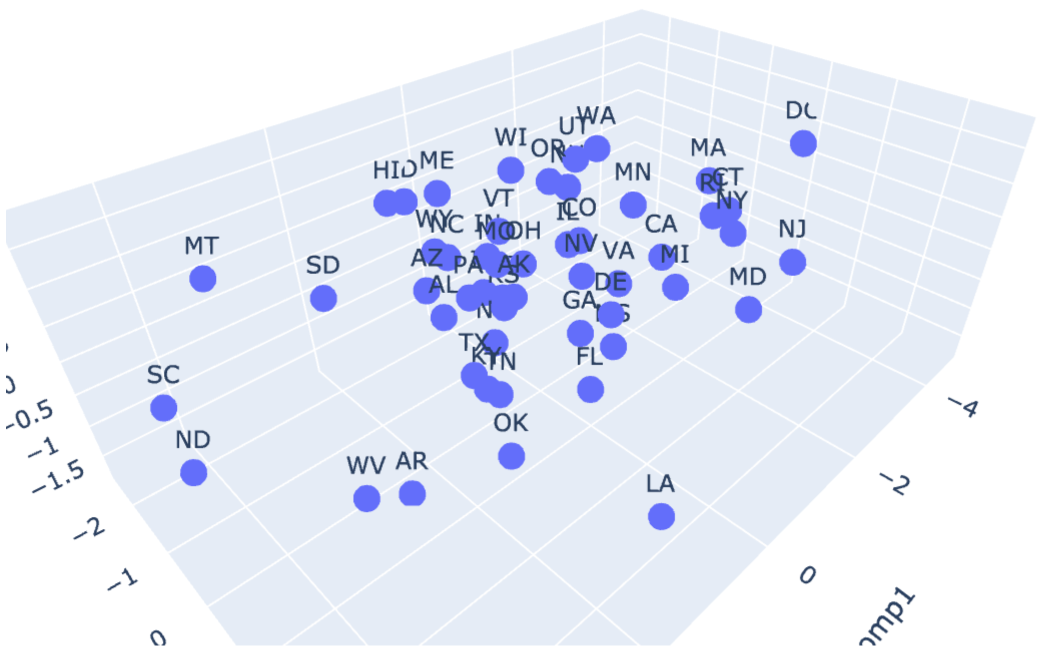
32) Compare the states you reported as potential outliers above to those that appear to be outliers from your plot. Do the same results seem to hold? The results still hold, we can tell that LA, DC, SC, and ND are still on the furthest edges of the plot.

33) Read how to generate a 3D scatterplot in seaborn or plotly, and use it to generate a scatterplot of the first 3 components.

[38]:
```
fig = px.scatter_3d(df_car_crashes_comps, x='comp1', y='comp2', z='comp3',␣
 ↪text=df_car_crashes.index, title="Component 1 vs Component 2 vs Component 3␣
 ↪of PCA")
fig.update_layout(scene = dict(xaxis_title='comp1', yaxis_title='comp2',␣
 ↪zaxis_title='comp3'))
fig.show()
```

34) Do the same outliers still stand out? Yes, they four outliers are still the furthest away from the center cluster of points. But it also looks like MT and AR are also quite far away and could be seen as outliers too.

Component 1 vs Component 2 vs Component 3 of PCA



34) Do the same outliers still stand out?

Yes, they four outliers are still the furthest away from the center cluster of points.
But it also looks like MT and AR are also quite far away and could be seen as outliers too.