Pricipal Coordinate Analysis (PCA)

At times, when you're working with complex data, you have so many variables that you're not sure where to start...It's in these cases, when you have many variables to consider that I often turn to PCA.

In these situations of variable-overload, I often struggle to understand the relationships between each variable. Am I overfitting a model -- its hard to tell with so many variables? I'm also often concerned that I may be violating assumptions of a model, especially that featurse are independent.

PCA helps to reduce the dimension of your feature space. By reducing the dimension of your feature space, you have fewer relationships between variables to consider and you are less likely to overfit your model. (Note: This doesn't immediately mean that overfitting, etc. are no longer concerns!)

Reducing the dimension of the feature space is called more officially "dimensionality reduction." There are many ways to achieve dimensionality reduction, but most of these techniques fall into one of two classes:

- Feature Elimination
- Feature Extraction

Feature elimination is what it sounds like: we reduce the feature space by eliminating features. Instead of considering all 100 features, we'll only use 10. Advantages of feature elimination methods include simplicity and maintaining interpretability of your variables. As a disadvantage, though, you gain no information from those variables you've dropped (and they may be important!).

Feature extraction, however, doesn't run into this problem. Say we have ten independent variables. In feature extraction, we create ten "new" independent variables, where each "new" independent variable is a combination of each of the ten "old" independent variables. However, we create these new independent variables in a specific way and order these new variables by how well they predict our dependent variable. In the Statquest video, these were the fitted eigenvectors he discussed.

Principal component analysis is a technique for feature extraction — so it combines our input variables in a specific way, then we can drop the "least important" variables while still retaining the most valuable parts of all of the variables! As an added benefit, each of the "new" variables after PCA are all independent of one another. This is a benefit because the assumptions of a linear model require our independent variables to be independent of one another. If we decide to fit a linear regression model with these "new" variables, this assumption will necessarily be satisfied.

When should PCA be used?

- Do you want to reduce the number of variables, but aren't able to identify variables to completely remove from consideration?

- Do you want to ensure your variables are independent of one another?
- Are you comfortable making your independent variables less interpretable?

If you answered “yes” to all three questions, then PCA is a good method to use. If you answered “no” to question 3, you should not use PCA.

Content based on https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c (https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c)

Dataset for PCA

We are going to be working with the digital images of tumor cells from our previous SVM tutorial. You'll remember that we have tumor images to predict whether the tumors are malignant or benign.

For each image, ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter) b) texture (standard deviation of gray-scale values) c) perimeter d) area e) smoothness (local variation in radius lengths) f) compactness (perimeter^2 / area - 1.0) g) concavity (severity of concave portions of the contour) h) concave points (number of concave portions of the contour) i) symmetry j) fractal dimension ("coastline approximation" - 1)

Additionally, the mean, standard error and “worst” or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features.

Let's start with bringing int he dataset and taking a quick look at it....

In [1]:
```
# Sometimes plots dont show up and this makes it better

%matplotlib inline
```

In [2]:
```
import pandas as pd
dataset = pd.read_csv('cancer.csv')
dataset.head()
```

Out[2]:

	ID	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_
0	842302	M	17.99	10.38	122.80	1001.0	0.
1	842517	M	20.57	17.77	132.90	1326.0	0.
2	84300903	M	19.69	21.25	130.00	1203.0	0.
3	84348301	M	11.42	20.38	77.58	386.1	0.
4	84358402	M	20.29	14.34	135.10	1297.0	0.

5 rows × 32 columns

In [3]:
```
print("Shape of our tumor cell dataset:", dataset.shape)
```

```
Shape of our tumor cell dataset: (569, 32)
```

In [4]:
```
print(dataset.columns)
```

```
Index(['ID', 'diagnosis', 'radius_mean', 'texture_mean', 'perimeter_
mean',
       'area_mean', 'smoothness_mean', 'compactness_mean', 'conc_mea
n',
       'conc_points_mean', 'symmetry_mean', 'fractral_mean', 'radius
_se',
       'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
       'compactness_se', 'conc_se', 'conc_points_se', 'symmetry_se',
       'fractral_se', 'radius_worst', 'texture_worst', 'perimeter_wo
rst',
       'area_worst', 'smoothness_worst', 'compactness_worst', 'conc_
worst',
       'conc_points_worst', 'symmetry_worst', 'fractral_worst'],
      dtype='object')
```

Let's select the mean, errors, and worst columns as separate dataframes. We've done this several different ways, usign iloc, using specific column names. This method is probably one I use a lot.

In [5]:
```
feature_mean = list(dataset.columns[2:12])
feature_error = list(dataset.columns[12:22])
feature_worst = list(dataset.columns[22:32])
```

In [6]:
```
print(feature_mean, feature_error, feature_worst)
```

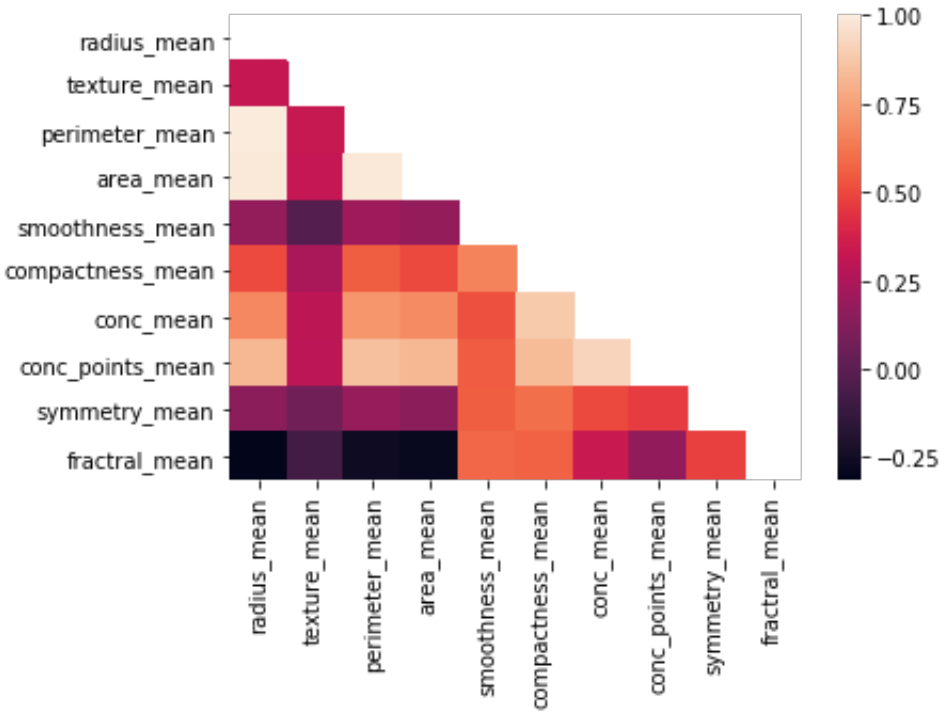
```
['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean', 'smoo
thness_mean', 'compactness_mean', 'conc_mean', 'conc_points_mean', '
symmetry_mean', 'fractral_mean'] ['radius_se', 'texture_se', 'perime
ter_se', 'area_se', 'smoothness_se', 'compactness_se', 'conc_se', 'c
onc_points_se', 'symmetry_se', 'fractral_se'] ['radius_worst', 'text
ure_worst', 'perimeter_worst', 'area_worst', 'smoothness_worst', 'co
mpactness_worst', 'conc_worst', 'conc_points_worst', 'symmetry_worst
', 'fractral_worst']
```

Make a correlation plot of the average values.

In [7]:
```
dataset_mean = dataset[list(dataset.columns[2:12])]
corr = dataset_mean.corr(method='pearson')
```

In [8]:
```
import seaborn as sns
import numpy as np
import matplotlib

# This removes the top half redundancy on heatmp
mask = np.zeros_like(corr)
mask[np.triu_indices_from(mask)] = True

# Plots the heatmap

sns.heatmap(corr, mask=mask)
```

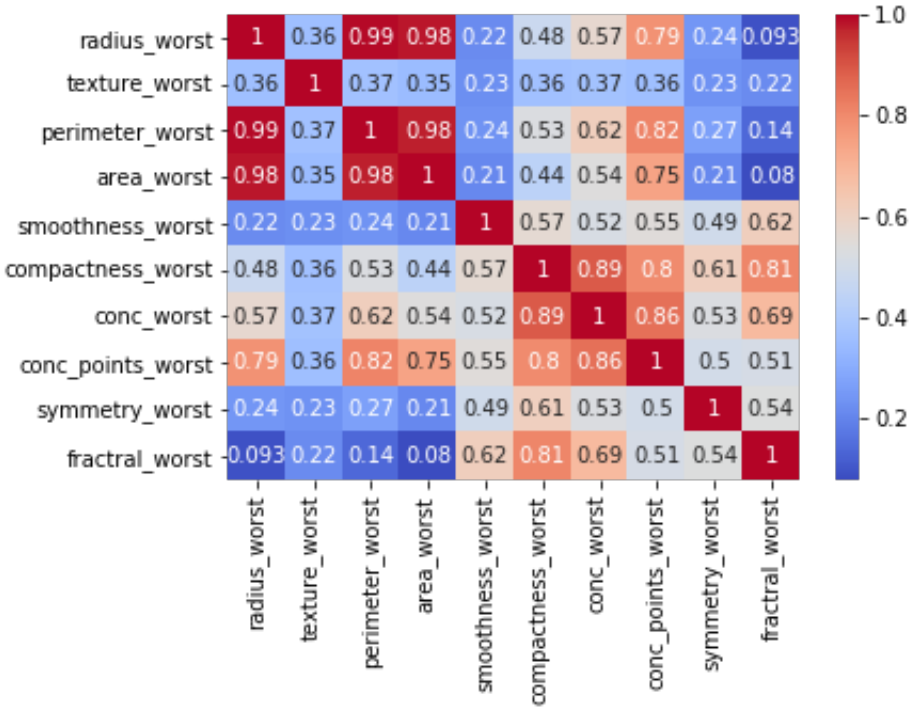
```
Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1e8ae128>
```



On your Own: Make at least one other plot to explore the correlations within this dataset. Make a correlation plot of the worst values.

In [9]:
```
dataset_worst = dataset[list(dataset.columns[22:32])]
corr2 = dataset_worst.corr(method='pearson')

sns.heatmap(corr2, annot = True, cmap = 'coolwarm')
```

Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1ecc1c88>



In [10]:
```
# Let's define our X and Y datasets for machine learning in scikit
X_data = dataset.iloc[:, 2:32]
Y_data = dataset.iloc[:,1]
```

Scaling data is important for PCAs

Feature scaling through standardization (or Z-score normalization) can be an important preprocessing step for many machine learning algorithms. Standardization involves rescaling the features such that they have the properties of a standard normal distribution with a mean of zero and a standard deviation of one.

While many algorithms (such as SVM, K-nearest neighbors, and logistic regression) require features to be normalized. Principle Component Analysis (PCA) is a prime example of when normalization is also important.

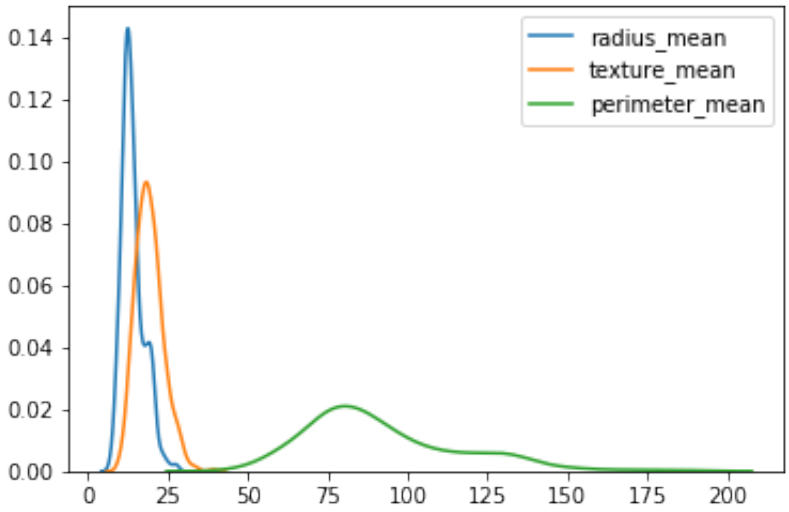
In PCA we are interested in the components that maximize the variance. If one component (e.g. human height) varies less than another (e.g. weight) because of their respective scales (meters vs. kilos), PCA might determine that the direction of maximal variance more closely corresponds with the 'weight' axis, if those features are not scaled. As a change in height of one meter can be considered much more important than the change in weight of one kilogram, this is clearly incorrect.

Source Documentation (https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html)

In [11]:
```
from sklearn.preprocessing import StandardScaler

# Documentation - https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

scaled_data = StandardScaler()
scaled_X = scaled_data.fit_transform(X_data)
```

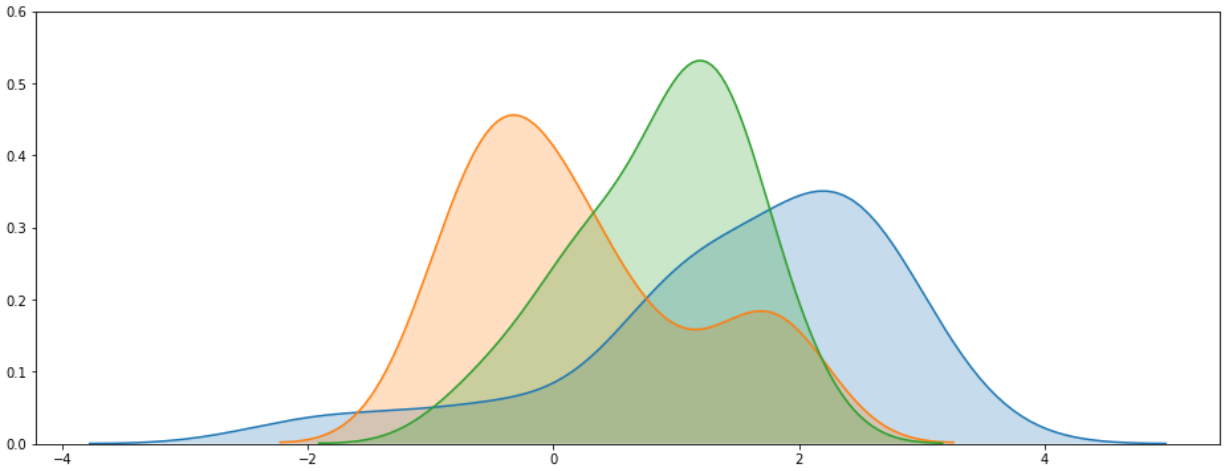
In [12]:
```
sns.kdeplot(X_data.iloc[:,0])
sns.kdeplot(X_data.iloc[:,1])
sns.kdeplot(X_data.iloc[:,2])
```

Out[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1e89f470>
```



In [13]:
```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(16, 6))
#Do you know what this does? Create a figure and set figure size

sns.kdeplot(scaled_X[0], shade = True)
sns.kdeplot(scaled_X[1], shade = True)
sns.kdeplot(scaled_X[2], shade = True)
plt.ylim(0, 0.6)
```

Out[13]:
```
(0, 0.6)
```



In the code above, what do the arguments 'stratify' and 'random_state' specify and when might you use them?

The argument 'stratify' is to specify whether the data is array-like or none(default), data will be splited in a stratified fashion using this as the labels array if not none. For 'random_state' argument, if it's set to be 1, the code will give same result every time; if it's set to be 2, it will happen as well but probably with different values than before; if it's set to be none(default), the code will choose random seed and get different result every time.

In [14]:
```
from sklearn.decomposition import PCA

pca1 = PCA(n_components=4)
# fit PCA on training set
pca1.fit(scaled_X)

train_pca1 = pca1.transform(scaled_X)
```

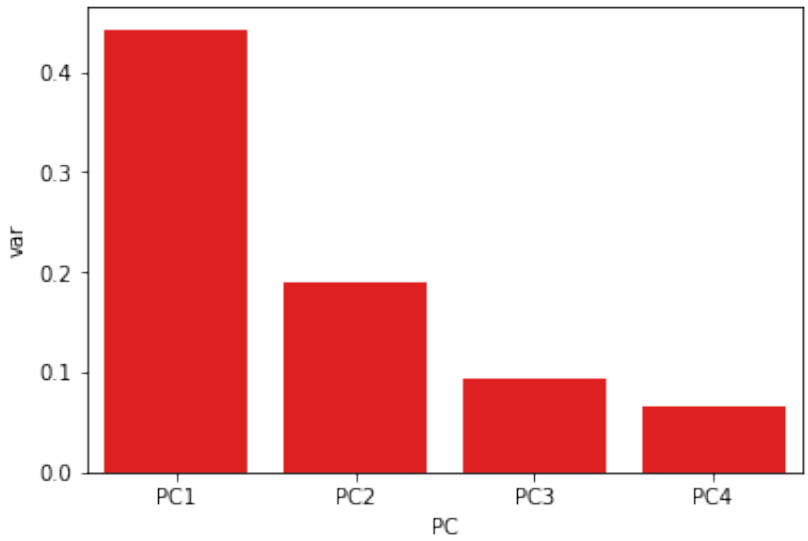
In [15]:
```
pc_df = pd.DataFrame(data = train_pca1, columns = ['PC1', 'PC2','PC3',
'PC4'])
pc_df['Cluster'] = Y_data
pc_df.head
```

Out[15]:
```
<bound method NDFrame.head of            PC1        PC2       PC3
PC4 Cluster
0     9.192837   1.948583 -1.123166  3.633731       M
1     2.387802  -3.768172 -0.529293  1.118264       M
2     5.733896  -1.075174 -0.551748  0.912083       M
3     7.122953  10.275589 -3.232790  0.152547       M
4     3.935302  -1.948072  1.389767  2.940639       M
5     2.380247   3.949929 -2.934877  0.941037       M
6     2.238883  -2.690031 -1.639913  0.149340       M
7     2.143299   2.340244 -0.871947 -0.127043       M
8     3.174924   3.391813 -3.119986 -0.601297       M
9     6.351747   7.727174 -4.341916 -3.375202       M
10   -0.810414  -2.659275 -0.488830 -1.672567       M
11    2.651100   0.066568 -1.526455  0.051262       M
12    8.185034   2.700976  5.730231 -1.112257       M
13    0.342126  -0.968279  1.717172 -0.595003       M
14    4.342379   4.861083 -2.816116 -1.454557       M
15    4.075656   2.977061 -3.125274 -2.458071       M
16    0.230055  -1.564758 -0.802519 -0.650583       M
17    4.418011   1.418670 -2.270319 -0.186272       M
18    4.948704  -4.114334 -0.314749 -0.088207       M
19   -1.237063  -0.188215 -0.593283  1.596346       B
20   -1.578161   0.572808 -1.801447  1.125276       B
21   -3.557336   1.662950  0.451187  2.073765       B
22    4.733211   3.304964 -1.466537  2.041150       M
```

```
23    4.208524  -5.128367 -0.752402 -0.862710       M
24    4.949632  -1.543752 -1.713194  0.046759       M
25    7.098563   2.018610 -0.029010  2.587951       M
26    3.510263   2.171625 -3.894546 -1.295760       M
27    3.064054  -1.876552  2.581748  0.128484       M
28    4.007264   0.537242 -2.761626 -1.898387       M
29    1.715310  -1.523705  0.146187  1.911386       M
..         ...        ...       ...       ...     ...
539  -1.142832   5.599458  1.301037 -2.188250       B
540  -1.665475   2.389618  1.502249  0.875951       B
541   1.011712   1.092390 -0.632698 -1.758518       B
542  -1.300930  -1.821415  0.373307 -1.848169       B
543  -2.373429  -1.681576  0.384528 -3.016729       B
544  -1.665871  -0.213963 -0.148072 -0.197052       B
545  -1.927678  -1.137740  0.478202 -1.157500       B
546  -4.237217   0.184272 -0.326418  0.588303       B
547  -2.677871   2.315793 -0.053848  0.340450       B
548  -3.836498   0.496250  0.923240 -0.551872       B
549  -2.551440   0.228330  1.414178 -1.970790       B
550  -4.694923  -0.767478  1.543965 -0.779019       B
551  -2.025037   1.261242  0.504926 -1.135527       B
552  -2.895948  -1.451636  0.780546 -2.970448       B
553  -3.502201   1.800832  2.766457 -0.866307       B
554  -2.153904  -0.830069  0.564797 -3.011756       B
555  -2.055084   1.616459  1.838959 -3.113535       B
556  -3.877290   1.084255  1.859944 -0.433740       B
557  -4.063862   0.122168  3.238773 -3.469183       B
558  -0.098667  -0.213560  0.388929 -1.012711       B
559  -1.089376   1.292848  1.429379 -3.372136       B
560  -0.481771  -0.178020  1.032108 -2.010280       B
561  -4.870310  -2.131106  3.414189 -5.133988       B
562   5.917613   3.482637 -3.262792 -3.917585       M
563   8.741338  -0.573855  0.897090  0.385150       M
564   6.439315  -3.576817  2.459487  1.177314       M
565   3.793382  -3.584048  2.088476 -2.506028       M
566   1.256179  -1.902297  0.562731 -2.089227       M
567  10.374794   1.672010 -1.877029 -2.356031       M
568  -5.475243  -0.670637  1.490443 -2.299157       B

[569 rows x 5 columns]>
```

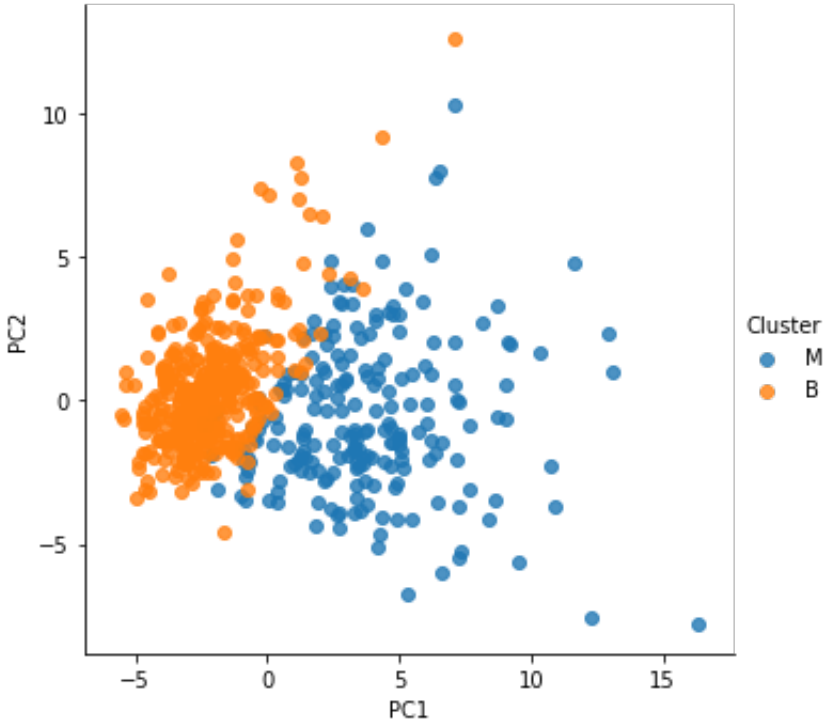
Let's take a look at our trained dataset and how much was explained by each principle coordinate.

In [16]:
```
df = pd.DataFrame({'var':pca1.explained_variance_ratio_, 'PC':['PC1','PC2','PC3','PC4']})
sns.barplot(x='PC',y="var", data=df, color="red");
```



In [17]:
```
p = sns.lmplot( x="PC1", y="PC2", data=pc_df, fit_reg=False, hue='Cluster', legend=True)
# specify the point size
p
```

Out[17]: <seaborn.axisgrid.FacetGrid at 0x1a20d3b240>



Some other tutorials (that are potentially useful):

1. PCA followed by regression: https://nirpyresearch.com/principal-component-regression-python/ (https://nirpyresearch.com/principal-component-regression-python/)
2. Manually doing a PCA, more math theory: https://sebastianraschka.com/Articles/2014_pca_step_by_step.html (https://sebastianraschka.com/Articles/2014_pca_step_by_step.html)
3. Generic PCA with a different dataset: https://medium.com/district-data-labs/principal-component-analysis-with-python-4962cd026465 (https://medium.com/district-data-labs/principal-component-analysis-with-python-4962cd026465)

Another Dataset: Wine

The dataset contain the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines.

Details can be **found here** (http://archive.ics.uci.edu/ml/datasets/Wine).

In [18]:
```
df = pd.read_csv('wine.data.csv', header=None)
df.columns = ['Class', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash', 'Magnesium', 'Total phenols', 'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity', 'Hue', 'OD280/OD315 of diluted wines', 'Proline']
df.head(10)
```

Out[18]:

	Class	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proa
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	
5	1	14.20	1.76	2.45	15.2	112	3.27	3.39	0.34	
6	1	14.39	1.87	2.45	14.6	96	2.50	2.52	0.30	
7	1	14.06	2.15	2.61	17.6	121	2.60	2.51	0.31	
8	1	14.83	1.64	2.17	14.0	97	2.80	2.98	0.29	
9	1	13.86	1.35	2.27	16.0	98	2.98	3.15	0.22	

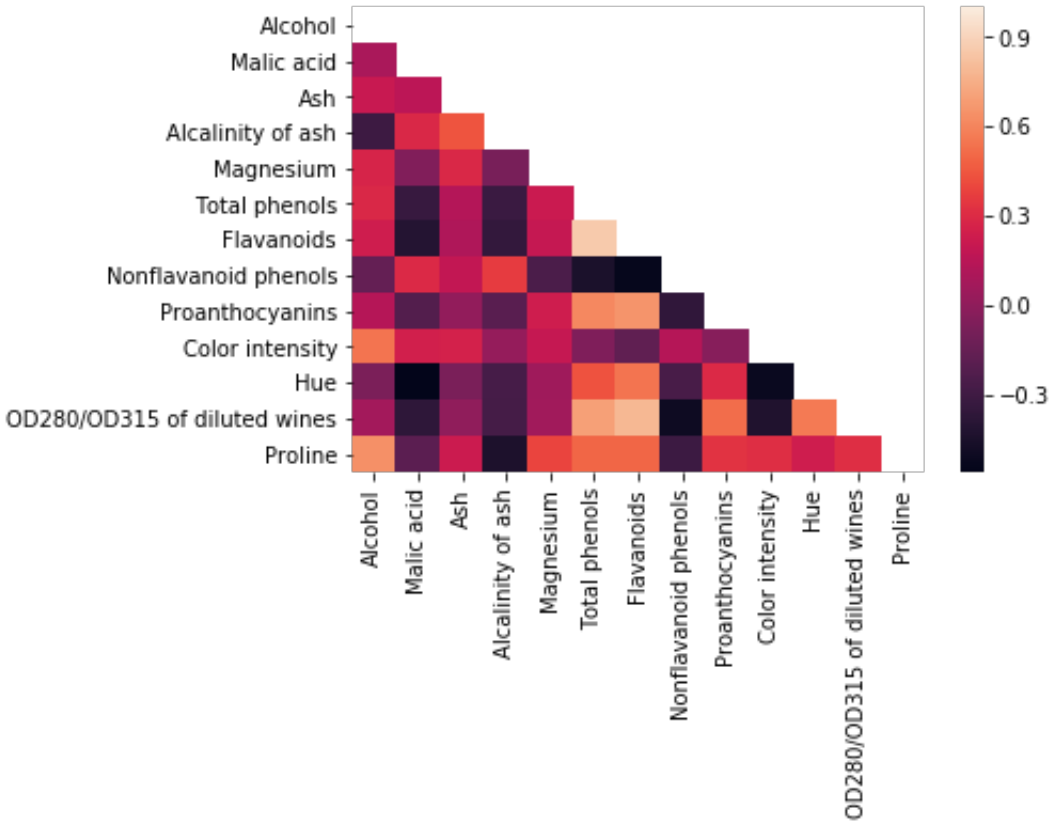
In [19]:
```
print("Shape of the wine dataset:", df.shape)
```

```
Shape of the wine dataset: (178, 14)
```

In [20]:
```
dfc = df[list(df.columns[1:14])]
corrw = dfc.corr(method='pearson')
```

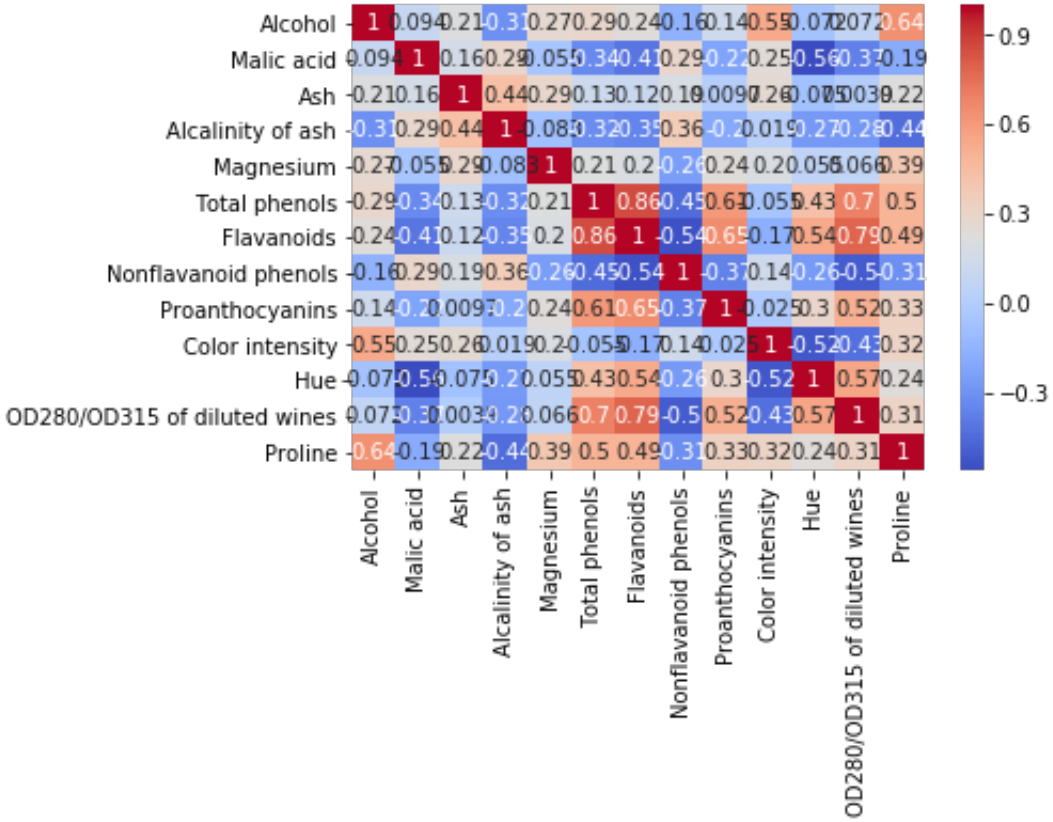
In [21]:
```
# removes the top half redundancy on heatmp
maskw = np.zeros_like(corrw)
maskw[np.triu_indices_from(maskw)] = True

# plot the heatmap

sns.heatmap(corrw, mask=maskw)
```

Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x1a20d3ba20>



In [22]:
```
sns.heatmap(corrw, annot = True, cmap = 'coolwarm')
```

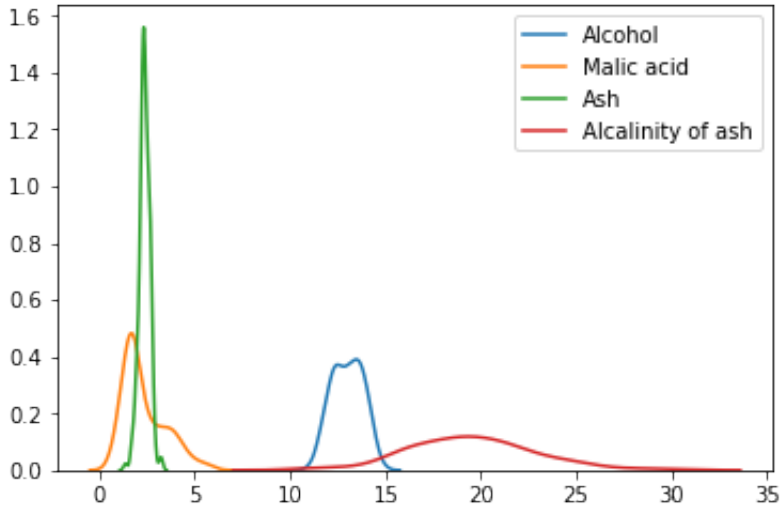
Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x11c374710>



In [23]:
```
# define our X and Y datasets for machine learning in scikit
X_dataw = df.iloc[:, 1:14]
Y_dataw = df.iloc[:,0]

scaled_Xw = scaled_data.fit_transform(X_dataw)

sns.kdeplot(X_dataw.iloc[:,0])
sns.kdeplot(X_dataw.iloc[:,1])
sns.kdeplot(X_dataw.iloc[:,2])
sns.kdeplot(X_dataw.iloc[:,3])
```

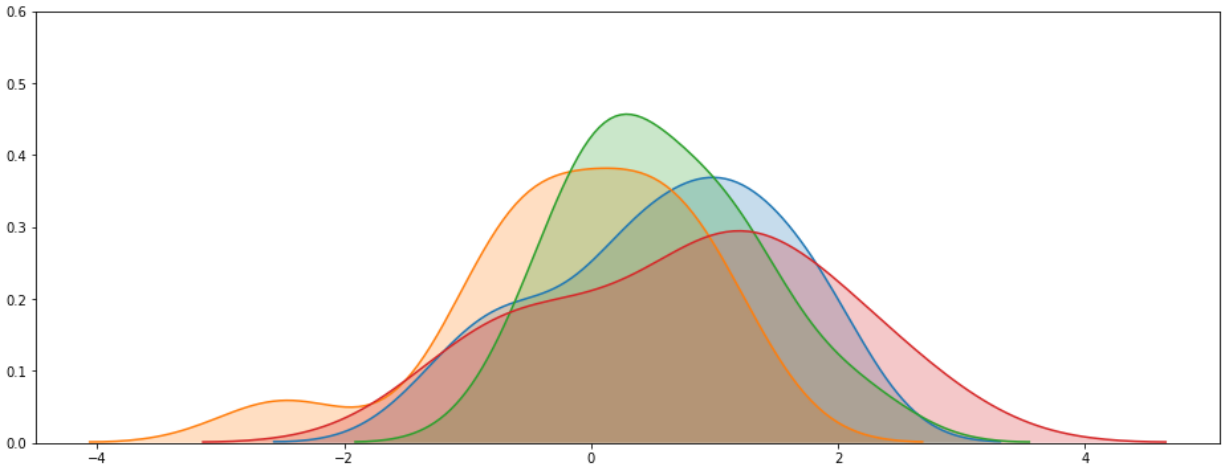
```
/Users/joylee/anaconda3/lib/python3.7/site-packages/sklearn/preproce
ssing/data.py:645: DataConversionWarning: Data with input dtype int6
4, float64 were all converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
/Users/joylee/anaconda3/lib/python3.7/site-packages/sklearn/base.py:
464: DataConversionWarning: Data with input dtype int64, float64 wer
e all converted to float64 by StandardScaler.
  return self.fit(X, **fit_params).transform(X)
```

Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x1a2115a6a0>



In [24]:
```
# Create a figure and set size
plt.figure(figsize=(16, 6))

sns.kdeplot(scaled_Xw[0], shade = True)
sns.kdeplot(scaled_Xw[1], shade = True)
sns.kdeplot(scaled_Xw[2], shade = True)
sns.kdeplot(scaled_Xw[3], shade = True)
plt.ylim(0, 0.6)
```

Out[24]:
```
(0, 0.6)
```



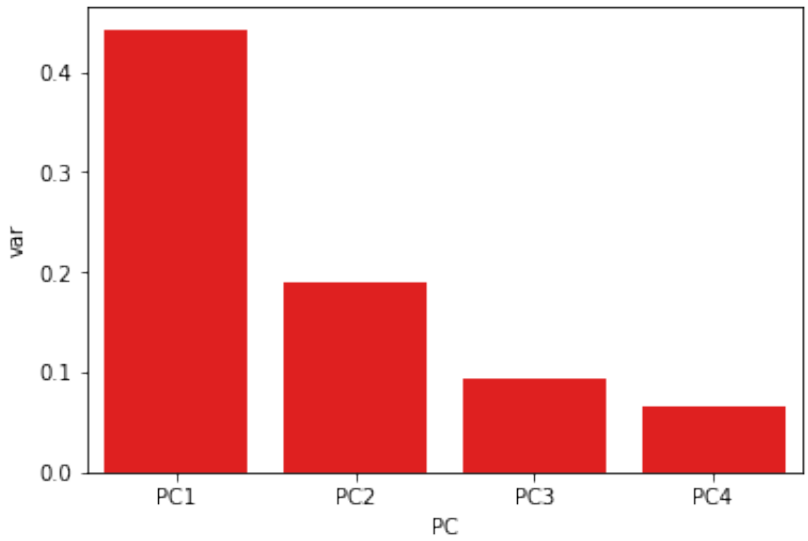
In [25]:
```
pca2 = PCA(n_components=4)
# fit PCA on training set
pca2.fit(scaled_Xw)

train_pca2 = pca2.transform(scaled_Xw)

pc_dfw = pd.DataFrame(data = train_pca2, columns = ['PC1', 'PC2','PC3'
,'PC4'])
pc_dfw['Cluster'] = Y_dataw
pc_dfw.head
```

Out[25]:
```
<bound method NDFrame.head of           PC1       PC2       PC3
PC4  Cluster
0    3.316751 -1.443463 -0.165739 -0.215631        1
1    2.209465  0.333393 -2.026457 -0.291358        1
2    2.516740 -1.031151  0.982819  0.724902        1
3    3.757066 -2.756372 -0.176192  0.567983        1
4    1.008908 -0.869831  2.026688 -0.409766        1
5    3.050254 -2.122401 -0.629396 -0.515637        1
6    2.449090 -1.174850 -0.977095 -0.065831        1
7    2.059437 -1.608963  0.146282 -1.192608        1
8    2.510874 -0.918071 -1.770969  0.056270        1
9    2.753628 -0.789438 -0.984247  0.349382        1
10   3.479737 -1.302333 -0.422735  0.026842        1
```

11	1.754753	-0.611977	-1.190878	-0.890164	1
12	2.113462	-0.675706	-0.865086	-0.356438	1
13	3.458157	-1.130630	-1.204276	0.162458	1
14	4.312784	-2.095976	-1.263913	0.305773	1
15	2.305188	-1.662552	0.217903	-1.440590	1
16	2.171955	-2.327305	0.831730	-0.912601	1
17	1.898971	-1.631369	0.794914	-1.082380	1
18	3.541985	-2.518344	-0.485459	-0.910323	1
19	2.084522	-1.061138	-0.164747	0.484997	1
20	3.124403	-0.786897	-0.364887	-0.025562	1
21	1.086570	-0.241744	0.936962	1.029910	1
22	2.535224	0.091841	-0.311933	-0.048391	1
23	1.644988	0.516279	0.143885	-0.413720	1
24	1.761576	0.317149	0.890286	-0.115116	1
25	0.990079	-0.940667	3.820908	-1.321561	1
26	1.775278	-0.686175	-0.086700	-0.232907	1
27	1.235424	0.089807	-1.386897	-0.495683	1
28	2.188406	-0.689570	1.394567	-0.777492	1
29	2.256109	-0.191462	-1.092657	0.286152	1
..	...	...	...	...	...
148	-2.807064	-1.570534	-0.472528	0.627358	3
149	-2.899659	-2.041057	-0.495960	0.471156	3
150	-2.320737	-2.356366	0.437682	-0.052260	3
151	-2.549831	-2.045283	-0.312268	0.386972	3
152	-1.812541	-1.527646	1.362590	-0.189396	3
153	-2.760145	-2.138932	-0.964629	0.668386	3
154	-2.737151	-0.409886	-1.190405	-0.663045	3
155	-3.604869	-1.802384	-0.094037	1.268840	3
156	-2.889826	-1.925219	-0.782323	1.324725	3
157	-3.392156	-1.311876	1.602026	-0.482842	3
158	-1.048182	-3.515090	1.160039	0.935329	3
159	-1.609912	-2.406638	0.548560	0.754310	3
160	-3.143131	-0.738161	-0.090999	0.980648	3
161	-2.240157	-1.175465	-0.101377	-1.165279	3
162	-2.847674	-0.556044	0.804215	-0.897888	3
163	-2.597497	-0.697966	-0.884940	-0.274229	3
164	-2.949299	-1.555309	-0.983401	0.015480	3
165	-3.530032	-0.882527	-0.466029	0.580790	3
166	-2.406111	-2.592356	0.428226	-0.184335	3
167	-2.929085	-1.274447	-1.213358	0.295316	3
168	-2.181413	-2.077537	0.763783	-0.389593	3
169	-2.380928	-2.588667	1.418044	0.588502	3
170	-3.211617	0.251249	-0.847129	-0.217065	3
171	-3.677919	-0.847748	-1.339420	-0.125176	3
172	-2.465556	-2.193798	-0.918781	0.018025	3
173	-3.370524	-2.216289	-0.342570	1.058527	3
174	-2.601956	-1.757229	0.207581	0.349496	3
175	-2.677839	-2.760899	-0.940942	0.312035	3
176	-2.387017	-2.297347	-0.550696	-0.688285	3
177	-3.208758	-2.768920	1.013914	0.596903	3

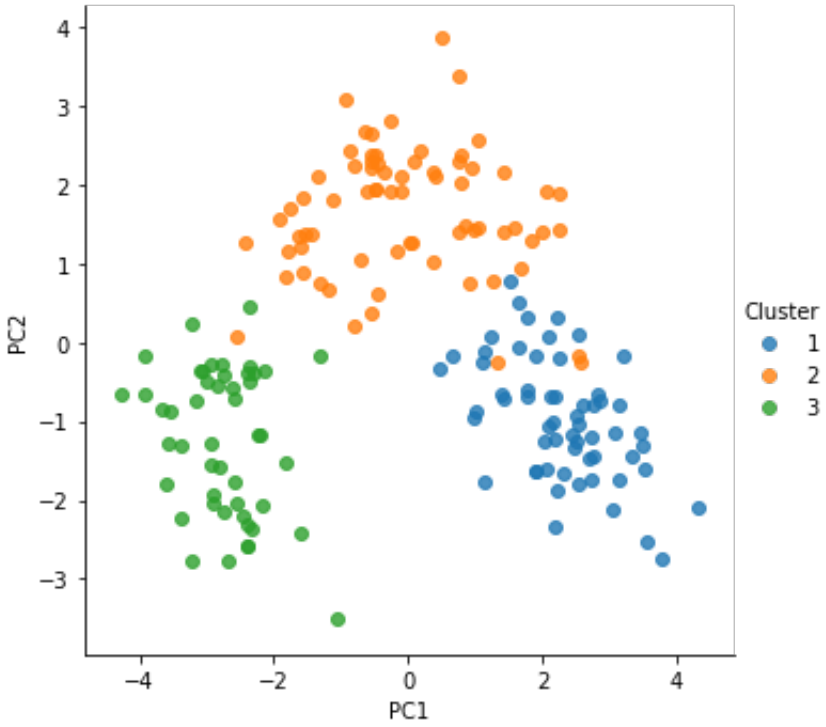
```
[178 rows x 5 columns]>
```

In [26]:
```
dfw = pd.DataFrame({'var':pca1.explained_variance_ratio_, 'PC':['PC1',
'PC2','PC3','PC4']})
sns.barplot(x='PC',y="var", data=dfw, color="red");
```



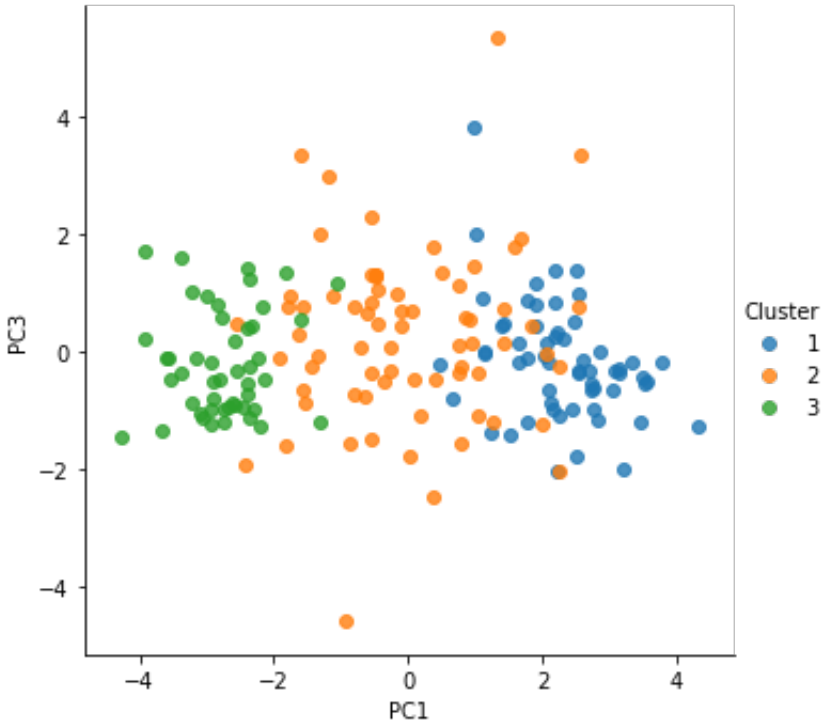
In [27]:
```
p1 = sns.lmplot( x="PC1", y="PC2", data=pc_dfw, fit_reg=False, hue='Cluster', legend=True)
p1
```

Out[27]: <seaborn.axisgrid.FacetGrid at 0x1a21326cf8>



In [28]:
```
p2 = sns.lmplot( x="PC1", y="PC3", data=pc_dfw, fit_reg=False, hue='Cluster', legend=True)
p2
```

Out[28]: <seaborn.axisgrid.FacetGrid at 0x1a2156c9b0>



The PCA model has been trained on the wine dataset with four principle coordinates. Based on the scatter plots of principle coordinates, PC1 and PC2 gives excellent result in terms of seperating three wine classes. To get more optimal classification, we can use three features for the model since this is a relatively small dataset. Original code and data are posted on github. (https://github.com/joyleeisu/ABE516X-PCA.git (https://github.com/joyleeisu/ABE516X-PCA.git))