



CS-109A Introduction to Data Science

Lab 8: Principle Component Analysis (PCA)

Harvard University

Fall 2019

Instructors: Pavlos Protopapas, Kevin Rader, Chris Tanner

Lab Instructors: Chris Tanner and Eleni Kaxiras.

Contributors: Will Claybaugh, David Sondak, Chris Tanner

```
In [1]: ## RUN THIS CELL TO PROPERLY HIGHLIGHT THE EXERCISES
import requests
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-
CS109A/master/content/styles/cs109.css").text
HTML(styles)
```

Out[1]:

Learning Goals

In this lab, we will look at how to use PCA to reduce a dataset to a smaller number of dimensions. The goal is for students to:

- Understand what PCA is and why it's useful
- Feel comfortable performing PCA on a new dataset
- Understand what it means for each component to capture variance from the original dataset
- Be able to extract the `variance explained` by components.
- Perform modelling with the PCA components

```
In [17]: %matplotlib inline
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.linear_model import LogisticRegressionCV
from sklearn.linear_model import LassoCV
from sklearn.metrics import accuracy_score
pd.set_option('display.width', 500)
pd.set_option('display.max_columns', 100)
pd.set_option('display.notebook_repr_html', True)

from sklearn.model_selection import train_test_split
```


Part 1: Introduction

What is PCA? PCA is a deterministic technique to transform data to a lowered dimensionality, whereby each feature/dimension captures the most variance possible.

Why do we care to use it?


- Visualizing the components can be useful
- Allows for more efficient use of resources (time, memory)
- Statistical reasons: fewer dimensions -> better generalization
- noise removal / collinearity (improving data quality)

Imagine some dataset where we have two features that are pretty redundant. For example, maybe we have data concerning elite runners. Two of the features may include **VO2 max** and **heartrate**. These are highly correlated. We probably don't need both, as they don't offer much additional information from each other. Using a [great visual example from online](#), let's say that this unlabelled graph (**always label your axis**) represents those two features:

 VO2 Max vs Heart Rate

Let's say that this is our entire dataset, just 2 dimensions. If we wish to reduce the dimensions, we can only reduce it to just 1 dimension. A straight line is just 1 dimension (to help clarify this: imagine your straight line as being the x-axis, and values can be somewhere on this axis, but that's it. There is no y-axis dimension for a straight line). So, how should PCA select a straight line through this data?

Below, the image shows all possible projects that are centered in the data:

 Animation of possible lines

PCA picks the line that:

- captures the most variance possible
- minimizes the distance of the transformed points (distance from the original to the new space)

The animation **suggests** that these two aspects are actually the same. In fact, this is objectively true, but the proof for which is beyond the scope of the material for now. Feel free to read more at [this explanation](#) and via [Andrew Ng's notes](#).

In short, PCA is a math technique that works with the covariance matrix -- the matrix that describes how all pairwise features are correlated with one another. Covariance of two variables measures the degree to which they moved/vary in the same direction; how much one variable affects the other. A positive covariance means they are positively related (i.e., x_1 increases as x_2 does); negative means negative correlation (x_1 decreases as x_2 increases).

In data science and machine learning, our models are often just finding patterns in the data this is easier if the data is spread out across each dimension and for the data features to be independent from one another (imagine if there's no variance at all. We couldn't do anything). Can we transform the data into a new set that is a linear combination of those original features?

PCA finds new dimensions (set of basis vectors) such that all the dimensions are orthogonal and hence linearly independent, and ranked according to the variance (eigenvalue). That is, the first component is the most important, as it captures the most variance.

Part 2: The Wine Dataset

Imagine that a wine sommelier has tasted and rated 1,000 distinct wines, and now that she's highly experienced, she is curious if she can more efficiently rate wines without even trying them. That is, perhaps her tasting preferences follow a pattern, allowing her to predict the rating a new wine without even trying it!

The dataset contains 11 chemical features, along with a quality scale from 1-10; however, only values of 3-9 are actually used in the data. The ever-elusive perfect wine has yet to be tasted.

NOTE: While this dataset involves the topic of alcohol, we, the CS109A staff, along with Harvard at large is in no way encouraging alcohol use, and this example should not be interpreted as any endorsement for such; it is merely a pedagogical example. I apologize if this example offends anyone or is off-putting.

Read-in and checking

First, let's read-in the data and verify it:

```
In [18]: wines_df = pd.read_csv("../data/wines.csv", index_col=0)
wines_df.head()
```

Out[18]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	8.9	0.590	0.50	2.0	0.337	27.0	81.0	0.99640	3.04	1.61	9.5	6
1	7.7	0.690	0.22	1.9	0.084	18.0	94.0	0.99610	3.31	0.48	9.5	5
2	8.8	0.685	0.26	1.6	0.088	16.0	23.0	0.99694	3.32	0.47	9.4	5
3	11.4	0.460	0.50	2.7	0.122	4.0	17.0	1.00060	3.13	0.70	10.2	5
4	8.8	0.240	0.54	2.5	0.083	25.0	57.0	0.99830	3.39	0.54	9.2	5

```
In [19]: wines_df.describe()
```

Out[19]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	quality
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1
mean	7.558400	0.397455	0.30676	4.489250	0.067218	25.29650	91.03100	0
std	1.559455	0.189923	0.16783	4.112419	0.046931	17.06237	59.57269	0
min	3.800000	0.080000	0.00000	0.800000	0.009000	1.00000	6.00000	0
25%	6.500000	0.260000	0.22000	1.800000	0.042000	12.00000	37.75000	0
50%	7.200000	0.340000	0.30000	2.400000	0.060000	22.00000	86.00000	0
75%	8.200000	0.520000	0.40000	6.100000	0.080000	35.00000	135.00000	0
max	15.500000	1.580000	1.00000	26.050000	0.611000	131.00000	313.00000	1

For this exercise, let's say that the wine expert is curious if she can predict, as a rough approximation, the **categorical quality** -- **bad, average, or great**. Let's define those categories as follows:

- **bad** is when for wines that have a quality <= 5
- **average** is when a wine has a quality of 6 or 7
- **great** is when a wine has a quality of >= 8

```
In [20]: # copy the original data so that we're free to make changes
wines_df_recode = wines_df.copy()

# use the 'cut' function to reduce a variable down to the aforementioned bins (inclusive boundaries)
wines_df_recode['quality'] = pd.cut(wines_df_recode['quality'], [0,5,7,10], labels=[0,1,2])
wines_df_recode.loc[wines_df_recode['quality'] == 1]

# drop the un-needed columns
x_data = wines_df_recode.drop(['quality'], axis=1)
y_data = wines_df_recode['quality']

x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=.2, random_state=8,
stratify=y_data)

# previews our data to check if we correctly constructed the labels (we did)
print(wines_df['quality'].head())
print(wines_df_recode['quality'].head())
```

```
0    6
1    5
2    5
3    5
4    5
Name: quality, dtype: int64
0    1
1    0
2    0
3    0
4    0
Name: quality, dtype: category
Categories (3, int64): [0 < 1 < 2]
```

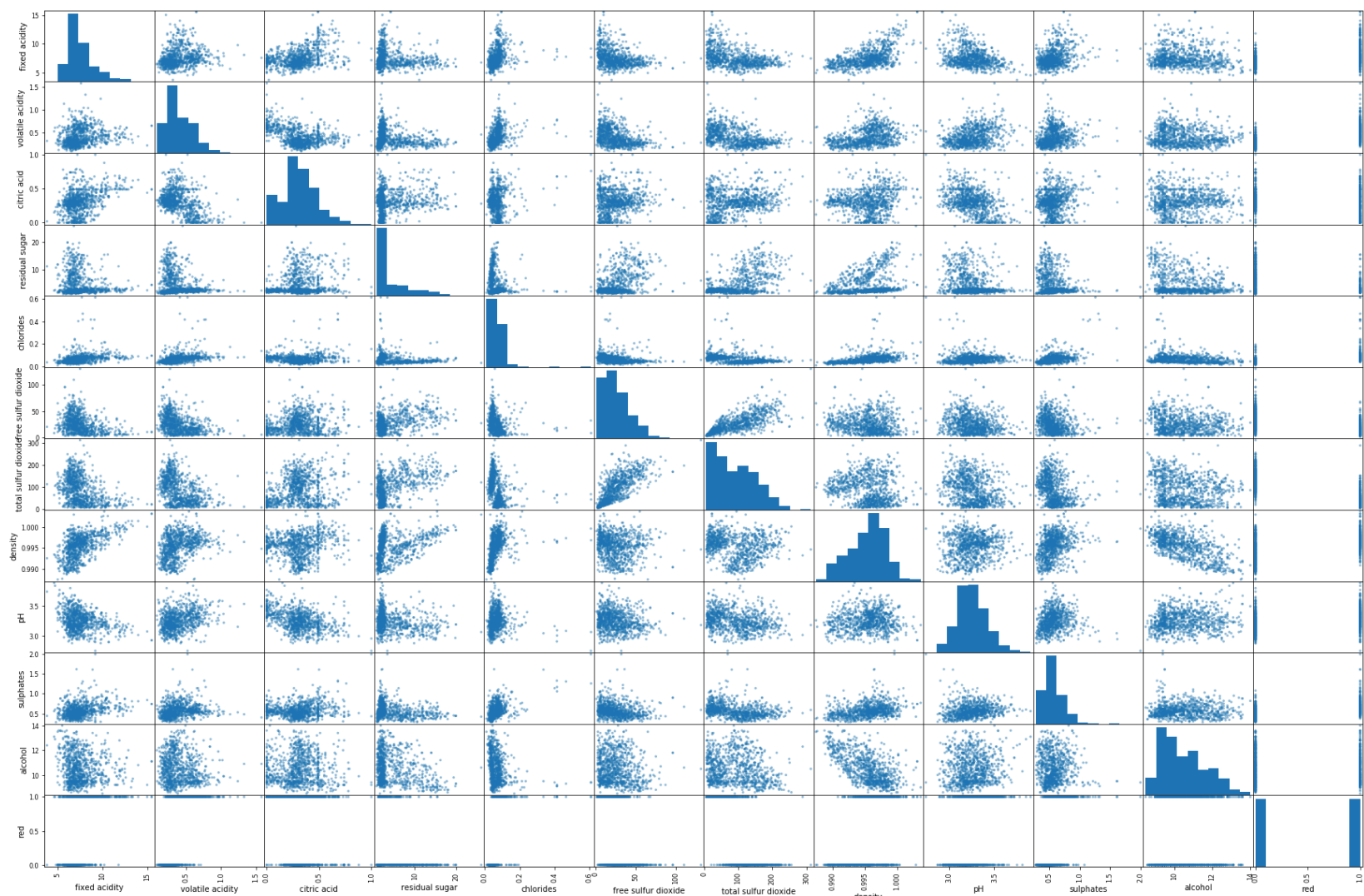
For sanity, let's see how many wines are in each category:

```
In [72]: y_data.value_counts()
```

```
Out[72]: 1    598
0    379
2     23
Name: quality, dtype: int64
```

Now that we've split the data, let's look to see if there are any obvious patterns (correlations between different variables).

```
In [22]: from pandas.plotting import scatter_matrix
scatter_matrix(wines_df_recode, figsize=(30,20));
```



It looks like there aren't any particularly strong correlations among the predictors (maybe **total sulfur dioxide** and **free sulfur dioxide**) so we're safe to keep them all.

Part 3: Prediction Models

Before we do anything too fancy, it's always best to start with something simple.

MLE Baseline

Maximum-likelihood estimate, as we discussed in Lab 6, is barely a model -- it simply returns the single label/class that maximizes the likelihood that the training data was observed. In other words, whichever label/class is most popular, it will always emit that as its answer, completely independent of any x-data. Above, we saw that the most popular label was **average**, represented by 598 of our 1,000 wines. Thus, our MLE should yield **average** for all inputs:

```
In [123]: mle_class = y_data.value_counts().idxmax()
mle_train_accuracy = len(y_train.loc[y_train == mle_class]) / len(y_train)
mle_test_accuracy = len(y_test.loc[y_test == mle_class]) / len(y_test)

# add it to a new dataframe of results
#model_results = {'MLE': [mle_train_accuracy, mle_test_accuracy]}

scores = [[mle_train_accuracy, mle_test_accuracy]]
names = ['MLE']
df_results = pd.DataFrame(scores, index=names, columns=['Train Accuracy', 'Test Accuracy'])
df_results
```

Out[123]:

	Train Accuracy	Test Accuracy
MLE	0.5975	0.6

MLE gives a predictive accuracy is **60.0%** on the test set. Hopefully we can do better than this.

Logistic Regression

Logistic regression is used for predicting categorical outputs, which is exactly what our task concerns. So, let's create a logistic regression model:

```
In [124]: from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1000000, solver='lbfgs', multi_class='ovr', max_iter=10000).fit(x_train,y_train)
print("Coefficients:", lr.coef_)
print("Intercepts:", lr.intercept_)

Coefficients: [[-9.06195681e-02  4.01354365e+00  8.29365457e-01 -7.59637666e-02
  7.90968174e-01 -7.24204216e-03  8.51373247e-03  5.69750326e+00
 -3.54671277e-01 -1.45265293e+00 -1.12361810e+00  3.66833904e-01]
 [ 9.89355891e-02 -3.62383390e+00 -7.82265871e-01  8.59208982e-02
  3.49438371e-01 -4.02021619e-04 -6.99279818e-03 -4.29100502e+00
  2.45590096e-01  7.49614440e-01  8.77241084e-01 -2.32631070e-01]
 [ 2.64583794e-01 -2.10729846e+00  5.78746442e-01 -1.40827248e-01
 -4.08817845e+01  2.67479043e-02 -2.64446974e-03 -1.24747277e+01
  3.38514567e+00  2.95824310e+00  7.01062478e-01 -1.85965151e-01]]
Intercepts: [  5.71308742 -4.35882732 -11.2307248 ]
```

Exercise: What is stored in `lr.coef_` and `lr.intercept_`? Why are there so many of them?

Answer was discussed in lab. In short, it's because there are N models, where N is our number of class labels (due to running logistic regression in a one-vs-rest manner).

Let's measure its performance:

```
In [125]: lr_train_accuracy = lr.score(x_train, y_train)
lr_test_accuracy = lr.score(x_test, y_test)

# appends results to our dataframe
names.append('Logistic Regression')
scores.append([lr_train_accuracy, lr_test_accuracy])
df_results = pd.DataFrame(scores, index=names, columns=['Train Accuracy', 'Test Accuracy'])
df_results
```

Out[125]:

	Train Accuracy	Test Accuracy
MLE	0.59750	0.60
Logistic Regression	0.73875	0.75

Yay, that's better than our MLE's performance. Can we do better with cross-validation?

Summary

- Logistic regression extends OLS to work naturally with a dependent variable that's only ever 0 and 1.
- In fact, Logistic regression is even more general and is used for predicting the probability of an example belonging to each of N classes.
- The code for the two cases is identical and just like `LinearRegression`: `.fit`, `.score`, and all the rest
- Significant predictors does not imply that the model actually works well. Significance can be driven by data size alone.
- The data aren't enough to do what we want

Warning: Logistic regression *tries* to hand back valid probabilities. As with all models, you can't trust the results until you validate them- if you're going to use raw probabilities instead of just predicted class, take the time to verify that if you pool all cases where the model says "I'm 30% confident it's class A" 30% of them actually are class A.

```
In [126]: logit_regr_lasso = LogisticRegressionCV(solver='liblinear', multi_class='ovr', penalty='l1',
max_iter=100000, cv=10)
logit_regr_lasso.fit(x_train,y_train) # fit

y_train_pred_lasso = logit_regr_lasso.predict(x_train) # predict the test set
y_test_pred_lasso = logit_regr_lasso.predict(x_test) # predict the test set

train_score_lasso = accuracy_score(y_train, y_train_pred_lasso) # get train accuracy
test_score_lasso = accuracy_score(y_test, y_test_pred_lasso) # get test accuracy

names.append('Logistic Regression w/ CV + Lasso')
scores.append([train_score_lasso, test_score_lasso])
df_results = pd.DataFrame(scores, index=names, columns=['Train Accuracy', 'Test Accuracy'])
df_results
```

Out[126]:

	Train Accuracy	Test Accuracy
MLE	0.59750	0.60
Logistic Regression	0.73875	0.75
Logistic Regression w/ CV + Lasso	0.74000	0.74

Exercise: Hmm, cross-validation didn't seem to offer improved results. Is this correct? Is it possible for cross-validation to not yield better results than non-cross-validation? If so, how and why?

```
In [39]: # answer was discussed in Lab
```

Part 4: Dimensionality Reduction

In attempt to improve performance, we may wonder if some of our features are redundant and are posing difficulties for our logistic regression model. Let's PCA to shrink the problem down to 2 dimensions (with as little loss as possible) and see if that gives us a clue about what makes this problem tough.

```
In [40]: from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

num_components = 2

# scale the datasets
scale_transformer = StandardScaler(copy=True).fit(x_train)
x_train_scaled = scale_transformer.transform(x_train)
x_test_scaled = scale_transformer.transform(x_test)

# reduce dimensions
pca_transformer = PCA(num_components).fit(x_train_scaled)
x_train_2d = pca_transformer.transform(x_train_scaled)
x_test_2d = pca_transformer.transform(x_test_scaled)

print(x_train_2d.shape)
x_train_2d[0:5,:]
```

(800, 2)

```
Out[40]: array([[ -1.23949587, -2.34876616],
[ -1.13615842,  0.37632328],
[  1.84037474, -0.10597198],
[  2.00455608, -0.14333293],
[  1.68553308,  4.82627254]])
```

NOTE:

- 1. Both scaling and reducing dimension follow the same pattern: we fit the object to the training data, then use .transform() to convert the training and test data. This ensures that, for instance, we scale the test data using the *training* mean and variance, not its own mean and variance

2. We need to equalize the variance of each feature before applying PCA; otherwise, certain dimensions will dominate the scaling. Our PCA dimensions would just be the features with the largest spread.

Exercise: Why didn't we scale the y-values (class labels) or transform them with PCA? Is this a mistake?

```
In [ ]: # answer was discussed in Lab
```

Exercise: Our data only has 2 dimensions/features now. What do these features represent?

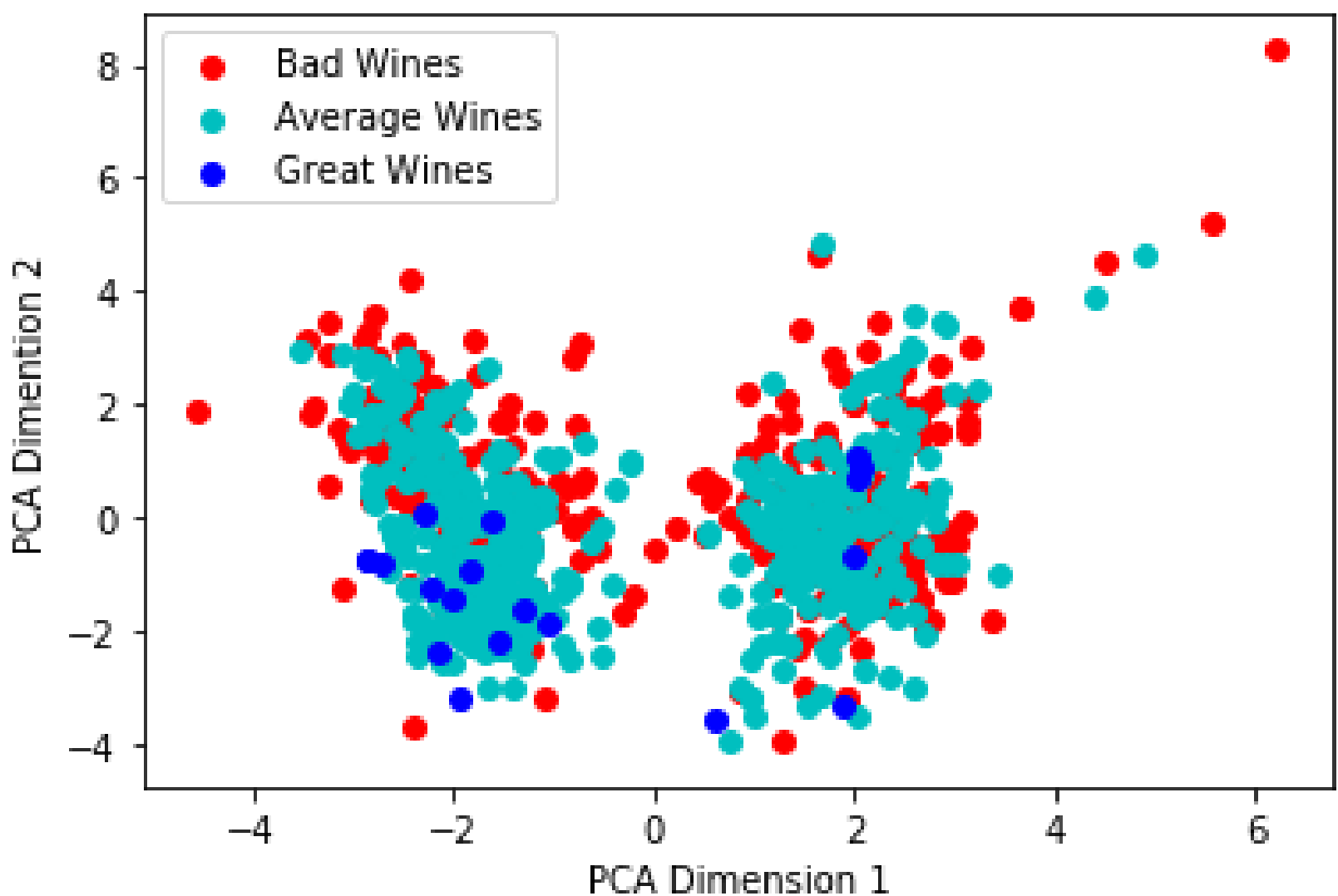
```
In [ ]: # answer heavily discussed in Lab
```

Since our data only has 2 dimensions now, we can easily visualize the entire dataset. If we choose to color each datum with respect to its associated label/class, it allows us to see how separable the data is. That is, it gives an indication as to how easy/difficult it is for a model to fit the new, transformed data.

```
In [12]: # notice that we set up lists to track each group's plotting color and label
colors = ['r','c','b']
label_text = ["Bad Wines", "Average Wines", "Great Wines"]

# and we loop over the different groups
for cur_quality in [0,1,2]:
    cur_df = x_train_2d[y_train==cur_quality]
    plt.scatter(cur_df[:,0], cur_df[:,1], c = colors[cur_quality], label=label_text[cur_quality])

# all plots need labels
plt.xlabel("PCA Dimension 1")
plt.ylabel("PCA Dimention 2")
plt.legend();
```



Well, that gives us some idea of why the problem is difficult! The bad, average, and great wines are all on top of one another. Not only are there few great wines, which we knew from the beginning, but there is no line that can easily separate the classes of wines.

Exercise:

1. What critique can you make against the plot above? Why does this plot not prove that the different wines are hopelessly similar?
2. The wine data we've used so far consist entirely of continuous predictors. Would PCA work with categorical data?

```
In [ ]: # answer was discussed in Lab
```

Looking at our PCA plot above, we see something peculiar: we have two disjoint clusters, both of which have immense overlap in the qualities of wines.

Exercise: What could cause this? What does this mean?

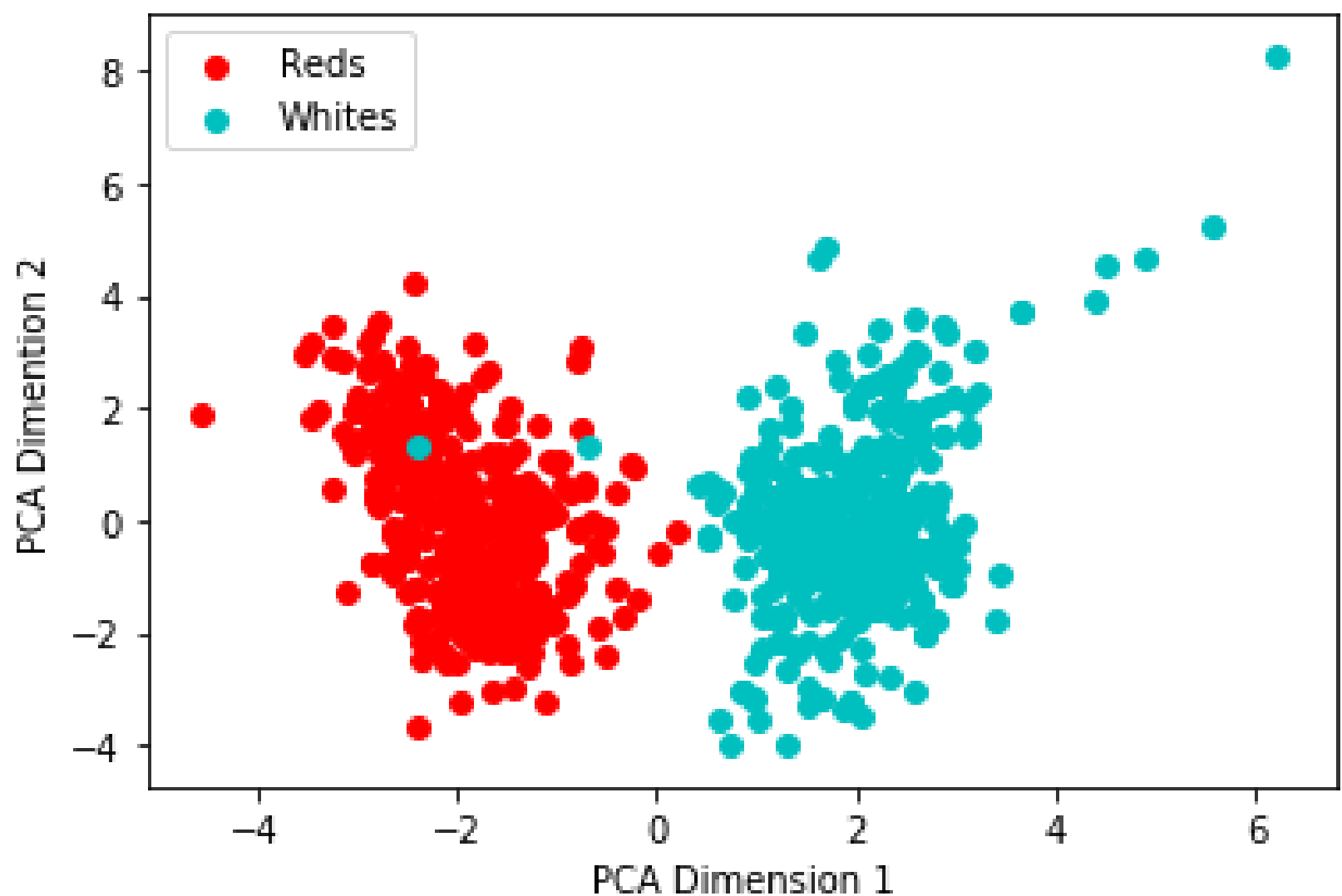
```
In [ ]: # answer was discussed in Lab
```

Let's plot the same PCA'd data, but let's color code them according to if the wine is red or white

```
In [13]: # notice that we set up lists to track each group's plotting color and label
colors = ['r','c','b']
label_text = ["Reds", "Whites"]

# and we loop over the different groups
for cur_color in [0,1]:
    cur_df = x_train_2d[x_train['red']==cur_color]
    plt.scatter(cur_df[:,0], cur_df[:,1], c = colors[cur_color], label=label_text[cur_color])

# all plots need labels
plt.xlabel("PCA Dimension 1")
plt.ylabel("PCA Dimention 2")
plt.legend();
```



Exercise: Wow. Look at that separation. Too bad we aren't trying to predict if a wine is red or white. Does this graph help you answer our previous question? Does it change your thoughts?

```
In [ ]: # answer was discussed in Lab
```

5. Evaluating PCA: Variance Explained and Predictions

One of the criticisms we made of the PCA plot was that it's lost something from the original data. Heck, we're only using 2 dimensions, we it's perfectly reasonable and expected for us to lose some information – our goal was that the information we were discarding was noise.

Let's investigate how much of the original data's structure the 2-D PCA captures. We'll look at the `explained_variance_ratio_` portion of the PCA fit. This lists, in order, the percentage of the x data's total variance that is captured by the nth PCA dimension.

```
In [41]: var_explained = pca_transformer.explained_variance_ratio_
print("Variance explained by each PCA component:", var_explained)
print("Total Variance Explained:", np.sum(var_explained))

Variance explained by each PCA component: [0.34021651 0.20128374]
Total Variance Explained: 0.5415002511091703
```

The first PCA dimension captures 33% of the variance in the data, and the second PCA dimension adds another 20%. Together, we've captured about half of the total variation in the training data with just these two dimensions. So far, we've used PCA to transform our data, we've visualized our newly transformed data, and we've looked at the variance that it captures from the original dataset. That's a good amount of inspection; now let's actually use our transformed data to make predictions.

Exercise: Use Logistic Regression (with and without cross-validation) on the PCA-transformed data. Do you expect this to outperform our original 75% accuracy? What are your results? Does this seem reasonable?

```
In [127]: # partial solution provided (not showing CV portion)
lr = LogisticRegression(C=1000000, solver='lbfgs', multi_class='ovr',
max_iter=10000).fit(x_train_2d,y_train)

lr_pca_train_accuracy = lr.score(x_train_2d, y_train)
lr_pca_test_accuracy = lr.score(x_test_2d, y_test)

names.append('Logistic Regression w/ PCA')
scores.append([lr_pca_train_accuracy, lr_pca_test_accuracy])
df_results = pd.DataFrame(scores, index=names, columns=['Train Accuracy', 'Test Accuracy'])
df_results
```

Out[127]:

	Train Accuracy	Test Accuracy
MLE	0.59750	0.60
Logistic Regression	0.73875	0.75
Logistic Regression w/ CV + Lasso	0.74000	0.74
Logistic Regression w/ PCA	0.61250	0.56

We're only using 2 dimensions. What if we increase our data to 10 PCA components?

- Exercise:**
1. Fit a PCA that finds the first 10 PCA components of our training data
 2. Use `np.cumsum()` to print out the variance we'd be able to explain by using n PCA dimensions for n=1 through 10
 3. Does the 10-dimension PCA agree with the 2d PCA on how much variance the first components explain? ****Do the 10d and 2d PCAs find the same first two dimensions? Why or why not?****
 4. Make a plot of number of PCA dimensions against total variance explained. What PCA dimension looks good to you?

Hint: `np.cumsum` stands for 'cumulative sum', so `np.cumsum([1,3,2,-1,2])` is `[1,4,6,5,7]`

The plot above can be used to inform of us when we reach diminishing returns on variance explained. That is, the 'elbow' of the line is probably an ideal number of dimensions to use, at least with respect to the amount of variance explained.

Exercise: Looking at your graph, what is the 'elbow' point / how many PCA components do you think we should use? Does this number of components imply that predictive performance will be optimal at this point? Why or why not?

6: PCA Debriefing:

- PCA maps a high-dimensional space into a lower dimensional space.
- The PCA dimensions are ordered by how much of the original data's variance they capture
 - There are other cool and useful properties of the PCA dimensions (orthogonal, etc.). See a [textbook](#).
- PCA on a given dataset always gives the same dimensions in the same order.
- You can select the number of dimensions by fitting a big PCA and examining a plot of the cumulative variance explained.

PCA is not guaranteed to improve predictive performance at all. As you've learned in class now, none of our models are guaranteed to always outperform others on all datasets; analyses are a roll of the dice. The goal is to have a suite of tools to allow us to gather, process, dissect, model, and visualize the data – and to learn which tools are better suited to which conditions. Sometimes our data isn't the most conducive to certain tools, and that's okay.

What can we do about it?

1. Be honest about the methods and the null result. Lots of analyses fail.
2. Collect a dataset you think has a better chance of success. Maybe we collected the wrong chemical signals...
3. Keep trying new approaches. Just beware of overfitting the data you're validating on. Always have a test set locked away for when the final model is built.
4. Change the question. Maybe something you noticed during analysis seems interesting or useful (classifying red versus white). But again, you the more you try, the more you might overfit, so have test data locked away.
5. Just move on. If the odds of success start to seem small, maybe you need a new project.