Lab 6

Today we continue exploring **dimension reduction** to help us get a handle on large dimensional data. Today's goals are:

1. Understand how to use the `sklearn` implementation for PCA
2. Determine what the "right" lower dimension is
3. Introduce *Singular Value Decomposition*
4. Compare and contrast PCA and SVD

In [ ]:
```
# Import block
%matplotlib notebook

from sklearn.decomposition import PCA
```

For easier comparisons, we will continue with the ever exciting `students_info.csv` file. Please import this in `pandas` and then create a `numpy` array with only the numerical variables

In [ ]:
```
# Import data
students =
```

In [ ]:
```
# Create justnum with only the numerical data
justnum = students[["coffee", "sleep", "gym", "gpa"]].to_numpy()
```

Preparing the data

Recall that for PCA, we need to have normalized data. Before continuing, create `justnum_std` that *standardizes* each variable. (Hint: you might want to grab this from another lab)

In [ ]:
```
# Standardizing our variables:
```

In [ ]:
```
# Check your creation of justnum_std
```

In [ ]:
```
# You may want to create a few plots
```

In [ ]:
```
# Another block for exploration
```

PCA in `sklearn`

Like kmeans, PCA is implemented in `sklearn` and it works much the way that kmeans did: there's a set-up phase and then an application phase. We first set up how many components we are looking for with PCA. We then apply that particular PCA that we have crafted to our data.

As we did with kmeans, we will take each step individually, exploring the output that we generate in each stage.

In the below code block, we have one possible setting of `PCA()` in `sklearn`.

- What *type* is the output and what information is contained within `PCA` ?
- What are the various parameters doing?

In [ ]:
```
# Step one: Set up PCA by defining the number of components
pca_alg = PCA(n_components=2)
```

In [ ]:
```
# Code block for further discovery
type(????)


# Try displaying the result from above.
#       What can you read easily and what is hidden?
```

Using `dir` for further exploration

We can use `dir()` to actually peek inside what kinds information is contained within `pca_alg`. The output from `dir()` will just list the attributes and names within the listed input.

WARNING: `dir` works differently in the **terminal** (or command prompt or anaconda prompt) that you use to access your notebooks.

Let's see this in operation:

In [ ]:
```
dir(pca_alg)
```

As with kmeans, now that we have set up our PCA, we can *fit* it to our data. We can `fit` our data and then `transform` it; or if we prefer, we can do both using `fit_transform`. In the next section, we will explore these three functions:

- `fit()`
- `transform()`
- `fit_transform()`

Using `.fit()`

This first fitting applies our PCA to the data. What *type* is the output and what information is contained within pfit ?

In [ ]:
```
pfit = pca_alg.fit(justnum_std)
```

In [ ]:
```
# Code block for further discovery
type(????)


# Try displaying the result from above.
#       What can you read easily and what is hidden?
```

In [ ]:
```

```

In [ ]:
```

```

It's not immediately obvious how to access the principal components. The result of `.fit()` wraps this information inside a class style object. We can use `.components_` to access the principal components:

In [ ]:
```
print("Shape of the resulting components\n", pfit.components_.shape)

print("\n Actual components \n",pfit.components_)
```

Is this what we expect to see? Why or why not?

Take a minute to explain this to a neighbor. Then working with your fellow machine learning practitioner, try to condense your explanation into something you could post on social media (like a tweet or a meme).

PCA `.fit` result

The output of `fit()` is a *transition* matrix. This is the matrix that "carries" our data from a higher dimension down to the lower one. What `.fit` does **not** do is carry out this tranformation. For that step, we need to use `.transform()` .

Using `.transform()`

So far, we have set-up our PCA and applied it to our data to get our transition matrix. To actually send our data to the lower dimensional space, we *transform* our data using `transform()` .

Note - Both `.predict` for kmeans and `transform` for PCA in `sklearn` are similar in the sense of *extending* the common applications of their algorithms respectively. However, in the case of PCA, simply stopping at the *transition* matrix feels a bit odd as the *dimension reduction* has not yet occurred. To complete the dimension reduction, we use `transform()` .

Before moving on: Write in **your** own words:

1. What is `.fit()` ?
2. What is `.transform()` ?
3. How are they different?

In [ ]:
```
justnum_intwo = pca_alg.transform(justnum_std)
```

In [ ]:
```
# Check the new lower dimensional result. -
#     What is the type and shape of our output?
type(????)


# Try displaying the result from above.
#       What can you read easily and what is hidden?
```

In [ ]:
```
# Create a 2D plot of the new lower dimensional result
```

Is this what you expect given what you know about this data? Why or why not?

Using `fit_transform()`

We can fit our PCA to our data and do the dimension reduction in one step using `fit_transform()` :

In [ ]:
```
justnum_intwo_onestep = pca_alg.fit_transform(justnum_std)
```

In [ ]:
```
# Add a layer to compare the previous result to this one.
# Your goal with this plot is to confirm that you have the same result

# Create the plot from above with larger but fainter circles:
plt.scatter(justnum_intwo[????], justnum_intwo[????], s = 100, alpha = 0.5)

# Layer with the new variable represented with little x markers
plt.scatter(????, ????, marker = "x", c = 'k')
```

Aside - `transform` is just matrix multiplication

All that `.transform` is doing is *right* multiplying our data with the transition matrix. In fact, if we define `trans_mat` as follows, we can

In [ ]:
```
trans_mat = pfit.components_
justnum_intwo_mult = np.dot(justnum_std,trans_mat.T)
```

In [ ]:
```
# Check that the shape is what we expect
```

In [ ]:
```
# Add a layer to compare the previous result to this one.
# Your goal with this plot is to confirm that you have
#      the same result using multiplication
# Hint - It should look exactly like the above one, but
#        with a different variable in the last line
```

In [ ]:
```

```

Wrapping up PCA

So far, we have applied PCA to a dataset of two-dimensions. This seems a bit odd since most computers can easily visualize data in two-dimensions. So this kind of dimension reduction feels unnecessary. And you're totally right! It is unnecessary from a machine learning perspective. **But** we did this dimension reduction precisely because we can see each part. We can visualize the starting point in 2-D, see the result in 1-D, and compare the two.

Before moving on, apply PCA to `justnum_std` for 1, 3, and 4 components. That is, create a 1-D starting point (or a dataset with just one column) and do PCA. Then repeat this process for a 3-D starting point (or a dataset with three columns) and for a 4-D starting point.

Then create visualizations for 1 and 3 dimensions. Considering the two dimensional ones that we created, what is the right number of components? **Justify** your choice.

Below are a few code blocks for your experiments and plots.

In [ ]:
```
# Apply PCA to justnum_std for 1 component

# Set up PCA with 1 component
pca_alg1 = PCA(n_components=??)

# Fit PCA to the data
pfit = pca_alg.fit(???)

# Project the data onto the resulting components
justnum_inone = pca_alg.???(???)
```

In [ ]:
```
# Visualize result of PCA with 1 component applied justnum_std
plt.scatter(????,np.zeros(???))
```

In [ ]:
```
# Apply PCA to justnum_std for 3 components

# Set up PCA with 3 components
pca_alg1 = PCA(n_components=??)

# Fit PCA to the data and transform it in one step
justnum_inthree = pca_alg.???(???)
```

In [ ]:
```
# Visualize result of PCA with 3 components applied justnum_std
fig = plt.figure()
ax = plt.axes(projection='3d')

# Create the SCATTER() plot with colors
ax.scatter(???,???,???);
```

In [ ]:
```
# Apply PCA to justnum_std for 4 components
```

In [ ]:
```
# Why is there no visulization block for this one?
# your thoughts -



# Yes this is intentionally a code block
```

What do you notice about your data? What is easier to see in 1, 2, or 3 D projections?

Wait here for a group discussion

Choosing the *lower* dimension

The goal of PCA is a dimension reduction. The question becomes how far should we reduce? What is the "right" lower dimension? After our adventure with k-means, it should come as no surprise that there is no universally accepted "answer" for choosing the right number for the lower dimension. If your primary reason for reducing the dimension is for data visualization, then many times the "answer" is 2 for the *target* dimension.

If your goal is to reduce your data to speed up algorithms or to simply remove redundant information, then you might look at how much your reduction *explains* of the original data. Encapsulated in the `explained_variance_ratio_` is a notion of explanation.

Total Explained Variation

Most data is spread out. Data that is more spread from each other with varying notions of "near" and "far", we call data of *high variance* and by contrast data with *low variance* is data that tends to be very tightly clustered together. The idea of *variance* is to give us a notion of how spread (or not) our data is. It can also be thought of as a notion of the variation in between our data points.

PCA seeks to be a summary of our original data. We want it to be a decent summary that gives us close approximation of the data. To give us an idea of how well our PCA works, we can compute how much of the original data's variation is *explained* by each principal component. In `sklearn` this information is captured in `explained_variance_ratio_` or the ratio of the **total** variance that is explained by a particular component.

The `explained_variance_ratio_` is an attribute of our fitted PCA. Compute PCA with four components and plot the explained_variance_ratio

In [ ]:

```
pca_alg = PCA(n_components=???)
pfit4 = pca_alg.fit(justnum_std)
```

In [ ]:

```
plt.plot(pfit4.explained_variance_ratio_, marker='*')
```

When the number of components matches the dimension of the data, the `sum` of the `explained_variance_ratio_` should equal one. Check that this is true:

In [ ]:

```
# Your check here:
```

Note - If yours is not exactly one, check that the difference between your answer and 1 is below *machine tolerance* or about 10^{-14}:

In [ ]:

```
mach_tol = 10e-14
print(np.abs(np.sum(pfit4.explained_variance_ratio_)-1) < mach_tol)
```

Take a minute to unpack what is in the second check. What does each piece do?

- `np.sum(pfit4.explained_variance_ratio_)` is doing _
- `np.abs(???-1)` is doing _
- `???? < mach_tol` is doing _

Wait here for a group discussion

Using `explained_variance_ratio_` to choose the "right" dimension

In `explained_variance_ratio_` we have how much each principal component explains. As we can see above, the added benefit of each one goes down. This is related to the fact that the computation of `explained_variance_ratio_` has to do with the eigenvalues associated to each principal component (which by construction are in decreasing order).

It would be much more helpful if we could see how much the cumulative contribution of the first component, the second component with the first, the third component with the first and the second, and so on. Now, while we could write a `for` loop, let's take a page

```
In [ ]: cum_variance = np.cumsum(pfit4.explained_variance_ratio_)
```

The function `cumsum()` takes cumulative sums in progressing order. To illustrate what this means run the following code block:

```
In [ ]: test_cumsum = np.cumsum([1,2,3,4,5])
        print(test_cumsum)
```

Notice that that the first entry in the output is the value of the first index. The second entry is the sum of the the values associated to the first two indices, and so on.

Returning to our cumulative variance, let's now plot this output against the individual contributions:

```
In [ ]: plt.step(range(1,5),cum_variance)
        plt.bar(range(1,5),pfit4.explained_variance_ratio_)

        # Plot based on image on Page 148 in _Python Machine Learning_
```

What we should see in this plot is how much each piece contributes. While this is not the most exciting example with our original data's dimension being only 4, we can see concretely how much each component explains and how much they consecutive components together.

Now the question still looms: how do we decide what the appropriate target dimension (ie. the lower dimension) should be? Again, if not desiring to make a two-dimensional visualization of our data, we would examine a plot similar to the above one and use a process similar to elbowology to determine the "right" target dimension.

What do you think the "right" target dimension is for this data? **Justify** your choice.

Singular Value Decomposition (SVD)

SVD is linear algebra at its finest (though I promise not to explore that tangent). There are several parallels to PCA (which we explore later, but in this section, we will focus on 1) what SVD is and 2) how to compute it.

SVD is based on the fact that for any matrix D, we can find a *decomposition* comprised of three matrices -- U, S, and V -- such that $D = USV^t$. In other words, we can find three matrices -- U, S, and V -- where D is equal to the matrix multiplications of U, S, and V transposed. (Recall that all that *transpose* means is "flip" the rows to be columns and the columns to be rows.)

For our data matrix D, suppose we have m observations and n variables. Then the matrices in this decomposition have a particular structure:

- U is an $m \times m$ matrix where the **columns** are each perpendicular to each other (like axes). These are called the *left singular vectors*.
- S is an $m \times n$ matrix where nearly all the entries are 0. Only the diagonal entries (ie. those in the i, i locations are non-zero. Like the eigenvalues in PCA, we the entries of S decrease as we step down the diagonal. These are called the *singular values*.
- V is an $n \times n$ matrix where the **rows** are each perpendicular to each other (like axes). These are called the *right singular vectors*.

The goal of SVD is to find the matrices U, S, and V, such that $D = USV^t$.

SVD in python

SVD is *just* a decomposition of a matrix. It's all linear algebra, and as such, it is implemented directly in *numpy* in the `linalg` sub-module:

In [ ]:
```
U,S,Vt = np.linalg.svd(justnum,full_matrices=True)
```

Warning

There is an inherent disconnect between the `numpy` implementation and the theory underlying SVD. The SVD decomposition is typically written: $D = U \cdot S \cdot V^t$, where V is a collection of **columns** and where ^{t} is the transpose of V.

In the `numpy` implementation, what is actually outputted is V^t. So this means that Vt is a collection of **rows.** This is an important distinction, especially when using `np.dot` (which we will use in a moment).

Takeawy - The `numpy` implementation for SVD is a bit in conflict with the typical notation for SVD. One needs to proceed carefully.

Let's check the shape of the things that we have now:

In [ ]:
```
# Code block for shape checks
print(U.shape)
print(S.shape)
print(Vt.shape)
```

Notice that the shape of S only has one value. This means that numpy is regarding this as an one-dimensional array. If we want to check that our matrix decomposition is close to the original data, we use the following trick from the svd helpfile:

In [ ]:
```
umax = S.shape[0]
np.allclose(justnum, np.dot(U[:, :umax] * S, Vt))
```

Let's take this line apart, piece by piece.

- The most interior part is `U[:, :umax] * S`. Here is a 2D array such that the i^{th} column of U is multiplied with the i^{th} entry of S. This creates a matrix that is 300 rows and just 4 columns.
- In `np.dot(U[:, :umax] * S, Vt)`, we are using the matrix multiplication version of `np.dot`. This means that we are taking the above resulting columns are multiplied by the rows of Vt. Since the Vt outputted from `linalg.svd` is actually what is called V^t in the theory (or V transposed), this is entirely correct (albeit confusing).
    - Finally, notice that in this "trick" we are using `np.allclose`. This method compares all the elements between two matrices and returns `True` if the entries in the same position are all close to each other. Put another way, if we want to compare A and B, then `np.allclose(A,B) = True` if (for all values of i and j)

SVD for dimension reduction

Like with PCA, we can use pieces of our matrice decomposition to perform a dimension reduction. Borrowing from the trick above, we can push our data into a lower dimension:

In [ ]:
```
justnum_svd2 = U[:, :2] * S[:2]
```

In [ ]:
```
# Create a scatter plot of the resulting dimension reduction
```

Resulting dimension reduction

Notice that here, while we have a two dimensional representation (in 2-D), we do not have nice axes that we can easily interpret. Part of this is due to not normalizing nor standardizing our data.

In [ ]:
```
# Perform SVD on the standardized data
U_std,S_std,V_std = np.linalg.svd(????,full_matrices=True)

justnum_std_svd2 =
```

In [ ]:
```
# Create a 2D plot of the output
```

Obtaining Low-dimensional approximations

As with PCA, we can create a low dimension approximation of our data in the original dimension. To create a k dimension approximation in the original dimension, we need:

- The first k columns of U
- The first k elements of S
- The first k rows of Vt

Putting these together creates the lower dimensional representation but in the original dimension

In [ ]:
```
k = 2
low_d = np.dot(U[:,:k] * S[:k], Vt[:k,:]))
```

In [ ]:
```
## Check the shape on LOW_D
# Is this what you expect?
```

`full_matrices` flag

You will notice that we used a flag in our SVD. This flag sets the output shape of our matrices, due to a technical theorectical point:

- The maximum number of singular values is the minimum of m and n.

So while we can create a matrix S that is of shape $m \times n$, so much of that matrix will have 0 entries. And just like with single numbers, when we have whole rows or columns of zeros in a matrix, multiplying with that matrix will yield rows or columns of zeros. This means that there are parts of either the U or V matrices that will never have an effect in our decomposition.

The `full_matrices` flag allows us to create a decomposition with only rows and columns that will be used.

In [ ]:
```
U_small,S_small,V_small = np.linalg.svd(justnum,full_matrices=False)
```

Lots of kinds of SVD

Similar to using the `full_matrices` flag, there are many versions of SVD that can be deployed in various circumstances. One version is called the truncated SVD where we decide in advance what dimension we want our data to be viewed in.

As the truncated SVD helpfile in sklearn states, truncated SVD is a good alternative for *sparse* data. So what is "sparse" data?

Sparse data

Sparse data is data where most of each observation's variables are equal to zero. Typically, when someone says "sparse", they mean that only about 10% of the total dataframe is non-zero. Although, you may hear of other specifications where someone says that their data has only 5% non-zero values.

Sparsity happens more often than you think. Consider a dataset about restaurants. We might have a variable for the type of food (ie. American, fusion, etc), another for whether they are open for breakfast (ie. yes or no), another for open for brunch, another for lunch, and another for dinner. We could have indicator variables (ie. ones with either "yes" or "no" for choices) for hundred options. For many of these options, restaurants may only have "yes" for a handful of them, and thus the data is likely *sparse*.

SVD on sparse data

To learn more about SVD on sparse data in python, check out this blog post

PCA & SVD - A Comparison

PCA and SVD are closely related (and with a bit of linear algebra, you can show this to be true). Both can be used for dimension reduction in different situations. Both also have their restrictions:

- PCA requires the data to be centered around 0
- If you want each variable to have even weight, then you also need to standardize your data
- SVD does not require the data to be normalized, but this imbues a sense of importance on each variable based on the span of the variable (think GPA vs hours slept)
- SVD can be performed on sparse data, while PCA cannot always be used

Both SVD and PCA are very powerful tools. Like choosing the target dimension, we have to choose whether to use PCA or SVD based on the situation that we have.

Final Thoughts

To finish up this lab, create ***one*** plot comparing PCA and SVD on the standardized data. Share your plot in a post on **#lab06_submission** channel on slack saying whether you prefer PCA or SVD and **why**. Cheer on your fellow Machine Learning practitioners as the votes in the epic discussion of PCA vs. SVD roll in!

If your have questions from this lab, post them to #lab_questions with the same preamble (i.e. starting with **Lab6**). If you have the same question, please use one of the emoji's to upvote the question. If you would like to answer someone's question, please use the thread function. This will tie your answer to their question.

References consulted

1. *Doing Data Science: Straight talk from the frontline* by C. O'Neil & R. Schutt (2014)
2. *Python Machine Learning*
3. PCA `sklearn` helpfile
4. Truncated SVD `sklearn` helpfile
5. SVD `numpy` helpfile
6. What is principal component analysis? from "Bits of DNA"
7. Explained variance in PCA

8. What is the intuitive relationship between SVD and PCA?
9. Reduced SVDs on Wikipedia
10. Sparse SVDs in Python
11. PCA and SVD explained with numpy
[illegible]