The curse of Dimensionality

Humans are bound by their perception of a maximum of three dimensions. We can't comprehend shapes/graphs beyond three dimensions. Often, data scientists get datasets which have thousands of features. They give birth to two kinds of problems:

- **Increase in computation time:** Majority of the machine learning algorithms they rely on the calculation of distance for model building and as the number of dimensions increases it becomes more and more computation-intensive to create a model out of it. For example, if we have to calculate the distance between two points in just one dimension, like two points on the number line, we'll just subtract the coordinate of one point from another and then take the magnitude:

Distance= $x_1 - x_2$

What if we need to calculate the distance between two points in two dimensions?

The same formula translates to: Distance= $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

What if we need to calculate the distance between two points in three dimensions?

The same formula translates to: Distance= $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$

And for N-dimensions, the formula becomes: Distance= $\sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2 + (c_1 - c_2)^2 + \ldots + (n_1 - n_2)^2}$

This is the effort of calculating the distance between two points. Just imagine the number of calculations involved for all the data points involved.

One more point to consider is that as the number of dimension increases, points are going far away from each other. This means that any new point that comes when we are testing the model is going to be farther away from our training points. This leads to a less reliable model, and it makes our model overfitted to the training data.

- **Hard (or almost impossible) to visualise the relationship between features:** As stated above, humans can not comprehend things beyond three dimensions. So, if we have an n-dimensional dataset, the only solution left to us is to create either a 2-D or 3-D graph out of it. Let's say for simplicity, we are creating 2-D graphs. Suppose we have 1000 features in the dataset. That results in a total (1000*999)/2= 499500 combinations possible for creating the 2-D graph.

Is it humanly possible to analyse all those graphs to understand the relationship between the variables?

The questions that we need to ask at this point are:

- Are all the features really contributing to decision making?
- Is there a way to come to the same conclusion using a lesser number of features?
- Is there a way to combine features to create a new feature and drop the old ones?
- Is there a way to remodel features in a way to make them visually comprehensible?

The answer to all the above questions is- *Dimensionality Reduction technique.*

What is a Dimensionality Reduction Technique?

Dimensionality reduction is a feature selection technique using which we reduce the number of features to be used for making a model without losing a significant amount of information compared to the original dataset. In other words, a dimensionality reduction technique projects a data of higher dimension to a lower-dimensional subspace.

When to use Dimensionality Reduction? Dimensionality reduction shall be used before feeding the data to a machine learning algorithm to achieve the following:

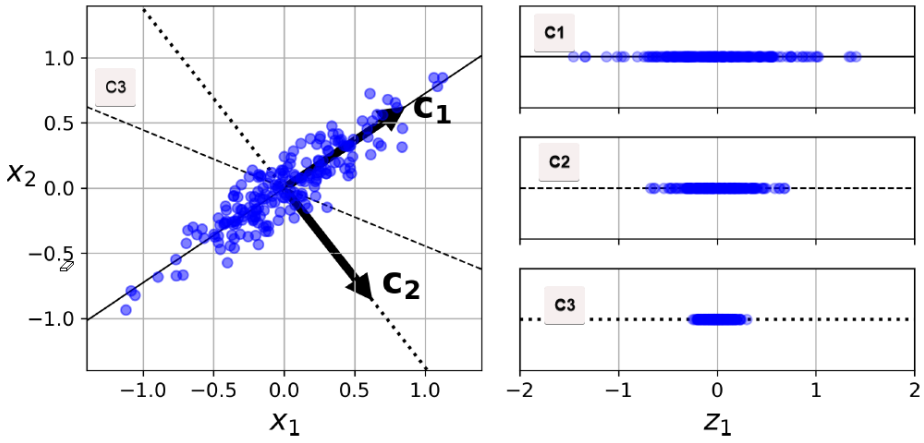
- It reduces the size of the space in which the distances are calculated, thereby improving machine learning algorithm performance.
- It reduces the degrees of freedom for our dataset avoiding chances of overfitting
- Reducing the dimensionality using dimensionality reduction techniques can simplify the dataset facilitating a better description, visualisation, and insight.

Principal Component Analysis:

The principal component analysis is an unsupervised machine learning algorithm used for feature selection using dimensionality reduction techniques. As the name suggests, it finds out the principal components from the data. PCA transforms and fits the data from a higher-dimensional space to a new, lower-dimensional subspace This results into an entirely new coordinate system of the points where the first axis corresponds to the first principal component that explains the most variance in the data.

What are the principal components? Principal components are the derived features which explain the maximum variance in the data. The first principal component explains the most variance, the 2nd a bit less and so on. Each of the new dimensions found using PCA is a linear combination of the old features.

Let's take the following example where the data is distributed like the diagram on the left:



In the diagram above, we are considering 3 orthogonal(*C3 is in the third dimension*) axes to show the distribution of data. If you notice the diagram on the right, the first two axes **C1** and **C2** successfully explain the maximum variation in the data whereas the axes **C3** only consists of a fewer number of points. Hence, while considering the principal components C1 and C2 will be our choices.

Mathematics Behind PCA

We are going to discuss PCA using a method called Singular Value Decomposition (SVD) which factorises the dataset matrix in such a way that it becomes a product of the multiplication of three individual matrices:

X(original Data)= $U * \Sigma * V^T$

Where V is the matrix that contains the principal components.

Pre-requisite: PCA assumes that the mean of all the individual columns is zero and the standard deviation is 1. So, before applying PCA, the data should be pre-processed appropriately.

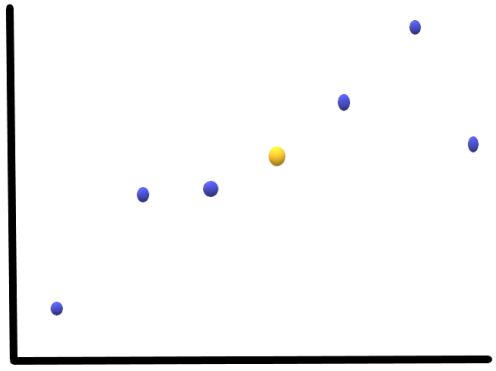
Let's take a simple example to understand it:

Let's suppose we have the following dataset:

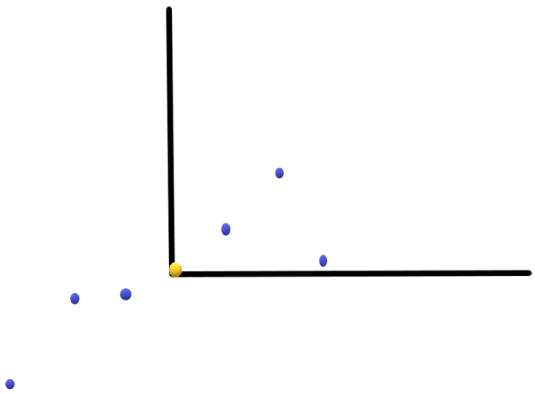
	Student1	Student2	Student3	Student4	Student5	Student6
Math	11	13	9	4	3	1
Computer	7	5	6	4	4	1

Steps to Calculate PCA

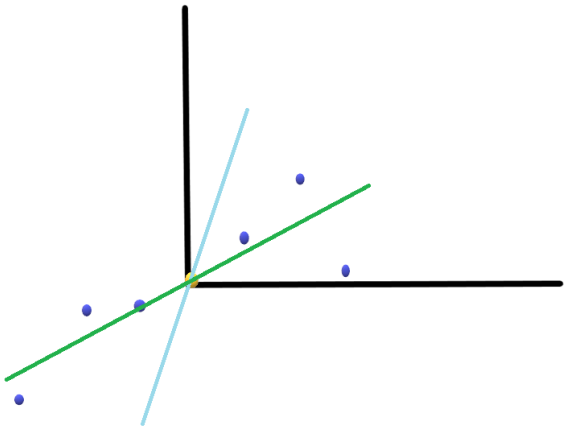
- Let's plot this on the XY plane and calculate the average of the magnitude of all the points. Blue ones are the actual points and the yellow one is the average point.



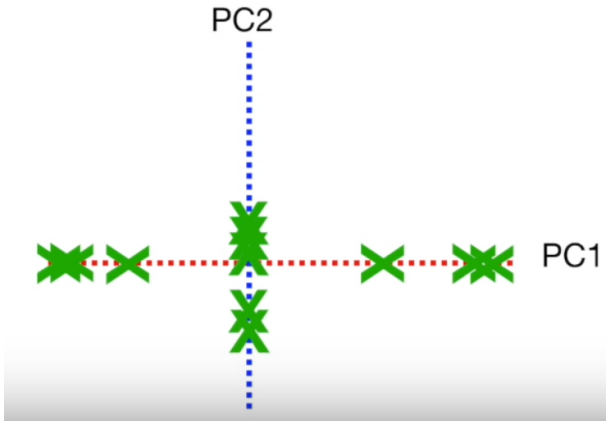
- Move the points so that the average point is on the origin. This is called a parallel translation. Although the coordinates of the points have changed, the corresponding distances among them remain the same.



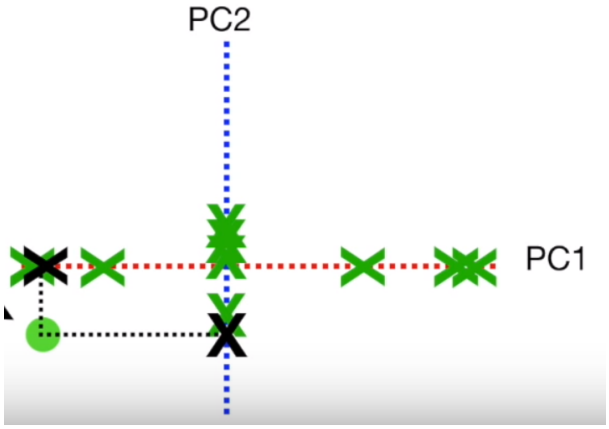
- Create the best fit line for the new data points. We first start with a random line(blue one), and then try to find the best fit line(the green one) so that the distance from individual data points is minimum and consequently the distance from origin is maximum. This best fit line is called Principal component1 or PC1.



- PC2 is a line perpendicular to the PC1.
- Then the axes PC1 and PC2 are rotated in a way that PC1 becomes the horizontal axis.



- Then based on the sample points the new points are projected using PC1 and PC2. Thus we get the derived features.



But the question is: if we talk about n dimensions, there are n-1 perpendicular lines possible on PC1. **How to select a line as PC2?**

And the next question is: **what is the optimum number of Principal components needed?**

Explained Variance Ratio

All of the above questions are answered using the *explained variance ratio*. It represents the amount of variance each principal component is able to explain.

For example, suppose if the square of distances of all the points from the origin that lie on PC1 is 50 and for the points on PC2 it's 5.

EVR of PC1= $\frac{Distance of PC1 points}{(Distance of PC1 points + Distance of PC2 points)} = \frac{50}{55} = 0.91$

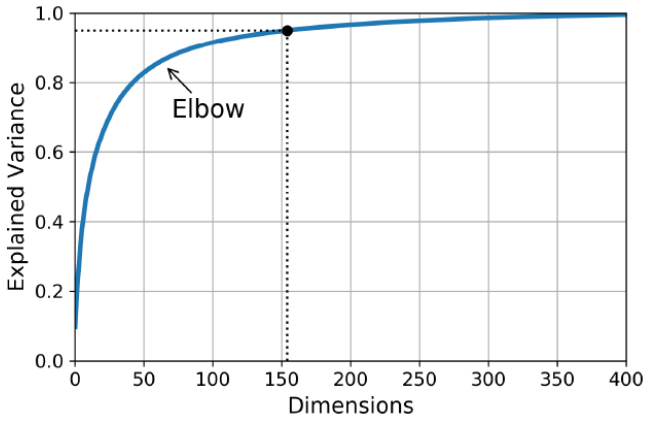
EVR of PC2= $\frac{Distance of PC2 points}{(Distance of PC1 points + Distance of PC2 points)} = \frac{5}{55} = 0.09$

Thus PC1 explains 91% of the variance of data. Whereas, PC2 only explains 9% of the variance. Hence we can use only PC1 as the input for our model as it explains the majority of the variance.

In a real-life scenario, this problem is solved using the **Scree Plots**

Scree Plots:

Scree plots are the graphs that convey how much variance is explained by corresponding Principal components.



As shown in the given diagram, around 75 principal components explain approximately 90 % of the variance. Hence, 75 can be a good choice based on the scenario

Explaining the Maths involved through code

In [55]:

```
import numpy as np
# Creating an Array
A = np.array([
        [ 3,  7],
        [-4, -6],
        [ 7,  8],
        [ 1, -1],
        [-4, -1],
        [-3, -7]
    ])

m,n = A.shape # m-observations, n-features

print("Array:")
print(A) # our array

print("---")
print("Dimensions:")
print(A.shape) # shape

print("---")
print("Mean across Rows:")
print(np.mean(A,axis=0))
```

```
Array:
[[ 3  7]
 [-4 -6]
 [ 7  8]
 [ 1 -1]
 [-4 -1]
 [-3 -7]]
---
Dimensions:
(6, 2)
---
Mean across Rows:
[0. 0.]
```

In [56]:

```
# Converting the array into a DataFrame ...
import pandas as pd
df = pd.DataFrame(A, columns = ['a0', 'a1'])
print(df)
```

```
   a0  a1
0   3   7
1  -4  -6
2   7   8
3   1  -1
4  -4  -1
5  -3  -7
```

In [57]:

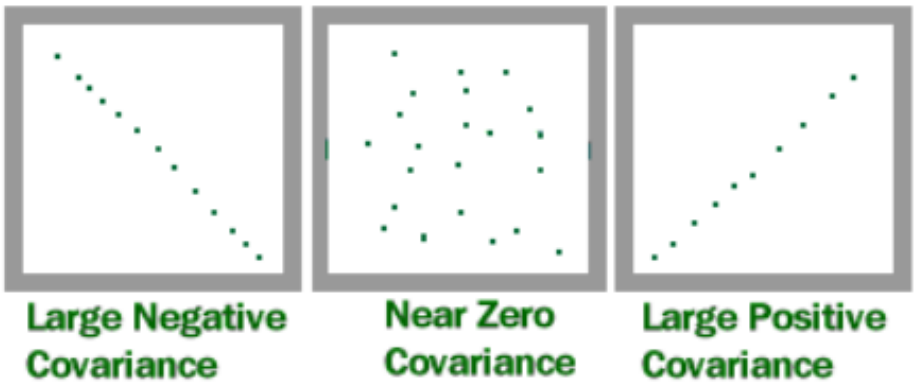
```
# ... and a dataframe can as easily be converted to an array
df.values
```

Out[57]:

```
array([[ 3,  7],
       [-4, -6],
       [ 7,  8],
       [ 1, -1],
       [-4, -1],
       [-3, -7]])
```

Covariance

Variance is the measure of how a variable changes or varies and *co* means together. Hence, *covariance* is the measure of how two variables change together.



If the covariance is high, it means that the variables are highly correlated and change in one results in a change in the other one too. Generally, we avoid using highly correlated variables in building a machine learning model.

In [94]:

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# makes charts pretty
import seaborn as sns
sns.set(color_codes=True)
```

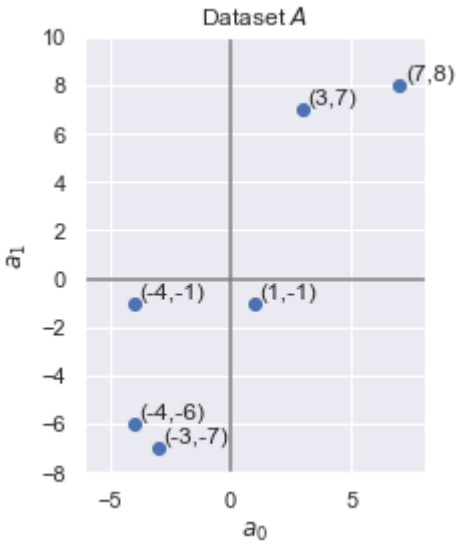
In [95]:

```
# plots
plt.scatter(A[:,0],A[:,1])   # create a scatter plot

# annotations
for i in range(m):
    plt.annotate('('+str(A[i,0])+','+str(A[i,1])+')',(A[i,0]+0.2,A[i,1]+0.2))

# axes
plt.plot([-6,8],[0,0],'grey') # x-axis
plt.plot([0,0],[-8,10],'grey') # y-axis
plt.axis([-6, 8, -8, 10])
plt.axes().set_aspect('equal')

# labels
plt.xlabel("$a_0$")
plt.ylabel("$a_1$")
plt.title("Dataset $A$")
```

```
C:\Users\virat\Anaconda3\lib\site-packages\matplotlib\figure.py:98: MatplotlibDeprecationWarning:
Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.  In a future version, a new instance will always be created and returned.  Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
  "Adding an axes using the same arguments as a previous axes "
```

Out[95]:

```
Text(0.5, 1.0, 'Dataset $A$')
```



Sample covariance between a_0 and a_1 :

$$cov_{a_0,a_1} = \frac{\sum_{k=0}^{m-1}(a_0^k - \bar{a_0})(a_1^k - \bar{a_1})}{m-1}$$

where $\bar{a_0}$ is the mean of column a_0 and $\bar{a_1}$ is the mean of column a_1

In [96]:

```
# Calculate covariance of a0 and a1
a0 = A[:,0]
a1 = A[:,1]
product = a0*a1 # element-wise product
print("Length of prod equals " + str(len(product)))
print("---")
print("Covariance:")
print(np.sum(prod)/(m-1))
```

```
Length of prod equals 6
---
Covariance:
25.0
```

In [97]:

```
# Get more stuff using NumPy's covariance method
np.cov(a0,a1)
```

Out[97]:

```
array([[20., 25.],
       [25., 40.]])
```

The Linear Algebra way:

$$\Sigma = \frac{A^T A}{(m-1)}$$

In [98]:

```
# What is A.T?
A.T # This is the transpose of matrix A
```

Out[98]:

```
array([[ 3, -4,  7,  1, -4, -3],
       [ 7, -6,  8, -1, -1, -7]])
```

In [99]:

```
# Matrix Multiplication, @ operator is used for calculating the dot product of two matr
A.T @ A # or np.dot(A.T,A)
```

Out[99]:

```
array([[100, 125],
       [125, 200]])
```

In [100]:

```
# As stated in the formula now we need to divide the  product by (m-1) to yield true Sc
# Let's call it Sigma
Sigma = (A.T @ A)/(m-1) # or np.cov(A.T)
Sigma
```

Out[100]:

```
array([[20., 25.],
       [25., 40.]])
```

Eigen-decomposition of Σ

According to Wikipedia article on PCA (https://en.m.wikipedia.org/wiki/Principal_component_analysis), *"PCA can be done by eigenvalue decomposition of a data covariance (or correlation) matrix or singular value decomposition of a data matrix."* The second approach has already been discussed above. Let's discuss the first approach now.

Σ is a real, symmetric matrix; thus, it has

1) real eigenvalues, and

2) orthogonal eigenvectors.

Definition:

An **eigenvector v** of a linear transformation **T** is a nonzero vector that, when **T** is applied to it, does not change direction. Applying **T** to the eigenvector only scales the eigenvector by the scalar value λ, called an **eigenvalue**. This condition can be written as the equation

$$T(\mathbf{v}) = \lambda \mathbf{v},$$

In [101]:

```
# obtaining the eigenvalues and eigen vectors for the matrix Sigma
l, X = np.linalg.eig(Sigma)
print("Eigenvalues:")
print(l)
print("---")
print("Eigenvectors:")
print(X)
```

```
Eigenvalues:
[ 3.07417596 56.92582404]
---
Eigenvectors:
[[-0.82806723 -0.56062881]
 [ 0.56062881 -0.82806723]]
```

Recall from your Linear Algebra class that the following should hold:

$$\Sigma x_0 = \lambda_0 x_0$$
$$\Sigma x_1 = \lambda_1 x_1$$

In [102]:

```
# let's check the first Eigenvalue, Eigenvector combination
print("Sigma times eigenvector:")
print(Sigma @ X[:,0]) # 2x2 times 2x1
print("Eigenvalue times eigenvector:")
print(l[0] * X[:,0]) # scalar times 2x1
```

```
Sigma times eigenvector:
[-2.54562438  1.72347161]
Eigenvalue times eigenvector:
[-2.54562438  1.72347161]
```

In [103]:

```
# ... and the product with the second eigenvalue
print("Sigma times eigenvector:")
print(Sigma @ X[:,1]) # 2x2 times 2x1
print("Eigenvalue times eigenvector:")
print(l[1] * X[:,1]) # scalar times 2x1, ANNOYING - MUST USE * vs. @
```

```
Sigma times eigenvector:
[-31.91425695 -47.13840945]
Eigenvalue times eigenvector:
[-31.91425695 -47.13840945]
```

In [104]:

```
print("The first principal component is eigenvector with largest evalue:")
print(X[:,1])
print("---")
print("Second principal component:")
print(X[:,0])
```

```
The first principal component is eigenvector with largest evalue:
[-0.56062881 -0.82806723]
---
Second principal component:
[-0.82806723  0.56062881]
```

In [105]:

```
# Are the two Principal components Orthogonal? If the dot product of two matrices is ze
X[:,1].T @ X[:,0]
```

Out[105]:

```
0.0
```

In [106]:

```
# plotting the Eigen Vectors
plt.scatter(A[:,0],A[:,1])
scale = 3 # increase this scaling factor to highlight these vectors
plt.plot([0,X[0,1]*scale],[0,X[1,1]*scale],'r') # First principal component
plt.plot([0,X[0,0]*scale],[0,X[1,0]*scale],'g') # Second principal component

# annotations
for i in range(m):
    plt.annotate('('+str(A[i,0])+','+str(A[i,1])+')',(A[i,0]+0.2,A[i,1]+0.2))

# axes
plt.plot([-6,8],[0,0],'grey') # x-axis
plt.plot([0,0],[-8,10],'grey') # y-axis
plt.axis([-6, 8, -8, 10])
plt.axes().set_aspect('equal')

# labels
plt.xlabel("$a_0$")
plt.ylabel("$a_1$")
plt.title("Eigenvectors of $\Sigma$")
```

```
C:\Users\virat\Anaconda3\lib\site-packages\matplotlib\figure.py:98: Matplo
tlibDeprecationWarning:
Adding an axes using the same arguments as a previous axes currently reuse
s the earlier instance.  In a future version, a new instance will always b
e created and returned.  Meanwhile, this warning can be suppressed, and th
e future behavior ensured, by passing a unique label to each axes instanc
e.
  "Adding an axes using the same arguments as a previous axes "
```

Out[106]:

```
Text(0.5, 1.0, 'Eigenvectors of $\\Sigma$')
```

Dimensionality Reduction: 2D to 1D

In [112]:

```
# change to matrix
Amat = np.asmatrix(A)
Xmat = np.asmatrix(X)
Amat
```

Out[112]:

```
matrix([[ 3,  7],
        [-4, -6],
        [ 7,  8],
        [ 1, -1],
        [-4, -1],
        [-3, -7]])
```

In [110]:

```
# Choose eigenvector with highest eigenvalue as first principal component
pc1 = Xmat[:,1]
pc1
```

Out[110]:

```
matrix([[-0.56062881],
        [-0.82806723]])
```

In [111]:

```
Acomp = Amat @ pc1 # the dot product of a 6x2 and 2x1 matrix yields a 6x1 matrix
print("Compressed version of A:")
print(Acomp)
```

```
Compressed version of A:
[[ -7.47835704]
 [  7.21091862]
 [-10.54893951]
 [  0.26743842]
 [  3.07058247]
 [  7.47835704]]
```

In [113]:

```
Arec = Acomp @ pc1.T # the dot product of a  6x1 matrix and 1x2 matrix results into a 6
print("Reconstruction from 1D compression of A:")
print(Arec)
```

```
Reconstruction from 1D compression of A:
[[ 4.1925824   6.1925824 ]
 [-4.04264872 -5.97112541]
 [ 5.9140394   8.73523112]
 [-0.14993368 -0.22145699]
 [-1.72145699 -2.54264872]
 [-4.1925824  -6.1925824 ]]
```

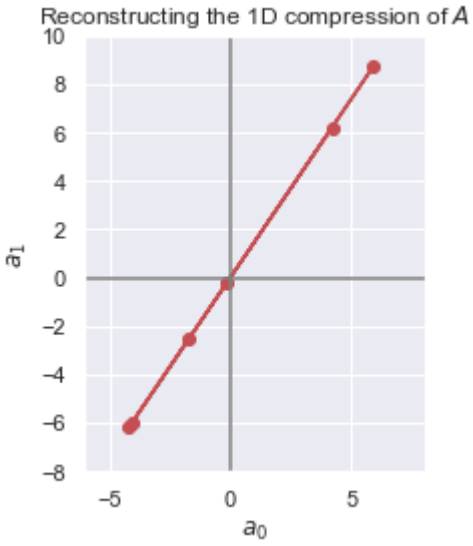
In [117]:

```
plt.plot(Arec[:,0],Arec[:,1],'r', marker='o') # Arec in RED

# axes
plt.plot([-6,8],[0,0],'grey') # x-axis
plt.plot([0,0],[-8,10],'grey') # y-axis
plt.axis([-6, 8, -8, 10])
plt.axes().set_aspect('equal')

# labels
plt.xlabel("$a_0$")
plt.ylabel("$a_1$")
plt.title("Reconstructing the 1D compression of $A$")
```

```
C:\Users\virat\Anaconda3\lib\site-packages\matplotlib\figure.py:98: Matplotl
ibDeprecationWarning:
Adding an axes using the same arguments as a previous axes currently reuses
the earlier instance.  In a future version, a new instance will always be cr
eated and returned.  Meanwhile, this warning can be suppressed, and the futu
re behavior ensured, by passing a unique label to each axes instance.
  "Adding an axes using the same arguments as a previous axes "
```

Out[117]:

```
Text(0.5, 1.0, 'Reconstructing the 1D compression of $A$')
```



In [118]:

```
print(np.linalg.matrix_rank(Amat)) # originally a Rank 2 matrix
print(np.linalg.matrix_rank(Arec)) # reconstructed matrix is Rank 1
```

```
2
1
```

By taking on the Rank-1 matrix related to the 2nd eigenvector you get back to the original data

In [119]:

```
# Add the Rank 1 matrix for the other vector to recover A completely
# Here we are taking the dot product of matrix A with the principal components and the
Amat @ Xmat[:,1] @ Xmat[:,1].T + Amat @ Xmat[:,0] @ Xmat[:,0].T
```

Out[119]:

```
matrix([[ 3.,  7.],
        [-4., -6.],
        [ 7.,  8.],
        [ 1., -1.],
        [-4., -1.],
        [-3., -7.]])
```

In [121]:

```
# Why does this work? Well, recall that the dot product of a matrix and its transpose (
# Hence the entire expression becomes equivalent to multiplying a matrix with a unit ma
A @ Xmat @ Xmat.T
```

Out[121]:

```
matrix([[ 3.,  7.],
        [-4., -6.],
        [ 7.,  8.],
        [ 1., -1.],
        [-4., -1.],
        [-3., -7.]])
```

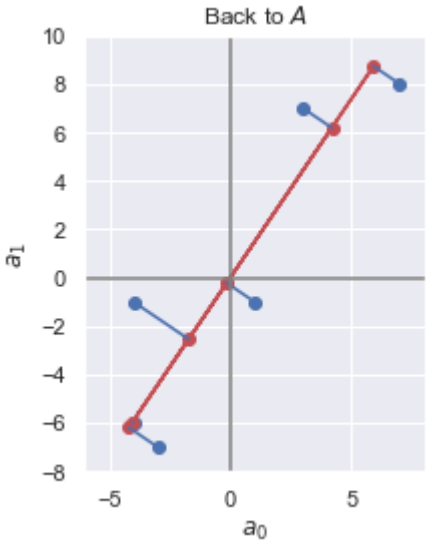
In [122]:

```
# plots
plt.scatter(A[:,0], A[:,1]) # A in blue
plt.plot(Arec[:,0],Arec[:,1],'r', marker='o') # Arec in RED

# across observations
for i in range(m):
    e = np.vstack((A[i],Arec[i]))
    plt.plot(e[:,0],e[:,1],'b') # BLUE

# axes
plt.plot([-6,8],[0,0],'grey') # x-axis
plt.plot([0,0],[-8,10],'grey') # y-axis
plt.axis([-6, 8, -8, 10])
plt.axes().set_aspect('equal')

# labels
plt.xlabel("$a_0$")
plt.ylabel("$a_1$")
plt.title("Back to $A$")
```

```
C:\Users\virat\Anaconda3\lib\site-packages\matplotlib\figure.py:98: Matplotl
ibDeprecationWarning:
Adding an axes using the same arguments as a previous axes currently reuses
the earlier instance.  In a future version, a new instance will always be cr
eated and returned.  Meanwhile, this warning can be suppressed, and the futu
re behavior ensured, by passing a unique label to each axes instance.
  "Adding an axes using the same arguments as a previous axes "
```

Out[122]:

```
Text(0.5, 1.0, 'Back to $A$')
```



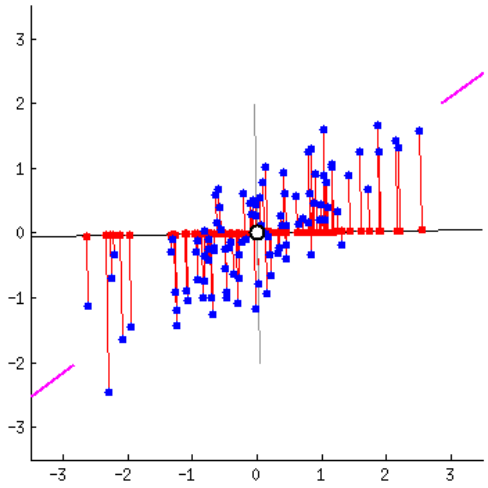
Summary of Eigen-decomposition Approach

1. Normalize columns of A so that each feature has a mean of zero
2. Compute sample covariance matrix $\Sigma = A^T A/(m-1)$
3. Perform eigen-decomposition of Σ using `np.linalg.eig(Sigma)`
4. Compress by ordering k evectors according to largest e-values and compute AX_k
5. Reconstruct from the compressed version by computing $AX_k X_k^T$

All the above steps can be summarized with the following gif. [Wicked animated GIF which illustrates PCA (http://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues)](http://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues)

Magically, eigen-decomposition (or PCA) finds the line where

1. the spread of values along the black line is **maximal**
2. the projection error (sum of red lines) is **minimal**



Python Implementation

In [31]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [32]:

```
# we are using the free glass datset.
# The objective is to tell the type of glass based on amount of other elements present.
data = pd.read_csv('glass.data')
```

In [33]:

```
data.head()
```

Out[33]:

	index	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Class
0	1	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0	1
1	2	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0	1
2	3	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.0	1
3	4	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0	1
4	5	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0	1

In [54]:

```
1 data.isna().sum()
```

Out[54]:

```
RI    0
Na    0
Mg    0
Al    0
Si    0
K     0
Ca    0
Ba    0
Fe    0
dtype: int64
```

In [42]:

```
1 data=data.drop(labels=['index','Class'], axis=1)
```

In [43]:

```
1 data.describe()
```

Out[43]:

	RI	Na	Mg	Al	Si	K	Ca	
count	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	21
mean	1.518365	13.407850	2.684533	1.444907	72.650935	0.497056	8.956963	
std	0.003037	0.816604	1.442408	0.499270	0.774546	0.652192	1.423153	
min	1.511150	10.730000	0.000000	0.290000	69.810000	0.000000	5.430000	
25%	1.516523	12.907500	2.115000	1.190000	72.280000	0.122500	8.240000	
50%	1.517680	13.300000	3.480000	1.360000	72.790000	0.555000	8.600000	
75%	1.519157	13.825000	3.600000	1.630000	73.087500	0.610000	9.172500	
max	1.533930	17.380000	4.490000	3.500000	75.410000	6.210000	16.190000	

We'll go ahead and standardise this data as all the data is on a different scale.

In [44]:

```
1 from sklearn.preprocessing import StandardScaler
2 scaler=StandardScaler()
3 scaled_data=scaler.fit_transform(data)
4
```

In [48]:

```
1 df=pd.DataFrame(data=scaled_data, columns= data.columns)
```

In [49]:

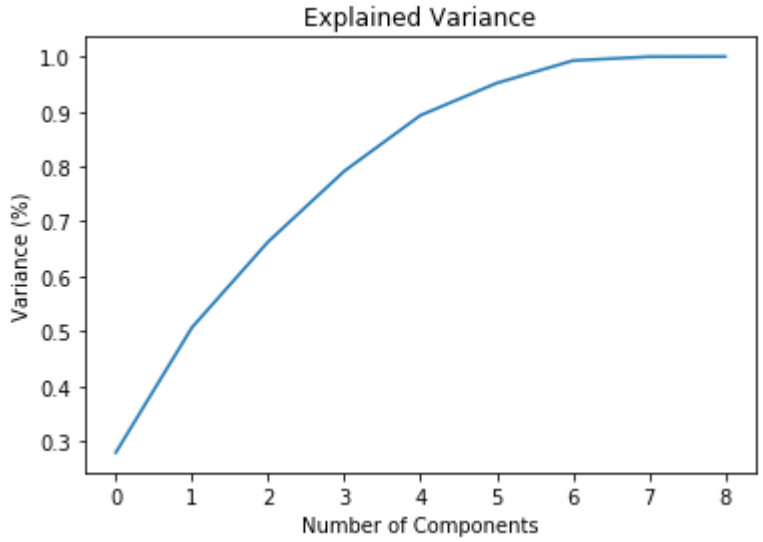
```
df.describe()
```

Out[49]:

	RI	Na	Mg	Al	Si	K
count	2.140000e+02	2.140000e+02	2.140000e+02	2.140000e+02	2.140000e+02	2.140000e+02
mean	-2.737478e-14	2.179980e-15	-2.801497e-16	-3.434428e-16	9.966067e-16	7.470660e-17
std	1.002345e+00	1.002345e+00	1.002345e+00	1.002345e+00	1.002345e+00	1.002345e+00
min	-2.381516e+00	-3.286943e+00	-1.865511e+00	-2.318616e+00	-3.676472e+00	-7.639186e-01
25%	-6.082728e-01	-6.141580e-01	-3.957744e-01	-5.117560e-01	-4.800288e-01	-5.756501e-01
50%	-2.262293e-01	-1.323817e-01	5.527787e-01	-1.704602e-01	1.799655e-01	8.905322e-02
75%	2.614331e-01	5.120326e-01	6.361680e-01	3.715977e-01	5.649621e-01	1.735820e-01
max	5.137232e+00	4.875637e+00	1.254639e+00	4.125851e+00	3.570524e+00	8.780145e+00

In [51]:

```
from sklearn.decomposition import PCA
pca = PCA()
principalComponents = pca.fit_transform(df)
plt.figure()
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Variance (%)') #for each component
plt.title('Explained Variance')
plt.show()
```



From the diagram above, it can be seen that 4 principal components explain almost 90% of the variance in data and 5 principal components explain around 95% of the variance in data.

So, instead of giving all the columns as input, we'd only feed these 4 principal components of the data to the machine learning algorithm and we'd obtain a similar result.

In [52]:

```
pca = PCA(n_components=4)
new_data = pca.fit_transform(df)
# This will be the new data fed to the algorithm.
principal_Df = pd.DataFrame(data = new_data
             , columns = ['principal component 1', 'principal component 2','principal c
```

In [53]:

```
principal_Df.head()
```

Out[53]:

	principal component 1	principal component 2	principal component 3	principal component 4
0	1.151140	-0.529488	-0.372096	1.728901
1	-0.574137	-0.759788	-0.556708	0.760232
2	-0.940160	-0.929836	-0.554907	0.206254
3	-0.142083	-0.961677	-0.117125	0.415724
4	-0.351092	-1.091249	-0.485079	0.069102

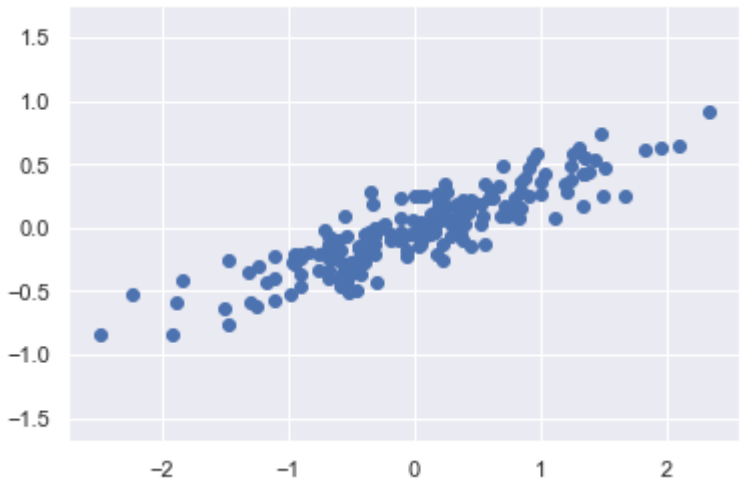
Here, we see that earlier we had 9 columns in the data earlier. Now with the help of Scree plot and PCA, we have reduced the number of features to be used for model building to 4. This is the advantage of PCA. *It drastically reduces the number of features, thereby considerably reducing the training time for the model.*

Visualizing the Principal components

As humans can only perceive 3dimensions, we'll take a dataset with less than 4 dimensions.

In [72]:

```
np.random.seed(1)
X = np.dot(np.random.random(size=(2, 2)), np.random.normal(size=(2, 200))).T
plt.plot(X[:, 0], X[:, 1], 'o')
plt.axis('equal');
```



PCA seeks to find the **Principal Axes** in the data, and explain how vital those axes are in describing the data

distribution

In [73]:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
print(pca.explained_variance_)
print(pca.components_)
```

```
[0.7625315 0.0184779]
[[-0.94446029 -0.32862557]
 [-0.32862557  0.94446029]]
```

In [74]:

```
#To see what these numbers mean, let's view them as vectors plotted on top of the data:

plt.plot(X[:, 0], X[:, 1], 'o', alpha=0.5)
for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)
    plt.plot([0, v[0]], [0, v[1]], '-k', lw=3)
plt.axis('equal');
```

Notice that one vector is longer than the other. In a sense, this tells us that that direction in the data is somehow more "important" than the other direction. The explained variance quantifies this measure of "importance" in a direction.

Another way to think of it is that the second principal component could be **completely ignored** without much loss of information! Let's see what our data look like if we only keep 95% of the variance

In [75]:

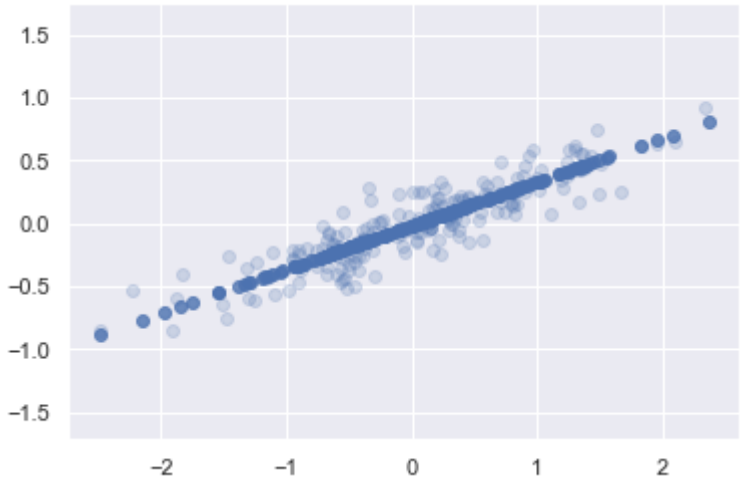
```
clf = PCA(0.95) # keep 95% of variance
X_trans = clf.fit_transform(X)
print(X.shape)
print(X_trans.shape)
```

```
(200, 2)
(200, 1)
```

By specifying that we want to throw away 5% of the variance, the data is now compressed by a factor of 50%! Let's see what the data look like after this compression:

In [76]:

```
X_new = clf.inverse_transform(X_trans)
plt.plot(X[:, 0], X[:, 1], 'o', alpha=0.2)
plt.plot(X_new[:, 0], X_new[:, 1], 'ob', alpha=0.8)
plt.axis('equal');
```



The lighter points are the original data, while the dark points are the projected version on the principal component axis. We see that after truncating 5% of the variance of this dataset and then reprojecting it, the "most important" features of the data are maintained, and we've compressed the data by 50%!

This is the sense in which "dimensionality reduction" works: if you can approximate a data set in a lower dimension, you can often have an easier time visualizing it or fitting complicated models to the data.

Application of PCA to the Digits Data

The dimensionality reduction might seem a bit abstract in two dimensions, but the projection and dimensionality reduction can be extremely useful when visualizing high-dimensional data. Let's implement PCA to the digits data. This data consists of a collection of different points in the plane to represent a digit

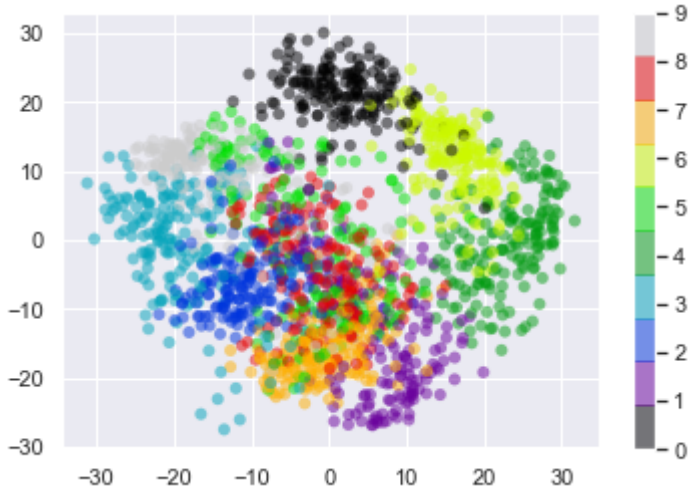
In [77]:

```
from sklearn.datasets import load_digits
digits = load_digits()
X = digits.data
y = digits.target

pca = PCA(2)  # project from 64 to 2 dimensions
Xproj = pca.fit_transform(X)
print(X.shape)
print(Xproj.shape)
```

```
(1797, 64)
(1797, 2)
```

In [78]:

```
# Creating a scatter plot of the datapoints
plt.scatter(Xproj[:, 0], Xproj[:, 1], c=y, edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('nipy_spectral', 10))
plt.colorbar();
```



This gives us an idea of the relationship between the datapoints. Essentially, we have found the optimal stretch and rotation in 64-dimensional space and tried to fit it to a 2-Dimensional space that allows us to see the layout of the digits, **without reference** to the labels.

What do the Components Mean?

This gives us an idea of the relationship between the datapoints. Essentially, we have made the data of 64 dimension fit to a 2-Dimensional space that allows us to see the layout of the digits, **without reference** to the labels.

$$x = [x_1, x_2, x_3 \cdots]$$

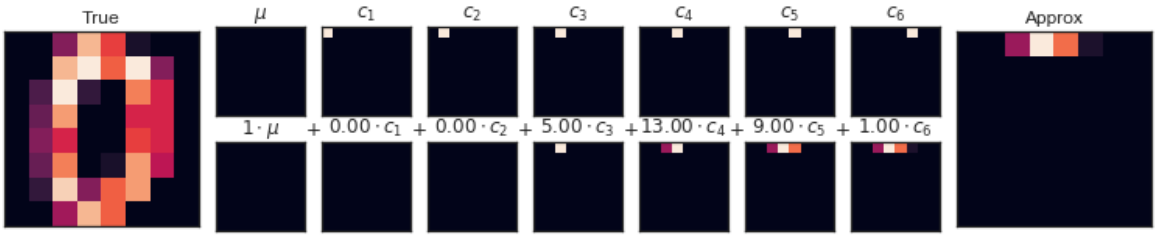
but what this really means is

$$image(x) = x_1 \cdot (\text{pixel 1}) + x_2 \cdot (\text{pixel 2}) + x_3 \cdot (\text{pixel 3}) \cdots$$

If we reduce the dimensionality in the pixel space to (say) 6, we recover only a partial image:

In [82]:

```
# Before running this, download the fig_code library from Git
from fig_code.figures import plot_image_components

sns.set_style('white')
plot_image_components(digits.data[0])
```



Pixel-wise representation of those digits is not the only choice we have. We can also use other *basis functions*, and show it like:

$$image(x) = \text{mean} + x_1 \cdot (\text{basis } 1) + x_2 \cdot (\text{basis } 2) + x_3 \cdot (\text{basis } 3) \cdots$$

What PCA does is to choose optimal **basis functions** so that only a few are needed to get a reasonable approximation. The low-dimensional representation of our data is the coefficients of this series, and the approximate reconstruction is the result of the sum:

In [83]:

```
from fig_code.figures import plot_pca_interactive
plot_pca_interactive(digits.data)
```

```
C:\Users\virat\Anaconda3\lib\site-packages\IPython\html.py:14: ShimWarning:
The `IPython.html` package has been deprecated since IPython 4.0. You should
import from `notebook` instead. `IPython.html.widgets` has moved to `ipywidg
ets`.
  "`IPython.html.widgets` has moved to `ipywidgets`.", ShimWarning)
```

```
interactive(children=(IntSlider(value=0, description='i', max=1796), Output
()), _dom_classes=('widget-interact…
```

Here we see that with only six PCA components, we recover a reasonable approximation of the input!

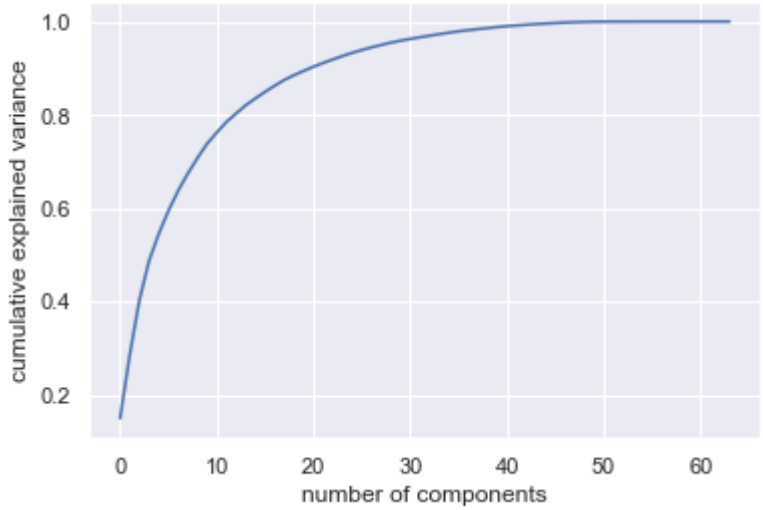
Thus we see that PCA can be viewed from two angles. It can be viewed as **dimensionality reduction**, or it can be viewed as a form of **lossy data compression** where the loss favours noise. In this way, PCA can be used as a **filtering** mechanism as well.

Choosing the Number of Components

But how much information have we thrown away? We can figure this out by looking at the **explained variance** as a function of the components:

In [84]:

```
sns.set()
pca = PCA().fit(X)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



From the Scree plot, it can be seen that 20 components are required to explain 90% of the variance which is still better than computing using all the 64 features. The explained variance threshold can be choosen based on the doamin and business requirements.

PCA for data compression

As mentioned, PCA can be used for a sort of data compression as well. Using a smaller value of `n_components` allows you to represent a higher dimensional point as a sum of just a few principal component vectors.

Here's what a single digit looks like when you change the number of components:

In [85]:

```
fig, axes = plt.subplots(8, 8, figsize=(8, 8))
fig.subplots_adjust(hspace=0.1, wspace=0.1)

for i, ax in enumerate(axes.flat):
    pca = PCA(i + 1).fit(X)
    im = pca.inverse_transform(pca.transform(X[20:21]))

    ax.imshow(im.reshape((8, 8)), cmap='binary')
    ax.text(0.95, 0.05, 'n = {0}'.format(i + 1), ha='right',
            transform=ax.transAxes, color='green')
    ax.set_xticks([])
    ax.set_yticks([])
```

Let's take another look at this by using IPython's `interact` functionality to view the reconstruction of several images at once:

In [87]:

```
from IPython.html.widgets import interact

def plot_digits(n_components):
    fig = plt.figure(figsize=(8, 8))
    plt.subplot(1, 1, 1, frameon=False, xticks=[], yticks=[])
    nside = 10

    pca = PCA(n_components).fit(X)
    Xproj = pca.inverse_transform(pca.transform(X[:nside ** 2]))
    Xproj = np.reshape(Xproj, (nside, nside, 8, 8))
    total_var = pca.explained_variance_ratio_.sum()

    im = np.vstack([np.hstack([Xproj[i, j] for j in range(nside)])
                    for i in range(nside)])
    plt.imshow(im)
    plt.grid(False)
    plt.title("n = {0}, variance = {1:.2f}".format(n_components, total_var),
                  size=18)
    plt.clim(0, 16)

interact(plot_digits, n_components=range(1, 64), nside=range(1, 8)) # A in blue

```

```
interactive(children=(Dropdown(description='n_components', options=(1, 2, 3,
4, 5, 6, 7, 8, 9, 10, 11, 12, 13,…
```

Out[87]:

```
<function __main__.plot_digits(n_components)>
```

In the diagram above, we can dynamically select the number of principal components and get to know the explained percentage of variance.

Pros of PCA:

- Correlated features are removed.
- Model training time is reduced.
- Overfitting is reduced.
- Helps in better visualizations
- Ability to handle noise

Cons of PCA

- The resultant principal components are less interpretable than the original data
- Can lead to information loss if the explained variance threshold is not considered appropriately.

Conclusion

From all the explanations above, we can conclude that PCA is a very powerful technique for reducing the dimensions of the data, projecting the data from a higher dimension to a lower dimension, helps in data visualization, helps in data compression and most of all increases the model training speed drastically by decreasing the number of variables involved in computation.

In [ ]:

```
1
```