K-Means Clustering

K-Means is a simple yet powerful clustering algorithm used in unsupervised learning. It partitions a dataset into **K clusters** based on feature similarity.

Overview

Goal: Divide data into (K) clusters where each point belongs to the cluster with the nearest mean (centroid).

Steps:

1. Initialize (K) centroids randomly.
2. Assign each data point to the nearest centroid.
3. Update the centroids by calculating the mean of all points assigned to each cluster.
4. Repeat steps 2 and 3 until convergence (no change in centroids or a maximum number of iterations is reached).

Step-by-Step Implementation

Here's a simple implementation of K-Means clustering from scratch using Python.

1. Import Libraries

```
import numpy as np
import matplotlib.pyplot as plt
```

2. Generate a Dataset

For simplicity, we will generate synthetic data.

```
from sklearn.datasets import make_blobs

# Generate sample data with 3 clusters
data, _ = make_blobs(n_samples=300, centers=3, random_state=42, cluster_std=1.

# Plot the dataset
plt.scatter(data[:, 0], data[:, 1], s=30, color='gray')
plt.title("Input Data")
plt.show()
```

`n_samples=300` : This specifies the total number of data points (samples) to generate. In this case, 300 samples will be created. `centers=3` : This sets the number of centers (or clusters) to generate. Here, the data will be grouped into 3 distinct clusters. `random_state=42` : This ensures reproducibility of the dataset. By setting a seed value (42 in this case), the same dataset will be generated each time the code is run. `cluster_std=1.0` : This controls the standard deviation of the clusters. A standard deviation of 1.0 means the clusters will have a moderate spread.

3. Initialize Centroids

Randomly select (K) data points as the initial centroids.

```
def initialize_centroids(data, k):
    """Randomly select k data points as initial centroids."""
    indices = np.random.choice(data.shape[0], size=k, replace=False)
    return data[indices]
```

4. Assign Clusters

For each data point, find the nearest centroid and assign it to that cluster.

```
def assign_clusters(data, centroids):
    """Assign each data point to the nearest centroid."""
```

```
    distances = np.linalg.norm(data[:, np.newaxis] - centroids, axis=2)
    return np.argmin(distances, axis=1)
```

5. Update Centroids

Recompute the centroids as the mean of all data points assigned to each cluster.

```
def update_centroids(data, labels, k):
    """Compute the new centroids as the mean of points in each cluster."""
    centroids = np.array([data[labels == i].mean(axis=0) for i in range(k)])
    return centroids
```

- The list comprehension [data[labels == i].mean(axis=0) for i in range(k)] iterates over each cluster index i from 0 to k-1.
- For each cluster i, it selects the data points that belong to that cluster (`data[labels == i]`).
- It then computes the mean of these points along the feature axis (`axis=0`), resulting in the new centroid for that cluster.
- Finally, it converts the list of centroids into a NumPy array.

6. Define the K-Means Algorithm

Combine all the steps into a single function.

```
def k_means(data, k, max_iters=100):
    """K-Means clustering algorithm."""
    # Step 1: Initialize centroids
    centroids = initialize_centroids(data, k)

    for _ in range(max_iters):
        # Step 2: Assign clusters
        labels = assign_clusters(data, centroids)

        # Step 3: Update centroids
        new_centroids = update_centroids(data, labels, k)

        # Convergence check
        if np.all(centroids == new_centroids):
            break
        centroids = new_centroids

    return labels, centroids
```

7. Run the Algorithm

```
k = 3  # Number of clusters
labels, centroids = k_means(data, k)

# Plot the results
for i in range(k):
    plt.scatter(data[labels == i][:, 0], data[labels == i][:, 1], label=f"Clus
plt.scatter(centroids[:, 0], centroids[:, 1], color='red', marker='x', s=200,
plt.legend()
plt.title("K-Means Clustering")
plt.show()
```

Explanation of Results

- The **clusters** represent groups of similar data points.
- The **centroids** are the central points of these clusters.

Notes

1. **Limitations**:
    - Sensitive to initial centroid placement (may converge to local minima).
    - Requires specifying (K) in advance.
2. **Improvements**:
    - Use the **K-Means++** algorithm to initialize centroids smartly.

Below is a basic tutorial to implement the **K-Nearest Neighbors (KNN)** algorithm from scratch in Python. KNN is a simple yet effective algorithm for classification and regression tasks.

https://www.youtube.com/watch?v=4b5d3muPQmA&ab_channel=StatQuestwithJoshStarmer

---

K-Nearest Neighbors (KNN)

https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning

1. What is KNN?

K-Nearest Neighbors is a supervised learning algorithm used for classification and regression. It classifies a data point based on the majority vote of its 'k' nearest neighbors. For regression, it predicts the average of the 'k' nearest neighbors.

2. Steps in KNN

1. Choose the number of neighbors, (k).
2. Compute the distance between the test data point and all training data points.
3. Sort the distances and select the (k) nearest neighbors.
4. For classification:
    - Take a majority vote among the (k) nearest neighbors.
    - Assign the most frequent class to the test data point.
5. For regression:
    - Calculate the mean value of the (k) nearest neighbors.

---

3. Implementation

3.1 Import Libraries

```
import numpy as np
from collections import Counter
```

3.2 Create the Distance Function

The most common distance metric is **Euclidean Distance**: [\text{Distance} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}]

```
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((np.array(point1) - np.array(point2)) ** 2))
```

3.3 Define the KNN Class

```
class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
```

```
        """
        Store the training data.
        """
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        """
        Predict the labels for the given input data.
        """
        predictions = [self._predict(x) for x in X]
        return np.array(predictions)

    def _predict(self, x):
        """
        Predict the label for a single data point.
        """
        # Calculate distances between x and all training points
        distances = [euclidean_distance(x, x_train) for x_train in self.X_trai

        # Get the indices of the k nearest neighbors
        k_indices = np.argsort(distances)[:self.k]

        # Get the labels of the k nearest neighbors
        k_neighbor_labels = [self.y_train[i] for i in k_indices]

        # Majority vote for classification
        most_common = Counter(k_neighbor_labels).most_common(1)
        return most_common[0][0]
```

---

4. Testing the KNN Class

4.1 Create a Simple Dataset

```
from sklearn.model_selection import train_test_split

# Example dataset
X = [[1, 2], [2, 3], [3, 1], [6, 5], [7, 7], [8, 6]]
y = [0, 0, 0, 1, 1, 1]  # Binary classification

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
```

4.2 Train and Test the KNN Classifier

```
# Instantiate the KNN model
knn = KNN(k=3)

# Fit the model with training data
knn.fit(X_train, y_train)

# Predict on the test data
predictions = knn.predict(X_test)

# Print the results
print("Predictions:", predictions)
print("Actual labels:", y_test)
```

---

5. Evaluate the Model

5.1 Accuracy

```
accuracy = np.mean(predictions == y_test)
print("Accuracy:", accuracy)
```

---

6. Customizations

- **Change Distance Metric:** Replace `euclidean_distance` with other metrics like Manhattan distance.
- **Hyperparameter (k):** Experiment with different values of (k).
- **Use for Regression:** Replace the voting mechanism with the mean of the (k)-nearest neighbors.

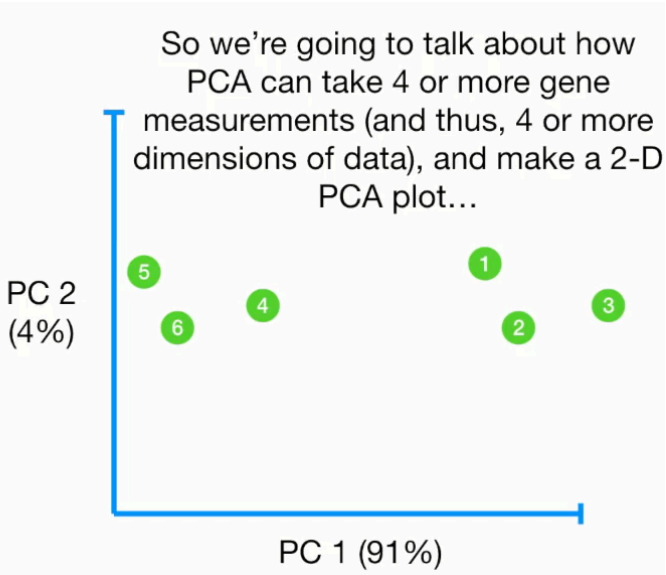
---

7. Visualizing the Results

(Optional) You can use `matplotlib` to visualize the decision boundary of KNN if your dataset has two features.

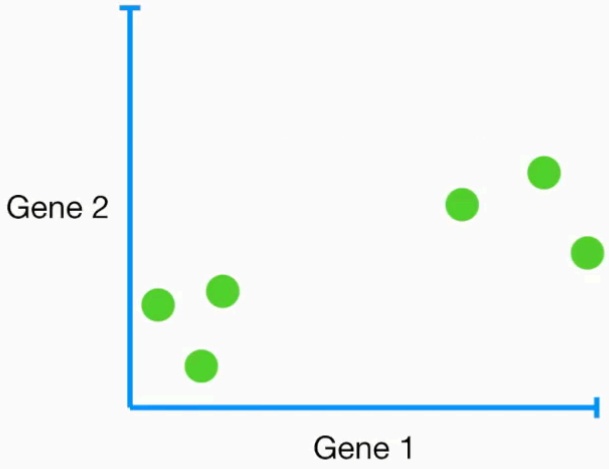
PCA

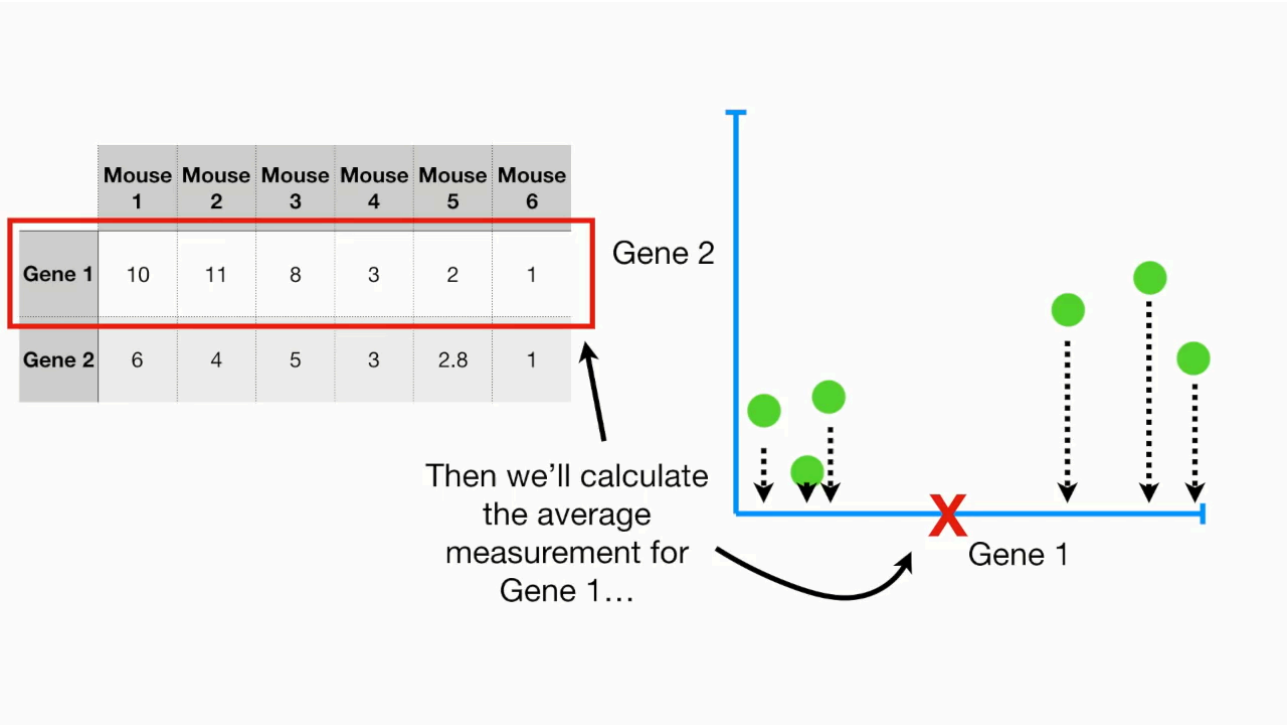
	Mouse 1	Mouse 2	Mouse 3	Mouse 4	Mouse 5	Mouse 6
Gene 1	10	11	8	3	2	1
Gene 2	6	4	5	3	2.8	1
Gene 3	12	9	10	2.5	1.3	2
Gene 4	5	7	6	2	4	7



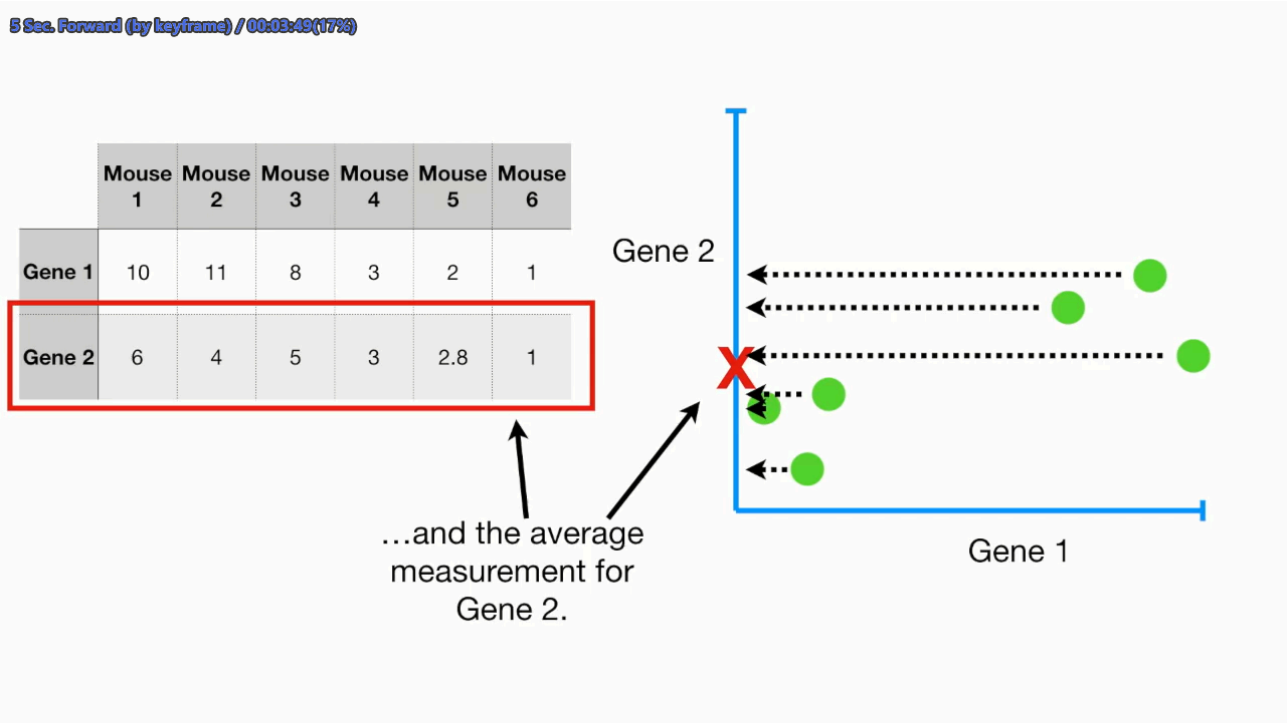
	Mouse 1	Mouse 2	Mouse 3	Mouse 4	Mouse 5	Mouse 6
Gene 1	10	11	8	3	2	1
Gene 2	6	4	5	3	2.8	1

We'll start by plotting the data…

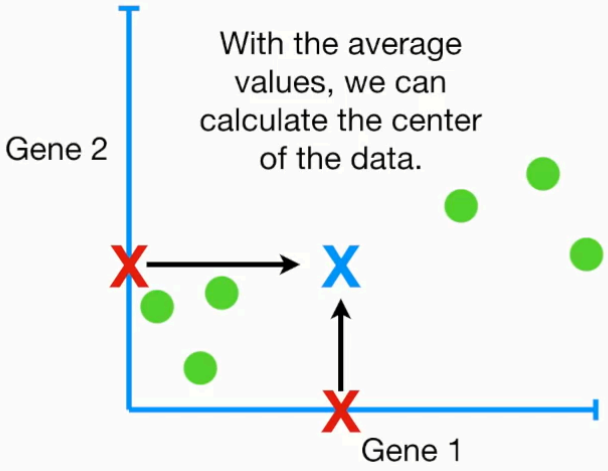


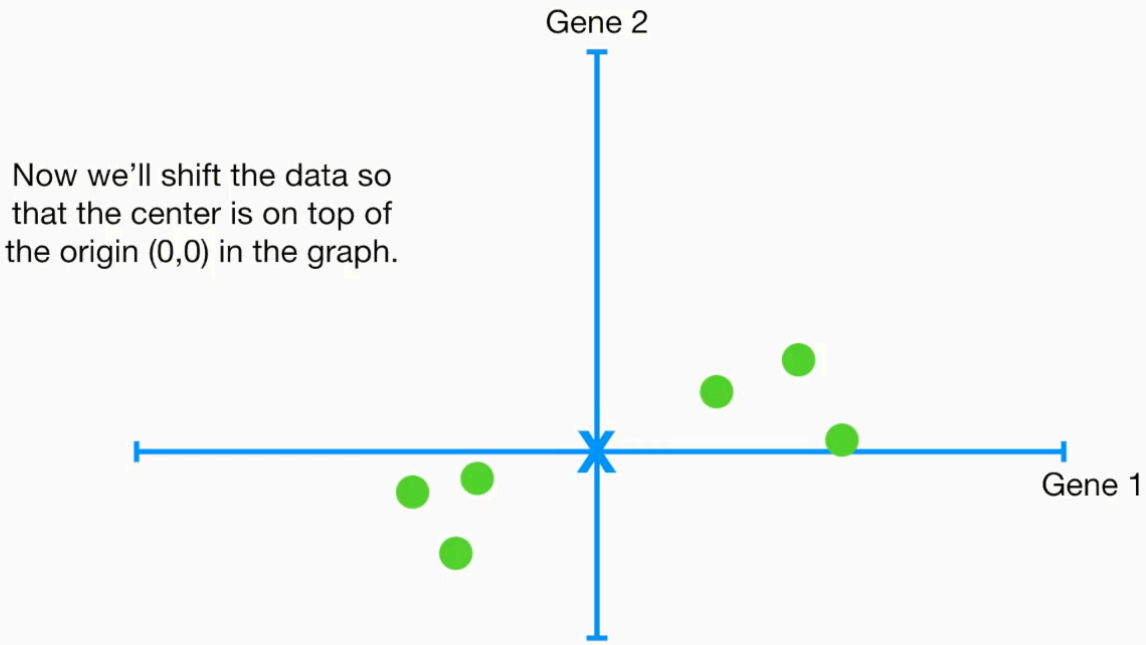


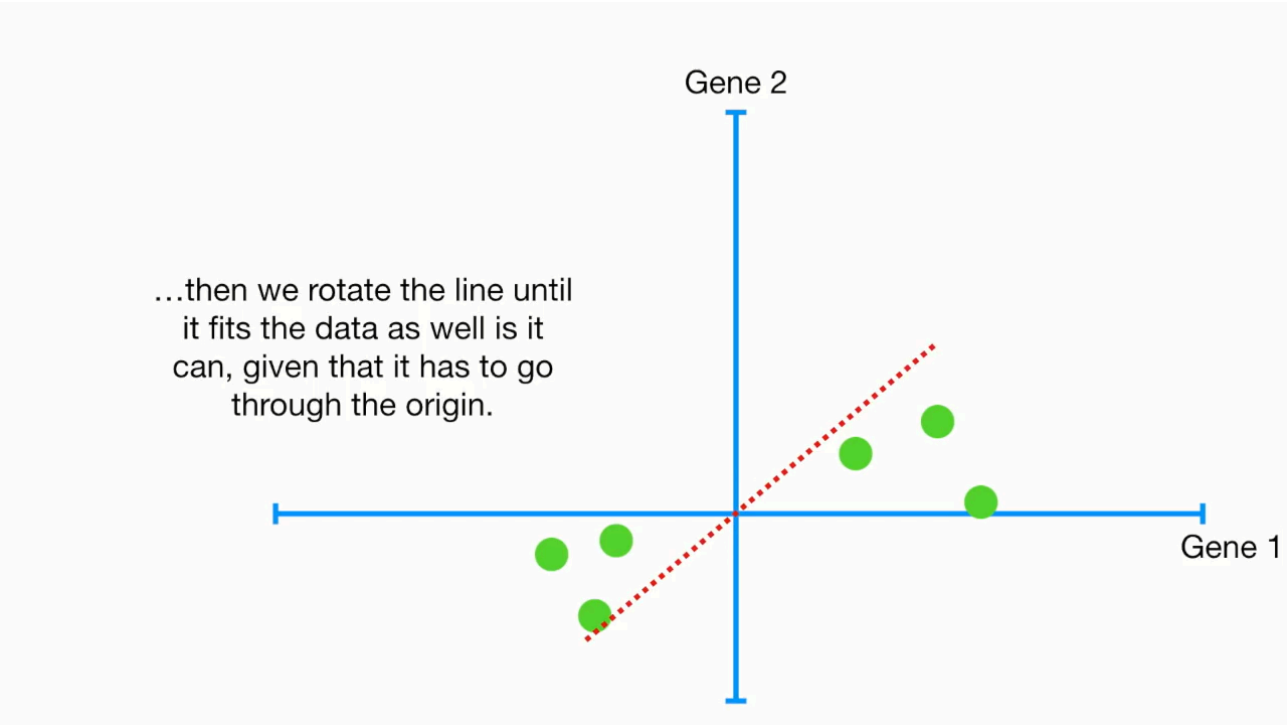
5 Sec. Forward (by keyframe) / 00:03:49(17%)

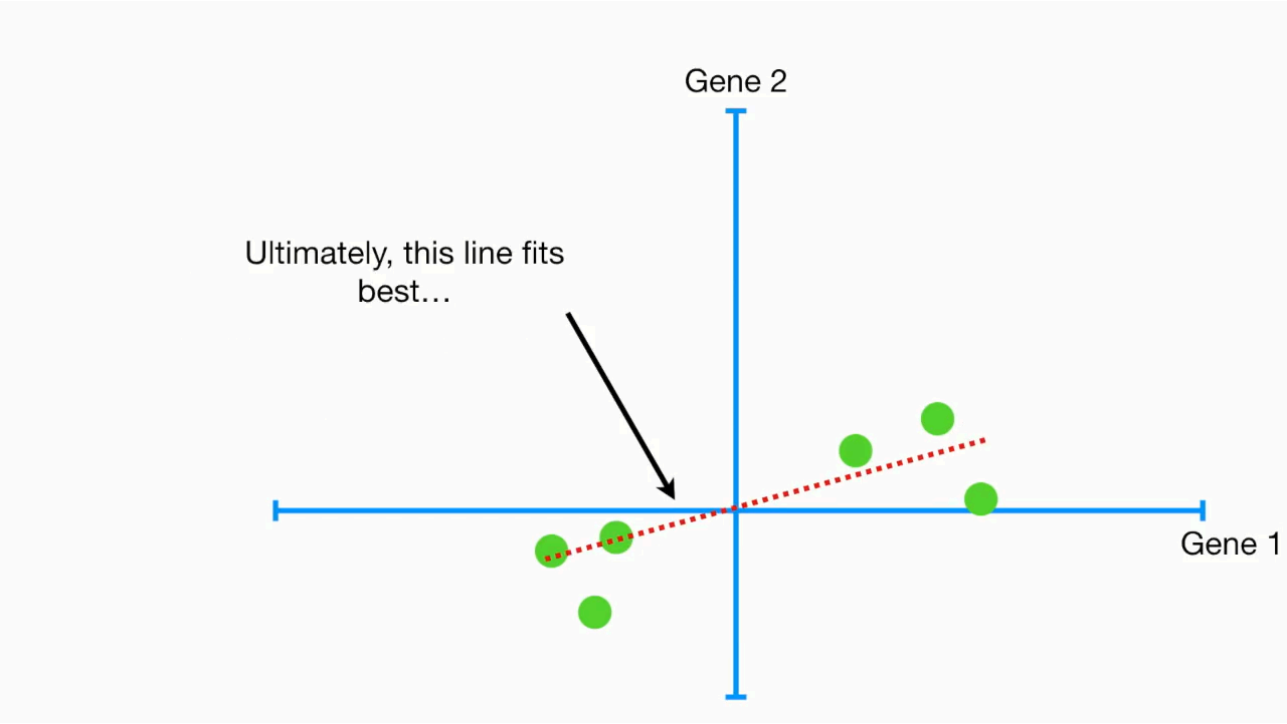


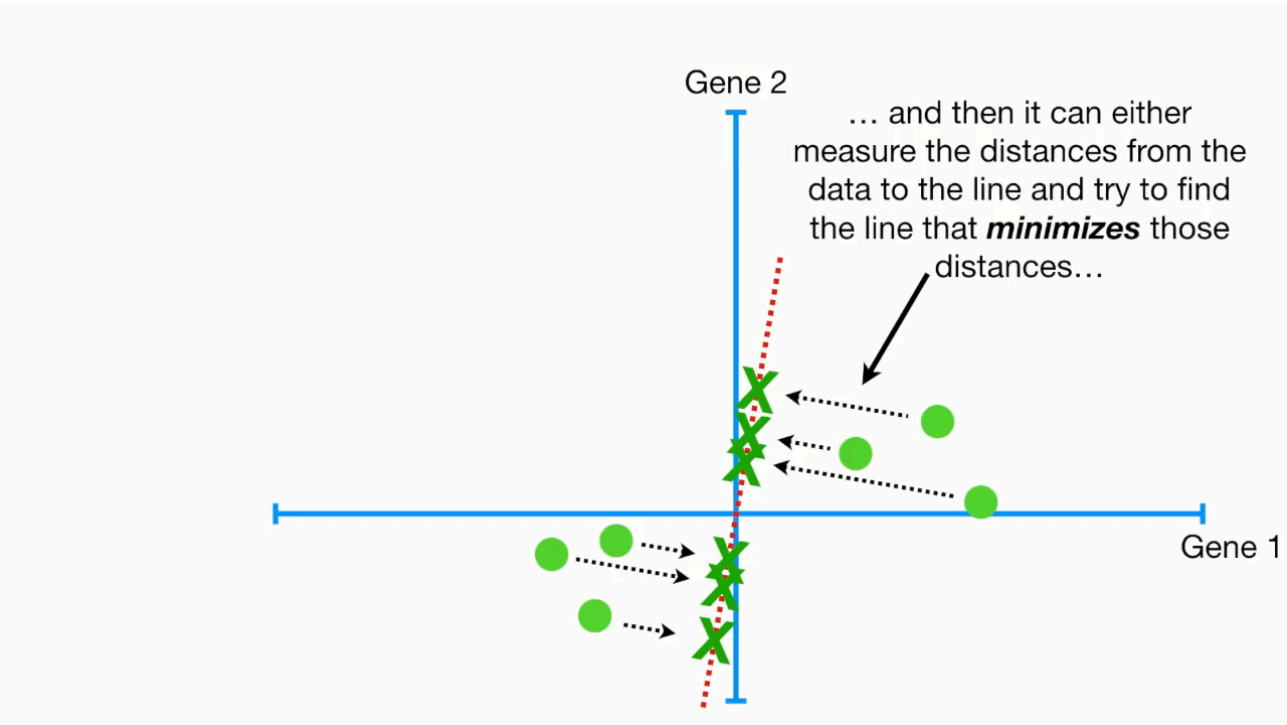
	Mouse 1	Mouse 2	Mouse 3	Mouse 4	Mouse 5	Mouse 6
Gene 1	10	11	8	3	2	1
Gene 2	6	4	5	3	2.8	1

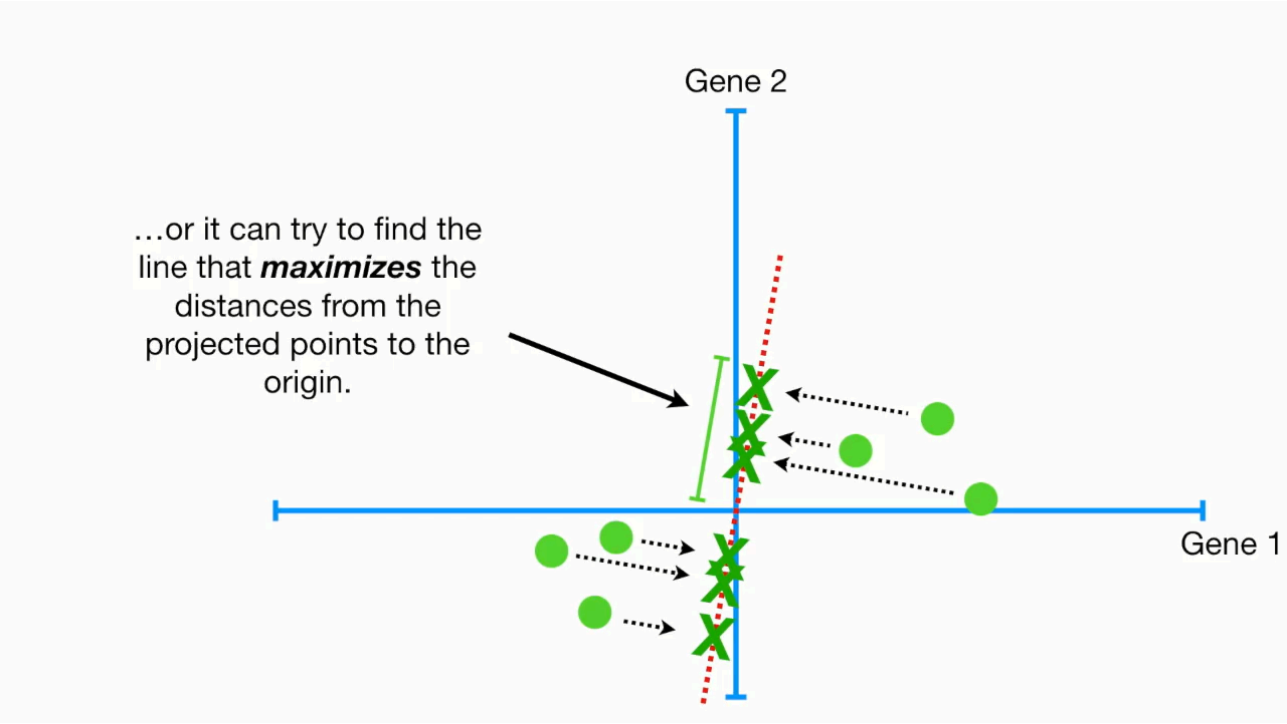


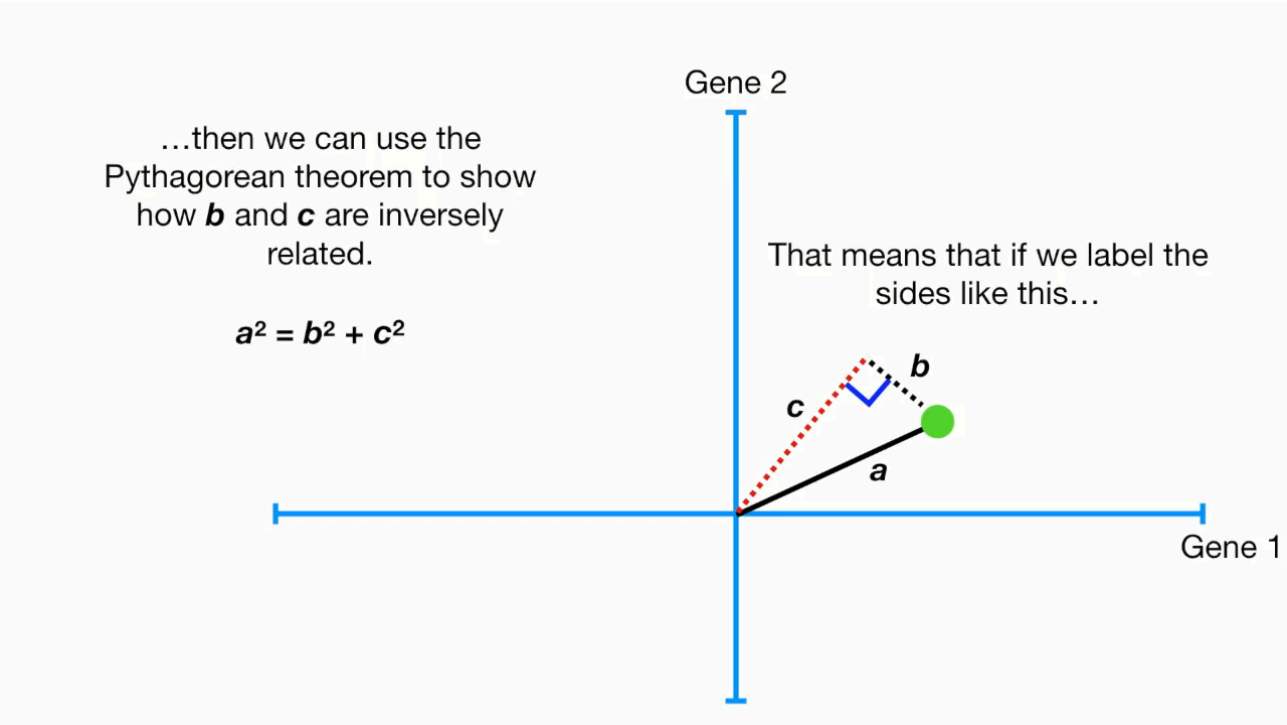


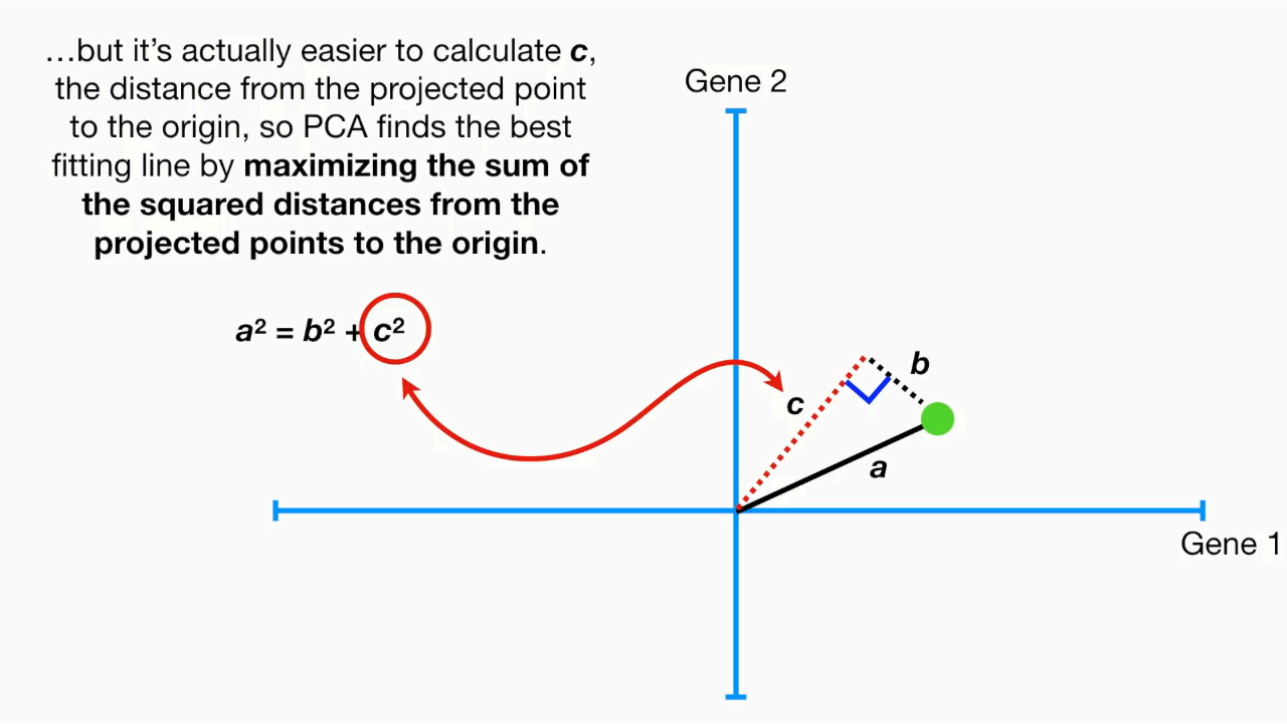

Gene 2
…then we rotate the line until it fits the data as well is it can, given that it has to go through the origin.
Gene 1

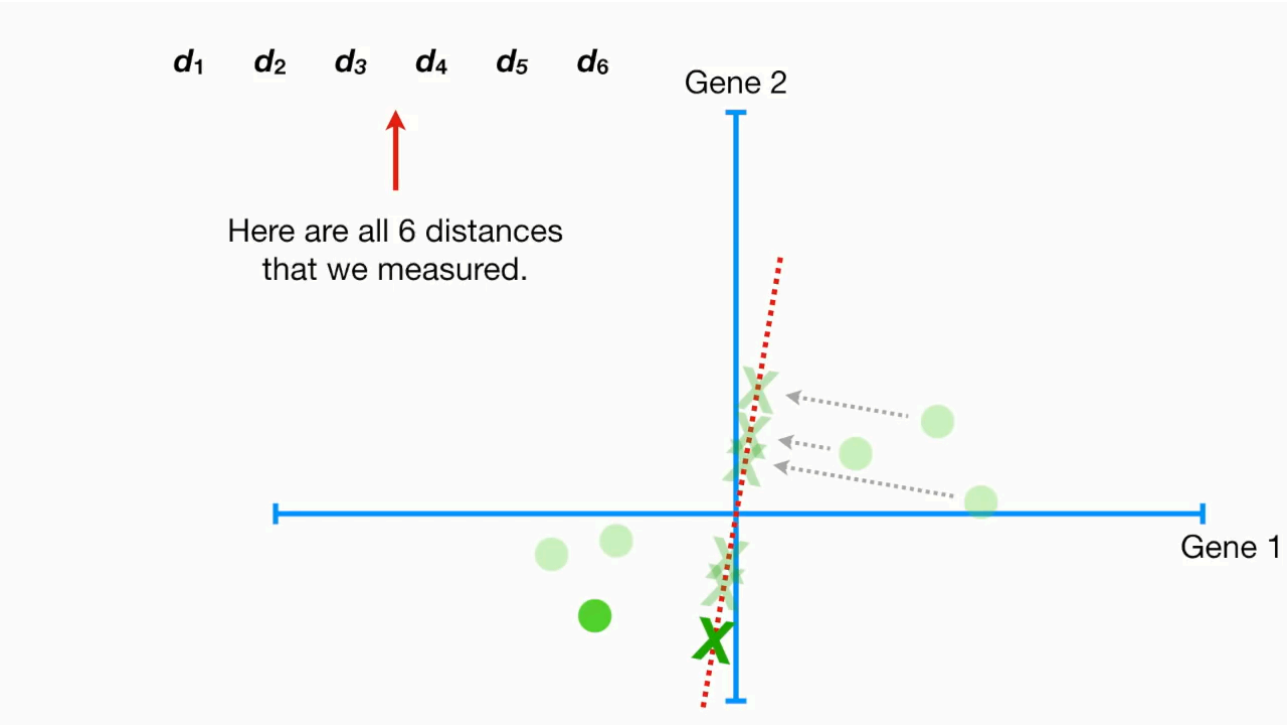

Gene 2
Ultimately, this line fits best…
Gene 1

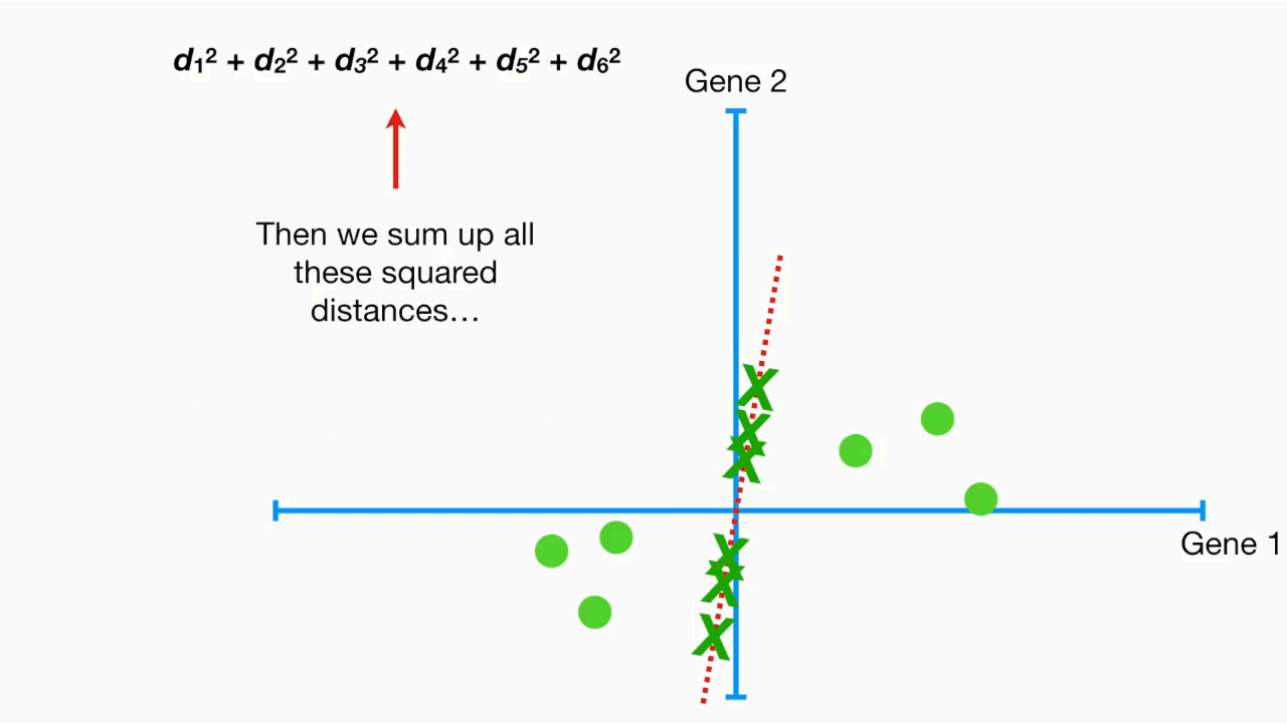

Gene 2
… and then it can either measure the distances from the data to the line and try to find the line that ***minimizes*** those distances…
Gene 1

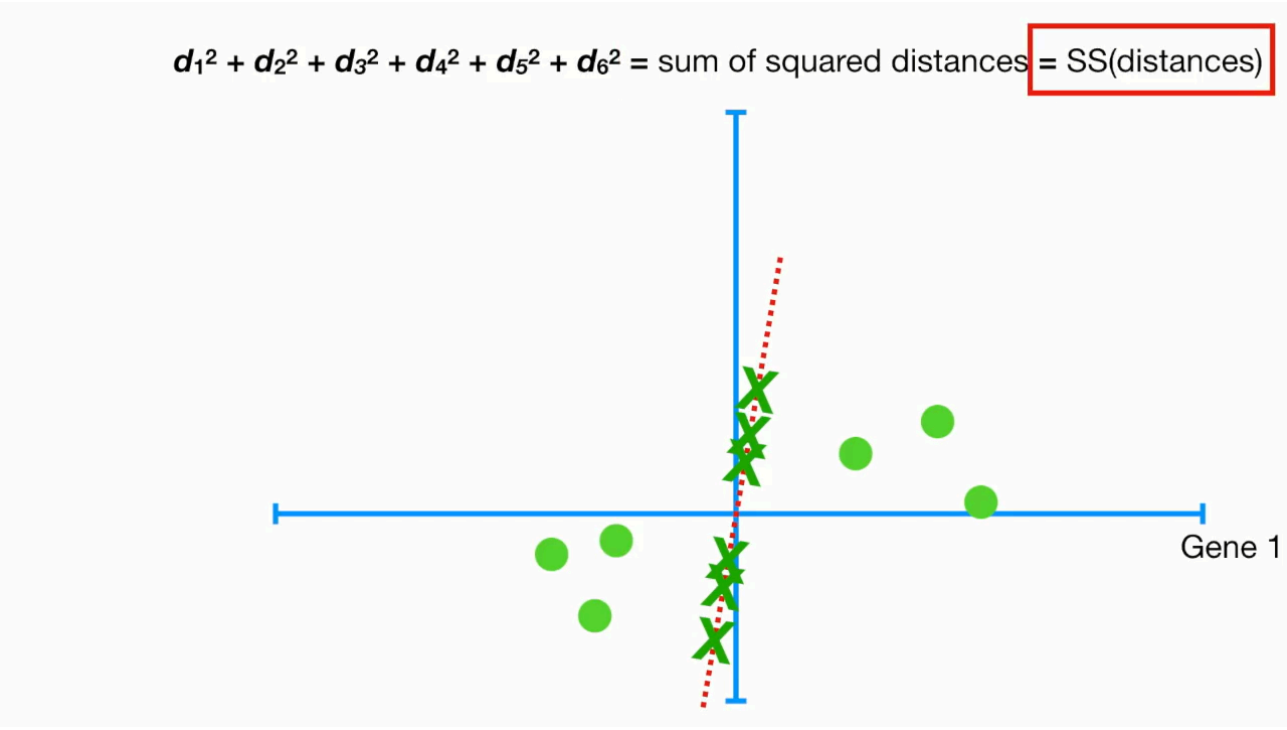

Gene 2
…or it can try to find the line that ***maximizes*** the distances from the projected points to the origin.
Gene 1

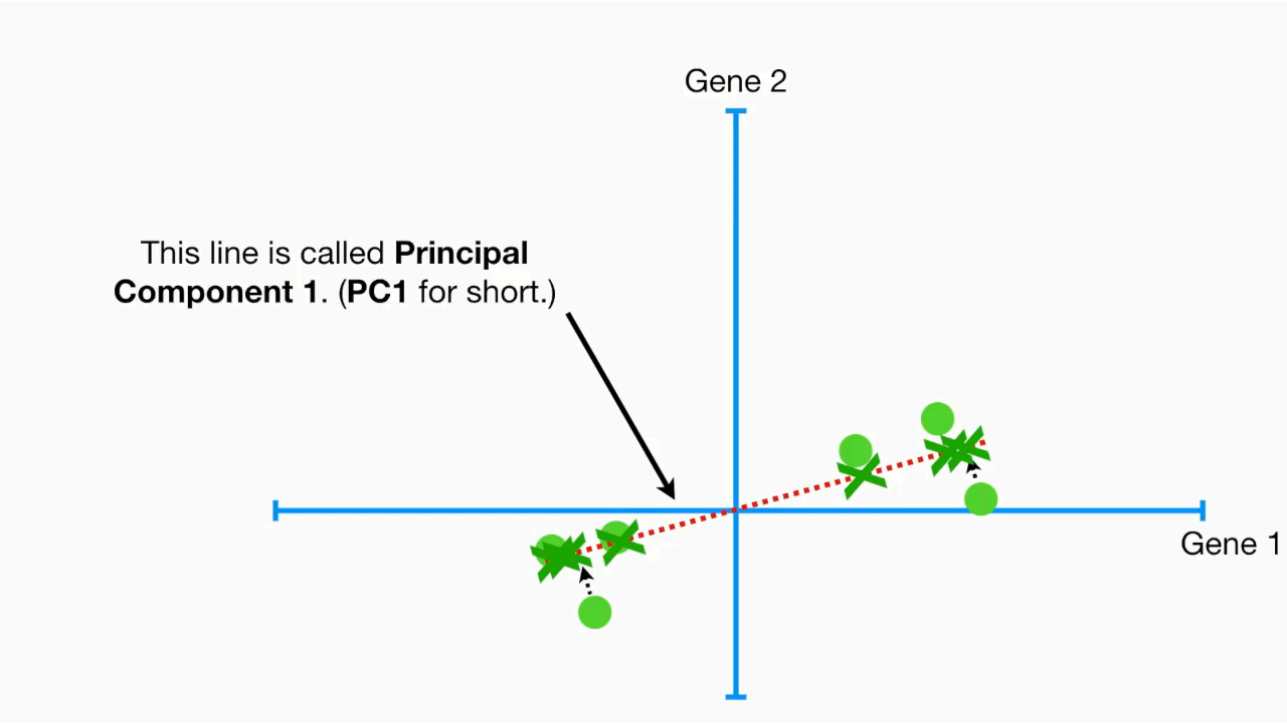

…then we can use the Pythagorean theorem to show how b and c are inversely related.
a² = b² + c²
Gene 2
That means that if we label the sides like this…
b
c
a
Gene 1

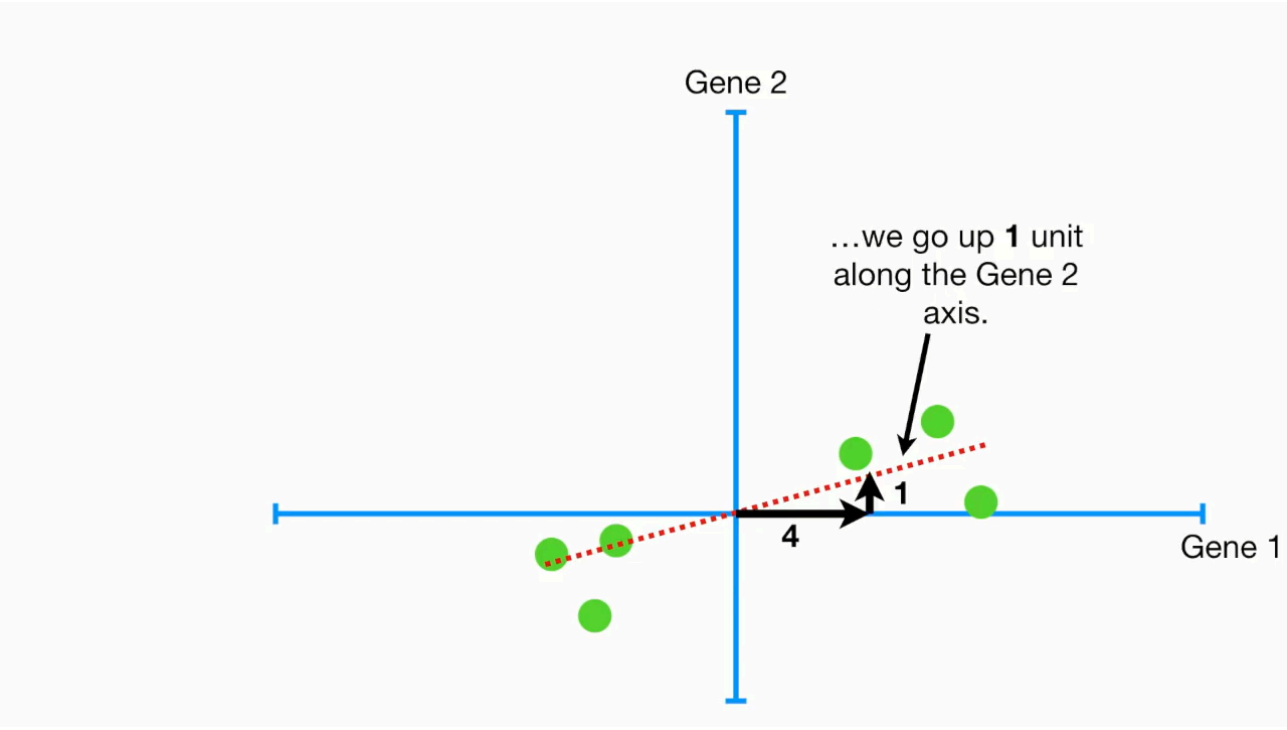

…but it's actually easier to calculate c, the distance from the projected point to the origin, so PCA finds the best fitting line by maximizing the sum of the squared distances from the projected points to the origin.
a² = b² + c²
Gene 2
b
c
a
Gene 1

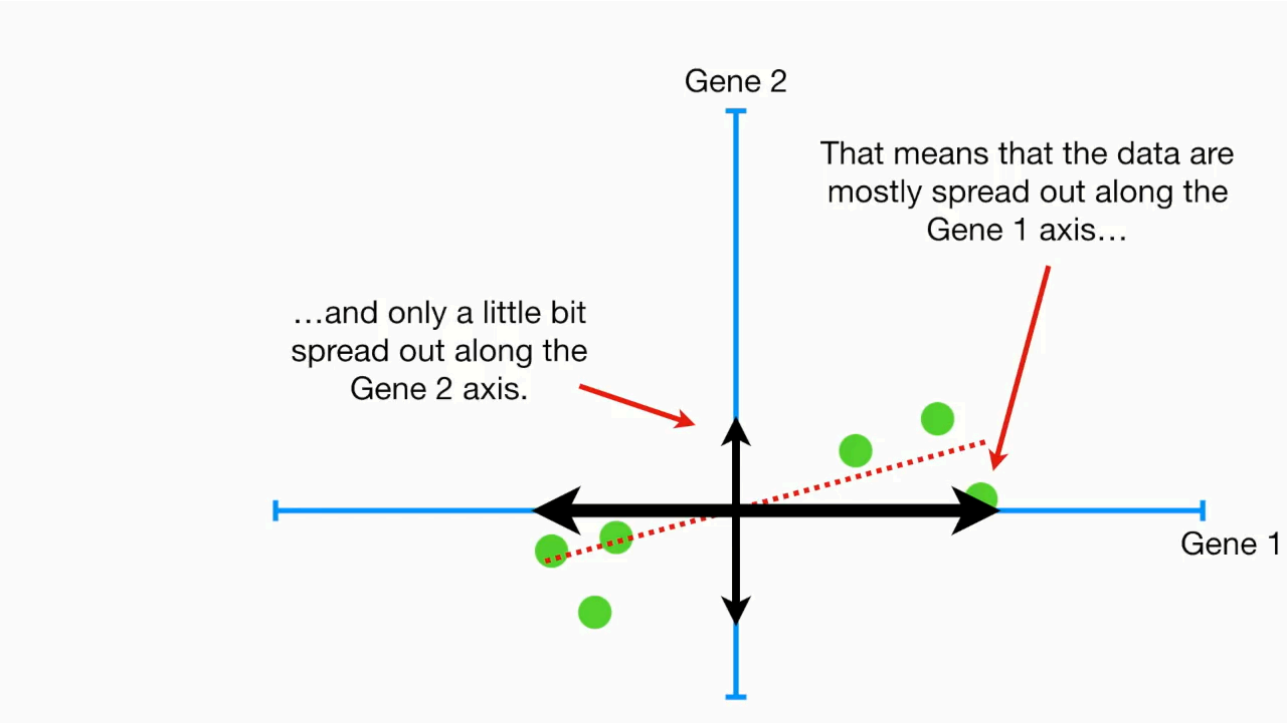

d_1 d_2 d_3 d_4 d_5 d_6
Here are all 6 distances that we measured.
Gene 2
Gene 1

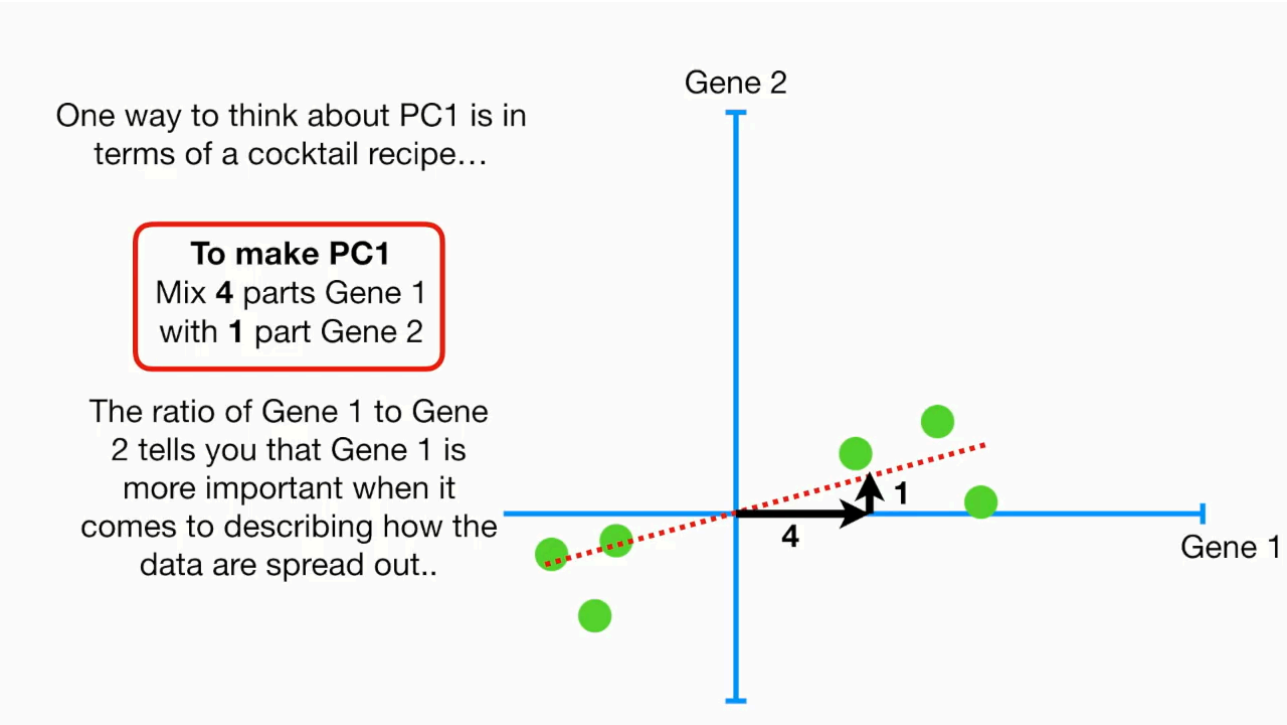

$d_1^2 + d_2^2 + d_3^2 + d_4^2 + d_5^2 + d_6^2$
Then we sum up all these squared distances…
Gene 2
Gene 1

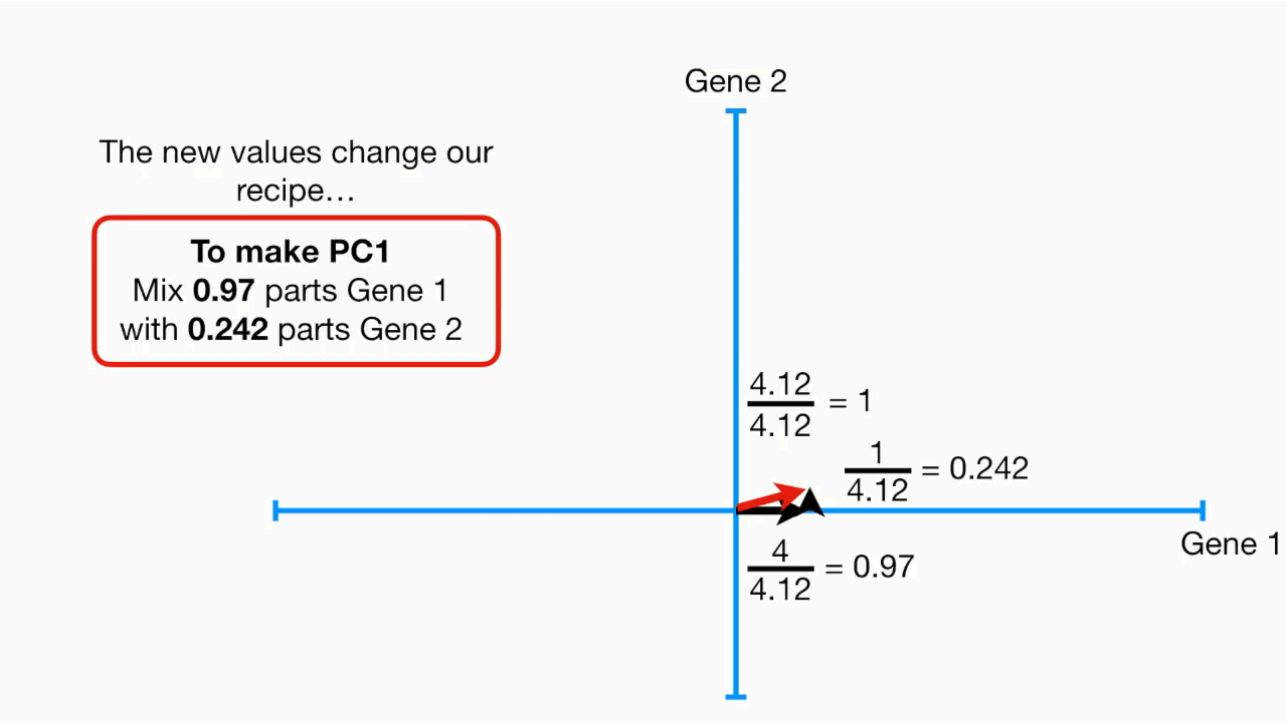

$d_1^2 + d_2^2 + d_3^2 + d_4^2 + d_5^2 + d_6^2$ = sum of squared distances = SS(distances)
Gene 1

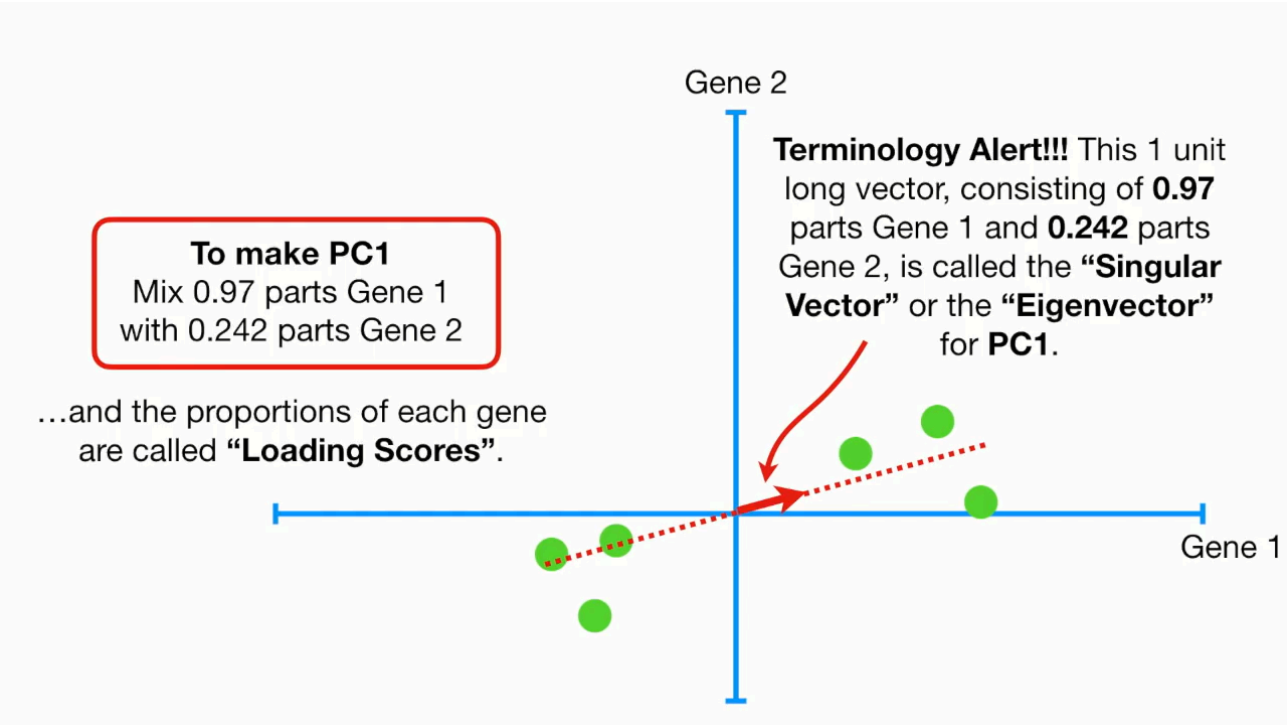

Gene 2
This line is called **Principal Component 1**. (**PC1** for short.)
Gene 1

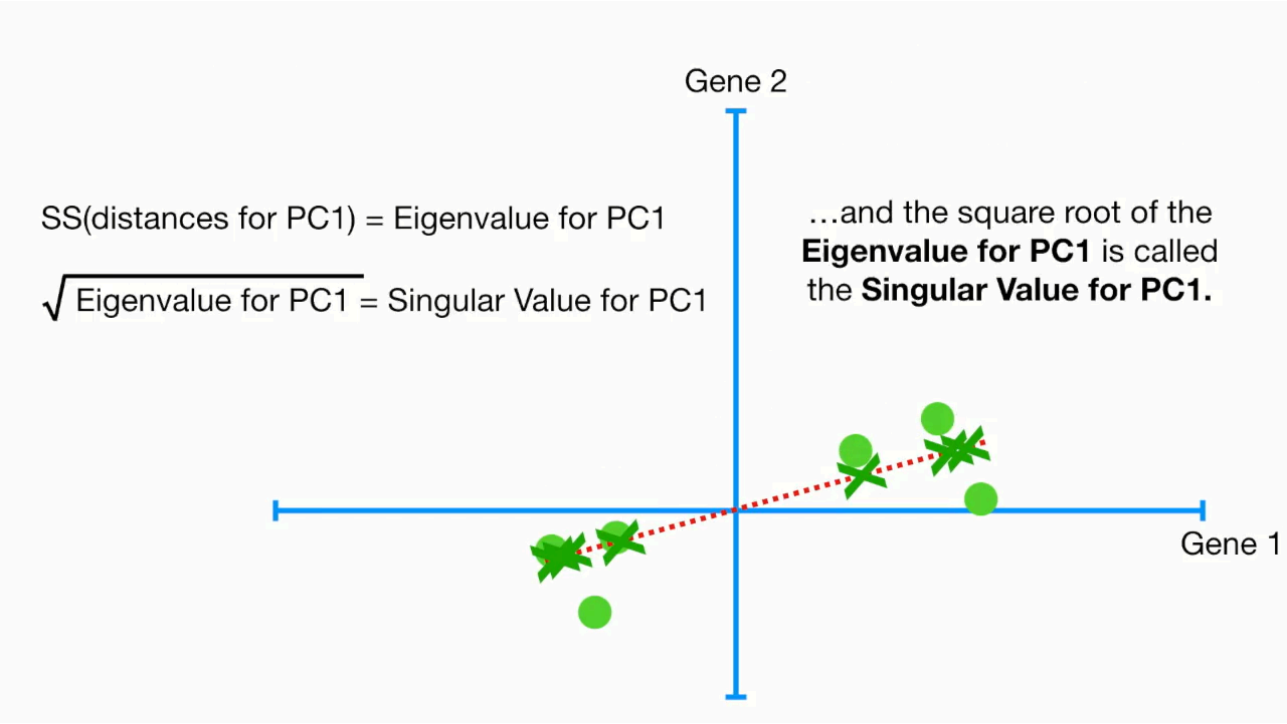

Gene 2
…we go up 1 unit along the Gene 2 axis.
1
4
Gene 1

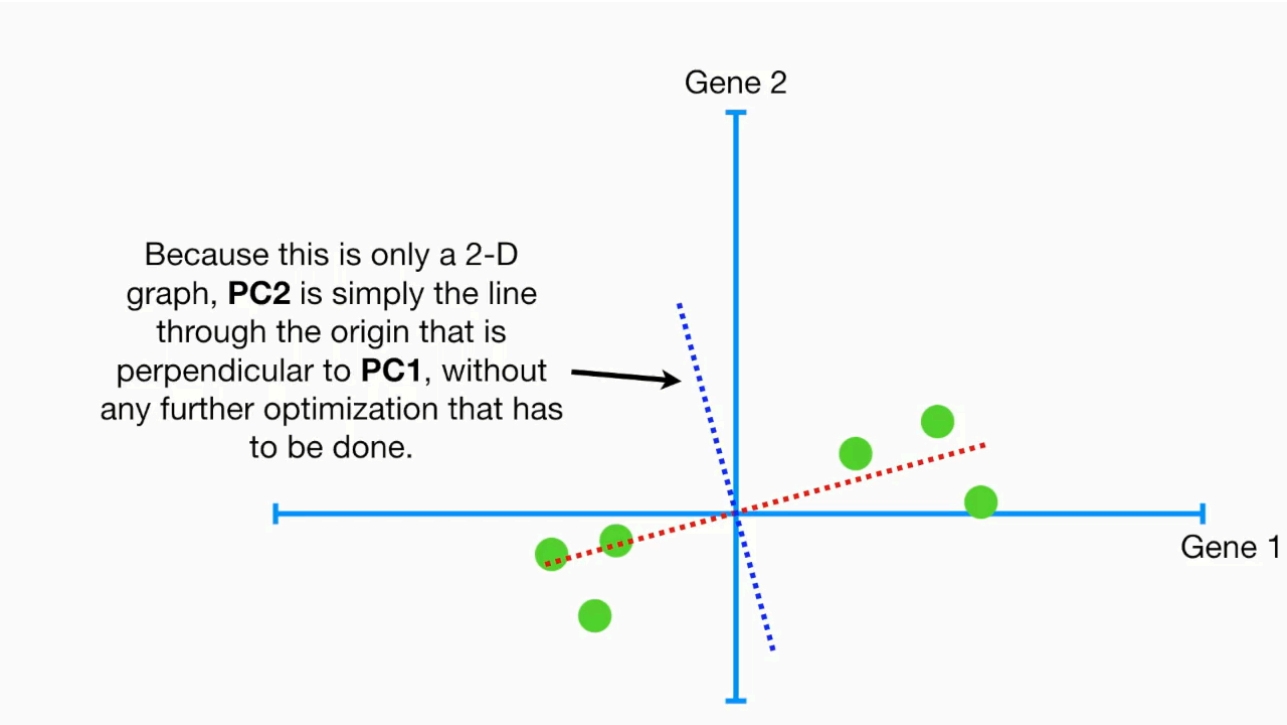

Gene 2
That means that the data are mostly spread out along the Gene 1 axis…
…and only a little bit spread out along the Gene 2 axis.
Gene 1

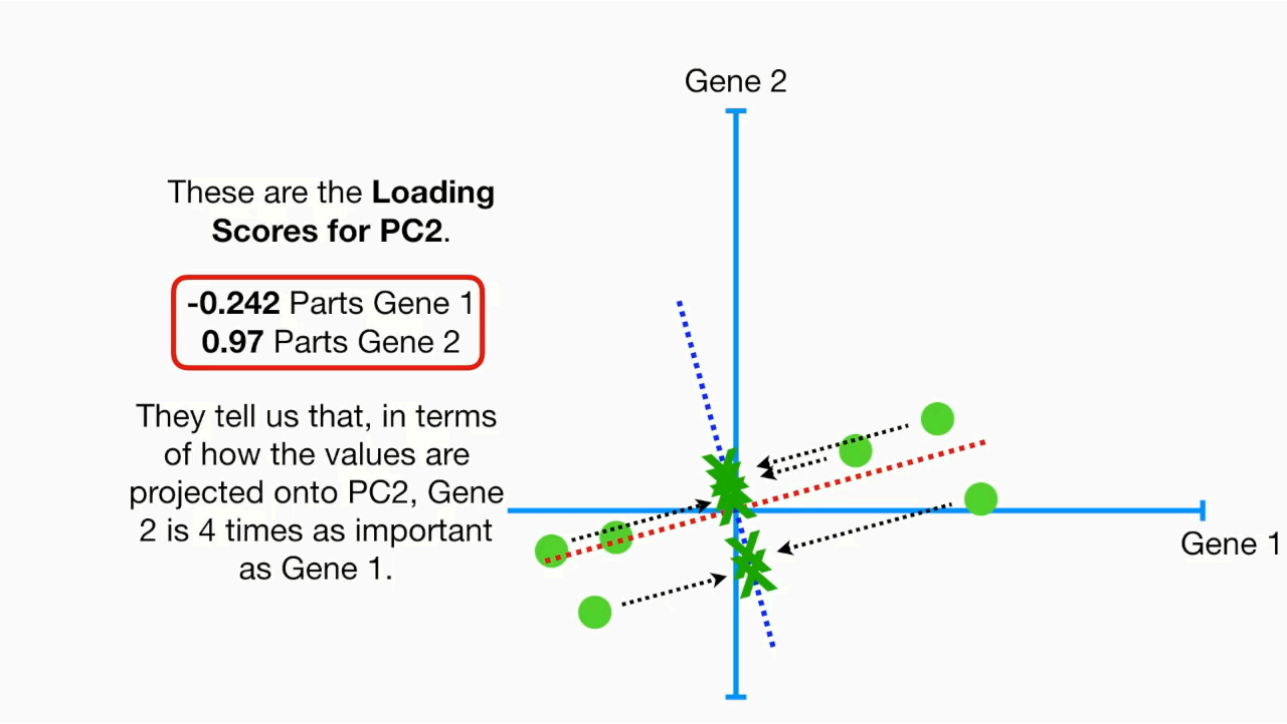

One way to think about PC1 is in terms of a cocktail recipe…
To make PC1
Mix 4 parts Gene 1 with 1 part Gene 2
The ratio of Gene 1 to Gene 2 tells you that Gene 1 is more important when it comes to describing how the data are spread out..
Gene 2
1
4
Gene 1

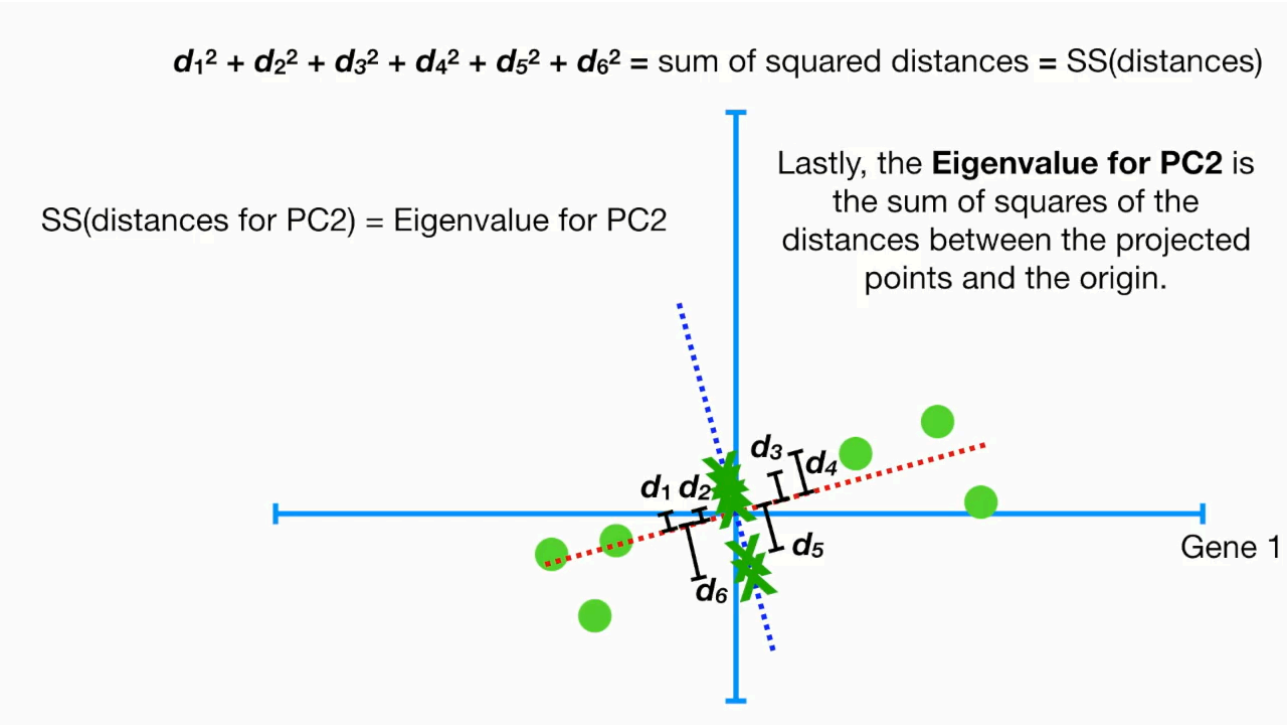

The new values change our recipe…
To make PC1
Mix 0.97 parts Gene 1 with 0.242 parts Gene 2
Gene 2
4.12/4.12 = 1
1/4.12 = 0.242
4/4.12 = 0.97
Gene 1

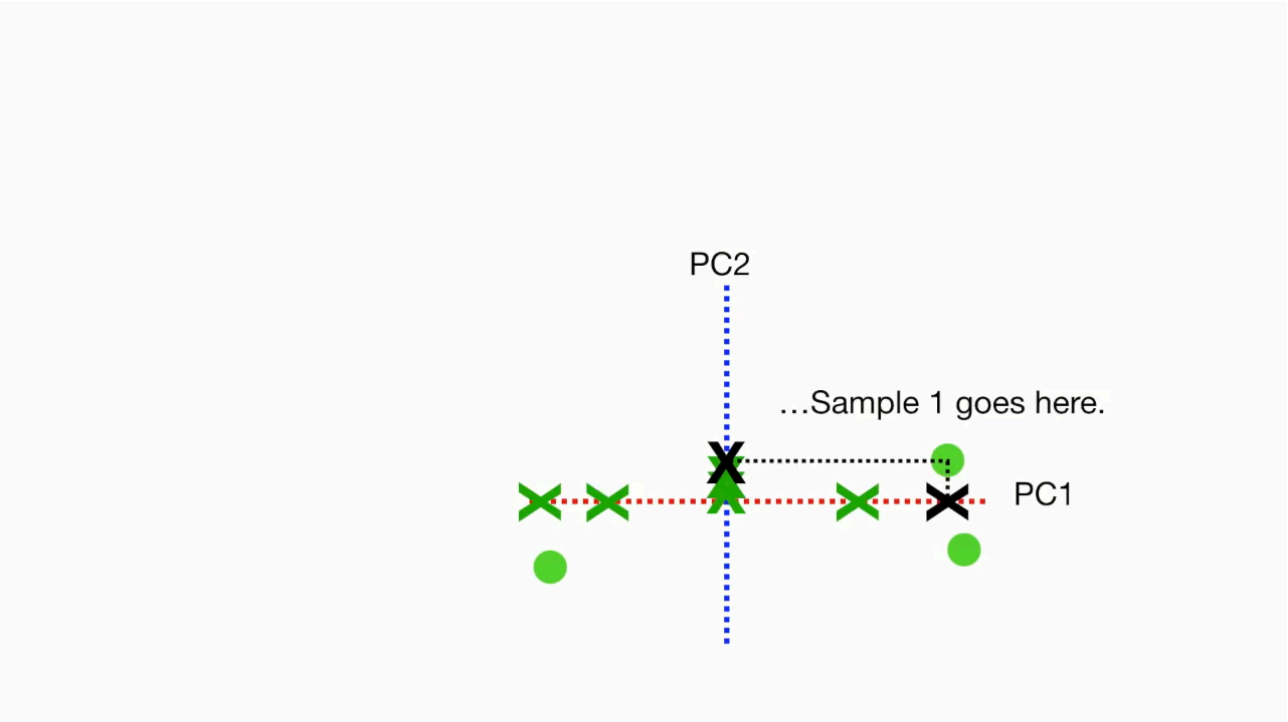

Gene 2
Terminology Alert!!! This 1 unit long vector, consisting of 0.97 parts Gene 1 and 0.242 parts Gene 2, is called the "Singular Vector" or the "Eigenvector" for PC1.
To make PC1
Mix 0.97 parts Gene 1
with 0.242 parts Gene 2
…and the proportions of each gene are called "Loading Scores".
Gene 1


Gene 2
SS(distances for PC1) = Eigenvalue for PC1
√Eigenvalue for PC1 = Singular Value for PC1
…and the square root of the Eigenvalue for PC1 is called the Singular Value for PC1.
Gene 1


Gene 2
Because this is only a 2-D graph, **PC2** is simply the line through the origin that is perpendicular to **PC1**, without any further optimization that has to be done.
Gene 1

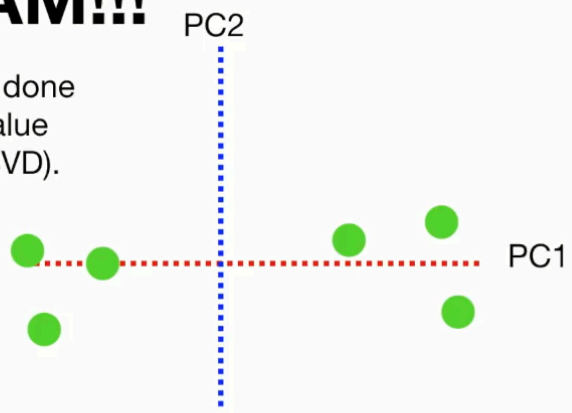

Gene 2
These are the **Loading Scores for PC2**.
-0.242 Parts Gene 1
0.97 Parts Gene 2
They tell us that, in terms of how the values are projected onto PC2, Gene 2 is 4 times as important as Gene 1.
Gene 1


$d_1^2 + d_2^2 + d_3^2 + d_4^2 + d_5^2 + d_6^2$ = sum of squared distances = SS(distances)
SS(distances for PC2) = Eigenvalue for PC2
Lastly, the **Eigenvalue for PC2** is the sum of squares of the distances between the projected points and the origin.
d_1 d_2
d_3
d_4
d_5
d_6
Gene 1


PC2
…Sample 1 goes here.
PC1

Double BAM!!!

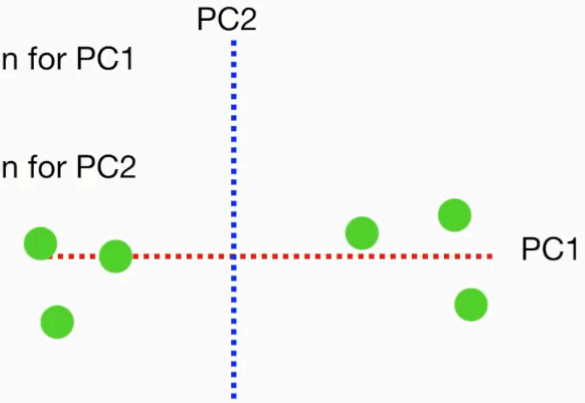
That's how PCA is done using Singular Value Decomposition (SVD).



We can convert them into variation around the origin (0, 0) by dividing by the sample size minus 1 (i.e. n - 1).

$$\frac{\text{SS(distances for PC1)}}{n - 1} = \text{Variation for PC1}$$

$$\frac{\text{SS(distances for PC2)}}{n - 1} = \text{Variation for PC2}$$

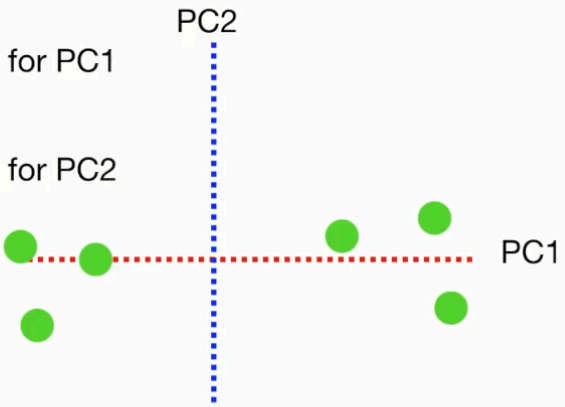


For the sake of the example, imagine that the Variation for **PC1 = 15**, and the variation for **PC2 = 3**.

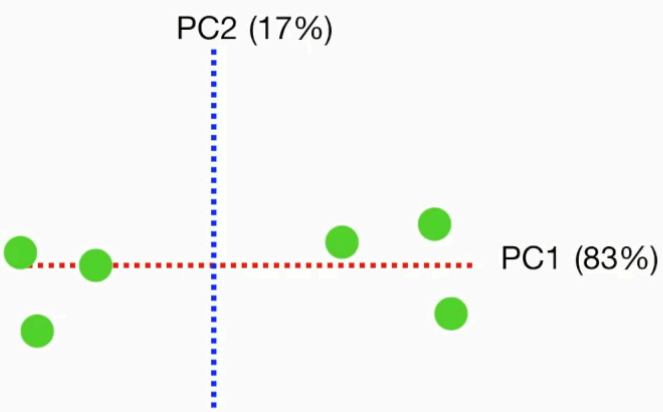
That means that the total variation around both PCs is **15 + 3 = 18**...

$$\frac{SS(\text{distances for PC1})}{n - 1} = \text{Variation for PC1}$$

$$\frac{SS(\text{distances for PC2})}{n - 1} = \text{Variation for PC2}$$



PC2 accounts for **3 / 18 = 0.17 = 17%** of the total variation around the PCs.



Background for PCA in math

- Covariance of two attributes:

$$\text{cov}(A_1, A_2) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{(n-1)}$$

$$\begin{pmatrix} \text{cov}(H,H) & \text{cov}(H,M) \\ \text{cov}(M,H) & \text{cov}(M,M) \end{pmatrix}$$

$$= \begin{pmatrix} \text{var}(H) & 104.5 \\ 104.5 & \text{var}(M) \end{pmatrix}$$

$$= \begin{pmatrix} 47.7 & 104.5 \\ 104.5 & 370 \end{pmatrix}$$

1. Given original data set S = $\{\mathbf{x}^1, ..., \mathbf{x}^k\}$, produce new set by subtracting the mean of attribute A_i from each x_i.

Data =

x	y
2.5	2.4
0.5	0.7
2.2	2.9
1.9	2.2
3.1	3.0
2.3	2.7
2	1.6
1	1.1
1.5	1.6
1.1	0.9
Mean: 1.81	**1.91**

DataAdjust =

x	y
.69	.49
-1.31	-1.21
.39	.99
.09	.29
1.29	1.09
.49	.79
.19	-.31
-.81	-.81
-.31	-.31
-.71	-1.01
Mean: **0**	**0**

2. Calculate the covariance matrix:

$$cov = \begin{array}{c} \\ \mathbf{x} \\ \mathbf{y} \end{array} \begin{array}{c} \begin{array}{cc} \mathbf{x} & \mathbf{y} \end{array} \\ \begin{pmatrix} .616555556 & .615444444 \\ .615444444 & .716555556 \end{pmatrix} \end{array}$$

Step 2: Covariance Matrix

The covariance matrix is given as:

$$\text{cov} = \begin{pmatrix} 0.616555556 & 0.615444444 \\ 0.615444444 & 0.716555556 \end{pmatrix}$$

This symmetric matrix represents the covariances between variables x and y. The diagonal elements are the variances ($\text{Var}(x)$ and $\text{Var}(y)$), and the off-diagonal elements are the covariances ($\text{Cov}(x, y)$).

Step 3: Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are computed from the covariance matrix by solving the eigenvalue equation:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

Where:

- $\mathbf{A}$ is the covariance matrix,
- λ is an eigenvalue,
- $\mathbf{v}$ is the corresponding eigenvector.

1. **Find the Eigenvalues**:
   The eigenvalues are solutions to the characteristic equation:

   $$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

   Here, $\mathbf{I}$ is the identity matrix. For this covariance matrix:

   $$\det\left(\begin{pmatrix} 0.616555556 - \lambda & 0.615444444 \\ 0.615444444 & 0.716555556 - \lambda \end{pmatrix}\right) = 0$$

   Expanding this determinant gives a quadratic equation in λ, which can be solved to find the eigenvalues:

   $$\lambda_1 = 0.0490833989, \quad \lambda_2 = 1.28402771$$

Let's compute the eigenvalues explicitly by expanding the determinant.

Step 1: Write the eigenvalue equation

The eigenvalues λ are the roots of:

$$\det\left(\mathbf{A} - \lambda\mathbf{I}\right) = 0$$

Substitute $\mathbf{A}$ (the covariance matrix) and $\mathbf{I}$ (the identity matrix):

$$\mathbf{A} - \lambda\mathbf{I} = \begin{pmatrix} 0.616555556 - \lambda & 0.615444444 \\ 0.615444444 & 0.716555556 - \lambda \end{pmatrix}$$

The determinant is:

$$\det\left(\begin{pmatrix} 0.616555556 - \lambda & 0.615444444 \\ 0.615444444 & 0.716555556 - \lambda \end{pmatrix}\right) = 0$$

Step 2: Expand the determinant

The determinant of a 2×2 matrix is given by:

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

Substitute the elements:

$$\det = (0.616555556 - \lambda)(0.716555556 - \lambda) - (0.615444444)(0.615444444)$$

Expand the terms:

$$\det = (0.616555556)(0.716555556) - (0.616555556 + 0.716555556)\lambda + \lambda^2 - (0.615444444)^2$$

Simplify the constants:

- $(0.616555556)(0.716555556) = 0.441287056$,
- $(0.615444444)^2 = 0.378742774$.

So:

$$\det = 0.441287056 - (0.616555556 + 0.716555556)\lambda + \lambda^2 - 0.378742774$$

$$\det = \lambda^2 - 1.333111112\lambda + (0.441287056 - 0.378742774)$$

Simplify further:

$$\det = \lambda^2 - 1.333111112\lambda + 0.062544282 = 0$$

Step 3: Solve the quadratic equation

The quadratic equation is:

$$\lambda^2 - 1.333111112\lambda + 0.062544282 = 0$$

Final Eigenvalues:

$$\lambda_1 = 1.28402771, \quad \lambda_2 = 0.0490833989$$

2. **Find the Eigenvectors**:

To compute the eigenvectors corresponding to the eigenvalues, we solve the equation:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

or equivalently:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0$$

Here $\mathbf{A}$ is the covariance matrix, λ is an eigenvalue, and $\mathbf{v}$ is the eigenvector.

Step 1: Eigenvector for $\lambda_1 = 1.28402771$

Substitute $\lambda_1 = 1.28402771$ into $\mathbf{A} - \lambda_1 \mathbf{I}$:

$$\mathbf{A} - \lambda_1 \mathbf{I} = \begin{pmatrix} 0.616555556 - 1.28402771 & 0.615444444 \\ 0.615444444 & 0.716555556 - 1.28402771 \end{pmatrix}$$

Simplify:

$$\mathbf{A} - \lambda_1 \mathbf{I} = \begin{pmatrix} -0.667472154 & 0.615444444 \\ 0.615444444 & -0.567472154 \end{pmatrix}$$

We solve:

$$\begin{pmatrix} -0.667472154 & 0.615444444 \\ 0.615444444 & -0.567472154 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

This gives two equations:

1. $-0.667472154 v_1 + 0.615444444 v_2 = 0$,
2. $0.615444444 v_1 - 0.567472154 v_2 = 0$.

From the first equation:

$$v_2 = \frac{0.667472154}{0.615444444} v_1 \approx 1.0843 v_1$$

The eigenvector is:

$$\mathbf{v}_1 = \begin{pmatrix} 1 \\ 1.0843 \end{pmatrix}$$

Normalize $\mathbf{v}_1$ to make it a unit vector:

$$\|\mathbf{v}_1\| = \sqrt{1^2 + (1.0843)^2} \approx \sqrt{2.1767} \approx 1.4754$$

$$\mathbf{v}_1 = \frac{1}{1.4754} \begin{pmatrix} 1 \\ 1.0843 \end{pmatrix} \approx \begin{pmatrix} 0.7352 \\ 0.6779 \end{pmatrix}$$

Step 2: Eigenvector for $\lambda_2 = 0.0490833989$

Substitute $\lambda_2 = 0.0490833989$ into $\mathbf{A} - \lambda_2 \mathbf{I}$:

$$\mathbf{A} - \lambda_2 \mathbf{I} = \begin{pmatrix} 0.616555556 - 0.0490833989 & 0.615444444 \\ 0.615444444 & 0.716555556 - 0.0490833989 \end{pmatrix}$$

Simplify:

$$\mathbf{A} - \lambda_2 \mathbf{I} = \begin{pmatrix} 0.567472157 & 0.615444444 \\ 0.615444444 & 0.667472157 \end{pmatrix}$$

We solve:

$$\begin{pmatrix} 0.567472157 & 0.615444444 \\ 0.615444444 & 0.667472157 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

This gives two equations:

1. $0.567472157 v_1 + 0.615444444 v_2 = 0$,
2. $0.615444444 v_1 + 0.667472157 v_2 = 0$.

From the first equation:

$$v_2 = -\frac{0.567472157}{0.615444444} v_1 \approx -0.9222 v_1$$

The eigenvector is:

$$\mathbf{v}_2 = \begin{pmatrix} 1 \\ -0.9222 \end{pmatrix}$$

Normalize $\mathbf{v}_2$ to make it a unit vector:

$$\|\mathbf{v}_2\| = \sqrt{1^2 + (-0.9222)^2} \approx \sqrt{1.8504} \approx 1.3603$$

$$\mathbf{v}_2 = \frac{1}{1.3603} \begin{pmatrix} 1 \\ -0.9222 \end{pmatrix} \approx \begin{pmatrix} 0.7352 \\ -0.6779 \end{pmatrix}$$

Final Eigenvectors

The unit eigenvectors are:

$$\mathbf{v}_1 = \begin{pmatrix} 0.7352 \\ 0.6779 \end{pmatrix}, \quad \mathbf{v}_2 = \begin{pmatrix} 0.7352 \\ -0.6779 \end{pmatrix}$$

https://www.youtube.com/watch?v=FgakZw6K1QQ&pp=ygUNcGNhIHN0YXRxdWVzdA%3D%3D

Defination

Principal Component Analysis (PCA) is a widely used dimensionality reduction technique in machine learning. It reduces the number of features in a dataset while preserving as much information as possible. PCA achieves this by identifying new axes (principal components) that maximize variance.

---

1. Why PCA?

- **High-Dimensional Data**: High-dimensional datasets can lead to overfitting and computational inefficiency.
- **Noise Reduction**: PCA eliminates features that contribute minimal variance, reducing noise.
- **Visualization**: PCA is often used to project data into 2D or 3D for visualization.

---

2. Steps in PCA

1. **Standardize the Data**: Scale the data to have zero mean and unit variance.
2. **Compute Covariance Matrix**: Identify relationships between features.
3. **Perform Eigen Decomposition**: Obtain eigenvalues (variance explained) and eigenvectors (principal components).
4. **Sort Eigenvectors**: Rank them by corresponding eigenvalues in descending order.
5. **Project the Data**: Transform the data into the new feature space using top components.

---

3. PCA in Python

Step 1: Import Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

Step 2: Load the Dataset

Use any dataset. Here, we'll use the Iris dataset as an example:

```
from sklearn.datasets import load_iris

# Load dataset
data = load_iris()
X = data.data
y = data.target

# Convert to DataFrame
df = pd.DataFrame(X, columns=data.feature_names)
df['target'] = y
```

Step 3: Standardize the Features

```
# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Step 4: Apply PCA

```
# Apply PCA
pca = PCA(n_components=2)  # Reduce to 2 components
X_pca = pca.fit_transform(X_scaled)

# Variance explained by each component
print("Explained variance ratio:", pca.explained_variance_ratio_)
```

Step 5: Visualize Results

```
# Plot PCA projection
plt.figure(figsize=(8, 6))
for target in np.unique(y):
    plt.scatter(X_pca[y == target, 0], X_pca[y == target, 1], label=f'Class {t

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA of Iris Dataset')
plt.legend()
plt.show()
```

---

4. Interpreting Results

- **Explained Variance Ratio**: Indicates how much information (variance) each principal component captures.
- **Reduced Features**: Data is now represented by fewer dimensions, making models faster and less prone to overfitting.

---

5. Practical Tips

- **Standardize Features**: PCA is sensitive to feature scaling.
- **Choose n_components Wisely**: Use `explained_variance_ratio_` to decide the number of components.
- **Interpretability**: PCA is unsupervised; ensure reduced features align with the problem's requirements.

---

6. PCA for High-Dimensional Data

For datasets with hundreds of features, PCA can drastically reduce dimensions:

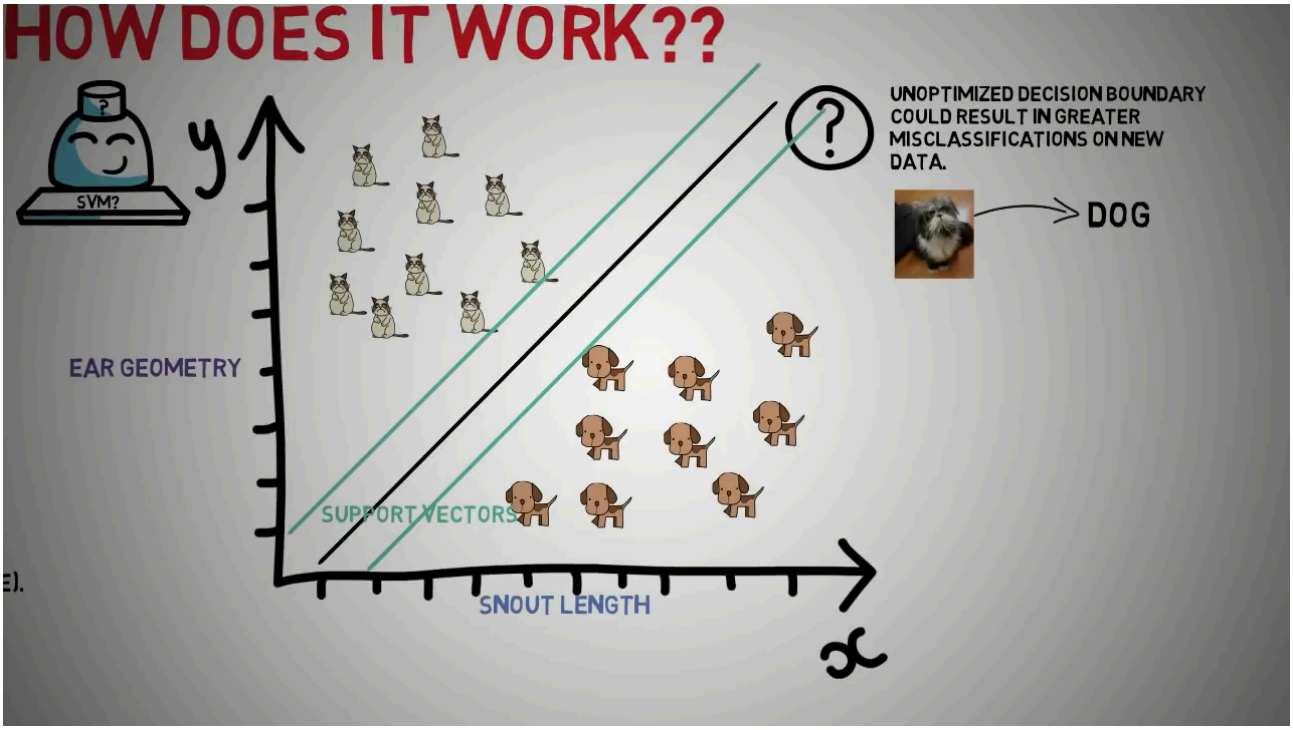
```
pca = PCA(n_components=0.95)  # Keep 95% of variance
X_reduced = pca.fit_transform(X_scaled)
print(f"Reduced dimensions: {X_reduced.shape[1]}")
```

Support Vectors machine

SVM is a supervised learning algorithm that finds the best decision boundary (hyperplane) to separate different classes in the feature space. It aims to maximize the margin between data points of different classes.

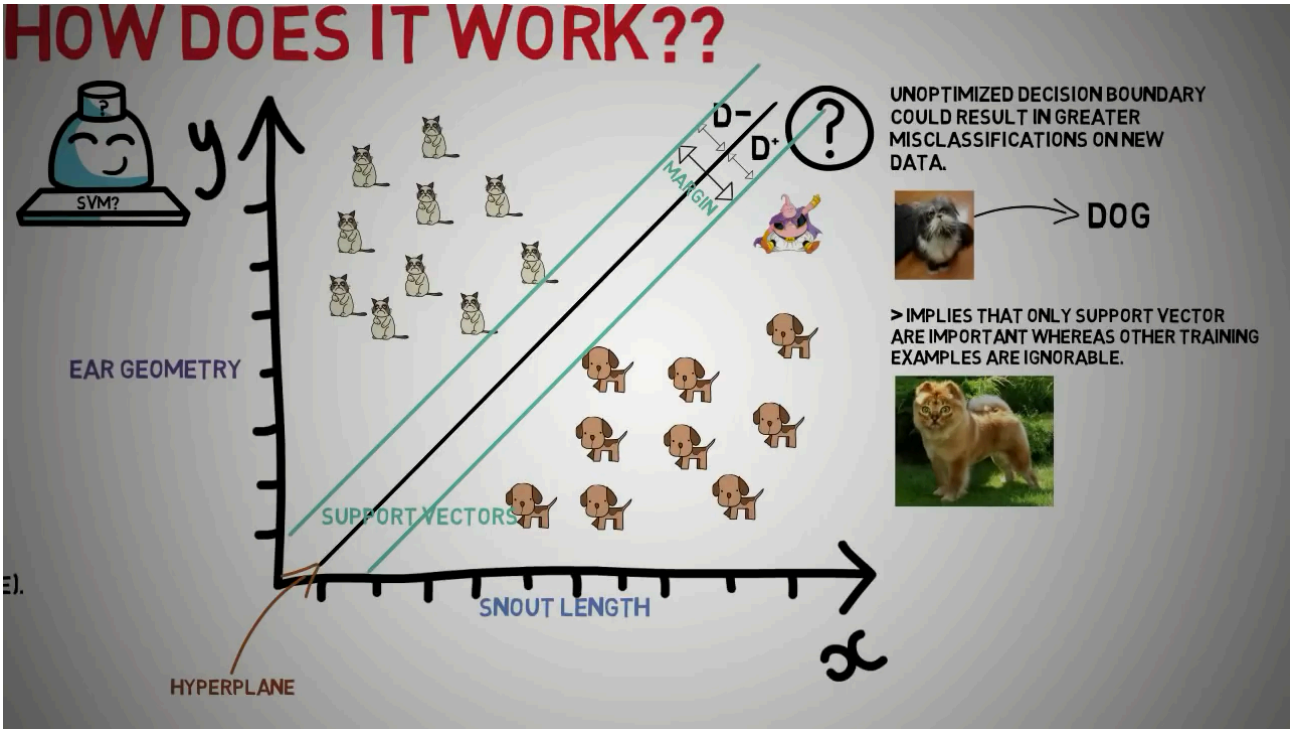
Key terms:

- **Hyperplane**: A boundary that separates data points of different classes.
- **Margin**: The distance between the hyperplane and the nearest data points from each class (support vectors).



Linear SVM

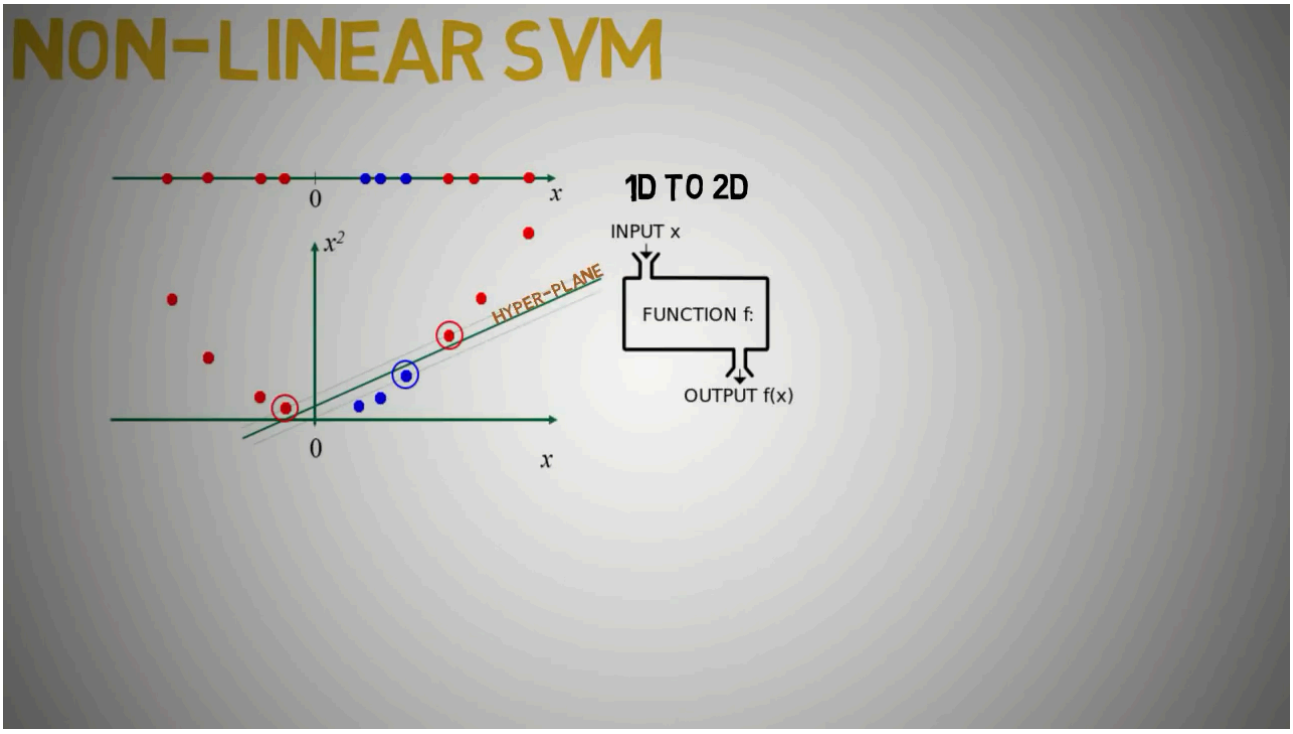
If the data is linearly separable, SVM finds a straight-line hyperplane (in 2D) or a flat hyperplane (in higher dimensions) to separate classes.

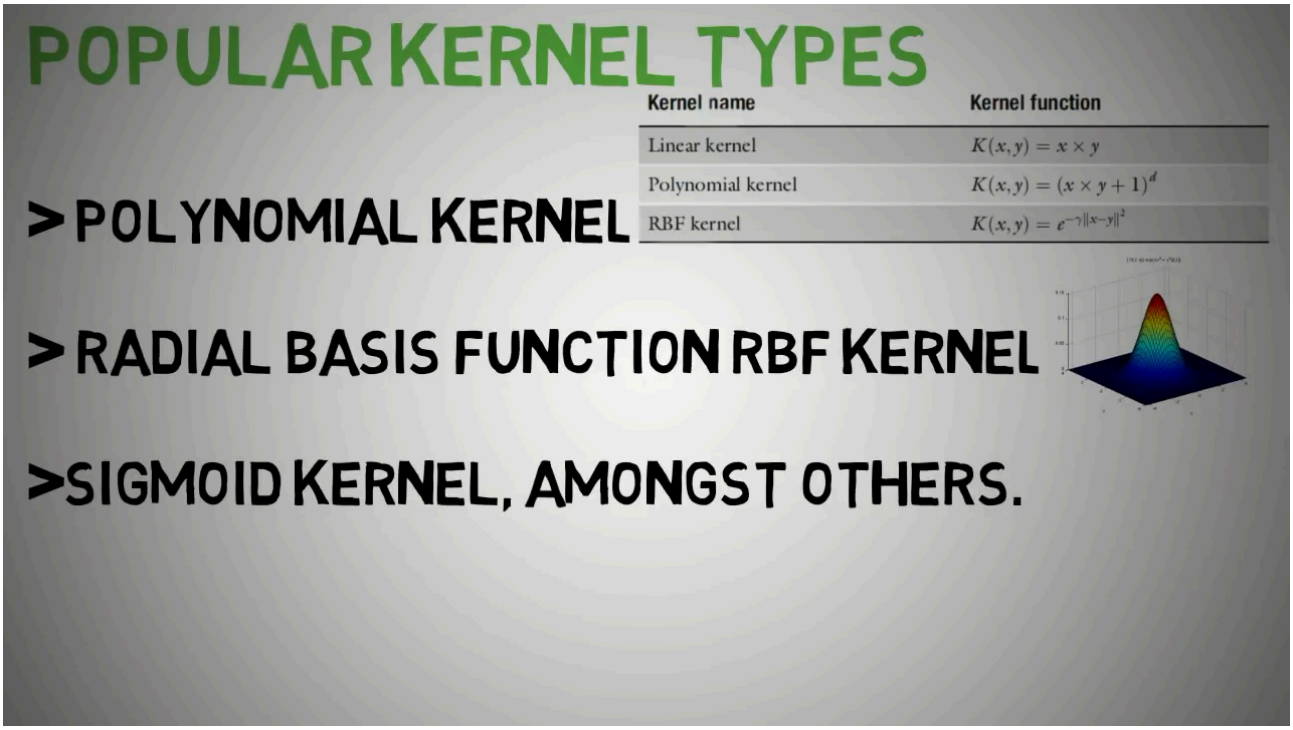


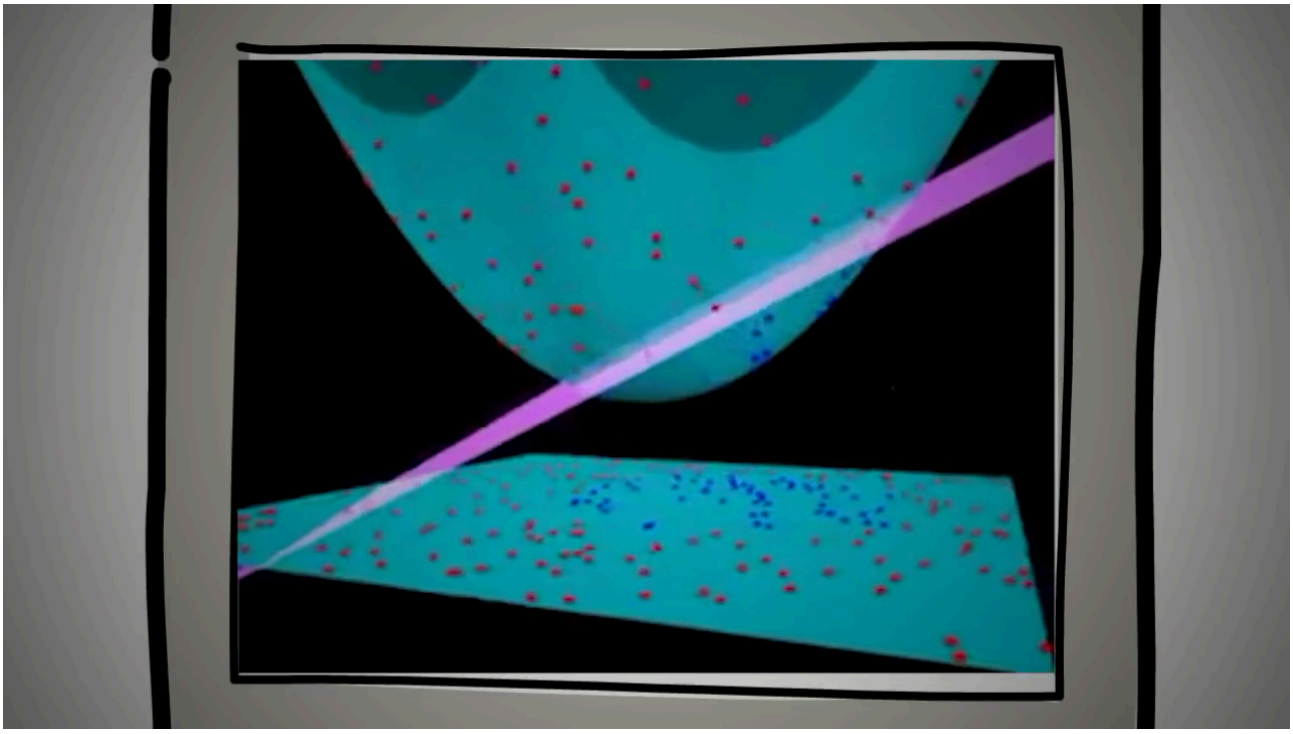
Non-linear SVM with Kernels

Real-world data is often non-linear. SVM uses the "kernel trick" to transform the data into a higher-dimensional space where it becomes linearly separable. Common kernels:

- **Linear Kernel**: For linearly separable data.
- **Polynomial Kernel**: For curved boundaries.
- **Radial Basis Function (RBF) Kernel**: Popular for non-linear problems.


NON-LINEAR SVM
0
x
x^2
HYPER-PLANE
0
x
1D TO 2D
INPUT x
FUNCTION f:
OUTPUT f(x)


POPULAR KERNEL TYPES
Kernel name
Kernel function
Linear kernel
$K(x,y) = x \times y$
Polynomial kernel
$K(x,y) = (x \times y + 1)^d$
RBF kernel
$K(x,y) = e^{-\gamma\|x-y\|^2}$
>POLYNOMIAL KERNEL
>RADIAL BASIS FUNCTION RBF KERNEL
>SIGMOID KERNEL, AMONGST OTHERS.



Advantages

- Effective in high-dimensional spaces.
- Works well for small to medium-sized datasets.
- Handles non-linear data with kernel functions.

Limitations

- Computationally intensive for large datasets.
- Choosing the right kernel and hyperparameters can be challenging.
- Not suitable for noisy datasets with overlapping classes.

---

4. Implementing SVM in Python

Installation

Ensure you have `scikit-learn` installed. You can install it using:

```
pip install scikit-learn
```

Linear SVM for Binary Classification

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

# Generate synthetic data
X, y = make_blobs(n_samples=100, centers=2, random_state=6)

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rando

# Create and train the SVM model
linear_svm = SVC(kernel='linear', C=1)
linear_svm.fit(X_train, y_train)

# Make predictions
y_pred = linear_svm.predict(X_test)

# Evaluate the model
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print(classification_report(y_test, y_pred))

# Plot decision boundary
def plot_decision_boundary(X, y, model):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o')
    plt.show()

plot_decision_boundary(X, y, linear_svm)
```

SVM with RBF Kernel for Non-linear Data

```
# Generate non-linear data
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=100, factor=0.5, noise=0.1)

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rando
```

```
# Create and train the SVM model with RBF kernel
rbf_svm = SVC(kernel='rbf', C=1, gamma=0.5)
rbf_svm.fit(X_train, y_train)

# Make predictions
y_pred = rbf_svm.predict(X_test)

# Evaluate the model
print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print(classification_report(y_test, y_pred))

# Plot decision boundary
plot_decision_boundary(X, y, rbf_svm)
```

---

5. Practical Tips and Use Cases

Tips

- Normalize the data to improve SVM performance.
- Use grid search or cross-validation to tune hyperparameters like `C`, `kernel`, and `gamma`.
- SVM is sensitive to outliers; consider preprocessing your data.

Use Cases

- Text classification (e.g., spam detection).
- Image classification.
- Bioinformatics (e.g., protein classification).
- Handwriting recognition.
