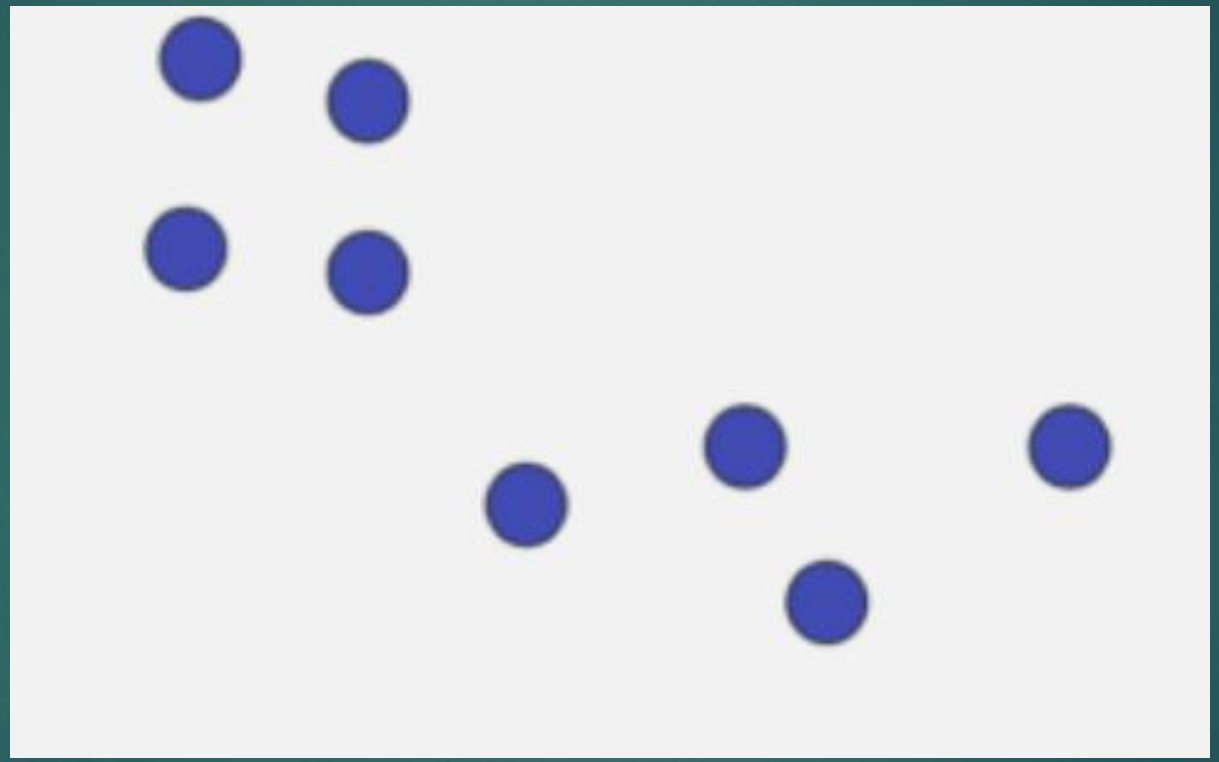
KMeans Clustering

- Find distinct groups (cluster within the data)

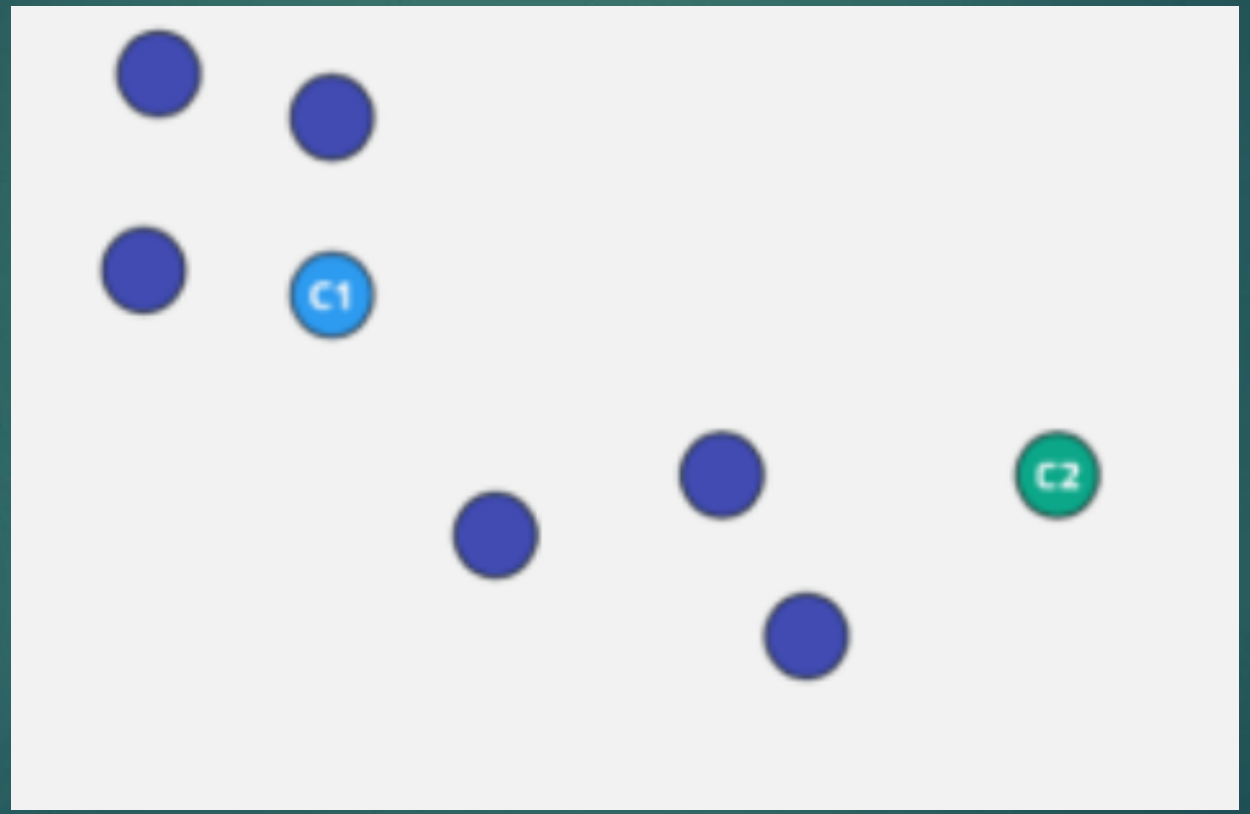
Algorithm

1. Randomly select k centroids
2. Assign each data point to nearest centroid
3. Calculate new centroid by assignment
4. Repeat 2 – 3 until no longer change or maximum iteration

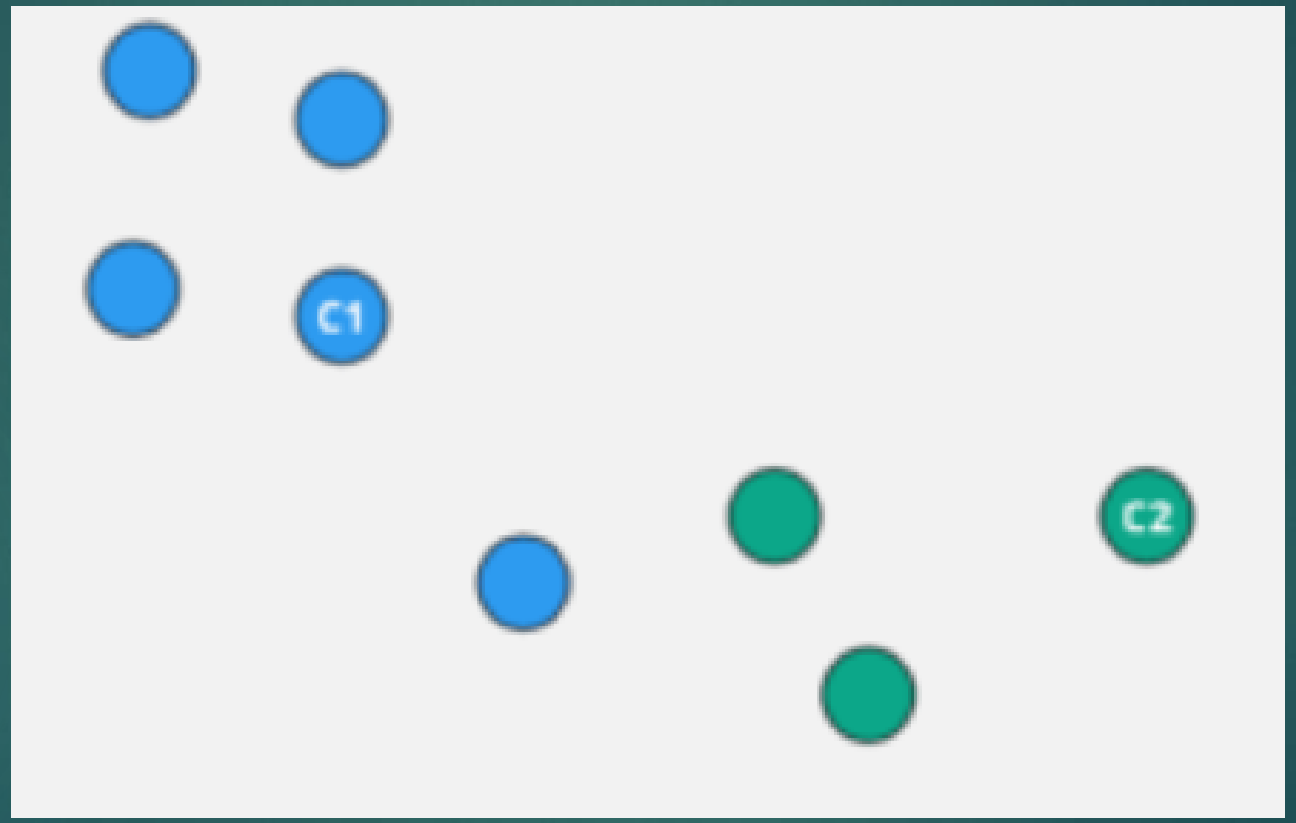
Data Points



Choose Centroid

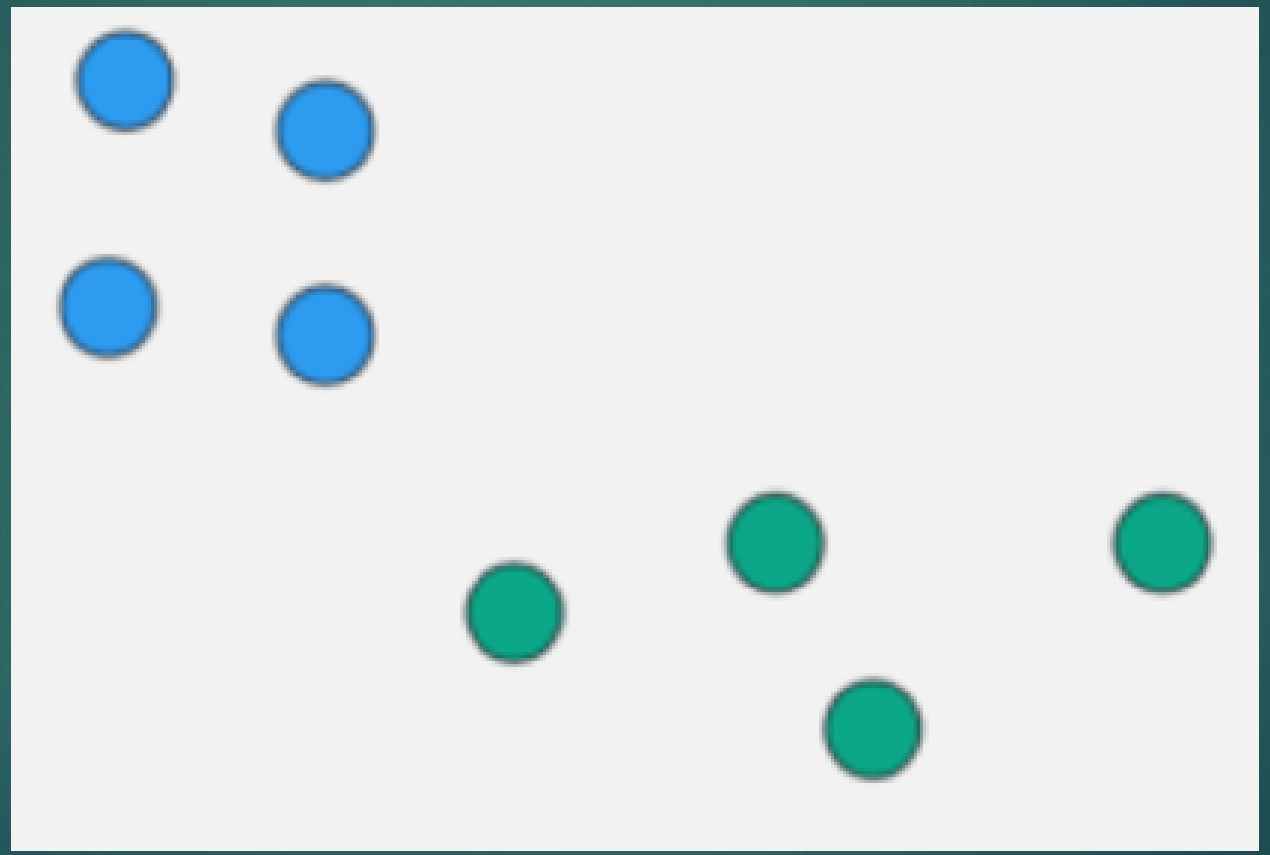


Assign Member

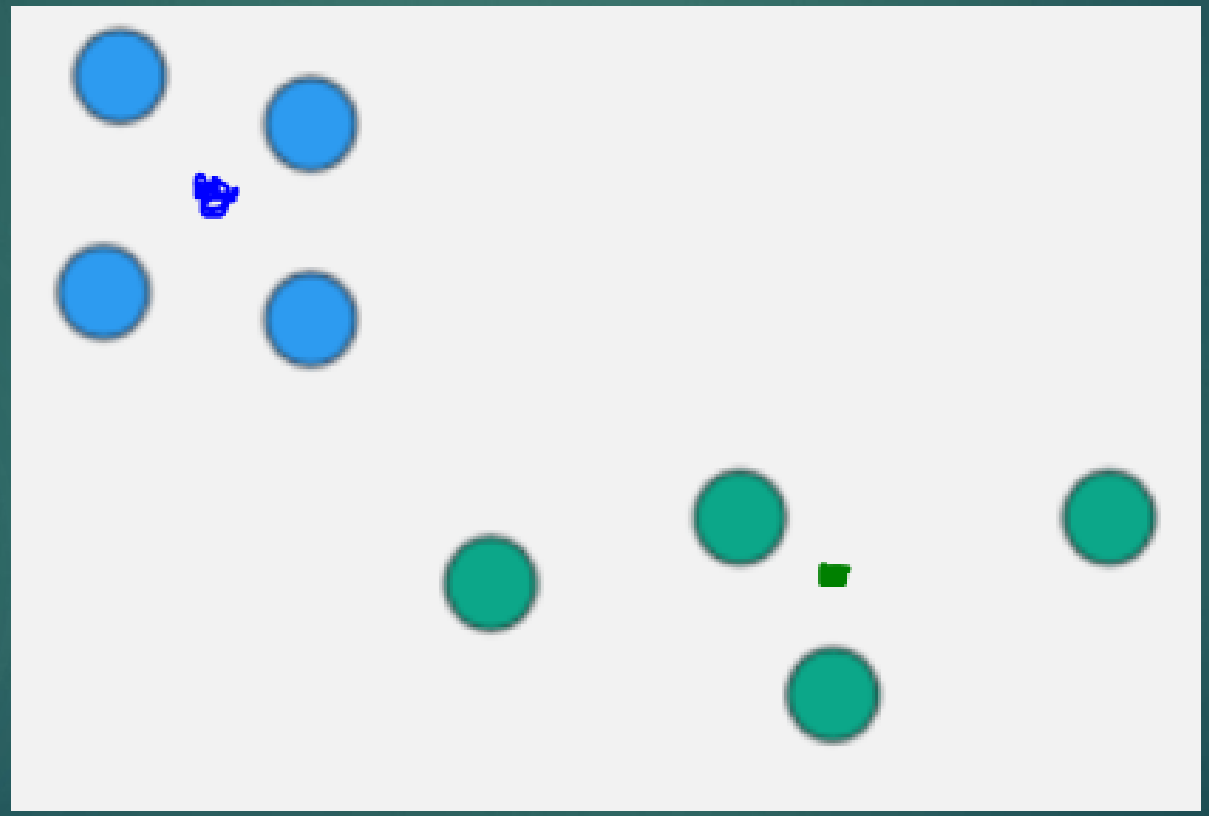


Calculate Centroid

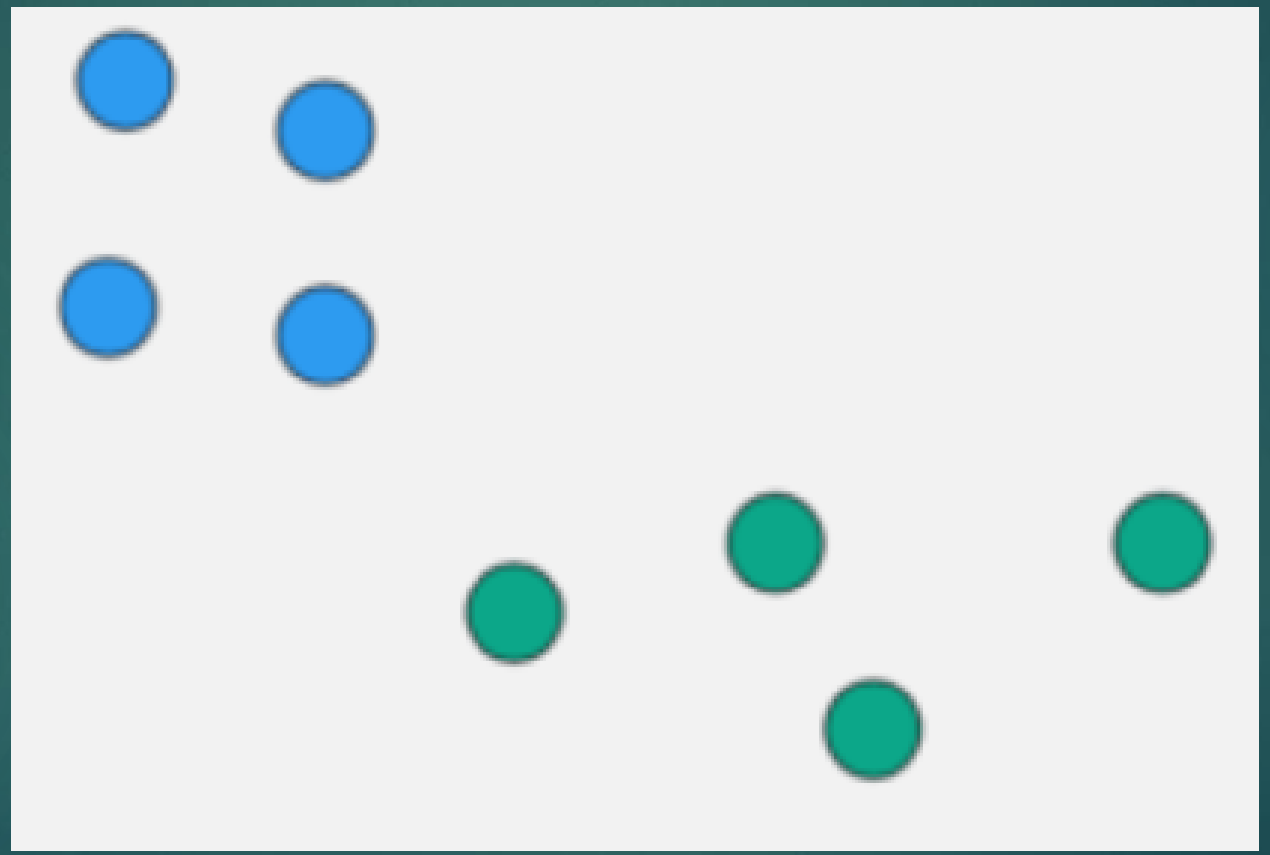
Re-assign Member



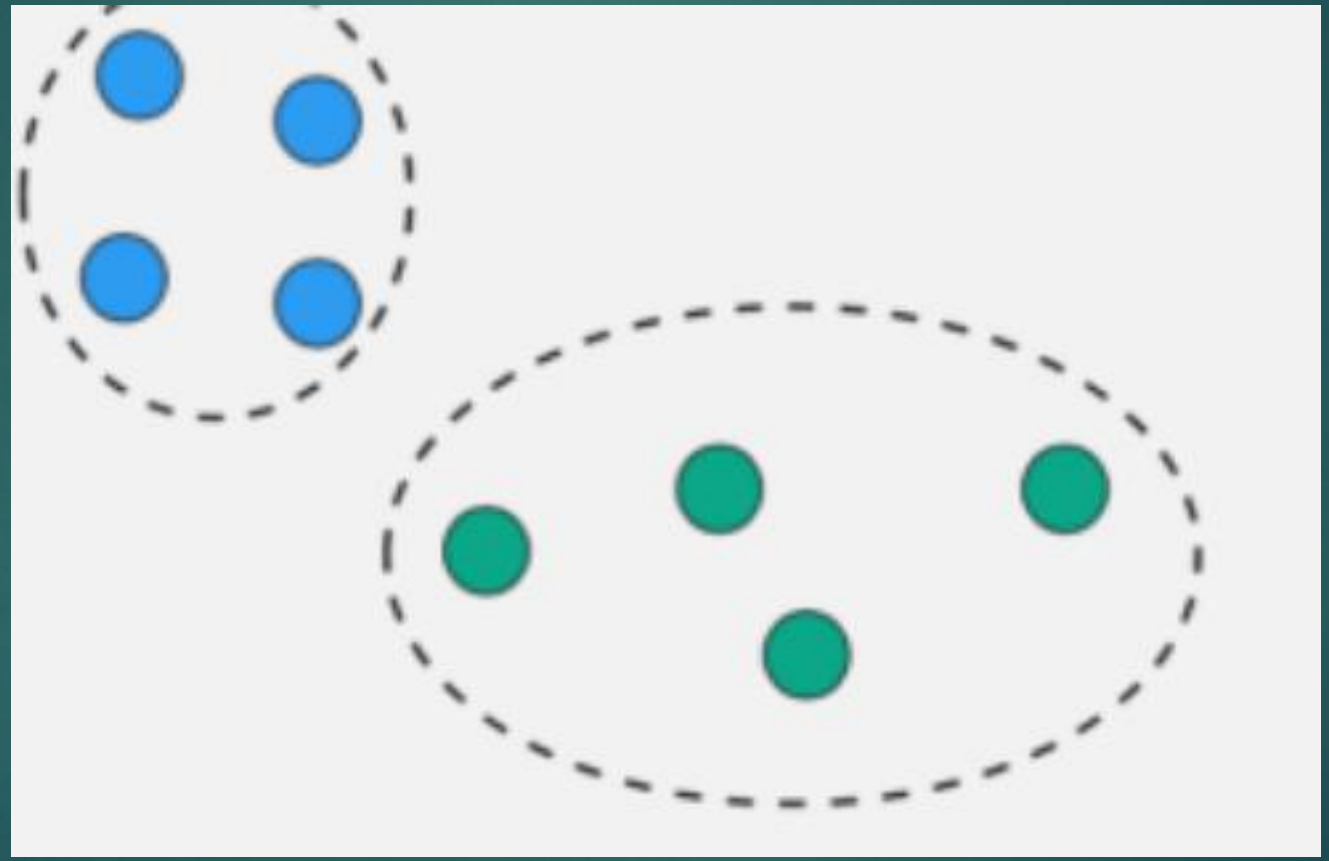
Calculate Centroid



Re-assign Member



Clusters Found



Python Code

```
from sklearn.cluster import KMeans
from sklearn import datasets
import matplotlib.pyplot as plt

X = datasets.load_iris().data
kmc = KMeans(n_clusters=3)
kmc.fit(X)
labels = kmc.predict(X)

# Scatter plot of any two variables(features)
plt.scatter(X[:, 0], X[:, 2], c=labels)

# Display centroid of trained models
centroids = kmc.cluster_centers_

# Plot of centroid along with data
plt.scatter(centroids[:, 0], centroids[:, 2], marker="*", s=150, facecolors="none",
    edgecolors="red", linewidths=2, label="Centroids",)

plt.show()
```

What is Best K ?

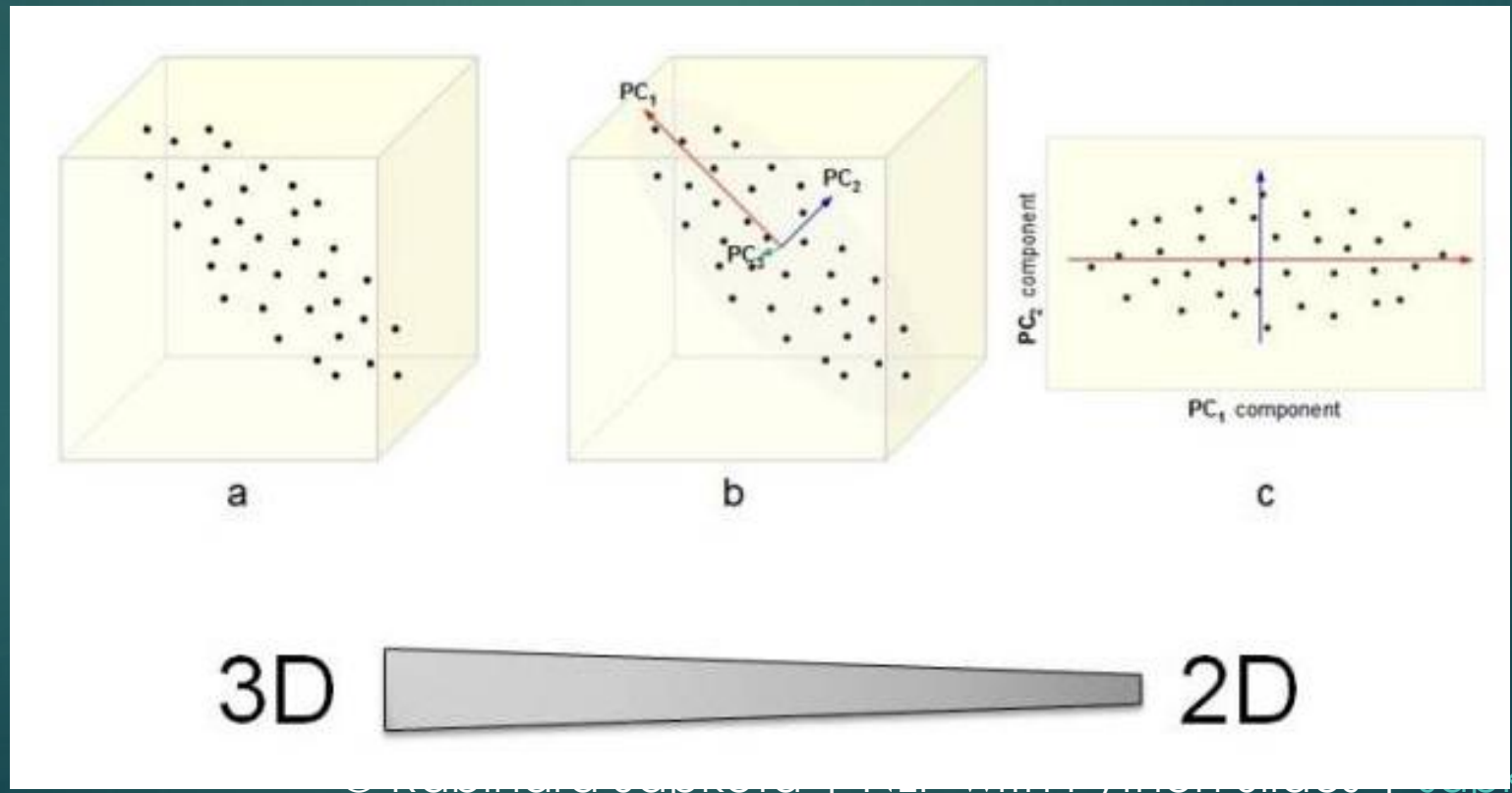
- Inertia measure (WCSS) is used for determining optimum value of K
- WCSS ➔ Within Cluster Sum of Squares
- Gives info on spread calculating the distance of sample from centroid
- With increase in value of k, inertia decrease
- Good cluster has low inertia but not too many clusters
- Plot of inertia vs Number of cluster is done
- Rule of Thumb: Choose elbow in a plot

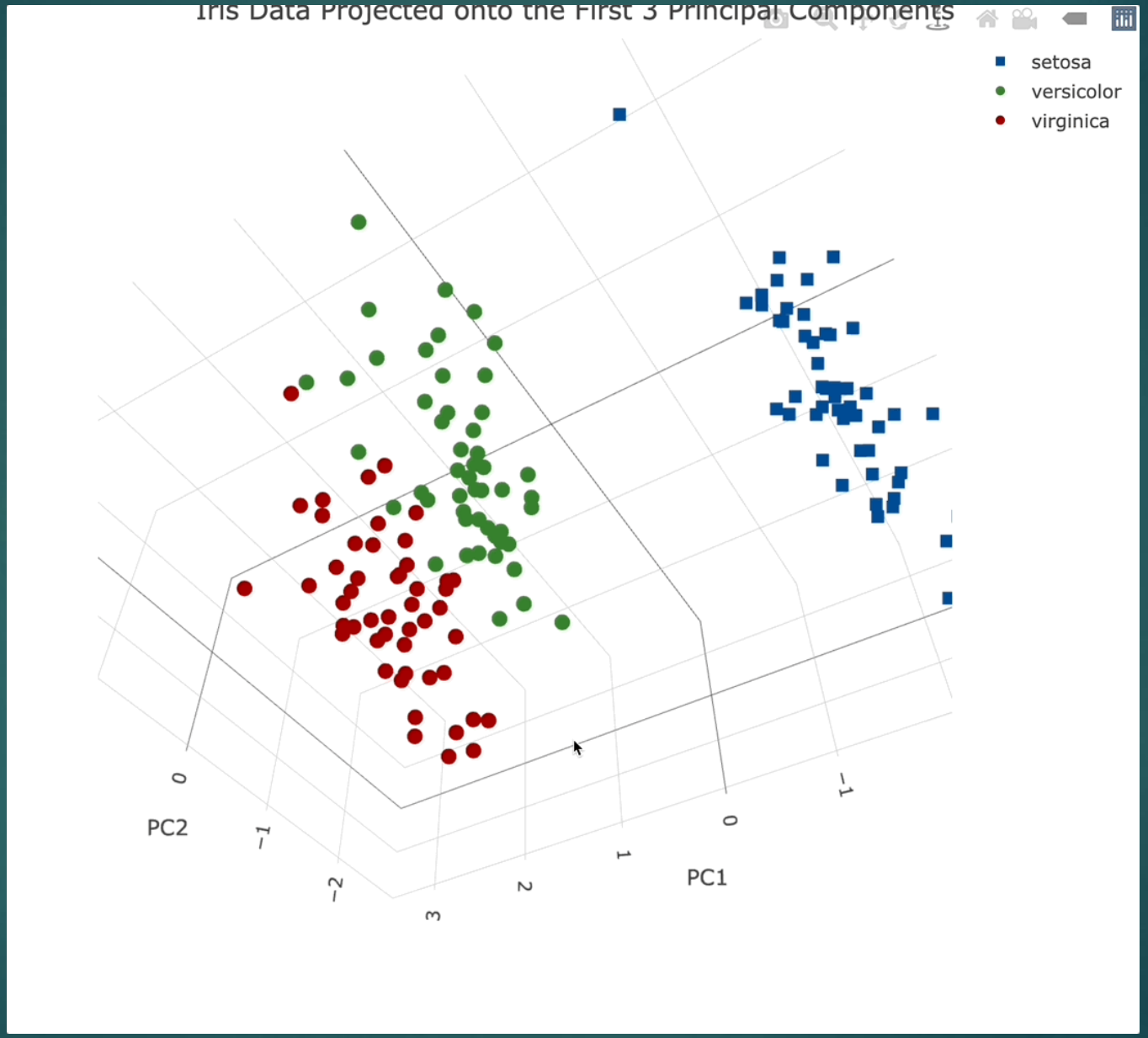
Python Code

```
from sklearn.cluster import KMeans
from sklearn import datasets
import matplotlib.pyplot as plt

X = datasets.load_iris().data
k_list = range(1, 10)
wcss = []

for k_value in k_list:
    kmc = KMeans(n_clusters=k_value)
    kmc.fit(X)
    wcss.append(kmc.inertia_)

# Plot of k vs inertia
plt.plot(k_list, wcss, "-o")
plt.xlabel("Number of clusters K")
plt.ylabel("Inertia / WCSS")
plt.title("WCSS VS Number of clusters")
plt.xticks(k_list)
plt.show()
```

PCA

- Principle Component Analysis
- Dimension Reduction Technique
- Rotates the data to Align with highest variance




Iris Data Projected onto the First 3 Principal Components
setosa
versicolor
virginica
PC2
PC1
0
−1
−2
3
2
1
0
−1

Variance vs PCA

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import pandas as pd

# Load and standardize data
iris = load_iris()
X = iris.data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Run PCA with all 4 components
pca = PCA(n_components=4)
X_pca = pca.fit_transform(X_scaled)

# Create DataFrame of variance explained
explained_var = pd.DataFrame({
        "Principal Component": [f"PC{i+1}" for i in range(4)],
        "Variance Explained Ratio": pca.explained_variance_ratio_,
        "Cumulative Variance": pca.explained_variance_ratio_.cumsum(),})

print(explained_var)
```

NLP

- Natural Language Processing
- Intersection of computer science, linguistics, and AI
- Enables machines to understand, interpret & generate human language
- NLP helps us to preprocess, clean, analyze & extract insight from text
- Use Case: Chat Bot, Language Translation, Spam Detection etc..
- Done with library nltk (`pip install nltk`) Natural Language Tool Kit
- nltk provides tools for tokenization, stop word removal, stemming, lemmatization, POS tagging and more

Tokenization

- Process in which piece of text is split into smaller meaningful units
- Tokens could be words, subwords, characters, or even sentences
- Tokens depends on the type of tokenization

Python Code

```
import nltk
from nltk.tokenize import word_tokenize    # sent_tokenize

# nltk.download("punkt")
# nltk.download("punkt_tab")

text = "NLTK is a great toolkit for working with text."
tokens = word_tokenize(text)
print(tokens)
```

Pre-Processing

- Before analyzing text, it must be pre-processed. Steps include

1. Lowercasing
2. Removing Punctuation / Number
3. Removing stop words
4. Tokenizing text

```
import string

sample_text = "Text Mining in Python, is FUN and powerful!!"

text = sample_text.lower()     # Lowercase
text = text.translate(str.maketrans('', '', string.punctuation))      # Remove punctuation
tokens = nltk.word_tokenize(text)        # Tokenize
print(tokens)
```

Regex Cleaning

- There could be pattern of data that is present in text but not relevant for understanding data
- We can make use of regex to clean these data

```
import re

text = "Contact us at support@example.com or visit https://example.com"

# Remove email
clean_text = re.sub(r"\S+@\S+", "", text)

# Remove URL
clean_text = re.sub(r"http\S+", "", clean_text)

print(clean_text)
```

Stop words

- Common words that are typically filtered out from text before processing
- Don't carry much meaningful information for many NLP tasks.
- So frequent that they are irrelevant to understand meaning of the text
- Example: Articles, Pre-position, conjunction, pronoun, aux verb
- the, is, in, an, and, etc.

Python Code

```
import nltk
from nltk.corpus import stopwords

# nltk.download("stopwords")
# nltk.download("wordnet")

print("Stopwords in English:", stopwords.words("english")[:20])
filtered  = [ w for w in tokens if w not in stopwords.words("english")]
```

Stemming

- Refers to reducing word to their root form
- Root of Play, Played, Playing, Plays ➔ Play
- Its rule based and may produce non words

Python Code

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()
words = ["playing", "played", "plays ", " happily"]
print([stemmer.stem(w) for w in words])
```

POS Tagging

- Part of Speech (POS) is labelling each text with grammatic category

```
import nltk

# Download required resources (only once)
nltk.download("punkt")
nltk.download("averaged_perceptron_tagger")

# Sample text
text = "Apple is launching a new iPhone next month, and many customers are excited."

# Step 1: Tokenize the text into words
tokens = nltk.word_tokenize(text)

# Step 2: Perform POS tagging
pos_tags = nltk.pos_tag(tokens)

print("POS Tags:")
print(pos_tags)
```

TF-IDF & Cosine Similarity

Vector Space Model

- Goal: Represent text documents numerically to measure similarity
- Bag of Words: Treat document as bag of words order ignored but frequency matters
- Vector Form V = {t1, t2 ... tm}. i.e. weight of words in vocabulary
- TF – IDF : Weighting scheme that balances terms importance within document (TF) and across corpus (IDF)

Term Frequency (TF): Count of a term (word) in a document

Inverse Document Frequency (IDF):

- If there are N document with k document having term t then IDF = log(N/k)
- Highlights discriminative terms removing biasing toward common term

```
import numpy as np; import pandas as pd; import re' from collections import Counter

docs = {"D1": "The cat sat on the mat", "D2": "The dog sat on the log", "D3": "Cats and dogs play together", "D4": "My cat likes to play on the mat",}

STOP = {"the", "on", "and", "to", "my"}
LEMMAS = {"cats": "cat", "dogs": "dog", "likes": "like"}

def preprocess(text):
    text = text.lower()
    text = re.sub(r"[^a-z\s]", " ", text)  # keep letters and spaces only
    tokens = [w for w in text.split() if w and w not in STOP]
    tokens = [LEMMAS.get(w, w) for w in tokens]
    return tokens

tokens = {k: preprocess(v) for k, v in docs.items()}
vocab = sorted({w for ts in tokens.values() for w in ts})

def tf_vector(tok_list, vocab):
    c = Counter(tok_list)
    return np.array([c.get(t, 0) for t in vocab], dtype=float)

TF = {k: tf_vector(ts, vocab) for k, ts in tokens.items()}

def find_cosine_similarity(vec_a, vec_b):
    dot_product = vec_a @ vec_b
    mag_a = np.sqrt(np.sum(vec_a**2))
    mag_b = np.sqrt(np.sum(vec_b**2))
    return dot_product / (mag_a * mag_b) if mag_a > 0 and mag_b > 0 else 0.0

def get_similar_document(doc_key):
    similarities = []
    vector_doc = TF.pop(doc_key)
    for doc, vector in TF.items():
        similarity_score = find_cosine_similarity(vector_doc, vector)
        similarities.append((doc, similarity_score))

    most_similar_doc, max_score = max(similarities, key=lambda x: x[1])
    return most_similar_doc, max_score

print("Most similar to D4:", get_similar_document("D4"))
```

Library

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# --- Existing corpus ---
documents = []

# Preprocessing
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

def preprocess(text):
    tokens = nltk.word_tokenize(text.lower())
    tokens = [lemmatizer.lemmatize(t) for t in tokens if t.isalpha() and t not in stop_words]
    return " ".join(tokens)

processed_docs = [preprocess(doc) for doc in documents]

# --- Fit TF-IDF vectorizer on existing corpus ---
vectorizer = TfidfVectorizer(lowercase=True, stop_words='english', max_features=5000)
tfidf_matrix = vectorizer.fit_transform(processed_docs)

# --- New document ---
new_doc = "Neural networks are a core part of deep learning."
processed_new_doc = preprocess(new_doc)
new_vec = vectorizer.transform([processed_new_doc])  # transform only, do NOT fit again

# --- Compute similarity with existing corpus ---
similarities = cosine_similarity(new_vec, tfidf_matrix)  # shape: (1, num_docs)
similarities = similarities.flatten()  # convert to 1D array

# --- Find the most similar existing document ---
most_similar_index = similarities.argmax()
print("Similarity score:", similarities[most_similar_index])
print("Most similar document:", documents[most_similar_index])
```