# DWM CODES

# Decision Tree (Classification)

| Day | Outlook | Temp | Humidity | Wind | Play Tennis |
|---|---|---|---|---|---|
| D1 | Sunny | Hot | High | Weak | No |
| D2 | Sunny | Hot | High | Strong | No |
| D3 | Overcast | Hot | High | Weak | Yes |
| D4 | Rain | Mild | High | Weak | Yes |
| D5 | Rain | Cool | Normal | Weak | Yes |
| D6 | Rain | Cool | Normal | Strong | No |
| D7 | Overcast | Cool | Normal | Strong | Yes |
| D8 | Sunny | Mild | High | Weak | No |
| D9 | Sunny | Cool | Normal | Weak | Yes |
| D10 | Rain | Mild | Normal | Weak | Yes |
| D11 | Sunny | Mild | Normal | Strong | Yes |
| D12 | Overcast | Mild | High | Strong | Yes |
| D13 | Overcast | Hot | Normal | Weak | Yes |
| D14 | Rain | Mild | High | Strong | No |

```
import math

# Tennis data (Outlook, Temp, Humidity, Wind, PlayTennis)
data = [
    ['Sunny', 'Hot', 'High', 'Weak', 'No'],
    ['Sunny', 'Hot', 'High', 'Strong', 'No'],
    ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'High', 'Weak', 'Yes'],
    ['Rain', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Cool', 'Normal', 'Strong', 'No'],
    ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
    ['Sunny', 'Mild', 'High', 'Weak', 'No'],
    ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'Normal', 'Weak', 'Yes'],
    ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
    ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
    ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'High', 'Strong', 'No']
]

features = ['Outlook', 'Temp', 'Humidity', 'Wind']

def entropy(values):
    counts = {}
    for v in values:
        counts[v] = counts.get(v, 0) + 1
```

```
    return -sum((count/len(values)) * math.log2(count/len(values))
            for count in counts.values())

def best_feature(data, features):
    base_entropy = entropy([row[-1] for row in data])
    best_gain = 0
    best_feat = None

    for i in range(len(features)):
        feat_values = {row[i] for row in data}
        feat_entropy = 0
        for value in feat_values:
            subset = [row[-1] for row in data if row[i] == value]
            feat_entropy += (len(subset)/len(data)) * entropy(subset)
        gain = base_entropy - feat_entropy
        if gain > best_gain:
            best_gain = gain
            best_feat = i
    return best_feat

def build_tree(data, features):
    outcomes = [row[-1] for row in data]
    if len(set(outcomes)) == 1:
        return outcomes[0]

    best_idx = best_feature(data, features)
    if best_idx is None:
        return max(set(outcomes), key=outcomes.count)

    tree = {features[best_idx]: {}}
    for value in {row[best_idx] for row in data}:
        subset = [row for row in data if row[best_idx] == value]
        tree[features[best_idx]][value] = build_tree(subset, features)
    return tree

tree = build_tree(data, features)
print(tree)
```

output

{'Outlook': {'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}, 'Rain': {'Wind': {'Weak': 'Yes', 'Strong': 'No'}}, 'Overcast': 'Yes'}}

# Naive Bayesian (Classification)

```
data = [
    ['Yes', 'No', 'Yes'],
    ['No', 'Yes', 'Yes'],
    ['Yes', 'Yes', 'Yes'],
    ['No', 'No', 'No'],
    ['Yes', 'No', 'Yes'],
    ['No', 'No', 'Yes'],
    ['Yes', 'No', 'Yes'],
    ['Yes', 'No', 'No'],
    ['No', 'Yes', 'Yes'],
    ['No', 'Yes', 'No'],
]

from collections import defaultdict

# Count classes
yes = sum(1 for row in data if row[2] == 'Yes')
no = len(data) - yes
total = yes + no

# Feature counts
counts = {
    'Covid': {'Yes': defaultdict(int), 'No': defaultdict(int)},
    'Flu': {'Yes': defaultdict(int), 'No': defaultdict(int)}
}

for row in data:
    covid, flu, fever = row
    counts['Covid'][fever][covid] += 1
    counts['Flu'][fever][flu] += 1

def predict(covid, flu):
    # Prior probabilities
    p_yes = yes / total
    p_no = no / total

    # Likelihoods without smoothing
    try:
        covid_yes = counts['Covid']['Yes'][covid] / yes
```

```
        flu_yes = counts['Flu']['Yes'][flu] / yes
        p_yes *= covid_yes * flu_yes
    except ZeroDivisionError:
        p_yes = 0

    try:
        covid_no = counts['Covid']['No'][covid] / no
        flu_no = counts['Flu']['No'][flu] / no
        p_no *= covid_no * flu_no
    except ZeroDivisionError:
        p_no = 0

    print(f"\nInput: Covid={covid}, Flu={flu}")
    print(f"P(Fever=Yes): {p_yes:.5f}")
    print(f"P(Fever=No):  {p_no:.5f}")

    return 'Yes' if p_yes > p_no else 'No'

# Tests
print("Predicted Fever:", predict('Yes', 'Yes'))  # Expected: Yes
```

## K-means - 1D

```
data = [2, 4, 10, 12, 3, 20, 30, 11, 25]

# Initial centroids
m1 = 2
m2 = 4

for _ in range(10):  # max 10 iterations
    g1 = []
    g2 = []

    # Assign to nearest cluster
    for x in data:
        if abs(x - m1) < abs(x - m2):
            g1.append(x)
        else:
            g2.append(x)

    # Recalculate means
    new_m1 = sum(g1) / len(g1)
```

```
    new_m2 = sum(g2) / len(g2)

    # Stop if centroids don’t change
    if new_m1 == m1 and new_m2 == m2:
        break

    m1 = new_m1
    m2 = new_m2

# Final output
print("Cluster 1:", g1)
print("Cluster 2:", g2)
print("Final Centroids:", round(m1, 2), "and", round(m2, 2))
```

## 2D K-means (Clustering)

```
# Sample data: (Age, Amount)
data = [
    (20, 500),   # c1
    (40, 1000),  # c2
    (30, 800),   # c3
    (18, 300),   # c4
    (28, 1200),  # c5
    (35, 1400),  # c6
    (45, 1800)   # c7
]

# Initial centroids (first two points)
centroids = [data[0], data[1]]  # (20, 500) and (40, 1000)

def euclidean_distance(p1, p2):
    return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5

def k_means_single_iteration(data, centroids):
    # Step 1: Create empty clusters
    clusters = [[] for _ in range(len(centroids))]

    # Step 2: Assign each point to the nearest centroid
    for point in data:
        distances = [euclidean_distance(point, centroid) for centroid in
centroids]
        closest = distances.index(min(distances))  # Find the closest centroid
        clusters[closest].append(point)  # Assign point to that cluster
```

```
    return centroids, clusters

# Run K-Means for just one iteration
centroids, clusters = k_means_single_iteration(data, centroids)

# Output the results
print("Centroids:", centroids)
for i, cluster in enumerate(clusters):
    print(f"Cluster {i + 1}: {cluster}")
```

# Agglomerative – Single

```
import math

# Data: (X, Y)
data = [
    (4, 3),  # s1
    (1, 4),  # s2
    (2, 1),  # s3
    (3, 8),  # s4
    (6, 9),  # s5
    (5, 1),  # s6
]

# Names for the points
names = ['s1', 's2', 's3', 's4', 's5', 's6']

# Function to calculate Euclidean distance
def euclidean_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

# Agglomerative clustering (single linkage)
def agglomerative_clustering(data, names):
    clusters = [[i] for i in range(len(data))]  # Each point starts as its own
cluster

    # Print initial clusters
    print("Initial Clusters:")
    for i, cluster in enumerate(clusters):
        cluster_names = [names[idx] for idx in cluster]
        print(f"Cluster {i + 1}: {cluster_names}")
    print()

    while len(clusters) > 1:
```

```
        min_dist = float('inf')
        closest_pair = None

        # Find the closest pair of clusters
        for i in range(len(clusters)):
            for j in range(i + 1, len(clusters)):
                # Find minimum distance between any two points in the
clusters (single linkage)
                dist = min([euclidean_distance(data[p1], data[p2]) for p1 in
clusters[i] for p2 in clusters[j]])

                if dist < min_dist:
                    min_dist = dist
                    closest_pair = (i, j)

        # Merge the closest clusters
        c1, c2 = closest_pair
        clusters[c1] += clusters[c2]
        clusters.pop(c2)

        # Print the current clusters after merging
        print(f"After merging clusters {c1 + 1} and {c2 + 1}:")
        for i, cluster in enumerate(clusters):
            cluster_names = [names[idx] for idx in cluster]
            print(f"Cluster {i + 1}: {cluster_names}")
        print()

    # Return the final clusters with names
    final_cluster = clusters[0]
    return [names[i] for i in final_cluster]

# Run the agglomerative clustering
final_clusters = agglomerative_clustering(data, names)
```

## Agglomerative – OVERALL (Hierarchical Clustering)

```
import math

# Data: (X, Y)
data = [
    (4, 3),  # s1
    (1, 4),  # s2
    (2, 1),  # s3
```

```
    (3, 8),  # s4
    (6, 9),  # s5
    (5, 1),  # s6
]

# Names for points
names = ['s1', 's2', 's3', 's4', 's5', 's6']

# Euclidean distance function
def euclidean(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

# Linkage distance calculation
def cluster_distance(c1, c2, method):
    distances = [euclidean(data[i], data[j]) for i in c1 for j in c2]
    if method == 'single':
        return min(distances)
    elif method == 'complete':
        return max(distances)
    elif method == 'average':
        return sum(distances) / len(distances)

# Agglomerative clustering
def agglomerative_clustering(data, names, linkage='single'):
    clusters = [[i] for i in range(len(data))]

    print(f"Initial Clusters ({linkage} linkage):")
    for i, cluster in enumerate(clusters):
        print(f"Cluster {i+1}: {[names[idx] for idx in cluster]}")
    print()

    while len(clusters) > 1:
        min_dist = float('inf')
        pair = (0, 1)

        # Find closest pair
        for i in range(len(clusters)):
            for j in range(i+1, len(clusters)):
                dist = cluster_distance(clusters[i], clusters[j], linkage)
                if dist < min_dist:
                    min_dist = dist
                    pair = (i, j)
```

```python
        # Merge clusters
        i, j = pair
        clusters[i] += clusters[j]
        clusters.pop(j)

        # Print current clusters
        print(f"After merging clusters {i+1} and {j+1}:")
        for k, cluster in enumerate(clusters):
            print(f"Cluster {k+1}: {[names[idx] for idx in cluster]}")
        print()

    return [names[i] for i in clusters[0]]

# Run for all linkage methods
print("\n=== SINGLE LINKAGE ===\n")
agglomerative_clustering(data, names, linkage='single')

print("\n=== COMPLETE LINKAGE ===\n")
agglomerative_clustering(data, names, linkage='complete')

print("\n=== AVERAGE LINKAGE ===\n")
agglomerative_clustering(data, names, linkage='average')
```

## Apriori (Association Rule Mining)

```python
# Transactions
data = [
    ['Bread', 'Butter', 'Jam', 'Milk'],
    ['Bread', 'Butter', 'Milk'],
    ['Bread', 'Juice', 'Cereal'],
    ['Bread', 'Milk', 'Juice'],
    ['Butter', 'Milk', 'Juice']
]

support_percent = 50
confidence_percent = 75
total = len(data)

# Get unique items
items = []
for t in data:
    for item in t:
        if item not in items:
            items.append(item)
```

```
# Count support
def count(items_list):
    c = 0
    for t in data:
        if all(i in t for i in items_list):
            c += 1
    return c

# Check 2-item combinations
for i in range(len(items)):
    for j in range(i + 1, len(items)):
        A = items[i]
        B = items[j]
        ab = count([A, B])
        support = (ab / total) * 100

        if support >= support_percent:
            a = count([A])
            b = count([B])
            conf_ab = (ab / a) * 100
            conf_ba = (ab / b) * 100

            if conf_ab >= confidence_percent:
                print(f"conf({A} -> {B}) = {ab}/{a} = {round(conf_ab)}%")

            if conf_ba >= confidence_percent:
                print(f"conf({B} -> {A}) = {ab}/{b} = {round(conf_ba)}%")
```

## Linear Regression (Prediction Algorithm)

```
# Data
experience = [3, 8, 9, 13, 3, 6, 11, 21, 1, 16]
salary =     [30, 57, 64, 72, 36, 43, 59, 90, 20, 83]

# Step 1: Calculate means
n = len(experience)
mean_x = sum(experience) / n
mean_y = sum(salary) / n

# Step 2: Calculate slope (m) and intercept (c)
```

```
numerator = sum((experience[i] - mean_x) * (salary[i] - mean_y) for i in
range(n))
denominator = sum((experience[i] - mean_x) ** 2 for i in range(n))

m = numerator / denominator
c = mean_y - m * mean_x

# Step 3: Predict salary for 10 years of experience
x_new = 10
y_pred = m * x_new + c

# Output
print("Slope (m):", m)
print("Intercept (c):", c)
print("Predicted Salary for 10 years experience:", y_pred)
```