Affinity Propagation

Affinity Propagation (AP) is a clustering algorithm that identifies exemplars (representative data points) among the data and forms clusters by assigning each data point to the exemplar. Unlike other clustering algorithms like k-means because it doesn't require the number of clusters to be specified previously.

**How Affinity Propagation Works:**

1. **Input Similarity Matrix**: Instead of working with raw data, AP works with a similarity matrix, where each entry represents the similarity between a pair of data points. This matrix is crucial because it drives how clusters are formed.
2. **Messages Exchange**: The algorithm iteratively sends two types of messages between data points:
   * **Responsibility**: Sent from data point i to candidate exemplar k, indicating how well-suited k is as the exemplar for i compared to other candidates.
   * **Availability**: Sent from candidate exemplar k to data point i, indicating how appropriate it would be for i to choose k as its exemplar.
3. **Exemplar Identification**: After several iterations, data points with the highest "net" responsibility and availability are chosen as exemplars, and other data points are assigned to these exemplars to form clusters.
4. **Preference Parameter:** the algorithm uses a preference parameter to influence the selection of exemplars; higher values result in exemplars.

**Types of Datasets that can be used for Affinity Propagation**

1. **Similarity-based Data**: AP works best with datasets where a meaningful similarity measure between data points can be defined. Examples include:
   * **Text Data**: Where cosine similarity or Jaccard similarity between documents can be used.
   * **Image Data**: Where pixel-based similarity or feature vector similarity is available.
   * **Biological Data**: Where genetic or protein similarity measures are available.
2. **Moderate-sized Datasets**: Due to its memory and computational requirements, AP is more suitable for datasets that are not extremely large. It performs well on moderate-sized datasets(i.e.. in between 100 – 1000 images), especially where the number of potential exemplars isn't overwhelming.
3. **Datasets with Unknown Number of Clusters**: AP is particularly useful when the number of clusters is not known in advance, which is often the case in exploratory data analysis.
4. **Datasets with Varying Cluster Shapes**: AP can handle complex cluster shapes, making it useful for datasets where clusters are not spherical or have varying densities.

Affinity Propagation is a powerful clustering method, especially when you have a well-defined similarity measure and no prior knowledge of the number of clusters. However, it may not be suitable for extremely large datasets or situations where computational resources are limited. It is most effective with similarity-based datasets of moderate size, where cluster shapes can be irregular.

Affinity Propagation (AP) can be used in conjunction with image data to perform clustering based on features extracted from the images. To understand how AP works with image data, let’s break down the process and how workflow is happening

**Workflow:**

Here is a simplified workflow for using Affinity Propagation with image data:

1. **Image Data Preprocessing**: Resize, normalize, or augment the images as necessary.
2. **Feature Extraction**: Use a method like CNN feature extraction to obtain a feature vector for each image depending upon which feature to be extracted.
3. **Similarity Matrix Calculation**: Compute the similarity matrix using an appropriate distance or similarity measure.
4. **Apply Affinity Propagation**: Run the Affinity Propagation algorithm on the similarity matrix.
5. **Analyze Clusters**: Interpret the clusters and exemplars identified by the algorithm.

When working with image data, Affinity Propagation can be a powerful clustering tool, especially when the dataset's structure is unknown, and the number of clusters is not predetermined. However, the success of the algorithm hinges on effective feature extraction and the choice of similarity measure. It’s best suited for applications with a moderate number of images where computational resources can handle the intensity of the algorithm.

**Pros and Cons of Affinity Propagation**

**Pros:**

1. **No Need to Specify Number of Clusters**: Unlike k-means, where you need to define the number of clusters in advance, AP automatically determines the number of clusters.
2. **Handles Different Cluster Sizes and Densities**: AP can effectively handle clusters of varying sizes and densities, which is challenging for algorithms like k-means.
3. **Automatic Cluster Number Detection**: this automatically detects the number of clusters based on the feature similarities.
4. **Flexible Similarity Measures**: You can tailor the similarity measure to suit the specific features extracted from the images.
5. **Handles Complex Structures**: this can cluster images that belong to non-convex or irregular groups based on their features.

**Cons:**

* **Computationally Intensive**: For large image datasets, calculating and storing the similarity matrix can be computationally expensive.
* **Feature Dependency**: The quality of the clustering depends heavily on the feature extraction method. Poor feature selection can lead to suboptimal clustering.
* **Memory Intensive**: AP can be memory-intensive, especially with large numbers of images, as it requires storing and processing a large similarity matrix.
* **Possible Convergence Issues**: For some datasets, AP may converge to non-optimal solutions or fail to converge altogether, especially if the damping factor (which controls message update rates) is not properly tuned.

Affinity Propagation (AP) is a clustering algorithm that works by sending messages between data points until a set of exemplars (representative points) and corresponding clusters emerge. Unlike algorithms like K-means, AP does not require specifying the number of clusters beforehand. Here’s how Affinity Propagation is applied in the code:

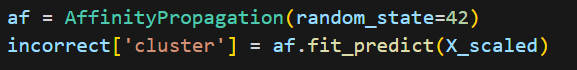
**How Affinity Propagation Works in the Code**

1. **Input Data Preprocessing:**
   * The code reads in a CSV file containing image data with columns like "Luminescence" and "Correct Prediction".
   * Data is split into two subsets: correct and incorrect predictions based on the 'Correct Prediction' column.
2. **Selecting Data for Clustering:**
   * Clustering is only performed on the incorrectly classified samples (incorrect DataFrame), focusing on the 'Luminescence' feature.

A screen shot of a computer

Description automatically generated

1. **Standardizing the Data:**
   * The code applies StandardScaler to normalize the luminescence values before clustering. This ensures that the clustering algorithm works effectively without being biased by the scale of the data.
2. **Clustering with Affinity Propagation:**
   * Affinity Propagation is initialized with a random state for reproducibility and is fit to the scaled data.
   * The algorithm identifies clusters based on "exemplar points," which are representative samples for each cluster.

]

**How AP Works Internally:**

* + **Message Passing:** The algorithm iteratively updates two types of messages between data points:
    - **Responsibility (r(i, k)):** Indicates how well-suited a point k is to be the exemplar for point i.
    - **Availability (a(i, k)):** Reflects how appropriate it is for point i to choose point k as its exemplar, considering other possible exemplars.
  + **Determining Exemplars:** Points that maximize the sum of their availability and responsibility become exemplars, representing a cluster. Each non-exemplar point is assigned to the cluster of its closest exemplar.

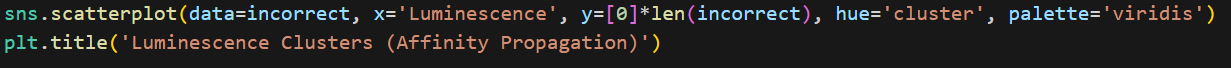
1. **Extracting Clusters and Statistics:**
   * The number of clusters is determined from the number of unique exemplars found by the algorithm:

A black background with white and blue text

Description automatically generated

* + For each cluster, the code computes:
    - The cluster center (mean luminescence value).
    - The range of luminescence values within the cluster.
    - The misclassification rate within the cluster (based on how many points fall into the cluster's range).

1. **Visualizing Clusters:**
   * A scatter plot visualizes the clusters formed by Affinity Propagation. The incorrect data points are coloured according to their cluster assignment, and additional annotations provide misclassification details.



Affinity Propagation in this code works by iteratively refining the relationships between data points to determine clusters based on exemplar points. The algorithm operates without needing a predefined number of clusters, making it suitable for this task where the number of misclassification clusters isn't known beforehand.

The visualization and statistics generated provide insights into how different ranges of luminescence values relate to incorrect classifications, offering a detailed breakdown of the model’s performance across different conditions.

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.cluster import AffinityPropagation

from sklearn.preprocessing import StandardScaler

# Constants

LUMINANCE\_RANGE = (0, 120) # Adjusted based on the actual data range

# Step 1: Load data

data = pd.read\_csv('classification\_results.csv')

# Convert 'Correct Prediction' to boolean

data['Correct Prediction'] = data['Correct Prediction'].map({'Yes': True, 'No': False})

correct = data[data['Correct Prediction']]

incorrect = data[~data['Correct Prediction']]

# Step 2: Affinity Propagation Clustering

X = incorrect[['Luminescence']]

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Initialize and fit Affinity Propagation

af = AffinityPropagation(random\_state=42)

incorrect['cluster'] = af.fit\_predict(X\_scaled)

# Get the number of clusters

N\_CLUSTERS = len(af.cluster\_centers\_indices\_)

print(f'Number of clusters: {N\_CLUSTERS}')

# Calculate cluster centers, ranges, and misclassification rates

cluster\_centers = []

cluster\_ranges = []

cluster\_misclassification\_rates = []

for cluster\_id in range(N\_CLUSTERS):

cluster\_data = incorrect[incorrect['cluster'] == cluster\_id]

cluster\_center = cluster\_data['Luminescence'].mean()

cluster\_range = (cluster\_data['Luminescence'].min(), cluster\_data['Luminescence'].max())

total\_in\_range = len(data[(data['Luminescence'] >= cluster\_range[0]) & (data['Luminescence'] <= cluster\_range[1])])

misclassified\_in\_range = len(cluster\_data)

misclassification\_rate = misclassified\_in\_range / total\_in\_range if total\_in\_range > 0 else 0

cluster\_centers.append(cluster\_center)

cluster\_ranges.append(cluster\_range)

cluster\_misclassification\_rates.append(misclassification\_rate)

# Affinity Propagation clustering plot

plt.figure(figsize=(12, 8))

sns.scatterplot(data=incorrect, x='Luminescence', y=[0]\*len(incorrect), hue='cluster', palette='viridis')

plt.title('Luminescence Clusters (Affinity Propagation)')

sns.scatterplot(data=correct, x='Luminescence', y=[-0.1]\*len(correct), color='green', marker='o', label='Correctly Classified')

sns.scatterplot(data=incorrect, x='Luminescence', y=[-0.1]\*len(incorrect), color='red', marker='x', label='Misclassified')

# Calculate positions for text annotations and avoid overlap

cluster\_centers\_sorted = sorted(cluster\_centers)

y\_positions = np.linspace(0.2, 1.2, len(cluster\_centers\_sorted))

plt.ylim(-0.5, 2)

for i, cluster\_id in enumerate(range(N\_CLUSTERS)):

cluster\_data = incorrect[incorrect['cluster'] == cluster\_id]

cluster\_center = cluster\_data['Luminescence'].mean()

misclassified\_count = len(cluster\_data)

total\_count = len(data[(data['Luminescence'] >= cluster\_data['Luminescence'].min()) & (data['Luminescence'] <= cluster\_data['Luminescence'].max())])

correctly\_classified\_count = total\_count - misclassified\_count

misclassification\_rate = misclassified\_count / total\_count if total\_count > 0 else 0

y\_position = y\_positions[i]

plt.text(

x=cluster\_centers\_sorted[i],

y=y\_position,

s=(f"Cluster {cluster\_id}\n"

f"Misclassification Rate: {misclassification\_rate:.2f}\n"

f"Correctly Classified: {correctly\_classified\_count}\n"

f"Misclassified: {misclassified\_count}"),

horizontalalignment='center',

fontsize=10,

bbox=dict(facecolor='white', alpha=0.8)

)

plt.legend()

plt.show()

# Function to create binned analysis

def create\_binned\_analysis(bin\_size):

bins = np.arange(LUMINANCE\_RANGE[0], LUMINANCE\_RANGE[1] + bin\_size, bin\_size)

bin\_centers = (bins[:-1] + bins[1:]) / 2

correct\_counts, \_ = np.histogram(correct['Luminescence'], bins=bins)

incorrect\_counts, \_ = np.histogram(incorrect['Luminescence'], bins=bins)

total\_counts = correct\_counts + incorrect\_counts

misclassification\_rates = np.divide(incorrect\_counts, total\_counts, out=np.zeros\_like(incorrect\_counts, dtype=float), where=total\_counts!=0)

return bin\_centers, correct\_counts, incorrect\_counts, total\_counts, misclassification\_rates

# Create plots for different bin sizes

bin\_sizes = [5, 7, 10]

fig, axes = plt.subplots(len(bin\_sizes), 1, figsize=(14, 6\*len(bin\_sizes)), sharex=True)

fig.suptitle('Luminescence Analysis with Different Bin Sizes', fontsize=16)

for i, bin\_size in enumerate(bin\_sizes):

bin\_centers, correct\_counts, incorrect\_counts, total\_counts, misclassification\_rates = create\_binned\_analysis(bin\_size)

ax = axes[i]

bar\_width = bin\_size

ax.bar(bin\_centers, correct\_counts, width=bar\_width, color='blue', label='Correctly Classified')

ax.bar(bin\_centers, incorrect\_counts, width=bar\_width, bottom=correct\_counts, color='red', label='Misclassified')

ax2 = ax.twinx()

ax2.plot(bin\_centers, misclassification\_rates, color='green', marker='o', linestyle='-', label='Misclassification Rate')

ax.set\_xlabel('Luminescence')

ax.set\_ylabel('Number of Images')

ax2.set\_ylabel('Misclassification Rate')

ax.set\_title(f'Bin Size: {bin\_size}')

ax.legend(loc='upper left')

ax2.legend(loc='upper right')

ax.grid(True)

ax.set\_xlim(LUMINANCE\_RANGE)

ax2.set\_ylim(0, 1)

plt.tight\_layout()

plt.show()

# Print cluster statistics

print("\nCluster Statistics:")

print(f"{'Cluster':<10} {'Size':<10} {'Range':<30} {'Center':<15} {'Misclassification Rate':<25}")

for i, (center, range\_, rate) in enumerate(zip(cluster\_centers, cluster\_ranges, cluster\_misclassification\_rates)):

cluster\_size = len(incorrect[incorrect['cluster'] == i])

print(f"{i:<10} {cluster\_size:<10} {str(range\_):<30} {center:<15.2f} {rate:.2f}")

**explanation of the code:**

This code performs an analysis on luminescence data to understand the patterns of misclassification using clustering and binning techniques. The key steps involve data preprocessing, clustering using Affinity Propagation, visualizing the clusters, performing binned analysis, and generating insightful plots. Here is a step-by-step explanation:

**Step 1: Import Libraries**

* **pandas** is used for data manipulation.
* **numpy** is used for numerical computations.
* **seaborn** and **matplotlib** are used for data visualization.
* **AffinityPropagation** from sklearn is used for clustering.
* **StandardScaler** from sklearn is used for scaling the data.

**Constants**

* This constant defines the range of luminescence values. It is used later when creating binned analysis plots.

**Step 2: Load and Preprocess Data**

* The data is loaded from a CSV file. The 'Correct Prediction' column is converted from 'Yes'/'No' to Boolean (True/False).
* The data is then split into two subsets: correctly classified data (correct) and misclassified data (incorrect).

**Step 3: Clustering Misclassified Data with Affinity Propagation**

* The clustering focuses on the 'Luminescence' feature of the misclassified data.
* The luminescence values are standardized using StandardScaler.
* Affinity Propagation is then applied to find clusters, and the cluster labels are added to the incorrect DataFrame.
* The number of clusters found is printed.

**Step 4: Calculate Cluster Statistics**

* For each cluster, the code computes:
  + **Cluster Center:** The mean luminescence value of the cluster.
  + **Cluster Range:** The minimum and maximum luminescence values in the cluster.
  + **Misclassification Rate:** The ratio of misclassified images to the total images within the cluster’s range.

**Step 5: Plot Clustering Results**

* This section visualizes the clustering results.
* The misclassified data is shown as points colored by cluster.
* Correctly classified data is plotted below the x-axis for contrast.
* Annotations are added for each cluster, showing the misclassification rate, the number of correctly classified, and misclassified images.

**Step 6: Binned Analysis Function**

* This function performs a binned analysis of luminescence values.
* It divides the luminescence range into bins of a specified size.
* For each bin, it calculates the number of correctly classified and misclassified images and the corresponding misclassification rate.

**Step 7: Plot Binned Analysis Results for Different Bin Sizes**

* This section generates subplots to analyze how different bin sizes impact the visualization.
* Each subplot shows the correctly classified (blue) and misclassified (red) counts as stacked bars.
* The misclassification rate is plotted as a green line.

**Step 8: Print Cluster Statistics**

* Finally, this section prints a summary table of each cluster, displaying the size, range, center, and misclassification rate for each cluster.

The code performs a detailed analysis of luminescence data by:

1. Clustering misclassified samples using Affinity Propagation.
2. Calculating statistics and visualizing the clusters.
3. Conducting a binned analysis to see how the misclassification rate changes across different luminescence ranges.
4. Providing a clear summary of cluster statistics.

Visualization of obtained outputs:

A screen shot of a graph

Description automatically generated

Using binsize 5:

A graph with green and red lines

Description automatically generated

Using binsize 7:

A graph with a line graph

Description automatically generated with medium confidence

Using binsize 10:

A graph with red and blue squares

Description automatically generated