(520|600).666

Information Extraction

Homework # 1

Q1. Read Chapter 1 from the Jelinek book.

Answer: Done

Q2. Write a one-page summary of the article

Answer:

Article provides a comprehensive review of the principles, architecture, and current state of large vocabulary continuous speech recognition (LVCSR) systems as of 1994. LVCSR refers to systems that can automatically transcribe natural continuous speech from any speaker using a large vocabulary.

Statistical Pattern Recognition Foundations: It explains that modern LVCSR systems are firmly grounded in the statistical pattern recognition techniques pioneered by researchers at IBM in the 1970s/80s. The goal is to find the word sequence W that maximizes the probability P(W|Y) given the observed acoustic speech signal Y. Using Bayes' rule, this is decomposed into the prior probability P(W) given by a language model and the likelihood P(Y|W) computed by an acoustic model.

Front-End Speech Parameterization: A key requirement is to convert the speech waveform into a compact sequence of acoustic vectors representing the short-time speech spectrum. The front-end needs to extract all necessary information to enable effective subsequent pattern matching. Popular features include cepstral coefficients derived from Fourier or LPC spectra. The mel-scale non-linear frequency resolution, log compression, temporal derivatives, and DCT transforms help satisfy acoustic modeling assumptions.

Acoustic Modeling: we need to find Y given w. for large sequence we decompose it to phones. HMMs model the sounds (phones) of speech. Triphone HMMs capture preceding and succeeding contextual effects. Tying acoustically similar states enables robust parameter sharing. Gaussian mixture models are now favored over discrete distributions. Tying parameters and using decision trees to determine state clustering mitigates overfitting given limited training data. The forward-backward algorithm efficiently computes output probabilities while Baum-Welch finds maximum likelihood estimates.

Language Modeling: N-gram language models effectively encode local word order constraints. Discounting redistributes probability mass from more frequent to less frequent N-grams. Backing

off handles unseen contexts. Despite limitations in capturing longer range dependencies, N-grams remain dominant due to computational efficiency and smoother model estimates.

Decoding: For searching best W, we need decoder to find it. There are 2 ways breadth first and depth first. The Viterbi search finds the optimal word sequence in reasonable time by pruning low probability hypotheses early. Beam search performs well with trigram language models and crossword context dependent acoustics but requires efficient early application of constraints. Tree-copy approaches meet this need.

Current Performance: Results from the 1993 ARPA benchmark evaluation are summarized to exemplify the state-of-the-art LVCSR performance levels. The best system achieved 7.2% average word error rate on speech dictation where 7 words in every 100 were mis transcribed. However, performance varied widely across speakers, with some speakers having over 20% error rates.

Key Challenges: In conclusion, while accuracy on dictated speech in controlled conditions is approaching usable levels, particularly for adapted systems, several key challenges remain to achieve widespread robust LVCSR deployment: Speaker adaptation - unsupervised incremental methods needed, Noise robustness - compensation insufficient thus far , Task dependence - language model adaptation required, Spontaneous speech - much worse accuracy levels currently, Computational efficiency - 100x+ faster decoding needed

Q3. Vector Quantization Code

Answer:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

# Function to perform vector quantization
def vector_quantize(dp_numpy, n_clusters):

    # Initialize centroids randomly
    #centeroids = dp_numpy[np.random.choice(len(dp_numpy), n_clusters, replace=False)]
    initial_centroid_indices = [0, 50, 99]  # Adjust these indices based on your preferences
    centeroids = dp_numpy[initial_centroid_indices]
    print("Initial Centers:", centeroids)

    # Initialize empty list for cluster labels and set iteration count to 0
    old_label = []
    iteration_count = 0

    # Iterate until convergence
    while True:
```

```
    # Initialize empty list for new cluster labels
    new_label = []

    # Assign each vector to the nearest centroid

    for each_dp in dp_numpy:
        distances = []

        for each_c in centeroids:
            distance = np.linalg.norm(each_dp - each_c)
            distances.append(distance)

        closest_center_index = np.argmin(distances)
        new_label.append(closest_center_index)


    # Check for convergence by comparing with previous labels
    if np.array_equal(old_label, new_label):
        break

    # Update labels and increment iteration count
    old_label = new_label
    iteration_count += 1

    # Plot the current iteration
    plot_iteration(dp_numpy, old_label, centeroids, iteration_count)

    # Update centroids based on mean of points in each cluster
    for cluster_index in range(n_clusters):
        current_cluster_points = []

        for data_point_index in range(len(dp_numpy)):
            if old_label[data_point_index] == cluster_index:
                current_cluster_points.append(dp_numpy[data_point_index])

        if len(current_cluster_points) == 0:
            continue

        current_cluster_points = np.array(current_cluster_points)
        centeroids[cluster_index, :] = np.mean(current_cluster_points, axis=0)

# Print the number of iterations and return the final labels and centroids
print(f'Number of iterations: {iteration_count}')
return old_label, centeroids
```

```
# Function to plot the current iteration
def plot_iteration(vectors, old_label, centers, iteration_count):
    plt.figure()
    plt.scatter(vectors[:, 0], vectors[:, 1], c=old_label, cmap='viridis', marker='o', s=50)
    plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x', s=200, label='Centroids')
    plt.title(f'Iteration {iteration_count}')
    plt.legend()
    plt.show()

# Main function
def main():
    # Read data from file
    f = open("/Users/lavanya/Library/CloudStorage/OneDrive-
JohnsHopkins/Courses/Spring2024/InformationExtraction/HW1/hw1-data (1).txt", "r")
    cluster = 3
    dp_list = []

    # Parse data and convert to numpy array
    for line in f:
        line = line.strip().split()
        dp_list.append((float(line[0]), float(line[1])))

    dp_numpy = np.array(dp_list)

    # Perform vector quantization
    vector_quantize(dp_numpy, cluster)

# Execute main function if the script is run as the main program
if __name__ == "__main__":
    main()
```

(a) the common tendencies

Answer:

These are the trails:

Trial 1:
Initial Centers: [[0.4738 0.5406]
 [0.412 0.5122]
 [0.404 0.4533]]
Final Centers: [[0.63435   0.50358158]
 [0.51913548 0.63927742]

[0.41170323 0.48437419]]
Number of iterations: 9

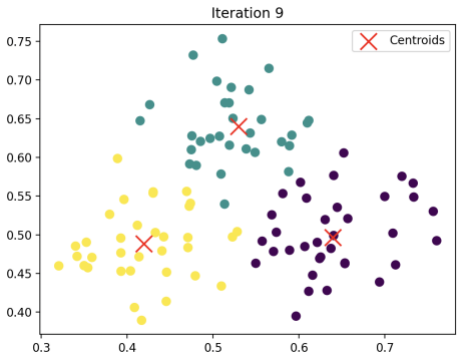
Trail 2:
Initial Centers: [[0.5891 0.615]
[0.5892 0.48]
[0.446 0.4516]]
Final Centers: [[0.53080588 0.63440294]
[0.63769706 0.49494118]
[0.41350625 0.4866125]]
Number of iterations: 3

Trial 3:
Initial Centers: [[0.657 0.5209]
[0.6116 0.6477]
[0.5213 0.6904]]
Final Centers: [[0.63834857 0.49862]
[0.52744194 0.64223226]
[0.41965882 0.48848235]]
Number of iterations: 5

Trail 4:
Initial Centers: [[0.7126 0.4611]
[0.4264 0.6681]
[0.5188 0.6704]]
Final Centers: [[0.64494242 0.49795455]
[0.41681515 0.48693333]
[0.52715588 0.63141176]]
Number of iterations: 3

Based on the multiple trials of k-means clustering, I can observe a few general tendencies:

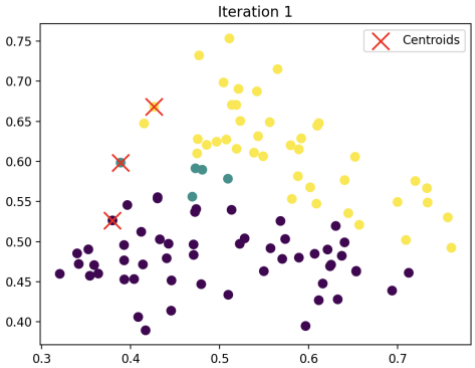
1. The clusters seem to converge towards similar final centers, even with different initializations. This suggests that k-means can identify the natural cluster in the data given.
2. 3 clusters emerging focused on left lower left, upper middle, and right lower left center points.
3. Across the trials, the lower left cluster seems most stable in its center location. The other two shift slightly more between trials as the algorithm iterates.
4. Most trials converge in 3-6 iterations, indicating fast separation into the natural clusters. Only one trial took 9 iterations to stabilize.
5. This was the final graph in most of the final iteration



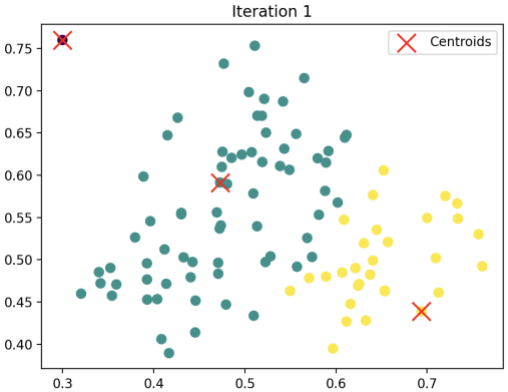
(b) the occasional outlier behavior of the clustering algorithm

answer:

1. based on the trails observed, even though an outlier was picked randomly on the first iteration, it was converged to a correct cluster in the end of the iteration.
2. For example, this was first iteration where a data point which is in the boundary was picked. But it was converged to a proper cluster in the end.



3. Since in the data provided there was not so many data point which was an extreme outlier. I modified the data and added an outlier. So, with these changes the k means algorithm was breaking and couldn't come out of it.
4. This is the graph, it was same even in the last iteration



Q4. Product Quantization

(a) Describe the space and assignment complexity of K-Means vs. product quantization.

Answer:

The Space Complexity for Kmeans is O(kD) where k= no of clusters and D = dimension of the vectors. For example, kD = 3056×128 = 391168. Whereas the assignment complexity is O(kD) where each data point needs to calculate the distance to the k centroids.

The Space Complexity for Product Quantization is mk*D* = O(k^(1/m))×D where k= no of clusters, m = no of sub-vectors and D = dimension of the vectors. For example: (k^(1/m))×D = (3048^(1/8))×128 = 349. As we see the space is reduced a lot when we apply product quantization. Whereas the assignment complexity is O(k^(1/m))×D) where each data point needs to calculate the distance to the subset of centroids.

Product quantizers are more memory-efficient than k-means, especially for large values of k. The assignment complexity of product quantizers is also lower compared to k-means.

(b) Using the above result, explain why this is useful.

Answer:

- As we saw, the space and assignment complexity is very low for product quantization.
- It as ability to significantly reduce memory requirements. By decomposing the vector space into subspaces and quantizing each sub vector independently, the overall memory footprint is substantially decreased. This makes it well-suited for applications dealing with large datasets and memory-constrained environments.
- It is especially effective in tasks that involve nearest neighbor search. The reduced assignment complexity allows for faster computation of distances between data points and centroids.
- It exhibits good scalability properties. As the dataset size or the number of clusters increases, the reduced computational complexity allows for efficient handling of larger data volumes.
- It provides a trade-off between memory efficiency and quantization precision. While there is some loss of accuracy due to quantization, the trade-off allows practitioners to fine-tune the approach based on the specific requirements of their application.