Evaluation of TPC-DC Active Learning for Low Budgets on CIFAR-10

Abdullah Al Mu'adh
King's College London
https://github.kcl.ac.uk/k20009401/ML_CW2

***Abstract*—Active Learning (AL) strategies often fail in low-budget, 'cold start' scenarios. The TypiClust algorithm [1] addresses this by selecting diverse and typical samples, contrasting with traditional methods that seek uncertain ones. This report details the implementation and evaluation of the TPC-DC (Deep Clustering) variant of TypiClust on CIFAR-10 across varying budget sizes. Performance was benchmarked against a random sampling baseline using the fully supervised framework [1]. Results demonstrate that TPC-DC provides a significant and consistent performance advantage over the random sampling. A modification using a cosine medoid selection strategy was also proposed and evaluated; while it outperformed the random baseline, it did not match the performance of the TPC-DC method.**

***Index Terms*—Active Learning, Deep Learning, Deep Clustering, TypiClust, Low Budget, CIFAR-10, SCAN, SimCLR, TPC-DC.**

I. Introduction

The cost of acquiring large, expertly-annotated datasets remains a key bottleneck in deep learning. Active Learning (AL) aims to mitigate this by selecting the most informative samples for labelling. However, many AL strategies fail in low-budget or 'cold start' scenarios and simple random sampling is more effective.

The work by Hacohen et al. [1] proposes that low-budget regimes benefit from an opposite strategy: selecting *typical* and representative samples to build a robust initial model, rather than querying uncertain ones. To implement this, TypiClust employs a two-stage process. First, it learns powerful feature representations from unlabelled data using a self-supervised pretext task. This report focuses on the TPC-DC variant, which utilizes the SCAN [3]. SCAN, in turn, leverages a contrastive learning framework like SimCLR [2] to learn its features by maximizing the agreement between different augmented views of the same image, clustering the data without labels. This report documents the implementation of TPC-DC on CIFAR-10, evaluates its performance against a random baseline, and presents a modification to its selection strategy.

II. Methodology

A. Original TPC-DC Algorithm

The TPC-DC variant was implemented as an iterative process. The core of the algorithm is a loop that progressively builds a labeled set (L) by selecting samples from a larger unlabeled pool (U). The batch size of new samples added at each step is fixed at $B = 10$.

The iterative steps are as follows:

1) **Representation Learning:** First, a pre-trained SimCLR model [2] is used to generate 512-dimension feature embeddings for the entire 50,000-image CIFAR-10 training set. This is a one-time step.
2) **Iterative Clustering and Selection:** The main active learning loop proceeds for each cumulative budget k (e.g., 10, 20, ..., 60). For each iteration i:
   a) **Re-clustering:** The entire dataset's embeddings are re-clustered into $k = |L_{i-1}| + B$ clusters using the SCAN algorithm [3]. For the first iteration, $k = 10$.
   b) **Identify Uncovered Clusters:** The algorithm identifies which of the k new clusters contain samples from the previously labeled set L_{i-1}. These are marked as "covered" clusters.
   c) **Typicality Selection:** From the remaining "uncovered" clusters, B new samples are selected. This is done by first finding the single most typical point within each uncovered cluster using the typicality formula [1]:

$$\text{Typicality}(x) = \left(\frac{1}{K} \sum_{x_i \in K\text{-NN}(x)} ||x - x_i||_2 \right)^{-1} \tag{1}$$

   Then, the B points with the highest typicality scores among these candidates are chosen.
   d) **Update Pools:** The B newly selected indices are added to the labelled set L and conceptually removed from U.

The effectiveness of the clustering step can be seen in t-SNE visualizations. Fig. 1 shows the clear partitioning of the data into 20 distinct clusters.

B. Baseline and Evaluation Frameworks

The TPC-DC selection was compared against a **Random Baseline** that selected k samples uniformly at random for each budget level. The visualization in Fig. 2 shows the class distribution for the random baseline at a budget of 30.

Performance was measured using the **Fully Supervised Framework**, where a ResNet-18 model is trained from scratch on the selected labelled samples for each budget level [1].

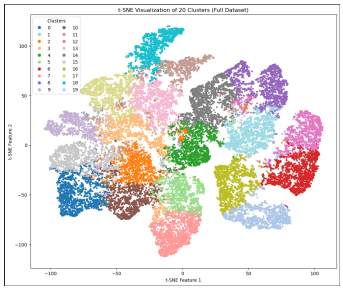


Fig. 1. t-SNE Visualization of 20 Clusters on the Full Dataset.

Fig. 2. Images Selected by typiclust (Budget = 30).

TABLE I
MEAN TEST ACCURACY (%) AND STANDARD DEVIATION ON CIFAR-10

Budget (k)	TPC-DC (TypiClust)	Medoid (Cosine)	Random Baseline
10	16.46 ± 0.84	12.79 ± 1.04	12.50 ± 1.13
20	17.07 ± 1.47	15.93 ± 0.71	12.51 ± 1.03
30	19.13 ± 0.70	18.07 ± 1.08	17.25 ± 0.78
40	21.46 ± 1.39	18.08 ± 0.54	16.51 ± 0.79
50	22.54 ± 1.14	19.09 ± 1.24	20.74 ± 1.24
60	23.77 ± 0.82	19.75 ± 1.29	19.49 ± 1.01

III. ALGORITHM MODIFICATION

A. Proposal: Cosine Medoid Selection

We propose a modification to the sample selection step (2c). Instead of using the paper's typicality formula, we select the **cosine medoid** from each uncovered cluster. The B medoids with the highest overall scores are then chosen.

B. Justification

This modification is proposed for two main reasons:

1) **Curse of Dimensionality:** In high-dimensional spaces like the 512-dimension embeddings, Euclidean distance can become less meaningful. Cosine similarity measures the angle between vectors, which can be more robust measure of semantic similarity.
2) **Global vs. Local Representation:** The paper's typicality formula is a *local* measure, considering only the k-nearest neighbours. The cosine medoid is a *global* measure of centrality, as it considers a point's average similarity to *all other points* in the cluster. This may lead to a more holistically representative sample selection.

IV. RESULTS AND DISCUSSION

The performance of the three selection strategies was evaluated across cumulative budgets from 10 to 60. Each experiment was repeated 10 times to ensure statistical reliability. The mean accuracy and standard deviation are summarized in Table I.

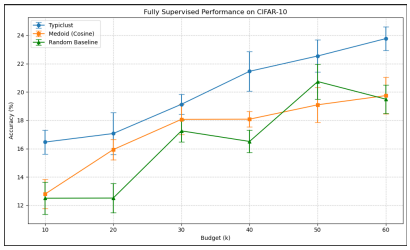


Fig. 3. Fully Supervised Performance on CIFAR-10.

A. TPC-DC vs. Random Baseline

As shown in Fig. 3 and Table I, the original TPC-DC algorithm consistently and significantly outperformed the random baseline. At the lowest budget of k=10, TPC-DC achieved a mean accuracy of 16.46%, nearly 4 percentage points higher than the random baseline's 12.50%. This performance gap was maintained and widened at higher budgets, with TPC-DC reaching 23.77% accuracy at k=60, compared to 19.49% for the random baseline. This result validates the central hypothesis of Hacohen et al. [1]: in low-budget scenarios, intelligently selecting typical and diverse samples is superior to random selection.

B. Evaluation of Modification

The proposed Cosine Medoid modification also consistently outperformed the random baseline, demonstrating that a global measure of centrality is a valid strategy. However, it did not manage to surpass the performance of the original TPC-DC algorithm. While competitive at lower budgets, the performance gap widened as the budget increased, with the Cosine Medoid method achieving 19.75% accuracy at k=60, over 4 percentage points lower than TPC-DC.

C. Impact and Limitations

The results strongly support using unsupervised feature learning to guide sample selection. The evaluation of the Cosine Medoid modification provided a valuable negative result, suggesting that for these fine-tuned embeddings, a *local* measure of density (k-nearest neighbors) is more effective than a *global* measure of cluster centrality (cosine medoid). This could be because the most informative "typical" samples exist in small, dense pockets within a larger cluster. A key limitation was restricting the evaluation to the fully supervised framework due to computational constraints.

V. CONCLUSION

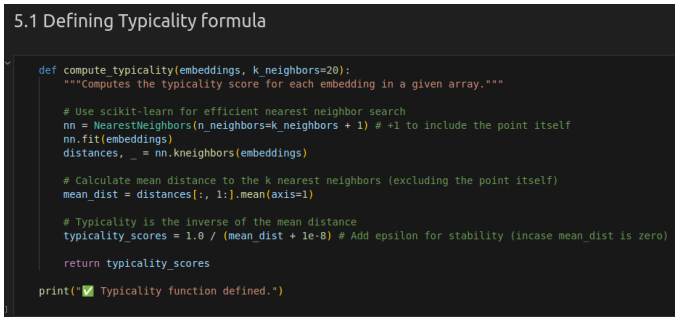
We successfully implemented the iterative TPC-DC active learning strategy, confirming its superiority over a random sampling baseline for low-budget supervised learning on CIFAR-10. A modification using a cosine medoid selection strategy was also evaluated. While this method also outperformed random selection, it did not achieve the performance of the original algorithm, suggesting that the local density estimation of the typicality formula is a more effective selection criterion in this context.

```
# Active Learning Loop using TypiClust
B = 10 # our budget
most_typical_points = []


# 0. Initialistion
labeled_indices = set(most_typical_points)
budgets_to_run = [10,20, 30, 40, 50, 60]

# Dictionary to store the cumulative list of indices at each budget
typiclust_cumulative_results = {10: list(labeled_indices)}

# 3. Iterative Selection Loop
for k in budgets_to_run:
    print(f"\n{'='*20} Running for Cumulative Budget k={k} {'='*20}")

    # a. Get the features and cluster assignments for the current budget
    current_features = all_features[k]
    current_assignments = all_cluster_assignments[k]

    # b. Identify which of the new 'k' clusters are "covered"
    covered_cluster_ids = set()
    for idx in labeled_indices:
        cluster_id = current_assignments[idx]
        covered_cluster_ids.add(cluster_id)
    print(f"Found {len(covered_cluster_ids)} covered clusters containing previously labeled points.")

    # c. Find the most typical point from each "uncovered" cluster
    candidate_points = [] # To store (typicality_score, original_index)

    for cluster_id in range(k):
        if cluster_id in covered_cluster_ids:
            continue # Skip covered clusters

        # Get data for this uncovered cluster
        original_indices_in_cluster = np.where(current_assignments == cluster_id)[0]
        if len(original_indices_in_cluster) < min_cluster_size:
            continue

        embeddings_in_cluster = current_features[original_indices_in_cluster]

        # Find the most typical point and its score
        typicality_scores = compute_typicality(embeddings_in_cluster, k_neighbors=k_neighbors)
        most_typical_local_idx = np.argmax(typicality_scores)
        highest_typicality_score = typicality_scores[most_typical_local_idx]
        most_typical_original_idx = original_indices_in_cluster[most_typical_local_idx]

        candidate_points.append((highest_typicality_score, most_typical_original_idx))

    print(f"Found {len(candidate_points)} candidate points from {k - len(covered_cluster_ids)} uncovered clusters.")

    # d. Select the top B candidates with the highest typicality scores
    # Sort candidates by typicality score in descending order
    candidate_points.sort(key=lambda x: x[0], reverse=True)

    # Select the top B (e.g., 10) indices
    newly_selected_indices = [index for score, index in candidate_points[:B]]

    # e. Update the labeled pool
    labeled_indices.update(newly_selected_indices)

    # f. Store the cumulative results
    typiclust_cumulative_results[k] = (list(labeled_indices))
    print(f"✅ Selected {len(newly_selected_indices)} new points. Total labeled points: {len(labeled_indices)}")
```

Fig. 4. TPC-DC implementation

```
from sklearn.metrics.pairwise import cosine_similarity

def select_medoids_iteratively(all_features, all_cluster_assignments,
                               budgets_to_run, initial_labeled_indices,
                               B=10, min_cluster_size=20):

    #1. Initialisation
    labeled_indices = set(initial_labeled_indices)
    cumulative_results_medoid = {10: sorted(list(labeled_indices))}

    # 2. Iterative Selection Loop
    for k in budgets_to_run:
        print(f"\n{'='*20} Running Medoid Selection for k={k} {'='*20}")

        current_features = all_features[k]
        current_assignments = all_cluster_assignments[k]

        # a. Identify covered clusters
        covered_cluster_ids = {current_assignments[idx] for idx in labeled_indices}
        print(f"Found {len(covered_cluster_ids)} covered clusters.")

        # b. Find the medoid from each "uncovered" cluster
        candidate_points = []

        uncovered_cluster_ids = [cid for cid in range(k) if cid not in covered_cluster_ids]

        for cluster_id in uncovered_cluster_ids:
            original_indices_in_cluster = np.where(current_assignments == cluster_id)[0]

            if len(original_indices_in_cluster) < min_cluster_size:
                continue

            embeddings_in_cluster = current_features[original_indices_in_cluster]

            # calculate cosine similarity matrix
            similarity_matrix = cosine_similarity(embeddings_in_cluster)

            # Calculate average similarity for each point (sum of a points similarity divided by total points in cluster)
            avg_similarity = similarity_matrix.mean(axis=1)

            # Find the medoid (point with max average similarity)
            medoid_local_idx = np.argmax(avg_similarity)
            highest_avg_similarity = avg_similarity[medoid_local_idx]
            medoid_original_idx = original_indices_in_cluster[medoid_local_idx]

            candidate_points.append((highest_avg_similarity, medoid_original_idx))

        print(f"Found {len(candidate_points)} candidate medoids from uncovered clusters.")

        # c. Select the top B candidates with the highest average similarity
        candidate_points.sort(key=lambda x: x[0], reverse=True)
        newly_selected_indices = [index for score, index in candidate_points[:B]]

        # d. Update the labeled pool
        labeled_indices.update(newly_selected_indices)

        # e. Store the sorted cumulative results
        cumulative_results_medoid[k] = sorted(list(labeled_indices))
        print(f"✅ Selected {len(newly_selected_indices)} new points. Total labeled points: {len(labeled_indices)}")

    return cumulative_results_medoid

✓ 0.0s
```

Fig. 5. Cosine medoid modification



```
5.1 Defining Typicality formula

def compute_typicality(embeddings, k_neighbors=20):
    """Computes the typicality score for each embedding in a given array."""

    # Use scikit-learn for efficient nearest neighbor search
    nn = NearestNeighbors(n_neighbors=k_neighbors + 1) # +1 to include the point itself
    nn.fit(embeddings)
    distances, _ = nn.kneighbors(embeddings)

    # Calculate mean distance to the k nearest neighbors (excluding the point itself)
    mean_dist = distances[:, 1:].mean(axis=1)

    # Typicality is the inverse of the mean distance
    typicality_scores = 1.0 / (mean_dist + 1e-8) # Add epsilon for stability (incase mean_dist is zero)

    return typicality_scores

print("✅ Typicality function defined.")
```

Fig. 6. Defined Typicality formula within paper[1]

REFERENCES

[1] G. Hacohen, A. Dekel, and D. Weinshall, "Active learning on a budget: Opposite strategies suit high and low budgets," in *Proc. 39th Int. Conf. Machine Learning (ICML)*, 2022.

[2] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proc. 37th Int. Conf. Machine Learning (ICML)*, 2020.

[3] W. Van Gansbeke, S. Vandenhende, S. Georgoulis, and L. Van Gool, "Scan: Learning to classify images without labels," in *Proc. European Conf. on Computer Vision (ECCV)*, 2020.