CSE251A Programming Project 1: Prototype selection for nearest neighbor

Marsol Torrent, Sergi (UC San Diego) [1]

Abstract

Nearest Neighbor classification achieves strong performance but scales poorly with training set size. Prototype selection mitigates this limitation by replacing the full training set with a small, representative subset that preserves accuracy while reducing computational cost. This work studies prototype selection for 1-Nearest Neighbor (1-NN) classification on MNIST, proposing a class-wise k-means-based strategy that emphasizes coverage of each class manifold while avoiding redundancy. Experiments across multiple prototype budgets show that coverage-oriented selection substantially outperforms uniform random sampling. Further refinements based on error-driven allocation and selective boundary-aware augmentation are evaluated through iterative experimentation.

1. Introduction

Nearest Neighbor (kNN) classification is a simple and effective non-parametric method, but its computational cost grows linearly with the size of the training set. In the 1-Nearest Neighbor (1-NN) setting, each test example is assigned the label of its closest training point under a distance metric, making storage and search efficiency critical (García et al., 2012).

This work focuses on the MNIST handwritten digit dataset, which consists of 60,000 training images and 10,000 test images of digits from 0 to 9. Each image is a 28×28 grayscale vector, resulting in a 784-dimensional feature space. The ten digit classes exhibit varying intra-class variability and inter-class similarity, making MNIST suitable for studying prototype selection strategies.

To reduce 1-NN computational cost while preserving accuracy, the full training set can be replaced by a carefully selected subset of representative examples, referred to as prototypes. Prototype selection has been extensively studied, including clustering-based strategies for k-NN (Vascon et al., 2013) and scalable optimization for large datasets (Plasencia-Calaña et al., 2017). The goal is to retain the essential data structure relevant for nearest neighbor classification. The following section introduces a coverage-oriented prototype selection strategy based on class-wise k-means clustering.

2. High-Level Prototype Selection Strategies

The objective is to reduce the size of the training set used by 1-Nearest Neighbor (1-NN) classification on MNIST while preserving accuracy. Prototype selection is treated as an iterative design process: starting from a coverage-oriented baseline, observations motivate progressively refined strategies that incorporate class difficulty, allocation stability, and decision boundary information. Each stage builds directly on the behavior observed in the previous one.

2.1. Class-wise k-means: coverage as a foundation

The initial prototype selection strategy is based on class-wise k-means clustering, a widely used approach for vector quantization and prototype selection in nearest neighbor systems (García et al., 2012; Vascon et al., 2013). For each digit class independently, k-means is applied to the training samples of that class, and the resulting cluster centroids are selected as prototypes. An equal number of prototypes is allocated to each class, ensuring all digits are represented regardless of their frequency or difficulty.

This approach formalizes prototype selection as a vector quantization problem: the centroids minimize the within-class squared distance between training samples and their nearest prototype. From the 1-NN perspective, this ensures each test example lies close to at least one prototype of the correct class. By spreading prototypes across each class's feature space, k-means captures dominant modes of variation while removing redundant points. The pseudocode for this initial approach is shown in Algorithm 1.

This coverage-driven baseline performs well because it preserves the geometric structure of each class manifold. However, allocating the same number of prototypes to all classes implicitly assumes uniform difficulty. Empirical results show that while easy digits are well represented, harder digits remain under-represented, suggesting coverage alone is insufficient when class complexity varies.

Algorithm 1 Equal-Allocation Class-wise k-means Prototype Selection

Input: training data (X, y), total prototypes M, number of classes C
Output: prototype set $(X_{\text{proto}}, y_{\text{proto}})$
Initialize $X_{\text{proto}} \leftarrow \emptyset$, $y_{\text{proto}} \leftarrow \emptyset$
Set $k \leftarrow \lfloor M/C \rfloor$ {prototypes per class}
for each class $c \in \{0, \ldots, C-1\}$ **do**
  Extract class data $X_c = \{x_i \mid y_i = c\}$
  Run k-means on X_c with k clusters
  for each cluster center μ **do**
    Find $x^* = \arg\min_{x \in X_c} \|x - \mu\|_2$
    Append x^* to X_{proto}
    Append label c to y_{proto}
  end for
end for
return $(X_{\text{proto}}, y_{\text{proto}})$

2.2. Error-driven allocation: adapting coverage to class difficulty

To account for differences in class difficulty, the prototype budget is reallocated based on observed per-class classification error. After evaluating 1-NN performance using the class-wise k-means prototypes, classes with higher error rates are assigned a larger share of the total prototype budget, while easier classes receive fewer prototypes. The total number of prototypes remains fixed. Algorithm 2 describes the single-pass error-driven reallocation strategy.

This strategy adapts coverage to difficulty: rather than representing all classes equally, it assigns more samples where the classifier struggles most. In practice, this improves accuracy for hard digits by providing finer-grained coverage of their manifolds. At the same time, it reveals an important trade-off: reducing prototypes for easy classes can slightly degrade their performance, indicating that some minimum coverage is still necessary even for well-separated classes.

These observations motivate the need for a more stable allocation mechanism that preserves the benefits of error-driven adaptation without sacrificing robustness.

2.3. Iterative error-driven reallocation with minimum coverage

To stabilize error-driven allocation, the reallocation process is applied iteratively. Starting from an initial equal allocation, per-class errors are computed, the prototype budget is redistributed accordingly, and class-wise k-means is rerun with the updated allocations. This process is repeated, recalculating errors after each iteration and adjusting the allocation until changes become negligible or a maximum number of iterations is reached.

Algorithm 2 Single-Pass Error-Driven Prototype Reallocation

Input: training data $(X_{\text{tr}}, y_{\text{tr}})$, test data $(X_{\text{te}}, y_{\text{te}})$, total prototypes M, minimum per-class budget $m_{\min}$
Output: prototype set $(X_{\text{proto}}, y_{\text{proto}})$
{Step 1: Initial equal allocation}
$(X^{(0)}_{\text{proto}}, y^{(0)}_{\text{proto}}) \leftarrow$ Algorithm 1$(X_{\text{tr}}, y_{\text{tr}}, M)$
Train 1-NN on $(X^{(0)}_{\text{proto}}, y^{(0)}_{\text{proto}})$
{Step 2: Estimate per-class errors}
for each class $c \in \{0, \ldots, C-1\}$ **do**
  Compute test accuracy a_c on samples with label c
  Set error $e_c \leftarrow 1 - a_c$
end for
{Step 3: Reallocate prototype budget}
Initialize allocation $k_c \leftarrow M \cdot \frac{e_c}{\sum_j e_j}$ for each class c
Enforce minimum budget: $k_c \leftarrow \max(k_c, m_{\min})$
if $\sum_c k_c > M$ **then**
  Scale all k_c proportionally so that $\sum_c k_c = M$
end if
{Step 4: Regenerate prototypes with new allocation}
$(X_{\text{proto}}, y_{\text{proto}}) \leftarrow$ Algorithm 1$(X_{\text{tr}}, y_{\text{tr}}, \{k_c\}_{c=0}^{C-1})$
return $(X_{\text{proto}}, y_{\text{proto}})$

The motivation for iteration is to allow the allocation to adapt as prototype placements improve: as hard classes receive more prototypes, their error decreases, potentially shifting difficulty toward other classes. To prevent easy classes from being progressively stripped of prototypes, a minimum number of prototypes per class is enforced at every iteration. This constraint guarantees a baseline level of coverage for all digits and prevents catastrophic degradation. Algorithm 3 extends the single-pass reallocation scheme by repeatedly updating the class prototype budget until convergence.

In practice, this iterative process converges quickly. After a small number of iterations, both the allocation and overall accuracy stabilize, and further updates produce only marginal or noisy changes. This behavior suggests that the remaining classification errors are not primarily due to insufficient prototype counts, but rather to limitations in where prototypes are placed. Once class-wise coverage has been optimized under reasonable constraints, reallocating prototypes alone is no longer sufficient to yield meaningful gains.

2.4. Selective boundary-aware augmentation: adding complementary information

Since further refinement of k-means centroids and allocation does not substantially improve performance, attention shifts away from improving global coverage and toward incorporating additional, complementary information. In

Algorithm 3 Iterative Error-Driven Prototype Reallocation

Input: training data $(X_{\text{tr}}, y_{\text{tr}})$, test data $(X_{\text{te}}, y_{\text{te}})$, total prototypes M, minimum per-class budget $m_{\min}$, max iterations T, convergence threshold ϵ
Output: prototype set $(X_{\text{proto}}, y_{\text{proto}})$, final allocation $\{k_c\}$
{Initialization}
Initialize allocation $k_c \leftarrow M/C$ for all classes c
Set $t \leftarrow 0$
repeat
  $t \leftarrow t + 1$
  Store previous allocation $k_c^{\text{old}} \leftarrow k_c$
  {Generate prototypes under current allocation}
  $(X_{\text{proto}}, y_{\text{proto}}) \leftarrow$ Algorithm 1$(X_{\text{tr}}, y_{\text{tr}}, \{k_c\})$
  {Estimate per-class errors}
  Train 1-NN on $(X_{\text{proto}}, y_{\text{proto}})$
  for each class $c \in \{0, \ldots, C-1\}$ **do**
    Compute test accuracy a_c
    Set error $e_c \leftarrow 1 - a_c$
  end for
  {Update allocation}
  Set $k_c \leftarrow M \cdot \frac{e_c}{\sum_j e_j}$ for each class c
  Enforce minimum budget: $k_c \leftarrow \max(k_c, m_{\min})$
  if $\sum_c k_c > M$ **then**
    Scale all k_c proportionally so that $\sum_c k_c = M$
  end if
until $\max_c |k_c - k_c^{\text{old}}| \leq \epsilon$ **or** $t = T$
{Final prototype generation}
$(X_{\text{proto}}, y_{\text{proto}}) \leftarrow$ Algorithm 1$(X_{\text{tr}}, y_{\text{tr}}, \{k_c\})$
return $(X_{\text{proto}}, y_{\text{proto}}), \{k_c\}$

particular, remaining errors are often concentrated near class boundaries, where centroids—by design—tend to be sparse and may fail to capture ambiguous regions of the feature space. This suggests that improving boundary representation, rather than global structure, may offer additional gains.

An initial attempt replaced a fraction of k-means prototypes directly with training points closest to inter-class decision boundaries, identified via nearest-enemy distances. However, this naive replacement substantially reduced performance, as many boundary points were noisy, unrepresentative, or overly specific to local variations, harming both easy and hard classes. This outcome highlighted that effective prototype sets must carefully balance boundary sensitivity with the need for stable global coverage.

To address this limitation, boundary-aware prototypes are introduced selectively for the hardest classes. Training samples are scored by their nearest-enemy distance, defined as the distance to the closest training point of a different class. Samples with small nearest-enemy distances are likely to lie near decision boundaries and represent difficult or ambiguous cases that are underrepresented by centroid-based prototypes. To avoid introducing noise, candidate boundary points are filtered using local label consistency, ensuring that they are not isolated or mislabeled samples.

Algorithm 4 Selective Boundary-Aware Prototype Refinement

Input: training data $(X_{\text{tr}}, y_{\text{tr}})$, test data $(X_{\text{te}}, y_{\text{te}})$, total prototypes M, number of hard classes H, boundary points per class B
Output: refined prototype set $(X_{\text{proto}}, y_{\text{proto}})$
{Phase 1: Error-driven prototype selection}
$(X_{\text{proto}}, y_{\text{proto}}, \{a_c\}, \{k_c\}) \leftarrow$ Algorithm 3$(X_{\text{tr}}, y_{\text{tr}}, X_{\text{te}}, y_{\text{te}}, M)$
{Phase 2: Identify hardest classes}
Compute per-class errors $e_c \leftarrow 1 - a_c$
Let $\mathcal{H}$ be the set of H classes with largest e_c
{Phase 3: Boundary-aware augmentation}
for each class $c \in \mathcal{H}$ **do**
  Identify candidate boundary samples in class c using nearest-enemy distance
  Select B closest boundary samples with local label consistency
  Replace B interior prototypes of class c with selected boundary samples
end for
return refined $(X_{\text{proto}}, y_{\text{proto}})$

Rather than modifying all classes, boundary-aware prototypes are added only to the hardest classes, replacing interior prototypes from the same class in a budget-neutral manner. Easy classes are left unchanged, preserving their stable coverage and strong baseline performance. This strategy introduces limited boundary sensitivity without discarding the coverage provided by k-means centroids. Algorithm 4 illustrates this final refinement stage, which selectively augments hard classes with boundary-aware prototypes.

Empirically, selective boundary augmentation avoids the performance degradation observed when boundary points are introduced globally (observed in intermediate experiment). However, improvements over the iterative allocation strategy remain modest. This indicates that, for MNIST, centroid-based coverage captures most of the useful structure for 1-NN classification, and that boundary information provides only limited additional benefit when added later.

3. Experimental Results

The prototype selection strategies were evaluated on the MNIST dataset for varying prototype budgets $M = 100, 200, 500, 1000, 2000, 5000, 10000$. Five approaches were compared:

Table 1. Classification accuracy of different prototype selection methods on MNIST for varying prototype budgets M. Random Stratified selection was repeated 10 times to compute standard deviations; other methods used a fixed seed for reproducibility.

Method	$M = 100$	$M = 200$	$M = 500$	$M = 1000$	$M = 2000$	$M = 5000$	$M = 10000$
Random Stratified	0.6555 ± 0.0202	0.7335 ± 0.0120	0.7980 ± 0.0062	0.8388 ± 0.0035	0.8650 ± 0.0035	0.8972 ± 0.0027	0.9146 ± 0.0026
Equal K-Means	0.7962	0.8206	0.8553	0.8749	0.8972	0.9091	0.9187
Error-Driven	**0.7962**	**0.8206**	0.8555	0.8849	**0.9030**	0.9141	0.9242
Iterative Error	0.7918	0.8191	**0.8619**	**0.8855**	0.8995	**0.9158**	**0.9247**
Selective Hybrid	0.7915	0.8177	0.8591	0.8833	0.8977	0.9143	0.9226

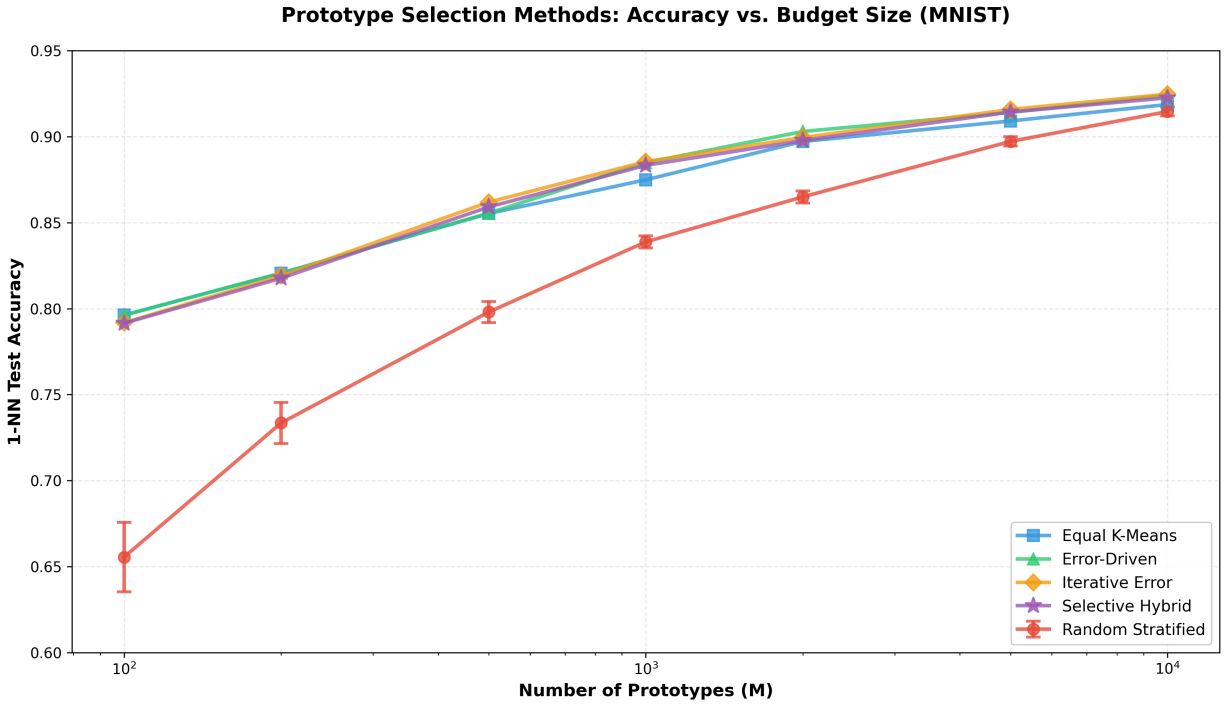

Figure 1. Classification accuracy of different prototype selection methods on MNIST for various prototype budgets M. Random stratified selection includes error bars (10 runs), while the other methods use a fixed k-means seed.

1. Completely randomized selection (baseline)
2. Equal class-wise k-means for coverage
3. Single-pass error-driven allocation
4. Iterative error-driven reallocation with minimum prototypes
5. Selective boundary-aware augmentation of hard classes

For the randomized baseline, each experiment was repeated $n = 10$ times with different seeds. The mean accuracy μ and standard deviation σ are computed as

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2},$$

where x_i is the accuracy of run i. Error bars in plots represent $\pm\sigma$. For deterministic k-means-based methods (equal-kmeans, error-driven, iterative, selective-hybrid), a fixed seed (42) is used, so only a single accuracy value is reported. A 95% confidence interval could be approximated as $\mu \pm 1.96 \cdot \sigma/\sqrt{n}$, but we report σ directly for simplicity.

Table 1 summarizes the overall classification accuracies and standard deviations for each approach across all M values. Figure 1 visualizes these results, highlighting trends in performance as the prototype budget increases.

It is immediately apparent that all the implemented prototype selection methods substantially outperform the randomized baseline, demonstrating the value of informed prototype choice. As the prototype budget M increases, the performance of the baseline approaches that of the other methods. With very small M, random selection is prone to picking poorly representative points, whereas the structured approaches provide consistent coverage and representation. For larger M, even the randomized baseline performs relatively well.

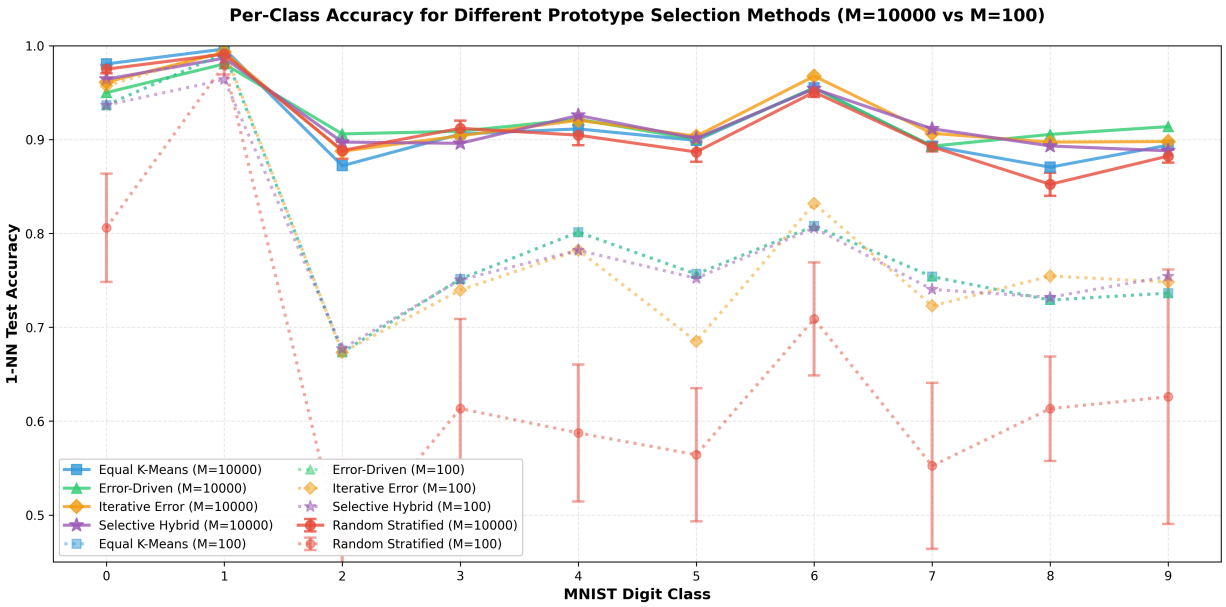

Figure 2. Per-class accuracies for MNIST using $M = 100$ and $M = 10000$. Random stratified selection includes error bars (10 runs), while the other methods use a fixed k-means seed.

Among the proposed approaches, error-driven allocation with class-wise k-means consistently improves upon equal allocation across classes. For small M, differences are negligible since few prototypes are available per class, but as M grows, allocating more prototypes to difficult classes becomes beneficial, resulting in a modest accuracy gain.

The iterative error-driven method yields slightly higher accuracy than single-pass error-driven allocation, particularly for higher M. By approximating an optimal number of prototypes per class while enforcing minimum coverage, it better balances class representation. However, this improvement comes at a significant computational cost: the iterative method takes roughly 10 times longer to run than the single-pass error-driven allocation, which itself requires approximately twice the runtime of the equal-allocation-per-class k-means baseline. Consequently, while the iterative approach can provide marginally higher accuracy, in most practical applications the single-pass error-driven allocation offers a better tradeoff between performance and runtime.

Finally, the selective hybrid approach, which augments difficult classes with boundary-aware prototypes, does not improve overall performance. In fact, it slightly reduces accuracy compared to the iterative error-driven method. While intended to improve representation near decision boundaries, these additional boundary points introduce misclassifications in other classes, outweighing any gains for hard cases. This indicates that, for MNIST, boundary-aware augmentation provides limited benefit and is not a reliable strategy for improving 1-NN performance.

Additionally, Figure 2 presents the per-class accuracies for $M = 100$ and $M = 10000$, highlighting extreme cases in the experiments and illustrating differences in class-level performance across methods. These two values of M were chosen to contrast scenarios with very few prototypes versus a large budget, allowing clear observation of how each approach manages class coverage and representation.

Several trends are evident from the figure. Some classes, such as digit 1, exhibit consistently high accuracy across all methods and M values, as they are visually distinct and rarely confused with other digits. In these cases, the error-driven allocation assigns very few prototypes to such easy classes, reflecting efficient use of the total budget.

Other classes, like digits 6 or 7, are more difficult to classify and show substantial improvement as M increases. For low M, the proposed k-means-based approaches provide much better accuracy than the randomized baseline, demonstrating the advantage of selecting representative prototypes. For higher M, the differences narrow, indicating that with a sufficiently large prototype set, careful selection becomes less critical.

Finally, certain classes, such as digit 0, show that k-means-based methods maintain relatively stable accuracy across different M, whereas the randomized baseline exhibits greater variability. This highlights the consistency of the structured prototype selection approaches and their ability to adapt efficiently to different prototype budgets, ensuring reliable performance even when M is small.

4. Conclusions

This study explored several prototype selection strategies to reduce the training set size for 1-Nearest Neighbor (1-NN) classification on MNIST while maintaining high accuracy. Across a wide range of prototype budgets M, the implemented approaches consistently outperformed random selection, demonstrating that carefully chosen prototypes provide better coverage of the class manifolds and more reliable representation of difficult classes.

The results confirm that the objectives of the project have been met: even with a substantially reduced training set, the proposed methods achieve very good and consistent classification accuracies. Among the structured strategies, the iterative error-driven approach achieved the highest scores, closely followed by the single-pass error-driven allocation. Considering runtime tradeoffs, the latter is generally preferred, as it provides nearly the same accuracy while requiring significantly less computation.

Depending on the application and desired tradeoff between runtime and accuracy, the prototype budget M can be adjusted. Larger M generally improves accuracy but increases computation, while smaller M may be sufficient when efficiency is critical. Notably, with $M = 2000$, all proposed methods already achieve approximately 90% test accuracy consistently, indicating that substantial reduction in training data can be achieved without sacrificing performance.

A natural next step would be to explore adaptive, data-dependent prototype refinement during training. For example, prototypes could be iteratively updated based on incoming test examples or misclassified points, allowing the prototype set to better adapt to challenging regions of the input space. Another direction could involve combining coverage and boundary-aware strategies with feature-space dimensionality reduction, which may improve efficiency and robustness for larger or more complex datasets. These extensions could further enhance the accuracy-runtime tradeoff and generalize the proposed methods beyond MNIST.

Software

The implementation consists of three main Python scripts. `select_prototypes.py` selects M prototypes using one of five methods (random, equal-kmeans, error-driven, iterative, or selective-hybrid) with a fixed seed for reproducibility. `run_experiments.py` executes experiments across all methods for a given M, outputs overall and per-class accuracies, and estimates error bars for the random baseline. `proj1.py` contains the core algorithm implementations, including k-means clustering, error-driven allocation, iterative refinement, and boundary-point selection. Detailed usage and parameter descriptions are provided in the included `README.md`.

AI Usage Statement

AI tools were used to assist in formatting and organizing the code, adding comments, generating the `README.md`, and refining technical writing. No source material or original research content was generated by AI.

References

García, S., Derrac, J., Cano, J. R., and Herrera, F. Prototype selection for nearest neighbor classification: taxonomy and empirical study. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(3):417–435, 2012.

Plasencia-Calaña, Y., Orozco-Alzate, M., Méndez-Vázquez, H., and Duin, R. P. Scalable prototype selection by genetic algorithms and hashing. *arXiv preprint arXiv:1712.09277*, 2017.

Vascon, S., Cristani, M., Pelillo, M., and Murino, V. Using dominant sets for k-nn prototype selection. In *Image Analysis and Processing – ICIAP 2013*, volume 8157 of *Lecture Notes in Computer Science*, pp. 131–140. Springer, 2013.