K-Means and NearMiss based Prototype Selection for Nearest Neighbor Classifier

Jintong Luo
jil386@ucsd.edu

Abstract

Accelerating Nearest Neighbor classification can be accomplished by replacing the extensive training set with a specially selected subset of 'prototypes'. To perform this prototype selection, we applied K-Means clustering and the NearMiss algorithm, targeting the dual challenges of large dataset management and imbalanced class distribution that typically impede NN classifier performance. The chosen prototypes were utilized for 1-NN classification and rigorously tested on the MNIST dataset.

1 Description

In this project, we concentrate on developing prototype subsets from the original training data that accurately embody the dataset during KNN inference. Our prototype selection mechanism ensures that each prototype mirrors the specific traits and distribution of its respective category. Utilizing K-Means clustering, we establish centroids that compactly encapsulate data points, allowing for a balanced and representative dataset with a limited number of prototypes. Yet, to address the complexities of large datasets and class imbalances, we incorporate the NearMiss algorithm. This algorithm adeptly highlights data outlines, catering to the nuances of imbalanced categories and outlier data crucial for KNN effectiveness. Our experiments deploy these prototype selection strategies on the MNIST dataset, using varying prototype counts to validate their accuracy, ultimately enhancing KNN classifier performance through precise prototype representation.

2 Pseudo-code

2.1 K-Means and NearMiss Prototype Selection

Input: Labelled Training Set of size N, Input Features: $X[N]$, Output Labels: $y[N]$

Returns: Prototype set of size M, Input Features: $X_{sampled}[M]$, Output Labels: $y_{sampled}[M]$.

1. Retrieve ratio of each category in the Labelled Training Set:
   - Balanced retrieval:

     $$ratio\_of\_label[label] = \frac{1}{\textit{Num of the labels in y[N]}}$$

   - Weighted retrieval:

     $$ratio\_of\_label[label] = \frac{\textit{Count of the label in y[N]}}{N}$$

2. Get the corresponding prototype size of each category in $y_{sampled}[M]$:
   For each label i:

   $$prototype\_size[i] = floor(\text{ratio\_of\_label}[i] \times M) + I(i < M - \Sigma_i floor(\text{ratio\_of\_label}[i] \times M))$$

3. Performing Prototype Selection corresponding to the number required for each label:
   - K-Means:
     For each label i:
     - $Index = \textit{Get\_index}(y[N] = i)$
     - $Centroids[prototype\_size[i]] = \textit{K-Means}(X[Index])$
     - For each Centroids k:

       $$Index_{sampled} = \textit{Get\_index}(\textit{Get\_Nearest}(Centroids[k], X[Index]))$$

     - Get result

       $$X_{sampled}[i] = X[Index_{sampled}],$$
       $$y_{sampled}[i] = y[Index_{sampled}]$$

   - NearMiss:
     For each label i:

- For each instance in the majority class, find the $prototype\_size[i]$ nearest instances from the minority class.
- Select the majority class instances with the largest average distance to the nearest minority class instances.

4. Combine prototypes obtained from step 3, for all labels

2.2 Mixed Prototype Selection

Input: Labelled Training Set of size N, Input Features: $X[N]$, Output Labels: $y[N]$, Predefined K-Means Prototype ratio: k
Returns: Prototype set of size M, Input Features: $X_{sampled}[M]$, Output Labels: $y_{sampled}[M]$.

1. Get the corresponding prototype size of K-Means and NearMiss:

$$prototype\_size[\textit{K-Means}] = floor(k \times M)$$
$$prototype\_size[\textit{NearMiss}] = M - prototype\_size[\textit{K-Means}]$$

2. Performing Prototype Selection of K-Means and NearMiss:

   - K-Means:

$$X_{sampled}[\textit{K-Means}], y_{sampled}[\textit{K-Means}] = \textit{K-Means Prototype Selection}(X[N], y[N])$$

   - NearMiss:

$$X_{sampled}[\textit{NearMiss}], y_{sampled}[\textit{NearMiss}] = \textit{NearMiss Prototype Selection}(X[N], y[N])$$

3. Combine prototypes obtained from step 2

$$X_{sampled} = X_{sampled}[\textit{K-Means}] + X_{sampled}[\textit{NearMiss}],$$
$$y_{sampled} = y_{sampled}[\textit{K-Means}] + y_{sampled}[\textit{NearMiss}]$$

3 Experimental result

In our experimental setup, we subject M prototypes to 1-NN classification. We take a comparative analysis across various hyperparameter configurations, evaluating the performance against several prototype selection methods. These methods include:

- Baseline: Utilizing the entire original dataset without prototype selection.
- RandomSelectionPrototype: Randomly choosing prototypes without consideration of balance or distribution.
- BalancedKMeansPrototype: Prototypes selected via K-Means clustering using Balanced ratio retrieval.
- WeightedKMeansPrototype: Prototypes selected via K-Means clustering using Weighted ratio retrieval.
- BalancedNearMissPrototype: NearMiss algorithm to perform under-sampling with Balanced ratio retrieval.
- WeightedNearMissPrototype: NearMiss algorithm to perform under-sampling with Weighted ratio retrieval.
- WeightedMixedPrototype: A hybrid approach combining K-Means and NearMiss algorithms, with weighting applied.

We set the hyperparameter for the number of prototypes M to a range of values: [20000, 10000, 5000, 1000, 500, 100]. To mitigate the influence of randomness in prototype selection, we execute the selection process using 10 different random seeds. The outcomes are then assessed based on the average accuracy and standard deviation across these iterations. Result shows in Table1, Figure 1 & 2.

4 Critical evaluation

4.1 Analysis

- Accuracy: Since the accuracy of KMeansPrototype and WeightedMixedPrototype was significantly better than RandomPrototype in all size of prototype M, our method easily outperformed random selection.The method that used KMeansPrototype to choose prototypes was the most accurate. Since MNIST

Method	M	Mean Accuracy	Standard Deviation	Average Time
Baseline	-	0.9691	-	3.0429
RandomSelectionPrototype	20000	0.9581	0.0012	1.1863
	10000	0.9486	0.0012	0.6875
	5000	0.9363	0.002	0.4712
	1000	0.8837	0.0037	0.2891
	500	0.8518	0.005	0.2654
	100	0.7116	0.0192	0.2425
BalancedKMeansPrototype	20000	0.9607	0.0013	333.0524
	10000	0.9555	0.0014	156.8553
	5000	0.9497	0.0012	87.2003
	1000	0.9286	0.0011	28.6516
	500	0.9154	0.0018	18.1512
	100	0.8644	0.0029	10.2447
WeightedKMeansPrototype	20000	0.9607	0.0007	333.394
	10000	0.9553	0.0011	159.8586
	5000	0.9491	0.0016	86.9784
	1000	0.9276	0.0017	28.6211
	500	0.9151	0.0013	18.3937
	100	0.8649	0.0017	9.9622
BalancedNearMissPrototype	20000	0.9376	0.0	3.0216
	10000	0.9053	0.0	2.5243
	5000	0.8715	0.0	2.274
	1000	0.7487	0.0	2.0957
	500	0.6836	0.0	2.0608
	100	0.5372	0.0	2.0494
WeightedNearMissPrototype	20000	0.9373	0.0	2.9679
	10000	0.9017	0.0	2.4987
	5000	0.8689	0.0	2.2643
	1000	0.7453	0.0	2.078
	500	0.6814	0.0	2.042
	100	0.5345	0.0	2.0313
WeightedMixedPrototype	20000	0.96	0.0012	290.4631
	10000	0.9547	0.0014	137.7424
	5000	0.9462	0.0016	79.402
	1000	0.919	0.002	27.7602
	500	0.9044	0.0018	19.5524
	100	0.8518	0.0033	10.4598

Table 1: Performance comparison of prototype selection methods.

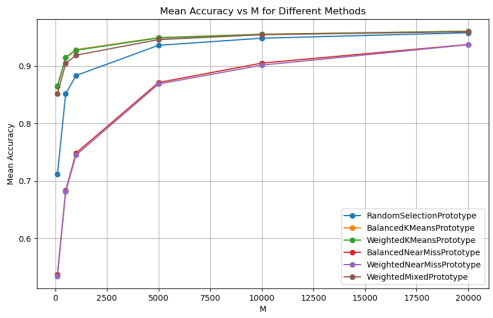

Figure 1: Mean Accuracy vs M for Different Methods

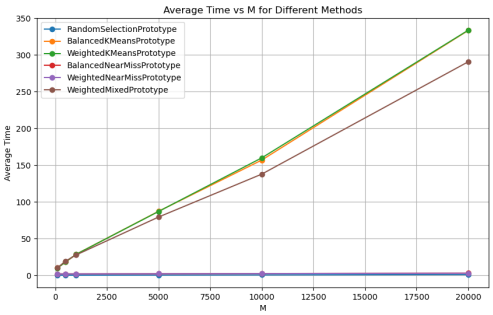

Figure 2: Average Time vs M for Different Methods

dataset do not have a significant imbalance on different categories, making Balanced and Weighted ratio retrieval not having a big difference. The WeightedMixedPrototype did just as well when we used more prototypes. On the other hand, the NearMissPrototype wasn't as accurate as just picking prototypes at random. This is because NearMiss pays more attention to the outer parts of the data, which means it doesn't represent the core as well. This also made the WeightedMixedPrototype less accurate when we use a small size of prototype M.

- Stability: We looked at how consistent the results were across different trials. The KMeansPrototype with a Weighted ratio led to a more consistent prototype selection, showing less variation in performance (lower standard deviation).

- Time Complexity: The NearMiss algorithm is faster than KMeans. So, when we combine it with KMeans in the WeightedMixedPrototype, it can greatly reduce the time needed, especially for a larger M. Hence, for large prototype sizes where accuracy is comparable, the WeightedMixedPrototype is preferable for its time efficiency.

4.2 Further scope

In our study using the MNIST dataset, which has a pretty even number of samples in each category and few outliers, the NearMiss algorithm didn't really stand out compared to KMeans. This is because the MNIST dataset is already well-balanced, and NearMiss is usually more helpful in datasets where some categories have a lot more samples than others.

However, if we were to try this on a dataset that was imbalanced, with large inner-category variance, our WeightedMixedPrototype method could likely do better than just using KMeansPrototype. This method combines the best parts of both NearMiss and KMeans, so it might be better at handling these more complicated situations, both in terms of accuracy and time complexity.

Appendix: Code

```
import numpy as np
from sklearn.neighbors import
    KNeighborsClassifier
from sklearn.cluster import KMeans
from sklearn.metrics import
    accuracy_score,
    classification_report
from tensorflow.keras.datasets import
    mnist
from scipy.spatial import distance
from imblearn.under_sampling import
    NearMiss
import matplotlib.pyplot as plt
import concurrent.futures
import random
import json
import time
import warnings
warnings.filterwarnings("ignore")
# Define Parameters
M = [20000, 10000, 5000, 1000, 500,
    100] # Number of samples to select
num_seeds = 10 # Number of different
    seeds for sampling
# M = [200] # Number of samples to
    select
# num_seeds = 2 # Number of different
    seeds for sampling
K = 1

# Load MNIST Dataset using Keras
(X_train, y_train), (X_test, y_test) =
    mnist.load_data()
X_train =
    X_train.reshape(X_train.shape[0],
    -1) / 255.0
X_test =
    X_test.reshape(X_test.shape[0], -1)
    / 255.0
```

```
class BasePrototype:
    def __init__(self, num_samples,
        seed):
        self.num_samples = num_samples
        self.seed = seed

    def get_samples(self):
        pass

    def get_result(self):
        knn =
            KNeighborsClassifier(n_neighbors=K)
        knn.fit(self.X_train_sampled,
            self.y_train_sampled)
        self.y_pred = knn.predict(X_test)
        return self.y_pred

    def analysis(self, full=False):
        accuracy = accuracy_score(y_test,
            self.y_pred)
        if full:
            print("Accuracy:", accuracy)
            print("Classification
                Report:\n",
                classification_report(y_test,
                self.y_pred))
        return accuracy

    def all(self, full=False):
        self.get_samples()
        self.get_result()
        return self.analysis(full)


class NoPrototype(BasePrototype):
    def get_samples(self):
        self.X_train_sampled = X_train
        self.y_train_sampled = y_train


class
    RandomSelectionPrototype(BasePrototype):
    def get_samples(self):
        np.random.seed(self.seed)
        indices =
            np.random.choice(X_train.shape[0],
            self.num_samples,
            replace=False)
        self.X_train_sampled =
            X_train[indices]
        self.y_train_sampled =
            y_train[indices]


def process_class(class_label,
    num_samples, seed):
    # print("K-Means of class {}
        started".format(class_label))
    # Isolate data for the current class
    class_data = X_train[y_train ==
        class_label]

    # Apply k-means
    kmeans =
        KMeans(n_clusters=num_samples,
        random_state=seed)
    kmeans.fit(class_data)
    centroids = kmeans.cluster_centers_

    # Find nearest samples to centroids
    nearest_samples = []
    for centroid in centroids:
        distances =
            distance.cdist([centroid],
            class_data, 'euclidean')
        nearest_index =
            np.argmin(distances)
        nearest_samples.append((class_data[nearest_index
            class_label))

    # print("K-Means of class {}
        finished".format(class_label))
    return nearest_samples


class KMeansPrototype(BasePrototype):
    # def assign_num_prototypes(self):
    #    pass

    def get_samples(self):
        sampled_training_data = []

        # print("Starting Parallel")
        # with
            concurrent.futures.ThreadPoolExecutor(max_wo
            as executor:
        #    futures =
            [executor.submit(process_class,
            class_label,
            self.num_samples_dict[class_label],
            self.seed)
        #            for class_label in
            np.unique(y_train)]
        #    for future in
            concurrent.futures.as_completed(futures):
        #
            sampled_training_data.extend(future.result()

        for class_label in
            np.unique(y_train):
            sampled_training_data.extend(process_class(cl
                self.num_samples_dict[class_label],
                self.seed))

        X_train_sampled, y_train_sampled
            = zip(*sampled_training_data)
        self.X_train_sampled =
            np.array(X_train_sampled)
        self.y_train_sampled =
            np.array(y_train_sampled)
        # self.X_train_sampled = []
        # self.y_train_sampled = []
        #
        # for class_label in
            np.unique(y_train):
        #    # Isolate data for the
            current class
        #    class_data = X_train[y_train
            == class_label]
        #
        #    # Apply k-means
        #    kmeans =
            KMeans(n_clusters=self.num_samples_dict[clas
            random_state=self.seed)
        #    kmeans.fit(class_data)
        #    print("K-Means of class
            {}".format(class_label))
```

```
        #    centroids =
             kmeans.cluster_centers_
        #
        #    # Find nearest samples to
             centroids
        #    for centroid in centroids:
        #       distances =
             distance.cdist([centroid],
             class_data, 'euclidean')
        #       nearest_index =
             np.argmin(distances)
        #
             self.X_train_sampled.append(class_data[nearest_index])
        #
             self.y_train_sampled.append(class_label)
        #
        # self.X_train_sampled =
             np.array(self.X_train_sampled)
        # self.y_train_sampled =
             np.array(self.y_train_sampled)

    def all(self, full=False):
        self.assign_num_prototypes()
        self.get_samples()
        self.get_result()
        return self.analysis(full)


class
    RandomKMeansPrototype(KMeansPrototype):
    def get_samples(self):
        self.X_train_sampled = []
        self.y_train_sampled = []

        kmeans =
            KMeans(n_clusters=self.num_samples,
            random_state=self.seed)
        kmeans.fit(X_train)
        # print("K-Means of random")
        centroids =
            kmeans.cluster_centers_

        # Find nearest samples to
            centroids
        for centroid in centroids:
            distances =
                distance.cdist([centroid],
                X_train, 'euclidean')
            nearest_index =
                np.argmin(distances)
            self.X_train_sampled.append(X_train[nearest_index])
            self.y_train_sampled.append(y_train[nearest_index])

        self.X_train_sampled =
            np.array(self.X_train_sampled)
        self.y_train_sampled =
            np.array(self.y_train_sampled)

    def all(self, full=False):
        self.get_samples()
        self.get_result()
        return self.analysis(full)


class BalancedSampler:
    def assign_num_prototypes(self):
        num_classes =
            len(np.unique(y_train))
        quotient = self.num_samples //
            num_classes
        remainder = self.num_samples -
            num_classes * quotient
        num_samples_per_class = [quotient
            + (_ < remainder) for _ in
            range(num_classes)]
        self.num_samples_dict = dict()
        for i, class_label in
            enumerate(np.unique(y_train)):
            self.num_samples_dict[class_label]
                = num_samples_per_class[i]


class WeightedSampler:
    def assign_num_prototypes(self):
        unique_classes, class_counts =
            np.unique(y_train,
            return_counts=True)
        class_ratios =
            np.floor(class_counts /
            y_train.shape[0] *
            self.num_samples).astype(int)
        remainder = self.num_samples -
            class_ratios.sum()
        self.num_samples_dict = dict()
        for i, (class_label, class_ratio)
            in
            enumerate(zip(unique_classes,
            class_ratios)):
            self.num_samples_dict[class_label]
                = class_ratio + (i <
                remainder)


class
    BalancedKMeansPrototype(KMeansPrototype,
    BalancedSampler):
    pass


class
    WeightedKMeansPrototype(KMeansPrototype,
    WeightedSampler):
    pass


class NearMissPrototype(BasePrototype):
    def __init__(self, *args, **kwargs):
        self.nearmiss_type =
            kwargs['nearmiss_type']
        kwargs.pop('nearmiss_type')
        super().__init__(*args, **kwargs)
    # def assign_num_prototypes(self):
    #    pass

    def get_samples(self):
        nearmiss =
            NearMiss(sampling_strategy=self.num_samples_
            version=self.nearmiss_type,
            n_neighbors=3,
                        n_neighbors_ver3=3)
        self.X_train_sampled,
            self.y_train_sampled =
            nearmiss.fit_resample(X_train,
            y_train)

    def all(self, full=False):
        self.assign_num_prototypes()
        self.get_samples()
        self.get_result()
```

```
        return self.analysis(full)

class BalancedNearMissPrototype(NearMissPrototype,
    BalancedSampler):
    pass

class WeightedNearMissPrototype(NearMissPrototype,
    WeightedSampler):
    pass

class WeightedMixedPrototype(BasePrototype):
    def __init__(self, *args, **kwargs):
        self.kmeans_ratio =
            kwargs['kmeans_ratio']
        kwargs.pop('kmeans_ratio')
        self.nearmiss_type =
            kwargs['nearmiss_type']
        kwargs.pop('nearmiss_type')
        super().__init__(*args, **kwargs)
        self.num_samples_kmeans =
            int(self.num_samples *
            self.kmeans_ratio)
        self.num_samples_nearmiss =
            self.num_samples -
            self.num_samples_kmeans
        self.nearmiss_prototype =
            WeightedNearMissPrototype(self.num_samples,
            self.seed,
            nearmiss_type=self.nearmiss_type)
        self.kmeans_prototype =
            WeightedKMeansPrototype(self.num_samples_kmeans,
            self.seed)

    def assign_num_prototypes(self):
        self.nearmiss_prototype.assign_num_prototypes()
        self.kmeans_prototype.assign_num_prototypes()

    def get_samples(self):
        self.nearmiss_prototype.get_samples()
        self.kmeans_prototype.get_samples()
        self.X_train_sampled =
            np.concatenate(
            (self.nearmiss_prototype.X_train_sampled,
                self.kmeans_prototype.X_train_sampled))
        self.y_train_sampled =
            np.concatenate(
            (self.nearmiss_prototype.y_train_sampled,
                self.kmeans_prototype.y_train_sampled))

    def all(self, full=False):
        self.assign_num_prototypes()
        self.get_samples()
        #
            self.kmeans_prototype.get_result()
        #
            self.kmeans_prototype.analysis(full)
        # print('\nabove
            kmeans\n-------------------------\nbelow
            nearmiss\n')
        self.get_result()
        return self.analysis(full)

works = {
    'RandomSelectionPrototype': lambda
        num_samples, seed:
    RandomSelectionPrototype(num_samples,
        seed).all(),
    # 'RandomKMeansPrototype': lambda
        num_samples, seed:
    # RandomKMeansPrototype(num_samples,
        seed).all(),
    'BalancedKMeansPrototype': lambda
        num_samples, seed:
    BalancedKMeansPrototype(num_samples,
        seed).all(),
    'WeightedKMeansPrototype': lambda
        num_samples, seed:
    WeightedKMeansPrototype(num_samples,
        seed).all(),
    'BalancedNearMissPrototype': lambda
        num_samples, seed:
    BalancedNearMissPrototype(num_samples,
        seed, nearmiss_type=1).all(),
    'WeightedNearMissPrototype': lambda
        num_samples, seed:
    WeightedNearMissPrototype(num_samples,
        seed, nearmiss_type=1).all(),
    'WeightedMixedPrototype': lambda
        num_samples, seed:
    WeightedMixedPrototype(num_samples,
        seed, nearmiss_type=1,
        kmeans_ratio=0.85).all()
}

print(works['WeightedMixedPrototype'](1000,
    100))
# WeightedMixedPrototype(10000, 100,
    nearmiss_type=1,
    kmeans_ratio=0.85).all(full=True)
# WeightedNearMissPrototype(10000,
    100,
    nearmiss_type=1).all(full=True)
# RandomSelectionPrototype(10000,
    100).all(full=True)
# WeightedKMeansPrototype(10000,
    100).all(full=True)
# NoPrototype(1000,
    100).all(full=True)

def plot(accuracies, std_deviation):
    # Optional: Plotting the Accuracies
        with Error Bars
    plt.errorbar(range(num_seeds),
        accuracies, yerr=std_deviation,
        fmt='o')
    plt.title("Accuracy with Error Bars
        for Different Training Seeds")
    plt.xlabel("Seed")
    plt.ylabel("Accuracy")
    plt.show()

def main():
    result_dict = dict()
    print(f"Using num_seeds={num_seeds}")

    start_time = time.time()
    baseline = NoPrototype(0, 0).all()
    end_time = time.time()
    avg_time = end_time - start_time
```

```
    print(f"Baseline, Mean Accuracy:
        {baseline}, Average Time:
        {avg_time}")
    result_dict['Baseline'] = (baseline,
        -1, avg_time)

    for work in works:
        result_dict[work] = dict()
        for num_samples in M:
            random_seeds =
                [random.randint(1, 10000)
                for _ in range(num_seeds)]
            accuracies = []
            start_time = time.time()

            # Step 2: Random Sample
                Selection and Training
            for seed in random_seeds:
                accuracies.append(works[work](num_samples,
                    seed))

            # Step 4: Calculate Accuracy
                and Error Bars
            mean_accuracy =
                np.mean(accuracies)
            std_deviation =
                np.std(accuracies)
            end_time = time.time()
            avg_time = (end_time -
                start_time)/len(random_seeds)

            print(f"{work}, M:
                {num_samples}, Mean
                Accuracy: {mean_accuracy},
                Standard Deviation:
                {std_deviation}, "
                 f"Average Time:
                     {avg_time}")
            result_dict[work][num_samples]
                = (mean_accuracy,
                std_deviation, avg_time)

        print('----------------------------------------')
        print(json.dumps(result_dict))
        print('----------------------------------------')


if __name__ == '__main__':
    main()
```