Simple and Scalable Sparse k-means Clustering via Feature ranking

Lloyd Fernandes (NETID : lloydf2)
Praveen Kumar (NETID :pkm4)
Omkar Mehta (NETID : omehta2)
Vincent Hoff (NETID : vhoff2)

December 2021

1 Introduction

In the context of data analysis and research, clustering is a technique for segmenting data into unique, homogeneous groups, with each group sharing similar characteristics. In any real-world data, only a handful of features are relevant to the problem of clustering, with the rest adding to the noise and the computational and memory requirements.

One of the most famous and widely used algorithms to perform clustering is the k-means algorithm due to its simplicity, speed, and familiarity. The k-means works by optimally assigning the data points to one of the k-nearest clusters according to the cluster's mean. A well-known algorithm to perform k-means is Lloyd's algorithm, where each feature is standardized to have zero mean and unit variance. The objective function of Lloyd's algorithm can be summarised with the following objective function.

$$\sum_{j=1}^{k} \sum_{\mathbf{x}_i \in C_j} ||\mathbf{x} - \theta_j||_2^2 = \sum_{i=1}^{n} \min_{1 \leq j \leq k} ||\mathbf{x} - \theta_j||_2^2, \tag{1}$$

where

$$\theta_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} \mathbf{x}_i.$$

Lloyd's algorithm, however, has the following issues, namely (i) the objective function in equation 1 (i.e., the within-cluster sum of squared error (WCSS)) is NP-hard (ii) the algorithm converges in a few iterations to the local optimum but not to the global optimum if the initial cluster centers are poorly assigned (iii) if the number of features in the data is very high, the signal to noise ratio decreases. As a result, the clustering accuracy decreases.

To alleviate the problem (iii) mentioned previously, many methods have been proposed which identify the proper subset of features from the data. These can be into the filter and wrapper-based methods. Filter methods are those where the subset of features is selected based on the clustering quality before clustering. These include entropy-based distances and Laplacian scores. Wrapper model methods (also called hybrid methods) first select subsets of attributes and finally select the subset post clustering based on certain cluster quality metrics. Clustering on Subset

of Attributes (COSA) is one such method. Here features are scaled with weights, such that the significant features are assigned higher weights. However, the COSA algorithm is not sparse in that it does not completely remove the noisy features. To incorporate sparsity, a new algorithm called Sparse k-means (SKM) was developed to reduce the feature space of clustering (achieved through ℓ-1 norm) and make the solution smooth, i.e., avoid $\mathbf{w}$ from having a single positive element (achieved through ℓ-2 norm) . The optimality is achieved using block descent. The objective of SKM is given by

$$\sum_{m=1}^{p} w_m \left(\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ijm} - \sum_{l=1}^{k} \frac{1}{|C_l|} \sum_{i,j \in C_l} d_{ijm} \right) \tag{2}$$

The algorithms proposed are computationally intensive, requiring large amounts of memory, and are complicated since they do not eliminate the uninformative features leading to the development of the more advanced algorithm called 'Sparse k-means with feature ranking' (SKFR) by Zhang et al., which we discuss in this paper [1]. SKFR adds a feature ranking procedure to eliminate uninformative features.

1.1 Pollard's population level loss function

Pollard uses the following population-level loss function to prove strong consistency in optimal centroids [2].

$$\Phi(\Theta, P) = \int \min_{\theta \in \Theta} ||\mathbf{x} - \theta||^2 P(d\mathbf{x}), \tag{3}$$

where $\mathbf{x}_n$ are independently sampled from a distribution P, and $\Theta \subset B_0(s)$ denotes a set of k points.

1.2 K-pod

In order to perform clustering on a dataset with missing features,[3] introduce k-pod with the objective of minimizing $\sum_{j=1}^{k} \sum_{i \in C_j} \sum_{(i,l) \in \Omega} (x_{il} - \theta_{jl})^2$. K-pod uses the majorization-minimization principle, wherein the objective is majorized by a surrogate $\sum_{i=1}^{n} \min_{1 \geq j \geq k} ||\mathbf{y}_i - \theta_j||_2^2$ and then minimizing this objective. Here $y_{il} = x_{il}$ when $(i, l) \in \Omega$, and $y_{il} = \theta_{il}$ when $(i, l) \notin \Omega$.

2 Main Summary

Zhang et al. proposes a novel method that incorporates feature selection into Lloyd's algorithm by adding a sorting step, preserving the algorithm's speed and scalability while directly addressing sparsity. The proposed algorithms forces the cluster centers, θ, into a s-sparse ball

$$B_0(s) := \{\mathbf{x} \in R^p, ||\mathbf{x}||_0 \leq s\}$$

with nonzero entries occurring along a shared set of feature indices. The projection into B_0 is achieved using feature ranking. SKFR1 has a global feature ranking wherein the feature importance is decided by calculating d_l after cluster centers were calculated.

$$d_l = \sum_{i}^{n} (x_{il} - 0)^2 - \sum_{j=1}^{k} \sum_{i \in C_j} (x_{il} - \mu_{jl})^2 = -\sum_{j=1}^{k} \mu_{jl}^2 |C_j| + 2 \sum_{j=1}^{k} \mu_{jl} \sum_{i \in C_j} x_{il} = \sum_{j} |C_j| \mu_{jl}^2$$

For SKFR2, the feature ranking is done locally for each cluster. The objective can be derived as

$$d_l = |C_j|\mu_{jl}^2.$$

Unlike the ℓ_1 norm, which induces sparsity in SKM by proxy, the sparsity parameter, s, is easily interpretable and can be inspired by the dataset and the problem at hand. The sparsity enforced by this algorithm ensures that important features are selected without undergoing any algebraic transformation, thus retaining the features and keeping the model interpretable. Other algorithms, like PCA, in concert with k-means, transform the original data while performing dimensional reduction.

Algorithm 1 SKFR1 algorithm pseudocode

Input: data $\mathbf{X} \in R^{n \times p}$ number of clusters k, sparsity level s, initial clusters C_j.
repeat
  for cluster j **do**
    $\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} \mathbf{x}_i$
  end for
  for feature l **do**
    Rank l by criterion $d_l = \sum_j |C_j|\mu_{jl}^2$
  end for
  Let L be the set of features l with $rank(l) \leq s$
  for sample i **do**
    Assign $\mathbf{x}_i$ to the cluster C_j such that j minimizes $\sum_{l \in L}(x_{il} - \mu_{jl})^2 + \sum_{l \notin L} x_{il}^2$
  end for
until convergence

Algorithm 2 SKFR2 algorithm pseudocode

Input: data $\mathbf{X} \in R^{n \times p}$ number of clusters k, sparsity level s, initial clusters C_j.
repeat
  for cluster j **do**
    $\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} \mathbf{x}_i$
    for feature l **do**
      Rank l by criterion $d_l = |C_j|\mu_{jl}^2$
    end for
    Let L_j be the set of features l with $rank(l) \leq s$
  end for
  for sample i **do**
    Assign $\mathbf{x}_i$ to the cluster C_j such that j minimizes $\sum_{l \in L}(x_{il} - \mu_{jl})^2 + \sum_{l \notin L} x_{il}^2$
  end for
until convergence

2.1 Properties

Zhang et al. establish the convergence of the proposed algorithms in terms of the objective function and the location of centroids.

2.1.1 Monotonicity of the objective

Each iteration of SKFR1 and SKFR2 monotonically decreases the k-means objective function

$$h(C, \theta) = \sum_{j=1}^{k} \sum_{\mathbf{x} \in C_j} ||\mathbf{x} - \theta_j||_2^2.$$

This monotonicity leads to the stability of these algorithms to a local optimum.

2.1.2 Strong consistency of centroids

Assume that for any neighborhood N of Θ^*, there exists $\eta > 0$ such that $\Phi(\Theta, P) > (\Theta^*, P) + \eta$ for every $\Theta \notin N$. Then, if Θ_n eventually lies in the same dimension as $\Theta^a st$ (i.e., the nonzero feature indices of Θ_n agree with $\Theta^a st$ as $n \to \infty$), we have $\Theta_n \xrightarrow{a.s} \Theta^*$ as $n \to \infty$.

2.2 Extensions to SKFR

2.2.1 Missing data

Missing features are usually handled by imputing missing features or deleting incomplete samples in a pre-processing step, leading to erroneous insertion of data and losing valuable information. Zhang et al. recommends using the k-pod algorithm for the SKFR algorithms, as it was implemented for Lloyd's algorithm and their optimization steps are the same.

2.2.2 Outliers

Treating outliers by trimming the outputs can be unwieldy on a high-dimensional dataset. Still, it can be assuaged by using the ℓ_1 norm in place of the ℓ_2 norm in the objective function, d_l, of SKFR1 and SKFR2 and using within-cluster median instead of within-cluster mean to identify cluster centers. It, however, requires d_l to be found by parsing through the entire dataset.

2.2.3 Choice of sparsity level

The sparsity level, s, in SKFR is interpretable and can be directly implemented in the algorithm if the sparsity level is known in advance. SKM, on the other hand, uses λ, which does not directly correspond to the level of sparsity needed. In cases when the sparsity level is a trainable parameter, the dataset can be randomly shuffled within each column of $\mathbf{X}$, say $\mathbf{X}_1, \ldots, \mathbf{X}_B$. The parameter s can be selected by maximizing the following :

$$s^* =_s Gap(s)$$

$$Gap(s) = \log O(\mathbf{X}, s) - \frac{1}{B} \sum_{b=1}^{B} \log O(\mathbf{X}_b, s)$$

where

$$O(\mathbf{X}, s) = \sum_{x_i \in \mathbf{X}} ||\mathbf{x}_i - \bar{\mathbf{x}}||_2^2 - \sum_{j=1}^{k} \sum_{i \in C_j} ||\mathbf{x}_i - \theta_{\mathbf{j}}||_2^2.$$

2.3 Flowcharts

The flow chart for SKFR1 can be seen in Figure1.

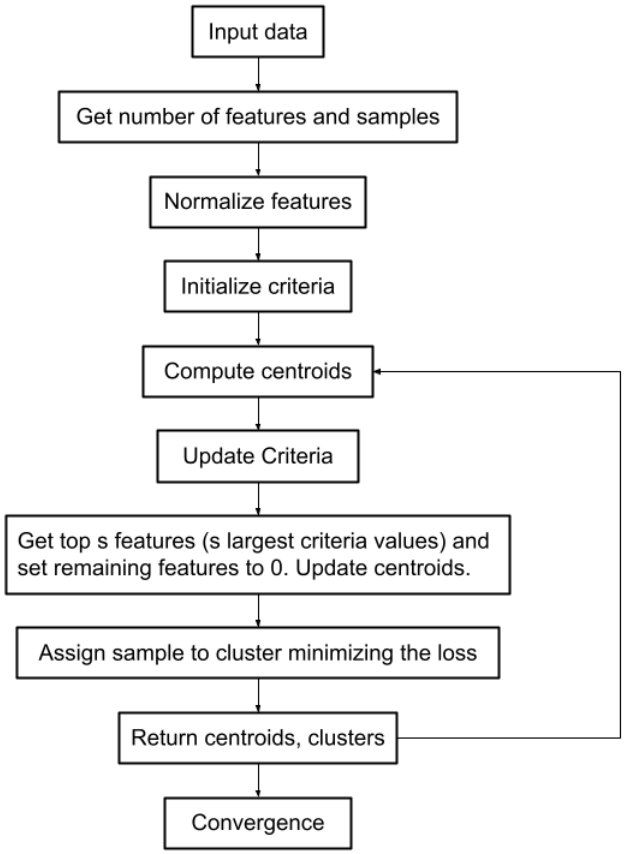

Figure 1: Flow Chart for SKFR1

The flow chart for SKFR2 can be seen in Figure2.

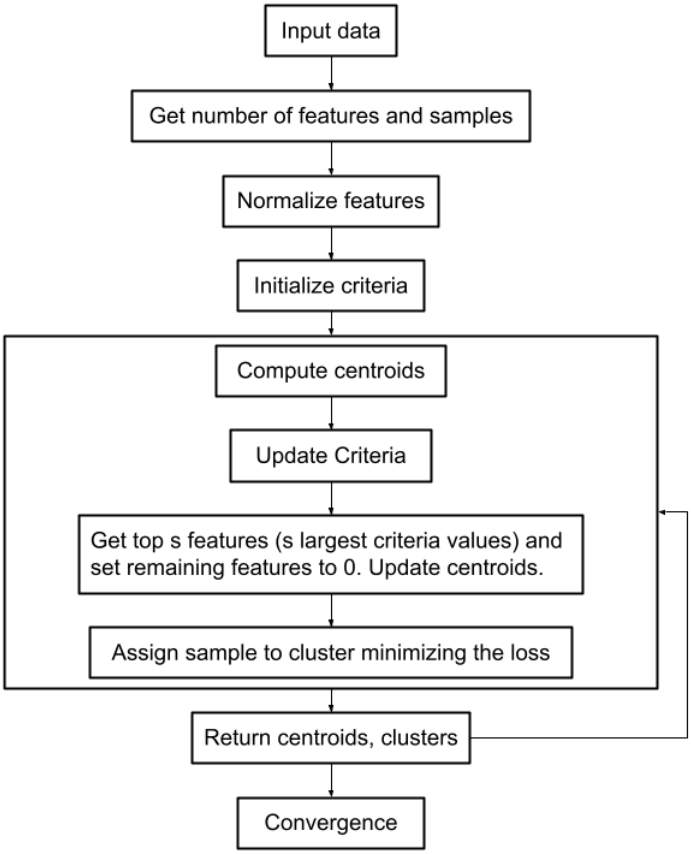

Figure 2: Flow Chart for SKFR2

3 Application

3.1 Algorithm Implementation

We would like to note that part of the SKFR1 algorithm was sourced from a Github repository [4]. The SKFR2 implementation described in this section is implemented by us and can be found in the appendix A.

For SKFR1 (the global version), the difference is $d_l = \sum_j |C_j|\mu_{jl}^2$ and for SKFR2 (the local version), the difference is $d_l = |C_j|\mu_{jl}^2$. SKFR2 allows for the set of relevant features to vary across clusters. The informative components are updated within each cluster j.

3.2 Data Collection and Creation

Testing of the SKFR1 and SKFR2 algorithm is done on randomly generated data. We have taken the following assumptions for our data: k (number of clusters) = 4, $features = 2$, $cases$ (samples) = 1000.

We are sampling normally distributed data so that each cluster has different means. For cluster

0, using the *np.random.normal* library with mean 0 and standard deviation 1, standard normal data is drawn at random using the features and cases to generate the shape. We increment the mean by four for the remaining clusters and keep the standard deviation at 2. Next, we negate the first row for cluster 1 and cluster 3. The code can be found in the appendix B.

3.3 Statistics of the application results and data

After implementing both algorithms on the generated data, the clustering assignments in Figure 3 are created.

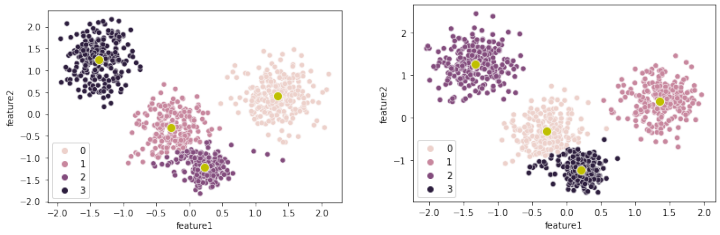

Figure 3: SKFR1 and SKFR2: Clusters

We confirm that the algorithms converge by looking at their loss plots. We report the loss shown in Figure 4, because in both algorithms, we assign a sample to the cluster such that the loss

$$\sum_{l \in L} (x_{il} - \mu_{jl})^2 + \sum_{l \notin L} x_{il}^2 \tag{4}$$

is minimized.

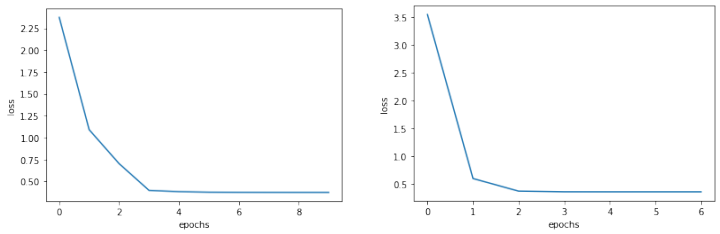

Figure 4: SKFR1 and SKFR2: Loss

While the authors have used the implementations on 10 benchmark datasets, we recreated some of the results for WDBC, Iris, Wine datasets. The clustering assignments are shown in Figure 5

While the clusters for WDBC dataset look ideal, the clusters for Iris and Wine datasets don't. Since the authors have not mentioned the sparsity values for each dataset, we tried our hands on different sparsity values.

3.4 Choice of Programming Language

The two most common choices for programming languages to solve data analysis problems, under which clustering falls, are Python and R. We implement the algorithm in Python rather than R.

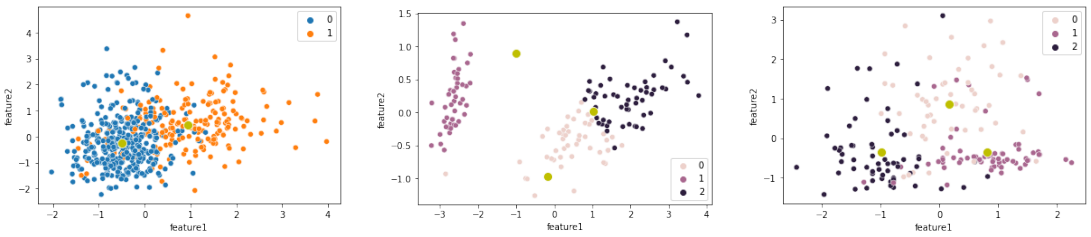

Figure 5: SKFR1 Clustering Assignments on WDBC, Iris, Wine (resp.)

One reason for this is the ease of implementing the PyTorch library for Python. As can be seen in our code, our implementation heavily depends on the use of this library. Another prevalent library that we use is the sci-kit-learn library. Scikit-learn has a unified interface for working with machine learning and data analysis methods and algorithms, of which there is usually only one implementation in Python. On the other hand, R has many smaller packages with unique algorithms that are often inconsistent in how they are accessed. The diversity in algorithms leads to a greater selection, with many coming directly from research labs. However, it comes at the cost of usability and ease of combining several parts.

3.5 Complexity Analysis

Cluster centers determine the rank of each of the features. Therefore there is no need to go through the entire dataset again to calculate d_l when $s = p$. The algorithm can be shown to be the same as Lloyd's algorithm. Therefore, the SKFR algorithms enjoy the same computational complexity as Lloyd's algorithm, $O(npk)$, as the sorting algorithm has a logarithmic computational cost.

4 Impact and Connection to Course

The algorithms preserve the interpretability and identifiability of features during and after the clustering process. The significant impact of these algorithms would be on the problems such as gene associations for rare diseases.

By providing hands-on experience with clustering, the project helps connect several foundational topics in data science taught in this course. Discussions about resource constraints expand on the mathematical foundations we usually consider during class and additional considerations for physical limitations. The consistency of centroids are proved using the almost surely convergence of probability covered in class

References

[1] Z. Zhang, K. Lange, and J. Xu, "Simple and scalable sparse k-means clustering via feature ranking," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 10 148–10 160. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/735ddec196a9ca5745c05bec0eaa4bf9-Paper.pdf

[2] D. Pollard, "Strong consistency of k-means clustering," *The Annals of Statistics*, vol. 9, no. 1, 1981.

[3] J. T. Chi, E. C. Chi, and R. G. Baraniuk, "k-pod: A method for k-means clustering of missing data," *The American Statistician*, vol. 70, no. 1, p. 91–99, 2016.

[4] "Skfr-python," https://github.com/aarunishsinha/SKFR-Python, [Online; accessed 8-December-2021].

A SKFR2

```
# PyTorch Implementation of the SKFR2 algorithm
# Input: data of shape (features, samples), clusters, k (number of clusters),
    ↪ sparsity
# Output: center of shape (features, clusters), clusters, list_loss
def skfr2(data, clusters, k, sparsity, maxiter):
    '''
    SKFR2 algorithm
    '''
    # get number of features and samples
    features, samples = data.shape
    # criteria of shape (features)
    criteria = torch.zeros(features).to(device)

    # normalize each feature
    for i in range(features):
        data[i, :] = zscore(data[i, :])

    # list_loss empty list
    list_loss = []
    # number of iterations
    num_iter = 1
    switched = True

    while switched and num_iter < maxiter:
        # initialize clusters
        center = torch.zeros(features, k).to(device)
        # members of size clusters
        members = torch.zeros(k).to(device)
        # for each sample
        for j in range(samples):
            # i as jth cluster
            i = clusters[j]
            # add sample to center
            center[:, i] = center[:, i] + data[:, j]
            # add 1 to members
            members[i] = members[i] + 1
        # for each cluster
        for j in range(k):
            # if members is not zero
            if members[j] > 0:
                # divide by members
                center[:, j] = center[:, j] / members[j]
                # Get criteria = number of members in cluster multiplied by center*
                    ↪ center
                criteria = members[j] * torch.mul(center[:, j], center[:, j])
                # get index from criteria
```

```
            index = torch.LongTensor([i for i in range(len(criteria))]).to(
                ↪ device)
            # sort criteria
            sorted_criteria = sorted(zip(criteria, index))
            J = [x[1] for x in sorted_criteria]
            # make long tensor of J
            J = torch.LongTensor(J).to(device)
            # get only features-sparsity features of J
            J = J[:features-sparsity]
            for i in range(len(J)):
                center[J[i], j] = 0
    # deleter members, criteria, index, sorted_criteria, J
    del members, criteria, index, sorted_criteria, J

    # get distamce as square root of sum of square of each feature
    distance = torch.sqrt(((data.T - center.T[:, np.newaxis])**2).sum(axis=2))
    switched = False
    # for each sample
    for i in range(samples):
        # get index of minimum distance
        j = torch.argmin(distance[:, i])
        # if this index is not same as cluster
        if j != clusters[i]:
            # update cluster
            switched = True
            clusters[i] = j

    # deleter distance
    del distance
    # WSS as sum of square of each feature. Initialize to zero
    WSS = torch.zeros(k).to(device)
    # for each cluster
    for cluster in range(k):
        # get temporary index where cluster is k
        temp_index = torch.LongTensor(np.where(clusters.cpu().numpy() ==
            ↪ cluster)[0]).to(device)
        # tempX as zero tensor of shape (features, len(temp_index))
        tempX = torch.zeros(features, len(temp_index)).to(device)
        # for each temp_index
        for j in range(len(temp_index)):
            # add data to tempX
            tempX[:, j] = data[:, temp_index[j]]
        # get WSS
        WSS[cluster] = torch.mean(((tempX.T - center[:, cluster]).T)**2)
    # get loss as sum of WSS
    loss = torch.sum(WSS)
    # add loss to list_loss
    list_loss.append(loss)
```

```
        # print iteration number and loss
        print('Iteration: {} Loss: {}'.format(num_iter, loss))
        # deleter tempX, temp_index, WSS, loss
        del tempX, temp_index, WSS, loss
        # update iteration number
        num_iter += 1
    return center, clusters, list_loss
```

B Data Collection

```
features = 2
cases = 1000
k = 4
sparsity = 2
X_random = np.random.normal(0,1,size=(features,cases//k))
print(f"X_random.shape: {X_random.shape}")
j = 4
p = 1
for i in range(k-1):
    tmp = np.random.normal(j,2,size=(features,cases//k))
    if i%2==0:
        tmp[0] = -1*tmp[0]
    j+=4
    X_random = np.concatenate((X_random,tmp),axis=1)

X_random = torch.FloatTensor(X_random)
X_random = X_random.to(device)
```