Clustering and Neural Networks

Yuan Gao

May 11, 2025

Part I

The Linear Programming Based K-Means Algorithm

1 Summary:

In the first part of this report, I studied the paper LP-based K-Means Algorithm by Borgwardt et al.[1], where I summarized the main theorems and provided clarifications and completions for some of the proofs. Then I implement the algorithm and reduced it to the original K-Means to verify the correctness. Then in the second part, after exploring this unsupervised learning algorithm, I was eager to investigate the field of supervised learning, specifically focusing on neural networks. In particular, I studied the paper by Xu et al.[7] that proposed improvements to the traditional GRU model for Multivariate Time Series Forecasting tasks, and I reproduced and described the architecture presented in that work. Finally, I applied the model on my created dataset to test the performance of the model.

2 Introduction

Clustering is a fundamental technique in unsupervised learning and data analysis, designed to partition a dataset into meaningful groups (clusters) based on similarity measures. Different clustering methods are tailored to various data structures and applications, for example now some newer methods are designed to handle the cluster bounds. This diversity make them essential for pattern recognition, anomaly detection, data organization and so on.

2.1 Common Clustering Methods

Several popular clustering methods are used in practice:

- K-means Clustering: Iteratively assigns each data point to the nearest cluster center and updates the centers to minimize intra-cluster variance. • Hierarchical Clustering: Builds a nested hierarchy of clusters by iteratively merging or splitting clusters based on similarity. The approach is either bottom-up (agglomerative) or top-down (divisive).

- Spectral Clustering: Uses eigenvalues of similarity matrices to perform clustering in a lower-dimensional space.

More detailed discussions of these methods can be found in [3].

2.2 Applications of Clustering

Clustering has widespread applications across various domains:

• Market Segmentation: Identifying groups of customers with similar purchasing behaviors. • Image Segmentation: Grouping pixels with similar characteristics to detect objects in images.

3 K-means

The K-means algorithm is one of the most popular clustering methods. Given a fixed number k, the goal is to partition a dataset into k clusters such that the data points within each group are as similar as possible, while the differences between points in different groups are as large as possible.

3.1 Algorithm

Given $k \in \mathbb{N}$, a set of data points $X = \{x_1,x_2,...,x_m\} \subseteq \mathbb{R}^d$, and a set of cluster centers $S = \{s_1,s_2,...,s_k\} \subseteq \mathbb{R}^d$, the K-means algorithm proceeds as follows:

1. Initialization: Randomly select k initial cluster centers $s_1,s_2,...,s_k$.
2. Assignment Step: According to the centroids chosen, assign each data point $x_i \in X$ to the cluster C_j whose center s_j is the closest:

   $$C_j = \{x_i \mid \|x_i - s_j\|^2 \leq \|x_i - s_\ell\|^2, \forall \ell \neq j\}.$$

   By doing this, we are minimising the total sum of square which could also be seen as the objective function of the K-means.
3. Update Step: Compute the new cluster centers as the mean of the assigned points:

   $$s_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i \quad .$$
4. Repeat Steps 2 and 3 until convergence (i.e., when cluster assignments no longer change).

3.1.1 Theoretical Results

Theorem 1. *Let $x_1,...,x_m \in \mathbb{R}^d$. The sum of squared distances of the x_i to a point $p \in \mathbb{R}^d$ is minimized when p is the centroid, i.e.,*

$$p = \frac{1}{m}\sum_{i=1}^{m} x_i .$$

Theorem 2. *For any data set X, set of sites S, and any $k \in \mathbb{N}$, the K-means algorithm decreases (from one iteration to the next) the sum of squared errors (SSE).*

Note. A similar proof for Theorem 1 and Theorem 2 will be given in Section 2.2.

Theorem 3. *For any data set X, set of sites S, and any $k \in \mathbb{N}$, the K-means algorithm converges in a finite number of steps.*

Proof: There are at most k^m ways to partition m points into k clusters (since each of the m points has k possible assignments). Thus we had at most k^m unique SSE values for these partitions. Thus, the algorithm must stop after a finite number of steps.

4 Linear programming

Linear programming (LP) is a mathematical method used for optimizing a linear objective function subject to linear equality and inequality constraints. In the context of clustering, LP is used to determine fractional assignments of data points to clusters while satisfying size and weight constraints. A standard LP problem has the form:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b, \end{aligned}$$

where x is the variable vector, A and b define the constraints, and c is the cost vector.

5 Linear Programming Based K-means Algorithm

The following introduced algorithm is based on the paper by Borgwardt et al. [1].

5.1 Motivation

The classical K-means algorithm is widely used for clustering, but it assumes that each data point is fully assigned to exactly one cluster. However, in many real-world applications, it is necessary to incorporate additional constraints such as:

- Weight: Some datasets include weighted points, representing different importance levels or repeated observations.
- Cluster size constraints: In business and operational settings, it is often required that clusters have prescribed lower and upper bounds on their sizes to ensure fairness or feasibility.
- Partial membership: Some applications involve data points that belong to multiple clusters simultaneously, requiring fractional assignments.

5.2 Example: Agricultural Land Redistribution

One practical application of the LP-based K-means algorithm is in the redistribution of agricultural land among farmers in a given region. Suppose an agricultural area consists of n land plots, each represented by a point in $\mathbb{R}^2$. These plots have different productivity levels, represented by weights.

Problem Statement:

- There are k farmers cultivating n plots and some plots may be shared by multiple farmers due to geographical constraints.
- Each farmer should receive a set of contiguous plots.
- The total productivity assigned to each farmer must remain within a given range to ensure fair distribution.

5.3 Notations

Let throughout the present paper $k,n,d \in \mathbb{N}$ with $n \geq k \geq 2$. We denote by $X := \{x_1,...,x_n\} \subset \mathbb{R}^d$ a dataset of distinct points with associated weights $\Omega := (\omega_1,...,\omega_n) \in \mathbb{R}^n$ where $\omega_j > 0$ for all $j \leq n$. we define lower and upper bounds for cluster sizes:

$$\kappa^- := (\kappa_1^-, \dots, \kappa_k^-)^T, \quad \kappa^+ := (\kappa_1^+, \dots, \kappa_k^+)^T \in \mathbb{R}^k, \quad 0 < \kappa_i^- \leq \kappa_i^+.$$

Further we assume that:

$$\sum_{i=1}^{k} \kappa_i^- \leq \sum_{j=1}^{n} \omega_j \leq \sum_{i=1}^{k} \kappa_i^+ .$$

A partial membership k-clustering is defined as $C := (C_1,...,C_k)$, where each cluster C_i is represented by an assignment vector:

$$y := (y_{11},...,y_{1n},...,y_{k1},...,y_{kn})^T \in [0,1]^{kn}$$

such that each data point x_j is fractionally assigned to clusters, satisfying:

$$\sum_{i=1}^{k} y_{ij} = 1, \quad \forall j \leq n.$$

Intuitively, y_{ij} represents the fraction of the point x_j that belongs to cluster C_i. The support of a cluster C_i is:

$$\mathrm{supp}(C_i) := \{x_j \mid y_{ij} > 0\}.$$

This can also be interpreted as the set of all points x_j that are assigned to cluster C_i.

The support of the clustering is the tuple:

$$\mathrm{supp}(C) := (\mathrm{supp}(C_1),\ldots,\mathrm{supp}(C_k)).$$

We denote the total weight or size of cluster C_i as:

$$|C_i| := \sum_{j=1}^{n} y_{ij}\omega_j.$$

The center of gravity c_i of a cluster C_i is given by:

$$c_i := \frac{1}{|C_i|} \sum_{j=1}^{n} y_{ij}\omega_j x_j .$$

We define weight-balanced clusterings as clusterings which satisfied:

$$\kappa^- \leq |C| \leq \kappa^+ \qquad \text{component-wise}.$$

A tuple $I = (k,n,d,X,\Omega,\kappa^\pm)$ that satisfies these properties defines an instance of the weight-balanced clustering problem.

For comparison, a trivial instance $I' = (k,n,d,X)$ refers to the standard k-means setting, where there are no weights and no cluster size constraints.

Weight-Balanced Assignment: We define a weight-balanced (S,κ^-,κ^+)least-squares assignment of X as the following optimization problem:

$$\min \sum_{i=1}^{k} \sum_{j=1}^{n} y_{ij}\omega_j \|x_j - s_i\|^2$$

subject to the constraints:

$$\kappa^- \leq |C_i| \leq \kappa^+ \qquad (i \leq k).$$

This formulation ensures that each cluster has a total weight within the given bounds κ^- and κ^+, while minimizing the squared Euclidean distance between data points and their assigned cluster centers.

Strict Weight-Balanced Assignment: The strict condition guarantees that for a given set of cluster sites S, the assignment vector y is uniquely determined. That is, for fixed S and X, there exists a unique optimal assignment. This uniqueness is particularly crucial in the linear programming (LP) formulation, since under this strict condition the solver can only return a vertex solution.

5.4 Power Diagram

Power diagrams are geometric objects that describe how the dataset X is partitioned with respect to a particular decision boundary. This boundary is defined as follows:

Given a set of sites $S = \{s_1,...,s_k\} \subset \mathrm{R}^d$ and weights $\Sigma = (\sigma_1,...,\sigma_k) \in \mathrm{R}^k$, the power cell P_i is defined by:

$$P_i := \left\{x \in \mathbb{R}^d : \|x - s_i\|^2 - \sigma_i \leq \|x - s_j\|^2 - \sigma_j,\ \forall j \neq i\right\}.$$

Thus, we can see that the two sources of parameters—sites and weights—define a power diagram.

Allow and Support Relation:

- A diagram P is said to *allow* a clustering C if $\mathrm{supp}(C_i) \subseteq P_i$ for each cluster C_i. That is, all points assigned to cluster C_i lie within the corresponding power cell P_i.
- A diagram P is said to *support* a clustering C if $\mathrm{supp}(C_i) = X \cap P_i$ for each C_i. This means that the support of each cluster exactly matches the intersection of the dataset $X \subseteq \mathrm{R}^d$ with the corresponding cell P_i, and thus the power diagram fully determines the clustering.

Strongly Feasible Power Diagram:

- The power diagram P *support* the clustering $C = (C_1,...,C_k)$. This support property is also the key property used in our prooff.
- Given a clustering $C = (C_1,...,C_k)$, we define a multigraph $G(C)$ whose vertices correspond to the clusters $C_1,...,C_k$. An edge labeled by x_j connects nodes C_i and C_l (for $i /= l$) if and only if $x_j \in \mathrm{supp}(C_i) \cap \mathrm{supp}(C_l)$. The power diagram P is called *strongly feasible* if $G(C)$ contains no cycle with two or more distinct edge labels. This condition prevents ambiguity in point-to-cluster assignments and ensures a consistent geometric decomposition.

Centroidal power diagram:

Finally, if P is a feasible power diagram for C and the sites s_i coincide with the centers of gravity c_i of the corresponding clusters, then P is called a

5.5 The LP based K-means algorithms

We present a generalization of the K-Means algorithm that allows the combination of weighted point sets and partial assignment with prescribed lower and upper bounds on the cluster sizes.

Step 1: Linear Programming Formulation

- **Initialize:** Define the dataset $X = \{x_1,...,x_n\} \subset \mathbb{R}^d$, associated weights $\omega = (\omega_1,...,\omega_n)$, and initial cluster centers $S = \{s_1,...,s_k\}$.
- **Define the objective function:** The clustering objective is to minimize the sum of squared Euclidean distances between data points and their assigned cluster centers:

  $$\min \sum_{j=1}^{n}\sum_{i=1}^{k} y_{ij}\omega_j \|x_j - s_i\|^2.$$

  Expanding the squared norm:

  $$\min \sum_{j=1}^{n}\sum_{i=1}^{k} y_{ij}\omega_j (x_j - s_i)^T(x_j - s_i).$$

  Expanding the inner product:

  $$\min \sum_{j=1}^{n}\sum_{i=1}^{k} y_{ij}\omega_j (x_j^T x_j - 2x_j^T s_i + s_i^T s_i).$$

  Since $x^T_j x_j$, $s^T_i s_i$, and $x^T_j s_i$ are fixed for given x_j and s_i, the objective function simplifies to:

  $$\min \sum_{j=1}^{n} y_{ij}\omega_j (s_i^T s_i - 2x_j^T s_i).$$

  Note that this objective function is linear in y_{ij}, ensuring the problem remains a linear programming (LP) problem.
- **Constraints:** The following constraints ensure weight-balanced clustering:

  $$\sum_{i=1}^{k} y_{ij} = 1, \qquad \forall j \le n, \qquad (*)$$

  $$\kappa_i^- \le \sum_{j=1}^{n} y_{ij}\omega_j \le \kappa_i^+, \qquad \forall i \le k, \qquad (\dagger)$$

  $$y_{ij} \ge 0, \qquad \forall i,j.$$

  * Each point is fully assigned: the sum over all clusters for a given point equals one.

† Cluster size constraints: total assigned weight to each cluster must lie within $[\kappa^-_i,\kappa^+_i]$.

Step 2: Update Centroids

• Compute new cluster centers based on the assigned points:

$$s_i = \frac{1}{|C_i|}\sum_{j=1}^{n} y_{ij}\omega_j x_j .$$

Repeat Steps 1 and 2 until convergence which means the objective function in the current iteration doesn't decrease comparing to the last iteration.

Notation: We refer to Step 1 as **Algorithm 1**, and the combination of Steps 1 and 2 as **Algorithm 3**.

Remark: The constraints in our linear programming formulation define a polyhedron, as they are represented by a system of linear inequalities and equalities. Moreover, since the solution set $\{y_{ij}\}$ is bounded within [0,1] and thus finite, these constraints further define a polytope, a convex hull which contains a finite number of points with each in a certain dimension. This dimension in our case is R^{n*k}.

Given that there exist linear programming solvers which always returns a vertex of the feasible region as the optimal solution, our algorithm is guaranteed to obtain a vertex of the polytope as its solution.

5.6 Theorems and Properties of the Algorithm

In this section, we list several theorems and proposition related to the LPbased K-means algorithm, which are also essential for Section 2.7 where we aims to establish an upper bound for the algorithm's performance. Due to space limitations, we provide the proof for Theorem 4 only.

Proposition 2 (Brieden & Gritzmann, 2012). Let *X* be a weighted data set, and let *C* be a (strict) weight-balanced least-squares assignment for *X*. Then *C* allows a (strongly) feasible power diagram. Furthermore, if *C* allows a strongly feasible power diagram, its assignment vector *y* contains at most $2(k - 1)$ fractional components, i.e., components satisfying $0 < y_{ij} < 1$.

Proposition 3 (Brieden & Gritzmann, 2012). Algorithm 1 computes a clustering *C* that allows a strongly feasible power diagram.

Theorem 4. *Algorithm 3 terminates with a clustering that allows a strongly feasible centroidal power diagram.*

Proof: First, we prove that for a given fixed clustering C where y_{ij} has already been determined, the optimal site for the weight-balanced least-squares assignment corresponds to the center of gravity of each cluster. This scenario arises when Algorithm 1 is executed for a single iteration, leaving us with the task of determining the new cluster sites s_i for the next iteration.

Let $C := (C_1,...,C_k)$ be a fixed clustering with centers of gravity $c_1,...,c_k$, and let $s_1,...,s_k$ be the optimal sites. Define $z_i = c_i - s_i$ for $i \leq k$. For each C_i, we then have:

$$\sum_{j=1}^{n} y_{ij}\omega_j \|x_j - s_i\|^2$$

Applying the transformation $s_i = c_i - z_i$:

$$= \sum_{j=1}^{n} y_{ij}\omega_j \|x_j - c_i + c_i - s_i\|^2$$

$$= \sum_{j=1}^{n} y_{ij}\omega_j \|x_j - (c_i - z_i)\|^2$$

$$= \sum_{j=1}^{n} y_{ij}\omega_j \left[x_j - (c_i - z_i)\right]^T \left[x_j - (c_i - z_i)\right]$$

$$= \sum_{j=1}^{n} y_{ij}\omega_j (x_j - c_i + z_i)^T (x_j - c_i + z_i).$$

Rearranging terms:

$$= \sum_{j=1}^{n} y_{ij}\omega_j x_j^T x_j - 2\sum_{j=1}^{n} y_{ij}\omega_j x_j^T (c_i - z_i)$$

$$+ \sum_{j=1}^{n} y_{ij}\omega_j (c_i - z_i)^T (c_i - z_i).$$

The first term remains constant since y_{ij} is fixed by assumption. Substituting the definitions of the center of gravity and the size of clusters:

$$c_i := \frac{1}{|C_i|} \sum_{j=1}^{n} y_{ij}\omega_j x_j$$

$$|C_i| := \sum_{j=1}^{n} y_{ij}\omega_j.$$

,

We replace $\sum_{j=1}^{n} y_{ij}\omega_j x_j^T$ with $|C_i|c^T_i$, and

$$\sum_{j=1}^{n} y_{ij}\omega_j$$

with $|C_i|$, and then expand the parentheses, yielding:

$$\begin{aligned}\sum_{j=1}^{n} y_{ij}\omega_j \|x_j - s_i\|^2 &= \sum_{j=1}^{n} y_{ij}\omega_j x_j^T x_j - 2|C_i| c_i^T c_i + 2|C_i| c_i^T z_i \\ &\quad + |C_i| c_i^T c_i - 2|C_i| c_i^T z_i + |C_i| z_i^T z_i \\ &= \sum_{j=1}^{n} y_{ij}\omega_j x_j^T x_j - |C_i| c_i^T c_i + |C_i| z_i^T z_i.\end{aligned}$$

$$\sum_{j=1}^{n} y_{ij}\omega_j \|x_j - s_i\|^2 = \sum_{j=1}^{n} y_{ij}\omega_j x_j^T x_j - 2|C_i| c_i^T c_i + 2|C_i| c_i^T z_i$$

$$+|C_i| c_i^T c_i - 2|C_i| c_i^T z_i + |C_i| z_i^T z_i.$$

$$= \sum_{j=1}^{n} y_{ij}\omega_j x_j^T x_j - |C_i| c_i^T c_i + |C_i| z_i^T z_i$$

.

Since the clustering is fixed which means that the first two terms are fixed and $|C_i| z_i^T z_i \geq 0$, the expression is minimized when $z_i = 0$, which implies that $s_i = c_i$. Hence, the optimal site s_i for a given cluster C_i is its center of gravity c_i.

Secondly, we use the idea above to prove that the algorithm is *monotonically decreasing* with respect to the objective function.

Let $\Theta(C,S)$ denote the objective value of the linear program, where $C^{(l)}$ represents the clustering assignment in the l-th iteration, and $S^{(l)}$ denotes the corresponding set of sites (centroids). Then the update process satisfies the following inequality:

$$\Theta(C_{(l)}, S_{(l)}) \geq \Theta(C_{(l)}, S_{(l+1)}) \geq \Theta(C_{(l+1)}, S_{(l+1)}).$$

The first inequality holds because we update the centroids $S^{(l+1)}$ as the center of mass (based on the idea discussed above).

The second inequality holds because, given the updated centroids $S^{(l+1)}$, solving the linear program again yields a new assignment $C^{(l+1)}$ that minimizes the objective function further.

Thus, since we know that the number of vertices of a polytope is bounded, and our algorithm is decreasing in each iteration which means we would land on different vertices of the polytope as a solution, thus the Algorithm 3 would terminates in a finite number of steps, and would thus finally terminates with a clustering with a feasible centroidal power diagram.

5.7 Proof of an Upper Bound of the Algorithm 3

In this section, we first introduce some necessary preliminaries. We then combine the previously stated theorems and properties to derive an upper bound for the algorithm 3.

5.7.1 Preliminaries

We define for a power diagram $P := (P_1,...,P_k)$, we define its *(X, P)-incidence pattern* as:

$$\mathrm{X}(P) := (X \cap P_1,...,X \cap P_k),$$

which describes how the data points X are distributed among the power cells. Note that if the set of sites $s_1,...,s_k$ and weights $\sigma_1,...,\sigma_k$ are fixed, then a specific power diagram P' is uniquely determined, and so is the incidence pattern $\mathrm{X}(P')$.

Although in Algorithm 3 the solution set y_{ij} could takes values in the continuous range (0,1), which theoretically allows for an infinite number of iterations, Proposition 3 guarantees that each clustering produced by the algorithm 3 allows a strongly feasible power diagram. This implies that each iteration corresponds to a specific (X,P')-incidence pattern. Therefore, if we can establish an upper bound on the total number of such distinct (X,P)-*incidence patterns*, we can consequently derive a finite upper bound on the number of iterations the algorithm can perform.

Sign-patterns of polynomial systems. Given a system of real polynomials $p_1(z),...,p_t(z)$ in s variables , the *sign-pattern* of the system at a point z is the tuple $v(z) := (v_1,...,v_t) \in \{-1,0,+1\}^t$,

where each entry is defined by

$$v_i = \begin{cases} -1 & \text{if } p_i(z) < 0, \\ 0 & \text{if } p_i(z) = 0, \\ +1 & \text{if } p_i(z) > 0. \end{cases}$$

That is, the sign-pattern records the sign of each polynomial evaluated at the given point $z \in \mathbb{R}^s$.

Proposition 5. Let $p_1,...,p_t$ be a system of real polynomials in s variables, all of degree at most l. If $s \leq 2t$, then the number of different sign-patterns of this system is bounded above by

$$\left(\frac{8e \cdot l \cdot t}{s}\right)^s.$$

Note: The number s denotes the total number of variables in the system, and e denotes the base of the natural logarithm (the exponential constant).

Theorem 6. Let $X := \{x_1,...,x_n\} \subset \mathbb{R}^d$ be a (weighted) data set that is to be partitioned with respect to a (strongly) feasible power diagram with k cells. Then the number of different power patterns $\mathrm{X}(P)$ that can arise is bounded above by

$$\left(\frac{8e \cdot (k-1)n}{d}\right)^{(d+1)(k-1)}.$$

Proof. Firstly, it is easy to see that An (S,Σ)-power diagram cells are the same as those $(S,\Sigma+\sigma')$-power diagram, where $\Sigma+\sigma' := \{\sigma_1+\sigma',...,\sigma_k+\sigma'\}$ for some $\sigma' \in \mathbb{R}$. By this invariance, we set sigma k = 0 to get one parameter free.

Thus, we define the total parameter vector as

$$(s_1,\sigma_1,...,s_{k-1},\sigma_{k-1},s_k)_T \in \mathbb{R}_{(d+1)(k-1)+d} = \mathbb{R}_{(d+1)k-1}.$$

Then we construct our systems of polynomials:

For any pair $i/= j$ and any point $x \in X$, we define a polynomial

$$p_{i,j,x}(g) := \|x - s_i\|^2 - \sigma_i - \left(\|x - s_j\|^2 - \sigma_j\right),$$

Thus there are

$$\binom{k}{2} \cdot n$$

polynomials in total.

And the sign of $p_{i,j,x}(g)$ indicates how $\|x-s_i\|^2-\sigma_i$ compares to $\|x-s_j\|^2-\sigma_j$ for the power diagram defined by g. Recall that a point x lies in the ith power cell if and only if

$$\|x - s_i\|^2 - \sigma_i \leq \|x - s_j\|^2 - \sigma_j \qquad \text{for all } j/= i.$$

We now apply Proposition 5. We have $\binom{k}{2} \cdot n$polynomials, each of degree at most 2, and our parameter vector lies in $\mathbb{R}^{(d+1)k-1}$, so the number of variables is $s = (d+1)k-1$. The condition in Proposition 5 requires $s \leq 2t$, which in our case becomes

$$(d+1)k - 1 \leq 2 \cdot \binom{k}{2} \cdot n.$$

This inequality is satisfied under the general assumptions in our algorithm: $d + 1 \leq n$ and $k \geq 2$, since

$$(d + 1)k - 1 < (d + 1)k \leq nk \leq (k - 1)kn.$$

Hence, applying Proposition 5, we obtain the upper bound for the sign patterns of our polynomial systems:

$$\begin{aligned}\left(\frac{8e \cdot 2\binom{k}{2}n}{(d+1)k-1}\right)^{(d+1)k-1} &= \left(\frac{8e \cdot k(k-1)n}{(d+1)k-1}\right)^{(d+1)k-1} \\ &\leq \left(\frac{8e \cdot k(k-1)n}{(d+1)k-k}\right)^{(d+1)k-1} \\ &= \left(\frac{8e \cdot (k-1)n}{d}\right)^{(d+1)(k-1)} .\end{aligned}$$

Finally, observe that for each point $x \in X$, if $x \in P_i$, then by definition we have $p_{i,j,x}(g) < 0$ for all $j \neq i$. This implies that the corresponding sign entries $v_{i,j} = -1$ for all such j. Meanwhile, the other sign entries $v_{x,y}$ for $x,y \neq i,j$ can take arbitrary values.

Therefore, each possible power pattern for X corresponds to at least one signpattern of the polynomial system. Thus, the upper bound established above also serves as an upper bound on the number of possible power patterns $X(P)$.

5.7.2 Upper Bound

Now, we aim to utilize the properties of sign patterns to construct polynomials that characterize the power diagrams, and then apply the previous results to establish an upper bound for algorithm 3.

Theorem 7. *The number of iterations (evaluations) of Algorithm 3 is bounded by*

$$(40ek^2n)^{(d+1)(k-1)}.$$

Proof. We omit the full proof here due to space limitations, but outline the two key ideas.

First, according to Proposition 2, in each assignment produced by the algorithm, at most $2(k-1)$ of the variables y_{ij} lie in the open interval $(0,1)$. Moreover, each such assignment belongs to a vertex of a certain polytope. This allows us to focus on a subclass of the original polytope, which we denote $Q'(y^*)$.

We define a parameter vector $z := (z_{11},\dots,z_{kn})^T \in \mathbb{R}^{kn}$ by:

$$z_{ij} := \begin{cases} 1 & \text{if, } y^*_{ij} = 1 \\ 0 & \text{otherwise.} \end{cases}$$

The corresponding constraints of this restricted polytope are:

$$\kappa_i^- - \sum_{j=1}^{n} \omega_j z_{ij} \le \sum_{j: y^*_{ij} \notin \{0,1\}} \omega_j y_{ij} \le \kappa_i^+ - \sum_{j=1}^{n} \omega_j z_{ij} \qquad (i \le k),$$

$$\sum_{i: y^*_{ij} \notin \{0,1\}} y_{ij} = 1 - \sum_{i=1}^{k} z_{ij} \qquad (j \le n : \exists i \text{ such that, } y^*_{ij} \notin \{0,1\})$$

$$y_{ij} \ge 0 \quad (i \le k, j \le n : y^*_{ij} \notin \{0,1\}).$$

Since there are at most $2(k-1)$ values where $0 < y^*_{ij} < 1$, the dimension of $Q'(y^*)$ is at most $2(k-1)$. Furthermore, this system has at most $5k-2$ constraints:

line 1 contributes at most $2k$, line 2 contributes at most $k-1$ (which equals to the number of variables), and line 3 at most $2(k-1)$.

Hence, the number of vertices of this subclass polytope is at most the number of bases of the system of inequalities, bounded by:

$$\binom{5k-2}{2(k-1)} \leq (5k)^{2k-2}.$$

Second, by Theorem 6, the number of distinct power patterns X(P) is at most

$$\left(\frac{8e\cdot(k-1)n}{d}\right)^{(d+1)(k-1)}.$$

Since each clustering corresponds to a power pattern, and each clusting corresponds to a vertex of the polytope, this also bounds the number of vertex of the original polytopes'.

Thus, for each iteration of the algorithm 3, the worst case is we need to evaluate on each vertices on a specific subclass of the polytope and get the minimum object value. Since we had proved that the algorithm is decreasing for each iteration which means that we must get different vertex as solutions for each iteration.

Thus, for the whole algorithm, the worst-case scenario is that we traverse each vertex as the best solution in every iteration, which yields the total number of iterations:

$$\begin{aligned}(5k)^{2k-2}\cdot\left(\frac{8e\cdot(k-1)n}{d}\right)^{(d+1)k-1} &\leq \left(\frac{5k\cdot 8e\cdot(k-1)n}{d}\right)^{(d+1)k-1}\\ &\leq (40ek^2n)^{(d+1)k-1}.\end{aligned}$$

□

5.8 Implementation

In this section, we implement the LP-based K-Means algorithm. To verify the correctness of our code, we assume that all data points have equal weights of 1 and there are no constraints for each clustering. This could reduce the algorithm to the traditional k-means method. In practice, we can easily incorporate weights for our data by modifying the objective function and updating the centroids accordingly in the code.

Listing 1: LP-Based K-Means Clustering

```

def kmeans_lp(X, init_centers, max_iter=1000):
""" 4    Solves K-Means clustering using Linear Programming (LP)

                  Case when we had identical weight for X

          Parameters:
```

```
    X: ndarray, shape (n_samples, d) - Input data points
    init_centers: ndarray, shape (k, d) - Initial cluster centers
    max_iter: int - Maximum number of iterations

    Returns:
    centers: Final cluster centers
    labels: Cluster assignments for each point
    obj: Final objective function value
    """

    # Get dataset shape
    n, d = X.shape # n: number of data points, d: dimensions
    k = init_centers.shape[0] # Number of clusters

    obj = 0
    pre_obj = float("inf") # Large initial value for convergence check

    for iteration in range(max_iter):
        pre_obj = obj # Store previous objective value

        # Initialize cluster centers in the first iteration
        if iteration == 0:
            centre_interation = init_centers.copy()

        # Define LP problem (Minimization)
        prob = pulp.LpProblem("KMeans_Clustering", pulp.LpMinimize)

        # Define decision variables (z[i][j] = 1 if X[i] belongs to cluster j)
        z = np.array([[pulp.LpVariable(f"z_{i}_{j}", lowBound=0, upBound=1) for j in range(k)] for i in
        range(n)])

        # Constraint: Each point belongs to one cluster
        for i in range(n):
            prob += pulp.lpSum(z[i][j] for j in range(k)) == 1

            # Objective function: Minimize total squared distance
            prob += sum(
            sum(sum((X[i, dim] - centre_interation[j, dim]) ** 2
                    for dim in range(d)) * z[i, j] for i in range(n))
            for j in range(k)
            )

        # Solve the LP problem
        prob.solve()

        # Retrieve objective value
        obj = pulp.value(prob.objective)

            # Update cluster centers
            centre_interation = np.array([
            [
            sum(pulp.value(z[i][j]) * X[i, dim] for i in range( n)) / sum(pulp.value(z[i][j]) for i in
            range(n)
                )
```

```
                    for dim in range(d)
                    ]
                    for j in range(k)
                    ])

        # Store the best assignment
        best_z = np.array([[pulp.value(z[i][j]) for j in range(k)] for i in range(n)])
        print(obj)
        # Convergence check
        if iteration != 0 and obj >= pre_obj:
                print('Converged!')
                return centre_interation, np.argmax(best_z, axis=1), pre_obj

    return centre_interation, np.argmax(best_z, axis=1), obj # Return final centers, labels, and objective value
```

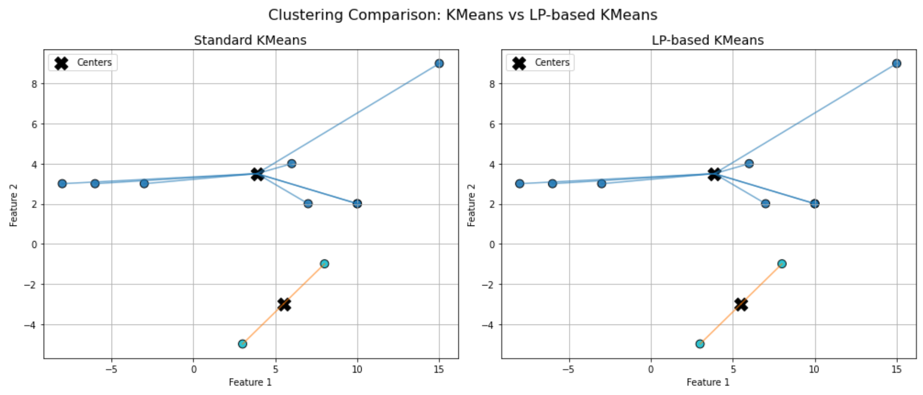

Figure 1: clustering of kmeans vs LP based Kmeans

Part II

Neural Network for Time Series Forecasting

6 Introduction

In Part II, we study a graph-enhanced neural network designed to improve multivariate time series forecasting by leveraging correlations between variables across different historical time steps. We first introduce the key architectural components of neural networks in the following sections. For a more comprehensive overview of neural networks, we refer the reader to Higham et al. [4]. Subsequently, in Section 4, we present the details of the model proposed by Xu et al. [7].

6.1 Definition of Neural Network

Artificial neural networks (ANNs) are a class of machine learning models inspired by the structure and function of biological neural networks. They are widely used in supervised learning tasks such as classification, regression, and time series forecasting.

6.2 Mechanism of Neural Network Training

Neural networks are trained by minimizing a loss function that measures the discrepancy between predicted and true outputs. This is achieved through an iterative optimization process that adjusts the model parameters—weights and biases—based on gradients of the loss.

Gradient-Based Optimization

The most common approach is **gradient descent**, which updates parameters θ at iteration t as: $\theta_{(t+1)} = \theta_{(t)} - \eta \cdot \nabla_{\theta} \mathrm{L}(\theta_{(t)})$,

where η is the learning rate. where η is the learning rate. In practice, variants such as **SGD**, **Adam**, and **RMSProp** are employed to improve convergence speed and stability. These methods differ from standard gradient descent by incorporating techniques such as mini-batch updates, adaptive learning rates which help escape poor local minima.

Backpropagation

To compute the gradients efficiently, neural networks use the **backpropagation** algorithm, which propagates errors backward from the output layer to the input layer. The process assumes a feedforward network with L layers.

Notation

- $a^{[l]}$: activation vector of layer l
- $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$: pre-activation of layer l
- $W^{[l]}, b^{[l]}$: weights and biases of layer l
- $C = \frac{1}{2}\|y - a^{[L]}\|^2$: quadratic cost
- $\delta_{[l]} = \partial z \underline{\partial C}_{[l]}$: error at layer l

- $\odot$: elementwise (Hadamard) product

Backpropagation Equations

$$\delta^{[L]} = \sigma'(z^{[L]}) \odot (a^{[L]} - y), \quad (1)$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \odot \left((W^{[l+1]})^\top \delta^{[l+1]}\right), \quad 2 \le l \le L-1 \quad ,$$

(2) $$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]}, \quad (3)$$

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} \cdot a_k^{[l-1]}. \quad (4)$$

By alternating between forward propagation (computing activations) and backpropagation (computing gradients), the network updates its parameters to reduce the loss and improve performance on unseen data.

6.3 The Gradient Vanishing Problem

The gradient vanishing problem arises when gradients become increasingly small during backpropagation through deep networks, leading to minimal parameter updates and stalled training. It is commonly mitigated by using non-saturating activation functions (e.g., ReLU), normalization techniques, or architectural changes such as residual connections.

6.3.1 Example

An example of addressing the vanishing gradient problem in the Transformer architecture is to swap the order of layer normalization and residual connections. A standard **residual unit** is defined as:

$$y_l = x_l + \mathrm{F}(x_l;\theta_l) \quad (5)$$

$$x_{l+1} = f(y_l) \quad (6)$$

Post-Norm In the original Transformer architecture by Vaswani et al. [5], layer normalization — which normalizes the input vector within this vector — is placed *after* the residual connection. That is, the output of the residual addition is normalized before being passed to the next layer.

$$x_{l+1} = \mathrm{LN}(x_l + \mathrm{F}(x_l;\theta_l)) = \mathrm{LN}(y_l) \quad (7)$$

Here, x_l denotes the input to the l-th layer, and x_{l+1} is the corresponding output. $\mathrm{F}(x_l;\theta_l)$ represents a learnable transformation, in this case is the Mutihead attention layer plus a feedforward layer, parameterized by θ_l.

Let L denote the loss term. By the chain rule, the gradient of the loss with respect to the input x_l of the l-th layer is given by:

$$\frac{\partial \mathcal{L}}{\partial x_l} = \frac{\partial \mathcal{L}}{\partial x_L} \cdot \prod_{k=l}^{L-1} \frac{\partial \mathrm{LN}(y_k)}{\partial y_k} \cdot \prod_{k=l}^{L-1} \left(1 + \frac{\partial \mathcal{F}(x_k;\theta_k)}{\partial x_k}\right). \tag{8}$$

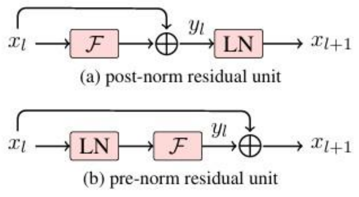

Figure 2: Post-Norm vs Pre-Norm(Wang et al. [6]).

Pre-Norm In Pre-Norm Transformers, layer normalization is applied before the residual connection:

$$x_L = x_l + \sum_{k=l}^{L-1} \mathcal{F}(\mathrm{LN}(x_k);\theta_k),$$

$$\frac{\partial \mathcal{E}}{\partial x_l} = \frac{\partial \mathcal{E}}{\partial x_L} \times \left(1 + \sum_{k=l}^{L-1} \frac{\partial \mathcal{F}(\mathrm{LN}(x_k);\theta_k)}{\partial x_l}\right)$$

$$x_{l+1} = x_l + \mathrm{F}(\mathrm{LN}(x_l);\theta_l) \tag{9}$$

By the recursive expansion:

(10)

. (11)

By analyzing the backpropagation process of both structures, we observe that in the Post-Norm setting, gradients must pass through LN(·) in each sublayer. This introduces the term $\prod_{k=l}^{L-1} \frac{\partial \mathrm{LN}(y_k)}{\partial y_k}$ into the right-hand side of Eq. (5), which increases the risk of gradient vanishing or explosion as the depth L grows.

6.4 Different Architectures of Neural Networks

Neural networks can adopt different architectures depending on the learning task and data structure. For time series forecasting, the following models are commonly used:

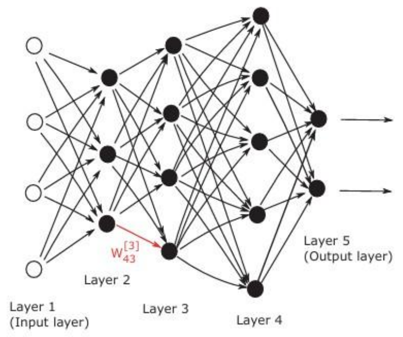

Figure 3: A Feed-forward Neural Network([4]).

1. **Feedforward Neural Network (FNN)** FNNs are the simplest networks, where information flows from input to output without cycles. They are suitable for fixed-size input features without temporal dependencies.

2. **Recurrent Neural Network (RNN)** RNNs incorporate feedback loops, allowing them to maintain hidden states across time steps. This makes them capable of modeling temporal dependencies.

$$h_t = \psi(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \qquad y_t = W_{hy}h_t + b_y,$$

where parameters are shared across time steps.

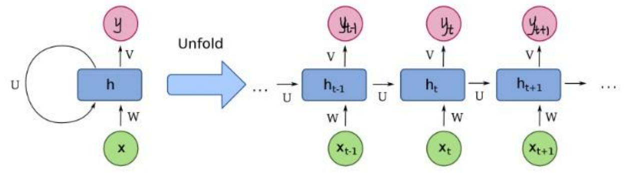

Figure 4: RNN (source: wikipedia)

3. **Gated Recurrent Unit (GRU)** GRUs improve upon RNNs by introducing gating mechanisms to better capture long-term dependencies and mitigate vanishing gradients. The update rules are:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}), \qquad r_t = \sigma(W_r x_t + U_r h_{t-1}), \tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1})), \qquad h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t.$$

For example, if the network needs to retain information from an early time step (e.g., $t = 1$) and use it much later (e.g., $t = 21$), the GRU can learn to keep the update gate z_t near zero for $t = 1$ to 20, effectively preserving the hidden state h_1 across time. When z_{21} becomes close to one, the network updates the state, demonstrating GRU's ability to maintain long-term memory through gating.

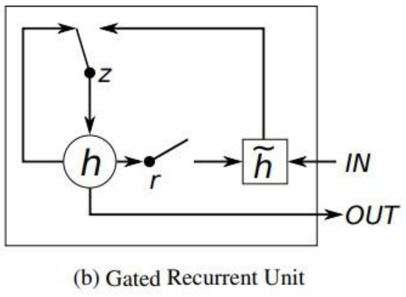

Figure 5: GRU([2])

7 A Graph Enhanced Neural Network For Multivariate Forecasting

7.1 Multivariate Time Series Forecasting

7.1.1 Definition

A multivariate time series contains multiple variables observed over time. Each variable may depend on its own past values and those of other variables. With the recent advancements in machine learning, time series models are generally categorized into two groups: *statistical-based models* and *machine learning-based models*.

Example: The Bivariate Vector Autoregressive Models Model with lag 2

By Zivot et al. [8]:

Let $\mathbf{y}_t = \begin{pmatrix} y_{1t} \\ y_{2t} \end{pmatrix}$ be a 2-dimensional time series. The VAR(2) model is:

$$\mathbf{y}_t = \mathbf{c} + \Pi_1 \mathbf{y}_{t-1} + \Pi_2 \mathbf{y}_{t-2} + \varepsilon_t, \tag{12}$$

where Π_1, Π_2 are 2×2 coefficient matrices, and ε_t is white noise. Expanded, this becomes:

$$y_{1t} = c_1 + \pi_{11}^{(1)} y_{1,t-1} + \pi_{12}^{(1)} y_{2,t-1} + \pi_{11}^{(2)} y_{1,t-2} + \pi_{12}^{(2)} y_{2,t-2} + \varepsilon_{1t},$$
$$y_{2t} = c_2 + \pi_{21}^{(1)} y_{1,t-1} + \pi_{22}^{(1)} y_{2,t-1} + \pi_{21}^{(2)} y_{1,t-2} + \pi_{22}^{(2)} y_{2,t-2} + \varepsilon_{2t}.$$

7.1.2 Related Problem

Traditional statistical models assume a linear dependency among variables. However, they suffer from two main drawbacks. First, they are incapable of capturing more complex, nonlinear relationships. Second, when the number of variables is n, the number of parameters to be estimated grows quadratically to n^2. This quadratic increase in model complexity introduces a large number of parameters, which may lead to overfitting.

With the development of deep learning, various neural network architectures such as LSTM and GRU have been employed to capture the nonlinear patterns in multivariate time series. More recently, some models have incorporated graph neural networks to model the correlations among variables. However, these neural networks often overlook the temporal interdependencies between a given variable and the historical sequences of other variables.

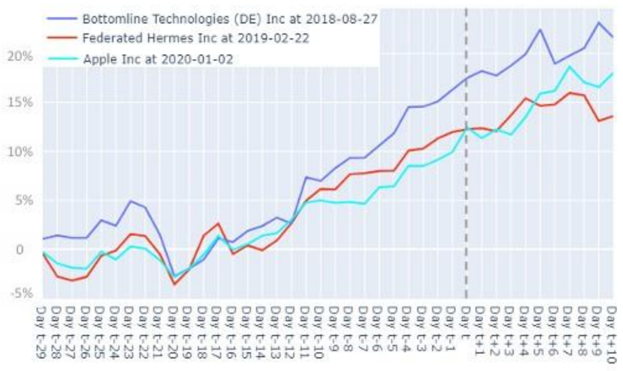

Figure 6: The cumulative return series of three stocks at different time stamps [7].

7.2 The Model Architecture

7.2.1 Dataset

We leverage a dataset consisting of daily exchange rates from eight countries. The features are constructed as a sequence $X_t = \{x_{t-d},...,x_t\}$, where each $x_t \in \mathbb{R}^n$, with n representing the number of countries and d denoting the length of the input time step. The model uses this historical sequence to predict $x_{t+h} \in \mathbb{R}^n$, i.e., the exchange rates of all n countries with h days into the future.

7.2.2 Definition: Series Instance

Given the previously defined $X_t \in \mathbb{R}^{n\times d}$, a *series instance* refers to a d-dimensional vector that represents the historical sequence of a particular variable. It is denoted by $X_{i,t}$, where i indicates the i-th variable.

For example, in the matrix $M \in \mathbb{R}^{n\times d}$ shown below, a series instance corresponds to a single row of the matrix. Specifically, the first row of M represents the historical values sequence of the first variable:

$$M = \begin{bmatrix} x_{t-d}^1 & x_{t-d+1}^1 & \cdots & x_t^1 \\ x_{t-d}^2 & x_{t-d+1}^2 & \cdots & x_t^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_{t-d}^n & x_{t-d+1}^n & \cdots & x_t^n \end{bmatrix}$$

7.2.3 The Architecture

Part 1: Sequence Encoding with GRU and MLP

Let X = $\{X_1, X_2, ..., X_T\}$ denote the input sequence, where each $X_i \in \mathbb{R}^{n\times d}$.

The model first encodes this sequence using a GRU followed by a feedforward neural network (MLP). Specifically, we reshape the input into a tensor with shape (N,T,n), where: - N is the batch size, - T is the sequence length, - n is the number of features.

The GRU processes this input and outputs a tensor of shape:

$$(N,T,L')$$

where L' is the hidden size of the GRU. We then extract the hidden state from the last time step, resulting in a tensor of shape:

$$(N,L')$$

Note that in our model, we let the mini-batch to be composed of all n instances at each time point t, so the batch size $N = n$, the sequence length $T = d$, and n = 1.

This output is then fed into a Multilayer Perceptron (MLP) defined as:

$$\text{MLP} : \mathbb{R}^{L'} \to \mathbb{R}^{L}$$

which transforms the input from dimension L' to L. The final output of this part is thus a tensor of shape:

$$(N,L)$$

where L is the output dimension of the MLP.

We apply this process for all $t \in T$, which means it is performed on all batches. As a result, the original sequence $X = \{X_1, ..., X_T\}$ is transformed into a new sequence $E = \{E_1, ..., E_T\}$, where each $E_i \in \mathbb{R}^{n\times L}$.

Moreover, that during this process, we stop the gradient updates. This step can be regarded as applying a nonlinear projection to the original data.

Part 2: Sequence Encoding with GRU and MLP

Then, we enable the gradient updates and apply the same procedure as in Part 1. As a result, we could obtain similar sequence of Matrix $H_t \in \mathbb{R}^{n\times L}$ at each time step t.

Part 3: Sampler

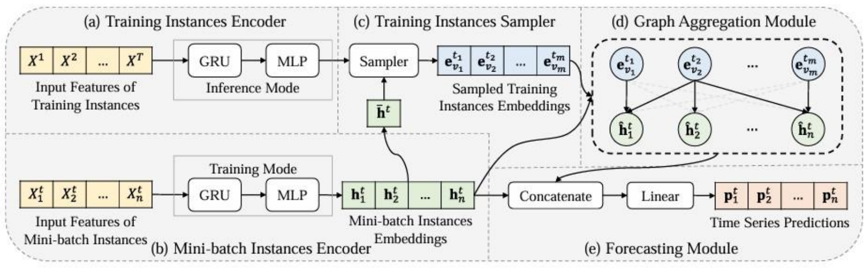

Figure 7: Overall Architecture ([7])

For each $H_t \in \mathbb{R}^{n\times L}$ obtained from Part 2, we split it by variables into n vectors $v_{i,t} \in \mathbb{R}^L$, where $v_{i,t}$ represents the feature instance of variable i at time step t.

For each E_t obtained from Part 1, where each row corresponds to the GRU+MLP representation of a variable, we perform a row-wise average over all variables at each time step t. Specifically, the aggregated representation $\bar{E}_t$ is computed as:

$$\bar{E}_t = \frac{1}{n}\sum_{i=1}^{n} E_{t,i}$$

where $E_{t,i}$ denotes the i-th row of E_t.

This produces a new sequence $\bar{E} = \{\bar{E}_1,...,\bar{E}_T\}$, where each $\bar{E}_t \in \mathbb{R}^L$.

Similarly, we apply the same row-wise averaging operation on H_t to obtain a new sequence $\bar{H} = \{\bar{H}_1,...,\bar{H}_T\}$, where each $\bar{H}_t \in \mathbb{R}^L$.

Then, at each time step t, we compute the cosine similarity between $\bar{H}_t \in \mathbb{R}^l$ and all other $\bar{E}_i \in \mathbb{R}^l$ for $i = 1,...,T$. We then select the top-k time steps according to the highest similarity scores. These selected time steps can be regarded as the k most relevant days to time t.

Part 4: Graph Aggregation

For each time step t, since we previously computed $\bar{E}_t$ as the average representation of the n instances at time t, for each selected similar day we have n sampled instances from H_t. Thus, at each t, we have a total of $m = n \times k$ sampled instances.

Similarly, we expand H_t at each time step into n instance vectors, each belonging to $\mathbb{R}^L$.

At this point, at each time step t, we have: - n instance vectors from H_t, each in $\mathbb{R}^L$; - $m = n \times k$ sampled instance vectors, each also in $\mathbb{R}^L$.

Now, we compute the cosine similarity between each instance vector $v_{i,t}$ and each sampled instance vector $v_{j,\text{sampled}}$, resulting in a similarity matrix. Specifically, the cosine similarity matrix $A \in \mathbb{R}^{n\times(n\times k)}$ is computed as:

$$A_{ij} = \text{Cosine}(W_h v_{i,t}, W_e v_{j,\text{sampled}}) = \frac{W_h v_{i,t} \cdot W_e v_{j,\text{sampled}}}{\| W_e v_{j,\text{sampled}} \|_2}, \quad \| W_h v_{i,t} \|_2$$

where W_h and W_e are two learnable linear mapping matrices applied to the mini-batch instance vectors and sampled instance vectors, respectively. The matrix A serves as the adjacency matrix connecting mini-batch instances to sampled instances.

Then, to mitigate the overfitting problem caused by the fully connected structure of the adjacency matrix, we adopt a sparsification strategy inspired by graph neural networks (GNNs). Specifically, for each row of the adjacency matrix A, we retain only the top-N largest entries and set the remaining entries to zero. The resulting sparsified adjacency matrix is denoted as $\hat{A}$.

Finally, we aggregate the sampled instances back to the n original instances to obtain new representations $\hat{h}^t_i$. The aggregation is performed using the following formula:

$$\hat{h}_i^t = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \hat{A}_{ij} . W_e . v_{j,\text{sampled}},$$

where $\mathcal{N}_i$ denotes the set of sampled neighbors for node i, and $|\mathcal{N}_i|$ is the number of selected neighbors.

Note that the above operation is performed for all time steps $t \in T$.

Part 5: Prediction

For each aggregated representation $\hat{h}^t_i$ obtained from Part 4, we concatenate it with the original instance representation h^t_i, and feed the concatenated vector into a Multilayer Perceptron (MLP) to obtain the final prediction for x^{t+h}_i. Formally, the prediction p^t_i is computed as:

$$p_i^t = \text{Linear}(\text{Concat}(\hat{h}_i^t, h_i^t)),$$

7.3 Training Details

7.3.1 Data Preprocessing

We split the dataset in chronological order into 60% for the training set, 20% for the validation set, and 20% for the test set. Then, we normalize each features in the three sets based on the maximum value of features in the training set

7.3.2 Evaluation Metrics

On the training set, we use either the Mean Squared Error (MSE) or the Mean Absolute Error (MAE) to compute the loss and its gradient for parameter updates. On the validation set, we evaluate the model's performance using the following two metrics:

- **Root Relative Squared Error (RSE)**:

$$RSE = \frac{\sqrt{\sum_{(i,t)\in\Omega_{Test}} (Y_{it} - \hat{Y}_{it})^2}}{\sqrt{\sum_{(i,t)\in\Omega_{Test}} (Y_{it} - \mathrm{mean}(Y))^2}}$$

  RSE is a scaled version of the widely used Root Mean Squared Error (RMSE). A lower RSE value indicates better performance.

- **Empirical Correlation Coefficient (CORR)**:

$$CORR = \frac{1}{n}\sum_{i=1}^{n} \frac{\sum_t (Y_{it} - \mathrm{mean}(Y_i))(\hat{Y}_{it} - \mathrm{mean}(\hat{Y}_i))}{\sqrt{\sum_t (Y_{it} - \mathrm{mean}(Y_i))^2 \sum_t (\hat{Y}_{it} - \mathrm{mean}(\hat{Y}_i))^2}}$$

  CORR measures the correlation between the true and predicted values for each variable. A higher CORR value indicates better performance.

Note that we choose these two metrics mainly because they are designed to normalize evaluation results regardless of the data scale. This allows us to fairly compare the model's performance across different datasets. Here, $Y, \hat{Y} \in \mathbb{R}^{n\times T}$ denote the ground truth and predicted values, respectively, where n is the number of variables and T is the number of time steps.

7.3.3 Training and Split Nodes Mechanism

We train the model for 100 epochs on the training set, evaluating its performance on the validation set after each epoch and recording the results. Although we do not apply an early stopping mechanism, we save the best-performing model across all epochs based on validation performance.

After constructing and preprocessing the training data, and completing Part 1 to obtain the GRU+MLP representations of all training instances, we introduce the *split nodes mechanism*. In short, for each training batch, we divide the data into several subsets based on a certain permutation of the input features. Each resulting subset contains time series data corresponding to only a few selected features. These subsets are then individually passed through the model, and their respective losses are aggregated before performing a parameter update.

However, this introduces a potential issue: during Part 2, when each subset is passed through the GRU+MLP, the resulting representations only capture partial information from a subset of features. This affects the accuracy of the aggregated representation $\hat{h}^t$ used for sampling in Part 3.

To mitigate this issue, we periodically reshuffle the feature permutations after several epochs. This introduces randomness into the construction of $\hat{h}^t$ and allows the model to gradually incorporate more complete feature interactions over time.

This splitting mechanism allows the model to process large multivariate time series datasets more efficiently by breaking them into manageable chunks and helps prevent memory overload during training.

7.4 A Small Experiment

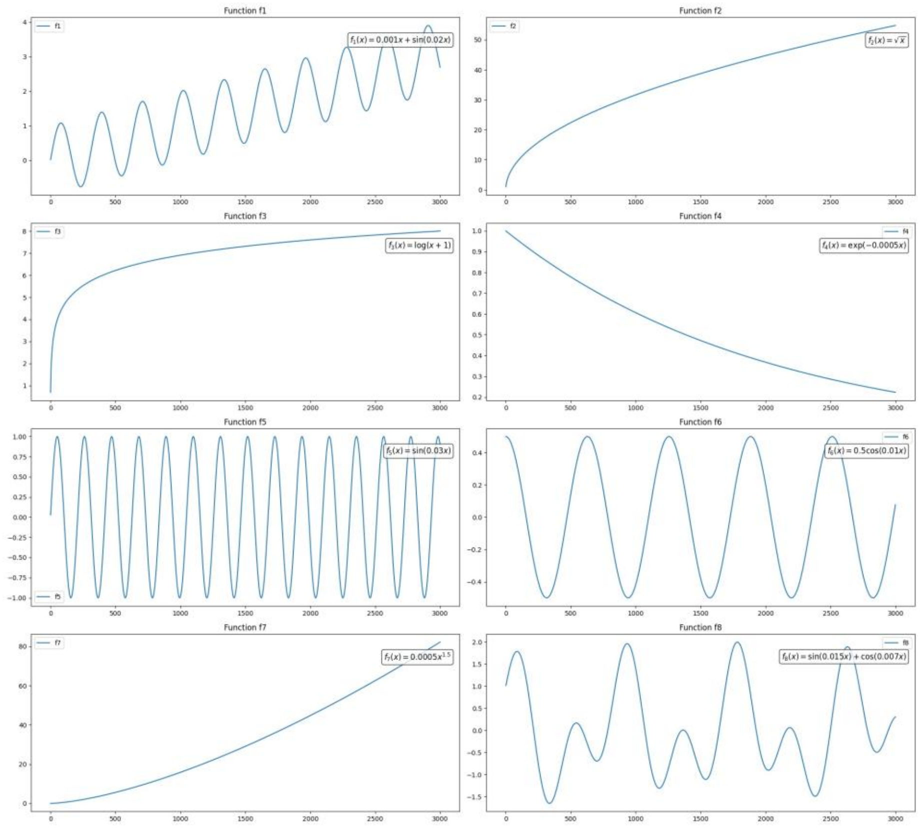

Figure 8: Function Set 1

We now conduct a small-scale experiment using a set of predefined functions to evaluate the model's performance. Theoretically, our model is particularly effective at predicting variables that exhibit similar time series patterns in different segments of their history. Although the functions may appear complex and diverse, some share similar dynamics over different time intervals, which aligns with the model's underlying assumptions.

By applying the model and evaluating it using the two metrics introduced in Section 4.3.2, we obtain a test set performance of **Corr** = 0.9993 and **RSE** = 0.0034. These results are comparable to those reported in the original paper, where the model was applied to the Electricity dataset and achieved a **Corr** of 0.9508 and **RSE** of 0.0740 with a prediction horizon of 3.

It is important to note that both metrics are normalized, which allows for meaningful comparisons across datasets of different scales.

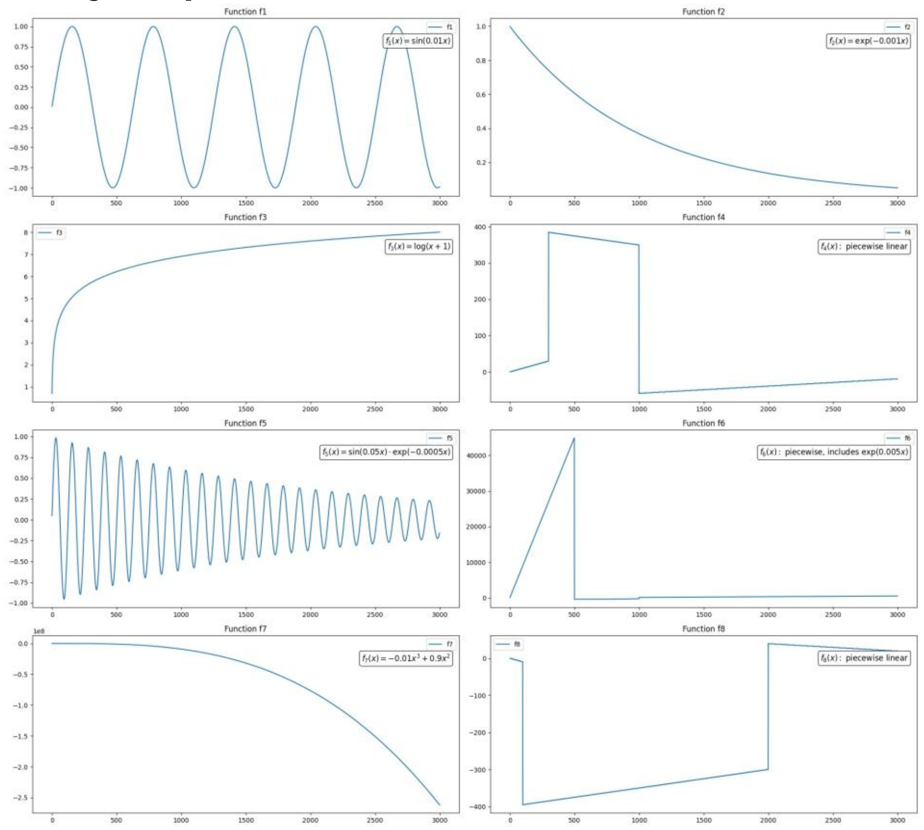

Figure 9: Function set 2

Next, we aim to challenge the core assumption of the model by introducing several piecewise-defined functions. Intuitively, as visualized in the plots, these functions disrupt the presence of similar patterns across different variables at different time steps. As a result, the key assumption that underpins the model no longer holds.

After applying the model under this setting, the performance degrades significantly, yielding a test set result of **Corr** = 0.2949 and **RSE** = 0.6025. This confirms the model's reliance on shared temporal patterns across variables in different timesteps for accurate forecasting.

References

[1] Steffen Borgwardt, Andreas Brieden, and Peter Gritzmann. An lp-based k-means algorithm for balancing weighted point sets. *European Journal of Operational Research*, 263(2):349–355, 2017.

[2] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[3] Trevor Hastie. The elements of statistical learning: data mining, inference, and prediction, 2009.

[4] Catherine F Higham and Desmond J Higham. Deep learning: An introduction for applied mathematicians. *SIAM review*, 61(4):860–891, 2019.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, L ukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[6] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. *arXiv preprint arXiv:1906.01787*, 2019.

[7] Wentao Xu, Weiqing Liu, Jiang Bian, Jian Yin, and Tie-Yan Liu. Instancewise graph-based framework for multivariate time series forecasting. *arXiv preprint arXiv:2109.06489*, 2021.

[8] Eric Zivot and Jiahui Wang. Vector autoregressive models for multivariate time series. *Modeling financial time series with S-PLUS®*, pages 385–429, 2006.

8 Appendix

Listing 2: Standard K-Means Clustering

```

from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
import pulp
# Define initial cluster centers
init_centers = np.array([[2, 2.5], [5, -5]])

                    # Define dataset
                    X = np.array([[7, 2], [3, -5], [-3, 3], [6, 4], [-8, 3], [10, 2],
                    [15, 9], [8, -1], [10, 2], [-6, 3]])

 # Run K-Means clustering
 kmeans = KMeans(n_clusters=2, init=init_centers, n_init=1, random_state=42)
 kmeans.fit(X)

 # Get cluster labels and final centers
 labels = kmeans.labels_
 centers = kmeans.cluster_centers_

 print(labels)
 print(centers)
print("Final objective function value (inertia):", kmeans.inertia_)


def kmeans_lp(X, init_centers, max_iter=1000):
    """
    Solves K-Means clustering using Linear Programming (LP)

        Parameters:
    X: ndarray, shape (n_samples, d) - Input data points
    init_centers: ndarray, shape (k, d) - Initial cluster centers
    max_iter: int - Maximum number of iterations

        Returns:
    centers: Final cluster centers
    labels: Cluster assignments for each point
    obj: Final objective function value
    """

    n, d = X.shape # Number of points, dimensions
    k = init_centers.shape[0] # Number of clusters

    obj = 0
    pre_obj = float("inf") # Large initial value

        for iteration in range(max_iter):
        pre_obj = obj

            if iteration == 0:
            centre_interation = init_centers.copy()

        # Define LP problem
        prob = pulp.LpProblem("KMeans_Clustering", pulp.LpMinimize)
        # Define binary decision variables (z[i][j] = 1 if X[i] belongs to cluster j)
        z = np.array([[pulp.LpVariable(f"z_{i}_{j}", cat=pulp. LpBinary) for j in range(k)] for i in range(n)])

# Constraint: Each point belongs to one cluster     for i in range(n):
                        prob += pulp.lpSum(z[i][j] for j in range(k)) == 1

            # Objective function: Minimize squared distance
            prob += sum(
            sum(sum((X[i, dim] - centre_interation[j, dim]) ** 2
                                    for dim in range(d)) * z[i, j] for i in range(n))
            for j in range(k)
            )

        # Solve the LP problem
        prob.solve()

        # Retrieve objective value
        obj = pulp.value(prob.objective)

                # Update cluster centers
                centre_interation = np.array([
                [
                sum(pulp.value(z[i][j]) * X[i, dim] for i in range( n)) / sum(pulp.value(z[i][j]) for i in
                range(n)
                    )
                for dim in range(d)
                ]
                for j in range(k)
                ])

        # Store the best assignment
        best_z = np.array([[pulp.value(z[i][j]) for j in range(k)] for i in range(n)])
        print(obj)

               if iteration != 0 and obj >= pre_obj:
               print('Converged!')
                return centre_interation, np.argmax(best_z, axis=1), pre_obj

             return centre_interation, np.argmax(best_z, axis=1), obj

# Example usage
init_centers = np.array([[2, 2.5], [5, -5]]) # Initial centers  X = np.array([[7, 2], [3, -5], [-3, 3], [6, 4], [-8,
3],[10,2],  [15,9],[8,-1],[10,2],[-6,3]]) # Data points

centers, labels, final_obj = kmeans_lp(X, init_centers)
```

```
print("Final Cluster Centers:\n", centers)
print("Cluster Assignments:\n", labels)
print("Final Objective Function Value:", final_obj)
```

Listing 3: The impletemetation of the Graph Enhanced GRU Network

```

class IGMTFModel(nn.Module):
    def __init__(self, d_feat=8, hidden_size=512, num_layers=2, dropout=0.0,
        base_model="GRU"):
        """
        Args:
            d_feat (int): Number of features per time step (only used for LSTM)
            hidden_size (int): Hidden dimension for RNN and MLP
            num_layers (int): Number of RNN layers
            dropout (float): Dropout rate inside RNN
            base_model (str): One of 'GRU' or 'LSTM'
        """
        super().__init__()

        # Choose GRU or LSTM as backbone
        if base_model == "GRU":
            self.rnn = nn.GRU(
                input_size=1,
                hidden_size=hidden_size,
                num_layers=num_layers,
                batch_first=True,
                dropout=dropout,
            )
        elif base_model == "LSTM":
            self.rnn = nn.LSTM(
                input_size=d_feat,
                hidden_size=hidden_size,
                num_layers=num_layers,
                batch_first=True,
                dropout=dropout,
            )
        else:
            raise ValueError("Unknown base model name %s" % base_model)

        # A 2-layer feedforward network after RNN
        self.lins = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.LeakyReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.LeakyReLU()
        )

        # Project embeddings for graph attention
        self.project1 = nn.Linear(hidden_size, hidden_size, bias=
            False)
        self.project2 = nn.Linear(hidden_size, hidden_size, bias=
            False)

        # Output layer after graph aggregation + direct encoding
        self.fc_out_pred = nn.Linear(hidden_size * 2, 1)

        self.leaky_relu = nn.LeakyReLU()
        self.d_feat = d_feat
        def cal_cos_similarity(self, x, y):
         """
        Compute cosine similarity between all rows of x and y.
        Args:           x: (n, d), y: (m, d) Returns:
(n, m) cosine similarity matrix """
        xy = x.mm(torch.t(y)) # Dot product
        x_norm = torch.sqrt(torch.sum(x * x, dim=1)).reshape(-1, 1)
        y_norm = torch.sqrt(torch.sum(y * y, dim=1)).reshape(-1, 1)
        return xy / (x_norm.mm(torch.t(y_norm)) + 1e-6)

          def sparse_dense_mul(self, s, d):
         """
            Multiply sparse tensor s with dense tensor d (only on nonzero positions of s).
         """
        i = s._indices()
        v = s._values()
        dv = d[i[0, :], i[1, :]]
        return torch.sparse.FloatTensor(i, v * dv, s.size())

        def forward(self, x, get_hidden=False, train_hidden=None, train_hidden_day=None, k_day=10,
        n_neighbor=5):
        device = x.device

        # Transpose input: (1, 168, F)       (F, 168, 1)
        x = x.permute(2, 1, 0)
        out, _ = self.rnn(x) # Shape: (F, T, H)
        out = out[:, -1, :] # Shape: (F, H)
        out = self.lins(out) # Shape: (F, H)          mini_batch_out = out # Feature-wise hidden encodings
                  if get_hidden:
            return mini_batch_out
            # Average across features to represent "the day"
            mini_batch_out_day = torch.mean(mini_batch_out, dim=0). unsqueeze(0)

        #Compute cosine similarity to all stored day-level hidden
            states
        day_similarity = self.cal_cos_similarity(mini_batch_out_day
            , train_hidden_day.to(device))

        # Get top-k most similar days from training memory
        day_index = torch.topk(day_similarity, k_day, dim=1)[1]

        # Retrieve hidden states for those days
        sample_train_hidden = train_hidden[day_index.long().cpu()]. squeeze()
        sample_train_hidden = [torch.from_numpy(h.astype(np.float32 )).to(device) for h in
        sample_train_hidden]
        sample_train_hidden = torch.cat(sample_train_hidden, dim=0)
```

```
                # Shape: (k_day * F, H)
        sample_train_hidden = self.lins(sample_train_hidden) #
            Refine memory encodings

        # Compute pairwise cosine similarity between query and
            memory
        cos_similarity = self.cal_cos_similarity(self.project1( mini_batch_out),
        self.project2(sample_train_hidden))

            # Build sparse top-k graph connection matrix (attention
            weights)
            row = (
            torch.arange(x.shape[0]).reshape([-1, 1])
            .repeat(1, n_neighbor)
            .reshape(1, -1)
            .to(device)
            )
            column = torch.topk(cos_similarity, n_neighbor, dim=1)[1]. reshape(1, -1)

            mask = torch.sparse_coo_tensor(
            torch.cat([row, column]),
            torch.ones([row.shape[1]]).to(device) / n_neighbor,
            (x.shape[0], sample_train_hidden.shape[0]),
            )

        # Apply attention weights to the memory states
        cos_similarity = self.sparse_dense_mul(mask, cos_similarity )
        agg_out = torch.sparse.mm(cos_similarity, self.project2( sample_train_hidden)) # Shape: (F,
        H)

        # Concatenate direct encoding + neighbor encoding     final
            prediction
        out = self.fc_out_pred(torch.cat([mini_batch_out, agg_out], axis=1)).squeeze() # Shape: (F,)

          return out
```

Note that, for simplicity, I only present the architecture of the network here. The full training and validation code consists of approximately 450 lines. For the complete implementation, please see the full source code at: https://github.com/xuang774/Deep-learning/blob/main/MATH499_appendix.py