Common Failure Modes of Subcluster-based Sampling in Dirichlet Process Gaussian Mixture Models – and a Deep-learning Solution

Vlad Winter*
winterv@post.bgu.ac.il
Ben-Gurion University

Or Dinari*
dinari@post.bgu.ac.il
Ben-Gurion University

Oren Freifeld
orenfr@cs.bgu.ac.il
Ben-Gurion University

Abstract

The Dirichlet Process Gaussian Mixture Model (DPGMM) is often used to cluster data when the number of clusters is unknown. One main DPGMM inference paradigm relies on sampling. Here we consider a known state-of-art sampler (proposed by Chang and Fisher III (2013) and improved by Dinari *et al.* (2019)), analyze its failure modes, and show how to improve it, often drastically. Concretely, in that sampler, whenever a new cluster is formed it is augmented with two subclusters whose labels are initialized at random. Upon their evolution, the subclusters serve to propose a split of the parent cluster. We show that the random initialization is often problematic and hurts the otherwise-effective sampler. Specifically, we demonstrate that this initialization tends to lead to poor split proposals and/or too many iterations before a desired split is accepted. This slows convergence and can damage the clustering. As a remedy, we propose two drop-in-replacement options for the subcluster-initialization subroutine. The first is an intuitive heuristic while the second is based on deep learning. We show that the proposed approach yields better splits, which in turn translate to substantial improvements in performance, results, and stability. Our code is publicly available.

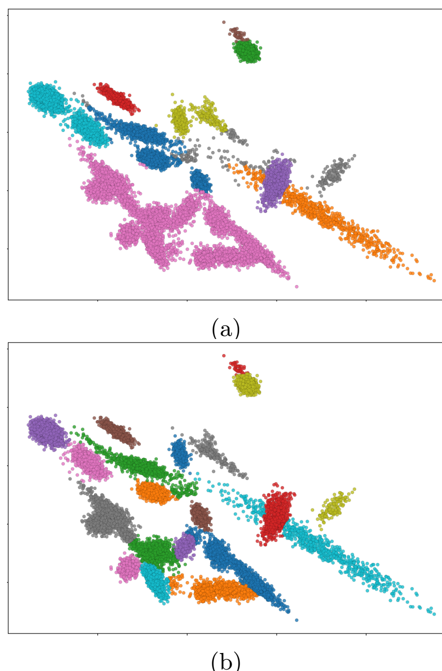


Figure 1: (a) A post-convergence clustering result of the SubC sampler (Chang and Fisher III, 2013) in a relatively-difficult dataset. Note the significant underestimation of the # of clusters. (b) With our proposed SplitNet for subcluster initializations, the same sampler quickly converges to a much better result.

1 INTRODUCTION

The Dirichlet Process Gaussian Mixture Model (DPGMM), a Bayesian Nonparametric (BNP) extension of the Gaussian Mixture Model (GMM), provides a flexible and principled approach to clustering when K, the number of clusters, is unknown. One appeal of BNP clustering is that, as K itself is inferred, the models adapt to the complexity of the data. Exact DPGMM inference being infeasible, the main inference paradigms are the variational approach and

Proceedings of the 25th International Conference on Artificial Intelligence and Statistics (AISTATS) 2022, Valencia, Spain. PMLR: Volume 151. Copyright 2022 by the author(s).

Acknowledgements. This work was supported by the Lynn and William Frankel Center at BGU CS, by the Israeli Council for Higher Education via the BGU Data Science Research Center, and by Israel Science Foundation Personal Grant #360/21. O.D. was also funded by the Jabotinsky Scholarship from Israel's Ministry of Technology and Science, and by BGU's Hi-Tech Scholarship.

the Markov Chain Monte Carlo (MCMC, Robert and Casella (2013)) sampling-based approach. Chang and Fisher III (2013) proposed a sampler, henceforth referred to as the SubC sampler (due to is usage of *subclusters*; see § 3) which is one of the best and fastest DPGMM samplers (especially via its recent and more scalable reimplementation by Dinari et al. (2019)). A key component of the SubC sampler is the splitting and merging of clusters, which allows changing K. For reasons to become clear shortly, our paper focuses on their splits. During the sampling iterations, whenever a new cluster is formed it is augmented with two subclusters. Next, in the subsequent iterations, these subclusters evolve with the rest of the model where, every once in a while, the sampler *proposes* to split the cluster into its two subclusters. Note well: *subclusters are created not only before running the sampler as part of the overall initialization but also many times during the inference process itself*; *i.e.*, whenever a cluster is split (during the sampler's run), its subclusters become clusters, and each of which must be augmented with their own pair of new subclusters.

A practical question arises: upon their creation, how should the subsclusters be initialized? Chang and Fisher III (2013) did so by randomly partitioning the cluster into its subclusters. Seemingly, this is a natural and benign choice. Another question, perhaps less obvious, then follows: *how much, if at all, does the choice of the initialization method matter*?

As we will show, *the surprising answer is that it can matter a lot and have drastic effects* on the SubC sampler's performance (*e.g.*, it can be the difference between poor clustering (Figure 1a) and a success (Figure 1b)), convergence speed, and stability.

We note here that the crust of the matter is that the random initializations tend to lead to poor split proposals (which are mostly rejected) and/or too many iterations before a desired split is accepted. Upon discovering that the subcluster initializations play such a crucial role, and upon characterizing the failure modes caused by the random initialization, we set out to investigate better subcluster-initialization methods. Of note, since subcluster initialization often occurs multiple times during the sampler's run, such methods must be fast (prohibiting the usage of relatively-expensive clustering methods). This leads us to propose two drop-in-replacement alternatives for subcluster initialization: 1) a simple-yet-effective heuristic based on 2-means (*i.e.*, the classical K-means (Lloyd, 1982) whose "K" is 2 and should not be confused with K, that in this paper denotes the total number of inferred clusters according to the model); 2) an even better alternative, a new Deep Neural Net, self-coined "SplitNet". The utility of those solutions, SplitNet in particular, is especially noticeable in the context of the aforementioned failure modes. In our experimental study, we compare the baseline SubC sampler, its two variants based on our proposed solutions, as well as several other popular DPGMM methods, and test them all using a comprehensive suite of synthetic simulations and real datasets. We show that with our better initializations, the sampler outperforms the other methods in inference time, stability, and various clustering evaluation metrics.

To summarize, our two main contributions are: 1) we characterize failure modes of the SubC sampler (Chang and Fisher III, 2013); 2) we propose two subcluster initialization methods that improve that sampler, the important among the two being a novel Deep Learning (DL) method. Finally, our code is publicly available at `https://github.com/BGU-CS-VIL/dpgmm_splitnet`.

2 RELATED WORK

Bayesian non-parametric (BNP) mixture models are commonly used in unsupervised tasks such as clustering, topic modeling, and density estimation (Müller et al., 2015). A common formulation is the Dirichlet Process Mixture Model (DPMM) (Ferguson, 1973; Antoniak, 1974), exemplified by the DPGMM.

Sequential and Parallel Samplers for DPMM. Earlier MCMC methods for DPMM inference were primarily sequential. Two of the chief examples are the Collapsed Gibbs Sampler (Neal, 2000) and a sampler based on the Chinese Restaurant Process (Aldous, 1985). See also Walker (2007) for a technique based on slice sampling (Damlen et al., 1999). Due to the slowness of the sequential samplers, and as an alternative, in recent years several parallel samplers have been proposed. For example, Papaspiliopoulos and Roberts (2008) relied on retrospective sampling while Williamson et al. (2013) proposed a parallel sampler which treats the DPMM as a mixture of DPMMs, allowing to divide the workload between several processes. The SubC sampler (Chang and Fisher III, 2013) is another type of a parallel Gibbs sampler, based on a split/merge MCMC framework (the latter was first introduced by Jain and Neal (2004)). The SubC sampler, which will be discussed in more depth in § 3, alternates between a restricted Gibbs sampler, which handles a fixed amount of components, and the split/merge framework. The latter can modify the number of instantiated components. That work was accompanied with a single-machine, multithreaded implementation (in MATLAB/C++) which uses a shared memory model. For a discussion on parallel samplers and possible associated pitfalls, see Gal and Ghahramani (2014).

Distributed computations in DPMM samplers.

In addition to parallelism, efforts have been made to distribute computations (*e.g.*: Ge et al. (2015); Wang and Lin (2017)). Of particular interest to us is Dinari et al. (2019), a more scalable and faster implementation of the SubC sampler (Chang and Fisher III, 2013). It is based on a distributed memory model and supports not only multiple cores but also multiple machines. In fact, it turns out that even when both the original and the new implementations of the SubC sampler are run on a *single* multi-core machine, the one from Dinari et al. (2019) is faster. Thus, we use it as our baseline.

Variational inference for the DPMM. While we focus on sampling, an important alternative is variational inference, first proposed in the DPMM context by Blei et al. (2006). That approach was also explored for DPMMs by others, *e.g.*, Hoffman et al. (2013); Hughes and Sudderth (2013); Kurihara et al. (2007); Wang et al. (2011); Wang and Blei (2012); Bryant and Sudderth (2012); Fei et al. (2019); Huynh et al. (2016).

3 BACKGROUND

Let D be the dimensionality of the data. One known DPGMM construction (out of several) is based on the following sampling procedure, where we also assume a Normal Inverse Wishart (NIW) prior:

$$\boldsymbol{\pi}|\alpha \sim \text{GEM}(\boldsymbol{\pi};\alpha), \tag{1}$$

$$\theta_k|\lambda \overset{i.i.d.}{\sim} \text{NIW}(\theta_k;\lambda), \qquad \forall k \in \{1,2,\ldots\}, \tag{2}$$

$$z_i|\boldsymbol{\pi} \overset{i.i.d.}{\sim} \text{Cat}(z_i;\boldsymbol{\pi}), \qquad \forall i \in \{1,2,\ldots,N\}, \tag{3}$$

$$\boldsymbol{x}_i|z_i,\theta_{z_i} \sim \mathcal{N}(\boldsymbol{x}_i;\theta_{z_i}), \qquad \forall i \in \{1,2,\ldots,N\}\,. \tag{4}$$

Here $\boldsymbol{\pi} = (\pi_k)_{k=1}^{\infty}$ is a weight vector of infinite length (*i.e.*, $\pi_k > 0$ for every k and $\sum_{k=1}^{\infty} \pi_k = 1$) drawn from the Griffiths-Engen-McCloskey stick-breaking process (GEM) (Pitman, 2002) with a concentration parameter $\alpha > 0$ while $\theta_k = (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ stands for the mean vector and covariance matrix of Gaussian k (so $\boldsymbol{\mu}_k \in \mathbb{R}^D$ and $\boldsymbol{\Sigma}_k$ is an D-by-D symmetric positive-definite matrix) sampled from an NIW distribution whose probability density function (pdf) is denoted by $\text{NIW}(\cdot;\lambda)$ where λ represent the hyperparameters of the prior. Each of the N *i.i.d.* observations $(\boldsymbol{x}_i)_{i=1}^{N} \subset \mathbb{R}^D$ is generated by first drawing a label, $z_i \in \mathbb{Z}^+$, from $\boldsymbol{\pi}$ (*i.e.*, $\text{Cat}(\cdot;\boldsymbol{\pi})$ is the categorical distribution), and then drawing $\boldsymbol{x}_i$ from Gaussian z_i where $\mathcal{N}(\cdot;\theta_{z_i})$ is a D-dimensional Gaussian pdf parameterized by $\theta_{z_i} = (\boldsymbol{\mu}_{z_i}, \boldsymbol{\Sigma}_{z_i})$. Note that the latent variables in Equations (1)–(4) above are $(\theta_k)_{k=1}^{\infty}$, $\boldsymbol{\pi}$, and $(z_i)_{k=1}^{N}$. For more details and alternative constructions, see Sudderth (2006).

3.1 The SubC Sampler

Due to space limits, and since most of the many details of the fairly-elaborated SubC sampler (Chang and Fisher III, 2013) are inessential for understanding our work, their entire algorithm (in the context of the DPGMM) appears in our appendix. Below we provide only a high-level review while also focusing on the part most relevant to us: the splits. The SubC sampler consists of a *restricted* Gibbs sampler and a split/merge framework, which together form an ergodic Markov chain. The operations in each step of the sampler are highly parallelizable while the splits/merges enable the sampler to perform *large moves* (on the optimization surface).

The augmented space. The latent variables, $(\theta_k)_{k=1}^{\infty}$, $\boldsymbol{\pi}$, and $(z_i)_{k=1}^{N}$, are augmented with auxiliary variables as follows. To each z_i, an additional *subcluster label*, $\bar{z}_i \in \{l, r\}$, is added, where "l" and "r" conceptually stand for "left" and "right", respectively. To each component θ_k two subcomponents are added, $\bar{\theta}_{k,l}, \bar{\theta}_{k,r}$, with weights $\bar{\boldsymbol{\pi}}_k$. See Figure 2 for a graphical model of the DPGMM with the augmented space.

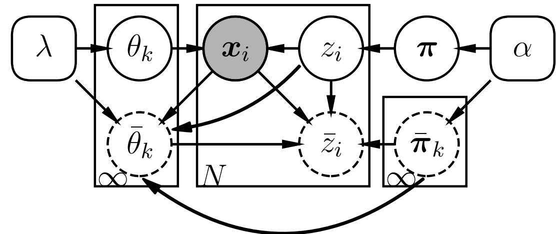


Figure 2: The DPGMM with the augmented space depicted as a graphical model. This figure is based on the one in Chang et al. (2014).

The restricted Gibbs sampler. This restricted sampler is not allowed to change K, the current number of estimated clusters; rather, it can update only the parameters of the existing clusters, and, when sampling the labels, it can assign an observation only to an existing cluster. During the iterations of the restricted sampler, the subclusters evolve together with the clusters. Concretely, in each pair of subclusters the latter evolve using Gibbs sampling in a 2-component GMM (except that each iteration is also conditioned on the cluster label).

The split/merge framework. Splits and merges allow the sampler to change the current number of instantiated components, using the Metropolis-Hastings framework (Hastings, 1970). Let us focus here on the splits. Every certain (user-defined) amount of iterations, the sampler proposes splitting an existing cluster into its subclusters. The split of cluster k is accepted

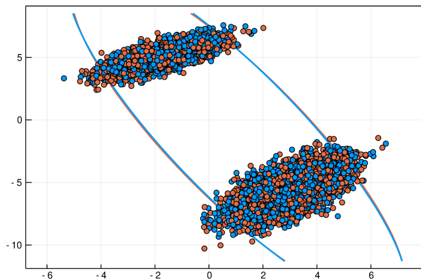


Figure 3: A random initialization (Chang and Fisher III, 2013) yields 2 almost-identical Gaussians (visualized by the almost-identical orange and blue ellipses). At this state, the proposed split will be rejected with a very high probability (Eq. (16)). It will take at least several iterations before the 2-component GMM will imply a split that is likely to be accepted.

with probability $\min(1, H)$ where

$$H = \frac{\alpha\Gamma(N_{k,l}) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_{k,l}};\lambda)\Gamma(N_{k,r}) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_{k,r}};\lambda)}{\Gamma(N_k) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_k};\lambda)} \tag{5}$$

is the Hastings ratio, Γ is the Gamma function, $N_{k,l}$ and $N_{k,r}$ are the number of points in subclusters l or r, respectively, $\boldsymbol{x}_{\mathcal{I}_k}$ denotes the points in cluster k, $\boldsymbol{x}_{\mathcal{I}_{k,s}} \subset \boldsymbol{x}_{\mathcal{I}_k}$ denotes the points in subcluster $s \in \{l, r\}$, and $f_{\boldsymbol{x}}(\cdot;\lambda)$ is the *marginal* data likelihood (see the appendix for the concrete expression). Upon a split acceptance, each of the new clusters is augmented with two subclusters (that must be initialized somehow).

4 METHOD

Let us start with the failure modes of the SubC sampler. In Chang and Fisher III (2013), the subcluster initialization is done by randomly partitioning the cluster of interest into two subclusters. Figure 3 shows an example. Although it is evident that there are, in fact, two distinct clusters, the random initialization of course gives rise to initial subclusters (indicated in the figure in different colors) that are nearly identical (in terms of their estimated 2-component GMM parameters). While simple, this incurs a heavy price in the subsequent iterations of the algorithm, as the restricted Gibbs sampler will relatively-slowly move points between the two subclusters until a sufficiently-good split proposal is reached; *i.e.*, until the resulting 2-component GMM will give rise to a high Hastings Ratio (Eq. (16)).

Moreover, and as we will show, slow convergence is not the only problem with random initializations. In Figure 3 we saw that even when clearly-distinct clusters exist, the two randomly-initialized subclusters are almost indistinguishable. Eventually, the restricted Gibbs sampler will give rise to the correct subclusters, but it will take time to get there. A more severe problem is shown in Figure 1a: on this challenging synthetic dataset, using random subcluster initializations the SubC sampler failed to achieve good clustering results, despite having converged; *i.e.*, it missed many desired splits, and by converging to a poor local maximum, grossly underestimated K. As demonstrated in Figure 4, this phenomenon notably occurs in challenging datasets characterized by densely-packed and partially-overlapping clusters and *especially in lower dimensions* as there it is often hard to distinguish between clusters.

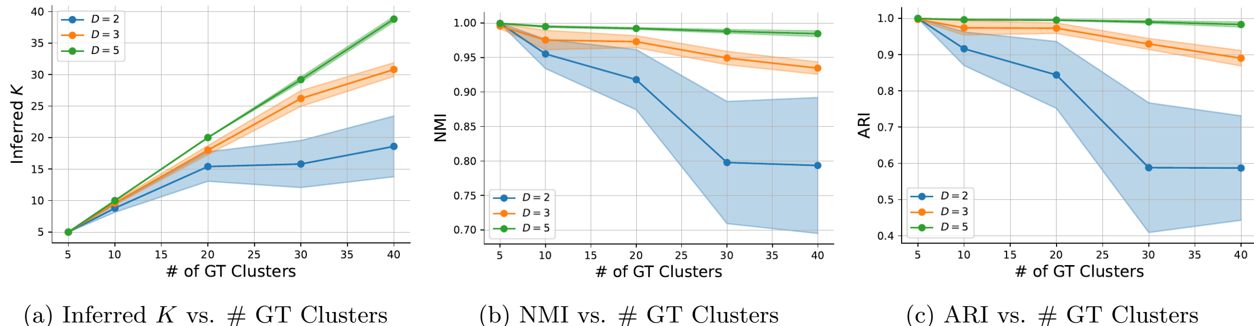
While random initialization is often adequate for clustering algorithms in general, we argue that it is a poor choice in the context of the SubC sampler. Recall that the whole point of maintaining the subcluster mixtures is to facilitate desired splits. It follows that this goal must be taken into consideration when initializing the subclusters. *We suspect this insight was missed because the original sampler (Chang and Fisher III, 2013) was already, overall, highly effective, producing at-the-time state-of-the-art results.*

4.1 The Proposed Solutions

In this section we propose two alternative subcluster initialization methods. The rationale behind proposing "smart splits" is based on the sampler's strong reliance on H (Eq. (16)): a split proposal is accepted mostly when the (marginal) likelihood according to the two subclusters sufficiently improves upon the (marginal) likelihood according to the original cluster whose possible split is under consideration. Particularly, when the subclusters are almost indistinguishable (*e.g.*, as in Figure 3), H tends to be low. When the subclusters are initialized randomly, then, by construction, the region "covered" by one subcluster is roughly the same as the other. Thus, *fast convergence* from such an initial state into a state where the two subclusters cover sufficiently-different regions to justify a split is unlikely. Given the above, it is preferable to initialize the subclusters in a way such that the covered regions are sufficiently distinct, resulting in an immediately higher H or at least one that will be achieved within a few iterations. Importantly, the initialization scheme, as it is called many times during the sampler's run, cannot be too slow or computationally expensive. Let us first consider a simple solution.

4.1.1 Subcluster initialization via K-means.

The first solution we propose is based on a simple heuristic: create the subclusters by running on the cluster the K-means algorithm, with $K = 2$. Other classical clustering algorithms such as K-medoids, EM-GMM, *etc.* are also optional, but this option is simple,



(a) Inferred K vs. # GT Clusters (b) NMI vs. # GT Clusters (c) ARI vs. # GT Clusters

Figure 4: Deterioration analysis of the SubC sampler (Chang and Fisher III, 2013). Results were obtained by running the SubC sampler on increasingly-denser synthetic GMM datasets. Note that it is the *lower* dimensions where the sampler struggles the most (as there it is harder to discover that a cluster needs to be split).

intuitive and computationally cheap. Also note that one obvious advantage of K-means (over a random initialization) is that it yields two **non-overlapping** subclusters. Another related advantage, verified empirically, is that usually, in situations where the cluster needs to be split but the initial Hastings Ratio is still too low, a SubC sampler that uses K-means subcluster iterations requires fewer iterations, until the ratio is high enough. In practice, K-means is fast, adding a relatively-insignificant computational overhead. However, as data becomes more densely packed and of higher dimension, the effectiveness of K-means initialization will decrease.

4.1.2 Subcluster initialization via SplitNet

Our second proposed solution is based on Deep-Learning (DL). The latter is attractive here as its feedforward pass is fast, and since it excels in learning complex data patterns. Specifically, given a cluster we want the deep net to predict useful subclusters. Below we explain the learning-task formulation, the choice of architecture, loss function, data generation, and training procedure. Importantly, no manual labeling of data is needed.

The first step is to formulate the task at hand in terms suitable for DL. Intuitively, we would like our so-called SplitNet model, denoted by f, to learn to distinguish between the two regions of the highest density. Let X be a cluster of N points in $\mathbb{R}^D$. SplitNet's **input** is X, represented as a matrix, $X \in \mathbb{R}^{N\times D}$, where each row corresponds to a a different point: $X = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_n]^T$. Its **output** (thought of as subcluster assignments) is a binary-valued N-length column vector: *i.e.*, $\widehat{Z} = (\widehat{z}_i)_{i=1}^N = \begin{bmatrix} \widehat{z}_1, \dots, \widehat{z}_N \end{bmatrix}$ where $\widehat{z}_i \in \{0, 1\}$.

Some inherent characteristics of the input and output must be considered when formulating the neural model and its accompanying training procedure. *First*, the loss should be invariant to a permutation of the rows of X and $\widehat{Z}$, provided the *same* permutation applies to both. In other words, we should consider the input/output as a set, $\{(x_i, \hat{z}_i)\}_{i=1}^N$. *Second*, we cannot know in advance the number of points, N. *Third*, the label ordering (*i.e.*, $(0, 1)$ or $(1, 0)$) is arbitrary and interchangeable; what is essential here is only the partition itself, not the "names" of the labels. The loss for predicting $\widehat{Z}$ should be the same if we replace *all* the $(\widehat{z}_i)_{i=1}^N$ with their 1-complement; namely, $(1 - \widehat{z}_i)_{i=1}^N$.

Let us summarize these considerations: 1) The model must be able to work with any number of points at the input (varying N); 2) The model needs to be invariant to permutations of the rows of X (as $\{(x_i, \hat{z}_i)\}_{i=1}^N$ is a set); 3) The model's output (subcluster assignments) must be interchangeable ($(1, 0) \leftrightarrow (0, 1)$).

Model Architecture: To fall in with the above considerations, we chose to base the model on a deep architecture that can process sets, and particularly, the Set Transformer (ST) architecture (Lee et al., 2019). Another option could be based on PointNet (Qi et al., 2017); however, we chose ST due to its relative simplicity, smaller model size and better performance (in our context). The ST architecture is permutation invariant, can work with any number of data points N (for a fixed d), and learns the relations and interactions between the data points through an attention mechanism.

We formulated the model prediction as a binary "segmentation" task: each point in the input is assigned to either of the two subclusters via a binary label "0" or "1". To allow this, we need the model to output a neuron per data-point in the input, corresponding to the subcluster assignment. To that end, we construct the ST in a particular way. Using known basic building blocks of the ST such as Induced Set Attention Block (ISAB), row Feed Forward (rFF), and Pooling by Multihead Attention (PMA), we use the following encoder and decoder: The **Encoder**, denoted by $H_X = \mathrm{ISAB}_L(X)$, is a stack of L components of ISAB(). The **Decoder**,

denoted by $Z = \text{rFF}(\text{PMA}_M(H_X))$, is a stack of M components of PMA() followed by a single row-Fully-Connected layer. For more details on the ST architecture, these building blocks, and the L and M hyperparameters, see the appendix.

Generating Data for Training As is usual in DL, the training data is one of the most critical factors for model performance. Therefore, it must be of high quality and provided with enough variability so that the model would be able to generalize well to unseen and challenging examples (recall that here each example is an entire cluster). Fortunately, in our case, where the overarching goal is improving a DPGMM method, we can assume the training data comes from GMM. Thus, it is relatively simple to define a data-generation process with full control over various parameters such as the Gaussians' proximity to each other, the size and shape of the covariance matrices, how many points are drawn from each Gaussian, the number of Gaussians, how overlapping they are, and more. Recall the intuition that we would like to learn how to distinguish between the two most highly-dense regions of the data. Through experimentation, we found that training the model on exactly 2-component GMM's was better than training it on more than 2 components. This is also where the model can learn difficult splits, where other methods (*e.g.* K-means) are likely to fail. We generate a dataset with specific characteristics: cluster imbalance (*i.e.*, clusters of different sizes), non-spherical covariance matrices, and the degree of overlap between the clusters. Another critical factor when generating the data, is ensuring that the data is "splittable", *i.e.*, that there exists a split with a relatively high Hastings ratio. Any sample with a low ratio is thus discarded. This mitigates the introduction of noisy data into the training. Generated examples of samples of varying difficulty are shown in Figure 5. Note that each training example is an entire "dataset", *i.e.*, a set of points drawn from a different 2-component GMM, where the number of points vary across the examples.

Loss Function. In each example, we know for each point from which Gaussian it was drawn. This provides synthetic Ground Truth (GT) labels: $Z = (z_i)_{i=1}^N$. Our supervised loss function, which uses these GT labels, is based on the standard Binary Cross-Entropy (BCE) used for binary classification, which in our case is per data point each in example dataset. Recall that for a GT label $z_i \in \{0,1\}$ and a prediction $\hat{z}_i \in [0,1]$, the regular BCE loss is:

$$\ell_{\text{BCE}}(z_i, \hat{z}_i) = -(z_i \log \hat{z}_i + (1 - z_i) \log (1 - \hat{z}_i)) . \quad (6)$$

It is thus tempting to define (for an example dataset):

$$\mathcal{L}_{\text{BCE}}(Z, \hat{Z}) = \tfrac{1}{N} \sum\nolimits_{i=1}^{N} \ell_{\text{BCE}}(z_i, \hat{z}_i) . \quad (7)$$

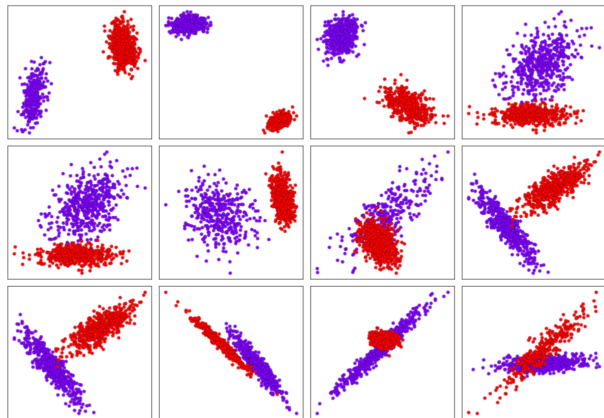


Figure 5: Examples of generated training data. Each panel represents a single training example; *i.e.*, a cluster that needs to be split into two, according to the (ground-truth) colors. First row: "easy" data, middle row: "medium-difficult" data, last row: "hard" data.

However, recall that label ordering is arbitrary. As we would not want to penalize the model for predicting, *e.g.*, a perfect split which is the exact mirror of the GT labels, we extend the BCE loss to ensure its invariance to such switching. First, we convert the GT labels to one-hot encoding vectors, $Z^{\text{oh}} = (z_i^{\text{oh}})_{i=1}^N$ where

$$z_i^{\text{oh}} = \begin{cases} (1,0), & \text{if } z = 0 \\ (0,1), & \text{if } z = 1 \end{cases} . \quad (8)$$

With this new matrix Z^{oh}, we compute the BCE loss per the two possible cases and then take the minimum, making the loss invariant to label-switching:

$$\mathcal{L}_{\text{SplitNet}}(Z^{\text{oh}}, \hat{Z}) = \min_{j \in \{0,1\}} \tfrac{1}{N} \sum\nolimits_{i=1}^{N} \ell_{\text{BCE}}(z_{i,j}^{\text{oh}}, \hat{z}_i) \quad (9)$$

where $z_{i,j}^{\text{oh}}$ is, in zero-based indexing, the j entry of z_i^{oh}.

Training. We trained three models for dimensions $D = 2, 10, 20$. The full training details (most of which are standard), as well as the curriculum learning that we used, appear in the appendix.

5 EXPERIMENTS AND RESULTS

We test the proposed approach in three different modes: 1) SplitNet's performance on test data similar to its training data; 2) qualitative comparisons between three variants of the SubC sampler (each with a different type of subcluster initializations); 3) quantitative comparisons between those variants as well as other key methods on synthetic and real datasets.

Performance Metrics. We use several metrics: Log-posterior (LP) (when applicable); Normalized Mutual

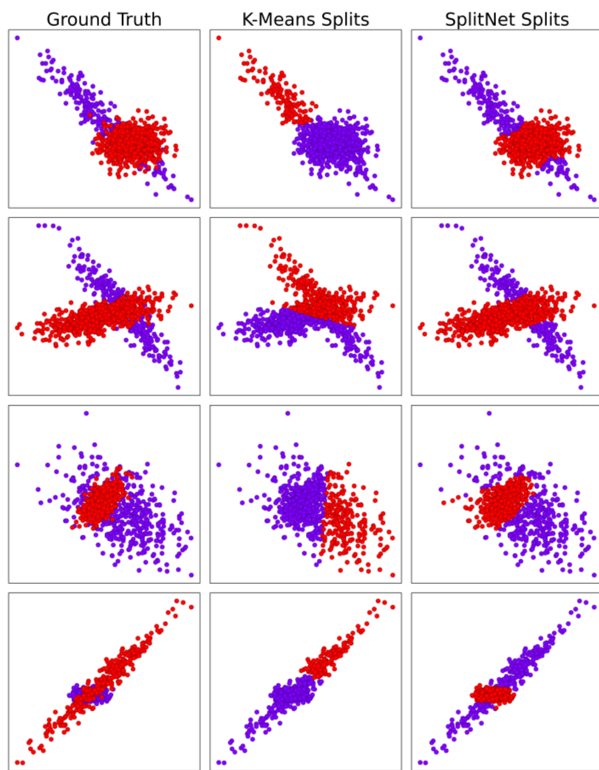


Figure 6: Splits on difficult test datasets. **Left column**: GT, **Middle**: K-means, **Right**: SplitNet (the last example is a success too, despite the reversed colors caused by label switching).

Information (NMI); Adjusted Rand Index (ARI); the inferred K (or, equivalently, K-MAE – the mean absolute error between true and the estimated K). In all of the experiments, the metrics reported (boxplots, mean+std. dev. values) correspond to 10 runs.

Choice of Test data. We divide the experiments into synthetic data consisting of GMM data of various dimensions and numbers of clusters, and real data derived from several computer-vision datasets.

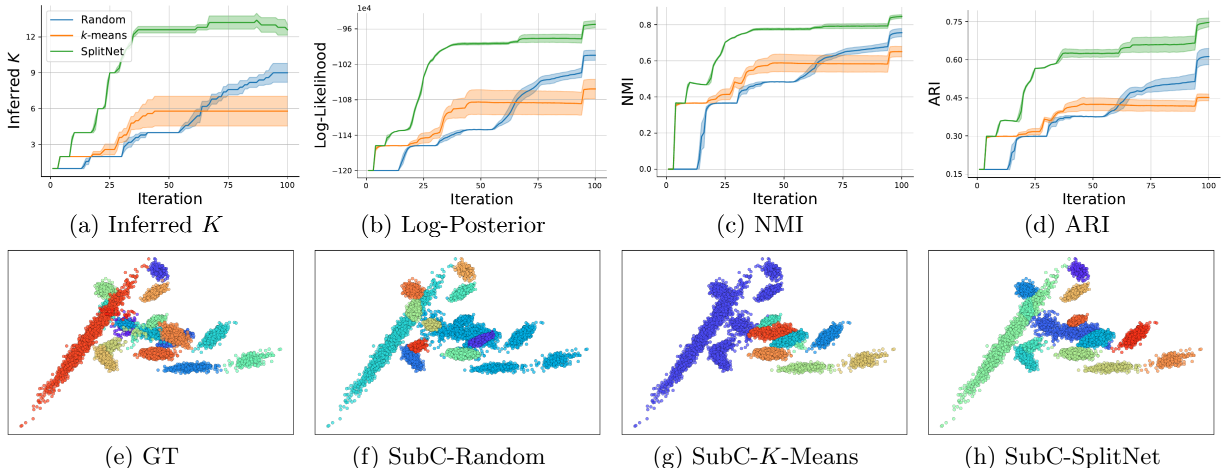
Algorithms. Chiefly, we seek to study the effect of the subcluster-initialization type on the SubC sampler. To that aim, we compare the same SubC sampler from Chang and Fisher III (2013) (using its more recent implementation (Dinari et al., 2019)) but with three different subcluster-initialization types: 1) Random (*i.e.*, the baseline), denoted as SubC-Random. 2) 2-means (using the K-means implementation from Julia's Clustering package), denoted as SubC-K-means. 3) SplitNet (based on the prediction of our trained deep net), denoted as SubC-SplitNet. We also included two other DPGMM methods: 4) A DPGMM implementation from scikit-learn (Pedregosa et al., 2011), denoted as SK-DPGMM. It is a Bayesian GMM with a Dirichlet process prior fitted with variational inference. That implementation requires the specification of an upper bound on K (set to 1000) and 5) The Memoized Online Variational Bayes (Hughes and Sudderth, 2013), denoted as moVB. Lastly, for completeness, we include the commonly-used clustering methods 7) K-means and 8) EM-GMM. (using scikit-learn's implementations), where to these methods we had to provide the GT K, giving them an unfair advantage.

5.1 Synthetic Data

We start with a sanity check. We visually compared SplitNet and K-means on the test sets generated from the same generative process used to create SplitNet's training data. Figure 6, showing select examples of "hard-to-split" datasets, suggests the SplitNet predicts better splits than those found by K-means. For more results see the appendix. We now return to the original task of clustering with an unknown K.

Qualitative Comparisons. First, we compare SubC-Random, SubC-K-means, and SubC-SplitNet on multiple 2D synthetic GMM datasets, and run each method 10 times on the *same* dataset. In Figure 7, which presents a fairly-difficult example of such a dataset, we see that the SubC-SplitNet achieves the fastest and better convergence and was also most stable (*i.e.*, low variance). Note that (in this particular example) SubC-K-means was better than SubC-Random at the early steps of the runs but was eventually outperformed by it. Additional examples appear in the appendix.

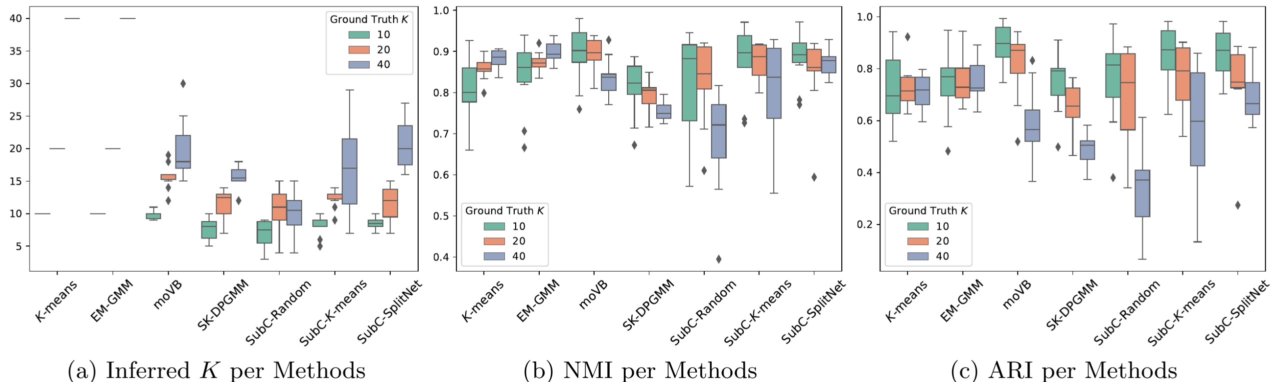
Quantitative Comparisons. In these comparisons we also add the other methods. We considered multiple GMM configurations. Each configuration consists of a choice of D, K, and n where in each configuration we generated 10 GMMs, drawing n points from each, to generate 10 different datasets. On each such dataset, all methods ran for a fixed number of 200 iterations (*i.e.* no early-stopping for convergence). Figure 8 presents the results for $D = 2$, $K \in \{10, 20, 40\}$, and $n = 20,000$ – computed for each method across the 10 datasets – in boxplot form. Note that the SplitNet-based sampler degrades the least as the number of clusters increases, while achieving the best metrics, with the least variance. Additional results, in higher dimensions and higher K values, are in the appendix. Of note, the slowest methods, by far, are the SK-DPGMM and moVB, as they were consistently slower by about 2 and 1 orders of magnitude, respectively, compared with all the SubC samplers (whose times were similar to each other – each was ~ 5 [sec] – as the number of iterations was fixed). We emphasize that all algorithms were run on a single thread (despite the fact that the SubC-* samplers support multiple threads).



(a) Inferred K (b) Log-Posterior (c) NMI (d) ARI

(e) GT (f) SubC-Random (g) SubC-K-Means (h) SubC-SplitNet

Figure 7: Subcluster initialization types: Comparison on 2D Data with K=20.



(a) Inferred K per Methods (b) NMI per Methods (c) ARI per Methods

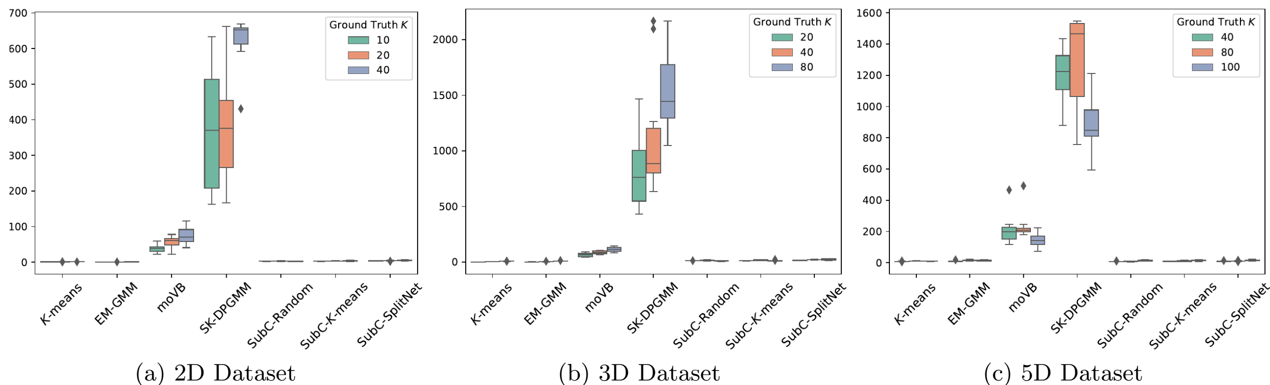
Figure 8: Performance on 2D Datasets.

Table 1: Results on Real Datasets

		SK-DPGMM	MoVB	SubC-Random	SubC-K-means	SubC-SplitNet	K-means	EM-GMM
MNIST	K-MAE:	$10.0 \pm .00$	1.00 ± 0.00	0.80 ± 0.44	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	–	–
	NMI:	0.68 ± 0.01	0.64 ± 0.00	0.66 ± 0.04	0.68 ± 0.01	$\mathbf{0.69 \pm 0.01}$	0.51 ± 0.01	0.67 ± 0.01
	ARI:	0.47 ± 0.02	0.46 ± 0.00	0.48 ± 0.04	0.51 ± 0.00	$\mathbf{0.52 \pm 0.01}$	0.37 ± 0.01	0.53 ± 0.04
FMNIST	K-MAE:	$10.0 \pm .00$	$\mathbf{0.00 \pm 0.00}$	1.40 ± 0.54	1.69 ± 0.89	1.20 ± 2.16	–	–
	NMI:	0.57 ± 0.01	$\mathbf{0.58 \pm 0.00}$	0.56 ± 0.01	0.57 ± 0.00	$\mathbf{0.58 \pm 0.01}$	0.51 ± 0.01	0.57 ± 0.02
	ARI:	0.35 ± 0.01	$\mathbf{0.39 \pm 0.00}$	0.36 ± 0.01	0.36 ± 0.01	0.38 ± 0.01	0.53 ± 0.00	0.38 ± 0.01
CIFAR10	K-MAE:	$10.0 \pm .00$	14.00 ± 0.00	1.30 ± 0.68	$\mathbf{1.00 \pm 0.82}$	2.00 ± 0.00	–	–
	NMI:	0.71 ± 0.01	0.68 ± 0.00	0.68 ± 0.02	0.72 ± 0.02	$\mathbf{0.74 \pm 0.00}$	0.79 ± 0.00	0.72 ± 0.00
	ARI:	0.58 ± 0.01	0.53 ± 0.00	0.55 ± 0.04	0.61 ± 0.05	$\mathbf{0.64 \pm 0.00}$	0.78 ± 0.00	0.60 ± 0.00
CIFAR20	K-MAE:	$20.0 \pm .00$	$\mathbf{2.00 \pm 0.00}$	6.90 ± 1.85	8.20 ± 1.47	8.50 ± 1.95	–	–
	NMI:	0.44 ± 0.01	0.41 ± 0.00	0.43 ± 0.01	$\mathbf{0.56 \pm 0.01}$	0.46 ± 0.01	0.49 ± 0.00	0.44 ± 0.00
	ARI:	0.20 ± 0.01	0.19 ± 0.00	0.25 ± 0.01	$\mathbf{0.28 \pm 0.01}$	0.27 ± 0.01	0.33 ± 0.00	0.24 ± 0.00
STL-10	K-MAE:	$10.0 \pm .00$	10.00 ± 0.00	1.80 ± 1.09	1.20 ± 0.83	$\mathbf{1.00 \pm 0.00}$	–	–
	NMI:	0.63 ± 0.00	0.62 ± 0.00	0.60 ± 0.03	0.63 ± 0.02	$\mathbf{0.67 \pm 0.00}$	0.70 ± 0.00	0.61 ± 0.00
	ARI:	0.51 ± 0.01	0.51 ± 0.00	0.47 ± 0.05	0.52 ± 0.04	$\mathbf{0.56 \pm 0.00}$	0.67 ± 0.00	0.46 ± 0.00

5.2 Real Data

We now turn to the real datasets. As these are high dimensional (*i.e.*, images), we first use dimensionality reduction. In the MNIST and Fashion MNIST datasets, we project the images themselves via Principal Component Analysis (PCA) to a 20-dimensional space. In CIFAR10, CIFAR20, and STL10, we compute deep features using a recent unsupervised



(a) 2D Dataset (b) 3D Dataset (c) 5D Dataset

Figure 9: Running times (in [sec]) on different datasets.

method (Van Gansbeke et al., 2020). The deep features are 512-dimensional, so we apply PCA on them, projecting the data to $\mathbb{R}^{20}$. Full details about the dataset can be found in the appendix. In each dataset, following the PCA, each method was run 10 times. Table 1 summarizes the results. We observe that in most cases, the SubC-K-Means and SubC-SplitNet outperform the others. In fact, they sometimes even beat the parametric methods (K-means; EM-GMM) that were provided with the GT K.

5.3 Analysis of the Results

Performance: In relatively-easy datasets with a low K (*e.g.*, $K = 10$ in Figure 8), most methods perform similarly. However, when the dataset become harder (higher K; more densely packed), the proposed SubC-K-Means and (more so) SubC-SplitNet usually start outperforming the others.

Convergence Rate, Speed, and Stability: As Figure 7 and similar figures in the appendix show, SubC-K-Means and SubC-SplitNet converge faster than SubC-Random. Those figures, together with the std. dev. values in Table 1 also demonstrate that SubC-SplitNet is the most stable (*i.e.*, has the lowest variance) among the three methods.

When compared with other methods (moVB; SK-DPGMM), the SubC-* methods are 1 or 2 orders of magnitude faster, as can be seen in Figure 9: moVB and (especially) SK-DPGMM are much slower than the SubC samplers. Of note, the fact that the figure might suggest that SubC-SplitNet sampler took (*very slightly!*) more time than the other two SubC methods is misleading: 1) Recall we fixed the number of iterations to 200, which sufficed for convergence for all methods. However, the SubC-SplitNet needed far fewer iterations as it converged faster. 2) the implementation of SplitNet-based method combines software from Julia (from Dinari et al. (2019)) and PyTorch (SplitNet). Currently, the cross talk between the languages costs a little overhead which we believe could be eliminated by some software engineering. We also emphasize that all algorithms were run on the same single machine using a single thread despite the fact that the SubC-* methods support both multiple threads and multiple machines.

Performance and Generalization on Real Datasets. Despite the fact that SplitNet was trained on synthetic 2-component GMM data, is it is able to perform well and generalize to real datasets, outperforming all other methods on most datasets and metrics.

6 CONCLUSION

In this paper we have identified failure modes of the SubC sampler (Chang and Fisher III, 2013). These failures are tied to the random subcluster initializations. As a remedy, we proposed two alternative subcluster initializations: K-means and a new DL method, called SplitNet. We showed that the SubC-SplitNet sampler usually outperforms both the baseline SubC sampler and the SubC-K-means sampler in performance, convergence speed, and stability. The added value is especially apparent in challenging datasets. SplitNet's limitation is that it is hard to scale it to very high dimensions; training on such data is expensive in both time and memory. However, in high dimensions, clusters are usually more easily separable and the failure modes of the baseline sampler are less frequent, thus better initializations are less needed there anyway.

Future work may explore similar ideas for non-Gaussian mixtures: while it is reasonable to believe that the proposed method will also work well for many other continuous distributions, extending it to discrete distributions (*e.g.*, multinomials) will require new ideas. Another interesting idea is to train SplitNet to directly maximize Hasting Ratio of the split.

References

D. J. Aldous. Exchangeability and related topics. In *École d'Été de Probabilités de Saint-Flour XIII—1983*. Springer, 1985. 2

C. E. Antoniak. Mixtures of Dirichlet processes with applications to Bayesian nonparametric problems. *The annals of statistics*, 1974. 2

D. M. Blei, M. I. Jordan, et al. Variational inference for Dirichlet process mixtures. *Bayesian analysis*, 2006. 2

M. Bryant and E. B. Sudderth. Truly nonparametric online variational inference for hierarchical Dirichlet processes. In *NIPS*, 2012. 2

J. Chang and J. W. Fisher III. Parallel sampling of DP mixture models using sub-cluster splits. In *NIPS*, 2013. 1, 2, 3.1, 3, 4, 4, 5, 6, B, 1

J. Chang et al. *Sampling in computer vision and Bayesian nonparametric mixtures*. PhD thesis, Massachusetts Institute of Technology, 2014. 2, C

A. Coates, A. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning. In *AISTATS*, pages 215–223. JMLR Workshop and Conference Proceedings, 2011. A.3

P. Damlen, J. Wakefield, and S. Walker. Gibbs sampling for bayesian non-conjugate and hierarchical models by using auxiliary variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 1999. 2

L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 2012. A.3

O. Dinari, A. Yu, O. Freifeld, and J. Fisher III. Distributed MCMC inference in Dirichlet process mixture models using Julia. In *IEEE CCGRID Workshop on High Performance Machine Learning*, 2019. 1, 2, 5, 5.3

Z. Fei, K. Liu, B. Huang, Y. Zheng, and X. Xiang. Dirichlet process mixture model based nonparametric Bayesian modeling and variational inference. In *Chinese Automation Congress*. IEEE, 2019. 2

T. S. Ferguson. A Bayesian analysis of some nonparametric problems. *The Annals of Statistics*, 1973. 2

Y. Gal and Z. Ghahramani. Pitfalls in the use of parallel inference for the Dirichlet process. In *ICML*, 2014. 2

H. Ge, Y. Chen, M. Wan, and Z. Ghahramani. Distributed inference for Dirichlet process mixture models. In *ICML*, 2015. 2

A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin. *Bayesian data analysis*. Chapman and Hall/CRC, 2013. C

C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 2020. D

W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. 1970. 3.1

M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *JMLR*, 2013. 2

M. C. Hughes and E. Sudderth. Memoized online variational inference for Dirichlet process mixture models. In *NIPS*, 2013. 2, 5

V. Huynh, D. Phung, and S. Venkatesh. Streaming variational inference for Dirichlet process mixtures. In *ACML*. PMLR, 2016. 2

S. Jain and R. M. Neal. A split-merge Markov chain monte carlo procedure for the Dirichlet process mixture model. *Journal of Computational and Graphical Statistics*, 2004. 2

A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. A.3

K. Kurihara, M. Welling, and N. Vlassis. Accelerated variational Dirichlet process mixtures. In *NIPS*, 2007. 2

J. Lee, Y. Lee, J. Kim, A. Kosiorek, S. Choi, and Y. W. Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *ICML*. PMLR, 2019. 4.1.2, F, F

S. Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory*, 1982. 1

P. Müller, F. A. Quintana, A. Jara, and T. Hanson. *Bayesian nonparametric data analysis*. Springer, 2015. 2

R. M. Neal. Markov chain sampling methods for Dirichlet process mixture models. *Journal of computational and graphical statistics*, 2000. 2

O. Papaspiliopoulos and G. O. Roberts. Retrospective markov chain monte carlo methods for dirichlet process hierarchical models. *Biometrika*, 2008. 2

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NIPS 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learn pdf. D

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *JMLR*, 2011. 5, D

Pitman. Combinatorial stochastic processes. Technical report, Technical Report 621, Dept. Statistics, UC Berkeley, 2002. Lecture notes, 2002. 3

C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*, 2017. 4.1.2

C. Robert and G. Casella. *Monte Carlo statistical methods*. Springer Science & Business Media, 2013. 1

E. B. Sudderth. *Graphical models for visual object recognition and tracking*. PhD thesis, Massachusetts Institute of Technology, 2006. 3

W. Van Gansbeke, S. Vandenhende, S. Georgoulis, M. Proesmans, and L. Van Gool. Scan: Learning to classify images without labels. In *ECCV*, 2020. 5.2, A.3

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017. F, F

P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020. D

S. G. Walker. Sampling the dirichlet mixture model with slices. *Communications in Statistics–Simulation and Computation®*, 2007. 2

C. Wang and D. M. Blei. Truncation-free online variational inference for Bayesian nonparametric models. In *NIPS*, 2012. 2

C. Wang, J. Paisley, and D. Blei. Online variational inference for the hierarchical Dirichlet process. In *AISTATS*, 2011. 2

R. Wang and D. Lin. Scalable estimation of Dirichlet process mixture models on distributed data. In *IJCAI*, 2017. 2

S. Williamson, A. Dubey, and E. Xing. Parallel Markov Chain Monte Carlo for nonparametric mixture models. In *ICML*, 2013. 2

H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017. A.3

Supplementary Material: Common Failure Modes of Subcluster-based Sampling in Dirichlet Process Gaussian Mixture Models – and a Deep-learning Solution

This appendix contains the following: 1) additional results and some more technical details about the experiments; 2) the SubC sampler's full algorithm; 3) the expressions for the posterior hyperparameters and Marginal Data Likelihood in a Gaussian Model with an NIW prior; 4) additional training details; 5) additional training data details; 6) details about the Set Transformer architecture.

A Additional Results and Technical Details about Experiments

For completeness, below we provide the additional results that were alluded do in the paper but were omitted there due to space limits.

A.1 Synthetic-data Experiments

A.1.1 SplitNet's Performance on Test Dataset.

Here we visually compare SplitNet and K-means splits on a few test sets generated from the same generative process used to create SplitNet's training data. To that aim, We created 3 test-sets, conceptually tagged as "easy" (Figure 10), "medium"(Figure 11) and "hard" (Figure 6).

Remark 1 *Note that in some of the examples (for either K-means or SplitNet) there exists a label switching w.r.t. the Ground Truth (GT), but that by itself does not indicate a failure of either of the methods since the ordering of the labels arbitrary.*

As the figures show, both methods succeed in the easy example. Both methods still do well with the medium-level examples, though it is fair to say that SplitNet slightly wins by a small margin. However, in most difficult example it is evident that SplitNet's significantly outperforms K-means.

A.1.2 Qualitative Comparisons.

Here we present two more quantitative comparisons (in addition to the one in the paper) between the SubC-Random, SubC-K-means, and SubC-SplitNet methods on two different 2D synthetic GMM datasets. The first is for data drawn from a GMM with $K = 10$ components, while in the second we used a GMM with $K = 40$ components. On each of the two datasets we run each method 10 times. The results are shown in Figure 12 ($K = 10$) and Figure 13 ($K = 40$). The results are consistent with ones shown in the paper, suggesting that SubC-SplitNet outperforms, overall, the other two in terms of performance, convergence speed and low variance.

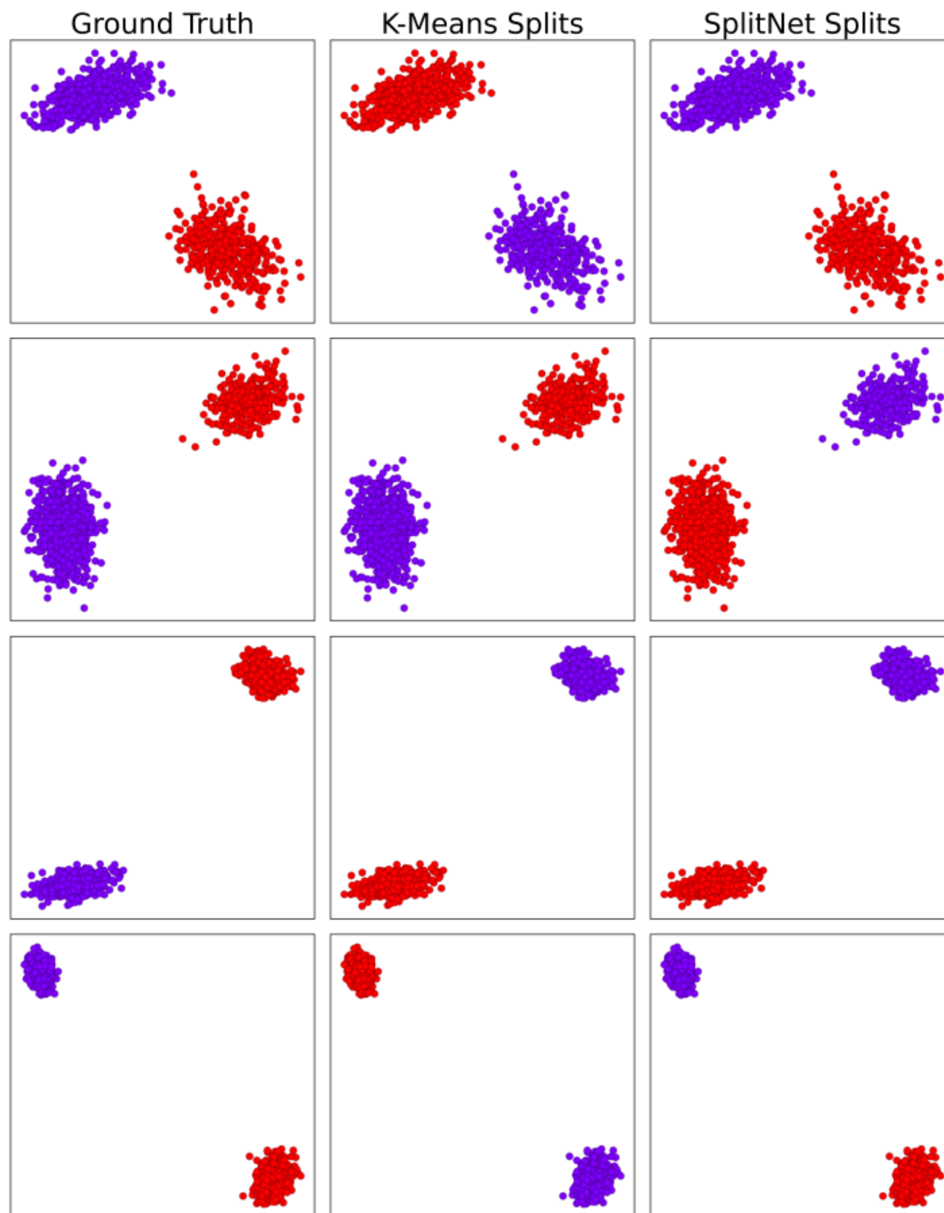


Figure 10: Splits on easy test datasets. **Left column**: GT, **Middle**: K-means, **Right**: SplitNet (Note that in some of the examples there is label switching w.r.t. the GT, but that by itself does not indicate a failure of either of the methods).

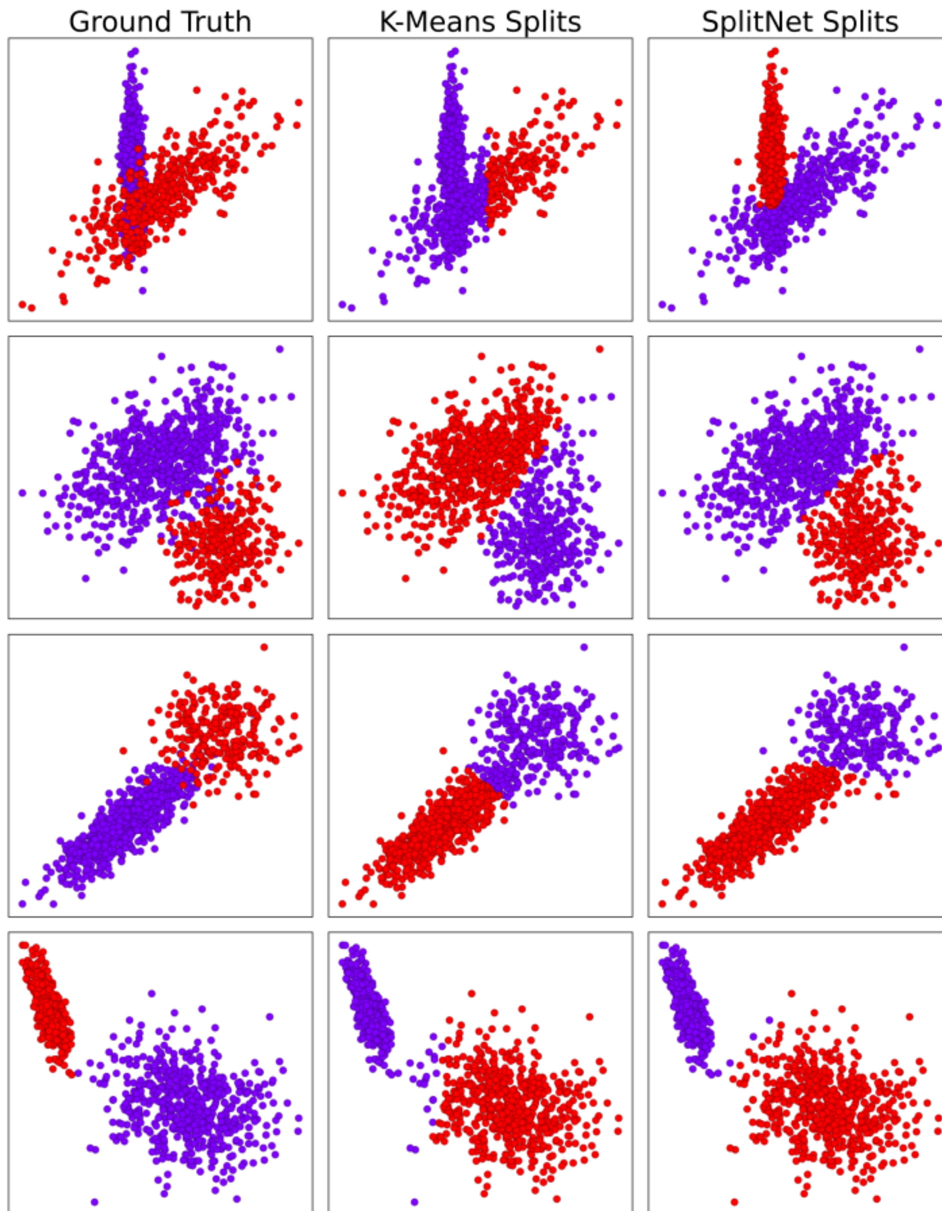
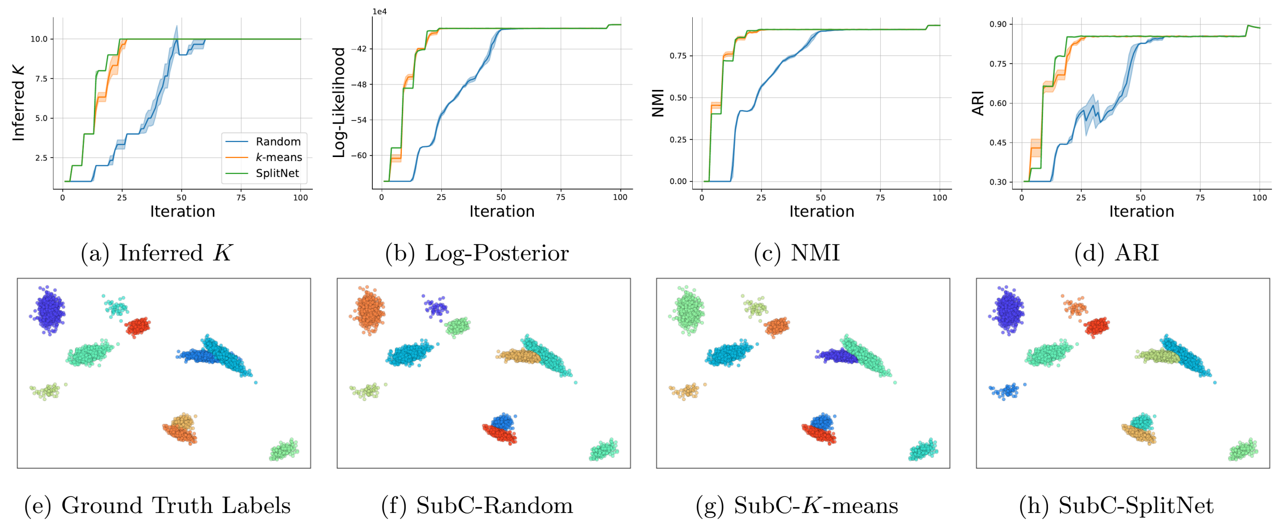


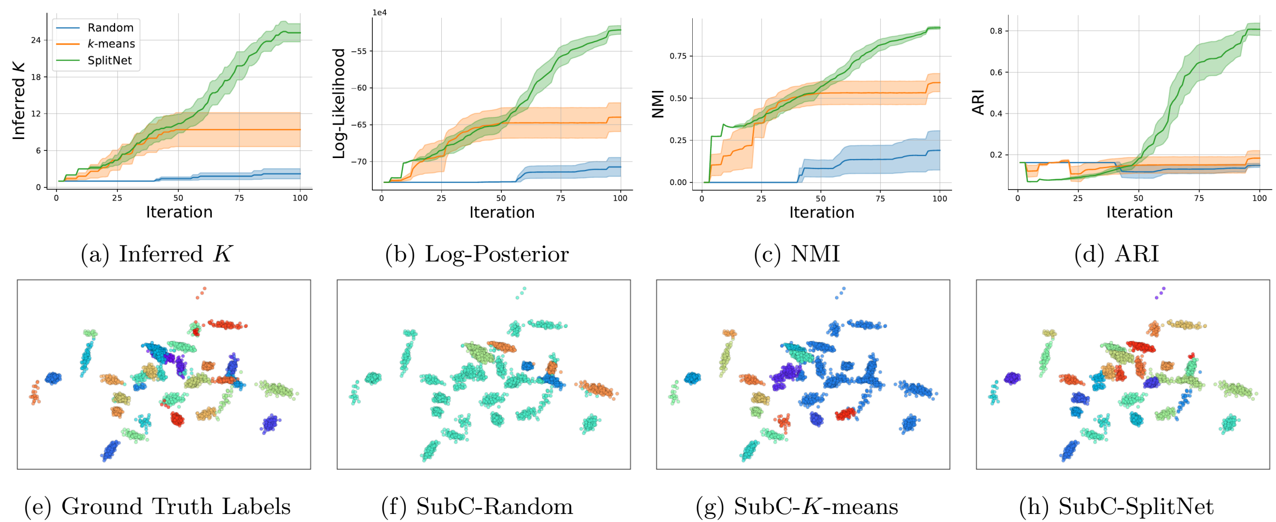
Figure 11: Splits on medium-difficulty test datasets. **Left column**: GT, **Middle**: K-means, **Right**: SplitNet (Note that in some of the examples there is label switching w.r.t. the GT, but that by itself does not indicate a failure of either of the methods).



(a) Inferred K (b) Log-Posterior (c) NMI (d) ARI

(e) Ground Truth Labels (f) SubC-Random (g) SubC-K-means (h) SubC-SplitNet

Figure 12: Comparison on 2D Data with $K = 10$.



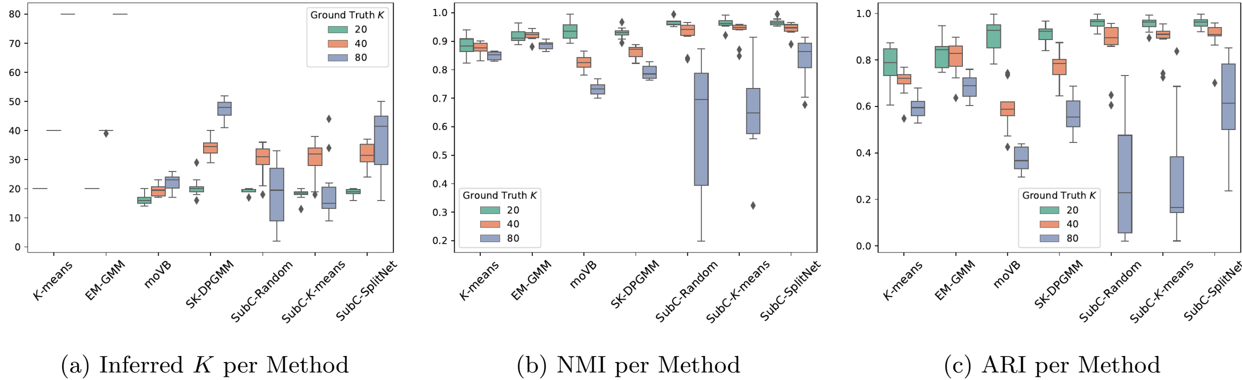
(a) Inferred K (b) Log-Posterior (c) NMI (d) ARI

(e) Ground Truth Labels (f) SubC-Random (g) SubC-K-means (h) SubC-SplitNet

Figure 13: Comparison on 2D Data with $K = 40$.

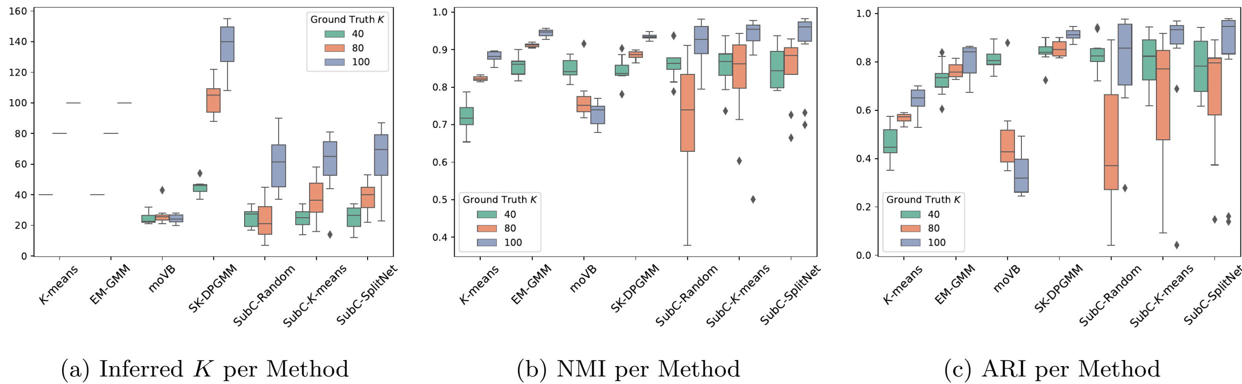
A.1.3 Quantitative Comparisons.

Here we present additional GMM test configurations with a greater number of GT clusters and on higher dimensions: $D \in \{3, 5, 10\}$, and the results are shown in Figure 14, Figure 15 and Figure 16 respectively. In each setting, we generate 10 different datasets. We observe that the SubC-SplitNet method generally outperforms the rest of the methods. Also, note that the SK-DPGMM method consistently over-estimates the number of GT clusters, which me explain why its NMI and ARI values are high. The performance of the moVB method deteriorates as the number of clusters increases. Again, we stress that the SK-DPGMM method is 2 orders of magnitude slower than the SubC-* methods (*e.g.*, 1500 seconds vs. 15 seconds).



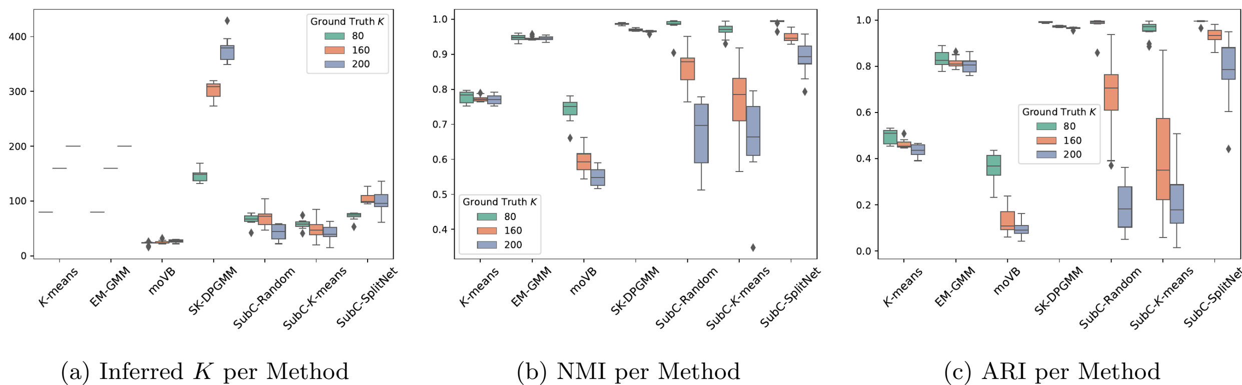
(a) Inferred K per Method (b) NMI per Method (c) ARI per Method

Figure 14: Performance on 3D Datasets.



(a) Inferred K per Method (b) NMI per Method (c) ARI per Method

Figure 15: Performance on 5D Datasets.



(a) Inferred K per Method (b) NMI per Method (c) ARI per Method

Figure 16: Performance on 10D Datasets.

A.2 Experiments Details

All the experiments and model training were done on an Ubuntu 16.04 machine with Intel® Xeon(R) CPU E5-2630 @ 2.20GHz Processor, with an NVIDIA Tesla P100 (16 Gb VRAM) GPU.

The following SW language/packages versions were used: Julia: 1.4, Python 3.8, PyTorch 1.9.

A.3 Details on Dataset used

For the first two datasets, MNIST and Fashion MNIST, we use the raw features of the images themselves, and project them via PCA to a lower dimension. For the rest of the datasets (CIFAR10, CIFAR20, and STL10), we compute deep features produced by a recent unsupervised method SCAN (Van Gansbeke et al., 2020) to learn and classify images. The deep features are 512-dimensional vectors, so we apply PCA and project the data to a 20-dimensional space.

MNIST

The MNIST dataset (Deng, 2012) is a dataset of 60,000 small square 28×28 pixel grayscale (784 total pixels) images of handwritten single digits between 0 and 9. We apply PCA on the flattened images and project the test data to $D = 20$.

Fashion MNIST

The Fashion-MNIST dataset (Xiao et al., 2017) contains 60,000 training images (and 10,000 test images) of fashion and clothing items, taken from 10 classes. Each image is a standardized 28×28 size in grayscale (784 total pixels). We apply PCA on the flattened images and project the test data to $D = 20$. Results are reported on the testset.

CIFAR10

The CIFAR-10 dataset (Krizhevsky et al., 2009) consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. Instead of raw image features, we use the deep features (embeddings) produced by SCAN, which are 512-dimensional. We further reduced them 20 dimensions via PCA. Results are reported on the testset.

CIFAR20

CIFAR100 (Krizhevsky et al., 2009) is an extension of CIFAR10, only with 100 classes. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs). The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Here, we treat the superclasses as the ground truth clusters. Features are processed similarly as in the CIFAR10 case. Results are reported on the test set images.

STL10

STL-10 is an image recognition dataset (Coates et al., 2011) consists of 96x96 color images, with a corpus of 100000 unlabeled images, 50000 training images, and 8000 test images. Features are processed similarly as in the CIFAR10 case. Results are reported on the testset.

B The SubC Sampler Algorithm

Algorithm 1 below is the full algorithm of the original SubC sampler (*i.e.*, SubC-Random in our terminology), proposed by (Chang and Fisher III, 2013), (where $\overset{\propto}{\sim}$ denotes sampling proportional to the right-hand side of the equation). Note that H_{merge} from Algorithm 1 here was called H in our paper. The rest of the notation is as in our paper.

Algorithm 1 The SubC Sampler (Chang and Fisher III, 2013)

1: **procedure** RESTRICTED GIBBS SAMPLING(X)

2: Sample cluster weights $\pi_1, \pi_2, \ldots, \pi_K$,:

$$(\pi_1, \ldots, \pi_K, \tilde{\pi}_{K+1}) \sim \text{Dir}(N_1, \ldots, N_K, \alpha). \tag{10}$$

3: Sample subcluster weights $\bar{\pi}_{k,l}, \bar{\pi}_{k,r}$ for each cluster $k \in \{1, \ldots, K\}$:

$$(\bar{\pi}_{k,l}, \bar{\pi}_{k,r}) \sim \text{Dir}(N_{k,l} + \alpha/2, N_{k,r} + \alpha/2). \tag{11}$$

4: Sample cluster parameters θ_k for each cluster k:

$$\theta_k \overset{\propto}{\sim} f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_k}; \theta_k) f_\theta(\theta_k; \lambda) \tag{12}$$

5: Sample subcluster parameters $\bar{\theta}_{k,h}$ for each cluster $k \in \{1, \ldots, K\}$ and $h \in \{l, r\}$:

$$\bar{\theta}_{k,h} \overset{\propto}{\sim} f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_{k,h}}; \bar{\theta}_{k,h}) f_\theta(\bar{\theta}_{k,h}; \lambda). \tag{13}$$

6: Sample cluster assignments z_i for each point $i \in \{1, \ldots, N\}$:

$$z_i \overset{\propto}{\sim} \sum\nolimits_{k=1}^{K} \pi_k f_{\boldsymbol{x}}(\boldsymbol{x}_i; \theta_k) \mathbf{1}_{z_i = k}. \tag{14}$$

7: Sample subcluster assignments $\bar{z}_i$ for each point $i \in \{1, \ldots, N\}$:

$$\bar{z}_i \overset{\propto}{\sim} \sum\nolimits_{h \in \{l,r\}} \pi_{z_i,h} f_{\boldsymbol{x}}(\boldsymbol{x}_i; \bar{\theta}_{z_i,h}) \mathbf{1}_{\bar{z}_i = h}. \tag{15}$$

8: **procedure** PROPOSE AND ACCEPT SPLITS()(X)

9: Propose to **randomly** split cluster k into its 2 subclusters for all $k \in \{1, 2, \ldots, K\}$.

10: Calculate the Hastings ratio H_{split} and accept the split with probability $\min(1, H_{\text{split}})$ where:

$$H_{\text{split}} = \frac{\alpha \Gamma(N_{k,l}) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_{k,l}}; \lambda) \Gamma(N_{k,r}) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_{k,r}}; \lambda)}{\Gamma(N_k) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_k}; \lambda)} \tag{16}$$

11: **procedure** PROPOSE AND ACCEPT MERGES()(X)

12: Propose to merge clusters k_1, k_2 for all pairs $k_1, k_2 \in \{1, 2, \ldots, K\}$.

13: Calculate the Hastings ratio H_{merge} and accept the merge with probability $\min(1, H_{\text{merge}})$ where

$$H_{\text{merge}} = \frac{\Gamma(N_{k_1} + N_{k_2}) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_{k_1} \cup \mathcal{I}_{k_2}}; \lambda)}{\alpha \Gamma(N_{k_1}) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_{k_1}}; \lambda) \Gamma(N_{k_2}) f_{\boldsymbol{x}}(\boldsymbol{x}_{\mathcal{I}_{k_2}}; \lambda)} \tag{17}$$

C Posteriors and Marginal Data Likelihood in a Gaussian Model

Here we provide the expressions, in a Gaussian model with an NIW prior, for the posterior hyperparameters and the marginal data likelihood. For more details see (Gelman et al., 2013) or (Chang et al., 2014).

In what follows, we assume we have m samples, each of dimension D, from a multivariate normal distribution:

$$\boldsymbol{x}_i|\boldsymbol{\mu}, \boldsymbol{\Sigma} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{18}$$

where $\boldsymbol{x}$, an $m \times d$ matrix, collects all the data such that $\boldsymbol{x}_i$ is i'th row of $\boldsymbol{x}$.

C.1 Posterior Distribution of the Parameters

When the mean and covariance matrix of the sampling distribution are unknown, one can place a Normal-Inverse-Wishart prior on the mean and covariance parameters jointly: $(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \sim \mathrm{NIW}(\boldsymbol{\mu}_0, \kappa, \boldsymbol{\Psi}, \nu)$ (where $(\boldsymbol{\mu}_0, \kappa, \boldsymbol{\Psi}, \nu)$, hyperparameters associated with the NIW distribution, were collectively denoted by λ in our paper).

By conjugacy, the resulting posterior distribution for the mean and covariance matrix will also be a Normal-Inverse-Wishart distribution with closed-form updates: $(\boldsymbol{\mu}, \boldsymbol{\Sigma}|\boldsymbol{x}) \sim \mathrm{NIW}(\boldsymbol{\mu}_m, \kappa_m, \boldsymbol{\Psi}_m, \nu_m)$: The posterior NIW four parameters are updated as follows:

$$\boldsymbol{\mu}_m = \frac{\kappa\boldsymbol{\mu}_0 + m\bar{\boldsymbol{x}}}{\kappa + m} \tag{19}$$

$$\kappa_m = \kappa + m \tag{20}$$

$$\nu_m = \nu + m \tag{21}$$

$$\boldsymbol{\Psi}_m = \boldsymbol{\Psi} + S + \frac{\kappa m}{\kappa + m}(\boldsymbol{m}\bar{\boldsymbol{x}} - \mu_0)^T(\boldsymbol{m}\bar{\boldsymbol{x}} - \mu_0) \tag{22}$$

$$\text{with, } \underset{\sim}{\mathrm{S}} = \sum_{i=1}^{m}(\boldsymbol{m}\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T(\boldsymbol{m}\boldsymbol{x}_i - \bar{\boldsymbol{x}}) \tag{23}$$

C.2 The Marginal Data Likelihood

When marginalizing over the parameters of a Gaussian (*i.e.*, its mean and covariance), one obtains the marginal data likelihood (given the hyperparameters of the NIW prior):

$$f_{\boldsymbol{x}}(\boldsymbol{x};\lambda) = f_{\boldsymbol{x}}(\boldsymbol{x};\boldsymbol{\mu}_0, \kappa, \boldsymbol{\Psi}, \nu) = \frac{1}{\pi^{\frac{md}{2}}} \frac{\Gamma_D(\nu_m/2)}{\Gamma_D(\nu/2)} \frac{|\nu\boldsymbol{\Psi}|^{\nu/2}}{|\nu_m\boldsymbol{\Psi}_m|^{\nu_m/2}} \left(\frac{\kappa}{\kappa_m}\right)^{d/2} \tag{24}$$

where Γ_d is the D-dimensional Gamma function.

D Model Training Details

The models, training code and data-generation related code, are all based on the PyTorch (Paszke et al., 2019), NumPy (Harris et al., 2020), SciPy (Virtanen et al., 2020), and SciKit-Learn (Pedregosa et al., 2011) Python packages.

To build and compare the SplitNet models across a variety of scenarios, we trained three SplitNet models, for dimensions $D = 2, 10, 20$.

D.1 Nuances for successful learning

Here are a few important design choices to aid with the training of SplitNet:

Input Normalization:

Each dataset $X = [x_1, \cdots, x_n]^T$ (during training and inference) is normalized by subtracting the mean and dividing by the standard deviation: $\hat{X} = \frac{X - \text{mean}(X)}{std(X)}$. This standardizes the input space in terms of range, and helps the model to learn faster.

Curriculum Learning:

As we mentioned earlier, we can control the training dataset difficulty level. We have found that gradually increasing the dataset difficulty level during training, helps the training speed. Specifically, we choose an initial “easy” NIW prior (for data generation) for the beginning of the training, a final “difficult” NIW prior, and the update frequency T (number of epochs). The choices for these priors are listed in Table 2.

Then, we (linearly) interpolate the NIW prior between the initial and final values (in $\frac{\text{Total Epochs}}{T}$ points), generate a more difficult data subset with those interpolated values, and replacing some existing data. So that at the end of the training, the entire dataset consists of varying-difficulty clusters, from easy to hard, and the model can perform well on all of them. Figure 17 shows concrete examples of different levels of difficulty.

Ensuring “Splittable” Clusters:

It is essential to ensure that the generated dataset is “splittable”w.r.t. the Hastings Ratio - *i.e.*, its log Hastings Ratio is higher than 1. So any generated datasets with a low Hastings’ Ratio are discarded to avoid training on “noisy” samples, which can only hinder the model’s performance.

D.2 Training Hyperparameters

The training details, data-related and model’s hyper-parameters are summarized in Table 2.

Table 2: Training Settings for SplitNet

	Parameter	Value		
		$D = 2$ **Model**	$D = 10$ **Model**	$D = 20$ **Model**
General	Optimizer	Adam	Adam	Adam
	Learning Rate	0.01	0.01	0.01
	Total Epochs	200	300	400
	Batch Size	64	64	32
Model	Hidden Layer Size	128	128	256
	# of Encoders (ISAB)	2	2	3
	# of Decoders (PMA)	2	2	3
	# of Inducing Seeds	64	64	128
	# of PMA Seeds	8	16	16
	# of Heads	4	8	8
Data	Size of Dataset (train)	10000	10000	10000
	Size of Dataset (validation)	1000	1000	1000
	NIW Prior - Initial $(\boldsymbol{\mu}_0, \boldsymbol{\Psi}, \nu, \kappa)$	$\mathbf{0}_{2\times1}, \boldsymbol{I}_{2\times2}$, 10, 0.1	$\mathbf{0}_{10\times1}, \boldsymbol{I}_{10\times10}$, 20, 0.1	$\mathbf{0}_{20\times1}, \boldsymbol{I}_{20\times20}$, 25, 0.1
	NIW Prior - Final $(\boldsymbol{\mu}_0, \boldsymbol{\Psi}, \nu, \kappa)$	$\mathbf{0}_{2\times1}, \boldsymbol{I}_{2\times2}$, 4, 2	$\mathbf{0}_{10\times1}, \boldsymbol{I}_{10\times10}$, 11, 5	$\mathbf{0}_{20\times20}, \boldsymbol{I}_{20\times20}$, 21, 2.5
	NIW Prior Update Frequency, T (# of epochs)	20	30	40

E Training Data Details

For deep learning in general, the data the model is trained on is one of the most critical factors for good model performance. Therefore, it must be of high quality and provided with enough variability so that the model would be able to generalize well on unseen and challenging datasets.

Fortunately, in the case of the DPGMM, we assume the data is a GMM, and it is relatively simple to define a generative process for producing an GMM with much control over various parameters such as the Gaussians' proximity to each other, the variance of their covariance matrices, the ratio of the number of points between the Gaussians, the number of clusters, how overlapping they are, and more. The generative algorithm we used is described in Algorithm 2 below. To facilitate the creation of a rich dataset space, we used the NIW distribution for sampling the Gaussian parameters.

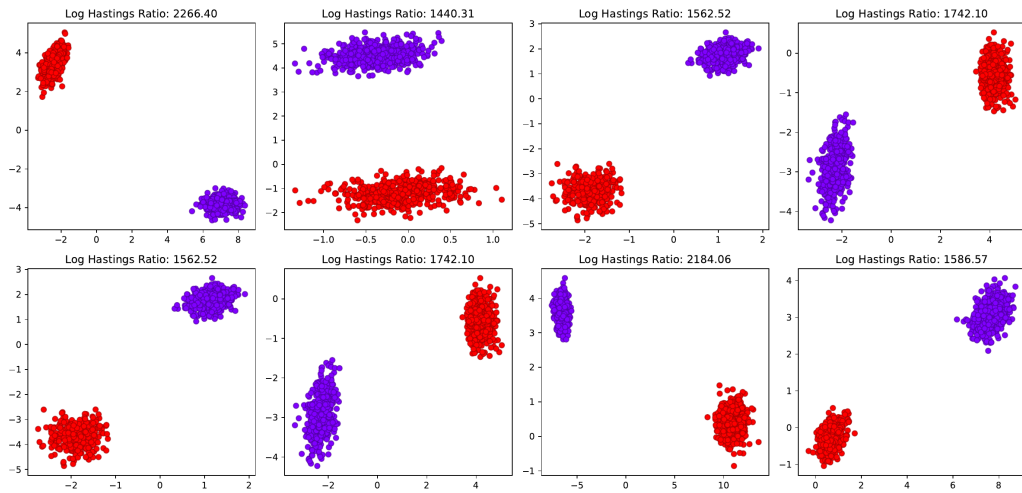
Algorithm 2 Training Data Generation Process

$p_1, p_2 \sim \mathrm{Dir}([\alpha_{\mathrm{Dir}}, \alpha_{\mathrm{Dir}}])$
$n_1, n_2 = \lceil p_1 \cdot N_{max} \rceil, \lceil p_2 \cdot N_{max} \rceil$
$\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1 \sim NIW(\nu, \kappa, \Psi, \boldsymbol{\mu}_0)$
$\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2 \sim NIW(\nu, \kappa, \Psi, \boldsymbol{\mu}_0)$
$X_l \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ - sample n_1 points.
$X_r \sim \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$ - sample n_2 points.
$X \longleftarrow [X_l, X_r]$
$Z \longleftarrow [\mathbf{0}_{n_1}, \mathbf{1}_{n_2}]$

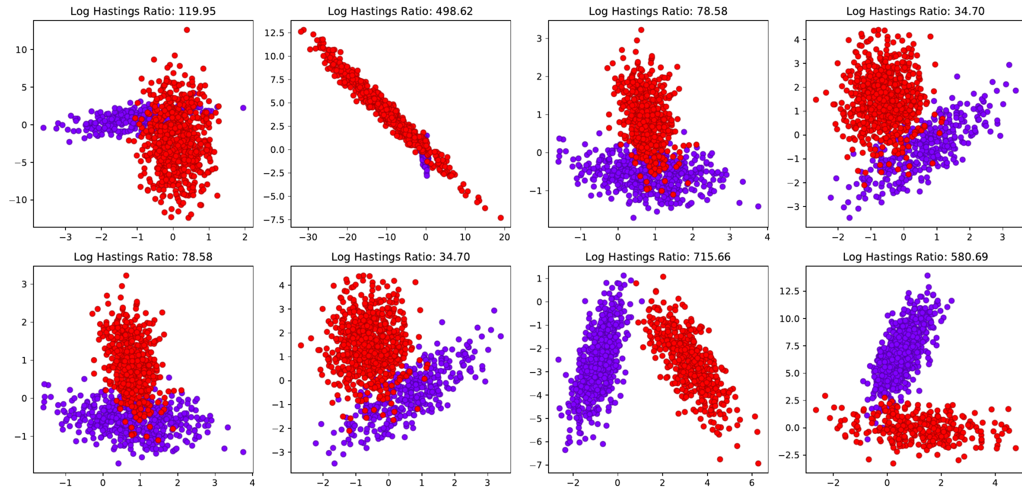
This is also where the model can learn "difficult" splits, where other methods (e.g., K-means splits) are most likely to fail. We generate a dataset with specific characteristics: clusters imbalance, non-spherical covariances, and overlapping clusters. The "knobs" with which we tune this are the following:

1. κ (one of the NIW hyperparameters) - the larger κ is, the more separable the clusters are.
2. ν (another NIW hyperparameter) - the smaller ν is, the more isotropic the cluster are.
3. α_{Dir} (a hyperparameter of the two-dimensional Dirichlet distribution) - the smaller α is, the more imbalanced the points allocation between the subclusters.

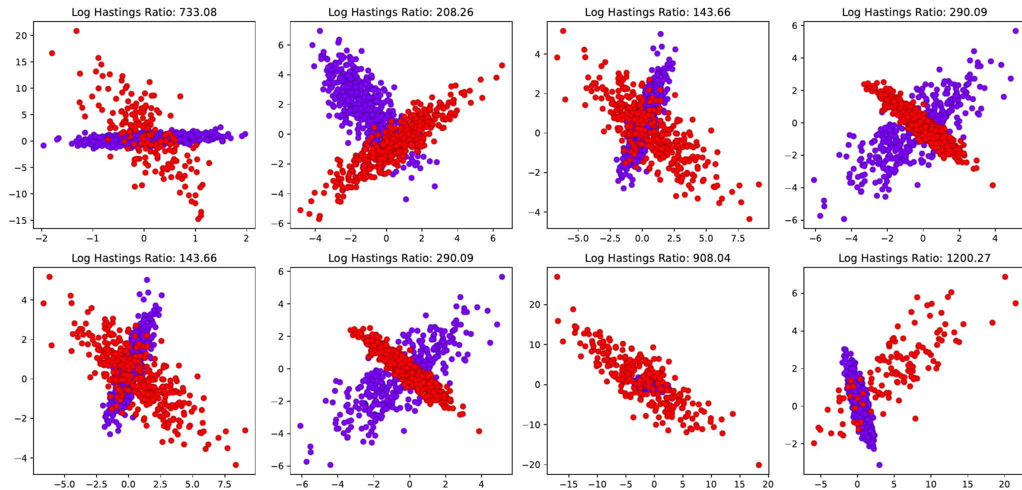
Examples of samples of varying difficulty datasets are depicted in Figure 17.



(a) Easy Data Samples



(b) Medium Data Samples



(c) Hard Data Samples

Figure 17: Examples of Generated Training Data of Various difficulty levels

F The Set Transformer

The Set Transformer (ST) (Lee et al., 2019) is a permutation-invariant set-input neural network that uses self-attention operations as building blocks. It utilizes multi-head attention (Vaswani et al., 2017) for both encoding elements of a set and decoding encoded features into outputs.

The attention mechanism allows processing every element in an input set, which enables the ST to naturally encode pairwise or higher-order interactions between elements in the set.

Assume we have n query vectors (corresponding to a set with n elements) each with dimension $d_q : Q \in \mathbb{R}^{n \times d_q}$. An attention function $\text{Att}(Q, K, V)$ is a function that maps queries Q to outputs using n_v key-value pairs $K \in \mathbb{R}^{n_v \times d_q}, V \in \mathbb{R}^{n_v \times d_v}$.

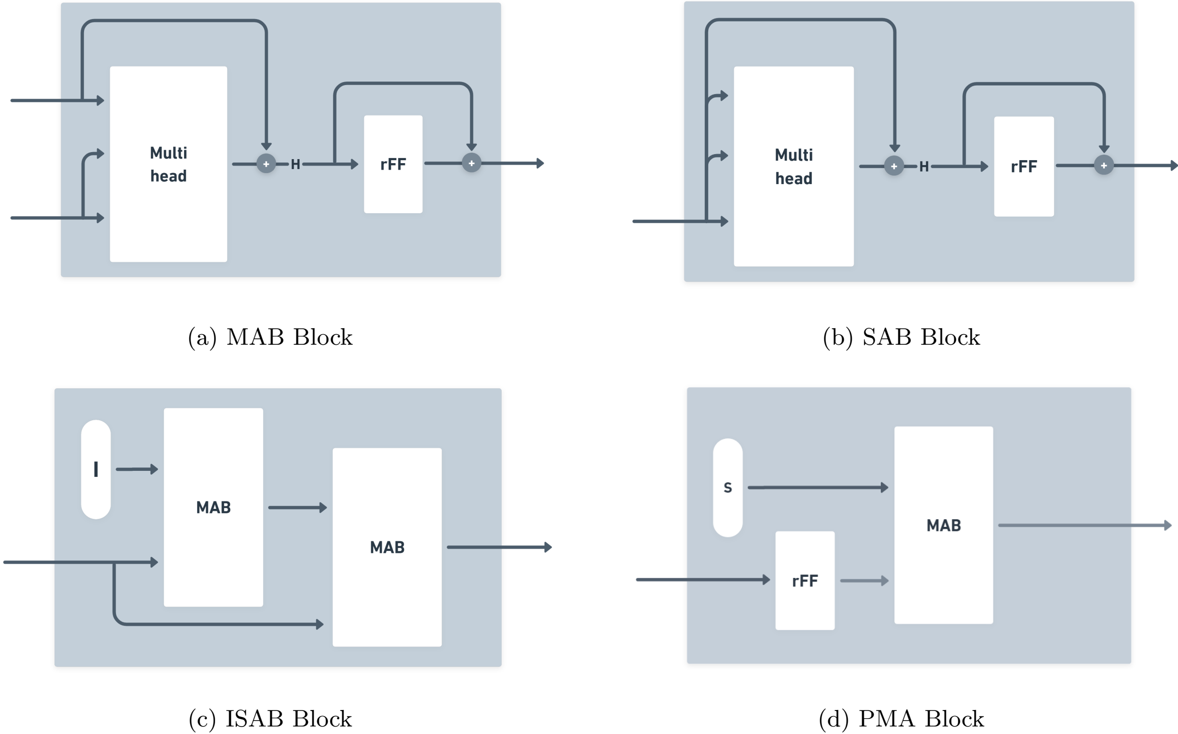
$$\text{Att}(Q, K, V; \omega) = \omega\left(QK^\top\right) V \tag{25}$$

The pairwise dot product $QK^T \in \mathbb{R}^{n \times n_v}$ measures how similar each pair of query and key vectors is, with weights computed with an activation function ω. The output $\omega(QK^T)V$ is a weighted sum of V where a value gets more weight if its corresponding key has larger dot product with the query.

Multihead Attention is an extension of this mechanism, originally introduced by (Vaswani et al., 2017), in which Q, K, V are first projected into h different d_q^M, d_k^M, d_v^M-dimensional vectors, respectively. Then the attention function is applied individually on these projections. The output is computed by concatenating the attention outputs and applying linear transformation on it:

$$\text{Multihead} - \text{Att}(Q, K, V; \lambda, \omega) = \text{concat}\left(O_1, \cdots, O_h\right) W^O \tag{26}$$

$$\text{where } O_j = \text{Att}\left(QW_j^Q, KW_j^K, VW_j^V; \omega_j\right) \tag{27}$$



(a) MAB Block

(b) SAB Block

(c) ISAB Block

(d) PMA Block

Figure 18: Basic Attention-based Blocks of the Set Transformer

The fundamental building block of a ST is the Multihead Attention Block (MAB), which takes two sets $X = [x_1, \ldots, x_n]$ and $Y = [y_1, ..., y_m]^T$ and outputs a set of the same size as X. An MAB is defined as

$$\text{MAB}(X, Y) = H + \text{rFF}(H) \tag{28}$$

where $H = X + \text{rFF}(\text{ Multihead-Att}(X, Y))$ and $\text{rFF}(\cdot)$ is a feed-forward layer applied row-wise (*i.e.*, for each element). $\text{MAB}(X, Y)$ computes the pairwise interactions between the elements in X and Y with sparse weights obtained from attention.

A Self-Attention Block (SAB) is simply MAB applied to the set itself:

$$\text{SAB}(X) = \text{MAB}(X, X). \tag{29}$$

We can model high-order interactions among the items in a set by stacking multiple SABs. Note that the time-complexity of SAB is $O(n^2)$ because of pairwise computation. To reduce this, the authors (Lee et al., 2019) proposed to use Induced Self-Attention Block (ISAB) defined as:

$$\text{ISAB}(X) = \text{MAB}(X, \text{MAB}(I, X)) \tag{30}$$

where $I = [i_1, \ldots, i_m]^T$ are trainable inducing points. ISAB indirectly compares the elements of X through the inducing points, reducing the time-complexity to $O(nm)$.

To summarize a set into a fixed-length representation, ST uses an operation called Pooling by Multihead Attention (PMA). A PMA is defined as

$$\text{PMA}_k(X) = \text{MAB}(S, X) \tag{31}$$

where $S = [s_1, \ldots, s_k]^T$ are trainable parameters.

A visual representation of these basic blocks is available in Figure 18.