Supplemental Material

A joint deep learning model enables simultaneous batch effect correction, denoising and clustering in single-cell transcriptomics

Justin Lakkis*, David Wang, Yuanchao Zhang, Gang Hu, Kui Wang, Huize Pan, Lyle Ungar, Muredach P. Reilly, Xiangjie Li*, Mingyao Li*

***Correspondence:**

Justin Lakkis (jlakkis@pennmedicine.upenn.edu)

Xiangjie Li (xiangjie631@outlook.com)

Mingyao Li, Ph.D. (mingyao@pennmedicine.upenn.edu)

Supplemental Table 1: CarDEC hyperparameters.

Hyperparameter	Default	Brief Description
n_high_var	2000	The number of genes to be retained as HVGs. The *n_high_var* genes with the highest variance as retained as HVGs, while all other genes are marked as LVGs.
n_clusters	No default	The number of cell type clusters specified by the user.
dims	[128, 32]	List of layer sizes for the HVG encoder, excluding the input layer which is inferred from the data. Decoder layers inferred automatically such that the decoder is symmetric to the encoder.
LVG_dims	[128, 32]	List of layer sizes for the LVG encoder, excluding the input layer which is inferred from the data. Decoder layers inferred automatically such that the decoder is symmetric to the encoder.
Clustering Weight	1 step 3	The weight α for balancing clustering loss with reconstruction loss for step 3 of CarDEC.
tol	0.005 step 3	Helps to monitor convergence of CarDEC. See online methods or *iteration_patience_ES* parameter for more details.
maxiter	1000 step 3	The maximum number of iterations for training the main CarDEC model, described in step 3 of the CarDEC workflow.
epochs	2000 step 2, 4	The maximum number of minibatch gradient descent epochs for training the neural networks for steps 2, 4.
patience_LR	3 step 2, 3, 4	If validation reconstruction loss does not improve after this many epochs/iterations, then decay the learning rate.
decay_factor	1/3	If validation reconstruction loss does not improve after

	step 2, 3, 4	*patience_LR* epochs or iterations, then decay the learning rate by this factor.
patience_ES	9 step 2, 4	If validation reconstruction loss does not improve after this many epochs when training a neural network from step 2 or 4, stop training.
iteration_patience_ES	6 step 3	If validation reconstruction loss does not decrease after this many iterations, and if the proportion of cells whose cluster assignment changed on the most recent iteration was less than *tol*, than stop training.
batch_size	64 step 2, 3, 4	Batch size for minibatch gradient descent
learning rate	0.0001 step 2, 3, 4	Learning rate for minibatch gradient descent

Supplemental Table 2. Datasets analyzed in this paper.

Species	Tissue	Data Source	Batches	Cell Types (number of Cells)*	Dataset Dimensions*	Protocol

Macaque	Retina
	
	(Bipolar cells)
		Peng *et al.* (2019)(Peng et al. 2019)
		
		(GSE11848)
			2 regions;
			
			4 animals;
			
			30 samples
				BB/GB* (1814)
				DB1 (995)
				DB2 (2243)
				DB3a (623)
				DB3b (2640)
				DB4 (2985)
				DB5* (3467)
				DB6 (658)
				FMB (4500)
				IMB (6150)
				OFFx (147)
				RB (4076)
					30298 cells
					
					18083 genes
						Drop-seq

Mouse	Cortex	Ding *et al.* (2020)(Ding et al. 2020) (SCP425)	4 batches (4 scRNA-seq protocols)	Astrocyte (1384) Endothelial (399) Excitatory neuron (7776) Inhibitory neuron (2471) Microglia (311) OPC (276) Oligodendrocyte (777) Pericyte (29)	13423 cells 20095 genes	10x Chromium DroNc-seq Smart-seq2 sci-RNA-seq
Human	PBMC	Ding *et al.* (2020)(Ding et al. 2020) (SCP424)	8 batches (5 scRNA-seq protocols)	B cell (4773) CD14+ monocyte (4896) CD16+ monocyte (777) CD4+ T cell (7188) Cytotoxic T cell (8504) Dendritic cell (411) Megakaryocyte (202) Natural killer cell (1515) Plasmacytoid dendritic cell (160)	28426 cells 17295 genes	10x Chromium v2 (three replicates) 10x Chromimum v3 CEL-Seq2 Drop-seq Seq-Well inDrops
Human	Pancreas	Grün D *et al.* (2016)(Grun et al. 2016) (GSE81076)	4 batches (4 scRNA-seq protocols)	acinar (711) activated_stellate (180) alpha (2281) beta (1172) delta (405)	6321 cells 21215 genes	CEL-Seq CEL-Seq2 Fluidigm C1

		Muraro *et al*. (2016)(Muraro et al. 2016) (GSE85241) Lawlor *et al*. (2017)(Lawlor et al. 2017) (GSE86469) Segerstolpe *et al*. (2015)(Segerstolpe et al. 2016) (E-MTAB-5061)		ductal (1065) endothelial (61) epsilon (14) gamma (359) macrophage (24) mast (17) quiescent_stellate (20) schwann (12)		SMART-Seq2
Human	Liver	Popescu *et al*. (2019)(Popescu et al. 2019) (E-MTAB-7407)	13 batches	B cell (920) DC precursor (293) DC1 (328) DC2 (3661) Early Erythroid (10652) Early lymphoid_T lymphocyte (714) Endothelial cell (2985) Fibroblast (1640) HSC_MPP (2996) Hepatocyte (2150) ILC precursor (1659) Kupffer Cell (23866) Late Erythroid (2821) MEMP (1219) Mast cell (952) Megakaryocyte (3621) Mid Erythroid (25706) Mono-Mac (6312) Monocyte (2389) Monocyte precursor (277) NK (6431)	104694 cells 21521 genes	10x Chromium

				Neutrophil-myeloid progenitor (542) Pre pro B cell (176) VCAM1+ EI macrophage (93) pDC precursor (236) pre-B cell (995) pro-B cell (1060)		
Human	Monocytes	Li *et al*. (2020)(Li et al. 2020) (GSE146974)	1 subject; 3 batches	-	10878 cells 11160 genes	10x Chromium

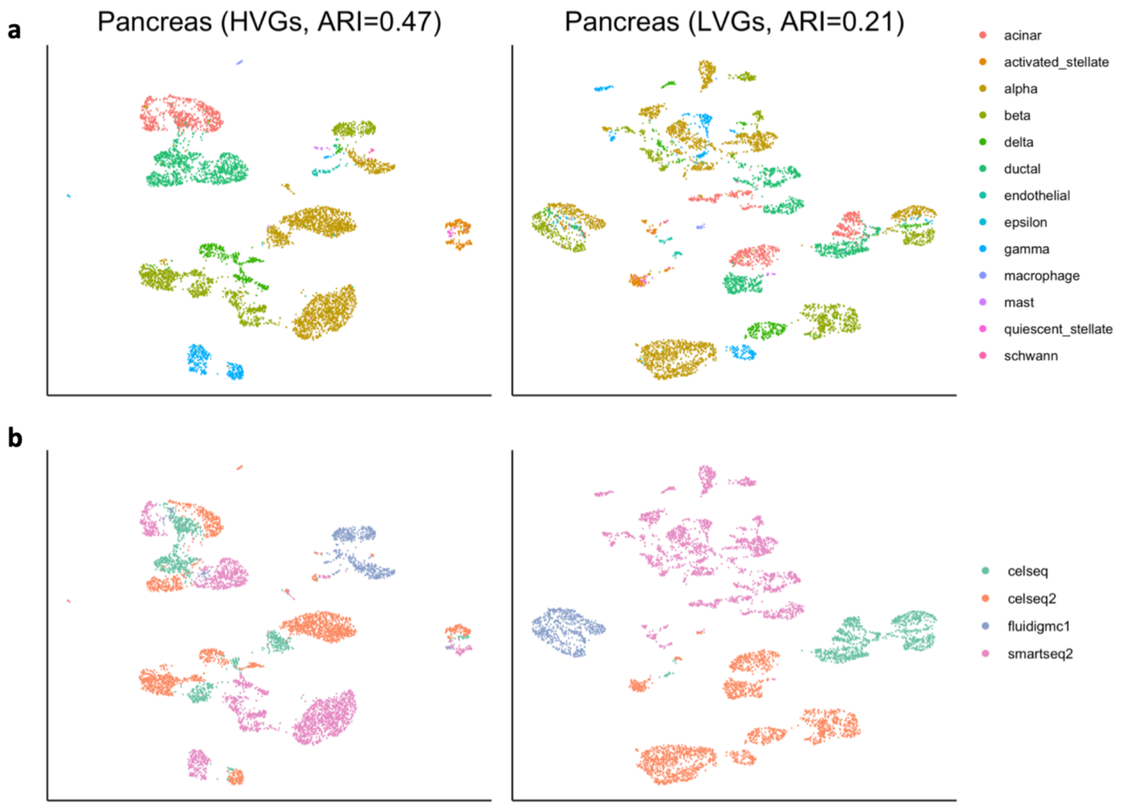
*These numbers are reported after quality control filtering of the datasets

Supplemental Table 3. Software compared with CarDEC.

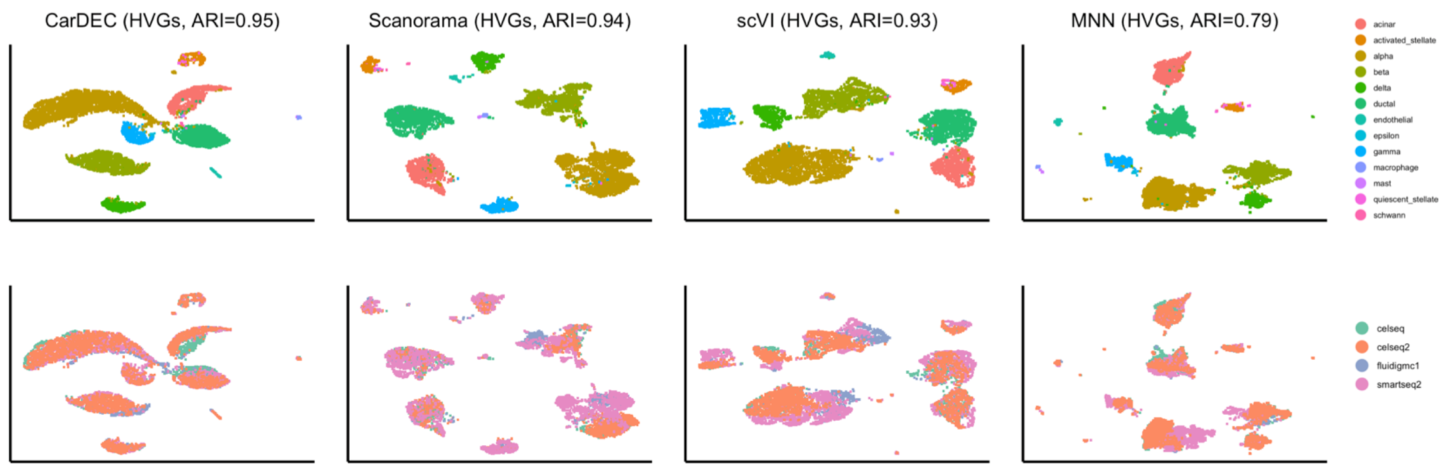
Method	Version	URL	Reference
DCA	0.2.3	https://pypi.org/project/DCA/0.2.3	Eraslan *et al*. (2019)
Combat	1.5.1	https://pypi.org/project/scanpy/1.5.1/	Johnson *et al*. (2007)
Scanorama	1.7	https://pypi.org/project/scanorama/1.7/	Hie *et al*. (2019)(Hie et al. 2019)
scvi-tools	0.7.0b0	https://pypi.org/project/scvi-tools/0.7.0b0/	Lopez *et al*. (2018)
MNN	1.2.4	https://bioconductor.org/packages/release/bioc/html/batchelor.html	Haghverdi *et al*. (2018)
scDeepCluster	Github commit: 4186eaf85aec3a04522eff04149223e7bcba90d3	https://github.com/ttgump/scDeepCluster/tree/4186eaf85aec3a04522eff04149223e7bcba90d3	Tian *et al*. (2019)

Comments: We did not install a dedicated package for Combat. Rather, we used an implementation in Scanpy: see documentation https://scanpy.readthedocs.io/en/stable/api/scanpy.pp.combat.html.

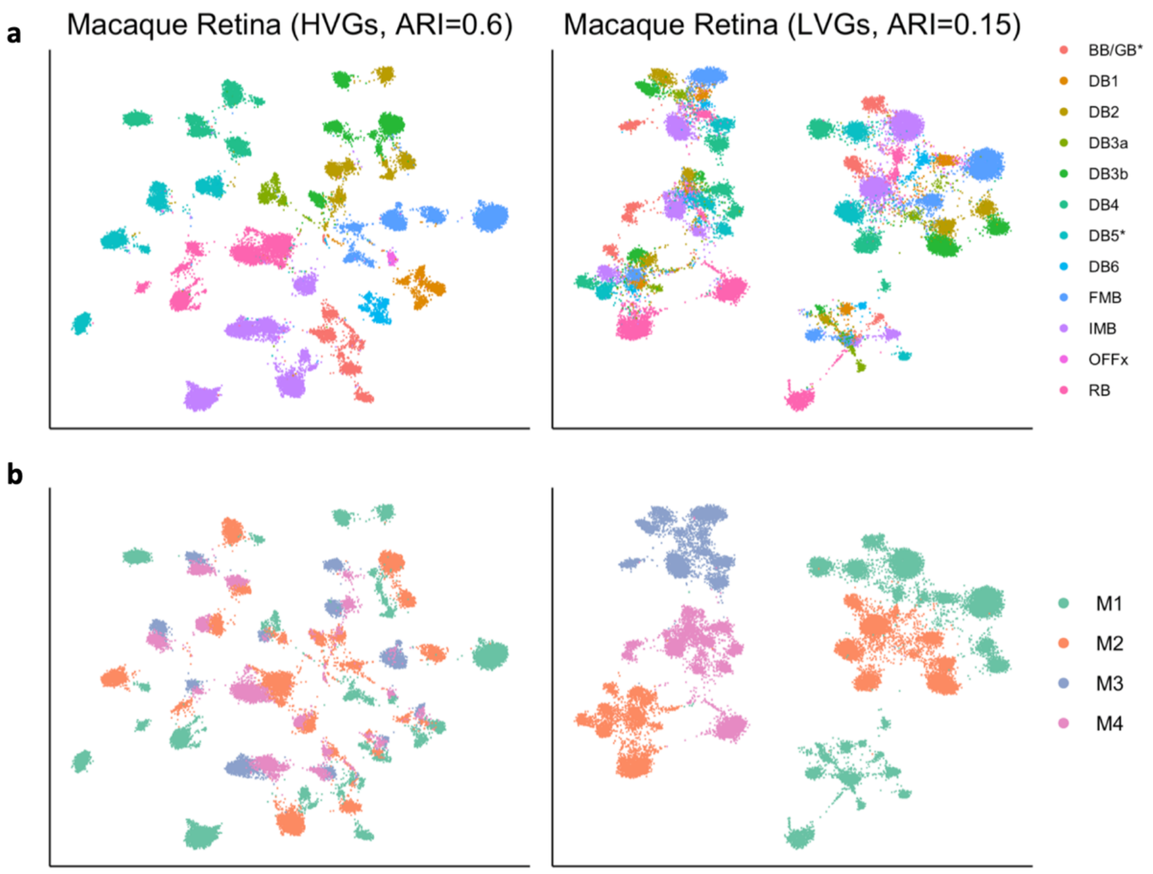
All packages can be installed into an Anaconda environment easily using this environment yml file (excluding R MNN package that must be handled separately), https://github.com/jlakkis/CarDEC_Codes/blob/main/cardec_alternatives.yml.



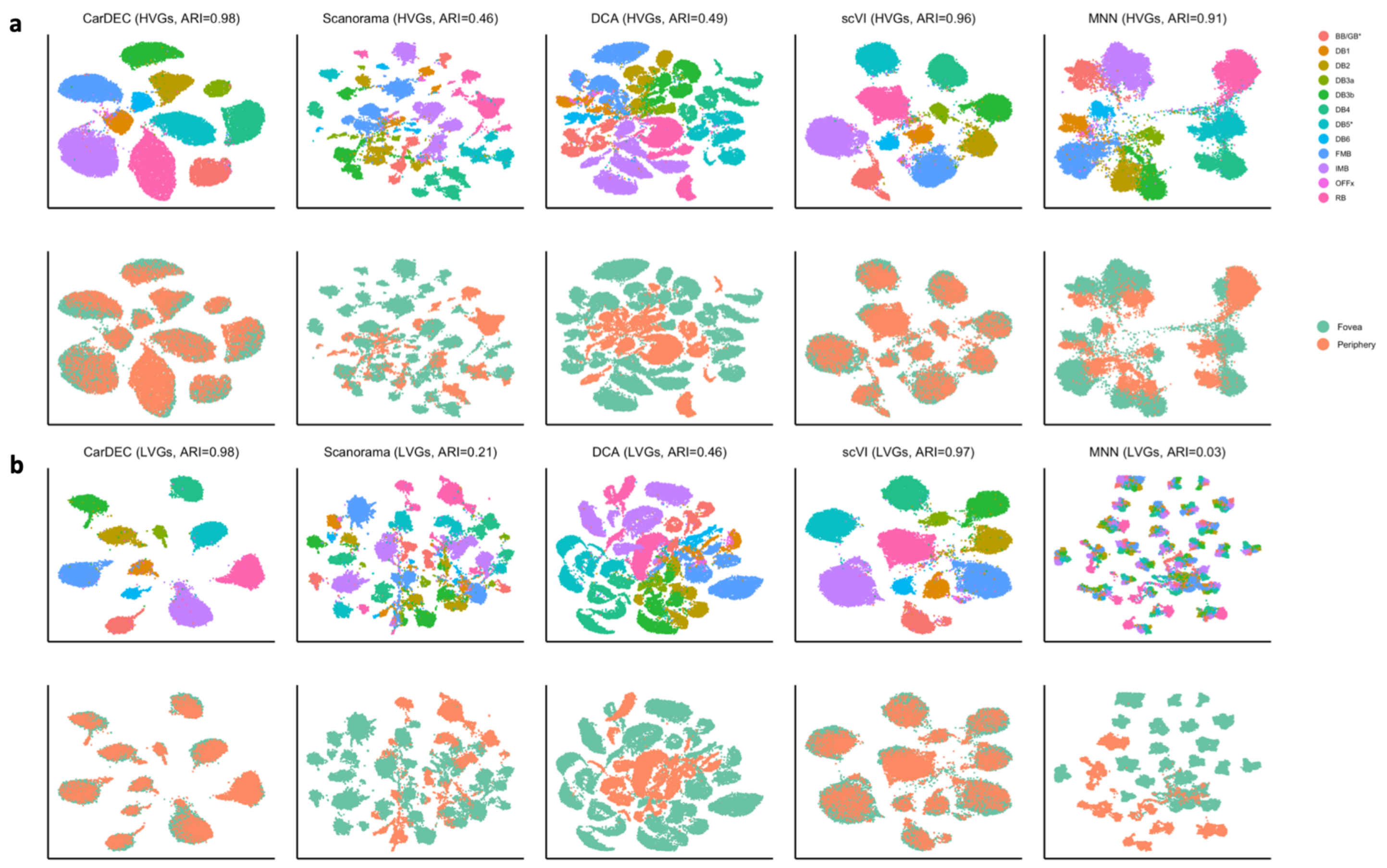
Supplemental Fig. S1. Raw counts analysis on the human pancreas data. We analyze the raw HVG and LVG counts for the pancreas data using Scanpy's Louvain workflow to provide a baseline for our analyses. The UMAPs obtained by analyzing the raw counts with the Scanpy Louvain workflow are provided here, along with their ARIs. (a) Here, we visualize the UMAP embeddings computed from the raw counts of HVGs. (b) Here, we visualize the UMAP embeddings computed from the raw counts of LVGs.



Supplemental Fig. S2. Methods comparison for batch correcting HVGs only on the pancreas data. Here, we fit batch correction methods on the HVGs only of the pancreas data, rather than using HVGs and LVGs together. We used the batch corrected HVGs as input for UMAP projection, and colored UMAPs both by cell type (top row) and batch annotation (bottom row).

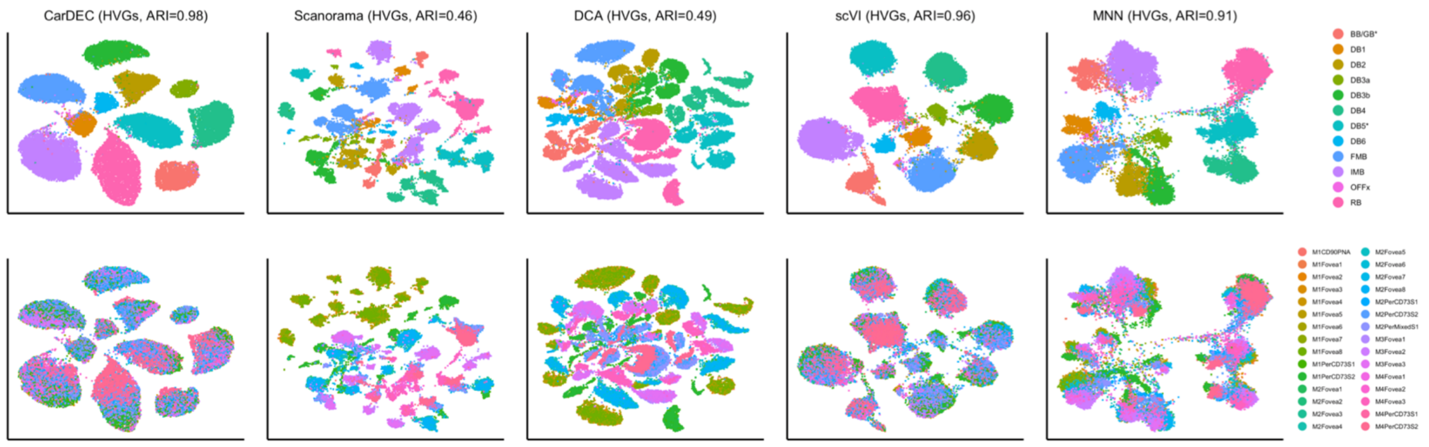


Supplemental Fig. S3. Raw counts analysis on the macaque retina data. We analyzed the raw HVG and LVG counts for the Macaque Retina dataset generated by Peng *et al*. (Peng et al. 2019) using Scanpy's Louvain workflow to provide a baseline for our analyses. The UMAPs obtained by analyzing the raw counts with the Scanpy Louvain workflow are provided here, along with their ARIs. (a) UMAP embeddings computed from the raw counts of HVGs (left) and LVGs (right) where the cells were colored by cell type. (b) UMAP embeddings computed from the raw counts of HVGs (left) and LVGs (right) where the cells were colored by macaque ID.

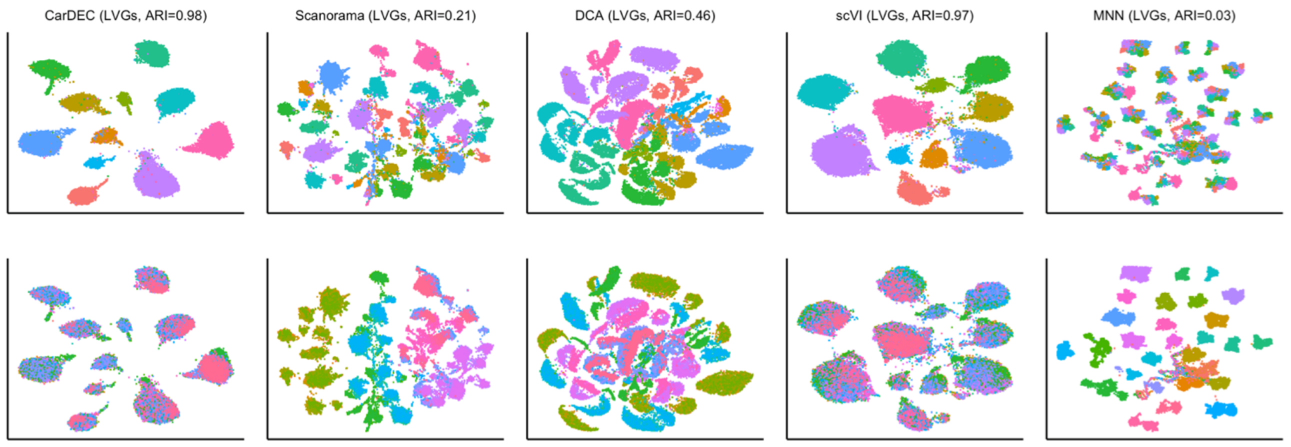


Supplemental Fig. S4. Methods comparison on the macaque retina data using denoised/batch corrected counts with region annotation. We analyzed the Macaque Retina dataset. Here, we present essentially the same UMAPs as those shown in Figure 4 from the main paper, but now the batch annotation used to color the UMAPs is “Region” rather than Macaque ID. (a) UMAP embedding computed from the denoised HVG counts for each method. Top row colored by cell type; bottom colored by batch. Cells are also clustered with Louvain’s algorithm, and resultant ARI is provided. (b) UMAP embedding computed from the denoised LVG counts for each method. Figure legends are the same as those in (a).

a

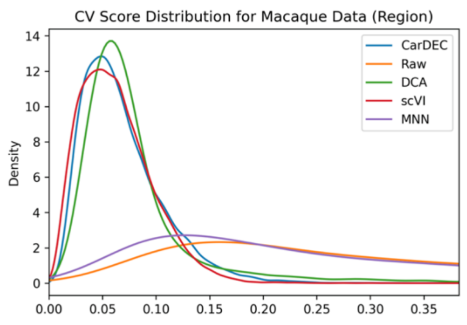


b

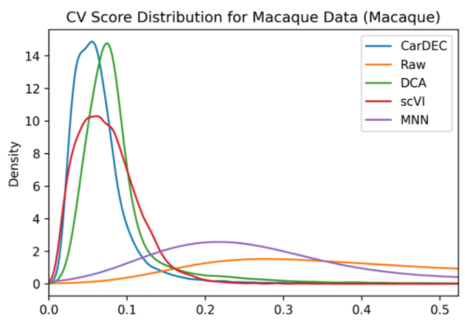


Supplemental Fig. S5. Methods comparison on the macaque retina data using denoised/batch corrected counts with sample annotation. Here, we present essentially the same UMAPs as those shown in Figure 4 from the main paper, but now the batch annotation used to color the UMAPs is "Sample" rather than Macaque ID. (a) UMAP embedding computed from the denoised HVG counts for each method. Top row colored by cell type; bottom colored by batch. Cells are also clustered with Louvain's algorithm, and resultant ARI is provided. (b) UMAP embedding computed from the denoised LVG counts for each method. Figure legends are the same as those in (a).

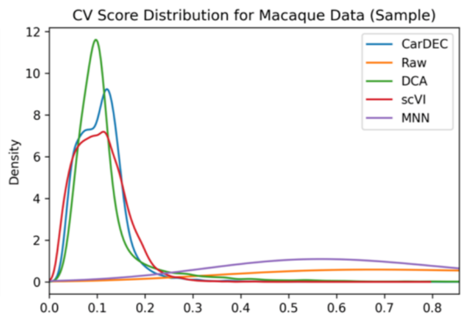
a



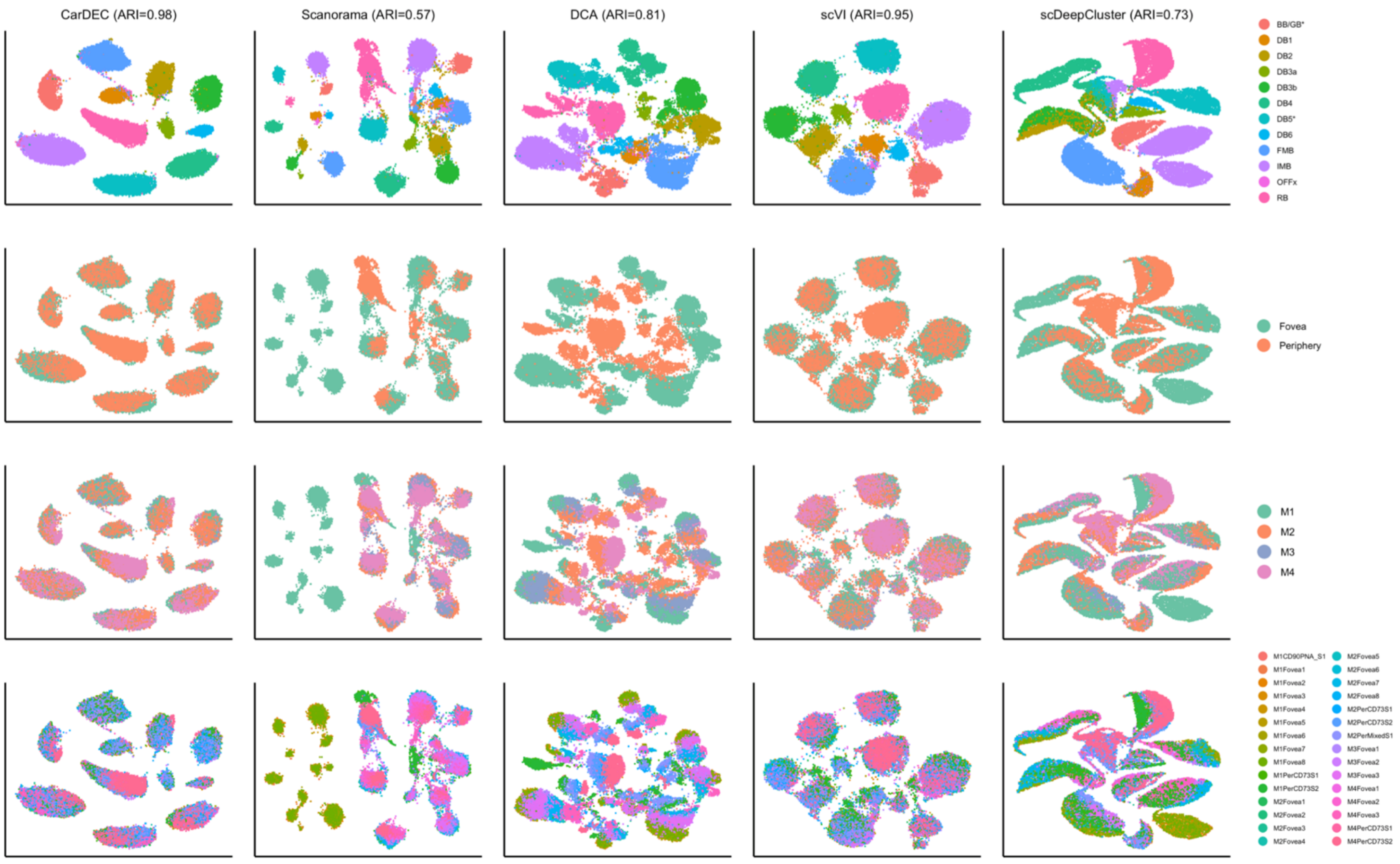
b



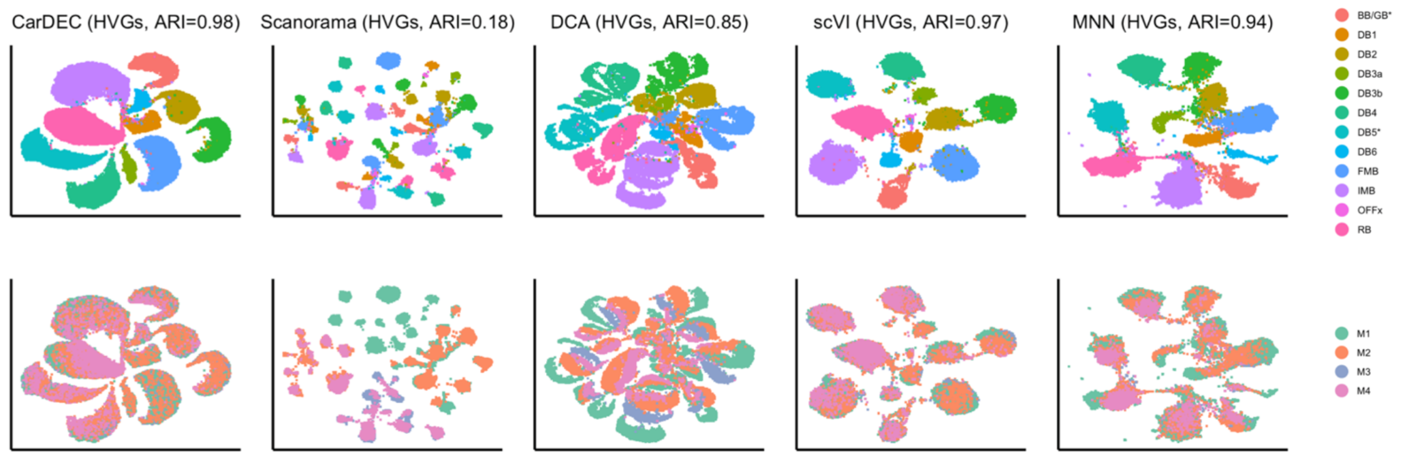
c



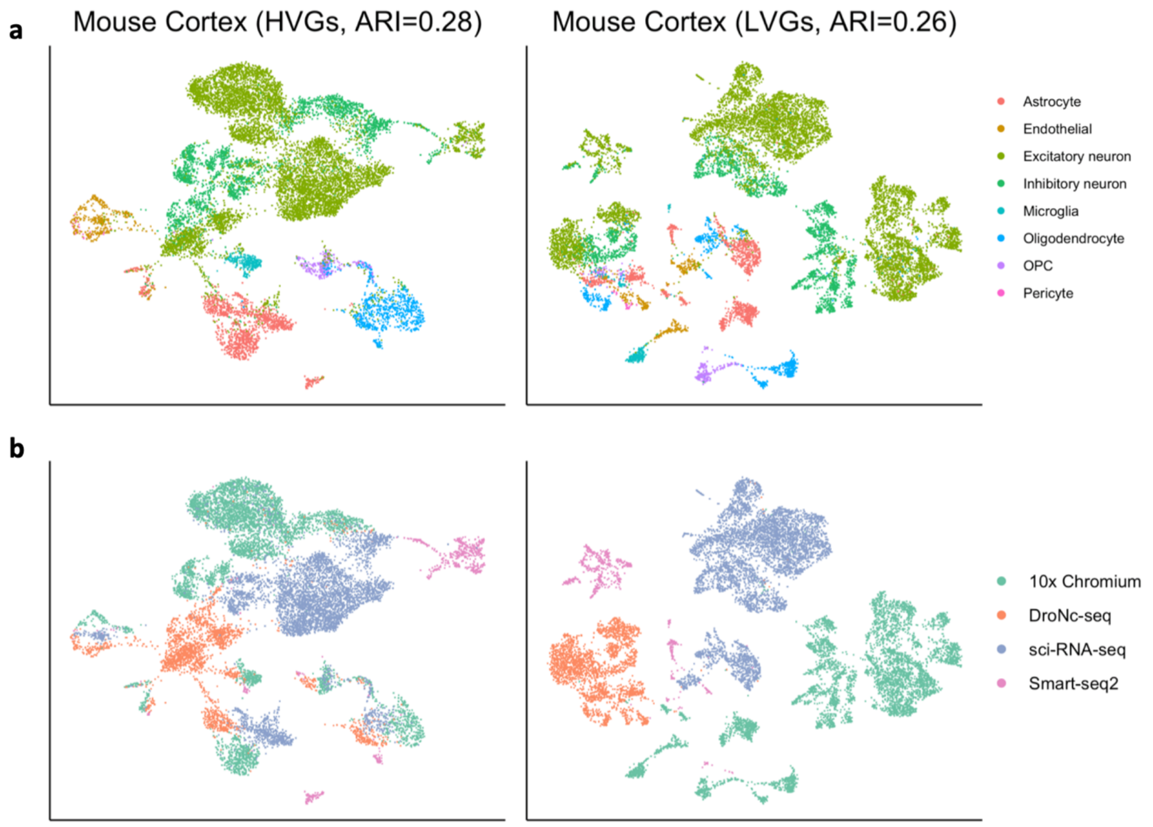
Supplemental Fig. S6. CV plots for all three macaque retina data batch annotations. Here, we show all three possible CV plots (computed using all genes) for the Macaque Retina dataset, one for each batch annotation definition. Recall that for a CV plot, we plot the density plot of genewise coefficient of variation (CV) among batch centroids for each method. The set of batch centroids is defined differently for each batch definition, so we show the CV Plots for all batch definitions here. (a) CV plot using Region as the definition for batch. (b) CV Plot using Macaque as the definition for batch. (c) CV plot using Sample as the definition for batch.



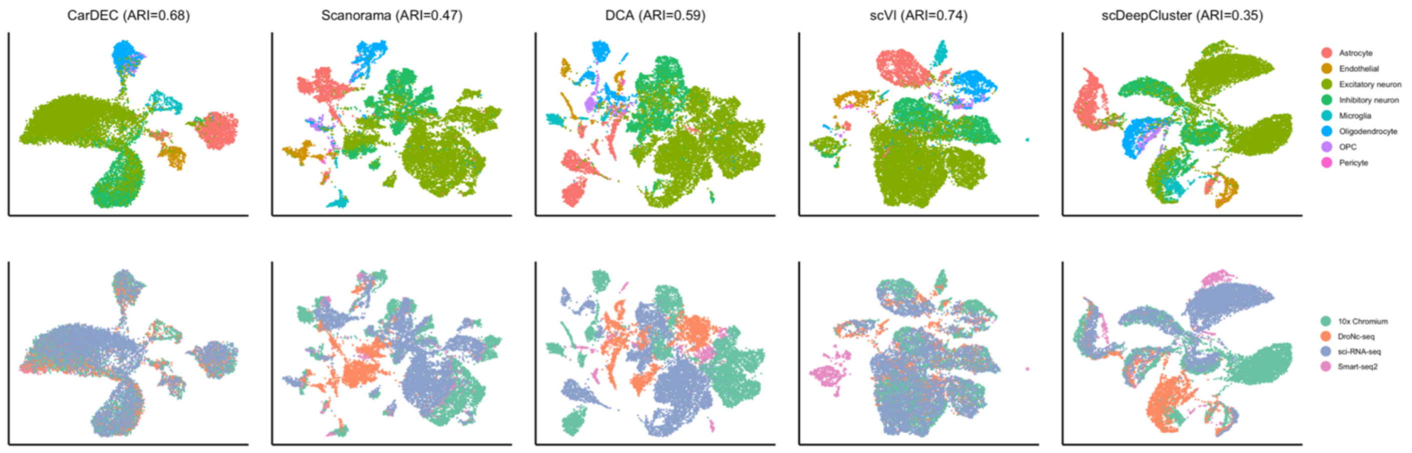
Supplemental Fig. S7. Methods comparison for clustering in the embedding space on the macaque retina data. Here, we used embedding space driven methods to cluster the Macaque Retina data, rather than using a full gene-space representation. We used the embedding as input for UMAP projection, and colored UMAPs both by cell type (top row), region batch annotation (second row), macaque batch annotation (third row), and sample batch annotation (last row).



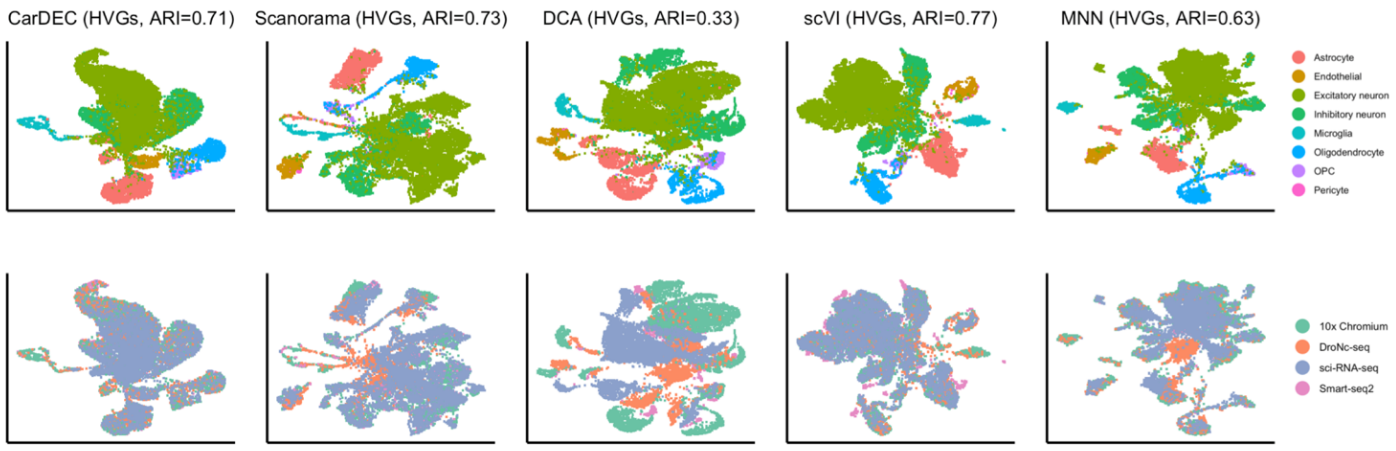
Supplemental Fig. S8. Methods comparison for batch correcting HVGs only on the macaque retina data. Here, we fit batch correction methods on the HVGs only of the Macaque Retina data, rather than using HVGs and LVGs together. We used the batch corrected HVGs as input for UMAP projection, and colored UMAPs both by cell type (top row) and macaque batch annotation (bottom row).



Supplemental Fig. S9. Raw counts analysis on the mouse cortex data. We analyzed the raw HVG and LVG counts for the Mouse Cortex dataset generated by Ding *et al*. (Ding et al. 2020) using Scanpy's Louvain workflow to provide a baseline for our analyses. The UMAPs obtained by analyzing the raw counts with the Scanpy Louvain workflow are provided here, along with their ARIs. (a) UMAP embeddings computed from the raw counts of HVGs. (b) UMAP embeddings computed from the raw counts of LVGs.

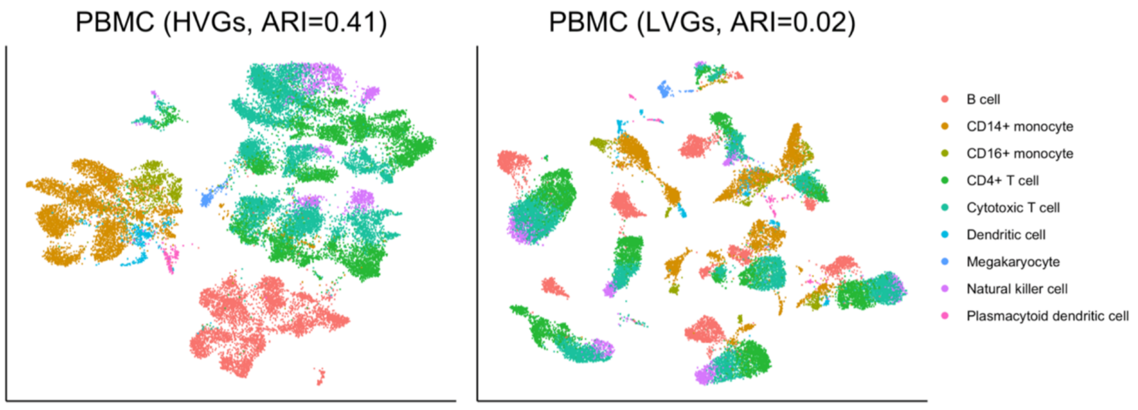


Supplemental Fig. S10. Methods comparison for clustering in the embedding space on the mouse cortex data. Here, we used embedding space driven methods to cluster the Mouse Cortex dataset, rather than using a full gene-space representation. We used the embedding as input for UMAP projection, and colored UMAPs both by cell type (top row) and batch annotation (bottom row).

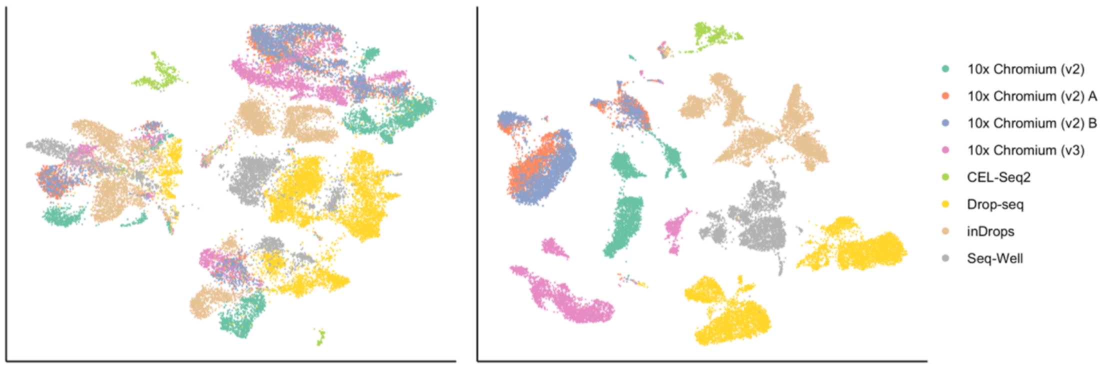


Supplemental Fig. S11. Methods comparison for batch correcting HVGs only on the mouse cortex data. Here, we fit batch correction methods on the HVGs only of the mouse cortex data, rather than using HVGs and LVGs together. We used the batch corrected HVGs as input for UMAP projection, and colored UMAPs both by cell type (top row) and batch annotation (bottom row).

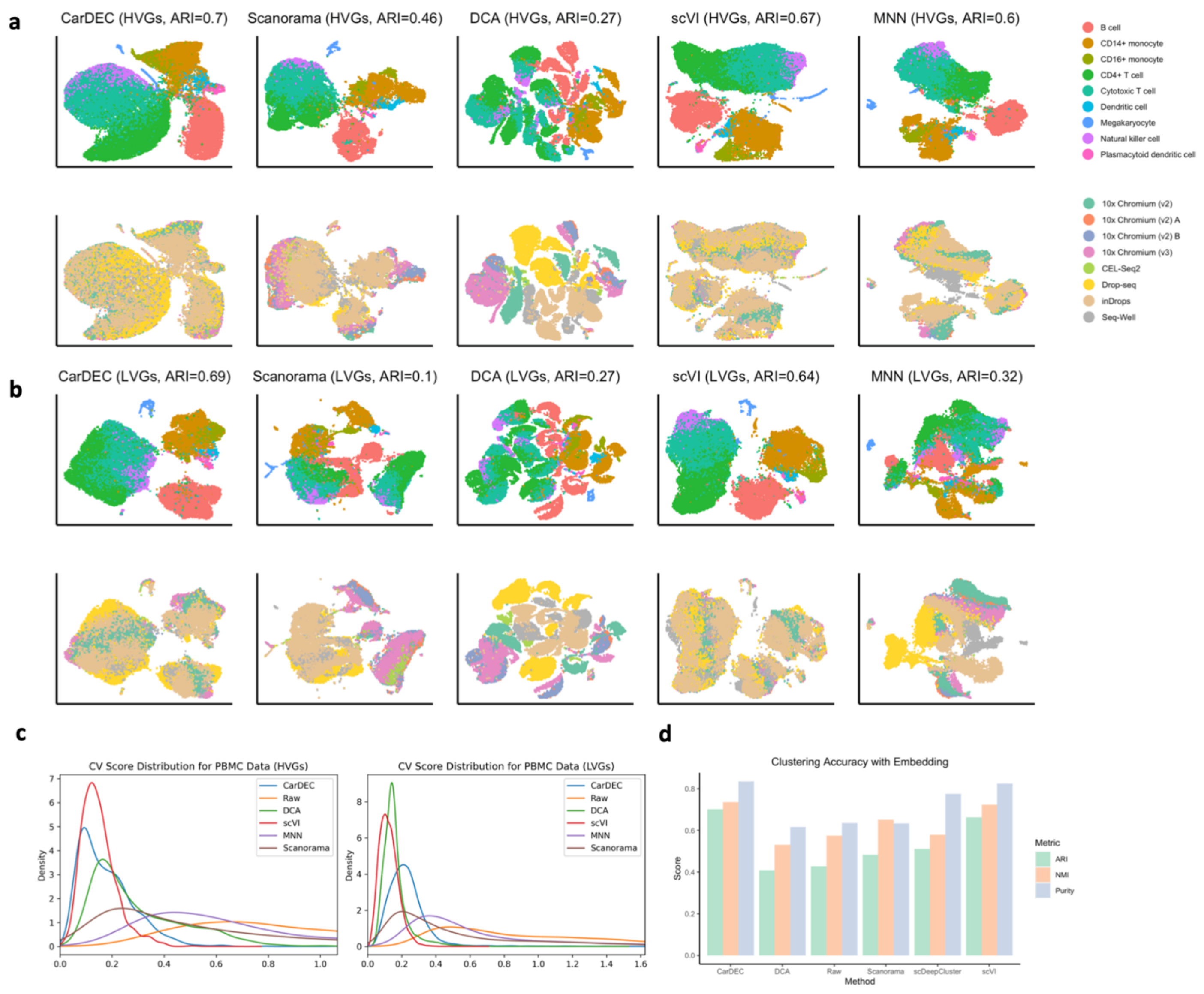
a



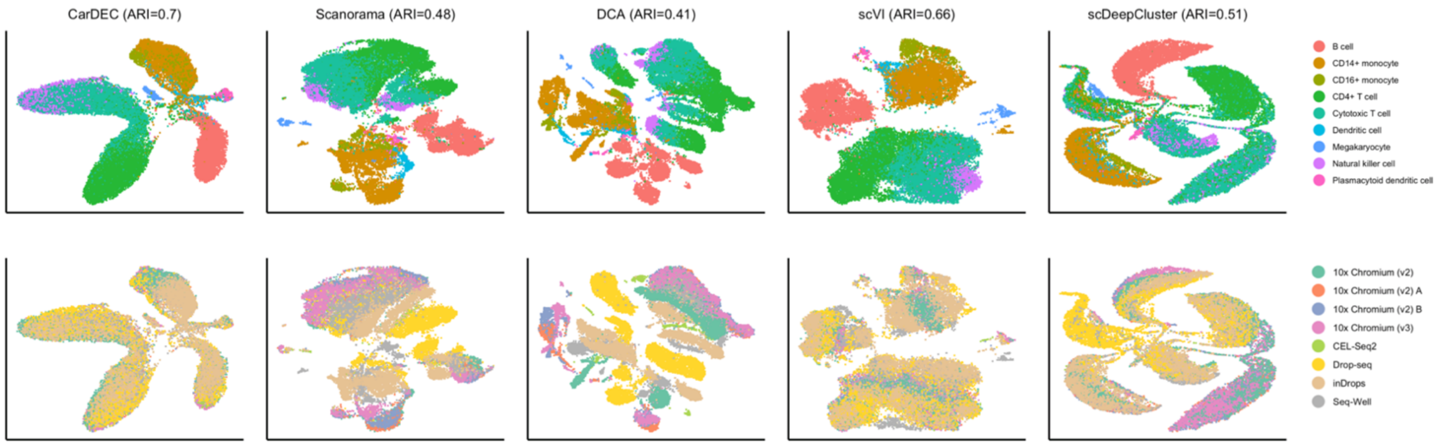
b



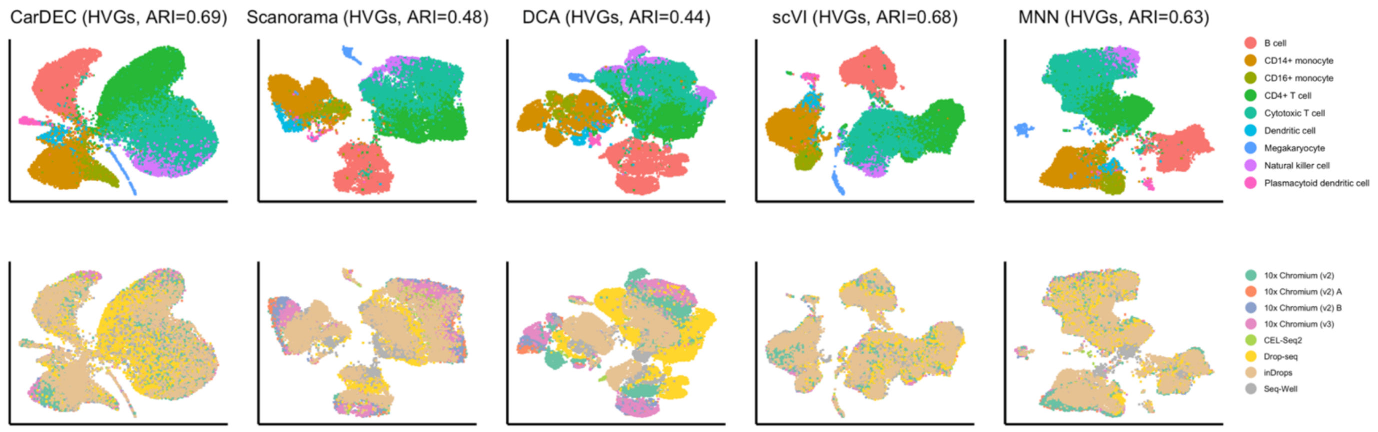
Supplemental Fig. S12. Raw counts analysis on the human PBMC data. We analyzed the raw HVG and LVG counts for the PBMC dataset generated by Ding *et al*. (Ding et al. 2020) using Scanpy's Louvain workflow to provide a baseline for our analyses. The UMAPs obtained by analyzing the raw counts with the Scanpy Louvain workflow are provided here, along with their ARIs. (a) UMAP embeddings computed from the raw counts of HVGs. (b) UMAP embeddings computed from the raw counts of LVGs.



Supplemental Fig. S13. Methods comparison on the human PBMC data using denoised and batch corrected counts. We benchmarked the performance of CarDEC and several competing methods on the human PBMC dataset. (a) UMAP embedding computed from the denoised HVG counts for each method. Top row was colored by cell type; bottom was colored by batch. Cells were clustered with Louvain's algorithm. (b) UMAP embedding computed from the denoised LVG counts for each method. Figure legends are the same as those in (a). (c) Density plot of genewise coefficient of variation (CV) among batch centroids. Density plots are provided for HVGs and LVGs separately. (d) Clustering accuracy metrics obtained using the embedding based methods to cluster the data, rather than running Louvain on the full gene expression space. Results for "Raw" were run using Louvain's algorithm on the original HVG counts, to provide a baseline with which to compare embedding based clustering results to.

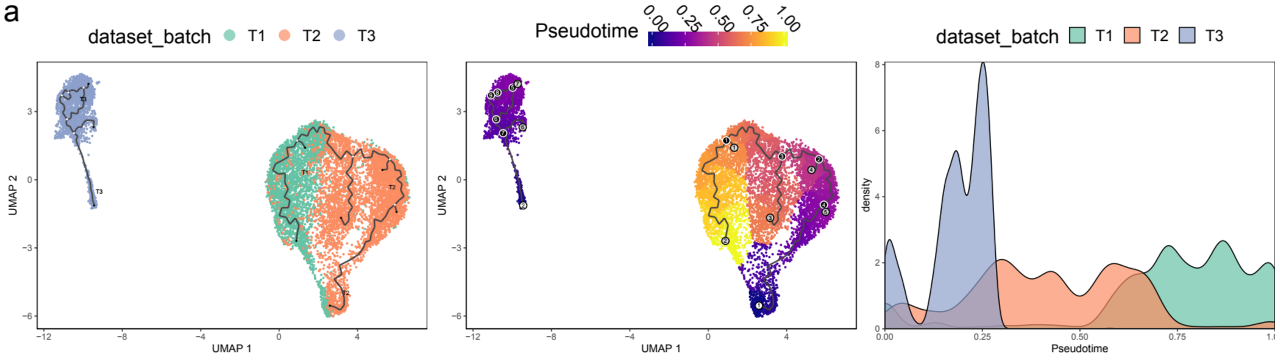


Supplemental Fig. S14. Methods comparison for clustering in the embedding space on the human PBMC data. Here, we used embedding space driven methods to cluster the data, rather than using a full gene-space representation. We used the embedding as input for UMAP projection, and colored UMAPs both by cell type (top row) and batch annotation (bottom row).

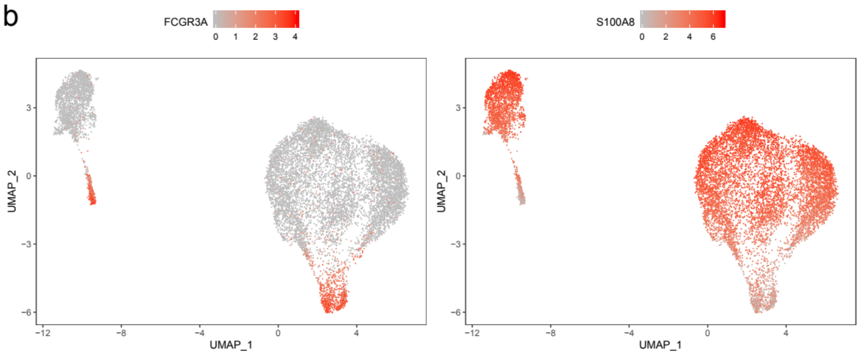


Supplemental Fig. S15. Methods comparison for batch correcting HVGs only on the PBMC data. Here, we fit batch correction methods on the HVGs only of the PBMC data, rather than using HVGs and LVGs together. We used the batch corrected HVGs as input for UMAP projection, and colored UMAPs both by cell type (top row) and batch annotation (bottom row).

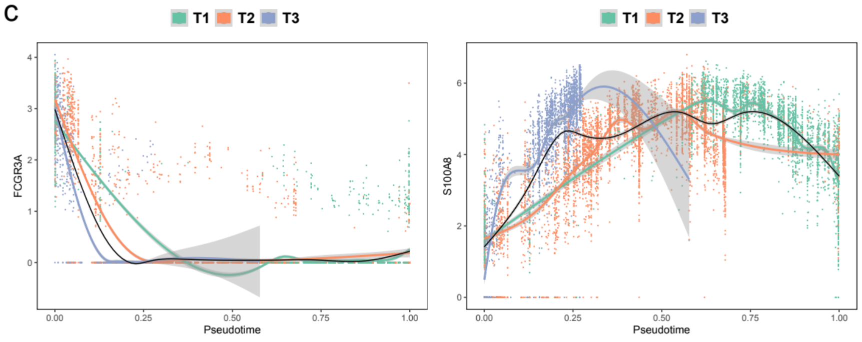
a



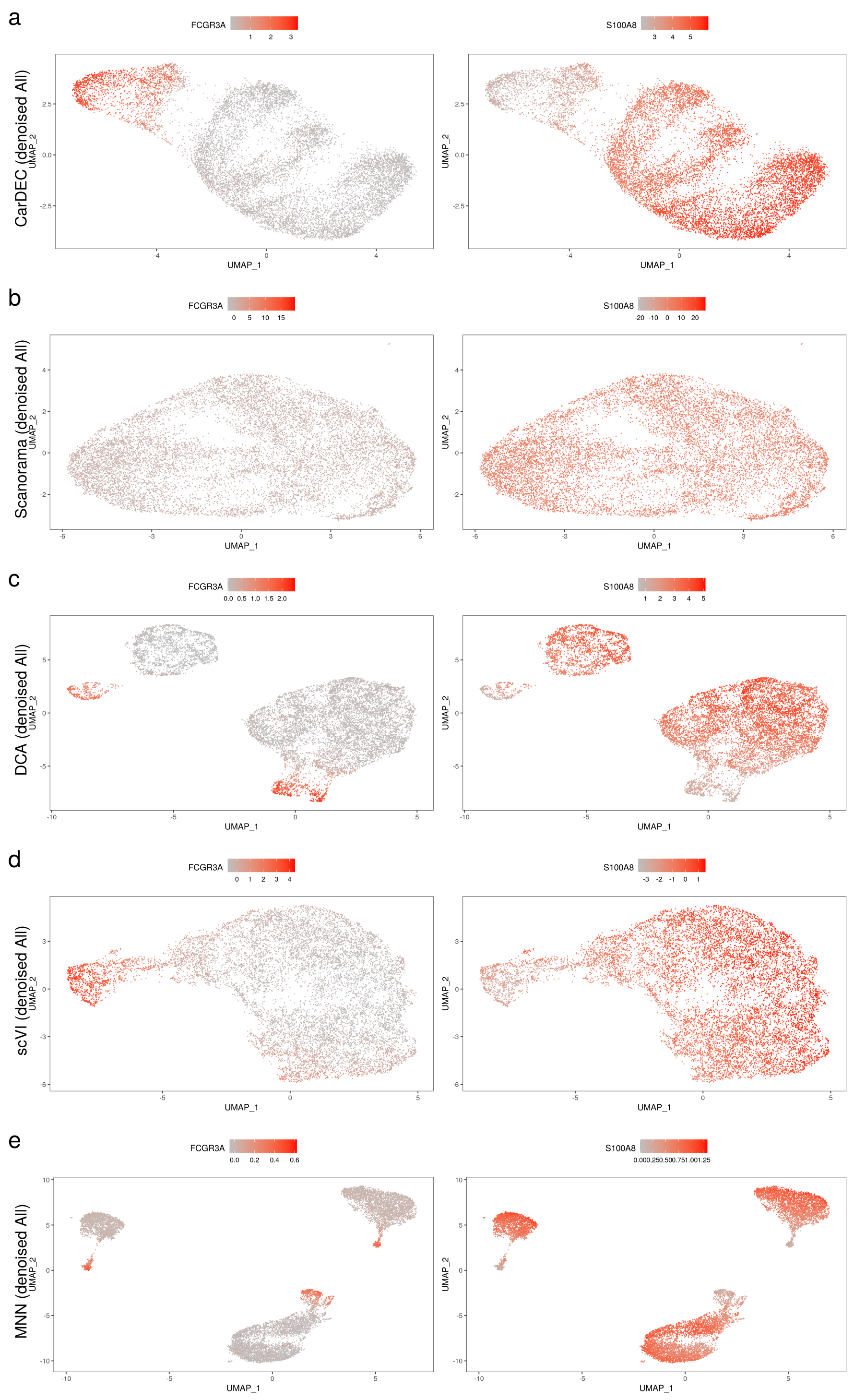
b



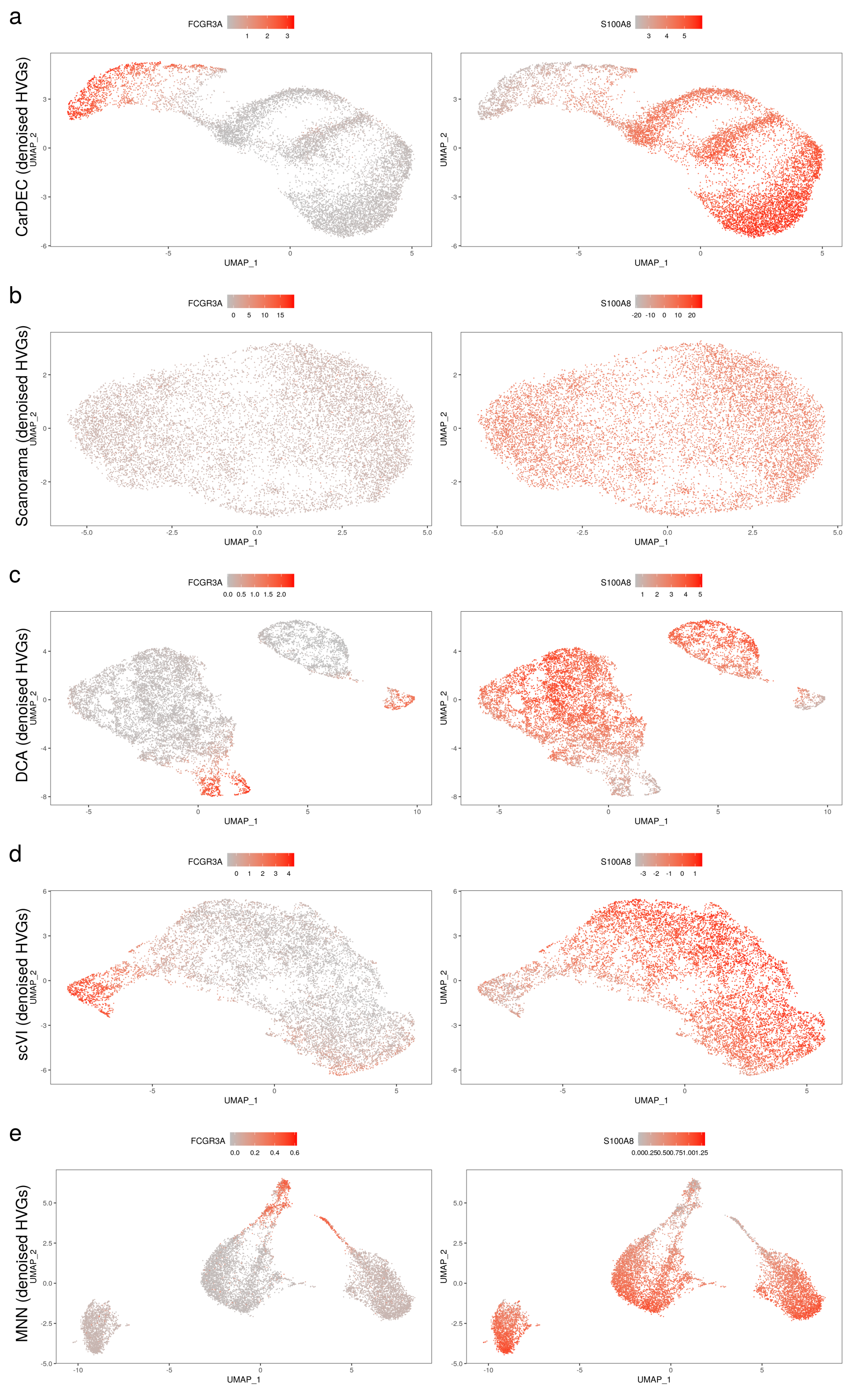
c



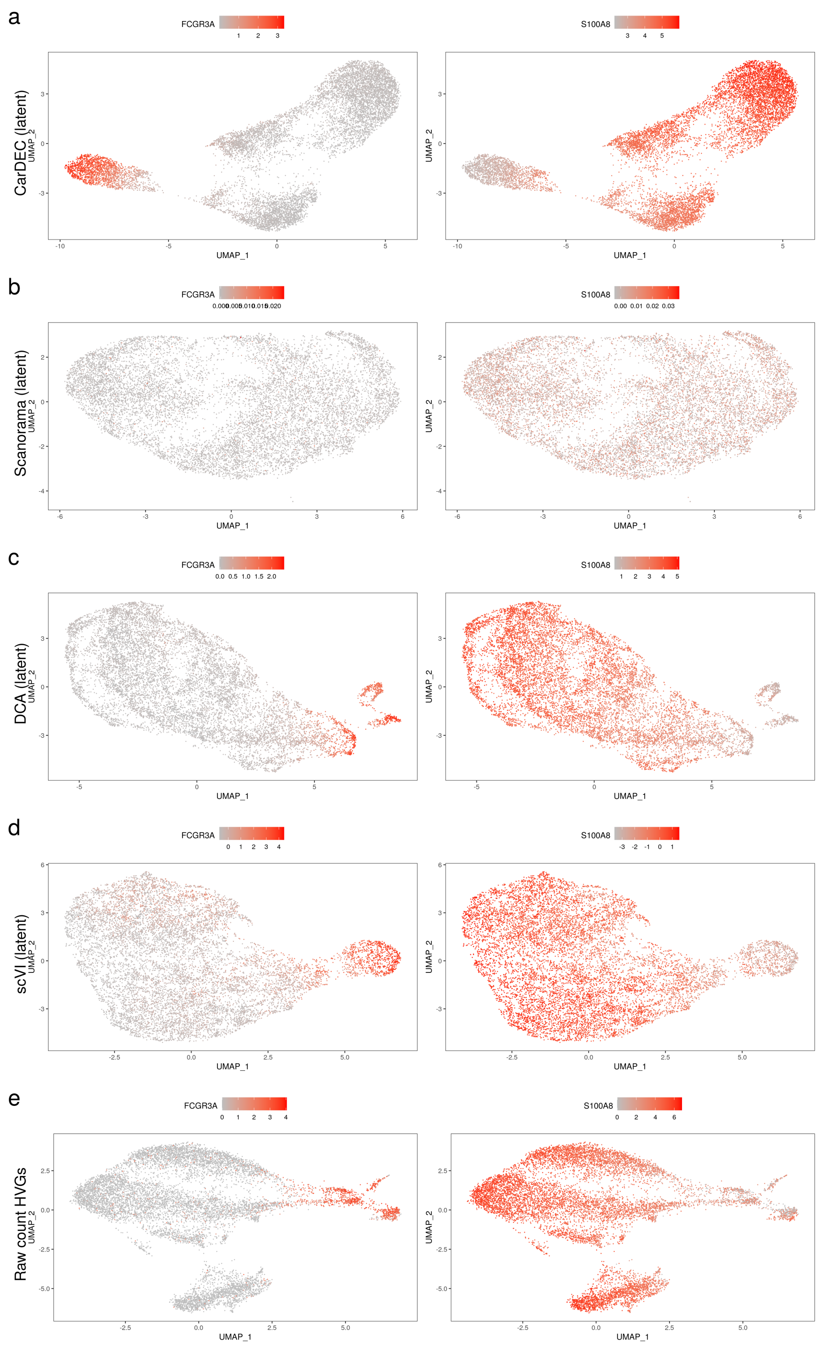
Supplemental Fig. S16. Baseline performance for pseudotime analysis on the human monocyte data. Here, we demonstrate the performance of Monocle 3 for pseudotime reconstruction with no denoising/batch correction method applied as a preprocessing step, to provide a baseline for improving pseudotime results. We fed in the matrix of raw counts for all genes as input to Monocle 3. (a) Monocle 3 UMAP embedding colored by batch (left), pseudotime (middle), and the kernel density distribution of pseudotime by batch (right). (b) Monocle 3 UMAP embedding colored by *FCGR3A* marker gene expression (left) and *S100A8* marker gene expression (right). (c) The distributions of marker genes *FCGR3A* (left) and *S100A8* (right) against pseudotime.



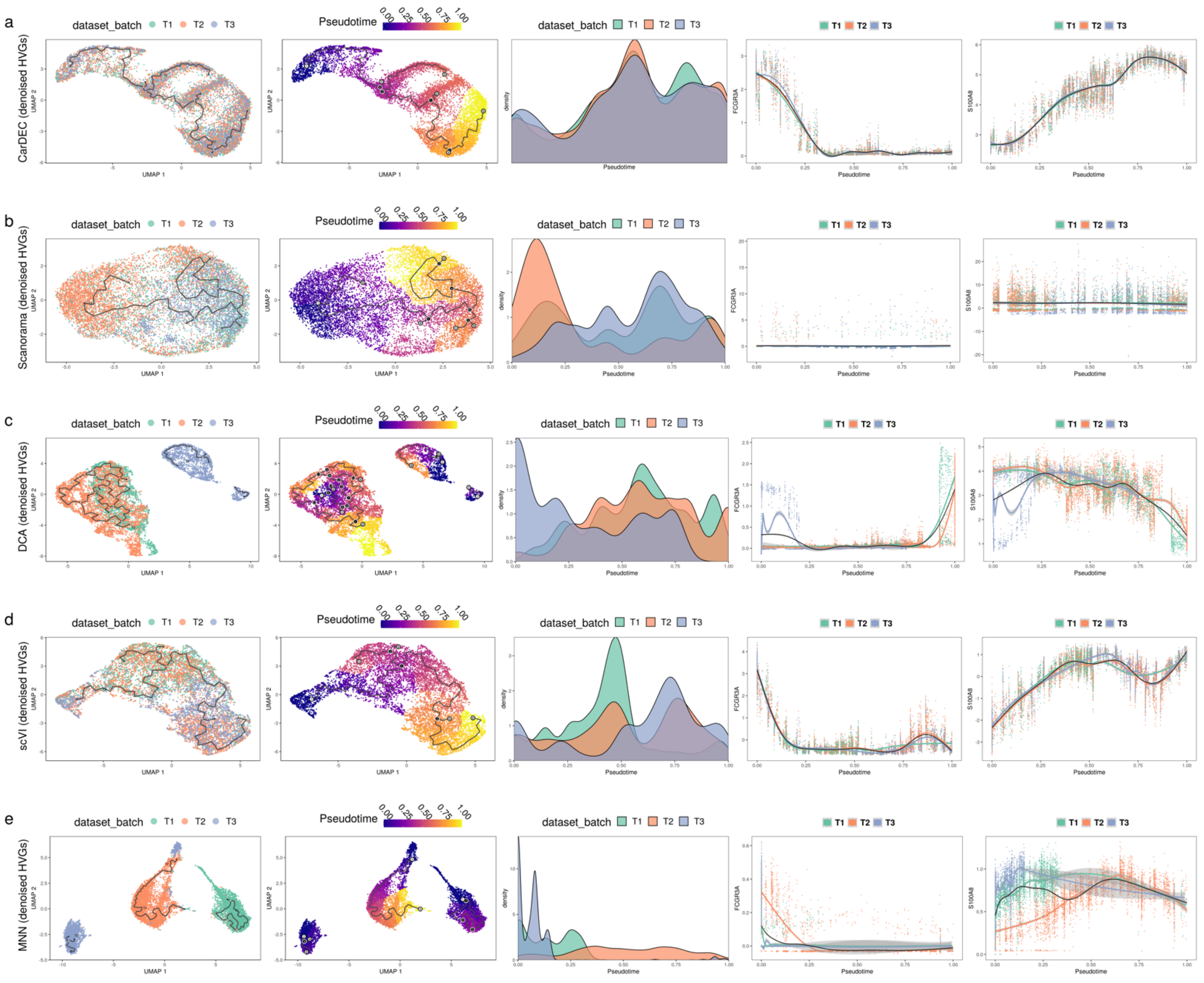
Supplemental Fig. S17. Marker gene expression colored UMAPs using denoised all genes as input to Monocle 3 on the human monocyte data. Monocle 3 UMAP embeddings colored by expression of marker genes *FCGR3A* (column 1) and *S100A8* (column 2). Note that in this case for each method, we ran the denoising/batch correction method and then passed the full denoised/batch corrected matrix as input to Monocle 3 as described in Figure 6. Marker gene expression colored UMAP embeddings using CarDEC denoised and batch corrected matrix (a), Scanorama batch corrected gene expression matrix (b), DCA denoised matrix with Combat post-hoc batch effect correction (c), scVI denoised and batch corrected matrix (d), and MNN batch corrected matrix (e), as input to Monocle 3.



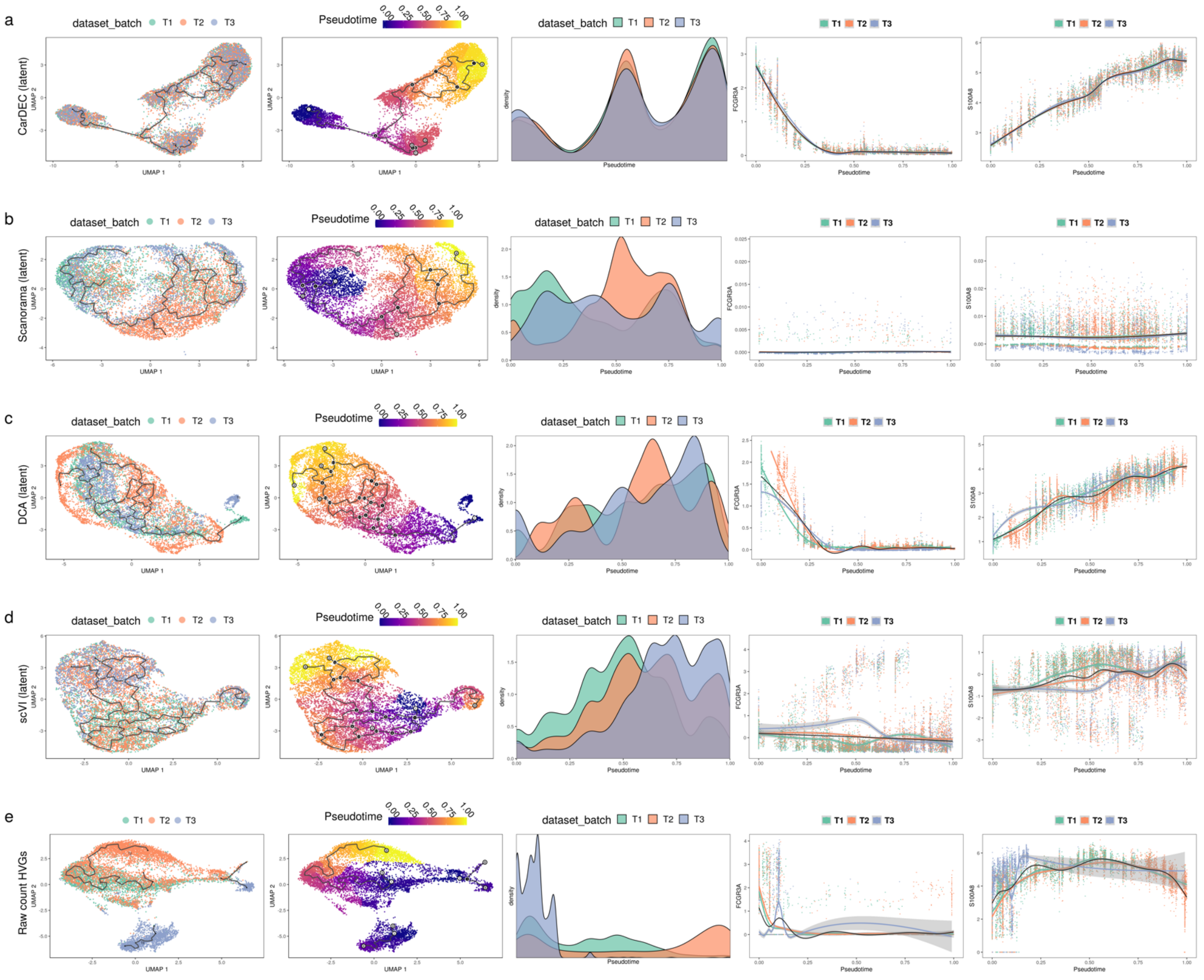
Supplemental Fig. S18. Marker gene expression colored UMAPs using denoised HVGs as input to Monocle 3 on the human monocyte data. Monocle 3 UMAP embeddings colored by expression of marker genes *FCGR3A* (column 1) and *S100A8* (column 2). Note that in this case for each method, we ran the denoising/batch correction method, subsetted the resulting denoised/corrected matrix to include only HVGs, and then passed the HVG only denoised/batch corrected matrix as input to Monocle 3 as described in Supplemental Fig. S17. Marker gene expression colored UMAP embeddings using CarDEC denoised and batch corrected HVGs (a), Scanorama batch corrected gene expression matrix (b), DCA denoised matrix with Combat post-hoc batch effect correction (c), scVI denoised and batch corrected matrix (d), and MNN batch corrected matrix (e), as input to Monocle 3.



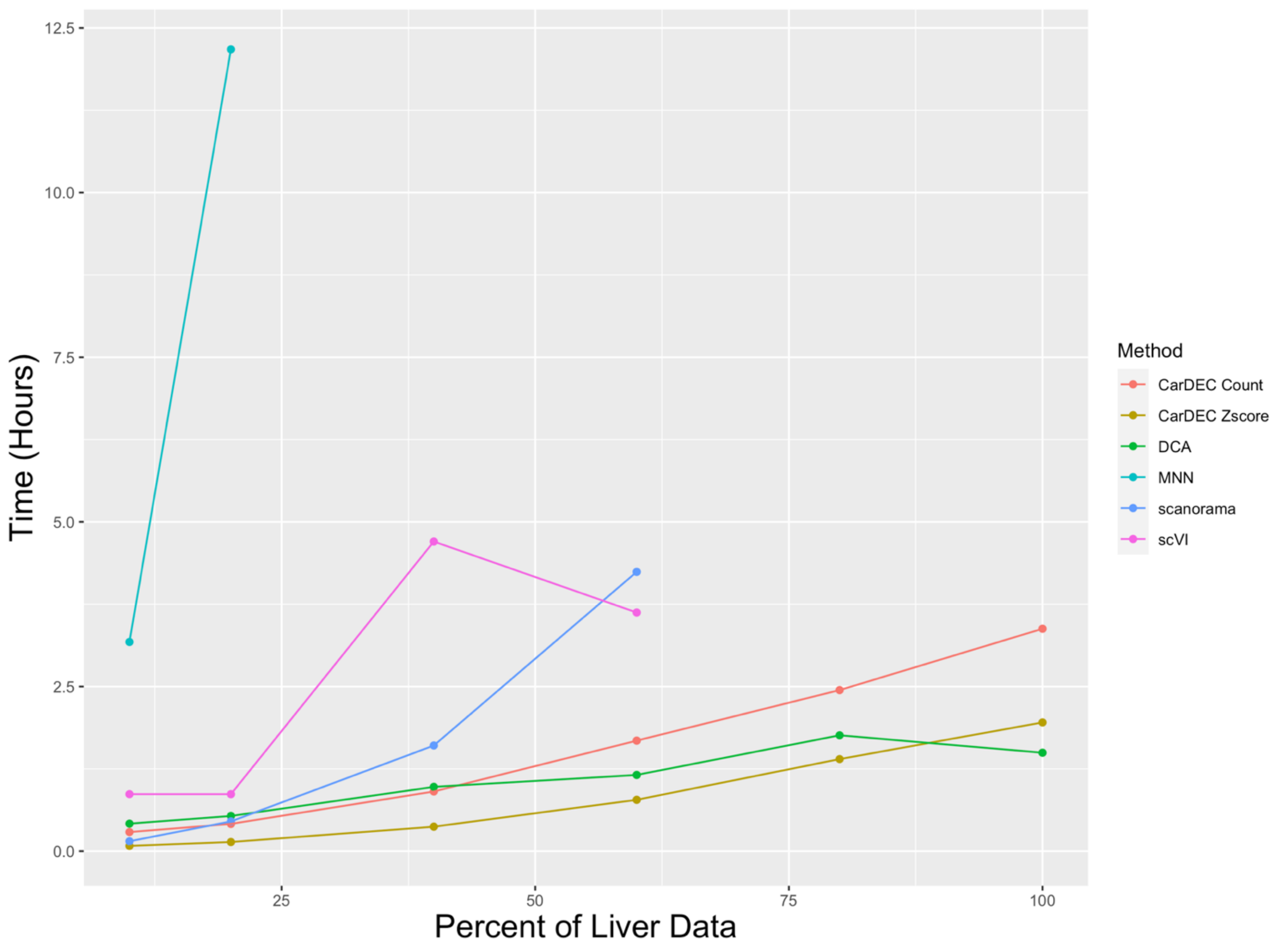
Supplemental Fig. S19. Marker gene expression colored UMAPs using HVG learned embeddings from each method as input to Monocle 3 on the human monocyte data. Monocle 3 UMAP embeddings colored by expression of marker genes *FCGR3A* (column 1) and *S100A8* (column 2). Note that in this case for each method, instead of using Monocle 3's PCA based approach for dimension reduction, we use the learned embedding from each method as the dimension reduced space for pseudotime graph construction as described in Supplemental Fig. S18. Marker gene expression colored UMAP embeddings using CarDEC denoised and batch corrected HVGs (a), Scanorama batch corrected gene expression matrix (b), DCA denoised matrix with Combat post-hoc batch effect correction (c), scVI denoised and batch corrected matrix (d), and MNN batch corrected matrix (e), as input to Monocle 3.



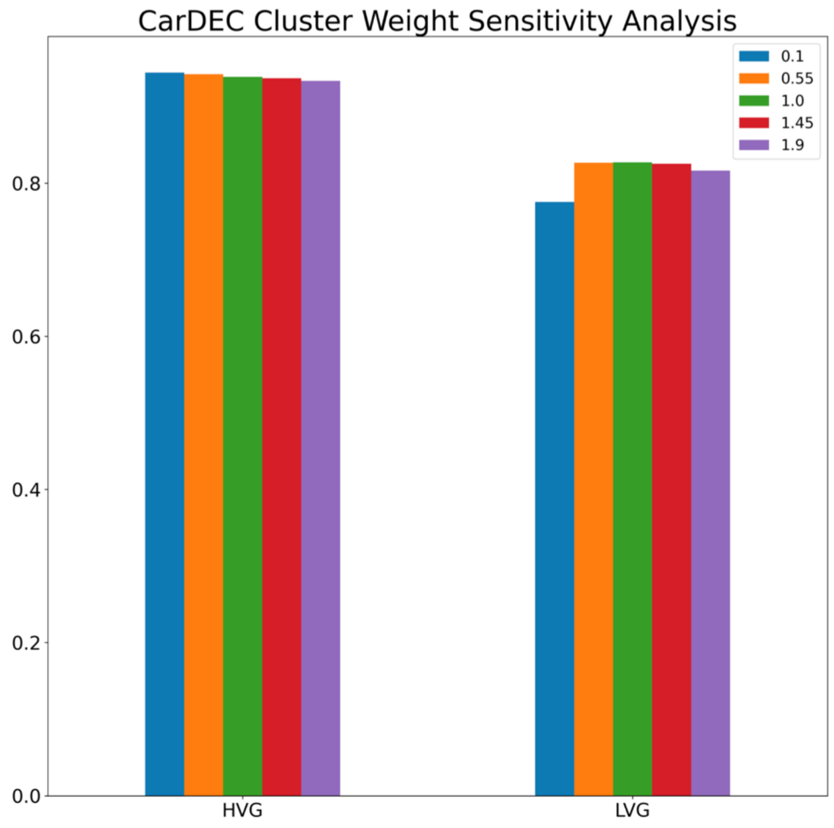
Supplemental Fig. S20. Methods comparison for pseudotime analysis on the human monocyte data using denoised and batch corrected HVGs only. We benchmarked the performance of CarDEC for improving pseudotime analysis using Monocle 3 on the Monocyte data generated by Li *et al*. (Li et al. 2020). For each method, the full dataset was denoised and batch corrected. Then we subsetted this dataset to include only HVGs, fed this to Monocle 3 for pseudotime analysis. This figure shows UMAP embedding colored by batch (column 1), pseudotime (column 2), the kernel density distribution of psuedotime by batch (column 3), and the distributions of marker genes *FCGR3A* (non-classifical monocytes) and *S100A8* (classical monocytes) against pseudotime (columns 4 and 5, respectively). (a) Pseudotime analysis when using denoised/corrected HVG gene matrix from CarDEC as input. (b) Pseudotime analysis when using batch corrected HVG gene matrix from Scanorama as input. (c) Pseudotime analysis when using denoised HVG gene matrix from DCA and batch corrected by Combat as input. (d) Pseudotime analysis when using denoised and batch corrected HVG gene matrix from scVI as input. (e) Pseudotime analysis when using batch corrected HVG gene matrix from MNN as input.



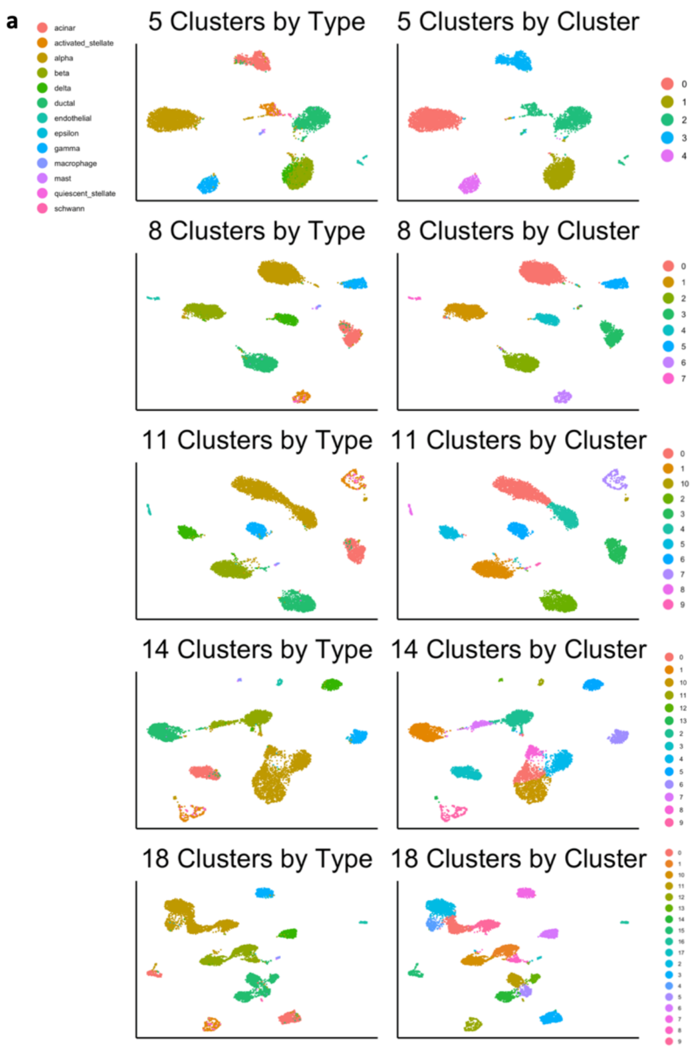
Supplemental Fig. S21. Methods comparison for pseudotime analysis on the human monocyte data using embedding. We benchmarked the performance of CarDEC for improving pseudotime analysis using Monocle 3. For each method, the full dataset was denoised/batch corrected. Then instead of using Monocle 3's PCA based approach for dimension reduction, we used the learned embedding from the corresponding method as the dimension reduced space for pseudotime graph construction. Since MNN had no embedding, it was omitted from this analysis. This figure shows UMAP embedding colored by batch (column 1), pseudotime (column 2), the kernel density distribution of psuedotime by batch (column 3), and the distributions of marker genes FCGR3A (non-classifical monocytes) and S100A8 (classical monocytes) against pseudotime (columns 4 and 5, respectively). (a) Pseudotime analysis when using denoised/corrected HVG gene matrix from CarDEC as input. (b) Pseudotime analysis when using batch corrected HVG gene matrix from Scanorama as input. (c) Pseudotime analysis when using denoised HVG gene matrix from DCA and batch corrected by Combat as input. (d) Pseudotime analysis when using denoised and batch corrected HVG gene matrix from scVI as input. (e) Pseudotime analysis when using batch corrected HVG gene matrix from MNN as input.

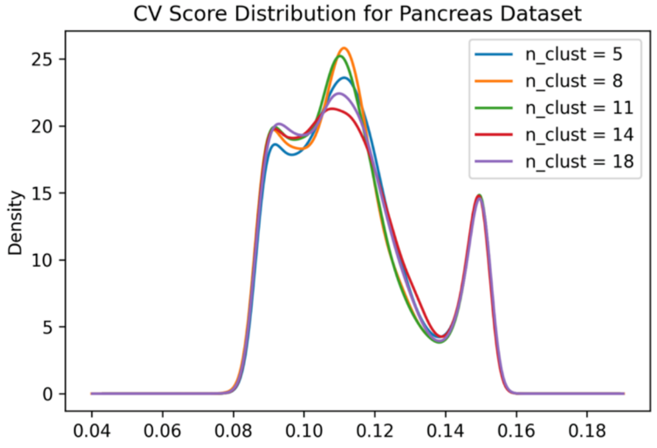


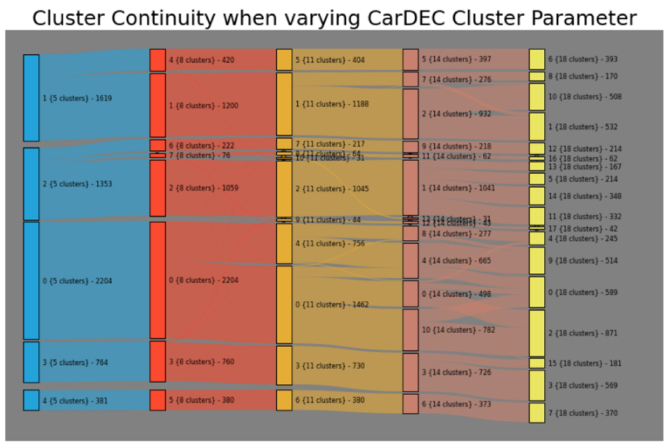
Supplemental Fig. S22. Scalability analysis. We recorded the runtimes of the denoising methods on the liver dataset of over 100,000 cells. For each method we recorded the runtime needed to process 10%, 20%, 40%, 60%, 80%, and 100% of the data. We included two variations of CarDEC: CarDEC Z-score, which only provides denoised expression in the Z-score space, and CarDEC Count, which provides denoised expression in the count space.



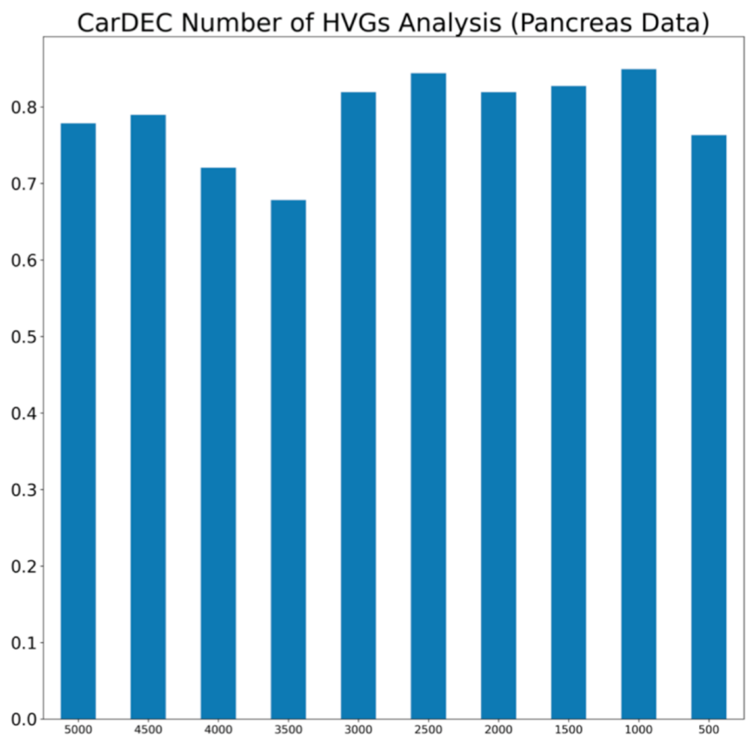
Supplemental Fig. S23. Clustering weight robustness analysis. In this supplementary figure, we explore the robustness of CarDEC to the choice of parameter α that governs the balancing of the reconstruction and clustering loss (see methods). We rerun CarDEC for various choices of α and measure the ARI of CarDEC's batch corrected counts on HVGs and LVGs at each value of α.

a


b


c


Supplemental Fig. S24. Number of clusters analysis. In this supplementary figure, we explore the robustness of CarDEC to the choice number of clusters. Since we cannot use ARI to score clustering accuracy effectively when the number of clusters differs from the gold standard labels, we use other approaches to evaluate CarDEC's robustness to choice of cluster number. We use the pancreas dataset for this analysis. (a) Here, we plot UMAP embeddings computed from CarDEC corrected counts obtained for various cluster number hyperparameter specifications. In the left column, we color the plot by cell type, and in the right column we color the plot by CarDEC cluster ID. (b) Here, we plot the CV score distribution for each run of CarDEC (with different cluster number specification). (c) In this plot, we use a Sankey plot to show how each cell is recategorized into a new cluster after CarDEC is rerun with more clusters. The thickness of a curve connecting one cluster to the next demonstrates how many cells in the first cluster were recategorized into the second cluster after CarDEC was rerun with a higher number of clusters specified. This third plot shows that when CarDEC is rerun with more clusters, existing clusters are usually just split into two (or three), with cells from different batches generally not mixing.



Supplemental Fig. S25. Number of HVGs robustness analysis. In this supplementary figure, we explore the robustness of CarDEC to the choice number of HVGs. We rerun CarDEC on the pancreas dataset multiple times, with differing numbers of HVGs for each run. At each run, we measure ARI on the bottom 16215 LVGs (which remains the same across all runs).

Supplemental Note 1: Details of the CarDEC algorithm

The full workflow of CarDEC is shown in **Fig. 1**. Below we describe each step in CarDEC in detail.

Step 1: preprocessing

There are four important tasks to be completed in preprocessing: cell normalization, log normalization, identify highly variable genes (HVGs), and Z-score normalization.

Let $\mathbf{X}$ be an $n \times p$ matrix of raw scRNA-seq data, with n cells and p genes. Also, let x_{ij} be the expression value of gene j in cell i. In cell normalization, we compute a *size factor* for each cell as follows. Let s_i be the size factor for cell i. For each cell, compute the sum of read counts in that cell over all genes, and let t_i denote the sum of read counts for cell i. Let m be the median of t_i for $i \in \{1,2, \dots, n\}$. Then for cell i, $s_i = t_i/m$. To perform cell normalization and get the cell normalized count y_{ij} we divide the expression of each gene in each cell by the cell's size factor as follows,

$$y_{ij} \leftarrow x_{ij}/s_i.$$

To update the normalized counts y_{ij}, so that they are log-normalized, we simply add a pseudocount and then take the natural logarithm elementwise as follows,

$$y_{ij} \leftarrow \log{(y_{ij} + 1)}.$$

At this point, we can use the log-normalized counts to determine which genes are highly variable and which are not. To do so, we use the approach for selecting HVGs introduced by the Seurat 3.0 paper(Stuart et al. 2019) and implemented in the Scanpy package(Wolf et al. 2018). Specifically, we call the Scanpy function "pp.highly_variable_genes" with "batch_key" parameter identifying the user-supplied vector of cellwise batch annotations, to select HVGs using within batch variance.

Lastly, we Z-score standardize the y_{ij}. However, we do not do a simple Z-score standardization that spans all cells. Rather, we Z-score standardize within each batch. More precisely, suppose we have batches b_m for $m = 1, 2, \dots, M$. Let B_m be the set of cell indices associated with cells sequenced in batch b_m. That is, $i \in B_m$ if and only if cell i is from batch b_m. Then for each gene j, we compute batch specific mean μ_{mj} and variance σ_{mj}^2 for each batch m as follows,

$$\mu_{mj} = \frac{\sum_{i \in B_m} y_{ij}}{\sum_{i \in B_m} 1},$$
$$\sigma_{mj}^2 = \frac{\sum_{i \in B_m} \left(y_{ij} - \mu_{mj}\right)^2}{\left(\sum_{i \in B_m} 1\right) - 1}.$$

Then, we Z-score standardize each y_{ij} using the mean and variance of the batch corresponding to cell i. More precisely, let m_i be defined such that $i \in B_{m_i}$. Then our batch-specific Z-score normalization is performed as follows:

$$y_{ij} \leftarrow \frac{y_{ij} - \mu_{m_i j}}{\sqrt{\sigma^2_{m_i j}}}.$$

Step 2: pretraining

The pretraining step is a straightforward implementation of an autoencoder. First, let $\mathbf{Y}_{HVG}$ be the $n \times p_{HVG}$ matrix of normalized expression from Step 1, subsetted to include only the p_{HVG} HVGs. Let $\boldsymbol{y}_{i,HVG}$ be the vector of HVGs in cell i, that is, the i^{th} row of $\boldsymbol{Y}_{HVG}$.

Define a standard autoencoder with encoder and decoder's functionally represented by $f_{E,HVG}(\,\cdot\,;\, W_{E,HVG})$ and $f_{D,HVG}(\,\cdot\,;\, W_{D,HVG})$. Both the encoder and decoder are flexibly user-definable compositions of network layers. The weights $W_{E,HVG}$ and $W_{D,HVG}$ are randomly initialized using the glorot uniform approach, and will be tuned during pretraining. The encoder maps to a low-dimensional embedding $\boldsymbol{z}_{i,HVG}$ as follows, where $\boldsymbol{z}_{i,HVG}$ has dimension $d \ll p_{HVG}$,

$$\boldsymbol{z}_{i,HVG} = f_{E,HVG}(\boldsymbol{y}_{i,HVG};\, W_{E,HVG}).$$

Then, $f_{D,HVG}(\,\cdot\,;\, W_{D,HVG})$ maps the low-dimensional embedding to a reconstruction $\hat{\boldsymbol{y}}_{i,HVG}$ in the original p_{HVG} dimension space. That is,

$$\hat{\boldsymbol{y}}_{i,HVG} = f_{D,HVG}(\boldsymbol{z}_{i,HVG};\, W_{D,HVG}).$$

For activation functions, we use the tanh activation for the output of the encoder, and the linear activation function for the output of the decoder. For all intermediate hidden layers in the encoder and decoder, we use the ReLu activation function.

We use the mean square error objective function to pretrain our autoencoder. Specifically, the loss for cell i is

$$l_i = \frac{1}{p_{HVG}} \left\| \hat{\boldsymbol{y}}_{i,HVG} - \boldsymbol{y}_{i,HVG} \right\|^2.$$

The autoencoder is then pretrained end-to-end using minibatch gradient descent, with the Adam optimizer(Kingma and Ba 2015) used to make gradient descent updated to the weights $W_{E,HVG}$ and $W_{D,HVG}$. The model is trained until the early stopping criterion is satisfied. Specifically, if the validation loss does not decrease for hyperparameter *patience_ES* epochs, then training is halted. We also decay the learning rate by a factor of hyperparameter *decay_factor* if validation loss does not decrease for hyperparameter *patience_LR* epochs. TensorFlow is used as the framework of choice for defining the network and performing reverse mode automatic differentiation for this step and all subsequent steps in CarDEC.

Step 3: denoising Z-scores

In this phase, we use an expanded, branching architecture to accommodate those genes that are not detected as HVGs, which we call as lowly variable genes (LVGs). Specifically, we introduce a clustering loss that regularizes the embedding and improve batch mixing and denoising especially in the gene space.

First, let $\mathbf{Y}_{HVG}$ be the $n \times p_{HVG}$ matrix of normalized expression from Step 1, subsetted to include only the p_{HVG} HVGs. Let $\boldsymbol{y}_{i,HVG}$ be the vector of HVGs in cell i, that is, the i^{th} row of $\boldsymbol{Y}_{HVG}$. Likewise, let $\mathbf{Y}_{LVG}$ be the $n \times p_{LVG}$ matrix of normalized expression from Step 1, subsetted to include only the p_{LVG} LVGs. Let $\boldsymbol{y}_{i,LVG}$ be the vector of LVGs in cell i, that is, the i^{th} row of $\mathbf{Y}_{LVG}$.

We retain the encoder and decoder mappings for HVGs, $f_{E,HVG}(\cdot\,; W_{E,HVG})$ and $f_{D,HVG}(\cdot\,; W_{D,HVG})$ from Step 2, including the learnt weights $W_{E,HVG}$ and $W_{D,HVG}$. We introduce a clustering layer which takes the HVG embedding $\boldsymbol{z}_{i,HVG}$ as input and returns for each cell a vector of cluster membership probabilities for h clusters, where h is a user specified number. For this clustering layer, we introduce an $h \times d$ matrix of trainable weights $\mathbf{M}$. The j^{th} row of $\mathbf{M}$ is a cluster centroid $\boldsymbol{\mu}_j$.

To initialize $\mathbf{M}$, we run Louvain's algorithm on the embeddings $\boldsymbol{z}_{i,HVG}$ for $i = 1, 2, \dots, n$ learned from the pretrained autoencoder, and we find the cluster centroid for each of the h clusters. Note that Louvain's algorithm takes a "resolution" parameter rather than the number of clusters directly, so we use a bisection algorithm to find a resolution that gives h clusters. This approach is described later in this section.

The clustering layer computes a vector of cluster membership probabilities for cell i, denoted by $\boldsymbol{q}_i$. Let q_{ij}, the j^{th} element of $\boldsymbol{q}_i$, denote the probability that cell i belongs to cluster j. Then the membership probabilities are computed using a *t*-distribution kernel as follows,

$$q_{ij} = \frac{\left(1 + \left\|\boldsymbol{z}_{i,HVG} - \mu_j\right\|^2\right)^{-1}}{\sum_{j'}\left(1 + \left\|\boldsymbol{z}_{i,HVG} - \mu_{j'}\right\|^2\right)^{-1}}.$$

Since we do not have a vector of cell type labels in a practical unsupervised analysis, we need "pseudo-labels" that can be used in place of real labels for optimizing clustering weights. These pseudo-labels are computed from the membership probabilities q_{ij} as follows,

$$p_{ij} = \frac{q_{ij}^2 / \sum_i q_{ij}}{\sum_j' q_{ij'}^2 / \sum_i q_{ij'}}.$$

Let $\boldsymbol{p}_i$ be an h dimensional vector whose j^{th} element is p_{ij}. Then the clustering loss for cell i is defined as the following Kullback–Leibler divergence (KLD),

$$l_{i,c} = KLD(\boldsymbol{p}_i || \boldsymbol{q}_i) = \sum_j p_{ij} \log\left(\frac{p_{ij}}{q_{ij}}\right).$$

This loss is a component of the total loss. Since it takes the embedding vectors as input $\boldsymbol{z}_{i,HVG}$, minimizing this objective function has the effect of refining the embedding, helping to remove batch effects from denoised counts computed using this embedding as input.

We also introduce encoder and decoder mappings $f_{E,LVG}(\,\cdot\,;\, W_{E,LVG})$ and $f_{D,LVG}(\,\cdot\,;\, W_{D,LVG})$ to address the problem of denoising and batch correcting the LVGs. Since these mappings were not included in the pretrain step, their weights $W_{E,LVG}$ and $W_{D,LVG}$ are randomly initialized using glorot uniform. This encoder and corresponding decoder are also flexible compositions of network layers, which the user can define independently of the HVG encoder and decoder. The encoder maps to a low-dimensional embedding $\boldsymbol{z}_{i,LVG}$ as follows, where $\boldsymbol{z}_{i,LVG}$ has dimension $d_2 \ll p_{LVG}$,

$$\boldsymbol{z}_{i,LVG} = f_{E,LVG}(\boldsymbol{y}_{i,LVG};\, W_{E,LVG}).$$

Unlike the HVG decoder $f_{D,HVG}$, the LVG decoder $f_{D,LVG}(\,\cdot\,;\, W_{D,HVG})$ does not map the low-dimensional embedding $\boldsymbol{z}_{i,LVG}$ alone to a reconstruction $\widehat{\boldsymbol{y}}_{i,LVG}$ in the original p_{LVG} dimension space. Rather, we concatenate the HVG and LVG embeddings together, and feed the combined vector $[\boldsymbol{z}_{i,HVG}\ \ \boldsymbol{z}_{i,LVG}]$ of length $d + d_2$ into the decoder to denoise and batch correct LVG expression in the original p_{LVG} dimension space. That is,

$$\widehat{\boldsymbol{y}}_{i,LVG} = f_{D,HVG}([\boldsymbol{z}_{i,HVG}\ \ \boldsymbol{z}_{i,LVG}];\, W_{D,LVG}).$$

This formulation allows CarDEC to only allow high signal HVGs to drive the clustering loss, while still using the rich, batch corrected embedding that was refined using this clustering loss to denoise and batch correct LVGs.

For activation functions, we again use the tanh activation for the outputs of the encoders, and the linear activation function for the outputs of the decoders. For all intermediate hidden layers in the encoders and decoders, we use the ReLu activation function. The clustering layer we introduced doesn't have any standard activation function that is typically used in deep learning models, although the t-distribution kernel can be thought of as an activation for this layer.

To train this branching model, we use a multi-component loss function. We already presented the cluster loss $l_{i,c}$, the KL divergence between $\boldsymbol{p}_i$ and $\boldsymbol{q}_i$. In addition, we also include two reconstruction losses, one for HVGs and one for LVGs computed as follows for cell i,

$$l_{i,HVG} = \frac{1}{p_{HVG}} \left\|\widehat{\boldsymbol{y}}_{i,HVG} - \boldsymbol{y}_{i,HVG}\right\|^2,$$
$$l_{i,LVG} = \frac{1}{p_{LVG}} \left\|\widehat{\boldsymbol{y}}_{i,LVG} - \boldsymbol{y}_{i,LVG}\right\|^2.$$

Let α be a hyperparameter ranging from 0 to 2, which balances reconstruction loss with clustering loss. Then the total loss is computed as follows:

$$l_i = \alpha l_{i,c} + (2 - \alpha)\frac{l_{i,HVG} + l_{i,LVG}}{2}.$$

This loss is minimized in an iterative fashion. After initializing the cluster centroids using Louvain's algorithm, we compute cluster assignment vectors $\boldsymbol{q}_i$ and pseudo-labels $\boldsymbol{p}_i$. Then we iterate back and forth between updating the weights and updating the target distribution. This iteration is necessary because we need pseudo-labels $\boldsymbol{p}_i$ to compute the loss so that the network weights can be optimized to minimize it, but the pseudo-labels themselves are a function of the network weights.

When updating the weights, we use TensorFlow to perform automatic differentiation on the loss function for minibatch gradient descent and update all weights with the Adam optimizer. When computing the loss for differentiation, we use the most up to date pseudolabels $\boldsymbol{p}_i$, which are treated as fixed during this step. The set of weights updated include HVG mapping weights $W_{E,HVG}$ and $W_{D,HVG}$, LVG mapping weights $W_{E,LVG}$ and $W_{D,LVG}$, and the cluster centroids $\mathbf{M}$. We loop through the entire dataset, updating these weights once for each minibatch until we have completed a single epoch of minibatch updates.

Once we complete an epoch of minibatch of gradient descent updates, we then switch to updating the target distribution $\boldsymbol{p}_i$. Now, we fix the weights. We compute the cluster assignment probabilities $\boldsymbol{q}_i$ and then compute $\boldsymbol{p}_i$ from $\boldsymbol{q}_i$. We fix this updated distribution $\boldsymbol{p}_i$ and switch back to updating the weights. We iterate back and forth until certain convergence criteria are satisfied.

To monitor convergence, we implement learning rate decay and early stopping. Let an iteration be defined by a single pair of target distribution update and minibatch gradient descent epoch steps. For learning rate decay, we monitor the reconstruction loss $\frac{l_{i,HVG}+l_{i,LVG}}{2}$. If the validation reconstruction loss fails to decrease after hyperparameter *iteration_patience_LR* iterations, then the learning rate is decayed by a factor of hyperparameter *iteration_decay_factor*. For early stopping we require two conditions to halt training. First, we require that the validation reconstruction loss not have decreased in hyperparameter *iteration_patience_ES* iterations. Secondly, we require that the proportion of cells whose most likely cluster assignment changed be less than hyperparameter *tol* on the most recent iteration. If both of these criteria are satisfied at the conclusion of an iteration, then training is halted.

Step 4: denoising counts

In Step 3, the reconstruction loss is the mean squared error between the reconstructed outputs of the decoder and the Z-score normalized input to the model. So naturally, the denoised expression values obtained from the model are on a Z-score scale and are not comparable to raw counts. To remedy this, we offer an optional downstream modeling step that will provide denoised expression values on the count scale. This strategy involves finding mean and dispersion parameters that maximize a negative binomial likelihood.

After the training in step 3, we have fine-tuned HVG and LVG encoders that can produce information rich, batch corrected low-dimension embeddings for each cell. For each cell i, we obtain low-dimension embeddings $\boldsymbol{z}_{i,HVG}$ and $\boldsymbol{z}_{i,LVG}$ from the fine-tuned HVG and LVG encoders. We will then use two separate neural networks to maximize negative binomial losses: one for HVGs and one for LVGs. These models are completely separate from one another but are trained almost identically with only minor differences.

First, we introduce two separate mappings $f_{C,HVG}(\cdot;\, W_{C,HVG})$ and $f_{C,LVG}(\cdot;\, W_{C,LVG})$ that are similar to decoders introduced in previous steps, but exclude the final output layer, which must be treated more carefully since we want two sets of parameters for each feature outputted.

$f_{C,HVG}(\cdot;\, W_{C,HVG})$ maps d −dimensional $\mathbf{z}_{i,HVG}$ to a $d' > d$ dimension vector $\tilde{\mathbf{z}}_{i,HVG}$. Similarly, $f_{C,LVG}(\cdot;\, W_{C,LVG})$ maps the concatenated $(d + d_2) -$ dimensional embedding $[\mathbf{z}_{i,HVG}\ \mathbf{z}_{i,LVG}]$ to a $d'' > (d + d_2)$ dimension vector $\tilde{\mathbf{z}}_{i,LVG}$. All activations for both of these mappings are ReLu. We require these count models to have the same hidden layer dimensions as the main CarDEC model's HVG and LVG decoders, respectively.

For both the HVGs and the LVGs, we then need to map these higher-dimensional embeddings into the full gene space to obtain mean and dispersion parameters for each gene. The exact same operation is done for both the HVGs and the LVGs, so without loss of generality we show the equations only for the HVGs. The vector of genewise means $\boldsymbol{\mu}_{i,HVG}$ and vector genewise dispersions $\boldsymbol{\theta}_{i,HVG}$ are given below,

$$\boldsymbol{\mu}_{i,HVG} = s_i \times exp(\mathbf{W}_{\mu,HVG} \times \tilde{\mathbf{z}}_{i,HVG}),$$
$$\boldsymbol{\theta}_{i,HVG} = softplus(\mathbf{W}_{\theta,HVG} \times \tilde{\mathbf{z}}_{i,HVG}).$$

Here, s_i is the size factor for cell i computed in Step 1, $\mathbf{W}_{\mu,HVG}$ and $\mathbf{W}_{\theta,HVG}$ are trainable weight matrices each of dimension $p_{HVG} \times d'$, and exp and $softplus$ are activation functions that are applied elementwise. The equations to obtain negative binomial parameters $\boldsymbol{\mu}_{i,LVG}$ and $\boldsymbol{\theta}_{i,LVG}$ for the LVGs are nearly identical and can be obtained by replacing all instances of "HVG" with "LVG" in the above equations. $\mathbf{W}_{\mu,LVG}$ and $\mathbf{W}_{\theta,LVG}$ will have dimensions $p_{LVG} \times d''$.

For each gene j in cell i, we can compute the negative log likelihood of the negative binomial distribution as

$$l_{ij} = -\log\left(\frac{\Gamma(x_{ij} + \theta_{ij})}{\Gamma(\theta_{ij})}\left(\frac{\theta_{ij}}{\theta_{ij} + \mu_{ij}}\right)^{\theta_{ij}}\left(\frac{\mu_{ij}}{\theta_{ij} + x_{ij}}\right)^{x_{ij}}\right),$$

where Γ is the gamma function, x_{ij} is the original count in HVG gene j for cell i, μ_{ij} and θ_{ij} are the j^{th} elements of $\boldsymbol{\mu}_{i,HVG}$ and $\boldsymbol{\theta}_{i,HVG}$, respectively.

For the HVG count model, the full loss for cell i is then $l_i = \frac{1}{p_{HVG}} \sum_{j=1}^{p_{HVG}} l_{ij}$. The loss function for LVGs is almost identical, but when defining $l_{i,j}$ for LVGs, let μ_{ij} and θ_{ij} instead be the jth elements $\boldsymbol{\mu}_{i,LVG}$ and $\boldsymbol{\theta}_{i,LVG}$ respectively, and let x_{ij} be the original count in LVG gene j. Then the loss for the LVG count model is $l_i = \frac{1}{p_{LVG}} \sum_{j=1}^{p_{LVG}} l_{ij}$.

Both the HVG and LVG count models are trained using their own early stopping and learning rate decay convergence monitoring. These early stopping and learning rate decay monitoring strategies are exactly the same as the ones used for pretraining the autoencoder as defined in Step 2, with their own independently settable patience and decay factor hyperparameters.

Prespecifying the number of clusters for Louvain's algorithm

CarDEC uses Louvain's algorithm to initialize cluster centroids. As previously stated, Louvain's algorithm does not directly accept a number of clusters as an argument, but rather a resolution hyperparameter which correlates with the number clusters identified. To allow CarDEC users to prespecify the number of clusters they desire, we need an approach that automatically finds the resolution that corresponds to the number of clusters desired. We describe a bisection-based approach to accomplish this here.

Let m' be the desired number of clusters. After building the nearest neighbor graph, we initialize two numbers: $R_L = 0$ and $R_U = 1000$. Then set $\delta = True$, $iter = 0$, and $maxiter = 50$ and proceed as follows,

1. Compute $R = \frac{R_L + R_U}{2}$.
2. Run Louvain's algorithm with a resolution of R. Denote the number of obtained clusters by m.
3. If $m = m'$, then set $\delta \leftarrow False$.
4. If $m > m'$, then $R_U \leftarrow R$.
5. Else if $m < m'$, then $R_L \leftarrow R$.
6. $iter \leftarrow iter + 1$.
7. If $iter = maxiter$, then $\delta \leftarrow False$.
8. If $\delta = False$, stop the algorithm and return R as the Louvain resolution of choice. Else return to step 1 and repeat the above steps.

Supplemental Note 2: Hyperparameters used in different methods

For CarDEC, we used the default hyperparameters described in **Supplemental Table 1** for all datasets except for the Monocyte dataset, where we used a batch size of 256.

For Scanorama, all hyperparameters were set to the default values from the method's package, except for the dimred parameter. The dimred parameter, which represents the dimension of the integrated embedding, was set to 50 for all evaluations.

For DCA, we used the negative binomial conditional dispersion model. All hyperparameters were set to the defaults from the Scanpy implementation of DCA, including the hidden layer widths of 64, 32, 64.

For scVI, we set n_layers = 2 when specifying the deep learning architecture. We trained the model for 200 epochs. To get normalized gene expression, we follow the scVI tutorial, https://docs.scvi-tools.org/en/stable/api/reference/scvi.model.SCVI.get_normalized_expression.html#scvi.model.SCVI.get_normalized_expression, and we set the library size to 10^4 and the number of posterior samples (denoted by n_samples) to 10. All other parameters were left at their default values.

For MNN, we used all default hyperparameters.

For all methods we used the same set of 2,000 HVGs selected using Scanpy. We also usually used the same number of clusters for each method, except in the case when this resulted in a large subpopulation of cells being split in half or two large subpopulations being merged together, significantly reducing ARI for a method. For example, in the mouse cortex dataset, setting the number of clusters to 6 for scVI resulted in low ARI because there are two big subpopulations of neurons that are combined. In cases like this, we modify the number of clusters slightly, specifically we either increase the number of clusters by 1 to avoid merging two big subpopulations, or we decrease the number of clusters by 1 to avoid splitting a big subpopulation in half. In the example of scVI on mouse cortex data, we increased the number of clusters to 7 where necessary in order to avoid merging the neuron subclusters, which results in improved ARIs for scVI and a more fair comparison with that method. The number of clusters are 12 (mouse retina), 11 (macaque retina), 8 (pancreas), 6 (mouse cortex), 6 (PBMC), and 4 (Monocytes when running CarDEC).

References

Ding J, Adiconis X, Simmons SK, Kowalczyk MS, Hession CC, Marjanovic ND, Hughes TK, Wadsworth MH, Burks T, Nguyen LT et al. 2020. Systematic comparison of single-cell and single-nucleus RNA-sequencing methods. *Nat Biotechnol* **38**: 737-746.

Grun D, Muraro MJ, Boisset JC, Wiebrands K, Lyubimova A, Dharmadhikari G, van den Born M, van Es J, Jansen E, Clevers H et al. 2016. De Novo Prediction of Stem Cell Identity using Single-Cell Transcriptome Data. *Cell Stem Cell* **19**: 266-277.

Hie B, Bryson B, Berger B. 2019. Efficient integration of heterogeneous single-cell transcriptomes using Scanorama. *Nature biotechnology* **37**: 685-691.

Kingma DP, Ba JL. 2015. ADAM: a method for stochastic optimization. *International Conference on Learning Representation*.

Lawlor N, George J, Bolisetty M, Kursawe R, Sun L, Sivakamasundari V, Kycia I, Robson P, Stitzel ML. 2017. Single-cell transcriptomes identify human islet cell signatures and reveal cell-type-specific expression changes in type 2 diabetes. *Genome Res* **27**: 208-222.

Li X, Wang K, Lyu Y, Pan H, Zhang J, Stambolian D, Susztak K, Reilly MP, Hu G, Li M. 2020. Deep learning enables accurate clustering with batch effect removal in single-cell RNA-seq analysis. *Nat Commun* **11**: 2338.

Muraro MJ, Dharmadhikari G, Grun D, Groen N, Dielen T, Jansen E, van Gurp L, Engelse MA, Carlotti F, de Koning EJ et al. 2016. A Single-Cell Transcriptome Atlas of the Human Pancreas. *Cell Syst* **3**: 385-394 e383.

Peng YR, Shekhar K, Yan W, Herrmann D, Sappington A, Bryman GS, van Zyl T, Do MTH, Regev A, Sanes JR. 2019. Molecular Classification and Comparative Taxonomics of Foveal and Peripheral Cells in Primate Retina. *Cell* **176**: 1222-1237 e1222.

Popescu DM, Botting RA, Stephenson E, Green K, Webb S, Jardine L, Calderbank EF, Polanski K, Goh I, Efremova M et al. 2019. Decoding human fetal liver haematopoiesis. *Nature* **574**: 365-371.

Segerstolpe A, Palasantza A, Eliasson P, Andersson EM, Andreasson AC, Sun X, Picelli S, Sabirsh A, Clausen M, Bjursell MK et al. 2016. Single-Cell Transcriptome Profiling of Human Pancreatic Islets in Health and Type 2 Diabetes. *Cell Metab* **24**: 593-607.

Stuart T, Butler A, Hoffman P, Hafemeister C, Papalexi E, Mauck WM, 3rd, Hao Y, Stoeckius M, Smibert P, Satija R. 2019. Comprehensive Integration of Single-Cell Data. *Cell* **177**: 1888-1902 e1821.

Wolf FA, Angerer P, Theis FJ. 2018. SCANPY: large-scale single-cell gene expression data analysis. *Genome Biol* **19**: 15.