Data Preprocessing for Single-Cell RNA-seq

Essential Steps for High-Quality Analysis

1. Cell Filtering

Remove low-quality cells, doublets, and empty droplets to ensure data integrity

2. Gene Filtering

Exclude lowly expressed genes and problematic gene categories

3. Normalization

Account for technical variations in sequencing depth and composition

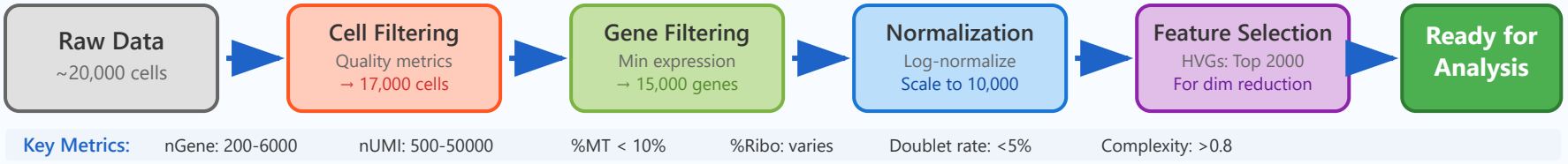
4. Imputation

Handle dropout events and technical zeros (use with caution)

5. Batch Effects

Identify and correct technical variations from sample processing

Quality Control Pipeline

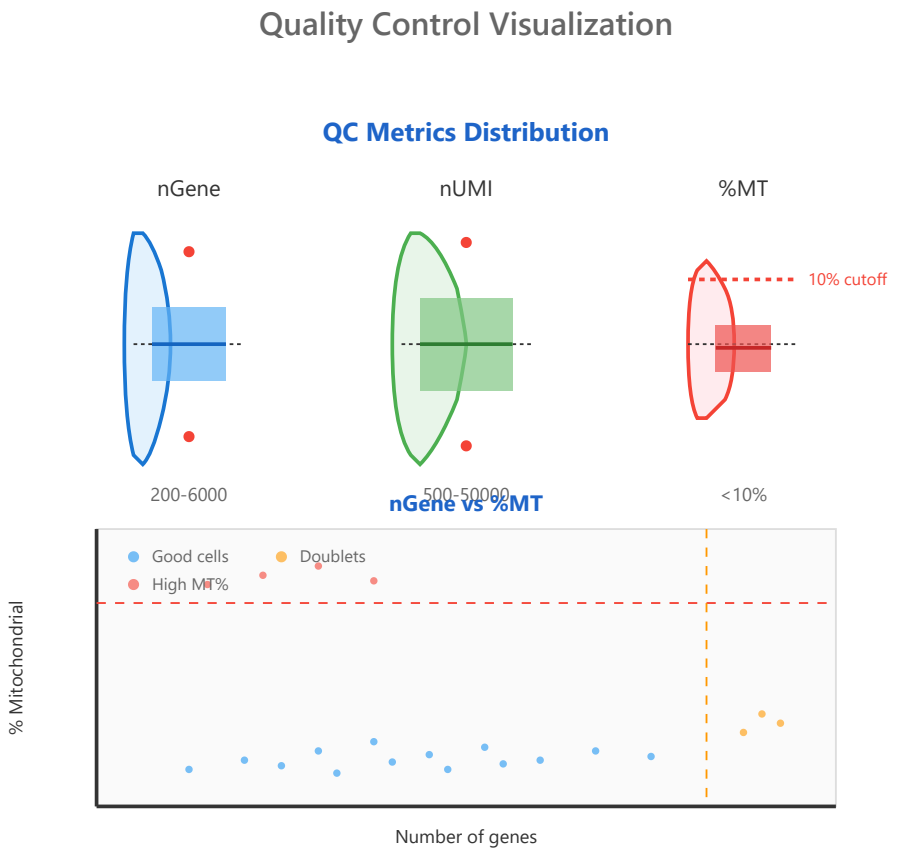


1 Cell Filtering

Cell filtering is the critical first step in single-cell RNA-seq analysis. This process removes low-quality cells, empty droplets, and doublets/multiplets that could compromise downstream analysis. High-quality cells are identified based on multiple quality control metrics that assess cellular integrity and technical artifacts.

Key Quality Metrics

- ✓ **nGene/nFeature:** Number of detected genes (typically 200-6000)
- ✓ **nUMI/nCount:** Total number of unique molecular identifiers
- ✓ **%Mitochondrial:** Percentage of reads mapping to MT genes (<10%)
- ✓ **%Ribosomal:** Percentage of reads from ribosomal genes
- ✓ **Complexity:** Library complexity (nGene/nUMI ratio)



Example Code (Scanpy/Python):

```
# Calculate QC metrics import scanpy as sc
sc.pp.calculate_qc_metrics(adata, qc_vars=['mt',
'ribo'], percent_top=None, log1p=False, inplace=True)
# Filter cells sc.pp.filter_cells(adata,
min_genes=200) sc.pp.filter_cells(adata,
max_genes=6000) adata =
adata[adata.obs['pct_counts_mt'] < 10, :]
```

⚠ **Important:** Thresholds should be dataset-specific. Always visualize distributions before setting cutoffs to avoid removing biologically relevant populations.

2 Gene Filtering

Gene filtering removes features that provide little information or could introduce noise into the analysis. This includes genes expressed in very few cells, housekeeping genes that show minimal variation, and genes from specific categories that might bias the analysis (e.g., mitochondrial, ribosomal genes for certain analyses).

Filtering Criteria

- ✓ **Minimum cells:** Remove genes expressed in <3-10 cells
- ✓ **Mitochondrial genes:** Often excluded from HVG selection

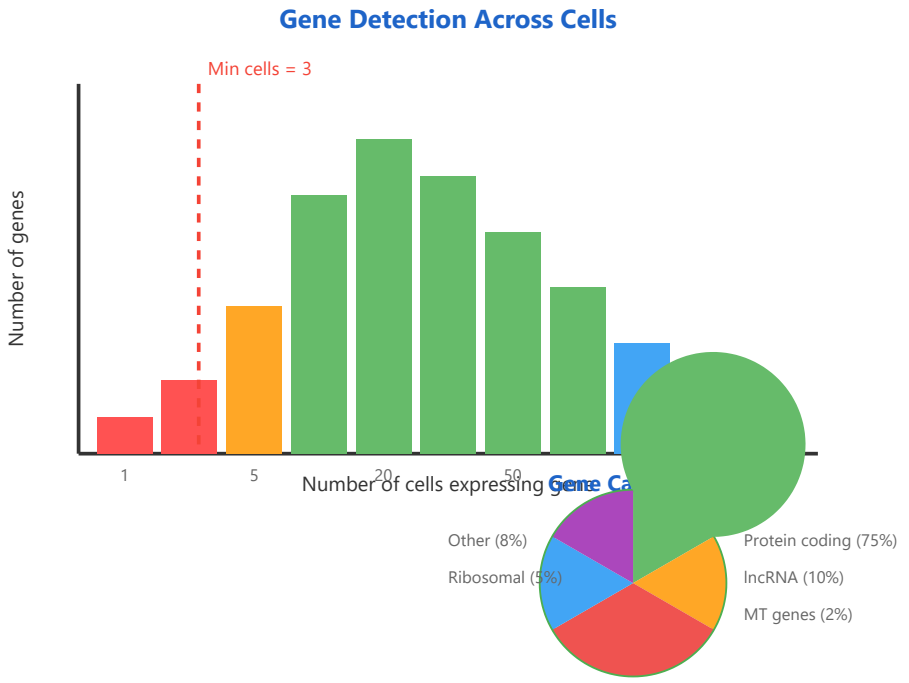
Gene Expression Distribution

- ✓ **Ribosomal genes:** May be removed for specific analyses
- ✓ **Cell cycle genes:** Can be regressed out if needed
- ✓ **Sex-specific genes:** Consider removing for certain studies

Example Code (Seurat/R):

```
# Filter genes library(Seurat) # Keep genes expressed in at least 3 cells min.cells <- 3 genes.use <- rowSums(counts > 0) >= min.cells counts <- counts[genes.use, ] # Remove MT and ribosomal genes mt.genes <- grep("^MT-", rownames(counts)) ribo.genes <- grep("^RP[SL]", rownames(counts)) counts <- counts[-c(mt.genes, ribo.genes), ]
```

💡 **Tip:** Consider keeping mitochondrial and ribosomal genes in the initial dataset for QC purposes, then exclude them during feature selection for dimensionality reduction.



3 Normalization Methods

Normalization corrects for technical differences between cells, particularly sequencing depth variations. Different normalization methods are suited for different data types and analysis goals. The choice of normalization can significantly impact downstream results, especially clustering and differential expression.

Normalization Effects

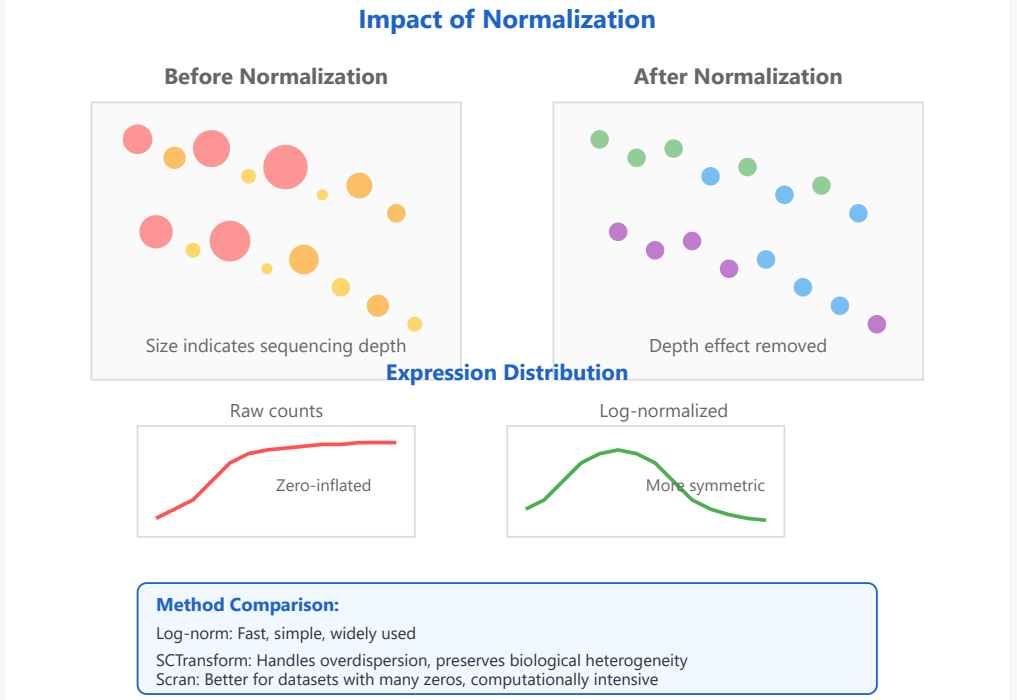
Common Methods

- ✓ **Log-normalization:** Scale to 10,000 UMIs + log transform (most common)
- ✓ **SCTransform:** Regularized negative binomial regression
- ✓ **Scran pooling:** Pool-based size factor estimation
- ✓ **TMM/DESeq2:** Methods adapted from bulk RNA-seq
- ✓ **Pearson residuals:** Variance-stabilizing transformation

Example Code (Multiple Methods):

```
# Method 1: Standard log-normalization
sc.pp.normalize_total(adata, target_sum=1e4)
sc.pp.log1p(adata) # Method 2: SCTransform (Seurat)
library(Seurat) seurat_obj <- SCTransform(seurat_obj,
vars.to.regress = "percent.mt", verbose = FALSE) #
Method 3: Scran normalization library(scran) sce <-
computeSumFactors(sce) sce <- logNormCounts(sce)
```

⚠ **Note:** Different normalization methods make different assumptions. Log-normalization assumes similar total RNA content across cells, which may not hold for all cell types.



4 Imputation Strategies

Imputation methods attempt to recover missing expression values (zeros) that may represent technical dropouts rather than true biological absence. However, imputation is controversial as it can introduce false signals and should be used cautiously, particularly for differential expression analysis.

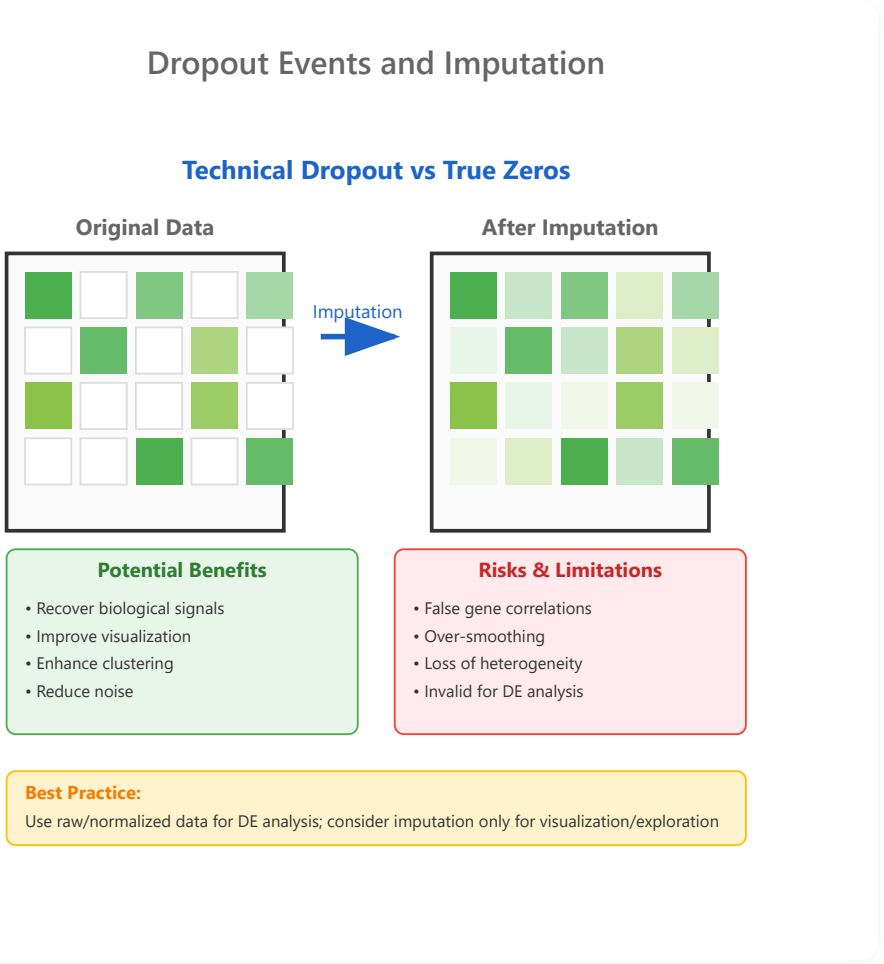
Imputation Approaches

- ✓ **MAGIC:** Markov affinity-based graph imputation
- ✓ **scImpute:** Statistical model-based imputation
- ✓ **SAVER:** Bayesian approach using gene-gene relationships
- ✓ **DCA:** Deep count autoencoder
- ✓ **No imputation:** Often the safest choice

Example Code (MAGIC):

```
import magic import scanpy as sc # Create MAGIC
operator magic_op = magic.MAGIC() # Run imputation
adata_magic = adata.copy()
magic_op.fit_transform(adata_magic.X) # Compare
before/after sc.pl.scatter(adata, x='gene1',
y='gene2', title='Original')
sc.pl.scatter(adata_magic, x='gene1', y='gene2',
title='After MAGIC')
```

⚠ **Caution:** Imputation can create false gene-gene correlations and should generally be avoided for differential expression analysis. Consider using methods designed to handle sparsity instead.



5 Batch Effect Correction

Batch effects are systematic technical variations introduced during sample processing, sequencing, or data generation. These can confound biological signals and must be carefully identified and corrected. Modern methods can integrate multiple datasets while preserving biological variation.

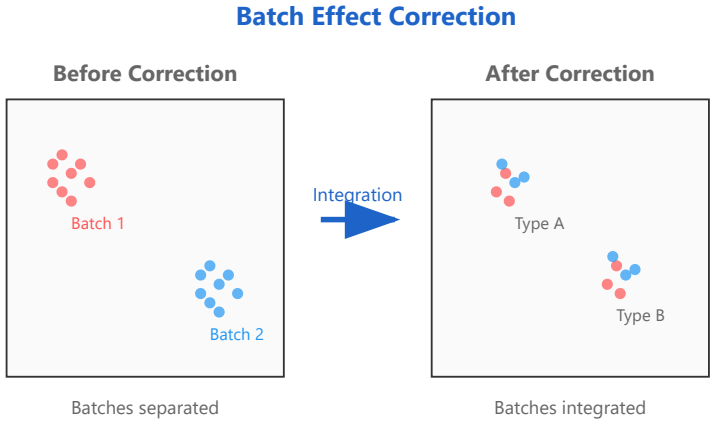
Integration Methods

- ✓ **Harmony:** Fast iterative correction in PCA space
- ✓ **Seurat Integration:** Anchor-based integration using CCA
- ✓ **scVI:** Variational autoencoder approach
- ✓ **MNN:** Mutual nearest neighbors correction
- ✓ **ComBat:** Empirical Bayes batch correction
- ✓ **LIGER:** Integrative non-negative matrix factorization

Example Code (Harmony):

```
import scanpy as sc import scanpy.external as sce # Compute PCA first sc.pp.pca(adata) # Run Harmony integration sce.pp.harmony_integrate( adata, key='batch', # batch variable in obs basis='X_pca', adjusted_basis='X_pca_harmony' ) # Use corrected
```

Batch Effect Visualization



Integration Method Comparison

Method	Speed	Scalability	Preservation
Harmony	Fast	Excellent	Good
Seurat CCA	Medium	Good	Excellent
scVI	Slow	Excellent	Excellent
ComBat	Fast	Limited	Variable

Key Consideration:
Always check biological signals are preserved post-correction using known markers

```
embeddings for UMAP sc.pp.neighbors(adata,
use_rep='X_pca_harmony') sc.tl.umap(adata)
```

💡 **Best Practice:** Always visualize data before and after batch correction. Check that biological variation (e.g., cell types) is preserved while technical variation is removed.

Processing Workflow Summary

1. **Quality Control:** Assess cell and gene quality metrics
2. **Cell Filtering:** Remove low-quality cells based on thresholds
3. **Gene Filtering:** Exclude lowly expressed and problematic genes
4. **Normalization:** Account for technical variations
5. **Feature Selection:** Identify highly variable genes
6. **Batch Correction:** Integrate multiple datasets if needed
7. **Dimensionality Reduction:** PCA, then UMAP/tSNE for visualization

> **Remember:** Each dataset is unique. Always visualize your data at each step and adjust parameters based on your specific biological system and experimental design.