UNIVERSITÀ
DI PAVIA

FACOLTÀ DI INGEGNERIA
DIPARTIMENTO DI INGEGNERIA INDUSTRIALE E
DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN BIOINGEGNERIA

TESI DI LAUREA

Progetto e sviluppo di strumenti interattivi in Orange per la correzione dei batch effect in scRNA-seq

Candidato: **Michele Gottinger**
Relatore: **Prof. Riccardo Bellazzi**

A.A. 2024/2025

Abstract

L'analisi dei dati di single-cell RNA sequencing (scRNA-seq) consente di studiare l'espressione genica a livello di singola cellula, offrendo nuove opportunità per la comprensione dell'eterogeneità cellulare in sistemi biologici complessi. Tuttavia, questi dati sono frequentemente affetti da effetti di batch, ovvero variazioni tecniche non biologiche introdotte da differenze tra esperimenti, tecnologie di sequenziamento o condizioni di laboratorio. Tali effetti possono compromettere l'interpretazione biologica dei dati e rendere difficoltosa l'integrazione di dataset provenienti da esperimenti diversi.

In questa tesi viene affrontato il problema della correzione dei batch effects, con l'obiettivo finale di estendere ed arricchire le funzionalità del software open-source Orange in ambito di analisi di dati scRNA-seq. Dopo un'analisi comparativa di diversi metodi presenti in letteratura, è stato selezionato il metodo Harmony per le sue buone prestazioni in termini di integrazione dei batch, preservazione dell'informazione biologica ed efficienza computazionale. L'algoritmo è stato implementato all'interno dell'Add-On Single Cell del software Orange mediante lo sviluppo di un nuovo widget dedicato alla correzione dei batch effects.

Inoltre, è stato progettato e implementato un secondo widget, denominato Batch Correction Evaluation, finalizzato alla valutazione quantitativa della qualità della correzione su dataset corretti o dell'impatto dei batch effects su dati non ancora corretti.

Il lavoro è stato svolto nell'ambito di un periodo di Erasmus Traineeship presso il laboratorio del Prof. Blaž Zupan, nel dipartimento di Computer and Information Science dell'Università di Lubiana, contribuendo allo sviluppo dell'ecosistema software Orange.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

Negli ultimi anni l'analisi computazionale di dati biologici ad alta dimensionalità ha assunto un ruolo sempre più centrale nella ricerca biomedica. In particolare, la tecnologia di single-cell RNA sequencing (scRNA-seq) consente di misurare l'espressione genica a livello di singola cellula, permettendo di studiare l'eterogeneità cellulare all'interno di tessuti complessi e di identificare popolazioni cellulari e stati funzionali che non risultano osservabili mediante tecniche di analisi in cui le cellule vengono analizzate congiuntamente, dette analisi bulk [1, 2].

A differenza delle tecniche di sequenziamento tradizionali, che forniscono una misura media dell'espressione genica su grandi popolazioni cellulari, l'analisi scRNA-seq permette infatti di caratterizzare la variabilità tra cellule individuali, fornendo informazioni cruciali per lo studio dello sviluppo cellulare, delle malattie e delle dinamiche dei sistemi biologici complessi [3, 1]. Questa tecnologia ha quindi trovato applicazione in numerosi ambiti della biologia e della medicina, tra cui l'immunologia, la ricerca sul cancro e lo studio dei processi di differenziamento cellulare.

Nonostante il grande potenziale di questa tecnologia, l'analisi dei dati scRNA-seq presenta numerose sfide. Una delle problematiche più rilevanti è rappresentata dai cosiddetti effetti di batch, termine con il quale si fa riferimento a variazioni tecniche non biologiche introdotte da differenze tra esperimenti, quali variazioni nei protocolli di preparazione dei campioni, nelle tecnologie di sequenziamento utilizzate, nelle condizioni di laboratorio o dovute a qualsiasi altro aspetto che non sia la pura variabilità biologica. Tali effetti possono alterare significativamente la struttura dei dati, introducendo separazioni artificiali tra cellule biologicamente simili o, al contrario, mascherando differenze biologiche reali [4, 5]. La presenza di batch effects rende, inoltre, particolarmente complessa anche l'integrazione di dataset provenienti da esperimenti differenti, una pratica sempre più comune nell'analisi di dati scRNA-seq su larga scala.

Per affrontare questo problema sono stati sviluppati diversi metodi di batch effect correction e data integration, che mirano a rimuovere la variabilità tecnica preser-

vando al contempo l'informazione biologica presente nei dati [6, 7]. Negli ultimi anni sono stati proposti numerosi approcci per la correzione dei batch effects nei dati single-cell, che spaziano da metodi statistici classici a metodi più recenti basati su tecniche di machine learning e deep learning.

Con l'aumentare della complessità logica di tali algoritmi e modelli, diventa necessario renderli fruibili anche a quel panorama di utenti con background prevalentemente biologico, ma con conoscenze informatiche limitate. Vi sono quindi software che mirano all'implementazione di questo tipo di strumenti mediante un approccio più semplificato e orientato all'utente. Tra questi si colloca Orange, una piattaforma open-source per l'analisi dei dati e il machine learning sviluppata dal team del laboratorio BioLab presso l'Università di Lubiana [8]. Il software è progettato per fornire un ambiente visuale per l'esplorazione interattiva, l'analisi e la visualizzazione di dataset complessi. Orange utilizza un'interfaccia grafica basata su workflow, in cui le possibili operazioni di analisi vengono rappresentate come widget collegabili tra loro, all'interno dei quali è possibile impostare alcuni parametri ed opzioni. Questo approccio consente agli utenti di costruire pipeline di analisi in modo intuitivo, senza richiedere necessariamente competenze avanzate di programmazione. L'obiettivo principale del software è rendere accessibili tecniche avanzate di analisi dei dati a una comunità ampia di utenti, favorendo al contempo l'esplorazione interattiva e l'utilizzo ai fini didattici, garantendo comunque riproducibilità e rigore scientifico nell'analisi [9]. Nel corso degli anni Orange è stato esteso tramite diversi Add-On dedicati a specifici ambiti applicativi. Tra questi, l'Add-On Single Cell fornisce strumenti per l'analisi di dati di single-cell RNA sequencing, includendo diverse funzionalità, ma con lacune per quanto riguarda la correzione dei batch effects, in quanto il metodo precedentemente implementato è ampiamente superato dalle alternative più moderne. Il lavoro presentato in questa tesi mira a colmare tali mancanze, contribuendo allo sviluppo dell'ambiente Orange.

L'attività è stata svolta nell'ambito di un periodo di Erasmus Traineeship presso il laboratorio BioLab, supervisionato dal Professor Blaž Zupan. Il laboratorio si trova presso la facoltà di Computer and Information Science dell'Università di Lubiana.

1.2 Obiettivi della tesi

Gli obiettivi principali del lavoro svolto sono i seguenti:

- paragonare diversi metodi di correzione dei batch effects nel contesto dell'analisi di dati scRNA-seq;
- selezionare un metodo adeguato all'integrazione nel software Orange, considerando sia le prestazioni sia i vincoli e le necessità dettati dal contesto;
- implementare il metodo scelto (Harmony) all'interno dell'Add-On Single Cell mediante lo sviluppo di un nuovo widget dedicato;
- sviluppare uno strumento per la valutazione quantitativa dell'impatto dei batch effects e dell'efficacia della correzione di tali effetti;

- applicare gli strumenti sviluppati a diversi dataset pubblici di scRNA-seq al fine di confrontarne le prestazioni con il metodo preesistente in Orange.

L'attività a me assegnata durante il periodo di Traineeship è quindi da definirsi di tipo progettuale; il mio compito è stato infatti quello di ampliare le funzionalità dell'Add-On Single Cell, calandomi nel ruolo di sviluppatore dell'ambiente Orange.

1.3 Contributi del lavoro

Nello specifico, l'attività svolta ha contribuito a colmare alcune necessità e lacune presenti nelle precedenti versioni dell'Add-On Single Cell di Orange.

L'aspetto più importante è sicuramente l'introduzione di un metodo di correzione dei batch effects, accuratamente selezionato per rispettare le molteplici esigenze correlate all'ambiente Orange. Il metodo introdotto è infatti computazionalmente leggero ed efficiente, ma anche versatile e scalabile, essendo adattato a molteplici contesti, grazie alla sua struttura logica e alla possibilità offerta dal widget di modificare appositi parametri. In precedenza non era possibile effettuare una correzione dei batch effects accurata, in quanto il metodo implementato era di vecchia concezione e limitato all'utilizzo su casistiche particolarmente semplici. Il widget Batch Effect Removal (versione preesistente in Orange) si limita infatti ad una regressione lineare globale (non locale), con cui si stima ed elimina l'effetto di batch. Tale approccio è piuttosto semplice e superato dai nuovi metodi di correzione disponibili in letteratura [10], in grado di effettuare una correzione più fine e mirata, preservando meglio l'informazione biologica ma rimuovendo l'effetto di batch.

Inoltre, il secondo widget implementato (Batch Correction Evaluation) è in grado di aiutare l'utente fornendo una valutazione quantitativa della qualità del dataset utilizzato. Nelle versioni precedenti non vi erano, infatti, strumenti che consentissero in prima battuta all'utente di comprendere quanto il dataset utilizzato fosse effettivamente affetto da batch effects e quanto fosse buono il mixing tra batch e tra tipi cellulari. Ciò dunque non permetteva, quantomeno in via quantitativa, di comprendere se fosse effettivamente necessario effettuare una correzione mediante un apposito algoritmo o se fosse sufficiente proseguire solamente con un opportuno preprocessing. Inoltre, non era possibile, dopo la correzione, avere una valutazione quantitativa della bontà di tale correzione. Con il nuovo widget progettato e implementato è ora possibile effettuare entrambe le operazioni sopra citate.

1.4 Organizzazione del lavoro di tesi

L'elaborato, come già affermato in questa fase introduttiva, si propone di descrivere dettagliatamente l'attività di ricerca e sviluppo svolta durante il periodo di tirocinio presso il laboratorio BioLab. La struttura di questa tesi ricalca fedelmente lo sviluppo cronologico e logico degli eventi affrontati durante il Traineeship. Questo approccio narrativo permette di giustificare ogni scelta progettuale come naturale conseguenza delle analisi e delle necessità emerse nelle fasi immediatamente precedenti, supportata dal confronto con professori e colleghi.

L'elaborato si articolerà nelle seguenti fasi:

- **Background:** in questo capitolo verranno fornite le informazioni necessarie a comprendere il contesto del lavoro, dalla tecnologia scRNA-seq al software Orange e alla sua filosofia. Inoltre, ci si soffermerà brevemente nel descrivere alcuni metodi di correzione dell'effetto di batch. Essi sono stati i metodi candidati, in prima battuta, all'implementazione in Orange, in quanto rappresentativi delle principali filosofie di approccio alla correzione dei batch effect. Verrà dunque introdotta la ratio alla base di ognuno dei metodi, piuttosto che aspetti particolarmente tecnici.
- **Benchmark e scelta del metodo:** il lavoro prosegue illustrando la comparazione dei metodi candidati su vari dataset pubblici. Questa fase, preliminare all'implementazione vera e propria, è servita a valutare oggettivamente e quantitativamente le performance, oltre ai vantaggi e agli svantaggi legati ai vincoli operativi di Orange, portando infine alla scelta motivata dell'algoritmo Harmony.
- **Implementazione del widget Harmony in Orange:** in questa sezione si esplorano i dettagli matematici più fini del metodo scelto, illustrando le scelte architetturali legate all'implementazione di Harmony, prima come modulo Python indipendente e successivamente come widget nativo per Orange. Si prosegue poi, nel capitolo successivo, con il confronto tra il nuovo strumento e il metodo preesistente nel software.
- **Batch Correction Evaluation Widget:** il capitolo in oggetto tratta le motivazioni pratiche che, a valle dell'implementazione di Harmony, hanno fatto emergere la necessità di progettare un secondo strumento con scopi diagnostici verso il dataset. Verranno descritti i dettagli matematici delle metriche implementate e gli aspetti architetturali legati allo sviluppo del codice di questo nuovo widget.

Capitolo 2

Background

2.1 Single-cell RNA sequencing

2.1.1 Panoramica della tecnologia

La tecnologia di single-cell RNA sequencing consente di misurare i profili di espressione genica a livello di ogni singola cellula. Ciò supera il limite delle tecniche di sequenziamento tradizionali, che forniscono solamente la media dei valori di espressione sull'insieme delle cellule analizzate (analisi bulk) [1]. A partire da una sospensione di cellule o nuclei, le singole unità vengono isolate e marcate con barcode molecolari che permettono di associare le letture di sequenziamento alla cellula di origine [11]. Dopo l'isolamento, l'RNA messaggero viene retrotrascritto in cDNA e amplificato, quindi preparato per il sequenziamento ad alta profondità, in modo da ottenere un profilo trascrittomico per ogni cellula [1, 2].

Il scRNA-seq ha trasformato lo studio dell'eterogeneità cellulare, rendendo possibile identificare sottopopolazioni cellulari, stati funzionali e traiettorie di differenziamento che rimarrebbero nascosti in dati bulk [1, 3]. Questa tecnologia trova applicazione in numerosi ambiti, tra cui lo studio dello sviluppo embrionale, la caratterizzazione di microambienti tumorali complessi, l'analisi della risposta immunitaria in condizioni fisiologiche e patologiche, lo studio delle malattie neurodegenerative (Alzheimer e Parkinson) [3, 12]. Di recente si è assistito a una rapida evoluzione delle piattaforme sperimentali, con un aumento della dimensionalità delle acquisizioni (decine o centinaia di migliaia di cellule per esperimento) e una progressiva riduzione dei costi, che ha favorito la diffusione di studi su larga scala e la necessità di strumenti computazionali dedicati [2, 13], che richiedono dunque una formazione sempre più orientata anche all'ingegneria e all'informatica.

2.1.2 Struttura dei dati

I dati di scRNA-seq sono tipicamente organizzati in una matrice di conteggi, in cui le righe rappresentano le cellule e le colonne rappresentano i geni di cui si misura l'e-

spressione; ogni elemento della matrice contiene un conteggio intero che approssima l'abbondanza degli RNA trascritti per un gene in una determinata cellula [14].

A questa matrice principale si affiancano abitualmente tabelle di metadati a livello di cellula, che includono informazioni quali il campione o il donatore di origine, la tecnologia di sequenziamento utilizzata, il lotto sperimentale, lo studio (in caso di dati integrati o studi multicentrici), eventuali annotazioni del tipo cellulare o del cluster di appartenenza ed altri attributi derivati da analisi a valle [14, 11].

Dal punto di vista computazionale, inoltre, la matrice di conteggi è spesso estremamente sparsa, poiché in ogni cellula solo una frazione dei geni risulta espressa sopra il limite di rilevazione, anche a causa del cosiddetto dropout tecnico (gene biologicamente espresso ma non rilevato a causa di limitazioni tecniche) [15]. Per facilitare le analisi esplorative, la matrice grezza viene in genere sottoposta a pre-processing, che include passi di filtraggio di cellule e geni di bassa qualità, normalizzazione dei conteggi, trasformazioni logaritmiche e selezione di geni altamente variabili [11, 12]. Sulla base di questi dati pre-processati vengono poi costruite rappresentazioni a bassa dimensionalità (e.g. tramite PCA) e grafi di similarità tra cellule, che costituiscono il punto di partenza per algoritmi di clustering, inferenza di traiettorie e metodi di correzione dei batch effects [12, 15]. In Orange sono implementati tutti i widget necessari per svolgere operazioni di filtraggio, normalizzazione e trasformazioni. Risulta invece più macchinosa la gestione delle matrici sparse, in quanto lo scheletro dell'Add-On Single Cell è stato originariamente implementato gestendo i dati come ordinarie matrici dense.

2.2 Effetti di batch nei dati scRNA-seq

2.2.1 Origine degli effetti di batch

Come introdotto in precedenza, i batch effects nei dati scRNA-seq sono variazioni non biologiche che emergono quando i dati vengono generati in esperimenti multipli condotti in momenti o condizioni diverse. Tali effetti derivano principalmente da differenze tecniche non controllate che si accumulano lungo l'intero workflow sperimentale, dalla preparazione del campione al sequenziamento, oltre all'integrazione di dataset provenienti da differenti studi o da uno studio multicentrico.

Tra le fonti più comuni si annoverano:

- **differenze nei protocolli di preparazione**: variazioni nei tempi di incubazione o efficienza degli enzimi di reverse transcription e PCR [16];
- **fluttuazioni nella qualità dei reagenti**: differenze tra lotti di enzimi, primer o barcodes molecolari che introducono bias sistematici nell'efficienza di cattura dell'mRNA [16, 4];
- **differenze tecnologiche**: piattaforme diverse (e.g. 10x Chromium e Smart-seq2, due opzioni molto comuni) o versioni aggiornate dello stesso strumento generano profili di espressione differenti, specialmente per geni a bassa espressione [4, 17];

- **variazioni biologiche controllate**: differenze tra donatori/pazienti, tessuti anatomici raccolti in momenti diversi [18, 17];
- **fattori ambientali e operatori**: temperatura, pH dei tamponi, umidità durante la manipolazione, differenze nella tecnica degli operatori di laboratorio [18];
- **differenze nel sequenziamento**: cicli di sequenziamento, profondità assegnata per batch, o rumore di lettura specifico del sequenziatore [19].

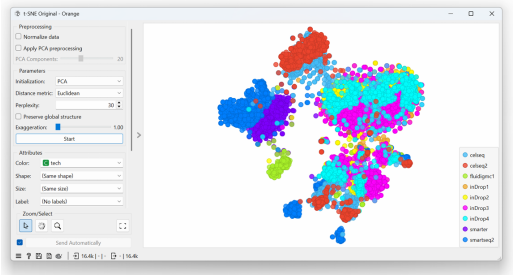
Questi effetti sono inevitabili nei moderni studi scRNA-seq su larga scala, dove è impossibile processare decine di migliaia di cellule in un singolo esperimento. Diventano particolarmente critici quando si cerca di integrare dataset provenienti da collaborazioni multicentriche o studi longitudinali, dove il numero di batch può raggiungere le decine o centinaia, con caratteristiche tecniche fortemente eterogenee [17].

2.2.2 Impatto sull'analisi scRNA-seq

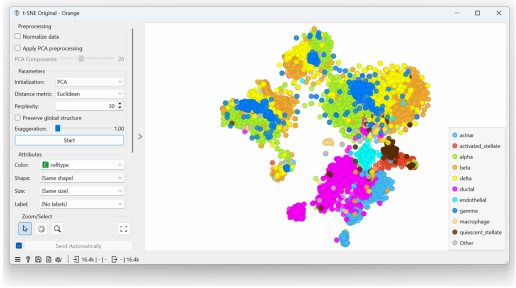
I batch effects dominano spesso la variabilità osservata nei dati scRNA-seq, alterando il segnale biologico e compromettendo le analisi a valle [17]. Il loro impatto si può manifestare lungo più fasi del workflow di una tipica analisi scRNA-seq standard:

- **clustering delle cellule**: algoritmi come Leiden o Louvain (clustering non supervisionato) [20] assegnano cellule a cluster basandosi principalmente sull'appartenenza al batch piuttosto che al tipo cellulare biologico [21]. Cellule di tipologia simile, ma provenienti da batch diversi, vengono quindi separate artificialmente in sottogruppi batch-specifici, portando ad una potenziale sovrastima del numero di tipi cellulare e identificazione di geni marker batch-specifici invece che biologici [21]. Inoltre, questo tipo di clustering non supervisionato viene utilizzato anche per la scoperta di nuovi tipi cellulari, che risulterebbero così potenzialmente oscurati (specialmente se appartenenti a popolazioni cellulari rare distribuite su più batch) [17];
- **visualizzazioni in embedding 2D/3D (t-SNE, UMAP)**: nella pratica si fa spesso uso di rappresentazioni dei dati a dimensionalità ridotta per una visualizzazione più intuitiva e immediata di essi. In metodi basati su trasformazioni di variabili in uno spazio latente, o spazio di embedding, come t-SNE e UMAP, i batch effects si manifestano come raggruppamenti evidenti per tipologia di batch, mascherando la struttura biologica [22] (Figura 2.1);
- **integrazione di dataset multipli**: la distanza tra cellule di batch diversi è influenzata anche dalla variabilità tecnica, impedendo l'allineamento corretto degli spazi di embedding [5];
- **analisi differenziale di espressione genica (DGE)**: geni con espressione variabile tra batch (per cause tecniche) vengono erroneamente identificati come geni marker di tipi cellulari [22, 23];

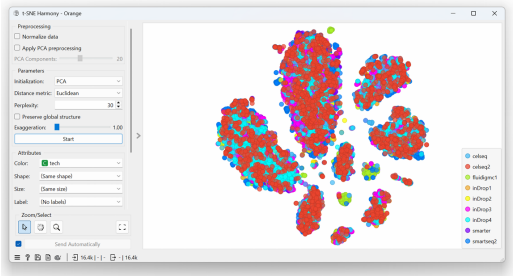
- **traiettorie pseudotemporali**: le traiettorie vengono distorte o spezzate ai confini tra batch, poiché la similarità cellulare è alterata dagli errori tecnici [5].



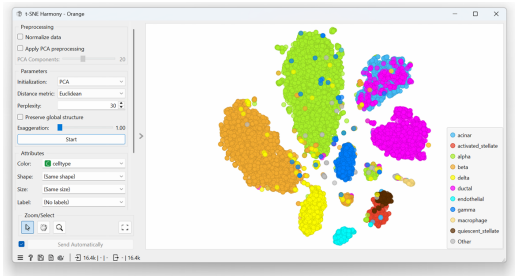
(a) Rappresentazione t-SNE dei dati colorati per batch.



(b) Rappresentazione t-SNE dei dati colorati per tipo cellulare.



(c) Rappresentazione t-SNE dei dati corretti da Harmony colorati per batch.



(d) Rappresentazione t-SNE dei dati corretti da Harmony colorati per tipo cellulare.

Figura 2.1: Rappresentazione grafica in Orange dei dati originali e corretti con il metodo Harmony. Si può notare come i raggruppamenti dei dati non corretti nello spazio t-SNE sono fortemente guidati dal batch di appartenenza.

Una serie di studi di benchmark ha mostrato che, senza correzione, fino all'80-90% della variabilità principale (prime componenti della PCA) può essere spiegata da effetti di batch, mentre la variabilità biologica (tipo cellulare, condizione sperimentale) emerge solo nelle componenti successive [24, 25]. I batch effects trasformano quindi l'eterogeneità cellulare biologica reale, obiettivo primario del scRNA-seq, in una struttura di correlazione artefatta influenzata (o, nei casi peggiori, dominata) da variabilità tecnica, rendendo indispensabili metodi di correzione prima di qualsiasi analisi esplorativa o inferenziale [17].

2.3 Metodi di batch correction

2.3.1 Benchmark presenti in letteratura

Per la valutazione del metodo di batch effect correction da implementare in Orange ci si è inizialmente appoggiati a benchmark preesistenti in letteratura. Nello specifico,

ci si è principalmente affidati al benchmark di Open Problems [10] (Figura 2.2).

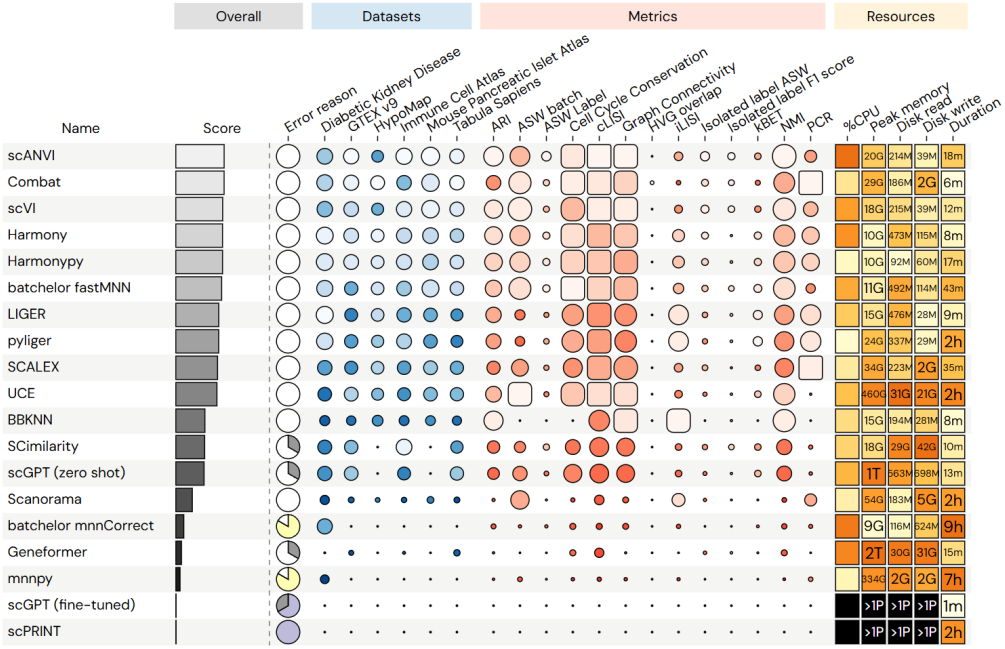


Figura 2.2: Tabella riassuntiva del benchmark di Open Problems riguardo i principali metodi di correzione degli effetti di batch.

Si è deciso di investigare, calandoli nel contesto di Orange, quattro metodi tra i più promettenti, che avessero però fondamenti teorici radicalmente differenti. La scelta è dunque ricaduta su ComBat (metodo statistico), Harmony (machine learning), scANVI (deep learning variational autoencoder) e scGPT (transformer pre-trained zero-shot).

Nel benchmark Open Problems è possibile osservare le prestazioni di oltre 50 metodi su metriche multiple (iLISI, cLISI, kBET), così come le risorse hardware e temporali richieste. I risultati sui metodi selezionati mostrano che Harmony eccelle su task di integrazione semplice-media con runtime ridotti (<1 minuto) anche su dataset di dimensioni medio-grandi (>100k cellule). Inoltre, scANVI performa bene su task complessi, ma richiede ingenti risorse computazionali. ComBat invece è robusto per dataset piccoli, ma non è efficiente per dataset di grandi dimensioni. Infine, scGPT (in modalità zero-shot, senza quindi affinare l'addestramento) è promettente ma ancora instabile su dataset eterogenei [10].

Nei seguenti paragrafi è riportata una breve descrizione dei principi su cui si fondano tali metodi, utile a comprendere l'approccio di selezione adottato, sia per quanto riguarda la scelta dei candidati, sia per ciò che concerne la designazione del metodo vincitore del confronto.

2.3.2 ComBat

ComBat [26], il metodo di concezione più datata tra quelli analizzati, assume che i batch effects consistano in correzioni di posizione e scala additive, uniformi lungo tutte le feature geniche. L'espressione osservata viene modellizzata come:

$$Y_{ijg} = \alpha_g + X_{ij}\beta_g + \gamma_{bg} + \delta_{bg} \cdot \epsilon_{ijg} \tag{2.1}$$

dove γ_{bg} rappresenta lo scostamento sistematico medio (effetto additivo) e δ_{bg} il fattore di scala della varianza, stimati per ogni coppia batch b e gene g, assumendo che l'effetto di batch si manifesti in modo uniforme lungo tutto lo spazio genico. Infine, il termine ϵ_{ijg} rappresenta l'errore residuo per la cellula i, il gene g e il batch j: esso modella la naturale variabilità e il rumore di fondo dell'espressione genica, che subiscono un'alterazione moltiplicativa proprio a causa del parametro δ_{bg} del batch.

Questo metodo è veloce, non richiede embedding e quindi preserva la struttura matriciale originaria del dataset ($\#cellule * \#geni$), utile per alcuni tipi di analisi downstream in scRNA-seq. Tuttavia, la correzione globale non gestisce bene l'eterogeneità dei batch o le sovrapposizioni con l'informazione biologica [6].

2.3.3 Harmony

Harmony [27] opera su embedding PCA (tipicamente 20-50 PC) e corregge gli effetti di batch tramite un approccio localmente adattivo basato su soft clustering iterativo. A differenza dei metodi tradizionali che applicano una traslazione globale (assumendo che il batch effect alteri tutte le cellule allo stesso modo), Harmony riconosce che il rumore tecnico può colpire regioni diverse dello spazio dei dati in direzioni e con intensità differenti.

Per farlo, l'algoritmo divide prima le cellule in K cluster non supervisionati, mediante un approccio soft-clustering basato su k-means. Successivamente, stima gli effetti di batch localmente all'interno di ciascun cluster mediante una regressione lineare. Ciò consente di correggere l'effetto di batch in modo ottimale, anche quando si manifesta con entità differente nello spazio dei dati. Per fare un esempio pratico semplificato: si supponga che una differenza di protocollo in un batch abbia alterato fortemente il profilo trascrizionale di una specifica sottopopolazione (e.g. linfociti T in un particolare stato di attivazione, mappati nel Cluster 1), lasciando invece inalterate o meno colpite altre sottopopolazioni (e.g. linfociti T a riposo, nel Cluster 2, e macrofagi, nel Cluster 3). In questo scenario, la regressione calcolerà un vettore di correzione ampio per il Cluster 1, lasciando invece pressoché invariati Cluster 2 e Cluster 3.

L'embedding viene quindi aggiornato in un processo iterativo, sottraendo il contributo tecnico, specifico del batch, calcolato per ogni cellula come somma pesata delle correzioni dei singoli cluster, tramite la seguente regola matematica:

$$Z'_i = Z_i - \sum_k R_{ki} W_k^\top \phi_i \tag{2.2}$$

dove R_{ki} è il grado di appartenenza (probabilità soft) della cellula i al cluster k, W_k rappresenta la matrice dei coefficienti di regressione batch-specifici calcolati per quel determinato cluster, e ϕ_i è il vettore one-hot che identifica il batch di origine della cellula i.

Grazie a queste operazioni su matrici a bassa dimensionalità, il metodo risulta computazionalmente molto veloce (richiedendo da pochi secondi ad alcuni minuti). Inoltre, proprio in virtù della sua correzione mirata e locale, si conferma come uno dei modelli più robusti all'overcorrection, preservando l'informazione biologica sottostante [10].

2.3.4 scANVI

Il metodo scANVI [28] rappresenta un'estensione diretta di scVI (Single-cell Variational Inference). Quest'ultimo affronta l'integrazione dei dati e la rimozione dei batch effect adottando un approccio basato sul Deep Learning, nello specifico implementando un Variational Autoencoder (VAE). In questa rete neurale generativa, lo strato di partenza (encoder) riceve in input l'espressione genica di una singola cellula e la comprime in uno spazio latente a dimensionalità ridotta; successivamente, il decoder tenta di ricostruire l'espressione genica originale a partire da tale rappresentazione compressa.

La peculiarità fondamentale del metodo per la correzione del bias tecnico risiede nella gestione dell'informazione di batch: quest'ultima viene codificata (tramite vettore one-hot) e concatenata al vettore dei dati sia all'ingresso dell'encoder che all'ingresso del decoder. Agendo come covariata per la ricostruzione del dato originario, la rete è disincentivata dal memorizzare le differenze tecniche all'interno dello spazio latente, poiché tale informazione è già esplicitamente e separatamente fornita al modello. Di conseguenza, la rete impara a confinare la variabilità del batch in questa covariata dedicata, producendo coordinate latenti depurate dal rumore tecnico e rappresentative della sola eterogeneità biologica.

A partire da queste fondamenta, scANVI (Single-Cell ANnotation using Variational Inference) estende l'architettura di scVI, trasformandola in un modello semi-supervisionato. L'algoritmo incorpora infatti, oltre all'informazione di batch, anche il tipo cellulare come informazione aggiuntiva, qualora le annotazioni siano disponibili nel dataset. A differenza dell'informazione di batch, che viene trattata come covariata di "disturbo" da ignorare nella rappresentazione compressa, il tipo cellulare assume un ruolo profondamente diverso. Esso viene utilizzato per modificare la distribuzione a priori dello spazio latente. Invece di proiettare tutte le cellule in un'unica distribuzione omogenea, scANVI modella lo spazio latente come un Gaussian Mixture Model (GMM), in cui ogni tipo cellulare noto è associato a una specifica distribuzione Gaussiana.

Durante la fase di addestramento, la funzione di costo impone che la rappresentazione latente di una cellula annotata si allinei alla distribuzione assegnata al proprio tipo cellulare. Questo meccanismo forza l'encoder a posizionare vicine tra loro cellule dello stesso tipo provenienti da esperimenti diversi, creando dei centri di riferimento, dipendenti dalla biologia, che resistono alla rimozione del rumore tecnico. Per le cellule originariamente prive di etichetta, l'algoritmo inferisce il tipo cellulare più probabile in base alla loro vicinanza spaziale a tali centri, realizzando in un unico passaggio sia l'integrazione robusta dei dati sia la classificazione automatica.

2.3.5 scGPT

scGPT [29] (Single-Cell Generative Pre-trained Transformer) è un modello basato sull'architettura Transformer, precedentemente allenato su un vasto atlante di cellule (1 miliardo di cellule da 33 specie diverse nella versione multi-specie, 1 milione nella versione di atlante umano). L'intuizione alla base di questo approccio è trattare l'espressione genica di una cellula come se fosse una frase testuale, in cui i singoli geni fungono da parole, applicando sostanzialmente lo stesso paradigma dei più noti modelli di NLP (Natural Language Processing). Per correggere i batch effect utilizza il modello già allenato adattandolo ai nuovi dati; in questo processo preserva perfettamente l'informazione biologica tramite i meccanismi di attention, che non analizzano i geni singolarmente, ma ne valutano le complesse interazioni reciproche.

Il punto di forza di questo modello è sicuramente la gestione di dataset complessi, come quelli multi-specie, ma anch'esso risulta computazionalmente piuttosto oneroso. Questo onere è legato alle decine di milioni di parametri [29] ed alla complessità di calcolo che cresce in modo quadratico all'aumentare del numero di geni analizzati.

2.4 Ambiente Orange

2.4.1 Approccio e obiettivi

Il software open-source Orange [8] (Figura 2.3) nasce dall'idea di rendere accessibili i metodi di machine learning (e, successivamente, deep learning ed altri modelli AI) a utenti non esperti in programmazione, con un focus dedicato, in particolare, al mondo della biologia. L'idea di fondo, come introdotto nei paragrafi precedenti, è quella di sviluppare degli strumenti con cui anche chi non ha un background di competenze di programmazione è in grado di interagire, esplorare, manipolare e visualizzare i dati a disposizione mediante tecniche che tale tipologia di utenti non avrebbe avuto modo di comprendere altrimenti. L'obiettivo di Orange è anche quello di fornire uno strumento utile ai fini didattici; esistono infatti workshop e corsi, organizzati talvolta dagli autori stessi, in cui Orange, grazie alla sua semplicità di utilizzo, funge da strumento cardine per un primo approccio alla comprensione di strumenti complessi, legati al mondo della statistica e del machine learning.

In particolare, nel recente periodo si può osservare una crescita quasi frenetica delle nuove applicazioni biologiche che comprendono l'utilizzo di questi metodi complessi, perlomeno per chi non ha competenze informatiche e matematiche avanzate. Risulta

dunque particolarmente innovativo poter disporre di un software che permetta di allenare ed avvicinare i biologi al mondo della data science. Orange funge quindi da ponte per addolcire la curva di apprendimento e smussare quella barriera che si frappone tra il biologo e i suoi dati, permettendo un accesso facilitato ai dati ed alla loro esplorazione, nonché consentendo di acquisire familiarità con gli strumenti utilizzati [9]. Un concetto fondamentale che gli sviluppatori di Orange mettono in primo piano, inoltre, è quello dell'interattività. Come verrà illustrato nei paragrafi successivi, ogni mezzo fornito all'utente avrà un approccio interattivo, così da poter coinvolgere l'utente stesso, consentendogli di essere parte integrante dell'attività che sta svolgendo e di non percepire davanti a sé una rappresentazione fredda e statica dei dati.

Figura 2.3: Logo ufficiale di Orange.

2.4.2 Organizzazione di Orange: Add-On e Widget

Il software è costituito da un nucleo comune di funzioni preinstallate e da numerosi Add-On aggiuntivi, i quali racchiudono funzioni legate ad ambiti particolarmente specifici, che richiedono strumenti dedicati (uno di questi è proprio l'Add-On Single Cell). Le funzioni necessarie all'analisi, manipolazione e rappresentazione dei dati sono racchiuse in elementi base, chiamati widget. Ogni widget implementa un aspetto diverso dell'analisi, dal più semplice al più complesso. Un esempio oltremodo banale può essere la combinazione dei widget `File` e `Data Table`, i quali permettono rispettivamente di leggere dati da un file (sotto forma di matrici, con diversi formati di file disponibili) e di stampare a schermo in un'apposita tabella l'input letto. Se connessi, dunque, i widget parlano tra di loro con un linguaggio semplice: collego un canale di output del widget di partenza ad un canale di input del widget di arrivo (Figura 2.4).

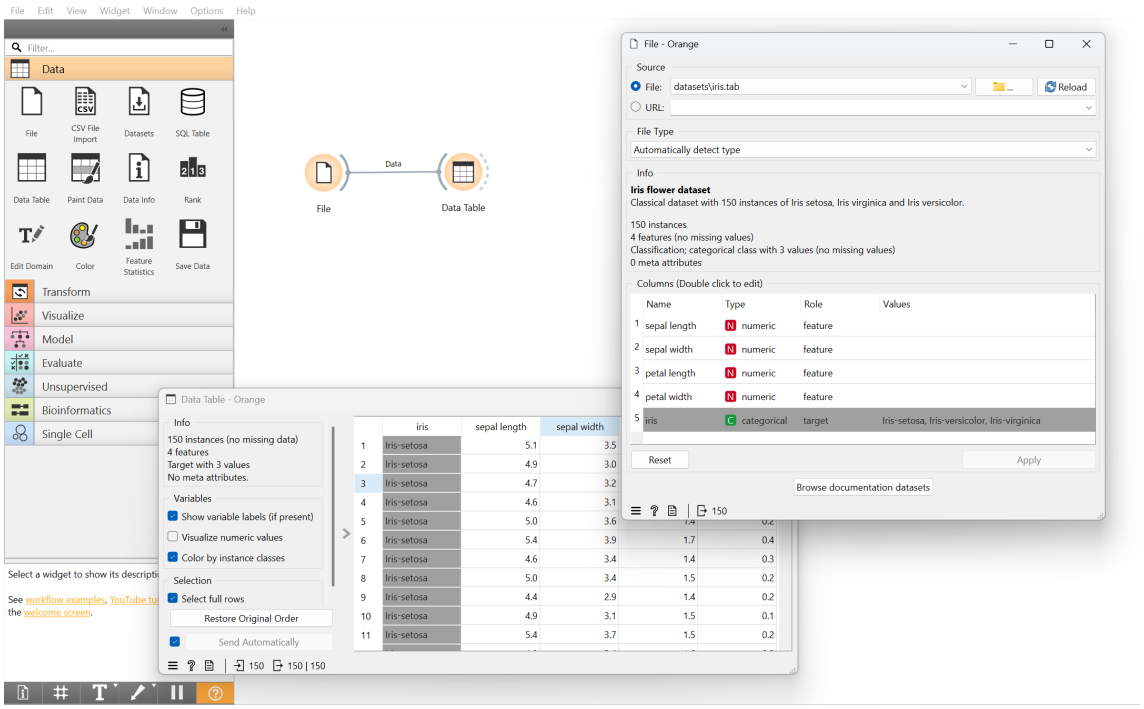


Figura 2.4: Interfaccia di Orange con a sinistra i pacchetti di funzioni disponibili e, nella parte centrale, l'interfaccia dove l'utente inserisce, connette tra loro ed interagisce con i widget. Sono mostrate, inoltre, le interfacce grafiche dei widget `File` e `Data Table` a seguito del collegamento tra essi e del caricamento di un dataset di esempio.

Il flusso di lavoro si articola quindi in un reticolo di widget connessi, che sono in grado di comunicare con gli altri widget nel reticolo. Orange ha quindi una logica intuitiva e lineare, in cui il flusso di lavoro viene costruito passo dopo passo. Questo aspetto è enfatizzato anche dal fatto che Orange esegue il codice, per citare gli autori, "on-the-fly": ogni volta che l'utente connette un nuovo widget, il flusso di lavoro viene eseguito, così da consentire un'analisi interattiva, in tempo reale, in cui l'utilizzatore è in grado di accorgersi ed immergersi immediatamente in ciò che sta succedendo ai dati. Inoltre, i widget di rappresentazione dei dati sono spesso interattivi. Essi offrono infatti la possibilità di selezionare parte delle righe o delle colonne di una tabella, nuvole di punti nei plot o barre specifiche nei barplot, per citare alcuni esempi, e tali selezioni influiscono istantaneamente sull'output dei widget stessi, che deve essere dunque in grado di variare dinamicamente (Figura 2.5 e Figura 2.6).

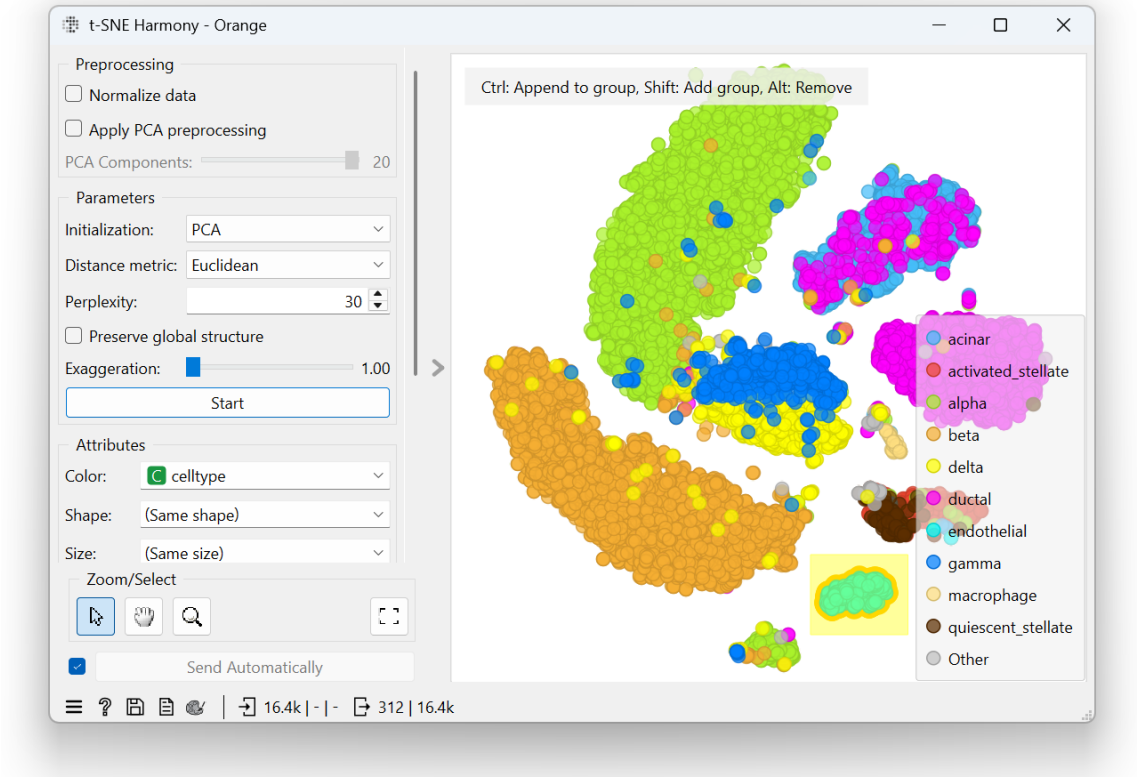


Figura 2.5: Esempio di utilizzo interattivo del widget `t-SNE`, connesso a `Data Table`. Nella figura si può notare come Orange offra la possibilità di selezionare con il mouse e inviare come output parte dei punti presenti nel grafico (rettangolo colorato di giallo).

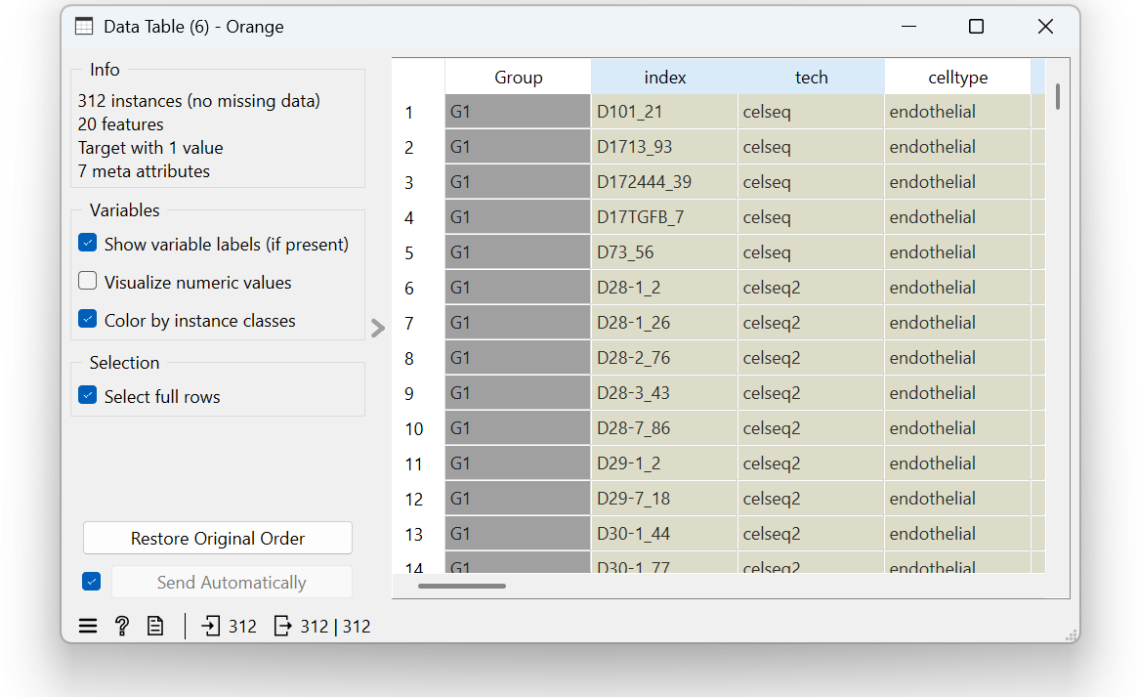


Figura 2.6: Visualizzazione mediante `Data Table` dei punti selezionati nel grafico del widget `t-SNE` in Figura 2.5.

Il nucleo di widget base di Orange consiste in funzioni di gestione e trasformazione dei dati, metodi di rappresentazione e visualizzazione (plot, tabelle, mappe a dimensionalità ridotta) e modelli di machine learning. Tale scheletro consente quindi di effettuare analisi di dati provenienti da qualsiasi contesto, non limitandosi per forza alle applicazioni biologiche. Orange si basa, inoltre, sul concetto di riproducibilità. Ogni progetto può essere infatti salvato e riprodotto fedelmente in un secondo momento. Inoltre, vi è la possibilità di creare pipeline adatte ad un utilizzo ripetuto, con dataset differenti. La grafica è chiara ed il flusso di lavoro è sempre mostrato a schermo in maniera evidente, con l'idea di mettere l'utente nella condizione di comprendere sempre ciò che sta avvenendo.

2.4.3 Add-On Single Cell

L'Add-On Single Cell nasce dall'esigenza di formare ed avvicinare biologi competenti nel panorama del sequenziamento RNA a singola cellula ai processi di manipolazione dei dati che fanno uso di strumenti di machine learning e data science [9]. L'implementazione di widget legati al mondo del scRNA vuole anche essere di aiuto, come sottolineato nei principi generici su cui si basa Orange, ma ancora più importante per quanto riguarda gli Add-On aggiuntivi, nell'organizzazione di corsi e workshop di avvicinamento alla materia, piuttosto che per un vero e proprio utilizzo professionale (pipeline logicamente più efficienti in Python o R). Principalmente, il pacchetto dedicato alla gestione di dati scRNA offre funzioni specifiche per il caricamento di dati (matrice di espressione genica, compatibile in diversi formati), preprocessing (dropout gene selection, normalizzazioni, selezione di cellule e geni mediante molteplici approcci), correzione degli effetti di batch e individuazione di marker genes (2.7).

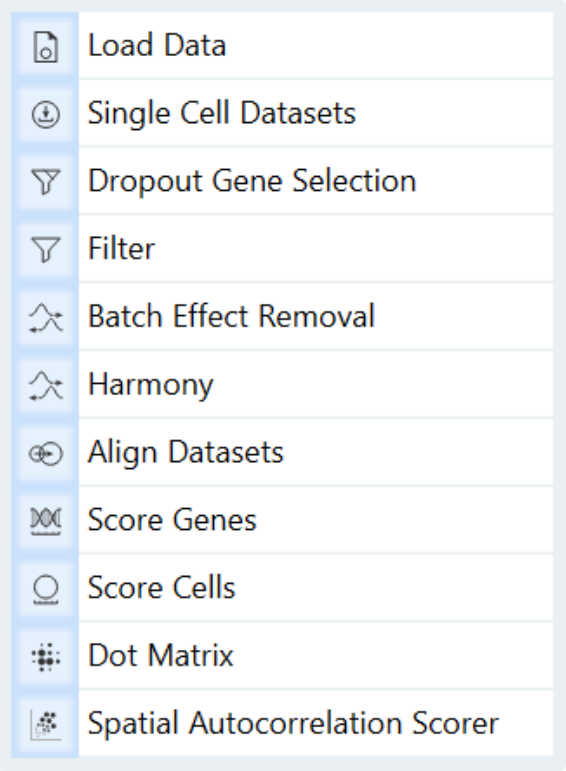


Figura 2.7: Pacchetto di funzioni presenti nell'Add-On Single Cell.

In particolare, il pacchetto Single Cell in Orange, come indicato dagli autori [9], è stato sviluppato per:

- workshop giornalieri (dalle 3 alle 10 ore);
- workshop che coprono la maggior parte degli argomenti legati all'analisi scRNA (e.g. preprocessing dedicato, clustering, DGE);
- lezioni "hands-on";
- ispirare i discenti mostrando risultati visuali, evidenziando le potenzialità di un approccio algoritmico all'analisi dei dati;
- qualsiasi altra applicazione in cui, almeno inizialmente, si vuole porre l'attenzione sull'intuizione biologica piuttosto che su quella matematica, statistica e informatica.

2.4.4 Architettura tecnica e motore dei widget

Nei paragrafi precedenti si è discusso di Orange dal punto di vista dell'utilizzatore e della filosofia alla base dello sviluppo di tale software. Tuttavia, è logico immaginare che lo scheletro sottostante l'elegante interfaccia richieda invece uno sviluppo, in termini di codice, ponderato e bene organizzato. Infatti, il motore dei widget di Orange, che ne consente l'utilizzo interattivo e modulare, è interamente basato sul linguaggio Python. In questo ecosistema, ogni widget è una porzione di codice

dinamica, progettata per gestire flussi di dati in ingresso e in uscita, reagendo in tempo reale alle interazioni dell'utente attraverso l'interfaccia grafica dedicata.

Organizzazione modulare dell'Add-on

Come sottolineato in precedenza, l'estensione delle funzionalità di Orange avviene tramite una struttura a pacchetti definiti Add-On. Nell'organizzazione del repository di un Add-On, così come per il nucleo di funzioni base di Orange, il codice è separato secondo principi legati alla programmazione modulare:

- **nucleo algoritmico (preprocess):** in questa sezione risiede la logica computazionale pura. Gli algoritmi vengono implementati utilizzando librerie ad alte prestazioni, come `NumPy`, evitando dipendenze grafiche per garantire che il calcolo sia separato dalla sua rappresentazione e venga eseguito in maniera fluida. Viene quindi creata una sottocartella in cui sono presenti tutti i file di tipo "preprocess", contenenti funzioni cardine per i widget veri e propri (nel caso di Harmony, si vedano le appendici A.2 e A.1);
- **livello visuale (widget):** contiene i file con il codice che dà vita a ciascun widget. Questi ultimi includono delle classi che definiscono il comportamento dei widget all'interno del flusso di lavoro. Ogni file in questa cartella funge da mediatore tra l'utente e l'algoritmo, definendo l'interfaccia grafica del widget (menù, tabelle, grafici), la dinamicità (callback, commit), gli input e gli output (nel caso di Harmony, si veda l'appendice A.3 o, nel caso di Batch Correction Evaluation, appendice B.1);
- **risorse e test:** la struttura include, inoltre, cartelle dedicate alle icone, che definiscono l'identità visiva del widget, e ai test automatici, essenziali per validare la stabilità del software durante l'integrazione di nuove funzionalità.

Anatomia e ciclo di vita di un Widget

Dal punto di vista dello sviluppo del codice vero e proprio, ogni widget è una classe che eredita dalla classe base `OWWidget` (disponibile nella principale libreria di Orange). La loro struttura segue uno schema pressoché standardizzato, che ne garantisce l'interoperabilità:

- **canali di comunicazione (Input/Output):** un widget comunica attraverso slot tipizzati. Gli input ricevono i dati dai nodi precedenti, mentre gli output trasmettono i risultati elaborati ai nodi successivi del flusso di lavoro;
- **interfaccia Grafica (GUI):** la costruzione della UI avviene nel costruttore della classe utilizzando il framework `Qt` e gli appositi moduli di astrazione (`Orange.widgets.gui`) che Orange mette a disposizione (si approfondirà nel paragrafo successivo);
- **persistenza e impostazioni (Settings):** il software permette di salvare lo stato del widget (parametri scelti dall'utente) attraverso i cosiddetti `Setting`. I `ContextSettings` permettono inoltre di memorizzare configurazioni specifiche per diversi dataset, garantendo che le scelte dell'utente non vadano perse

al riavvio del software. Ciò richiama il concetto di riproducibilità del flusso di lavoro, citato in precedenza;

- **meccanismo di esecuzione (Commit):** la logica del widget è generalmente racchiusa in una funzione di commit. Questa può essere attivata manualmente tramite un pulsante "Apply", visibile sull'interfaccia grafica che l'utente finale vede a schermo, o automaticamente ("Auto-commit"), permettendo una risposta fluida e interattiva: ogni volta che un parametro cambia, il widget ricalcola i dati e aggiorna istantaneamente tutto il workflow a valle (anche questo concetto di esecuzione in tempo reale è stato precedentemente introdotto).

Ereditarietà e integrazione con le librerie principali

Come evidenziato dall'analisi della struttura del codice, Orange non è semplicemente un aggregato di script isolati, ma un ecosistema software basato su un'architettura orientata agli oggetti. Lo sviluppatore di un Add-On non opera quindi nel vuoto, ma sfrutta un meccanismo di ereditarietà, che permette di riutilizzare strumenti comuni a tutta la piattaforma, garantendo coerenza funzionale ed estetica. Tra i concetti fondamentali su cui si basa tale architettura troviamo:

- **il nucleo centrale di Orange:** dalla libreria principale del software (`Orange`) vengono importati i componenti fondamentali che costituiscono l'ossatura di ogni widget. La classe `OWWidget`, come citato precedentemente, funge da "classe madre": ereditando da essa, il nuovo widget riceve nativamente la capacità di comparire nel canvas, di gestire il salvataggio automatico dello stato e di interagire con le aree predefinite della finestra, come la *controlArea* (la barra laterale dei parametri nella GUI del widget) e la *mainArea* (l'area dedicata alla visualizzazione di dati e/o informazioni);
- **astrazione della GUI e framework Qt:** sebbene l'interfaccia si appoggi al framework `Qt` (tramite il wrapper `AnyQt`), Orange fornisce un ulteriore livello di astrazione mediante il modulo `Orange.widgets.gui`. Questo strumento permette allo sviluppatore di definire elementi grafici complessi, come spin boxes, check boxes o pulsanti di auto-commit, con poche righe di codice, legandoli direttamente alle variabili di impostazione (Settings). Tale approccio garantisce che ogni widget rispetti gli standard di "visual programming" e interattività che caratterizzano il software Orange;
- **la struttura dati (`Orange.data.Table`):** un concetto fondamentale per l'interoperabilità dei widget è l'utilizzo dell'oggetto `Table` come unità universale per lo scambio di informazioni tra essi. Ogni widget riceve e trasmette dati incapsulati in questa struttura, che, per fare un esempio connesso a quanto trattato in questo elaborato, non contiene solo la matrice numerica delle espressioni geniche, bensì anche il dominio del dataset. Il dominio organizza i metadati in variabili di classe, attributi continui e variabili "meta" (come i nomi dei geni o le etichette dei batch), permettendo a widget diversi di comprendere la natura biologica del dato senza dover riesaminare l'intero file originale.

In definitiva, questo sistema di ereditarietà trasforma lo sviluppo di un nuovo strumento di analisi in un processo di estensione, più rapido, coerente ed efficace. Lo sviluppatore personalizza le capacità del nucleo centrale di Orange, concentrandosi principalmente sulla logica del nuovo algoritmo o delle nuove funzioni da implementare e delegando al nucleo preesistente la gestione della complessa macchina che sta alla base dell'interfaccia utente.

Per concludere, l'intera parentesi legata all'aspetto tecnico che risiede dietro le quinte del funzionamento di Orange, è da sottolineare nuovamente come questa architettura permetta di trasformare una pipeline di analisi dati, intrinsecamente complessa e statica, in un ambiente di esplorazione visuale, dove l'utente può interrogare i dati manipolando direttamente gli oggetti grafici. L'obiettivo dello sviluppatore diventa quindi quello di convertire i concetti che fungono da principi su cui si basa Orange in concrete porzioni di codice e scelte progettuali.

Capitolo 3

Benchmark e scelta del metodo

Dopo aver introdotto il contesto, sia in termini di panorama scientifico (analisi scRNA-seq), sia tecnico (architettura di Orange), in questo capitolo si entrerà nella narrazione vera e propria dell'attività svolta. Verrà illustrato il processo di scelta del metodo di correzione degli effetti di batch più adatto all'implementazione in Orange. Si noti che le scelte effettuate sono strettamente dipendenti dal contesto di lavoro e differiscono di conseguenza da quelle che avrebbe potuto fare un ricercatore atto a cimentarsi in un'analisi scRNA-seq al di fuori del contesto di Orange (si vedano le motivazioni dello sviluppo dell'Add-On Single Cell nel capitolo precedente).

3.1 Dataset utilizzati

Per valutare e confrontare i metodi di batch effect correction in diversi scenari biologici e tecnici sono stati utilizzati 8 dataset pubblici di scRNA-seq, selezionati da benchmark recenti in letteratura e da studi dedicati alla correzione dei batch effects. I dataset coprono tessuti umani e murini, differenti condizioni di sviluppo, esperimenti di mixing controllato e integrazione multimodale, con numerosità (sia in termini di batch che di cellule e geni) e complessità di batch variabili. Di seguito se ne riportano le principali caratteristiche, con le rispettive sigle utilizzate da qui in avanti per fare riferimento ad essi. Si osservi che il numero di dataset raccolti è limitato, principalmente per una questione di qualità dei dataset pubblici (di dimensione e interesse consoni al contesto in cui occorre lavorare, ovvero Orange), che spesso risultavano inconsistenti o incompleti.

HP – Human Pancreas Il dataset HP è un dataset ampiamente utilizzato nei benchmark di integrazione, originariamente aggregato e pre-processato nello studio "Benchmarking atlas-level data integration in single-cell genomics" di Luecken et al. [7]. HP combina dati di pancreas umano provenienti da diversi studi e tecnologie (Baron, Muraro, Segerstolpe, Xin, Wang, Lawlor, Grün), armonizzati in un unico oggetto. Il dataset contiene 16 382 cellule e 19 093 geni, suddivisi in 9 batch corrispondenti a differenti protocolli sperimentali e sorgenti di dati. I batch effects

sono quindi legati a differenze di tecnologia di sequenziamento e laboratorio. Nella versione utilizzata in questo lavoro i dati sono già normalizzati e log-trasformati [7].

RE – Mouse Retina Il dataset RE è stato introdotto nello studio "Batch alignment of single-cell transcriptomics data using deep metric learning" (scDML) come benchmark multi-batch per la correzione dei batch effects [30]. Esso contiene cellule di retina murina, con 23 494 cellule e 13 166 geni, distribuiti in 6 batch che rappresentano replicati biologici/tecnici ottenuti con lo stesso protocollo. In questo caso i batch effects sono dovuti principalmente a differenze tra replicati (giorno di esperimento, preparazione del campione), a parità di tecnologia. I dati sono forniti come raw counts, rendendo necessario un pre-processing standard (normalizzazione, log-trasformazione) prima dell'analisi [30].

LA – Lung Atlas Il dataset LA è uno dei task di integrazione definiti da Luecken et al. nello stesso studio di benchmark atlas-level precedentemente citato [7]. Esso aggrega dati di polmone umano, includendo campioni provenienti da diversi studi, laboratori, protocolli di campionamento e localizzazioni anatomiche. Il dataset comprende 32 472 cellule e 15 148 geni, suddivisi in 16 batch corrispondenti a diversi donatori e condizioni sperimentali. I batch effects riflettono quindi variazioni inter-donatore, di sito di campionamento e di protocollo. I dati sono stati pre-processati dagli autori (log-normalizzati) [7, 31].

IH – Immune Human Il dataset IH rappresenta un'integrazione di dati di cellule immunitarie umane provenienti da sangue periferico e midollo osseo, anch'esso utilizzato nello studio di Luecken et al. [7]. Il dataset include 33 506 cellule e 12 302 geni, suddivisi in 10 batch che corrispondono a donatori provenienti da 5 studi o dataset differenti. Anche in questo caso, i dati sono forniti come log-normalizzati [7, 31].

IM – Immune (Human + Mouse) Il dataset IM estende ulteriormente lo scenario combinando cellule immunitarie umane e murine [7]. Esso contiene 97 861 cellule e 8 135 geni, distribuiti in 23 batch che riflettono differenti donatori, tessuti (sangue e midollo), specie (umano e topo) e laboratori. I dati sono forniti nuovamente come log-normalizzati [7, 31].

CL – Cell Lines Il dataset CL è stato utilizzato come benchmark per la correzione dei batch effects nello studio FedscGen [32]. Esso deriva dall'esperimento 293t_jurkat di Zheng et al. (2017) e contiene 9 531 cellule e 1 126 geni [33]. I 3 batch corrispondono a tre esperimenti di mixing controllato: (1) campione puro 293T, (2) campione puro Jurkat T, (3) mix 50:50 delle due linee cellulari. I dati sono forniti come conteggi pre-processati con selezione di geni altamente variabili [32].

TC – Thymus Cells Il dataset TC contiene dati di scRNA-seq del timo umano, con cellule stromali ed epiteliali in diverse fasi dello sviluppo [34]. Il dataset comprende 68 008 cellule e 804 geni, suddivisi in 5 batch che corrispondono a differenti stadi di sviluppo: Fetal 19 settimane, Fetal 23 settimane, Postnatale 6 giorni,

Postnatale 10 mesi, Adulto 25 anni. I batch effects si sovrappongono quindi a variazioni biologiche reali legate allo sviluppo, rendendo questo dataset ideale per testare la capacità dei metodi di non confondere l'effetto di batch e l'effetto di sviluppo. I dati sono pre-processati selezionando geni altamente variabili e applicando log-normalizzazione [34, 35].

OP – Open Problems (BMMC/HSPC) Il dataset OP è stato utilizzato nell'ambito di Open Problems per il benchmarking di metodi di integrazione multimodale e correzione dei batch effects su dati di cellule ematopoietiche umane [10]. Contiene 69 249 cellule e 129 921 geni, provenienti da 13 batch che riflettono differenze di donatore, tempo e sito di raccolta, oltre che di protocollo. I dati sono forniti come matrice di conteggi grezzi [10].

3.2 Metriche di valutazione

Per valutare la bontà della correzione dei batch effects in un dataset di single-cell RNA sequencing è necessario considerare diversi aspetti del problema. In primo luogo, è desiderabile che la dipendenza dei dati dal batch di provenienza venga ridotta il più possibile, in modo da evitare che variazioni tecniche non biologiche influenzino l'analisi. Tuttavia, questo non rappresenta l'unico criterio rilevante. Un metodo di correzione efficace deve infatti ridurre l'informazione associata al batch senza compromettere la struttura biologica dei dati, che si manifesta tipicamente nella separazione tra i diversi tipi cellulari.

Un ulteriore aspetto da considerare è che nessuna metrica di valutazione, se considerata singolarmente, è in grado di catturare completamente la bontà di una procedura di batch correction. Per questo motivo è pratica comune utilizzare più metriche, ciascuna delle quali valuta un diverso aspetto del problema, come il mixing dei batch, la preservazione della struttura biologica o la predicibilità del batch nei dati corretti.

Di seguito sono riportate le metriche quantitative utilizzate nel presente lavoro. Tutte le metriche sono state normalizzate nell'intervallo $[0, 1]$. Per la maggior parte di esse il valore 1 corrisponde allo scenario più desiderabile, mentre il valore 0 rappresenta la situazione peggiore. Solo nel caso delle metriche basate sulla predizione del batch, come l'accuratezza e l'AUC della regressione logistica, i valori non sono stati invertiti, per preservarne l'interpretabilità intuitiva: in tali casi un valore vicino a 0 indica una bassa predicibilità del batch a partire dai dati corretti (dunque uno scenario desiderabile), mentre un valore vicino a 1 indica che il batch rimane facilmente identificabile nei dati.

- **kBET (k-nearest neighbor Batch Effect Test)**

  La metrica kBET valuta il grado di mescolamento dei batch nel vicinato locale delle cellule nello spazio delle feature (tipicamente, come in questo caso, uno spazio di embedding come la PCA). L'idea di base è che, in assenza di batch effects, la distribuzione dei batch tra i k vicini più prossimi di una cellula debba riflettere la distribuzione globale dei batch nel dataset.

Formalmente, per ogni cellula i si considera l'insieme dei suoi k vicini più prossimi $N_k(i)$ e si confronta la distribuzione osservata dei batch all'interno di tale vicinato con la distribuzione globale dei batch nel dataset tramite un test statistico di tipo χ^2. Più precisamente, indicando con B il numero di batch e con $n_i = (n_{i1}, \ldots, n_{iB})$ il vettore dei conteggi dei batch nel vicinato della cellula i, l'ipotesi nulla assume che tali conteggi siano generati da una distribuzione multinomiale con probabilità pari alle frequenze globali dei batch $p = (p_1, \ldots, p_B)$ nel dataset.

Se il test rigetta l'ipotesi nulla, significa che il vicinato locale non è rappresentativo della distribuzione globale dei batch, indicando la presenza di batch effects.

Il punteggio kBET è definito come la frazione di test non rigettati:

$$\text{kBET} = 1 - \frac{\text{numero di test rigettati}}{\text{numero totale di test}} \tag{3.1}$$

Un valore elevato di kBET indica quindi che i vicinati locali delle cellule presentano una distribuzione dei batch simile a quella globale, suggerendo un buon mixing dei batch. Grazie alla sua natura basata su test statistici locali, kBET risulta particolarmente sensibile alla presenza di strutture residue di batch anche quando queste coinvolgono solo porzioni limitate del dataset, rendendolo una metrica complementare ad altre misure di integrazione.

- **iLISI (integration Local Inverse Simpson's Index)**

  Anche la metrica iLISI misura il grado di integrazione dei batch nel vicinato locale delle cellule, con un approccio differente rispetto a kBET. Essa si basa sull'indice di Simpson inverso, una misura di diversità comunemente utilizzata in molti studi di analisi scRNA-seq.

  Per ogni cellula i, considerando il suo vicinato $N_k(i)$, si definisce la frazione di vicini appartenenti alla categoria b come

$$p_{i,b} = \frac{1}{k} \sum_{j \in N_k(i)} \mathbf{1}[z_j = b] \tag{3.2}$$

  dove $\mathbf{1}[\cdot]$ è la funzione indicatrice e z_j rappresenta il batch della cellula j.

  L'indice LISI è quindi definito come:

$$LISI(i) = \frac{1}{\sum_b p_{i,b}^2} \tag{3.3}$$

  dove la somma è estesa a tutti i batch presenti nel dataset.

Questo indice assume valore minimo pari a 1 quando tutti i vicini appartengono allo stesso batch, e cresce all'aumentare della diversità dei batch nel vicinato. Nel caso ideale di perfetto mixing dei batch, le proporzioni $p_{i,b}$ risultano uniformi e il valore dell'indice si avvicina al numero totale di batch.

Nel contesto del presente lavoro il valore medio di LISI è stato normalizzato nell'intervallo $[0, 1]$. Indicando con B il numero totale di batch, la normalizzazione può essere espressa come:

$$\text{iLISI} = \frac{\overline{LISI} - 1}{B - 1} \tag{3.4}$$

in modo tale che valori prossimi a 0 indichino assenza di mixing tra batch mentre valori prossimi a 1 rappresentino uno scenario di integrazione ideale.

- **cLISI (cell-type Local Inverse Simpson's Index)**

  La metrica cLISI è concettualmente quasi del tutto analoga a iLISI, ma viene calcolata utilizzando le etichette di tipo cellulare anziché quelle di batch. L'obiettivo è valutare se la correzione dei batch effects preserva la separazione biologica tra i diversi tipi cellulari.

  Anche in questo caso si calcola l'indice LISI nel vicinato locale, ma considerando le proporzioni $p_{i,c}$ delle diverse classi cellulari:

$$LISI_{cell}(i) = \frac{1}{\sum_c p_{i,c}^2} \tag{3.5}$$

  dove $p_{i,c}$ rappresenta la frazione di cellule appartenenti al tipo cellulare c nel vicinato della cellula i.

  Nel caso ideale, cellule appartenenti allo stesso tipo cellulare dovrebbero essere vicine nello spazio delle feature; di conseguenza il valore di LISI calcolato sulle etichette biologiche dovrebbe risultare basso. Indicando con C il numero totale di classi cellulari, nel presente lavoro la metrica è stata normalizzata come

$$\text{cLISI} = 1 - \frac{\overline{LISI} - 1}{C - 1} \tag{3.6}$$

  in modo tale che valori prossimi a 1 indichino che i vicini appartengono prevalentemente allo stesso tipo cellulare, suggerendo una buona preservazione della struttura biologica locale.

- **LogReg batch prediction accuracy**

  Un ulteriore modo per valutare la presenza di informazione residua sul batch nei dati consiste nel verificare quanto sia facile predire il batch di appartenenza delle cellule a partire dalle loro feature.

A questo scopo è stato addestrato un classificatore di regressione logistica per predire la variabile di batch utilizzando come input l'embedding dei dati. Formalmente, la regressione logistica stima la probabilità che una cellula i appartenga al batch b come:

$$P(y_i = b \mid x_i) = \frac{e^{w_b^\top x_i}}{\sum_{b'} e^{w_{b'}^\top x_i}} \tag{3.7}$$

dove x_i rappresenta il vettore di feature della cellula e w_b i parametri del modello associati al batch b.

L'accuratezza del classificatore è definita come:

$$\text{accuratezza} = \frac{\text{numero di predizioni corrette}}{\text{numero totale di campioni}} \tag{3.8}$$

Un valore elevato di accuratezza indica che il batch è facilmente predicibile a partire dai dati, suggerendo che una quantità significativa di informazione legata al batch è ancora presente nel dataset. Al contrario, valori bassi indicano che il batch non è più distinguibile, suggerendo una correzione efficace.

- **LogReg batch prediction AUC**

  Oltre all'accuratezza, è stata considerata l'Area Under the ROC Curve (AUC), una metrica che valuta la capacità del classificatore di distinguere tra le diverse classi indipendentemente dalla soglia decisionale scelta. Rispetto all'accuratezza, l'AUC risulta più informativa in presenza di dataset sbilanciati, poiché valuta la separabilità dei batch basandosi sulla distribuzione dei punteggi di probabilità assegnati dal modello piuttosto che su una classificazione rigida.

  Formalmente, in un contesto binario, l'AUC rappresenta la probabilità che il classificatore assegni un punteggio maggiore a un campione estratto casualmente dalla classe positiva rispetto a uno estratto casualmente dalla classe negativa. Nel contesto dell'integrazione di dati scRNA-seq, dove i batch sono frequentemente più di due, la metrica è stata estesa seguendo una strategia one-vs-rest con aggregazione di tipo media macro. In questo approccio viene calcolato un valore di AUC per ciascun batch b trattandolo come classe positiva contro tutti gli altri batch aggregati come classe negativa. Il punteggio globale è definito come la media non pesata delle AUC individuali:

$$AUC_{total} = \frac{1}{B} \sum_{b=1}^{B} AUC_b \tag{3.9}$$

  dove B rappresenta il numero totale di batch presenti nel dataset.

  Dal punto di vista dell'integrazione dei dati, l'AUC funge da misura operativa del batch effect residuo: valori prossimi a 1 indicano che i batch rimangono

facilmente distinguibili nell'embedding, segnalando una correzione incompleta. Al contrario, valori prossimi a 0.5 indicano che il classificatore non è in grado di distinguere l'origine delle cellule meglio di un'assegnazione casuale, suggerendo che l'informazione legata al batch è stata efficacemente rimossa e i dataset risultano correttamente integrati.

- **ARI (Adjusted Rand Index)**

  L'Adjusted Rand Index misura la similarità tra due diverse partizioni di uno stesso insieme di dati. Nel contesto di questa analisi viene utilizzato per confrontare il clustering ottenuto sui dati corretti con le etichette di tipo cellulare, considerate note nei dataset a disposizione.

  In questo lavoro il clustering è stato ottenuto mediante l'algoritmo di Leiden applicato al grafo di vicinato costruito nell'embedding PCA, una scelta comunemente adottata nell'analisi di dati single-cell per identificare comunità cellulari coerenti.

  Data una partizione U e una partizione V, l'ARI è definito come:

$$ARI = \frac{RI - \mathbb{E}[RI]}{\max(RI) - \mathbb{E}[RI]} \tag{3.10}$$

  dove RI è il Rand Index, che misura la frazione di coppie di campioni che sono assegnate allo stesso cluster in entrambe le partizioni oppure a cluster diversi in entrambe ed $\mathbb{E}[\cdot]$ è il valore atteso.

  L'ARI assume valore 1 quando le due partizioni coincidono perfettamente e valore 0 quando la similarità tra le due partizioni è pari a quella attesa per assegnazioni casuali.

- **NMI (Normalized Mutual Information)**

  La Normalized Mutual Information misura la quantità di informazione condivisa tra due partizioni dei dati. Nel caso del clustering di dati scRNA-seq, viene utilizzata per quantificare quanto le assegnazioni ai cluster riflettano la struttura biologica definita dalle etichette di tipo cellulare.

  Siano U e V due partizioni dei dati. L'informazione mutua è definita come:

$$I(U,V) = \sum_{u \in U} \sum_{v \in V} p(u,v) \log \frac{p(u,v)}{p(u)p(v)} \tag{3.11}$$

  La NMI è quindi ottenuta normalizzando l'informazione mutua rispetto alle entropie delle due partizioni:

$$NMI = \frac{2I(U,V)}{H(U) + H(V)} \tag{3.12}$$

dove $H(\cdot)$ rappresenta l'entropia di Shannon.

Il valore della NMI è compreso tra 0 e 1: valori elevati indicano una forte corrispondenza tra clustering e tipi cellulari, suggerendo che la struttura biologica dei dati è stata preservata.

3.3 Protocollo sperimentale

L'obiettivo di questa fase preliminare del lavoro, come sottolineato in precedenza, è quello di selezionare il metodo migliore tra i candidati. La fase di testing dei metodi individuati vuole quindi essere un confronto di tali algoritmi condotto in contesti di dataset reali, di dimensionalità e caratteristiche variabili, come apprezzabile dalla sezione riguardante la descrizione dei dataset scelti. Vista la possibilità di combinare molti parametri, che danno origine ad un numero di permutazioni potenzialmente infinito, è stato deciso di testare i metodi con le opzioni di default, poiché ci si aspetta che le prestazioni risultino ugualmente solide (sebbene non ottimizzate al massimo del loro potenziale). I metodi testati sono stati quelli citati nel capitolo precedente, con l'aggiunta dei dati "Raw", normalizzati e log-trasformati, come gruppo di controllo:

Metodo:	Raw	Harmony	ComBat	scANVI	scGPT
Paradigma:	controllo	machine learning	statistico	deep learning	transformer

Tabella 3.1: Metodi presi in considerazione nell'analisi, con relativo principio teorico alla base.

Le metriche valutate sono anch'esse quelle citate nel paragrafo precedente, salvo per l'accuratezza della regressione logistica, utilizzata intrinsecamente per il calcolo dell'AUC della curva ROC, ma non considerata ai fini dell'analisi statistica. Le metriche sono dunque:

Metrica:	kBET	iLISI	cLISI	ARI	NMI	AUC
Significato:	batch mix	batch mix	biology	biology	biology	batch prediction

Tabella 3.2: Metriche tenute in considerazione per il confronto dei metodi di correzione degli effetti di batch, con relativo significato.

Il protocollo sperimentale vero e proprio è molto semplice, poiché il contesto non impone un'analisi statistica eccessivamente arricchita da numerose metriche e scenari complessi, in quanto si dispone già di solide basi in letteratura, oltre al contesto operativo di Orange, che impone comunque dei limiti nella complessità dello scenario valutato (non sarebbe in grado, per la gestione grafica e a matrici dense, di sopportare scenari oltremodo complessi).

I dataset, previa normalizzazione, log-trasformazione e selezione dei 3000 geni maggiormente significativi mediante dropout selection, sono stati elaborati con i metodi scelti. Si ottengono dunque 8 valori (corrispondenti ai dataset scelti) per ogni metrica prevista, per ogni metodo di correzione degli effetti di batch considerato (un totale dunque di 240 valori). Da un punto di vista quantitativo, come si tratterà nel paragrafo successivo, sono stati condotti test statistici per confrontare l'efficacia dei metodi sotto diversi profili (diverse metriche). Si propone, inoltre, anche un approccio più visuale, caratterizzato principalmente da heat map e grafici atti a fornire una prima impressione qualitativa dello scenario.

3.4 Risultati del confronto

In questa sezione sono presentati i risultati del confronto tra i metodi individuati in precedenza.

Occorre, tuttavia, fare una doverosa premessa prima di trattare l'illustrazione vera e propria dei risultati. Per simulare il contesto realistico di un flusso di lavoro in Orange, è stato deciso di evitare l'utilizzo di GPU durante l'allenamento dei modelli e l'esecuzione degli algoritmi. Ciò ha portato, come intuibile da ragioni evidenziate nei capitoli precedenti, ad evidenti difficoltà legate all'utilizzo di modelli più complessi e computazionalmente onerosi, quali scANVI e scGPT. Il runtime dei suddetti modelli oscillava tra le decine di minuti, nei casi oltremodo semplici, ad intere ore nei casi di media complessità. In un utilizzo tipico di Orange (e.g. workshop, didattica, scoperta del machine learning) tale comportamento sarebbe del tutto incompatibile per ragioni legate soprattutto a limiti di carattere temporale. Inoltre, ciò si scontra con la logica di esecuzione in tempo reale del flusso di lavoro di Orange. Risulterebbe incoerente implementare un metodo di concezione complessa, nel momento in cui esso può essere utilizzato, in un contesto operativo realistico, solamente su dataset semplici, in cui non risulterebbe necessario un metodo così complesso. Dunque, tenendo conto anche che i benchmark presenti in letteratura non evidenziano vittorie schiaccianti di tali modelli rispetto ad approcci più semplici, soprattutto su problemi di medio-bassa complessità (tipici di Orange), è stato deciso di escludere scANVI ed scGPT dalle opzioni plausibili. Per questo motivo e per motivi di scorrevolezza dell'elaborato, non verranno mostrati i risultati ad essi legati nel presente paragrafo, in quanto ininfluenti per il confronto. Si è comunque deciso di trattare, anche se brevemente, di tali metodi in questo elaborato per rispettare la narrazione cronologica e, soprattutto, per evidenziare l'importanza in fase progettuale di contestualizzare il lavoro nell'ambito di sviluppo del software Orange.

Per valutare quantitativamente l'efficacia dei metodi di integrazione, è stata condotta un'analisi statistica sulle metriche calcolate per gli 8 dataset in esame. Le statistiche descrittive (media e deviazione standard) per ciascuna metrica, suddivise per scenario di analisi (Raw, Har, Com), sono riportate nella Tabella 3.3.

Considerando la limitata numerosità campionaria ($N = 8$), che non garantisce il rispetto delle assunzioni di normalità e omoschedasticità, si è optato per un approccio non parametrico. Nello specifico, le differenze globali tra le distribuzioni dei

punteggi nei tre scenari sono state valutate mediante il test di Friedman. Nei casi in cui è emersa una significatività statistica globale (considerando $\alpha = 0.05$), sono stati eseguiti confronti post-hoc a coppie utilizzando il test dei ranghi con segno di Wilcoxon per dati appaiati. Al fine di controllare l'inflazione del tasso di falsi positivi (Family-Wise Error Rate) derivante dai confronti multipli, i p-value ottenuti sono stati corretti mediante il metodo di Holm-Bonferroni. I risultati completi dei test statistici sono riassunti nella Tabella 3.4.

Tabella 3.3: Statistiche descrittive delle metriche di valutazione calcolate sugli 8 dataset. I valori sono espressi come media ± deviazione standard per i dati non corretti (Raw) e per i dati integrati con i metodi Harmony e ComBat.

Metrica	**Raw**	**Harmony**	**ComBat**
kBET	0.0732 ± 0.2038	0.0969 ± 0.2011	0.0282 ± 0.0786
iLISI	0.1805 ± 0.2230	0.3318 ± 0.2001	0.0895 ± 0.1783
cLISI	0.9818 ± 0.0122	0.9775 ± 0.0164	0.9949 ± 0.0038
ARI	0.4644 ± 0.1226	0.5025 ± 0.1513	0.4401 ± 0.1478
NMI	0.6505 ± 0.0818	0.6549 ± 0.1087	0.7092 ± 0.0846
AUC	0.9281 ± 0.0996	0.6851 ± 0.0865	0.8299 ± 0.1487

Tabella 3.4: Risultati dei test statistici. Nella prima colonna numerica è riportato il p-value del test globale di Friedman. Nelle successive colonne sono riportati i p-value corretti (p_{corr}, metodo di Holm-Bonferroni) relativi ai confronti post-hoc eseguiti con il test di Wilcoxon. I valori statisticamente significativi ($p \leq \alpha = 0.05$) sono evidenziati con l'asterisco (*). I trattini indicano i casi in cui il test post-hoc non è stato eseguito per assenza di significatività globale.

Metrica	**Friedman** (p)	**Confronti Post-Hoc** (p_{corr})		
		Raw vs Har	**Raw vs Com**	**Har vs Com**
kBET	0.0030^*	0.0546	0.0546	0.0533
iLISI	0.0003^*	0.0234^*	0.0234^*	0.0234^*
cLISI	0.0008^*	0.0234^*	0.0234^*	0.0234^*
ARI	0.8825	–	–	–
NMI	0.4169	–	–	–
AUC	0.0022^*	0.0234^*	0.0547	0.0312^*

L'analisi dei risultati evidenzia chiaramente come l'algoritmo Harmony (Har) offra le prestazioni migliori in termini di rimozione del batch effect. Per le metriche iLISI e AUC della regressione logistica, il test di Friedman ha rivelato differenze globali

altamente significative. I successivi test post-hoc hanno confermato che la correzione con Harmony migliora significativamente il mescolamento locale nello spazio latente rispetto ai dati non corretti (valore medio di iLISI che passa da 0.1805 a 0.3318, $p_{corr} = 0.0234$) e riduce in modo sostanziale la predicibilità del batch (l'AUC cala da 0.9281 a 0.6851, $p_{corr} = 0.0234$). Al contrario, il metodo ComBat (Com) non riesce a garantire un'integrazione altrettanto efficace, producendo valori di iLISI addirittura inferiori al dato grezzo (0.0895, probabilmente perché alcuni scenari analizzati hanno una complessità non sostenibile da ComBat) e un'AUC di 0.8299 che non si discosta in modo statisticamente significativo da quella dei dati pre-correzione ($p_{corr} = 0.0547$). Inoltre, nel confronto diretto, Harmony sovraperforma significativamente ComBat sull'AUC ($p_{corr} = 0.0312$). Per quanto riguarda la metrica kBET, pur essendo emersa una significatività globale ($p = 0.0030$), la penalizzazione imposta dalla correzione di Holm-Bonferroni su un campione così ridotto non ha permesso di isolare differenze statisticamente significative nei confronti a coppie ($p_{corr} \approx 0.054$), sebbene le medie indichino un netto trend di mixing favorevole ad Harmony e p_{corr} sfiori il valore soglia α.

Parallelamente, è stata valutata la capacità dei metodi di preservare la struttura biologica dei dati, aspetto fondamentale per scongiurare fenomeni di overcorrection. I risultati confermano l'eccellente profilo conservativo degli algoritmi: il test di Friedman non ha rilevato alcuna variazione significativa per l'indice ARI ($p = 0.8825$) né per la NMI ($p = 0.4169$), confermando che l'informazione biologica dei cluster e la loro separabilità globale non vengono stravolte dai metodi di correzione rispetto ai dati raw. Analizzando cLISI, il test ha evidenziato differenze significative ($p = 0.0008$); tuttavia, queste si traducono in fluttuazioni minime in termini assoluti. Sebbene Harmony induca un lieve ma fisiologico calo rispetto al dato raw (0.9775 contro 0.9818, $p_{corr} = 0.0234$), i valori si mantengono ampiamente prossimi all'unità. Questi risultati suggeriscono che la purezza dei vicinati locali rispetto al vero tipo cellulare viene preservata in modo eccellente, dimostrando come Harmony offra il miglior compromesso tra integrazione dei batch e sicurezza biologica.

I risultati sono inoltre riportati di seguito mediante boxplot e heatmap (Figure 3.1, 3.2).

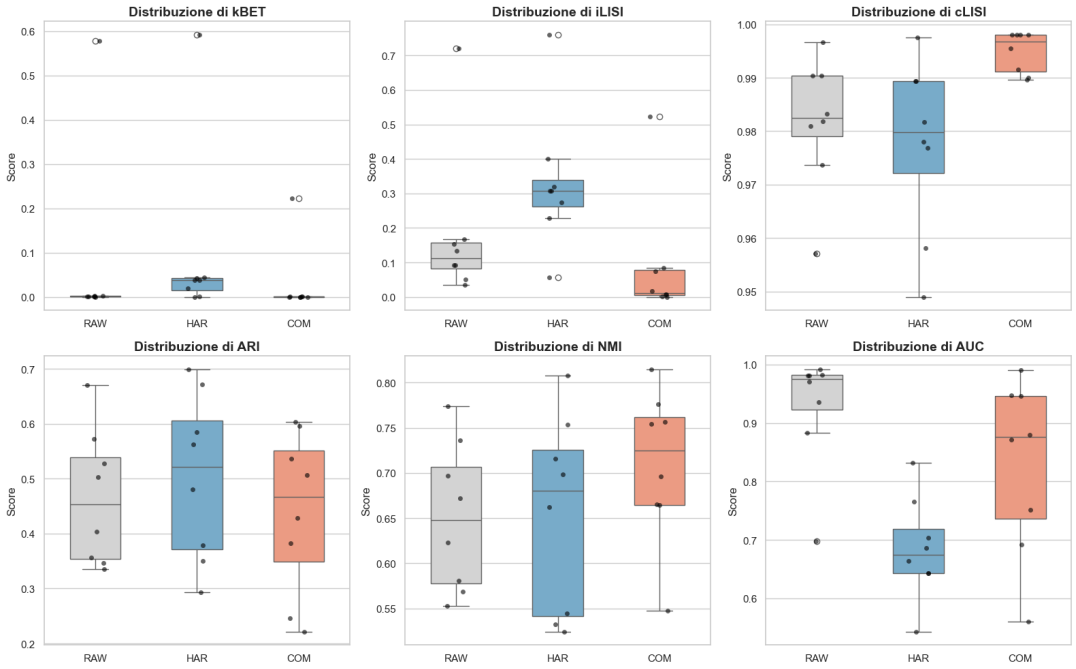


Figura 3.1: Boxplot delle metriche (kBET, iLISI, cLISI, ARI, NMI e AUC) valutate sui metodi di integrazione (Raw, Harmony, ComBat) per ciascun dataset. Si noti che la scala sull'asse delle ordinate varia per ogni metrica, per questioni di leggibilità dai grafici.

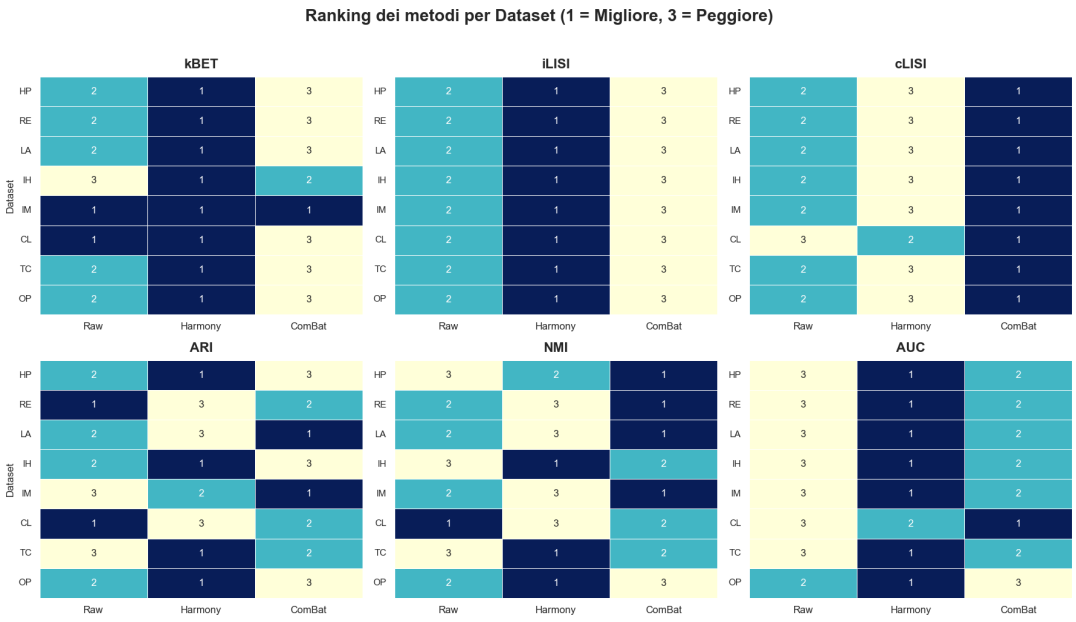


Figura 3.2: Ranking dei metodi di integrazione per ciascun dataset. I valori indicano il posizionamento relativo (1 = performance migliore, 3 = performance peggiore) di ciascun metodo (Raw, Harmony, ComBat) valutato secondo le metriche kBET, iLISI, cLISI, ARI, NMI e AUC.

3.5 Motivazioni della scelta di Harmony: dati e integrazione in Orange

Nel complesso, dunque, l'analisi statistica conferma che l'implementazione di Harmony rappresenta probabilmente il miglior compromesso tra l'integrazione tecnica dei batch e la conservazione dell'eterogeneità biologica originaria. Inoltre, Harmony è un metodo di concezione più moderna ed elaborata rispetto a ComBat, sebbene preservi un'eccellente velocità di elaborazione. I runtime appartengono, infatti, a scale temporali paragonabili (nell'ordine di grandezza dei secondi o, al più, pochi minuti per scenari complessi). Dalla letteratura emerge anche come Harmony sia generalmente poco propenso all'overcorrection [10]. Per natura stessa dell'algoritmo, in aggiunta, Harmony offre caratteristiche di scalabilità e possibilità di interazione con l'algoritmo (e quindi con il widget); i parametri di principale importanza (si veda il capitolo successivo riguardante l'implementazione del widget) consentono infatti di regolare aspetti che influenzano il processo di correzione, permettendo di gestire dimensionalità, complessità del problema, performance e overcorrection.

La scelta, di conseguenza, è ricaduta sul metodo Harmony.

Capitolo 4

Implementazione del widget Harmony in Orange

In questo capitolo vengono approfonditi dettagliatamente gli aspetti principali di Harmony, a seguito della scelta dello stesso come metodo designato all'implementazione in Orange. Verranno trattati concetti legati al metodo originariamente concepito dagli autori, alla sua implementazione in Python mediante la libreria `harmonypy` ed infine all'implementazione vera e propria all'interno dell'ecosistema Orange.

4.1 Metodo Harmony: i principi

Come anticipato in parte nel capitolo 3, Harmony è un metodo di correzione degli effetti di batch proposto da Korsunsky et al. [27]. Si era evidenziato che una caratteristica fondamentale di Harmony fosse il fatto che esso operi direttamente su una rappresentazione a bassa dimensionalità dei dati, tipicamente ottenuta mediante Analisi delle Componenti Principali (PCA), piuttosto che sulla matrice originale di espressione genica ad alta dimensionalità. Questa caratteristica consente a Harmony di essere efficiente anche su dataset di grandi dimensioni e di essere applicato in modo indipendente dalle strategie di normalizzazione o selezione delle feature adottate a monte (metodi come scANVI, ad esempio, richiedono i conteggi raw, non normalizzati, spesso difficili da trovare in dataset pubblici). Entreremo ora nei dettagli della matematica alla base del metodo, in quanto è un passo necessario alla comprensione di quest'ultimo e alla successiva implementazione.

Sia quindi $Z \in \mathbb{R}^{d \times N}$ l'embedding a bassa dimensionalità dei dati, dove Z_i rappresenta l'embedding d-dimensionale della cellula i, e sia ϕ_i il vettore di assegnazione al batch associato alla cellula i. L'obiettivo di Harmony è calcolare un embedding corretto Z' della stessa dimensionalità di Z, tale che cellule che condividono lo stesso stato biologico risultino allineate tra batch differenti, mentre gli effetti specifici di batch vengano rimossi.

4.1.1 Ottimizzazione iterativa e funzione obiettivo

Harmony applica un processo iterativo che consiste in due fasi principali:

1. **soft clustering** delle cellule nello spazio di embedding, con un vincolo esplicito sulla diversità dei batch all'interno dei cluster;
2. **regressione e correzione degli effetti di batch**, eseguite in modo indipendente per ciascun cluster.

Queste due fasi vengono ripetute fino a convergenza, producendo assegnamenti ai cluster e stime degli effetti di batch progressivamente più accurate. Il procedimento ricorda uno schema di tipo expectation–maximization, in cui le appartenenze ai cluster e i parametri di correzione vengono aggiornati alternativamente.

Durante la fase di soft clustering, basata sull'algoritmo k-means, a ciascuna cellula i vengono assegnati pesi $R_{ki} \geq 0$ rispetto a K cluster, tali che $\sum_k R_{ki} = 1$. Le assegnazioni e i centroidi dei cluster vengono ottenuti minimizzando una funzione obiettivo complessa, che bilancia la coesione del clustering, l'entropia delle assegnazioni e la diversità dei batch all'interno di ogni cluster [27]:

$$J(R, Y) = \sum_{i,k} R_{ki} \|Z_i - Y_k\|^2 + \sigma \sum_{i,k} R_{ki} \log R_{ki} + \sigma\theta \sum_{i,k} R_{ki} \log \left(\frac{O_{ki}}{E_{ki}} \right) \tag{4.1}$$

dove:

- $Z_i \in \mathbb{R}^d$ è l'embedding della cellula i nello spazio PCA;
- $Y_k \in \mathbb{R}^d$ rappresenta il centroide del cluster k;
- $R_{ki} \in [0, 1]$ è la probabilità (soft assignment) che la cellula i appartenga al cluster k, con il vincolo $\sum_k R_{ki} = 1$;
- σ è un parametro di regolarizzazione che controlla la morbidezza del clustering (softness);
- θ è il parametro che regola la penalità per lo sbilanciamento dei batch (diversity penalty).

Il primo termine della funzione corrisponde ad un obiettivo di tipo k-means pesato, che incoraggia le cellule a essere vicine ai centroidi Y_k.

Il secondo termine introduce una regolarizzazione entropica, controllata dal parametro σ, che penalizza assegnazioni troppo rigide e favorisce appartenenze soft ai cluster, migliorando la stabilità numerica.

Il terzo termine è specifico di Harmony e impone che ogni cluster sia rappresentato in modo equilibrato da tutti i batch disponibili. Siano B il numero di batch e $Pr(b) = \frac{N_b}{N}$ la frequenza globale del batch b nel dataset. Definiamo le seguenti quantità :

- **massa osservata (O_{kb}):** la quantità di cellule del batch b effettivamente assegnata al cluster k:
$$O_{kb} = \sum_i 1_{i \in b} R_{ki} \tag{4.2}$$

- **massa attesa (E_{kb}):** la quantità di cellule del batch b che ci si aspetterebbe nel cluster k sotto l'ipotesi di indipendenza tra batch e cluster:
$$E_{kb} = Pr(b) \sum_i R_{ki} \tag{4.3}$$

Questo termine penalizza cluster la cui composizione di batch si discosta significativamente dalla distribuzione globale, scoraggiando la formazione di cluster dominati da un singolo batch. L'intensità di tale penalizzazione è controllata dal parametro θ, che regola il compromesso tra coesione biologica e mescolamento dei batch.

Per massimizzare l'efficienza, Harmony utilizza la distanza coseno. Se i vettori Z_i e Y_k sono normalizzati L_2, la distanza euclidea quadratica si riduce a:

$$dist_{ki} = 2(1 - Y_k^\top Z_i) \tag{4.4}$$

L'algoritmo alterna fasi di clustering e correzione fino a convergenza. L'aggiornamento delle responsabilità R_{ki} deriva direttamente dalla minimizzazione della Lagrangiana associata alla funzione obiettivo, portando alla forma operativa:

$$R_{ki} \propto \exp\Big(- \frac{\text{dist}_{ki}}{\sigma} \Big) \cdot \Big(\frac{O_{k,b(i)}}{E_{k,b(i)}} \Big)^{\theta}. \tag{4.5}$$

Poiché O_{kb} ed E_{kb} dipendono globalmente da R, Harmony implementa degli aggiornamenti a blocchi [27]. Il dataset viene diviso in piccoli blocchi (e.g. 5% del totale); per ogni blocco, si rimuove temporaneamente il suo contributo dalle statistiche globali, si aggiornano le responsabilità R e si reinseriscono i nuovi valori (si approfondirà meglio questo aspetto più avanti nel capitolo).

4.1.2 Correzione lineare tramite Mixture of Experts (MoE)

Una volta ottenuto un clustering batch-indipendente, Harmony modellizza la variazione residua tramite un modello Mixture of Experts [27]. Per ogni cluster k, si definisce una regressione lineare Ridge che mette in relazione le coordinate PCA (Z) con la matrice dei batch (Φ^*), arricchita da un termine di intercetta:

$$W_k = (\Phi^* \text{diag}(R_k) \Phi^{*\top} + \lambda I)^{-1} \Phi^* \text{diag}(R_k) Z^\top \tag{4.6}$$

dove λ è un parametro di regolarizzazione Ridge che previene la singolarità della matrice. La correzione finale dell'embedding avviene sottraendo dai dati originali la componente predetta dalle covariate di batch, ponderata per la probabilità di appartenenza della cellula ai vari cluster:

$$\hat{Z}_i = Z_i - \sum_k R_{ki} W_{k[1:B,\cdot]}^{\top} \phi_{i[1:B]} \tag{4.7}$$

In questa fase, l'intercetta $W_{k[0,\cdot]}$ non viene sottratta, poiché essa rappresenta il segnale biologico intrinseco del cluster, legato al tipo cellulare (variabilità biologica).

4.2 Implementazione di Harmony in Orange

4.2.1 Analisi dell'implementazione pre-esistente in Python: paper originale vs `harmonypy`

Prima di procedere allo sviluppo dell'integrazione per Orange, è stata condotta un'analisi dell'implementazione di riferimento in Python, nella libreria dedicata `harmonypy`, confrontandola con le definizioni matematiche presentate nel paper originale di Korsunsky et al. [27]. Lo scopo di questa analisi è stato quello di valutare, in primo luogo, la possibilità di un'integrazione in Orange diretta delle funzioni presenti in `harmonypy`. In secondo luogo, si rendeva necessario verificare l'integrità di questa implementazione e la coerenza con il paper d'origine, essendo essa il riferimento principale in ambiente Python (e, dunque, utile come riferimento anche in caso di necessità di una nuova implementazione dell'algoritmo in Orange).

Nel complesso, `harmonypy` rappresenta una traduzione molto fedele dell'algoritmo operativo: mantiene la stessa distanza coseno, la medesima struttura della funzione obiettivo, lo stesso schema di aggiornamento a blocchi per le responsabilità R e la correzione mediante Mixture of Experts (MoE).

Tuttavia, emergono alcune differenze legate a scelte numeriche e pratiche, necessarie per trasformare il modello teorico in un software robusto.

Smoothing additivo e stabilità numerica

Nel paper, le statistiche di diversità O_{kb} (massa osservata) e E_{kb} (massa attesa) guidano la penalizzazione dei batch sovrarappresentati. Matematicamente, il termine di diversità dipende dal rapporto $\frac{O_{kb}}{E_{kb}}$. Tuttavia, in fasi iniziali o per cluster molto piccoli, è possibile che $O_{kb} \to 0$ o $E_{kb} \to 0$, portando a instabilità numerica e divisioni per zero. `harmonypy` risolve questo problema introducendo uno smoothing additivo (+1) nel calcolo del rapporto:

$$\Omega(R) = \log\left(\frac{O_{kb}+1}{E_{kb}+1}\right). \tag{4.8}$$

Questa modifica garantisce che il logaritmo sia sempre ben definito, senza alterare la dinamica asintotica del termine di sbilanciamento dei batch.

Inoltre, nell'algoritmo implementato in `harmonypy`, l'aggiornamento operativo assume la forma:

$$R_{ki} \propto \exp\Big(-\frac{\mathrm{dist}_{ki}}{\sigma_k}\Big) \cdot \Big(\frac{E_{k,b(i)}+1}{O_{k,b(i)}+1}\Big)^{\theta_{b(i)}}. \tag{4.9}$$

Sebbene l'espressione originale (Equazione 4.5) derivi direttamente dal gradiente della funzione obiettivo, questa nuova forma viene adottata per garantire una dinamica correttiva stabile durante l'ottimizzazione iterativa. In particolare, il fattore $(E/O)^\theta$ riduce le responsabilità associate a batch sovrarappresentati in un cluster e le aumenta nel caso di batch sottorappresentati, guidando progressivamente il sistema verso configurazioni in cui $O_{k,b} \approx E_{k,b}$.

Nel formalismo di Harmony, il termine $\log(O/E)$ va quindi interpretato come componente dell'obiettivo da minimizzare, mentre il fattore $(E/O)^\theta$ rappresenta un aggiornamento coerente.

Criteri di convergenza

L'articolo originale non specifica i dettagli sul termine delle iterazioni, limitandosi a indicare che l'algoritmo procede "fino a convergenza". `harmonypy` introduce criteri euristici precisi:

- **convergenza interna (clustering):** viene valutata su una finestra mobile e l'algoritmo considera il clustering stabilizzato se la variazione dell'errore all'interno di questa finestra scende sotto una determinata soglia;
- **convergenza esterna (Harmony):** l'interruzione del ciclo di Expectation-Maximization si basa sul calcolo del rapporto relativo tra i valori della funzione obiettivo in iterazioni successive $\left(\frac{|J_t - J_{t-1}|}{J_{t-1}}\right)$, arrestando il processo quando il miglioramento è trascurabile.

Aggiornamento a blocchi

Il paper prevede aggiornamenti a blocchi delle responsabilità [27], poiché le quantità O ed E dipendono globalmente da R. Un aggiornamento simultaneo di tutte le colonne di R utilizzando valori obsoleti di O ed E non approssimerebbe correttamente la dinamica desiderata.

L'implementazione `harmonypy.update_R()` realizza pertanto:

1. il calcolo della base
$$\tilde{R}_{ki} \propto \exp\Big(- \frac{\text{dist}_{ki}}{\sigma_k} \Big);$$
2. la suddivisione casuale delle cellule in blocchi di dimensione `block_size`;
3. per ciascun blocco:
    - rimozione temporanea del contributo del blocco da O ed E;
    - aggiornamento delle responsabilità tramite il fattore $(E+1)/(O+1)$;
    - normalizzazione delle colonne di R;
    - reinserimento del contributo aggiornato in O ed E.

4.2.2 La scelta di `NumPy`

Nonostante la coerenza di `harmonypy` con i principi cardine di Harmony [27], l'inclusione di quest'ultimo nell'Add-On Single Cell di Orange ha richiesto una reimplementazione ex novo dell'algoritmo in Python, totalmente indipendente dalla libreria `harmonypy`. Tale esigenza nasce da alcuni fattori tecnici, strettamente legati all'integrazione nell'ambiente di Orange:

1. **interoperabilità e coerenza dati:** Orange gestisce i dati internamente attraverso oggetti `Orange.data.Table`, che si basano su matrici NumPy. L'uso di `harmonypy` avrebbe richiesto la conversione dei dati in `pandas.DataFrame` o `AnnData`, introducendo un'operazione non necessaria e aumentando il rischio di errori di formattazione tra i widget;
2. **performance ed efficienza:** in un ambiente grafico interattivo, la velocità di esecuzione è fondamentale, come già trattato in precedenza. L'uso di `NumPy` "puro" permette di sfruttare appieno l'ottimizzazione delle operazioni vettoriali su array senza il sovraccarico di astrazioni di alto livello date da librerie più specifiche, garantendo che la correzione avvenga in pochi secondi anche per migliaia di cellule;
3. **riduzione delle dipendenze:** evitare librerie esterne pesanti semplifica notevolmente la manutenzione dell'Add-On e ne facilita la distribuzione agli utenti finali, riducendo i conflitti di versioni tra i vari pacchetti scientifici del sistema.

La versione sviluppata è stata quindi ottimizzata per operare in tale contesto, mantenendo la massima fedeltà al paper originale e basandosi sulla logica computazionale introdotta in `harmonypy`.

4.2.3 Architettura in Orange: `harmonypy` vs implementazione `NumPy`

L'implementazione del metodo è dunque proseguita verso la realizzazione di un codice indipendente, basato solamente, per quanto concerne gli aspetti strettamente computazionali (e non legati ad interfaccia grafica), sulla libreria `NumPy` (Appendice A.1). Le differenze principali tra il codice sviluppato per Orange e quello presente in `harmonypy` riguardano, inoltre, la gestione dei dati, le dipendenze e l'ottimizzazione delle operazioni matriciali:

Strutture dati e interoperabilità

Come introdotto in precedenza, `harmonypy` è progettato per interfacciarsi con l'ecosistema dell'analisi dati standard in Python, facendo largo uso di `pandas.DataFrame` per la gestione dei metadati, dei batch e, più recentemente, del formato `AnnData` (Annotated Data, sviluppato appositamente per la gestione di dati scRNA-seq), standard per la bioinformatica single-cell. Al contrario, Orange basa la sua intera architettura di passaggio dati sulla classe `Orange.data.Table`, che è essenzialmente un wrapper su array `NumPy` ad alte prestazioni.

Utilizzare `harmonypy` avrebbe richiesto al modulo `harmony_preprocess.py` (e al widget `owharmony.py`) di:

1. estrarre i dati continui e categorici da `Orange.data.Table`;
2. convertirli iterativamente in `pandas.DataFrame` o `AnnData`;
3. eseguire la correzione;
4. riconvertire l'output in un nuovo oggetto `Table` per passarlo ai widget successivi.

Questo processo di conversione avrebbe compromesso l'efficienza in termini di tempo di calcolo e di allocazione di memoria (si parla di matrici con decine di migliaia righe e migliaia di colonne). L'implementazione sviluppata riceve invece direttamente la matrice X nativa, processando i dati vettorialmente tramite `NumPy`.

Risoluzione numerica ottimizzata per la Ridge Regression

L'uso dell'inversione esplicita in `harmonypy` per il calcolo dei pesi della regressione lineare locale può presentare limiti di stabilità numerica. Nell'implementazione sviluppata per Orange (metodo `_moe_correct_ridge_numpy`), il sistema lineare associato alla regressione Ridge non viene risolto invertendo esplicitamente la matrice di covarianza $(\Phi^* \text{diag}(R_k)\Phi^{*\top} + \lambda I)^{-1}$, bensì utilizzando metodi di risoluzione nativi e più stabili di `NumPy` (`numpy.linalg.solve`). Tale accortezza assicura una maggiore affidabilità della correzione, soprattutto quando vi è un forte sbilanciamento nel numero di cellule appartenenti a determinati batch all'interno dei cluster.

4.3 Specifiche del widget

Conclusa la fase di analisi e implementazione dell'algoritmo, il lavoro è proseguito con lo sviluppo del widget all'interno dell'ambiente Orange. In fase di progettazione si è resa necessaria un'attenta definizione delle specifiche tecniche e delle funzionalità a disposizione dell'utente.

Input e Output

Fondamentale per la gestione dell'intero codice è stata la definizione degli input e degli output con cui il widget interagisce. La scelta finale ha propeso per una gestione semplice e lineare: si è deciso, infatti, di includere l'operazione di PCA all'interno del widget stesso, per evitare confusione all'utente. L'approccio ideato prevede dunque che vengano forniti in ingresso i dati da correggere, per ottenere direttamente in uscita il dataset corretto. Le operazioni di preprocessing, tuttavia, rimangono separate dal funzionamento del widget Harmony. Difatti, mentre la PCA è un'opzione obbligata per l'utilizzo dell'algoritmo stesso, le scelte in fase di preprocessing sono a discrezione dell'utente e concettualmente separate in Orange (è presente un widget dedicato). Risulterebbe, inoltre, eccessivamente complesso come widget, in quanto si tratta di uno strumento specifico per l'integrazione di dati e/o la correzione di effetti di batch. Per quanto riguarda l'output, il paradigma rimane

analogo. Come risulterebbe intuitivo pensare, l'unico output fornito dal widget è l'embedding corretto.

Parametri gestibili dall'utente

Occorre chiedersi, successivamente, quali siano i parametri sui quali si intende lasciare maggiore libertà all'utilizzatore. Nei capitoli precedenti è stata sviscerata l'importanza del ritrovare l'interattività in ogni aspetto di Orange; tuttavia, è altresì importante discernere tra ciò che risulta davvero fondamentale da gestire per l'utente e ciò che è invece superfluo, confusionario e di ingombro a livello grafico.

Scegliere di implementare l'opportunità di selezionare manualmente il valore di alcuni parametri, lasciando logicamente la possibilità di utilizzare i valori di default, è anche un modo per consentire all'utente più curioso di cimentarsi in un approccio più consapevole al problema.

La scelta finale è ricaduta sui seguenti parametri, alcuni dei quali già introdotti nel corso dell'elaborato:

- **softness (σ):** consente di modificare la rigidità nelle assegnazioni dell'algoritmo di soft clustering;
- **imbalance penality (θ):** penalizza lo sbilanciamento di batch tra classi. L'utente può scegliere l'aggressività della penalizzazione in base alla complessità e allo sbilanciamento del problema che deve affrontare;
- **regularization (λ):** termine di regolarizzazione;
- **clusters:** numero di cluster, che può essere modificato a seconda della complessità e della dimensionalità del problema;
- **max iterations:** numero massimo di iterazioni dell'algoritmo Harmony, a prescindere dal raggiungimento della convergenza. Consente all'utente di gestire anche problemi complessi, trovando un compromesso tra accuratezza della correzione e runtime;
- **principal components:** numero di Componenti Principali calcolate durante la PCA. Fondamentale per definire la dimensione dell'output ed il compromesso tra eliminazione dell'effetto di batch e preservazione della variabilità biologica.

4.4 Dettagli implementativi e architettura del codice

L'integrazione dell'algoritmo Harmony all'interno dell'ambiente Orange ha richiesto un'attenta progettazione architetturale per allineare il flusso computazionale del metodo con i paradigmi di programmazione a oggetti previsti dal framework. Il codice sorgente è stato suddiviso in tre moduli principali, garantendo la separazione tra l'interfaccia grafica (front-end, modulo `OWHarmony`, Appendice A.3), la gestione

delle strutture dati di Orange (middleware, `harmony_preprocess.py`, Appendice A.2) e il motore computazionale matematico (back-end, `harmony_full_numpy.py`, Appendice A.1).

4.4.1 Gestione dell'interfaccia grafica e delegati Qt

Il modulo `owharmony.py` definisce la classe `OWHarmony`, che eredita direttamente da `OWWidget` (disponibile nella libreria principale di Orange). Questa classe gestisce l'interazione con l'utente e il ciclo di vita dei dati in ingresso e in uscita.

Un aspetto tecnico rilevante nell'implementazione della GUI è stata la formattazione dei parametri numerici (come i valori di σ, θ, λ e i numeri interi per le iterazioni). Per garantire una corretta visualizzazione e allineamento all'interno delle tabelle e dei controlli Qt, sono state implementate due classi delegate custom: `IntegralDelegate` e `RealDelegate`, entrambe derivate da `QStyledItemDelegate` della libreria `AnyQt`. Sovrascrivendo il metodo `initStyleOption`, questi delegati forzano l'allineamento a destra (`Qt.AlignRight | Qt.AlignVCenter`) e, nel caso dei numeri reali, formattano il testo imponendo un numero fisso di cifre decimali, migliorando la leggibilità dell'interfaccia rispetto al comportamento di default.

4.4.2 Persistenza dello Stato tramite `ContextHandler`

In un ambiente di data mining visuale, è fondamentale che i widget abbiano memoria delle configurazioni dell'utente (come la selezione della variabile di batch) quando lo stesso dataset viene ricaricato o quando un workflow viene riaperto.

Per implementare questa funzionalità senza dover salvare esplicitamente i nomi delle variabili nel file di progetto in modo statico, la classe `OWHarmony` fa uso dei `ContextSetting`. La gestione dei contesti è affidata alla classe `PerfectDomain ContextHandler`. Quando un dataset entra nel widget, il gestore genera una "firma" (signature) basata sulle feature e sui metadati del dominio (`data.domain`). Se la firma coincide con una precedentemente incontrata, il widget ripristina dinamicamente i `ContextSetting` associati a quel dominio, riassegnando automaticamente le variabili di batch corrette nell'interfaccia.

4.4.3 Astrazione del preprocessing e ricostruzione del dominio

La logica di preparazione dei dati e il collegamento con il nucleo computazionale non risiedono nel widget visivo, ma sono stati astratti nel modulo `harmony_preprocess.py`. In Orange, le operazioni di trasformazione dei dati devono tipicamente conformarsi a una struttura a cascata. A tale scopo è stata creata la classe `HarmonyNormalizer`, che eredita dalla classe base `Orange.preprocess.preprocess.Preprocess`.

Il metodo `transform` (chiamato dall'operatore `__call__`) riceve l'oggetto `Table` originale. Da esso estrae la matrice dei dati continui (`X_raw`) e le colonne relative alle variabili di batch tramite la funzione `data.get_column(v)`. Tali strutture vengono quindi passate alla funzione `harmony_numpy_fun` del nucleo computazionale.

L'aspetto implementativo più critico di questa fase è la cosiddetta Domain Reconstruction. Poiché l'algoritmo restituisce un nuovo embedding (Z_{corr}), lo spazio dimensionale originale delle features continue non è più valido. La classe `HarmonyModel` si occupa quindi di:

1. generare nuove variabili continue, istanziando oggetti `ContinuousVariable (f"PC{i+1}")` per ogni colonna della nuova matrice corretta;
2. estrarre le liste originali delle variabili categoriche (`data.domain.class_vars`) e dei metadati (`data.domain.metas`), che contengono le informazioni biologiche necessarie per le valutazioni a valle;
3. istituire un nuovo oggetto `Domain` unendo le nuove componenti principali alle vecchie annotazioni;
4. assemblare infine una nuova istanza di `Table`, incapsulando il nuovo dominio, la matrice corretta Z_{corr}, e i vettori originali Y e `metas`.

4.4.4 Robustezza e Gestione delle Eccezioni

Il flusso di controllo nel widget è stato progettato per operare in sicurezza e prevenire crash non gestiti. All'interno del metodo di elaborazione (eseguito alla ricezione di nuovi segnali in ingresso), l'istanziazione del `HarmonyNormalizer` e la successiva correzione sono racchiuse all'interno di un blocco `try-except`.

Qualora l'algoritmo fallisca, ad esempio per l'assenza di variabili di batch valide, per matrici di covarianza singolari dovute a selezioni errate o problemi numerici intrinseci ai dati, l'eccezione viene intercettata. Il widget invoca il modulo nativo di gestione degli errori (`self.Error.general_error(str(e))`), che notifica l'utente tramite un indicatore visivo. Contestualmente, la variabile contenente i dati viene forzata a `None` e inviata al canale `Outputs.data.send()`. Questo pattern garantisce che la propagazione di dati corrotti venga interrotta immediatamente, preservando l'integrità del flusso di lavoro complessivo.

4.5 Interfaccia grafica

Il tassello finale della fase implementativa di un widget è la realizzazione dell'interfaccia grafica. Quasi ironicamente, gli aspetti progettuali legati a quest'ultima fase sono i primi ad avere un impatto significativo per l'utilizzo dal punto di vista dell'utente. Occorre dunque non trascurare alcun dettaglio che possa generare confusione nell'utilizzatore o scarsa intuitività dell'uso predefinito. Tale fase, nel contesto di lavoro descritto, si è concretizzata con l'inserimento nel widget di:

- **infobox "Info":** piccola sezione posizionata nella parte alta della finestra, in cui si è deciso di fornire una breve panoramica riassuntiva delle caratteristiche del dataset in ingresso (che può fungere anche da controllo per l'utente). In particolare, sono forniti il numero di cellule, il numero di geni e la quantità di features rappresentanti metadati;

- **infobox "Parameters":** posizionato nella zona centrale dell'interfaccia del widget, è il menù che racchiude i parametri regolabili dall'utente, ne mostra il valore e ne dà possibilità di modifica;
- **checkbox "Batch Variable Selection":** area in cui vengono stampate a schermo le variabili meta selezionabili come variabili di batch e il numero dei batch stessi. L'utilizzatore ha la possibilità di selezionare una o più variabili.

In luce di quanto detto nel capitolo riguardante l'ambiente Orange e la struttura dei widget, ogni altro aspetto non esplicitamente citato è da ricondursi al principio di ereditarietà dalla classe che funge da scheletro per ciascun widget.

Esempio di utilizzo in Orange

Nella figura seguente è mostrato un possibile utilizzo del widget Harmony, oltre all'interfaccia del widget stesso (Figura 4.1). Inoltre, sono mostrate le rappresentazioni di un dataset esempio (HP, si veda il capitolo legato alla descrizione dei dataset) precedenti e successive alla correzione con Harmony (Figura 4.2), estratte dal workflow in Figura 4.1.

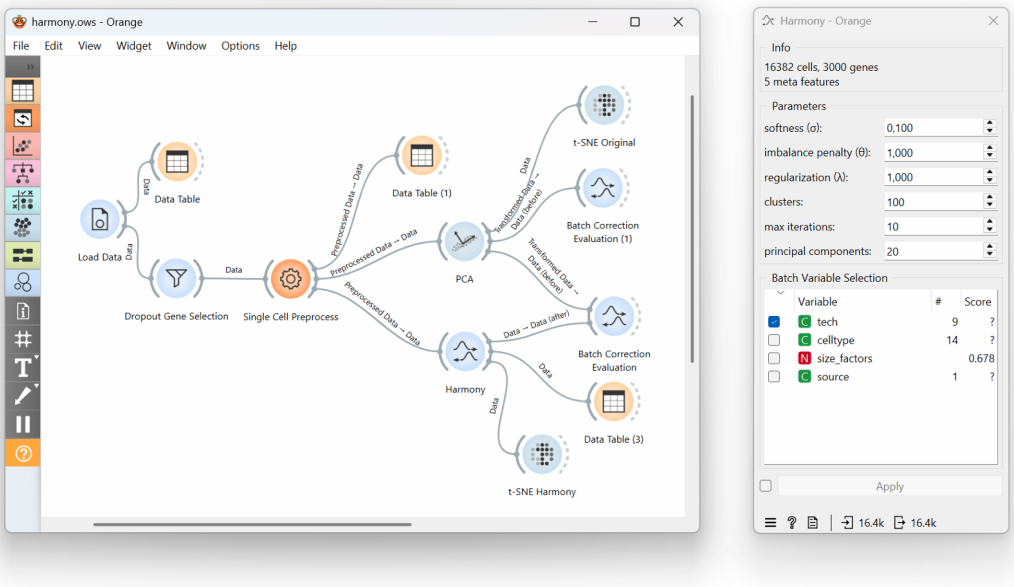
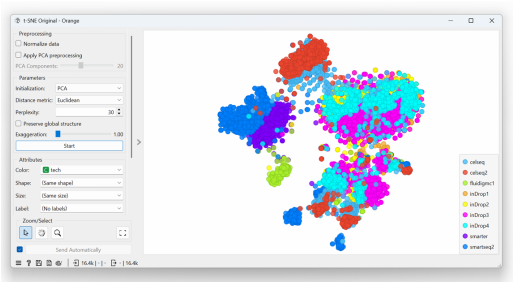
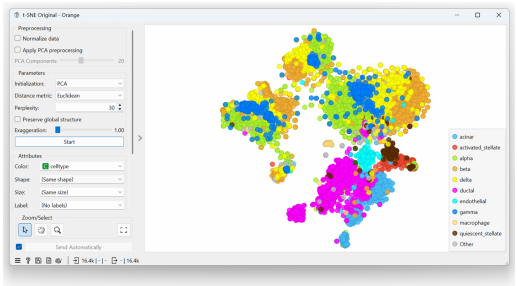


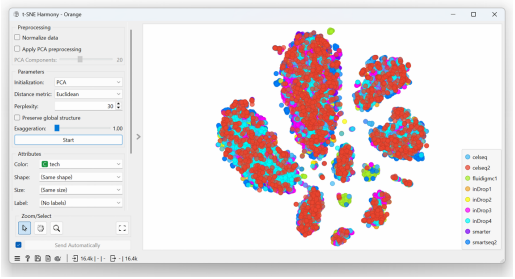
Figura 4.1: Esempio di flusso di lavoro per un'analisi scRNA-seq in Orange. Sulla destra si può osservare come appare l'interfaccia grafica del widget Harmony all'utente finale.



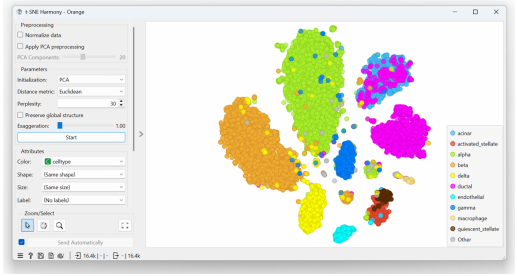
(a) Rappresentazione t-SNE dei dati HP colorati per batch.



(b) Rappresentazione t-SNE dei dati HP colorati per tipo cellulare.



(c) Rappresentazione t-SNE dei dati HP corretti da Harmony colorati per batch.



(d) Rappresentazione t-SNE dei dati HP corretti da Harmony colorati per tipo cellulare.

Figura 4.2: Rappresentazione grafica in Orange dei dati HP originali e corretti con il widget Harmony.

Capitolo 5

Confronto tra Harmony e il metodo pre-implementato

5.1 Il metodo di batch correction in Orange

Il modulo per la correzione degli effetti di batch precedentemente integrato in Orange è basato su un classico approccio di regressione lineare multipla univariata. A differenza di metodi più recenti e complessi che operano in spazi multivariati (come Harmony), questo algoritmo interviene direttamente sulla matrice dei conteggi, analizzando l'espressione di un singolo gene alla volta per stimare e sottrarre il bias tecnico.

Dal punto di vista matematico, il metodo converte inizialmente le meta-variabili tecniche (le etichette dei batch) in una matrice Z, costruita tramite una codifica one-hot (variabili dummy), affiancata da una prima colonna di valori unitari che funge da termine di intercetta, rappresentante l'espressione basale teorica in assenza di perturbazioni tecniche:

$$Z = [1, Z_{batch}].$$

Sia Y_j il vettore colonna contenente i livelli di espressione del gene j-esimo misurati attraverso tutte le cellule del dataset. L'algoritmo modella l'espressione genica come combinazione lineare delle covariate tecniche. I coefficienti di regressione W_j (che quantificano il peso dell'effetto di batch specifico per quel gene) vengono stimati minimizzando l'errore quadratico medio tramite il metodo dei Minimi Quadrati Ordinari (OLS):

$$\hat{W}_j = (Z^\top Z)^{-1} Z^\top Y_j. \tag{5.1}$$

Una volta calcolato il vettore dei pesi $\hat{W}_j$, il contributo tecnico perturbativo stimato per ciascuna cellula è dato dal prodotto $Z\hat{W}_j$. La fase di correzione vera e propria si concretizza sottraendo questo rumore additivo dai dati originali:

$$Y_j^{(corretto)} = Y_j - Z\hat{W}_j \tag{5.2}$$

Questo approccio si rivela intrinsecamente globale e rigido: la correzione stimata $\hat{W}_j$ per un determinato batch viene sottratta in egual misura a tutte le cellule di quel lotto, ignorando la loro reale identità biologica. Tale limite giustifica la ricerca e l'implementazione di metodi più moderni, localmente adattivi, capaci di calibrare l'entità della correzione in base ai cluster cellulari (come avviene in Harmony) o a sottostrutture dello spazio dei dati.

5.2 Risultati del confronto

In questa sezione sono presentati i risultati del confronto diretto tra il nuovo metodo implementato, Harmony, e lo strumento di integrazione dei batch nativamente presente all'interno dell'Add-On Single Cell di Orange. L'obiettivo in questa fase è comparare esplicitamente le performance dei due metodi in un reale scenario d'uso, operando sui medesimi 8 dataset pubblici, al fine di dimostrare le intuizioni da cui si è partiti per lo sviluppo di questo lavoro e validare definitivamente la scelta di Harmony.

Le metriche di valutazione analizzate sono coerenti con il resto del lavoro e mantengono la loro interpretazione canonica: per kBET, iLISI, cLISI, ARI e NMI valori prossimi a 1 indicano le performance ideali, mentre per l'AUC il punteggio ottimale, associato a un'assegnazione del batch randomica, è pari a 0.5. Le statistiche descrittive (media e deviazione standard) registrate per i due metodi sono esposte nella Tabella 5.1.

Essendo il confronto articolato su due soli gruppi appaiati, la valutazione inferenziale è stata effettuata mediante il test non parametrico dei ranghi con segno di Wilcoxon, come per il confronto dei metodi durante la fase iniziale di benchmarking, ma, in questo scenario, non prevedendo confronti multipli per la singola metrica, i p-value calcolati non necessitano di ulteriori fattori di correzione. I risultati dell'analisi statistica, valutati con un livello di significatività $\alpha = 0.05$, sono riportati nella Tabella 5.2.

Tabella 5.1: Statistiche descrittive delle metriche di valutazione. I valori sono espressi come media ± deviazione standard per i dati elaborati con il nuovo metodo (Harmony) e con lo strumento basato su regressione, già presente (Orange).

Metrica	**Harmony**	**Orange**
kBET	0.0969 ± 0.2011	0.0557 ± 0.1546
iLISI	0.3318 ± 0.2001	0.1621 ± 0.2161
cLISI	0.9775 ± 0.0164	0.9645 ± 0.0454
ARI	0.5025 ± 0.1513	0.4125 ± 0.0860
NMI	0.6549 ± 0.1087	0.5774 ± 0.1204
AUC	0.6851 ± 0.0865	0.4618 ± 0.0386

Tabella 5.2: Risultati del test dei ranghi con segno di Wilcoxon. I p-value valutano la significatività della differenza tra le performance di Harmony e di Orange. I valori statisticamente significativi ($p \leq \alpha = 0.05$) sono evidenziati con l'asterisco (*).

Metrica	**Wilcoxon (p-value)**
kBET	0.0277^*
iLISI	0.0156^*
cLISI	0.6406
ARI	0.0547
NMI	0.0234^*
AUC	0.0078^*

L'analisi integrata dei risultati giustifica pienamente l'implementazione del nuovo widget, dimostrando in modo statisticamente solido come Harmony garantisca prestazioni sensibilmente superiori a quelle dello strumento nativo.

Per quanto concerne la mitigazione del bias tecnico a livello locale, il test di Wilcoxon rileva miglioramenti significativi a favore di Harmony sia per l'indice iLISI ($p = 0.0156$) che per kBET ($p = 0.0277$). Il raddoppio della media di iLISI (0.3318 contro 0.1621) certifica che il nuovo algoritmo riesce a costruire vicinati cellulari molto più omogenei dal punto di vista del mescolamento dei batch rispetto al metodo di base di Orange.

Ancora più marcato è il vantaggio in termini di preservazione della struttura biologica: Harmony supera Orange per qualità dei cluster biologici post-integrazione. L'informazione mutua normalizzata (NMI) risulta significativamente superiore (0.6549 contro 0.5774, con $p = 0.0234$), e parallelamente l'Adjusted Rand Index (ARI) registra un forte incremento medio (0.5025 contro 0.4125), attestandosi al limite della significatività statistica ($p = 0.0547$). Infine, la metrica cLISI, pur non presentando differenze statisticamente significative ($p = 0.6406$), si mantiene su valori medi prossimi all'unità per entrambi i metodi.

Un'anomalia apparente, su cui vale la pena riflettere, in luce anche dei punteggi sulle metriche già analizzate, risiede nei punteggi di AUC. Lo strumento implementato in Orange restituisce un valore medio di 0.4618 (statisticamente più vicino all'ideale 0.5 rispetto allo 0.6851 di Harmony, $p = 0.0078$), con punteggi che, quasi nella totalità dei casi, risultano inferiori al valore di 0.5. Alla luce dei peggioramenti contemporanei di Orange nelle metriche biologiche (NMI, ARI), questo punteggio riflette chiaramente una dinamica di overcorrection globale: lo strumento base applica traslazioni così aggressive da ingannare totalmente il classificatore del batch, ma al prezzo di mutare negativamente anche l'eterogeneità biologica autentica. Al contrario, Harmony agisce in modo localmente mirato: offre un eccellente compromesso, unendo un mescolamento locale rigoroso (iLISI e kBET elevati) a una conservazione ottimale dei segnali biologici (NMI, ARI e cLISI) e rimuovendo una parte sostanziale dell'informazione legata al batch nei dati (AUC).

Un quadro riassuntivo e visivo dell'andamento sugli scenari analizzati è illustrato mediante le distribuzioni a boxplot (Figura 5.1) e le heatmap dei rank (Figura 5.2).

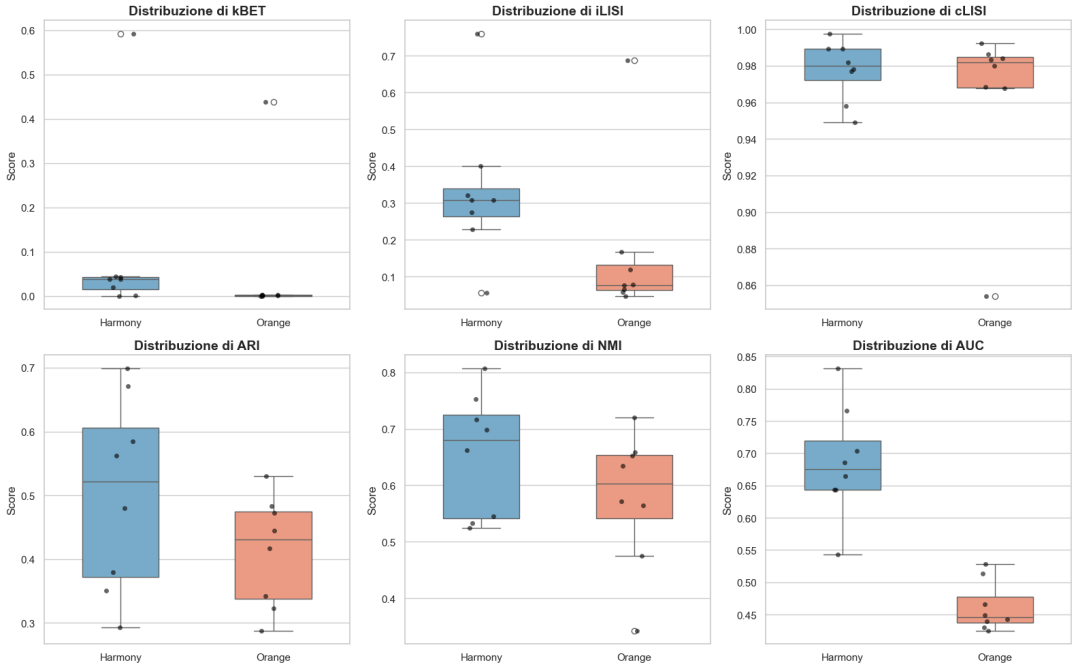


Figura 5.1: Boxplot delle sei metriche valutate (kBET, iLISI, cLISI, ARI, NMI e AUC) sui due scenari operativi (Harmony e Orange) per gli 8 dataset analizzati.

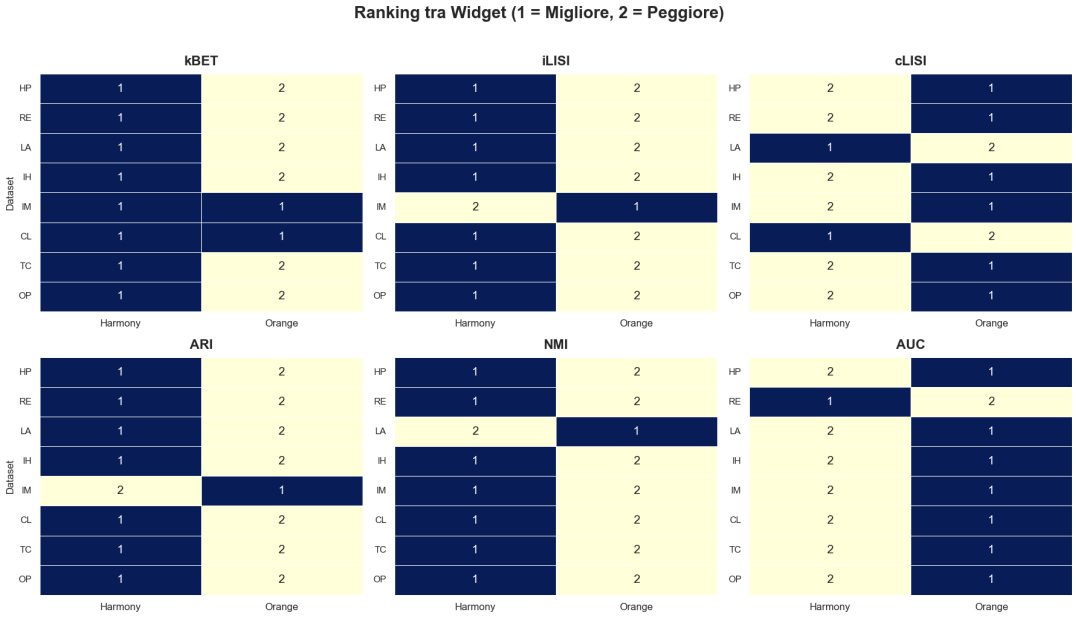


Figura 5.2: Ranking relativo dei due strumenti di integrazione per ciascun dataset (1 = performance migliore, 2 = performance peggiore). Per l'AUC, il rank migliore indica la minor distanza dal valore ideale di predizione causale pari a 0.5.

Capitolo 6

Batch Correction Evaluation widget

Durante il proseguo del lavoro presso il laboratorio BioLab, sono emerse ulteriori possibilità di ampliamento dell'Add-On Single Cell. Restando vicini concettualmente a quanto svolto per il widget Harmony, si è optato per sviluppare un nuovo widget che arricchisse il panorama di funzioni connesse al fenomeno degli effetti di batch. In questo capitolo verrà illustrato il processo che ha portato allo sviluppo di tale widget, spaziando dalle motivazioni all'implementazione del codice alla base.

6.1 Motivazioni

Sebbene frequentemente si incontrino, durante un'analisi di dati scRNA-seq, situazioni in cui il fenomeno degli effetti di batch è evidente e ben marcato, non bisogna pensare che la modalità corretta di approccio sia correggere qualsiasi dataset passi sotto mano, alla cieca. Occorre dunque rendere consapevole l'utente intenzionato ad avvicinarsi a questo tipo di tecniche riguardo al fatto che il machine learning non sia una formula magica di cui abusare, bensì talvolta risulta addirittura dannoso, se applicato a dataset che inizialmente non risentono di alcun fenomeno di batch effect (fenomeni di overcorrection in cui il metodo inventa l'effetto di batch). In letteratura Harmony, occorre sottolinearlo, è indicato come poco propenso all'overcorrection [10]. Tuttavia, l'idea scaturita dopo aver realizzato e ottimizzato il widget Harmony è stata quella di progettare uno strumento che potesse aiutare l'utente a valutare se ci sia concretamente la necessità di un'eventuale correzione del batch effect. In altre parole, un widget che cerchi di diagnosticare la condizione, in termini di batch effect, del dataset. Inizialmente è stato dunque sviluppato un widget a questo proposito.

Successivamente ci si è accorti che, in un utilizzo naturale del widget in fase sperimentale, la tendenza fosse quella di collegare un'unità del nuovo widget ai dati non corretti e, in aggiunta, anche a una medesima a quelli corretti. Ciò veniva fatto con l'intenzione di valutare lo stato del dataset corretto e, quindi, la bontà della correzione. L'evoluzione del prototipo di widget ha dunque previsto l'integrazione della possibilità di di effettuare un confronto quantitativo tra i due dataset (origi-

nale e corretto) per determinare la qualità della correzione, previa connessione di un secondo input con il dataset corretto, introducendo così una duplice modalità di utilizzo.

6.2 Specifiche del widget

Il nuovo widget, denominato Batch Correction Evaluation, ha una complessità computazionale sicuramente di gran lunga inferiore a quanto richiesto per Harmony, in quanto non implementa nessun algoritmo e, di conseguenza, non richiede moduli di preprocess e gestione performante della parte computazionale. Tuttavia, occorre comunque avere l'accortezza di rispettare i paradigmi legati all'ecosistema Orange, di cui si è discusso in questo elaborato. Le specifiche, dunque, devono essere semplici ma concrete, con lo sguardo sempre rivolto all'utilizzatore finale.

6.2.1 Input e Output

Come per il caso di Harmony, definire input e output è fondamentale per lo sviluppo del widget. Essi costituiscono infatti la prima informazione che viene fornita all'utente quando si redige la documentazione per l'utilizzo del widget. Nel caso in oggetto, il widget accetta due possibili input: `Data (before)` e `Data (after)`. Essi sono entrambi due matrici rappresentanti dataset scRNA-seq, siano essi embedding a dimensione ridotta o dataset a dimensionalità piena. Il widget richiede almeno uno dei due input connessi e necessita tassativamente, in caso di connessione di entrambi gli input, che questi ultimi abbiano la medesima dimensione.

L'output fornito dal widget consiste, invece, in una tabella mono riga in cui vengono riportate alcune metriche quantitative, di cui si discuterà in questo capitolo.

6.2.2 Parametri gestibili dall'utente

L'aspetto interattivo del widget in questione consiste nel dare all'utente la possibilità di selezionare e modificare i parametri fondamentali.

Come illustrato nel paragrafo successivo, la maggior parte delle metriche calcolate si basa sul concetto di vicinato locale di una cellula. Si è dunque deciso di implementare la possibilità per l'utilizzatore di variare il valore del parametro k, che rappresenta il numero di vicini considerati per ciascuna cellula durante il calcolo delle metriche. Tale decisione ha motivazioni legate soprattutto alla possibilità di utilizzare dataset con dimensioni radicalmente differenti, vista la buona capacità di scalare del metodo Harmony.

Il secondo aspetto interattivo del widget è, logicamente, legato alla possibilità di scegliere la variabile di batch e, se presente, la variabile che rappresenta il tipo cellulare (o, più in generale, la classe di appartenenza).

6.2.3 Metriche implementate

Nella presente sezione sono illustrate le metriche implementate. Come già sottolineato in precedenza, si è cercato di ottenere una valutazione completa dello scenario in esame, secondo paradigmi e punti di vista tra loro differenti. Chiaramente ci si soffermerà maggiormente sulle metriche non ancora trattate in questo elaborato, mentre si rimanda ai capitoli precedenti per un eventuale approfondimento sulle statistiche già utilizzate in precedenza. In particolare, soprattutto per l'utilizzo del widget a singolo input, quindi con l'obiettivo di valutare lo stato del dataset in termini di impatto dell'effetto di batch, si è deciso di implementare iLISI, cLISI e AUC in maniera invariata rispetto al capitolo 3.

Sono state, inoltre, introdotte quattro metriche comparative, appositamente pensate per il confronto tra dataset originario e corretto.

iLISI: before, after e improvement

Come già introdotto in precedenza, nella fase di confronto dei metodi di batch effect correction sui dataset presi in esame, l'indice `iLISI` è una metrica che si propone di rappresentare il grado di mescolamento locale dei batch. Nel contesto del widget Batch Correction Evaluation, `iLISI` è stato implementato in tre differenti varianti:

- **iLISI before:** indice calcolato sul dataset non corretto;
- **iLISI after:** indice calcolato sul dataset corretto;
- **iLISI improvement:** metrica normalizzata nell'intervallo $[0, 1]$, che rappresenta il miglioramento percentuale di `iLISI after` su `iLISI before`, rispetto al massimo miglioramento possibile. Tale scenario è quello in cui `iLISI after` sale a 1, con l'aggiunta di un termine ε per ragioni di stabilità numerica:

$$\text{iLISI\_improvement} = \frac{\text{iLISI}_{\text{after}} - \text{iLISI}_{\text{before}}}{1 - \text{iLISI}_{\text{before}} + \varepsilon}. \tag{6.1}$$

cLISI: before, after e retention

Similmente a quanto descritto per `iLISI`, sono state definite tre differenti interpretazioni di `cLISI` (rappresentante il grado di separazione cellulare locale), anch'esso già introdotto in precedenza:

- **cLISI before:** indice calcolato sul dataset non corretto;
- **cLISI after:** indice calcolato sul dataset corretto;
- **cLISI retention:** metrica normalizzata nell'intervallo $[0, 1]$, che calcola un'eventuale ritenzione dell'indice `cLISI after` rispetto a `cLISI before` in seguito alla correzione, mediante il rapporto tra esse:

$$\text{cLISI\_retention} = \frac{\text{cLISI}_{\text{after}}}{\text{cLISI}_{\text{before}} + \varepsilon}. \tag{6.2}$$

Neighbor Overlap

`Neighbor Overlap` è una metrica nuova, non ancora introdotta in questo elaborato. Essa è stata implementata poiché si è definita la necessità di comprendere quanto invasiva fosse la correzione del metodo. In altre parole, per arricchire la descrizione del contesto, si è cercato di trovare un indice che non rispondesse più alle domande "quanto bene sono mischiati i batch?" o "quanto bene sono separati i tipi cellulari?", bensì "quanto è stata forte la correzione?" o "quanto l'algoritmo di batch correction ha modificato la posizione dei dati originali?". Questo indice è, dunque, pensato per essere utilizzato in combinazione con una valutazione del batch mixing e della separazione biologica, per diagnosticare eventuali fenomeni di overcorrection. L'idea è quindi che se `Neighbor Overlap` ci dice che la correzione è stata piuttosto invasiva, posso ricadere principalmente in due scenari, distinguibili con l'aiuto delle altre metriche presenti:

- il problema era complesso e l'effetto di batch molto marcato (`iLISI before` basso), dunque è stata necessaria una forte correzione. Vedo quindi che `cLISI` è rimasto circa invariato (o è migliorato);
- l'algoritmo ha interpretato male lo scenario e corretto più del dovuto: vedo un netto calo di `cLISI`, indice di overcorrection.

L'indice in oggetto è definito, dunque, sfruttando il concetto di similarità di Jaccard. Esso misura la stabilità della struttura locale tra dataset originario e corretto. Per ciascuna cellula i, si considerano gli insiemi di k vicini:

$$\mathcal{N}_k^{\text{before}}(i), \qquad \mathcal{N}_k^{\text{after}}(i),$$

e si calcola la similarità di Jaccard:

$$J(i) = \frac{\left|\mathcal{N}_k^{\text{before}}(i) \cap \mathcal{N}_k^{\text{after}}(i)\right|}{\left|\mathcal{N}_k^{\text{before}}(i) \cup \mathcal{N}_k^{\text{after}}(i)\right|}. \tag{6.3}$$

Il valore globale è la media:

$$\text{neighbor\_overlap} = \frac{1}{n}\sum_{i=1}^{n} J(i). \tag{6.4}$$

Interpretazione:

- ≈ 1: i vicinati sono quasi invariati (correzione "gentile");
- ≈ 0: la struttura locale è fortemente modificata (possibile distorsione).

Predizione del batch con regressione logistica (AUC): before, after e drop

Similmente ai casi legati a `iLISI` e `cLISI`, anche per la metrica `AUC`, già utilizzata e descritta nelle fasi precedenti del lavoro, è stata proposta una triade di valori:

- **AUC before:** indice calcolato sul dataset non corretto;
- **AUC after:** indice calcolato sul dataset corretto;

- **AUC drop:** metrica che calcola banalmente il decremento dell'AUC in seguito alla correzione dell'effetto di batch:

$$\mathrm{AUC}_{\mathrm{drop}} = \mathrm{AUC}_{\mathrm{before}} - \mathrm{AUC}_{\mathrm{after}}. \tag{6.5}$$

6.2.4 Modalità ad uno o due input

Come introdotto nel capitolo in oggetto, il widget prevede una duplice modalità di funzionamento. La modalità a singolo input è pensata per valutare la condizione del singolo dataset. Implementa infatti solamente tre metriche, già citate e utilizzate in altre sezioni dell'elaborato: iLISI, cLISI e AUC. Il significato è quello di fornire una panoramica sullo stato del dataset, ovvero quanto i batch sono mischiati, in che misura i tipi cellulari sono separati e la predicibilità del batch a partire dai dati, in modo da fornire una descrizione della situazione sotto differenti aspetti.

Nel funzionamento a due input, invece, lo scenario viene arricchito. Le metriche implementate passano a dieci. Sebbene un numero così elevato di statistiche a primo impatto possa risultare confusionario, lo scenario è descritto, almeno in parte, con gli stessi strumenti: iLISI before e iLISI after sul dataset prima e dopo la correzione, cLISI before e after, AUC before e after. Queste metriche sono lasciate come riferimento all'utente per valutare anche lo stato dei dataset presi singolarmente. Le statistiche che racchiudono l'essenza del confronto, tuttavia, sono quelle di natura comparativa: iLISI improvement, cLISI retention, LogReg AUC drop e neighbor overlap. Tali valori rappresentano quantitativamente il miglioramento (o il peggioramento) del dataset dovuto alla correzione secondo angoli di visione radicalmente differenti l'uno dall'altro.

La scelta effettuata in termini di metriche presentate predilige lasciare all'utente gli strumenti per un'interpretazione quantitativa ma mirata alla comprensione dello scenario, piuttosto che fornire un unico score che combini tali metriche per fornire un freddo "voto" alla performance dell'algoritmo. Esistono infatti molteplici scenari, in cui l'utilizzatore potrebbe avere obiettivi diversi per situazioni diverse (e.g. minimizzare la predicibilità del batch, oppure massimizzare la separazione tra tipi cellulari).

6.3 Dettagli implementativi

Sebbene il widget Batch Correction Evaluation condivida con il widget Harmony alcune logiche di base legate all'ecosistema Orange (come la gestione dei contesti e la cattura strutturata delle eccezioni), la sua natura di strumento di valutazione, piuttosto che di elaborazione dei dati (come era per Harmony), ha imposto sfide architetturali differenti.

6.3.1 Gestione flessibile dell'input e validazione

Una delle principali differenze strutturali rispetto al widget Harmony è la necessità di gestire flussi di dati simultanei, supportando sia una modalità a singolo input, sia una modalità comparativa a doppio input. La classe principale instrada questi flussi tramite metodi dedicati, adattando quindi in maniera dinamica le metriche da calcolare.

Poiché nella modalità comparativa (a due input) il widget valuta l'effetto della correzione, è fondamentale che i due dataset siano semanticamente e dimensionalmente sovrapponibili. A tale scopo, la routine di esecuzione all'interno del codice verifica preventivamente che le due matrici abbiano lo stesso numero di campioni (cellule) e solleva un'eccezione esplicita qualora vi sia una discrepanza. Questo meccanismo di validazione arresta il processo a monte delle successive fasi di elaborazione, consentendo quindi di prevenire l'allocazione di risorse inutili.

6.3.2 Rappresentazione grafica adattiva

Nel widget Harmony, l'interfaccia grafica è stata arricchita con delegati custom per il semplice allineamento del testo. Per il widget di valutazione, invece, l'esigenza era mostrare i risultati delle metriche (tutte normalizzate nell'intervallo $[0, 1]$) in modo visivamente intuitivo. A tal fine è stato sviluppato il `BarDelegate`, una classe derivata da `QStyledItemDelegate`. Questa classe sovrascrive sia il metodo `sizeHint` (per garantire un'altezza ottimale delle righe), sia il metodo di disegno nativo `paint`, sfruttando l'oggetto `QPainter` della libreria Qt. Tramite la definizione di due nuovi ruoli per il passaggio dei dati (`BarValueRole` e `BarColorRole`), il delegato intercetta il valore numerico della metrica e disegna una barra orizzontale (`QRectF`) la cui larghezza è strettamente proporzionale al punteggio dell'indice considerato.

Un altro aspetto rilevante è l'implementazione di una logica di colorazione per le barre, calcolata dal back-end tramite il metodo `_bar_style` e passata all'interfaccia. Il significato dei diversi colori sarà esplicitato nel paragrafo legato all'interfaccia grafica del widget.

6.3.3 Integrazione con librerie di Machine Learning standard

Mentre per lo sviluppo del widget Harmony la scelta architetturale è ricaduta su un'implementazione basata esclusivamente sulla libreria `NumPy`, al fine di evitare l'introduzione di dipendenze esterne specifiche quali `pandas` o `AnnData`, per il widget di valutazione si è optato per un approccio differente, facendo leva sulla libreria `scikit-learn`.

Tale scelta è giustificata da due motivazioni principali: in primo luogo, a differenza dei pacchetti bioinformatici (particolarmente settoriali), `scikit-learn` non costituisce una dipendenza esterna, bensì una libreria nativa e strutturale dell'ecosistema Orange, che la utilizza come motore computazionale per la quasi totalità dei suoi nodi standard di classificazione e regressione. Di conseguenza, il suo impiego non

appesantisce il processo di installazione dell'Add-On e non introduce alcun rischio di conflitti di versione per l'utente finale.

In secondo luogo, `scikit-learn` opera direttamente con strutture di tipo array `NumPy`. Questo aspetto è fondamentale, poiché permette un'integrazione banale con l'oggetto `Table` di Orange, senza dovere convertire il dataset in strutture dati complesse e radicalmente differenti come `AnnData` (processo che andrebbe ad intaccare le prestazioni del widget).

6.3.4 Costruzione dinamica dell'output

L'output di questo widget differisce radicalmente dai nodi di preprocessing convenzionali, come il widget Harmony. Non restituisce un dataset di cellule trasformato, bensì un "metadato" riassuntivo. Per rendere questo output compatibile con i nodi a valle di Orange (ad esempio, per salvare i risultati su un file CSV tramite il widget Save Data), i risultati calcolati nel dizionario interno devono essere convertiti in un oggetto `Table`.

Il widget implementa quindi una routine di generazione del nuovo dominio. Iterando sui risultati ottenuti, la funzione interna `_safe_var_name` prende i nomi delle metriche e per ognuna istanzia un nuovo oggetto `ContinuousVariable`. Viene quindi creato un `Domain` ex novo, popolato con una singola riga di valori numerici (come citato in precedenza). Questo approccio permette all'output di integrarsi perfettamente nel flusso di lavoro di Orange.

6.4 Interfaccia grafica

Anche per il widget Batch Correction Evaluation, così come per ogni widget Orange, l'interfaccia grafica assume rilevante importanza, visto il paradigma sul quale è sviluppato il software stesso.

Per il widget in oggetto, gli elementi hanno un comportamento dinamico, che può variare a seconda dell'utilizzo a singolo o doppio input. In particolare, come si può osservare in Figura 6.2, la configurazione completa si articola in:

- **infobox "Info":** similmente ad Harmony, viene mostrato un piccolo riassunto delle informazioni principali in alto a sinistra, con numero di geni e di cellule per entrambi i dataset (funge anche da controllo per l'utente);
- **infobox "Parameters":** anche in questo caso, analogamente ad Harmony, vi è una sezione dedicata alla modifica dei parametri (in questo caso solamente il numero dei vicini locali);
- **checkbox "Batch Variable":** come intuibile dal nome stesso, è reso disponibile un checkbox dove l'utilizzatore può selezionare le variabili di batch;
- **checkbox "Cell Type Variable":** analogamente, si è predisposto anche un checkbox dove l'utente può selezionare la variabile riguardante il tipo cellulare;

- **main area "Results":** nell'area principale del widget sono riportati i risultati delle metriche calcolate. Come introdotto in precedenza, è stato aggiunto un codice colore per consentire all'utente di destreggiarsi a colpo d'occhio fra gli indicatori riportati dal widget:
  - **azzurro:** assegnato alle metriche di stato assoluto (e.g. `iLISIbefore/after`, `AUC before/after`), che fotografano il dataset senza esprimere un giudizio comparativo;
  - **verde:** assegnato alle metriche di successo (e.g. `iLISI improvement`, `cLISI retention`, `AUC drop`), dove un riempimento maggiore indica che l'algoritmo di integrazione ha raggiunto il suo traguardo in modo efficace;
  - **arancione:** assegnato alle metriche diagnostiche (`Neighbor Overlap`), che fungono da campanello d'allarme visivo per indicare una potenziale alterazione strutturale eccessiva (overcorrection).

Nelle figure seguenti (Figura 6.1 e Figura 6.2) è mostrata l'interfaccia grafica del widget integrato in un flusso di lavoro che prevede l'utilizzo a doppio input. In Figura 6.3 è invece mostrata la versione diagnostica, a singolo input (in riferimento sempre al flusso di lavoro mostrato in Figura 6.1).

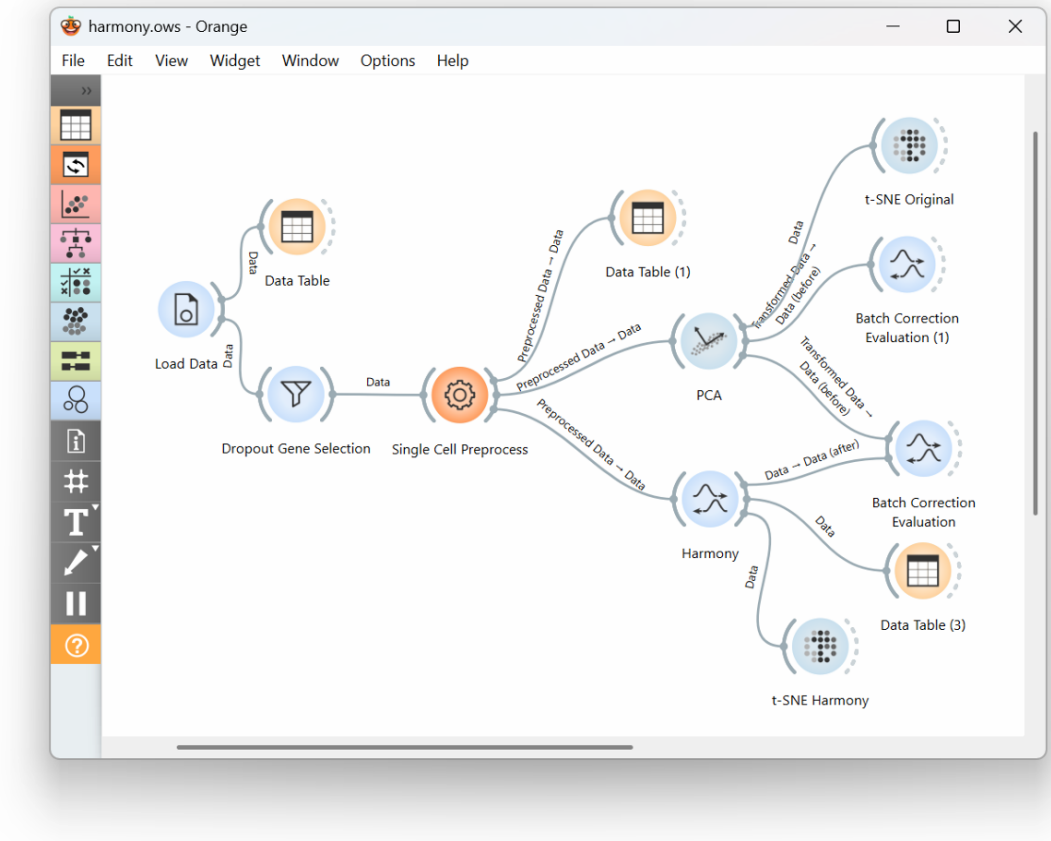


Figura 6.1: Flusso di lavoro in cui viene integrato il widget Batch Correction Evaluation sia in modalità a singolo input (indicata in Figura con (1)) che a doppio input.

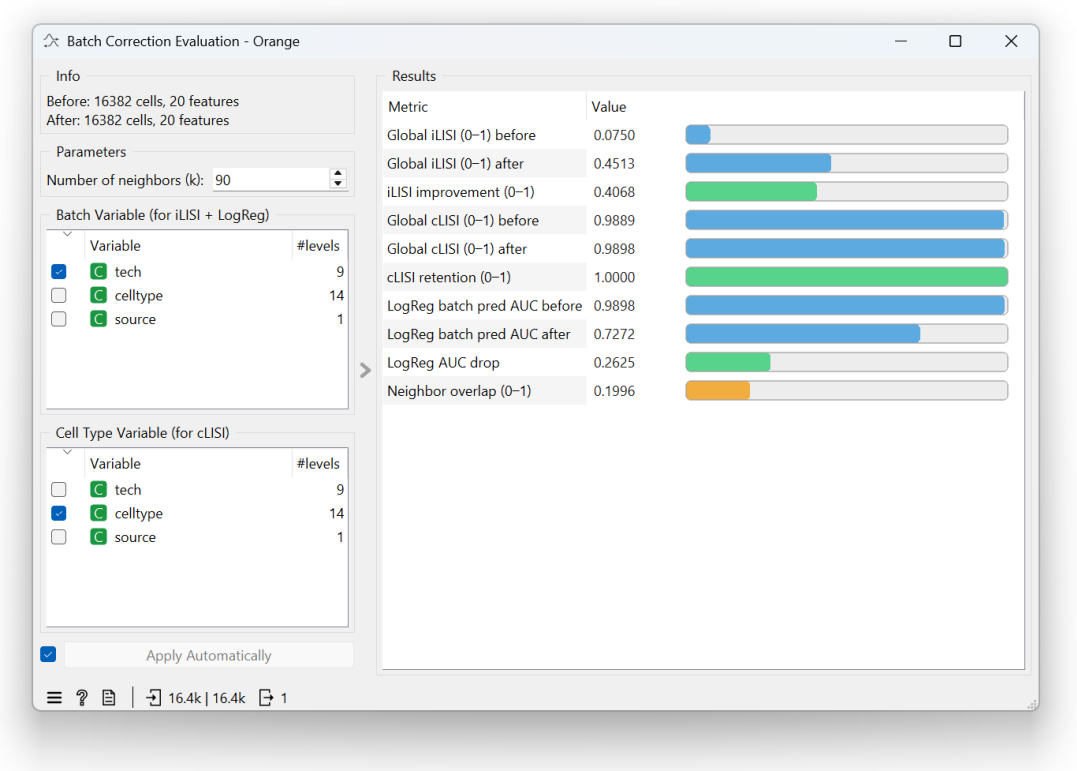


Figura 6.2: Interfaccia grafica del widget Batch Correction Evaluation, utilizzato in modalità doppio input.

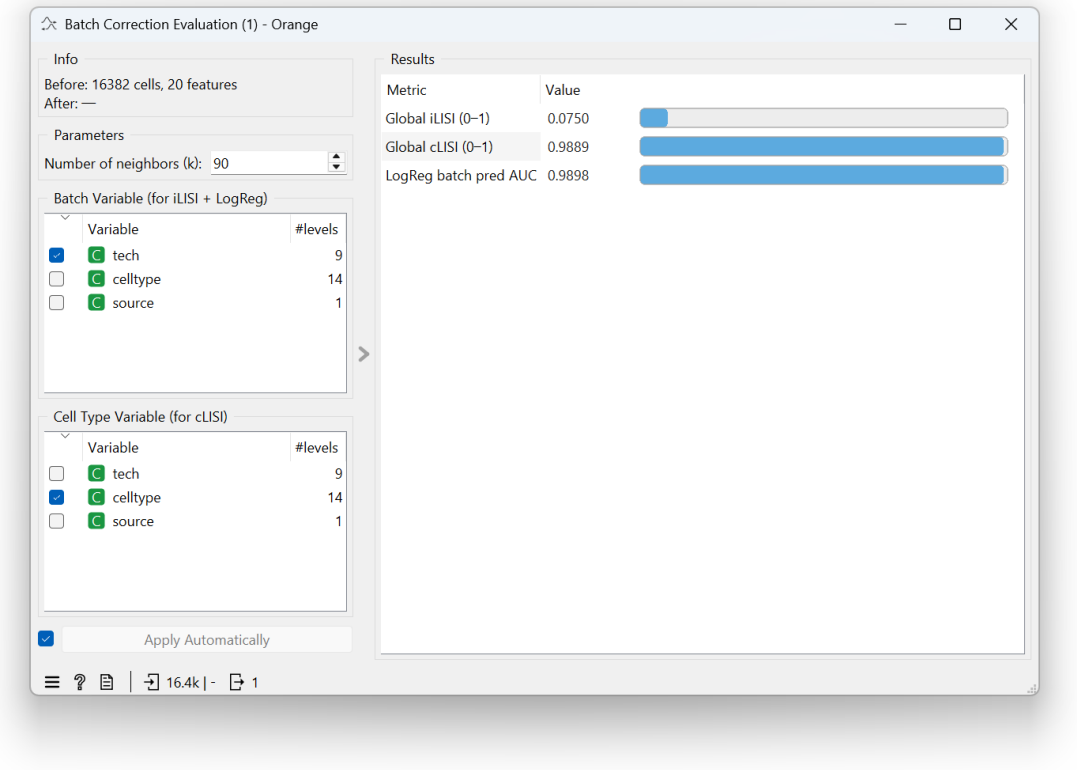


Figura 6.3: Interfaccia grafica del widget Batch Correction Evaluation, utilizzato in modalità singolo input.

Capitolo 7

Considerazioni finali

7.1 Limiti del lavoro

Durante l'esperienza presso BioLab all'Università di Lubiana, di cui si è trattato in questo elaborato, sono stati raggiunti gli obiettivi concordati in termini di nuove funzionalità da implementare in Orange, arricchite da argomentazioni e validazioni a sostegno di esse. Tuttavia, ogni applicazione ingegneristica presenta intrinsecamente delle limitazioni.

Nel contesto di questo lavoro, le limitazioni principali associate al progetto risiedono principalmente in quattro ambiti distinti:

- **limitazioni intrinseche di Harmony:** Harmony, come qualsiasi altro algoritmo di machine learning, racchiude in sé compromessi imputabili alla logica stessa dell'algoritmo. Seppur effettuando correzioni locali (al contrario, per esempio, di ComBat o del metodo precedentemente implementato in Orange), Harmony modella l'effetto di batch nei cluster con una regressione lineare. In condizioni di effetto di batch fortemente non lineare l'algoritmo risulta quindi inadeguato (alcuni approcci basati su deep learning funzionano meglio in questi casi). Tuttavia, va detto che quest'ultimo è uno scenario piuttosto complesso e non frequente, e dunque è al di fuori del contesto di impiego previsto per Orange;
- **gestione della memoria in Orange Single Cell:** come sottolineato all'interno dell'elaborato, Orange è strutturato per lavorare con matrici dense. In ambito scRNA-seq, tuttavia, la matrice di espressione genica è tipicamente fortemente sparsa. L'Add-On Single Cell opera, tuttavia, utilizzando la struttra dati base di Orange (basata su matrici dense), compromettendo quindi l'efficienza in presenza di dataset di dimensione onerosa;
- **influenza dei parametri sui risultati:** sebbene concedere all'utente la possibilità di modificare i parametri nei widget implementati sia stata una scelta ragionata ed esplicitamente voluta, essa potrebbe nascondere insidie. L'utente deve dunque considerare il fatto che lo scostamento del valore di alcuni parametri dall'opzione di default potrebbe influenzare anche in maniera negativa

il risultato finale. In particolare, il numero di componenti principali scelte può avere effetti sulla prestazione dell'algoritmo [36], e il numero k dei vicini locali considerati nel calcolo delle metriche può influenzare il risultato delle stesse. Viene comunque implementata una selezione di default automatica delle componenti di Harmony;

- **correzione su embedding PCA:** da un punto di vista bioinformatico, lavorare su uno spazio PCA è un punto a favore per quanto riguarda efficienza e fluidità di lavoro, oltre ad essere ottimo per rappresentazioni grafiche (t-SNE o UMAP), clustering, traiettorie pseudotemporali e annotazione dei tipi cellulari. Tuttavia, è limitante per gli ambiti che fanno utilizzo esplicito della matrice di espressione genica nella sua forma originale (e.g. Differential Gene Expression). In luce di ciò, è stato comunque preservato il preesistente widget Batch Effect Removal.

7.2 Conclusioni

Il lavoro di tesi presentato ha affrontato la progettazione, lo sviluppo e la validazione di nuovi strumenti interattivi all'interno del software Orange, mirati a risolvere una delle problematiche più critiche nell'analisi dei dati di single-cell RNA sequencing: la correzione degli effetti di batch. Gli obiettivi raggiunti in questo progetto si articolano su tre direttrici principali, che uniscono l'accessibilità degli strumenti bioinformatici al rigore dell'ingegneria del software.

Democratizzazione del machine learning per l'analisi bioinformatica

Il primo e più evidente contributo consiste nell'aver partecipato attivamente a colmare un divario tecnologico e formativo significativo. L'utilizzo di algoritmi di integrazione allo stato dell'arte, come Harmony, è tipicamente appannaggio esclusivo di bioinformatici con solide competenze di programmazione in R o Python. L'implementazione di questo metodo all'interno dell'Add-On Single Cell di Orange ha permesso di abbattere tale barriera all'ingresso e di rendere fruibile tale strumento anche a chi si sta avvicinando e formando nel mondo della data science. Attraverso un'interfaccia di programmazione visuale intuitiva si può accedere ad analisi ad alta complessità, manipolando i dati in tempo reale senza la necessità di scrivere codice, favorendo un approccio inclusivo e interattivo.

Rigore ingegneristico e ottimizzazione

Dal punto di vista dello sviluppo software, il progetto mostra come sia possibile integrare algoritmi matematici e statistici complessi all'interno di un applicativo visuale mantenendo prestazioni di livello. Rinunciando a dipendenze esterne e implementando il motore di calcolo del widget Harmony in puro `NumPy`, è stata garantita una computazione rapida ed efficiente in termini di memoria, ad eccezione delle limitazioni intrinseche relative alla struttura dei dati in Orange. Inoltre, l'adozione di un'architettura software modulare, caratterizzata da un netto disaccoppiamento tra il front-end grafico, il middleware di gestione dei dati di Orange e il back-end

algoritmico, assicura la robustezza, la manutenibilità e la portabilità del codice sviluppato.

Approccio diagnostico e consapevolezza analitica

Infine, la filosofia alla base del lavoro si è discostata dalla semplice fornitura di una "scatola nera" per la correzione dei dati. Riconoscendo i rischi legati all'abuso degli algoritmi di machine learning, lo sviluppo del widget Batch Correction Evaluation ha dotato l'utente di uno strumento di diagnostica dei dati e dei risultati. Attraverso il calcolo di metriche avanzate (come il mixing dei batch, la retenzione della separazione biologica e il rischio di overcorrection), l'utente è ora messo nella condizione di valutare criticamente lo stato iniziale del dataset e la reale necessità di intervento, promuovendo un'analisi dei dati a singola cellula non solo tecnicamente avanzata, ma soprattutto consapevole e scientificamente rigorosa.

Complessivamente, l'implementazione congiunta di un modulo correttivo ad alte prestazioni e di uno strumento diagnostico dota l'Add-On Single Cell di una suite ora più completa, capace di guidare l'utente dall'identificazione del bias tecnico fino alla sua risoluzione e validazione.

7.3 Sviluppi futuri

7.3.1 Implementazione della struttura dati `AnnData`

Guardando alle prospettive di evoluzione per l'ecosistema Orange e per l'Add-On Single Cell, le possibilità di arricchimento del software sono molteplici. Un intervento prioritario, capace di ottimizzare radicalmente il flusso di lavoro sui dati scRNA-seq, consisterebbe nella reimplementazione dell'intero Add-On volta a supportare nativamente la struttura dati `AnnData`, attuale stato dell'arte in Python per l'analisi di dati scRNA-seq. Da un punto di vista computazionale, questa transizione consentirebbe di abbattere il consumo di memoria RAM sfruttando l'utilizzo di matrici sparse. L'oggetto `AnnData`, inoltre, funge da contenitore integrato capace di incapsulare simultaneamente l'intero stato dell'analisi: dai metadati cellulari alle matrici dei conteggi (grezzi e corretti dai batch effect), fino alle coordinate delle molteplici proiezioni dimensionali (come PCA, t-SNE e UMAP), tutto in un unico oggetto. Garantirebbe dunque maggiore interoperabilità con l'ecosistema bioinformatico moderno, considerando anche che il formato `.h5ad`, estensione dei file in formato compatibile con `AnnData`, è utilizzato sempre più frequentemente nell'ambito della ricerca scRNA-seq.

Naturalmente, un simile lavoro architetturale si preannuncia oneroso in termini di sviluppo: esso comporterebbe infatti la transizione dalla classe standardizzata `Orange.data.Table`, perdendo temporaneamente i vantaggi legati alle funzioni native e alle ottimizzazioni già integrate nel framework di Orange. Tuttavia, un aggiornamento di questa portata risulterebbe strategico a lungo termine. Consentirebbe infatti di scalare l'utilizzo dell'Add-On Single Cell verso scenari computazionali molto più complessi, avvicinando definitivamente l'ambiente visuale di Orange alle

reali pipeline sperimentali e di produzione impiegate quotidianamente nella ricerca bioinformatica.

7.3.2 Ampliamento degli algoritmi per l'integrazione dei dati e la correzione degli effetti di batch

In stretta continuità con gli obiettivi di questo lavoro di tesi, una naturale prospettiva di sviluppo consiste nell'ampliamento della libreria di algoritmi disponibili per la correzione degli effetti di batch all'interno dell'ambiente Orange. Qualora l'Add-On Single Cell venisse rielaborato per supportare nativamente il formato `AnnData`, si aprirebbe la strada all'integrazione agevole e modulare di molteplici metodi allo stato dell'arte, ciascuno caratterizzato da paradigmi teorici e ambiti di applicazione ottimali differenti [10].

Oltre ad Harmony, che fonda la sua efficienza su correzioni lineari locali e clustering iterativo, si potrebbe prevedere l'implementazione di widget dedicati a metodi basati sulla topologia dei grafi [37] (come Mutual Nearest Neighbors o BBKNN), su architetture generative di Deep Learning [28] (come i Variational Autoencoders impiegati in scVI e scANVI) o su transformer [29] (scGPT). Un'alternativa progettuale più raffinata consisterebbe, altrimenti, nello sviluppo di un singolo widget modulare, concepito come un hub centralizzato per l'integrazione dei dati e la correzione degli effetti di batch, all'interno del quale l'operatore possa scegliere con flessibilità l'algoritmo di correzione più idoneo al proprio scenario.

Mettere a disposizione dell'utente un ventaglio così diversificato di approcci permetterebbe di adattare la strategia di integrazione alle specifiche peculiarità e alla complessità del dataset in esame (e.g. elevata sparsità, presenza di tipi cellulari rari, sbilanciamento tra i batch o necessità di analisi a valle che richiedono un formato preciso di rappresentazione dei dati). Inoltre, lavorando in sinergia con lo strumento di valutazione diagnostica sviluppato in questa tesi, tale pluralità di metodi trasformerebbe l'ambiente Orange Single Cell in una vera e propria piattaforma di benchmarking interattiva. Ciò consentirebbe al ricercatore di confrontare empiricamente le performance di diversi algoritmi in tempo reale sul proprio dataset, selezionando la pipeline che garantisce il miglior compromesso tra la rimozione del bias tecnico e la conservazione dell'eterogeneità biologica.

Appendice A

Codice sorgente Harmony

In questa appendice sono riportati i codici sorgente completi dei moduli sviluppati per l'Add-On Single Cell di Orange, discussi nei capitoli precedenti. I file originali sono scritti in linguaggio Python e sfruttano le librerie PyQt, NumPy e Scikit-learn.

A.1 Modulo Base: `harmony_full_numpy.py`

Questo script contiene il motore computazionale matematico dell'algoritmo Harmony.

```
import numpy as np

def _l2norm_cols(X, eps=1e-12):
    nrm = np.linalg.norm(X, axis=0, keepdims=True)
    nrm = np.maximum(nrm, eps)
    return X / nrm

def _one_hot_labels(labels):
    labels = np.asarray(labels)
    levels, inv = np.unique(labels, return_inverse=True)
    B = len(levels)
    N = labels.shape[0]
    Phi = np.zeros((B, N), dtype=float)
    Phi[inv, np.arange(N)] = 1.0
    counts = Phi.sum(1)
    return Phi, levels, counts

def _one_hot_multi(vars_list):
    if not isinstance(vars_list, (list, tuple)):
        vars_list = [vars_list]
    Phis, levels_all, counts_all = [], [], []
    for v in vars_list:
        Phi, lev, cnt = _one_hot_labels(v)
        Phis.append(Phi)
        levels_all.append(lev)
        counts_all.append(cnt)
    Phi = np.concatenate(Phis, axis=0)
    counts_concat = np.concatenate(counts_all)
```

```
    return Phi, levels_all, counts_concat

def _expand_theta(theta, counts_per_var):
    counts_per_var = np.asarray(counts_per_var)
    if np.isscalar(theta):
        return np.repeat(float(theta), counts_per_var.sum())
    theta = np.asarray(theta, dtype=float)
    if theta.ndim == 1 and theta.size == counts_per_var.size:
        return np.concatenate([np.repeat(theta[j], counts_per_var[j
    ]) for j in range(len(counts_per_var))])
    if theta.ndim == 1 and theta.size == counts_per_var.sum():
        return theta
    raise ValueError("theta: unknown shape.")

def _kpp_cosine_init(Z_cos_T, K, rng):
    N, d = Z_cos_T.shape
    centers = np.empty((K, d), dtype=float)
    idx = rng.integers(0, N)
    centers[0] = Z_cos_T[idx]
    D = 1.0 - (Z_cos_T @ centers[0][:, None]).squeeze()
    D = np.maximum(D, 0.0)
    for k in range(1, K):
        probs = D / (D.sum() + 1e-12)
        idx = rng.choice(N, p=probs)
        centers[k] = Z_cos_T[idx]
        dots = Z_cos_T @ centers[k][:, None]
        D = np.minimum(D, 1.0 - dots.squeeze())
        D = np.maximum(D, 0.0)
    centers /= np.linalg.norm(centers, axis=1, keepdims=True).clip(
    min=1e-12)
    return centers

def _compute_dist_cos(Y_T, Z_cos):
    return 2.0 * (1.0 - (Y_T @ Z_cos))

def _update_R_blocks(R, dist_mat, sigma, Phi, Pr_b, theta_level,
    block_size, rng, E, O):
    K, N = R.shape
    B = Phi.shape[0]

    score = -dist_mat / sigma[:, None]
    score -= np.max(score, axis=0, keepdims=True)
    base = np.exp(score)

    order = np.arange(N)
    rng.shuffle(order)
    n_blocks = int(np.ceil(1.0 / block_size))
    blocks = np.array_split(order, n_blocks)

    for b in blocks:
        if b.size == 0:
            continue

        mass = R[:, b].sum(axis=1)
        E -= np.outer(mass, Pr_b)
        O -= R[:, b] @ Phi[:, b].T
```

```
82
83          penalty_kb = np.power((E + 1.0) / (O + 1.0), theta_level)
84          penalty_cells = penalty_kb @ Phi[:, b]
85          R[:, b] = base[:, b] * penalty_cells
86          colsum = np.sum(R[:, b], axis=0, keepdims=True).clip(min=1e
    -12)
87          R[:, b] /= colsum
88
89          mass_new = R[:, b].sum(axis=1)
90          E += np.outer(mass_new, Pr_b)
91          O += R[:, b] @ Phi[:, b].T
92
93      return R, E, O
94
95  def _moe_correct_ridge_numpy(Z_orig, Z_corr, Z_cos, R, Phi_moe,
    lamb):
96      d, N = Z_orig.shape
97      K = R.shape[0]
98      Z_corr[:] = Z_orig
99      for k in range(K):
100         Phi_Rk = Phi_moe * R[k, :][None, :]
101         X = Phi_Rk @ Phi_moe.T
102         X += lamb
103         RHS = Phi_Rk @ Z_orig.T
104         W = np.linalg.solve(X, RHS)
105         W[0, :] = 0.0
106         Z_corr -= (W.T @ Phi_Rk)
107     Z_cos[:] = _l2norm_cols(Z_corr, 1e-12)
108     return Z_corr, Z_cos
109
110 def _safe_entropy(R):
111     out = R * np.log(R, where=(R>0), out=np.zeros_like(R))
112     return out
113
114 def _compute_objective_full(R, dist, sigma, theta_level, E, O, Phi)
    :
115     kmeans_error = float(np.sum(R * dist))
116     ent = float(np.sum(_safe_entropy(R) * sigma[:, None]))
117     y = np.tile(theta_level[None, :], (R.shape[0], 1))
118     z = np.log((O + 1.0) / (E + 1.0))
119     w = (y * z) @ Phi
120     cross = float(np.sum((R * sigma[:, None]) * w))
121     total = kmeans_error + ent + cross
122     return total, kmeans_error, ent, cross
123
124 def harmony_numpy_fun(
125     X_raw,
126     batches,
127     n_pcs=50,
128     K=None,
129     sigma=0.1,
130     theta=1.0,
131     lamb=1.0,
132     tau=0.0,
133     block_size=0.05,
134     max_iter_harmony=10,
```

```
    max_iter_kmeans=20,
    eps_kmeans=1e-5,
    eps_harmony=1e-4,
    random_state=0,
    print_every=1
):
    rng = np.random.default_rng(random_state)

    X = np.asarray(X_raw, dtype=float)
    Xc = X - X.mean(axis=0, keepdims=True)
    U, S, Vt = np.linalg.svd(Xc, full_matrices=False)
    Z = (U[:, :n_pcs] * S[:n_pcs])

    N, d = Z.shape

    Phi, levels_all, counts_concat = _one_hot_multi(batches)
    B = Phi.shape[0]
    Pr_b = Phi.sum(axis=1) / N

    counts_per_var = np.array([len(lev) for lev in levels_all])
    theta_level = _expand_theta(theta, counts_per_var)

    if tau > 0 and K is not None:
        theta_level = theta_level * (1.0 - np.exp(-(counts_concat /
    (K * tau))**2))

    if np.isscalar(lamb):
        lamb_vec = np.full(B + 1, float(lamb))
        lamb_vec[0] = 0.0
        lamb_mat = np.diag(lamb_vec)
    else:
        lamb_vec = np.asarray(lamb, dtype=float)
        lamb_mat = np.diag(lamb_vec)

    Phi_moe = np.vstack([np.ones((1, N)), Phi])

    Z_T = Z.T.copy()
    colmax = np.max(np.abs(Z_T), axis=0, keepdims=True).clip(min=1e
    -12)
    Z_cos = _l2norm_cols(Z_T / colmax, 1e-12)
    Z_corr = Z_T.copy()

    if K is None:
        K = int(min(round(N / 30.0), 100))

    if np.isscalar(sigma):
        sigma = np.full(K, float(sigma))
    else:
        sigma = np.asarray(sigma, dtype=float)

    Y = _kpp_cosine_init(Z_cos.T, K, rng)
    Y /= np.linalg.norm(Y, axis=1, keepdims=True).clip(min=1e-12)

    dist = _compute_dist_cos(Y, Z_cos)
    score = -dist / sigma[:, None]
    score -= np.max(score, axis=0, keepdims=True)
```

```
    R = np.exp(score)
    R /= np.sum(R, axis=0, keepdims=True)

    E = np.outer(R.sum(axis=1), Pr_b)
    O = R @ Phi.T

    obj_hist = []
    total, km, ent, cross = _compute_objective_full(R, dist, sigma,
    theta_level, E, O, Phi)
    obj_hist.append(total)
    print(f"[init] Obj={total:.6e} | kmeans={km:.6e} | entropy={ent
:.6e} | cross={cross:.6e}")

    for it in range(1, max_iter_harmony + 1):
        for it_k in range(1, max_iter_kmeans + 1):

            Y = (R @ Z_cos.T)
            Y /= np.linalg.norm(Y, axis=1, keepdims=True).clip(min
=1e-12)

            dist = _compute_dist_cos(Y, Z_cos)
            R, E, O = _update_R_blocks(R, dist, sigma, Phi, Pr_b,
theta_level,
                                       block_size, rng, E, O)

        Z_corr, Z_cos = _moe_correct_ridge_numpy(Z_T, Z_corr, Z_cos
, R, Phi_moe, lamb_mat)

        total, km, ent, cross = _compute_objective_full(R, dist,
sigma, theta_level, E, O, Phi)
        obj_hist.append(total)

        if it % print_every == 0:
            print(f"[outer {it:02d}] Obj={total:.6e} | kmeans={km
:.6e} | entropy={ent:.6e} | cross={cross:.6e}")

        if len(obj_hist) >= 2:
            rel = (obj_hist[-2] - obj_hist[-1]) / (abs(obj_hist
[-2]) + 1e-12)
            if rel < eps_harmony:
                print(f"[stop] Convergence achieved:  rel ={rel:.3e
} < eps_harmony={eps_harmony}")
                break

    return Z_corr.T
```

Listing A.1: Codice sorgente del nucleo algoritmico `harmony_full_numpy.py`.

A.2 Preprocessore: `harmony_preprocess.py`

Questo codice racchiude il preprocessore, che funge da ponte tra l'algoritmo matematico ed il widget stesso.

```
from Orange.data import Table, Domain
from Orange.preprocess.preprocess import Preprocess
```

```
from .harmony_full_numpy import harmony_numpy_fun
from Orange.data import Domain, ContinuousVariable

class HarmonyModel:
    def __init__(self, batch_vars, harmony_params):
        """
        batch_vars      = lista di Orange.Variable
        harmony_params  = parametri da passare a harmony_numpy_fun
        """
        self.batch_vars = batch_vars
        self.harmony_params = harmony_params

    def transform(self, data):
        X_raw = data.X
        batch_list = [data.get_column(v) for v in self.batch_vars]

        # PCA + Harmony
        Z_corr = harmony_numpy_fun(
            X_raw=X_raw,
            batches=batch_list,
            **self.harmony_params
        )

        k = Z_corr.shape[1]
        attrs = [ContinuousVariable(f"PC{i+1}") for i in range(k)]

        class_vars = list(data.domain.class_vars)
        metas = list(data.domain.metas)

        new_domain = Domain(attrs, class_vars, metas)
        new_Y = data.Y
        new_metas = data.metas
        new_data = Table(new_domain, Z_corr, new_Y, new_metas)

        return new_data

    def __call__(self, data):
        return self.transform(data)

class HarmonyNormalizer(Preprocess):
    def __init__(self, batch_vars=(), **harmony_params):
        """
        batch_vars      = meta variables names list (string)
        harmony_params  = harmony_numpy_fun parameters
        """
        self.batch_vars = batch_vars
        self.harmony_params = harmony_params

    def __call__(self, data):

        vars_obj = [data.domain[b] for b in self.batch_vars]

        model = HarmonyModel(vars_obj, self.harmony_params)

```

```
57          return model.transform(data)
```

Listing A.2: Codice sorgente del preprocess `harmony_preprocess.py`.

A.3 Widget Harmony: `owharmony.py`

Il seguente codice è l'implementazione vera e propria del widget Harmony, descritta nel capitolo dedicato.

```
import numbers
import sys
from collections import namedtuple
from enum import IntEnum
import numpy as np

from AnyQt.QtCore import Qt
from AnyQt.QtGui import QStandardItemModel, QStandardItem
from AnyQt.QtWidgets import (
    QStyledItemDelegate, QTreeView, QHeaderView, QApplication
)

from Orange.data import Table
from Orange.widgets import gui
from Orange.widgets.settings import (
    ContextSetting, Setting, PerfectDomainContextHandler
)
from Orange.widgets.widget import Input, Output, Msg, OWWidget

from orangecontrib.single_cell.preprocess.scbnorm import (
    ScBatchScorer, LINKS
)

from orangecontrib.single_cell.preprocess.harmony_preprocess import
    HarmonyNormalizer

VariableRole = next(gui.OrangeUserRole)

class IntegralDelegate(QStyledItemDelegate):
    def initStyleOption(self, option, index):
        super().initStyleOption(option, index)
        data = index.data(Qt.DisplayRole)
        if isinstance(data, numbers.Number):
            option.displayAlignment = Qt.AlignRight | Qt.
   AlignVCenter

class RealDelegate(QStyledItemDelegate):
    def initStyleOption(self, option, index):
        super().initStyleOption(option, index)
        data = index.data(Qt.DisplayRole)
        if isinstance(data, numbers.Number):
            option.displayAlignment = Qt.AlignRight | Qt.
   AlignVCenter
            option.text = "?" if np.isnan(data) else "{:.3f}".
   format(data)

```

```
class LinkMethod(IntEnum):
    IDENTITY_LINK, LOG_LINK = range(2)

    @staticmethod
    def items():
        return sorted(list(LINKS.keys()))

class OWHarmony(OWWidget):
    name = "Harmony"
    description = "Batch effect removal with Harmony method on
   Single Cell RNA sequencing data set."
    icon = "icons/BatchEffectRemoval.svg"
    priority = 230

    class Inputs:
        data = Input("Data", Table)

    class Outputs:
        data = Output("Data", Table)

    class Error(OWWidget.Error):
        general_error = Msg({})
        discrete_attributes = Msg("Data with discrete attributes "
                                  "can not be processed.")

    class Warning(OWWidget.Warning):
        missing_values = Msg("Missing values have been replaced
   with 0.")
        negative_values = Msg("Unable to use current settings due "
                              "to negative values in data.")

    resizing_enabled = False
    want_main_area = False

    settingsHandler = PerfectDomainContextHandler()
    batch_vars = ContextSetting([])
    link_method = Setting(LinkMethod.IDENTITY_LINK)
    skip_zeros = Setting(False)
    auto_commit = Setting(True)

    # Harmony tunable parameters
    sigma = Setting(0.1)
    theta = Setting(1.0)
    lamb = Setting(1.0)
    K = Setting(10)
    max_iter_harmony = Setting(10)
    n_pcs = Setting(50)

    def __init__(self, parent=None):
        super().__init__(parent)
        self.data = None

        infobox = gui.widgetBox(self.controlArea, "Info")
        self.info_label = gui.widgetLabel(infobox, "No data on
   input.")

```

```
        method_box = gui.widgetBox(self.controlArea, "Parameters")

        # Harmony tunable parameters
        gui.doubleSpin(
            method_box, self, "sigma",
            minv=0.01, maxv=10.0, step=0.05,
            label="softness (\u03C3):",
            decimals=3,
            callback=self.commit.deferred
        )

        gui.doubleSpin(
            method_box, self, "theta",
            minv=0.0, maxv=10.0, step=0.1,
            label="imbalance penalty (\u03B8):",
            decimals=3,
            callback=self.commit.deferred
        )

        gui.doubleSpin(
            method_box, self, "lamb",
            minv=0.0, maxv=10.0, step=0.1,
            label="regularization (\u03BB):",
            decimals=3,
            callback=self.commit.deferred
        )

        gui.spin(
            method_box, self, "K",
            minv=1, maxv=100, step=1,
            label="clusters:",
            callback=self.commit.deferred
        )

        gui.spin(
            method_box, self, "max_iter_harmony",
            minv=1, maxv=1000, step=1,
            label="max iterations:",
            callback=self.commit.deferred
        )

        gui.spin(
            method_box, self, "n_pcs",
            minv=1, maxv=100, step=1,
            label="principal components:",
            callback=self.commit.deferred
        )

        # Batch Variable Selection
        header_shema = (
            ("selected", ""),
            ("variable", "Variable"),
            ("count", "#"),
            ("score", "Score"))
        header_labels = labels = [label for _, label in
header_shema]
```

```
151         header = namedtuple("header", [tag for tag, _ in
    header_shema])
152         self.Header = header(*[index for index, _ in enumerate(
    labels)])
153
154         batch_box = gui.widgetBox(self.controlArea, "Batch Variable
     Selection")
155         self.view = QTreeView()
156         self.model = QStandardItemModel()
157         self.model.itemChanged.connect(self.
    __selected_batch_vars_changed)
158         self.model.setHorizontalHeaderLabels(header_labels)
159         batch_box.layout().addWidget(self.view)
160         self._setup_view()
161
162         gui.auto_commit(self.controlArea, self, "auto_commit",
163                         "Apply", "Apply Automatically")
164
165     def __selected_batch_vars_changed(self, item):
166         if item.checkState():
167             self.batch_vars.append(item.data(VariableRole))
168         else:
169             self.batch_vars.remove(item.data(VariableRole))
170         self.commit.deferred()
171
172     def _setup_view(self):
173         self.view.setModel(self.model)
174         self.view.setSelectionMode(QTreeView.NoSelection)
175         self.view.setSortingEnabled(True)
176         self.view.setRootIsDecorated(False)
177         self.view.setItemDelegateForColumn(self.Header.count,
178                                            IntegralDelegate(self))
179         self.view.setItemDelegateForColumn(self.Header.score,
180                                            RealDelegate(self))
181         self.view.header().setSectionResizeMode(QHeaderView.
    ResizeToContents)
182         self.view.header().setStretchLastSection(False)
183         self.view.header().setSectionResizeMode(self.Header.
    variable,
184                                                 QHeaderView.Stretch
    )
185         self.view.setFocus()
186
187     @Inputs.data
188     def set_data(self, data):
189         self.closeContext()
190         self.clear()
191         self.data = data
192         self._setup_info_label()
193         self._check_data()
194         self.openContext(data)
195
196         if self.data is not None:
197             if hasattr(self.data, 'X') and self.data.X is not None:
198                 num_samples = len(self.data.X)
199                 self.K = min(max(int(num_samples * 0.05), 1), 100)
```

```
200             else:
201                 self.K = 10
202 
203             self.batch_vars = [data.domain[v.name] for v in self.
    batch_vars]
204             self._setup_model()
205 
206         self.commit.deferred()
207 
208     def clear(self):
209         self.batch_vars = []
210         if self.model:
211             n_rows = self.model.rowCount()
212             self.model.removeRows(0, n_rows)
213 
214     def _setup_info_label(self):
215         text = "No data on input."
216         if self.data is not None:
217             domain, attrs = self.data.domain, self.data.domain.
    attributes
218             text = "{} cells, {} genes\n".format(len(self.data),
    len(attrs))
219             text += "{} meta features".format(len(domain.metas)) \
220                 if len(domain.metas) else "(no meta features)"
221         self.info_label.setText(text)
222 
223     def _check_data(self):
224         self.clear_messages()
225         if self.data and self.data.domain.has_discrete_attributes()
    :
226             self.data = None
227             self.Error.discrete_attributes()
228         if self.data is not None and np.isnan(self.data.X).any():
229             self.data = self.data.copy()
230             with self.data.unlocked_reference(self.data.X):
231                 self.data.X = np.nan_to_num(self.data.X)
232             self.Warning.missing_values()
233 
234     def _setup_model(self):
235         estimator = ScBatchScorer()
236         for var in self.data.domain.class_vars + self.data.domain.
    metas:
237             if not var.is_primitive():
238                 continue
239             try:
240                 score = float(estimator.score_data(self.data, var))
241             except Exception:
242                 score = np.nan
243             self.model.appendRow([
244                 self.__selected_item(var),
245                 self.__variable_item(var),
246                 self.__count_item(var),
247                 self.__score_item(score)
248             ])
249 
250     def __selected_item(self, var):
```

```
        item = QStandardItem()
        item.setData(var, VariableRole)
        item.setCheckable(True)
        select = var in self.batch_vars
        item.setCheckState(Qt.Checked if select else Qt.Unchecked)
        item.setEditable(False)
        return item

    def __variable_item(self, var):
        item = QStandardItem()
        item.setData(var.name, Qt.DisplayRole)
        item.setData(gui.attributeIconDict[var], Qt.DecorationRole)
        item.setEditable(False)
        return item

    def __count_item(self, var):
        item = QStandardItem()
        if var.is_discrete:
            item.setData(len(var.values), Qt.DisplayRole)
        item.setEditable(False)
        return item

    def __score_item(self, score):
        item = QStandardItem()
        item.setData(score, Qt.DisplayRole)
        item.setEditable(False)
        return item

    @gui.deferred
    def commit(self):
        data = None
        self.Error.general_error.clear()
        self.Warning.negative_values.clear()

        if self.data is None:
            self.Outputs.data.send(None)
            return

        try:
            batch_vars = [v.name for v in self.batch_vars]

            if len(batch_vars) == 0:
                self.Outputs.data.send(self.data)
                return

            normalizer = HarmonyNormalizer(
                batch_vars=batch_vars,
                K=self.K,
                sigma=self.sigma,
                theta=self.theta,
                lamb=self.lamb,
                tau=0.0,
                max_iter_harmony=self.max_iter_harmony,
                n_pcs=self.n_pcs
            )

```

```
307             data = normalizer(self.data)
308
309         except Exception as e:
310             self.Error.general_error(str(e))
311             data = None
312
313         self.Outputs.data.send(data)
314
315     def send_report(self):
316         method = LinkMethod.items()[self.link_method]
317         if self.skip_zeros:
318             method += " (Skip zero expressions)"
319         variables = ", ".join([v.name for v in self.batch_vars]) \
320             if self.batch_vars else "None"
321         self.report_items("", [("Method", method),
322                                ("Batch variable selection",
    variables)])
323
324 def main(args=None):
325     app = QApplication(args or [])
326     w = OWHarmony()
327     w.set_data(Table("iris"))
328     w.show()
329     w.raise_()
330     app.exec_()
331     w.saveSettings()
332     w.onDeleteWidget()
333
334 if __name__ == "__main__":
335     sys.exit(main())
```

Listing A.3: Codice sorgente del widget Harmony `owharmony.py`.

Appendice B

Codice sorgente Batch Correction Evaluation

Lo script seguente mostra l'implementazione del widget Batch Correction Evaluation, descritta anch'essa nell'apposito capitolo.

```
import numbers
import sys
from collections import namedtuple
import numpy as np

from AnyQt.QtCore import Qt, QRectF
from AnyQt.QtGui import (
    QStandardItemModel, QStandardItem, QPainter, QColor, QPen,
   QBrush
)
from AnyQt.QtWidgets import (
    QStyledItemDelegate, QTreeView, QHeaderView, QApplication
)

from sklearn.neighbors import NearestNeighbors
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

from Orange.data import Table, Domain, ContinuousVariable
from Orange.widgets import gui
from Orange.widgets.settings import ContextSetting, Setting,
   PerfectDomainContextHandler
from Orange.widgets.widget import Input, Output, Msg, OWWidget

VariableRole = next(gui.OrangeUserRole)
BarValueRole = Qt.UserRole + 1001
BarColorRole = Qt.UserRole + 1002


class IntegralDelegate(QStyledItemDelegate):
    def initStyleOption(self, option, index):
        super().initStyleOption(option, index)
```

```
        data = index.data(Qt.DisplayRole)
        if isinstance(data, numbers.Number):
            option.displayAlignment = Qt.AlignRight | Qt.
    AlignVCenter


class BarDelegate(QStyledItemDelegate):
    def sizeHint(self, option, index):
        s = super().sizeHint(option, index)
        s.setHeight(max(s.height(), 24))
        return s

    def paint(self, painter, option, index):
        if index.column() != 1:
            super().paint(painter, option, index)
            return

        text = index.data(Qt.DisplayRole)
        bar_val = index.data(BarValueRole)
        color = index.data(BarColorRole)

        try:
            if bar_val is None or (isinstance(bar_val, float) and
    np.isnan(bar_val)):
                super().paint(painter, option, index)
                return
        except Exception:
            super().paint(painter, option, index)
            return

        v = float(bar_val)
        v = 0.0 if v < 0 else (1.0 if v > 1 else v)

        if not isinstance(color, QColor):
            color = QColor(200, 200, 200)

        painter.save()
        painter.setClipRect(option.rect)

        if option.state & getattr(option, "State_Selected", 0):
            painter.fillRect(option.rect, option.palette.highlight
    ())
        else:
            painter.fillRect(option.rect, option.palette.base())

        rect = option.rect
        pad = 6
        gap = 8
        right_pad = 14
        text_w = 70

        text_rect = rect.adjusted(pad, 0, -(rect.width() - pad -
    text_w), 0)
        vpad = 4
        bar_rect = rect.adjusted(pad + text_w + gap, vpad, -
    right_pad, -vpad)
```

```
83
84          if bar_rect.width() < 6 or bar_rect.height() < 6:
85              painter.setPen(QColor(30, 30, 30))
86              painter.drawText(rect.adjusted(pad, 0, -pad, 0),
87                               Qt.AlignVCenter | Qt.AlignLeft, str(
    text))
88              painter.restore()
89              return
90
91          radius = min(4, int(bar_rect.height() / 2))
92
93          painter.setPen(Qt.NoPen)
94          painter.setBrush(QBrush(QColor(230, 230, 230, 180)))
95          painter.drawRoundedRect(QRectF(bar_rect), radius, radius)
96
97          fill = QRectF(bar_rect)
98          fill.setWidth(bar_rect.width() * v)
99          painter.setBrush(QBrush(QColor(color.red(), color.green(),
    color.blue(), 200)))
100         painter.drawRoundedRect(fill, radius, radius)
101
102         painter.setBrush(Qt.NoBrush)
103         painter.setPen(QPen(QColor(160, 160, 160, 160), 1))
104         painter.drawRoundedRect(QRectF(bar_rect), radius, radius)
105
106         if option.state & getattr(option, "State_Selected", 0):
107             painter.setPen(option.palette.highlightedText().color()
    )
108         else:
109             painter.setPen(QColor(30, 30, 30))
110         painter.drawText(text_rect, Qt.AlignVCenter | Qt.AlignLeft,
     str(text))
111
112         painter.restore()
113
114
115 def _lisi_from_neighbor_labels(neigh_labels):
116     labels = np.asarray(neigh_labels)
117     if labels.dtype.kind == "f":
118         labels = labels.copy()
119         labels[np.isnan(labels)] = -999999.0
120     _, counts = np.unique(labels, return_counts=True)
121     p = counts.astype(float) / counts.sum()
122     return 1.0 / np.sum(p * p)
123
124 def lisi_per_cell(X, labels, k=90, metric="euclidean"):
125     X = np.asarray(X, dtype=float)
126     n = X.shape[0]
127     if n < 2:
128         return np.array([np.nan] * n, dtype=float)
129     k_eff = int(min(max(1, k), n - 1))
130     nn = NearestNeighbors(n_neighbors=k_eff + 1, metric=metric)
131     nn.fit(X)
132     neigh_idx = nn.kneighbors(return_distance=False)[:, 1:]
133     labels = np.asarray(labels)
134     out = np.empty(n, dtype=float)
```

```
    for i in range(n):
        out[i] = _lisi_from_neighbor_labels(labels[neigh_idx[i]])
    return out

def _mixing_score_from_lisi(lisi_values, n_categories):
    if n_categories <= 1:
        return np.nan
    lisi_mean = float(np.nanmean(lisi_values))
    return (lisi_mean - 1.0) / (n_categories - 1.0)

def _separation_score_from_lisi(lisi_values, n_categories):
    if n_categories <= 1:
        return np.nan
    lisi_mean = float(np.nanmean(lisi_values))
    return 1.0 - (lisi_mean - 1.0) / (n_categories - 1.0)

def _neighbor_overlap(before, after, k):
    n = before.shape[0]
    if n < 2:
        return np.nan
    k_eff = int(max(1, min(k, n - 1)))
    nn1 = NearestNeighbors(n_neighbors=k_eff + 1, metric="euclidean
    ").fit(before)
    nn2 = NearestNeighbors(n_neighbors=k_eff + 1, metric="euclidean
    ").fit(after)
    idx1 = nn1.kneighbors(return_distance=False)[:, 1:]
    idx2 = nn2.kneighbors(return_distance=False)[:, 1:]
    overlaps = np.empty(n, dtype=float)
    for i in range(n):
        a = set(idx1[i])
        b = set(idx2[i])
        inter = len(a & b)
        union = len(a | b)
        overlaps[i] = inter / union if union else 1.0
    return float(np.nanmean(overlaps))

def _improvement(after, before, eps=1e-12):
    if np.isnan(after) or np.isnan(before):
        return np.nan
    return float(np.clip((after - before) / (1.0 - before + eps),
    0.0, 1.0))

def _retention(after: float, before: float, eps: float = 1e-12) ->
    float:
    if np.isnan(after) or np.isnan(before):
        return np.nan
    return float(np.clip(after / (before + eps), 0.0, 1.0))

def _batch_pred_auc_logreg(X, y, test_size=0.2, random_state=0):
    X = np.asarray(X, dtype=float)
    y = np.asarray(y).reshape(-1)
    if y.dtype.kind == "f":
        mask = ~np.isnan(y)
    else:
        mask = np.array([v is not None for v in y], dtype=bool)
    X = X[mask]
```

```
    y = y[mask]
    n = X.shape[0]
    if n < 5:
        return np.nan
    classes, counts = np.unique(y, return_counts=True)
    if len(classes) < 2:
        return np.nan
    stratify = y if np.all(counts >= 2) else None
    try:
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=test_size, random_state=random_state,
    stratify=stratify
        )
    except Exception:
        return np.nan
    if len(np.unique(y_train)) < 2 or len(np.unique(y_test)) < 2:
        return np.nan
    try:
        clf = LogisticRegression(max_iter=2000, solver="lbfgs",
    multi_class="auto")
        clf.fit(X_train, y_train)
        proba = clf.predict_proba(X_test)
        if proba.shape[1] == 2:
            return float(roc_auc_score(y_test, proba[:, 1]))
        return float(roc_auc_score(y_test, proba, multi_class="ovr"
    , average="macro"))
    except Exception:
        return np.nan



class OWBatchCorrectionEvaluation(OWWidget):
    name = "Batch Correction Evaluation"
    description = ("Compare global iLISI, cLISI, batch prediction
    AUC (Logistic Regression) "
                   "and other metrics before vs after batch
    correction or integration.")
    icon = "icons/BatchEffectRemoval.svg"
    priority = 235

    class Inputs:
        before = Input("Data (before)", Table)
        after = Input("Data (after)", Table)

    class Outputs:
        metrics = Output("Metrics", Table)

    class Error(OWWidget.Error):
        general_error = Msg({})
        missing_inputs = Msg("At least one input is required.")
        size_mismatch = Msg("Inputs must have the same number of
    rows (cells).")
        no_continuous = Msg("Input has no continuous attributes (no
     embedding).")
        need_discrete = Msg("Selected variable must be discrete.")
```

```
235         no_discrete_vars = Msg("No discrete variables found in
    input.")
236
237     class Warning(OWWidget.Warning):
238         missing_values = Msg("Missing values have been replaced
    with 0.")
239         too_few_points = Msg("Not enough rows to compute neighbors.
    ")
240         logreg_not_computable = Msg("Batch prediction AUC could not
     be computed.")
241
242     resizing_enabled = True
243     want_main_area = True
244
245     settingsHandler = PerfectDomainContextHandler()
246
247     batch_var = ContextSetting(None)
248     celltype_var = ContextSetting(None)
249
250     k_neighbors = Setting(90)
251     auto_commit = Setting(True)
252
253     def __init__(self, parent=None):
254         super().__init__(parent)
255
256         self.setMinimumWidth(850)
257         self.setMinimumHeight(550)
258
259         self.data_before = None
260         self.data_after = None
261
262         infobox = gui.widgetBox(self.controlArea, "Info")
263         self.info_label = gui.widgetLabel(infobox, "No data on
    input.")
264
265         pbox = gui.widgetBox(self.controlArea, "Parameters")
266         gui.spin(
267             pbox, self, "k_neighbors",
268             minv=5, maxv=500, step=5,
269             label="Number of neighbors (k):",
270             callback=self.commit.deferred
271         )
272
273         header_schema = (
274             ("selected", ""),
275             ("variable", "Variable"),
276             ("count", "#levels"),
277         )
278         header_labels = [lbl for _, lbl in header_schema]
279         header = namedtuple("header", [tag for tag, _ in
    header_schema])
280         self.Header = header(*range(len(header_labels)))
281
282         batch_box = gui.widgetBox(self.controlArea, "Batch Variable
     (for iLISI + LogReg)")
283         self.batch_view = QTreeView()
```

```
        self.batch_model = QStandardItemModel()
        self.batch_model.setHorizontalHeaderLabels(header_labels)
        self.batch_model.itemChanged.connect(self.__batch_changed)
        batch_box.layout().addWidget(self.batch_view)
        self._setup_view(self.batch_view, self.batch_model)

        cell_box = gui.widgetBox(self.controlArea, "Cell Type
    Variable (for cLISI)")
        self.cell_view = QTreeView()
        self.cell_model = QStandardItemModel()
        self.cell_model.setHorizontalHeaderLabels(header_labels)
        self.cell_model.itemChanged.connect(self.__celltype_changed
    )
        cell_box.layout().addWidget(self.cell_view)
        self._setup_view(self.cell_view, self.cell_model)

        box = gui.widgetBox(self.mainArea, "Results")
        self.results_view = QTreeView()
        self.results_model = QStandardItemModel()
        self.results_model.setHorizontalHeaderLabels(["Metric", "
    Value"])
        self.results_view.setModel(self.results_model)
        self.results_view.setRootIsDecorated(False)
        self.results_view.setAlternatingRowColors(True)
        self.results_view.setEditTriggers(QTreeView.NoEditTriggers)

        hdr = self.results_view.header()
        hdr.setSectionResizeMode(0, QHeaderView.ResizeToContents)
        hdr.setSectionResizeMode(1, QHeaderView.Stretch)

        self.results_view.setItemDelegate(BarDelegate(self))
        self.results_view.setUniformRowHeights(True)
        box.layout().addWidget(self.results_view)

        gui.auto_commit(self.controlArea, self, "auto_commit",
                        "Apply", "Apply Automatically")

    def _setup_view(self, view, model):
        view.setModel(model)
        view.setSelectionMode(QTreeView.NoSelection)
        view.setSortingEnabled(True)
        view.setRootIsDecorated(False)
        view.setItemDelegateForColumn(self.Header.count,
    IntegralDelegate(self))
        view.header().setSectionResizeMode(QHeaderView.
    ResizeToContents)
        view.header().setStretchLastSection(False)
        view.header().setSectionResizeMode(self.Header.variable,
    QHeaderView.Stretch)

    def clear(self):
        self.batch_var = None
        self.celltype_var = None
        for m in (self.batch_model, self.cell_model):
            m.removeRows(0, m.rowCount())
        if hasattr(self, "results_model"):
```

```
            self._clear_results_table()

    def _setup_info_label(self):
        if self.data_before is None and self.data_after is None:
            self.info_label.setText("No data on input.")
            return

        def _shape(d):
            if d is None:
                return "   "
            return f"{len(d)} cells, {len(d.domain.attributes)} features"

        self.info_label.setText(f"Before: {_shape(self.data_before)}\nAfter: {_shape(self.data_after)}")

    def _clear_results_table(self):
        self.results_model.removeRows(0, self.results_model.rowCount())

    @staticmethod
    def _bar_style(metric_name: str, value: float):
        if value is None or (isinstance(value, float) and np.isnan(value)):
            return np.nan, None

        name = (metric_name or "").lower()
        v = float(value)

        green_success = QColor(46, 204, 113)
        blue_state    = QColor(52, 152, 219)
        orange_diag   = QColor(243, 156, 18)

        if "improvement" in name or "retention" in name or "drop" in name:
            return float(np.clip(v, 0.0, 1.0)), green_success

        if "overlap" in name:
            return float(np.clip(v, 0.0, 1.0)), orange_diag

        return float(np.clip(v, 0.0, 1.0)), blue_state

    def _set_results(self, rows):
        self._clear_results_table()
        for name, val in rows:
            name_item = QStandardItem(str(name))
            name_item.setEditable(False)

            value_item = QStandardItem()
            value_item.setEditable(False)

            if val is None:
                value_item.setData("   ", Qt.DisplayRole)
                value_item.setData(np.nan, BarValueRole)
            elif isinstance(val, (float, np.floating)):
                if np.isnan(val):
```

```
                    value_item.setData("NaN", Qt.DisplayRole)
                    value_item.setData(np.nan, BarValueRole)
                else:
                    value_item.setData(f"{float(val):.4f}", Qt.
DisplayRole)
                    bar_v, col = self._bar_style(str(name), float(
val))
                    value_item.setData(bar_v, BarValueRole)
                    if isinstance(col, QColor):
                        value_item.setData(col, BarColorRole)
            else:
                value_item.setData(str(val), Qt.DisplayRole)
                value_item.setData(np.nan, BarValueRole)

            self.results_model.appendRow([name_item, value_item])

    def _check_embedding(self, data):
        if data is None:
            return None
        if data.X is None or data.X.size == 0:
            self.Error.no_continuous()
            return None
        if np.isnan(data.X).any():
            d2 = data.copy()
            with d2.unlocked_reference(d2.X):
                d2.X = np.nan_to_num(d2.X)
            self.Warning.missing_values()
            return d2
        if len(data) < 2:
            self.Warning.too_few_points()
        return data

    def _source_for_vars(self):
        return self.data_before or self.data_after

    def _selected_item(self, var, selected=False):
        item = QStandardItem()
        item.setData(var, VariableRole)
        item.setCheckable(True)
        item.setCheckState(Qt.Checked if selected else Qt.Unchecked
)
        item.setEditable(False)
        return item

    def _variable_item(self, var):
        item = QStandardItem()
        item.setData(var.name, Qt.DisplayRole)
        item.setData(gui.attributeIconDict[var], Qt.DecorationRole)
        item.setEditable(False)
        return item

    def _count_item(self, var):
        item = QStandardItem()
        item.setData(len(var.values), Qt.DisplayRole)
        item.setEditable(False)
        return item
```

```
438
439     def _populate_models(self):
440         self.batch_model.removeRows(0, self.batch_model.rowCount())
441         self.cell_model.removeRows(0, self.cell_model.rowCount())
442
443         src = self._source_for_vars()
444         if src is None:
445             return
446
447         discrete_vars = []
448         for var in list(src.domain.class_vars) + list(src.domain.
    metas):
449             if var.is_discrete and var.is_primitive():
450                 discrete_vars.append(var)
451
452         if not discrete_vars:
453             self.Error.no_discrete_vars()
454             return
455
456         batch_name = self.batch_var.name if self.batch_var else ""
457         cell_name = self.celltype_var.name if self.celltype_var
    else ""
458
459         for var in discrete_vars:
460             self.batch_model.appendRow([
461                 self._selected_item(var, selected=(var.name ==
    batch_name)),
462                 self._variable_item(var),
463                 self._count_item(var),
464             ])
465             self.cell_model.appendRow([
466                 self._selected_item(var, selected=(var.name ==
    cell_name)),
467                 self._variable_item(var),
468                 self._count_item(var),
469             ])
470
471     @staticmethod
472     def _uncheck_others(model, keep_item):
473         model.blockSignals(True)
474         for r in range(model.rowCount()):
475             it = model.item(r, 0)
476             if it is not keep_item and it.checkState() == Qt.
    Checked:
477                 it.setCheckState(Qt.Unchecked)
478         model.blockSignals(False)
479
480     def __batch_changed(self, item):
481         if item.column() != self.Header.selected:
482             return
483         var = item.data(VariableRole)
484         if item.checkState() == Qt.Checked:
485             self._uncheck_others(self.batch_model, item)
486             self.batch_var = var
487         else:
488             if self.batch_var == var:
```

```
                self.batch_var = None
        self.commit.deferred()

    def __celltype_changed(self, item):
        if item.column() != self.Header.selected:
            return
        var = item.data(VariableRole)
        if item.checkState() == Qt.Checked:
            self._uncheck_others(self.cell_model, item)
            self.celltype_var = var
        else:
            if self.celltype_var == var:
                self.celltype_var = None
        self.commit.deferred()

    @Inputs.before
    def set_before(self, data):
        self.closeContext()
        self.clear()

        self.data_before = self._check_embedding(data)
        self._setup_info_label()

        if self.data_before is not None:
            self.openContext(self.data_before)
        elif self.data_after is not None:
            self.openContext(self.data_after)

        src = self._source_for_vars()
        if src is not None:
            if self.batch_var is not None and self.batch_var.name
in src.domain:
                self.batch_var = src.domain[self.batch_var.name]
            else:
                self.batch_var = None

            if self.celltype_var is not None and self.celltype_var.
name in src.domain:
                self.celltype_var = src.domain[self.celltype_var.
name]
            else:
                self.celltype_var = None

        self._populate_models()
        self.commit.deferred()

    @Inputs.after
    def set_after(self, data):
        if self.data_before is None:
            self.closeContext()
            self.clear()

        self.data_after = self._check_embedding(data)
        self._setup_info_label()

        src_ctx = self._source_for_vars()
```

```
        if src_ctx is not None:
            self.openContext(src_ctx)

            if self.batch_var is not None and self.batch_var.name
    in src_ctx.domain:
                self.batch_var = src_ctx.domain[self.batch_var.name
    ]
            else:
                self.batch_var = None

            if self.celltype_var is not None and self.celltype_var.
    name in src_ctx.domain:
                self.celltype_var = src_ctx.domain[self.
    celltype_var.name]
            else:
                self.celltype_var = None

        self._populate_models()
        self.commit.deferred()

    @gui.deferred
    def commit(self):
        self.clear_messages()
        self.Error.general_error.clear()
        self.Error.need_discrete.clear()
        self.Error.no_continuous.clear()
        self.Error.no_discrete_vars.clear()
        self.Warning.logreg_not_computable.clear()

        self.Outputs.metrics.send(None)
        self._clear_results_table()

        if self.data_before is None and self.data_after is None:
            self.Error.missing_inputs()
            return

        compare_mode = (self.data_before is not None and self.
    data_after is not None)

        if compare_mode and len(self.data_before) != len(self.
    data_after):
            self.Error.size_mismatch()
            return

        try:
            k = int(self.k_neighbors)
            src = self.data_before if self.data_before is not None
    else self.data_after

            if compare_mode:
                Xb = np.asarray(self.data_before.X, dtype=float)
                Xa = np.asarray(self.data_after.X, dtype=float)
            else:
                X = np.asarray(src.X, dtype=float)

            overlap = np.nan
```

```
            ilisi_before = ilisi_after = ilisi_impr = np.nan
            clisi_before = clisi_after = clisi_reten = np.nan
            batch_auc_before = batch_auc_after = batch_auc_drop =
    np.nan

            ilisi_single = np.nan
            clisi_single = np.nan
            batch_auc_single = np.nan

            if compare_mode:
                overlap = _neighbor_overlap(Xb, Xa, k)

            if self.batch_var is not None:
                bv = src.domain[self.batch_var.name] if self.
    batch_var.name in src.domain else None
                if bv is None or not bv.is_discrete:
                    self.Error.need_discrete()
                    return

                batch_labels = src.get_column(bv)
                B = len(bv.values)

                if compare_mode:
                    lisi_b = lisi_per_cell(Xb, batch_labels, k=k)
                    lisi_a = lisi_per_cell(Xa, batch_labels, k=k)

                    ilisi_before = _mixing_score_from_lisi(lisi_b,
    B)
                    ilisi_after = _mixing_score_from_lisi(lisi_a, B
    )
                    ilisi_impr = _improvement(ilisi_after,
    ilisi_before)

                    batch_auc_before = _batch_pred_auc_logreg(Xb,
    batch_labels, test_size=0.2, random_state=0)
                    batch_auc_after = _batch_pred_auc_logreg(Xa,
    batch_labels, test_size=0.2, random_state=0)

                    if np.isnan(batch_auc_before) or np.isnan(
    batch_auc_after):
                        self.Warning.logreg_not_computable()
                    else:
                        batch_auc_drop = float(batch_auc_before -
    batch_auc_after)

                else:
                    lisi_x = lisi_per_cell(X, batch_labels, k=k)
                    ilisi_single = _mixing_score_from_lisi(lisi_x,
    B)

                    batch_auc_single = _batch_pred_auc_logreg(X,
    batch_labels, test_size=0.2, random_state=0)
                    if np.isnan(batch_auc_single):
                        self.Warning.logreg_not_computable()

            if self.celltype_var is not None:
```

```
                cv = src.domain[self.celltype_var.name] if self.celltype_var.name in src.domain else None
                if cv is None or not cv.is_discrete:
                    self.Error.need_discrete()
                    return

                ct_labels = src.get_column(cv)
                C = len(cv.values)

                if compare_mode:
                    lisi_b = lisi_per_cell(Xb, ct_labels, k=k)
                    lisi_a = lisi_per_cell(Xa, ct_labels, k=k)

                    clisi_before = _separation_score_from_lisi(lisi_b, C)
                    clisi_after = _separation_score_from_lisi(lisi_a, C)
                    clisi_reten = _retention(clisi_after, clisi_before)
                else:
                    lisi_x = lisi_per_cell(X, ct_labels, k=k)
                    clisi_single = _separation_score_from_lisi(lisi_x, C)

            rows = []
            if compare_mode:
                if not np.isnan(ilisi_before):
                    rows.append(("Global iLISI (0  1  ) before", ilisi_before))
                    rows.append(("Global iLISI (0  1  ) after", ilisi_after))
                    rows.append(("iLISI improvement (0  1  )", ilisi_impr))

                if not np.isnan(clisi_before):
                    rows.append(("Global cLISI (0  1  ) before", clisi_before))
                    rows.append(("Global cLISI (0  1  ) after", clisi_after))
                    rows.append(("cLISI retention (0  1  )", clisi_reten))

                if not np.isnan(batch_auc_before) and not np.isnan(batch_auc_after):
                    rows.append(("LogReg batch pred AUC before", batch_auc_before))
                    rows.append(("LogReg batch pred AUC after", batch_auc_after))
                    rows.append(("LogReg AUC drop", batch_auc_drop))

                if not np.isnan(overlap):
                    rows.append(("Neighbor overlap (0  1  )", overlap))
            else:
                if not np.isnan(ilisi_single):
```

```
                    rows.append(("Global iLISI (0  1  )",
    ilisi_single))
                if not np.isnan(clisi_single):
                    rows.append(("Global cLISI (0  1  )",
    clisi_single))
                if not np.isnan(batch_auc_single):
                    rows.append(("LogReg batch pred AUC",
    batch_auc_single))

            self._set_results(rows)

            def _safe_var_name(s: str) -> str:
                s = (s or "").strip()
                s = s.replace("   ", "-").replace("   ", "-")
                s = s.replace("(", "").replace(")", "")
                s = s.replace("/", "_").replace(" ", "_").replace("
    __", "_")
                return s

            out_attrs = []
            out_vals = []

            for metric_name, metric_val in rows:
                var_name = _safe_var_name(str(metric_name))
                out_attrs.append(ContinuousVariable(var_name))

                if metric_val is None:
                    out_vals.append(np.nan)
                else:
                    try:
                        out_vals.append(float(metric_val))
                    except Exception:
                        out_vals.append(np.nan)

            out_domain = Domain(out_attrs)
            out_row = np.array([out_vals], dtype=float)

            self.Outputs.metrics.send(Table(out_domain, out_row))

        except Exception as e:
            self.Error.general_error(str(e))
            self.Outputs.metrics.send(None)

    def send_report(self):
        self.report_items("", [
            ("k neighbors", self.k_neighbors),
            ("Batch variable", self.batch_var.name if self.
    batch_var else "None"),
            ("Cell type variable", self.celltype_var.name if self.
    celltype_var else "None"),
        ])


def main(args=None):
    app = QApplication(args or [])
    w = OWBatchCorrectionEvaluation()
```

```
726     w.show()
727     w.raise_()
728     app.exec_()
729     w.saveSettings()
730     w.onDeleteWidget()
731
732
733 if __name__ == "__main__":
734     sys.exit(main())
```

Listing B.1: Codice sorgente del widget Batch Correction Evaluation `owbatchcorrectionevaluation.py`.

Bibliografia

[1] X. Pan *et al.*, "Single-cell rna sequencing technologies and applications: A brief overview," *Frontiers in Genetics*, 2022. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC8964935/

[2] X. Zheng *et al.*, "The evolution of single-cell rna sequencing technology and application: Progress and perspectives," *Frontiers in Molecular Biosciences*, 2023. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC9918030/

[3] J. Smith *et al.*, "Applications and techniques of single-cell rna sequencing across biomedical research," *Briefings in Bioinformatics*, vol. 26, no. 4, p. bbaf354, 2025. [Online]. Available: https://academic.oup.com/bib/article/26/4/bbaf354/8210809

[4] P.-Y. Tung, J. D. Blischak, C. J. Hsiao, S. L. Battle, J. B. Zaugg, M. Stephens, and R. L. Gierách, "Batch effects and the effective design of single-cell gene expression studies," *Nature methods*, vol. 14, no. 9, pp. 827–830, 2017.

[5] L. Haghverdi, A. T. Lun, M. D. Morgan, and J. C. Marioni, "Batch effects in single-cell rna-sequencing data are corrected by matching mutual nearest neighbors," *Nature biotechnology*, vol. 36, no. 5, pp. 421–427, 2018.

[6] H. T. N. Tran *et al.*, "A benchmark of batch-effect correction methods for single-cell rna sequencing data," *Genome Biology*, vol. 21, no. 1, p. 12, 2020. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/31948481/

[7] M. D. Luecken, M. Büttner, K. Chaichoompu, A. Danese, M. Interlandi, R. López-Pérez, M. Klein, N. F. Müller, A. Figueras, M. Förster *et al.*, "Benchmarking atlas-level data integration in single-cell genomics," *Nature Methods*, vol. 19, no. 6, pp. 675–686, 2022.

[8] J. Demšar and B. Zupan, "Orange: Data mining fruitful and fun - a historical perspective," *Informatica*, vol. 37, no. 1, pp. 55–60, 2013. [Online]. Available: http://www.informatica.si/index.php/informatica/article/view/434

[9] M. Stražar, L. Žagar, J. Kokošar, V. Tanko, A. Erjavec, P. G. Policar, A. Starič, J. Demšar, G. Shaulsky, V. Menon, A. Lemire, A. Parikh, and B. Zupan, "scorange—a tool for hands-on training of concepts from single-cell data analytics," *Bioinformatics*, vol. 35, 2019, iSMB/ECCB 2019.

[10] M. D. Luecken, S. Gigante, D. B. Burkhardt, R. Cannoodt, D. C. Strobl, N. S. Markov, L. Zappia, G. Palla, W. Lewis, D. Dimitrov, M. E. Vinyard, D. S. Magruder, M. F. Mueller, A. Andersson, E. Dann, Q. Qin, D. J. Otto, M. Klein, O. B. Botvinnik, L. Deconinck, K. Waldrant, S. N. Yasa, A. Szałata, A. Benz, Z. Li, O. P. J. Members, B. Rieck, C. Ahlmann-Eltze, E. da Veiga Beltrame, C. Bravo González-Blas, A. T. Chen, B. DeMeo, C. Ergen, S. Floc'hlay, A. Gayoso, S. Hicks, Y. Ji, V. Kleshchevnikov, G. La Manno, M. G. Lombardo, R. Lopez, D. Righelli, H. Sarkar, V. Svensson, A. Tong, G. Xing, C. Xu, J. M. Bloom, A. O. Pisco, J. Saez-Rodriguez, D. Wulsin, L. Pinello, Y. Saeys, F. J. Theis, and S. Krishnaswamy, "Defining and benchmarking open problems in single-cell analysis," *Nature Biotechnology*, vol. 43, no. 7, pp. 1035–1040, 2025. [Online]. Available: https://doi.org/10.1038/s41587-025-02694-w

[11] M. Heidari *et al.*, "Single-cell rna sequencing procedures and data analysis," in *StatPearls*. Treasure Island (FL): StatPearls Publishing, 2024. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK569559/

[12] Y. Li *et al.*, "Single-cell transcriptomics: Current methods and challenges in neuroscience," *Frontiers in Neuroscience*, 2021. [Online]. Available: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2021.591122/full

[13] Y. Wang *et al.*, "Single-cell rna sequencing technology landscape in 2023," *Current Issues in Molecular Biology*, 2023. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/37934608/

[14] D. Osumi-Sutherland *et al.*, "Cell-level metadata are indispensable for documenting single-cell datasets," *PLOS Biology*, vol. 19, no. 12, 2021. [Online]. Available: https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.3001077

[15] H. Chen *et al.*, "Advances and challenges in single-cell rna sequencing data analysis," *Computational and Structural Biotechnology Journal*, 2026. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC12860385/

[16] T. S. Andrews and M. Hemberg, "Batch effects and the effective design of single-cell gene expression studies," *Scientific reports*, vol. 7, no. 1, p. 2969, 2017.

[17] M. D. Luecken and F. J. Theis, "Current best practices in single-cell rna-seq analysis: a tutorial," *Molecular systems biology*, vol. 15, no. 6, p. e8746, 2019.

[18] N. Team, "Understanding batch effect and normalization in scrna-seq data," https://www.nygen.io/resources/blog/batch-effect-normalization-techniques-scrnaseq, 2023, accessed: 2026-03-06.

[19] B. J. Stewart and et al., "Data matrix normalization and merging strategies minimize batch effects for single-cell rna-seq," *bioRxiv*, 2021.

[20] V. A. Traag, L. Waltman, and N. J. van Eck, "From louvain to leiden: guaranteeing well-connected communities," *Scientific reports*, vol. 9, no. 1, p. 5237, 2019.

[21] T. Stuart, A. Butler, P. Hoffman, C. Hafemeister, E. Papalexi, W. M. Mauck III, Y. Hao, M. Stoeckius, P. Smibert, and R. Satija, "Comprehensive integration of single cell data," *Cell*, vol. 177, no. 7, pp. 1888–1902, 2019.

[22] R. A. Amezquita *et al.*, "Orchestrating single-cell analysis with bioconductor," *Nature Methods*, vol. 17, no. 2, pp. 137–145, 2020. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC7358058/

[23] C. Soneson, M. I. Love, and M. D. Robinson, "Differential analyses for rna-seq: transcript-level analyses and beyond," *Genome biology*, vol. 19, no. 1, pp. 1–15, 2018.

[24] E. Team, "Batch effect in single-cell rna-seq: Frequently asked questions," 2023. [Online]. Available: https://www.elucidata.io/blog/batch-effect-in-single-cell-rna-seq-frequently-asked-questions-and-answers

[25] L. Tian and et al., "Deep learning enables accurate clustering with batch effect removal in single-cell rna-seq analysis," *Nature communications*, vol. 11, no. 1, p. 2760, 2020.

[26] W. E. Johnson, C. Li, and A. Rabinovic, "Adjusting batch effects in microarray expression data using empirical bayes methods," *Biostatistics*, vol. 8, no. 1, pp. 118–127, 2007.

[27] I. Korsunsky, N. Millard, J. Fan, K. Slowikowski, F. Zhang, K. Wei, Y. Baglaenko, M. Brenner, P.-R. Loh, and S. Raychaudhuri, "Fast, sensitive and accurate integration of single-cell data with harmony," *Nature methods*, vol. 16, no. 12, pp. 1289–1296, 2019.

[28] C. Xu, R. Lopez, E. Mehlman, J. Regier, M. Klein, A. Gayoso, and N. Yosef, "scanvi uses stochastic processes and phenotypic information to correct for batch effects in single-cell transcriptomics," *Nature communications*, vol. 12, no. 1, pp. 1–12, 2021.

[29] B. Li, S. Zhang, H. Li, K. Zhang, Y. Zhang, Y. Zhang, J. Chen, Z. Wang, H. Zhang, Y. Zhang *et al.*, "scgpt: toward building a foundation model for single-cell multi-omics using generative ai," *Nature methods*, vol. 21, no. 5, pp. 773–786, 2024.

[30] S. Xu, Q. Liu, X. Su *et al.*, "Batch alignment of single-cell transcriptomics data using deep metric learning," *Nature Communications*, vol. 14, no. 1, p. 1075, 2023.

[31] M. D. Luecken *et al.*, "Benchmarking atlas-level data integration in single-cell genomics - integration task datasets (immune and pancreas)," https://doi.org/10.6084/m9.figshare.12420968.v5, 2020, figshare dataset associated with [7].

[32] M. Bakhtiari, S. Bonn, F. J. Theis, O. Zolotareva, and J. Baumbach, "Fedscgen: privacy-preserving federated batch effect correction of single-cell rna sequencing data," *Genome Biology*, vol. 26, no. 1, p. 216, 2025.

[33] G. X. Zheng, J. M. Terry, P. Belgrader, P. Ryvkin, Z. W. Bent, R. Wilson, S. B. Ziraldo, T. D. Wheeler, G. P. McDermott, J. Zhu *et al.*, "Massively parallel digital transcriptional profiling of single cells," *Nature Communications*, vol. 8, p. 14049, 2017.

[34] J. L. Bautista, N. T. Cramer, C. N. Miller, J. Chavez, A. S. Weinmann, A. G. Farr *et al.*, "Developmental dynamics of thymic stromal and epithelial cells revealed by single-cell rna sequencing," *Cell Reports*, vol. 31, no. 7, p. 107510, 2020.

[35] J. L. Bautista *et al.*, "Single-cell rna-seq of human thymus across development," https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE147520, 2020, gEO accession for thymus dataset.

[36] C. Padron-Manrique, A. Vázquez-Jiménez, D. A. Esquivel-Hernandez, Y. E. Martinez-Lopez, D. Neri-Rosario, D. Giron-Villalobos, E. Mixcoha, J. P. Sánchez-Castañeda, and O. Resendis-Antonio, "Diffusion on pca-umap manifold: The impact of data structure preservation to denoise high-dimensional single-cell rna sequencing data," *Biology*, vol. 13, no. 7, p. 512, 2024. [Online]. Available: https://doi.org/10.3390/biology13070512

[37] L. Haghverdi *et al.*, "Batch effects in single-cell rna-sequencing data are corrected by matching mutual nearest neighbors," *Nature Biotechnology*, vol. 36, no. 5, pp. 421–427, 2018. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/29608177/