NSCC Neuro-Symbolic Constraint Checker

Configuratore di build PC neuro-simbolico con vincoli (Knowledge Graph + CSP + ML)

Gruppo di lavoro

- Mattia Dimonopoli —799872—m.dimonopoli1@studenti.uniba.it

A.A. 2025–2026

Abstract

Il progetto NSCC (Neuro-Symbolic Constraint Checker) implementa un configuratore di PC che integra apprendimento supervisionato, una base di conoscenza espressa come Knowledge Graph (RDF) e un correttore basato su Constraint Satisfaction Problem (CSP). Dato un profilo/insieme di preferenze e un budget, un predittore neurale multi-output propone una configurazione plausibile; successivamente il correttore CSP ripara eventuali incoerenze (compatibilità hardware e vincoli di potenza/budget) e, opzionalmente, ottimizza la soluzione per avvicinarsi al budget e alle preferenze dell'utente. La valutazione sperimentale è condotta tramite k-fold cross validation e confronto tra predizione "grezza" e predizione corretta, misurando sia metriche ML (precision/recall/accuracy macro) sia metriche di coerenza (tasso di consistenza e violazioni di budget). Il contributo principale del progetto consiste nel dimostrare come un correttore basato su CSP, derivato automaticamente da una Knowledge Base esplicita, possa garantire coerenza logica e rispetto di vincoli economici, mantenendo prestazioni predittive comparabili a quelle di un modello di apprendimento supervisionato non vincolato.

Sommario esecutivo

Componenti principali del sistema:

- Catalogo di componenti (domini finiti, prezzi, attributi tecnici).
- Knowledge Graph RDF che rappresenta attributi e relazioni di compatibilità, usato per generare vincoli.
- Compilazione dei vincoli in una forma efficiente (incompatibilità e implicazioni).
- Modello di apprendimento supervisionato (MultiOutputClassifier + MLPClassifier) per predire una build.
- Correttore CSP con ricerca backtracking e funzione obiettivo (vicinanza al ML + budget + preferenze).
- Interfacce di utilizzo: CLI interattiva e GUI (Tkinter).
- Script di valutazione: cross-validation (run_experiment.py) e benchmark multi-budget (run_benchmark.py).

Elenco argomenti del corso toccati dal progetto

- Knowledge Graph e Ontologie (PDF 6): triple RDF, classi/proprietà, modellazione di un dominio reale e reificazione.
- Rappresentazione e Ragionamento Relazionale (PDF 5): uso di predicati/relazioni per generalizzare conoscenza (schema catalogo, attributi, compatibilità).
- Ragionamento con Vincoli (PDF 3): CSP, vincoli hard e soft, ricerca con potatura, ottimizzazione.
- Apprendimento Supervisionato (PDF 7): definizione del task, training/test, cross-validation, metriche (precision/recall/accuracy, log-loss come concetto di costo).

- Reti Neurali e Apprendimento Profondo (PDF 8): MLP come esempio di rete feed-forward per classificazione.

1. Introduzione e obiettivi

Configurare un PC richiede soddisfare simultaneamente molte condizioni: compatibilità meccanica ed elettrica (socket CPU–scheda madre, standard RAM DDR4/DDR5, potenza alimentatore in funzione della GPU), vincoli economici (budget massimo) e spesso anche preferenze qualitative (gaming vs office, equilibrio CPU/GPU, ecc.). Un modello puramente data-driven può proporre combinazioni incoerenti se non "conosce" esplicitamente tali regole; un approccio puramente simbolico può essere coerente ma poco "naturale" o lento nel proporre configurazioni sensate in funzione di un profilo. NSCC integra i due mondi: l'ML fornisce una prima proposta, il simbolico la rende valida e ottimale.

1.1 Requisiti funzionali

1. Dato un budget e un profilo, generare una configurazione completa (CPU, Motherboard, RAM, GPU, PSU, Case, Storage).
2. Garantire coerenza rispetto ai vincoli di compatibilità hardware.
3. Gestire vincoli economici: cercare soluzioni <= budget e, se impossibile, minimizzare lo sforamento.
4. Fornire una modalità "conservativa": se la predizione ML è già coerente e nel budget, non modificarla (in assenza di preferenze esplicite).
5. Consentire preferenze utente (gaming/office, qualità, richiesta GPU/CPU/RAM) come vincoli soft nell'ottimizzazione.
6. Supportare esecuzione riproducibile: seed, k-fold CV, output di metriche e grafici.

1.2 Struttura del repository e file principali

Organizzazione del progetto (zip consegnato):

Percorso	Contenuto / ruolo
README.md	Istruzioni di esecuzione e comandi per CV/benchmark/GUI.
requirements.txt	Dipendenze Python (numpy, scikit-learn, rdflib, matplotlib).
scripts/run_experiment.py	Esegue k-fold cross-validation e salva summary + grafici.
scripts/run_benchmark.py	Benchmark multi-seed e multi-budget per report (media ± std).
scripts/gui.py	Interfaccia grafica Tkinter per configurazione guidata.
scripts/interactive_cli.py	CLI per test rapido (budget, preferenze).

src/nscc/kg.py	Catalogo + costruzione Knowledge Graph + estrazione vincoli.
src/nscc/constraints.py	Compilazione vincoli e funzioni di consistenza.
src/nscc/model.py	Modello ML multi-output (MLP + MultiOutputClassifier).
src/nscc/csp_corrector.py	Correttore CSP: ricerca e ottimizzazione.
src/nscc/eval.py	Metriche e routine di valutazione in CV.
src/nscc/data.py	Generazione dataset (feature e label).

2. Architettura complessiva e flusso di esecuzione

Il sistema è progettato secondo una pipeline modulare. La KB (Knowledge Graph) definisce il dominio e viene compilata in vincoli; il modello ML opera in uno spazio di feature numeriche; il correttore CSP combina output probabilistici del modello e vincoli per produrre una soluzione valida.

Schema della pipeline (alto livello):

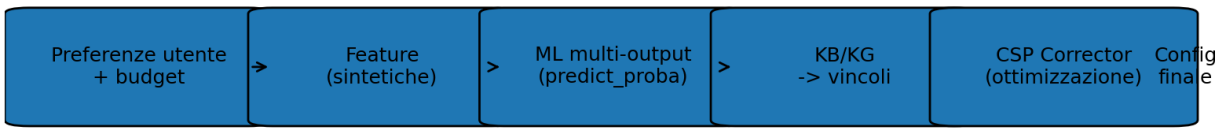

Figura 1 — Pipeline end-to-end: preferenze/budget → ML → vincoli da KB/KG → CSP → configurazione finale.

2.1 Dati, domini e rappresentazione delle configurazioni

Una configurazione è rappresentata come assegnazione totale a un insieme di variabili discrete: CPU, Motherboard, RAM, GPU, PSU, Case, Storage. Ogni variabile ha un dominio finito di valori (ID stringhe stabili). Il catalogo associa a ciascun valore un prezzo e attributi tecnici (socket, standard RAM, watt, ecc.). Questa scelta consente di modellare il problema sia come classificazione multi-classe (ML) sia come CSP su domini finiti.

2.2 Catalogo interno e catalogo esteso

Il progetto supporta due modalità: (i) un catalogo interno, piccolo ma realistico, definito direttamente nel codice; (ii) un catalogo esteso opzionale generato da scripts/fetch_catalog.py che produce data/catalog_curated.json. All'avvio, la funzione build_default_catalog (src/nscc/kg.py) tenta di caricare automaticamente il catalogo curato e fa fallback al catalogo interno se il file non esiste o non è valido.

3. Knowledge Base come Knowledge Graph (RDF) e generazione dei vincoli

La componente simbolica del progetto è una Knowledge Base (KB) implementata come Knowledge Graph (KG) RDF. L'obiettivo non è solo memorizzare un catalogo come database, ma formalizzare relazioni che inducono vincoli sulle configurazioni ammissibili. In particolare, la KB esprime: (i) attributi dei componenti (socket, tipo RAM, watt, ecc.), (ii) relazioni di compatibilità/incompatibilità tra componenti, (iii) regole che trasformano attributi in vincoli tra variabili del CSP.

3.1 Scelte di rappresentazione

Nel KG, ogni componente è un individuo (URI) e le proprietà tecniche sono rappresentate come triple RDF. Per mantenere il progetto gestibile, le regole di compatibilità sono implementate in modo deterministico e compilate in vincoli ground (tra variabili e valori). Questo approccio bilancia espressività e semplicità: il KG rende esplicita la conoscenza di dominio, mentre il CSP usa una rappresentazione più 'operativa' (liste di incompatibilità e implicazioni).

3.2 Catalogo: variabili, domini e attributi

La classe Catalog (src/nscc/kg.py) incapsula: l'elenco di variabili, i domini per ciascuna variabile, i prezzi e attributi tecnici necessari per le regole. Un estratto della definizione (semplificato):

```
NSCC = Namespace("http://example.org/nscc/")


@dataclass(frozen=True)
class Catalog:
    """Catalogo dei componenti (domini delle variabili, prezzi e attributi utili).

    Nota: i valori (IDs) sono stringhe stabili e vengono usati in ML/CSP.
    `display_name` serve solo per output leggibile in CLI/GUI.
    """

    variables: List[str]
    domains: Dict[str, List[str]]
    prices: Dict[str, int]  # value_id -> price EUR
    display_name: Dict[str, str]  # value_id -> pretty label

    # attributi per regole (usati sia per feature sintetiche sia per vincoli)
    cpu_socket: Dict[str, str]  # cpu_id -> socket
    mb_socket: Dict[str, str]  # mb_id -> socket
    mb_ram: Dict[str, str]  # mb_id -> ddr4/ddr5
    ram_type: Dict[str, str]  # ram_id -> ddr4/ddr5
    gpu_psu_min: Dict[str, int]  # gpu_id -> W min consigliati
    psu_watt: Dict[str, int]  # psu_id -> W

    # score/metadata per ottimizzazione (proxy prestazioni)
    cpu_score: Dict[str, int]  # 0..100
    gpu_score: Dict[str, int]  # 0..100
    ram_gb: Dict[str, int]
    storage_gb: Dict[str, int]


@dataclass(frozen=True)
```

```
class ConstraintSpec:
    """Vincoli estratti dalla KB in forma utilizzabile dal CSP.

    - `incompatible`: lista di coppie ground che NON possono coesistere.
      Ogni elemento e' ((var1, value1), (var2, value2)).
    - `requires`: lista di implicazioni ground. Ogni elemento e'
      ((var1, value1), (var2, value2)) e significa:
        se var1=value1 allora var2=value2.

    Nota: più implicazioni con stesso antecedente e stesso consequente-var
    rappresentano una OR (var2 può assumere *uno qualunque* dei valori ammessi).
    """

    incompatible: List[Tuple[Tuple[str, str], Tuple[str, str]]]
    requires: List[Tuple[Tuple[str, str], Tuple[str, str]]]


def _u(name: str) -> URIRef:
    return NSCC[name]
```

Gli attributi principali usati nei vincoli sono: cpu_socket e mb_socket per la compatibilità CPU↔MB; mb_ram e ram_type per DDR4/DDR5; gpu_psu_min e psu_watt per la compatibilità GPU↔alimentatore. Il catalogo contiene anche metadati per l'ottimizzazione (cpu_score, gpu_score, ram_gb, storage_gb), usati per modellare preferenze soft senza ricorrere a benchmark reali.

3.3 Dal KG ai vincoli (compilazione)

La KB viene 'compilata' in un oggetto ConstraintSpec che contiene due famiglie di vincoli:

- Incompatibilità: coppie di assegnazioni che non possono coesistere (es. CPU socket AM5 con MB LGA1700).
- Implica/Requires: se una variabile assume un valore, allora un'altra variabile deve assumere un valore compatibile (es. una GPU che richiede PSU ≥ X).

Questi vincoli sono poi trasformati in una struttura CompiledConstraints (src/nscc/constraints.py) più efficiente per controlli ripetuti e per la ricerca. L'idea è coerente con il corso: partire da una rappresentazione dichiarativa (KG) e ricavarne un formalismo operativo per il ragionamento (CSP).

3.4 Esempi concreti di regole

Esempi (descrittivi) di vincoli modellati nel dominio:

- Socket CPU–Motherboard: la CPU deve avere lo stesso socket della scheda madre.
- Standard RAM: la RAM deve essere DDR4 se la motherboard è DDR4, DDR5 altrimenti.
- PSU–GPU: la potenza dell'alimentatore deve essere almeno pari al minimo raccomandato per la GPU (vincolo hard).
- Budget: somma dei prezzi dei componenti non deve superare il budget (hard se possibile, altrimenti soft).

Nel codice, tali regole vengono costruite in modo deterministico a partire dagli attributi del catalogo, e poi espanse in vincoli ground sulle coppie (variabile, valore).

4. Ragionamento con vincoli: correttore CSP e ottimizzazione

Il correttore CSP è il componente che rende “affidabile” il sistema: prende in input le probabilità del modello ML (una distribuzione per ciascuna variabile) e produce una configurazione completa che soddisfa i vincoli hard. Quando presenti, integra anche vincoli soft (budget, spend_near_budget e preferenze qualitative) come termini di costo in una funzione obiettivo.

4.1 Struttura dei vincoli e test di consistenza

I vincoli compilati comprendono: (i) una lista di incompatibilità, e (ii) un dizionario requires_any che rappresenta implicazioni in cui il consequente può essere un insieme di valori ammessi (OR). Il controllo di consistenza è implementato in src/nscc/constraints.py ed è usato sia per valutazione sia per pruning.

4.2 Funzione di costo: vicinanza al modello + vincoli soft

La ricerca non mira solo a trovare una configurazione valida, ma la “migliore” rispetto a un criterio composito. In csp_corrector.py la funzione obiettivo combina:

- Distanza dal modello ML: somma di -log(p) per le assegnazioni scelte (costruita da pred_proba).
- Penalità di sforamento budget (se budget soft): costo quadratico proporzionale all’over-budget.
- Penalità di underuse (opzionale): spinge a spendere vicino al budget quando richiesto (es. build gaming).
- Penalità per mismatch con preferenze utente (gpu_need, cpu_need, ram_need, quality) basate su score proxy.

Estratto delle funzioni di costo (src/nscc/csp_corrector.py):

```
                return True

    return False


def _assignment_cost(val: str, proba: Optional[Mapping[str, float]]) -> float:
    """Costo (da minimizzare) dell'assegnazione in base alle probabilita' ML."""
    if not proba:
        return 0.0
    p = float(proba.get(val, 0.0))
    p = max(p, 1e-9)
    return -math.log(p)


def _budget_penalty(price: int, budget: Optional[int], budget_weight: float) -> float:
    """Penalizza lo sforamento budget (soft-constraint).

    - se price <= budget: 0
    - altrimenti: penalità quadratica, scalata da budget_weight
    """

    if budget is None or budget_weight <= 0:
        return 0.0
    over = max(0, price - int(budget))
    return float(budget_weight) * (over / 250.0) ** 2
```

```
def _spend_near_budget_penalty(price: int, budget: Optional[int], spend_weight: float)
-> float:
    """Penalizza l'"underuse" del budget, per spingere a spendere vicino al budget.

    Questo serve a spiegare perché con budget alto si può ottenere una build più cara:
    se spend_weight>0, il solver preferisce soluzioni con prezzo vicino a budget.
    """

    if budget is None or spend_weight <= 0:
        return 0.0
    under = max(0, int(budget) - price)
    return float(spend_weight) * (under / 350.0) ** 2


def _prefs_penalty(
    cfg: Mapping[str, str],
    user_prefs: Optional[Mapping[str, float]],
    budget: Optional[int] = None,
) -> float:
    """Penalità (soft) per mismatch con preferenze utente.

    user_prefs attesi (0..1): gpu_need, cpu_need, ram_need, quality
    - gpu_need: quanto conta la GPU (gaming)
    - cpu_need: quanto conta la CPU (produttività)
    - ram_need: desiderio di RAM (0≈16GB, 1≈64GB)
    - quality: target globale, serve a non proporre "4K" a budget bassi.
    """

    if not user_prefs:
        return 0.0

    gpu_need = float(user_prefs.get("gpu_need", 0.5))
    cpu_need = float(user_prefs.get("cpu_need", 0.5))
    ram_need = float(user_prefs.get("ram_need", 0.5))
    quality = float(user_prefs.get("quality", 0.5))

    # Cap del target di qualità in base al budget: evita richieste "4K" con budget
bassi.
    # - Sotto ~700€ -> cap ~0.2
    # - Oltre ~2500€ -> cap ~1.0
    if budget is not None:
        budget_cap = max(0.0, min(1.0, (int(budget) - 700) / (2500 - 700)))
        quality = min(quality, float(budget_cap))

    gpu_sc = CATALOG.gpu_score[cfg["GPU"]]  # 0..100
    cpu_sc = CATALOG.cpu_score[cfg["CPU"]]  # 0..100
    ram_gb = CATALOG.ram_gb[cfg["RAM"]]

    # target in 0..100
    # qualità globale sposta tutti i target verso l'alto/basso
    base_target = 25 + 70 * quality  # 25..95
    gpu_target = base_target * (0.6 + 0.4 * gpu_need)
    cpu_target = base_target * (0.6 + 0.4 * cpu_need)
    ram_target_gb = 16 + int(round(48 * ram_need))  # 16..64

    # pesi: GPU pesa di più quando gpu_need è alto, ecc.
    w_gpu = 1.8 * (0.3 + 0.7 * gpu_need)
    w_cpu = 1.3 * (0.3 + 0.7 * cpu_need)
    w_ram = 0.6 * (0.3 + 0.7 * ram_need)

    # costo quadratico normalizzato
```

```
    gpu_term = w_gpu * ((gpu_sc - gpu_target) / 35.0) ** 2
    cpu_term = w_cpu * ((cpu_sc - cpu_target) / 30.0) ** 2
    ram_term = w_ram * ((ram_gb - ram_target_gb) / 24.0) ** 2

    # Penalizza anche componenti "overkill" quando l'utente vuole office/base:
    # - PSU troppo sopra il minimo raccomandato
    # - Case full tower quando non richiesto
    psu_over = 0.0
    if "PSU" in cfg and "GPU" in cfg:
        min_psu = CATALOG.gpu_psu_min.get(cfg["GPU"], 650)
        w = CATALOG.psu_watt.get(cfg["PSU"], 650)
        # penalizza solo quando quality è bassa
        psu_over = (max(0, w - min_psu) / 350.0) ** 2 * (1.2 * (1.0 - quality))

    case_over = 0.0
    if cfg.get("Case") == "case_atx_full_tower":
        case_over = 0.6 * (1.0 - quality)

    return float(gpu_term + cpu_term + ram_term + psu_over + case_over)


def correct_prediction(
    pred_proba: Mapping[str, Mapping[str, float]],
    cons: CompiledConstraints,
    budget: Optional[int] = None,
    budget_weight: float = 3.0,
    spend_weight: float = 0.0,
    user_prefs: Optional[Mapping[str, float]] = None,
    domains: Mapping[str, List[str]] | None = None,
```

4.3 Algoritmo di ricerca e potatura

Il solver usa una ricerca backtracking (depth-first) su assegnazioni parziali, con diverse euristiche di potatura:

- Ordine variabili per ‘sharpness’: prima le variabili con massima probabilità più alta (scelte più ‘certe’).
- Bound ottimistico sul costo rimanente: se cost_so_far + bound >= best_cost allora potatura.
- Potatura su vincoli: _partial_violates intercetta incompatibilità e implicazioni violate da una parziale.
- Budget hard come fase preferita: se esiste soluzione <= budget, la ricerca pota tutto ciò che non può rientrare (bound sul prezzo minimo rimanente).

Queste scelte riflettono gli elementi discussi nel modulo di Ragionamento con Vincoli (CSP): l’efficienza dipende criticamente da ordinamento delle variabili e pruning.

4.4 Politica “conservativa” e gestione delle preferenze

Per un uso realistico, il correttore adotta una politica conservativa: se la configurazione ottenuta scegliendo l’argmax per ogni variabile è già consistente e nel budget, viene restituita senza modifiche (early return). Questa politica viene disabilitata quando l’utente fornisce preferenze esplicite, perché in quel caso l’obiettivo non è solo la coerenza ma anche l’allineamento a un profilo (es. office può preferire componenti meno costosi anche se la predizione ML è valida).

5. Apprendimento supervisionato: predizione di configurazioni

La componente di apprendimento supervisionato ha lo scopo di proporre rapidamente una configurazione plausibile in base a un vettore di feature (preferenze sintetiche, budget e indicatori). Il problema è formulato come classificazione multi-output: una classe per ciascuna variabile della configurazione.

5.2 Modello: MultiOutputClassifier con MLP

Il modello è implementato in src/nscc/model.py. Si usa uno StandardScaler seguito da MultiOutputClassifier, dove ogni output è un MLPClassifier (rete feed-forward) addestrato con Adam. Questa scelta consente di ottenere sia predizioni di classe sia distribuzioni di probabilità per output.

Estratto rilevante (src/nscc/model.py):

```
from sklearn.pipeline import Pipeline
from sklearn.exceptions import ConvergenceWarning

from .data import VARIABLES


@dataclass
class TrainedModel:
    pipeline: Pipeline
    classes_: Dict[str, List[str]]  # var -> list of classes in estimator order


def train_model(
    X_train: np.ndarray,
    y_train: Dict[str, np.ndarray],
    seed: int = 0,
    hidden_layer_sizes: Tuple[int, ...] = (64, 32),
    alpha: float = 1e-4,
    max_iter: int = 500,
) -> TrainedModel:
    """Allena un predittore multi-output (una testa per variabile).

    Scelta tecnica: MultiOutputClassifier con MLPClassifier base.
    """

    Y = np.column_stack([y_train[v] for v in VARIABLES])

    bs = int(min(128, X_train.shape[0]))

    base = MLPClassifier(
        hidden_layer_sizes=hidden_layer_sizes,
        activation="relu",
        solver="adam",
        alpha=alpha,
        batch_size=bs,
        learning_rate_init=1e-3,
        max_iter=max_iter,
        random_state=seed,
        early_stopping=False,
        n_iter_no_change=10,
        verbose=False,
    )
```

```
    clf = MultiOutputClassifier(base)
    pipe = Pipeline([
        ("scaler", StandardScaler()),
        ("clf", clf),
    ])

    with warnings.catch_warnings():
        warnings.filterwarnings('ignore', category=ConvergenceWarning)
        pipe.fit(X_train, Y)

    classes_: Dict[str, List[str]] = {}
    for v, est in zip(VARIABLES, pipe.named_steps["clf"].estimators_):
        classes_[v] = list(est.classes_)

    return TrainedModel(pipeline=pipe, classes_=classes_)


def predict_labels(model: TrainedModel, X: np.ndarray) -> Dict[str, np.ndarray]:
    Y_pred = model.pipeline.predict(X)
    return {v: Y_pred[:, i] for i, v in enumerate(VARIABLES)}


def predict_proba(model: TrainedModel, X: np.ndarray) -> List[Dict[str, Dict[str,
float]]]:
    """Ritorna, per ogni esempio, un dict: var -> {label: prob}.

    Nota: MultiOutputClassifier espone predict_proba come lista di matrici,
    una per variabile.
    """

    probas =
model.pipeline.named_steps["clf"].predict_proba(model.pipeline.named_steps["scaler"].tr
ansform(X))
    # probas: list of (n_samples, n_classes_var)

    out: List[Dict[str, Dict[str, float]]] = []
    n = X.shape[0]
    for i in range(n):
        ex: Dict[str, Dict[str, float]] = {}
        for var_idx, v in enumerate(VARIABLES):
            cls = model.classes_[v]
            pvec = probas[var_idx][i]
            ex[v] = {cls[j]: float(pvec[j]) for j in range(len(cls))}
        out.append(ex)
    return out
```

5.3 Predizione probabilistica e uso nel CSP

L’integrazione neuro-simbolica avviene perché il CSP usa le probabilità del modello come ‘costo’ di deviazione: una scelta con probabilità alta ha costo basso (-log(p)), una scelta improbabile ha costo alto. In questo modo, il correttore tende a modificare il meno possibile la proposta ML, ma è autorizzato a farlo quando necessario per soddisfare i vincoli o migliorare l’allineamento alle preferenze.

6. Valutazione sperimentale

La valutazione segue la linea-guida del corso: niente singolo run, ma stima robusta tramite k-fold cross-validation e, quando necessario per la relazione, benchmark ripetuti su più seed e budget (media ± deviazione standard). Si valutano sia prestazioni predittive (metriche ML) sia qualità della correzione simbolica (coerenza, budget).

6.1 Metriche considerate

- Precision/Recall/Accuracy macro (media sulle classi, e poi media sulle variabili).
- Exact Match Accuracy: percentuale di esempi in cui tutte le variabili sono predette correttamente (molto severa). Il valore ridotto dell'exact match è atteso in un problema multi-output con domini ampi: la probabilità di predire correttamente tutte le variabili simultaneamente è intrinsecamente più bassa rispetto alle metriche marginali.
- Consistency rate: percentuale di configurazioni che rispettano tutti i vincoli hard.
- Budget violation rate: percentuale di configurazioni che sforano il budget (quando il budget è attivo).
- Avg changes: numero medio di variabili modificate dal correttore rispetto alla predizione raw (misura di 'intervento').

6.2 Risultati (cross-validation) — raw vs corrected

Di seguito si riportano i risultati ottenuti dallo script scripts/run_experiment.py (esempio: --seed 0 --n-samples 2000 --folds 5). I valori sono media ± std sui fold.

Metrica	Raw (ML)	Corrected (ML + CSP)
Exact match accuracy	0.0033 ± 0.0058	0.0033 ± 0.0058
Consistency rate	0.9433 ± 0.0306	1.0000 ± 0.0000
Budget violation rate	0.0000 ± 0.0000	0.0000 ± 0.0000
Avg #changes (solo corrected)	0.0567 ± 0.0306	0.0567 ± 0.0306
Precision macro	0.4230 ± 0.0086	0.4206 ± 0.0067
Recall macro	0.4097 ± 0.0099	0.4082 ± 0.0077
Accuracy macro	0.4262 ± 0.0157	0.4243 ± 0.0129

Osservazione chiave: il correttore porta la consistenza a 1.0 (100%) a costo di un numero medio molto ridotto di modifiche (~0.057 variabili per configurazione nel run riportato). Le metriche di accuratezza macro restano simili, perché la correzione privilegia la coerenza: in alcuni casi può sostituire una scelta ML corretta ma incompatibile con altre variabili, oppure scegliere un valore compatibile ma non quello esatto del ground truth sintetico.

6.3 Grafici prodotti automaticamente

Il progetto salva in results/ diversi grafici utili per la relazione. Di seguito alcuni esempi.

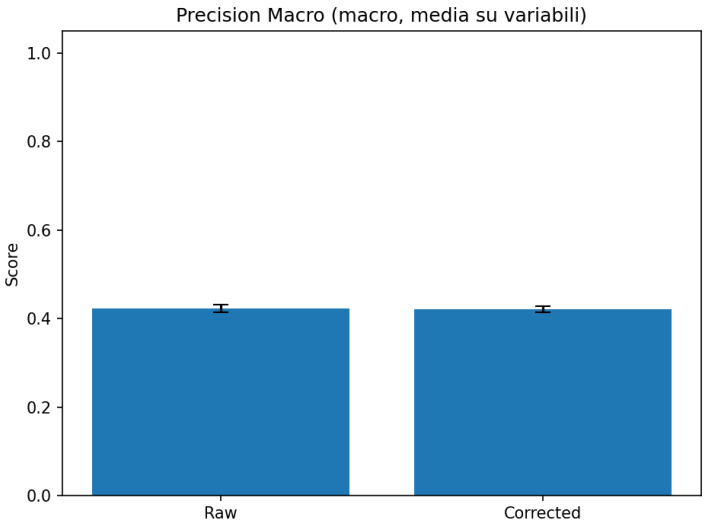

Figura 2 — Precision macro (raw vs corrected).

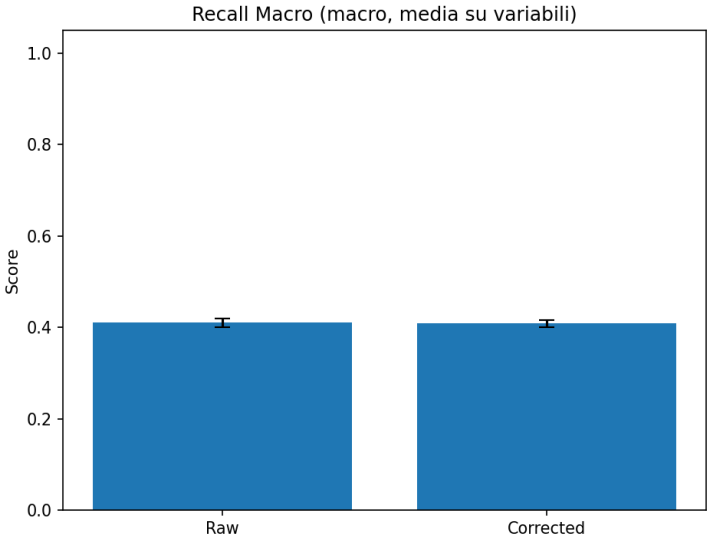

Figura 3 — Recall macro (raw vs corrected).

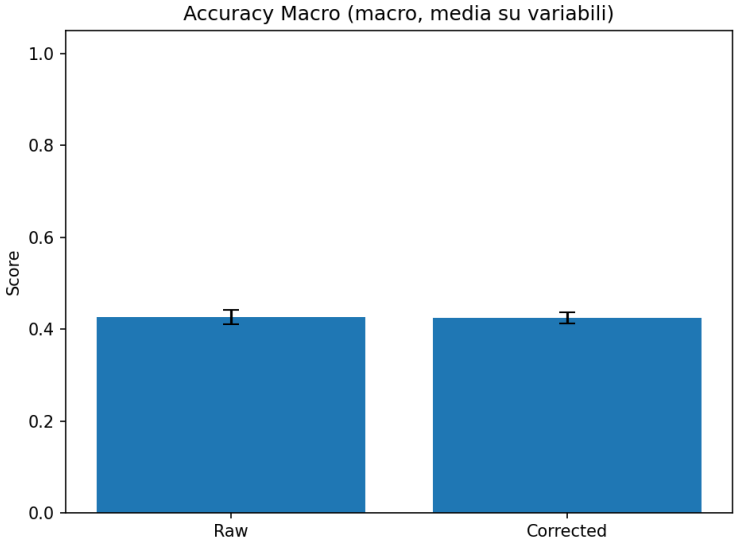

Figura 4 — Accuracy macro (raw vs corrected).

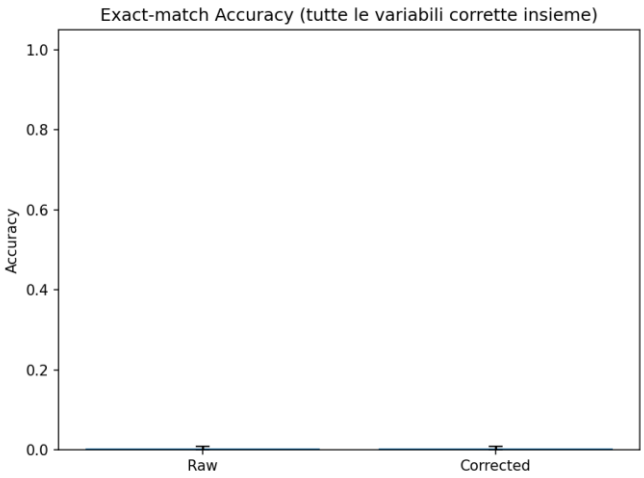

Figura 5 — Exact match accuracy (raw vs corrected).

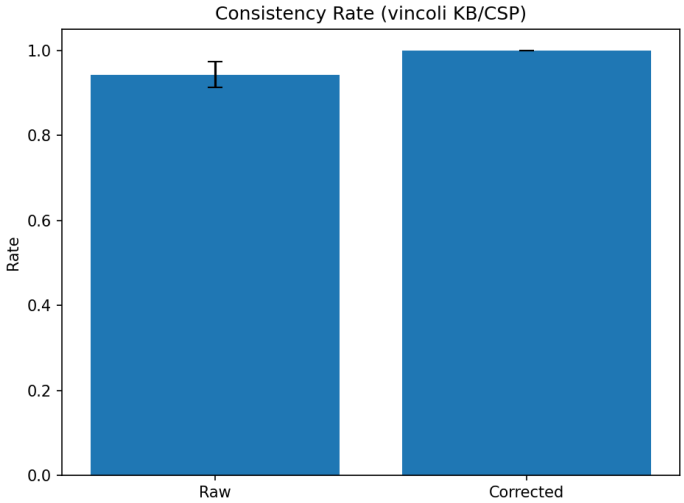

Figura 6 — Consistency rate (raw vs corrected).

6.4 Benchmark multi-budget e multi-seed (per analisi approfondita)

Per una valutazione ancora più robusta , scripts/run_benchmark.py consente di ripetere la cross-validation su più seed e per diversi budget, salvando un file summary_by_budget.json e grafici metriche-vs-budget. Questo è particolarmente utile per discutere la stabilità del sistema e l'effetto del budget sulle scelte del correttore (es. quando spend_near_budget è attivo).

```
py -3 scripts/run_benchmark.py --seeds 0..9 --budgets 800,1000,1200,1400,1600,1800,2000
\
  --n-samples 15000 --folds 5 --outdir results/bench_big --plot
```

7. Discussione

7.1 Perché l'integrazione ML + KB/KG + CSP è necessaria

Il progetto evidenzia un punto centrale del corso ICON: nessuna delle tre componenti (apprendimento supervisionato, rappresentazione della conoscenza e ragionamento con vincoli) è sufficiente se considerata isolatamente. Il modello di apprendimento supervisionato fornisce una proposta iniziale plausibile, la Knowledge Base rende esplicita la conoscenza di dominio, mentre il correttore CSP garantisce proprietà globali come la coerenza logica e il rispetto dei vincoli economici. L'integrazione consente di combinare flessibilità statistica e garanzie simboliche in modo controllato ed efficace.

7.2 Limiti del dataset sintetico

L'uso di dati sintetici è una scelta consapevole: garantisce riproducibilità e controllo del dominio, ma limita la validità esterna. In particolare: (i) i 'ground truth' riflettono euristiche definite a mano, non scelte reali di utenti; (ii) i proxy di prestazione (score) sono basati su quantili di prezzo e non su benchmark. Queste limitazioni non impattano l'obiettivo del corso (integrazione neuro-simbolica), ma sono importanti per interpretare i risultati.

7.3 Complessità e scalabilità

Il solver CSP usa backtracking su domini finiti; con il catalogo interno (pochi valori per variabile) la ricerca è veloce. Con un catalogo esteso (PCPartPicker) i domini crescono e la ricerca può diventare più costosa. Le euristiche introdotte (ordinamento per sharpness, bound sul costo, potatura su vincoli e budget) mitigano il problema, ma una scalabilità industriale richiederebbe tecniche aggiuntive (es. CP-SAT, ricerca locale, decomposizioni, o vincoli globali).

7.4 Possibili estensioni

- Aggiungere vincoli realistici: form factor (case↔motherboard), connettori PSU, lunghezza GPU, compatibilità dissipatore.
- Rendere esplicite le regole nel KG come regole Datalog/SHACL e compilare automaticamente (meno codice procedurale).
- Integrare un dataset reale (scelte utenti o build pubbliche) per addestramento e confronto.
- Visualizzazione del KG e spiegazioni: mostrare all'utente quali vincoli hanno causato la correzione (explainability).
- Sostituire il backtracking con un solver CP/ILP esterno per cataloghi molto grandi.

8. Istruzioni di esecuzione e riproducibilità

Setup (Windows), come da README.md:

```
python -m venv venv
venv\Scripts\activate
pip install -r requirements.txt
```

Esempi di comandi principali:

```
python scripts\run_experiment.py --seed 0 --n-samples 2000 --folds 5 --plot
python scripts\interactive_cli.py --budget 1500 --seed 0 --spend-weight 0.5
python scripts\gui.py
```

9. Riferimenti

Dispense del corso ICon (pdf forniti): Knowledge Graph e Ontologie; Ragionamento con Vincoli; Apprendimento Supervisionato; Reti Neurali e Apprendimento Profondo; Rappresentazione e Ragionamento Relazionale.

Librerie principali: scikit-learn (MLPClassifier, MultiOutputClassifier), RDFLib (RDF/Graph), matplotlib (plot).

Inquadramento delle Macro-Aree ICON e Collegamento alla Teoria

Inquadramento delle Macro-Aree ICON e collegamento alla teoria

Questa sezione riassume il collegamento tra le scelte progettuali adottate e le macro-aree teoriche del corso ICON, evidenziandone l'integrazione concreta nel sistema sviluppato.

Knowledge Graph e Ontologie.

Il progetto utilizza un Knowledge Graph RDF come strato dichiarativo per rappresentare la conoscenza di dominio sui componenti hardware e derivarne vincoli simbolici. Il KG non è usato come semplice database, ma come base esplicita da cui vengono compilati vincoli operativi.

Ragionamento con Vincoli (CSP).

Il problema di configurazione è formalizzato come CSP su domini finiti, con vincoli hard di compatibilità e vincoli soft di ottimizzazione. Il correttore CSP applica tecniche di potatura e bounding, in linea con quanto discusso nel modulo di Ragionamento con Vincoli.

Apprendimento Supervisionato.

La componente di apprendimento supervisionato affronta un problema multi-output e fornisce una proposta iniziale di configurazione. Le probabilità predette sono integrate nel CSP tramite un termine di costo, realizzando una vera integrazione neuro-simbolica.

Nel complesso, il progetto dimostra come l'integrazione tra apprendimento automatico, rappresentazione della conoscenza e ragionamento con vincoli consenta di ottenere sistemi più affidabili, interpretabili e coerenti con i principi teorici del corso ICON.

Conclusioni

Il progetto ha mostrato come l'integrazione tra apprendimento supervisionato, rappresentazione esplicita della conoscenza e ragionamento con vincoli consenta di affrontare in modo efficace un problema di configurazione complesso, garantendo coerenza logica e rispetto di vincoli globali che un approccio puramente statistico non sarebbe in grado di assicurare.

I risultati sperimentali, valutati tramite cross-validation e aggregati su più run, evidenziano come l'introduzione del correttore CSP migliori sistematicamente la consistenza delle soluzioni senza degradare in modo significativo le prestazioni predittive del modello di apprendimento supervisionato, confermando il valore dell'approccio neuro-simbolico adottato.

Per ragioni di tempo e di complessità, alcune problematiche non sono state affrontate nel presente lavoro. In particolare, la Knowledge Base potrebbe essere ulteriormente arricchita con vincoli più articolati e relazioni di ordine superiore, mentre l'integrazione tra apprendimento e ragionamento potrebbe essere estesa introducendo meccanismi di feedback dal CSP al processo di addestramento del modello. Ulteriori sviluppi potrebbero inoltre riguardare l'utilizzo di dataset reali più ampi e l'analisi di tecniche di ottimizzazione più avanzate per la risoluzione del CSP. Queste estensioni rappresentano naturali direzioni di evoluzione del progetto e potrebbero costituire la base per lavori futuri o per un approfondimento in un contesto di tesi.