Documentazione del Progetto:

Dataset Builder Ontology

Vincenzo Buttari, Francesco Ciavarella

7 febbraio 2025

Indice

Prefazione **1**

1 Struttura del Progetto **2**

2 Gestione dell'Ontologia **3**
2.1 La Classe `OntologyManager` 3
2.1.1 Esempio di Utilizzo 3

3 Modelli Predittivi Tradizionali e Ottimizzati **4**
3.1 Modello Base e Grid Search 4
3.2 Modello Base 4
3.2.1 Esempio di utilizzo 4
3.3 Modello Grid Search 5
3.3.1 Esempio di utilizzo 5
3.4 Confronto delle Prestazioni 5
3.4.1 Esempio di utilizzo 6

4 Utilizzo di Modelli Predefiniti con PyKEEN **7**
4.1 Visualizzazione e Clustering degli Embedding 8
4.2 Riduzione della Dimensionalità con PCA e t-SNE 8

5 Interfacce Utente: CLI e Web App **10**
5.1 Interfaccia a Riga di Comando (CLI) 10
5.1.1 Esempi di Esecuzione dal CLI 10
5.2 Interfaccia Web 11

6 Istruzioni per l'Installazione **12**
6.1 Prerequisiti 12
6.1.1 Installazione di Java 12
6.2 Esecuzione 13

7 Conclusioni e Sviluppi Futuri **14**

Dataset Builder Ontology

Nome: Vincenzo Buttari

GitHub:
https://github.com/zeltarave/dataset_builder_ontology

Matricola: 776274

Nome: Francesco Ciavarella

GitHub:
https://github.com/zeltarave/dataset_builder_ontology

Matricola: 782015

7 febbraio 2025

Prefazione

Questo documento descrive in dettaglio il progetto **Dataset Builder Ontology**. Il progetto nasce con l'obiettivo di:

- Popolare automaticamente un'ontologia OWL con dati realistici.
- Estrarre dati strutturati dall'ontologia per costruire un dataset.
- Addestrare modelli predittivi tradizionali (ad esempio, regressione logistica) e ottimizzati (tramite Grid Search) per attività di classificazione.
- Integrare modelli neurali per imparare rappresentazioni vettoriali (embeddings) degli oggetti dell'ontologia, sfruttando modelli predefiniti con PyKEEN.
- Visualizzare gli embedding tramite tecniche di riduzione della dimensionalità (PCA e t-SNE) e applicare algoritmi di clustering per identificare gruppi di entità simili.
- Fornire interfacce a riga di comando (CLI) e una web app con form interattivi per configurare la suddivisione del dataset (train/test).

Il documento illustra la struttura del progetto, i moduli principali, gli esempi di esecuzione e le possibili direzioni di sviluppo futuro.

Capitolo 1

Struttura del Progetto

La struttura delle cartelle è organizzata in modo modulare per separare le responsabilità.
La gerarchia delle cartelle è la seguente:

```
dataset_builder_ontology/
 data/
    ontology.owl       % Ontologia OWL popolata (output)
    dataset.csv        % Dataset generato (output)
 src/
    flask/
       templates/
          compare.html
          grid_search.html
          index.html
          dataset.html
          train.html
       app.py                   % Interfaccia web con Flask
    owl/
       logger_config.py         % Configurazione del logging
       ontology_manager.py      % Classe OntologyManager per gestione ontologia
    predictive_model/
       predictive_model.py      % Modello predittivo base
       grid_search_model.py     % Ottimizzazione tramite Grid Search
       compare_models.py        % Confronto tra modelli
    cli.py                       % Interfaccia a riga di comando
    pykeen_leaner/
       learn_knowledge.py       % Implementazione di pyKEEN attraverso TransE
 requirements.txt                 % Dipendenze del progetto
 README.md
```

Capitolo 2

Gestione dell'Ontologia

2.1 La Classe `OntologyManager`

Il modulo `ontology_manager.py` definisce la classe `OntologyManager`, che incapsula le seguenti operazioni:

- **load()**: Carica l'ontologia dal file specificato o, se il file non esiste, crea una nuova ontologia (file di default `ontology.owl`).
- **populate()**: Definisce classi (ad es. `Person` e `Course`), proprietà (ad es. `has_name`, `has_age`, `teaches`, `takes`, `course_title`, `course_description`) e popola l'ontologia con individui generati automaticamente.
- **reason()**: Esegue il ragionamento sull'ontologia tramite il ragionatore di `Owlready2`.
- **extract_features()**: Estrae le informazioni rilevanti degli individui della classe `Person` presenti nell'ontologia.

2.1.1 Esempio di Utilizzo

```
from owl.ontology_manager import OntologyManager
import os

ontology_path = os.path.join("data", "ontology.owl")
onto = OntologyManager(ontology_file)

onto.populate() # Populate carica automaticamente la ontology
onto.extract_features() # Extract_features esegue il ragionamento sull'
    ontologia
```

Listing 2.1: Utilizzo della classe OntologyManager

Capitolo 3

Modelli Predittivi Tradizionali e Ottimizzati

3.1 Modello Base e Grid Search

Il progetto addestra un modello di regressione logistica sul dataset estratto, utilizzando due approcci:

- **Modello Base:** Addestrato con parametri fissi.
- **Modello con Grid Search:** Utilizza `GridSearchCV` per ottimizzare iperparametri come C e il tipo di penalizzazione.

Le funzioni sono state aggiornate per utilizzare file di suddivisione (train/test) configurabili tramite un form della web app.

3.2 Modello Base

Il modulo `predictive_model.py` addestra un modello di regressione logica con parametri fissi utilizzando:

- La feature `age` e il numero di corsi seguiti.
- Un target `teacher` definito in base alla presenza di corsi insegnati.

3.2.1 Esempio di utilizzo

```
from predictive_model.predictive_model import train_predictive_model

dataset_path = os.path.join("data", "dataset.csv")
base_model = train_predictive_model(dataset_path, random_state=42)
```

Listing 3.1: Utilizzo del modello predittivo base

3.3 Modello Grid Search

Il modulo `grid_search_model.py` implementa una procedura di Grid Search con 5-fold cross-validation per:

- Ottimizzare i parametri del modello (es: il parametro C e il tipo di penalizzazione `l1` o `l2`).
- Trovare la configurazione ottimale in base all'accuracy media ottenuta durante la cross-validation.

Nel modello Grid search utilizziamo `np.logspace(-5,5,11)` per testare valori di C che vanno da 10^{-5} a 10^{5}. Questo permette di verificare se un'ampia variazione del grado di regolarizzazione influenza le prestazioni del modello.

- Per la **penalità** `l1`: testiamo i solver `liblinear` e `saga` (entrambi supportano `l1`)
- Per la **penalità** `l2`: testiamo `liblinear`, `saga`, `newton-cg` e `lbfgs`.

In questo modo, la Grid Search esplorerà combinazioni diverse per vedere se una configurazione ottimale riesce a gestire meglio il rumore e lo sbilanciamento.

Utilizziamo anche la metrica `balanced_accuracy` poiché è più adatta per problemi con classi sbilanciate, in quanto tiene conto del recall per entrambe le classi.

3.3.1 Esempio di utilizzo

```
from predictive_model.grid_search import train_with_grid_search

dataset_path = os.path.join("data", "dataset.csv")
best_model = train_with_grid_search(dataset_path, random_state=42)
```

Listing 3.2: Utilizzo del modello predittivo con Grid Search

Per entrambi i modelli usiamo il parametro `class_weight = balanced` per assegnare pesi maggiori alla classe minoritaria.

3.4 Confronto delle Prestazioni

Il modulo `compare_models.py` confronta le prestazioni tra il modello base e quello ottimizzato:

- Suddivide il dataset in training e test set.
- Calcola l'accuracy e il classification report per ciascun modello.
- Stampa a video i risultati per facilitarne il confronto.

3.4.1 Esempio di utilizzo

```
from predictive_model.compare_models import compare_models

dataset_path = os.path.join("data", "dataset.csv")
compare_models(dataset_path)
```

Listing 3.3: Comparare i due modelli: base e grid search

Capitolo 4

Utilizzo di Modelli Predefiniti con PyKEEN

Utilizziamo PyKEEN per addestrare modelli di embedding sul knowledge graph. Le triple vengono caricate direttamente come lista di triple, poi suddivise in training, testing e validation tramite il metodo `split`. Esempio:

```
import numpy as np
from pykeen.pipeline import pipeline
from pykeen.triples import TriplesFactory
def extract_triples(self):
    onto.load()
    self.triples = []
    try:
        Person = onto.ontology.Person
    except AttributeError:
        raise ValueError("La classe 'Person' non e' definita nell' ontologia.")

    for person in Person.instances():
        head = person.name
        if hasattr(person, "takes") and person.takes:
            for course in person.takes:
                self.triples.append((head, "takes", course.name))
        if hasattr(person, "teaches") and person.teaches:
            for course in person.teaches:
                self.triples.append((head, "teaches", course.name))

# Converti la lista di triple in un array NumPy
def train_model(self):
    self.extract_triples()
    triples_array = np.array(self.triples)

    tf = TriplesFactory.from_labeled_triples(triples_array)
    print(tf)

    tf_train, tf_test, tf_valid = tf.split([0.8, 0.1, 0.1])

    result = pipeline(
        training=tf_train,
        testing=tf_test,
```

```
        validation=tf_valid,
        model="TransE",  # ad es., usa il modello TransE
        training_kwargs=dict(num_epochs=100),
    )
```

Listing 4.1: Utilizzo di PyKEEN per il training degli embedding

4.1 Visualizzazione e Clustering degli Embedding

Una volta ottenuti gli embedding da PyKEEN, si possono esplorare in 2D/3D e analizzare la struttura semantica:

4.2 Riduzione della Dimensionalità con PCA e t-SNE

Utilizziamo PCA e t-SNE per ridurre la dimensionalità e visualizzare i dati:

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

entity_embedding = result.model.entity_representations[0]
embeddings_tensor = entity_embedding()
self.embeddings = embeddings_tensor.detach().cpu().numpy()

entity_to_id = tf.entity_to_id
id_to_entity = {v: k for k, v in entity_to_id.items()}
self.entity_labels = [id_to_entity[i] for i in range(len(id_to_entity))]

self.pca()
self.tsne2D()
self.pca3D()

# # --- A) Usando PCA per una visualizzazione 2D
def pca(self):
    pca = PCA(n_components=2)
    embeddings_2d = pca.fit_transform(self.embeddings)

    plt.figure(figsize=(10, 8))
    plt.scatter(embeddings_2d[:, 0], embeddings_2d[:, 1], s=50, alpha
        =0.7)
    for i, label in enumerate(self.entity_labels):
        plt.annotate(self.find_name(label), (embeddings_2d[i, 0],
            embeddings_2d[i, 1]), fontsize=8, alpha=0.75)
    plt.title("Visualizzazione␣2D␣delle␣entita'␣(PCA)")
    plt.xlabel("PC1")
    plt.ylabel("PC2")
    plt.grid(True)
    plt.show()

# --- B) Usando t-SNE per una visualizzazione 2D (puo' evidenziare
    strutture non lineari)
def tsne2D(self):
    tsne_2d = TSNE(n_components=2, random_state=42, init="random")
```

```
35      embeddings_tsne_2d = tsne_2d.fit_transform(self.embeddings)
36
37      plt.figure(figsize=(10, 8))
38      plt.scatter(embeddings_tsne_2d[:, 0], embeddings_tsne_2d[:, 1], s
            =50, alpha=0.7, c='green')
39      for i, label in enumerate(self.entity_labels):
40          plt.annotate(self.find_name(label), (embeddings_tsne_2d[i, 0],
                embeddings_tsne_2d[i, 1]), fontsize=8, alpha=0.75)
41      plt.title("Visualizzazione␣2D␣delle␣entita'␣(t-SNE)")
42      plt.xlabel("Dimensione␣1")
43      plt.ylabel("Dimensione␣2")
44      plt.grid(True)
45      plt.show()
46
47  # --- C) Visualizzazione 3D con PCA
48  def pca3D(self):
49      pca_3d = PCA(n_components=3)
50      embeddings_3d = pca_3d.fit_transform(self.embeddings)
51
52      fig = plt.figure(figsize=(10, 8))
53      ax = fig.add_subplot(111, projection='3d')
54      ax.scatter(embeddings_3d[:, 0], embeddings_3d[:, 1], embeddings_3d
            [:, 2], s=50, alpha=0.7)
55      for i, label in enumerate(self.entity_labels):
56          ax.text(embeddings_3d[i, 0], embeddings_3d[i, 1], embeddings_3d[
                i, 2], self.find_name(label), size=8, zorder=1, color='k')
57      ax.set_title("Visualizzazione␣3D␣delle␣entita'␣(PCA)")
58      ax.set_xlabel("PC1")
59      ax.set_ylabel("PC2")
60      ax.set_zlabel("PC3")
61      plt.show()
```

Listing 4.2: Visualizzazione con PCA e t-SNE

Capitolo 5

Interfacce Utente: CLI e Web App

5.1 Interfaccia a Riga di Comando (CLI)

Il file `cli.py` gestisce le operazioni come:

- Popolare l'ontologia.
- Estrarre il dataset.
- Addestrare modelli predittivi (base e con Grid Search).
- Addestrare modelli neurali per apprendere embedding (PyKEEN).
- Visualizzare gli embedding.

5.1.1 Esempi di Esecuzione dal CLI

Per eseguire le operazioni via CLI, aprire il terminale nella root del progetto ed eseguire:

```
# Popola l'ontologia:
python src/cli.py populate

# Estrae il dataset e lo salva in data/dataset.csv:
python src/cli.py extract

# Addestra il modello base:
python src/cli.py train

# Addestra il modello ottimizzato con Grid Search:
python src/cli.py grid_search

# Confronta le prestazioni tra il modello base e quello ottimizzato:
python src/cli.py compare

# Comando per addrestrare il modello con TransE
python src/cli.py learn_graph
```

5.2 Interfaccia Web

L'applicazione web (definita in `app.py`) consente di eseguire le stesse operazioni tramite un'interfaccia grafica. Per avviare l'app:

```
python app.py
```

Successivamente, aprire il browser e navigare all'indirizzo:

```
http://127.0.0.1:4996/
```

Da qui è possibile interagire con l'applicazione per visualizzare il dataset estratto e i risultati dell'addestramento del modello.

Capitolo 6

Istruzioni per l'Installazione

6.1 Prerequisiti

- Python 3.6 o superiore.
- Le dipendenze elencate in `requirements.txt` (installabili con `pip install -r requirements.txt`).
- Java 21 o superiore.

6.1.1 Installazione di Java

Per eseguire alcune parti del nostro progetto, è necessario avere installato il Java Development Kit (JDK). Di seguito sono riportate le istruzioni per installare Java sui principali sistemi operativi.

Windows

1. **Scarica il JDK:** Visita il sito ufficiale di Oracle a `https://www.oracle.com/java/technologies/javase-downloads.html` oppure utilizza una distribuzione open-source come **Adoptium/Temurin** all'indirizzo `https://adoptium.net/`.
2. **Installa il JDK:** Esegui il file di installazione scaricato e segui le istruzioni a schermo.
3. **Verifica l'installazione:** Apri il Prompt dei Comandi e digita:

   ```
   java --version
   ```

   Dovresti vedere un output che conferma la versione di Java installata.

Linux (Ubuntu)

1. **Aggiorna i repository:** Apri un terminale ed esegui:

```
sudo apt update
```

2. **Installa OpenJDK:** Per installare OpenJDK 11, esegui:

```
sudo apt install openjdk-11-jdk
```

3. **Verifica l'installazione:** Digita nel terminale:

```
java --version
```

Dovresti vedere l'output relativo alla versione di OpenJDK installata.

macOS

1. **Utilizzo di Homebrew:** Se non hai già Homebrew installato, segui le istruzioni sul sito ufficiale (`https://brew.sh/`). Una volta installato, apri il Terminale ed esegui:

```
brew update
brew install openjdk
```

2. **Configura Java:** Dopo l'installazione, potrebbe essere necessario aggiungere Java al PATH del sistema: se stai utilizzando la shell zsh

```
echo 'export PATH="/opt/homebrew/opt/openjdk/bin:$PATH"' >> ~/.zshrc
```

Invece per bash

```
echo 'export PATH="/opt/homebrew/opt/openjdk/bin:$PATH"' >> ~/.bashrc
```

3. **Verifica l'installazione:** Digita nel Terminale:

```
java --version
```

Dovresti visualizzare la versione di OpenJDK installata.

6.2 Esecuzione

- Per eseguire il progetto via CLI, utilizzare i comandi indicati nella sezione "Esempi di Esecuzione dal CLI".
- Per avviare l'interfaccia web, eseguire `python app.py`
  e accedere all'indirizzo `http://127.0.0.1:4996/`.

Capitolo 7

Conclusioni e Sviluppi Futuri

Il progetto **Dataset Builder Ontology** rappresenta una piattaforma integrata per:

- La creazione e la gestione di ontologie.
- L'estrazione e il preprocessing di dataset da ontologie.
- L'addestramento di modelli predittivi tradizionali e neurali, con ottimizzazione automatica degli iperparametri.
- La generazione di embedding semantici (modelli PyKEEN) e la loro analisi tramite visualizzazione e clustering.
- L'interazione tramite CLI e una web app interattiva per configurazioni e monitoraggio dei processi.

Gli sviluppi futuri potrebbero includere:

- L'integrazione di ulteriori algoritmi di deep learning e interpretabilità dei modelli (ad es. SHAP o LIME).
- L'implementazione di dashboard interattive per monitorare le performance in tempo reale.
- L'espansione del knowledge graph con ulteriori dati e relazioni, migliorando così la qualità degli embedding.

Bibliografia

[1] Owlready2: A module for ontology-oriented programming in Python. https://pypi.org/project/Owlready2/

[2] Scikit-learn Developers, Scikit-learn: Machine Learning in Python. https://scikit-learn.org/

[3] Scikit-learn Developers, GridSearchCV Documentation. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[4] Flask: A lightweight WSGI web application framework. https://flask.palletsprojects.com/

[5] PyKEEN: A Python library for learning and evaluating knowledge graph embeddings. https://pykeen.readthedocs.io/

[6] Matplotlib: A 2D graphics environment. Available at: https://matplotlib.org/.

[7] Array programming with NumPy. Available at: https://numpy.org/.