SOFTWARE ARCHITECTURE AND PATTERN DESIGN

ENHANCING MACHINE LEARNING DATASETS THROUGH ONTOLOGY-DRIVEN FEATURE AUGMENTATION

STUDENTS: Alessio Mattiace, Claudio Saponaro

INDEX

ABSTRACT

This project leverages the integration of a diabetes ontology for enhancing predictive power in diabetes prediction given a set of features.

Ontologies, as structured frameworks for organizing knowledge, allow for the enrichment of datasets by incorporating domain expertise, semantic relationships, and contextual information. In our work, a diabetes ontology was specifically tested.

After the integration, a set of formal rules was applied to uncover more complex and personalized relationships related to the patient, derived from their clinical data.

Two approaches were tested to obtain a dense representation which is crucial to train the model: OWL2Vec* and Node2Vec.

The enriched dataset was then employed to train the ML model, which was subsequently evaluated against the model trained on the simple dataset using the specific classification metrics.

The results suggest that the proposed system, given an adequately sized input dataset, can improve the performance of a machine learning model. This is particularly significant for tasks requiring high precision, such as those in the medical or banking domains.

SECTION 1 – STATE OF ART

This section discusses the state-of-the-art of the tested and implemented embedding models, namely OWL2Vec* and Node2Vec.

1.1 - OWL2Vec*

Firstly, we have used this framework since it, as the paper suggests, provides and end-to-end approach on translating the ontology structure and semantics into a dense numerical representation (embedding) which is crucial for training the machine learning model.

The OWL2Vec* framework involves three main steps:

- reasoning new axioms from SWRL rules
- extracting a corpus of text sentences from the ontology
- training a word embedding model with the obtained corpus.

The corpus includes three documents: a structured document capturing the ontology's graph structure, a lexical document that captures the semantics, the combined document combines the topological structure obtained from the structured document and the semantics obtained from the lexical one.
Then the combined document is used to train a Word2Vec model.

In our initial implementation, we have followed the research methodology proposed in the OWL2Vec* paper, fine-tuning a pre-trained Word2Vec model on ontology-based semantics: the embeddings are not generated only based on the information in our ontology, but also by using general semantic knowledge through a pre-trained Word2Vec model.

Overall framework

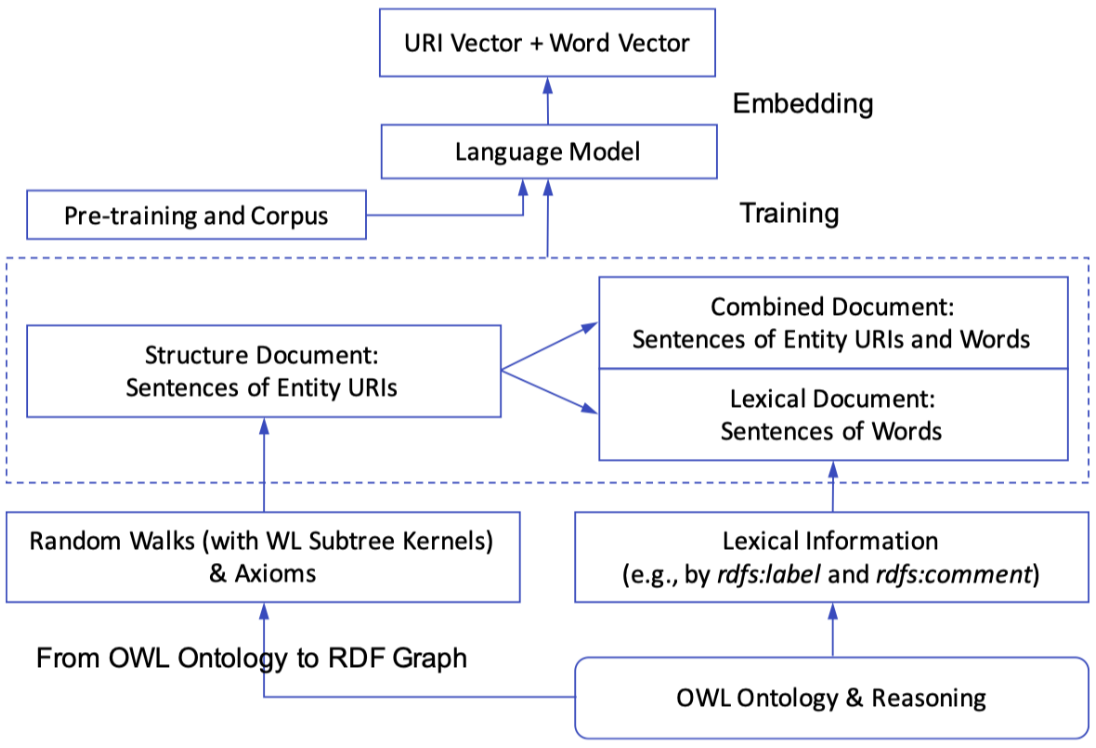

URI Vector + Word Vector
Embedding
Language Model
Pre-training and Corpus
Training
Combined Document: Sentences of Entity URIs and Words
Structure Document: Sentences of Entity URIs
Lexical Document: Sentences of Words
Random Walks (with WL Subtree Kernels) & Axioms
Lexical Information (e.g., by *rdfs:label* and *rdfs:comment*)
From OWL Ontology to RDF Graph
OWL Ontology & Reasoning

Fig. 1.1 – The overall structure of OWL2Vec framework [2].*

First, we can observe in the figure that there are 2 branches which followed the reasoning steps:

The first branch ('From OWL Ontology to RDF Graph') provides a strategy for converting the ontology to an RDF graph using the projection rules.

The structure document aims at capturing both the graph structure and the logical constructors of the ontology obtained by random walking in the graph at the previous step [2].

The lexical document, on the other hand, includes two types of word sentences: the first type is generated from the entity IRI sentences in the structure document, while the second is extracted from the relevant lexical annotation axioms in the ontology (the branch that extends from 'OWL Ontology and Reasoning' to 'Lexical Information').

Lastly in the dashed part of the figure the lexical document and the structure document are combined into the Mix document to obtain a single document that will be used for training the Word2Vec model from scratch or, in our case, to fine-tune the existent pre-trained Word2Vec model.

Word2Vec is a well-known group of sequence feature learning techniques for learning word embeddings from a large corpus of data and was initially developed by a team at Google; it can be configured to use either skip-gram or CBOW architectures [2].

1.2 - Node2Vec

Node2Vec is a semi-supervised algorithm for scalable feature learning in networks. It is designed to create vector representations of nodes in a graph which we have used to train our machine learning model.

Since our ontology is inherently a graph, this approach is very suitable for our purpose.

Let's examine what are the key concepts behind this algorithm:

1. Graph representation: Node2Vec uses random walks to explore the graph, generating sequences of nodes that can be treated like sentences in natural language processing (NLP).
   It's important to underline that *p* and *q* parameters affect the behavior of the walks; the *q* parameter is related more to BFS (Breadth-first Search) kind of exploration while the *p* parameter enhances a DFS (Depth-first Search) kind of exploration in the graph.
   Since our graph is not very deep, we preferred using a high value of q compared to p, focusing on traversing the graph in a Breadth-First Search manner.

2. NLP: These resulting sequences obtained after traversing the graph, are then feed into the Skip-gram model (Word2Vec-like model) to learn the embeddings.

Node2Vec Algorithm:

```
**LearnFeatures** (Graph $G = (V, E, W)$, Dimensions $d$, Walks per
  node $r$, Walk length $l$, Context size $k$, Return $p$, In-out $q$)
  $\pi$ = PreprocessModifiedWeights($G, p, q$)
  $G' = (V, E, \pi)$
  Initialize $walks$ to Empty
  **for** $iter = 1$ **to** $r$ **do**
    **for all** nodes $u \in V$ **do**
      $walk$ = node2vecWalk($G', u, l$)
      Append $walk$ to $walks$
  $f$ = StochasticGradientDescent($k$, $d$, $walks$)
  **return** $f$
```

```
**node2vecWalk** (Graph $G' = (V, E, \pi)$, Start node $u$, Length $l$)
  Inititalize $walk$ to $[u]$
  **for** $walk\_iter = 1$ **to** $l$ **do**
    $curr = walk[-1]$
    $V_{curr}$ = GetNeighbors($curr$, $G'$)
    $s$ = AliasSample($V_{curr}, \pi$)
    Append $s$ to $walk$
  **return** $walk$
```

Fig. 1.2 - Node2Vec algorithm.[1]

First of all, let's understand in more detail what these parameters mean:

1. *r* : number of different walks that the node has, in other terms how many times the graph is explored differently.
2. *l* : dimension of the walk, (l=3 means that from the source node we have maximum a walk composed of 3 hops).
3. *k* : the dimension of the context that will be used later in skipgram training to learn the embeddings of the nodes.

The *PreprocessModifiedWeights* function instantiates the transition probabilities between nodes in the random walk using the *pi* variable.

The *GetNeighbors* function serves to retrieve the neighbor of the source node while the *AliasSample* function aims to select a neighbor based on the transition probabilities, this process is repeated until the walk length is reached.

Lastly the *StocasticGradientDescent* function updates the embeddings vectors in the Skipgram's architecture to minimize the loss function.

SECTION 2 – BACKGROUND & DESIGN

2.1 - Model Decision

As previously mentioned, for the generation of embeddings, we initially used OWL2Vec* as it is a more complex and comprehensive architecture. However, we later opted for the faster, more scalable, and efficient Node2Vec model.

Its speed allowed us to perform fine-tuning of its hyperparameters to achieve the best possible result.

2.2 – Activity Plan

First, we have adopted an agile methodology where we tested the code incrementally before going to the next step.

However, in the first phase we have thought simultaneously so we have used a design thinking approach to find a solution since there was not a previous solution to the problem of integrating an ontology to enhance a ML model.

So, summarising, we have first thought about an innovative solution to make possible integration an ontology type of knowledge within the dataset; after the creative process we have started coding toward the solution.

We have first adapted the ontology to our dataset to make possible the integration and then training the model.

2.3 – Technology Stack

In the realization process, we have used a lot of different technologies, so it is helpful to create a quick list which summarizes the application domain of this technologies:

- *OWL*: a semantic web language designed to represent complex knowledge about things, groups of things, and their relationships that are a key aspect in application domains such as finance and medicine in which a lot of variables come into play.

- *Protégé*: an open-source ontology editor and framework for building and modifying ontologies.

In our case we have used Protégé to modify an existing ontology based on diabetes to be compatible in the ‘loading process’ (explained in detail in section 2).

We have used the SWRL Tab in Protégé for building the SWRL logical rules which are a key aspect in creating different relationships depending on the patient personal data (this difference represents a crucial aspect in the embedding creation step, explained in detail in section 3).

- *Owlready2*: a Python library for populating the ontology with the data taken from the dataset.
- *Pellet*: an inference engine capable of deducing new relationships between entities based on SWRL rules.
- *Rdflib*: a Python library for getting a graph structure from the ontology which serves as input for the main Word2Vec model.
- *Sklearn*: a Python library for automatizing regular machine learning processes such as splitting and scaling the dataset.
- *Torch*: A library providing data structures for multi-dimensional tensors and performing high-speed mathematical operations, particularly leveraging GPUs for acceleration.
- *NumPy*: a Python library for optimizing the array’s operation over more dimensions.
- *Pandas*: Python library for dataset pre-processing.
- *Matplotlib* and *Seaborn*: Python libraries for visualizing the results.
- *OWL2Vec**: a framework for encoding both the semantics of the ontology and its topological structure [2]; first architectural choice, then discarded.
- *Node2Vec*: an algorithmic framework for learning continuous representations for nodes in networks [1]; second architectural choice.

2.4 – Non-functional requirements

Below are the main non-functional requirements that we considered during the process of creating our system.

Performances:
The architectural choices were made to maximize the values of accuracy, precision, and recall after training the ML model; results were always compared with the model trained using only the dataset without enhancement.

Scalability:

We have aimed to make our architecture as scalable as possible by adopting a faster and more efficient embedding model.

By optimizing resource allocation, we have ensured that the system can handle increasing workloads by using Node2Vec embedding model instead of OWL2Vec* and PyTorch tensors in ML model training ensuring GPU usage.

Usability:
The integration between the dataset and its associated ontology is very straightforward at a formal level; therefore, it is easily generalizable to new scenarios involving the use of different datasets and ontologies.

2.5 – Design approach & MLops

Since this is an experimental project, we have not deployed the machine learning model, but if we have done so we would have used this specific approach:

- GCP (Google Cloud AI Platform): for using the GPU resources in the model training phase, for real-case large dataset and architectures scenarios.
- Docker: for containerizing and isolate our services which are the embeddings creation and the model training.
- Spark: employed for real-case scenarios where large dataset are used; the *pyspark* library provides a high-level API for distributed big data-processing.
- MLflow: an open-source platform that offers a comprehensive solution for managing the entire ML lifecycle, including model versioning.

- Prometeus: an open-source monitoring and alerting toolkit for tracking the CPU, GPU and memory usage employed for the embedding creation and model training.

SECTION 3 – ONTOLOGY AND DATASET STRUCTURE: RULES AND FORMALISM

The architecture we designed requires the dataset and ontology to follow specific rules for the system to function; in this section, a formal structure will be presented that can serve as a guideline for potential users, along with a graphical representation of the dataset and ontology we used for testing.

3.1 – Dataset and ontology rules

For both classification and regression tasks, the input dataset for our system must exhibit three specific characteristics:

1) The first column must represent the ID of the *main_individual*, i.e., the primary entity on which the task is based (the subject to which each data sample refers). If a row in our dataset corresponds to a single person, the first column could be named "Person" and contain unique IDs for each individual.

2) The last column must necessarily contain the target variable, also referred to as the outcome, which is essential for calculating the loss function during the training of our model.

3) All the intermediate columns must refer to the features used to train the model.

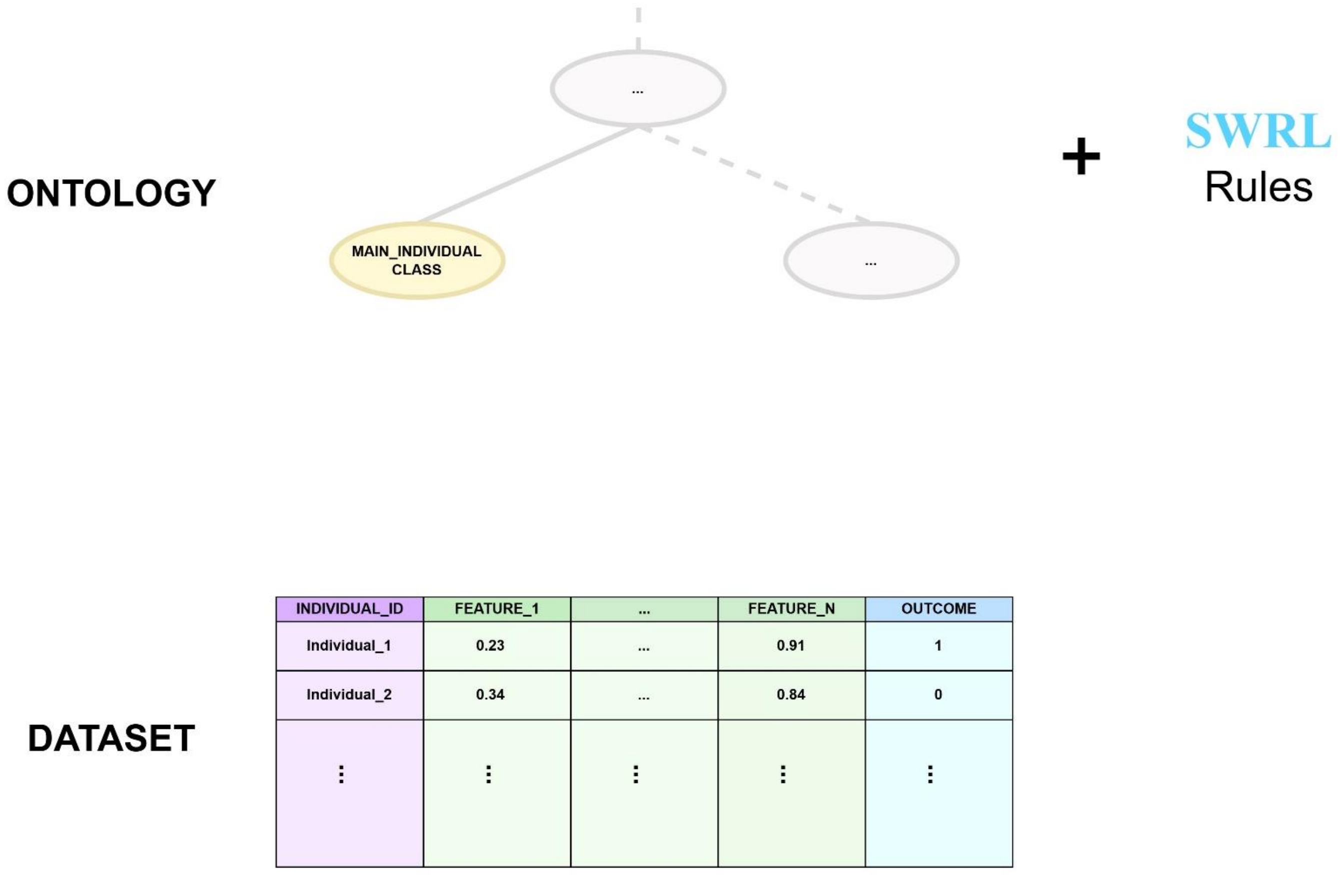
For the ontology, there are only two main rules:

1) There must be a class with the same name as the first column of the dataset, corresponding to the *main_individual* previously mentioned (e.g., a class named "Person").

2) It must contain SWRL rules that allow, after inference with an appropriate reasoner such as Pellet or Hermit, the deduction of additional relationships. An example of a logical rule could be:

$$\textit{if Person\_1 hasAge} > 70 \quad \overset{(so)}{\rightarrow} \quad \textit{Person\_1 is Old.}$$

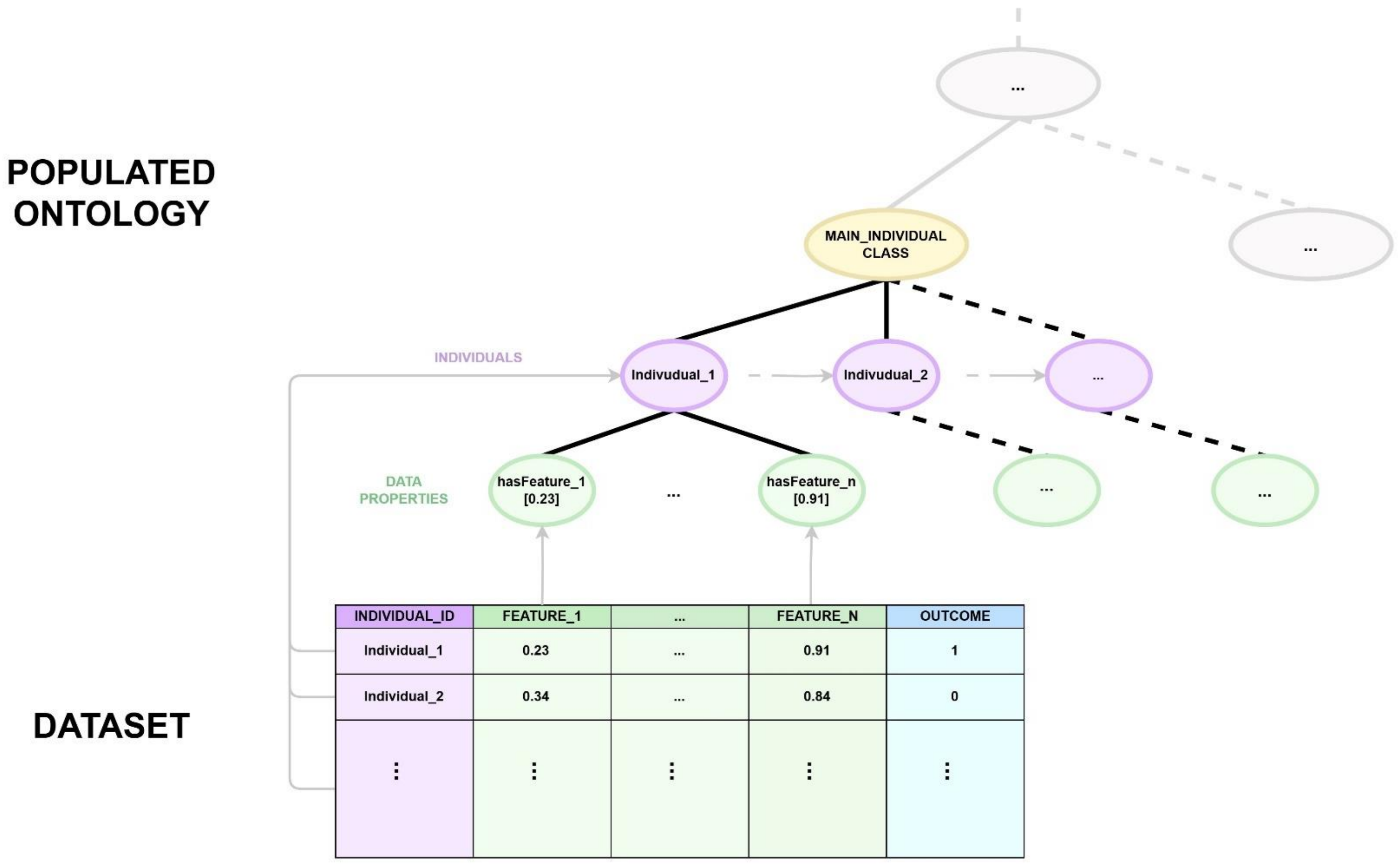
Below is a graphical representation of the structure that the ontology and dataset must have in order to comply with all the mentioned rules:



INDIVIDUAL_ID	FEATURE_1	...	FEATURE_N	OUTCOME
Individual_1	0.23	...	0.91	1
Individual_2	0.34	...	0.84	0
⋮	⋮	⋮	⋮	⋮

Fig. 3.1 – Ontology and dataset standard structure visualization.

If a suitable dataset and ontology are provided as input to the system, the first step is to populate the ontology with the information contained in the dataset. Below is a representation of how this process takes place:



INDIVIDUAL_ID	FEATURE_1	...	FEATURE_N	OUTCOME
Individual_1	0.23	...	0.91	1
Individual_2	0.34	...	0.84	0
⋮	⋮	⋮	⋮	⋮

Fig. 3.2 – ontology population using dataset information.

The ontology is populated with as many individuals of the Main_individual class as there are rows in the dataset. These individuals will be named according to the individual_ID present in the first column (for example, if each row refers to a person, the class "Person" will be populated with person_1, person_2, etc.).

Next, for each individual data properties are assigned, which correspond to the features obtained from the dataset. These properties will have the same name, prefixed by "has" (for example, the feature "Price" will be mapped to the data property "hasPrice" in the ontology).

Each data property will have an associated value; for example, "Vehicle_23" might have the property "hasPrice=9000").

3.2 - Dataset and ontology tested

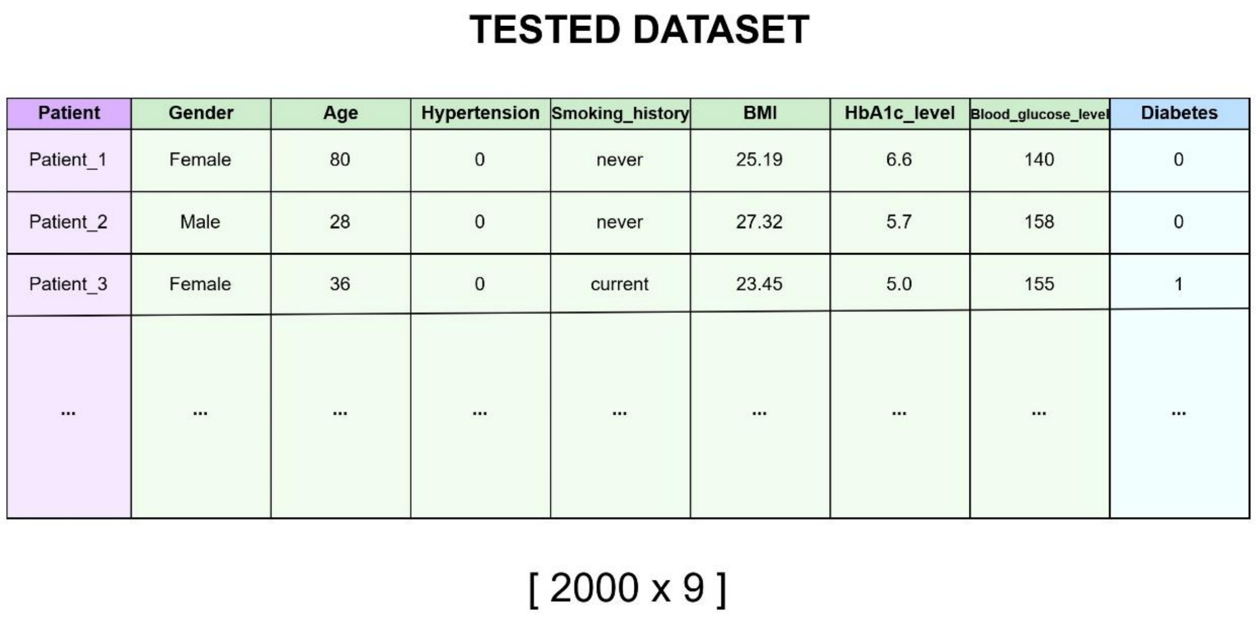
We used a diabetes dataset to train a machine learning model on classification tasks to recognize the presence or absence of diabetes. The dataset contains a column for IDs, 7 features, and a column for outcomes, as labels (with a size of 2000 data samples).

It was downloaded from:

https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset

The ontology, on the other hand, was modelled by us to be suitable, based on the one available at the following address:

https://bioportal.bioontology.org/ontologies/CDMONTO/?p=classes&conceptid=http%3A%2F%2Fwww.semanticweb.org%2Fkhaled%2Fontologies%2F2024%2F7%2FCDMOnto%23Diabetes&lang=en



TESTED DATASET

Patient	Gender	Age	Hypertension	Smoking_history	BMI	HbA1c_level	Blood_glucose_level	Diabetes
Patient_1	Female	80	0	never	25.19	6.6	140	0
Patient_2	Male	28	0	never	27.32	5.7	158	0
Patient_3	Female	36	0	current	23.45	5.0	155	1
...	...	...	...	...	...	...	...	...

[2000 x 9]

Fig. 3.3 – Tested dataset representation.

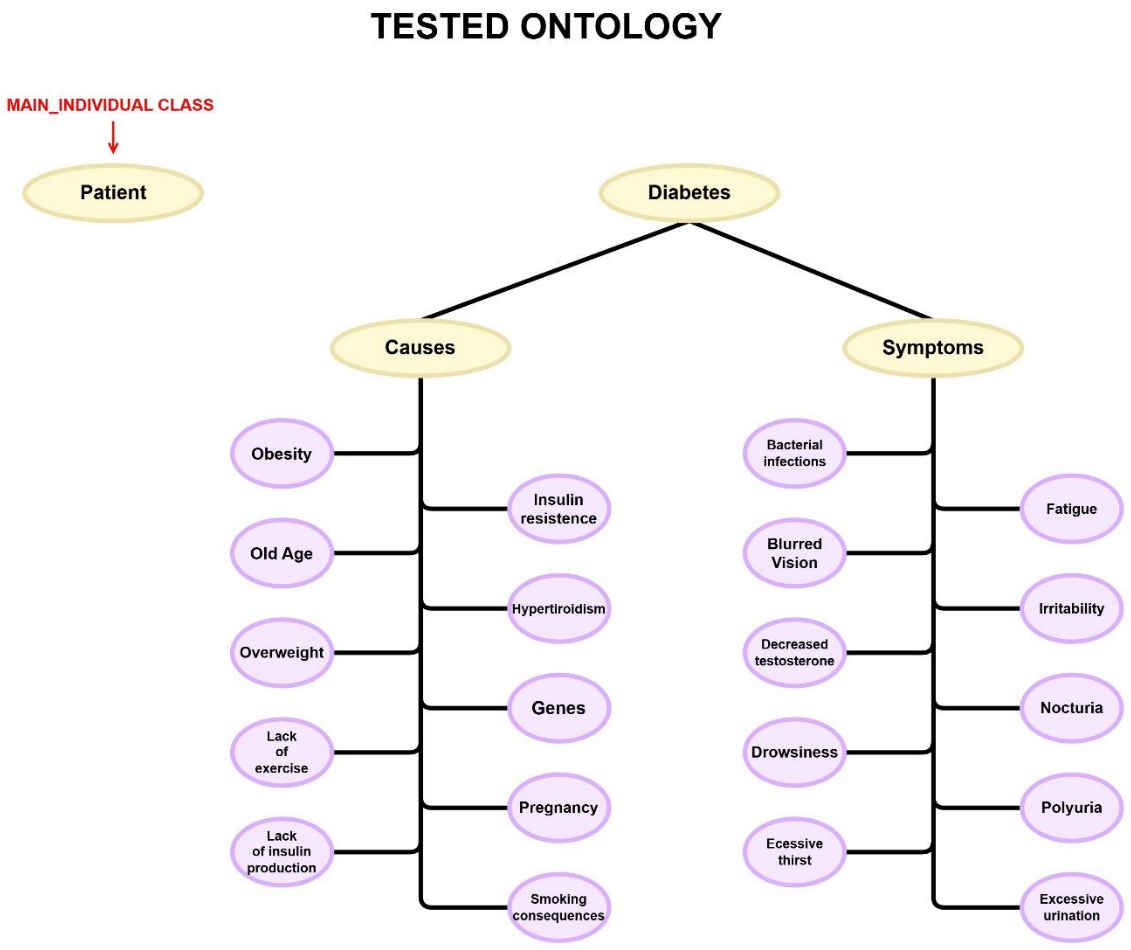


Fig. 2.4 – Tested ontology representation.

SECTION 4 – ONTOLOGY EMBEDDINGS GENERATION

In this section, each function involved in the embedding creation process in the script *OWL_Embedding_Generator_N2V.py* will be explained. This program, given the dataset and the ontology as input, returns the embeddings obtained using Node2Vec. These embeddings are generated "Offline," meaning not during model training, but rather in a separate file saved in ".npy" format, which can be used in the Main.

There will also be a brief representation of the process pipeline using OWL2Vec* (in the script *OWL_Embedding_Generator_O2V.py*).

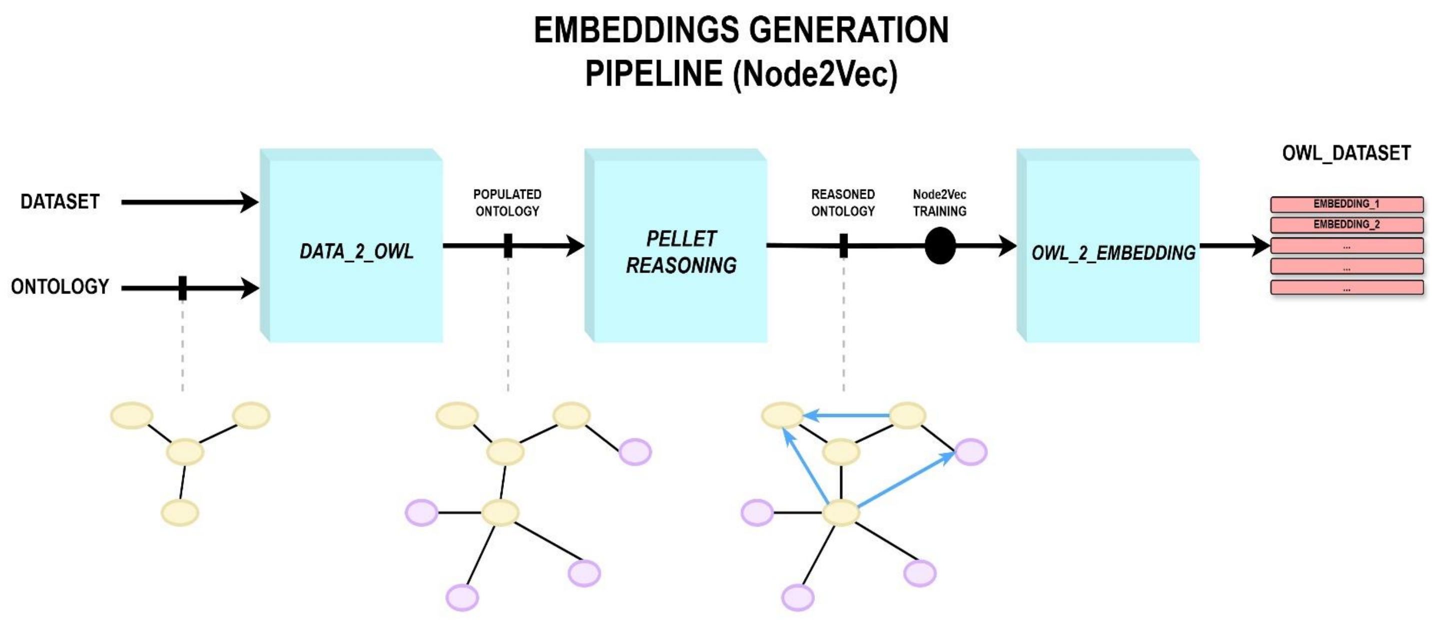


Fig. 4.1 – Embedding generation pipeline with Node2Vec.

The ontology is first processed using the function data_2_owl, which populates the ontology according to the procedure already outlined. Next, reasoning is performed to infer new axioms using SWRL rules with Pellet.

After training the Node2Vec model, the embeddings are calculated, and a vertical stack is created to obtain the OWL_dataset (a dataset containing only the information derived from the ontology).

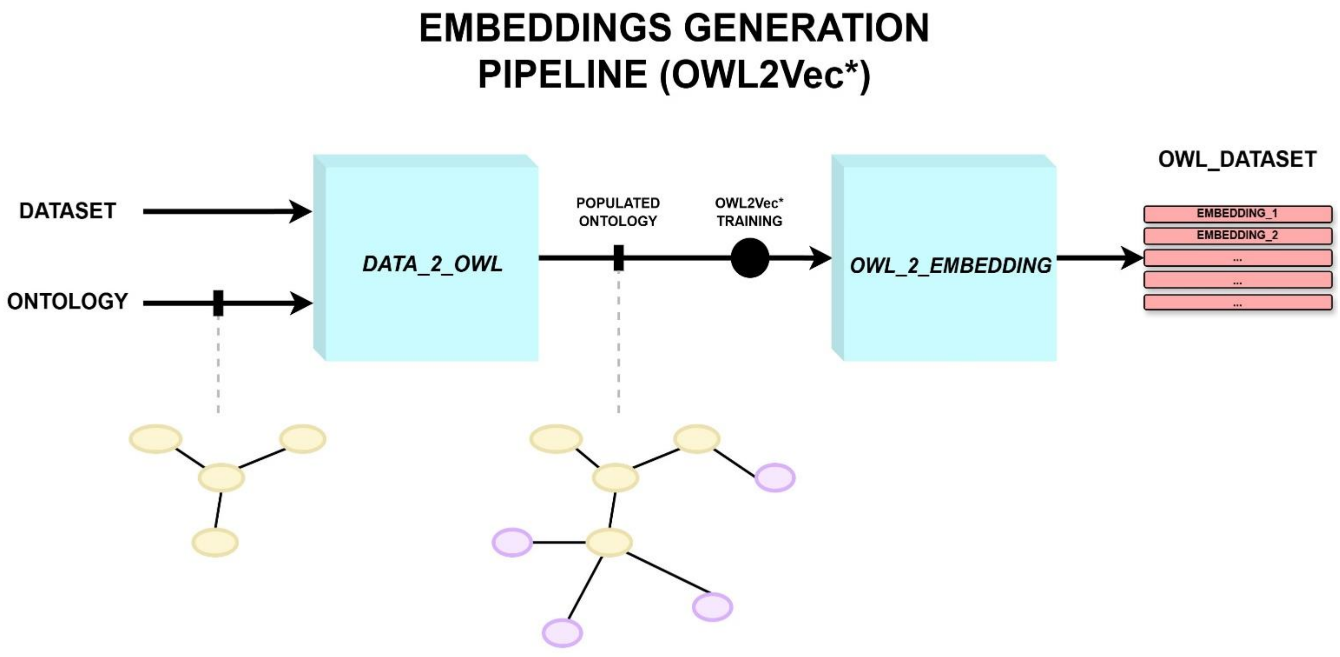


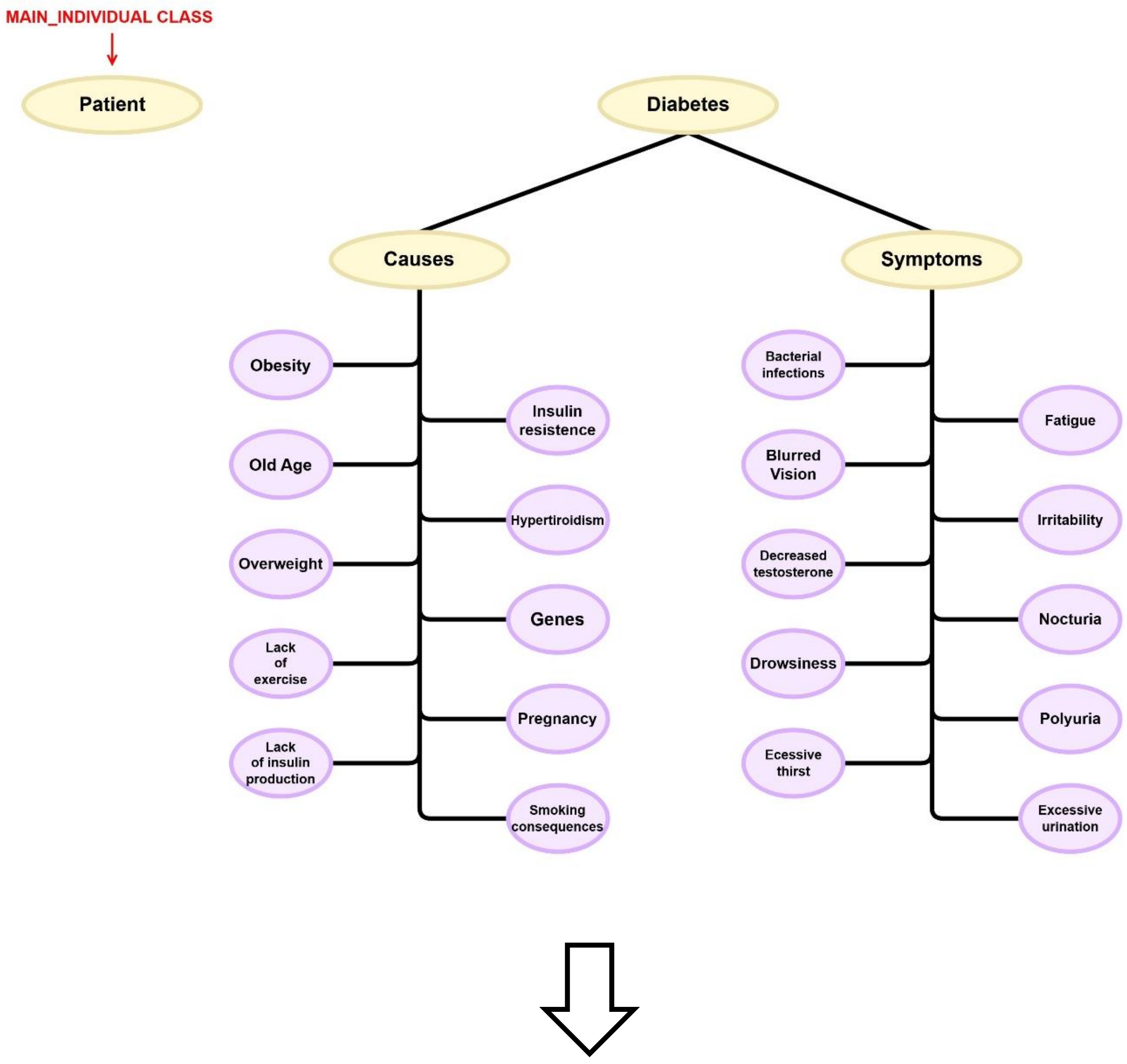
Fig. 4.2 – Embedding generation pipeline with OWL2Vec.*

In the initial implementation, using OWL2Vec* as the model, the pipeline is very similar, but the function for reasoning new axioms is not present, as it is integrated within the model itself and executed just before training.

4.1 – data_2_OWL process

The function that allows populating the ontology based on the information present in the dataset identifies the Main_individual class. It first iterates over the rows of the dataset to populate it with individuals, each given a unique name based on the ID present in the first column. Then, it assigns values to each individual's data properties by iterating over the features.

Below is a representation of how the ontology we tested was populated:



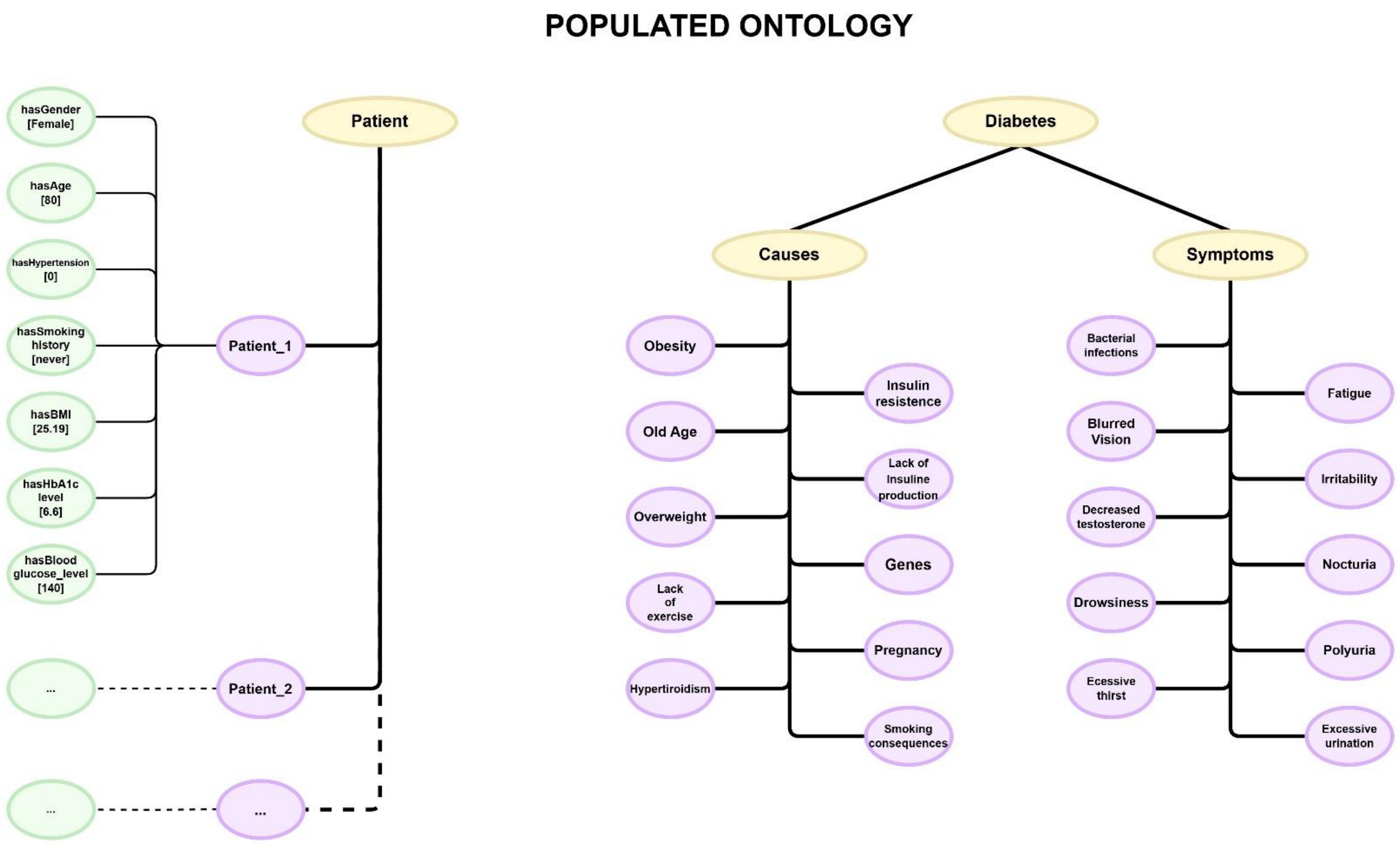


Fig. 4.3 – tested ontology population representation.

4.2 – SWRL rules reasoning

As already mentioned, in addition to the semantics of the sentences, the model primarily learns from the structure of the graph. For this reason, it is very important to define specific SWRL rules that, once inferred from the populated ontology, enable the creation of new relationships between entirely new entities.

In our scenario, for example, if Patient_53 has a BMI index above 30, he is considered obese: it is therefore very useful to deduce a rule that links the patient to the "Obesity" entity, which turn as a cause for Diabetes. The Node2Vec model can learn from these relationships, creating similar embeddings for similar patients (and different ones for different patients).

Using a specific function, we then perform reasoning with the Pellet inference engine.

Below is a representation of the ontology after the reasoning:

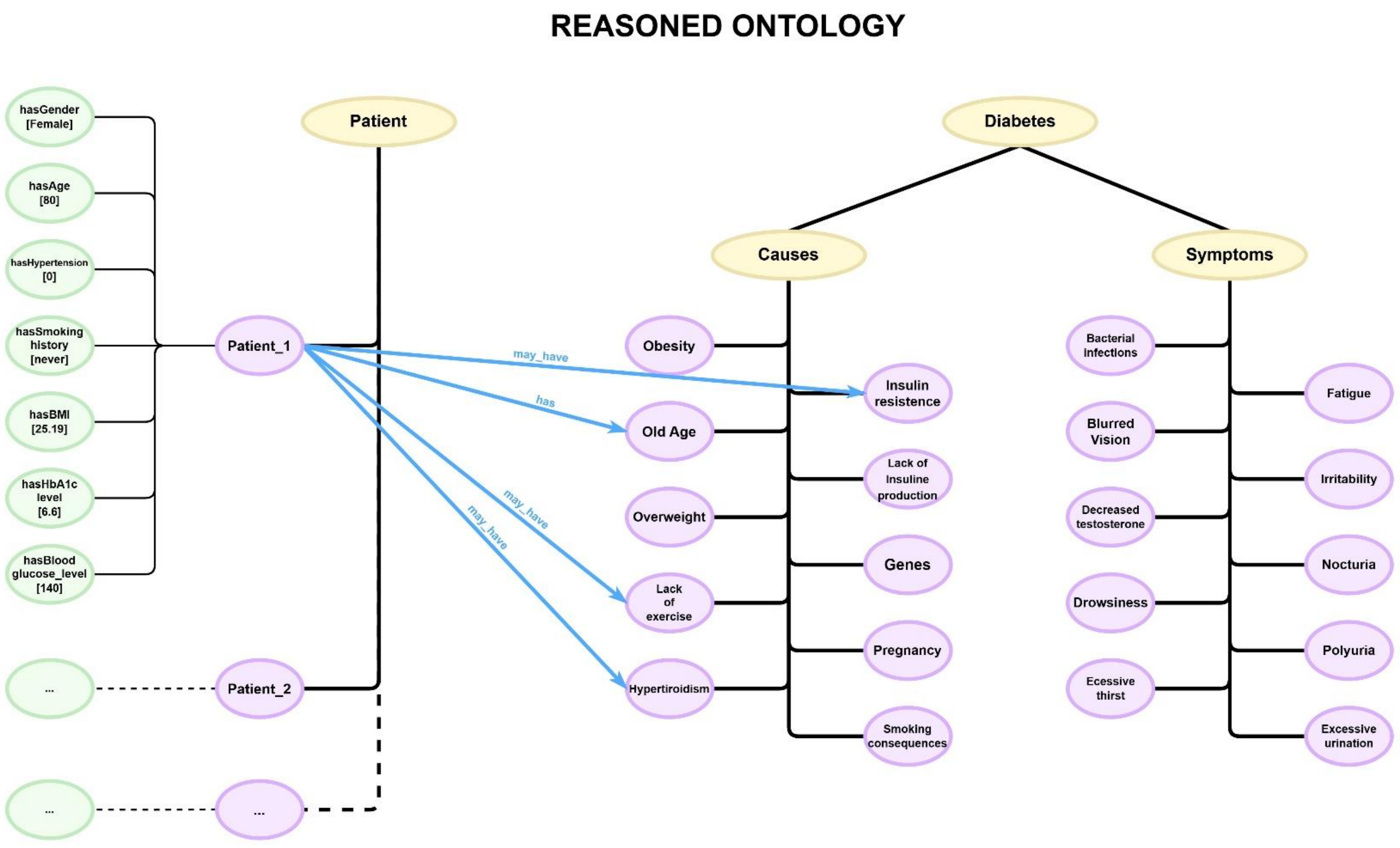


Fig. 4.4 – SWRL rules reasoning in tested ontology.

4.3 – OWL_2_embedding process

Once the final ontology is obtained, it can be transformed into a graph and used as input for training Node2Vec; in our case, it contained several thousand edges and nodes. It is important to select the appropriate hyperparameter values for the Node2Vec model, such as the vector size, the number of random walks, etc.

After training the model, it is enough to compute the embedding value for each individual in the Main_individual class and perform a simple vertical stack to obtain the OWL_dataset.

This procedure is fast and efficient for Node2Vec, but very slow for OWL2Vec*, requiring more than 36 hours on a regular laptop and cannot be optimize since we are not working with tensor yet.

SECTION 4 – MACHINE LEARNING PROCESS

This section will outline the pre-processing method and the machine learning model's architecture to obtain the prediction of the analyzed patients.

Before deep diving into the explanation, a visual overview of the pipeline of our ML process:

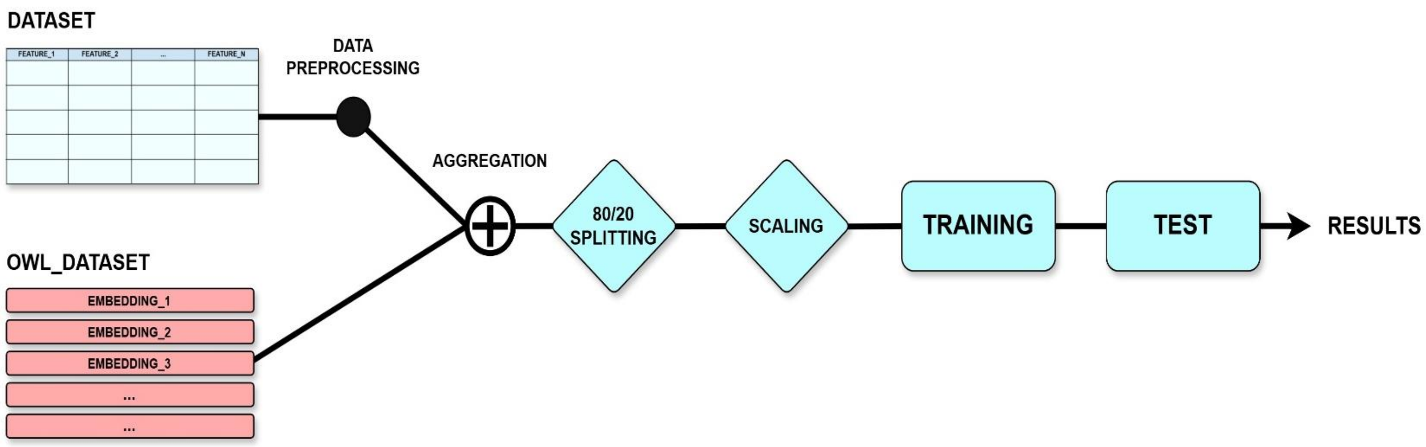


Fig. 4.1 – Machine learning process pipeline.

Below is an illustration of how OWL_dataset is concatenated with the original dataset (data aggregation or enhancing):

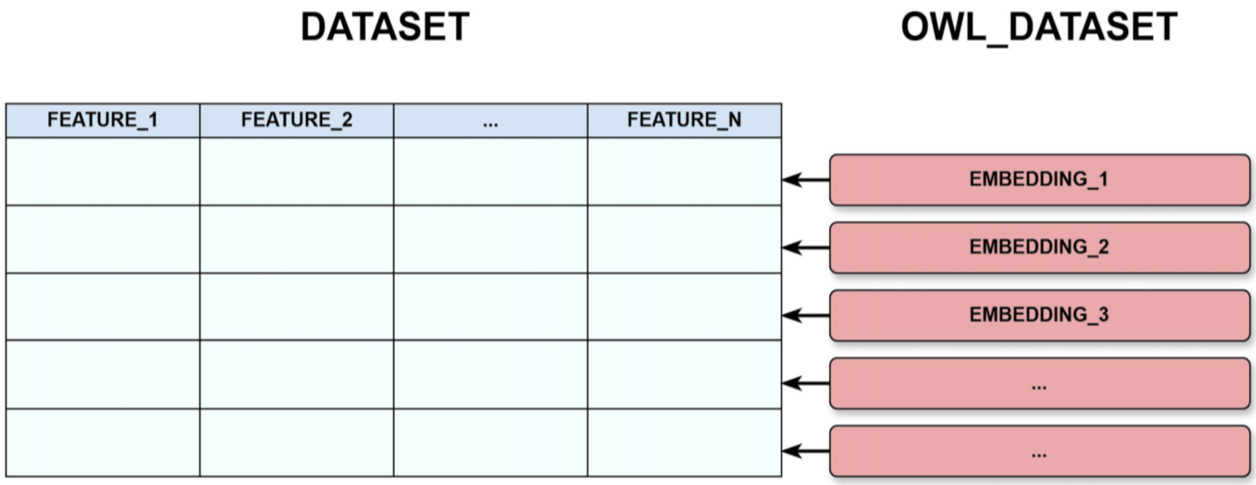


Fig. 4.2 - Visual representation of the embedding's horizontal stack.

4.1 – Neural Network model

The model used for classifying patients was a Neural Network with three layers. Below is the representation of its architecture:

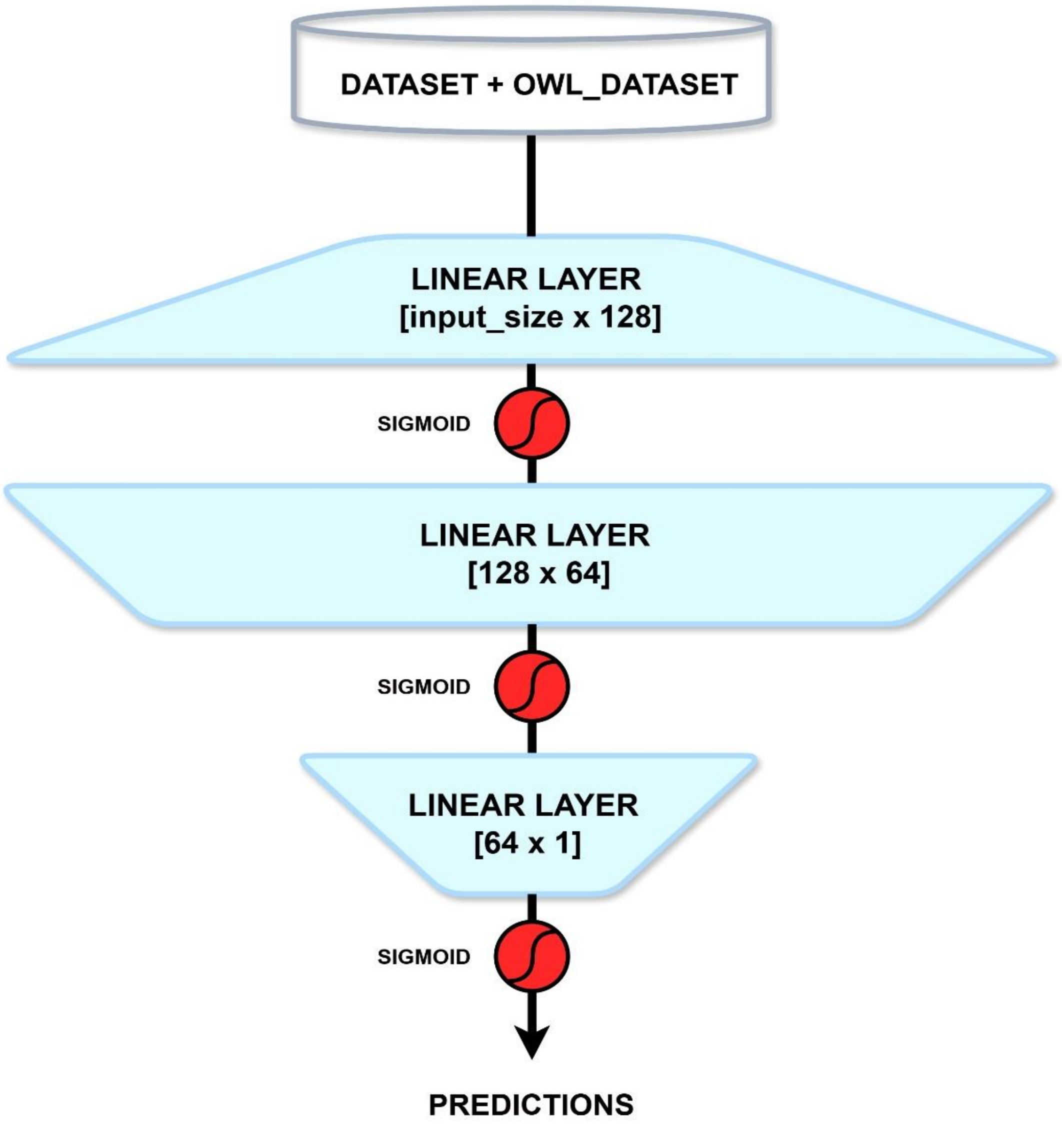


Fig. 4.3 – Neural Network model architecture.

The model takes as input the enhanced dataset, with an input size not defined a priori (in our case, 22, but this depends on the embedding_size and the size of the original dataset). We used three linear layers followed by sigmoid activation functions.

The final layer reduces the dimension to 1, producing the prediction as a single numerical value vector.

4.2 – Preprocessing, training and testing

Before performing the aggregation between the original dataset and the *OWL_dataset*, a light preprocessing is required: the first column, which contains the IDs, must be removed since it is a unique feature for each individual and not useful for training. Additionally, categorical features need to be processed.

In our test scenario, we performed the following operations:

- **Splitting the enhanced dataset** into a training set and a test set (in our case, 80% training and 20% test).
- **Scaling the features** to normalize the values; in our case, we used a standard scaler from the sklearn library.
- **Training the model weights** on the training set using CUDA (GPU) or CPU; we have used Adam as optimizer for better learning, a learning rate of 0.001, and 800 epochs (*Train.py* file).
- **Testing the performance** on the test set and printing the classification metrics and confusion matrix (*Test.py* and *Metrics_plot.py* files).

SECTION 6 – RESULTS

In the final part of our report, we have analyzed the results obtained from our architecture.

We have based on a dataset with 92,000 data samples, so, due to memory limitations and computational power of the machines on which the tests were conducted, we reduced the size to 2000 samples and performed tests on a further reduced version of 1000 samples (stratifying based on the target column).

The tested ontology, on the other hand, was modeled and adapted to our dataset.

The section is structured as follows:

- In the first part, we analyzed the results obtained with the reduced dataset (1000 samples):
    - Results using the initial dataset without additional ontology information (used for comparison).
    - Results using OWL2Vec*
    - Results using Node2Vec

- In the second part, we analyze the results obtained with the complete dataset (2000 samples):
    - Results using the initial dataset without additional ontology information (used for comparison).
    - Results using Node2Vec

6.1 – Results on 1000 samples dataset

DATASET_1000

Results obtained by training the model using the reduced dataset only without any OWL dataset integration.

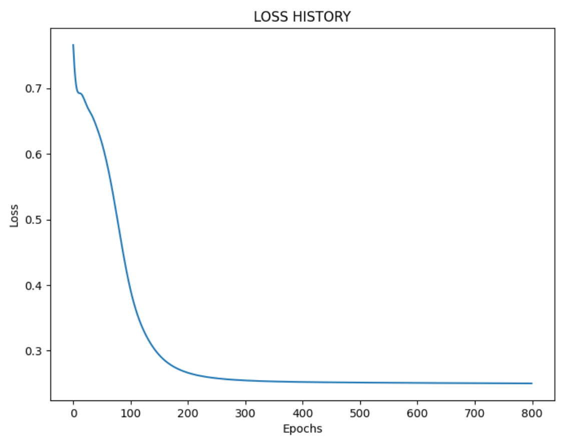
CLASSIFICATION METRICS

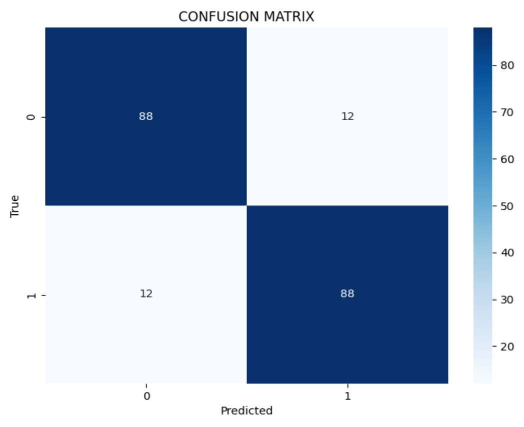
Accuracy: 88.00 %

Precision: 88.00%

Recall: 88.00%

F1_Score: 88.00%





OWL_DATASET_1000 [OWL2Vec*]

Results obtained by training the model using only the embeddings generated by OWL2Vec*.

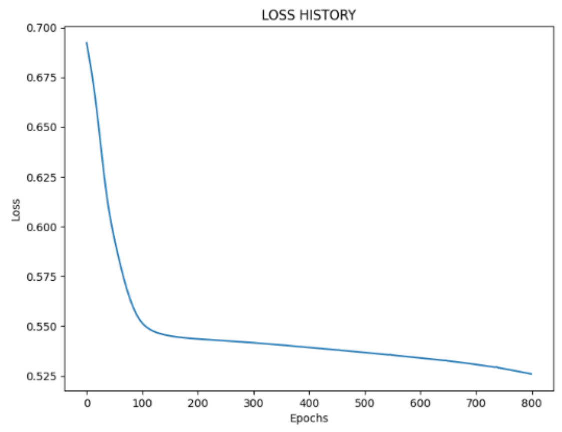
CLASSIFICATION METRICS

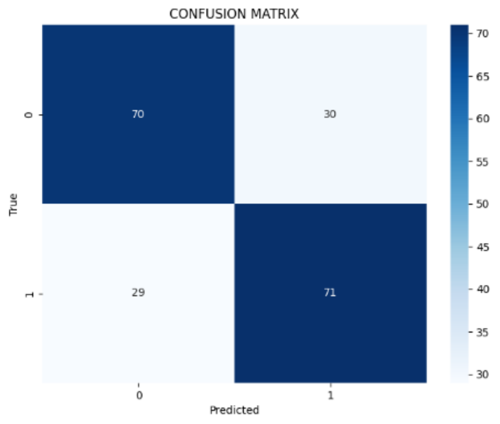
Accuracy: 70.50 %

Precision: 70.30%

Recall: 71.00%

F1_Score: 70.65%





OWL_DATASET_1000 [Node2Vec]

Results obtained by training the model using only the embeddings generated by Node2Vec.

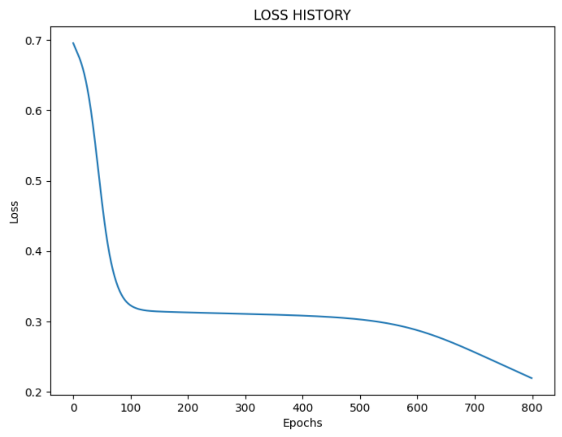
CLASSIFICATION METRICS

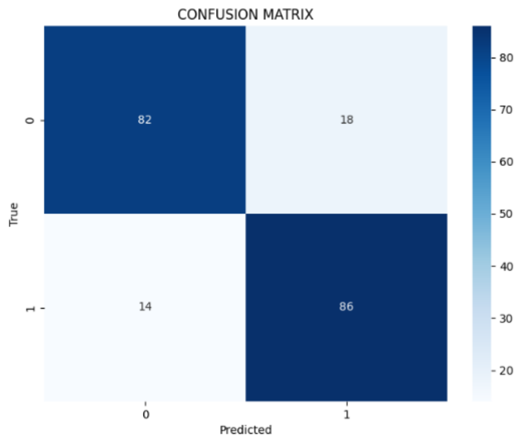
Accuracy: 84.00 %

Precision: 82.69%

Recall: 86.00%

F1_Score: 84.31%





ENHANCED_DATASET_1000 [OWL2Vec*]

Results obtained by training the model using the enhanced dataset with embeddings by OWL2Vec*.

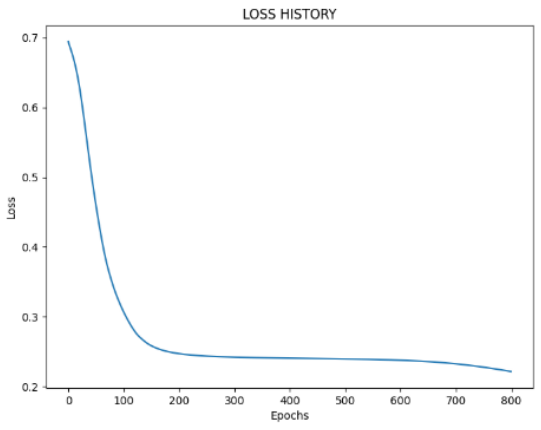
CLASSIFICATION METRICS

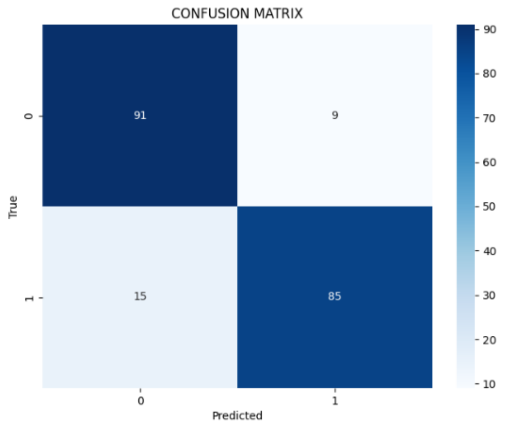
Accuracy: 88.00 %

Precision: 90.43%

Recall: 85.00%

F1_Score: 87.63%





ENHANCED_DATASET_1000 [Node2Vec]

Results obtained by training the model using the enhanced dataset with embeddings by Node2Vec.

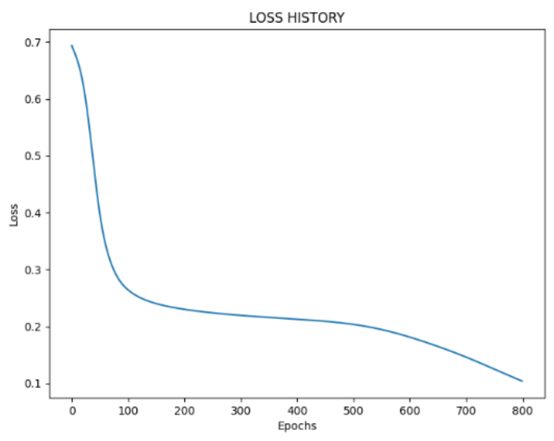
CLASSIFICATION METRICS

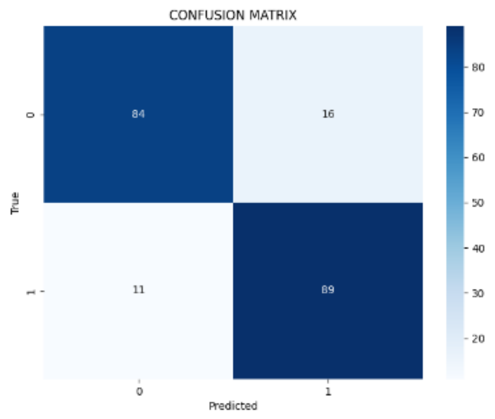
Accuracy: 86.50 %

Precision: 84.76%

Recall: 89.00%

F1_Score: 86.83%





As we can see, the training was performed with only the original reduced dataset (1000 samples) and already shows a fairly high percentage of accuracy, recall, and precision: this means that, in our case, the model learns very well with just the base dataset even with a small number of samples without the ontology integration.
The contribution of the OWL datasets, on the other hand, is minimal: for both models used, the performance did not improve at all, and in the case of Node2Vec, it actually worsened.
This means that it is necessary to at least double the size of the dataset in order to achieve good results.

6.2 – Results on 2000 samples dataset

DATASET_2000

Results obtained by training the model using main dataset only.

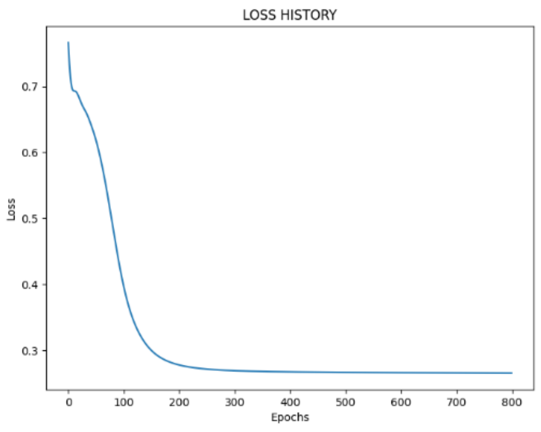
CLASSIFICATION METRICS

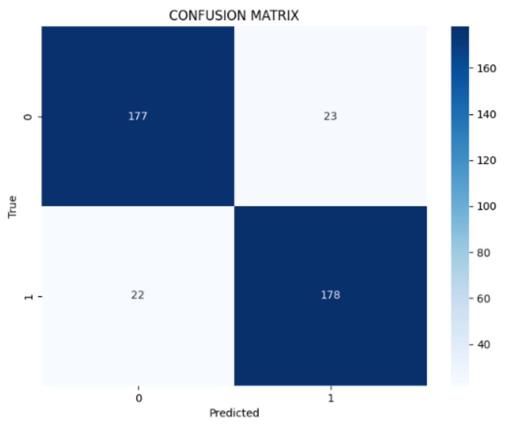
Accuracy: 88.75 %

Precision: 88.56%

Recall: 89.00%

F1_Score: 88.78%





OWL_DATASET_2000 [Node2Vec]

Results obtained by training the model using only the embeddings generated by Node2Vec.

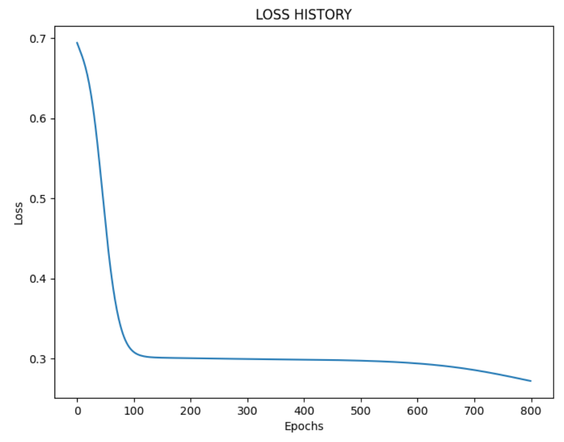
CLASSIFICATION METRICS

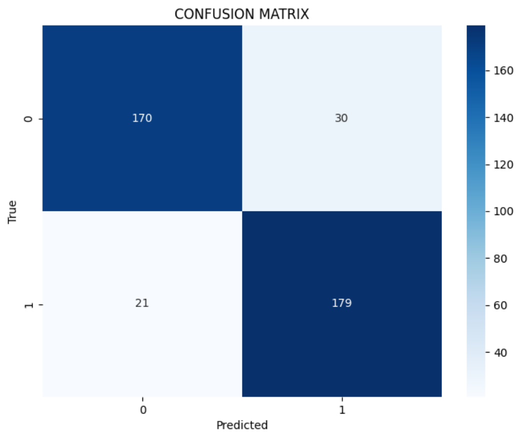
Accuracy: 87.25 %

Precision: 85.65%

Recall: 89.50%

F1_Score: 87.53%





ENHANCED_DATASET_2000 [Node2Vec]

Results obtained by training the model using the enhanced dataset with embeddings by Node2Vec.

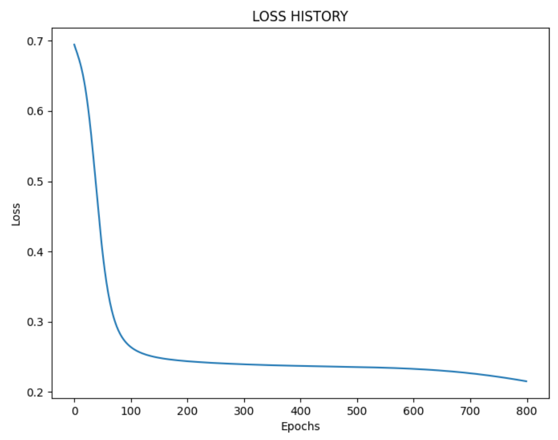
CLASSIFICATION METRICS

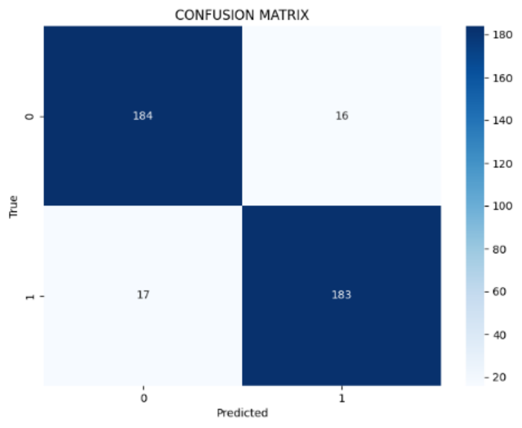
Accuracy: 91.75 %

Precision: 91.96%

Recall: 91.50%

F1_Score: 91.73%





By doubling the size of the dataset to 2000 samples, we indeed obtained excellent results.

First, as expected, the model trained only on the original dataset of 2000 samples showed a slight increase in accuracy compared to the one with 1000 samples (from 88% to 88.75%).

Furthermore, we documented a significant improvement in the model's performance when trained on the enhanced dataset using Node2Vec: reaching an accuracy of almost 92%, which represents a 3% increase on the test set; other metrics like precision, recall and F1-score also improve significantly.

This explains how our system, using Node2Vec, allows for a significant improvement in the performance of machine learning models, while being fast and scalable.

The results using OWL2Vec*, however, were not documented due to the excessive computational cost required by the model, which led to several errors related to insufficient memory on our machine.

CONCLUSION AND FURTHER DEVELOPMENTS

This project focused on developing a machine learning model for predicting whether a patient has diabetes.

Unlike the traditional approach, based only on dataset analysis (preprocessing and model prediction), we have decided to enhance the model knowledge by structuring an ontology which stores the key relationship between patient and his clinical data.

The model demonstrated a higher accuracy, as expected, when trained on the enhanced dataset; however, there are a lot of improvements which can be added.

First of all, we have used a simple ontology in respect to a real-case diabetes scenario in which an ontology can be built by a domain expert of this field in order unlock more complex knowledge pattern in the predictions. At the same time, also the formal logical rules used for reasoning on the knowledge base had to be defined by a domain expert to ensure a more personalized representation of the patient's health situation. Another point could be the adoption of a richer and more detailed dataset with more features and data samples.

Two different approaches were used for generating embeddings: OWL2Vec* and Node2Vec). A possible future development could involve adopting a more advanced architecture that balances the complexity of OWL2Vec* with the efficiency of Node2Vec, leveraging a model like BERT to also encode more complex semantic information.

While neural networks are robust and capable of identifying complex patterns, exploring other architectures such as Support Vector Machines (SVMs) or classification trees could offer insights into performance trade-off.

Regarding the neural network, other hyperparameters can be experimented with, for instance learning rates, number of epochs, number of hidden layers, activation functions, etc.

In conclusion, this project represents a meaningful step toward predictive analysis in machine learning tasks.

BIBLIOGRAPHY

[1] Aditya Grover, Jure Leskovec, node2vec: Scalable Feature Learning for Networks, paper

[2] Jiaoyan Chen, Pan Hu, Ernesto Jimenez-Ruiz, Ole Magnus Holter, Denvar Antonyrajah, Ian Horrocks, OWL2Vec∗: Embedding of OWL Ontologies, paper