EJSIT
European Journal of Science, Innovation and Technology
ISSN: 2786-4936
www.ejsit-journal.com
Volume 3 | Number 1 | 2023

ML Ontological Solution for Blood Donation System

Sana Rizwan [1], Muhammad Salman Ahmed [2], Akash ur Rehman [3], Muhammad Abu Hurairah [4]
[1]Assistant Professor, Department of Computer Science, COMSATS University Islamabad, Lahore Campus, Pakistan
[2,3,4]Student of BSSE, Department of Computer Science, COMSATS University Islamabad, Lahore Campus, Pakistan

Abstract

Blood donation is basically a process where an individual can voluntarily donate his/her blood for future transfusions. He/she donates his/her blood to the organizations that require that blood for treatment purposes like hospitals etc. People are majorly focused on helping others in case of emergency circumstances. Aim of this working is to provide a platform where they can donate their blood and can seek help in any case of emergency. To solve this real-world problem, adopt machine learning and ontology learning approaches to make the system learn by themselves. Automated blood donation system will use in emergency and make the blood donation an easy process by facilitating people with no hassle in search of blood pints from blood banks. Patients can achieve the blood by contacting the donor through the internet or a personal contact number. This facilitation process starts from blood screening, capturing the data of donors from datasets, adding new donors/donor agencies, applying machine learning algorithms to filter the required data of donor(s) with respect to screening results and diseases and will end on request completion and feedback statements. As the study on the semantic web is progressing, many domain ontologies are being built on the defined case. Ontologies are built for variety of reasons, provide a specification of a particular domain in an explicit format, ontologies serve as a common vocabulary for different people i.e., stakeholders, to communicate about a specific domain, providing a framework for capturing and sharing the knowledge etc. Ontologies help to integrate data from different sources by particular domain knowledge. The trained dataset will be imported and merged with self-created ontological structure, this schema will check by SPARQL queries and exported to web platform.

Keywords: RDF, Ontology, Classes, SPARQL, Object Properties, Data Properties, OWL Wiz, Logistic Regression, SVM, Decision Trees, KNN, Gradient Roasting, Random Forest Classifire, MLP Classifier

Introduction

To build a complete ontology, this study tried to merge the heterogeneous domain ontologies. Although there have been many studies about the method for building the ontology using ontology tools, there are still limited to build the perfect ontology.

Ontology merging can be a problem because it involves combining two or more ontologies that may have overlapped or conflicting concepts, properties, and axioms. When merging ontologies, it is important to ensure that the resulting ontology is coherent, consistent, and semantically meaningful. However, achieving this can be challenging due to the following reasons:

1. Inconsistencies: The merged ontology may contain inconsistencies in the form of contradictory statements or definitions. These inconsistencies can arise when the ontologies being merged have different or conflicting definitions for the same concept or property.
2. Conceptual heterogeneity: The ontologies being merged may have different

conceptualizations of the same domain, leading to conceptual heterogeneity. This can result in the presence of multiple, overlapping, or redundant concepts in the merged ontology.
3. Structural heterogeneity: The ontologies being merged may have different structural characteristics, such as different taxonomies or hierarchy structures. This can make it difficult to merge the ontologies and create a coherent structure.
4. Axiomatic heterogeneity: The ontologies being merged may have different axioms or logical constraints, which can lead to conflicts when attempting to merge them.
5. Scalability: As the number of ontologies to be merged increases, the problem of ontology merging becomes more complex, making it difficult to identify and resolve conflicts and inconsistencies.

The symmetric approach in ontologies is a way of representing relationships between concepts where the relationship is symmetric or bidirectional. In other words, if concept A is related to concept B through a symmetric relationship, then it is also true that concept B is related to concept A through the same relationship.

The symmetric approach is often contrasted with the asymmetric approach, where relationships between concepts are unidirectional. For example, the "parent of" relationship between two concepts is asymmetric because if concept A is the parent of concept B, it does not necessarily mean that concept B is the parent of concept A.

In ontologies, the symmetric approach is often used to represent relationships between concepts in a more flexible and natural way. It allows for more accurate representation of complex relationships between concepts and can be useful in a wide range of applications, such as in natural language processing, knowledge representation, and semantic web technologies.

In this paper, we have created ontologies and applied the concepts using the symmetric approach, we created an application to perform this research. Our research is not completely original as many of the studies have been made on this merging and mapping of the ontology, but our approach is original as we are manually merging and mapping the ontologies. Ontology merging, and as its name implies, merges the concepts that match semantically with each other into a single concept, and then produces a unique ontology from two source ontologies.

Methodology

Integration of blood banks dataset with data ontology after accessing best accuracy and precision of data via various machine learning and deep learning algorithms. Map and import one to one data and keep maintain RDF store, versatile queries will ensure the presence and absence of data via SPARQL patterns and show the resultant in react interface. Figure 1 shows the high-level structure of a system having its modules, components, and their relationships. Beside these SPARQL is a query language used to retrieve data from semantic web data sources. It is used to query data expressed in RDF (Resource Description Framework) format, which is a standard for representing and exchanging information on the web.

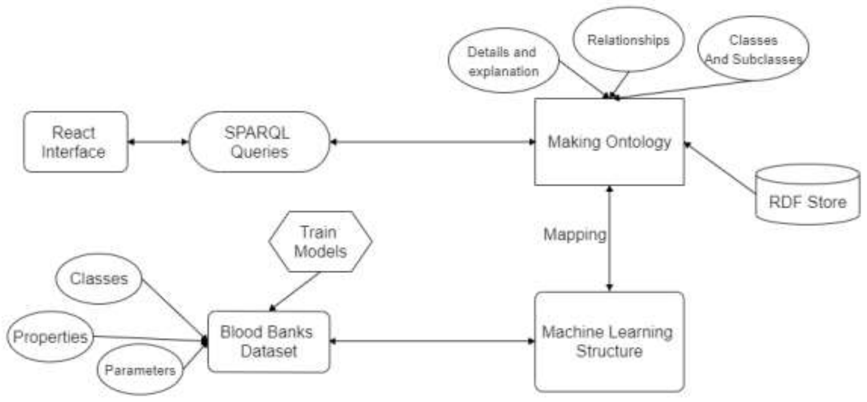


Figure 1: System Architecture

The methodology employ in the development of the envisioned system is Waterfall. In the waterfall method firstly define function and non-functional requirements. Make the design of system from these requirements. Then implement the design using different tools and techniques. The last testing and maintenance phase will be performed for deploying our web application of blood donation system.

Protégé

Protégé is a popular open-source ontology editor and development environment that provides a range of tools and features to help users create and manage ontologies. Here are some ways in which Protégé can help in the making of ontologies:

- User-friendly interface: Protégé provides a user-friendly interface that allows users to create and edit ontologies without requiring any programming knowledge.
- Ontology visualization: Protégé provides a graphical representation of ontologies, making it easy for users to visualize the relationships between classes and properties.
- Automated reasoning: Protégé supports automated reasoning, which helps users to identify inconsistencies and errors in their ontologies.
- Collaboration: Protégé provides tools for collaboration, allowing multiple users to work on the same ontology simultaneously.
- Customization: Protégé allows users to customize the ontology development environment to suit their specific needs, by providing a range of plugins and add-ons.

Benefits of Using Protégé:

There are several benefits of using Protégé for ontology development, some of which are listed below:

Firstly, Protégé provides a user-friendly interface that allows users to create and edit ontologies without requiring any programming knowledge. This makes it easy for domain experts and non-technical users to participate in the ontology development process, thus enabling collaboration and knowledge sharing across different domains.

Secondly, Protégé provides a range of ontology visualization tools that allow users to visualize the relationships between classes and properties. This helps users to better understand the structure of their ontologies and identify potential issues or areas for improvement.

Thirdly, Protégé supports automated reasoning, which helps users to identify inconsistencies and errors in their ontologies. This can save time and effort by detecting errors early in the development process before they become more difficult to correct.

Fourthly, Protégé provides tools for collaboration, allowing multiple users to work on the same ontology simultaneously. This enables teams to work more efficiently and effectively, by allowing them to share their knowledge and expertise in real-time.

Finally, Protégé allows users to customize the ontology development environment to suit their specific needs, by providing a range of plugins and add-ons. This allows users to tailor the environment to their specific domain or ontology, and to add new features as needed.

Creation of Ontology

An ontology for a blood donation system would consist of a set of concepts and relationships between them that are relevant to the domain of blood donation. Some key concepts in this domain might include blood types, blood banks, donors, recipients, and donations. Properties might include relationships between these entities, such as "donatesTo", "receivesFrom", "hasBloodType", and "belongsToBloodBank".

Using Protégé, we could create classes for each of these concepts, along with their corresponding properties. For example, we could create a class for "BloodType" with subclasses for each type of blood (A, B, AB, O) and properties such as "isCompatibleWith" to represent the compatibility between different blood types. Similarly, we could create classes for "Donor" and "Recipient" with properties such as "hasBloodType" and "donatesTo".

We could also create a class for "BloodBank" with properties such as "hasDonors" and "receivesFrom". Instances of this class could represent individual blood banks and their respective donors and recipients. We could also create a class for "Donation" with properties such as "hasDonor", "hasRecipient", and "hasDate", representing individual blood donations and their associated information.

By creating an ontology in this way, we could facilitate data integration and retrieval across multiple blood donation systems. For example, we could use reasoning to infer the compatibility between donors and recipients based on their blood types and other properties. We could also use the ontology to develop decision support systems that help blood banks and healthcare providers make informed decisions about blood transfusions.

Additionally, we could use Protégé's visualization tools to create graphical representations of the ontology, making it easier to understand and analyze. We could also use plugins such as OntoGraf to visualize the relationships between different entities in the ontology.

In summary, creating an ontology in Protégé for a blood donation system would provide a formal representation of the key concepts and relationships in the domain, facilitating data integration, retrieval, and analysis. It would also provide a foundation for the development of decision support systems and other applications that could improve the efficiency and effectiveness of blood donation systems.

Ontology Building

OWL ontology is built by following steps:

- Create the classes, subclasses and provide information about them. All the classes will be subclass of owl:Thing.
- Create data properties and provide domain that will be the class in the ontology. Enter range of the data property which will be the int, double or string. All the data properties will be the subclass of owl: topDataProperty.
- Create object properties and provide domain and range in this case both will be the class. Object properties are used to define the relation between two objects also called individuals in OWL ontology. All the object properties are subclass of owl: topObjectProperty. Provide the characteristic of object property and check the appropriate option.
- Create individuals in the individuals tab. Individuals are the instances of class which is set in the “type” description. Create the object assertion properties and data assertion properties.
- Set disjoint sets where two classes cannot contain same instance.

Classes in Ontology Building

In Protégé, classes are the most basic building blocks of an ontology. A class represents a category or type of entity in a domain. It defines a set of instances or individuals that share common characteristics and relationships.

To create a class in Protégé, you can follow these steps:

1. Open the Protégé application and create a new ontology.
2. In the "Classes" tab, click the "+" button to create a new class.
3. Enter the name of the class and any other information you want to include, such as a description or comments.
4. Add any necessary properties to the class, such as subclasses, super classes, or equivalent classes.

Here are some common types of classes you might encounter in an ontology:

1. Entity classes: These are classes that represent individual entities in the domain, such as "Person" or "Organization".
2. Attribute classes: These are classes that represent attributes or properties of entities, such as "Age" or "Gender".
3. Process classes: These are classes that represent processes or actions that occur in the domain, such as "Payment" or "Registration".
4. Relationship classes: These are classes that represent relationships between entities, such as "HasChild" or "HasParent".
5. Abstract classes: These are classes that represent abstract concepts or categories that are not directly instantiated, such as "Animal" or "Vehicle".
6. Utility classes: These are classes that are used to define utility concepts or functions that are needed to support the ontology, such as "Collection" or "List".

It's important to note that classes can also have subclasses and super classes. Subclasses are more specific types of a given class, while super classes are more general. For example, "Cat" might be a subclass of "Animal", while "Animal" is a superclass of "Cat".

Overall, classes are an essential component of an ontology in Protégé. By defining classes and their properties, we can create a structured and formal representation of the entities and relationships in a domain. This facilitates data integration, retrieval, and analysis, and can be used to develop applications that improve the efficiency and effectiveness of a system.

Relationships in Ontology

In Protégé, relationships in ontology are represented using properties. Properties are binary relationships between classes, where one class is the domain and the other class is the range. There are two main types of properties in Protégé: object properties and data properties.

Object properties- are relationships between two classes, where one class represents the subject of the relationship and the other class represents the object. For example, in a blood donation system ontology, we might define an object property called "hasDonor" that connects a class representing a blood bank to a class representing a donor. This relationship can be represented in Protégé as an arrow between the two classes.

Data properties- on the other hand, are relationships between a class and a data value. For example, we might define a data property called "donationDate" that connects a class representing a blood donation to a data value representing the date of the donation. This relationship can be represented in Protégé as a line between the class and the data value.

In addition to these basic relationship types, Protégé also supports more complex relationship types, such as transitive properties, inverse properties, functional properties, and inverse functional properties. These properties allow us to define more sophisticated relationships between classes and entities in an ontology.

For example, we might define a transitive property called "isParentOf" that connects a class representing a parent to a class representing a child. This property would allow us to infer that if A is a parent of B, and B is a parent of C, then A is also a parent of C. This relationship can be represented in Protégé as a series of arrows between the classes.

Similarly, we might define an inverse property called "hasDonation" that connects a class representing a donor to a class representing a blood donation. This property would allow us to infer that if A has donated B, then B was made by A. This relationship can be represented in Protégé as an arrow pointing in the opposite direction to the original arrow.

Overall, relationships in ontology are a critical component of an ontology in Protégé. By defining relationships between classes and entities, we can create a more comprehensive and accurate representation of a domain. This can be used to support a wide range of applications, from data integration and retrieval to reasoning and inference.

Ontology and its use as a Database

Ontologies can be used as a database for storing and retrieving information in a structured way. In fact, some researchers argue that ontologies are more than just a database, as they provide a formal, machine-readable representation of knowledge in a particular domain that can be used to support a wide range of applications.

Ontologies can be used as a database in several ways. Firstly, ontologies provide a structured way to organize data, as they define a set of classes, properties, and relationships that can be used to represent different types of entities and their attributes. For example, in a blood donation system ontology, we might define classes for donors, blood banks, and donations, and properties to represent attributes such as donor name, donation date, blood type, and so on. By structuring the data in this way, we can ensure that it is organized and easily searchable.

Secondly, ontologies provide a powerful querying mechanism that can be used to retrieve information from the database. In Protégé, for example, the SPARQL query language can be used to query ontologies and retrieve information based on certain criteria. This makes it easy to search the ontology for specific information, such as all donors with a particular blood type, or all donations made by a particular donor.

Thirdly, ontologies can be used to support reasoning and inference. By defining relationships between entities and classes, we can use automated reasoning tools to infer new information based on the existing data in the ontology. For example, if we have an ontology that defines relationships between diseases, symptoms, and treatments, we can use automated reasoning to infer the most likely treatment for a patient based on their symptoms and medical history.

Finally, ontologies can be used to support data integration and interoperability between different databases and applications. By defining a common vocabulary and set of relationships, ontologies can be used to bridge the gap between different databases and systems that use different terminologies and data structures. This makes it easier to combine data from different sources and perform complex analysis and decision-making tasks.

Overall, ontologies can be a powerful tool for storing and retrieving information in a structured and machine-readable way. By defining a set of classes, properties, and relationships, we can create a powerful database that can be used to support a wide range of applications, from data integration and retrieval to reasoning and inference.

Classification of Ontology

With reference to our Project, the classification of ontology can be done in several ways using the Protégé software. Here are some possible classifications:

1. *Blood types:* Blood types can be classified into four main groups: A, B, AB, and O. Each group can be further subcategorized into positive or negative based on the presence or absence of Rh factor. This classification can help in identifying the blood type of donors and recipients.

2. *Donor information:* Donor information can be classified into different categories such as name, age, gender, blood type, contact information, health history, and donation history. This classification can help in identifying suitable donors for specific recipients.
3. *Blood donation process:* The blood donation process can be classified into different stages such as donor screening, blood collection, testing, processing, and storage. Each stage can be further subcategorized based on the specific procedures and protocols involved. This classification can help in ensuring that the blood donation process is safe and efficient.
4. *Blood products:* Blood products can be classified into different categories such as whole blood, red blood cells, plasma, platelets, and cryoprecipitate. Each category can be further subcategorized based on the specific components and uses. This classification can help in identifying the appropriate blood product for a specific medical condition.

SPARQL Queries

SPARQL (SPARQL Protocol and RDF Query Language) is a standard query language used for querying data stored in RDF (Resource Description Framework) format. In the context of ontologies, SPARQL queries can be used to retrieve information from ontologies, which are also typically stored in RDF format. Here are some benefits of using SPARQL queries in ontologies:

Querying ontology data: SPARQL allows querying the data stored in an ontology, which makes it possible to retrieve specific information from the ontology. This can be helpful in a variety of applications such as data integration, data mining, and knowledge discovery.

Ontology maintenance: SPARQL queries can help in maintaining the ontology by checking for inconsistencies, redundancies, and other errors. For example, one can use SPARQL to identify missing or incorrect data in the ontology, which can then be corrected.

Integration with other tools: SPARQL queries can be integrated with other tools to create more powerful applications. For example, one can use SPARQL to query the ontology data and then use the results to generate reports or visualizations.

Interoperability: SPARQL is a standard query language, which means that it can be used across different ontologies and software applications. This helps in promoting interoperability and data sharing among different systems.

Flexibility: SPARQL queries are flexible and can be adapted to different use cases. For example, one can use SPARQL to retrieve information about specific classes or instances, or to query data across multiple ontologies.

Overall, SPARQL queries provide a powerful and flexible way to retrieve information from ontologies and can help in a variety of applications related to ontology development, maintenance, and integration.

Implementation

(Braga, J., Dias, J. L., & Regateiro, F. (2020). A Machine Learning Ontology. ResearchGate, p. 10, November 25, 2022.)

Object Properties

In ontology, an object property is used to describe the relationships between individuals or entities. Specifically, an object property expresses a relationship between two objects, where one object is the subject and the other is the object of the property.

In the context of a blood donation ontology, there could be several object properties that describe the relationships between different entities involved in the blood donation process as shown in Figure 2.

Donor-Recipient Relationship: This object property would relate a blood donor to the recipient of their blood donation. For example, a blood donor could donate blood to their spouse, and this relationship could be expressed using the object property "hasSpouseRecipient".

Donor-Blood Type Relationship: This object property would relate a blood donor to their blood type. For example, a donor with type O blood would have the object property "hasBloodTypeO".

Donor-Donation Center Relationship: This object property would relate a blood donor to the donation center where they made their donation. For example, a donor who made their donation at a Red Cross donation center would have the object property "donatedAtRedCrossCenter".

Recipient-Blood Type Relationship: This object property would relate a blood recipient to their blood type. For example, a recipient with type A blood would have the object property "hasBloodTypeA".

Recipient-Transfusion Relationship: This object property would relate a blood recipient to the transfusion of blood they received. For example, a recipient who received a transfusion of type B blood would have the object property "receivedTypeBTransfusion".

By using object properties in a blood donation ontology, we can describe the relationships between different entities involved in the blood donation process in a structured and formal way. This can help to improve the accuracy and consistency of data related to blood donation, which can ultimately lead to better outcomes for patients who need blood transfusions.

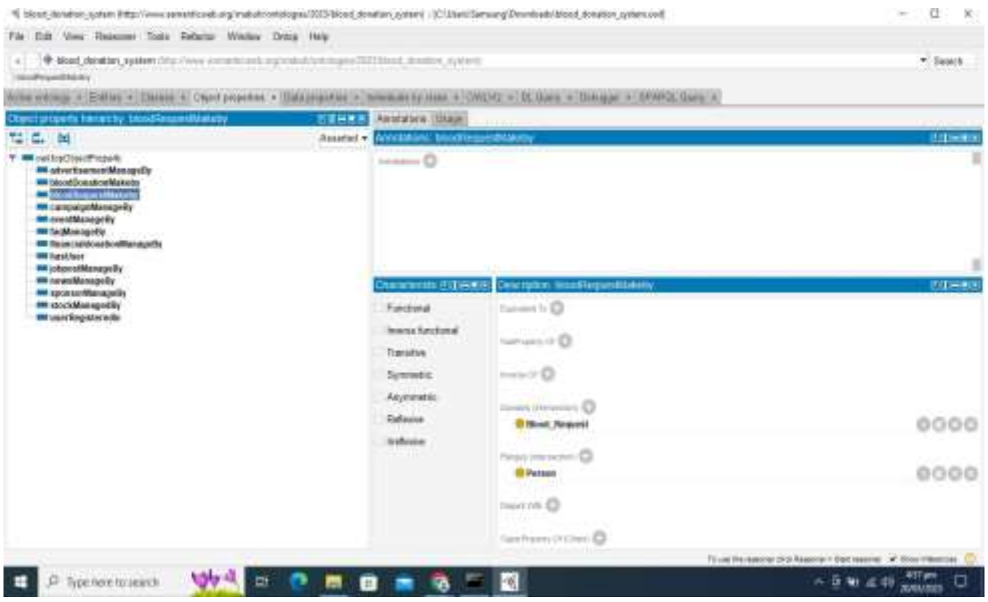


Figure 2: Object Properties

Data Properties

In an ontology, data properties are used to define attributes or characteristics of individuals or instances of a particular class. These properties are used to specify the values that can be assigned to the attributes, such as strings, numbers, and dates. In the context of a blood donation system ontology, data properties can be used to define attributes of blood donors, recipients, blood banks, and other relevant entities.

Here are some examples of data properties that are used in a blood donation system ontology as shown in Figure 3.

Donor Name: This property can be used to specify the name of the blood donor. It can be represented as a string value.

Donor Age: This property can be used to specify the age of the blood donor. It can be represented as an integer value.

Donor Blood Type: This property can be used to specify the blood type of the donor. It can be represented as a string value, such as "A+", "B-", or "O+".

Donor Last Donation Date: This property can be used to specify the date when the donor last donated blood. It can be represented as a date value.

Recipient Name: This property can be used to specify the name of the blood recipient. It can be represented as a string value.

Recipient Age: This property can be used to specify the age of the blood recipient. It can be represented as an integer value.

Recipient Blood Type: This property can be used to specify the blood type of the recipient. It can be represented as a string value.

Blood Bank Name: This property can be used to specify the name of the blood bank. It can be represented as a string value.

Blood Bank Address: This property can be used to specify the address of the blood bank. It can be represented as a string value.

Blood Bank Phone Number: This property can be used to specify the phone number of the blood bank. It can be represented as a string value.

These are just a few examples of data properties that can be used in a blood donation system ontology. Other properties can be added as needed to capture the relevant information about the entities in the system. By defining these properties, the ontology can help to standardize the representation of data across different systems and ensure that information is consistent and accurate.

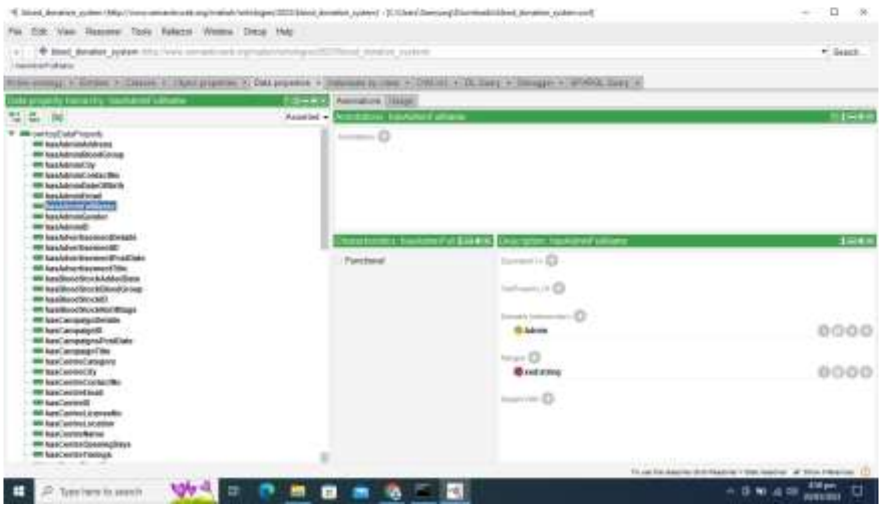


Figure 3: Data Properties

Ontology Graph

An ontology graph in Protégé is a graphical representation of the concepts, classes, and relationships between them within a domain. In the context of a blood donation system, an ontology graph can help to define and organize the knowledge related to blood donation, including the concepts, relationships, and properties that are relevant to the domain.

As shown in Figure 4, the ontology graph for a blood donation system could include classes such as "Blood Bank," "Donor," "Recipient," "Blood Type," "Blood Donation," "Blood Component," and "Blood Test." These classes would represent the main concepts within the domain and would be connected by relationships such as "Donates," "Receives," "Has Blood Type," "Is Component Of," and "Is Tested By."

For example, a "Donor" class would have properties such as "Name," "Age," "Blood Type," and "Contact Information," and it would be connected to a "Blood Donation" class through a "Donates" relationship. The "Blood Donation" class would have properties such as "Donation Date," "Donation Location," and "Blood Component," and it would be connected to a "Blood Bank" class through an "Is Stored In" relationship.

The ontology graph could also include axioms and rules that define the relationships between the classes, such as "A donor can donate blood multiple times," or "A recipient must have a compatible blood type with the donated blood."

Overall, the ontology graph in Protégé provides a visual representation of the knowledge within a domain and can help to organize and formalize the concepts and relationships involved in a blood donation system.

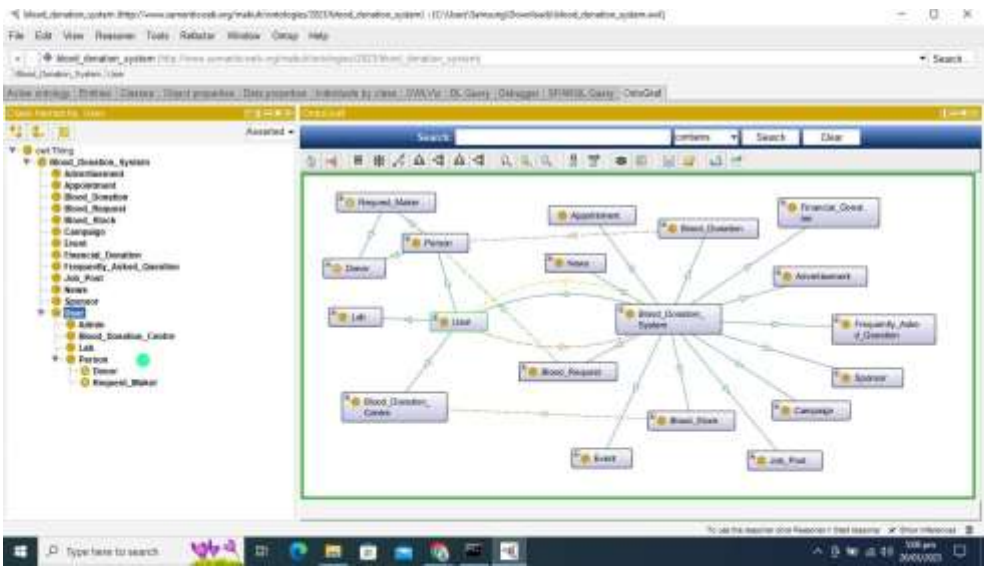


Figure 4: Ontology Graph

Classes in Ontology

In ontology, as shown in Figure 4 a class is a group or category of similar things or concepts that share common attributes, behaviors, or relationships. In the context of a blood donation system, classes can be used to represent the different entities or concepts that are relevant to the system.

Some of the classes that might be used in an ontology for a blood donation system could include:

Donor - a class representing individuals who donate blood. This class might have properties such as name, age, gender, blood type, and contact information.

Blood Bank - a class representing a facility where donated blood is collected, processed, and stored. This class might have properties such as name, location, contact information, and inventory.

Recipient - a class representing individuals who receive blood transfusions. This class might have properties such as name, age, gender, blood type, and medical history.

Blood Type - a class representing the different blood types that exist, such as A, B, AB, and O. This class might have properties such as compatibility rules for blood transfusions.

Blood Donation - a class representing a specific instance of a donor donating blood to a blood bank. This class might have properties such as the date, time, location, and quantity of the donation.

Blood Component - a class representing the different components that can be extracted from donated blood, such as red blood cells, plasma, and platelets. This class might have properties such as compatibility rules and shelf life.

Blood Test - a class representing the different tests that are performed on donated blood to ensure its safety and compatibility for transfusion. This class might have properties such as the type of test, the date it was performed, and the results.

These classes can be related to each other through properties or relationships to define the different types of associations or interactions that can exist between them. For example, a "Donor" class can be related to a "Blood Donation" class through a "donates" relationship, and a "Blood Bank" class can be related to a "Blood Donation" class through a "stores" relationship.

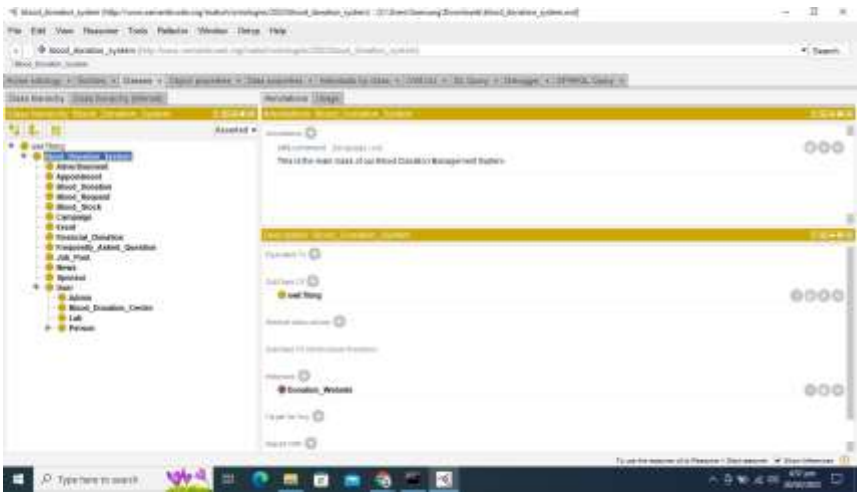


Figure 5: Classes in ontology

OWL Wiz (Hierarchy of Classes)

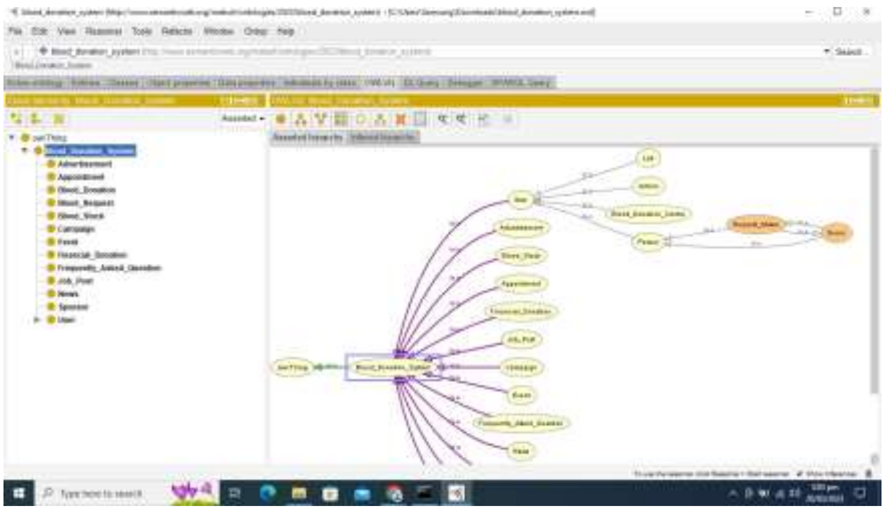


Figure 6: OWLWiz

As shown in Figure 6, hierarchy of classes represents a structure of classes and subclasses organized in a hierarchical manner. This hierarchy helps to organize and classify different entities based on their properties and relationships with other entities.

Working with Machine Learning Algorithms

An Ontology-based Approach for Making Machine Learning Systems Accountable (Esnaola-Gonzalez, 2021)

Blood donation systems rely heavily on timely and efficient donor management to ensure a stable supply of safe blood for transfusions. Machine learning and ontology can be used in blood donation systems to improve the efficiency of donor management and enhance the safety of the donated blood.

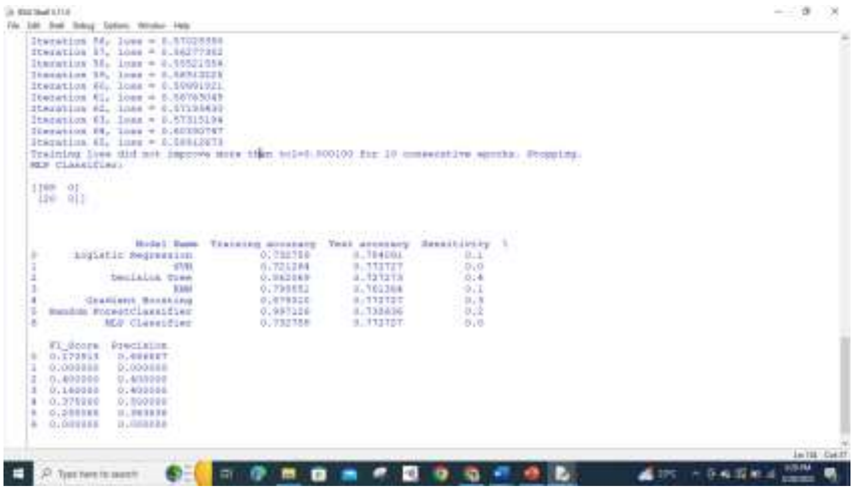
Machine learning algorithms like Gradient boosting, Random forest classifier and MLP classifier etc. can be used to analyze the data collected from donors and predict which donors are most likely to donate in the future. This information can be used to tailor donor recruitment campaigns to specific groups of potential donors. For example, if machine learning algorithms predict that younger donors are more likely to donate in the future, recruitment campaigns can be targeted towards this demographic to increase the likelihood of obtaining donations.

Machine learning can also be used to improve the safety of donated blood by analyzing the results of pre-donation health screenings. By analyzing the results of pre-donation health screenings, machine learning algorithms can identify potential risk factors for bloodborne diseases and other health issues. This information can be used to flag potential donors who may be at increased risk of transmitting bloodborne diseases or other health issues and prevent them from donating.

Ontologies can be used to standardize the terminology used to describe the different aspects of the blood donation system. By standardizing the terminology, it becomes easier to integrate data from different sources and improve data quality. For example, an ontology can be used to standardize the terminology used to describe donor demographics, blood types, and health screening results. This standardized terminology can then be used to develop a database that can be queried to provide insights into donor behavior, donation patterns, and other aspects of the blood donation system.

In summary, the implementation of machine learning and ontology in blood donation systems can help to improve the efficiency of donor management and enhance the safety of donated blood. By using machine learning algorithms to predict donor behavior and identify potential risk factors, blood donation systems can tailor their donor recruitment campaigns and prevent potentially unsafe blood from being donated. By using ontologies to standardize terminology, blood donation systems can improve data quality and gain insights into donor behavior and donation patterns.

Comparison Table



```
Iteration 56, loss = 0.57028588
Iteration 57, loss = 0.56277362
Iteration 58, loss = 0.55521558
Iteration 59, loss = 0.56843228
Iteration 60, loss = 0.50991921
Iteration 61, loss = 0.56765049
Iteration 62, loss = 0.57198430
Iteration 63, loss = 0.57315194
Iteration 64, loss = 0.60390787
Iteration 65, loss = 0.58812873
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
MLP Classifier:

[[69  0]
 [20  0]]


              Model Name  Training accuracy  Test accuracy  Sensitivity  \
0    Logistic Regression           0.732758       0.784091          0.1
1                    SVM           0.721284       0.772727          0.0
2          Decision Tree           0.982169       0.727273          0.4
3                    KNN           0.798852       0.761364          0.1
4      Gradient Boosting           0.879320       0.772727          0.3
5 Random ForestClassifier          0.997126       0.738636          0.2
6         MLP Classifier           0.732758       0.772727          0.0

   F1_Score  Precision
0  0.173913   0.666667
1  0.000000   0.000000
2  0.400000   0.400000
3  0.160000   0.400000
4  0.375000   0.500000
5  0.285714   0.363636
6  0.000000   0.000000
```

Figure 7: Comparison Table

As shown in the above Figure 7, we are provided with different models like Logistic Regression, SVM, Decision Trees, KNN, Gradient Roasting, Random Forest Classifier and MLP classifier. Details of the models are given below.

Logistic Regression is a linear model that is used for binary classification problems, where the goal is to predict whether an input belongs to one of two classes. In our model this algorithm gave us Training accuracy 0.732759 and Test accuracy of 0.784091 with a sensitivity of 0.1.

SVM is another popular algorithm for binary classification that is widely used in a variety of applications. SVM tries to find the best boundary between classes in the feature space. In our model this algorithm gave us Training accuracy 0.7221264 and Test accuracy of 0. 772727 with a sensitivity of 0.0.

Decision Trees are a type of supervised learning algorithm used for both classification and regression tasks. Decision trees build a tree-like model of decisions and their possible consequences. In our model this algorithm gave us Training accuracy 0.862069 and Test accuracy of 0. 727273 with a sensitivity of 0.4.

KNN is a non-parametric algorithm used for classification and regression tasks. The basic idea of KNN is to find the K-nearest neighbors of a new data point based on some distance metric, and then use the labels or values of these neighbors to predict the label or value of the new point. In our model this algorithm gave us Training accuracy 0.798851 and Test accuracy of 0. 761364 with a sensitivity of 0.1.

Gradient Boosting is an ensemble method that uses a combination of weak learners (usually decision trees) to create a strong learner. It iteratively trains a sequence of models, where each subsequent model tries to improve the error of the previous model by fitting the residual errors. In our model this algorithm gave us Training accuracy 0.879310 and Test accuracy of 0. 772727 with a sensitivity of 0.3.

Random Forest Classifier is another ensemble method that uses a combination of decision trees. Random Forest builds multiple decision trees using a subset of the available features and a subset of the available training data. The final output is then based on the majority vote of the individual decision trees. In our model this algorithm gave us Training accuracy 0.997126 and Test accuracy of 0. 738636 with a sensitivity of 0.2.

MLP Classifier is a type of neural network that uses multiple layers of neurons to learn complex relationships between the input and output data. It uses an activation function and back propagation algorithm to update the weights and biases of the network to minimize the error between the predicted and actual output. In our model this algorithm gave us Training accuracy 0.732759 and Test accuracy of 0. 772727 with a sensitivity of 0.0.

Details of the features

Figure 8: Details of results

Graphical representation of the features playing the role in the Model

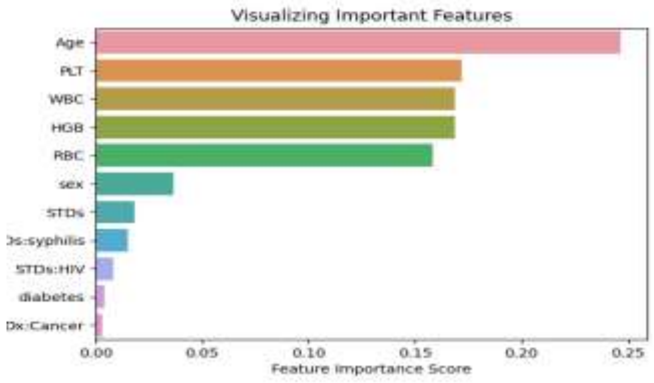
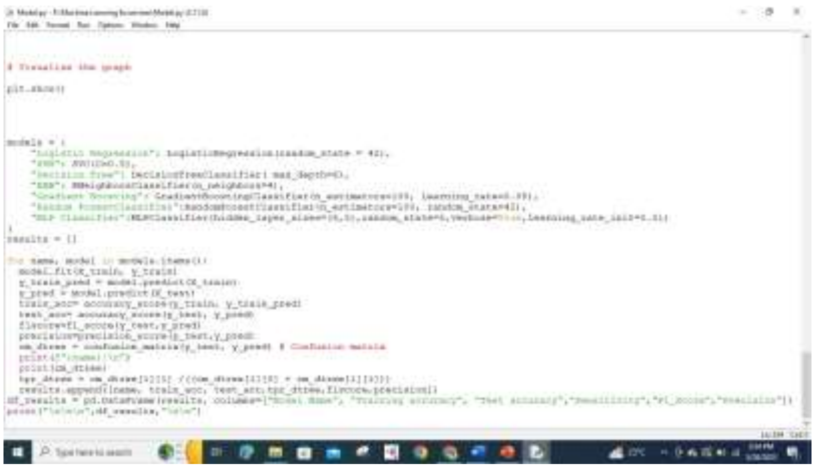


Figure 9: Graphical Representation

As shown in the Figure 8 and 9 the age feature is contributing the most in training of the model. Then we have PLT, WCB, HGB, RBC, Sex, STDs and so on. These features are contributing the most according to the order of their placement in the training of the model.

Algorithms



```
# Visualize the graph

plt.show()



models = {
    "Logistic Regression": LogisticRegression(random_state = 42),
    "SVM": SVC(C=0.5),
    "Decision Tree": DecisionTreeClassifier( max_depth=[illegible]),
    "KNN": KNeighborsClassifier(n_neighbors=4),
    "Gradient Boosting": GradientBoostingClassifier(n_estimators=100, learning_rate=0.05),
    "Random ForestClassifier":RandomForestClassifier(n_estimators=100, random_state=42),
    "MLP Classifier":MLPClassifier(hidden_layer_sizes=(5,5),random_state=1,verbose=True,learning_rate_init=0.01)
}
results = []

for name, model in models.items():
  model.fit(X_train, y_train)
  y_train_pred = model.predict(X_train)
  y_pred = model.predict(X_test)
  train_acc= accuracy_score(y_train, y_train_pred)
  test_acc= accuracy_score(y_test, y_pred)
  f1score=f1_score(y_test,y_pred)
  precision=precision_score(y_test,y_pred)
  cm_dtree = confusion_matrix(y_test, y_pred) # Confusion matrix
  print(f"{name}:\n")
  print(cm_dtree)
  tpr_dtree = cm_dtree[1][1] /((cm_dtree[1][0] + cm_dtree[1][1]))
  results.append([name, train_acc, test_acc,tpr_dtree,f1score,precision])
df_results = pd.DataFrame(results, columns=["Model Name", "Training accuracy", "Test accuracy","Sensitivity","F1_score","Precision"])
print("\n\n",df_results,"\n\n")
```

Figure 10: Machine Learning Algorithm

As shown in the Figure 10 we are provided with different algorithms used in Machine learning. The details of all the models are given as follows:

Logistic Regression: This is a linear model that is used for binary classification problems, where the goal is to predict whether an input belongs to one of two classes. It models the probability of the output using a logistic function and can be trained using gradient descent. The parameter 'random_state' is used to set the random seed for reproducibility.

SVM: This is a popular algorithm for binary classification that is widely used in a variety of applications. SVM tries to find the best boundary between classes in the feature space. The parameter 'C' is used to set the regularization parameter of the SVM, which controls the trade-off between maximizing the margin and minimizing the classification error.

Decision Tree: This is a type of supervised learning algorithm used for both classification and regression tasks. Decision trees build a tree-like model of decisions and their possible consequences. The parameter 'max_depth' is used to set the maximum depth of the decision tree, which controls the complexity of the model and prevents overfitting.

KNN: This is a non-parametric algorithm used for classification and regression tasks. The basic idea of KNN is to find the K-nearest neighbors of a new data point based on some distance metric, and then use the labels or values of these neighbors to predict the label or value of the new point. The parameter 'n_neighbors' is used to set the number of neighbors to consider when making a prediction.

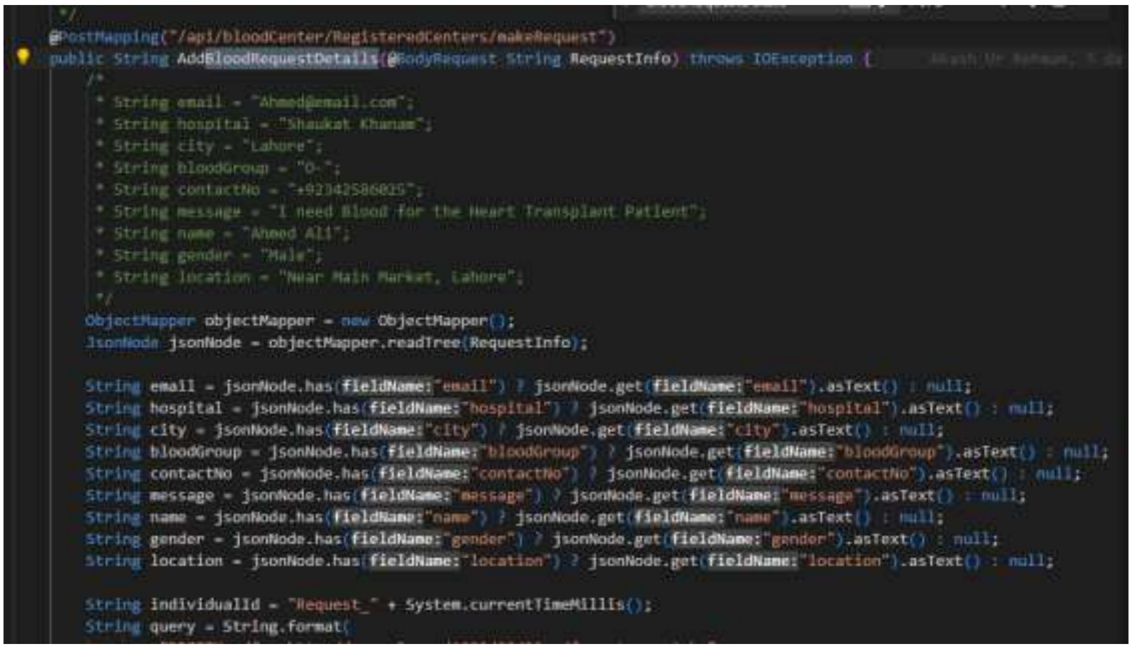
Gradient Boosting: This is an ensemble method that uses a combination of weak learners (usually decision trees) to create a strong learner. It iteratively trains a sequence of models, where each subsequent model tries to improve the error of the previous model by fitting the residual errors. The parameter 'n_estimators' is used to set the number of weak learners to use, and the parameter 'learning_rate' controls the contribution of each weak learner.

Random Forest Classifier: This is another ensemble method that uses a combination of decision trees. Random Forest builds multiple decision trees using a subset of the available features and a subset of the available training data. The final output is then based on the majority vote of the individual decision trees. The parameter 'n_estimators' is used to set the number of decision trees to build, and 'random_state' sets the random seed for reproducibility.

MLP Classifier: This is a type of neural network that uses multiple layers of neurons to learn complex relationships between the input and output data. It uses an activation function and backpropagation algorithm to update the weights and biases of the network to minimize the error between the predicted and actual output. The parameter 'hidden_layer_sizes' is used to set the number of neurons in each hidden layer, 'random_state' sets the random seed for reproducibility, 'verbose' enables verbose output during training, and 'learning_rate_init' sets the initial learning rate used during optimization.

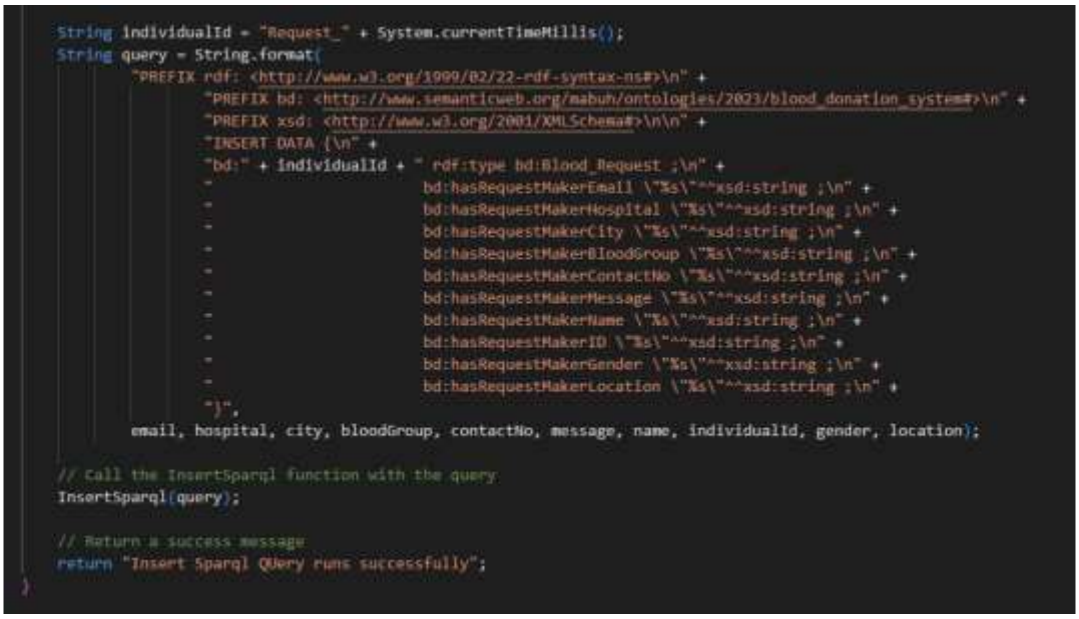
SPARQL Queries

Post Query:



```
    */
    @PostMapping("/api/bloodCenter/RegisteredCenters/makeRequest")
    public String AddBloodRequestDetails(@RequestBody String RequestInfo) throws IOException {
        /*
         * String email = "Ahmed@email.com";
         * String hospital = "Shaukat Khanam";
         * String city = "Lahore";
         * String bloodGroup = "O-";
         * String contactNo = "+923425860025";
         * String message = "I need Blood for the Heart Transplant Patient";
         * String name = "Ahmed Ali";
         * String gender = "Male";
         * String location = "Near Main Market, Lahore";
         */
        ObjectMapper objectMapper = new ObjectMapper();
        JsonNode jsonNode = objectMapper.readTree(RequestInfo);

        String email = jsonNode.has(fieldName:"email") ? jsonNode.get(fieldName:"email").asText() : null;
        String hospital = jsonNode.has(fieldName:"hospital") ? jsonNode.get(fieldName:"hospital").asText() : null;
        String city = jsonNode.has(fieldName:"city") ? jsonNode.get(fieldName:"city").asText() : null;
        String bloodGroup = jsonNode.has(fieldName:"bloodGroup") ? jsonNode.get(fieldName:"bloodGroup").asText() : null;
        String contactNo = jsonNode.has(fieldName:"contactNo") ? jsonNode.get(fieldName:"contactNo").asText() : null;
        String message = jsonNode.has(fieldName:"message") ? jsonNode.get(fieldName:"message").asText() : null;
        String name = jsonNode.has(fieldName:"name") ? jsonNode.get(fieldName:"name").asText() : null;
        String gender = jsonNode.has(fieldName:"gender") ? jsonNode.get(fieldName:"gender").asText() : null;
        String location = jsonNode.has(fieldName:"location") ? jsonNode.get(fieldName:"location").asText() : null;

        String individualId = "Request_" + System.currentTimeMillis();
        String query = String.format(
```

Figure 11: SPARQL Queries

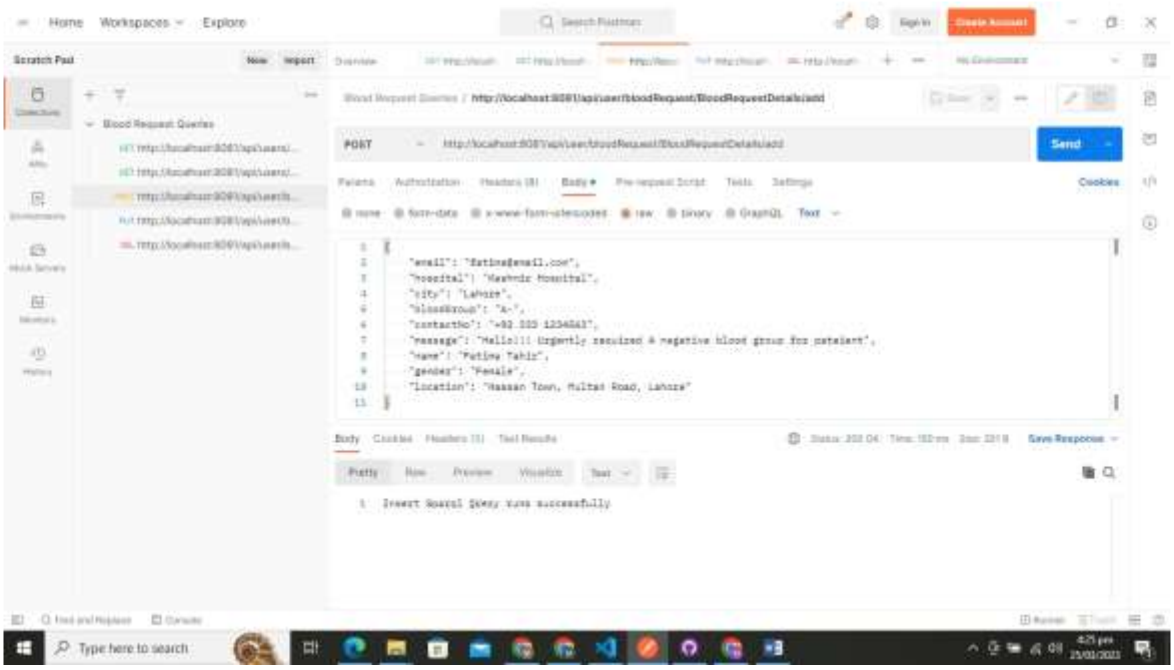


```
    String individualId = "Request_" + System.currentTimeMillis();
    String query = String.format(
            "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n" +
                    "PREFIX bd: <http://www.semanticweb.org/mabuh/ontologies/2023/blood_donation_system#>\n" +
                    "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>\n\n" +
                    "INSERT DATA {\n" +
                    "bd:" + individualId + " rdf:type bd:Blood_Request ;\n" +
                    "                         bd:hasRequestMakerEmail \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerHospital \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerCity \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerBloodGroup \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerContactNo \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerMessage \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerName \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerID \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerGender \"%s\"^^xsd:string ;\n" +
                    "                         bd:hasRequestMakerLocation \"%s\"^^xsd:string ;\n" +
                    "}",
            email, hospital, city, bloodGroup, contactNo, message, name, individualId, gender, location);

    // Call the InsertSparql function with the query
    InsertSparql(query);

    // Return a success message
    return "Insert Sparql QUery runs successfully";
}
```

Figure 12: SPARQL queries

As shown in Figures 11 & 12, we have written sparl queries to get data from the user and store it into the API. This is a method in a Spring Boot controller that handles a POST request to add details of a blood request. The request body is expected to be in JSON format, which is deserialized using the ObjectMapper class from the Jackson library. The method extracts the relevant data from the JSON object and creates an RDF triple with the data using the SPARQL INSERT query. The query is executed using the InsertSparql method, which is not shown in this code snippet. Finally, the method returns a success message. The results of the query are shown in Figure 13 given below.

Results



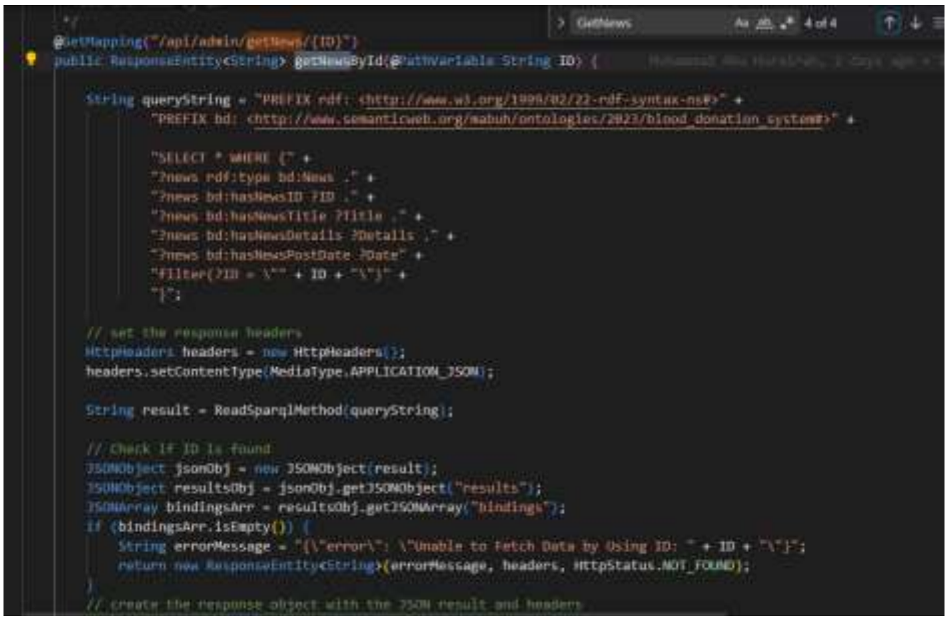
POST http://localhost:8081/api/user/bloodRequest/BloodRequestDetails/add

```
{
    "email": "fatima@email.com",
    "hospital": "Mashrik Hospital",
    "city": "Lahore",
    "bloodGroup": "A-",
    "contactNo": "+92 333 1234563",
    "message": "Hello!!! Urgently required A negative blood group for patient",
    "name": "Fatima Tahir",
    "gender": "Female",
    "location": "Hassan Town, Multan Road, Lahore"
}
```

```
Insert Sparql QUery runs successfully
```

Figure 13: INSERT Query Outcomes

View Query:



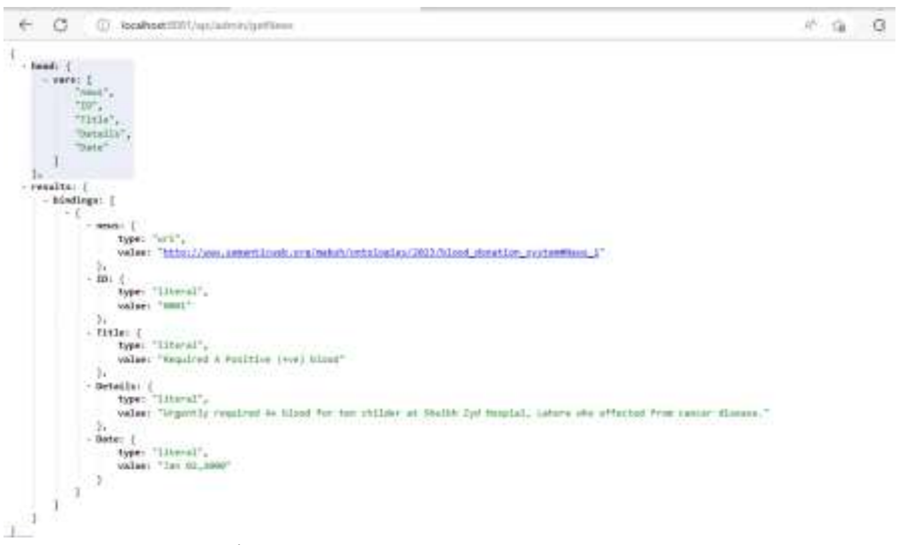
```
*/
@GetMapping("/api/admin/getNews/{ID}")
public ResponseEntity<String> getNewsById(@PathVariable String ID) {

    String queryString = "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>" +
            "PREFIX bd: <http://www.semanticweb.org/mabuh/ontologies/2023/blood_donation_system#>" +

            "SELECT * WHERE {" +
            "?news rdf:type bd:News ." +
            "?news bd:hasNewsID ?ID ." +
            "?news bd:hasNewsTitle ?Title ." +
            "?news bd:hasNewsDetails ?Details ." +
            "?news bd:hasNewsPostDate ?Date" +
            "Filter(?ID = \"" + ID + "\")" +
            "}";

    // set the response headers
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);

    String result = ReadSparqlMethod(queryString);

    // Check if ID is found
    JSONObject jsonObj = new JSONObject(result);
    JSONObject resultsObj = jsonObj.getJSONObject("results");
    JSONArray bindingsArr = resultsObj.getJSONArray("bindings");
    if (bindingsArr.isEmpty()) {
        String errorMessage = "{\"error\": \"Unable to Fetch Data by Using ID: " + ID + "\"}";
        return new ResponseEntity<String>(errorMessage, headers, HttpStatus.NOT_FOUND);
    }
    // create the response object with the JSON result and headers
```

Figure 14: View Query

In the above Figure 14, we are writing sparql queries to get the data from the database, This is a GET API endpoint that retrieves a news article by its ID. The endpoint takes the news ID as a path variable and constructs a SPARQL query to retrieve the news article's details from the ontology.

The SPARQL query selects all the properties of a News individual that has the provided news ID. It then calls a method ReadSparqlMethod(queryString) that executes the SPARQL query and returns the result as a JSON string. The method checks if the result contains any bindings, and if not, returns a JSON error message with a 404 HTTP status code.

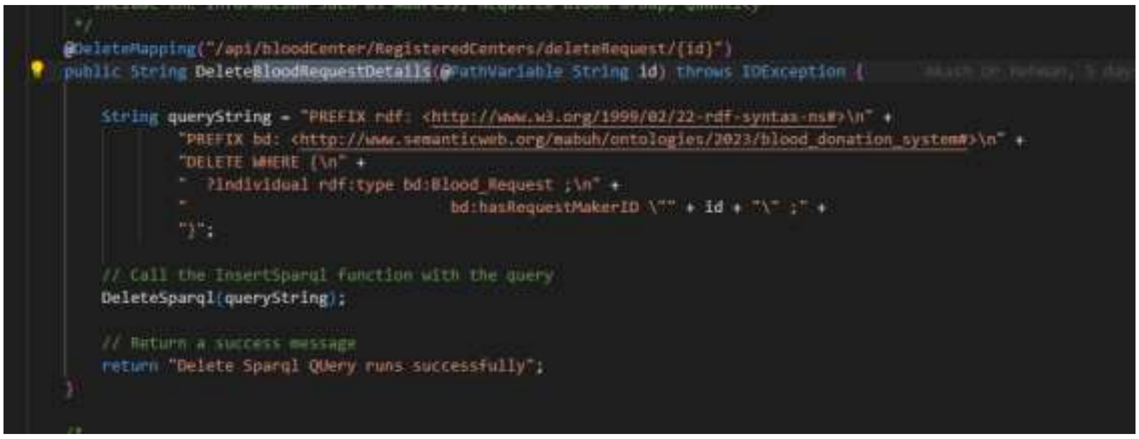
The method constructs and returns a ResponseEntity object with the result JSON string and HTTP headers. If the news article is found, it returns the result JSON with a 200 HTTP status code. If the news article is not found, it returns an error JSON with a 404 HTTP status code.this query is used to fetch the data from the database. The results of the query are as shown in the Figure 15.



```
localhost:8081/api/admin/getNews

{
 - head: {
   - vars: [
       "news",
       "ID",
       "Title",
       "Details",
       "Date"
     ]
   },
 - results: {
   - bindings: [
     - {
       - news: {
           type: "uri",
           value: "http://www.semanticweb.org/mabuh/ontologies/2023/blood_donation_system#News_1"
         },
       - ID: {
           type: "literal",
           value: "N001"
         },
       - Title: {
           type: "literal",
           value: "Required A Positive (+ve) blood"
         },
       - Details: {
           type: "literal",
           value: "Urgently required A+ blood for ten children at Shaikh Zyd Hospial, Lahore who effected from cancer disease."
         },
       - Date: {
           type: "literal",
           value: "Jan 02,2000"
         }
       }
     ]
   }
}
```

Figure 15: VIEW Query outcomes

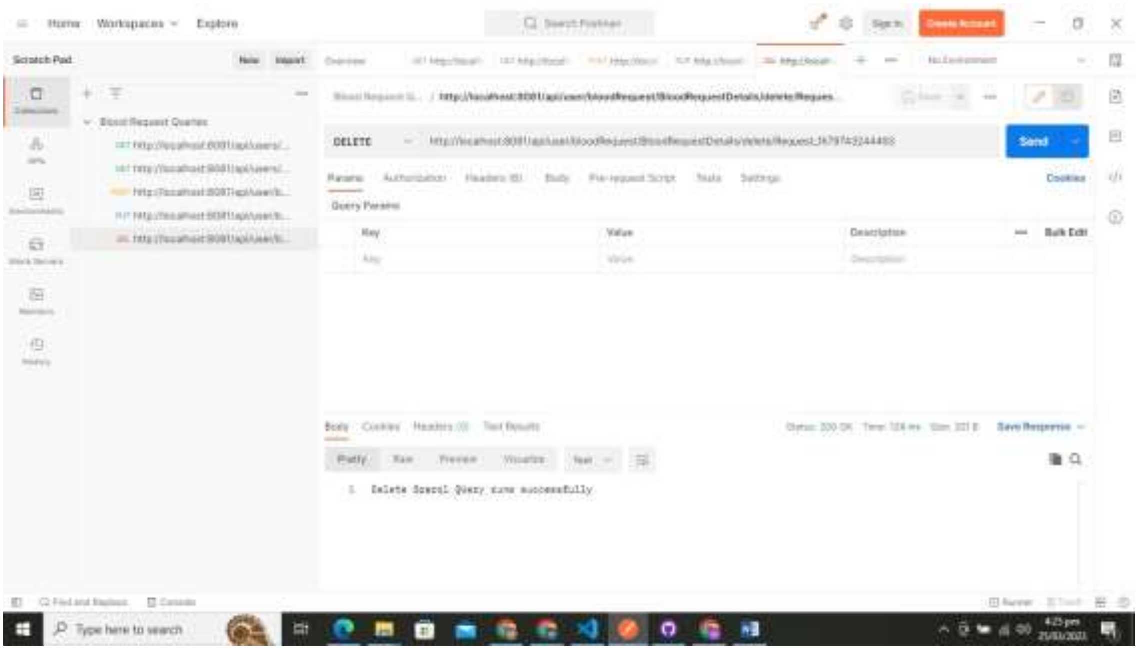
Delete Query:



```
    */
    @DeleteMapping("/api/bloodCenter/RegisteredCenters/deleteRequest/{id}")
    public String DeleteBloodRequestDetails(@PathVariable String id) throws IOException {

        String queryString = "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n" +
                "PREFIX bd: <http://www.semanticweb.org/mabuh/ontologies/2023/blood_donation_system#>\n" +
                "DELETE WHERE {\n" +
                "  ?individual rdf:type bd:Blood_Request ;\n" +
                "                              bd:hasRequestMakerID \"" + id + "\" ;" +
                "}";

        // Call the InsertSparql function with the query
        DeleteSparql(queryString);

        // Return a success message
        return "Delete Sparql QUery runs successfully";
    }
```

Figure 16: DELETE Query

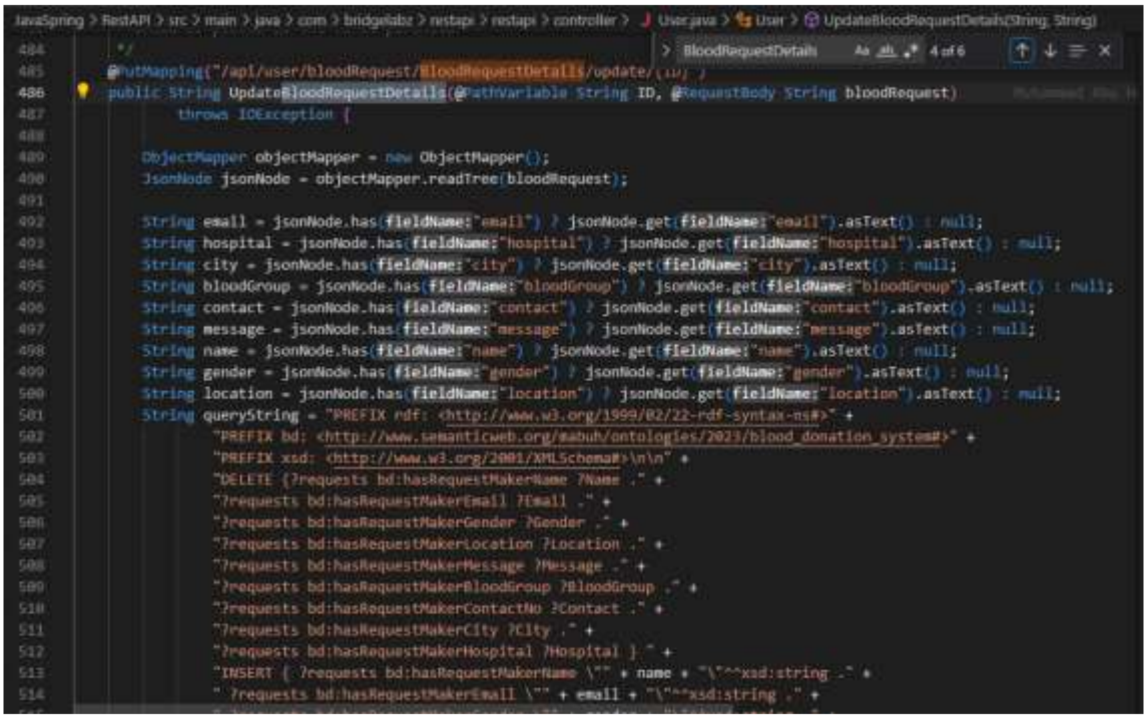
As shown in Figure 16, this is a query to delete the Blood Request from the database, This is a Spring Boot controller method for handling HTTP DELETE requests to delete blood donation requests from a database using SPARQL. The method takes a single parameter, "id," which represents the unique identifier of the blood donation request to be deleted. The SPARQL DELETE query uses this identifier to match the corresponding blood donation request in the database and delete it. The method returns a success message after executing the SPARQL query. Results of this query are shown in the Figure 17 given below.



```
DELETE    http://localhost:8081/api/user/bloodRequest/BloodRequestDetails/deleteRequest_1679743244488

Status: 200 OK

1  Delete Sparql QUery runs successfully
```

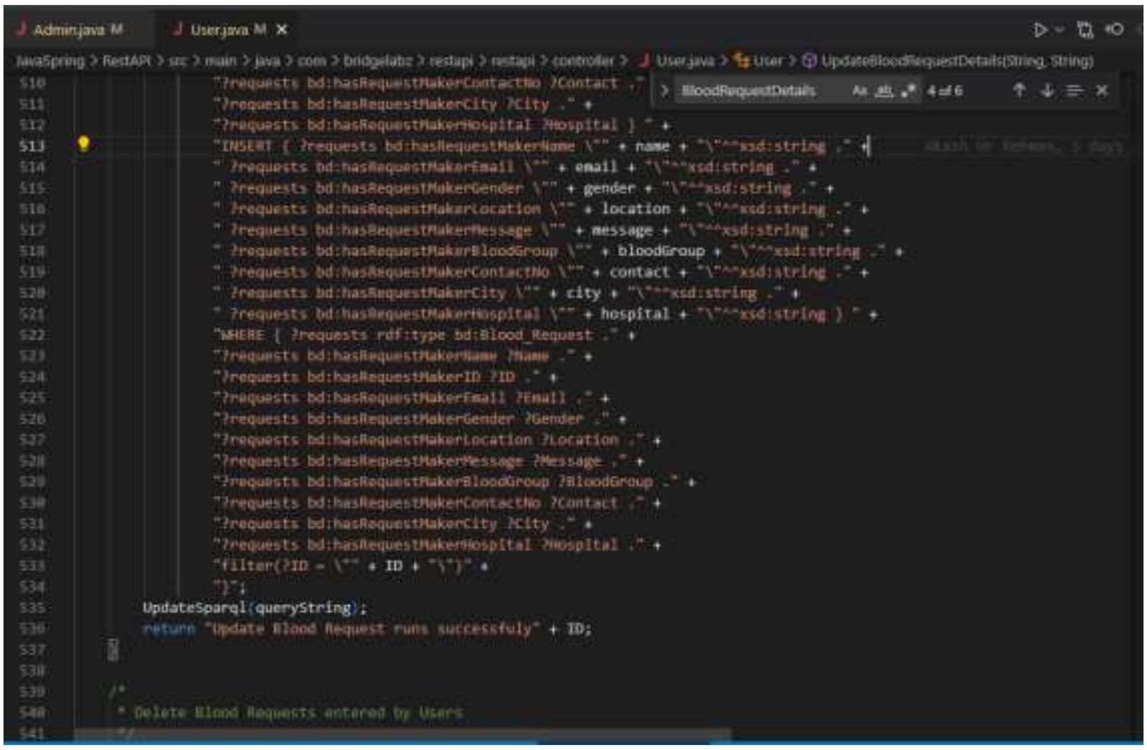
Figure 17: DELETE Query outcomes

Put Query:



```
JavaSpring > RestAPI > src > main > java > com > bridgelabz > restapi > restapi > controller > User.java > User > UpdateBloodRequestDetails(String, String)

    */
    @PutMapping("/api/user/bloodRequest/BloodRequestDetails/update/{ID}")
    public String UpdateBloodRequestDetails(@PathVariable String ID, @RequestBody String bloodRequest)
            throws IOException {

        ObjectMapper objectMapper = new ObjectMapper();
        JsonNode jsonNode = objectMapper.readTree(bloodRequest);

        String email = jsonNode.has(fieldName:"email") ? jsonNode.get(fieldName:"email").asText() : null;
        String hospital = jsonNode.has(fieldName:"hospital") ? jsonNode.get(fieldName:"hospital").asText() : null;
        String city = jsonNode.has(fieldName:"city") ? jsonNode.get(fieldName:"city").asText() : null;
        String bloodGroup = jsonNode.has(fieldName:"bloodGroup") ? jsonNode.get(fieldName:"bloodGroup").asText() : null;
        String contact = jsonNode.has(fieldName:"contact") ? jsonNode.get(fieldName:"contact").asText() : null;
        String message = jsonNode.has(fieldName:"message") ? jsonNode.get(fieldName:"message").asText() : null;
        String name = jsonNode.has(fieldName:"name") ? jsonNode.get(fieldName:"name").asText() : null;
        String gender = jsonNode.has(fieldName:"gender") ? jsonNode.get(fieldName:"gender").asText() : null;
        String location = jsonNode.has(fieldName:"location") ? jsonNode.get(fieldName:"location").asText() : null;
        String queryString = "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>" +
                "PREFIX bd: <http://www.semanticweb.org/mabuh/ontologies/2023/blood_donation_system#>" +
                "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>\n\n" +
                "DELETE {?requests bd:hasRequestMakerName ?Name ." +
                "?requests bd:hasRequestMakerEmail ?Email ." +
                "?requests bd:hasRequestMakerGender ?Gender ." +
                "?requests bd:hasRequestMakerLocation ?Location ." +
                "?requests bd:hasRequestMakerMessage ?Message ." +
                "?requests bd:hasRequestMakerBloodGroup ?BloodGroup ." +
                "?requests bd:hasRequestMakerContactNo ?Contact ." +
                "?requests bd:hasRequestMakerCity ?City ." +
                "?requests bd:hasRequestMakerHospital ?Hospital } " +
                "INSERT { ?requests bd:hasRequestMakerName \"" + name + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerEmail \"" + email + "\"^^xsd:string ." +
```

Figure 18: PUT Query

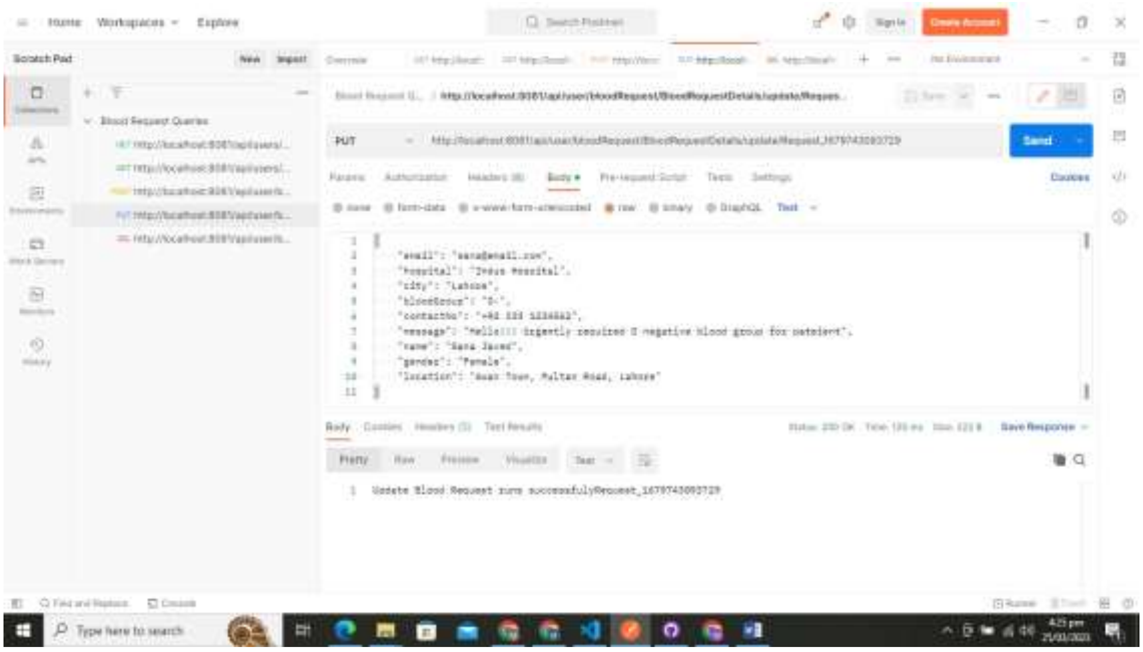


```
Admin.java M    User.java M ×
JavaSpring > RestAPI > src > main > java > com > bridgelabz > restapi > restapi > controller > User.java > User > UpdateBloodRequestDetails(String, String)

                "?requests bd:hasRequestMakerContactNo ?Contact ." +
                "?requests bd:hasRequestMakerCity ?City ." +
                "?requests bd:hasRequestMakerHospital ?Hospital } " +
                "INSERT { ?requests bd:hasRequestMakerName \"" + name + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerEmail \"" + email + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerGender \"" + gender + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerLocation \"" + location + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerMessage \"" + message + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerBloodGroup \"" + bloodGroup + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerContactNo \"" + contact + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerCity \"" + city + "\"^^xsd:string ." +
                " ?requests bd:hasRequestMakerHospital \"" + hospital + "\"^^xsd:string } " +
                "WHERE { ?requests rdf:type bd:Blood_Request ." +
                "?requests bd:hasRequestMakerName ?Name ." +
                "?requests bd:hasRequestMakerID ?ID ." +
                "?requests bd:hasRequestMakerEmail ?Email ." +
                "?requests bd:hasRequestMakerGender ?Gender ." +
                "?requests bd:hasRequestMakerLocation ?Location ." +
                "?requests bd:hasRequestMakerMessage ?Message ." +
                "?requests bd:hasRequestMakerBloodGroup ?BloodGroup ." +
                "?requests bd:hasRequestMakerContactNo ?Contact ." +
                "?requests bd:hasRequestMakerCity ?City ." +
                "?requests bd:hasRequestMakerHospital ?Hospital ." +
                "filter(?ID = \"" + ID + "\")" +
                "}";
        UpdateSparql(queryString);
        return "Update Blood Request runs successfuly" + ID;
    }


    /*
     * Delete Blood Requests entered by Users
     */
```

Figure 19: PUT Query

In the above Figure 18 and 19, we have the query to edit the record that is already existing in the database. This code defines a REST endpoint that handles an HTTP PUT request to update details of a blood donation request in a blood donation system. The endpoint accepts a request body in JSON format containing updated values for the blood request details. The code uses the Jackson ObjectMapper library to parse the JSON request body into a JsonNode object.

The code then extracts the updated values from the JsonNode object and constructs a SPARQL query to update the corresponding blood donation request in the RDF graph. The SPARQL query deletes the existing values of the blood request details and inserts the new values. The WHERE clause of the SPARQL query filters the blood donation request by its ID and retrieves its existing values. The UpdateSparql() function executes the constructed SPARQL query to update the blood donation request. Finally, the function returns a success message indicating that the update was successful. The results of this query are shown in the Figure 20 given below.



```
PUT   http://localhost:8081/api/user/bloodRequest/BloodRequestDetails/updateRequest_1679743093729
```

```
{
    "email": "sana@email.com",
    "hospital": "Indus Hospital",
    "city": "Lahore",
    "bloodGroup": "O-",
    "contactNo": "+92 333 1234562",
    "message": "Hello!!! Urgently required O negative blood group for patient",
    "name": "Sana Javed",
    "gender": "Female",
    "location": "Awan Town, Multan Road, Lahore"
}
```

```
Update Blood Request runs successfulyRequest_1679743093729
```

Figure 20: PUT Query outcomes

Conclusion

Problem statement is about blood donations. To work on it employ machine learning and ontology learning methodologies because they enable the system to learn on its own. Blood donation is essentially a process where a person can willingly donate his or her blood for subsequent transfusions. The procedure of giving blood is extremely important, and it may be made simple by employing machine learning. In this project, we suggest an automated blood donation system to facilitate blood donation and relieve emergency situations The main goal of creating this system or web application was to allow users to look for blood in blood banks and obtain blood in any situation. By contacting the donor online or at a personal phone number, patients can obtain the blood.

Acknowledgment

The Authors wish to thank Department of Computer Science COMSATS University Islamabad, Lahore Campus, Pakistan for support and help in the field of Semantic web structuring integrated ML Algorithms.

References

Braga, J., Dias, J. L., & Regateiro, F. (2020). A Machine Learning Ontology. ResearchGate, p. 10, November 25, 2022.

Esnaola-Gonzalez, I. (2021). An Ontology-based approach for making Machine learning systems Accountable. IOS Press, p. 17.

Khan, A. R., & Qureshi, M. S. (2009). Web-based information system for blood donation. *International Journal of Digital Content Technology and its Applications*, *3*(2), 137-142.

Mostafa, M. M. (2009). Profiling blood donors in Egypt: A neural network analysis. *Expert Systems with Applications*, *36*(3), 5031-5038.

Rutkin, A. (2015). Big data aims to boost New York blood donation. *New Scientist*, *225*(3006), 19.

Sinha, S., Seth, T., Colah, R. B., & Bittles, A. H. (2020). Haemoglobinopathies in India: estimates of blood requirements and treatment costs for the decade 2017–2026. *Journal of Community Genetics*, *11*(1), 39-45.

Srivastava, D. K., Tanwar, U., Krishna Rao, M.G., & Manohar, P. (2021). A Research Paper on Blood Donation Management. *International Journal of creative research thoughts*, *9*(5).