Лабораторная работа №7-8, Часть 1: Знакомство с онтологиями в Protégé

Цель работы: Освоить базовые принципы работы с онтологиями и семантическими технологиями через инструмент Protégé. Получить практические навыки изучения структуры онтологий, работы с классами, свойствами и индивидами.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Инструмент:** Protégé Desktop
- **Форматы:** OWL, RDF, Turtle
- **Языки:** OWL 2, RDFS
- **Датасет:** Pizza ontology (образовательная онтология)

Теоретическая часть

1. Онтологии в компьютерных науках Онтология — формальное представление знаний в виде иерархии понятий и отношений между ними. Ключевые компоненты:

- **Классы (Concepts):** Категории объектов предметной области
- **Свойства (Properties):** Отношения между объектами
- **Индивиды (Individuals):** Конкретные экземпляры классов
- **Аксиомы (Axioms):** Правила и ограничения

2. Язык OWL (Web Ontology Language) Стандарт W3C для создания онтологий:

- **OWL DL:** Поддержка сложных логических конструкций
- **Семантика описательной логики:** Формальная основа для логического вывода
- **Инференс:** Автоматическое выведение новых знаний

3. Protégé Ведущий open-source инструмент для работы с онтологиями:

- **Визуальный редактор:** Графическое создание и редактирование онтологий
- **Поддержка推理:** Интеграция с reasoners (HermiT, Pellet)
- **Расширяемость:** Поддержка плагинов

Задание на практическую реализацию

Этап 1: Установка и настройка Protégé

1. **Установка Java (требование для Protégé):**

```
sudo apt update
sudo apt install openjdk-17-jdk
java -version
```

2. **Скачивание и установка Protégé:**

```
wget https://github.com/protegeproject/protege-
distribution/releases/download/v5.6.2/protege-5.6.2-platform.zip
unzip protege-5.6.2-platform.zip
cd protege-5.6.2
```

3. **Запуск Protégé:**

```
./run.sh
```

Этап 2: Загрузка и изучение образовательной онтологии

1. **Загрузка Pizza Ontology:**
   - В меню Protégé выберите: `File → Open from URL...`
   - Введите URL: `https://protege.stanford.edu/ontologies/pizza/pizza.owl`
   - Сохраните локальную копию: `File → Save As → pizza.owl`

2. **Изучение интерфейса Protégé:**
   - **Active Ontology:** Метаинформация об онтологии
   - **Entities:** Основные сущности (Classes, Properties, Individuals)
   - **Class Description:** Описание выбранного класса
   - **OWL Viz:** Визуализация иерархии классов

Этап 3: Анализ структуры онтологии

1. **Изучение вкладки "Classes":**

```
# Создание файла для заметок
touch ontology_analysis.txt
```

   Задание: Изучите иерархию классов и запишите наблюдения:
   - Корневые классы онтологии
   - Глубина иерархии
   - Основные категории пицц

2. **Анализ свойств объекта:**
   - Перейдите во вкладку **Object Properties**
   - Изучите основные отношения:
     - `hasTopping`

- `hasBase`
- `hasSpiciness`

3. **Изучение индивидов:**
   - Перейдите во вкладку **Individuals**
   - Найдите примеры конкретных пицц
   - Изучите их свойства и связи

Этап 4: Работа с классами и свойствами

1. **Создание нового класса:**
   - В **Classes** нажмите `Add subclass`
   - Создайте класс: `RussianPizza`
   - Добавьте аннотацию: `Comment: Traditional Russian pizza variations`
2. **Определение свойств класса:**
   - В **Class Description** выберите `RussianPizza`
   - Добавьте ограничение: `hasTopping some RedOnion`
   - Добавьте ограничение: `hasTopping some Sausage`
3. **Создание объектаных свойств:**
   - В **Object Properties** создайте свойство: `hasTraditionalTopping`
   - Установите домен: `Pizza`
   - Установите диапазон: `PizzaTopping`

Этап 5: Использование reasoner для логического вывода

1. **Выбор и запуск reasoner:**
   - В меню выберите: `Reasoner → HermiT`
   - Запустите: `Reasoner → Start reasoner`
2. **Анализ результатов:**
   - Проверьте автоматическую классификацию пицц
   - Изучите непротиворечивость онтологии
   - Найдите новые inferred классы
3. **Создание запроса:**
   - Перейдите во вкладку **DL Query**
   - Введите запрос: `Pizza and hasTopping value Mushroom`
   - Проанализируйте результаты

Этап 6: Сохранение и экспорт онтологии

1. **Сохранение в разных форматах:**

```
# Сохранение в формате Turtle
File → Save As → pizza_russian.ttl

# Экспорт в RDF/XML
File → Save As → pizza_russian.rdf
```

2. **Создание отчета о онтологии:**

```
# Скрипт для анализа онтологии
from rdflib import Graph
import pandas as pd

def analyze_ontology(file_path):
    g = Graph()
    g.parse(file_path, format="turtle")

    # Статистика онтологии
    classes = list(g.subjects(predicate="http://www.w3.org/1999/02/22-rdf-
syntax-ns#type",
                              object="http://www.w3.org/2002/07/owl#Class"))

    properties = list(g.subjects(predicate="http://www.w3.org/1999/02/22-
rdf-syntax-ns#type",

object="http://www.w3.org/2002/07/owl#ObjectProperty"))

    individuals = list(g.subjects(predicate="http://www.w3.org/1999/02/22-
rdf-syntax-ns#type",

object="http://www.w3.org/2002/07/owl#NamedIndividual"))

    print(f"Классы: {len(classes)}")
    print(f"Свойства: {len(properties)}")
    print(f"Индивиды: {len(individuals)}")

    return {
        "classes": len(classes),
        "properties": len(properties),
        "individuals": len(individuals)
    }

# Анализ оригинальной и модифицированной онтологии
stats_original = analyze_ontology("pizza.owl")
stats_modified = analyze_ontology("pizza_russian.ttl")

# Создание отчета
report = pd.DataFrame([stats_original, stats_modified],
                      index=["Original", "Modified"])
report.to_csv("ontology_report.csv")
```

Этап 7: Документирование онтологии

1. **Создание документации:**

```
# Анализ онтологии Pizza

## Основные классы
- Pizza: Базовый класс всех пицц
- PizzaTopping: Ингредиенты для пиццы
- PizzaBase: Основа пиццы

## Добавленные элементы
- Класс: RussianPizza
- Свойство: hasTraditionalTopping

## Статистика
| Метрика | Оригинал | Модифицированная |
|---------|----------|------------------|
| Классы | X | Y |
| Свойства | X | Y |
| Индивиды | X | Y |
```

Требования к оформлению и отчету

Критерии оценки для Части 1:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (установка Protégé, загрузка онтологии). Создана локальная копия онтологии.
- **Хорошо:** Дополнительно успешно выполнен Этап 3-4 (анализ структуры, создание новых классов и свойств). Модифицированная онтология сохранена.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 5-7: использование reasoner, экспорт в разные форматы, создание отчета статистики. Проанализированы изменения в онтологии.

Рекомендуемая литература

1. **Protégé Documentation:** https://protegeproject.github.io/protege/
2. **OWL 2 Primer:** https://www.w3.org/TR/owl2-primer/
3. **Pizza Ontology Tutorial:** https://protege.stanford.edu/ontologies/tutorials/protege-owl-tutorial.pdf
4. **Semantic Web Technologies:** https://www.w3.org/standards/semanticweb/
5. **RDFLib Documentation:** https://rdflib.readthedocs.io/

Лабораторная работа №7-8, Часть 2: Работа с SPARQL-запросами

Цель работы: Освоить язык запросов SPARQL для работы с семантическими данными. Получить практические навыки подключения к семантическому хранилищу, выполнения различных типов запросов и анализа результатов.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Хранилище:** Apache Jena Fuseki
- **Библиотеки:** `SPARQLWrapper`, `rdflib`, `pandas`
- **Язык запросов:** SPARQL 1.1
- **Форматы данных:** RDF, OWL, Turtle

Теоретическая часть

1. Язык SPARQL SPARQL (SPARQL Protocol and RDF Query Language) — стандартный язык запросов для RDF-данных. Основные типы запросов:

- **SELECT:** Возвращает таблицу результатов
- **CONSTRUCT:** Создает новый RDF-граф
- **ASK:** Возвращает boolean-ответ
- **DESCRIBE:** Возвращает RDF-описание ресурса

2. Apache Jena Fuseki Сервер SPARQL с веб-интерфейсом для работы с RDF-данными:

- **Поддержка SPARQL 1.1:** Полная реализация стандарта
- **Веб-интерфейс:** Интерактивное выполнение запросов
- **REST API:** Программный доступ к данным

3. Структура SPARQL-запроса

- **PREFIX:** Определение пространств имен
- **SELECT/CONSTRUCT:** Цель запроса
- **WHERE:** Шаблон для сопоставления
- **FILTER:** Условия фильтрации
- **OPTIONAL:** Необязательные совпадения
- **ORDER BY/LIMIT:** Сортировка и ограничения

Задание на практическую реализацию

Этап 1: Установка и запуск Apache Jena Fuseki

1. **Скачивание и установка:**

```
wget https://archive.apache.org/dist/jena/binaries/apache-jena-fuseki-
4.10.0.tar.gz
tar -xzf apache-jena-fuseki-4.10.0.tar.gz
cd apache-jena-fuseki-4.10.0
```

2. **Запуск Fuseki сервера:**

```
./fuseki-server --mem --update /ds
```

3. **Проверка работы:**
   - Откройте браузер: `http://localhost:3030`
   - Убедитесь, что сервер доступен

Этап 2: Загрузка онтологии в Fuseki

1. **Загрузка данных через веб-интерфейс:**
   - Перейдите в `Manage Datasets` → `Add new dataset`
   - Выберите `in-memory` хранилище
   - Имя: `pizza_ds`
   - Нажмите `Create`
2. **Загрузка онтологии:**
   - Перейдите в `Upload data`
   - Загрузите файл `pizza.owl`
   - Нажмите `Upload`
3. **Проверка загрузки:**
   - Перейдите во вкладку `Query`
   - Выполните тестовый запрос:

```
SELECT (COUNT(*) AS ?count) WHERE { ?s ?p ?o }
```

Этап 3: Написание базовых SPARQL-запросов

1. **Создание скрипта для работы с SPARQL:**

```
touch sparql_queries.py
```

2. **Настройка подключения:**

```
from SPARQLWrapper import SPARQLWrapper, JSON, XML
import pandas as pd

# Настройка SPARQL endpoint
sparql = SPARQLWrapper("http://localhost:3030/ds/sparql")
sparql.setReturnFormat(JSON)

def run_query(query):
    sparql.setQuery(query)
    try:
        results = sparql.query().convert()
        return results
    except Exception as e:
        print(f"Ошибка выполнения запроса: {e}")
        return None
```

3. **Запрос 1: Получение всех классов онтологии**

```
query1 = """
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?class ?label
WHERE {
    ?class a owl:Class .
    OPTIONAL { ?class rdfs:label ?label }
}
ORDER BY ?class
"""

results1 = run_query(query1)
print("Классы онтологии:")
for result in results1["results"]["bindings"]:
    print(f"{result['class']['value']} - {result.get('label',
{}).get('value', 'No label')}")
```

4. **Запрос 2: Поиск всех пицц**

```
query2 = """
PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?pizza ?name
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza rdfs:label ?name .
}
ORDER BY ?name
"""
```

```
results2 = run_query(query2)
print("\nВсе пиццы:")
for result in results2["results"]["bindings"]:
    print(result['name']['value'])
```

Этап 4: Сложные запросы с фильтрацией

1. **Запрос 3: Пиццы с определенной начинкой**

```
query3 = """
PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?pizza ?name ?topping
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza rdfs:label ?name .
    ?pizza pizza:hasTopping ?toppingObj .
    ?toppingObj rdfs:label ?topping .
    FILTER (CONTAINS(LCASE(?topping), "mushroom"))
}
"""

results3 = run_query(query3)
print("\nПиццы с грибами:")
for result in results3["results"]["bindings"]:
    print(f"{result['name']['value']} - {result['topping']['value']}")
```

2. **Запрос 4: Статистика по начинкам**

```
query4 = """
PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?topping (COUNT(?pizza) AS ?count)
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza pizza:hasTopping ?toppingObj .
    ?toppingObj rdfs:label ?topping .
}
GROUP BY ?topping
ORDER BY DESC(?count)
LIMIT 10
"""

results4 = run_query(query4)
print("\nПопулярные начинки:")
```

```
for result in results4["results"]["bindings"]:
    print(f"{result['topping']['value']}: {result['count']['value']}")
```

Этап 5: CONSTRUCT-запросы для создания новых данных

1. **Запрос 5: Создание RDF-графа вегетарианских пицц**

```
query5 = """
PREFIX pizza: <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ex: <http://example.org/vegetarian#>

CONSTRUCT {
    ?pizza ex:isVegetarian true .
    ?pizza ex:hasTopping ?topping .
}
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza rdfs:label ?name .
    ?pizza pizza:hasTopping ?toppingObj .
    ?toppingObj rdfs:label ?topping .
    FILTER NOT EXISTS {
        ?toppingObj a pizza:MeatTopping .
    }
}
"""

# Для CONSTRUCT запросов меняем формат вывода
sparql.setReturnFormat(XML)
results5 = run_query(query5)
print("CONSTRUCT запрос выполнен")

# Сохранение результатов
with open("vegetarian_pizzas.rdf", "w") as f:
    f.write(results5.toxml())
```

Этап 6: Работа с онтологией через RDFLib

1. **Альтернативный способ работы с данными:**

```
from rdflib import Graph, Namespace
from rdflib.plugins.stores import sparqlstore

# Создание графа с SPARQL endpoint
store = sparqlstore.SPARQLUpdateStore()
store.open(('http://localhost:3030/ds/sparql',
'http://localhost:3030/ds/update'))

g = Graph(store)
```

```
# Определение namespace
PIZZA = Namespace("http://www.co-ode.org/ontologies/pizza/pizza.owl#")
RDFS = Namespace("http://www.w3.org/2000/01/rdf-schema#")

# Запрос через RDFLib
query6 = """
SELECT ?pizza ?name
WHERE {
    ?pizza a pizza:Pizza .
    ?pizza rdfs:label ?name .
}
LIMIT 5
"""

results6 = g.query(query6, initNs={"pizza": PIZZA, "rdfs": RDFS})
print("\nРезультаты через RDFLib:")
for row in results6:
    print(f"{row.pizza} - {row.name}")
```

Этап 7: Создание комплексных отчетов

1. **Генерация отчета по онтологии:**

```
def generate_ontology_report():
    queries = {
        "total_classes": """
            SELECT (COUNT(DISTINCT ?class) AS ?count)
            WHERE { ?class a owl:Class }
        """,
        "total_properties": """
            SELECT (COUNT(DISTINCT ?prop) AS ?count)
            WHERE { ?prop a owl:ObjectProperty }
        """,
        "total_individuals": """
            SELECT (COUNT(DISTINCT ?ind) AS ?count)
            WHERE { ?ind a owl:NamedIndividual }
        """
    }

    report = {}
    for name, query in queries.items():
        results = run_query(query)
        if results:
            count = results["results"]["bindings"][0]["count"]["value"]
            report[name] = count

    # Сохранение отчета
    df = pd.DataFrame([report])
    df.to_csv("ontology_report.csv", index=False)
    return report
```

```
ontology_stats = generate_ontology_report()
print("\nСтатистика онтологии:")
for key, value in ontology_stats.items():
    print(f"{key}: {value}")
```

Этап 8: Интеграционные тесты

1. **Тестирование различных endpoint:**

```
def test_endpoints():
    endpoints = [
        "http://localhost:3030/ds/sparql",
        "http://dbpedia.org/sparql",
        "http://query.wikidata.org/sparql"
    ]

    test_query = "SELECT (COUNT(*) AS ?count) WHERE { ?s ?p ?o } LIMIT 1"

    for endpoint in endpoints:
        try:
            sparql = SPARQLWrapper(endpoint)
            sparql.setQuery(test_query)
            sparql.setReturnFormat(JSON)
            results = sparql.query().convert()
            print(f"{endpoint}: Работает ({results['results']['bindings'][0]
['count']['value']} triplets)")
        except:
            print(f"{endpoint}: Не доступен")

test_endpoints()
```

Требования к оформлению и отчету

Критерии оценки для Части 2:

- **Удовлетворительно:** Успешно выполнены Этапы 1-3 (запуск Fuseki, базовые SELECT-запросы). Получены списки классов и пицц.
- **Хорошо:** Дополнительно успешно выполнен Этап 4-5 (сложные запросы с фильтрацией, CONSTRUCT-запросы). Создан RDF-граф вегетарианских пицц.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 6-8: работа через RDFLib, генерация отчетов, интеграционные тесты. Проанализирована структура онтологии через SPARQL.

Рекомендуемая литература

1. **SPARQL 1.1 Specification:** https://www.w3.org/TR/sparql11-query/

2. **Apache Jena Documentation:** https://jena.apache.org/documentation/fuseki2/
3. **SPARQLWrapper Documentation:** https://sparqlwrapper.readthedocs.io/
4. **RDFLib Documentation:** https://rdflib.readthedocs.io/
5. **Learning SPARQL:** https://www.learnsparql.com/

Лабораторная работа №7-8, Часть 3: Извлечение данных с помощью LLM

Цель работы: Исследовать возможности использования языковых моделей для генерации SPARQL-запросов по текстовым описаниям на естественном языке. Получить практические навыки интеграции LLM с семантическими технологиями.

Стек технологий:

- **ОС:** Ubuntu 24.04 LTS
- **Окружение:** Conda (`mlops-lab`)
- **Библиотеки:** `transformers`, `sparqlwrapper`, `rdflib`, `openai` (опционально)
- **Модели:** GPT-3.5/4, Llama 2, Mistral (через Hugging Face)
- **Инструменты:** Jena Fuseki, Protégé

Теоретическая часть

1. Генерация SPARQL через NL-to-SPARQL Преобразование естественного языка (Natural Language) в SPARQL:

- **Zero-shot подход:** Генерация без примеров
- **Few-shot подход:** Генерация с несколькими примерами
- **Fine-tuning:** Специализированное обучение на парах (вопрос-SPARQL)

2. Архитектура решения

- **Вход:** Текстовый запрос на естественном языке
- **Обработка:** LLM генерирует SPARQL-запрос
- **Валидация:** Проверка синтаксиса и выполнение запроса
- **Итерация:** Исправление ошибок через feedback loop

3. Оценка качества

- **Синтаксическая корректность:** Правильность SPARQL-синтаксиса
- **Семантическая корректность:** Соответствие intent пользователя
- **Эффективность:** Оптимальность выполнения запроса

Задание на практическую реализацию

Этап 1: Настройка окружения и подключение к LLM

1. **Установка необходимых пакетов:**

```
conda activate mlops-lab
pip install transformers sparqlwrapper rdflib openai
```

2. **Создание скрипта для работы с LLM:**

```
touch llm_sparql_generation.py
```

3. **Настройка подключения к LLM (Hugging Face):**

```
from transformers import pipeline, AutoTokenizer, AutoModelForCausalLM
import torch

class SPARQLGenerator:
    def __init__(self, model_name="mistralai/Mistral-7B-Instruct-v0.2"):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForCausalLM.from_pretrained(
            model_name,
            torch_dtype=torch.float16,
            device_map="auto"
        )
        self.tokenizer.pad_token = self.tokenizer.eos_token

    def generate_sparql(self, natural_language_query):
        prompt = f"""
        Convert the following natural language query to SPARQL for the Pizza
ontology.
        Use prefixes: PREFIX pizza: <http://www.co-
ode.org/ontologies/pizza/pizza.owl#>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

        Natural language: {natural_language_query}
        SPARQL:
        """

        inputs = self.tokenizer(prompt, return_tensors="pt",
truncation=True, max_length=512)
        with torch.no_grad():
            outputs = self.model.generate(
                **inputs,
                max_new_tokens=200,
                temperature=0.7,
                do_sample=True,
                pad_token_id=self.tokenizer.eos_token_id
            )

        generated_text = self.tokenizer.decode(outputs[0],
skip_special_tokens=True)
        sparql_query = generated_text.split("SPARQL:")[-1].strip()

        return sparql_query
```

Этап 2: Базовая генерация SPARQL-запросов

1. **Функция для тестирования генерации:**

```
def test_basic_generation():
    generator = SPARQLGenerator()

    test_queries = [
        "Find all pizzas that have mushroom as topping",
        "Show me vegetarian pizzas",
        "List pizzas with spicy toppings",
        "Find pizzas that are not too spicy",
        "Show me pizzas with cheese and tomato"
    ]

    for query in test_queries:
        print(f"\nNatural language: {query}")
        sparql = generator.generate_sparql(query)
        print(f"Generated SPARQL: {sparql}")
        print("-" * 50)

test_basic_generation()
```

Этап 3: Валидация и выполнение сгенерированных запросов

1. **SPARQL валидатор и исполнитель:**

```
from SPARQLWrapper import SPARQLWrapper, JSON, SPARQLExceptions
import re

class SPARQLValidator:
    def __init__(self, endpoint="http://localhost:3030/ds/sparql"):
        self.endpoint = endpoint
        self.sparql = SPARQLWrapper(endpoint)
        self.sparql.setReturnFormat(JSON)

    def validate_syntax(self, query):
        """Проверка синтаксиса SPARQL"""
        try:
            # Базовая проверка структуры
            if not query.strip().upper().startswith(('SELECT', 'CONSTRUCT',
'ASK', 'DESCRIBE')):
                return False, "Query must start with SELECT, CONSTRUCT, ASK
or DESCRIBE"

            # Проверка наличия WHERE clause
            if "WHERE" not in query.upper():
                return False, "Missing WHERE clause"

            return True, "Syntax appears valid"
        except Exception as e:
            return False, f"Syntax validation error: {e}"
```

```
    def execute_query(self, query):
        """Выполнение SPARQL-запроса"""
        try:
            self.sparql.setQuery(query)
            results = self.sparql.query().convert()
            return True, results
        except SPARQLExceptions.QueryBadFormed as e:
            return False, f"Malformed query: {e}"
        except Exception as e:
            return False, f"Execution error: {e}"

def test_generated_queries():
    generator = SPARQLGenerator()
    validator = SPARQLValidator()

    test_cases = [
        "Find all pizzas with mushroom",
        "Show me non-vegetarian pizzas",
        "List pizzas with exactly two toppings"
    ]

    for query in test_cases:
        print(f"\n{'='*60}")
        print(f"Testing: {query}")

        # Генерация SPARQL
        sparql = generator.generate_sparql(query)
        print(f"Generated: {sparql}")

        # Валидация синтаксиса
        is_valid, syntax_msg = validator.validate_syntax(sparql)
        print(f"Syntax valid: {is_valid} - {syntax_msg}")

        # Выполнение запроса
        if is_valid:
            success, result = validator.execute_query(sparql)
            if success:
                print("Query executed successfully!")
                if "results" in result and "bindings" in result["results"]:
                    bindings = result["results"]["bindings"]
                    print(f"Results: {len(bindings)} found")
                    for i, binding in enumerate(bindings[:3]):
                        print(f"  {i+1}. {binding}")
            else:
                print(f"Execution failed: {result}")
```

Этап 4: Few-shot обучение через промпты

1. **Улучшенный генератор с примерами:**

```
class ImprovedSPARQLGenerator(SPARQLGenerator):
    def __init__(self, model_name="mistralai/Mistral-7B-Instruct-v0.2"):
        super().__init__(model_name)
        self.examples = [
            {
                "nl": "Find all pizzas with mushroom topping",
                "sparql": """
                PREFIX pizza: <http://www.co-
ode.org/ontologies/pizza/pizza.owl#>
                PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

                SELECT ?pizza ?name
                WHERE {
                    ?pizza a pizza:Pizza .
                    ?pizza rdfs:label ?name .
                    ?pizza pizza:hasTopping ?topping .
                    ?topping rdfs:label ?toppingName .
                    FILTER (CONTAINS(LCASE(?toppingName), "mushroom"))
                }
                """
            },
            {
                "nl": "Show me vegetarian pizzas",
                "sparql": """
                PREFIX pizza: <http://www.co-
ode.org/ontologies/pizza/pizza.owl#>
                PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

                SELECT ?pizza ?name
                WHERE {
                    ?pizza a pizza:Pizza .
                    ?pizza rdfs:label ?name .
                    FILTER NOT EXISTS {
                        ?pizza pizza:hasTopping ?topping .
                        ?topping a pizza:MeatTopping .
                    }
                }
                """
            }
        ]

    def generate_with_examples(self, natural_language_query):
        prompt = "Convert natural language queries to SPARQL for Pizza
ontology.\n\n"

        # Добавление примеров
        for example in self.examples:
            prompt += f"NL: {example['nl']}\nSPARQL:
{example['sparql']}\n\n"

        prompt += f"NL: {natural_language_query}\nSPARQL:"

        inputs = self.tokenizer(prompt, return_tensors="pt",
```

```
truncation=True, max_length=1024)
        with torch.no_grad():
            outputs = self.model.generate(
                **inputs,
                max_new_tokens=300,
                temperature=0.3,  # Более детерминированное поколение
                do_sample=True
            )

        generated_text = self.tokenizer.decode(outputs[0],
skip_special_tokens=True)
        sparql_query = generated_text.split("SPARQL:")[-1].strip()

        return sparql_query
```

Этап 5: Интеграция с семантическим валидатором

1. **Расширенная валидация с семантической проверкой:**

```
class SemanticValidator(SPARQLValidator):
    def validate_ontology_compatibility(self, query):
        """Проверка совместимости с онтологией"""
        try:
            # Проверка использования правильных префиксов
            if "pizza:" not in query:
                return False, "Missing pizza prefix usage"

            # Проверка существующих классов/свойств
            class_check = self.execute_query("""
                PREFIX pizza: <http://www.co-
ode.org/ontologies/pizza/pizza.owl#>
                SELECT DISTINCT ?class WHERE { ?class a owl:Class }
            """)

            if class_check[0]:
                valid_classes = [str(r['class']['value']) for r in
class_check[1]['results']['bindings']]
                # Простая проверка на наличие pizza:Pizza в запросе
                if "pizza:Pizza" in query and "http://www.co-
ode.org/ontologies/pizza/pizza.owl#Pizza" in valid_classes:
                    return True, "Ontology compatibility check passed"

            return False, "May reference non-existent ontology elements"

        except Exception as e:
            return False, f"Ontology validation error: {e}"

def comprehensive_test():
    generator = ImprovedSPARQLGenerator()
    validator = SemanticValidator()
```

```
    complex_queries = [
        "Find pizzas that have both cheese and tomato toppings",
        "Show me spicy pizzas that are not too expensive",
        "List vegetarian pizzas with exactly three toppings"
    ]

    results = []
    for query in complex_queries:
        print(f"\nTesting complex query: {query}")

        # Генерация с примерами
        sparql = generator.generate_with_examples(query)

        # Многоуровневая валидация
        syntax_ok, syntax_msg = validator.validate_syntax(sparql)
        ontology_ok, ontology_msg =
validator.validate_ontology_compatibility(sparql)

        result = {
            "query": query,
            "generated_sparql": sparql,
            "syntax_valid": syntax_ok,
            "ontology_compatible": ontology_ok,
            "execution_result": None
        }

        if syntax_ok and ontology_ok:
            success, exec_result = validator.execute_query(sparql)
            result["execution_success"] = success
            result["execution_result"] = exec_result if success else
str(exec_result)

        results.append(result)

        # Вывод результатов
        print(f"Syntax: {syntax_ok} ({syntax_msg})")
        print(f"Ontology: {ontology_ok} ({ontology_msg})")
        if result.get("execution_success"):
            print("Execution: Successful")

    return results
```

Этап 6: Оценка качества и метрики

1. **Система оценки сгенерированных запросов:**

```
def evaluate_sparql_generation():
    test_dataset = [
        {
            "nl": "Find pizzas with mushroom",
            "expected_patterns": ["pizza:hasTopping", "mushroom", "FILTER"],
```

```
            "min_results": 1
        },
        {
            "nl": "Show vegetarian pizzas",
            "expected_patterns": ["FILTER NOT EXISTS", "pizza:MeatTopping"],
            "min_results": 3
        }
    ]

    generator = ImprovedSPARQLGenerator()
    validator = SemanticValidator()

    evaluation_results = []

    for test_case in test_dataset:
        nl_query = test_case["nl"]
        print(f"\nEvaluating: {nl_query}")

        # Генерация
        sparql = generator.generate_with_examples(nl_query)

        # Проверка ожидаемых паттернов
        pattern_matches = sum(1 for pattern in
test_case["expected_patterns"] if pattern in sparql)
        pattern_score = pattern_matches /
len(test_case["expected_patterns"])

        # Выполнение и проверка результатов
        exec_success, exec_result = validator.execute_query(sparql)
        result_count = len(exec_result["results"]["bindings"]) if
exec_success else 0
        result_score = 1.0 if result_count >= test_case["min_results"] else
result_count / test_case["min_results"]

        # Общая оценка
        total_score = (pattern_score * 0.6) + (result_score * 0.4)

        evaluation_results.append({
            "query": nl_query,
            "generated_sparql": sparql,
            "pattern_score": pattern_score,
            "result_score": result_score,
            "total_score": total_score,
            "status": "PASS" if total_score >= 0.7 else "FAIL"
        })

        print(f"Score: {total_score:.2f} (Patterns: {pattern_score:.2f},
Results: {result_score:.2f})")
        print(f"Status: {evaluation_results[-1]['status']}")

    # Сохранение результатов оценки
    import json
    with open("sparql_generation_evaluation.json", "w") as f:
        json.dump(evaluation_results, f, indent=2)
```

```
    return evaluation_results
```

Этап 7: Создание демонстрационного интерфейса

1. **Простой веб-интерфейс с Flask:**

```
from flask import Flask, request, jsonify
import threading

app = Flask(__name__)
generator = ImprovedSPARQLGenerator()
validator = SemanticValidator()

@app.route('/generate-sparql', methods=['POST'])
def generate_sparql_endpoint():
    data = request.json
    nl_query = data.get('query', '')

    if not nl_query:
        return jsonify({"error": "No query provided"}), 400

    try:
        # Генерация SPARQL
        sparql = generator.generate_with_examples(nl_query)

        # Валидация
        syntax_ok, syntax_msg = validator.validate_syntax(sparql)
        ontology_ok, ontology_msg =
validator.validate_ontology_compatibility(sparql)

        response = {
            "natural_language_query": nl_query,
            "generated_sparql": sparql,
            "validation": {
                "syntax": {"valid": syntax_ok, "message": syntax_msg},
                "ontology": {"valid": ontology_ok, "message": ontology_msg}
            }
        }

        return jsonify(response)

    except Exception as e:
        return jsonify({"error": str(e)}), 500

def run_flask_app():
    app.run(host='0.0.0.0', port=5001, debug=False)

# Запуск в отдельном потоке
# flask_thread = threading.Thread(target=run_flask_app)
# flask_thread.start()
```

Требования к оформлению и отчету

Критерии оценки для Части 3:

- **Удовлетворительно:** Успешно выполнены Этапы 1-2 (настройка LLM, базовая генерация). Сгенерированы SPARQL-запросы для простых вопросов.
- **Хорошо:** Дополнительно успешно выполнен Этап 3-4 (валидация, few-shot обучение). Реализована система проверки синтаксиса и семантики.
- **Отлично:** Все задания выполнены в полном объеме. Реализованы Этапы 5-7: семантическая валидация, система оценки, демонстрационный интерфейс. Проведен комплексный анализ качества генерации.

Рекомендуемая литература

1. **NL-to-SPARQL Survey:** https://arxiv.org/abs/2104.07281
2. **GPT-3 for SPARQL Generation:** https://arxiv.org/abs/2210.07812
3. **Hugging Face Transformers:** https://huggingface.co/docs/transformers
4. **SPARQL 1.1 Specification:** https://www.w3.org/TR/sparql11-query/
5. **Prompt Engineering Guide:** https://www.promptingguide.ai/