Mestrado Integrado em Engenharia Informática

Scripting no Processamento de Linguagem Natural

TP3: Web scraping + ontologia

Rafael Silva
(A74264)

25 de Junho de 2020

Conteúdo

1 Introdução **2**

2 Desenvolvimento **2**

2.1 Scrapy e Web-Crawling 2

2.1.1 Web-Site 2

2.1.2 Script e Web-Crawler 3

2.1.3 Recolha de informação 5

2.1.4 Notas 7

2.2 Ontologia 8

2.2.1 Criação da Ontologia 8

2.2.2 Povoamento da Ontologia 10

3 Extra **11**

3.1 GraphDB 11

Capítulo 1

Introdução

Este projecto consiste em utilizar a livraria **scrapy** de *python* para a construção de uma ontologia com escolha dos dados a recair sobre o autor.
De seguida encontra-se explicada em que consiste a livraria **scrapy** bem como o seu uso, e o termo denominado por *Web-Crawling*.
Também irei apresentar o *web-site* de onde foi recolhida a informação, bem como a metodologia para extrair a informação, o respectivo tratamento da mesma e, por fim, como utilizar essa informação para a construção de uma ontologia.

Capítulo 2

Desenvolvimento

Neste capítulo irei falar do desenvolvimento do projecto bem como dos *scripts* e ferramentas que foram utilizadas para a sua resolução.

2.1 Scrapy e Web-Crawling

De uma forma breve, a livraria **scrapy** consiste numa *framework* de *web-crawling*. Originalmente idealizado para *web-scraping*, também pode ser utilizado para extração de dados usando APIs ou como *web-crawler* de forma mais abrangente.
A arquitectura do **scrapy** é construída em torno de "*spiders*", que são crawlers autônomos que recebem um conjunto de instruções.
Um *web-crawler* é um programa que navega pela Internet de uma forma metódica e automatizada. O mesmo utiliza o código HTML da página fonte para fazer essa navegação.
De seguida, irei explicar como foi desenvolvido o *script* em *python* para fazer *web-crawler* e o *web-site* onde o mesmo foi utilizado.

2.1.1 Web-Site

O *web-site* de onde foi recolhida a informação é denominado por AutoEvolution, o mesmo contém informação acerca de todas as marcas de carro a nível mundial que existem até a data de hoje, bem como os modelos de cada marca e as suas respectivas versões e especificações dessas versões, também contem informação extra, tal como a história da marca, quantos modelos a mesma possui, fotos de cada modelo e versões (irão ser guardados todos os links necessários) e entre outros que serão explicados mais a frente.

2.1.2 Script e Web-Crawler

Nesta secção irei explicar como foi construído o *web-crawler* e a respectiva travessia pelo *web-site* em questão. O projecto foi construído utilizando a livraria **scrapy**, o mesmo foi gerado com o seguinte comando *bash*:

```
scrapy startproject autoevolution_scraper
```

Este comando irá criar um projecto default do **scrapy**, de seguida iremos efetuar o comando para a criação de um "spider"seguido do link do nosso web-site:

```
scrapy genspider AutoEvolution autoevolution.com
```

Com o nosso "spider"criado já podemos começar a construir o nosso *web-crawler*. O mesmo terá um formato default da seguinte forma:

```
import scrapy
class AutoevolutionSpider(scrapy.Spider):
    name = 'AutoEvolution'
    allowed_domains = ['autoevolution.com']
    start_urls = ['http://autoevolution.com/cars/']
    def parse(self, response):
        pass
```

Como é possível observar, existem duas lista, a primeira denominada por *allowed_domains*, que nos indica quais sãos os respectivos domínios permitidos para o nosso *web-crawler* navegar e de seguida temos a lista *start_urls*, que nos indica uma sucessão de links pelos quais o nosso *web-crawler* irá navegar, começando sempre pelo primeiro elemento dessa lista.
O mesmo irá começar sempre pela função **parse**, que irá utilizar o primeiro elemento da lista *start_urls*.
Após uma investigação rápida utilizando a ferramenta inspeccionar do *browser* encontramos o nosso elemento principal que queremos fazer *web-crawler*.

A navegação irá ser efectuada da seguinte forma:

- Primeiro iremos aceder a página principal onde se encontram todas as marcas.
- De seguida iremos percorrer marca a marca.
- Dentro da respectiva marca iremos visitar modelo a modelo.
- E por fim iremos visitar a respectiva versão de cada modelo com as respectivas especificações de cada versão.

Um enxerto de código de seguida mostra como a primeira navegação é efectuada:

```
...
def parse(self, response):
    BRAND_LIST_SELECTOR = '.carman'
    for brand in response.css(BRAND_LIST_SELECTOR):
        ...
        URL_BRAND_SELECTOR = 'h5 a ::attr(href)'
        urlBrand = brand.css(URL_BRAND_SELECTOR).extract_first()
```

```
        brand = {
            ...
            'urlBrand': urlBrand,
            ...
        }
        if urlBrand:
            request = scrapy.Request(urlBrand, callback=self.parseBrand, meta={'brand
                ': brand})
            yield request
...
```

Como podemos ver no excerto do código, verifica-se que cada marca é constituída pela classe em *CSS* denominada por **.carman** e, de seguida, é efectuado um ciclo sobre essa classe para visitar todos os elementos da respectiva classe.
Podemos já reparar que o **scrapy** utiliza a resposta efectuada pela visita ao mesmo *web-site* para ser efetuada a respectiva navegação, utilizando para isso a função *response.css('SELECTOR-CLASS')* ou também poderá utilizar outra denominado por *response.xpath('SELECTOR-XPATH')*.
Ambas podem utilizar as funções *extract()* que irá extrair toda a informação ou *extract_first()* que irá extrair a primeira informação encontrada.
Olhando para o código em cima podemos reparar que o **scrapy** irá efetuar a sua navegação para o respetivo URL utilizando um *Request*, este mesmo irá receber um URL, seguido de um *callback*, que é a nossa função que pretendemos utilizar para fazer o novo parser, e finalmente é passada em *meta* a informação recolhida anteriormente.

De seguida temos um enxerto do código da função acima definida como **parseBrand**:

```
...
def parseBrand(self, response):
    ...
    CAR_LIST_SELECTOR = '.carmod'
    for car in response.css(CAR_LIST_SELECTOR):
        ...
        URL_CAR = '.fl > a::attr(href)'
        url_car = car.css(URL_CAR).extract_first()
        model = {
            ...
            'url': url_car,
            ...
        }
        if url_car:
            request = scrapy.Request(url_car, callback=self.parseModelVersion, meta={
                'model': model, 'brand': brand})
            yield request
...
```

Como podemos o reparar o processo é o mesmo que o anterior explicado, que consiste em recolher a informação respectiva ao próximo URL, navegar para o mesmo e passar a informação que foi anteriormente recolhida.

Outro excerto de código mostra a navegação dentro do próprio modelo e a recolha da respectiva informação necessária.

```
...
def parseModelVersion(self, response):
    ...
    VERSION_SELECTOR = '.carmodel'
    for version in response.css(VERSION_SELECTOR):
        ...
        VERSION_SELECTOR_URL = '.col2width > .col12width > .col2width > .engitm > a
            ::attr(href)'
        version_url = version.css(VERSION_SELECTOR_URL).extract_first()
        version = {
            ...
            'version_url': version_url,
            ...
        }
        if version_url:
            request = scrapy.Request(version_url, callback=self.parseSpecsVersion,
                meta={'model': model, 'brand': brand, 'version': version})
            yield request
...
```

A navegação é feita de maneira igual a anterior recolhendo do próximo URL e navegando para o mesmo, neste caso será para a versão do respectivo modelo.

Por fim, temos a última função de *parser* que irá guardar todo o objecto recolhido, esse processo será explicado adiante, e, posteriormente, irá ser realizado o respectivo *parser* a página de especificações.

```
...
def parseSpecsVersion(self, response):
    ...
    version['info'] = info
    model['version'] = version
    ...
...
```

De seguida irei explicar a informação recolhida em cada *parser* e como essa informação foi guardada no final.

2.1.3 Recolha de informação

No primeiro parser foi recolhida a seguinte informação:

- **nameBrand** - Nome da marca.
- **urlBrand** - URL da marca.
- **imageBrand** - Respetivo URL com a imagem da marca.

Esta informação foi passada para o próximo *parser* denominado por *parseBrand* que recolhe a seguinte informação:

- **history** - História da marca.
- **productionModels** - Número de modelos em produção.
- **discontinuedModels** - Número de modelos descontinuados.

Ainda dentro da respectiva marca foi recolhida a informação acerca dos modelos da mesma:

- **name** - Nome do modelo.
- **url** - URL do modelo.
- **img** - URL da imagem do modelo.
- **class** - Classe que o modelo pertence.
- **fuels** - Lista de combustíveis que o modelo utiliza.
- **nrGenerations** - Número de gerações do modelo.
- **modelYears** - Intervalo de tempo que existe o modelo.

Esta informação foi passada para o próximo *parser* denominado por *parseModelVersion* que recolhe a informação individual de cada versão de cada modelo:

- **gallery** - URL com a galeria da versão.
- **years** - Intervalo de tempo que existe a versão.
- **version_url** - URL da versão.
- **version_name** - Nome da versão.

Por fim, temos o último *parser* denominado por *parseSpecsVersion* que irá receber toda a informação recolhida atrás e irá recolher toda a informação relacionada com a versão, tal como as especificações do motor, da carroçaria, a descrição e introdução da versão, o consumo de combustível e etc.
Este último *parser* é responsável pela criação de um objecto denominado por **auto_evolution_scraper_item**, que irá guardar toda a informação necessária para depois ser guardada num ficheiro com o formato *JSON*.

Na figura em baixo podemos visualizar esse objeto:

```
...
def parseSpecsVersion(self, response):
    ...
    auto_evolution_scraper_item = AutoevolutionScraperItem(
        brand=brand,
        model=model
    )
    yield auto_evolution_scraper_item
...
```

Este objecto é criado da seguinte forma:

```
...
class AutoevolutionScraperItem(scrapy.Item):
    brand = scrapy.Field()
    model = scrapy.Field()
```

Sendo uma classe com o nome *AutoevolutionScraperItem*.

Este mesmo objeto é depois escrito seguindo um processo de *pipelines* do **scrapy**:

```
import json
class AutoevolutionScraperPipeline:
    def process_item(self, item, spider):
        line = json.dumps(dict(item), indent=2) + ',\n'
        self.file.write(line)
        return item
    def open_spider(self, spider):
        self.file = open('autoevolution.txt', 'w')
        line = '[\n'
        self.file.write(line)
    def close_spider(self, spider):
        line = ']\n'
        self.file.write(line)
        self.file.close()
```

Para utilizar os *pipelines* do *scrapy* foi necessário descomentar a seguinte linha de código do ficheiro *settings.py*:

```
ITEM_PIPELINES = {
   'autoevolution_scraper.pipelines.AutoevolutionScraperPipeline': 300,
}
```

Por fim, é apenas necessário executar o seguinte comando para que o "*spider*"execute:

```
    scrapy runspider autoevolution_scraper/spiders/AutoEvolution.py
```

2.1.4 Notas

Nesta secção irei explicar algumas notas que foram dificuldades na resolução do projecto.
Como o *web-site* que foi utilizado usa o ficheiro *Robots.txt*, este ficheiro é utilizado quando o *browser* acede ao site e acede as suas "definições"antes de o visitar, dentro desse ficheiro encontrei *flags* para evitar *web-crawler*, para evitar esse problema o **scrapy** tem um *flag* que permite desactivar a leitura desse ficheiro, essa *flag* encontra-se no ficheiro *settings.py*:

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = False
```

De seguida, ocorreu um obstáculo, uma vez que o *web-site* ficava bloqueado após múltiplos pedidos seguidos e de maneira a evitar essa situação foi necessário alterar as seguintes variáveis do ficheiro *settings.py*:

```
# Configure maximum concurrent requests performed by Scrapy (default: 16)
CONCURRENT_REQUESTS = 20
...
# Configure a delay for requests for the same website (default: 0)
AUTOTHROTTLE_START_DELAY = 2
# The maximum download delay to be set in case of high latencies
AUTOTHROTTLE_MAX_DELAY = 60
# The average number of requests Scrapy should be sending in parallel to
# each remote server
AUTOTHROTTLE_TARGET_CONCURRENCY = 5
DOWNLOAD_DELAY = 2
```

Isto permite que cada *Request* seja feito com um atraso, evitando que o web-site bloqueei o meu acesso, e também introduzi um máximo de pedidos paralelos a decorrer.

2.2 Ontologia

Neste capitulo irei explicar o processo de criação da ontologia, ou seja, as suas classes, as suas relações e o tipo de dados que são guardados e por fim como é que a ontologia foi povoada e finalizada.

2.2.1 Criação da Ontologia

A mesma ontologia foi construída utilizando o programa *Protégé*, o mesmo permite criar uma ontologia e defini-la como pretendermos.
A imagem em baixo mostra a interface do *Protégé* já com a ontologia criada e as suas classes.

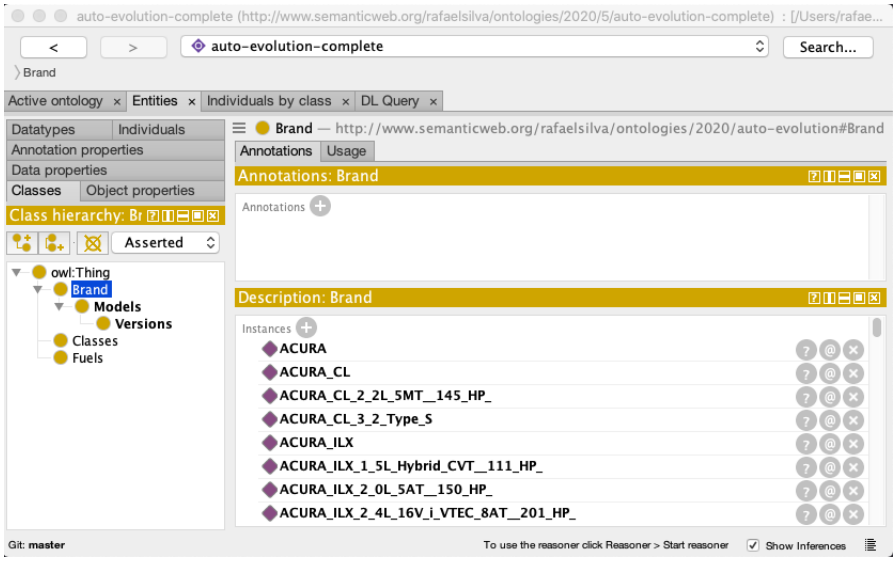

Figura 2.1: Ontologia

Como podemos verificar na figura anterior, a nossa ontologia possui três classes principais sendo que uma delas terá duas sub-classes. Seguindo a seguinte estrutura:

- **Brand** - Classe responsável por representar uma marca em questão e guardar a sua respectiva informação.
- **Models** - Sub-Classe de **Brand** responsável por representar um modelo em questão e guardar a sua respectiva informação.
- **Version** - Sub-Classe de **Models** responsável por representar uma versão de um modelo em questão e guardar a sua respectiva informação.
- **Fuels** - Classe responsável por representar um combustível em questão e guardar a sua respectiva informação.
- **Classes** - Classe responsável por representar uma classe a que um modelo pertence e guardar a sua respectiva informação.

De seguida, estão definidas as relações da nossa ontologia, a mesma terá relações inversas para serem depois usadas por inferências.

- **pertenceClass** - Relação que define que um **Models** pertence a uma **Classes**, o seu inverso será **classDoModelo**.
- **temModelo** - Relação que define que uma **Brand** possui **Models**, o seu inverso será **pertenceMarca**.
- **temVersao** - Relação que define que um **Models** possui **Versions**, o seu inverso será **pertenceModelo**.
- **usaCombustivel** - Relação que define que um **Models** utiliza **Fuels**, o seu inverso será **combustivelUsado**.

Na figura em baixo podemos verificar as relações explicadas em cima:

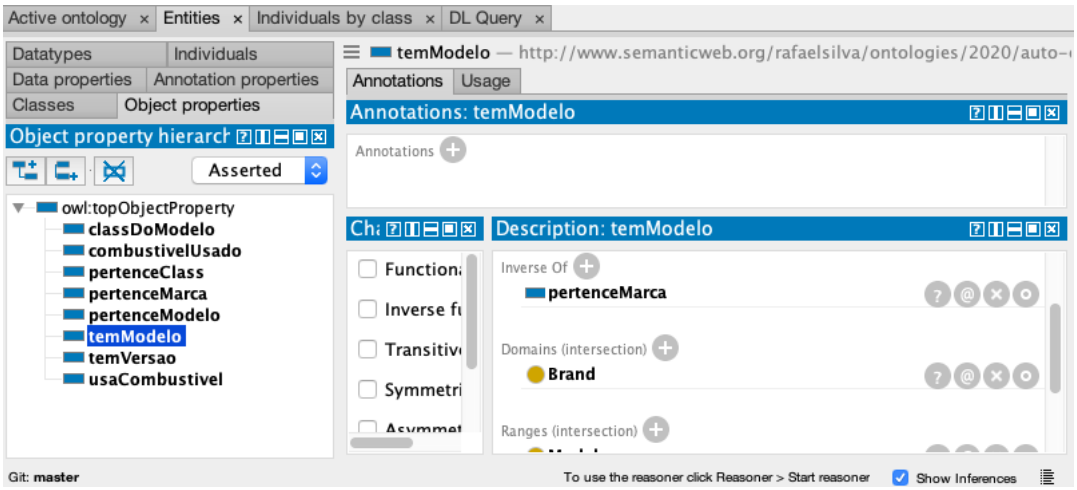

Figura 2.2: Relações da Ontologia

A seguinte figura representa o tipo de dados que foram guardados na nossa ontologia:

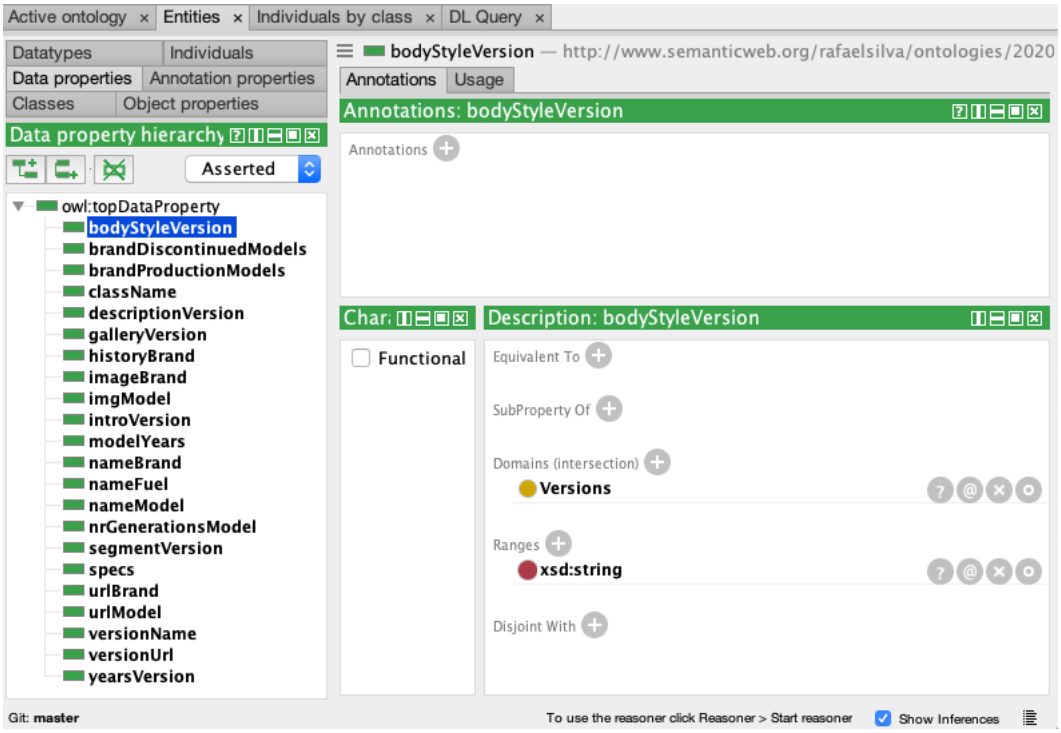

Figura 2.3: Data property da Ontologia

2.2.2 Povoamento da Ontologia

Nesta secção irei explicar o formato de cada classe que foi utilizada.
O povoamento das mesmas foi feito recorrendo a um *script* em *python* que lê o ficheiro com as dados completos, que é o objeto que foi descrito em cima, com a informação da marca, do modelo e da sua versão.

Em baixo encontra-se um exemplo do povoamento da classe **Fuels**:

```
### http://www.semanticweb.org/rafaelsilva/ontologies/2020/auto-evolution#Gasoline
:Gasoline rdf:type owl:NamedIndividual ,
    :Fuels ;
    :nameFuel "Gasoline"^^xsd:string .
```

A mesma classe foi povoada utilizando o script em cima descrito do seguinte modo:

```
def populate_fuels(data):
    with open("auto_evolution.ttl", "a") as auto_evolution:
        for fuel in data:
            fuel_aux = '''
### http://www.semanticweb.org/rafaelsilva/ontologies/2020/auto-evolution#{0}
:{0} rdf:type owl:NamedIndividual ,
```

```
                    :Fuels ;
            :nameFuel "{0}"^^xsd:string .
''').format(fuel['fuel'])
            auto_evolution.write(fuel_aux)
            auto_evolution.write('\n')
        auto_evolution.close()
```

Outro exemplo do povoamento da classe **Models**:

```
def populate_model(data):
    with open("auto_evolution.ttl", "a") as auto_evolution:
        for model in data:
            brand_name = model['nameBrand']
            ...
            model_aux = '''
###  http://www.semanticweb.org/rafaelsilva/ontologies/2020/auto-evolution#{0}
:{0} rdf:type owl:NamedIndividual ,
            :Models ;
    :pertenceClass :{1} ;
    :pertenceMarca :{2} ;
    :usaCombustivel {3}
    :imgModel "{4}"^^xsd:string ;
    :modelYears "{5}"^^xsd:string ;
    :nameModel "{0}"^^xsd:string ;
    :nrGenerationsModel "{6}"^^xsd:string ;
    :urlModel "{7}"^^xsd:string .
''').format(model_name,model_class,brand_name,fuels_aux,model_img,modelYears,
    model_nrGenerations,model_url)
        ...
```

Verificamos pelo excerto de código que as relações já estão a ser efectuadas com a respectiva informação.

Respectivamente as outras classes foi feito de maneira semelhante em que obtínhamos a informação necessária para povoar a respectiva classe e graças as nossas relações inversas no final do povoamento a própria ontologia ligando a ferramenta de *Reasoner* (ferramenta responsável por inferir relações) fará as próprias inferências pela totalidade da ontologia.

Capítulo 3

Extra

3.1 GraphDB

Por fim, foi efectuado um passo extra neste projecto que foi carregar a ontologia na ferramenta *GraphDB*, para confirmar que a ontologia não tinha nenhum erro semântico e que estava tudo correcto com a mesma.

Podemos visualizar as seguintes figuras da mesma ontologia no *GraphDB*.

Class	Links
:Models	60K
:Brand	29K
:Fuels	11K
:Classes	8K
:Versions	6K

Figura 3.1: Número de Relações da Ontologia no GraphDB

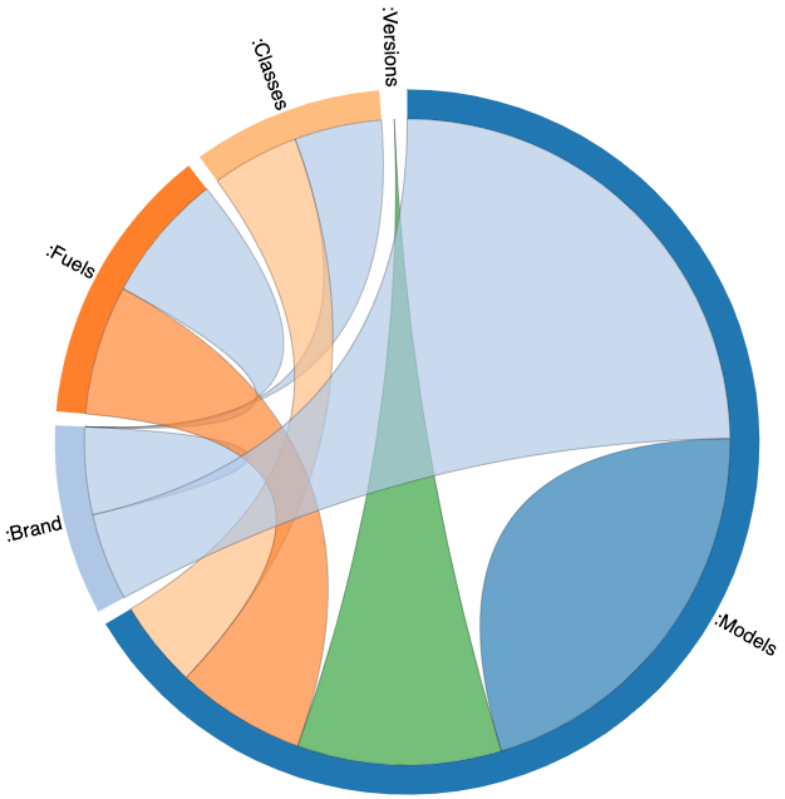

Figura 3.2: Gráfico com as relações e classes da Ontologia no GraphDB

Tirando no final o número total de cada classe:

```
Num Fuels: 10
Num Classes: 8
Num Brands: 109
Num Models: 2581
Num Versions: 6279
```