Pre-Entrega
Sistemas de Recuperación de Información

Universidad de la Habana

Integrantes

- Leismael Sosa Hernández
- Alejandro Yero Valdes

Resumen

El presente trabajo corresponde con la pre-entrega del proyecto final de la asignatura de Sistemas de Recuperación, y se tiene como objetivo implementar un Modelo de Recuperación a elección personal indexando 1 de las 3 colecciones de pruebas ofrecidas por el colectivo de profesores.
Nuestro equipo se decidió por implementar el Modelo Vectorial Clásico dado en conferencias usando como *corpus* la colección **Cran** que contiene 1400 documentos y en el resto del documento detallaremos las decisiones de implementación tomadas hasta ahora

Como ejecutar el proyecto

Nosotros consideramos que deberíamos comenzar por aquí el informe debido a que antes de restarles tiempo leyendo lo próximo sería interesante que realmente vean que esto está funcional.

Para ejecutar el proyecto es necesario tener instalado las siguientes librerías de python:

- nltk

Con `nltk` necesitarán descargar unos datasets que cuando intenten ejecutar el proyecto los **errores** les dirán como instalarlos. En un futuro esto lo plasmaremos en el informe

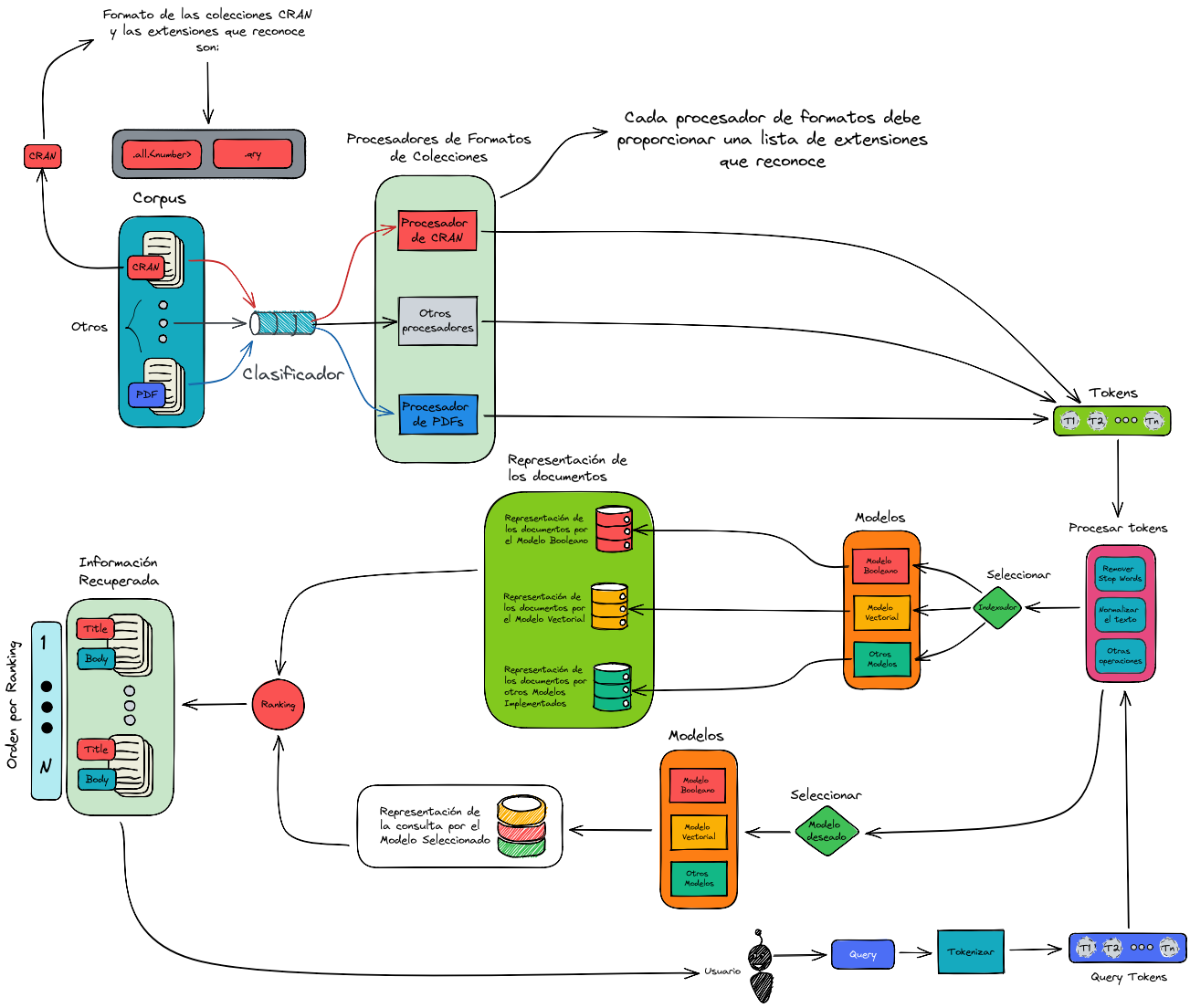
Entonces, la forma de ejecutar el proyecto es la siguiente (dentro de la raíz del proyecto):

```
python cli.py
```

Arquitectura actual del proyecto

La arquitectura actual del proyecto está pensada con el objetivo de poder implementar de forma sencilla los 3 Modelos de Recuperación de Información que se piden como requisito para la entrega final. Por esto tuvimos que pensar en una buena jerarquía de clases para lograr la flexibilidad deseada

La siguiente imagen, creada por nosotros, muestra la idea general del flujo de nuestro proyecto:



Architecture

Siguiendo la imagen anterior, el flujo de nuestro programa es:

- El usuario selecciona un archivo(s) que represente la colección deseada para indexar junto con el *Parser* que se quiere que las procese. Ejemplo: Colección *Cran* con el *Parser* de *CranParser*
- El *Parser* devuelve una colección de documentos en un formato estándar en nuestro sistema (clase Document). Estos documentos una vez creados procesan el texto usando la librería *nltk* para poder remover *stopwords*, realizar el proceso de *Stemming*, transformar las palabras a minúsculas, entre otras operaciones.
- Estos documentos son enviados al *Modelo de Recuperación* que el usuario desee para que puedan ser indexados por el mismo
- Luego de indexados es que el usuario puede comenzar a escribir *queries*
- Una vez el usuario escribe una *query*, el sistema se encarga de enviársela al *Modelo* seleccionado para que procese la *query* y devuelva la colección de documentos más relevantes, ordenados de mayor a menor. Este orden está dado por la función de *Ranking* asociada a cada *Modelo*

Jerarquía de Clases

Class IRS

Nosotros creamos la clase *IRS* (por Information Retrieval System) para manejar la lógica general del sistema. Su definición simplificada es la siguiente:

```
class IRS:
    """This class has the purpose of orchestrate the IRS process.
    """

    def __init__(self, storage: Storage):
        self.storage = storage
        self.models: dict[str, Model] = {}
        self.parsers: dict[str, Parser] = {}

    def add_model(self, model: Model):
        """This method add a model to the IRS"""

    def list_models(self) -> list[str]:
        """Returns the name of the models loaded for the system"""

    def add_parser(self, parser:Parser):
        """Add a new Parser to the IRS. Note that two different parsers with the
same 'pretty_name' will be recognized as the same"""

    def list_parsers(self) -> list[str]:
        """Returns all the names of the parsers loaded in the system"""

    def add_document(self, document:Document):
        """Add a document to the Index"""

    def add_document_collection(self, path:str, parser_name:str,
model_name:str):
        """This add a collection of documents to the Index of a Model"""

    def get_ranking(self, query:str, model_name:str, first_n_results:int):
        """This function returns the 'first_n_results' relevants documents for
the given query"""

    def save(self):
        """This method save all indexes created for each model in the system"""
```

Class Model

Model es la clase base para todos los Modelos de Recuperación que se quieran implementar en el sistema. Todos tienen que seguir su definición

```
class Model:

    def get_model_name(self) -> str:
        """This returns the name of a model in a pretty format"""

    def add_document(self, document: Document):
        """This method index the document according to an specific
implementation of a model"""

    def get_ranking(self, query:str, first_n_results:int, lang:str = 'english')
-> list[Document]:
        """Given a query this method returns the first n more relevant
results"""
```

Class Document

La clase **Document** representa el formato que los modelos saben como indexar.

```
class Document:
    """This is a Document object containing the original Document and the
normalized version of it.
    """
    def __init__(self, doc_id: int, doc_name: str, doc_body: str,
text_processor: Callable[[str,str],list[str]], doc_lang: str):
        self.doc_id = doc_id
        self.doc_name = doc_name
        self.doc_body = doc_body
        self.doc_lang = doc_lang
        self.text_processor = text_processor
        self.doc_normalized_name = text_processor(doc_name, doc_lang)
        self.doc_normalized_body = text_processor(doc_body, doc_lang)

    def get_doc_id(self):
        return self.doc_id

    def get_doc_name(self):
        return self.doc_name

    def get_doc_content(self):
        return self.doc_content

    def set_doc_id(self, doc_id):
        self.doc_id = doc_id

    def set_doc_name(self, doc_name):
        self.doc_name = doc_name

    def set_doc_content(self, doc_content):
        self.doc_content = doc_content
```

```
    def __str__(self):
        return f'doc_id: {self.doc_id}, doc_title: {self.doc_name}'
```

Se puede apreciar que esta clase solo guarda el titulo de un documento, su cuerpo y su versión normalizada

Class Parser

La clase **Parser** se encarga de parsear los documentos que representen la colección a indexar y devuelve una lista de documentos en el formato estándar (class Document)

```
class Parser:

    def __init__(self, text_processor: Callable[[str, str], list[str]], lang:str
= 'english'):
        self.text_processor = text_processor
        """This is the function in charge of tokenize the text and return it in
a normalized form"""
        self.lang = lang

    def get_pretty_name(self) -> str:
        """Returns the pretty name of this parser

        Returns:
            str: The pretty name of the parser
        """

    def get_extension_list(self) -> list[str]:
        """Returns the list of the formats this parser handles
        Returns:
            list[str]: A list with all the formats. Each element has the form
'ext' not '.ext'
        """

    def parse(self, file: TextIOWrapper):
        """This method receives a file and parse it's contents returning a list
of documents

        Args:
            file (_type_): _description_
        Returns:
            list[Document]: A list with the normalized documents
        """
```

Class Storage

Esta clase se encarga de salvar (serializar) y cargar (deserializar) la información indexada por los modelos de recuperación y de otras funciones deseadas por la clase IRS

```
class Storage:
    """Class for handling all storage processes of an IRS
    """

    def __init__(self, path_to_index='./irs_data'):
        """Initialize a storage object looking for info in a root path

        Args:
            path_to_index (str, optional): This is the root directory where all
the IRS models will store there info. Defaults to './irs_data'.
        """
        self.path_to_index = path_to_index

    def get_storage_path(self, name: str) -> tuple[bool,str]:
        """This method returns the path to a specific storage. Useful for IRS
models
        INFO: This method is for future optimizations in the storage process.
Not for now

        Args:
            name (str): The name of the storage. In the case of an IRS model is
the model name

        Returns:
            str: The path to the storage
        """

    def create_storage_path(self, name: str):
        """This method create a new storage directory with the given name.
        INFO: This method is for future optimizations in the storage process.
Not for now

        Args:
            name (str): Name of the new storage path
        """

    def save_model(self, name: str, obj: Model):
        """This method saves an object in a specific storage

        Args:
            name (str): The name of the storage
            obj (object): The object to be saved
        """

    def load_model(self, name: str) -> Model:
        """This method loads an object from a specific storage
```

```
        Args:
            name (str): The name of the storage

        Returns:
            object: The object loaded from the storage
        """

    def model_exists(self, name: str) -> bool:
        """This method checks if a model exists in the storage

        Args:
            name (str): The name of the model

        Returns:
            bool: True if the model exists, False otherwise
        """
```

Modelo Vectorial Clásico

Nuestra implementación del Modelo Vectorial Clásico es exactamente como se describe en las conferencias. No hay mucho interesante de lo que hablar en nuestra implementación, salvo quizás, que en vez de crear los vectores de los documentos de dimensión n (donde n es la cardinalidad del vocabulario de términos indexados por el sistema), pues solo se crearon de la misma dimensión que la cantidad de términos que presenta el documento (para ahorrar memoria). Por lo anterior, se tuvo que definir una función de similitud que tiene en cuenta esta optimización, pero que al final hace lo que tiene que hacer

Futuras mejoras

Por ahora se tiene cargado en la memoria del Modelo Vectorial (en memoria RAM) los documentos que presenta, así como el diccionario de `tf`, el del `idf` y el vector de cada documento. El tener los documentos cargados en la memoria del sistema presenta un costo extra que se puede remover cambiando algunos detalles de la implementación presente, pero por falta de tiempo se dejará para la entrega final

Conclusiones

Nuestro equipo considera que la jerarquía de clases presente en nuestra implementación es lo suficientemente flexible como para mantenerla prácticamente intacta a lo largo de las próximas iteraciones de cara a la entrega final. De ahí que los cambios que hagamos serán más destinados a optimizar lo ya hecho, a desarrollar los restantes modelos y a darle soporte a características extras como **retroalimentación**, **clustering** y demás.