Resumen CBRkit

CBRkit es un toolkit modular en Python para desarrollar aplicaciones aplicando Razonamiento Basado en Casos.

Actualmente se encuentra en desarrollo y en su versión actual (0.28.5) permite:

- cargar casos
- definir medidas de similaridad y realizar la recuperación sobre bases de casos en memoria
- definir funciones de adaptación para utilizarlas en la fase de reutilización.

Más detalles:

- Código y descarga
- Documentación oficial
- Paper y video
- Tutorial (en español)

Los siguientes módulos son parte de la versión actual de CBRkit:

- **cbrkit.loaders**: funciones para cargar casos y consultas.
- **cbrkit.sim**: funciones de similaridad para tipos de datos comunes: cadenas, números, colecciones, etc.
- **cbrkit.retrieval**: utilidades para definir y aplicar pipelines de recuperación.
- **cbrkit.adapt**: funciones de adaptación de casos para tipos de datos comunes: cadenas, números,

colecciones, etc.

- **cbrkit.reuse**: utilidades para definir y aplicar pipelines de reutilización de casos.
- **Otros**
  - **cbrkit.typing**: definiciones de tipos genéricos para definir funciones personalizadas.
  - **cbrkit.eval**: métricas de rendimiento
  - **cbrkit.cycle**: implementación del ciclo básico de los sistemas CBR
  - **cbrkit.synthesis**: integración con LLMs para generación de ''casos sintéticos''

---

1. Carga de casos (módulo cbrkit.loaders)

La librería CBRkit incluye un módulo de cargadores que permite leer datos de diferentes formatos de archivo y convertirlos en una "base de casos" (*Casebase*).

- Los mismos métodos pueden emplearse para cargar "consultas" (casos a resolver).
- El tipo de dato *Casebase* encapsula y unifica la gestión de las bases de casos que pueden cargarse desde diferentes formatos (JSON, YAML CVS, XML, Pandas, Polars, ...)
- Adicionalmente, en la carga de "bases de conocimiento" es posible forzar la validación de tipo de los casos cargados utilizando definiciones de la libreria Pydantic

1.1 Funciones de carga de "bases de casos"

- `cbrkit.loaders.file(path)`: Convierte un archivo (CSV, JSON, XML, YAML, etc) en un objeto *Casebase* "base de casos".

  Delega en métodos específicos la carga de casos desde estos formatos
  - `cbrkit.loaders.csv(path)`: Lee un archivo CSV y lo convierte en un diccionario.
  - `cbrkit.loaders.json(path)`: Lee un archivo JSON y lo convierte en un diccionario.
  - `cbrkit.loaders.yaml(path)`: Lee un archivo YAML y lo convierte en un diccionario.
  - `cbrkit.loaders.xml(path)`: Lee un archivo XML y lo convierte en un diccionario.
  - `cbrkit.loaders.pandas(DataFrame)`: Encapsula un *DataFrame* de Pandas ofreciendo un interfaz equivalente al de un diccionario
  - `cbrkit.loaders.polars(DataFrame)`: Encapsula un *DataFrame* de Polars ofreciendo un interfaz equivalente al de un diccionario
- `cbrkit.loaders.directory(path, pattern)`: Convierte todos los archivos en una carpeta que coincidan con un patrón específico en un *Casebase*.
- `cbrkit.loaders.validate(data, validation_model)`: Valida los datos (diccionario de casos) contra un modelo de Pydantic, arrojando errores si los datos no coinciden con el modelo esperado.

Las mismas funciones que se utilizan para cargar las "bases de casos" se pueden utilizar para cargar el caso o casos "a resolver" por el sistema CBR.

- La única restricción es que el tipos de los "casos a resolver" (casos *query*) que utilicen una "base de casos" debe de ser compatible con el tipo de los casos almacenados en la misma.

En el caso de la carga desde archivos JSON que se emplea en el ejemplo de la tasación, se espera que el fichero de entrada contenga una lista/array de objetos JSON o un diccionario (tabla hash) de objetos JSON.

- En ambos caso la "base de casos" cargada será un diccionario Python con los casos.
- En el caso de que el fichero de entrada contuviera lista de objetos se usará como clave e identificador del caso su índice en la lista/array.
- Si el fichero JSON de entrada ya era un diccionario/tabla hash se mantienen las claves utilizadas en el mismo.

---

2. Fase "Recuperar" (módulo cbrkit.retrieval)

El módulo **cbrkit.retrieval** ofrece clases y funciones de utilidad para dar soporte a la **recuperación de casos** basada en métricas de similaridad y un conjunto de tipos de datos para encapsular los resultados de una búsqueda por similaridad en la base de casos

2.1 Construcción de recuperadores (`cbrkit.retrieval.build()`)

- La función de utilidad `cbrkit.retrieval.build(similarity_func)` permite **crear funciones de recuperación** personalizadas basadas en una función de similaridad.
    - El valor de retorno es una "función de recuperación" (`cbrkit.typing.RetrieverFunc`) que puede ser utilizada por las funciones `apply_query()` o `apply_queries()` para consultar una "base de casos

2.2 Filtrado de recuperadores (`cbrkit.retrieval.dropout()`)

- La función de utilidad `cbrkit.retrieval.dropout(retriever_func, limit, min_similarity, max_similarity)` permite **filtrar** la salida de las funciones de recuperación.
    - Se pueden establecer límites en el número de casos devueltos y filtrar por similaridad mínima y máxima.
    - El valor de retorno es de nuevo una "función de recuperación" (`cbrkit.typing.RetrieverFunc`) que puede ser utilizada por las funciones `apply()` o `mapply()` para consultar una "base de casos"

2.3 Recuperación de casos similares (`cbrkit.retrieval.apply_query()` y `cbrkit.retrieval.apply_queries()`)

- La función de utilidad `cbrkit.retrieval.apply_query(casebase, query, retriever/s)` lanza un *"caso consulta"* (caso a resolver) `query` contra una "base de casos" `casebase` aplicando las métricas de similaridad de uno o varios `retriever` proporcionados por la función `build()` para **recuperar** los **casos más similares**.
- La función de utilidad `cbrkit.retrieval.apply_queries(casebase, queries, retriever/s)` es una generalziación de la anterior que permite lanzar múltiples *"casos consulta"* (*queries*) a una base de casos.

2.4 Resultados de la recuperación

- La clase `Result`, devuelta por `apply_query()` y `apply_queries()` ,encapsula los resultados de la recuperación.
  En concreto ofrece:
  - atributo `ranking`: lista ordenada con los `ids` (claves) de los *n* casos más similares
  - atributo `similarities`: lista ordenada de *n* valores de tipo `float` con los valores de similaridad de los *n* casos más similares
  - atributo `casebase`: base casos (~ diccionario) con los *n* casos más similares
  - atributo `steps`: lista ordenada de objetos `ResultStep` con la información de los resultados intermedios en el caso de utilizar multiples *Retrievers*
- La clase `ResultStep` representa la información de cada paso en el proceso de recuperación (para los casos con varias *RetrieverFunc*), incluyendo similaridades y rankings.

---

3. Métricas de similaridad (módulo cbrkit.sim)

El módulo **cbrkit.sim** (y sus submódulos) incluye un conjunto de métricas de similaridad para distintos tipos de datos, como números, cadenas de texto, listas y datos genéricos.

- Proporciona una implementación de diversos tipos de métricas específicas, que junto con funciones de similaridad definidas por el programador, pueden ser utilizadas para configurar los `Retrievers` creados con la función `cbrkit.retrieval.build()`
- Ofrece la función de utilidad **`cbrkit.sim.attribute_value()`** que permite definir una métrica de similaridad compleja que combine un conjunto de funciones de similaridad específicas para cada atributo de los casos procesados.
- Define la función de utilidad `cbrkit.sim.aggregator()` que permite especificar el tipo de combinación de métricas (media, máximo, etc) y su ponderación

3.1 Especificación de métricas de atributos (`cbrkit.sim.attribute_value()`)

La función de utilidad `attribute_value(attributes, aggregator, value_getter, default)` permite definir una **métrica de similaridad compuesta** que calcule la similaridad entre dos casos a partir de la combinación de valores de similaridad entre pares de atributos.

- En el parámetro `attributes` se vincula a cada *nombre de atributo* del caso con la *función de similaridad* a utilizar al comparar sus valores
  - Las funciones de similaridad pueden ser las propocionadas en los submódulos `cbrkit.sim.*` o funciones de similaridad específicas definidas por el programador
- En el parámetro `aggregator` se especifica el método de combinación de métricas a utilizar (`'mean', 'fmean', 'geometric_mean', 'harmonic_mean', 'median', 'median_low', 'median_high', 'mode', 'min', 'max', 'sum'`) y, opcionalmente, los pesos con los que se combinan las métricas de atributos

3.2 Funciones de similaridad para atributos numéricos (`cbrkit.sim.numbers`)

El módulo **cbrkit.sim.numbers** proporciona varias funciones de similaridad para valores numéricos

- `linear_interval(min, max)`: Calcula la similaridad lineal entre dos valores dentro de un intervalo definido por un mínimo y un máximo. La similaridad se basa en la distancia relativa de los valores dentro de este rango.
- `linear(max[, min])`: Similar a `linear_interval`, pero no se limita a un rango específico. Define un mínimo y un máximo, donde la similaridad es 1.0 en el mínimo y 0.0 en el máximo.
- `threshold(umbral)`: Devuelve una similaridad de 1.0 si la diferencia absoluta entre dos valores es menor o igual a un umbral definido; de lo contrario, devuelve 0.0.
- `exponential(alpha)`: Utiliza una función exponencial para calcular la similaridad, controlada por un parámetro *alpha*. Un valor de alpha más alto provoca una disminución más rápida de la similaridad.
- `sigmoid(alpha, theta)`: Implementa una función sigmoide para calcular la similaridad, donde *alpha* controla la pendiente de la curva y *theta* determina el punto en el que la similaridad es 0.5.

3.3 Funciones de similaridad para colecciones (`cbrkit.sim.collections`)

El módulo **cbrkit.sim.collections** proporciona varias funciones para calcular la similaridad entre atributos que almacenan colecciones y secuencias

- `jaccard()`: Calcula la similaridad de Jaccard entre dos colecciones, midiendo la razón entre las cardinalidades de la intersección (elementos comunes) sobre la unión (elementos totales).
- `isolated_mapping(element_similarity)`: Compara cada elemento de una secuencia *query* (y) con todos los elementos de la otra (x).
    - Para esa comparación entre elementos utiliza la función de similaridad proporcionada (`element_similarity`) quedándose con el máximo de similaridad para cada elemento de la secuencia *query* (y).
    - La salida es la media de estas "mejores" similaridades parciales
- `mapping(similarity_function, max_queue_size)`: Implementa un algoritmo A* para encontrar la mejor coincidencia entre elementos basándose en la función de similaridad proporcionada
- `sequence_mapping(element_similarity, exact)`: Asume secuencias ordenadas. Calcula la similaridad entre dos secuencias utilizando la función de similaridad proporcionada (`element_similarity`) para comparar sus elementos, posición a posición.
- `sequence_correctness(worst_case_sim)`: Asumen secuencias ordenadas. Evalúa la similaridad de dos secuencias comparando sus elementos, otorgando el valor `worst_case_sim` cuando todos los pares son discordantes y valores proporcionales cuando hay algunas correspondencias
- `smith_waterman(match_score, mismatch_penalty, gap_penalty)`: Realiza un alineamiento de Smith-Waterman, permitiendo ajustar los parámetros de puntuación
- `dtw()` (*Dynamic Time Warping*): Calcula la similaridad entre secuencias que pueden variar en longitud.

3.4 Funciones de similaridad para atributos de tipo String basadas en caracteres (`cbrkit.sim.strings`)

El módulo **cbrkit.sim.strings** ofrece varias funciones para calcular la similaridad entre cadenas de texto a nivel de caracteres

- `ngram(n, case_sensitive, tokenizer)`: Mide la similaridad en base a la coincidencia de n-gramas de caracteres entre las dos cadenas
- `levenshtein(score_cutoff, case_sensitive)`: Calcula la similaridad normalizada entre dos cadenas basándose en la distancia de Levenshtein o distancia de edición (número de inserciones, eliminaciones y sustituciones) sobre sus respectivos caracteres. Puede especificarse un umbral y la diferenciación entre mayúsculas y minúsculas.
- `jaro(score_cutoff)`: Calcula la similaridad entre dos cadenas utilizando el algoritmo de Jaro.
- `jaro_winkler(score_cutoff, prefix_weight)`: Variante del algoritmo de Jaro que tiene en cuenta los prefijos comunes entre las cadenas

3.5 Funciones de similaridad para atributos de tipo String basadas en *embeddings* (`cbrkit.sim.embed`)

El módulo **cbrkit.sim.embed** ofrece varias funciones para calcular la similaridad "semántica" entre cadenas de texto, basándose en la comparación de **representaciones vectoriales**.

- Ofrece adaptadores para utilizar diversos **modelos del lenguaje** para convertir las cadenas en "vectores semánticos" (*embeddings*) [tanto a nivel de palabra (*word embeddings*) como de frase (*sentence embeddings*)]
- Ofrece diversas **métricas de comparación** de representaciones **vectoriales** de cadenas.

3.5.1 Función de utilidad `build()`

La función de utilidad `cbrkit.sim.embed.build(conversion_func, sim_func, query_conversion_func)` permite especificar:

1. Los modelos de *embeddings* a usar para transformar las cadenas en vectores semánticos (parámetros `conversion_func`, para el procesamiento de texto en los casos, y `query_conversion_func`, para el procesamiento de texto de las *queries*)
2. La métrica de similaridad entre vectores a utilizar (parámetro `sim_func`).

Ejemplo:

```
similaridad = cbrkit.sim.embed.build(
    conversion_func=cbrkit.sim.embed.sentence_transformers(model="all-MiniLM-L6-v2"),
    sim_func=cbrkit.sim.embed.cosine()
    )
```

Crea una métrica de similaridad entre cadenas usando el modelo de *embeddings* preentrenado `all-MiniLM-L6-v2` (vectores de 384 elementos, con entrada de hasta 256 tokens) de Sentence Tranformers (tanto para casos como para *queries*).
La métrica de comparación entre vectores individuaes es la medida del coseno.

3.5.2 Modelos del lenguaje

- `spacy(model_name)`: Utiliza un modelo de la librería NLP spaCy para calcular la similaridad semántica entre pares de textos mediante vectores de palabras
    - Para la similaridad entre frases, spaCy calcula similaridades palabra-a-palabra (*word embeddings*) y los agrega calculando la media
    - Se ofrece la función de utilidad `load_spacy()` para cargar los modelos disponibles (ver modelos sPacy)
- `sentence_transformers(model_name)`: Usa un modelo preentrenado de la librería Sentence Transformers y calcula la similaridad semántica entre textos usando vectores de palabras (ver modelos Sentence Transformers)
- `openai(model_name)`: Utiliza los modelos de *embeddings* disponibles en el API de OpenAI (requiere una *API key*) para calcular la similaridad semántica entre pares de textos
- Otros: `ollama()`, `cohere()`, `voyageai()`

3.5.3 Métricas de similaridad entre vectores

- `cosine`: coseno entre dos vectores
- `dot`: producto escalar de dos vectores
- `angular`: ángulo entre vectores
- `euclidean`: distancia euclídea entre vectores
- `manhatan`: ditancia manhatan entre vectores.

3.5.4 Caché de vectores (*embeddings*)

El cálculo de *embeddings* suele ser muy costoso. Para evitar reptición de cálculos entre sucesivas ejecuciones del ciclo CBR, la librería *CBRKit* permite almacenar los *embeddings* ya calculados en una caché en disco (base de datos SQLite).

Se ofrece la función de utilidad `cbrkit.sim.embed.cache(func, path, table)` para configurar un "decorador" sobre una función de vectorización data (`func`) y especificar la ruta a la BD SQLite y la tabla donde se almacenarán los vectores "cacheados".

- La función de vectorización resultante puede usarse en `cbrkit.sim.embed.build()` para crear la métrica de similaridad entre cadenas final.

Ejemplo:

```
embedding_con_cache = cbrkit.sim.embed.cache(
    func=cbrkit.sim.embed.sentence_transformers(model="all-MiniLM-L6-v2"),
    path="embeddings_cache.npz"
    )

similaridad_con_cache = cbrkit.sim.embed.build(
    conversion_func=embedding_con_cache,
    sim_func=cbrkit.sim.embed.cosine()
    )
```

3.6 Funciones de similaridad para atributos organizados en Taxonomias (`cbrkit.sim.taxonomy`)

El módulo **cbrkit.sim.taxonomy** permite trabajar con **taxonomías**, que son estructuras jerárquicas de categorías.

- Cada categoría se representa como un nodo con atributos como nombre, peso y posibles hijos
- Las clases `Taxonomy` y `TaxonomyNode` del módulo se usan para representar la organización jerárquica de las categorías de la Taxonomía
- Se dispone de la función/clase de utilidad `load()` para cargar los elementos de una Taxonomía desde ficheros en formato JSON, YAML o TOML y de métricas de similaridad que explotan las relaciones jerárquicas de los nodos

Ejemplo: Taxonomia `paint_color.yaml` (codificada en YAML)

```yaml
name: color
children:
  - name: dark
    children:
      - name: brownish
        children:
          - name: brown
          - name: beige
          - name: bronze
      - name: others
        children:
          - name: purple
          - name: black
          - name: grey
          - name: violet
  - name: light
    children:
      - name: colorful
        children:
          - name: red
          - name: yellow
          - name: orange
          - name: gold
          - name: blue
      - name: others_2
        children:
          - name: custom
          - name: white
          - name: silver
          - name: green
```

3.6.1 Carga de Taxonomías y función de similaridad (`cbrkit.sim.taxonomy.build()`)

La función/clase de utilidad `cbrkit.sim.taxonomy.load(taxonomy, func)` carga una taxonomía desde un archivo (en formato JSON, YAML o TOML) y le vincula una métrica de similaridad sobre taxonomías para devolver una función para medir la similaridad entre categorias.

Las **métricas de similaridad** sobre taxonomias disponibles son:

- `wu_palmer()`: Calcula la similaridad entre dos nodos de la taxonomía utilizando el método de Wu & Palmer basado en la profundidad de los nodos y la de su ancestro común más cercano (LCA)
- `weights(source, strategy)`: Calcula la similaridad entre nodos basándose en pesos vinculados a cada nodo (definidos por el usuario en el archivo YAML de la taxonomía [`user`] o calculados en base a la profundidad [`auto`])
- `levels(strategy)`: Mide la similaridad entre nodos según su nivel en la jerarquía
- `paths(weight_up, weight_down)`: Mide la similaridad basándose en los pasos hacia arriba y hacia abajo desde el ancestro común más cercano (LCA)

3.7 Funciones de similaridad genéricas (`cbrkit.sim.generic`)

El módulo **cbrkit.sim.generic** ofrece funciones de similaridad que no están limitadas a tipos de datos específicos

- `static_table(entries, default, symmetric)`: Permite asignar valores de similaridad desde una tabla definida por una lista de entradas (`entries`) que relacionan pares de valores con su similaridad.
    - Se puede definir si la tabla es simétrica y establecer un valor de similaridad predeterminado para pares que no estén en la tabla.
- `equality()`: Devuelve una similaridad de 1.0 sólo si los dos valores son iguales y 0.0 si son diferentes.

---

4. Fase "Reutilizar" (módulos `cbrkit.adapt` y `cbrkit.reuse`)

Las versiones recientes de CBRkit incluyen componentes para dar soporte a la fase **Reutilizar** del ciclo CBR.
La organización es similar a cómo se plantea la fase **Recuperar** en CBRkit: funciones de similaridad + definición del ''recuperardor''.
En este caso se separan las funciones de adaptación individuales de la definición del ''pipeline'' de reutilización a emplear.

- El módulo `cbrkit.adapt` ofrece una colección de funciones para describir cómo se adapta un caso al nuevo problema.
    - Cuenta con funciones de adaptación específicas para cada tipo de datos: `cbrkit.adapt.string`, `cbrkit.adapt.numbers`, `cbrkit.adapt.generic`
    - Ofrece una función de utilidad `attribute_value(attibutes)` para agregar un conjunto de funciones de adaptación individuales, indicando cómo aplicarlas sobre los atributos de un caso.

- El módulo `cbrkit.reuse` permite definir cómo aplicar las funciones de adaptación a los resultados de la recuperación.
    - Ofrece una función de utilidad `build()` para defnir el "pipeline" de reutilización, especificando las funciones de adaptación a utilziar
    - Las funciones `apply_result()`, `apply_query()` y `apply_queries()` permiten aplicar las funciones de adaptación del "pipeline" indicado sobre una *query* (o conjunto de *queries*) y un caso recuperado (o un *Casebase* con un conjunto de casos recuerados).
    - Las clases `Result`, `ResultStep` y `QueryResultStep` permiten recuperar los nuevos casos resultantes de aplicar las adaptaciones indicadas en el "pipeline" de reutilización.

NOTA: En el ejemplo de uso de CBRKit (`tasador-25`) y en el entregable de la Práctica 2 no se utiliza el soporte ofrecido por CBRkit para la fase de Recuperación, implementándose manualmente las correspondientes adaptaciones.