

# **PRÁCTICA 5**

## **APLICACIÓN PARA GESTIÓN DE UNA LIBRERÍA**

Equipo:

Beltrán Saucedo Axel Alejandro

Cerón Samperio Lizeth Montserrat

Higuera Pineda Angel Abraham

Lorenzo Silva Abad Rey

4BV1.

ESCUELA SUPERIOR DE  
CÓMPUTO

**TECNOLOGÍAS PARA EL DESARROLLO DE APLICACIONES  
WEB**

# **Índice**

<b>1. Introducción</b>	<b>2</b>
<b>2. Fundamentos Teóricos</b>	<b>2</b>
<b>3. Diagrama UML</b>	<b>7</b>
<b>4. Implementación</b>	<b>8</b>
<b>Referencias</b>	<b>14</b>

# 1. Introducción

## Planteamiento del problema

Se pretende crear una API centrada en la gestion del catalogo de una libreria, cuyo principal objetivo sera proporcionar un sistema que permita la administracion de informacion de los libros y sus autores, manejando las relaciones que hay entre ellos. Nuestro sistema debe ser capaz de lo siguiente:

- Permitir la creacion y consulta de autores en la base de datos.
- Permitir la creacion de nuevos libros y poder asociarlos a uno o varios autores.
- Implementar de forma acertada las relaciones muchos a mucho entre libros y autores.
- Administrar de forma eficiente la propia informacion de un libro, como lo es su editorial, categorias, publico objetivo, etc.
- Ofrecer endpoints para poder consultar libros basados en la filtracion de diferentes criterios.

## Propuesta de solución

La solución propuesta es el desarrollo de una API con el uso de FastAPI, la cual usara SQLModel para gestionar la persistencia de datos. Los componentes clave para lograr nuestra solucion son:

- **SQLModel:** Para definir los modelos de datos principales como lo son las tablas SQL, ademas de una tabla de enlaces que gestionara la relacion muchos a muchos.
- **FastAPI:** Para construir la API y definir los endpoints en el modulo rutas.py, ademas de manejar las solicitudes HTTP.
- **Arquitectura modular:** Se refiere a la estructura que tendra la practica, dividiendola en los modulos: modelo.py, esquemas.py, servicios.py, rutas.py y database.py

# 2. Fundamentos Teóricos

Para la correcta creacion de nuestra práctica, es importante que entendamos las relaciones entre las tecnologías y los conceptos que ya revisados para la creación de una API, esta vez implementando persistencia de datos.

## Entorno de Trabajo y Herramientas Principales

- **Python:** Es el lenguaje de programación sobre el que se construye toda la lógica de la aplicación. Su sencilla sintaxis y su amplia cantidad de librerías lo hacen ideal para el desarrollo web.[2]
- **FastAPI:** Framework web moderno para construir APIs con Python. Sus características más destacadas son la rapidez, la validación de datos automática mediante Pydantic y la generación de documentación interactiva , que fue crucial para probar los endpoints de nuestra API.
- **Uvicorn:** Es un servidor ASGI ultrarrápido, utilizado para ejecutar la aplicación FastAPI. Permite que la API maneje múltiples peticiones de forma asíncrona, mejorando el rendimiento.[3]

## Persistencia de Datos

La persistencia de datos es la capacidad de un sistema para poder conservar la información posterior de la duración de una sola ejecución. En nuestra práctica pasada, los datos eran volátiles, ya que se almacenaban en una lista en memoria. En esta práctica, se implementa la persistencia a través de los siguientes componentes:

- **SQLite:** Es un motor de base de datos relacional, autocontenido y que no requiere un servidor. Almacena toda la base de datos en un único archivo (en nuestro caso, database.db). Es ideal para desarrollo y aplicaciones de pequeña a mediana escala por su simplicidad y portabilidad.
- **SQLAlchemy:** Es una librería que funciona como un Mapeador Objeto-Relacional (ORM). Un ORM es una técnica que actúa como un "traductor.entre el código orientado a objetos y las tablas de una base de datos relacional. Permite manipular la base de datos utilizando código Python en lugar de escribir consultas SQL directamente.
- **SQLModel:** Es una librería que combina **SQLAlchemy** y **Pydantic**, creada por el mismo autor de FastAPI. Permite definir la estructura de los datos, las validaciones y el esquema de la base de datos en una sola clase, reduciendo la duplicación de código y simplificando el desarrollo. En nuestra práctica, las clases como Item son modelos de SQLModel que representan tanto la tabla en la base de datos como los datos que la API recibe y envía.[4]

## Relaciones en Bases de Datos

- **Relación Unívoca:** Cada valor de clave primaria se relaciona con sólo un registro en la tabla relacionada.
- **Uno a varios:** La tabla de claves primaria sólo contiene un registro que se relaciona con ninguno, uno o varios registros en la tabla relacionada.
- **Varios a varios:** Cada registro en ambas tablas puede estar relacionado con varios registros en la otra tabla. Este tipo de relaciones requieren una tercera tabla, denominada tabla de enlace o asociación, porque los sistemas relacionales no pueden alojar directamente la relación. [5]

## API CRUD

Una API CRUD es una interfaz de programación que permite Crear, Leer, Actualizar y Borrar datos. [6]

## Algoritmo Genético Simple (AGS)

Está basado en el proceso genetico de los organismos vivos. Los Algoritmos Genéticos usan una analogía directa con el comportamiento natural.

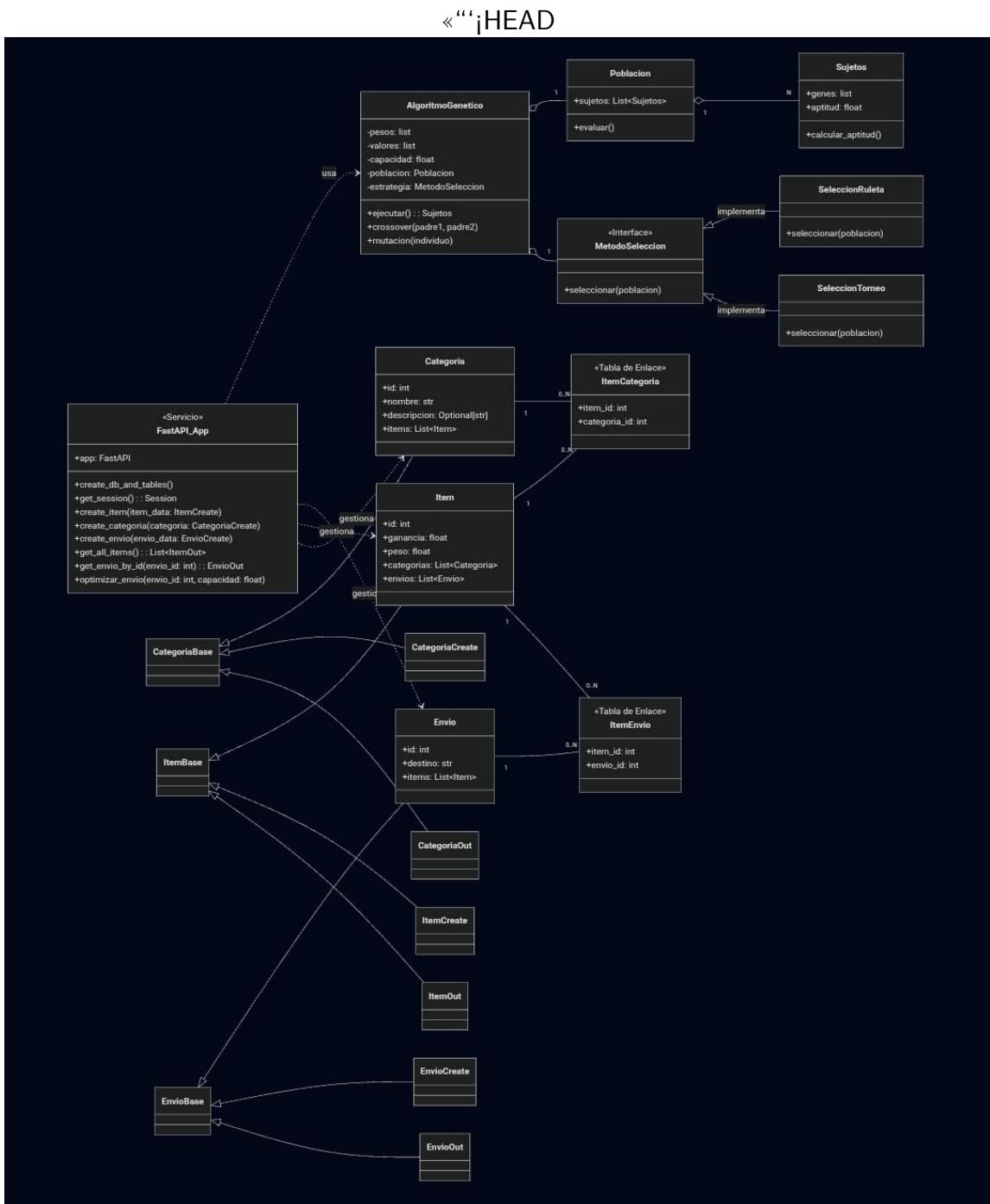
- **Población:** Trabajan con una poblacion de individuos, cada uno de los cuales representa una solucion factible a un problema dado. A cada individuo se le asigna un valor o puntuacion, relacionado con la bondad de dicha solucion.
- **Fitness:** Cuanto mayor sea la adaptacion de un individuo al problema, mayor sera la probabilidad de que el mismo sea seleccionado.
- **Cruzamiento:** Será seleccionado para reproducirse, cruzando su material genetico con otro individuo seleccionado de igual forma. Este cruce producira nuevos individuos descendientes de los anteriores los cuales comparten algunas de las caracteristicas de sus padres. Cuanto menor sea la adaptacion de un individuo, menor sera la probabilidad de que dicho individuo sea seleccionado para la reproduccion, y por tanto de que su material genetico se propague en sucesivas generaciones.[7]

De esta manera se produce una nueva poblacion de posibles soluciones, la cual reemplaza a la anterior y verifica la interesante propiedad de que contiene una mayor proporcion de buenas caracteristicas en comparacion con la poblacion anterior. Así a lo

largo de las generaciones las buenas caractersticas se propagan a traves de la poblacion. Favoreciendo el cruce de los individuos mejor adaptados, van siendo exploradas las areas mas prometedoras del espacio de busqueda. Si el Algoritmo Genetico ha sido bien disenado, la poblacion convergera hacia una solucion optima del problema.



### 3. Diagramma UML



»"»"»j7d864be4b5a5b04f7bcc156fd569caccca3a98e0

## 4. Implementación

### Código:

En esta sección, explicaremos las modificaciones y adiciones clave que transforman la API. El cambio fundamental es la introducción de **relaciones** en la base de datos, pasando de un solo modelo 'Item' a un sistema de tres modelos interconectados: Item, Categoria, y Envio.

«“¡HEAD

```
class ItemCategoria(SQLModel, table=True):
    """
    Tabla de enlace (asociativa) para la relación muchos-a-muchos
    entre Item y Categoria. Contiene las claves foráneas de ambas tablas.
    """

    item_id: Optional[int] = Field(
        default=None, foreign_key="item.id", primary_key=True # Parte de la clave primaria compuesta.
    )
    categoria_id: Optional[int] = Field(
        default=None, foreign_key="categoria.id", primary_key=True # Parte de la clave primaria compuesta.
    )
"""

class ItemEnvio(SQLModel, table=True):
    """
    Tabla de enlace (asociativa) para la relación muchos-a-muchos
    entre Item y Envio. Contiene las claves foráneas de ambas tablas.
    """

    item_id: Optional[int] = Field(
        default=None, foreign_key="item.id", primary_key=True # Parte de la clave primaria compuesta.
    )
    envio_id: Optional[int] = Field(
        default=None, foreign_key="envio.id", primary_key=True # Parte de la clave primaria compuesta.
    )
```

Figura 1: Definición de las tablas de enlace 'ItemCategoria' y 'ItemEnvio'.

»'''La Práctica 4 introduce relaciones de **muchos a muchos**. Un ítem puede pertenecer a múltiples categorías, y un ítem puede estar en múltiples envíos. Para lograr esto en una base de datos, se requieren "tablas de enlace".

- '■ **ItemCategoria**: Actúa como puente entre 'Item' y 'Categoria'. Contiene 'item\_id' y 'categoria\_id' como claves foráneas y juntas forman la clave primaria.
- '■ **ItemEnvio**: De forma similar, une 'Item' y 'Envio' usando 'item\_id' y 'envio\_id'.

```

class Categoria(CategoriaBase, table=True):
    """Modelo de la tabla 'categoria'. Incluye ID y la relación con Items."""
    id: Optional[int] = Field(default=None, primary_key=True) # Clave primaria autogenerada.

    # MODIFICADO: Relación muchos-a-muchos: Una categoría puede tener muchos 'Item'.
    # 'back_populates' conecta esta relación con el atributo 'categorias' (plural) en el modelo 'Item'.
    items: List["Item"] = Relationship(back_populates="categorias", link_model=ItemCategoria)
>"""

class Item(ItemBase, table=True):
    """Modelo de la tabla 'item'. Incluye ID y relaciones con Categoria y Envio."""
    id: Optional[int] = Field(default=None, primary_key=True) # Clave primaria autogenerada.

    # MODIFICADO: Se elimina la relación 'categoria' (muchos-a-uno) anterior.

    # --- NUEVO ---
    # Relación muchos-a-muchos: Un item puede tener muchas 'Categoria'.
    # 'link_model=ItemCategoria' especifica la tabla de enlace a usar.
    # 'back_populates' conecta con el atributo 'items' en 'Categoria'.
    categorias: List[Categoria] = Relationship(back_populates="items", link_model=ItemCategoria)

    # Relación muchos-a-muchos: Un item puede estar en muchos 'Envio'.
    # 'link_model=ItemEnvio' especifica la tabla de enlace a usar.
    # 'back_populates' conecta con el atributo 'envios' en 'Envio'.
    envios: List["Envio"] = Relationship(back_populates="envios", link_model=ItemEnvio)

class Envio(EnvioBase, table=True):
    """Modelo de la tabla 'envio'. Incluye ID y la relación con Items."""
    id: Optional[int] = Field(default=None, primary_key=True) # Clave primaria autogenerada.

    # Relación muchos-a-muchos: Un envío puede contener muchos 'Item'.
    items: List[Item] = Relationship(back_populates="envios", link_model=ItemEnvio)

```

Figura 2: Modelos de tabla Categoria, Item (modificado) y Envio.

»'''Con las tablas de enlace definidas, los modelos principales ahora usan el campo 'Relationship' de SQLModel:

- '■ Categoria: Define 'items: List[Item] = Relationship(...)'.
- '■ Item: Es el modelo central en donde se definen dos relaciones: 'categorias: List[Categoria]' y 'envios: List[Envio]'.'
- '■ Envio: Define 'items: List[Item] = Relationship(...)'.

»'''El parámetro `link_model` le dice a SQLModel qué tabla de enlace usar, y `back_populates` conecta los dos lados de la relación.

```

class ItemCreate(ItemBase):
    """Esquema de datos esperado al crear un Item."""
    # MODIFICADO: Se usa una lista de nombres de categorías en lugar de un ID.
    categoria_nombres: List[str] = Field(default=[], description="Lista de nombres de categorías a las que pertenece el ítem.")

    class EnvioCreate(EnvioBase):
        """Esquema de datos esperado al crear un Envío. Se añade lista de IDs de Items."""
        item_ids: List[int] = [] # Lista de IDs de los ítems a incluir inicialmente.

    >"""
class ItemOut(ItemBase):
    """Cómo se verá un Item en la respuesta (incluye ID y datos de sus categorías)."""
    # MODIFICADO: Se cambia 'categoria' por 'categorias' (plural).
    id: int
    categorias: List[CategoriaOut] = [] # Muestra la lista de categorías asociadas.

class EnvioOut(EnvioBase):
    """Cómo se verá un Envío en la respuesta (incluye ID y la lista de Items)."""
    id: int
    items: List[ItemOut] = [] # Muestra los ítems completos asociados.

```

Figura 3: Modelos de Creación (Input) y Salida (Output).

»'''Para manejar estas relaciones en la API, los modelos de "Creación"(Input) y "Salida"(Output) fueron modificados:

- **ItemCreate:** En lugar de pedir un 'categoria.id', que es lo que se haría en una relación uno-a-muchos, ahora solicita 'categoria\_nombres: List[str]'. Esto resulta mas comodo ya que no se necesitan saber los IDs.
- **EnvioCreate:** Pide una lista de 'item\_ids: List[int]' para asociar ítems existentes al nuevo envío.
- **ItemOut y EnvioOut:** Se actualizan para mostrar la lista completa de objetos relacionados, dando una respuesta más completa.

```

# Define un endpoint para crear un nuevo ítem.
# MODIFICADO: Se cambia la lógica para aceptar nombres de categorías (M2M).
@app.post("/items/", response_model=ItemOut, status_code=status.HTTP_201_CREATED, tags=["Items"])
def create_item(item_data: ItemCreate, db: SessionDep):
    """Crea un nuevo ítem, asignándolo a una o más categorías existentes por nombre."""

    categorias = []
    # 1. Buscar las categorías por nombre si se proporcionaron
    if item_data.categoria_nombres:
        # Busca en la BD todas las Categorías cuyos nombres estén en la lista.
        categorias = db.query(Categoría).filter(Categoría.nombre.in_(item_data.categoria_nombres)).all()

        # 2. Validación: Comprueba si se encontraron todas las categorías solicitadas.
        if len(categorias) != len(set(item_data.categoria_nombres)):
            raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Una o más categorías no fueron encontradas")

    # 3. Crea el diccionario de datos del ítem base (excluyendo la lista de nombres).
    item_dict = item_data.model_dump(exclude={"categoria_nombres"})
    # 4. Crea la instancia del modelo Item.
    new_item = Item(**item_dict)

    # 5. Asigna la lista de objetos Categoría encontrados a la relación.
    # SQLModel se encargará de crear las entradas en la tabla de enlace 'ItemCategoria'.
    new_item.categorias = categorias

    # 6. Guarda en la BD
    db.add(new_item)
    db.commit()
    db.refresh(new_item)
    # SQLModel cargará automáticamente las categorías para el 'ItemOut'.
    return new_item
"""

```

Figura 4: Lógica modificada del endpoint POST /items/.

»'''La lógica de los endpoints se vuelve un poco mas compleja cuando manejamos las relaciones. Al crear un ítem 'POST /items/', no solo insertamos un registro, sino que ahora debe:

- '1. Recibir la lista de **nombres** de categorías 'categoria\_nombres'.
- '2. Buscar en la base de datos los **objetos** Categoría que coincidan con esos nombres.
- '3. Validar que todas las categorías solicitadas existan.
- '4. Crear el nuevo objeto 'Item'.
- '5. Asignar la lista de objetos 'Categoría' encontrados al campo 'new\_item.categorias'.
- '6. Al hacer 'db.commit()', SQLModel gestiona la tabla de enlace 'ItemCategoria'.

»'''Esta lógica es similar a la aplicada al crear 'Envíos' y al actualizar (PATCH) los recursos, donde las listas de relaciones se reemplazan.

```

@app.post("/optimizar/{envio_id}", tags=["Algoritmo Genético"])
def optimizar_envio(envio_id: int, capacidad: float):
    with Session(engine) as session:
        #Obtenemos el envío por su ID
        envio = session.get(Envio, envio_id)
        if not envio:
            raise HTTPException(status_code=404, detail="Envío no encontrado")

        #Los items se obtienen directamente desde la relación
        items = envio.items
        if not items:
            raise HTTPException(status_code=400, detail="Este envío no tiene items")

        # Obtenemos pesos y ganancias
        pesos = [i.peso for i in items]
        ganancias = [i.ganancia for i in items]

        # Ejecutamos el algoritmo genético
        ag = AlgoritmoGenetico(pesos, ganancias, capacidad, SeleccionRuleta())

        # 1. 'ejecutar()' devuelve un objeto 'Sujetos', lo llamamos 'mejor_solucion'
        mejor_solucion = ag.ejecutar()

        # 2. Obtenemos la LISTA de genes desde el objeto
        mejor_genes_lista = mejor_solucion.genes

        # 3. La ganancia total es la aptitud ya calculada en el objeto
        ganancia_total = mejor_solucion.aptitud

        # 4. Calculamos el peso total usando la lista de genes
        peso_total = 0
        for i, gen in enumerate(mejor_genes_lista):
            if gen == 1:
                peso_total += pesos[i]
    """
    # 5. Creamos la lista de items usando 'mejor_genes_lista'
    items_seleccionados = [
        {
            "indice": i,
            "id": items[i].id,
            # Aquí tenías "nombre", pero el item no tiene "nombre".
            # Asumo que querías el nombre de la CATEGORÍA.
            "nombres_categorias": [cat.nombre for cat in items[i].categorias] if items[i].categorias else [],
            "peso": items[i].peso,
            "ganancia": items[i].ganancia,
        }
    for i, gen in enumerate(mejor_genes_lista) # <-- Usamos la lista
        if gen == 1
    ]

    # 6. Devolvemos la lista de genes y los valores correctos
    return [
        "envio_id": envio.id,
        "destino": envio.destino,
        "mejor_genes": mejor_genes_lista, # <-- Devolvemos la lista
        "ganancia_total": ganancia_total,
        "peso_total": peso_total,
        "items_seleccionados": items_seleccionados,
    ]

```

Figura 5: Nuevo endpoint POST para el Algoritmo Genético.

'''Aqui tenemos la adición más importante, que es el endpoint /optimizar/{envio\_id}. Este endpoint integra el algoritmo genético importado desde Algoritmo\_genetico.py con nuestra API de envíos:

- '1. Recibe un 'envio\_id' y una capacidad de peso.
- '2. Obtiene el objeto 'Envio' de la base de datos.
- '3. Accede a todos los items asociados a ese envío simplemente usando la relación: envio.items.
- '4. Extrae los pesos y ganancias de esa lista de items.
- '5. Inicializa y ejecuta el 'AlgoritmoGenetico' con esos datos.
- '6. La solución se usa para filtrar la lista de items y devolver solo los seleccionados que maximizan la ganancia sin superar la capacidad.

```
'''=====
```

```
''''''7d864be4b5a5b04fabcc156fd569cacca3a98e0
```

## Referencias

- [1] IBM. (s.f.). ¿Qué es una API REST? Recuperado el 14 de octubre de 2025, de <https://www.ibm.com/mx-es/topics/rest-apis>
- [2] Python Software Foundation. (s.f.). Acerca de Python™. Resumen Ejecutivo. Recuperado el 14 de octubre de 2025, de <https://www.python.org/doc/essays/blurb-es/>
- [3] Ramírez, S. (s.f.). FastAPI. Recuperado el 14 de octubre de 2025, de <https://fastapi.tiangolo.com/es/>
- [4] Ramírez, S. (s.f.). SQLAlchemy. Recuperado el 14 de octubre de 2025, de <https://sqlmodel.tiangolo.com/es/>
- [5] IBM Control Desk. (s. f.). Relaciones en Bases de Datos. Recuperado el 31 de octubre de 2025, de <https://www.ibm.com/docs/es/control-desk/7.6.1?topic=structure-database-relationships>
- [6] Jain, A. (2024, 8 septiembre). What is CRUD API? DEV Community. Recuperado el 31 de octubre de 2025, de <https://dev.to/ankitjaininfo/what-is-crud-api-502i>
- [7] Tema 2. Algoritmos genéticos. (s. f.). Departamento de Ciencias de la Computación E Inteligencia Artificial. Recuperado el 31 de octubre de 2025, de <http://www.scehu.es/ccwbayes/docencia/mmcc/docs/t2geneticos>