Guía técnica para el GPT – Uso de `gpt_portable_predictor.py` y documentos RAG de predicción

Este documento describe cómo debe trabajar el GPT con:

- El **motor de predicción portable** implementado en `gpt_portable_predictor.py`.
- Los artefactos de modelos contenidos en `modelos_portable.zip`.
- Los documentos JSON de salida tipo `rag_case_doc_ejemplo.json`.

Su objetivo es que el GPT pueda:

1. Recoger datos de una licitación (a partir de diálogo con el usuario).
2. Normalizarlos a un formato tabular compatible con el motor.
3. Ejecutar el motor en Python para obtener un documento RAG por expediente.
4. Explicar los resultados al usuario, apoyándose en dicho JSON y en la documentación adicional (tesis, metodología, normativa).

---

1. Flujo general del sistema

El sistema completo funciona en dos capas:

1. **Motor de predicción portable (Python)**
   - Ficheros principales:
     - `gpt_portable_predictor.py`
     - `modelos_portable.zip`
   - Se ejecuta en el entorno Python del GPT.
   - Produce, para cada expediente, un único objeto JSON estructurado (`rag_case_doc`) con:
     - Entrada normalizada.
     - Salidas de modelos (RF, XGB).
     - Predicción final del ensemble.
     - Vecinos kNN con información detallada.
     - Metadatos y campos pensados para RAG.

2. **GPT-RAG orquestador**
   - Mantiene una conversación con el usuario.
   - Traduce las respuestas del usuario a las variables que requiere el motor.
   - Llama a Python para generar el `rag_case_doc`.
   - Usa ese JSON como **fuente factual principal** para explicar el resultado, combinándolo con:
     - Metodología RAG.
     - Capítulos de tesis.
     - Normativa relacionada, si es relevante.

El GPT **no debe re-entrenar modelos ni modificar artefactos**. Únicamente debe:

- Preparar inputs,
- Invocar el motor,
- Interpretar y explicar las salidas.

---

2. Motor de predicción portable (`gpt_portable_predictor.py`)

2.1. Carga de modelos

El punto de entrada es:

```python
from gpt_portable_predictor import load_predictor_from_zip

predictor = load_predictor_from_zip("modelos_portable.zip")
```

- `modelos_portable.zip` contiene:
  - `preprocess_pre_portable.json`
  - `rf_base_portable.json`
  - `rf_124_portable.json`
  - `xgb_baja_portable.json`
  - `xgb_baja_model.json`

```
  - `knn_space_portable.json`
  - `knn_train.npz`
  - `knn_all.npz`
  - `ensemble_portable.json`
  - `centroides_pre.json`
- El motor **no utiliza sklearn ni xgboost**. Reimplementa:
  - Árboles de RandomForest desde JSON.
  - Booster de XGBoost desde JSON.
  - Preprocesado (imputación + escalado + one-hot encoding).
  - Búsqueda de vecinos kNN en espacio PRE_NUM.

### 2.2. Clases y funciones principales

Las piezas internas se organizan en clases (no es necesario usarlas directamente desde el GPT, salvo a través de la fachada):

- `PortablePreprocessor`
- `RandomForestPortable`
- `XGBBinaryPortable`
- `KNNPortable`
- `EnsemblePredictor`
- `PortablePredictor` (fachada principal)

La interfaz pública relevante es:

```python
from gpt_portable_predictor import (
    load_predictor_from_zip,
    build_prediction_payload,
    build_rag_case_document,
    generate_and_save_rag_case_document,
)
```

2.2.1. `predictor.predict(df_raw)`

```python
resultados = predictor.predict(df_raw)
```

Devuelve un diccionario con al menos:

- `"rf_base"`:
  - `"pred"`: Series con clase predicha (cluster) para cada fila.
  - `"proba"`: DataFrame con probabilidades por clase.
- `"rf_124"`:
  - Igual estructura que `"rf_base"` pero para el modelo especializado en clases 1,2,4.
- `"xgb_baja"`:
  - `"p_bajo"`: probabilidad continua de "baja competencia" según XGB.
  - `"baja_comp"`: flag binario (1 si `p_bajo >= 0.38`, 0 en caso contrario).
  - `"threshold_bajo"`: umbral **fijado explícitamente en 0.38**.
  - `"labels"`: lista de etiquetas de salida del XGB (ej. `["BAJO","MEDIO","ALTO"]`).

2.2.2. `predictor.knn_neighbors(df_raw, topk, space)`

```python
vecinos = predictor.knn_neighbors(df_raw, topk=10, space="all")
```

Devuelve:

- `"indices"`: array de índices de las filas vecinas (en `Z_train` o `Z_all`).
- `"distances"`: array de distancias euclídeas en el espacio PRE_NUM.
- `"similarity_percent"`: similitud relativa en *[%]*, derivada de la distancia y un valor de referencia `p95_nn1_train`.
- `"p95_nn1_train"`: percentil 95 de la distancia al primer vecino en TRAIN.

> **Nota**: el script de generación de RAG ya "resuelve" estos índices y rellena los datos completos de los vecinos; el GPT suele trabajar directamente con el JSON RAG y no con esta función.
```

2.2.3. Funciones de alto nivel para RAG

- `build_prediction_payload(predictor, df_raw, row_index, knn_topk, knn_space)`
  Construye un payload estructurado (dict JSON-safe) con:
  - Entrada (`input`),
  - Salidas de modelos (`models`),
  - Ensemble (`ensemble`),
  - kNN (`knn`),
  - Metadatos (`meta`).

- `build_rag_case_document(predictor, df_raw, row_index, knn_topk, knn_space)`
  Construye el documento RAG completo (dict).

- `generate_and_save_rag_case_document(...)`
  Igual que la anterior, pero además lo guarda en un `.json` en disco.

---

3. Contrato de entrada para nuevas predicciones

3.1. Estructura general del input

Para un nuevo expediente, el GPT debe construir un `DataFrame` de una sola fila con **nombres de columna compatibles** con el modelo (los mismos que en `df_test_train_v8.xlsx` / `df_norm`).

Es suficiente que el GPT construya un `dict` con las Variables de **PRE-licitación** (numéricas y categóricas):

```python
user_input = {
    "Presupuesto_licitacion_lote_c": ...,
    "Plazo_m": ...,
    "C_precio_p": ...,
    "C_juicio_valor_p_c": ...,
    "Mes_lici": ...,
    "Tipo_de_Administracion_c": ...,
    "Tipo_de_procedimiento_c": ...,
    "N_lotes_c": ...,
    "N_clasi_empresa_C": ...,
    "Tipo_de_contrato_c": ...,
    "Codigo_Postal_c": ...,
    "N_CPV_c": ...,
}
df_norm = pd.DataFrame([user_input])
```

Ese `df_norm` se pasa al motor (`predictor` y funciones RAG).

3.2. Variables clave (las más importantes)

El GPT debería priorizar la recogida y normalización correcta de, al menos, estas variables:

Variable	Tipo	Unidad / dominio	Comentario técnico
`Presupuesto_licitacion_lote_c`	float	euros	Presupuesto de licitación del lote. Siempre en euros, valor positivo.
`Plazo_m`	int	meses	Plazo contractual en meses.
`C_precio_p`	float	% en 0-100	Ponderación del precio en la valoración.
`C_juicio_valor_p_c`	float	% en 0-100	Ponderación de criterios de juicio de valor.

`Mes_lici`	int	1-12	Mes de licitación (enero=1,…, diciembre=12).
`Tipo_de_Administracion_c`	str	categoría	Tipo de administración (ej.: `"Autoridad local"`, `"Administración General"`, etc.).
`Tipo_de_procedimiento_c`	str	categoría	Procedimiento de contratación (ej.: `"Abierto"`, `"Abierto simplificado"`, etc.).
`Provincia2`	int	código numérico	Código de provincia obtenido del codigo postal.
`N_CPV`	int	unidades enteras	Numero de clasificadores CPV de la licitación.
`N_lotes`	int	unidades enteras	Número de lotes en los que se divide la licitación.
`N_clasi_empresa`	int	unidades enteras	Numero de requerimientos empresariales exigidos en la licitacion.
`Tipo_de_contrato_c`	str	categoría	Tipo de contrato (ej.: `"Obras"`, `"Servicios"`, etc.).

Además, existen otras variables (por ejemplo, fechas de publicación, identificadores PLACSP, etc.) que pueden completarse si el usuario las proporciona, pero no son imprescindibles para la predicción.

3.3. Reglas de normalización desde lenguaje natural

El GPT debe aplicar reglas claras para transformar entradas textuales del usuario en valores numéricos/categóricos:

- Meses:
  - `"Octubre"`, `"octubre (10)"`, `"mes 10"` → `Mes_lici = 10`.
- Porcentajes:
  - `"75%"`, `"75 %"` → `75.0`.
  - `"0.75"` **interpretar preferentemente como 75 %**, salvo que el contexto indique explícitamente 0-1.
- Plazos:
  - `"14 meses"`, `"14 m"`, `"duración de 14"` → `Plazo_m = 14`.
- Presupuesto:
  - `"893.145 €"`, `"893145"`, `"0,893 M€"` → `Presupuesto_licitacion_lote_c ≈ 893145.0`
    - El GPT debe intentar entender unidades (miles, millones) y expresarlo siempre en euros.

Si la interpretación es ambigua, el GPT debe:

- Pedir aclaración, o
- Explicitar la suposición que va a realizar y advertir de la limitación.

3.4. Gestión de valores faltantes

Si el usuario no puede proporcionar alguna variable:

- El motor seguirá funcionando gracias a la imputación interna (medias, modas).
- Sin embargo, desde el punto de vista del DSS, la predicción será menos fiable.

El GPT debe:

- Indicar al usuario qué campos se han imputado o dejado por defecto.
- Señalar explícitamente que esto reduce la confianza en la interpretación, aunque el motor devuelva un resultado numérico.

---

4. Estructura del documento RAG (`rag_case_doc`)

4.1. Esquema de alto nivel

El documento JSON generado por `build_rag_case_document` tiene esta estructura:

```json
{
  "doc_type": "licitacion_riesgo_prediccion",
  "rag_metodologia": { ... },
  "source": {
    "input": { ... },
    "models": { ... },
    "ensemble": { ... },
    "knn": { ... },
    "meta": { ... }
  },
  "retriever": { ... },
  "evaluation": { ... }
}
```

4.2. `source.input`

Ejemplo:

```json
"input": {
  "index_label": "0",
  "row_index": 0,
  "features": {
    "Identificador": 4353889,
    "Presupuesto_licitacion_lote_c": 520161.64,
    "Plazo_m": 4,
    "C_precio_p": 100.0,
    "C_juicio_valor_p_c": 0.0,
    "Mes_lici": 12,
    "Tipo_de_Administracion_c": "Autoridad local",
    "Tipo_de_procedimiento_c": "Abierto simplificado",
    "N_CPV": "3",
    ...
  }
}
```

- `features` contiene **todas las columnas** del `df_norm` para ese caso.
- Fechas se representan como cadenas ISO (`"YYYY-MM-DDTHH:MM:SS"`).
- Valores faltantes se representan como `null` (JSON estándar).

4.3. `source.models`

El bloque `models` contiene, al menos:

4.3.1. `xgb_baja`

```json
"xgb_baja": {
  "p_bajo": 0.4697,
  "baja_comp": 1,
  "threshold_bajo": 0.38,
  "labels": ["BAJO", "MEDIO", "ALTO"],
  "proba": {
    "BAJO": 0.4697,
    "MEDIO": 0.4175,
    "ALTO": 0.1126
  },
  "risk_label": "BAJO"
}
```

Interpretación:

- `p_bajo`: probabilidad continua de “baja competencia” (clase BAJO).
- `baja_comp`: flag binario, calculado como `p_bajo >= 0.38`.
- `proba`: distribución completa entre BAJO / MEDIO / ALTO.
- `risk_label`:
  - `"BAJO"` cuando `baja_comp = 1` y existe etiqueta BAJO.

  - Si `baja_comp = 0`, es la etiqueta más probable entre el resto de clases (típicamente MEDIO o ALTO).

4.3.2. `rf_base` y `rf_124`

Ambos tienen estructura similar, por ejemplo:

```json
"rf_base": {
  "pred": "4",
  "proba": {
    "0": 0.037,
    "1": 0.072,
    "2": 0.068,
    "3": 0.172,
    "4": 0.496,
    "5": 0.154
  },
  "classes": ["0","1","2","3","4","5"]
}
```

- `rf_base`: modelo general de clasificación en 6 clusters.
- `rf_124`: modelo especializado que sólo discrimina entre clases 1, 2 y 4.

Las probabilidades del RF portable pueden diferir hasta ~0.02 respecto a sklearn. Este margen es aceptado y está documentado en `meta.notes`.

4.4. `source.ensemble`

Ejemplo:

```json
"ensemble": {
  "pred_final": "1",
  "confianza_pred": 0.7151,
  "fuente_confianza": "rf_124_basebc",
  "flip_ensemble": true,
  "baja_comp": 1,
  "p_bajo": 0.4697
}
```

Interpretación:

- `pred_final`: clase de cluster final (texto `"0"`-`"5"`).
- `confianza_pred`: probabilidad de la clase ganadora en el modelo que se ha usado como fuente de decisión.
- `fuente_confianza`:
  - `"rf_base"` o `"rf_124_basebc"`, indica qué modelo fue elegido.
- `flip_ensemble`:
  - `true` indica que se ha cambiado la predicción de RF base por la de RF_124.
- `baja_comp`, `p_bajo`:
  - Copia de los valores de XGB para integrarlos en la explicación global.

Regla de combinación implementada:

1. Se ejecutan RF base y RF_124.
2. Si RF base predice una clase **dentro de {1,2,4}**, se compara la probabilidad top-1 de RF base y RF_124.
3. Se elige el modelo con mayor probabilidad top-1:
   - Si se elige RF_124 → `flip_ensemble = true`.
   - Si se mantiene RF base → `flip_ensemble = false`.

4.5. `source.knn`

Ejemplo de estructura:

```json
"knn": {
  "space": "all",
  "topk": 10,
```

```
  "indices": [0, 999, 49, ...],
  "distances": [0.0, 0.0168, ...],
  "similarity_percent": [100.0, 98.9, ...],
  "p95_nn1_train": 1.5866,
  "neighbors": [
    {
      "rank": 1,
      "knn_index": 0,
      "case_id": 4353889,
      "summary": {
        "Ahorro_final": 9.72,
        "Presupuesto_licitacion_lote_c": 520161.64,
        "N_ofertantes": 8,
        "Plazo_m": 4,
        "Baja_p": 20.4,
        "C_precio_p": 100.0
      },
      "features": { ... fila completa del vecino ... }
    },
    ...
  ]
}
```

- `neighbors` ya incluye:
  - `summary` con las 6 variables más relevantes (ahorro y 5 económicas).
  - `features` con la fila completa del `df_norm` del vecino.
- El GPT **no necesita acceder a ningún otro dataset** para explicar los casos parecidos; toda la información está en este bloque.

4.6. `source.meta`

Contiene, al menos:

```json
"meta": {
  "zip_path": "ruta/relativa/a/modelos_portable.zip",
  "notes": "Probabilidades RF portables pueden diferir hasta ~0.02 respecto a sklearn; threshold_bajo del XGB se fija explícitamente en 0.38.",
  "model_set_version": "v1.0_portable_rag",
  "generated_at": "2025-11-16T15:45:04.240573Z"
}
```

- `model_set_version`: etiqueta de versión de artefactos.
- `generated_at`: timestamp de inferencia en UTC (ISO8601, con sufijo `Z`).

4.7. `retriever` y `evaluation`

- `retriever` agrupa claves útiles para indexado y filtrado:

  ```json
  "retriever": {
    "case_id": "0",
    "row_index": 0,
    "business_ids": {
      "Identificador": 4353889,
      "Identificador_lote": "4353889//NEGOBP-2019/1200",
      "Identificador_lici": "NEGOBP-2019/1200"
    },
    "labels": {
      "pred_ensemble": "1",
      "baja_comp_flag": 1,
      "p_bajo": 0.4697,
      "rf_base_pred": "4",
      "rf_124_pred": "1",
      "xgb_risk_label": "BAJO"
    },
    "key_features": {
      "Presupuesto_licitacion_lote_c": 520161.64,
      "N_ofertantes": 8,
      "Plazo_m": 4,
```

```
        "Baja_p": 20.4,
        "C_precio_p": 100.0,
        "C_juicio_valor_p_c": 0.0,
        "Mes_lici": 12,
        "Tipo_de_Administracion_c": "Autoridad local",
        "Tipo_de_procedimiento_c": "Abierto simplificado",
        "Provincia2": 33,
        "CPV_c": "45233252-0",
        "CPV_lote_c": "45233252-0"
      },
      "available_feature_keys": [ ... lista completa de campos ... ]
    }
    ```

- `evaluation` es un placeholder para métricas y feedback, que puede rellenarse en etapas posteriores.

---

5. Pautas para el GPT al usar este sistema

5.1. Cuándo ejecutar Python

El GPT debe ejecutar código Python cuando:

- El usuario solicite **una nueva predicción** ("quiero estimar el riesgo de esta licitación…").
- El usuario quiera **recalcular** o **analizar un caso específico** con nuevos datos.

Secuencia típica:

1. Conversación para recoger datos.
2. Normalización a un `dict` con las variables clave.
3. En Python:
    ```python
    from gpt_portable_predictor import load_predictor_from_zip, generate_and_save_rag_case_document
    import pandas as pd

    predictor = load_predictor_from_zip("modelos_portable.zip")
    df_norm = pd.DataFrame([user_input_dict])
    rag_doc = generate_and_save_rag_case_document(
        predictor=predictor,
        df_raw=df_norm,
        row_index=0,
        knn_topk=10,
        knn_space="all",
        output_path="rag_case_doc_tmp.json",
    )
    ```
4. De vuelta al usuario: explicación basada en `rag_doc`.

5.2. Cómo explicar resultados

El GPT debe:

- Basarse **primariamente** en `rag_doc["source"]` para:
  - Clase final (`ensemble.pred_final`) y confianza (`ensemble.confianza_pred`).
  - Riesgo de baja competencia (`xgb_baja.p_bajo`, `xgb_baja.risk_label`, `baja_comp`).
  - Comparación con vecinos (`knn.neighbors[i].summary` y `features`).
- Usar la tesis/metodología como **contexto**:
  - para interpretar clusters,
  - para justificar por qué determinadas variables son relevantes,
  - para conectar con conceptos estadísticos y de gestión de proyectos.

Debe dejar claro al usuario:

- Qué partes de la explicación se basan en datos concretos del JSON.
- Qué partes son interpretación o contexto general (tesis, normativa).

5.3. Qué hacer si la entrada es deficiente o incoherente

Si, tras conversar, el GPT detecta:

- Campos críticos sin valor (p. ej. presupuesto, nº de ofertas, baja).
- Valores claramente fuera de rango (p. ej. `Baja_p = -5`, `N_ofertantes = 0`).

Debe:

1. Intentar aclarar con el usuario.
2. Si no es posible, explicitar:
   - Qué valores se han imputado o ajustado.
   - Que la fiabilidad de la predicción se ve afectada.

El motor **no lanza errores en estos casos**; simplemente imputa. El GPT es responsable de advertir de esta situación.

5.4. Errores que debe evitar el GPT

- No debe:
  - Re-entrenar modelos.
  - Modificar los archivos del ZIP.
  - Inventar columnas que el motor no usa.
- No debe asumir:
  - Igualdad numérica exacta con sklearn/xgboost originales.
    - Hay una tolerancia aceptada de ≈0,02 en probabilidades de RF.
- No debe reinterpretar:
  - Los clusters como categorías que no estén respaldadas por la tesis/metodología.
  - El riesgo de baja competencia sin respetar `risk_label` y `threshold_bajo`.

Este documento debe usarse como **guía interna del GPT** para interactuar correctamente con el motor portable y los documentos RAG, garantizando:

- Coherencia en la construcción de inputs.
- Interpretación fiel de los outputs.
- Transparencia con el usuario sobre supuestos, límites y nivel de confianza.

---

7. Modo light sin Python (usuarios sin herramienta de código)

En algunos entornos o tipos de cuenta el GPT **no puede ejecutar código Python**. En ese escenario:

- No es posible usar `gpt_portable_predictor.py` ni cargar `modelos_portable.zip`.
- No se pueden recalcular distancias kNN ni probabilidades RF/XGB.
- El sistema solo puede trabajar con la **información ya precomputada** y cargada como conocimiento estático.

Para estos casos se ha preparado un archivo adicional:

- `casos_referencia_light.md`

Este archivo contiene, para todos los casos TRAIN+TEST de la base utilizada en el DSS:

- Variables de **PRE-licitación** (numéricas y categóricas), incluyendo, entre otras:
  - `Presupuesto_licitacion_lote_c`
  - `C_precio_p`
  - `Plazo_m`
  - `N_lotes`
  - `N_CPV`
  - `N_clasi_empresa`
  - `C_juicio_valor_p_c`
  - `Intervalo_lici_d_c`
  - `Tipo_de_contrato_c`
  - `Tipo_de_Administracion_c`
  - `Tipo_de_procedimiento_c`
  - `Codigo_Postal_c`
- Las cinco variables económico□competitivas principales:
  - `Presupuesto_licitacion_lote_c`
  - `N_ofertantes`

  - `Plazo_m`
  - `Baja_p`
  - `C_precio_p`
- El resultado histórico para cada caso:
  - `Ahorro_final`
  - `Cluster_6` (en el campo `cluster_final`)
- Coordenadas normalizadas `z_*` para las cinco variables principales, calculadas sobre toda la base histórica.

Cada caso se presenta como un bloque `json` con la siguiente información mínima:

- `case_id`: identificador del caso (normalmente `Identificador` de la BD).
- `dataset_index`: índice de fila en el dataset original.
- `cluster_final`: valor de `Cluster_6`.
- `Ahorro_final`.
- Las cinco variables principales y sus `z_*`.
- Un diccionario `pre_variables` con todas las variables de PRE□licitación disponibles.

7.1. Qué debe hacer el GPT en modo light

Cuando no haya Python disponible, el GPT debe seguir este enfoque:

1. **Recoger variables clave del usuario**
   Al menos:
   - `Presupuesto_licitacion_lote_c` (euros),
   - `Plazo_m`,
   - `C_precio_p`,
   usando las mismas reglas de normalización que en el modo completo (porcentajes 0–100, meses 1–12, etc.).
   Siempre que sea posible, también debería preguntar por otras variables de PRE□licitación (tipo de contrato, administración, procedimiento, código postal, etc.).

2. **Formular una consulta RAG sobre `casos_referencia_light.md`**
   La consulta debe reflejar explícitamente los valores o rangos de:
   - `Presupuesto_licitacion_lote_c`,
   - `C_precio_p`,
   - `Plazo_m`,
   - y, si se conocen, el resto de PRE, ya que el RAG trabaja sobre texto.

3. **Recuperar varios casos candidatos y seleccionar los 3 más relevantes**
   Entre los casos devueltos por el RAG, el GPT escoge los 3 que considere más coherentes con el caso del usuario, fijándose en:
   - similitud de órdenes de magnitud (presupuesto),
   - rango de `N_ofertantes`, `Plazo_m`, `Baja_p`, `C_precio_p`,
   - coincidencia en tipo de administración, contrato, procedimiento, etc.

4. **Calcular una medida aproximada de similitud**
   El propio archivo `casos_referencia_light.md` incluye las medias, desviaciones estándar y la fórmula recomendada.
   Para cada uno de los 3 casos seleccionados, el GPT:
   - Asigna z□scores al caso nuevo usando esas medias/desviaciones.
   - Usa los `z_*` del caso de referencia para calcular la distancia euclídea:
     $d = \sqrt{\sum_k (z_{k,\text{nuevo}} - z_{k,\text{ref}})^2}$
   - Convierte esa distancia en similitud porcentual con:
     $\text{similitud} = \max(0, 100 \cdot (1 - d / D_{\max}))$, con $D_{\max} = 5.0$.

   Este cálculo solo debe hacerse sobre esos 3 casos, nunca sobre toda la base.

5. **Elegir el caso de referencia principal y un cluster de referencia**
   - Se ordenan los 3 casos por similitud descendente.
   - El caso con mayor similitud se considera el **más parecido**.
   - Si los 3 casos pertenecen a clusters distintos, se toma el `cluster_final` (Cluster_6) del caso más parecido como *cluster de referencia* para el nuevo expediente.

6. **Explicar el resultado como comparación con casos históricos**
   El GPT debe presentar al usuario:
   - Los 3 casos más parecidos, mostrando para cada uno:
     - `Presupuesto_licitacion_lote_c`, `N_ofertantes`, `Plazo_m`, `Baja_p`,

`C_precio_p`, `Ahorro_final`,
651 - el `cluster_final`,
652 - el % de similitud aproximado.
653 - Un resumen interpretativo del tipo:
654 - qué patrones de ahorro y competencia se observan en esos casos,
655 - qué sugiere eso para el expediente del usuario,
656 - en qué cluster histórico encaja mejor por analogía.
657
658 Siempre debe dejar claro que:
659 - En modo light **no se han ejecutado los modelos RF/XGB/ensemble**,
660 - las conclusiones se basan en **comparaciones históricas**, no en predicciones probabilísticas exactas.
661
662 ### 7.2. Limitaciones y buenas prácticas en modo light
663
664 - El GPT no debe prometer niveles de precisión equivalentes al modo completo.
665 Debe presentar las conclusiones como orientativas y condicionadas a la calidad de los datos aportados por el usuario.
666 - No debe intentar aplicar fórmulas de la tesis que requieran parámetros que solo están disponibles vía Python (por ejemplo, detalles internos del SOM, centroides PRE, etc.).
667 - Debe usar siempre que sea posible el vocabulario de la tesis (clusters, baja competencia, ahorro, etc.), pero dejando claro el **modo de trabajo**:
668 - "predicción basada en modelos" cuando hay Python,
669 - "comparación con casos históricos similares" cuando no lo hay.
670
671 Con este apéndice, la misma guía sirve tanto para el escenario de uso avanzado con entorno de código como para el escenario restringido sin Python, manteniendo idénticos contratos de variables y criterios de interpretación, y cambiando únicamente el mecanismo de cálculo.
672