JULIO DE 2024


**UNIVERSIDAD DISTRITAL**
FRANCISCO JOSÉ DE CALDAS

# MANUAL TÉCNICO DE MODELOS DE APRENDIZAJE AUTOMÁTICO EN PYTHON

PREDICCIÓN DEL RENDIMIENTO ACÁDEMICO DE ESTUDIANTES DE INGENIERÍA

EQUIPO DE DESARROLLO

# Tabla de Contenido

# Índice de Ilustraciones

# 1. Introducción

En la era de la información, el análisis de datos y la predicción de resultados se han convertido en elementos fundamentales para la toma de decisiones en diversos campos, desde el marketing y las finanzas hasta la medicina, la ingeniería y la educación. Los modelos de aprendizaje automático (machine learning) son herramientas poderosas que permiten a las organizaciones extraer conocimientos valiosos de sus datos y hacer predicciones precisas. Estos modelos no solo ayudan a identificar patrones y tendencias, sino que también pueden proporcionar información predictiva que es crucial para la planificación estratégica y la mejora continua.

Este manual técnico está diseñado para proporcionar una guía completa sobre la implementación y el uso de varios modelos de aprendizaje automático en Python. El lenguaje de programación Python se ha establecido como uno de los lenguajes más populares y versátiles para el análisis de datos y el desarrollo de modelos de machine learning, gracias a su extensa biblioteca de paquetes y su comunidad activa. Python ofrece un ecosistema robusto que facilita el desarrollo rápido y eficiente de modelos complejos, permitiendo a los analistas y científicos de datos centrarse en la interpretación y aplicación de los resultados.

A lo largo de este manual, exploraremos una amplia variedad de modelos de aprendizaje automático, cada uno con sus propias aplicaciones específicas en el ámbito educativo, más específicamente en la predicción del rendimiento o promedio de los estudiantes que forman parte de la Facultad de Ingeniería de la Universidad Distrital Francisco José de Caldas. La educación superior enfrenta desafíos significativos en la mejora del rendimiento estudiantil y en la retención de estudiantes, y los modelos de machine learning pueden proporcionar insights críticos para abordar estos desafíos.

Cada capítulo de este manual abordará ejemplos prácticos de su implementación en Python, proporcionando código detallado y explicaciones claras. Al final de este manual, los lectores tendrán una comprensión sólida de cómo aplicar estos modelos a sus propios conjuntos de datos y problemas específicos. Además, se proporcionarán estrategias para evaluar y optimizar el rendimiento de los modelos, garantizando que los usuarios puedan sacar el máximo provecho de estas poderosas herramientas en sus proyectos educativos.

# 2. Ámbitos del Sistema

## 2.1 Diagrama del sistema

A continuación, se muestra la estructura de la aplicación por bloques, compuesta por una interfaz de usuario, la aplicación en Python y el servidor de aplicaciones (ver Ilustración 1).

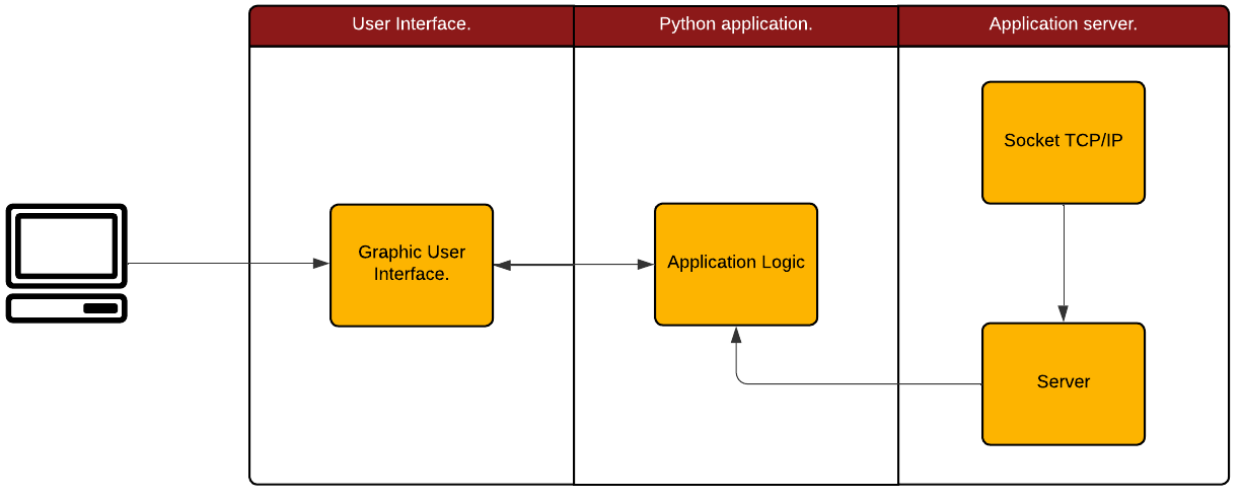


***Ilustración 1**. Diagrama del Sistema.*

## 2.2 Descripción del Sistema

El aplicativo predice el rendimiento académico utilizando algoritmos de aprendizaje supervisado basado en variables académicas, demográficas y sociodemográficas. En este trabajo se seleccionan las variables más influyentes en el transcurso de la vida académica de los estudiantes mediante métodos de filtro, embebidos y de ensamble. Así como las características semestre a semestre utilizando algoritmos de machine learning (árbol de decisión, KNN, SVC, Naive Bayes, LDA) implementados en Python.

## 2.3 Diagrama Jerárquico del Sistema

El aplicativo cuenta con 5 módulos: El primer módulo “Carga y limpieza de datos” permite hacer la carga de archivos e internamente ser procesados para modificar datos que dificulten el procesamiento de la información; El módulo “Analítica Descriptiva” muestra todas las carreras de la facultad y por medio de graficas le permite al usuario hacer un filtrado de la información; El módulo “Analítica Diagnostica” por medio de selección de variables permite generar aún más graficas para complementar el módulo de Analítica Descriptiva; El módulo “Analítica Predictiva” predice el rendimiento académico haciendo una selección de variables; Finalmente el módulo practico realiza una predicción del rendimiento académico ya sea grupal o individual (ver Ilustración 2).

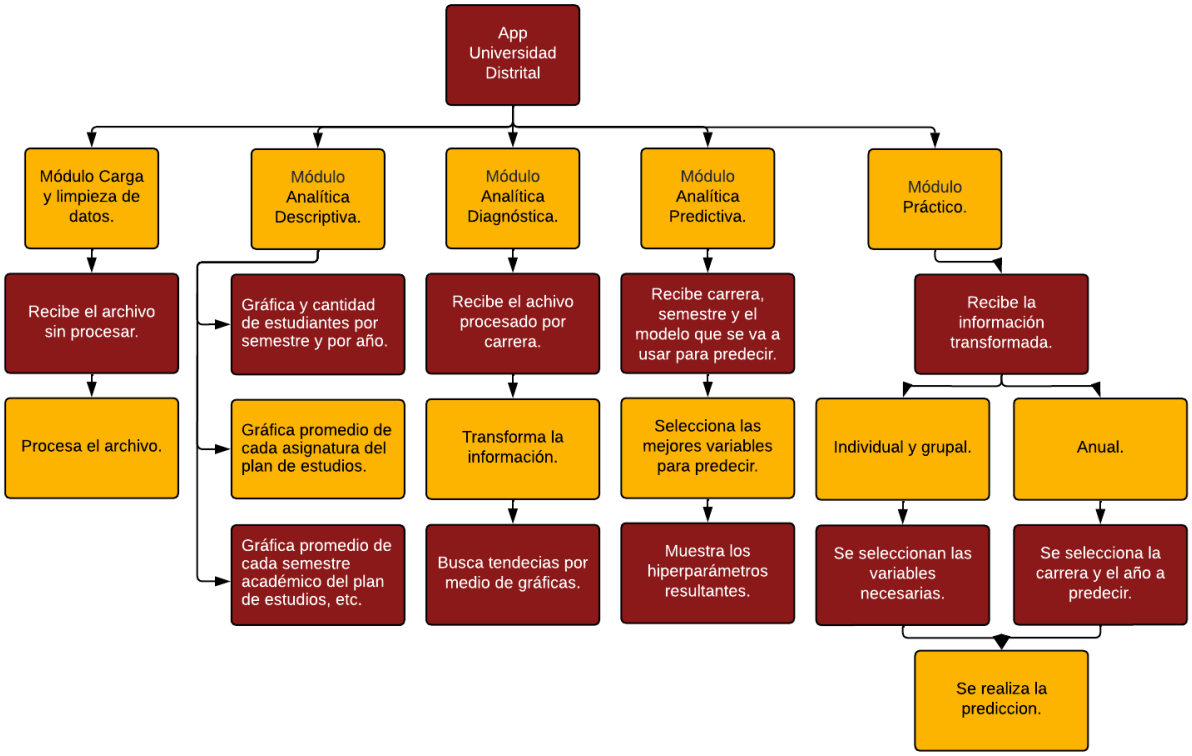


***Ilustración 2**. Diagrama jerárquico del sistema.*

## 2.4 Requerimientos del Sistema

El sistema requiere para su funcionamiento en la aplicación ejecutable:

- Uno o dos computadores con conexión local
- Sistema operativo: Windows 10
- Memoria RAM: 90 GB
- Almacenamiento raíz: 750 GB

Para compilar el aplicativo desde el código fuente:

- Permisos de administrador en la cuenta del usuario del sistema operativo.
- Python 3.11.0 o superior instalado en el sistema operativo.
- Pip en la versión 24.0 o superior.

Librerias:

- base64: 3.12.3
- catboost: 1.2.5
- csv: 3.12.3
- fastapi: 0.111.0
- glob: 3.12.3
- google: 3.0.0
- io: 3.12.3
- json: 3.12.3
- lightgbm: 1.3.0
- matplotlib: 8.9.0
- mlens: 0.2.3
- numpy: 1.23.5

- openpyxl: 3.1.2
- os: 3.12.3
- os.path: 3.12.3
- pandas: 2.2.2
- pickle: 3.12.3
- pydantic: 2.7.3
- seaborn: 0.13.2
- sklearn: 1.5.0
- torch: 2.3.0
- typing: 3.12.3
- unidecode: 1.3.8
- urllib.error: 3.12.3
- uvicorn: 0.30.1
- warnings: 3.12.3
- xgboost: 2.0.3

Librerias, dependencias del Front-End y su versión:

- @angular-devkit/build-angular: ^14.0.4
- @angular/cli: ^14.2.13
- @angular/compiler-cli: ^14.0.0
- @types/jasmine: ~4.0.0
- jasmine-core: ~4.1.0
- karma: ~6.3.0
- karma-chrome-launcher: ~3.1.0
- karma-coverage: ~2.2.0
- karma-jasmine: ~5.0.0
- karma-jasmine-html-reporter: ~1.7.0
- typescript: ~4.7.2

Dependencias de Software:

- @angular/animations: ^14.0.0
- @angular/cdk: ^14.0.0
- @angular/common: ^14.0.0
- @angular/compiler: ^14.0.0
- angular/core: ^14.0.0
- @angular/forms: ^14.0.0
- @angular/material: ^14.0.0
- @angular/platform-browser: ^14.0.0
- @angular/platform-browser-dynamic: ^14.0.0
- @angular/router: ^14.0.0
- bootstrap: ^5.2.2
- chart.js: ^4.0.1
- i: ^0.3.7
- jquery: ^3.6.1
- ngx-csv: ^0.3.2
- rxjs: ~7.5.0
- tslib: ^2.3.0
- zone.js: ~0.11.4

# 3. Módulo de Carga y Limpieza

La limpieza de datos se refiere a todo tipo de tareas y actividades para detectar y reparar errores en los datos. Esta actividad que puede consumir mucho tiempo, es esencial en el camino de la obtención de futuros buenos resultados al aplicar algoritmos de aprendizaje automático. Es claro que dependiendo del conjunto de datos que se disponga a analizar, también así serán las diferentes estrategias para su limpieza. Surge entonces la idea acerca de cuáles podían ser los pasos a seguir para preparar los datos antes de aplicar los algoritmos en un proyecto de aprendizaje automático. Los siguientes pasos pueden proporcionar una guía clara y estructurada en este proceso:

- Limpieza: son técnicas que identifican y corrigen errores presentes en los datos, cuyas características fueron descritas anteriormente en este apartado.
- Transformación: cambio de la escala o distribución de variables. Algunos algoritmos suponen que cada variable de entrada, y quizás la variable objetivo, tienen una distribución de probabilidad gaussiana. Esto significa que, si se tienen variables de entrada que no lo son, es mejor cambiarlas a gaussianas.

## 3.1 Limpieza de Información

Se inicia la limpieza definiendo una función para la carga del conjunto de los datos de una carrera sin formular en formato ".csv" separado por ";".

```
archivo_base_path = 'Ruta del archivo'
def procesar_archivos(archivo_base_path):
    industrial = pd.read_csv(archivo_base_path, sep=",")
```

A partir de este punto se empezarán a asignar una serie de condicionales los cuales manipularán la información dentro del DataFrame, asignando una distancia a cada localidad representada por un numero dentro del rango del 1 al 7. (Por motivos de integridad del manual, el código fuente se fragmento con el fin de explicar el funcionamiento del mismo y no extender de manera excesiva el documento.)

```
industrial["DISTANCIA"] = np.where(((industrial["LOCALIDAD"] == "BARRIOS UNIDOS") |
(industrial["LOCALIDAD"] == "RAFAEL URIBE URIBE") | (industrial["LOCALIDAD"] ==
"ANTONIO NARINO") | (industrial["LOCALIDAD"] == "ENGATIVA") | (industrial["LOCALIDAD"]
== "KENNEDY")), "3", industrial["DISTANCIA"])
```

## 3.2 Transformación de Datos

### *3.2.1 Reemplazo de valores por columna*

Después de haber asignado las distancias, se realiza la primera transformación de los valores dentro del DataFrame actualizado, todos los valores de tipo texto son convertidos a tipo numérico asignándoseles un identificador único. (Por motivos de integridad del manual, el código fuente se fragmento con el fin de explicar el funcionamiento del mismo y no extender de manera excesiva el documento.)

```
industrial["GENERO"] = industrial["GENERO"].replace({"MASCULINO": 0, "FEMENINO": 1})
industrial = industrial.apply(pd.to_numeric)
```

## 3.3 Creación de Nuevas Columnas y Asignación de Nuevos Valores

### *3.3.1 Primer semestre*

Haciendo uso del nuevo DataFrame con los nuevos valores transformados, se definen una serie de listas que contendrán las notas, veces cursadas, y créditos para diferentes materias del primer semestre de la carrera.

```
NOTAS_INDUSTRIAL_UNO = [
      "NOTA_DIFERENCIAL",
      "NOTA_DIBUJO",
      "NOTA_QUIMICA",
```

```
    ...
]
VECES_INDUSTRIAL_UNO = [
    "VECES_DIFERENCIAL",
    "VECES_DIBUJO",
    "VECES_QUIMICA",
    ...
]
CREDITOS_INDUSTRIAL_UNO = [
    "CREDITOS_DIFERENCIAL",
    "CREDITOS_DIBUJO",
    "CREDITOS_QUIMICA",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el primer semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
        for nota_col, veces_col, creditos_col in zip(NOTAS_INDUSTRIAL_UNO,
VECES_INDUSTRIAL_UNO, CREDITOS_INDUSTRIAL_UNO):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_UNO"] = promedios_semestre
```

Cálculo de métricas por semestre.

- CAR_UNO: Cantidad de asignaturas repetidas
- NAC_UNO: Numero de asignaturas cursadas.
- NAA_UNO: Numero de asignaturas aprobadas.
- NAP_UNO: Numero de asignaturas no aprobadas.
- EPA_UNO: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_UNO"]= industrial[VECES_INDUSTRIAL_UNO].apply(lambda x: (x > 1).sum(),
axis=1)
industrial["NAC_UNO"]= industrial[VECES_INDUSTRIAL_UNO].apply(lambda x: (x > 0).sum(),
axis=1)
industrial["NAA_UNO"]=industrial[NOTAS_INDUSTRIAL_UNO].apply(lambda    x:    (x    >=
30).sum(), axis=1)
industrial["NAP_UNO"] = industrial["NAC_UNO"] - industrial["NAA_UNO"]
industrial["EPA_UNO"] = industrial.apply(lambda row: 1 if row["PROMEDIO_UNO"] < 30 or
row["NAP_UNO"] >= 3 else 2, axis=1)
```

Cálculo de créditos cursados y aprobados.

- NCC_UNO: Calcula los créditos cursados por el estudiante.
- NCA_UNO: Calcula los créditos aprobados por el estudiante.
- NCP_UNO: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_UNO, CREDITOS_INDUSTRIAL_UNO):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_UNO"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_UNO, CREDITOS_INDUSTRIAL_UNO):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_UNO"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_UNO"] = industrial["NCC_UNO"] - industrial["NCA_UNO"]
```

Asignación de rendimiento por semestre.
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_UNO"] >= 0) & (industrial["PROMEDIO_UNO"] < 30),
    (industrial["PROMEDIO_UNO"] >= 30) & (industrial["PROMEDIO_UNO"] < 40),
    (industrial["PROMEDIO_UNO"] >= 40) & (industrial["PROMEDIO_UNO"] < 45),
    (industrial["PROMEDIO_UNO"] >= 45) & (industrial["PROMEDIO_UNO"] < 50),
    ]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_UNO"] = pd.Series(pd.cut(industrial["PROMEDIO_UNO"], bins=[0,
30, 40, 45, 50], labels=values))
```

### *3.3.2 Segundo semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del segundo semestre de la carrera.

```
NOTAS_INDUSTRIAL_DOS = [
    "NOTA_ALGEBRA",
    "NOTA_INTEGRAL",
    "NOTA_FISICA_UNO",
    ...
]
VECES_INDUSTRIAL_DOS = [
    "VECES_ALGEBRA",
    "VECES_INTEGRAL",
    "VECES_FISICA_UNO",
    ...
]
```

```
CREDITOS_INDUSTRIAL_DOS = [
    "CREDITOS_ALGEBRA",
    "CREDITOS_INTEGRAL",
    "CREDITOS_FISICA_UNO",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el segundo semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
    for nota_col, veces_col, creditos_col in zip(NOTAS_INDUSTRIAL_DOS,
VECES_INDUSTRIAL_DOS, CREDITOS_INDUSTRIAL_DOS):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_DOS"] = promedios_semestre
```

Cálculo de métricas por semestre.

- CAR_DOS: Cantidad de asignaturas repetidas.
- NAC_DOS: Número de asignaturas cursadas.
- NAA_DOS: Número de asignaturas aprobadas.
- NAP_DOS: Número de asignaturas no aprobadas.
- EPA_DOS: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_DOS"] = industrial[VECES_INDUSTRIAL_DOS].apply(lambda x: (x >
1).sum(), axis=1)
industrial["NAC_DOS"] = industrial[VECES_INDUSTRIAL_DOS].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_DOS"] = industrial[NOTAS_INDUSTRIAL_DOS].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_DOS"] = industrial["NAC_DOS"] - industrial["NAA_DOS"]
industrial["EPA_DOS"] = industrial.apply(lambda row: 1 if row["PROMEDIO_DOS"] < 30 or
row["NAP_DOS"] >= 3 else 2, axis=1
```

Cálculo de créditos cursados y aprobados.

- NCC_DOS: Calcula los créditos cursados por el estudiante.
- NCA_DOS: Calcula los créditos aprobados por el estudiante.
- NCP_DOS: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
```

```
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_DOS,
CREDITOS_INDUSTRIAL_DOS):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_DOS"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_DOS, CREDITOS_INDUSTRIAL_DOS):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_DOS"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_DOS"] = industrial["NCC_DOS"] - industrial["NCA_DOS"]
```

Asignación de rendimiento por semestre.
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_DOS"] >= 0) & (industrial["PROMEDIO_DOS"] < 30),
    (industrial["PROMEDIO_DOS"] >= 30) & (industrial["PROMEDIO_DOS"] < 40),
    (industrial["PROMEDIO_DOS"] >= 40) & (industrial["PROMEDIO_DOS"] < 45),
    (industrial["PROMEDIO_DOS"] >= 45) & (industrial["PROMEDIO_DOS"] < 50),]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_DOS"] = pd.Series(pd.cut(industrial["PROMEDIO_DOS"], bins=[0,
30, 40, 45, 50], labels=values))
```

### *3.3.3 Tercer semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del tercer semestre de la carrera.

```
NOTAS_INDUSTRIAL_TRES = [
    "NOTA_MULTIVARIADO",
    "NOTA_ESTADISTICA_UNO",
    "NOTA_TERMODINAMICA",
    ...
]
VECES_INDUSTRIAL_TRES = [
    "VECES_MULTIVARIADO",
    "VECES_ESTADISTICA_UNO",
    "VECES_TERMODINAMICA",
    ...
]
CREDITOS_INDUSTRIAL_TRES = [
    "CREDITOS_MULTIVARIADO",
    "CREDITOS_ESTADISTICA_UNO",
    "CREDITOS_TERMODINAMICA",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el tercer semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
        for   nota_col,   veces_col,   creditos_col   in   zip(NOTAS_INDUSTRIAL_TRES,
VECES_INDUSTRIAL_TRES, CREDITOS_INDUSTRIAL_TRES):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_TRES"] = promedios_semestre
```

Cálculo de métricas por semestre.

- CAR_TRES: Cantidad de Asignaturas Repetidas
- NAC_TRES: Numero de Asignaturas Cursadas.
- NAA_TRES: Numero de Asignaturas Aprobadas.
- NAP_TRES: Numero de Asignaturas no aprobadas.
- EPA_TRES: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_TRES"] = industrial[VECES_INDUSTRIAL_TRES].apply(lambda x: (x >
1).sum(), axis=1)
industrial["NAC_TRES"] = industrial[VECES_INDUSTRIAL_TRES].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_TRES"] = industrial[NOTAS_INDUSTRIAL_TRES].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_TRES"] = industrial["NAC_TRES"] - industrial["NAA_TRES"]
industrial["EPA_TRES"] = industrial.apply(lambda row: 1 if row["PROMEDIO_TRES"] < 30
or row["NAP_TRES"] >= 3 else 2, axis=1)
```

Cálculo de créditos cursados y aprobados.

- NCC_TRES: Calcula los créditos cursados por el estudiante.
- NCA_TRES: Calcula los créditos aprobados por el estudiante.
- NCP_ TRES: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_TRES,
CREDITOS_INDUSTRIAL_TRES):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_TRES"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
```

```
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_TRES,
CREDITOS_INDUSTRIAL_TRES):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_TRES"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_TRES"] = industrial["NCC_TRES"] - industrial["NCA_TRES"]
```

Asignación de rendimiento por semestre
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_TRES"] >= 0) & (industrial["PROMEDIO_TRES"] < 30),
    (industrial["PROMEDIO_TRES"] >= 30) & (industrial["PROMEDIO_TRES"] < 40),
    (industrial["PROMEDIO_TRES"] >= 40) & (industrial["PROMEDIO_TRES"] < 45),
    (industrial["PROMEDIO_TRES"] >= 45) & (industrial["PROMEDIO_TRES"] < 50),]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_TRES"] = pd.Series(pd.cut(industrial["PROMEDIO_TRES"],
bins=[0, 30, 40, 45, 50], labels=values))
```

### *3.3.4 Cuarto semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del cuarto semestre de la carrera.

```
NOTAS_INDUSTRIAL_CUATRO = [
    "NOTA_ECONOMIA_UNO",
    "NOTA_ECUACIONES",
    "NOTA_ESTADISTICA_DOS",
    ...
]
VECES_INDUSTRIAL_CUATRO = [
    "VECES_ECONOMIA_UNO",
    "VECES_ECUACIONES",
    "VECES_ESTADISTICA_DOS",
    ...
]
CREDITOS_INDUSTRIAL_CUATRO = [
    "CREDITOS_ECONOMIA_UNO",
    "CREDITOS_ECUACIONES",
    "CREDITOS_ESTADISTICA_DOS",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el cuarto semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
```

```
    creditos_estudiante = []
    for nota_col, veces_col, creditos_col in zip(NOTAS_INDUSTRIAL_CUATRO,
VECES_INDUSTRIAL_CUATRO, CREDITOS_INDUSTRIAL_CUATRO):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_CUATRO"] = promedios_semestre
```

Cálculo de métricas por semestre

- CAR_CUARTO: Cantidad de asignaturas repetidas
- NAC_ CUARTO: Numero de asignaturas cursadas.
- NAA_ CUARTO: Numero de asignaturas aprobadas.
- NAP_ CUARTO: Numero de asignaturas no aprobadas.
- EPA_ CUARTO: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_CUATRO"] = industrial[VECES_INDUSTRIAL_CUATRO].apply(lambda x: (x >
1).sum(), axis=1)
industrial["NAC_CUATRO"] = industrial[VECES_INDUSTRIAL_CUATRO].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_CUATRO"] = industrial[NOTAS_INDUSTRIAL_CUATRO].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_CUATRO"] = industrial["NAC_CUATRO"] - industrial["NAA_CUATRO"]
industrial["EPA_CUATRO"] = industrial.apply(lambda row: 1 if row["PROMEDIO_CUATRO"] <
30 or row["NAP_CUATRO"] >= 3 else 2, axis=1)
```

Cálculo de créditos cursados y aprobados

- NCC_CUARTO: Calcula los créditos cursados por el estudiante.
- NCA_CUARTO: Calcula los créditos aprobados por el estudiante.
- NCP_CUARTO: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_CUATRO,
CREDITOS_INDUSTRIAL_CUATRO):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_CUATRO"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_CUATRO,
CREDITOS_INDUSTRIAL_CUATRO):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_CUATRO"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_CUATRO"] = industrial["NCC_CUATRO"] - industrial["NCA_CUATRO"]
```

Asignación de rendimiento por semestre.
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_CUATRO"] >= 0) & (industrial["PROMEDIO_CUATRO"] < 30),
    (industrial["PROMEDIO_CUATRO"] >= 30) & (industrial["PROMEDIO_CUATRO"] < 40),
    (industrial["PROMEDIO_CUATRO"] >= 40) & (industrial["PROMEDIO_CUATRO"] < 45),
    (industrial["PROMEDIO_CUATRO"] >= 45) & (industrial["PROMEDIO_CUATRO"] < 50)]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_CUATRO"] = pd.Series(pd.cut(industrial["PROMEDIO_CUATRO"],
bins=[0, 30, 40, 45, 50], labels=values))
```

### *3.3.5 Quinto semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del quinto semestre de la carrera.

```
NOTAS_INDUSTRIAL_CINCO = [
    "NOTA_METODOLOGIA",
    "NOTA_FISICA_TRES",
    "NOTA_CONTABILIDAD",
    ...
]
VECES_INDUSTRIAL_CINCO = [
    "VECES_METODOLOGIA",
    "VECES_FISICA_TRES",
    "VECES_CONTABILIDAD",
    ...
]
CREDITOS_INDUSTRIAL_CINCO = [
    "CREDITOS_METODOLOGIA",
    "CREDITOS_FISICA_TRES",
    "CREDITOS_CONTABILIDAD",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el quinto semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
    for nota_col, veces_col, creditos_col in zip(NOTAS_INDUSTRIAL_CINCO,
VECES_INDUSTRIAL_CINCO, CREDITOS_INDUSTRIAL_CINCO):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
```

```
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_CINCO"] = promedios_semestre
```

Cálculo de métricas por semestre.

- CAR_CINCO: Cantidad de asignaturas repetidas
- NAC_CINCO: Numero de asignaturas cursadas.
- NAA_CINCO: Numero de asignaturas aprobadas.
- NAP_CINCO: Numero de asignaturas no aprobadas.
- EPA_CINCO: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_CINCO"] = industrial[VECES_INDUSTRIAL_CINCO].apply(lambda x: (x >
1).sum(), axis=1)
industrial["NAC_CINCO"] = industrial[VECES_INDUSTRIAL_CINCO].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_CINCO"] = industrial[NOTAS_INDUSTRIAL_CINCO].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_CINCO"] = industrial["NAC_CINCO"] - industrial["NAA_CINCO"]
industrial["EPA_CINCO"] = industrial.apply(lambda row: 1 if row["PROMEDIO_CINCO"] <
30 or row["NAP_CINCO"] >= 3 else 2, axis=1)
```

Cálculo de créditos cursados y aprobados

- NCC_CINCO: Calcula los créditos cursados por el estudiante.
- NCA_CINCO: Calcula los créditos aprobados por el estudiante.
- NCP_CINCO: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_CINCO,
CREDITOS_INDUSTRIAL_CINCO):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_CINCO"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_CINCO,
CREDITOS_INDUSTRIAL_CINCO):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_CINCO"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_CINCO"] = industrial["NCC_CINCO"] - industrial["NCA_CINCO"]
```

Asignación de rendimiento por semestre.
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_CINCO"] >= 0) & (industrial["PROMEDIO_CINCO"] < 30),
    (industrial["PROMEDIO_CINCO"] >= 30) & (industrial["PROMEDIO_CINCO"] < 40),
    (industrial["PROMEDIO_CINCO"] >= 40) & (industrial["PROMEDIO_CINCO"] < 45),
    (industrial["PROMEDIO_CINCO"] >= 45) & (industrial["PROMEDIO_CINCO"] < 50),]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_CINCO"] = pd.Series(pd.cut(industrial["PROMEDIO_CINCO"],
bins=[0, 30, 40, 45, 50], labels=values))
```

### *3.3.6 Sexto semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del sexto semestre de la carrera.

```
NOTAS_INDUSTRIAL_SEIS = [
    "NOTA_ECONOMIA_DOS",
    "NOTA_PLINEAL",
    "NOTA_DERECHO",
    ...
]
VECES_INDUSTRIAL_SEIS = [
    "VECES_ECONOMIA_DOS",
    "VECES_PLINEAL",
    "VECES_DERECHO",
    ...
]
CREDITOS_INDUSTRIAL_SEIS = [
    "CREDITOS_ECONOMIA_DOS",
    "CREDITOS_PLINEAL",
    "CREDITOS_DERECHO",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el sexto semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
    for nota_col, veces_col, creditos_col in zip(NOTAS_INDUSTRIAL_SEIS,
VECES_INDUSTRIAL_SEIS, CREDITOS_INDUSTRIAL_SEIS):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_SEIS"] = promedios_semestre
```

Cálculo de métricas por semestre

- CAR_SEXTO: Cantidad de asignaturas repetidas

- NAC_SEXTO: Numero de asignaturas cursadas.
- NAA_SEXTO: Numero de asignaturas aprobadas.
- NAP_SEXTO: Numero de asignaturas no aprobadas.
- EPA_SEXTO: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_SEIS"] = industrial[VECES_INDUSTRIAL_SEIS].apply(lambda x: (x >
1).sum(), axis=1)
industrial["NAC_SEIS"] = industrial[VECES_INDUSTRIAL_SEIS].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_SEIS"] = industrial[NOTAS_INDUSTRIAL_SEIS].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_SEIS"] = industrial["NAC_SEIS"] - industrial["NAA_SEIS"]
industrial["EPA_SEIS"] = industrial.apply(lambda row: 1 if row["PROMEDIO_SEIS"] < 30
or row["NAP_SEIS"] >= 3 else 2, axis=1)
```

Cálculo de créditos cursados y aprobados

- NCC_SEXTO: Calcula los créditos cursados por el estudiante.
- NCA_SEXTO: Calcula los créditos aprobados por el estudiante.
- NCP_SEXTO: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_SEIS,
CREDITOS_INDUSTRIAL_SEIS):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_SEIS"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_SEIS,
CREDITOS_INDUSTRIAL_SEIS):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_SEIS"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_SEIS"] = industrial["NCC_SEIS"] - industrial["NCA_SEIS"]
```

Asignación de rendimiento por semestre.
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_SEIS"] >= 0) & (industrial["PROMEDIO_SEIS"] < 30),
    (industrial["PROMEDIO_SEIS"] >= 30) & (industrial["PROMEDIO_SEIS"] < 40),
    (industrial["PROMEDIO_SEIS"] >= 40) & (industrial["PROMEDIO_SEIS"] < 45),
    (industrial["PROMEDIO_SEIS"] >= 45) & (industrial["PROMEDIO_SEIS"] < 50),]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_SEIS"] = pd.Series(pd.cut(industrial["PROMEDIO_SEIS"],
bins=[0, 30, 40, 45, 50], labels=values))
```

### *3.3.7 Séptimo semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del séptimo semestre de la carrera.

```
NOTAS_INDUSTRIAL_SIETE = [
    "NOTA_EMPRENDIMIENTO",
    "NOTA_GRAFOS",
    "NOTA_CALIDAD_UNO",
    ...
]
VECES_INDUSTRIAL_SIETE = [
    "VECES_EMPRENDIMIENTO",
    "VECES_GRAFOS",
    "VECES_CALIDAD_UNO",
    ...
]
CREDITOS_INDUSTRIAL_SIETE = [
    "CREDITOS_EMPRENDIMIENTO",
    "CREDITOS_GRAFOS",
    "CREDITOS_CALIDAD_UNO",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el séptimo semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
    for nota_col, veces_col, creditos_col in zip(NOTAS_INDUSTRIAL_SIETE,
VECES_INDUSTRIAL_SIETE, CREDITOS_INDUSTRIAL_SIETE):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_SIETE"] = promedios_semestre
```

Cálculo de métricas por semestre.

- CAR_SIETE: Cantidad de asignaturas repetidas.
- NAC_SIETE: Numero de asignaturas cursadas.
- NAA_SIETE: Numero de asignaturas aprobadas.
- NAP_SIETE: Numero de asignaturas no aprobadas.
- EPA_SIETE: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_SIETE"] = industrial[VECES_INDUSTRIAL_SIETE].apply(lambda x: (x >
1).sum(), axis=1)
```

```
industrial["NAC_SIETE"] = industrial[VECES_INDUSTRIAL_SIETE].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_SIETE"] = industrial[NOTAS_INDUSTRIAL_SIETE].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_SIETE"] = industrial["NAC_SIETE"] - industrial["NAA_SIETE"]
industrial["EPA_SIETE"] = industrial.apply(lambda row: 1 if row["PROMEDIO_SIETE"] <
30 or row["NAP_SIETE"] >= 3 else 2, axis=1)
```

Cálculo de créditos cursados y aprobados.

- NCC_SIETE: Calcula los créditos cursados por el estudiante.
- NCA_SIETE: Calcula los créditos aprobados por el estudiante.
- NCP_SIETE: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_SIETE,
CREDITOS_INDUSTRIAL_SIETE):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_SIETE"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_SIETE,
CREDITOS_INDUSTRIAL_SIETE):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_SIETE"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_SIETE"] = industrial["NCC_SIETE"] - industrial["NCA_SIETE"]
```

Asignación de rendimiento por semestre
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_SIETE"] >= 0) & (industrial["PROMEDIO_SIETE"] < 30),
    (industrial["PROMEDIO_SIETE"] >= 30) & (industrial["PROMEDIO_SIETE"] < 40),
    (industrial["PROMEDIO_SIETE"] >= 40) & (industrial["PROMEDIO_SIETE"] < 45),
    (industrial["PROMEDIO_SIETE"] >= 45) & (industrial["PROMEDIO_SIETE"] < 50)]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_SIETE"] = pd.Series(pd.cut(industrial["PROMEDIO_SIETE"],
bins=[0, 30, 40, 45, 50], labels=values))
```

### *3.3.8 Octavo semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del octavo semestre de la carrera.

```
NOTAS_INDUSTRIAL_OCHO = [
    "NOTA_LOG_UNO",
    "NOTA_GOPERACIONES",
```

```
    "NOTA_CALIDAD_DOS",
    ...
]
VECES_INDUSTRIAL_OCHO = [
    "VECES_LOG_UNO",
    "VECES_GOPERACIONES",
    "VECES_CALIDAD_DOS",
    ...
]
CREDITOS_INDUSTRIAL_OCHO = [
    "CREDITOS_LOG_UNO",
    "CREDITOS_GOPERACIONES",
    "CREDITOS_CALIDAD_DOS",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el octavo semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
    for nota_col, veces_col, creditos_col in zip(NOTAS_INDUSTRIAL_OCHO,
VECES_INDUSTRIAL_OCHO, CREDITOS_INDUSTRIAL_OCHO):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_OCHO"] = promedios_semestre
```

Cálculo de métricas por semestre.

- CAR_OCHO: Cantidad de asignaturas repetidas
- NAC_OCHO: Numero de asignaturas cursadas.
- NAA_OCHO: Numero de asignaturas aprobadas.
- NAP_OCHO: Numero de asignaturas no aprobadas.
- EPA_OCHO: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_OCHO"] = industrial[VECES_INDUSTRIAL_OCHO].apply(lambda x: (x >
1).sum(), axis=1)
industrial["NAC_OCHO"] = industrial[VECES_INDUSTRIAL_OCHO].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_OCHO"] = industrial[NOTAS_INDUSTRIAL_OCHO].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_OCHO"] = industrial["NAC_OCHO"] - industrial["NAA_OCHO"]
industrial["EPA_OCHO"] = industrial.apply(lambda row: 1 if row["PROMEDIO_OCHO"] < 30
or row["NAP_OCHO"] >= 3 else 2,axis=1)
```

Cálculo de créditos cursados y aprobados

- NCC_OCHO: Calcula los créditos cursados por el estudiante.
- NCA_OCHO: Calcula los créditos aprobados por el estudiante.
- NCP_OCHO: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_OCHO,
CREDITOS_INDUSTRIAL_OCHO):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_OCHO"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_OCHO,
CREDITOS_INDUSTRIAL_OCHO):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_OCHO"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_OCHO"] = industrial["NCC_OCHO"] - industrial["NCA_OCHO"]
```

Asignación de rendimiento por semestre.
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_OCHO"] >= 0) & (industrial["PROMEDIO_OCHO"] < 30),
    (industrial["PROMEDIO_OCHO"] >= 30) & (industrial["PROMEDIO_OCHO"] < 40),
    (industrial["PROMEDIO_OCHO"] >= 40) & (industrial["PROMEDIO_OCHO"] < 45),
    (industrial["PROMEDIO_OCHO"] >= 45) & (industrial["PROMEDIO_OCHO"] < 50)]

values = [1, 2, 3, 4]
industrial["RENDIMIENTO_OCHO"] = pd.Series(pd.cut(industrial["PROMEDIO_OCHO"],
bins=[0, 30, 40, 45, 50], labels=values))
```

### *3.3.9 Noveno semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del noveno semestre de la carrera.

```
NOTAS_INDUSTRIAL_NUEVE = [
    "NOTA_GRADO_UNO",
    "NOTA_LOG_DOS",
    "NOTA_DECISION",
    ...
]
VECES_INDUSTRIAL_NUEVE = [
    "VECES_GRADO_UNO",
    "VECES_LOG_DOS",
    "VECES_DECISION",
```

```
    ...
]
CREDITOS_INDUSTRIAL_NUEVE = [
    "CREDITOS_GRADO_UNO",
    "CREDITOS_LOG_DOS",
    "CREDITOS_DECISION",
    ...
]
```

Después, se calcula el promedio ponderado para cada estudiante en el noveno semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
    for nota_col, veces_col, creditos_col in zip(NOTAS_INDUSTRIAL_NUEVE,
VECES_INDUSTRIAL_NUEVE, CREDITOS_INDUSTRIAL_NUEVE):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_NUEVE"] = promedios_semestre
```

Cálculo de métricas por semestre.

- CAR_NUEVE: Cantidad de asignaturas repetidas.
- NAC_NUEVE: Numero de asignaturas cursadas.
- NAA_NUEVE: Numero de asignaturas aprobadas.
- NAP_NUEVE: Numero de asignaturas no aprobadas.
- EPA_NUEVE: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_NUEVE"] = industrial[VECES_INDUSTRIAL_NUEVE].apply(lambda x: (x >
1).sum(), axis=1)
industrial["NAC_NUEVE"] = industrial[VECES_INDUSTRIAL_NUEVE].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_NUEVE"] = industrial[NOTAS_INDUSTRIAL_NUEVE].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_NUEVE"] = industrial["NAC_NUEVE"] - industrial["NAA_NUEVE"]
industrial["EPA_NUEVE"] = industrial.apply(lambda row: 1 if row["PROMEDIO_NUEVE"] <
30 or row["NAP_NUEVE"] >= 3 else 2, axis=1)
```

Cálculo de créditos cursados y aprobados

- NCC_NUEVE: Calcula los créditos cursados por el estudiante.
- NCA_NUEVE: Calcula los créditos aprobados por el estudiante.
- NCP_NUEVE: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_NUEVE,
CREDITOS_INDUSTRIAL_NUEVE):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_NUEVE"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_NUEVE,
CREDITOS_INDUSTRIAL_NUEVE):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_NUEVE"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_NUEVE"] = industrial["NCC_NUEVE"] - industrial["NCA_NUEVE"]
```

Asignación de rendimiento por semestre.
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_NUEVE"] >= 0) & (industrial["PROMEDIO_NUEVE"] < 30),
    (industrial["PROMEDIO_NUEVE"] >= 30) & (industrial["PROMEDIO_NUEVE"] < 40),
    (industrial["PROMEDIO_NUEVE"] >= 40) & (industrial["PROMEDIO_NUEVE"] < 45),
    (industrial["PROMEDIO_NUEVE"] >= 45) & (industrial["PROMEDIO_NUEVE"] < 50)]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_NUEVE"] = pd.Series(pd.cut(industrial["PROMEDIO_NUEVE"],
bins=[0, 30, 40, 45, 50], labels=values))
```

### *3.3.10 Decimo semestre*

Se definen las nuevas listas con los nombres de las columnas que contienen las notas, veces cursadas y créditos para diferentes materias del décimo semestre de la carrera.

```
NOTAS_INDUSTRIAL_DIEZ = [
    "NOTA_GRADO_DOS",
    "NOTA_LOG_TRES",
    "NOTA_SIMULACION",
    ...
]
VECES_INDUSTRIAL_DIEZ = [
    "VECES_GRADO_DOS",
    "VECES_LOG_TRES",
    "VECES_SIMULACION",
    ...
]
CREDITOS_INDUSTRIAL_DIEZ = [
    "CREDITOS_GRADO_DOS",
    "CREDITOS_LOG_TRES",
    "CREDITOS_SIMULACION",
    ...]
```

Después, se calcula el promedio ponderado para cada estudiante en el décimo semestre. Si el estudiante cursó la materia al menos una vez y tiene una nota registrada, se agrega la nota y los créditos correspondientes a listas temporales. Luego, se calcula el promedio ponderado. Si el estudiante no cursó ninguna materia, se asigna un promedio de 0.

```
promedios_semestre = []
for index, row in industrial.iterrows():
    notas_estudiante = []
    creditos_estudiante = []
    for nota_col, veces_col, creditos_col in zip(
        NOTAS_INDUSTRIAL_DIEZ, VECES_INDUSTRIAL_DIEZ, CREDITOS_INDUSTRIAL_DIEZ
    ):
        if row[veces_col] > 0 and not pd.isnull(row[nota_col]):
            notas_estudiante.append(row[nota_col])
            creditos_estudiante.append(row[creditos_col])
    if len(notas_estudiante) > 0:
        promedio_semestre = np.average(notas_estudiante, weights=creditos_estudiante)
    else:
        promedio_semestre = 0
    promedios_semestre.append(promedio_semestre)
industrial["PROMEDIO_DIEZ"] = promedios_semestre
```

Cálculo de métricas por semestre.

- CAR_DIEZ: Cantidad de asignaturas repetidas.
- NAC_DIEZ: Numero de asignaturas cursadas.
- NAA_DIEZ: Numero de asignaturas aprobadas.
- NAP_DIEZ: Numero de asignaturas no aprobadas.
- EPA_DIEZ: Asigna un valor de 1 si el promedio es menor a 30 o si el número de materias no aprobadas es mayor o igual a 3, y 2 en caso contrario.

```
industrial["CAR_DIEZ"] = industrial[VECES_INDUSTRIAL_DIEZ].apply(lambda x: (x >
1).sum(), axis=1)
industrial["NAC_DIEZ"] = industrial[VECES_INDUSTRIAL_DIEZ].apply(lambda x: (x >
0).sum(), axis=1)
industrial["NAA_DIEZ"] = industrial[NOTAS_INDUSTRIAL_DIEZ].apply(lambda x: (x >=
30).sum(), axis=1)
industrial["NAP_DIEZ"] = industrial["NAC_DIEZ"] - industrial["NAA_DIEZ"]
industrial["EPA_DIEZ"] = industrial.apply(lambda row: 1 if row["PROMEDIO_DIEZ"] < 30
or row["NAP_DIEZ"] >= 3 else 2, axis=1)
```

Cálculo de créditos cursados y aprobados.

- NCC_DIEZ: Calcula los créditos cursados por el estudiante.
- NCA_DIEZ: Calcula los créditos aprobados por el estudiante.
- NCP_ DIEZ: Calcula los créditos pendientes, es decir, la diferencia entre los créditos cursados y aprobados.

```
def calcular_creditos_cursados(row):
    creditos_cursados = 0
    for veces_col, creditos_col in zip(VECES_INDUSTRIAL_DIEZ,
CREDITOS_INDUSTRIAL_DIEZ):
        if row[veces_col] > 0:
            creditos_cursados += row[creditos_col]
    return creditos_cursados
industrial["NCC_DIEZ"] = industrial.apply(calcular_creditos_cursados, axis=1)
def calcular_creditos_aprobados(row):
```

```
    creditos_aprobados = 0
    for nota_col, creditos_col in zip(NOTAS_INDUSTRIAL_DIEZ,
CREDITOS_INDUSTRIAL_DIEZ):
        if row[nota_col] >= 30:
            creditos_aprobados += row[creditos_col]
    return creditos_aprobados
industrial["NCA_DIEZ"] = industrial.apply(calcular_creditos_aprobados, axis=1)
industrial["NCP_DIEZ"] = industrial["NCC_DIEZ"] - industrial["NCA_DIEZ"]
```

Asignación de rendimiento por semestre.
Se asigna un valor de rendimiento basado en el promedio del estudiante:

- 1: Promedio entre 0 y 30.
- 2: Promedio entre 30 y 40.
- 3: Promedio entre 40 y 45.
- 4: Promedio entre 45 y 50.

```
conditions = [
    (industrial["PROMEDIO_DIEZ"] >= 0) & (industrial["PROMEDIO_DIEZ"] < 30),
    (industrial["PROMEDIO_DIEZ"] >= 30) & (industrial["PROMEDIO_DIEZ"] < 40),
    (industrial["PROMEDIO_DIEZ"] >= 40) & (industrial["PROMEDIO_DIEZ"] < 45),
    (industrial["PROMEDIO_DIEZ"] >= 45) & (industrial["PROMEDIO_DIEZ"] < 50)]
values = [1, 2, 3, 4]
industrial["RENDIMIENTO_DIEZ"] = pd.Series(pd.cut(industrial["PROMEDIO_DIEZ"],
bins=[0, 30, 40, 45, 50], labels=values))
```

# 4. Módulo Analítica Diagnóstica

Antes de explorar las diversas formas de obtener estadísticas de un conjunto de datos, es fundamental comprender la naturaleza de las variables que constituyen dicho conjunto. En el ámbito de la estadística y el análisis de datos, las variables pueden clasificarse en diferentes tipos. Esencialmente, estas clasificaciones incluyen variables numéricas, variables categóricas y variables ordinales. Las variables numéricas representan cantidades numéricas y pueden subdividirse en variables continuas y discretas. Por otro lado, las variables categóricas contienen categorías o etiquetas sin un orden específico y se denominan nominales, mientras que las variables ordinales reflejan categorías con un orden inherente. Esta comprensión inicial de las variables sienta las bases para aplicar métodos estadísticos adecuados y realizar análisis significativos sobre el conjunto de datos.

Con este módulo se espera que el usuario pueda identificar mediante el cruce de variables las relaciones que surgen entre ellas. Para esto se propone un módulo que permite por carrera y por semestre crear un gráfico de barras que muestre la tendencia de los datos según sean de interés para el usuario.

## 4.1 Carga y Transformación de Datos

Se define una función para la carga de datos en formato ".csv" separados por medio de ";", y realiza varias transformaciones en los datos cargados utilizando métodos de pandas y numpy para reemplazar códigos numéricos con etiquetas descriptivas en las columnas:

- "GENERO"
- "CALENDARIO"
- "TIPO_COLEGIO"
- "LOCALIDAD_COLEGIO"
- "DEPARTAMENTO"
- "LOCALIDAD"
- "INSCRIPCION"
- "MUNICIPIO"

Después de realizada la transformación se devuelve el DataFrame "datos" transformado.

```
def cargar_datos(carrera, semestre):
    ruta_archivo = f"C:/Users/Intevo/Desktop/UNIVERSIDAD DISTRITAL PROYECTO
FOLDER/UNIVERSIDAD-DISTRITAL-
PROYECTO/MODULO_ANALITICA_DIAGNOSTICA/DATOS/{carrera}{semestre}.csv"
    datos = pd.read_csv(ruta_archivo, sep=";")
    datos['GENERO'] = datos['GENERO'].replace({0:'MASCULINO', 1:'FEMENINO'})
    datos['GENERO']= np.where(datos['GENERO']==2,'FEMENINO',datos['GENERO'])
    datos['CALENDARIO'] = datos['CALENDARIO'].replace({0:'NO REGISTRA',0:'O', 1:'A',
2:'B', 3:'F'})

    datos['TIPO_COLEGIO'] = datos['TIPO_COLEGIO'].replace({0:'NO REGISTRA',
1:'OFICIAL', 2:'NO OFICIAL'})

    datos['LOCALIDAD_COLEGIO'] = datos['LOCALIDAD_COLEGIO'].replace({0:'NO REGISTRA',
1:'USAQUEN', 2:'CHAPINERO', 3:'SANTA FE', 3:'SANTAFE', 4:'SAN CRISTOBAL', 5:'USME',
6:'TUNJUELITO', 7:'BOSA', 8:'KENNEDY', 9:'FONTIBON', 10:'ENGATIVA',11: 'SUBA',12:
'BARRIOS UNIDOS', 13:'TEUSAQUILLO',14: 'LOS MARTIRES',15: 'ANTONIO NARINO',
16:'PUENTE ARANDA', 17: 'LA CANDELARIA', 18:'RAFAEL URIBE URIBE', 18:'RAFAEL URIBE',
19:'CIUDAD BOLIVAR', 20:'FUERA DE BOGOTA', 20:'SIN LOCALIDAD', 21:'SOACHA',22:'FUERA
DE BOGOTA'})

    datos["DEPARTAMENTO"] = np.where((datos["DEPARTAMENTO"] >= 34), "NO REGISTRA",
datos["DEPARTAMENTO"])
```

```
    datos["DEPARTAMENTO"] = datos["DEPARTAMENTO"].replace({"0": "NO REGISTRA", "1":
"AMAZONAS", "2": "ANTIOQUIA", "3": "ARAUCA", "4": "ATLANTICO", "5": "BOGOTA", "6":
"BOLIVAR", "7": "BOYACA", "8": "CALDAS", "9": "CAQUETA", "10": "CASANARE", "11":
"CAUCA", "12": "CESAR", "13": "CHOCO", "14": "CORDOBA", "15": "CUNDINAMARCA", "16":
"GUAINIA", "17": "GUAVIARE", "18": "HUILA", "19": "LA GUAJIRA", "20": "MAGDALENA",
"21": "META", "22": "NARINO", "23": "NORTE SANTANDER", "24": "PUTUMAYO", "25":
"QUINDIO", "26": "RISARALDA", "27": "SAN ANDRES Y PROVIDENCIA", "28": "SANTANDER",
"29": "SUCRE", "30": "TOLIMA", "31": "VALLE", "32": "VAUPES", "33": "VICHADA",})

    datos["LOCALIDAD"] = np.where((datos["LOCALIDAD"] >= 20), "FUERA DE BOGOTA",
datos["LOCALIDAD"])
    datos["LOCALIDAD"] = datos["LOCALIDAD"].replace({"0": "NO REGISTRA", "1":
"USAQUEN", "2": "CHAPINERO", "3": "SANTA FE", "4": "SAN CRISTOBAL", "5": "USME", "6":
"TUNJUELITO", "7": "BOSA", "8": "KENNEDY", "9": "FONTIBON", "10": "ENGATIVA", "11":
"SUBA", "12": "BARRIOS UNIDOS", "13": "TEUSAQUILLO", "14": "LOS MARTIRES", "15":
"ANTONIO NARINO", "16": "PUENTE ARANDA", "17": "LA CANDELARIA", "18": "RAFAEL URIBE
URIBE", "19": "CIUDAD BOLIVAR",})

    datos["INSCRIPCION"] = np.where(datos["INSCRIPCION"] > 11, "NO REGISTRA",
datos["INSCRIPCION"])
    datos["INSCRIPCION"] = datos["INSCRIPCION"].replace({"0": "NO REGISTRA", "1":
"BENEFICIARIOS LEY 1081 DE 2006", "2": "BENEFICIARIOS LEY 1084 DE 2006", "3":
"CONVENIO ANDRES BELLO", "4": "DESPLAZADOS", "5": "INDIGENAS", "6": "MEJORES
BACHILLERES COL. DISTRITAL OFICIAL", "7": "MINORIAS ETNICAS Y CULTURALES", "8":
"MOVILIDAD ACADEMICA INTERNACIONAL", "9": "NORMAL", "10": "TRANSFERENCIA EXTERNA",
"11": "TRANSFERENCIA INTERNA",})

    datos["MUNICIPIO"] = np.where((datos["MUNICIPIO"] >= 225), "NO REGISTRA",
datos["MUNICIPIO"])
    datos["MUNICIPIO"] = datos["MUNICIPIO"].replace({"0": "NO REGISTRA", "1":
"ACACIAS", "2": "AGUACHICA", "3": "AGUAZUL", "4": "ALBAN", "5": "ALBAN (SAN JOSE)",
"6": "ALVARADO", "7": "ANAPOIMA", "8": "ANOLAIMA", "9": "APARTADO", "10": "ARAUCA",
"11": "ARBELAEZ", "12": "ARMENIA", "13": "ATACO", "14": "BARRANCABERMEJA", "15":
"BARRANQUILLA", "16": "BELEN DE LOS ANDAQUIES", "17": "BOAVITA", "18": "BOGOTA",
"19": "BOJACA", "20": "BOLIVAR", "21": "BUCARAMANGA", "22": "BUENAVENTURA", "23":
"CABUYARO", "24": "CACHIPAY", "25": "CAICEDONIA", "26": "CAJAMARCA", "27": "CAJICA",
"28": "CALAMAR", "29": "CALARCA", "30": "CALI", "31": "CAMPOALEGRE", "32":
"CAPARRAPI", "33": "CAQUEZA", "34": "CARTAGENA", "35": "CASTILLA LA NUEVA", "36":
"CERETE", "37": "CHAPARRAL", "38": "CHARALA", "39": "CHIA", "40": "CHIPAQUE", "41":
"CHIQUINQUIRA", "42": "CHOACHI", "43": "CHOCONTA", "44": "CIENAGA", "45": "CIRCASIA",
"46": "COGUA", "47": "CONTRATACION", "48": "COTA", "49": "CUCUTA", "50": "CUMARAL",
"51": "CUMBAL", "52": "CURITI", "53": "CURUMANI", "54": "DUITAMA", "55": "EL BANCO",
"56": "EL CARMEN DE BOLIVAR", "57": "EL COLEGIO", "58": "EL CHARCO", "59": "EL
DORADO", "60": "EL PASO", "61": "EL ROSAL", "62": "ESPINAL", "63": "FACATATIVA",
"64": "FLORENCIA", "65": "FLORIDABLANCA", "66": "FOMEQUE", "67": "FONSECA", "68":
"FORTUL", "69": "FOSCA", "70": "FUNZA", "71": "FUSAGASUGA", "72": "GACHETA", "73":
"GALERAS (NUEVA GRANADA)", "74": "GAMA", "75": "GARAGOA", "76": "GARZON", "77":
"GIGANTE", "78": "GIRARDOT", "79": "GRANADA", "80": "GUACHUCAL", "81": "GUADUAS",
"82": "GUAITARILLA", "83": "GUAMO", "84": "GUASCA", "85": "GUATEQUE", "86":
"GUAYATA", "87": "GUTIERREZ", "88": "IBAGUE", "89": "INIRIDA", "90": "INZA", "91":
"IPIALES", "92": "ITSMINA", "93": "JENESANO", "94": "LA CALERA", "95": "LA DORADA",
"96": "LA MESA", "97": "LA PLATA", "98": "LA UVITA", "99": "LA VEGA", "100":
"LIBANO", "101": "LOS PATIOS", "102": "MACANAL", "103": "MACHETA", "104": "MADRID",
"105": "MAICAO", "106": "MALAGA", "107": "MANAURE BALCON DEL 12", "108": "MANIZALES",
"109": "MARIQUITA", "110": "MEDELLIN", "111": "MEDINA", "112": "MELGAR", "113":
```

```
"MITU", "114": "MOCOA", "115": "MONTERIA", "116": "MONTERREY", "117": "MOSQUERA",
"118": "NATAGAIMA", "119": "NEIVA", "120": "NEMOCON", "121": "OCANA", "122": "ORITO",
"123": "ORTEGA", "124": "PACHO", "125": "PAEZ (BELALCAZAR)", "126": "PAICOL", "127":
"PAILITAS", "128": "PAIPA", "129": "PALERMO", "130": "PALMIRA", "131": "PAMPLONA",
"132": "PANDI", "133": "PASCA", "134": "PASTO", "135": "PAZ DE ARIPORO", "136": "PAZ
DE RIO", "137": "PITALITO", "138": "POPAYAN", "139": "PUENTE NACIONAL", "140":
"PUERTO ASIS", "141": "PUERTO BOYACA", "142": "PUERTO LOPEZ", "143": "PUERTO SALGAR",
"144": "PURIFICACION", "145": "QUETAME", "146": "QUIBDO", "147": "RAMIRIQUI", "148":
"RICAURTE", "149": "RIOHACHA", "150": "RIVERA", "151": "SABOYA", "152": "SAHAGUN",
"153": "SALDAÑA", "154": "SAMACA", "155": "SAMANA", "156": "SAN AGUSTIN", "157": "SAN
ANDRES", "158": "SAN BERNARDO", "159": "SAN EDUARDO", "160": "SAN FRANCISCO", "161":
"SAN GIL", "162": "SAN JOSE DEL FRAGUA", "163": "SAN JOSE DEL GUAVIARE", "164": "SAN
LUIS DE PALENQUE", "165": "SAN MARCOS", "166": "SAN MARTIN", "167": "SANDONA", "168":
"SAN VICENTE DEL CAGUAN", "169": "SANTA MARTA", "170": "SANTA SOFIA", "171":
"SESQUILE", "172": "SIBATE", "173": "SIBUNDOY", "174": "SILVANIA", "175": "SIMIJACA",
"176": "SINCE", "177": "SINCELEJO", "178": "SOACHA", "179": "SOATA", "180":
"SOCORRO", "181": "SOGAMOSO", "182": "SOLEDAD", "183": "SOPO", "184": "SORACA",
"185": "SOTAQUIRA", "186": "SUAITA", "187": "SUBACHOQUE", "188": "SUESCA", "189":
"SUPATA", "190": "SUTAMARCHAN", "191": "SUTATAUSA", "192": "TABIO", "193": "TAMESIS",
"194": "TARQUI", "195": "TAUSA", "196": "TENA", "197": "TENJO", "198": "TESALIA",
"199": "TIBANA", "200": "TIMANA", "201": "TOCANCIPA", "202": "TUBARA", "203":
"TULUA", "204": "TUMACO", "205": "TUNJA", "206": "TURBACO", "207": "TURMEQUE", "208":
"UBATE", "209": "UMBITA", "210": "UNE", "211": "VALLEDUPAR", "212": "VELEZ", "213":
"VENADILLO", "214": "VENECIA (OSPINA PEREZ)", "215": "VILLA DE LEYVA", "216":
"VILLAHERMOSA", "217": "VILLANUEVA", "218": "VILLAPINZON", "219": "VILLAVICENCIO",
"220": "VILLETA", "221": "YACOPI", "222": "YOPAL", "223": "ZIPACON", "224":
"ZIPAQUIRA"})
    return datos
```

## 4.2 Declaración de Variables Globales

Estas líneas inicializan variables globales que se usarán para almacenar los valores recibidos a través de las solicitudes.

```
carrera = ""
semestre = ""
x = ""
y = ""
z = ""
```

Esta clase define la estructura esperada de los datos que se recibirán en las solicitudes.
Tiene cinco campos:

- carrera: Debe ser una cadena de texto (str) que representa la carrera.
- semestre: Debe ser un entero (int) que representa el semestre.
- x, y, z: Todos deben ser cadenas de texto (str) que representan valores específicos que se utilizarán en el procesamiento posterior.

```
class InputData(BaseModel):
    carrera: str
    semestre: int
    x: str
    y: str
    z: str
```

- def procesar_datos(data: InputData): Esta es una función asincrónica que toma un parámetro data de tipo InputData.

- global x, y, z, carrera, semestre: Declara que las variables x, y, z, carrera y semestre se manejarán como globales dentro de la función.
- carrera = data.carrera, semestre = data.semestre, x = data.x, y = data.y, z = data.z: Asigna los valores recibidos en el objeto data a las variables globales correspondientes. Esto permite que las variables globales se actualicen con los datos recibidos en cada llamada a esta función.

```
def procesar_datos(data: InputData):
    global x, y, z, carrera, semestre
carrera = data.carrera
semestre = data.semestre
x = data.x
y = data.y
z = data.z
```

## 4.3 Creación de Gráficos

Parámetros de entrada:

- x, y, z: Son los nombres de las columnas del DataFrame que se utilizarán para generar el gráfico.
- df: Es el DataFrame que contiene los datos que se utilizarán para generar el gráfico.

```
def generar_grafico(x, y, z, df):
    print(x, y, z, df)
```

### *4.3.1 Configuración de Seaborn:*

- sns.set_theme(style="whitegrid"): Configura el tema de Seaborn para utilizar un fondo blanco con líneas de cuadrícula.
- custom_palette: Es una lista de códigos de colores hexadecimales que se utilizarán para colorear las barras del gráfico.

```
# Configurar el tema de Seaborn
sns.set_theme(style="whitegrid")
custom_palette = ["#8c1919", "#fdb400","#000"]
```

### *4.3.2 Creación del gráfico:*

- sns.catplot(...): Esta función de Seaborn crea un gráfico de barras (kind="bar") con las siguientes especificaciones:
- data=df: Utiliza el dataframe df como fuente de datos.
- x=x, y=y, hue=z: Define las variables x, y y z para el eje x, eje y y agrupación respectivamente.
- errorbar="sd": Muestra una barra de error que representa la desviación estándar.
- palette=custom_palette: Aplica la paleta de colores personalizada definida anteriormente.
- alpha=0.6: Define la transparencia de las barras.
- height=6, aspect=2: Ajusta la altura y el aspecto del gráfico para mejorar la visualización.

```
# Crear el gráfico utilizando Seaborn
g = sns.catplot(
data=df,kind="bar",x=x,y=y,hue=z,errorbar="sd",palette=custom_palette,
alpha=0.6,height=6,aspect=2)
```

### *4.3.3 Visualización y almacenamiento del gráfico:*

- plt.xticks(rotation=90): Rota las etiquetas del eje x 90 grados para mejorar la legibilidad.
- plt.show(): Muestra el gráfico en la ventana de salida.
- g.savefig(...): Guarda el gráfico generado en un archivo temporal en formato PNG.
- Se lee el archivo temporal y se codifica su contenido en formato Base64.
- El código Base64 se guarda en un diccionario imagen_base64.

```
# Guardar el gráfico en un archivo temporal
plt.xticks(rotation=90) #Rotación 90° de etiquetas eje x
plt.show()
temp_file = io.BytesIO()
g.savefig(temp_file, format="png")
temp_file.seek(0)
base64_image = base64.b64encode(temp_file.read()).decode("utf-8")
imagen_base64 = {"data": base64_image}
```

*4.3.4 Guardado en archivo JSON:*

- ruta_especifica: Especifica la ruta donde se guardará el archivo JSON que contendrá el código Base64 de la imagen.
- json.dump(imagen_base64, json_file): Guarda el diccionario imagen_base64 en el archivo JSON especificado.

```
ruta_especifica = "C:/Users/Intevo/Desktop/UNIVERSIDAD DISTRITAL PROYECTO
FOLDER/UNIVERSIDAD-DISTRITAL-
PROYECTO/MODULO_ANALITICA_DIAGNOSTICA/Imagenes_Transformacion.json"
with open(ruta_especifica, "w") as json_file:
       json.dump(imagen_base64, json_file)
   print(f"Los códigos Base64 de las imágenes han sido guardados en
'{ruta_especifica}'.")
```

*4.3.5 Archivo de variables cruzadas*

- Esta función toma tres nombres de columnas (x, y, z), la carrera y el semestre.
- Filtra el DataFrame para incluir solo las columnas.
- Guarda el resultado filtrado como un archivo CSV en una ruta específica, incluyendo el nombre de la carrera y el semestre en el nombre del archivo.
- Convierte el dataframe filtrado en un diccionario y lo guarda como un archivo JSON en la misma carpeta con el mismo nombre.

```
def generar_archivo(x,y,z,carrera,semestre):
    df_grafico=df[[x,y,z]]
    ruta='C:/Users/Intevo/Desktop/'
    df_grafico.to_csv(f"C:/Users/Intevo/Desktop/UNIVERSIDAD DISTRITAL PROYECTO
FOLDER/UNIVERSIDAD-DISTRITAL-
PROYECTO/MODULO_ANALITICA_DIAGNOSTICA/DATOS_{carrera}{semestre}.csv",index=False)
    data_with_columns = df_grafico.to_dict(orient='records')
    diccionario_dataframes = [
        {
            'dataTransformacion': data_with_columns,
        }
    ]
    ruta = f"C:/Users/Intevo/Desktop/UNIVERSIDAD DISTRITAL PROYECTO
FOLDER/UNIVERSIDAD-DISTRITAL-
PROYECTO/MODULO_ANALITICA_DIAGNOSTICA/DATOS_{carrera}{semestre}.json"
    with open(ruta, "w") as json_file:
        json.dump({"data": diccionario_dataframes}, json_file, indent=4)
    print(f"El archivo JSON ha sido guardado en '{ruta}'.")

    return df_grafico
```

# 5. Módulo Analítica Predictiva Selección de Características

Los métodos de selección de características son herramientas cruciales para identificar y eliminar atributos innecesarios que no mejoran la precisión del modelo y pueden incluso perjudicar su rendimiento. Además, demuestran que no siempre es necesario tener muchos datos o una optimización exhaustiva de hiperparámetros para lograr un buen desempeño.

## 5.1 Carga y Filtrado de Datos

Se realiza la carga de datos por carrera que se toman en formato ".csv" y separados por medio de ";" y se crea una lista nombrada "conjunto_variables" que sera actualizada con las variables seleccionadas para el filtrado del DataFrame.

```
df=pd.read_csv("C:/Users/Intevo/Desktop/electrica.csv", sep=";")
conjunto_variables = []
df = df[conjunto_variables]
```

El DataFrame resultante solo contiene las columnas promedio con valores Mayores o iguales a 5 convertidas a número entero.

```
promedio_columns = df.filter(like="PROMEDIO_").columns
df = df[df[promedio_columns].ge(5).all(axis=1)]
df[promedio_columns] = df[promedio_columns].round(0)
df.astype(int)
df = df.apply(pd.to_numeric)
```

Se crean diferentes copias del DataFrame, esto con el fin de que los cambios que se realizados en las copias no afecten el DataFrame original.

```
X_R = X.copy(deep=True)
X_E = X.copy(deep=True)
X_N = X.copy(deep=True)
X_ROB = X.copy(deep=True)
X_T_BOX = X.copy(deep=True)
X_T_JOHNSON = X.copy(deep=True)
```

## 5.2 Métodos de Filtro

Los métodos de filtro para datos son técnicas de selección de características que evalúan la relevancia de cada variable de entrada respecto a la variable objetivo sin considerar el modelo específico que se utilizará. Su objetivo principal es reducir la dimensionalidad eliminando características que no contribuyen significativamente a la predicción o clasificación. Se basan en medidas estadísticas para asignar una puntuación a cada característica y seleccionar aquellas con la relación más fuerte con la variable objetivo.

### *5.2.1 Correlación de pearson*

La correlación de Pearson mide la relación lineal entre dos variables continuas. Su coeficiente proporciona información sobre la dirección y la fuerza de la relación, con valores que van de -1 a 1, indicando desde una correlación negativa perfecta hasta una positiva perfecta, y 0 indicando ninguna correlación lineal.
A continuación, se configura el estilo de los gráficos de Seaborn a un tema blanco, lo que proporciona un fondo blanco y un aspecto limpio para los gráficos.

```
sns.set(style="white")
```

Se copia el DataFrame "Xpandas_T_JOHNSON" en "df" y se calcula la matriz de correlación de Pearson.

```
df = (Xpandas_T_JOHNSON.copy())
corr = df.corr()
```

Se crea una máscara para la matriz de correlación que cubre la mitad superior, evitando así la duplicación de información en el mapa de calor.

```
mask = np.zeros_like(corr, dtype=bool)
mask[np.triu_indices_from(mask)] = True
```

Se configura la figura de matplotlib Y genera una paleta de colores divergente personalizada. Después, dibuja el mapa de calor con la matriz de correlación, aplicando la máscara y usando la paleta de colores y calcula la correlación absoluta con la columna "REND_UNO".

```
f, ax = plt.subplots(figsize=(10, 10))
cmap = sns.diverging_palette(220, 10, as_cmap=False)
sns.heatmap(corr,mask=mask,cmap=cmap,vmax=0.4,vmin=-0.4,square=True,linewidths=0.5,
ax=ax)
cor_target = abs(corr[REND_UNO])
```

Se selecciona características cuya correlación absoluta con "REND_UNO" es mayor que 0.02. Crea una instancia de "SelectKBest" con "f_regression"como función de puntuación y selecciona las 5 mejores características.

```
características_relevantes = cor_target[cor_target > 0.02]
prueba = SelectKBest(score_func=f_regression, k=5)
```

Se ajusta SelectKBest con los datos de "Xpandas_T_JOHNSON" y la columna "REND_UNO". Posteriormente se establece la precisión decimal de la salida de Numpy a 3.

```
entrenamiento = prueba.fit(Xpandas_T_JOHNSON, REND_UNO)
np.set_printoptions(precision=3)
```

Transforma "Xpandas_T_JOHNSON" para obtener las características seleccionadas.

```
caracteristicas_f = entrenamiento.transform(Xpandas_T_JOHNSON)
```

Obtiene los nombres de las columnas originales.

```
columnas = Xpandas_T_JOHNSON.columns
```

Convierte la matriz de características seleccionadas en un DataFrame de pandas, con nombres de columnas específicos y muestra las primeras dos filas del nuevo DataFrame.

```
cambioN_Xpandas_E=pd.DataFrame(data=caracteristicas_f,columns=["GENERO",
"TIPO_COLEGIO", "LOCALIDAD_COLEGIO", "CALENDARIO", "MUNICIPIO"])
print(cambioN_Xpandas_E.head(2))
```

Crea un conjunto de datos con los nombres de las columnas originales y los coeficientes de puntuación de SelectKBest.

```
Pearson = pd.DataFrame({"Pearson": columnas, "Coeficiente": prueba.scores_})
```

Filtra las filas para mantener solo las características con coeficientes mayores a 1.

```
Pearson = Pearson[Pearson["Coeficiente"] > 1]
```

### *5.2.2 ANOVA*

El Análisis de Varianza (ANOVA) compara las medias de tres o más grupos distintos en un conjunto de datos, descomponiendo la variabilidad total en componentes atribuibles a diversas fuentes de variación. Esta utiliza el estadístico F para evaluar la relación entre la varianza dentro de los grupos.

El primer paso es preparar y ajustar el selector de características. El código obtiene los nombres de las columnas del conjunto de datos "Xpandas_T_JOHNSON" y crea un objeto que utiliza "f_classif" como la función de puntuación, seleccionando las 5 mejores características. Luego, ajusta el selector de

características a los datos “Xpandas_T_JOHNSON” y la variable objetivo “REND_UNO” (este paso no solo será aplicado en la selección ANOVA, Tambien sera usada en los demás métodos).

```
# Asumiendo que Xpandas_T_JOHNSON es un DataFrame de pandas
columnas = Xpandas_T_JOHNSON.columns
# Características del método de prueba
prueba = SelectKBest(score_func=f_classif, k=5)
# Datos para el modelo
entrenamiento = prueba.fit(Xpandas_T_JOHNSON, REND_UNO)
```

El código a continuación configura la precisión de los decimales en la salida a 3 dígitos y transforma el conjunto de datos original para seleccionar las características más importantes. Obtiene los nombres de las características seleccionadas. Crea un nuevo conjunto de datos con las características seleccionadas e imprime las dos primeras filas del conjunto de datos de características seleccionadas.

```
# Decimales de la respuesta = 3
set_printoptions(precision=3)
# Resumen de características seleccionadas como matriz Numpy
caracteristicas_f = entrenamiento.transform(Xpandas_T_JOHNSON)
# Obtener los nombres de las características seleccionadas
caracteristicas_seleccionadas= [columnas[i] for i in prueba.get_support(indices=True)]
cambioN_Xpandas_E=pd.DataFrame(data=caracteristicas_f,columns=caracteristicas_selecci
onadas)cambioN_Xpandas_E.head(2)
Anova = pd.DataFrame({"Anova": columnas, "Coeficiente": prueba.scores_})
Anova = Anova[Anova["Coeficiente"] > 1.3]
```

### *5.2.3 Chi cuadrado*

La prueba de Chi-cuadrado determina si hay una relación significativa entre dos variables categóricas en un conjunto de datos, comparando las frecuencias observadas y esperadas en una tabla de contingencia. Rechaza la hipótesis nula si el estadístico de Chi-cuadrado es mayor que un valor crítico, indicando una asociación significativa entre las variables.

El primer paso es preparar y ajustar el selector de características. El código obtiene los nombres de las columnas del conjunto de datos “Xpandas_N” y crea un objeto que utiliza “chi2” como la función de puntuación, seleccionando las 5 mejores características. Luego, ajusta el selector de características a los datos “Xpandas_N” y la variable objetivo “REND_UNO”. Es importante destacar que “chi2” no funciona con datos negativos, por lo que se asegura de que “Xpandas_N” sea adecuado para esta operación.

```
columnas = Xpandas_N.columns
prueba = SelectKBest(score_func=chi2, k=5)
entrenamiento = prueba.fit(Xpandas_N, REND_UNO)
set_printoptions(precision=3)
print(entrenamiento.scores_)
```

Luego, se transforma el conjunto de datos original para seleccionar las características más importantes. La transformación se realiza con “entrenamiento.transform(Xpandas_N)”, y los nombres de las características seleccionadas se obtienen utilizando “prueba.get_support(indices=True)”. Se crea un nuevo conjunto de datos “cambioN_Xpandas_N” con las características seleccionadas, que se imprimen para verificar las dos primeras filas del nuevo conjunto de datos.

```
caracteristicas_chi = entrenamiento.transform(Xpandas_N)
caracteristicas_seleccionadas=[columnas[i] for i in prueba.get_support(indices=True)]
cambioN_Xpandas_N=pd.DataFrame(data=caracteristicas_chi,columns=caracteristicas_selec
cionadas)
print(cambioN_Xpandas_N.head(2))
```

Finalmente, se muestra el puntaje de cada característica junto con su nombre para facilitar la interpretación. Esto se hace iterando sobre los puntajes y nombres de las columnas. Se crea un conjunto de datos "ChiCuadrado" que contiene los nombres de las características y sus respectivos coeficientes Chi-Cuadrado. Este conjunto de datos se filtra para mantener solo aquellas características con coeficientes mayores a 0.01, proporcionando un resumen claro de las características más relevantes según la prueba de Chi-Cuadrado. (Al igual que con la configuración del selector y la salida de decimales, este paso se utilizará en el resto de métodos).

```
for i in range(len(prueba.scores_)):
    print("%s: %f" % (columnas[i], prueba.scores_[i]))
ChiCuadrado = pd.DataFrame({"ChiCuadrado": columnas, "Coeficiente": prueba.scores_})
ChiCuadrado = ChiCuadrado[ChiCuadrado["Coeficiente"] > 0.01]
```

### *5.2.4 Información Mutua*

La información mutua mide la dependencia o relación entre dos variables, cuantificando cuánta información proporciona una variable sobre otra. Los valores altos indican una mayor dependencia entre las variables.

Después de ajustar el selector y los decimales de la salida a 3 dígitos, se transforma el conjunto de datos original para seleccionar las características más importantes.

```
cambioN_Xpandas_N=pd.DataFrame(data=caracteristicas_mutual,
columns=caracteristicas_seleccionadas)
```

La transformación se realiza con:

```
caracteristicas_mutual = entrenamiento.transform(Xpandas_T_JOHNSON)
```

Y los nombres de las características seleccionadas se obtienen utilizando:

```
caracteristicas_seleccionadas=[columnas[i] for i in prueba.get_support(indices=True)]
```

Se crea un nuevo conjunto de datos "cambioN_Xpandas_N" con las características seleccionadas.

### *5.2.5 RFE – Regresion Logistica*

La Regresión Logística es un algoritmo de aprendizaje supervisado utilizado principalmente para problemas de clasificación binaria. Funciona modelando la relación entre un conjunto de características independientes y una variable dependiente binaria utilizando una función logística. En este proceso se estiman los coeficientes de las características independientes que maximizan la probabilidad de observar los datos de entrenamiento dados. La Regresión Logística utiliza una función sigmoide para convertir las predicciones lineales en probabilidades, permitiendo así clasificar las observaciones en una de las dos categorías posibles. En combinación con técnicas como RFE (Recursive Feature Elimination), se pueden seleccionar las características más relevantes para mejorar la precisión del modelo y reducir la dimensionalidad del conjunto de datos.

Se inicializa y ajusta un modelo de Regresión Logística con un máximo de 10000 iteraciones, y se configura RFE para seleccionar 20 características, transformando el conjunto de datos de entrada.

```
modelo = LogisticRegression(max_iter=10000)
selector = RFE(estimator=modelo, n_features_to_select=20)
entrenamiento = selector.fit(Xpandas_T_JOHNSON, REND_UNO)
caracteristicas_seleccionadas = columnas[entrenamiento.support_]
modelo.fit(Xpandas_T_JOHNSON[caracteristicas_seleccionadas], REND_UNO)
```

Se convierte el soporte booleano en una lista de nombres de las características seleccionadas y se entrena nuevamente el modelo de Regresión Logística utilizando solo las características seleccionadas para obtener los coeficientes. Finalmente, se obtienen los coeficientes del modelo entrenado y el conjunto de datos

"RFEregresionlogistica" se organiza a manera de lista y se filtran las características que tienen una importancia superior a 0.04 de manera descendente.

```
coeficientes = modelo.coef_[0]
RFEregresionlogistica=pd.DataFrame({"RFEregresionlogistica":caracteristicas_seleccionadas, "Coeficiente": coeficientes,})
RFEregresionlogistica = RFEregresionlogistica[RFEregresionlogistica["Coeficiente"] > 0.04]
```

#### *5.2.6 RFE – SVC*

SVC (Support Vector Classifier) es un algoritmo de aprendizaje supervisado utilizado para problemas de clasificación. Su objetivo es encontrar el hiperplano óptimo que separe las clases en el espacio de características, maximizando el margen entre las instancias de diferentes clases. Utiliza un kernel, como el kernel lineal, para transformar los datos y encontrar el hiperplano en un espacio de mayor dimensión si es necesario. La combinación de SVC con Recursive Feature Elimination (RFE) permite seleccionar las características más relevantes al entrenar el modelo iterativamente y eliminar las menos importantes, mejorando así la precisión y eficiencia del modelo.

Después de iniciar y configurar un modelo SVC con un kernel lineal para seleccionar 20 características se ajusta el selector para transformar el conjunto de datos de entrada mediante "RFE".

```
modelo = SVC(kernel="linear")
selector = RFE(estimator=modelo, n_features_to_select=20)
entrenamiento = selector.fit(Xpandas_T_JOHNSON, REND_UNO)
```

Se entrena nuevamente el modelo SVC utilizando solo las características seleccionadas para obtener los coeficientes.

```
modelo.fit(Xpandas_T_JOHNSON[caracteristicas_seleccionadas], REND_UNO)
```

Finalmente, se obtienen los coeficientes del modelo entrenado y el conjunto de datos "RFE_SVC" se organiza a manera de lista y se filtran las características que tienen un coeficiente superior a 0.04 de manera descendente.

```
RFE_SVC = pd.DataFrame({"RFE_SVC": caracteristicas_seleccionadas, "Coeficiente": coeficientes})
RFE_SVC = RFE_SVC[RFE_SVC["Coeficiente"] > 0.04]
```

### 5.3 Métodos de Envoltura (Wrapper)

Los métodos de envoltura seleccionan el mejor grupo de características evaluando diferentes combinaciones y determinando cuáles producen el mejor rendimiento predictivo. Incluyen métodos como la Selección Hacia Adelante, Eliminación Hacia Atrás y Eliminación Recursiva de Características (RFE).

#### *5.3.1 Eliminación de Características Recursivas (RFE)*

RFE elimina recursivamente las características menos importantes hasta alcanzar el número deseado de características o mejorar el rendimiento del modelo, utilizando la importancia de las características proporcionada por el modelo de aprendizaje automático.

#### *5.3.2 Eliminación hacia atrás*

Este método comienza con un modelo que incluye todas las características y elimina iterativamente la menos importante hasta que se alcanza un criterio de parada predefinido.

Primero, se divide el conjunto de datos "Xpandas_T_JOHNSON" y la variable objetivo "REND_UNO" en datos de entrenamiento y prueba. La división se realiza con un tamaño de prueba del 30% y una semilla aleatoria de 2. A continuación, se configura y ajusta el selector de características secuencial hacia atrás

(SFS). Se crea un objeto que utiliza "LinearRegression" como el modelo de regresión, con características en el rango de 1 a 8, configurado para realizar selección hacia atrás, sin selección flotante, utilizando la métrica de "r2" para la evaluación y con una validación cruzada de 10 pliegues.

```
X_trn,X_tst,Y_trn,Y_tst=train_test_split(Xpandas_T_JOHNSON,REND_UNO,test_size=0.3,ran
dom_state=int(2))
sfs1 = SFS(LinearRegression(),k_features=(1, 8), forward=False,floating=False,
       scoring="r2",cv=10,) sfs1.fit(X_trn, Y_trn)
```

El selector se ajusta a los datos de entrenamiento X_trn y Y_trn. Luego, se ajusta nuevamente el selector de características secuencial hacia atrás con una configuración similar, pero con 10 características seleccionadas. Después del ajuste, se obtienen los índices de las columnas y los nombres de las características seleccionadas. Estos se guardan en listas "Numero_columna".

```
modelo, LinearRegression()
sbs = SFS(modelo, k_features=(10), forward=False, floating=False, scoring="r2", cv=10)
sbs.fit(X_trn, Y_trn)
Numero_columna = list(sbs.k_feature_idx_)
Numero_columna = list(sbs.k_feature_names_)
```

Finalmente, se crea un conjunto de datos "EliminacionAtras" que contiene los nombres de las características seleccionadas.

```
EliminacionAtras = pd.DataFrame({"EliminacionAtras": list(sbs.k_feature_names_)})
```

### *5.3.3 Selección hacia adelante*

Comienza con un conjunto vacío de características y agrega iterativamente nuevas características que mejoran el rendimiento del modelo hasta alcanzar un criterio de detención predefinido.

```
modelo, LinearRegression()
sfs = SFS(modelo, k_features=(1, 15), forward=True, floating=False, scoring="r2",
cv=10)
sfs.fit(X_trn, Y_trn)
```

Despues de configurar y ajustar el selector de caracteristicas hacia adelante ("SFS"). Se crea un objeto que utiliza "LinearRegression" como el modelo de regresión, con características en el rango de 1 a 15, configurado para realizar selección hacia adelante, sin selección flotante, utilizando la métrica de "r2" para la evaluación y con una validación cruzada de 10 pliegues. El selector se ajusta a los datos de entrenamiento X_trn y Y_trn.

```
modelo, LinearRegression()
sbs = SFS(modelo, k_features=(9), forward=True, floating=False, scoring="r2", cv=10)
sbs.fit(X_trn, Y_trn)
```

Luego, se ajusta nuevamente el selector de características secuencial hacia adelante con una configuración similar, pero con 9 características seleccionadas y se realiza el mismo proceso de selección y guardado. Finalmente, se crea un conjunto de datos "EliminacionAdelante" que contiene los nombres de las características seleccionadas.

```
EliminacionAdelante=pd.DataFrame({"EliminacionAdelante": list(sbs.k_feature_names_)})
```

### *5.3.4 Eliminación bidireccional*

Combina la Eliminación Hacia Atrás y la Selección Hacia Adelante de manera iterativa para encontrar un conjunto óptimo de características que maximice el rendimiento del modelo. Después de configurar y ajustar el selector de características secuencial hacia adelante y hacia atrás. Se crea un objeto que utiliza "LinearRegression" como el modelo de regresión, con características en el rango de 1 a 15, configurado

para realizar selección hacia atrás, sin selección flotante, utilizando la métrica de "r2" para la evaluación y con una validación cruzada de 10 pliegues. El selector se ajusta a los datos de entrenamiento X_trn y Y_trn.

```
sfs1=SFS(LinearRegression(),k_features=(1,15),forward=False,floating=False,scoring="r
2",cv=10)
sfs1.fit(X_trn, Y_trn)
```

Luego, se ajusta nuevamente el selector de características secuencial con una configuración similar, pero con 10 características seleccionadas y se realiza el mismo proceso de selección y guardado.

```
modelo, LinearRegression()
sbs = SFS(modelo, k_features=(10), forward=False, floating=False, scoring="r2", cv=10)
sbs.fit(X_trn, Y_trn)
```

Finalmente, se crea un conjunto de datos "EliminacionBidirec" que contiene los nombres de las características seleccionadas.

```
EliminacionBidirec=pd.DataFrame({"EliminacionBidirec":list(sbs.k_feature_names_)})
```

## 5.4 Métodos Embebidos

En estos métodos, la selección de características se realiza durante el proceso de entrenamiento del modelo. Utilizan penalizaciones a los coeficientes de las características menos importantes para reducir su influencia en la predicción final, lo que mejora la eficiencia del modelo y reduce el riesgo de sobreajuste.

### *5.4.1 Regresión lineal*

Modelo que relaciona una variable dependiente con una o más variables independientes, estimando coeficientes que minimicen la suma de los errores. Primero, se define el modelo de regresión lineal que se utilizará en el proceso de RFE.

```
modelo = LinearRegression()
```

Se inicializa el modelo RFE especificando el modelo de regresión lineal como el estimador y el número de características a seleccionar (en este caso, 20).

```
selector = RFE(estimator=modelo, n_features_to_select=20)
```

El modelo "RFE" se ajusta al conjunto de datos "Xpandas_T_JOHNSON" y la variable objetivo "REND_UNO".

```
entrenamiento = selector.fit(Xpandas_T_JOHNSON, REND_UNO)
```

A continuación, se extraen los nombres de las características seleccionadas utilizando el soporte booleano del modelo RFE y se convierten en una lista de nombres de características.

```
caracteristicas_seleccionadas = columnas[entrenamiento.support_].tolist()
```

Se entrena nuevamente el modelo de regresión lineal utilizando únicamente las características seleccionadas para obtener los coeficientes de cada una de ellas.

```
modelo.fit(Xpandas_T_JOHNSON[caracteristicas_seleccionadas], REND_UNO)
coeficientes = modelo.coef_
```

Finalmente, se crea un DataFrame para mostrar los coeficientes de correlación de las características seleccionadas. Se filtran las características que tienen un coeficiente mayor a 0.04.

```
RFEregresionlineal = pd.DataFrame(
                    {"RFEregresionlineal":caracteristicas_seleccionadas,"Coeficiente":
coeficientes.flatten()})
RFEregresionlineal = RFEregresionlineal[RFEregresionlineal["Coeficiente"] > 0.04]
```

### *5.4.2 Regularización lasso*

Utiliza la regularización L1 para penalizar los coeficientes de las características menos importantes, forzando algunos a cero y facilitando la selección de características y la reducción de la dimensionalidad. Primero, se define el modelo de validación cruzada usando "RepeatedStratifiedKFold". Este modelo divide los datos en 10 splits y repite el proceso 3 veces, manteniendo la reproducibilidad con "random_state=1". Se inicializa el modelo "LassoCV" con validación cruzada para optimizar automáticamente el parámetro de regularización (alfa). Este modelo se ajusta a los datos "Xpandas_T_JOHNSON" con la variable objetivo "REND_UNO".

```
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
```

Se obtienen los coeficientes del modelo ajustado y se asocian a las columnas del conjunto de datos "Xpandas_T_JOHNSON". Luego, se filtran las características seleccionadas (caracteristicas_seleccionadas) para incluir solo aquellas con coeficientes diferentes de cero y cuyo valor absoluto sea mayor que 0.04.

```
reg = LassoCV(cv=cv)
reg.fit(Xpandas_T_JOHNSON, REND_UNO)
```

Se ordenan y filtran los coeficientes para visualización y se crea un conjunto de datos (RegresionLasso) que muestra las características seleccionadas (RegresionLasso) y sus coeficientes (Coeficiente), proporcionando así una visión clara de las variables más importantes según el modelo de regresión Lasso.

```
coef = pd.Series(reg.coef_, index=Xpandas_T_JOHNSON.columns)
caracteristicas_seleccionadas                                    =
caracteristicas_seleccionadas[caracteristicas_seleccionadas.abs()  >
0.04].index.tolist()
imp_coef = coef[coef != 0].sort_values()
imp_coef_filtered = imp_coef[imp_coef.abs() > 0.04]
RegresionLasso=pd.DataFrame({"RegresionLasso":imp_coef_filtered.index, "Coeficiente":
imp_coef_filtered.values})
```

## 5.5 Métodos de Ensamble

Estos métodos evalúan las características de entrada utilizando un modelo para clasificarlas según su importancia. Incluyen métodos como los Árboles de Clasificación y Regresión (CART), Bosques Aleatorios, y el Aumento de Gradiente.

### *5.5.1 Árboles de Decisión (CART)*

Modelos predictivos que dividen repetidamente los datos en subconjuntos más pequeños para maximizar la homogeneidad dentro de cada subconjunto, utilizando medidas como la ganancia de información, el índice Gini o la entropía. Primero, se definen las columnas del conjunto de datos y se inicializa el modelo de árbol de decisión que se utilizará en el proceso de RFE.
Se inicializa el modelo RFE especificando el modelo de árbol de decisión como el estimador y el número de características a seleccionar (en este caso, 9) y el modelo RFE se ajusta al conjunto de datos "Xpandas_T_JOHNSON" y la variable objetivo "REND_UNO".

```
columnas = Xpandas_T_JOHNSON.columns
model = DecisionTreeClassifier()
rfe = RFE(estimator=model, n_features_to_select=9)
X_rfe = rfe.fit_transform(Xpandas_T_JOHNSON, REND_UNO)
```

Se obtiene la importancia de las características del modelo ajustado (en este caso, del árbol de decisión).

```
importancia_caracteristicas = model.feature_importances_
```

Finalmente, se crea un DataFrame para mostrar la importancia de las características seleccionadas, filtrando aquellas que tienen una importancia mayor a 0.04.

```
RFEarboldecision = pd.DataFrame({"RFEarboldecision": caracteristicas_seleccionadas,
        "Importancia": importancia_caracteristicas,})
RFEarboldecision = RFEarboldecision[RFEarboldecision["Importancia"] > 0.04]
```

### *5.5.2 Bosques aleatorios*

Construyen múltiples árboles de decisión y dividen los nodos utilizando subconjuntos aleatorios de características. Cada árbol vota por una clase particular, y la clase con más votos se selecciona como resultado final. Para empezar, se define "X" como el conjunto de datos de características, "y" como la variable objetivo y se inicializa el modelo Random Forest para clasificación.

```
X = Xpandas_T_JOHNSON
y = REND_UNO
modelo = RandomForestClassifier()
```

Se utiliza RFECV (Recursive Feature Elimination with Cross-Validation) con Random Forest para seleccionar características de manera recursiva. RFECV ajusta el modelo y utiliza validación cruzada para evaluar la precisión.

```
rfecv = RFECV(estimator=modelo, step=1, cv=5, scoring="accuracy")
rfecv.fit(X, y)
```

Se crean dos variables:

- selected_features: Contiene las características seleccionadas basadas en "rfecv.support", que indica qué características fueron seleccionadas durante la eliminación recursiva.
- importances: Contiene las importancias de las características según el estimador.

```
Selected_features = X.columns[rfecv.support_]
importances = rfecv.estimator_.feature_importances_
```

Se crea un conjunto de datos ("randomforest") para visualizar las características seleccionadas y sus importancias, ordenadas por importancia en orden descendente.

```
Randomforest=pd.DataFrame({"randomforest":selected_features,"importance":
importances}).sort_values(by="importance", ascending=False)
```

Finalmente, se filtran las características con importancias mayores que 0.04 y se obtiene una lista con los nombres de las variables más importantes.

```
Randomforest = randomforest[randomforest["importance"] > 0.04]
important_features = randomforest["randomforest"].tolist()
```

### *5.5.3 ExtraTreesClassifier (Árboles extremadamente aleatorios)*

El Bosque Aleatorio, conocido como Random Forest en inglés, es un algoritmo de aprendizaje supervisado que utiliza múltiples árboles de decisión. Cada árbol es construido de manera independiente utilizando subconjuntos aleatorios de características del conjunto de datos. En problemas de clasificación y regresión, este método combina las predicciones de todos los árboles para determinar la salida final. Es eficaz para manejar grandes conjuntos de datos con muchas características, reduciendo el sobreajuste en comparación con un único árbol de decisión gracias a su enfoque de aleatorización en la selección de características y la construcción de árboles. Después de haber asignado los datos de entrada y la variable objetivo, se selecciona "ExtraTreesClassifier" como estimador, configurado con un estado aleatorio definido como 100.

```
X = Xpandas_T_JOHNSON
y = REND_UNO
```

```
modelo = ExtraTreesClassifier(random_state=100)
```

Utilizando el modelo. RFECV selecciona automáticamente las mejores características, evaluando la precisión mediante validación cruzada estratificada con 10 folds y estas son guardadas en un nuevo conjunto de datos.

```
Rfecv = RFECV(estimator=modelo, step=1, cv=StratifiedKFold(10), scoring="accuracy")
rfecv.fit(X, y)
selected_features = X.columns[rfecv.support_]
importances = rfecv.estimator_.feature_importances_
ExtraTreesClassifier = pd.DataFrame(
{"ExtraTreesClassifier": selected_features, "importance": importances}
).sort_values(by="importance", ascending=False)
```

Finalmente, se genera una lista de las variables incluyendo solo aquellas con una importancia mayor a 0.04, almacenadas en "important_features" para su uso posterior en análisis o visualización.

```
ExtraTreesClassifier = ExtraTreesClassifier[ExtraTreesClassifier["importance"] > 0.04]
important_features = ExtraTreesClassifier["ExtraTreesClassifier"].tolist()
```

### *5.5.4 XGBoost*

XGBoost, abreviatura de "Extreme Gradient Boosting", es un algoritmo avanzado de aprendizaje supervisado utilizado ampliamente en clasificación y regresión. Destacándose por su eficiencia y precisión, utiliza árboles de decisión como base y emplea técnicas de aumento de gradiente para mejorar gradualmente el rendimiento del modelo. Clasificado como un algoritmo de conjunto, XGBoost optimiza la selección de atributos para maximizar el rendimiento del modelo, evaluando la importancia de cada atributo según su contribución en la construcción de árboles de decisión potenciados. Inicialmente desarrollado como parte de la investigación en el grupo DMLC, XGBoost se ha convertido en una herramienta fundamental en aprendizaje automático.

```
Modelo = XGBClassifier()
modelo.fit(Xpandas_T_JOHNSON, REND_UNO)
coef_importancia = modelo.feature_importances_
XGBoost=pd.DataFrame({"XGBoost":Xpandas_T_JOHNSON.columns,"Importancia":
coef_importancia})
```

Una vez entrenado el modelo, se determinan los coeficientes de importancia de las características y se crea un conjunto de datos "XGBoost "para visualizar las características del conjunto de datos "Xpandas_T_JOHNSON" y sus respectivas importancias calculadas. Finalmente, el conjunto de datos "XGBoost" se organiza a manera de lista y se filtran las características que tienen una importancia superior a 0.04 de manera descendente.

```
XGBoost = XGBoost.sort_values(by="Importancia", ascending=False)
XGBoost = XGBoost[XGBoost["Importancia"] > 0.04]
XGBoosts_importantes = XGBoost["XGBoost"].tolist()
```

### *5.5.5 CatBoost*

CatBoost es una biblioteca de aprendizaje automático de código abierto desarrollada por Yandex, diseñada para resolver problemas de clasificación y regresión. Su principal innovación radica en la capacidad de manejar datos categóricos de manera nativa, eliminando la necesidad de preprocesamiento adicional. Esto es posible gracias a su técnica de "Codificación de Impacto", que asigna valores a las categorías según su relevancia para la variable objetivo. Similar a XGBoost, CatBoost construye múltiples árboles de decisión y evalúa la importancia de cada atributo, asegurando que las características más influyentes guíen las decisiones del modelo final. Esta capacidad especializada no solo mejora el rendimiento en diversas tareas

de aprendizaje automático, sino que también reduce el riesgo de sobreajuste en conjuntos de datos complejos y heterogéneos.
Se inicializa un modelo "CatBoost" utilizando "CatBoostClassifier" suprimiendo la salida durante el ajuste del modelo y se obtienen los coeficientes de importancia de las características del modelo entrenado.

```
modelo = CatBoostClassifier(verbose=0)
modelo.fit(Xpandas_T_JOHNSON, REND_UNO)
coef_importancia = modelo.get_feature_importance()
```

A continuación, se crea un conjunto de datos "importancia_df" para visualizar las características del conjunto de datos "Xpandas_T_JOHNSON" y sus respectivas importancias calculadas por CatBoost.

```
importancia_df = pd.DataFrame({"CatBoost": Xpandas_T_JOHNSON.columns, "Importancia":
coef_importancia})
```

Finalmente, el conjunto de datos "CatBoost" se organiza a manera de lista y se filtran las características que tienen una importancia superior a 4 de manera descendente.

```
importancia_df = importancia_df.sort_values(by="Importancia", ascending=False)
importancia_df = importancia_df[importancia_df["Importancia"] > 4]
caracteristicas_importantes = importancia_df["CatBoost"].tolist()
CatBoost = importancia_df
```

### *5.5.6 LightGBM*

LightGBM (Light Gradient Boosting Machine) es una biblioteca de aprendizaje automático de código abierto desarrollada por Microsoft, basada en el algoritmo de aumento de gradiente. Es especialmente eficiente en el entrenamiento rápido y efectivo de modelos, siendo ideal para conjuntos de datos grandes y de alta dimensionalidad. Similar a otros métodos de Boosting como XGBoost y CatBoost, LightGBM construye múltiples árboles de decisión y evalúa la importancia de cada atributo mediante puntuaciones que reflejan su contribución en la toma de decisiones clave. Sin embargo, se distingue por su enfoque en la subdivisión vertical de nodos, lo que acelera el crecimiento de los árboles y reduce considerablemente el tiempo de entrenamiento. Además, al igual que CatBoost, LightGBM maneja datos categóricos de manera nativa, simplificando el proceso de preprocesamiento de datos antes de aplicar algoritmos de clasificación o regresión.
Se inicializa un modelo "LightGBM" utilizando "LGBMClassifier" para clasificación obteniendo los coeficientes de importancia de las características del modelo entrenado.

```
modelo = LGBMClassifier()
modelo.fit(Xpandas_T_JOHNSON, REND_UNO)
coef_importancia = modelo.feature_importances_
```

Se crea un conjunto de datos "LightGBM" para visualizar las características del conjunto de datos "Xpandas_T_JOHNSON" y sus respectivas importancias calculadas por LightGBM.

```
lightGBM=pd.DataFrame({"LightGBM":Xpandas_T_JOHNSON.columns,"Importancia":
coef_importancia})
```

A continuación, se seleccionan y organizan las 12 caracteristicas más importantes y se almacenan en "LightGBM_12" en orden decendente.

```
lightGBM = LightGBM.sort_values(by="Importancia", ascending=False)
lightGBM = LightGBM_12
caracteristicas_importantes_12 = LightGBM["LightGBM"].tolist()
```

Se procede a combinar las características seleccionadas de varios conjuntos de datos, incluyendo LightGBM y se crea un nuevo conjunto de datos "df_resultado" para almacenar la incidencia de cada característica en los diferentes conjuntos de datos.

```
variables_seleccionadas = []
for df_name in ["CatBoost", "LightGBM", "XGBoost", "ExtraTreesClassifier",
"randomforest", "ArbolCart", "RegresionLasso", "EliminacionBidirec",
"EliminacionAdelante", "EliminacionAtras", "RFEarboldecision", "RFEregresionlineal",
"RFE_SVC", "RFEregresionlogistica", "InformacionMutua", "Pearson", "Anova",
"ChiCuadrado",]:
    df = globals()[df_name]
    variables_seleccionadas.extend(df[df_name].tolist())
variables_seleccionadas = sorted(list(set(variables_seleccionadas)))
df_resultado = pd.DataFrame(
    index=variables_seleccionadas,
     columns=["variable", "CatBoost", "LightGBM", "XGBoost", "ExtraTreesClassifier",
"randomforest","ArbolCart","RegresionLasso","EliminacionBidirec",
"EliminacionAdelante","EliminacionAtras","RFEarboldecision","RFEregresionlineal",
"RFE_SVC","RFEregresionlogistica","InformacionMutua","Pearson","Anova",
"ChiCuadrado",])
df_resultado["variable"] = df_resultado.index
```

Se itera sobre cada variable seleccionada y se verifica su presencia en los diferentes DataFrames, registrando su incidencia.

```
for var in variables_seleccionadas:
    for df_name in df_resultado.columns[1:]:
        df = globals()[df_name]
        df_resultado.loc[var, df_name] = 1 if var in df[df_name].tolist() else 0
```

Finalmente, se calcula la suma de incidencias de cada variable y se ordena el conjunto de datos "df_resultado" según esta suma en orden descendente y se exporta en formato Excel (".xlsx") en la ruta especificada.

```
df_resultado["Incidencia"] = df_resultado.iloc[:, 1:].sum(axis=1)
df_resultado = df_resultado.sort_values(by="Incidencia", ascending=False)
df_resultado.to_excel("C:/Users/Intevo/Desktop/©     de     caracteristicas/Variables
seleccionadas/ELECTRICA/Variables electrica10.xlsx", index=False)
```

# 6. Módulo Analítica Predictiva Transformación de Variables

Es común encontrar conjuntos de datos con variables en distintas escalas de medida que puede afectar negativamente a algoritmos de aprendizaje automático como la regresión lineal, la regresión logística, los k vecinos más cercanos y las máquinas de vectores de soporte (SVM). Para solucionar esto, se realiza una transformación de escala en las variables de entrada durante el preprocesamiento, ajustando sus rangos para mejorar la precisión y estabilidad de los modelos. Técnicas populares como el reescalado, la estandarización y la normalización, implementadas en Python con métodos como MinMaxScaler, StandardScaler y Normalizer, ayudan a que los datos se asemejen a una distribución cuasi-gaussiana, mejorando el rendimiento de los modelos. La elección de la técnica de escalamiento depende del conjunto de datos, el algoritmo y las características del problema, por lo que es esencial evaluar cómo cada técnica afecta el rendimiento del modelo para obtener los mejores resultados.

## 6.1 Carga de Datos

Se define una función para la carga de datos que se toman en formato ".csv" y separados por medio de ";".

```
def cargar_datos(carrera, semestre):
    ruta_archivo = f'C:/Users/Intevo/Desktop/UNIVERSIDAD DISTRITAL PROYECTO
FOLDER/UNIVERSIDAD-DISTRITAL-
PROYECTO/MODULO_ANALITICA_DIAGNOSTICA/DATOS/{carrera}{semestre}.csv'
    datos = pd.read_csv(ruta_archivo,sep=";")
    return datos
```

## 6.2 Copia de Datos

Se utiliza el método copy(prec=True) de Pandas para crear copias profundas del objeto X. Una copia profunda crea nuevos objetos independientes en la memoria para cada variable (X1, X_R1, X_E1, X_N1, X_ROB1, X_T_BOX1, X_T_JOHNSON1). Esto es crucial porque garantiza que cualquier modificación hecha en una copia no afectará al objeto original X ni a las otras copias. Es útil cuando se desea manipular y transformar los datos sin alterar los datos originales, manteniendo la integridad y la independencia de los objetos de datos.

```
X1=X.copy(deep=True)
X_R1 = X.copy(deep=True)
X_E1 = X.copy(deep=True)
X_N1 = X.copy(deep=True)
X_ROB1 = X.copy(deep=True)
X_T_BOX1 = X.copy(deep=True)
X_T_JOHNSON1 = X.copy(deep=True)
```

### *6.2.1 Transformación de datos – Reescalado*

El proceso de reescalar los datos es crucial para evitar que las diferencias en las escalas influyan desproporcionadamente en el proceso de modelado, especialmente en algoritmos sensibles a estas diferencias como la regresión lineal, KNN, la regresión logística y las máquinas de vectores de soporte. Este proceso asegura que todas las variables contribuyan de manera equitativa al modelo, mejorando su precisión y estabilidad.

Este proceso requiere determinar los valores mínimos y máximos de cada variable en el conjunto de datos y se realiza mediante técnicas como Min-Max Scaler, que transforma linealmente los datos asegurando que el valor mínimo sea 0 y el máximo sea 1.

```
def reescalar_datos(X):
    transformador = MinMaxScaler(feature_range=(0, 1)).fit(X)
    datos_reescalados = transformador.transform(X)
    set_printoptions(precision=3)
```

```
    print(datos_reescalados[:5, :])
    datos_reescalados_df = pd.DataFrame(data=datos_reescalados, columns=X.columns)
    return datos_reescalados_df
Xpandas_R1 = reescalar_datos(X_R1)
Xpandas_R1.head(2)
```

En este caso el reescalado de datos se utiliza para mejorar la convergencia en los modelos y evitar que las características con escalas mayores predominen sobre las características de menor escala. Para esto se hace uso de este proceso para que todas las características del objeto "X" tengan un escalado dentro del rango de 0 a 1 conservando el nombre de todas las columnas.

### *6.2.2 Transformación de datos – Estandarización*

El propósito de la estandarización es asegurar que todas las variables tengan escalas comparables, lo cual puede mejorar el rendimiento de ciertos algoritmos de aprendizaje automático. Al estandarizar los datos, se elimina cualquier sesgo relacionado con la escala de las variables, facilitando la comparación entre ellas y evitando que una variable predomine sobre las demás debido a su magnitud. Además, la estandarización puede hacer que los datos sean más adecuados para algoritmos de optimización como el descenso del gradiente, que tienden a converger más rápido cuando las variables tienen una escala similar.
La estandarización y el reescalado son técnicas de ajuste de escala utilizadas en el preprocesamiento de datos. La estandarización supone que los datos se ajustan a una distribución normal, mientras que el reescalado no asume ninguna distribución específica. Si los datos no siguen una distribución normal, es recomendable considerar la normalización antes de aplicar un algoritmo de aprendizaje automático.

```
def estandarizar_datos(X):
    transformador = StandardScaler().fit(X)
    datos_estandarizados = transformador.transform(X)
    set_printoptions(precisión=3)
    print(datos_estandarizados[:5, :])
    datos_estandarizados_df = pd.DataFrame(data=datos_estandarizados,
columns=X.columns)
    return datos_estandarizados_df
Xpandas_E1 = estandarizar_datos(X_E1)
Xpandas_E1.head(2)
```

### *6.2.3 Transformación de datos – Normalización*

La función de normalización opera a nivel de filas en lugar de columnas, a diferencia de otros transformadores como el reescalado y la estandarización. En vez de ajustar las escalas de las variables, la normalización recalibra cada observación (fila) para que tenga una longitud de 1, conocida como norma unitaria o vector de longitud 1. Esto se logra dividiendo cada valor de la observación por su magnitud, resultando en un vector con una longitud constante de 1, transformando así todas las características para que sus valores queden entre -1 y 1.
Este método de preprocesamiento es especialmente útil en conjuntos de datos dispersos, caracterizados por numerosos ceros y atributos con escalas variables. Es particularmente valioso para algoritmos que ponderan los valores de entrada, como las redes neuronales, y para aquellos que utilizan medidas de distancia, como KNN (k vecinos más cercanos). La normalización ayuda a asegurar que todas las observaciones contribuyan equitativamente al modelo, independientemente de sus magnitudes originales, mejorando así el rendimiento y la estabilidad de los algoritmos de aprendizaje automático.

```
def normalizar_datos(X):
    transformador = Normalizer().fit(X)
    datos_normalizados = transformador.transform(X)
    set_printoptions(precisión=3)
```

```
    print(datos_normalizados[:5, :])
    datos_normalizados_df = pd.DataFrame(data=datos_normalizados, columns=X.columns)
    return datos_normalizados_df
Xpandas_N1 = normalizar_datos(X_N1)
Xpandas_N1.head(2)
```

### *6.2.4 Transformación de datos – Estandarización Robusta*

La estandarización robusta es una técnica de preprocesamiento de datos que se utiliza para escalar características numéricas de manera que sean menos sensibles a los valores atípicos. En lugar de utilizar la media y la desviación estándar, se basa en la mediana y el rango intercuartílico (IQR), medidas más resistentes a los valores extremos. Esta técnica es especialmente útil cuando se trabaja con conjuntos de datos que contienen valores atípicos, ya que estos pueden sesgar las estimaciones en la estandarización estándar. Además, es adecuada cuando las variables no siguen una distribución gaussiana y presentan una distribución sesgada.

La estandarización robusta se implementa para asegurar que la escala de las características numéricas sea resistente a valores atípicos, mejorando así la capacidad de los modelos de aprendizaje automático para manejar estos datos. Al emplear medidas como la mediana y el rango intercuartílico, esta técnica ofrece una estimación más precisa de la distribución de los datos, siendo menos afectada por valores extremos.

Para este proceso se define una función que toma como entrada el objeto "X" y se crea una instancia denominada "transformador", la que se encargara de dar una estandarización escalando las características en rangos intercuartílicos del 25% al 75%, lo que significa que se utiliza una medida robusta para la escala. Para la instancia de "datos_reescalados" estandariza todas las características dentro de "X" utizando los parámetros que aprendio durante el ajuste. Finalmente, los datos que fueron estandarizados son guardados en un nuevo DataFrame llamado "datos_reescaldados_df" conservando los nombres de las columnas del objeto "X".

```
def estandarizacion_robusta(X):
    transformador = RobustScaler(quantile_range=(25, 75)).fit(X)
    datos_estandarizados = transformador.transform(X)
    set_printoptions(precisión=3)
    print(datos_estandarizados[:5, :])
    datos_estandarizados_df = pd.DataFrame(data=datos_estandarizados,
columns=X.columns)
    return datos_estandarizados_df
Xpandas_ROB1 = estandarizacion_robusta(X_ROB1)
Xpandas_ROB1.head(2)
```

### *6.2.5 Transformación de datos – Box Cox*

La transformación de Box-Cox es una técnica estadística diseñada para ajustar variables a una distribución normal o gaussiana, propuesta por George Box y Sir David Cox. Es esencial en análisis estadístico para mejorar la estabilidad de la varianza y cumplir con supuestos de normalidad en modelos como regresiones lineales y logísticas, así como Naive Bayes. Sin embargo, es crucial notar que no todos los conjuntos de datos siguen una distribución normal, pudiendo presentar sesgos, valores atípicos o distribuciones exponenciales o logarítmicas.

En este caso la transformación box-cox se utiliza para eliminar las columnas con una desviación 0, y aplicar un rescalado y transformación a los datos restantes, después de haber realizado la transformación, se identifican las columnas constantes en el DataFrame original "X1" y son añadidas al DataFrame final llamado "Xpandas_T_BOX".

```
def transformacion_minmax_boxcox(X):
    std_dev = X.std()
    constantes = std_dev[std_dev == 0].index
```

```
    X_sin_constantes = X.drop(constantes, axis=1)
    minmax_scaler = MinMaxScaler(feature_range=(1, 2))
    Reescalar_X_R = minmax_scaler.fit_transform(X_sin_constantes)
    print("Reescalamiento Min-Max:")
    print(Reescalar_X_R[:5])
    boxcox_transformer = PowerTransformer(method='box-cox', standardize=True)
    Reescalar_X_T_BOX = boxcox_transformer.fit_transform(pd.DataFrame(Reescalar_X_R,
columns=X_sin_constantes.columns))
    print("\nTransformación Box-Cox:")
    print(Reescalar_X_T_BOX[:5])
    Xpandas_T_BOX = pd.DataFrame(data=Reescalar_X_T_BOX,
columns=X_sin_constantes.columns)
    return Xpandas_T_BOX
Xpandas_T_BOX1 = transformacion_minmax_boxcox(X_T_BOX1)
Xpandas_T_BOX1.head(2)

std_dev = X1.std()
constantes = std_dev[std_dev == 0].index
print("Columnas constantes:", constantes)
columnas_constantes = std_dev[std_dev == 0].index
columnas_recuperar = [col for col in columnas_constantes if col in
Xpandas_T_BOX1.columns]
columnas_recuperadas = X1[columnas_recuperar]
Xpandas_T_BOX1 = pd.concat([Xpandas_T_BOX1, columnas_recuperadas], axis=1)
```

### *6.2.6 Transformación de datos – Yeo Johnson*

La transformación Yeo-Johnson es una técnica de preprocesamiento de datos que corrige asimetrías para hacer que los datos se asemejen más a una distribución normal o gaussiana. A diferencia de la transformación Box-Cox, esta metodología es capaz de manejar valores negativos y cero en los datos. Esta transformación es especialmente útil cuando se enfrentan datos con sesgos o asimetrías, ya que puede mejorar la precisión y el desempeño de los modelos predictivos al reducir el impacto de valores atípicos y facilitar la captura de patrones subyacentes en los datos.

```
def transformacion_johnson(X):
    transformador_johnson = PowerTransformer(method='yeo-johnson',
standardize=True).fit(X)
    datos_transformados = transformador_johnson.transform(X)
    set_printoptions(precisión=3)
    print(datos_transformados[:5, :])
    datos_transformados_df = pd.DataFrame(data=datos_transformados,
columns=X.columns)
    return datos_transformados_df
Xpandas_T_JOHNSON1 = transformacion_johnson (X_T_JOHNSON1)
Xpandas_T_JOHNSON1.head(2)
```

## 6.3 Visualización de Datos Transformados

El siguiente código se diseñó para seleccionar y visualizar la distribución de los datos de ciertas columnas en un DataFrame de pandas mediante gráficos de densidad, utilizando la biblioteca seaborn para la visualización. Inicialmente, el código obtiene una lista de columnas a partir de un diccionario denominado diccionario_seleccion. Si la clave 'columnas' no existe en el diccionario o la lista asociada está vacía, se utiliza todo el DataFrame Xpandas_N. En caso contrario, solo se seleccionan las columnas especificadas.
Una vez determinadas las columnas a utilizar, el código procede a crear una figura de tamaño 4x6 pulgadas. Luego, itera sobre cada columna del DataFrame seleccionado (X_seleccionado) y genera un gráfico de

densidad para cada una utilizando la función sns.kdeplot. Esto permite visualizar la distribución de los valores de cada columna en un mismo gráfico. El título del gráfico se establece con la función “ax.set_title”, y se etiquetan los ejes con ‘Valor’ para el eje x y ‘Densidad’ para el eje y. Además, se añade una leyenda para identificar las curvas de densidad de cada columna.

```
    columnas_seleccionadas = diccionario_seleccion.get('columnas', [])
    if columnas_seleccionadas:
        Xpandas_N = X_seleccionado[columnas_seleccionadas]
    else:
        Xpandas_N = X_pandas

    fig4, ax = plt.subplots(figsize=(4, 6))
    for columna in Xpandas.columns:
        sns.kdeplot(Xpandas[columna], ax=ax, label=columna)
    ax.set_title('Transformación Aplicada')
    ax.set_xlabel('Valor')
    ax.set_ylabel('Densidad')
    ax.legend()
    plt.show()
    print("normalizado")
```

A continuación, el proceso se encarga de convertir figuras de gráficos en cadenas de texto codificadas en Base64, almacenarlas en un archivo JSON y preparar múltiples DataFrames con distintas transformaciones de datos para finalmente guardarlos en otro archivo JSON.

Primero, se define un diccionario figuras que contiene referencias a varias figuras (fig1 a fig7). Luego, se inicializa una lista vacía ©_base64 para almacenar las representaciones en Base64 de estas figuras.

El código itera sobre cada figura en el diccionario figuras. Para cada figura, se crea un buffer en memoria utilizando io.BytesIO(), y se guarda la figura en este buffer en formato PNG con figura.savefig(buf, format=’png’). Posteriormente, el buffer se reinicia a la posición inicial con buf.seek(0), y la imagen en el buffer se convierte a una cadena Base64 mediante base64.b64encode(buf.getvalue()).decode(‘utf-8’). Esta cadena se añade a la lista ©_base64 y se imprime un mensaje en la consola indicando que el código Base64 de la figura ha sido guardado.

Tras procesar todas las figuras, se abre un archivo JSON llamado ©_Transformacion.json en modo escritura y se guardan las cadenas Base64 de las imágenes en el archivo utilizando json.dump({“data”: ©_base64}, json_file). Un mensaje en la consola confirma que los códigos Base64 se han guardado correctamente.

A continuación, se preparan varios DataFrames, cada uno con una columna adicional llamada “TRANSFORMACION” que indica el tipo de transformación aplicada. Por ejemplo, X1 se etiqueta como “SIN TRANSFORMACION”, Xpandas_R1 como “REESCALADO”, Xpandas_E1 como “ESTANDARIZACION”, y así sucesivamente para otros DataFrames. Estos DataFrames se concatenan verticalmente en uno solo llamado X_values1 usando pd.concat(). Luego, se rellenan los valores faltantes con ceros mediante X_values1.fillna(0, inplace=True). El DataFrame resultante se convierte en una lista de diccionarios con X_values1.to_dict(orient=’records’).

Se crea un diccionario diccionario_dataframes que contiene dos elementos: la lista de diccionarios resultante (dataTransformacion) y una lista de los nombres de las columnas del DataFrame (columnas). Finalmente, se guarda este diccionario en un archivo JSON llamado Dataframe_Transformacion.json con json.dump({“data”: diccionario_dataframes}, json_file, indent=4). Un mensaje en la consola confirma que los DataFrames se han guardado correctamente.

```
    Figuras = {
        "fig1": fig1,
        "fig2": fig2,
        "fig3": fig3,
```

```
        “fig4”: fig4,
        “fig5”: fig5,
        “fig6”: fig6,
        “fig7”: fig7}
    imagenes_base64 = []
    for nombre_figura, figura in figuras.items():
        buf = io.BytesIO()
        figura.savefig(buf, format=’png’)
        buf.seek(0)
        imagen_base64 = base64.b64encode(buf.getvalue()).decode(‘utf-8’)
        ©_base64.append(imagen_base64)
        print(f”El código Base64 de la imagen {nombre_figura} ha sido guardado.”)

    with open(“Imagenes_Transformacion.json”, “w”) as json_file:
        json.dump({“data”: ©_base64}, json_file)
    print(“Los códigos Base64 de las imágenes han sido guardados en
‘©_Transformacion.json’.”)

    X1[“TRANSFORMACION”]=”SIN TRANSFORMACION”
    Xpandas_R1[“TRANSFORMACION”]=”REESCALADO”
    Xpandas_E1[“TRANSFORMACION”]=”ESTANDARIZACION”
    Xpandas_N1[“TRANSFORMACION”]=”NORMALIZADO”
    Xpandas_ROB1[“TRANSFORMACION”]=”ESTANDARIZACION ROBUSTA”
    Xpandas_T_BOX1[“TRANSFORMACION”]=”BOX COX”
    Xpandas_T_JOHNSON1[“TRANSFORMACION”]=”YEO JHONSON”
    X_values1=pd.concat([X1,Xpandas_R1,Xpandas_E1,Xpandas_N1,
                         Xpandas_ROB1,Xpandas_T_BOX1,Xpandas_T_JOHNSON1],axis=0)
    X_values1.fillna(0,inplace=True)
    data_with_columns = X_values1.to_dict(orient=’records’)

    diccionario_dataframes = [
        {   ‘dataTransformacion’: data_with_columns,
            ‘columnas’: X_values1.columns.tolist()
        }
    ]

    # Guardar en archivo JSON
    with open(“Dataframe_Transformacion.json”, “w”) as json_file:
        json.dump({“data”: diccionario_dataframes}, json_file, indent=4)

    print(“Los DataFrames han sido guardados en ‘Dataframe_Transformacion.json’.”)
```

# 7. Módulo Analítica Predictiva Hiperparámetros Regresión y Clasificación

## 7.1 Selección, Transformación y Preparación de Datos

### *7.1.1 Selección de Variables por Carrera y Semestre*

Inicialmente, se define un diccionario denominado variables_por_carrera, que contiene las variables relevantes para diferentes carreras y semestres en una universidad. Este diccionario incluye cinco carreras:

1. Ingeniería Industrial
2. Ingeniería de Sistemas
3. Ingeniería Catastral
4. Ingeniería Eléctrica
5. Ingeniería Electrónica

Para cada carrera, se enumeran las variables específicas para cada semestre, identificadas por números del 1 al 10. Las variables incluyen nombres como 'PG_ICFES', 'CON_MAT_ICFES', 'FISICA_ICFES', entre otros, que representan diferentes tipos de calificaciones y atributos de los estudiantes.

```
variables_por_carrera = {
    'industrial': {
        1: ['PG_ICFES', 'CON_MAT_ICFES', 'FISICA_ICFES', ... ],
        2: ['PG_ICFES', 'CON_MAT_ICFES', 'FISICA_ICFES', ... ],
        # Resto de semestres
    },
    'sistemas': {
        1: ['PG_ICFES', 'CON_MAT_ICFES', 'FISICA_ICFES', ... ],
        2: ['PG_ICFES', 'CON_MAT_ICFES', 'FISICA_ICFES', ... ],
        # Resto de semestres
    },
    # Otras carreras}
```

Se define una función cargar_datos que toma como argumentos la carrera y el semestre. Esta función construye la ruta del archivo CSV correspondiente a los datos del semestre de la carrera específica y luego carga estos datos en un DataFrame de pandas usando pd.read_csv, especificando que el separador de columnas es un punto y coma ";". Finalmente, devuelve el DataFrame con los datos cargados.

```
def cargar_datos(carrera, semestre):
        ruta_archivo = f'C:/Users/DISTRITAL-
PROYECTO/MODULO_ANALITICA_PREDICTIVA/DATOS/{carrera}{semestre}.csv'
        datos = pd.read_csv(ruta_archivo,sep=";")
        return datos
```

Después de cargar los datos, el código filtra las columnas del DataFrame para seleccionar solo aquellas que son relevantes para la carrera y el semestre especificados. Estas columnas se obtienen del diccionario variables_por_carrera. El DataFrame filtrado, que contiene solo las columnas seleccionadas, se imprime y luego se convierte a tipo entero.

```
Datos = cargar_datos(carrera, semestre)
columnas_filtradas = variables_por_carrera[carrera][semestre]
df = datos[columnas_filtradas]
print("DataFrame con columnas filtradas:")
df=df.astype(int)
df
```

### *7.1.2 Separación de Variables Independientes y Dependiente*

El código separa los datos en dos conjuntos: *X* y *Y*. *X* contiene todas las columnas excepto aquellas cuyo nombre contiene "PROMEDIO_" seguido del semestre en letras, mientras que *Y* contiene solo las columnas que incluyen este patrón en su nombre:

```
    X = df.loc[:, ~df.columns.str.contains(f'PROMEDIO_{semestre_en_letras.upper()}')]
    Y = df.loc[:,
df.columns.str.contains(f'PROMEDIO_{semestre_en_letras.upper()}')]
    print("Separación de datos usando Pandas")
    print(X.shape, Y.shape)
    print(X.shape, Y.shape)
```

### *7.1.3 Transformación de Datos Yeo Johnson y Split de Entrenamiento y Prueba*

Se realiza una copia profunda del conjunto de datos *X* en *X_T_JOHNSON1*. Posteriormente, se define una función tranformacion_johnson que aplica una transformación Johnson al conjunto de datos, utilizando la clase PowerTransformer de scikit-learn con el método 'yeo-johnson':

```
    X_T_JOHNSON1 = X.copy(deep=True)
    def transformacion_johnson(X):
        transformador_johnson = PowerTransformer(method='yeo-johnson',
standardize=True).fit(X)
        datos_transformados = transformador_johnson.transform(X)
        set_printoptions(precisión=3)
        print(datos_transformados[:5, :])
        datos_transformados_df = pd.DataFrame(data=datos_transformados,
columns=X.columns)
        return datos_transformados_df
    Xpandas_T_JOHNSON1 = transformacion_johnson(X_T_JOHNSON1)
    Xpandas_T_JOHNSON1.head(2)
```

La función train_test_split se utiliza para dividir matrices o DataFrames en subconjuntos aleatorios de entrenamiento y prueba. Esta función es muy útil para evaluar el rendimiento de los modelos de aprendizaje automático, ya que permite probar el modelo en un conjunto de datos que no ha visto durante el entrenamiento.

Parámetros de train_test_split

1. Xpandas_T_JOHNSON1: Este es el DataFrame que contiene las características (variables independientes) transformadas mediante la transformación Johnson. Es decir, este DataFrame contiene las columnas relevantes de las variables predictoras que serán usadas para entrenar el modelo.
2. Y: Este es el DataFrame que contiene las etiquetas o valores objetivo (variables dependientes) que se quieren predecir. En este contexto, se refiere a los promedios de los estudiantes por semestre.
3. test_size=0.3: Este parámetro indica el porcentaje del conjunto de datos que se utilizará para el conjunto de prueba. En este caso, el 30% de los datos se reservará para la prueba, mientras que el 70% se utilizará para el entrenamiento.
4. random_state=2: Este parámetro se usa para asegurar que la división de los datos sea reproducible. Es decir, si se utiliza el mismo random_state, se obtendrá la misma división cada vez que se ejecute el código. Esto es útil para la consistencia en la evaluación de modelos y para la reproducibilidad de los resultados.

Salida de train_test_split

La función devuelve cuatro conjuntos de datos:

- X_trn: Este es el subconjunto de entrenamiento de las características (70% de Xpandas_T_JOHNSON1).

- X_tst: Este es el subconjunto de prueba de las características (30% de Xpandas_T_JOHNSON1).
- Y_trn: Este es el subconjunto de entrenamiento de las etiquetas (70% de Y).
- Y_tst: Este es el subconjunto de prueba de las etiquetas (30% de Y).

```
 X_trn, X_tst, Y_trn, Y_tst = train_test_split(Xpandas_T_JOHNSON1, Y, test_size=0.3,
random_state=2)
```

## 7.2 Construcción de Modelo y Obtención de Hiperparámetros

La función entrenar_modelo_knn_con_transformacion ha sido desarrollada para optimizar y entrenar un modelo de regresión mediante la técnica de búsqueda en malla (GridSearchCV). Esta función se encarga de encontrar los mejores hiperparámetros del modelo, así como de entrenarlo con estos parámetros óptimos. A continuación, se detallan los pasos y la lógica implementada en esta función:

```
    def entrenar_modelo_seleccionado_con_transformacion(X_trn, Y_trn):
        X_trn_transformado = X_trn
        parameters = {Parametros_del_mododelo}
        modelo = Modelo_Desarrollado()
        semilla = 5
        num_folds = 10
        kfold =StratifiedKFold(n_splits=num_folds, random_state=semilla,shuffle=True)
        metrica = 'neg_mean_squared_error'
        grid = GridSearchCV(estimator=modelo, param_grid=parameters, scoring=metrica,
cv=kfold, n_jobs=-1)
        grid_resultado = grid.fit(X_trn_transformado, Y_trn)
        mejor_modelo = Modelo_Desarrollado(**grid_resultado.best_params_)
        mejor_modelo.fit(X_trn_transformado, Y_trn)
        mejores_hiperparametros =grid_resultado.best_params_
        return mejor_modelo,grid_resultado.best_params_
    modelo,mejores_hiperparametros =
entrenar_modelo_seleccionado_con_transformacion(X_trn, Y_trn)
    mejores_hiperparametros
```

Nota: La construcción de los modelos se explican en el apartado de analítica práctica tanto para los de regresión como clasificación.

Búsqueda de Hiperparámetros:

- grid = GridSearchCV(estimator=modelo, param_grid=parameters, scoring=metrica, cv=kfold, n_jobs=-1): Se crea una instancia de GridSearchCV para buscar los mejores hiperparámetros usando la métrica neg_mean_squared_error.

Entrenamiento y Selección del Mejor Modelo:

- grid_resultado = grid.fit(X_trn_transformado, Y_trn): Se ajusta el modelo usando los datos de entrenamiento.
- mejor_modelo = Modelo_Seleccionado(**grid_resultado.best_params_): Se crea un nuevo modelo con los mejores hiperparámetros encontrados.
- mejor_modelo.fit(X_trn_transformado, Y_trn): Se entrena el mejor modelo con los datos transformados de entrenamiento.

Salida de la Función:

- La función retorna el mejor modelo entrenado (mejor_modelo) y los mejores hiperparámetros (grid_resultado.best_params_).

## 7.3 Cálculo de Métricas de Entrenamiento y Prueba para Regresión

El código inicia con la creación de un DataFrame vacío llamado resultados_df, que tiene columnas para almacenar las métricas y sus valores. Luego, se utiliza el modelo entrenado para predecir las etiquetas

(Y_pred) del conjunto de datos de entrenamiento y prueba (X_trn, X_tst). A partir de estas predicciones, se calculan diversas métricas de error: Error Cuadrático Medio (MSE), Raíz del Error Cuadrático Medio (RMSE), Error Absoluto Medio (MAE), Coeficiente de Determinación ($R^2$), Error Logarítmico Cuadrático Medio (MSLE), y Raíz del Error Logarítmico Cuadrático Medio (RMSLE).

Además de estas métricas, se calcula el Coeficiente de Determinación Ajustado ($R^2$ Ajustado) para tener en cuenta el número de predictores (k) y el tamaño de la muestra (n). Este ajuste es importante para evaluar de manera más precisa la capacidad explicativa del modelo, especialmente cuando se utiliza un número considerable de predictores.

Cada una de estas métricas se guarda en DataFrames separados para una mejor organización. Por ejemplo, df_MSE contiene el valor del Error Cuadrático Medio, mientras que df_RMSE guarda la Raíz del Error Cuadrático Medio, y así sucesivamente para las otras métricas. Estos DataFrames se combinan posteriormente en un único DataFrame llamado resultados_df_entrenamiento y resultados_df_prueba, el cual consolida toda la información de las métricas de evaluación del modelo sobre el conjunto de entrenamiento y prueba.

Finalmente, se añaden dos columnas adicionales a resultados_df_entrenamiento y resultados_df_prueba para especificar el modelo utilizado y el tipo de datos evaluados. Esta estructuración facilita la interpretación y comparación de los resultados, proporcionando una visión integral del rendimiento del modelo. Las métricas calculadas permiten evaluar desde errores promedio hasta la capacidad explicativa del modelo, ajustándose además por el número de predictores utilizados, lo que resulta crucial para entender el comportamiento del modelo en el conjunto de datos de entrenamiento.

```
Resultados_df= pd.DataFrame(columns=['MÉTRICA', 'VALOR'])
Y_pred = modelo.predict(X_trn)
mse = mean_squared_error(Y_trn, Y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(Y_trn, Y_pred)
r2 = r2_score(Y_trn, Y_pred)
msle = mean_squared_log_error(Y_trn, Y_pred)
rmsle = np.sqrt(msle)
n = len(Y_trn)
k = X_trn.shape[1]
r2_ajustado = 1 - ((1 - r2) * (n - 1) / (n - k - 1))

df_MSE = pd.DataFrame({'MÉTRICA': ['Error Cuadrático Medio (MSE)'], 'VALOR':
[round(mse, 2)]})
df_RMSE= pd.DataFrame({'MÉTRICA': ['Raíz del Error Cuadrático Medio (RMSE)'],
'VALOR': [round(rmse, 2)]})
df_MAE = pd.DataFrame({'MÉTRICA': ['Error Absoluto Medio (MAE)'], 'VALOR':
[round(mae, 2)]})
df_R2 = pd.DataFrame({'MÉTRICA': ['Coeficiente de Determinación'], 'VALOR':
[round(r2*100, 2)]})
df_R2A = pd.DataFrame({'MÉTRICA': ['Coeficiente de Determinación Ajustado'], 'VALOR':
[round(r2_ajustado*100, 2)]})
df_MSLE=pd.DataFrame({'MÉTRICA': ['Error Logarítmico Cuadrático Medio (MSLE)'],
'VALOR': [round(msle*100, 2)]})
df_RMSLE=pd.DataFrame({'MÉTRICA': ['Raíz del Error Logarítmico Cuadrático Medio
(RMSLE)'], 'VALOR': [round(rmsle*100, 2)]})
resultados_df_entrenamiento = pd.concat([resultados_df,
df_MSE,df_RMSE,df_MAE,df_R2,df_R2A,df_MSLE,df_RMSLE], ignore_index=True)
resultados_df_entrenamiento["MODELO"]='MODELO'
resultados_df_entrenamiento["TIPO_DE_DATOS"]='Entrenamiento'
resultados_df_entrenamiento
```

```
resultados_df= pd.DataFrame(columns=['MÉTRICA', 'VALOR'])
Y_pred = modelo.predict(X_tst)
mse = mean_squared_error(Y_tst, Y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(Y_tst, Y_pred)
r2 = r2_score(Y_tst, Y_pred)
msle = mean_squared_log_error(Y_tst, Y_pred)
rmsle = np.sqrt(msle)
n = len(Y_tst)
k = X_tst.shape[1]
r2_ajustado = 1 - ((1 - r2) * (n - 1) / (n - k - 1))

df_MSE = pd.DataFrame({'MÉTRICA': ['Error Cuadrático Medio (MSE)'], 'VALOR':
[round(mse, 2)]})
df_RMSE= pd.DataFrame({'MÉTRICA': ['Raíz del Error Cuadrático Medio (RMSE)'],
'VALOR': [round(rmse, 2)]})
df_MAE = pd.DataFrame({'MÉTRICA': ['Error Absoluto Medio (MAE)'], 'VALOR':
[round(mae, 2)]})
df_R2 = pd.DataFrame({'MÉTRICA': ['Coeficiente de Determinación'], 'VALOR':
[round(r2*100, 2)]})
df_R2A = pd.DataFrame({'MÉTRICA': ['Coeficiente de Determinación Ajustado'], 'VALOR':
[round(r2_ajustado*100, 2)]})
df_MSLE=pd.DataFrame({'MÉTRICA': ['Error Logarítmico Cuadrático Medio (MSLE)'],
'VALOR': [round(msle*100, 2)]})
df_RMSLE=pd.DataFrame({'MÉTRICA': ['Raíz del Error Logarítmico Cuadrático Medio
(RMSLE)'], 'VALOR': [round(rmsle*100, 2)]})
resultados_df_prueba = pd.concat([resultados_df,
df_MSE,df_RMSE,df_MAE,df_R2,df_R2A,df_MSLE,df_RMSLE], ignore_index=True)
resultados_df_prueba["MODELO"]='DecisionTree'
resultados_df_prueba["TIPO_DE_DATOS"]='Prueba'
resultados_df_prueba

mejores_hiperparametros= modelo.get_params()
mejores_hiperparametros

cadena_hiperparametros= ', '.join([f"{key}: {value}" for key, value in
mejores_hiperparametros.items()])
df_hiperparametros= pd.DataFrame({
    'MÉTRICA': ['Mejores Hiperparametros'],
    'VALOR': [cadena_hiperparametros],
    'MODELO': ['MODELO'],
    'TIPO_DE_DATOS': ['Hiperparametros del modelo']
})
resultados_df=
pd.concat([resultados_df_prueba,resultados_df_entrenamiento,df_hiperparametros],
ignore_index=True)
resultados_df
```

Métricas calculadas de los modelos de regresión:

Error Cuadrático Medio (MSE):

- Cálculo: El MSE se calcula promediando los cuadrados de las diferencias entre las predicciones del modelo y los valores reales (observados):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

- Donde $n$ es el número de muestras, $Y_i$ son los valores reales y $\hat{Y}_i$ son las predicciones del modelo.
- Uso: El MSE mide la magnitud promedio de los errores de predicción al cuadrado. Un MSE más bajo indica que el modelo tiene un mejor ajuste a los datos.

Raíz del Error Cuadrático Medio (RMSE):

- Cálculo: El RMSE es simplemente la raíz cuadrada del MSE para que esté en la misma escala que los valores originales:

$$RMSE = \sqrt{MSE}$$

- Uso: El RMSE también proporciona una medida de la magnitud de los errores de predicción, pero en la misma escala que los datos originales. Es más interpretable que el MSE directamente.

Error Absoluto Medio (MAE):

- Cálculo: El MAE es el promedio de las diferencias absolutas entre las predicciones y los valores reales:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$$

- Uso: El MAE proporciona una medida del tamaño promedio de los errores en las unidades originales de los datos. Es menos sensible a valores atípicos en comparación con el MSE.

Coeficiente de Determinación ($R^2$):

- Cálculo: El $R^2$ mide la proporción de la varianza en la variable dependiente que es predecible a partir de las variables independientes en el modelo. Se calcula como:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$$

- Donde $\bar{Y}$ es la media de los valores observados $Y_i$.
- Uso: $R^2$ indica qué tan bien el modelo se ajusta a los datos observados. Un $R^2$ más alto (más cercano a 1) sugiere un mejor ajuste del modelo a los datos.

Error Logarítmico Cuadrático Medio (MSLE):

- Cálculo: El MSLE se calcula de manera similar al MSE, pero primero se toma el logaritmo natural de las predicciones y los valores reales:

$$MSLE = \frac{1}{n}\sum_{i=1}^{n}(\log(1 + Y_i) - \log(1 + \hat{Y}_i))^2$$

- Uso: El MSLE es útil cuando los valores objetivo tienen una gran dispersión y queremos penalizar más fuertemente los errores en las predicciones de los valores altos.

Raíz del Error Logarítmico Cuadrático Medio (RMSLE):

- Cálculo: Es la raíz cuadrada del MSLE para ponerlo en la misma escala que los datos originales:
- Uso: El RMSLE es una medida del error en las predicciones de valores altos, ajustada por el logaritmo de los valores reales y predichos. Es útil en problemas donde la escala de los valores objetivo varía ampliamente.

$$RMSLE = \sqrt{MSLE}$$

## 7.4 Cálculo de Métricas de Entrenamiento y Prueba para Clasificación

Se inicializa un DataFrame vacío llamado resultados_df que contendrá métricas de evaluación del modelo implementado en el conjunto de datos de entrenamiento. Posteriormente, se utiliza el modelo entrenado para predecir las etiquetas (Y_pred_entrenamiento, Y_pred_Prueba) del conjunto de datos de

entrenamiento y prueba (X_trn, X_tst). A continuación, se calculan varias métricas de evaluación clave para evaluar el desempeño del modelo.

Primero, se calcula la Precisión, que mide la proporción de predicciones correctas entre todas las etiquetas predichas. La Exhaustividad (Recall) indica la proporción de etiquetas positivas que el modelo ha predicho correctamente. La Puntuación F1 es la media armónica de Precisión y Exhaustividad, proporcionando una métrica equilibrada de precisión y exhaustividad. La Exactitud mide la proporción de predicciones correctas en comparación con todas las predicciones realizadas. La Información Mutua Normalizada (NMI) evalúa la similitud entre las etiquetas verdaderas y las predichas, ajustando para el número de etiquetas y la distribución de clases. Finalmente, el Índice Kappa de Cohen mide la concordancia entre las etiquetas verdaderas y las predicciones, considerando la probabilidad de que las predicciones se hayan dado por casualidad.

Cada una de estas métricas se almacena en DataFrames individuales (df_precision, df_recall, df_f1, df_accuracy, df_nmi y df_kappa), donde cada DataFrame contiene la métrica y su valor redondeado correspondiente. Luego, estos DataFrames se concatenan en resultados_df_knn_entrenamiento y df_knn_prueba para consolidar todas las métricas de evaluación del modelo implementado sobre los datos de entrenamiento.

Finalmente, se añaden dos columnas adicionales a resultados_df_entrenamiento y resultados_df_prueba: "MODELO", que especifica el nombre del modelo evaluado y "TIPO_DE_DATOS", que indica que las métricas corresponden al conjunto de datos de entrenamiento.

Métricas calculadas de los modelos de clasificación:

**Precisión**: La precisión se define como la proporción de verdaderos positivos (TP) sobre todos los ejemplos etiquetados como positivos (ya sea correctamente o incorrectamente):

$$Precisión = \frac{TP}{TP + FP}$$

Donde:

$TP$ son los verdaderos positivos (instancias correctamente clasificadas como positivas).

$FP$ son los falsos positivos (instancias incorrectamente clasificadas como positivas).

En el contexto de clasificación multiclase, se calcula promediando la precisión de cada clase ponderada por su soporte.

**Exhaustividad (Recall)**: La exhaustividad (o recall) mide la proporción de verdaderos positivos (TP) sobre todas las instancias que son realmente positivas:

$$Exhaustividad = \frac{TP}{TP + FN}$$

Donde:

$FN$ son los falsos negativos (instancias incorrectamente clasificadas como negativas).

Al igual que la precisión, en clasificación multiclase se calcula promediando la exhaustividad de cada clase ponderada por su soporte.

**Puntuación F1 (F1 Score)**: La puntuación F1 es la media armónica de la precisión y la exhaustividad. Proporciona un equilibrio entre ambas métricas y es especialmente útil cuando hay clases desequilibradas en los datos:

$$F1\ Score = 2 \cdot \frac{Precisión \cdot Exhaustividad}{Precisión + Exhaustividad}$$

La puntuación F1 alcanza su mejor valor en 1 (precisión perfecta y exhaustividad) y su peor valor en 0.

**Exactitud (Accuracy)**: La exactitud mide la proporción de predicciones correctas en relación con el total de predicciones realizadas:

$$Exactitud = \frac{TP + TN}{TP + TN + FP + FN}$$

Donde:

$TN$ son los verdaderos negativos (instancias correctamente clasificadas como negativas).

$FP$ y $FN$ son los falsos positivos y negativos, respectivamente.

La exactitud es útil cuando las clases están balanceadas en los datos.

Información Mutua Normalizada (Normalized Mutual Information, NMI): La NMI mide la similitud entre dos distribuciones, en este caso, entre las etiquetas verdaderas y las predichas. Está normalizada para proporcionar un valor entre 0 (sin similitud) y 1 (similitud perfecta):

$$NMI(Y, \hat{Y}) = \frac{I(Y; \hat{Y})}{\sqrt{H(Y) \cdot H(\hat{Y})}}$$

Donde:

$I(Y; \hat{Y})$ es la información mutua entre $Y$ (etiquetas verdaderas) y $\hat{Y}$ (etiquetas predichas).

$H(Y) \cdot H(\hat{Y})$ son las entropías de $Y$ y $\hat{Y}$, respectivamente.

Índice Kappa de Cohen (Cohen's Kappa): El índice Kappa de Cohen mide la concordancia entre las etiquetas verdaderas y las predichas, ajustando para el acuerdo que se esperaría por casualidad:

$$Kappa = \frac{P_o - P_e}{1 - P_e}$$

Donde:

$P_o$ es la proporción de acuerdos observados entre las etiquetas verdaderas y las predichas.

$P_e$ es la proporción de acuerdos esperados por casualidad.

El valor de Kappa varía de -1 (acuerdo completo en sentido opuesto) a 1 (acuerdo perfecto). Un valor de 0 indica que el acuerdo observado es igual al que se esperaría por casualidad.

## 7.5 Concatenado de Resultados de Modelos (métricas e hiperpámetros)

El código a continuación tiene como objetivo principal consolidar y almacenar métricas de varios modelos de aprendizaje automático en un archivo JSON estructurado. En esta sección, se utiliza la función pd.concat() de Pandas para concatenar varios dataframes (resultados_df_knn, resultados_df_svc, etc.) a lo largo del eje 0, es decir, por filas. Esto crea un nuevo dataframe llamado Metricas_Modelos que contiene todas las métricas de los modelos seleccionados. Cada uno de estos dataframes probablemente contiene métricas para diferentes configuraciones de modelos de aprendizaje automático.

```
Metricas_Modelos=pd.concat([resultados_df_knn,resultados_df_svc,resultados_df_tree,re
sultados_df_gaussian,resultados_df_LDA,resultados_df_BG,resultados_df_random,resultad
os_df_extra,resultados_df_ADA,resultados_df_GD,resultados_df_XB,resultados_df_CB,resu
ltados_df_LIGHT,resultados_df_voting,resultados_df_stacking_lineal,
resultados_df_stacking_nolineal,resultados_df_superaprendiz,resultados_df_superaprend
iz_dos_capas],axis=0)
```

En esta línea, se filtra el dataframe Metricas_Modelos para incluir solo las filas donde el valor de la columna 'MODELO' está presente en la lista modelos_seleccionados. Esto significa que estamos interesados específicamente en las métricas de los modelos que han sido preseleccionados o que cumplen con ciertos criterios de calidad, relevancia o desempeño previamente definidos.

```
Metricas_Modelos=
Metricas_Modelos[Metricas_Modelos["MODELO"].isin(modelos_seleccionados)]
```

Aquí, se convierte el dataframe Metricas_Modelos en una lista de diccionarios utilizando el método to_dict() de Pandas con orientación 'records'. Cada diccionario en esta lista representa una fila del

dataframe original, donde las claves del diccionario son los nombres de las columnas y los valores son los datos correspondientes de cada métrica y modelo. Esta transformación es útil para preparar los datos en un formato que puede ser fácilmente serializado y almacenado en formato JSON. En esta sección, se crea un diccionario llamado diccionario_dataframes que contiene una lista con un solo elemento. Este elemento es otro diccionario con una única clave 'dataTransformacion', cuyo valor es la lista de diccionarios data_with_columns. Este paso es crucial para estructurar los datos de manera que sean compatibles con el formato JSON que se va a generar y guardar posteriormente.

Finalmente, en este bloque se abre un archivo JSON llamado "Metricas_Modelos_Regresion.json" en modo escritura ("w"), y se utiliza la función json.dump() para escribir el contenido del diccionario {"data": diccionario_dataframes} en el archivo JSON. El parámetro indent=4 se utiliza para asegurar que la salida JSON esté formateada de manera legible con sangrías de 4 espacios. Además, se imprime un mensaje indicando que el proceso de guardado ha sido completado satisfactoriamente.

```
    Data_with_columns = Metricas_Modelos.to_dict(orient='records')
    diccionario_dataframes = [
            {
                'dataTransformacion': data_with_columns,
            }
        ]
    with open("Metricas_Modelos_Regresion.json", "w") as json_file:
        json.dump({"data": diccionario_dataframes}, json_file, indent=4)
        print("Los DataFrames han sido guardados en
'Metricas_Modelos_Regresion.json'.")
```

# 8. Módulo Analítica Práctica Predicción Individual Regresión

## 8.1 Configuración Inicial y Carga de Bibliotecas

El primer paso en el desarrollo del script es la configuración inicial y la importación de las bibliotecas necesarias. Estas bibliotecas son fundamentales para la manipulación de datos, la visualización, el preprocesamiento, el modelado, y la evaluación de modelos. A continuación, se detallan las bibliotecas y configuraciones utilizadas.

### *8.1.1 Advertencias y configuración de visualización*

Para evitar advertencias innecesarias que puedan entorpecer la salida del programa, se desactivan las advertencias. Además, se configuran opciones de visualización para mejorar la legibilidad de las salidas.

```
Import warnings
warnings.filterwarnings('ignore')
```

### *8.1.2 Manejo y visualización de datos*

Se utilizan varias bibliotecas para la manipulación y visualización de datos:

- NumPy: Para operaciones matemáticas y manejo de arrays.
- Pandas: Para manipulación y análisis de datos.
- Matplotlib y Seaborn: Para visualización de datos.
- Scatter Matrix de Pandas: Para la creación de matrices de dispersión.

```
From numpy import set_printoptions
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

### *8.1.3 Manejo de archivos y formatos diversos*

Se importan bibliotecas para manejar archivos en diferentes formatos, incluyendo Excel, imágenes, y JSON:

- io y base64: Para manejo de streams y codificación de datos en base64.
- json: Para manejar datos en formato JSON.
- openpyxl: Para trabajar con archivos de Excel.
- os y os.path: Para manejo de rutas y archivos en el sistema operativo.
- unidecode: Para normalizar texto, removiendo acentos y caracteres especiales.

```
Import io
import base64
import json
from openpyxl import Workbook
from openpyxl.drawing.image import Image as XLImage
import os
import os.path
from unidecode import unidecode
```

### *8.1.4 Preprocesamiento de datos*

Para preprocesar los datos antes de entrenar los modelos, se utilizan diversas técnicas de escalado y transformación de características:

- LabelEncoder: Para codificación de etiquetas categóricas.
- MinMaxScaler, StandardScaler, Normalizer, RobustScaler, PowerTransformer: Para diferentes métodos de escalado y normalización.

```
From sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import PowerTransformer
```

### *8.1.5 Modelos y validación*

Se utilizan varias estrategias de validación y selección de modelos, incluyendo validación cruzada y búsqueda de hiperparámetros con grid search:

- GridSearchCV, Kfold, StratifiedKFold, train_test_split, cross_val_score: Para validación y precisió de datos.

```
From sklearn.model_selection import GridSearchCV
from sklearn.model_selection import Kfold
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
```

### *8.1.6 Algoritmos de regresión*

El script incluye una amplia variedad de algoritmos de regresión para comparar y seleccionar el mejor modelo:

- Modelos de Ensamble: Bagging, Random Forest, Extra Trees, AdaBoost, Gradient Boosting, XGBoost, CatBoost, LDA, LightGBM, Voting, Stacking, SuperLearner.
- Modelos Lineales: Linear Regression.
- Modelos de Vecinos Cercanos: KneighborsRegressor.
- Máquinas de Soporte Vectorial: SVR.
- Modelos Gaussianos: Gaussian Process Regressor con distintos kernels.

```
From sklearn.ensemble import BaggingRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.ensemble import GradientBoostingRegressor
from xgboost import XGBRegressor
from catboost import CatBoostRegressor
from lightgbm import LGBMRegressor
from sklearn.ensemble import VotingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import StackingRegressor
from mlens.ensemble import SuperLearner
```

### *8.1.7 Métricas de evaluación*

Para evaluar el rendimiento de los modelos, se utilizan varias métricas, incluyendo error cuadrático medio, error absoluto medio, coeficiente de determinación, entre otros:

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.metrics import mean_squared_log_error, median_absolute_error
```

## 8.2 Proceso de Carga y Transformación de Datos

En este punto se detalla el procedimiento para cargar y transformar datos académicos para diferentes carreras y semestres en la Universidad Distrital. El proceso incluye la selección de columnas específicas para

cada carrera y semestre, la carga de datos desde archivos CSV y la aplicación de la transformación Yeo-Johnson.

### *8.2.1 Configuración inicial de variables por carrera*

Se tienen definidos los conjuntos de variables relevantes para cada carrera y semestre en un diccionario llamado "variables_por_carrera". Este diccionario mapea cada carrera y semestre a una lista de variables específicas.

```
Variables_por_carrera = {
    'industrial': {
        '1': ['PG_ICFES', 'CON_MAT_ICFES',
'FISICA_ICFES','QUIMICA_ICFES','IDIOMA_ICFES','LOCALIDAD','PROMEDIO_UNO'],
    …
    },
    'sistemas': {
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','LOCALIDAD_COLEGIO','BIOLOGIA_ICFES','QUIMICA_ICFES',
'PG_ICFES','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    },
    'catastral': {
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','BIOLOGIA_ICFES','GENERO','FILOSOFIA_ICFES','LOCALIDA
D','LITERATURA_ICFES','QUIMICA_ICFES','PG_ICFES','PROMEDIO_UNO'],
   …
    },
    'electrica':{
        '1':
['IDIOMA_ICFES','PG_ICFES','FISICA_ICFES','ANO_INGRESO','APT_VERB_ICFES','QUIMICA_ICF
ES','FILOSOFIA_ICFES','GENERO','BIOLOGIA_ICFES','PROMEDIO_UNO'],

    },
    'electronica':{
        '1':
['CON_MAT_ICFES','QUIMICA_ICFES','PG_ICFES','FISICA_ICFES','IDIOMA_ICFES','BIOLOGIA_I
CFES','GENERO','LOCALIDAD','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    }
}
```

### *8.2.2 Función para cargar datos*

Para cargar los datos de una carrera específica y un semestre determinado, se utiliza la función "cargar_datos". Esta función lee un archivo CSV ubicado en una ruta específica y retorna un DataFrame con los datos cargados.

```
def cargar_datos(carrera, semestre):
    ruta_archivo = f'C:/Users/Desktop/UNIVERSIDAD DISTRITAL PROYECTO
FOLDER/UNIVERSIDAD-DISTRITAL-
PROYECTO/MODULO_ANALITICA_PRACTICO/DATOS/{carrera}{semestre}.csv'
    datos = pd.read_csv(ruta_archivo,sep=";")
    return datos
```

#### *8.2.3 Cargar datos y seleccionar columnas*

Cargamos los datos y seleccionamos las columnas específicas definidas para la carrera y semestre correspondientes.

```
Datos = cargar_datos(carrera, semestre)
columnas_filtradas = variables_por_carrera[carrera][semestre]
df = datos[columnas_filtradas]
print("DataFrame con columnas filtradas:")
df=df.astype(int)
df
```

#### *8.2.4 Separación de variables independientes y dependientes*

Se separan las variables independientes (X) de la variable dependiente (Y), correspondiente al promedio del semestre.

```
X = df.loc[:, ~df.columns.str.contains(f'PROMEDIO_{semestre_en_letras.upper()}')]
Y = df.loc[:, df.columns.str.contains(f'PROMEDIO_{semestre_en_letras.upper()}')]
df.columns.str.contains(f'PROMEDIO_{semestre_en_letras.upper()}')]
print("Separación de datos usando Pandas")
print(X.shape, Y.shape)
```

#### *8.2.5 Conversión de tipos de datos*

Las variables independientes se convierten a tipo float32 para la posterior transformación.

```
X = X.astype('float32')
print(X.shape, Y.shape)
```

#### *8.2.6 Transformación Yeo-Johnson*

Se realiza una transformación Johnson en los datos para normalizarlos y hacerlos adecuados para el modelado.

```
X_T_JOHNSON1 = X.copy(deep=True)
def transformacion_johnson(X):
    transformador_johnson = PowerTransformer(method='yeo-johnson',
standardize=True).fit(X)
    datos_transformados = transformador_johnson.transform(X)
    set_printoptions(precisión=3)
    print(datos_transformados[:5, :])
    datos_transformados_df = pd.DataFrame(data=datos_transformados,
columns=X.columns)
    return datos_transformados_df
Xpandas_T_JOHNSON1 = transformacion_johnson(X_T_JOHNSON1)
Xpandas_T_JOHNSON1.head(2)
```

### 8.3 División de Datos en Conjuntos de Entrenamiento y Prueba

La división de los datos en conjuntos de entrenamiento y prueba es un paso crucial en el proceso de modelado de datos. Esta técnica permite evaluar el rendimiento de un modelo con datos que no se utilizaron durante su entrenamiento, proporcionando una estimación más precisa de su capacidad de generalización a datos nuevos. Utilizamos la función train_test_split del módulo model_selection de scikit-learn para realizar esta división.

La función train_test_split toma como entrada las características (X) y las etiquetas (Y), y divide los datos en cuatro subconjuntos:

- X_trn: Características para el conjunto de entrenamiento.
- X_tst: Características para el conjunto de prueba.
- Y_trn: Etiquetas para el conjunto de entrenamiento.

- Y_tst: Etiquetas para el conjunto de prueba.

Los parámetros usados son:

- X_transformado: Las características del conjunto de datos.
- Y: Las etiquetas del conjunto de datos.
- test_size=0.3: Especifica que el 30% de los datos se asignarán al conjunto de prueba y el 70% restante al conjunto de entrenamiento.
- random_state=2: Fija la semilla para la generación de números aleatorios, garantizando que la división de datos sea reproducible.

```
X_transformado = transformacion_johnson(X)
X_trn, X_tst, Y_trn, Y_tst = train_test_split(X_transformado, Y, test_size=0.3,
random_state=2)
```

## 8.4 Desarrollo de Modelos de Machine Learning – Regresión

### *8.4.1 Modelo K-Nearest Neighbors (KNN) con búsqueda de hiperparámetros*

El modelo K-Nearest Neighbors (KNN) es un algoritmo de aprendizaje supervisado utilizado tanto para clasificación como para regresión. En el caso de la regresión, el valor de la predicción es la media (o ponderada) de los valores de los k vecinos más cercanos al punto de interés. La idea es que puntos similares tendrán valores de salida similares.

Parámetros para la búsqueda:

```
parameters = {'n_neighbors': [i for i in range(1, 18, 1)],
        'metric': ['euclidean', 'manhattan', 'minkowski'],
        'algorithm': ['auto'],'p': [i for i in range(1, 6)],
        'weights': ['uniform']}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- n_neighbors: Número de vecinos a considerar (de 1 a 17).
- metric: Métrica de distancia a utilizar (Euclidean, Manhattan, Minkowski).
- algorithm: Algoritmo a utilizar para la computación de vecinos (auto selecciona automáticamente el algoritmo más adecuado).
- p: Parámetro para la métrica Minkowski (relevante solo si metric es Minkowski).
- weights: Tipo de ponderación (uniforme asigna el mismo peso a todos los vecinos).

Inicialización del Modelo:

Se instancia un modelo KNN para regresión sin especificar aún los hiperparámetros.

```
Modelo = KneighborsRegressor()
```

### *8.4.2 Modelo SVR con búsqueda de hiperparámetros*

El SVR es un algoritmo de aprendizaje supervisado utilizado para predecir valores numéricos (continuos) en lugar de clasificar etiquetas como en las SVM tradicionales. Funciona encontrando un hiperplano en un espacio dimensional superior que minimiza una función de pérdida, con el objetivo de maximizar el margen entre los puntos de datos y este hiperplano.

Parámetros para la búsqueda:

```
parameters = {'kernel':  ['rbf', 'poly', 'sigmoid','linear'],
            'C': [i/10000 for i in range(8,12,1)],
            'max_iter': [i for i in range(1,3,1)],
            'gamma' : [i/100 for i in range(90,110,5)]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- Kernel Function: El parámetro kernel en SVR define el tipo de función kernel a utilizar, que determina cómo se transforman los datos a un espacio de características de mayor dimensión donde se puede encontrar un hiperplano separador. Los kernels comunes incluyen:
    - RBF (Radial Basis Function): Función gaussiana que mide la distancia entre un punto y los demás puntos.
    - Polynomial: Transformación polinómica que puede manejar relaciones no lineales entre los datos.
    - Sigmoid: Similar a una función de activación en redes neuronales, puede manejar funciones no lineales.
    - Linear: Utiliza una función lineal para la transformación, útil para problemas lineales.
- Regularización ©: Controla el balance entre el ajuste exacto de los datos de entrenamiento y la generalización del modelo para nuevos datos. Un valor más alto de C permite un ajuste más preciso de los datos de entrenamiento, pero podría llevar a un sobreajuste.
- Gamma (γ): Este parámetro afecta la influencia de cada ejemplo de entrenamiento. Un valor más alto de gamma significa que solo los puntos de datos cercanos tendrán un efecto significativo en la predicción, lo que conduce a un modelo más ajustado a los datos de entrenamiento (puede llevar a sobreajuste si se configura muy alto).
- Máximo de Iteraciones (max_iter): Establece el número máximo de iteraciones permitidas para la optimización del modelo. Aumentar este valor puede ser necesario si el modelo no converge con el número predeterminado de iteraciones.

Inicialización del Modelo:
Se crea una instancia del modelo SVR sin ningún hiperparámetro especificado.

```
Modelo = SVR()
```

### *8.4.3 Modelo Decision Tree con búsqueda de hiperparámetros*

Un árbol de decisión es una estructura jerárquica que modela decisiones en forma de árbol. Cada nodo interno del árbol representa una característica o atributo, y cada borde representa una regla de decisión basada en ese atributo. Los nodos hoja representan la salida o la predicción numérica en el caso de la regresión.
El árbol de decisión es un algoritmo de aprendizaje supervisado utilizado tanto para problemas de clasificación como de regresión. En este caso, se está utilizando para predecir valores numéricos continuos (regresión). Este modelo divide recursivamente el conjunto de datos en subconjuntos más pequeños basándose en las características que mejor separan las clases o predicen valores en el caso de la regresión. Los criterios como max_depth, min_samples_leaf y max_features controlan la estructura y la complejidad del árbol para evitar sobreajuste y mejorar la generalización. Para predecir, se desciende por el árbol desde el nodo raíz hasta un nodo hoja, siguiendo las reglas de decisión establecidas por los parámetros y los datos de entrenamiento.
Parámetros para la búsqueda:

```
    parameters = {
            'max_depth':[i for i in range(1,7,1)],
            'min_samples_leaf' : [i for i in range(1,7,1)],
            'max_features' : [i for i in range(1,7,1)],
            'splitter': ["best", "random"],
            'random_state': [i for i in range(1,7,1)]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- max_depth: Controla la profundidad máxima del árbol. Limita la cantidad de divisiones o nodos que puede tener el árbol.

- min_samples_leaf: Especifica el número mínimo de muestras requeridas para ser un nodo hoja (final) en el árbol.
- max_features: Especifica el número máximo de características a considerar al buscar la mejor división en cada nodo.
- splitter: Determina la estrategia utilizada para elegir la división en cada nodo. “best” elige la mejor división y “random” elige la mejor división aleatoria.
- random_state: Controla la aleatoriedad en la generación del árbol para asegurar resultados reproducibles.

Inicialización del Modelo:
Se crea una instancia del modelo DecisionTreeRegressor sin ningún hiperparámetro especificado, lo que significa que se utilizarán los valores por defecto.

```
Modelo = DecisionTreeRegressor()
```

### *8.4.4 Modelo Gaussian Process Regressor con búsqueda de hiperparámetros*

El Regresor de Proceso Gaussiano modela la relación entre las variables de entrada X y las salidas Y como una distribución sobre funciones posibles, en lugar de una función determinística como en los modelos lineales. Esto permite manejar incertidumbres y proporcionar predicciones probabilísticas.

- Funciones de Covarianza: El GPR utiliza una función de covarianza para medir la similitud entre puntos de datos. Esto permite cuantificar la incertidumbre asociada con las predicciones.
- Predicción Probabilística: En lugar de proporcionar un solo valor de predicción, el GPR produce una distribución de probabilidad sobre los posibles valores de salida para cada entrada X.

El Regresor de Proceso Gaussiano (GPR) es un método no paramétrico que puede ser utilizado para resolver problemas de regresión. A diferencia de los modelos lineales, el GPR no asume una estructura particular para la función subyacente, sino que modela la distribución sobre todas las posibles funciones que son consistentes con los datos observados.
Parámetros para la búsqueda:

```
parameters = {‘alpha’: [1e-10,1e-5,1e-2,1e-1],
        ‘n_restarts_optimizer’: [0,1,2,3],’normalize_y’: [True, False],
        ‘optimizer’: [‘fmin_l_bfgs_b’]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- preci: Parámetro de regularización que controla la magnitud del ruido en las observaciones. Un valor más alto de preci permite que el modelo se ajuste más flexiblemente a los datos, pero puede aumentar el sobreajuste.
- n_restarts_optimizer: Número de reinicios del optimizador para mejorar la convergencia del modelo. Un valor más alto puede mejorar la precisión del modelo, pero también aumenta el tiempo de entrenamiento.
- normalize_y: Indica si se debe normalizar la variable de respuesta *Y* antes de ajustar el modelo. Esto puede ser útil para mejorar la convergencia en algunos casos.
- precisión: Método de optimización utilizado para ajustar los hiperparámetros del modelo. En este caso, ‘fmin_l_bfgs_b’ es un método específico de optimización utilizado en la implementación de scikit-learn para GPR.

Inicialización del Modelo:

```
modelo = GaussianProcessRegressor()
```

### *8.4.5 Modelo LDA Regressor con búsqueda de hiperparámetros*

LDA es una técnica clásica que busca encontrar las direcciones (llamadas discriminantes lineales) que maximizan la separación entre múltiples clases en el espacio de características. LDA a menudo se compara

con PCA, ya que ambos métodos buscan reducir la dimensionalidad. Sin embargo, LDA tiene en cuenta la información de las etiquetas de clase para maximizar la separación entre clases, mientras que PCA no considera las etiquetas. En el contexto de clasificación, LDA utiliza las direcciones discriminantes para proyectar los datos en un espacio de menor dimensión donde las clases sean más fácilmente separables.
El Análisis Discriminante Lineal (LDA) es una técnica utilizada para la clasificación supervisada, que también se puede adaptar para la reducción de dimensionalidad.
Parámetros para la búsqueda:

```
parameters = { 'solver':  ['svd','lsqr','eigen'],
            'n_components':[1,2,3,4,5,6,7,8,9,10],
            'shrinkage': ['auto', 0.001, 0.01, 0.1, 0.5,1,10,100,1000]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- solver: Especifica el método utilizado para resolver el problema de optimización en LDA.
  - 'svd': Descomposición en valores singulares, adecuado para datos de alta dimensión.
  - 'lsqr': Mínimos cuadrados con regularización, útil cuando hay multicolinealidad.
  - 'eigen': Descomposición de autovalores, adecuado para problemas donde n_samples > n_features.
- n_components: Número de componentes a extraer en caso de usar LDA para reducción de dimensionalidad.
- shrinkage: Parámetro de regularización para mejorar la estabilidad y rendimiento de LDA cuando solver='lsqr'.

Inicialización del Modelo:
Se crea una instancia del modelo LinearDiscriminantAnalysis sin ningún hiperparámetro especificado, lo que significa que se utilizarán los valores por defecto.

```
modelo = LinearDiscriminantAnalysis()
```

### *8.4.6 Modelo Bagging Regressor con búsqueda de hiperparámetros*

Bagging Regressor mejora la precisión de los modelos reduciendo la varianza, promediando múltiples modelos entrenados en diferentes subconjuntos de datos. Esto ayuda a mitigar el sobreajuste y mejorar la generalización del modelo. Genera múltiples subconjuntos de datos mediante muestreo con reemplazo (bootstraping), permitiendo que cada estimador base vea diferentes perspectivas del conjunto de datos original. Las predicciones finales se obtienen promediando las predicciones de todos los estimadores base, reduciendo así la sensibilidad a variaciones pequeñas en los datos de entrenamiento.
Estimador Base: Puede ser cualquier tipo de modelo de regresión o clasificación, en este caso DecisionTreeRegressor, ajustado con los mejores hiperparámetros encontrados previamente.
Bagging (Bootstrap Aggregating) es una técnica de ensamblaje que mejora la estabilidad y la precisión de los modelos al promediar múltiples modelos base entrenados en diferentes subconjuntos de datos. En este caso, se aplica para problemas de regresión utilizando DecisionTreeRegressor como estimador base.
Parámetros para la búsqueda:

```
    parameters = {'n_estimators': [i for i in range(750,760,5)],
            'max_samples' : [i/100.0 for i in range(70,90,5)],
            'max_features': [i/100.0 for i in range(75,85,5)],
            'bootstrap': [True], 'bootstrap_features': [True]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- n_estimators: Número de estimadores base en el ensamblaje.
- max_samples: Tamaño máximo de cada subconjunto de datos generado mediante bootstraping, como fracción del tamaño total de los datos.
- max_features: Número máximo de características a considerar en cada estimador base.
- precisión: Indica si se utiliza precisión para muestrear los datos de entrenamiento.

- precisión_features: Indica si se utiliza precisión para muestrear las características.

Inicialización del Estimador Base:
Se inicializa un estimador base DecisionTreeRegressor utilizando los mejores hiperparámetros obtenidos previamente (pasados como mejores_hiperparametros_tree).

```
base_estimator= DecisionTreeRegressor(**mejores_hiperparametros_tree)
```

Inicialización del Modelo:

```
modelo = BaggingRegressor(estimator=base_estimator)
```

### *8.4.7 Modelo Random Forest Regressor con búsqueda de hiperparámetros*

Random Forest Regressor es una técnica de ensamblaje que construye múltiples árboles de decisión durante el entrenamiento y promedia sus predicciones para mejorar la precisión y controlar el sobreajuste. Random Forest Regressor mejora la precisión y la generalización al promediar múltiples árboles de decisión entrenados en diferentes subconjuntos de datos y características. Construye múltiples árboles de decisión independientes durante el entrenamiento, cada uno entrenado en un subconjunto aleatorio de datos (bootstraping) y características (subconjunto aleatorio). Para la regresión, las predicciones finales se obtienen promediando las predicciones de todos los árboles individuales.
Parámetros para la búsqueda:

```
parameters = { 'min_samples_split' : [1,2,3,4,6,8,10,15,20],
               'min_samples_leaf': [1,3,5,7,9,12,15]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- min_samples_split: Número mínimo de muestras requeridas para dividir un nodo interno.
- min_samples_leaf: Número mínimo de muestras requeridas en un nodo hoja.

Estos parámetros controlan la complejidad de los árboles individuales en el ensamblaje.
Inicialización del Modelo:
Se crea una instancia del modelo RandomForestRegressor sin ningún hiperparámetro especificado, lo que significa que se utilizarán los valores por defecto.

```
modelo = RandomForestRegressor()
```

### *8.4.8 Modelo Extra Trees Regressor con búsqueda de hiperparámetros*

Extra Trees Regressor (Extremely Randomized Trees Regressor) es una técnica de ensamblaje similar a Random Forest pero con diferencias clave en la forma en que se construyen los árboles de decisión y se maneja la aleatorización.

- Aleatorización Adicional: En Extra Trees, las divisiones en cada nodo se realizan de manera aún más aleatoria que en Random Forest, lo que puede llevar a una mayor varianza, pero también a una reducción en el sobreajuste.
- Menor Costo Computacional: Debido a la mayor aleatorización, los árboles individuales en Extra Trees pueden construirse más rápidamente que en Random Forest.
- Promedio de Predicciones: Al igual que Random Forest, las predicciones finales se obtienen promediando las predicciones de todos los árboles individuales.

Parámetros para la búsqueda:

```
parameters = {'min_samples_split' : [i for i in range(1,10,1)],
              criterion: 'absolute_error', 'squared_error',
              'friedman_mse','poisson')}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- min_samples_split: Número mínimo de muestras requeridas para dividir un nodo interno.
- criterion: Criterio utilizado para medir la calidad de una división en el árbol.

- absolute_error: Error absoluto medio.
- squared_error: Error cuadrático medio.
- precisió_mse: Error cuadrático medio con mejoras de Friedman.
- poisson: Error de Poisson.

Inicialización del Modelo:

```
modelo = ExtraTreesRegressor(random_state=semilla, n_estimators=40,bootstrap=True)
```

### *8.4.9 Modelo AdaBoost Regressor con búsqueda de hiperparámetros*

AdaBoost Regressor se diferencia de otros métodos de ensamblaje en cómo se ponderan los modelos débiles y cómo se actualizan los pesos de las instancias en cada iteración. AdaBoost construye un modelo fuerte secuencialmente agregando modelos débiles, ajustando los pesos de las instancias en cada iteración para enfocarse más en las instancias que fueron mal clasificadas por modelos anteriores. Cada instancia en el conjunto de datos tiene un peso asociado que se ajusta durante el entrenamiento para dar más importancia a las instancias difíciles de clasificar. AdaBoost (Adaptive Boosting) es una técnica de ensamblaje que construye un modelo fuerte combinando múltiples modelos débiles en secuencia.

Parámetros para la búsqueda:

```
    parameters = {'learning_rate' : [i/10000.0 for i in range(5,20,5)],
                  'n_estimators': [i for i in range(1,50,1)]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- learning_rate: Tasa de aprendizaje que controla la contribución de cada modelo débil al modelo final. Valores más altos hacen que los modelos débiles tengan más peso.
- n_estimators: Número de modelos débiles utilizados en el ensamblaje (número de iteraciones).

Inicialización del Modelo:

```
modelo = AdaBoostRegressor(estimator = None,random_state= None)
```

### *8.4.10 Modelo Gradient Boosting Regressor con búsqueda de hiperparámetros*

Gradient Boosting Regressor es una técnica de ensamblaje que construye un modelo fuerte secuencialmente mediante la adición de árboles de decisión débiles, optimizando una función de pérdida diferenciable. Gradient Boosting Regressor se destaca por su capacidad para optimizar una función de pérdida diferenciable, ajustando secuencialmente los modelos para mejorar la precisión en las predicciones:

- Optimización Gradiente: A diferencia de métodos como Random Forest, Gradient Boosting optimiza una función de pérdida diferenciable utilizando gradientes descendentes, ajustando los modelos en secuencia para minimizar el error residual.
- Construcción Secuencial: Cada árbol en el ensamblaje se ajusta para corregir los errores cometidos por los árboles anteriores, enfocándose más en las áreas donde el modelo actual falla.
- Regularización: Utiliza regularización interna y el parámetro de tasa de aprendizaje para controlar el sobreajuste y mejorar la generalización.

Parámetros para la búsqueda:

```
 parameters = { 'learning_rate' : [0.01, 0.05, 0.1,0.15],
                'loss': 'absolute_error', 'squared_error', 'quantile', 'huber'),
                'criterion':['friedman_mse']}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- learning_rate: Tasa de aprendizaje que controla la contribución de cada árbol al modelo. Valores más bajos requieren más árboles en el ensamblaje para alcanzar la misma precisión.
- pre: Función de pérdida utilizada para optimizar el modelo. Las opciones incluyen:
    - absolute_error: Error precisió.

- squared_error: Error cuadrático.
- quantile: Pérdida de cuantil para estimar intervalos de predicción.
- huber: Pérdida de Huber, combina las propiedades de la pérdida cuadrática y lineal.
- criterion: Criterio utilizado para medir la calidad de una división en cada árbol base. En este caso, 'friedman_mse' está especificado, que es específico de Gradient Boosting y utiliza mejoras de Friedman para evaluar la división.

Inicialización del Modelo:

```
modelo = GradientBoostingRegressor()
```

### *8.4.11 Modelo XGBoost Regressor con búsqueda de hiperparámetros*

El modelo XGBoost es un algoritmo de aprendizaje supervisado que utiliza Gradient Boosting para mejorar la precisión de los modelos de manera eficiente. XGBoost es una implementación optimizada de Gradient Boosting que ha demostrado ser extremadamente efectiva en competiciones de ciencia de datos y aplicaciones industriales debido a su capacidad para manejar grandes volúmenes de datos y generar modelos con alta precisión. Se basa en la técnica de ensamblaje llamada Gradient Boosting, que construye un modelo predictivo fuerte combinando múltiples modelos débiles en secuencia. A diferencia de otros métodos de ensamblaje, como Random Forest que construye árboles de decisión de manera independiente, Gradient Boosting optimiza iterativamente un modelo añadiendo nuevos árboles que corrigen los errores cometidos por los modelos anteriores.

Parámetros para la búsqueda:

```
parameters = {'reg_alpha': [0,0.1,0.2,0.3,0.4,0.5],
              'reg_lambda':  [i/1000.0 for i in range(100,150,5)],
              'colsample_bytree': [0.1,0.3, 0.5,0.6,0.7,0.8, 0.9, 1,1.1],
              'max_features': ('sqrt','log2')}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- reg_alpha: Parámetro de regularización L1 (Lasso). Ayuda a prevenir el sobreajuste penalizando los coeficientes grandes.
- reg_lambda: Parámetro de regularización L2 (Ridge). Similar a reg_alpha, pero penaliza los coeficientes cuadráticos grandes.
- colsample_bytree: Fracción de columnas a considerar al construir cada árbol. Ayuda a controlar el sobreajuste al muestrear características.
- max_features: Número máximo de características a considerar al buscar la mejor división. Puede ser 'sqrt' (raíz cuadrada de características) o 'log2' (logaritmo base 2 de características). reg_alpha y reg_lambda: Parámetros de regularización que controlan la penalización de los coeficientes para evitar el sobreajuste.

Inicialización del Modelo:

```
modelo = XGBRegressor(random_state=semilla,subsample =1,max_depth =2)
```

### *8.4.12 Modelo CatBoostRegressor con búsqueda de hiperparámetros*

El modelo CatBoostRegressor es una implementación avanzada de Gradient Boosting desarrollada por Yandex, optimizada para manejar eficientemente variables categóricas y mejorar la precisión de los modelos de regresión. Este modelo es conocido por su manejo automático de variables categóricas, regularización eficaz para prevenir el sobreajuste, y su eficiencia computacional que permite entrenar modelos rápidamente y con menos uso de memoria. Además, CatBoostRegressor puede procesar datos con valores perdidos sin necesidad de preprocesamiento adicional y admite el procesamiento en paralelo. Estas características lo hacen adecuado para una amplia gama de aplicaciones, incluyendo finanzas,

marketing, salud y más, ofreciendo alta precisión y facilidad de uso en la construcción de modelos predictivos robustos.

Inicialización del Modelo CatBoostRegressor:

```
modelo = CatBoostRegressor(random_state=semilla, verbose =0)
```

### *8.4.13 Modelo LightGBM con búsqueda de hiperparámetros*

LightGBM es una implementación de Gradient Boosting diseñada para ser eficiente en memoria y rápida en términos de velocidad de entrenamiento y predicción. Es particularmente adecuado para conjuntos de datos grandes con alta dimensionalidad y valores faltantes. LightGBM es altamente eficiente en términos de memoria y velocidad de entrenamiento. Utiliza histogramas para agrupar los valores de las características, lo que reduce significativamente el tiempo de entrenamiento y la complejidad computacional.

Parámetros para la búsqueda:

```
parameters = {'min_child_samples': [i for i in range(1, 1000,
100)],'colsample_bytree': [0.6, 0.8, 1.0,1.5],'boosting_type': ['gbdt', 'dart',
'goss'],'objective': ['binary', 'multiclass'],'random_state': [42]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- min_child_samples: Número mínimo de muestras (datos) que debe tener un nodo hijo.
- boosting_type: Tipo de algoritmo de boosting que se utilizará.
- colsample_bytree: Fracción de características (features) a considerar al dividir un nodo.
- boosting_type: Tipo de algoritmo de boosting que se utilizará.
- objective: Función de pérdida que el modelo tratará de minimizar.
- random_state: Semilla utilizada por el generador de números aleatorios.

Inicialización del Modelo:

```
modelo = LGBMRegressor()
```

### *8.4.14 Modelo Voting con búsqueda de hiperparámetros*

VotingRegressor es un modelo de ensamble que combina las predicciones de varios modelos base para mejorar la precisión y la robustez del modelo final. A continuación, se describen los conceptos clave y las ventajas de usar un VotingRegressor.

Características Principales

- Ensamble de Modelos: VotingRegressor combina varios modelos base entrenados independientemente y hace la predicción final tomando la media (para regresión) de las predicciones individuales de los modelos.
- Diversidad de Modelos: En este caso, el VotingRegressor incluye modelos de Gradient Boosting, AdaBoost, Extra Trees, Random Forest, Bagging, Decision Tree y XGBoost, cada uno con sus propios mejores hiperparámetros. Esta diversidad ayuda a capturar diferentes aspectos de los datos y a reducir la varianza del modelo final.
- Robustez y Generalización: Al combinar varios modelos, el VotingRegressor tiende a ser más robusto y generaliza mejor a nuevos datos en comparación con un modelo individual. Esto es porque los errores individuales de los modelos tienden a cancelarse entre sí.
- Reducción del Sobreajuste: El ensamble de múltiples modelos ayuda a reducir el riesgo de sobreajuste, ya que la combinación de múltiples predicciones puede suavizar las variaciones excesivas de un solo modelo.

```
def entrenar_modelo_voting_con_transformacion(X_trn, Y_trn, X_tst, Y_tst,
    mejores_hiperparametros_GD,mejores_hiperparametros_tree,
    mejores_hiperparametros_ADA,mejores_hiperparametros_extra,
    mejores_hiperparametros_random,mejores_hiperparametros_BG,
    mejores_hiperparametros_XB):
```

```
    semilla= 5
    modelo1 = GradientBoostingRegressor(**mejores_hiperparametros_GD)
    base_estimator=DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo2 = AdaBoostRegressor(**mejores_hiperparametros_ADA)
    modelo3 = ExtraTreesRegressor(**mejores_hiperparametros_extra)
    modelo4 = RandomForestRegressor(**mejores_hiperparametros_random)
    model = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo5 = BaggingRegressor(**mejores_hiperparametros_BG)
    modelo6 = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo7 = XGBRegressor(**mejores_hiperparametros_XB)
    metrica ='neg_mean_squared_error'
    preci_modelo = VotingRegressor(
    estimators=[('Gradient', modelo1), ('Adaboost', modelo2),
                ('Extratrees', modelo3),('RandomForest',modelo4),
                ('Bagging',modelo5),('Decision tree',modelo6),
                ('XGB',modelo7)])
```

### *8.4.15 Modelo StackingLineal Regressor con búsqueda de hiperparámetros*

El modelo StackingRegressor es una técnica de ensamble en aprendizaje automático donde múltiples modelos base (también conocidos como estimadores) son entrenados en el conjunto de datos y luego combinados usando un modelo final (también conocido como meta-modelo) para realizar la predicción final. A continuación, se describen los conceptos clave y las ventajas de usar un StackingRegressor:

Características Principales

- Ensamble de Modelos Base: StackingRegressor combina las predicciones de varios modelos base entrenados independientemente. En este caso, los modelos base son AdaBoost, Extra Trees, Random Forest y Decision Tree.
- Modelo Final (Meta-modelo): Las predicciones de los modelos base se utilizan como entradas para el modelo final. En este caso, el modelo final es una regresión lineal (LinearRegression). Este meta-modelo aprende cómo combinar las predicciones de los modelos base para mejorar la precisión global.
- Validación Cruzada: Para evitar el sobreajuste y asegurar una evaluación justa, el StackingRegressor puede utilizar validación cruzada para generar las predicciones de los modelos base que luego son usadas para entrenar el meta-modelo. En este caso, se utiliza StratifiedKFold con 10 particiones (n_splits=10).

```
def entrenar_modelo_stacking_lineal_con_transformacion(X_trn, Y_trn, X_tst,
Y_tst,mejores_hiperparametros_tree,mejores_hiperparametros_ADA,
mejores_hiperparametros_extra,mejores_hiperparametros_random):
    semilla= 5
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    base_estimator=DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo2 = AdaBoostRegressor(**mejores_hiperparametros_ADA)
    modelo3 = ExtraTreesRegressor(**mejores_hiperparametros_extra)
    modelo4 = RandomForestRegressor (**mejores_hiperparametros_random)
    model = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo5 = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    estimador_final = LinearRegression()
    metrica ='neg_mean_squared_error'
    mejor_modelo = StackingRegressor(
    estimators=[ ('Adaboost', modelo2), ('Extratrees', modelo3),
                ('Random Forest',modelo4),
                ('Decision tree',modelo5)], final_estimator=estimador_final)
```

### *8.4.16 Modelo Stacking no Lineal Regressor con búsqueda de hiperparámetros*

El modelo StackingRegressor es una técnica de ensamble en aprendizaje automático que combina múltiples modelos base para mejorar la precisión de la predicción. En esta implementación, se utiliza un estimador final no lineal (ExtraTreesRegressor) en lugar de un modelo lineal, lo que puede capturar relaciones más complejas entre las características y el objetivo. A continuación, se describen los conceptos clave y las ventajas de usar un StackingRegressor no lineal:

Características Principales

- Ensamble de Modelos Base: Combina las predicciones de varios modelos base. En este caso, los modelos base son AdaBoost, Extra Trees, Random Forest y Decision Tree.
- Modelo Final (Meta-modelo) No Lineal: Utiliza un ExtraTreesRegressor como meta-modelo final. Este tipo de modelo puede capturar relaciones complejas y no lineales entre las predicciones de los modelos base y el valor objetivo.
- Validación Cruzada: Utiliza StratifiedKFold para realizar validación cruzada con 10 particiones, lo que ayuda a evaluar de manera justa el rendimiento del modelo y a reducir el riesgo de sobreajuste.

```
def entrenar_modelo_stacking_nolineal_con_transformacion(X_trn, Y_trn, X_tst,
Y_tst,mejores_hiperparametros_tree,mejores_hiperparametros_ADA,
mejores_hiperparametros_extra,mejores_hiperparametros_random):
    semilla= 5
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    base_estimator=DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo2 = AdaBoostRegressor(**mejores_hiperparametros_ADA)
    modelo3 = ExtraTreesRegressor(**mejores_hiperparametros_extra)
    modelo4 = RandomForestRegressor (**mejores_hiperparametros_random)
    model = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo5 = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    estimador_final = ExtraTreesRegressor()
    metrica ='neg_mean_squared_error'
    mejor_modelo = StackingRegressor(
    estimators=[ ('Adaboost', modelo2), ('Extratrees', modelo3),
                ('Random Forest',modelo4),
                ('Decision tree',modelo5)
                ], final_estimator=estimador_final)
```

### *8.4.17 Modelo SuperLearner Regressor con búsqueda de hiperparámetros*

El modelo Super Learner es una estrategia avanzada de ensamble en aprendizaje automático que integra múltiples modelos base, como Extra Trees, Random Forest, Decision Tree y Gradient Boosting, utilizando un meta-modelo para mejorar la precisión de las predicciones. Este enfoque se beneficia de la diversidad y las fortalezas complementarias de los diferentes modelos base, mitigando así el riesgo de sobreajuste mediante la validación cruzada. Al combinar las predicciones de modelos diversos, el Super Learner optimiza el rendimiento general del modelo, ofreciendo una solución robusta y flexible para una amplia gama de problemas de predicción y análisis de datos en diversas aplicaciones prácticas.

La función entrenar_modelo_super_aprendiz_con_transformacion entrena un modelo de regresión utilizando la técnica de ensamble conocida como Super Learner. El Super Learner combina múltiples modelos base y un meta-modelo para mejorar la precisión de las predicciones.

```
def entrenar_modelo_super_aprendiz_con_transformacion(X_trn, Y_trn, X_tst, Y_tst,
mejores_hiperparametros_GD,mejores_hiperparametros_tree,
mejores_hiperparametros_extra,mejores_hiperparametros_random):
    semilla = 5
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    modelo1 = ExtraTreesRegressor(**mejores_hiperparametros_extra)
```

```
    modelo2 = RandomForestRegressor(**mejores_hiperparametros_random)
    model = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo4 = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo5 = GradientBoostingRegressor(**mejores_hiperparametros_GD)
    estimadores = [('Extratrees', modelo1),
                ('Random Forest', modelo2),
                ('Decision tree', modelo4),
                ('Gradient',modelo5)]
    mejor_modelo = SuperLearner(folds=10, random_state=semilla, verbose=2)
    mejor_modelo.add(estimadores)
    estimador_final = ExtraTreesRegressor(n_estimators=100, max_features=None,
    bootstrap=False, max_depth=11,min_samples_split=4,min_samples_leaf=1)
    mejor_modelo.add_meta(estimador_final)
    mejor_modelo.fit(X_trn, Y_trn)
```

### *8.4.18 Modelo SuperLearner dos Capas Regressor con búsqueda de hiperparámetros*

El modelo Super Aprendiz de Dos Capas es una variante avanzada del Super Learner que utiliza dos capas de modelos base y un meta-modelo para mejorar la precisión de las predicciones en aprendizaje automático.

Se emplean modelos base como Extra Trees, Random Forest y Decision Tree, cada uno con sus mejores hiperparámetros ajustados previamente. Estos modelos se integran en el Super Learner, que utiliza validación cruzada para combinar sus predicciones de manera óptima. El meta-modelo final, un Extra Trees Regressor con parámetros específicos, se entrena para aglutinar las predicciones de los modelos base y generar una predicción final más precisa. Este enfoque estratégico no solo aumenta el rendimiento general del modelo, sino que también fortalece su robustez frente a diversos conjuntos de datos y aplicaciones prácticas en análisis predictivo y de datos.

```
Def entrenar_modelo_super_aprendiz_dos_capas(X_trn, Y_trn, X_tst, Y_tst,
mejores_hiperparametros_tree,mejores_hiperparametros_extra,
mejores_hiperparametros_random):
    semilla = 5
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    modelo1 = ExtraTreesRegressor(**mejores_hiperparametros_extra)
    modelo2 = RandomForestRegressor(**mejores_hiperparametros_random)
    model = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    modelo3 = DecisionTreeRegressor(**mejores_hiperparametros_tree)
    estimadores = [('Extratrees', modelo1),
                ('Random Forest', modelo2),
                ('Decision tree', modelo3)]
    mejor_modelo = SuperLearner(folds=10, random_state=semilla, verbose=2)
    mejor_modelo.add(estimadores)
    preci_modelo.add(estimadores)
    estimador_final = ExtraTreesRegressor(n_estimators=100, max_features=None,
    bootstrap=False, max_depth=11, min_samples_split=4,min_samples_leaf=1)
    mejor_modelo.add_meta(estimador_final)
    mejor_modelo.fit(X_trn, Y_trn)
```

## 8.5 Configuración de Validación Cruzada Estratificada

Se configura la validación cruzada con StratifiedKFold para asegurar que las divisiones de los datos preserven la proporción de clases y se mantenga la aleatoriedad.

```
Semilla=5
num_folds=10
kfold =StratifiedKFold(n_splits=num_folds, random_state=semilla, shuffle=True)
```

## 8.6 Definición de la Métrica de Evaluación

Se especifica la métrica de evaluación que se utilizará para optimizar los hiperparámetros del modelo, en este caso, el error cuadrático medio negativo.

```
Metrica = 'neg_mean_squared_error'
```

La métrica neg_mean_squared_error (error cuadrático medio negativo) se utiliza en la evaluación de modelos de regresión y es simplemente el valor negativo del error cuadrático medio (MSE), que mide el promedio de los cuadrados de las diferencias entre los valores reales y los valores predichos por el modelo. Mientras que el MSE es siempre positivo y se minimiza para mejorar el rendimiento del modelo, el neg_mean_squared_error convierte este valor en negativo para que las funciones de optimización puedan maximizarlo. Esto facilita la evaluación y comparación de modelos mediante técnicas como la validación cruzada, ya que maximizar neg_mean_squared_error es equivalente a minimizar el MSE.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$neg\_MSE = -MSE$$

## 8.7 Obtener los Mejores Hiperparámetros

Se realiza una búsqueda en cuadrícula (Grid Search) de los mejores hiperparámetros para un modelo de machine learning, utilizando validación cruzada.

```
Grid = GridSearchCV(estimator=modelo, param_grid=parameters, scoring=metrica,
cv=kfold, n_jobs=-1)
grid_resultado = grid.fit(X_trn, Y_trn)
mejores_hiperparametros_knn = grid_resultado.best_params_
```

GridSearchCV es una clase de sklearn.model_selection que realiza una búsqueda exhaustiva sobre un conjunto especificado de hiperparámetros para un estimador (modelo de machine learning).

- Estimator: modelo: Aquí modelo es el estimador (por ejemplo, un modelo K-Nearest Neighbors, Support Vector Machine, etc.) que se va a optimizar.
- param_grid: parameters: parameters es un diccionario donde las claves son los nombres de los hiperparámetros y los valores son las listas de los valores a probar para esos hiperparámetros.
- Scoring: metrica: metrica es el criterio utilizado para evaluar las combinaciones de hiperparámetros.
- kfold: kfold es la estrategia de validación cruzada a utilizar. Puede ser un número entero para el número de divisiones (folds) o un objeto de validación cruzada (como Kfold, StratifiedKFold, etc.).
- n_jobs: -1: Indica el número de trabajos (procesos) a ejecutar en paralelo. -1 utiliza todos los núcleos disponibles del procesador para acelerar el proceso.

## 8.8 Entrenar el Modelo con los Mejores Hiperparámetros

Se crea y entrena un nuevo modelo utilizando los mejores hiperparámetros encontrados.

```
Mejor_modelo = modelo_implementado(**grid_resultado.best_params_)
mejor_modelo.fit(X_trn, Y_trn)
```

Posteriormente se calcula el coeficiente de determinación $R^2$, que mide la proporción de la varianza en la variable dependiente que es predecible a partir de las variables independientes.

```
R2 = r2_score(Y_tst, predicciones)
return mejor_modelo, r2, mejores_hiperparametros_knn
```

## 8.9 Entrenamiento de Modelos

Esta sección entrena múltiples modelos utilizando diversas funciones. Cada función devuelve el modelo entrenado, el $R^2$ score y los mejores hiperparámetros.

```
Modelo_knn, r2_knn, mejores_hiperparametros_knn =
entrenar_modelo_knn_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)

modelo_svc, r2_svc, mejores_hiperparametros_svc=
entrenar_modelo_svc_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)

modelo_tree, r2_tree,
mejores_hiperparametros_tree=entrenar_modelo_tree_con_transformacion(X_trn, Y_trn,
X_tst, Y_tst)

modelo_gaussian, r2_gaussian,
mejores_hiperparametros_gaussian=entrenar_modelo_gaussian_con_transformacion(X_trn,
Y_trn, X_tst, Y_tst)

modelo_LDA, r2_LDA,
mejores_hiperparametros_LDA=entrenar_modelo_LDA_con_transformacion(X_trn, Y_trn,
X_tst, Y_tst)

modelo_BG, r2_BG,
mejores_hiperparametros_BG=entrenar_modelo_BG_con_transformacion(X_trn, Y_trn, X_tst,
Y_tst, mejores_hiperparametros_tree)

modelo_random, r2_random,
mejores_hiperparametros_random=entrenar_modelo_random_con_transformacion(X_trn,
Y_trn, X_tst, Y_tst)

modelo_extra, r2_extra, mejores_hiperparametros_extra=
entrenar_modelo_extra_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)

modelo_ADA, r2_ADA, mejores_hiperparametros_ADA=
entrenar_modelo_ADA_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)
modelo_GD, r2_GD, mejores_hiperparametros_GD=
entrenar_modelo_GD_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)

modelo_XB, r2_XB, mejores_hiperparametros_XB=
entrenar_modelo_XB_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)

modelo_CB, r2_CB,
mejores_hiperparametros_CB=entrenar_modelo_CB_con_transformacion(X_trn, Y_trn, X_tst,
Y_tst)

modelo_LIGHT, r2_LIGHT, mejores_hiperparametros_LIGHT=
entrenar_modelo_LIGHT_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)

modelo_voting, r2_voting= entrenar_modelo_voting_con_transformacion(X_trn, Y_trn,
X_tst, Y_tst,mejores_hiperparametros_GD,mejores_hiperparametros_tree,
        mejores_hiperparametros_ADA,mejores_hiperparametros_extra,
        mejores_hiperparametros_random,mejores_hiperparametros_BG,
        mejores_hiperparametros_XB)
```

```
        modelo_stacking_lineal, r2_stacking_lineal,
mejores_hiperparametros_stacking_lineal=entrenar_modelo_stacking_lineal_con_transform
acion(X_trn, Y_trn, X_tst, Y_tst,mejores_hiperparametros_tree,
        mejores_hiperparametros_ADA,mejores_hiperparametros_extra,
        mejores_hiperparametros_random)

modelo_stacking_nolineal, r2_stacking_nolineal,
mejores_hiperparametros_stacking_nolineal=entrenar_modelo_stacking_nolineal_con_trans
formacion(X_trn, Y_trn, X_tst, Y_tst, mejores_hiperparametros_tree,
        mejores_hiperparametros_ADA,mejores_hiperparametros_extra,
        mejores_hiperparametros_random)

modelo_super_aprendiz,
r2_super_aprendiz=entrenar_modelo_super_aprendiz_con_transformacion(X_trn, Y_trn,
X_tst, Y_tst, mejores_hiperparametros_GD,mejores_hiperparametros_tree,
mejores_hiperparametros_extra,mejores_hiperparametros_random)

modelo_super_aprendiz_dos_capas,
r2_super_aprendiz_dos_capas=entrenar_modelo_super_aprendiz_dos_capas(X_trn, Y_trn,
X_tst, Y_tst, mejores_hiperparametros_tree,
         mejores_hiperparametros_extra,mejoreshiperparametros_random)

        modelos = {
            'Kneighbors': (modelo_knn, r2_knn),
            'SVC': modelo_svc, r2_svc),
            'DecisionTree': modelo_tree, r2_tree),
            'NaiveBayes': modelo_gaussian,r2_gaussian),
            'LDA': (modelo_LDA, r2_LDA),
            'Bagging': modelo_BG, r2_BG),
            'RandomForest': (modelo_random, r2_random),
            'Extratrees': (modelo_extra, r2_extra),
            'ADA': modelo_ADA, r2_ADA),
            'GradientBoosting': (modelo_GD, r2_GD),
            'XGB': (modelo_XB, r2_XB),
            'CatBoost': (modelo_CB, r2_CB),
            'LIGHT': (modelo_LIGHT, r2_LIGHT),
            'Voting': modelo_voting, r2_voting),
            'Stacking_Lineal': modelo_stacking_lineal, r2_stacking_lineal),
            'Stacking_noLineal': modelo_stacking_nolineal, r2_stacking_nolineal),
            'Super_Aprendiz': (modelo_super_aprendiz,r2_super_aprendiz),
            'Super_Aprendiz_dos_Capas': modelo_super_aprendiz_dos_capas,
             r2_super_aprendiz_dos_capas)}

        mejor_modelo_nombre = max(modelos, key=lambda x: modelos[x][1])
        mejor_modelo, mejor_precision = modelos[mejor_modelo_nombre]
        print("Mejor modelo:", mejor_modelo_nombre)
        print("Exactitud del modelo:", mejor_precision)
    except Exception as e:
        print("Error al cargar y entrenar el modelo:", e)
```

# 9. Módulo Analítica Práctica Predicción Grupal Regresión

## 9.1 Definir Variables por Carrera

Se tienen definidos los conjuntos de variables relevantes para cada carrera y semestre en un diccionario llamado variables_por_carrera. Este diccionario mapea cada carrera y semestre a una lista de variables específicas para entrenar los modelos a implementar. Se define un segundo diccionario el cual filtra el archivo recibido para la predicción masiva de estudiantes.

```
Variables_por_carrera = {
    'industrial': {
        '1': ['PG_ICFES', 'CON_MAT_ICFES',
'FISICA_ICFES','QUIMICA_ICFES','IDIOMA_ICFES','LOCALIDAD','PROMEDIO_UNO'],
    …
    },
    'sistemas': {
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','LOCALIDAD_COLEGIO','BIOLOGIA_ICFES','QUIMICA_ICFES',
'PG_ICFES','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    },
    'catastral': {
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','BIOLOGIA_ICFES','GENERO','FILOSOFIA_ICFES','LOCALIDA
D','LITERATURA_ICFES','QUIMICA_ICFES','PG_ICFES','PROMEDIO_UNO'],
   …
    },
    'electrica':{
        '1':
['IDIOMA_ICFES','PG_ICFES','FISICA_ICFES','ANO_INGRESO','APT_VERB_ICFES','QUIMICA_ICF
ES','FILOSOFIA_ICFES','GENERO','BIOLOGIA_ICFES','PROMEDIO_UNO'],

    },
    'electronica':{
        '1':
['CON_MAT_ICFES','QUIMICA_ICFES','PG_ICFES','FISICA_ICFES','IDIOMA_ICFES','BIOLOGIA_I
CFES','GENERO','LOCALIDAD','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    }
}
```

```
variables_por_carrera2 = {
    'industrial': {
        '1': ['PG_ICFES', 'CON_MAT_ICFES',
'FISICA_ICFES','QUIMICA_ICFES','IDIOMA_ICFES','LOCALIDAD','PROMEDIO_UNO'],
    …
    },
    'sistemas': {
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','LOCALIDAD_COLEGIO','BIOLOGIA_ICFES','QUIMICA_ICFES',
'PG_ICFES','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    },
    'catastral': {
```

```
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','BIOLOGIA_ICFES','GENERO','FILOSOFIA_ICFES','LOCALIDA
D','LITERATURA_ICFES','QUIMICA_ICFES','PG_ICFES','PROMEDIO_UNO'],
   …
    },
    'electrica':{
        '1':
['IDIOMA_ICFES','PG_ICFES','FISICA_ICFES','ANO_INGRESO','APT_VERB_ICFES','QUIMICA_ICF
ES','FILOSOFIA_ICFES','GENERO','BIOLOGIA_ICFES','PROMEDIO_UNO'],

    },
    'electronica':{
        '1':
['CON_MAT_ICFES','QUIMICA_ICFES','PG_ICFES','FISICA_ICFES','IDIOMA_ICFES','BIOLOGIA_I
CFES','GENERO','LOCALIDAD','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    }
}
```

## 9.2 Resultados de Predicción Grupal y Selección de Modelo de Regresión

En este paso se crea un diccionario llamado "exactitudes" que mapea el nombre de cada modelo con su respectiva precisión ($R^2$ score). Aquí, r2_(modelo) son las precisiones de cada modelo. Por su parte modelos_entrenados es un diccionario que mapea el nombre de cada modelo con el modelo entrenado correspondiente. Aquí, modelo son las instancias de cada modelo entrenado.

```
        Exactitudes = {
            'KNEIGHBORSCLASSIFIER': r2_knn,
            'SVC': r2_svc,
            'DECISION_TREE': r2_tree,
            'NAÏVE_BAYES': r2_gaussian,
            'LDA': r2_LDA,
            'BAGGING': r2_BG,
            'RANDOM_FOREST': r2_random,
            'EXTRATREE': r2_extra,
            'ADA': r2_ADA,
            'GRADIENTBOOST': r2_GD,
            'XGBOOST': r2_XB,
            'CATBOOST': r2_CB,
            'LIGHT': r2_LIGHT,
            'VOTING': r2_voting,
            'STACKING_LINEAL': r2_stacking_lineal,
            'STACKING_NO_LINEAL': r2_stacking_nolineal,
            'SUPER_APRENDIZ': r2_super_aprendiz,
            'SUPER_APRENDIZ_DOS_CAPAS': r2_super_aprendiz_dos_capas}

        modelos_entrenados = {
            'KNEIGHBORSCLASSIFIER': modelo_knn,
            'SVC': modelo_svc,
            'DECISION_TREE': modelo_tree,
            'NAIVE_BAYES': modelo_gaussian,
            'LDA': modelo_LDA,
            'BAGGING': modelo_BG,
            'RANDOM_FOREST':modelo_random,
            'EXTRATREE': modelo_extra,
            'ADA': modelo_ADA,
```

```
            'GRADIENTBOOST': modelo_GD,
            'XGBOOST': modelo_XB,
            'CATBOOST': modelo_CB,
            'LIGHT': modelo_LIGHT,
            'VOTING': modelo_voting,
            'STACKING_LINEAL': modelo_stacking_lineal,
            'STACKING_NO_LINEAL': modelo_stacking_nolineal,
            'SUPER_APRENDIZ': modelo_super_aprendiz,
            'SUPER_APRENDIZ_DOS_CAPAS': modelo_super_aprendiz_dos_capas
            }
        mejor_modelo = max(exactitudes, key=exactitudes.get)
        modelo_seleccionado = modelos_entrenados.get(mejor_modelo)
        print("Mejor modelo:", mejor_modelo)
        print(exactitudes)
```

A continuación, verifica si el modelo seleccionado no es None y, si es así, realiza predicciones sobre nuevos datos transformados utilizando una transformación Johnson. Luego, agrega estas predicciones a un DataFrame y guarda el DataFrame en un archivo CSV. Convierte el DataFrame a una lista de diccionarios y guarda estos datos en un archivo JSON. Finalmente, imprime un mensaje confirmando que los datos se han guardado en el archivo JSON. Si el modelo seleccionado es None, imprime un mensaje indicando que el modelo no está disponible. Es importante destacar que los modelos implementados para este módulo cumplen la estructura anteriormente indicada en el apartado de "Módulo Analítica Práctica Predicción Individual Regresión".

```
        If modelo_seleccionado is not None:
            predicciones_nuevos_datos =
modelo_seleccionado.predict(transformacion_johnsonX_prediccion))
            df_prediccion[f'PROMEDIO_{semestre_en_letras.upper()}']=predicciones_nuev
os_datos
            df_prediccion
            df_prediccion.to_csv(f'Prediccion_Regresion_{carrera}_{semestre}.csv',sep
=";",index=False)
            data_with_columns = df_prediccion.to_dict(orient='records')
            diccionario_dataframes = [
                    {
                        'dataTransformacion': data_with_columns,
                    }
                ]
            with open("Prediccion_Regresion.json", "w") as json_file:
                json.dump({"data": diccionario_dataframes}, json_file, indent=4)
                print("Los DataFrames han sido guardados en
'Prediccion_Regresion.json'.")
        else:
            print("El modelo seleccionado no está disponible.")

    except Exception as e:
        print("Error al cargar y entrenar el modelo:", e)
```

## 10. Módulo Analítica Práctica Predicción Individual Anual Regresión

La predicción anualizada dispone de las variables influyentes derivadas del módulo de selección de características agrupadas por tres años. Cada uno de estos consecutivamente van tomando variables de semestre pasados sin contemplar las del semestre referente al año indicado. Por ejemplo, si se desea predecir el promedio o rendimiento para el primer año, se tomarán las variables pre universitarias más las del primer semestre, sin contemplar las variables correspondientes al segundo semestre.

```
Variables_por_carrera = {
    ‘industrial’: {
        ‘1’:
[‘PG_ICFES’,’CON_MAT_ICFES’,’FISICA_ICFES’,’QUIMICA_ICFES’,’IDIOMA_ICFES’,’PROMEDIO_U
NO’,’LOCALIDAD_COLEGIO’,’BIOLOGIA_ICFES’,
             ‘LOCALIDAD’,’CAR_UNO’,’NCC_UNO’,’NAA_UNO’,’NOTA_DIFERENCIAL’,’NOTA_DIBU
JO’,’NOTA_QUIMICA’,’NOTA_CFJC’,’NOTA_TEXTOS’,’NOTA_SEMINARIO’,’NOTA_EE_UNO’,
             ‘PROMEDIO_DOS’,’PROMEDIO_TRES’],

        ‘2’: [‘PG_ICFES’, ‘CON_MAT_ICFES’,
‘FISICA_ICFES’,’QUIMICA_ICFES’,’IDIOMA_ICFES’,’LOCALIDAD’, ‘PROMEDIO_UNO’,
‘LOCALIDAD_COLEGIO’,
             ‘BIOLOGIA_ICFES’, ‘CAR_UNO’, ‘NCC_UNO’, ‘NAA_UNO’, ‘NOTA_DIFERENCIAL’,
‘NOTA_DIBUJO’, ‘NOTA_QUIMICA’, ‘NOTA_CFJC’, ‘NOTA_TEXTOS’, ‘NOTA_SEMINARIO’,
‘NOTA_EE_UNO’,
             ‘PROMEDIO_DOS’,’NCC_DOS’, ‘NCA_DOS’, ‘NAA_DOS’, ‘NOTA_ALGEBRA’,
‘NOTA_INTEGRAL’, ‘NOTA_MATERIALES’, ‘NOTA_PBASICA’, ‘NOTA_EE_DOS’,
             ‘PROMEDIO_TRES’,’NAA_TRES’, ‘NOTA_MULTIVARIADO’,
‘NOTA_ESTADISTICA_UNO’, ‘NOTA_TERMODINAMICA’, ‘NOTA_TGS’,
‘NOTA_EE_TRES’,’PROMEDIO_CUATRO’, ‘PROMEDIO_CINCO’],

        ‘3’: [‘PG_ICFES’, ‘CON_MAT_ICFES’,
‘FISICA_ICFES’,’QUIMICA_ICFES’,’IDIOMA_ICFES’,’LOCALIDAD’,’PROMEDIO_UNO’,
‘LOCALIDAD_COLEGIO’, ‘BIOLOGIA_ICFES’, ‘CAR_UNO’,
             ‘NCC_UNO’, ‘NAA_UNO’, ‘NOTA_DIFERENCIAL’, ‘NOTA_DIBUJO’,
‘NOTA_QUIMICA’, ‘NOTA_CFJC’, ‘NOTA_TEXTOS’, ‘NOTA_SEMINARIO’,
‘NOTA_EE_UNO’,’PROMEDIO_DOS’, ‘NCC_DOS’,
             ‘NCA_DOS’, ‘NAA_DOS’, ‘NOTA_ALGEBRA’, ‘NOTA_INTEGRAL’,
‘NOTA_MATERIALES’, ‘NOTA_PBASICA’, ‘NOTA_EE_DOS’,’PROMEDIO_TRES’, ‘NAA_TRES’,
‘NOTA_MULTIVARIADO’, ‘NOTA_ESTADISTICA_UNO’,
             ‘NOTA_TERMODINAMICA’, ‘NOTA_TGS’, ‘NOTA_EE_TRES’,’PROMEDIO_CUATRO’,
‘NOTA_ECUACIONES’, ‘NOTA_ESTADISTICA_DOS’, ‘NOTA_FISICA_DOS’, ‘NOTA_MECANICA’,
‘NOTA_PROCESOSQ’,
             ‘PROMEDIO_CINCO’, ‘NOTA_EE_CUATRO’, ‘NOTA_PROCESOSM’,
‘NOTA_ADMINISTRACION’, ‘NOTA_LENGUA_UNO’, ‘NOTA_EI_UNO’,
‘NOTA_EI_DOS’,’PROMEDIO_SEIS’, ‘NCA_SEIS’, ‘NOTA_PLINEAL’, ‘NOTA_DISENO’,
             ‘NOTA_EI_TRES’,’PROMEDIO_SIETE’, ‘NOTA_IECONOMICA’, ‘NAA_SIETE’,
‘NOTA_GRAFOS’, ‘NOTA_CALIDAD_UNO’, ‘NOTA_ERGONOMIA’,
‘NOTA_EI_CINCO’,’PROMEDIO_OCHO’,’PROMEDIO_NUEVE’]
    },
‘sistemas’: {
    ‘1’: …
    ‘2’: …
    ‘3’: …
    },
‘catastral: {
    ‘1’: …
```

```
    ‘2’: …
    ‘3’: …
    },
‘electrica: {
    ‘1’: …
    ‘2’: …
    ‘3’: …
    },
‘electronica: {
    ‘1’: …
    ‘2’: …
    ‘3’: …
    }
}
```

La función cargar_entrenar_modelo define una variable global mejor_modelo y selecciona una columna específica de rendimiento académico (PROMEDIO_TRES, PROMEDIO_CINCO, o PROMEDIO_NUEVE) basada en el año proporcionado (anio). Si el año no es válido (distinto de 1, 2, o 3), lanza un error. Luego, intenta cargar datos para una carrera y año específicos, filtra las columnas relevantes para esa combinación de carrera y año, convierte los datos a enteros, y separa las características (X) de la variable objetivo (Y) excluyendo la columna de promedio seleccionado.

```
Def cargar_entrenar_modelo():
    global mejor_modelo

    if anio == “1”:
        rendimiento_columna = ‘PROMEDIO_TRES’
        print(rendimiento_columna)
    elif anio == “2”:
        rendimiento_columna = ‘PROMEDIO_CINCO’
        print(rendimiento_columna)
    elif anio == “3”:
        rendimiento_columna = ‘PROMEDIO_NUEVE’
        print(rendimiento_columna)
    else:
        raise ValueError(“anio no válido. Solo se permiten los anios 1, 2 y 3.”)

    try:
        datos = cargar_datos(carrera, anio)
        columnas_filtradas = variables_por_carrera[carrera][anio]
        df = datos[columnas_filtradas].astype(int)
        anio_en_letras = numero_a_letras(anio)
        X = df.loc[:, ~df.columns.str.contains(rendimiento_columna)]
        Y = df.loc[:, df.columns.str.contains(rendimiento_columna)]
```

```
        modelos = {
            ‘Kneighbors’: (modelo_knn, r2_knn),
            ‘SVC’: modelo_svc, r2_svc),
            ‘DecisionTree’: modelo_tree, r2_tree),
            ‘NaiveBayes’: modelo_gaussian,r2_gaussian),
            ‘LDA’: (modelo_LDA, r2_LDA),
            ‘Bagging’: modelo_BG, r2_BG),
            ‘RandomForest’: (modelo_random, r2_random),
            ‘Extratrees’: (modelo_extra, r2_extra),
```

```
            'ADA': modelo_ADA, r2_ADA),
            'GradientBoosting': (modelo_GD, r2_GD),
            'XGB': (modelo_XB, r2_XB),
            'CatBoost': (modelo_CB, r2_CB),
            'LIGHT': (modelo_LIGHT, r2_LIGHT),
            'Voting': modelo_voting, r2_voting),
            'Stacking_Lineal': modelo_stacking_lineal, r2_stacking_lineal),
            'Stacking_noLineal': modelo_stacking_nolineal, r2_stacking_nolineal),
            'Super_Aprendiz': (modelo_super_aprendiz,r2_super_aprendiz),
            'Super_Aprendiz_dos_Capas':
modelo_super_aprendiz_dos_capas,r2_super_aprendiz_dos_capas)
        }
        mejor_modelo_nombre = max(modelos, key=lambda x: modelos[x][1])
        mejor_modelo, mejor_precision = modelos[mejor_modelo_nombre]
        print("Mejor modelo:", mejor_modelo_nombre)
        print("Exactitud del modelo:", mejor_precision)
    except Exception as e:
        print("Error al cargar y entrenar el modelo:", e)
```

# 11. Módulo Analítica Práctica Predicción Individual Clasificación

## 11.1 Configuración Inicial y Carga de Bibliotecas

### *11.1.1 Advertencias y configuración de visualización*

Para evitar advertencias innecesarias que puedan entorpecer la salida del programa, se desactivan las advertencias. Además, se configuran opciones de visualización para mejorar la legibilidad de las salidas.

```
Import warnings
warnings.filterwarnings('ignore')
```

### *11.1.2 Manejo y visualización de datos*

Se utilizan varias bibliotecas para la manipulación y visualización de datos:

- NumPy: Para operaciones matemáticas y manejo de arrays.
- Pandas: Para manipulación y análisis de datos.
- Matplotlib y Seaborn: Para visualización de datos.
- Scatter Matrix de Pandas: Para la creación de matrices de dispersión.

```
From numpy import set_printoptions
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

### *11.1.3 Manejo de archivos y formatos diversos*

Se importan bibliotecas para manejar archivos en diferentes formatos, incluyendo Excel, imágenes, y JSON:

- io y base64: Para manejo de streams y codificación de datos en base64.
- json: Para manejar datos en formato JSON.
- openpyxl: Para trabajar con archivos de Excel.
- os y os.path: Para manejo de rutas y archivos en el sistema operativo.
- unidecode: Para normalizar texto, removiendo acentos y caracteres especiales.

```
Import io
import base64
import json
from openpyxl import Workbook
from openpyxl.drawing.image import Image as XLImage
import os
import os.path
from unidecode import unidecode
```

### *11.1.4 Preprocesamiento de datos*

Para preprocesar los datos antes de entrenar los modelos, se utilizan diversas técnicas de escalado y transformación de características:

- LabelEncoder: Para codificación de etiquetas categóricas.
- MinMaxScaler, StandardScaler, Normalizer, RobustScaler, PowerTransformer: Para diferentes métodos de escalado y normalización.

```
From sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import PowerTransformer
```

### *11.1.5 Modelos y validación*

Se utilizan varias estrategias de validación y selección de modelos, incluyendo validación cruzada y búsqueda de hiperparámetros con grid search:

- GridSearchCV, Kfold, StratifiedKFold, train_test_split, cross_val_score: Para validación y precisió de datos.

```
From sklearn.model_selection import GridSearchCV
from sklearn.model_selection import Kfold
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
```

### *11.1.6 Algoritmos de regresión*

El script incluye una amplia variedad de algoritmos de clasificación para comparar y seleccionar el mejor modelo:

- Modelos de Ensamble: Bagging, Random Forest, Extra Trees, AdaBoost, Gradient Boosting, XGBoost, CatBoost, LDA, LightGBM, Voting, Stacking, SuperLearner.
- Modelos Lineales: Linear Regression.
- Modelos de Vecinos Cercanos: KneighborsClassifier.
- Máquinas de Soporte Vectorial: SVR.
- Modelos Gaussianos: Gaussian Process Regressor con distintos kernels.
- Redes Neuronales.

```
From sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KneighborsClassifier
from sklearn.svm import SVC
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from mlens.ensemble import SuperLearner
import torch
import torch.nn as nn
import torch.optim as optim
```

### *11.1.7 Métricas de evaluación*

Para evaluar el rendimiento de los modelos, se utilizan varias métricas, incluyendo error cuadrático medio, error absoluto medio, coeficiente de determinación, entre otros:

```
from sklearn.metrics import accuracy_score
```

## 11.2 Proceso de Carga y Transformación de Datos

En este punto se detalla el procedimiento para cargar y transformar datos académicos para diferentes carreras y semestres en la Universidad Distrital. El proceso incluye la selección de columnas específicas para

cada carrera y semestre, la carga de datos desde archivos CSV y la aplicación de la transformación Yeo-Johnson.

### *11.2.1 Configuración inicial de variables por carrera*

Se tienen definidos los conjuntos de variables relevantes para cada carrera y semestre en un diccionario llamado "variables_por_carrera". Este diccionario mapea cada carrera y semestre a una lista de variables específicas.

```
Variables_por_carrera = {
    'industrial': {
        '1': ['PG_ICFES', 'CON_MAT_ICFES',
'FISICA_ICFES','QUIMICA_ICFES','IDIOMA_ICFES','LOCALIDAD','PROMEDIO_UNO'],
    …
    },
    'sistemas': {
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','LOCALIDAD_COLEGIO','BIOLOGIA_ICFES','QUIMICA_ICFES',
'PG_ICFES','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    },
    'catastral': {
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','BIOLOGIA_ICFES','GENERO','FILOSOFIA_ICFES','LOCALIDA
D','LITERATURA_ICFES','QUIMICA_ICFES','PG_ICFES','PROMEDIO_UNO'],
   …
    },
    'electrica':{
        '1':
['IDIOMA_ICFES','PG_ICFES','FISICA_ICFES','ANO_INGRESO','APT_VERB_ICFES','QUIMICA_ICF
ES','FILOSOFIA_ICFES','GENERO','BIOLOGIA_ICFES','PROMEDIO_UNO'],

    },
    'electronica':{
        '1':
['CON_MAT_ICFES','QUIMICA_ICFES','PG_ICFES','FISICA_ICFES','IDIOMA_ICFES','BIOLOGIA_I
CFES','GENERO','LOCALIDAD','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    }
}
```

### *11.2.2 Función para cargar datos*

Para cargar los datos de una carrera específica y un semestre determinado, se utiliza la función "cargar_datos". Esta función lee un archivo CSV ubicado en una ruta específica y retorna un DataFrame con los datos cargados.

```
Def cargar_datos(carrera, semestre):
    ruta_archivo = f'C:/Users/Desktop/UNIVERSIDAD DISTRITAL PROYECTO
FOLDER/UNIVERSIDAD-DISTRITAL-
PROYECTO/MODULO_ANALITICA_PRACTICO/DATOS/{carrera}{semestre}.csv'
    datos = pd.read_csv(ruta_archivo,sep=";")
    return datos
```

### *11.2.3 Cargar datos y seleccionar columnas*

Cargamos los datos y seleccionamos las columnas específicas definidas para la carrera y semestre correspondientes.

```
Datos = cargar_datos(carrera, semestre)
columnas_filtradas = variables_por_carrera[carrera][semestre]
df = datos[columnas_filtradas]
print("DataFrame con columnas filtradas:")
df=df.astype(int)
df
```

#### *11.2.4 Separación de variables independientes y dependientes*

Se separan las variables independientes (X) de la variable dependiente (Y), correspondiente al rendimiento del semestre. Adicionalmente se transforman los datos de la variable dependiente a etiquetas.

```
X = df.loc[:, ~df.columns.str.contains(f'RENDIMIENTO_{semestre_en_letras.upper()}')]
Y = df.loc[:, df.columns.str.contains(f'RENDIMIENTO_{semestre_en_letras.upper()}')]
Y = LabelEncoder().fit_transform(Y.astype('str'))
```

#### *11.2.5 Conversión de tipos de datos*

Las variables independientes se convierten a tipo float32 para la posterior transformación.

```
X = X.astype('float32')
print(X.shape, Y.shape)
```

#### *11.2.6 Transformación Yeo-Johnson*

Se realiza una transformación Johnson en los datos para normalizarlos y hacerlos adecuados para el modelado.

```
X_T_JOHNSON1 = X.copy(deep=True)
def transformacion_johnson(X):
    transformador_johnson = PowerTransformer(method='yeo-johnson',
standardize=True).fit(X)
    datos_transformados = transformador_johnson.transform(X)
    set_printoptions(precisión=3)
    print(datos_transformados[:5, :])
    datos_transformados_df = pd.DataFrame(data=datos_transformados,
columns=X.columns)
    return datos_transformados_df
Xpandas_T_JOHNSON1 = transformacion_johnson(X_T_JOHNSON1)
Xpandas_T_JOHNSON1.head(2)
```

## 11.3 División de Datos en Conjuntos de Entrenamiento y Prueba

La división de los datos en conjuntos de entrenamiento y prueba es un paso crucial en el proceso de modelado de datos. Esta técnica permite evaluar el rendimiento de un modelo con datos que no se utilizaron durante su entrenamiento, proporcionando una estimación más precisa de su capacidad de generalización a datos nuevos. Utilizamos la función train_test_split del módulo model_selection de scikit-learn para realizar esta división.

La función train_test_split toma como entrada las características (X) y las etiquetas (Y), y divide los datos en cuatro subconjuntos:

- X_trn: Características para el conjunto de entrenamiento.
- X_tst: Características para el conjunto de prueba.
- Y_trn: Etiquetas para el conjunto de entrenamiento.
- Y_tst: Etiquetas para el conjunto de prueba.

Los parámetros usados son:

- X_transformado: Las características del conjunto de datos.
- Y: Las etiquetas del conjunto de datos.

- test_size=0.3: Especifica que el 30% de los datos se asignarán al conjunto de prueba y el 70% restante al conjunto de entrenamiento.
- random_state=2: Fija la semilla para la generación de números aleatorios, garantizando que la división de datos sea reproducible.

```
X_transformado = transformacion_johnson(X)
X_trn, X_tst, Y_trn, Y_tst = train_test_split(X_transformado, Y, test_size=0.3,
random_state=2)
```

## 11.4 Desarrollo de Modelos de Machine Learning – Clasificación

### *11.4.1 Modelo K-Nearest Neighbors (KNN) con búsqueda de hiperparámetros*

El modelo K-Nearest Neighbors (KNN) es un algoritmo de machine learning utilizado para problemas de clasificación y regresión. Es un método basado en instancias que clasifica un punto nuevo de datos basándose en la mayoría de las categorías de sus vecinos más cercanos en el espacio de características. Para clasificación, KNN asigna la clase más común entre sus k vecinos más cercanos.
Parámetros para la búsqueda:

```
parameters = {'n_neighbors': [i for i in range(1, 18, 1)],
              'metric': ['euclidean', 'manhattan', 'minkowski'],
              'algorithm': ['auto', 'kd_tree','ball_tree','brute'],
              'weights': ['uniform']}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- n_neighbors: Número de vecinos a considerar (de 1 a 18).
- metric: Métrica de distancia a utilizar (Euclidean, Manhattan, Minkowski).
- algorithm: Algoritmo automático para calcular vecinos ('auto' selecciona automáticamente el más adecuado, también puede usar 'kd_tree', 'ball_tree', o 'brute').
- weights: Tipo de ponderación (uniforme asigna el mismo peso a todos los vecinos).

Inicialización del Modelo:

```
modelo = KneighborsClassifier()
```

### *11.4.2 Modelo Support Vector Classifier (SVC) con búsqueda de hiperparámetros*

El modelo Support Vector Classifier (SVC) es un algoritmo de aprendizaje supervisado utilizado para problemas de clasificación. Se basa en encontrar el hiperplano óptimo que mejor separa las clases en el espacio de características, maximizando el margen entre las clases.
Parámetros para la búsqueda:

```
parameters = {'kernel':   ['rbf', 'poly', 'sigmoid','linear'],
                'C': [i/10000 for i in range(8,12,1)],
                'max_iter':[i for i in range(1,3,1)],
                'gamma' : [i/100 for i in range(80,110,5)],
                'random_state':[i for i in range(1,5,1)]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- Kernel Function: El parámetro kernel en SVR define el tipo de función kernel a utilizar, que determina cómo se transforman los datos a un espacio de características de mayor dimensión donde se puede encontrar un hiperplano separador. Los kernels comunes incluyen:
    - RBF (Radial Basis Function): Función gaussiana que mide la distancia entre un punto y los demás puntos.
    - Polynomial: Transformación polinómica que puede manejar relaciones no lineales entre los datos.

  - Sigmoid: Similar a una función de activación en redes neuronales, puede manejar funciones no lineales.
  - Linear: Utiliza una función lineal para la transformación, útil para problemas lineales.
- Regularización ©: Controla el balance entre el ajuste exacto de los datos de entrenamiento y la generalización del modelo para nuevos datos. Un valor más alto de C permite un ajuste más preciso de los datos de entrenamiento, pero podría llevar a un sobreajuste.
- Gamma (γ): Este parámetro afecta la influencia de cada ejemplo de entrenamiento. Un valor más alto de gamma significa que solo los puntos de datos cercanos tendrán un efecto significativo en la predicción, lo que conduce a un modelo más ajustado a los datos de entrenamiento (puede llevar a sobreajuste si se configura muy alto).
- Máximo de Iteraciones (max_iter): Establece el número máximo de iteraciones permitidas para la optimización del modelo. Aumentar este valor puede ser necesario si el modelo no converge con el número predeterminado de iteraciones.
- random_state: Descripción: Semilla utilizada por el generador de números aleatorios para reproducibilidad (dentro del rango de 1 a 5).

Inicialización del Modelo:

```
modelo = SVC()
```

### *11.4.3 Modelo Decision Tree Classifier con búsqueda de hiperparámetros*

El modelo Decision Tree Classifier es un algoritmo de aprendizaje supervisado utilizado para problemas de clasificación. Construye un árbol de decisiones recursivamente, dividiendo el conjunto de datos en subconjuntos cada vez más pequeños mientras maximiza la pureza de las clases en los nodos hoja.

Parametros para la búsqueda:

```
parameters = {'max_depth':[i for i in range(1,7,1)],
              'min_samples_split' :  [i for i in range(1,7,1)],
              'min_samples_leaf' : [i for i in range(1,7,1)],
              'max_features' : [i for i in range(1,7,1)],
              'splitter': ["best", "random"],
              'random_state': [i for i in range(1,5,1)]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- max_depth: La profundidad máxima del árbol, controla la profundidad máxima del árbol, limitando el número de divisiones.
- min_samples_split: Número mínimo de muestras requeridas para dividir un nodo interno, controla la cantidad mínima de muestras requeridas para considerar una división en el árbol.
- min_samples_leaf: Número mínimo de muestras requeridas en un nodo hoja, controla la cantidad mínima de muestras requeridas en los nodos hoja del árbol.
- max_features: Número máximo de características a considerar para realizar la mejor división, limita el número de características que se consideran al buscar la mejor división en cada nodo.
- splitter: Estrategia utilizada para elegir la división en cada nodo, determina cómo se selecciona la división en cada nodo del árbol. ("best" (elige la mejor división) o "random" (elige la mejor división aleatoria)).
- random_state: Semilla utilizada por el generador de números aleatorios para reproducibilidad. Garantiza que el modelo se comporte de manera reproducible y consistente en diferentes ejecuciones.

Inicialización del Modelo:

```
modelo = DecisionTreeClassifier()
```

### *11.4.4 Modelo Gaussian Process Classifier con búsqueda de hiperparámetros*

El modelo Naive Bayes es un método de aprendizaje supervisado basado en el teorema de Bayes, que asume que la presencia de una característica en una clase no está relacionada con la presencia de ninguna otra característica. A continuación se explica su funcionamiento para clasificación:

- Teorema de Bayes: El modelo Naive Bayes se basa en el teorema de Bayes, que establece cómo se puede actualizar la probabilidad de una hipótesis dada la evidencia.
- Independencia condicional: Naive Bayes asume que todas las características son independientes entre sí, dado el valor de la variable de clase. Esto se conoce como la "naive" o ingenua suposición de independencia. Aunque esta suposición rara vez es cierta en la práctica, el modelo puede ser sorprendentemente efectivo en muchas situaciones.
- Modelo generativo: Naive Bayes es un modelo generativo, lo que significa que calcula la probabilidad conjunta de las características y la variable de clase. Utiliza esta probabilidad conjunta para calcular la probabilidad posterior de la clase dada una instancia de datos.
- Entrenamiento: Durante la fase de entrenamiento, el modelo calcula las siguientes probabilidades:
    - Probabilidades a priori: Probabilidad de que una instancia pertenezca a una clase específica sin considerar ninguna característica.
    - Probabilidades condicionales: Probabilidad de que una instancia tenga ciertos valores de características dados que pertenece a una clase específica.
    - Estas probabilidades se estiman a partir de los datos de entrenamiento utilizando la frecuencia de ocurrencia de las características y las clases.
- Predicción: Para predecir la clase de una nueva instancia de datos, Naive Bayes utiliza el teorema de Bayes para calcular la probabilidad posterior de cada clase dado el conjunto de características. Se selecciona la clase con la mayor probabilidad posterior como la predicción del modelo.

Inicialización del Modelo:

```
modelo = GaussianNB()
```

### *11.4.5 Modelo Linear Discriminant Analysis (LDA) con búsqueda de hiperparámetros*

El modelo Linear Discriminant Analysis (LDA) es un método de aprendizaje supervisado utilizado para la clasificación y la reducción de dimensionalidad. LDA busca maximizar la separabilidad entre múltiples clases proyectando los datos en un espacio de menor dimensión mientras preserva la información discriminativa.
Parametros para la búsqueda:

```
parameters = {'solver': ['svd','lsqr','eigen'],
              'n_components':[1,2,3,4,5,6,7,8,9,10],
              'shrinkage': ['auto', 0.001, 0.01, 0.1, 0.5,1,10,100,1000],
              'tol':[i/1000 for i in range(1,100,1)]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- solver: Método utilizado para resolver el problema de optimización en LDA.
    - 'svd': Descomposición de valores singulares, adecuado para datos grandes.
    - 'lsqr': Mínimos cuadrados, eficiente para matrices grandes y densas.
    - 'eigen': Descomposición eigen, adecuado para problemas de pequeño tamaño.
- n_components: Número de componentes (dimensiones) a mantener después de la reducción de dimensionalidad
- shrinkage: Parámetro de regularización para mejorar la estabilidad y rendimiento de LDA cuando solver='lsqr'.
- tol: Define el nivel de precisión requerido para la convergencia en los métodos iterativos.

Inicialización del Modelo:

```
modelo = LinearDiscriminantAnalysis()
```

### *11.4.6 Modelo Bagging Classifier con búsqueda de hiperparámetros*

El modelo Bagging (Bootstrap Aggregating) es una técnica de ensamblado utilizada para mejorar la precisión y estabilidad de los modelos de aprendizaje automático. Consiste en entrenar múltiples modelos base utilizando diferentes muestras del conjunto de datos y combinar sus predicciones para mejorar el rendimiento general del modelo.

Parametros para la búsqueda:

```
parameters = {'n_estimators': [i for i in range(750,760,5)],
              'max_samples' : [i/100.0 for i in range(70,90,5)],
              'max_features': [i/100.0 for i in range(75,85,5)],
              'bootstrap': [True], 'bootstrap_features': [True]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- n_estimators: Controla la cantidad de modelos base en el ensamble, cada uno entrenado en una muestra precisión diferente del conjunto de datos.
- max_samples: Determina el tamaño de la muestra precisión para cada estimador, afectando la variabilidad de cada modelo base.
- max_features: Controla la variabilidad de los modelos base al limitar las características utilizadas en cada uno, lo que promueve la diversidad en el ensamble.
- precisión: Permite que cada estimador base se entrene en una muestra precisión del conjunto de datos original, mejorando la robustez y la generalización del modelo final.
- precisión_features: Permite que cada estimador base se entrene en subconjuntos aleatorios de características, lo que ayuda a mejorar la diversidad y la precisión del ensamble.

Inicialización del Estimador Base:

El base_estimator se refiere al modelo base que se utilizará en cada uno de los estimadores individuales dentro del ensamble de Bagging.

```
Base_estimator= DecisionTreeClassifier(**mejores_hiperparametros_tree)
```

Inicialización del Modelo:

```
modelo = BaggingClassifier(estimator=base_estimator)
```

### *11.4.7 Modelo Random Forest Classifier con búsqueda de hiperparámetros*

El modelo Random Forest Classifier es un algoritmo de ensamble que utiliza múltiples árboles de decisión entrenados en diferentes subconjuntos del conjunto de datos y promedia sus predicciones para mejorar la precisión y la generalización del modelo.

Parámetros para la búsqueda:

```
parameters={'min_samples_split':[1,2,3,4,6,8,10,15,20],
            'min_samples_leaf':[1,3,5,7,9,12,15],
'criterion': ('gini','entropy','log_loss')}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- min_samples_split: Controla la cantidad mínima de muestras que deben estar presentes en un nodo para que se considere una división.
- min_samples_leaf: Controla la cantidad mínima de muestras que deben estar presentes en un nodo hoja después de una división.
- criterion: Función para medir la calidad de una división en un árbol de decisión.
    - 'gini': Utiliza el índice de Gini como criterio para la pureza de la división.
    - 'entropy': Utiliza la ganancia de información (entropía) como criterio para la pureza de la división.

- ‘log_loss’: Utiliza la pérdida logarítmica (también conocida como entropía cruzada) como criterio para la calidad de la división.

Inicialización del Modelo:

```
modelo = RandomForestClassifier()
```

### *11.4.8 Modelo Extratree classifier con búsqueda de hiperparámetros*

El modelo Extra Trees Classifier es otro tipo de algoritmo de ensamble basado en árboles de decisión, similar al Random Forest, pero con diferencias clave en cómo se construyen los árboles y se calculan las divisiones.

Parámetros para la busqueda:

```
parameters = {‘min_samples_split’ : [i for i in range(1,10,1)], ‘min_samples_leaf’ :
[i for i in range(0,10,1)],
‘min_samples_leaf’:[iforiinrange(0,10,1)],’min_samples_split’:[iforiinrange(0,10,1)],
                ‘criterion’: (‘gini’,’entropy’,’log_loss’)}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- min_samples_split: Controla la cantidad mínima de muestras que deben estar presentes en un nodo para que se considere una división.
- min_samples_leaf: Controla la cantidad mínima de muestras que deben estar presentes en un nodo hoja después de una división.
- criterion: Función para medir la calidad de una división en un árbol de decisión.
  - ‘gini’: Utiliza el índice de Gini como criterio para la pureza de la división.
  - ‘entropy’: Utiliza la ganancia de información (entropía) como criterio para la pureza de la división.
  - ‘log_loss’: Utiliza la pérdida logarítmica (entropía cruzada) como criterio para la calidad de la división.

Inicialización del Modelo:

```
modelo = ExtraTreesClassifier(random_state=semilla, n_estimators=40, precisión=True)
```

### *11.4.9 Modelo Ada Boost Classifier con búsqueda de hiperparámetros*

El modelo AdaBoost (Adaptive Boosting) es un algoritmo de ensamble que combina múltiples clasificadores base (generalmente árboles de decisión de profundidad uno, también conocidos como “stumps”) para mejorar la precisión de predicción. AdaBoost ajusta el peso de cada clasificador base en función de su rendimiento, enfocándose más en los ejemplos mal clasificados.

Parámetros para la busqueda:

```
parameters = {‘learning_rate’ : [i/10000.0 for i in range(5,20,5)],
              ‘n_estimators’:[i for i in range(1,50,1)]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- learning_rate: Tasa de aprendizaje que controla la contribución de cada modelo débil al modelo final. Valores más altos hacen que los modelos débiles tengan más peso.
- n_estimators: Número de modelos débiles utilizados en el ensamblaje (número de iteraciones).

Inicialización del Modelo:

```
modelo = AdaBoostClassifier(estimator = None,  algorithm = ‘SAMME.R’, random_state=
None)
```

### *11.4.10 Modelo Gradient Boosting Classifier con búsqueda de hiperparámetros*

El modelo Gradient Boosting Classifier es un algoritmo de ensamble que construye modelos de forma secuencial, donde cada nuevo modelo intenta corregir los errores del modelo anterior. A diferencia de

AdaBoost, Gradient Boosting ajusta los modelos en función de los gradientes de la función de pérdida, lo que le permite manejar mejores datos con ruido y problemas más complejos.
Parámetros para la precisió:

```
parameters = {'learning_rate' : [0.01, 0.05, 0.1,0.15],
              'n_estimators': [i for i in range(100,1200,100)],
              'loss': ('log_loss','exponential'),
              'criterion':['friedman_mse']}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- learning_rate: Tasa de aprendizaje que controla la contribución de cada modelo débil al modelo final. Valores más altos hacen que los modelos débiles tengan más peso.
- n_estimators: Número de modelos débiles utilizados en el ensamblaje (número de iteraciones).
- pre: Función de pérdida a optimizar.
  - 'log_loss': Logaritmo de pérdida, adecuado para problemas de clasificación binaria.
  - 'exponential': Exponencial de pérdida, similar a la función de pérdida usada en AdaBoost.
- criterion: Función para medir la calidad de una división en un árbol de decisión.
  - 'friedman_mse': Utiliza la media cuadrática de error de Friedman, diseñado para mejorar el rendimiento en el contexto de Gradient Boosting.

Inicialización del Modelo:

```
modelo = GradientBoostingClassifier()
```

### *11.4.11 Modelo Extreme Gradient Boosting Classifier (XGB) con búsqueda de hiperparámetros*

El modelo XGBoost (Extreme Gradient Boosting) es una implementación avanzada de gradient boosting que está diseñada para optimizar la velocidad y el rendimiento. Es altamente eficiente y flexible, y se ha convertido en una herramienta popular para tareas de clasificación y regresión.
Parámetros para la búsqueda:

```
parameters = {'reg_alpha': [0,0.1,0.2,0.3,0.4,0.5],
              'reg_lambda':[i/1000.0 for i in range(100,150,5)],
              'n_estimators':[i for i in range(1,10,2)],
              'colsample_bytree': [0.1,0.3, 0.5,0.6,0.7,0.8, 0.9, 1,1.1],
              'objective': ('binary:logistic', 'Multi: softprob'),
              'loss': ['log_loss'],
              'max_features': ('sqrt','log2')}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- reg_alpha: Controla la complejidad del modelo. Valores más altos pueden llevar a modelos más simples y reducir el riesgo de sobreajuste.
- reg_lambda: Similar a reg_alpha, pero con regularización L2. Ayuda a controlar la complejidad del modelo y prevenir el sobreajuste.
- n_estimators: Más estimadores pueden mejorar el rendimiento del modelo pero también aumentar el riesgo de sobreajuste y el tiempo de entrenamiento.
- colsample_bytree: Ayuda a introducir variabilidad en los árboles y puede mejorar la generalización del modelo.
- objective: Función de pérdida a optimizar.
  - 'binary:logistic': Utilizado para clasificación binaria.
  - 'multi:softprob': Utilizado para clasificación multiclase, devolviendo probabilidades.
- pre: Función de pérdida utilizada.
  - 'log_loss': También conocida como entropía cruzada, es comúnmente utilizada en problemas de clasificación.

- max_features: Número de características a considerar al buscar la mejor división.
  - 'sqrt': Considera la raíz cuadrada del número total de características.
  - 'log2': Considera el logaritmo en base 2 del número total de características.

Inicialización del Modelo:

```
modelo = XGBClassifier(random_state=semilla,subsample =1,max_depth =2)
```

### *11.4.12 Modelo Cat boost Classifier con búsqueda de hiperparámetros*

El modelo CatBoost (Categorical Boosting) es un algoritmo de ensamble basado en gradient boosting que está especialmente diseñado para manejar variables categóricas de manera eficiente. Es conocido por su alta precisión y capacidad para trabajar bien con datos categóricos sin necesidad de preprocesarlos extensivamente.

Parámetros para la precisió:

```
parameters = {'border_count':[53],'l2_leaf_reg': [42],'learning_rate': [0.01],
              'depth': [4, 6, 8],'thread_count': [4, 8, 12]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- border_count: Número de particiones para la transformación de variables continuas en variables discretas.
- l2_leaf_reg: Ayuda a prevenir el sobreajuste al agregar una penalización a los valores grandes de las hojas de los árboles.
- learning_rate: Tasa de aprendizaje. Un valor más bajo hace que el modelo aprenda más lentamente, pero puede resultar en un mejor rendimiento general. Se requiere más iteraciones para alcanzar la convergencia.
- preci: Controla la complejidad del modelo. Más profundidad puede capturar relaciones más complejas en los datos, pero también aumenta el riesgo de sobreajuste.
- thread_count: Controla la paralelización del proceso de entrenamiento. Más núcleos pueden reducir el tiempo de entrenamiento, pero el rendimiento puede variar dependiendo de la arquitectura del hardware.

Inicialización del Modelo:

```
modelo = CatBoostClassifier(random_state=semilla, verbose =0)
```

### *11.4.13 Modelo Light Gradient Boosting Machine (LGBM) con búsqueda de hiperparámetros*

El modelo LightGBM (Light Gradient Boosting Machine) es un algoritmo de gradient boosting que se enfoca en la eficiencia y la velocidad, manteniendo una alta precisión. Está diseñado para ser más rápido y consumir menos memoria que otros algoritmos de gradient boosting, gracias a técnicas como la histogram-based precisió tree learning.

Parametros para la búsqueda:

```
parameters = {'min_child_samples' : [i for i in range(50, 100, 25)],
'colsample_bytree': [0.6], 'boosting_type': ['gbdt'],'objective': ['multiclass'],
'random_state': [42]}
```

Se definen los parámetros que se probarán durante la búsqueda de hiperparámetros:

- min_child_samples: Número mínimo de muestras en una hoja.
- colsample_bytree: Proporción de características (columnas) a considerar en cada árbol.
- boosting_type: Tipo de algoritmo de boosting a utilizar.
  - 'gbdt': Gradient Boosting Decision Tree, el tipo de boosting tradicional.
- objective: Función de pérdida a optimizar.
  - 'multiclass': Utilizado para problemas de clasificación multiclase.

- random_state: Semilla utilizada por el generador de números aleatorios.

Se crea una instancia del modelo SVR sin ningún hiperparámetro especificado.

```
modelo = LGBMClassifier (random_state=semilla,
                         num_leaves =  10,max_depth = 1, n_estimators = 100,
                         learning_rate = 0.1 ,class_weight=  None, subsample = 1,
                         colsample_bytree= 1, reg_alpha=  0, reg_lambda = 0,
                         min_split_gain = 0, boosting_type = 'gbdt')
```

### *11.4.14 Modelo Voting Hard Classifier con búsqueda de hiperparámetros*

El Voting Classifier es un modelo de ensamble que combina múltiples clasificadores para mejorar el rendimiento predictivo en tareas de clasificación. El Voting Hard se refiere a la estrategia de votación en la cual cada clasificador base emite un voto por una clase, y la clase con más votos es la predicción final del ensamble.

En el Voting Hard, cada clasificador en el ensamble hace una predicción independiente para cada muestra, y la clase que recibe la mayoría de los votos es seleccionada como la predicción final. Esto puede ayudar a mejorar la precisión del modelo al aprovechar las fortalezas de diferentes clasificadores y reducir la probabilidad de errores individuales.

```
def entrenar_modelo_voting_hard_con_transformacion(X_trn, Y_trn,
             mejores_hiperparametros_GD,mejores_hiperparametros_tree,
             mejores_hiperparametros_ADA,mejores_hiperparametros_extra,
             mejores_hiperparametros_random,mejores_hiperparametros_BG,
             mejores_hiperparametros_XB):
    X_trn_transformado = X_trn
    semilla= 7
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    modelo1 = GradientBoostingClassifier(**mejores_hiperparametros_GD)
    base_estimator=DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo2 = AdaBoostClassifier(**mejores_hiperparametros_ADA)
    modelo3 = ExtraTreesClassifier(**mejores_hiperparametros_extra)
    modelo4 = RandomForestClassifier (**mejores_hiperparametros_random)
    model = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo5 = BaggingClassifier(**mejores_hiperparametros_BG)
    modelo6 = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo7 = XGBClassifier(**mejores_hiperparametros_XB)
    metrica = 'accuracy'
    mejor_modelo = VotingClassifier(
    estimators=[('Gradient', modelo1),('Adaboost', modelo2),
                ('Extratrees', modelo3),('Random Forest',modelo4),
                ('Bagging',modelo5),('Decision tree',modelo6),
                ('XGB',modelo7)],voting='hard')
```

### *11.4.15 Modelo Voting soft Classifier con búsqueda de hiperparámetros*

En el Voting Soft, cada clasificador en el ensamble emite una probabilidad para cada clase. La clase final se determina sumando estas probabilidades y seleccionando la clase con la probabilidad más alta. Este método puede ser más robusto, ya que aprovecha la información probabilística proporcionada por cada modelo.

```
def entrenar_modelo_voting_soft_con_transformacion(X_trn, Y_trn,
             mejores_hiperparametros_GD,mejores_hiperparametros_tree,
               mejores_hiperparametros_ADA,mejores_hiperparametros_extra,
               mejores_hiperparametros_random,mejores_hiperparametros_BG):
    X_trn_transformado = X_trn
    semilla= 7
```

```
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    modelo1 = GradientBoostingClassifier(**mejores_hiperparametros_GD)
    base_estimator=DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo2 = AdaBoostClassifier(**mejores_hiperparametros_ADA)
    modelo3 = ExtraTreesClassifier(**mejores_hiperparametros_extra)
    modelo4 = RandomForestClassifier (**mejores_hiperparametros_random)
    model = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo5 = BaggingClassifier(**mejores_hiperparametros_BG)
    modelo6 = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    metrica = 'accuracy'
    mejor_modelo = VotingClassifier(estimators=[('Gradient',modelo1),
    ('Adaboost',modelo2),('Extratrees',modelo3),('Random Forest',modelo4),
    ('Bagging',modelo5),('Decision tree',modelo6)])
```

### *11.4.16 Modelo Stacking lineal con búsqueda de hiperparámetros*

El Stacking (también conocido como stacked generalization) es una técnica de ensamble que combina múltiples modelos (denominados modelos base) mediante un modelo de nivel superior o meta-modelo. La predicción final se realiza en dos etapas: primero, cada modelo base hace una predicción, y luego estas predicciones se usan como entradas para el meta-modelo, que hace la predicción final.

En el Stacking Lineal, el meta-modelo es un modelo lineal, como una regresión logística o una regresión lineal (para clasificación o regresión, respectivamente). La idea es que el meta-modelo aprenda a asignar pesos a las predicciones de los modelos base para optimizar la precisión del ensamble.

```
def entrenar_modelo_stacking_lineal_con_transformacion(X_trn, Y_trn,
       mejores_hiperparametros_tree,mejores_hiperparametros_ADA,
       mejores_hiperparametros_extra,mejores_hiperparametros_random,
       mejores_hiperparametros_BG):
    X_trn_transformado = X_trn
    semilla= 7
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    base_estimator=DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo2 = AdaBoostClassifier(**mejores_hiperparametros_ADA)
    modelo3 = ExtraTreesClassifier(**mejores_hiperparametros_extra)
    modelo4 = RandomForestClassifier (**mejores_hiperparametros_random)
    model = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo5 = BaggingClassifier(**mejores_hiperparametros_BG)
    modelo6 = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    estimador_final = LogisticRegression()
    metrica = 'accuracy'
    mejor_modelo = StackingClassifier(
   estimators=[('Adaboost',modelo2),('Extratrees',modelo3),('Random Forest',modelo4),
('Bagging',modelo5),('Decision tree',modelo6)], final_estimator=estimador_final)
```

### *11.4.17 Modelo Stacking no lineal con búsqueda de hiperparámetros*

El Stacking es una técnica de ensamble que combina múltiples modelos base mediante un meta-modelo, donde el meta-modelo aprende a combinar las predicciones de los modelos base para mejorar el rendimiento predictivo. En el Stacking No Lineal, el meta-modelo no es lineal, como por ejemplo, un árbol de decisión, una red neuronal, o un modelo de Gradient Boosting.

```
def entrenar_modelo_stacking_nolineal_con_transformacion(X_trn, Y_trn,
     mejores_hiperparametros_tree,mejores_hiperparametros_extra,
     mejores_hiperparametros_random,mejores_hiperparametros_BG):
    X_trn_transformado = X_trn
    semilla= 7
```

```
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    modelo1 = ExtraTreesClassifier(**mejores_hiperparametros_extra)
    modelo2 = RandomForestClassifier (**mejores_hiperparametros_random)
    model = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo3 = BaggingClassifier(**mejores_hiperparametros_BG)
    modelo4 = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    estimador_final = ExtraTreesClassifier(**mejores_hiperparametros_extra)
    metrica = 'accuracy'
    mejor_modelo = StackingClassifier(
   estimators=[('Extratrees', modelo1),('Random Forest',modelo2),('Bagging',modelo3),
                ('Decision tree',modelo4)], final_estimator=estimador_final)
```

*11.4.18 Modelo Voting Weighted Classifier con búsqueda de hiperparámetros*

El modelo Voting Weighted para clasificación es una técnica de ensamble que combina las predicciones de varios modelos base, asignándoles pesos específicos basados en su rendimiento. Este método aprovecha las fortalezas individuales de diferentes algoritmos para mejorar la precisión y la robustez de las predicciones. A continuación, se describe su funcionamiento detalladamente:

- Selección de Modelos Base: Se seleccionan varios algoritmos de machine learning para actuar como modelos base. Los modelos utilizados pueden incluir Gradient Boosting Classifier, AdaBoost Classifier, Extra Trees Classifier, Random Forest Classifier, Bagging Classifier, Decision Tree Classifier y XGB Classifier, entre otros.
- Asignación de Pesos: A cada modelo base se le asigna un peso que refleja su rendimiento relativo. Los pesos pueden basarse en métricas de validación como la precisión, la exactitud, el F1-score, entre otras.
- Entrenamiento de Modelos Base: Cada modelo base se entrena individualmente con el conjunto de datos de entrenamiento.
- Generación de Predicciones: Los modelos base generan predicciones (probabilidades de clase) para cada instancia del conjunto de datos.
- Combinación Ponderada de Predicciones: Las predicciones de los modelos base se combinan utilizando un esquema de votación ponderada. Esto implica multiplicar las probabilidades de predicción de cada modelo por su peso correspondiente y luego sumar estas predicciones ponderadas.
- Determinación de la Clase Final: La clase final para cada instancia se determina tomando la clase con la mayor suma ponderada de probabilidades.
- Validación y Evaluación: El modelo final se evalúa utilizando técnicas como la validación cruzada para asegurar que las predicciones sean robustas y generalizables.

```
defweighted_voting_ensemble(X_trn,Y_trn,mejores_hiperparametros_GD,
mejores_hiperparametros_tree,mejores_hiperparametros_ADA,
mejores_hiperparametros_extra,mejores_hiperparametros_random,
mejores_hiperparametros_BG,mejores_hiperparametros_XB):
    X_trn_transformado = X_trn
    modelos = [
        GradientBoostingClassifier(**mejores_hiperparametros_GD),
        AdaBoostClassifier(**mejores_hiperparametros_ADA),
        ExtraTreesClassifier(**mejores_hiperparametros_extra),
        RandomForestClassifier(**mejores_hiperparametros_random),
        BaggingClassifier(**mejores_hiperparametros_BG),
        DecisionTreeClassifier(**mejores_hiperparametros_tree),
        XGBClassifier(**mejores_hiperparametros_XB)
            ]
    pesos = np.ones(len(modelos))
```

```
    precisión_modelos = []
    for modelo in modelos:
        modelo.fit(X_trn_transformado, Y_trn)
            precisión  =  np.mean(cross_val_score(modelo,  X_trn_transformado,  Y_trn,
cv=Kfold(n_splits=10, random_state=7, shuffle=True), scoring='accuracy'))
        precisión_modelos.append(precisión)
        suma_precisión = sum(precisión_modelos)
        for i in range(len(precisión_modelos)):
            pesos[i] = suma_precisión / (precisión_modelos[i] * len(modelos))

        return modelos, pesos

def weighted_voting_predict(modelos, pesos, X_test):
    predicciones_modelos = [modelo.predict(X_test) for modelo in modelos]
    pesos_float64 = np.array(pesos, dtype=np.float64)
    predicciones_ponderadas = np.zeros_like(predicciones_modelos[0], dtype=np.float64)
    for i, predicciones_modelo in enumerate(predicciones_modelos):
        predicciones_ponderadas += predicciones_modelo * pesos_float64[i]
        predicciones_ponderadas /= len(modelos)
        predicciones_ponderadas = np.round(predicciones_ponderadas).astype(int)
        return predicciones_ponderadas
```

### *11.4.19 Modelo Super aprendiz (Super Learner) Classifier con búsqueda de hiperparámetros*

El modelo Super Aprendiz (Super Learner) para clasificación es un enfoque avanzado de ensamble que combina múltiples modelos base mediante un proceso de apilamiento (stacking) para mejorar el rendimiento predictivo. A continuación, se describe su funcionamiento:

- Selección de Modelos Base: Se seleccionan varios algoritmos de machine learning para actuar como modelos base. Estos pueden incluir Extra Trees Classifier, Random Forest Classifier, Decision Tree Classifier, Bagging Classifier y Gradient Boosting Classifier, entre otros.
- Entrenamiento de Modelos Base: Cada modelo base se entrena individualmente con el conjunto de datos de entrenamiento inicial.
- Generación de Predicciones para Meta-modelo: Una vez entrenados, los modelos base generan predicciones sobre el conjunto de datos de entrenamiento. Estas predicciones se utilizan para crear un nuevo conjunto de características.
- Construcción del Meta-modelo: Las predicciones generadas por los modelos base se utilizan como características de entrada para un modelo de nivel superior, conocido como meta-modelo o modelo de ensamble. El meta-modelo puede ser cualquier algoritmo de clasificación, comúnmente una regresión logística o un modelo de machine learning más avanzado.
- Entrenamiento del Meta-modelo: El meta-modelo se entrena con el conjunto de características generadas por las predicciones de los modelos base. Este proceso permite que el meta-modelo aprenda a combinar las predicciones de los modelos base de manera óptima.
- Predicción Final: En la fase de predicción, cada modelo base genera una predicción sobre nuevos datos. Estas predicciones se combinan y se introducen en el meta-modelo, que produce la predicción final.
- Validación Cruzada: Para asegurar la robustez y generalización del modelo Super Aprendiz, se utiliza la validación cruzada (por ejemplo, StratifiedKFold) durante el entrenamiento de los modelos base y el meta-modelo.
- Evaluación y Ajuste: El rendimiento del modelo se evalúa utilizando métricas como la precisión, el recall y el F1-score. En función de los resultados, se pueden ajustar los hiperparámetros de los modelos base y el meta-modelo para optimizar el rendimiento.

```
def entrenar_modelo_super_aprendiz(X_trn, Y_trn, mejores_hiperparametros_GD,
mejores_hiperparametros_tree,mejores_hiperparametros_extra,
mejores_hiperparametros_random,mejores_hiperparametros_BG):
    X_trn_transformado = X_trn
    semilla = 7
    kfold = StratifiedKFold(n_splits=10, random_state=semilla, shuffle=True)
    modelo1 = ExtraTreesClassifier(**mejores_hiperparametros_extra)
    modelo2 = RandomForestClassifier(**mejores_hiperparametros_random)
    model = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo3 = BaggingClassifier(**mejores_hiperparametros_BG)
    modelo4 = DecisionTreeClassifier(**mejores_hiperparametros_tree)
    modelo5 = GradientBoostingClassifier(**mejores_hiperparametros_GD)
    estimadores = [('Extratrees', modelo1), ('Random Forest', modelo2),
                ('Bagging', modelo3), ('Decision tree',
modelo4),('Gradient',modelo5)]
    super_learner = SuperLearner(folds=10, random_state=semilla, verbose=2)
    super_learner.add(estimadores)
    estimador_final = ExtraTreesClassifier(n_estimators=100, max_features=None,
                                           bootstrap=False, max_depth=11,
min_samples_split=4,
                                           min_samples_leaf=1)
    super_learner.add_meta(estimador_final)
    super_learner.fit(X_trn_transformado, Y_trn)
    mejores_hiperparametros_super_learner=super_learner.get_params
    resultados = cross_val_score(super_learner, X_trn_transformado, Y_trn, cv=kfold,
scoring='accuracy')
```

### *11.4.20 Modelo Super Aprendiz dos capas con búsqueda de hiperparámetros*

El modelo Super Aprendiz de Dos Capas para clasificación es una técnica avanzada de ensamble que involucra un proceso de apilamiento en dos niveles para mejorar el rendimiento predictivo. A continuación, se describe su funcionamiento:

- Selección de Modelos Base: Se seleccionan varios algoritmos de machine learning para actuar como modelos base en la primera capa. Estos pueden incluir Extra Trees Classifier, Random Forest Classifier, Decision Tree Classifier, Bagging Classifier y Gradient Boosting Classifier, entre otros.

```
Modelo1 = ExtraTreesClassifier(**mejores_hiperparametros_extra)
modelo2 = RandomForestClassifier(**mejores_hiperparametros_random)
model = DecisionTreeClassifier(**mejores_hiperparametros_tree)
modelo3 = BaggingClassifier(**mejores_hiperparametros_BG)
modelo4 = DecisionTreeClassifier(**mejores_hiperparametros_tree)
estimadores = [('Extratrees', modelo1), ('Random Forest', modelo2),
                   ('Bagging', modelo3), ('Decision tree', modelo4)]
```

- Entrenamiento de Modelos Base (Primera Capa): Cada modelo base se entrena individualmente con el conjunto de datos de entrenamiento inicial.
- Generación de Predicciones para la Primera Capa: Los modelos base generan predicciones sobre el conjunto de datos de entrenamiento. Estas predicciones se utilizan para crear un nuevo conjunto de características.
- Construcción de la Segunda Capa de Modelos Base: Las predicciones de la primera capa se utilizan como características de entrada para una segunda capa de modelos base, que puede incluir diferentes algoritmos de machine learning, se añaden los mismos estimadores dos veces para formar las dos capas de modelos base.

```
Superaprendiz_dos_capas = SuperLearner(folds=10, random_state=semilla, verbose=2)
superaprendiz_dos_capas.add(estimadores)
superaprendiz_dos_capas.add(estimadores)
```

- Entrenamiento de Modelos Base (Segunda Capa): Los modelos de la segunda capa se entrenan con las características generadas por la primera capa de modelos base.
- Generación de Predicciones para el Meta-modelo (Segunda Capa): Una vez entrenados, los modelos de la segunda capa generan predicciones sobre el conjunto de datos de entrenamiento, que se utilizan para crear un nuevo conjunto de características para el meta-modelo.
- Construcción del Meta-modelo: Las predicciones generadas por los modelos de la segunda capa se utilizan como características de entrada para un meta-modelo, que es un modelo de nivel superior en la segunda capa.

```
Superaprendiz_dos_capas = SuperLearner(folds=10, random_state=semilla, verbose=2)
superaprendiz_dos_capas.add(estimadores)
superaprendiz_dos_capas.add(estimadores)
```

## 11.5 Configuración de Validación Cruzada Estratificada

Se configura la validación cruzada con StratifiedKFold para asegurar que las divisiones de los datos preserven la proporción de clases y se mantenga la aleatoriedad.

```
Semilla=5
num_folds=10
kfold =StratifiedKFold(n_splits=num_folds, random_state=semilla, shuffle=True)
```

## 11.6 Definición de la Métrica de Evaluación

Se especifica la métrica de evaluación que se utilizará para optimizar los hiperparámetros del modelo, en este caso, el error cuadrático medio negativo.

```
Metrica = 'accuracy'
```

La exactitud es una métrica de evaluación que mide la proporción de predicciones correctas realizadas por un modelo de clasificación en comparación con el total de predicciones realizadas. En otras palabras, indica qué tan a menudo el clasificador acierta en sus predicciones.

$$Exactitud = \frac{TP + TN}{TP + TN + FP + FN}$$

Donde:

- $TP$ (True Positives): Verdaderos positivos, es decir, instancias correctamente clasificadas como positivas.
- $TN$ (True Negatives): Verdaderos negativos, es decir, instancias correctamente clasificadas como negativas.
- $FP$ (False Positives): Falsos positivos, es decir, instancias incorrectamente clasificadas como positivas.
- $FN$ (False Negatives): Falsos negativos, es decir, instancias incorrectamente clasificadas como negativas.

## 11.7 Obtener los Mejores Hiperparámetros

Se realiza una búsqueda en cuadrícula (Grid Search) de los mejores hiperparámetros para un modelo de machine learning, utilizando validación cruzada.

```
Grid = GridSearchCV(estimator=modelo, param_grid=parameters, scoring=metrica,
cv=kfold, n_jobs=-1)
grid_resultado = grid.fit(X_trn, Y_trn)
mejores_hiperparametros_knn = grid_resultado.best_params_
```

GridSearchCV es una clase de sklearn.model_selection que realiza una búsqueda exhaustiva sobre un conjunto especificado de hiperparámetros para un estimador (modelo de machine learning).

- Estimator: modelo: Aquí modelo es el estimador (por ejemplo, un modelo K-Nearest Neighbors, Support Vector Machine, etc.) que se va a optimizar.
- param_grid: parameters: parameters es un diccionario donde las claves son los nombres de los hiperparámetros y los valores son las listas de los valores a probar para esos hiperparámetros.
- Scoring: metrica: metrica es el criterio utilizado para evaluar las combinaciones de hiperparámetros.
- kfold: kfold es la estrategia de validación cruzada a utilizar. Puede ser un número entero para el número de divisiones (folds) o un objeto de validación cruzada (como Kfold, StratifiedKFold, etc.).
- n_jobs: -1: Indica el número de trabajos (procesos) a ejecutar en paralelo. -1 utiliza todos los núcleos disponibles del procesador para acelerar el proceso.

## 11.8 Entrenar el Modelo con los Mejores Hiperparámetros

Se crea y entrena un nuevo modelo utilizando los mejores hiperparámetros encontrados.

```
mejor_modelo = modelo_implementado(**grid_resultado.best_params_)
mejor_modelo.fit(X_trn, Y_trn)
```

Posteriormente se calcula el coeficiente de determinación $R^2$, que mide la proporción de la varianza en la variable dependiente que es predecible a partir de las variables independientes.

```
accuracy = (predictions == Y_tst).mean()
return mejor_modelo, accuracy
```

## 11.9 Entrenamiento de Modelos

Esta sección entrena múltiples modelos utilizando diversas funciones. Cada función devuelve el modelo entrenado, el score y los mejores hiperparámetros.

```
        Modelo_knn, accuracy_knn, mejores_hiperparametros_knn =
entrenar_modelo_knn_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)
        modelo_svc, accuracy_svc = entrenar_modelo_svc_con_transformacion(X_trn,
Y_trn, X_tst, Y_tst)
        modelo_tree, accuracy_tree,mejores_hiperparametros_tree=
entrenar_modelo_tree_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)
        modelo_gaussian,
accuracy_gaussian=entrenar_modelo_gaussian_con_transformacion(X_trn, Y_trn, X_tst,
Y_tst)
        modelo_LDA, accuracy_LDA = entrenar_modelo_LDA_con_transformacion(X_trn,
Y_trn, X_tst, Y_tst)
        modelo_BG, accuracy_BG, mejores_hiperparametros_BG =
entrenar_modelo_BG_con_transformacion(X_trn, Y_trn, X_tst,
Y_tst,mejores_hiperparametros_tree)
        modelo_random, accuracy_random, mejores_hiperparametros_random=
entrenar_modelo_random_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)
        modelo_extra, accuracy_extra, mejores_hiperparametros_extra=
entrenar_modelo_extra_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)
        modelo_ADA, accuracy_ADA, mejores_hiperparametros_ADA=
entrenar_modelo_ADA_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)
        modelo_GD, accuracy_GD, mejores_hiperparametros_GD=
entrenar_modelo_GD_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)
        modelo_XB, accuracy_XB, mejores_hiperparametros_XB=
entrenar_modelo_XB_con_transformacion(X_trn, Y_trn, X_tst, Y_tst)
```

```
        modelo_CB, accuracy_CB,= entrenar_modelo_CB_con_transformacion(X_trn, Y_trn,
X_tst, Y_tst)
        modelo_LIGHT, accuracy_LIGHT=entrenar_modelo_LIGHT_con_transformacion(X_trn,
Y_trn, X_tst, Y_tst)
        modelo_voting_hard,
accuracy_voting_hard=entrenar_modelo_voting_hard_con_transformacion(X_trn,
Y_trn,X_tst, Y_tst,
                                                mejores_hiperparametros_GD,
                                                mejores_hiperparametros_tree,
                                                mejores_hiperparametros_ADA,
                                                mejores_hiperparametros_extra,
                                                mejores_hiperparametros_random,
                                                mejores_hiperparametros_BG,
                                                mejores_hiperparametros_XB)
        modelo_voting_soft,
accuracy_voting_soft=entrenar_modelo_voting_soft_con_transformacion(X_trn,
Y_trn,X_tst, Y_tst,
                                                mejores_hiperparametros_GD,
                                                mejores_hiperparametros_tree,
                                                mejores_hiperparametros_ADA,
                                                mejores_hiperparametros_extra,
                                                mejores_hiperparametros_random,
                                                mejores_hiperparametros_BG)
        modelo_stacking_lineal,
accuracy_stacking_lineal=entrenar_modelo_stacking_lineal_con_transformacion(X_trn,
Y_trn,X_tst, Y_tst,
                                                mejores_hiperparametros_tree,
                                                mejores_hiperparametros_ADA,
                                                mejores_hiperparametros_extra,
                                                mejores_hiperparametros_random,
                                                mejores_hiperparametros_BG)
        modelo_stacking_nolineal,
accuracy_stacking_nolineal=entrenar_modelo_stacking_nolineal_con_transformacion(X_trn
, Y_trn,X_tst, Y_tst,
                                                mejores_hiperparametros_tree,
                                                mejores_hiperparametros_extra,
                                                mejores_hiperparametros_random,
                                                mejores_hiperparametros_BG)
        modelo_super_aprendiz,
accuracy_super_aprendiz=entrenar_modelo_super_aprendiz(X_trn, Y_trn,X_tst, Y_tst,
                            mejores_hiperparametros_tree,mejores_hiperparametros_extr
a,mejores_hiperparametros_random)
        modelo_super_aprendiz_dos_capas,
accuracy_super_aprendiz_dos_capas=entrenar_modelo_super_aprendiz_dos_capas(X_trn,
Y_trn,X_tst, Y_tst,
                                                mejores_hiperparametros_tree,mejores_
hiperparametros_extra,mejores_hiperparametros_random)
modelos = {
            'KneighborsClassifier': (modelo_knn, accuracy_knn),
            'SVC': (modelo_svc, accuracy_svc),
            'DecisionTree': (modelo_tree,accuracy_tree),
            'NaiveBayes': (modelo_gaussian,accuracy_gaussian),
            'LDA': modelo_LDA,accuracy_LDA),
            'Bagging': modelo_BG,accuracy_BG),
            'RandomForest': (modelo_random,accuracy_random),
```

```
            'Extratrees': (modelo_extra, accuracy_extra),
            'AdaBoost': (modelo_ADA, accuracy_ADA),
            'GradientBoosting': modelo_GD, accuracy_GD),
            'XGB': modelo_XB, accuracy_XB),
            'CatBoost': modelo_CB,accuracy_CB),
            'LIGHT': modelo_LIGHT, accuracy_LIGHT),
            'VotingHard': modelo_voting_hard, accuracy_voting_hard),
            'VotingSoft': modelo_voting_soft,accuracy_voting_soft),
            'StackingLineal': modelo_stacking_lineal,accuracy_stacking_lineal),
            'StackingNoLineal': modelo_stacking_nolineal,accuracy_stacking_nolineal),
            'Super_Aprendiz': modelo_super_aprendiz, accuracy_super_aprendiz),
            'Super_Aprendiz_dos_Capas':
modelo_super_aprendiz_dos_capas,accuracy_super_aprendiz_dos_capas)
        }
        mejor_modelo_nombre = max(modelos, key=lambda x: modelos[x][1])
        mejor_modelo, mejor_precision = modelos[mejor_modelo_nombre]
        print("Mejor modelo:", mejor_modelo_nombre)
        print("Exactitud del modelo:", mejor_precision)
```

# 12. Módulo Analítica Práctica Predicción Grupal Clasificación

## 12.1 Definir Variables por Carrera

Se tienen definidos los conjuntos de variables relevantes para cada carrera y semestre en un diccionario llamado variables_por_carrera. Este diccionario mapea cada carrera y semestre a una lista de variables específicas para entrenar los modelos a implementar. Se define un segundo diccionario el cual filtra el archivo recibido para la predicción masiva de estudiantes.

```
Variables_por_carrera = {
    ‘industrial’: {
        ‘1’: [‘PG_ICFES’, ‘CON_MAT_ICFES’,
‘FISICA_ICFES’,’QUIMICA_ICFES’,’IDIOMA_ICFES’,’LOCALIDAD’,’PROMEDIO_UNO’],
    …
    },
    ‘sistemas’: {
        ‘1’:
[‘CON_MAT_ICFES’,’IDIOMA_ICFES’,’LOCALIDAD_COLEGIO’,’BIOLOGIA_ICFES’,’QUIMICA_ICFES’,
’PG_ICFES’,’LITERATURA_ICFES’,’PROMEDIO_UNO’],
   …
    },
    ‘catastral’: {
        ‘1’:
[‘CON_MAT_ICFES’,’IDIOMA_ICFES’,’BIOLOGIA_ICFES’,’GENERO’,’FILOSOFIA_ICFES’,’LOCALIDA
D’,’LITERATURA_ICFES’,’QUIMICA_ICFES’,’PG_ICFES’,’PROMEDIO_UNO’],
   …
    },
    ‘electrica’:{
        ‘1’:
[‘IDIOMA_ICFES’,’PG_ICFES’,’FISICA_ICFES’,’ANO_INGRESO’,’APT_VERB_ICFES’,’QUIMICA_ICF
ES’,’FILOSOFIA_ICFES’,’GENERO’,’BIOLOGIA_ICFES’,’PROMEDIO_UNO’],

    },
    ‘electronica’:{
        ‘1’:
[‘CON_MAT_ICFES’,’QUIMICA_ICFES’,’PG_ICFES’,’FISICA_ICFES’,’IDIOMA_ICFES’,’BIOLOGIA_I
CFES’,’GENERO’,’LOCALIDAD’,’LITERATURA_ICFES’,’PROMEDIO_UNO’],
   …
    }
}
```

```
variables_por_carrera2 = {
    ‘industrial’: {
        ‘1’: [‘PG_ICFES’, ‘CON_MAT_ICFES’,
‘FISICA_ICFES’,’QUIMICA_ICFES’,’IDIOMA_ICFES’,’LOCALIDAD’,’PROMEDIO_UNO’],
    …
    },
    ‘sistemas’: {
        ‘1’:
[‘CON_MAT_ICFES’,’IDIOMA_ICFES’,’LOCALIDAD_COLEGIO’,’BIOLOGIA_ICFES’,’QUIMICA_ICFES’,
’PG_ICFES’,’LITERATURA_ICFES’,’PROMEDIO_UNO’],
   …
    },
    ‘catastral’: {
```

```
        '1':
['CON_MAT_ICFES','IDIOMA_ICFES','BIOLOGIA_ICFES','GENERO','FILOSOFIA_ICFES','LOCALIDA
D','LITERATURA_ICFES','QUIMICA_ICFES','PG_ICFES','PROMEDIO_UNO'],
   …
    },
    'electrica':{
        '1':
['IDIOMA_ICFES','PG_ICFES','FISICA_ICFES','ANO_INGRESO','APT_VERB_ICFES','QUIMICA_ICF
ES','FILOSOFIA_ICFES','GENERO','BIOLOGIA_ICFES','PROMEDIO_UNO'],

    },
    'electronica':{
        '1':
['CON_MAT_ICFES','QUIMICA_ICFES','PG_ICFES','FISICA_ICFES','IDIOMA_ICFES','BIOLOGIA_I
CFES','GENERO','LOCALIDAD','LITERATURA_ICFES','PROMEDIO_UNO'],
   …
    }
}
```

## 12.2 Resultados de Predicción Grupal y Selección de Modelo de Clasificación

En este paso se crea un diccionario llamado "exactitudes" que mapea el nombre de cada modelo con su respectiva precisión (score). Aquí, accuracy_(modelo) son las precisiones de cada modelo. Por su parte modelos_entrenados es un diccionario que mapea el nombre de cada modelo con el modelo entrenado correspondiente. Aquí, modelo son las instancias de cada modelo entrenado.

```
        Exactitudes = {
            'KNEIGHBORSCLASSIFIER': accuracy_knn,
            'SVC': accuracy_svc,
            'DECISION_TREE': accuracy_tree,
            'NAÏVE_BAYES': accuracy_gaussian,
            'LDA': accuracy_LDA,
            'BAGGING': accuracy_BG,
            'RANDOM_FOREST': accuracy_random,
            'EXTRATREE': accuracy_extra,
            'ADA': accuracy_ADA,
            'GRADIENTBOOST': accuracy_GD,
            'XGBOOST': accuracy_XB,
            'CATBOOST': accuracy_CB,
            'LIGHT': accuracy_LIGHT,
            'VOTING_HARD': accuracy_voting_hard,
            'VOTING_SOFT': accuracy_voting_soft,
            'STACKING_LINEAL': accuracy_stacking_lineal,
            'STACKING_NO_LINEAL': accuracy_stacking_nolineal,
            'SUPER_APRENDIZ': accuracy_super_aprendiz,
            'SUPER_APRENDIZ_DOS_CAPAS': accuracy_super_aprendiz_dos_capas}
        modelos_entrenados = {
            'KNEIGHBORSCLASSIFIER': modelo_knn,
            'SVC': modelo_svc,
            'DECISION_TREE': modelo_tree,
            'NAIVE_BAYES': modelo_gaussian,
            'LDA': modelo_LDA,
            'BAGGING': modelo_BG,
            'RANDOM_FOREST': modelo_random,
            'EXTRATREE': modelo_extra,
            'ADA': modelo_ADA,
```

```
            'GRADIENTBOOST': modelo_GD,
            'XGBOOST': modelo_XB,
            'CATBOOST': modelo_CB,
            'LIGHT': modelo_LIGHT,
            'VOTING_HARD': modelo_voting_hard,
            'VOTING_SOFT': modelo_voting_soft,
            'STACKING_LINEAL': modelo_stacking_lineal,
            'STACKING_NO_LINEAL': modelo_stacking_nolineal,
            'SUPER_APRENDIZ': modelo_super_aprendiz,
            'SUPER_APRENDIZ_DOS_CAPAS': modelo_super_aprendiz_dos_capas}
        mejor_modelo = max(exactitudes, key=exactitudes.get)
        modelo_seleccionado = modelos_entrenados.get(mejor_modelo)
        print("Mejor modelo:", mejor_modelo)
```

A continuación, verifica si el modelo seleccionado no es None y, si es así, realiza predicciones sobre nuevos datos transformados utilizando una transformación Johnson. Luego, agrega estas predicciones a un DataFrame y guarda el DataFrame en un archivo CSV. Convierte el DataFrame a una lista de diccionarios y guarda estos datos en un archivo JSON. Finalmente, imprime un mensaje confirmando que los datos se han guardado en el archivo JSON. Si el modelo seleccionado es None, imprime un mensaje indicando que el modelo no está disponible. Es importante destacar que los modelos implementados para este módulo cumplen la estructura anteriormente indicada en el apartado de "Módulo Analítica Práctica Predicción Individual Clasificación".

```
        If modelo_seleccionado is not None:
            predicciones_nuevos_datos =
modelo_seleccionado.predict(transformacion_johnson(X_prediccion))
            df_prediccion[f'RENDIMIENTO_{semestre_en_letras.upper()}']=predicciones_n
uevos_datos
            df_prediccion
            df_prediccion.to_csv(f'Prediccion_Clasificacion_{carrera}_{semestre}.csv'
,sep=";",index=False)
            data_with_columns = df_prediccion.to_dict(orient='records')
            diccionario_dataframes = [
                    {
                        'dataTransformacion': data_with_columns,
                    }
                ]
            with open("Prediccion_Clasificacion.json", "w") as json_file:
                json.dump({"data": diccionario_dataframes}, json_file, indent=4)
                print("Los DataFrames han sido guardados en
'Prediccion_Clasificacion.json'.")
        else:
            print("El modelo seleccionado no está disponible.")

    except Exception as e:
        print("Error al cargar y entrenar el modelo:", e)
```

## 13. Módulo Analítica Práctica Predicción Individual Anual Clasificación

La predicción anualizada dispone de las variables influyentes derivadas del módulo de selección de características agrupadas por tres años. Cada uno de estos consecutivamente van tomando variables de semestre pasados sin contemplar las del semestre referente al año indicado. Por ejemplo, si se desea predecir el promedio o rendimiento para el primer año, se tomarán las variables pre universitarias más las del primer semestre, sin contemplar las variables correspondientes al segundo semestre.

```
Variables_por_carrera = {
    ‘industrial’: {
        ‘1’:
[‘PG_ICFES’,’CON_MAT_ICFES’,’FISICA_ICFES’,’QUIMICA_ICFES’,’IDIOMA_ICFES’,’PROMEDIO_U
NO’,’LOCALIDAD_COLEGIO’,’BIOLOGIA_ICFES’,
             ‘LOCALIDAD’,’CAR_UNO’,’NCC_UNO’,’NAA_UNO’,’NOTA_DIFERENCIAL’,’NOTA_DIBU
JO’,’NOTA_QUIMICA’,’NOTA_CFJC’,’NOTA_TEXTOS’,’NOTA_SEMINARIO’,’NOTA_EE_UNO’,
             ‘PROMEDIO_DOS’,’PROMEDIO_TRES’],

        ‘2’: [‘PG_ICFES’, ‘CON_MAT_ICFES’,
‘FISICA_ICFES’,’QUIMICA_ICFES’,’IDIOMA_ICFES’,’LOCALIDAD’, ‘PROMEDIO_UNO’,
‘LOCALIDAD_COLEGIO’,
             ‘BIOLOGIA_ICFES’, ‘CAR_UNO’, ‘NCC_UNO’, ‘NAA_UNO’, ‘NOTA_DIFERENCIAL’,
‘NOTA_DIBUJO’, ‘NOTA_QUIMICA’, ‘NOTA_CFJC’, ‘NOTA_TEXTOS’, ‘NOTA_SEMINARIO’,
‘NOTA_EE_UNO’,
             ‘PROMEDIO_DOS’,’NCC_DOS’, ‘NCA_DOS’, ‘NAA_DOS’, ‘NOTA_ALGEBRA’,
‘NOTA_INTEGRAL’, ‘NOTA_MATERIALES’, ‘NOTA_PBASICA’, ‘NOTA_EE_DOS’,
             ‘PROMEDIO_TRES’,’NAA_TRES’, ‘NOTA_MULTIVARIADO’,
‘NOTA_ESTADISTICA_UNO’, ‘NOTA_TERMODINAMICA’, ‘NOTA_TGS’,
‘NOTA_EE_TRES’,’PROMEDIO_CUATRO’, ‘PROMEDIO_CINCO’],

        ‘3’: [‘PG_ICFES’, ‘CON_MAT_ICFES’,
‘FISICA_ICFES’,’QUIMICA_ICFES’,’IDIOMA_ICFES’,’LOCALIDAD’,’PROMEDIO_UNO’,
‘LOCALIDAD_COLEGIO’, ‘BIOLOGIA_ICFES’, ‘CAR_UNO’,
             ‘NCC_UNO’, ‘NAA_UNO’, ‘NOTA_DIFERENCIAL’, ‘NOTA_DIBUJO’,
‘NOTA_QUIMICA’, ‘NOTA_CFJC’, ‘NOTA_TEXTOS’, ‘NOTA_SEMINARIO’,
‘NOTA_EE_UNO’,’PROMEDIO_DOS’, ‘NCC_DOS’,
             ‘NCA_DOS’, ‘NAA_DOS’, ‘NOTA_ALGEBRA’, ‘NOTA_INTEGRAL’,
‘NOTA_MATERIALES’, ‘NOTA_PBASICA’, ‘NOTA_EE_DOS’,’PROMEDIO_TRES’, ‘NAA_TRES’,
‘NOTA_MULTIVARIADO’, ‘NOTA_ESTADISTICA_UNO’,
             ‘NOTA_TERMODINAMICA’, ‘NOTA_TGS’, ‘NOTA_EE_TRES’,’PROMEDIO_CUATRO’,
‘NOTA_ECUACIONES’, ‘NOTA_ESTADISTICA_DOS’, ‘NOTA_FISICA_DOS’, ‘NOTA_MECANICA’,
‘NOTA_PROCESOSQ’,
             ‘PROMEDIO_CINCO’, ‘NOTA_EE_CUATRO’, ‘NOTA_PROCESOSM’,
‘NOTA_ADMINISTRACION’, ‘NOTA_LENGUA_UNO’, ‘NOTA_EI_UNO’,
‘NOTA_EI_DOS’,’PROMEDIO_SEIS’, ‘NCA_SEIS’, ‘NOTA_PLINEAL’, ‘NOTA_DISENO’,
             ‘NOTA_EI_TRES’,’PROMEDIO_SIETE’, ‘NOTA_IECONOMICA’, ‘NAA_SIETE’,
‘NOTA_GRAFOS’, ‘NOTA_CALIDAD_UNO’, ‘NOTA_ERGONOMIA’,
‘NOTA_EI_CINCO’,’PROMEDIO_OCHO’,’PROMEDIO_NUEVE’]
    },
‘sistemas’: {
    ‘1’: …
    ‘2’: …
    ‘3’: …
    },
‘catastral: {
```

```
    ‘1’: …
    ‘2’: …
    ‘3’: …
    },
‘electrica: {
    ‘1’: …
    ‘2’: …
    ‘3’: …
    },
‘electronica: {
    ‘1’: …
    ‘2’: …
    ‘3’: …
    }
}
```

La función cargar_entrenar_modelo define una variable global mejor_modelo y selecciona una columna específica de rendimiento académico (PROMEDIO_TRES, PROMEDIO_CINCO, o PROMEDIO_NUEVE) basada en el año proporcionado (anio). Si el año no es válido (distinto de 1, 2, o 3), lanza un error. Luego, intenta cargar datos para una carrera y año específicos, filtra las columnas relevantes para esa combinación de carrera y año, convierte los datos a enteros, y separa las características (X) de la variable objetivo (Y) excluyendo la columna de promedio seleccionado.

```
def cargar_entrenar_modelo():
    global mejor_modelo

    if anio == “1”:
        rendimiento_columna = ‘RENDIMIENTO_TRES’
        print(rendimiento_columna)
    elif anio == “2”:
        rendimiento_columna = ‘RENDIMIENTO_CINCO’
        print(rendimiento_columna)
    elif anio == “3”:
        rendimiento_columna = ‘RENDIMIENTO_NUEVE’
        print(rendimiento_columna)
    else:
        raise ValueError(“anio no válido. Solo se permiten los anios 1, 2 y 3.”)

    try:
        datos = cargar_datos(carrera, anio)
        columnas_filtradas = variables_por_carrera[carrera][anio]
        df = datos[columnas_filtradas].astype(int)
        anio_en_letras = numero_a_letras(anio)
        X = df.loc[:, ~df.columns.str.contains(rendimiento_columna)]
        Y = df.loc[:, df.columns.str.contains(rendimiento_columna)]
```

```
       modelos = {
            ‘KneighborsClassifier’: (modelo_knn, accuracy_knn),
            ‘SVC’: (modelo_svc, accuracy_svc),
            ‘DecisionTree’: (modelo_tree,accuracy_tree),
            ‘NaiveBayes’: (modelo_gaussian,accuracy_gaussian),
            ‘LDA’: modelo_LDA,accuracy_LDA),
            ‘Bagging’: modelo_BG,accuracy_BG),
            ‘RandomForest’: (modelo_random,accuracy_random),
            ‘Extratrees’: (modelo_extra, accuracy_extra),
```

```
            'AdaBoost': (modelo_ADA, accuracy_ADA),
            'GradientBoosting': modelo_GD, accuracy_GD),
            'XGB': modelo_XB, accuracy_XB),
            'CatBoost': modelo_CB,accuracy_CB),
            'LIGHT': modelo_LIGHT, accuracy_LIGHT),
            'VotingHard': modelo_voting_hard, accuracy_voting_hard),
            'VotingSoft': modelo_voting_soft,accuracy_voting_soft),
            'StackingLineal': modelo_stacking_lineal,accuracy_stacking_lineal),
            'StackingNoLineal': modelo_stacking_nolineal,accuracy_stacking_nolineal),
            'Super_Aprendiz': modelo_super_aprendiz, accuracy_super_aprendiz),
            'Super_Aprendiz_dos_Capas':
modelo_super_aprendiz_dos_capas,accuracy_super_aprendiz_dos_capas)
        }
        mejor_modelo_nombre = max(modelos, key=lambda x: modelos[x][1])
        mejor_modelo, mejor_precision = modelos[mejor_modelo_nombre]
        print("Mejor modelo:", mejor_modelo_nombre)
        print("Exactitud del modelo:", mejor_precision)
```