3. Machine Learning

Aprendizaje Automático

In [ ]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from varname import nameof
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import PolynomialFeatures, StandardScaler, MinMaxScaler
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.model_selection import train_test_split
```

In [ ]:
```
df = pd.read_csv('./data/Student_performance_data.csv')
df
```

Métodos de transformación y escalado

La preparación de datos para algoritmos de machine learning es un paso importante para asegurar la fiabilidad de los modelos y aumentar su calidad.

Para llevar a cabo la transformación y escalado existen diversos métodos y es fundamental comprender cuándo y como utilizarlos. Debido a la naturaleza de los datos de estudiados, el metodo a usar será el StandardScaler.

- Escala los datos para que tengan una media de 0 y una desviación estandar de 1.
- Se utiliza para centrar y escalar los valores
- Se usa cuando los datos se distribuyen normalmente
- Para algoritmos sensibles a la escala de los datos, como regresión lineal, regresión logística, SVM, y redes neuronales

Escalar Datos

In [ ]:
```
# Identificamos outliers en las variables numéricas y los eliminamos
variables_numericas = ['Age', 'StudyTimeWeekly', 'Absences']
df_numericas = df[variables_numericas]

def identificar_outliers(df, col_categorica, col_cuantitativa):
  outliers = pd.DataFrame()

  for categoria in df[col_categorica].unique():
    data_categoria = df[df[col_categorica] == categoria][col_cuantitativa]
    Q1 = data_categoria.quantile(0.25)
    Q3 = data_categoria.quantile(0.75)
    IQR = Q3 - Q1
    limite_inferior = Q1 - 1.5 * IQR
    limite_superior = Q3 + 1.5 * IQR
    outliers_categoria = data_categoria[(data_categoria < limite_inferior) | (data_categoria > limite_superior)
    outliers = pd.concat([outliers, outliers_categoria])
  return outliers

outliers = identificar_outliers(df, 'GPA', 'Absences')
print(f"Outliers identificados:\n{outliers}")

indices_outliers = outliers.index
df = df.drop(indices_outliers)
```

In [ ]:
```
# Uso del StandardScaler en las columnas numericas

scaler_standard = StandardScaler()
df_standard = pd.DataFrame(scaler_standard.fit_transform(df), columns=df.columns)
df_standard
```

In [ ]:
```
# Uso del MinMaxScaler en las columnas
scaler_minmax = MinMaxScaler()
df_minmax = pd.DataFrame(scaler_minmax.fit_transform(df), columns=df.columns)

df_minmax
```

In [ ]:
```
# Funcion para comparar modelos en un dataframe
```

```
def compare_models(model_dict):
    results = []

    for name, (mse, r2, mae, rmse, pred) in model_dict.items():
        stats = [name, mse, r2, mae, rmse]
        results.append(stats)
    df_results = pd.DataFrame(results, columns=['Modelo', 'MSE', 'R2', 'MAE', 'RMSE'])
    return df_results
```

Modelos de regresión Escalar Datos

Regresion Lineal

In [ ]:
```
# Función de regresión lineal
def regresion_lineal_category(X_train, X_test, y_train, y_test, x_n, y_n, df_n):

    dir = str(f'./graph/1_regresion_lineal/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    # Crear un modelo de regresión lineal
    model = LinearRegression()

    # Ajustar el modelo a los datos
    model.fit(X_train, y_train)

    # Hacer predicciones para todos los valores de X
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)

    # Calcular métricas
    mse = mean_squared_error(y_test, y_pred_test)
    r2 = r2_score(y_test, y_pred_test)
    mae = mean_absolute_error(y_test, y_pred_test)
    rmse = mean_squared_error(y_test, y_pred_test, squared=False)  # squared=False returns the RMSE

    # Generar gráfico
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))  # Crear una figura con dos subgráficos

    # Añadir jitter a los datos binomiales
    jitter_strength = 0.02
    y_train_jitter = y_train + np.random.uniform(-jitter_strength, jitter_strength, y_train.shape)
    y_test_jitter = y_test + np.random.uniform(-jitter_strength, jitter_strength, y_test.shape)

    # Primer subgráfico: Datos de train
    ax[0].scatter(X_train, y_train_jitter, label='Datos de train', alpha=0.5)
    ax[0].plot(X_train, y_pred_train, color='red', label='Recta de regresión')
    ax[0].set_title(f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n} - train. DF: {df_n}')
    ax[0].set_xlabel(f'{x_n}')
    ax[0].set_ylabel(f'{y_n}')
    ax[0].legend()

    # Segundo subgráfico: Datos de test
    ax[1].scatter(X_train, y_train_jitter, label='Datos de train', alpha=0.5)
    ax[1].plot(X_train, y_pred_train, color='red', label='Recta de regresión')
    ax[1].set_title(f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n} - test. DF: {df_n}')
    ax[1].set_xlabel(f'{x_n}')
    ax[1].set_ylabel(f'{y_n}')
    ax[1].legend()

    plt.tight_layout()
    file = str(f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    #plt.show()

    return mse, r2, mae, rmse, y_pred_test
```

In [ ]:
```
# Función de regresion lineal binomial

def regresion_lineal_binomial(X_train, X_test, y_train, y_test, x_n, y_n, df_n):
    dir = str(f'./graph/1_regresion_lineal/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    # Crear un modelo de regresión lineal
    model = LinearRegression()

    # Ajustar el modelo a los datos
    model.fit(X_train, y_train)

    # Hacer predicciones para todos los valores de X
    y_pred_train = model.predict(X_train)
```

```
    y_pred_test = model.predict(X_test)

    # Calcular métricas
    mse = mean_squared_error(y_test, y_pred_test)
    r2 = r2_score(y_test, y_pred_test)
    mae = mean_absolute_error(y_test, y_pred_test)
    rmse = mean_squared_error(y_test, y_pred_test, squared=False)  # squared=False returns the RMSE

    # Normalizar los datos de y si el rango es muy pequeño
    if np.max(y_train) - np.min(y_train) < 0.1:
        scaler = MinMaxScaler()
        y_train = scaler.fit_transform(y_train.reshape(-1, 1)).flatten()
        y_test = scaler.transform(y_test.reshape(-1, 1)).flatten()
        y_pred_train = scaler.transform(y_pred_train.reshape(-1, 1)).flatten()
        y_pred_test = scaler.transform(y_pred_test.reshape(-1, 1)).flatten()
        y_n = f'Normalizado {y_n}'

    # Generar gráfico
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))  # Crear una figura con dos subgráficos

    # Crear función para gráficos de área apilada
    def stacked_area_plot(ax, X, y, y_pred, title):
        # Asegurarse de que X y y sean arrays unidimensionales
        X = X.flatten()
        y = y.flatten()
        # Clasificar los datos según X
        sorted_indices = np.argsort(X)
        X_sorted = X[sorted_indices]
        y_sorted = y[sorted_indices]
        y_pred_sorted = y_pred[sorted_indices]

        # Ajustar tamaño de la ventana para datos con poco rango
        window_size = min(50, len(y_sorted) // 2)  # Tamaño de ventana adaptativo
        if len(y_sorted) <= window_size:
            window_size = len(y_sorted) - 1  # Asegúrate de que la ventana no sea más grande que el número de m

        proportions_0 = [np.mean(y_sorted[i:i+window_size] <= 0.5) for i in range(len(y_sorted) - window_size +
        proportions_1 = [np.mean(y_sorted[i:i+window_size] > 0.5) for i in range(len(y_sorted) - window_size + 1
        X_windows = X_sorted[:len(proportions_0)]

        # Graficar las proporciones
        ax.fill_between(X_windows.flatten(), proportions_0, label=f'<= 0.5 {y_n}', alpha=0.5)
        ax.fill_between(X_windows.flatten(), proportions_1, label=f'> 0.5 {y_n}', alpha=0.5)

        # Graficar la recta de regresión
        ax.plot(X_sorted, y_pred_sorted, color='red', label='Recta de regresión')
        ax.set_title(title)
        ax.set_xlabel(f'{x_n}')
        ax.set_ylabel(f'{y_n}')
        ax.legend()

    # Primer subgráfico: Datos de train
    stacked_area_plot(ax[0], X_train, y_train, y_pred_train, f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n}

    # Segundo subgráfico: Datos de test
    stacked_area_plot(ax[1], X_test, y_test, y_pred_test, f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n} - T

    plt.tight_layout()
    file = str(f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    #plt.show()

    return mse, r2, mae, rmse, y_pred_test
```

In [ ]:

```
# Función de regresion lineal politomica

def regresion_lineal_politomica(X_train, X_test, y_train, y_test, x_n, y_n, df_n):

    dir = str(f'./graph/1_regresion_lineal/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    # Crear un modelo de regresión lineal
    model = LinearRegression()

    # Ajustar el modelo a los datos
    model.fit(X_train, y_train)

    # Hacer predicciones para todos los valores de X
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)

    # Calcular métricas
    mse = mean_squared_error(y_test, y_pred_test)
```

```
    r2 = r2_score(y_test, y_pred_test)
    mae = mean_absolute_error(y_test, y_pred_test)
    rmse = mean_squared_error(y_test, y_pred_test, squared=False)  # squared=False returns the RMSE

    # Preparar datos para gráficos
    train_data = np.concatenate([X_train, y_train.reshape(-1, 1)], axis=1)
    test_data = np.concatenate([X_test, y_test.reshape(-1, 1)], axis=1)

    # Generar gráfico
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))  # Crear una figura con dos subgráficos

    def boxplot_with_regression(ax, data, title):
        X = data[:, 0].reshape(-1, 1)
        y = data[:, 1]

        # Graficar el gráfico de cajas
        sns.boxplot(x=y, y=X.flatten(), ax=ax, palette='Set2')

        # Determinar los límites del gráfico de cajas
        box_limits = ax.get_xlim()

        # Crear nuevos puntos de X para la línea de regresión desde el primer hasta el último cuadro
        X_new = np.linspace(box_limits[0], box_limits[1], 100).reshape(-1, 1)
        y_pred_new = model.predict(X_new)

        # Graficar la línea de regresión
        ax.plot(X_new, y_pred_new, color='red', linewidth=2, label='Línea de Regresión')

        # Configuración del gráfico
        ax.set_title(title)
        ax.set_xlabel(f'{x_n}')
        ax.set_ylabel(f'{y_n}')
        ax.legend(title='Leyenda')
        ax.grid(True)

    # Primer subgráfico: Datos de train
    boxplot_with_regression(ax[0], train_data, f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n} - train. DF: 

    # Segundo subgráfico: Datos de test
    boxplot_with_regression(ax[1], test_data, f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n} - test. DF: {d

    plt.tight_layout()
    file = str(f'1. REGRESIÓN LINEAL Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    #plt.show()

    return mse, r2, mae, rmse, y_pred_test
```

In [ ]:
```
def regresion_lineal(X_train, X_test, y_train, y_test, x_n, y_n, df_n):
    if y_n == "Tutoring" or y_n == "Extracurricular" or y_n == "Sports" or y_n == "Music":
        return regresion_lineal_binomial(X_train, X_test, y_train, y_test, x_n, y_n, df_n)
    elif y_n == "ParentalSupport":
        return regresion_lineal_politomica(X_train, X_test, y_train, y_test, x_n, y_n, df_n)
    else:
        return regresion_lineal_category(X_train, X_test, y_train, y_test, x_n, y_n, df_n)
```

Regresión Polinómica

In [ ]:
```
# Función de regresion polinomica category

def regresion_polinomica_category(X_train, X_test, y_train, y_test, x_n, y_n, df_n):
    # Crear el directorio para guardar los gráficos
    dir = str(f'./graph/2_regresion_polinomica/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    # Crear características polinómicas
    poly = PolynomialFeatures(degree=15)  # Cambiar degree para ajustar la complejidad del modelo

    # Crear el modelo de regresión polinómica
    x_poly_train = poly.fit_transform(X_train)
    x_poly_test = poly.transform(X_test)  # Cambiar fit_transform a transform para datos de test

    model = LinearRegression()

    # Ajustar el modelo a los datos
    model.fit(x_poly_train, y_train)

    # Hacer predicciones
    y_poly_pred_train = model.predict(x_poly_train)
    y_poly_pred_test = model.predict(x_poly_test)

    # Evaluar el modelo
```

```
    mse_poly = mean_squared_error(y_test, y_poly_pred_test)
    r2_poly = r2_score(y_test, y_poly_pred_test)
    mae_poly = mean_absolute_error(y_test, y_poly_pred_test)
    rmse_poly = mean_squared_error(y_test, y_poly_pred_test, squared=False)  # squared=False returns the RMSE

    # Obtener el valor mínimo y máximo de x para el rango de valores
    x_min = X_train.min()
    x_max = X_train.max()

    # Crear un rango de valores para x para dibujar la curva de regresión
    x_range = np.linspace(x_min, x_max, 100).reshape(-1, 1)
    x_range_poly = poly.transform(x_range)
    y_range_pred = model.predict(x_range_poly)

    # Añadir jitter a los datos
    jitter_strength = 0.02
    y_train_jitter = y_train + np.random.uniform(-jitter_strength, jitter_strength, y_train.shape)
    y_test_jitter = y_test + np.random.uniform(-jitter_strength, jitter_strength, y_test.shape)

    # Generar gráfico
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))  # Crear una figura con dos subgráficos

    # Primer subgráfico: Datos de train
    ax[0].scatter(X_train, y_train_jitter, label='Datos de train', alpha=0.5)
    ax[0].plot(x_range, y_range_pred, color='red', label='Curva de regresión')
    ax[0].set_title(f'2. REGRESIÓN POLINÓMICA Relación entre {x_n} y {y_n} - train. DF: {df_n}')
    ax[0].set_xlabel(f'{x_n}')
    ax[0].set_ylabel(f'{y_n}')
    ax[0].legend()

    # Segundo subgráfico: Datos de test
    ax[1].scatter(X_test, y_test_jitter, color='green', label='Datos de test', alpha=0.5)
    ax[1].plot(x_range, y_range_pred, color='red', label='Curva de regresión')
    ax[1].set_title(f'2. REGRESIÓN POLINÓMICA Relación entre {x_n} y {y_n} - test. DF: {df_n}')
    ax[1].set_xlabel(f'{x_n}')
    ax[1].set_ylabel(f'{y_n}')
    ax[1].legend()

    plt.tight_layout()

    # Guardar el gráfico en la carpeta correspondiente
    file = str(f'2. REGRESIÓN POLINÓMICA Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    #plt.show()

    return mse_poly, r2_poly, mae_poly, rmse_poly, y_poly_pred_test
```

In [ ]:

```
# Función de regresion polinomica binomial

def regresion_polinomica_binomial(X_train, X_test, y_train, y_test, x_n, y_n, df_n, poly_degree=15):
    # Crear el directorio para guardar los gráficos
    dir = str(f'./graph/2_regresion_polinomica/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    # Crear características polinómicas
    poly = PolynomialFeatures(degree=poly_degree)  # Cambiar degree para ajustar la complejidad del modelo

    # Transformar los datos
    x_poly_train = poly.fit_transform(X_train)
    x_poly_test = poly.transform(X_test)  # Cambiar fit_transform a transform para datos de test

    model = LinearRegression()

    # Ajustar el modelo a los datos
    model.fit(x_poly_train, y_train)

    # Hacer predicciones
    y_poly_pred_train = model.predict(x_poly_train)
    y_poly_pred_test = model.predict(x_poly_test)

    # Evaluar el modelo
    mse_poly = mean_squared_error(y_test, y_poly_pred_test)
    r2_poly = r2_score(y_test, y_poly_pred_test)
    mae_poly = mean_absolute_error(y_test, y_poly_pred_test)
    rmse_poly = mean_squared_error(y_test, y_poly_pred_test, squared=False)  # squared=False returns the RMSE

    # Obtener el valor mínimo y máximo de x para el rango de valores
    x_min = X_train.min()
    x_max = X_train.max()

    # Crear un rango de valores para x para dibujar la curva de regresión
    x_range = np.linspace(x_min, x_max, 100).reshape(-1, 1)
    x_range_poly = poly.transform(x_range)
```

```
    y_range_pred = model.predict(x_range_poly)

    # Añadir jitter a los datos
    jitter_strength = 0.02
    y_train_jitter = y_train + np.random.uniform(-jitter_strength, jitter_strength, y_train.shape)
    y_test_jitter = y_test + np.random.uniform(-jitter_strength, jitter_strength, y_test.shape)

    # Generar gráfico
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))  # Crear una figura con dos subgráficos

    # Crear función para gráficos de área apilada
    def stacked_area_plot(ax, X, y, y_pred, title):
        # Asegurarse de que X y y sean arrays unidimensionales
        X = X.flatten()
        y = y.flatten()
        # Clasificar los datos según X
        sorted_indices = np.argsort(X)
        X_sorted = X[sorted_indices]
        y_sorted = y[sorted_indices]
        y_pred_sorted = y_pred[sorted_indices]

        # Ajustar tamaño de la ventana para datos con poco rango
        window_size = min(50, len(y_sorted) // 2)  # Tamaño de ventana adaptativo
        if len(y_sorted) <= window_size:
            window_size = len(y_sorted) - 1  # Asegúrate de que la ventana no sea más grande que el número de m

        proportions_0 = [np.mean(y_sorted[i:i+window_size] <= 0.5) for i in range(len(y_sorted) - window_size +
        proportions_1 = [np.mean(y_sorted[i:i+window_size] > 0.5) for i in range(len(y_sorted) - window_size + 1
        X_windows = X_sorted[:len(proportions_0)]

        # Graficar las proporciones
        ax.fill_between(X_windows.flatten(), proportions_0, label=f'<= 0.5 {y_n}', alpha=0.5)
        ax.fill_between(X_windows.flatten(), proportions_1, label=f'> 0.5 {y_n}', alpha=0.5)

        # Graficar la curva polinómica de regresión
        ax.plot(X_sorted, y_pred_sorted, color='red', label='Curva polinómica de regresión')
        ax.set_title(title)
        ax.set_xlabel(f'{x_n}')
        ax.set_ylabel(f'{y_n}')
        ax.legend()

    # Primer subgráfico: Datos de train
    stacked_area_plot(ax[0], X_train, y_train_jitter, y_poly_pred_train, f'2. REGRESIÓN POLINÓMICA Relación ent

    # Segundo subgráfico: Datos de test
    stacked_area_plot(ax[1], X_test, y_test_jitter, y_poly_pred_test, f'2. REGRESIÓN POLINÓMICA Relación entre

    plt.tight_layout()

    # Guardar el gráfico en la carpeta correspondiente
    file = str(f'2. REGRESIÓN POLINÓMICA Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    #plt.show()

    return mse_poly, r2_poly, mae_poly, rmse_poly, y_poly_pred_test
```

In [ ]:

```
# Función de regresion polinomica politomica

def regresion_polinomica_politomica(X_train, X_test, y_train, y_test, x_n, y_n, df_n, poly_degree=15):
    # Crear el directorio para guardar los gráficos
    dir = str(f'./graph/2_regresion_polinomica/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    # Crear características polinómicas
    poly = PolynomialFeatures(degree=poly_degree)  # Cambiar degree para ajustar la complejidad del modelo

    # Transformar los datos
    x_poly_train = poly.fit_transform(X_train)
    x_poly_test = poly.transform(X_test)  # Cambiar fit_transform a transform para datos de test

    model = LinearRegression()

    # Ajustar el modelo a los datos
    model.fit(x_poly_train, y_train)

    # Hacer predicciones
    y_poly_pred_train = model.predict(x_poly_train)
    y_poly_pred_test = model.predict(x_poly_test)

    # Evaluar el modelo
    mse_poly = mean_squared_error(y_test, y_poly_pred_test)
    r2_poly = r2_score(y_test, y_poly_pred_test)
    mae_poly = mean_absolute_error(y_test, y_poly_pred_test)
```

```
    rmse_poly = mean_squared_error(y_test, y_poly_pred_test, squared=False)  # squared=False returns the RMSE

    # Preparar datos para gráficos
    train_data = np.concatenate([X_train, y_train.reshape(-1, 1)], axis=1)
    test_data = np.concatenate([X_test, y_test.reshape(-1, 1)], axis=1)

    # Obtener los valores mínimos y máximos de x para el rango de valores
    x_min = X_train.min()
    x_max = X_train.max()

    # Generar gráfico
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))  # Crear una figura con dos subgráficos

    def boxplot_with_regression(ax, data, title):
        X = data[:, 0].reshape(-1, 1)
        y = data[:, 1]

        # Graficar el gráfico de cajas
        sns.boxplot(x=y, y=X.flatten(), ax=ax, palette='Set2')

        # Determinar los límites del gráfico de cajas
        box_limits = ax.get_xlim()

        # Ajustar los límites de la línea de regresión para comenzar y terminar en las cajas
        x_start = max(x_min, box_limits[0])
        x_end = min(x_max, box_limits[1])

        # Crear nuevos puntos de X para la línea de regresión desde el borde de la primera caja hasta el borde
        X_new = np.linspace(x_start, x_end, 100).reshape(-1, 1)
        y_pred_new = model.predict(poly.transform(X_new))

        # Graficar la curva de regresión polinómica
        ax.plot(X_new, y_pred_new, color='red', linewidth=2, label='Curva Polinómica de Regresión')

        # Configuración del gráfico
        ax.set_title(title)
        ax.set_xlabel(f'{x_n}')
        ax.set_ylabel(f'{y_n}')
        ax.legend(title='Leyenda')
        ax.grid(True)

    # Primer subgráfico: Datos de train
    boxplot_with_regression(ax[0], train_data, f'2. REGRESIÓN POLINÓMICA Relación entre {x_n} y {y_n} - train. I

    # Segundo subgráfico: Datos de test
    boxplot_with_regression(ax[1], test_data, f'2. REGRESIÓN POLINÓMICA Relación entre {x_n} y {y_n} - test. DF

    plt.tight_layout()
    file = str(f'2. REGRESIÓN POLINÓMICA Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    #plt.show()

    return mse_poly, r2_poly, mae_poly, rmse_poly, y_poly_pred_test
```

In [ ]:
```
def regresion_polinomica(X_train, X_test, y_train, y_test, x_n, y_n, df_n):
    if y_n == "Tutoring" or y_n == "Extracurricular" or y_n == "Sports" or y_n == "Music":
        return regresion_polinomica_binomial(X_train, X_test, y_train, y_test, x_n, y_n, df_n)
    elif y_n == "ParentalSupport":
        return regresion_polinomica_politomica(X_train, X_test, y_train, y_test, x_n, y_n, df_n)
    else:
        return regresion_polinomica_category(X_train, X_test, y_train, y_test, x_n, y_n, df_n)
```

Red Neuronal

In [ ]:
```
# Funcion de regresion neuronal category

def gen_graph_neuron(x_train, y_train, x_test, y_test, y_pred_train, y_pred_test, x_n, y_n, model, df_n):

    # Crear el directorio para guardar los gráficos
    dir = str(f'./graph/3_regresion_neuronal/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    plt.figure(figsize=(20, 6))
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))

    x_min = min(x_train.min(), x_test.min())
    x_max = max(x_train.max(), x_test.max())
    x_range = np.linspace(x_min, x_max, 100).reshape(-1, 1)

    x_range_norm = (x_range - np.mean(x_train)) / np.std(x_train)
    y_range_pred_norm = model.predict(x_range_norm)
    y_range_pred = y_range_pred_norm * np.std(y_train) + np.mean(y_train)
```

```
    # Añadir jitter a los datos
    jitter_strength = 0.02
    y_train_jitter = y_train + np.random.uniform(-jitter_strength, jitter_strength, y_train.shape)
    y_test_jitter = y_test + np.random.uniform(-jitter_strength, jitter_strength, y_test.shape)

    ax[0].scatter(x_train, y_train_jitter, label='Datos de entrenamiento', alpha=0.5)
    ax[0].plot(x_range, y_range_pred, color='red', label='Curva de regresión')
    ax[0].set_title(f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} - train. DF: {df_n}')
    ax[0].set_xlabel(x_n)
    ax[0].set_ylabel(y_n)
    ax[0].legend()

    ax[1].scatter(x_test, y_test_jitter, color='green', label='Datos de prueba', alpha=0.5)
    ax[1].plot(x_range, y_range_pred, color='red', label='Curva de regresión')
    ax[1].set_title(f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} - test. DF: {df_n}')
    ax[1].set_xlabel(x_n)
    ax[1].set_ylabel(y_n)
    ax[1].legend()

    file = str(f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    plt.tight_layout()
    #plt.show()

def regresion_neuronal_category(x, y, x_n, y_n, df_n):
    x = np.array(x).reshape(-1, 1)
    y = np.array(y).reshape(-1, 1)

    x_mean = np.mean(x)
    x_std = np.std(x)
    y_mean = np.mean(y)
    y_std = np.std(y)

    x_norm = (x - x_mean) / x_std
    y_norm = (y - y_mean) / y_std

    x_train, x_test, y_train, y_test = train_test_split(x_norm, y_norm, test_size=0.2, random_state=42)

    model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, input_shape=(x_train.shape[1],), activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1)
    ])

    model.compile(optimizer='adam', loss='mean_squared_error')

    history = model.fit(x_train, y_train, epochs=100, validation_split=0.2, verbose=0)

    y_pred_train_norm = model.predict(x_train)
    y_pred_test_norm = model.predict(x_test)
    y_pred_all_norm = model.predict(x_norm)

    y_pred_train = y_pred_train_norm * y_std + y_mean
    y_pred_test = y_pred_test_norm * y_std + y_mean
    y_pred_all = y_pred_all_norm * y_std + y_mean

    x_train_orig = x_train * x_std + x_mean
    x_test_orig = x_test * x_std + x_mean
    x_all_orig = x_norm * x_std + x_mean

    y_train_orig = y_train * y_std + y_mean
    y_test_orig = y_test * y_std + y_mean
    y_all_orig = y_norm * y_std + y_mean

    mse = mean_squared_error(y_all_orig, y_pred_all)
    r2 = r2_score(y_all_orig, y_pred_all)
    mae = mean_absolute_error(y_all_orig, y_pred_all)
    rmse = mean_squared_error(y_all_orig, y_pred_all, squared=False)

    gen_graph_neuron(x_train_orig, y_train_orig, x_test_orig, y_test_orig, y_pred_train, y_pred_test, x_n, y_n,
    return mse, r2, mae, rmse, y_pred_all
```

In [ ]:

```
# Función de regresion neuronal binomial

def stacked_area_plot(ax, X, y, y_pred, title, x_n, y_n):
    # Asegurarse de que X y y sean arrays unidimensionales
    X = X.flatten()
    y = y.flatten()
    # Clasificar los datos según X
    sorted_indices = np.argsort(X)
    X_sorted = X[sorted_indices]
    y_sorted = y[sorted_indices]
```

```
    y_pred_sorted = y_pred[sorted_indices]

    # Ajustar tamaño de la ventana para datos con poco rango
    window_size = min(50, len(y_sorted) // 2)  # Tamaño de ventana adaptativo
    if len(y_sorted) <= window_size:
        window_size = len(y_sorted) - 1  # Asegúrate de que la ventana no sea más grande que el número de muest

    proportions_0 = [np.mean(y_sorted[i:i+window_size] <= 0.5) for i in range(len(y_sorted) - window_size + 1)]
    proportions_1 = [np.mean(y_sorted[i:i+window_size] > 0.5) for i in range(len(y_sorted) - window_size + 1)]
    X_windows = X_sorted[:len(proportions_0)]

    # Graficar las proporciones
    ax.fill_between(X_windows.flatten(), proportions_0, label=f'<= 0.5 {y_n}', alpha=0.5)
    ax.fill_between(X_windows.flatten(), proportions_1, label=f'> 0.5 {y_n}', alpha=0.5)

    # Graficar la curva polinómica de regresión
    ax.plot(X_sorted, y_pred_sorted, color='red', label='Curva de regresión neuronal')
    ax.set_title(title)
    ax.set_xlabel(x_n)
    ax.set_ylabel(y_n)
    ax.legend()

def gen_graph_neuron_binomial(x_train, y_train, x_test, y_test, y_pred_train, y_pred_test, x_n, y_n, model, df_
    # Crear el directorio para guardar los gráficos
    dir = str(f'./graph/3_regresion_neuronal/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    # Obtener el rango de x para la curva de regresión
    x_min = min(x_train.min(), x_test.min())
    x_max = max(x_train.max(), x_test.max())
    x_range = np.linspace(x_min, x_max, 100).reshape(-1, 1)

    x_range_norm = (x_range - np.mean(x_train)) / np.std(x_train)
    y_range_pred_norm = model.predict(x_range_norm)
    y_range_pred = y_range_pred_norm * np.std(y_train) + np.mean(y_train)

    # Añadir jitter a los datos
    jitter_strength = 0.02
    y_train_jitter = y_train + np.random.uniform(-jitter_strength, jitter_strength, y_train.shape)
    y_test_jitter = y_test + np.random.uniform(-jitter_strength, jitter_strength, y_test.shape)

    # Crear una figura con dos subgráficos
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))

    # Primer subgráfico: Datos de entrenamiento
    stacked_area_plot(ax[0], x_train, y_train_jitter, y_pred_train,
                      f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} - train. DF: {df_n}',
                      x_n, y_n)

    # Segundo subgráfico: Datos de prueba
    stacked_area_plot(ax[1], x_test, y_test_jitter, y_pred_test,
                      f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} - test. DF: {df_n}',
                      x_n, y_n)

    plt.tight_layout()
    file = str(f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    #plt.show()

def regresion_neuronal_binomial(x, y, x_n, y_n, df_n):
    # Convertir a arrays y hacer reshape
    x = np.array(x).reshape(-1, 1)
    y = np.array(y).reshape(-1, 1)

    # Dividir los datos en entrenamiento y prueba
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

    # Normalizar los datos
    x_mean = np.mean(x_train)
    x_std = np.std(x_train)
    y_mean = np.mean(y_train)
    y_std = np.std(y_train)

    x_train_norm = (x_train - x_mean) / x_std
    x_test_norm = (x_test - x_mean) / x_std

    # Definir y entrenar el modelo de red neuronal
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, input_shape=(x_train_norm.shape[1],), activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1)
    ])
```

```
    model.compile(optimizer='adam', loss='mean_squared_error')

    history = model.fit(x_train_norm, y_train, epochs=100, validation_split=0.2, verbose=0)

    # Hacer predicciones
    y_pred_train_norm = model.predict(x_train_norm)
    y_pred_test_norm = model.predict(x_test_norm)
    y_pred_all_norm = model.predict((np.concatenate([x_train, x_test]) - x_mean) / x_std)

    # Desnormalizar las predicciones
    y_pred_train = y_pred_train_norm * y_std + y_mean
    y_pred_test = y_pred_test_norm * y_std + y_mean
    y_pred_all = y_pred_all_norm * y_std + y_mean

    # Desnormalizar los datos
    x_train_orig = x_train
    x_test_orig = x_test
    x_all_orig = np.concatenate([x_train, x_test])

    y_train_orig = y_train
    y_test_orig = y_test
    y_all_orig = np.concatenate([y_train, y_test])

    # Evaluar el modelo
    mse = mean_squared_error(y_all_orig, y_pred_all)
    r2 = r2_score(y_all_orig, y_pred_all)
    mae = mean_absolute_error(y_all_orig, y_pred_all)
    rmse = mean_squared_error(y_all_orig, y_pred_all, squared=False)

    # Generar gráficos
    gen_graph_neuron_binomial(x_train_orig, y_train_orig, x_test_orig, y_test_orig, y_pred_train, y_pred_test, x

    return mse, r2, mae, rmse, y_pred_test
```

In [ ]:

```
# Función de regresion neuronal politomica

def regresion_neuronal_politomica(x, y, x_n, y_n, df_n, poly_degree=15, hidden_layer_sizes=(100,), test_size=0.2
    # Asegúrate de que x e y sean arrays de NumPy
    x = np.asarray(x).reshape(-1, 1) if x.ndim == 1 else np.asarray(x)
    y = np.asarray(y).reshape(-1, 1) if y.ndim == 1 else np.asarray(y)

    # Dividir los datos en conjunto de entrenamiento y prueba
    X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=test_size, random_state=0)

    # Crear el directorio para guardar los gráficos
    dir = str(f'./graph/3_regresion_neuronal/{df_n}/')
    os.makedirs(dir, exist_ok=True)

    # Crear características polinómicas
    poly = PolynomialFeatures(degree=poly_degree)

    # Transformar los datos
    x_poly_train = poly.fit_transform(X_train)
    x_poly_test = poly.transform(X_test)

    # Crear el modelo de red neuronal
    model = MLPRegressor(hidden_layer_sizes=hidden_layer_sizes, max_iter=1000, random_state=0)

    # Ajustar el modelo a los datos
    model.fit(x_poly_train, y_train.ravel())  # .ravel() convierte a una 1D array

    # Hacer predicciones
    y_poly_pred_train = model.predict(x_poly_train)
    y_poly_pred_test = model.predict(x_poly_test)

    # Evaluar el modelo
    mse_poly = mean_squared_error(y_test, y_poly_pred_test)
    r2_poly = r2_score(y_test, y_poly_pred_test)
    mae_poly = mean_absolute_error(y_test, y_poly_pred_test)
    rmse_poly = mean_squared_error(y_test, y_poly_pred_test, squared=False)  # squared=False returns the RMSE

    # Preparar datos para gráficos
    train_data = np.concatenate([X_train, y_train], axis=1)
    test_data = np.concatenate([X_test, y_test], axis=1)

    # Obtener los valores mínimos y máximos de x para el rango de valores
    x_min = X_train.min()
    x_max = X_train.max()

    # Generar gráfico
    fig, ax = plt.subplots(1, 2, figsize=(20, 6))  # Crear una figura con dos subgráficos

    def boxplot_with_regression(ax, data, title):
```

```
        X = data[:, 0].reshape(-1, 1)
        y = data[:, 1]

        # Graficar el gráfico de cajas
        sns.boxplot(x=y, y=X.flatten(), ax=ax, palette='Set2')

        # Determinar los límites del gráfico de cajas
        box_limits = ax.get_xlim()

        # Ajustar los límites de la línea de regresión para comenzar y terminar en las cajas
        x_start = max(x_min, box_limits[0])
        x_end = min(x_max, box_limits[1])

        # Crear nuevos puntos de X para la línea de regresión desde el borde de la primera caja hasta el borde
        X_new = np.linspace(x_start, x_end, 100).reshape(-1, 1)
        X_poly_new = poly.transform(X_new)
        y_pred_new = model.predict(X_poly_new)

        # Graficar la curva de regresión neuronal
        ax.plot(X_new, y_pred_new, color='red', linewidth=2, label='Curva Neuronal de Regresión')

        # Configuración del gráfico
        ax.set_title(title)
        ax.set_xlabel(f'{x_n}')
        ax.set_ylabel(f'{y_n}')
        ax.legend(title='Leyenda')
        ax.grid(True)

    # Primer subgráfico: Datos de train
    boxplot_with_regression(ax[0], train_data, f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} - train. DF

    # Segundo subgráfico: Datos de test
    boxplot_with_regression(ax[1], test_data, f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} - test. DF:

    plt.tight_layout()
    file = str(f'3. REGRESIÓN NEURONAL Relación entre {x_n} y {y_n} DF: {df_n}.png')
    plt.savefig(dir + file)
    #plt.show()

    return mse_poly, r2_poly, mae_poly, rmse_poly, y_poly_pred_test
```

In [ ]:
```
def regresion_neuronal(x, y, x_n, y_n, df_n):
    if y_n == "Tutoring" or y_n == "Extracurricular" or y_n == "Sports" or y_n == "Music":
        return regresion_neuronal_binomial(x, y, x_n, y_n, df_n)
    elif y_n == "ParentalSupport":
        return regresion_neuronal_politomica(x, y, x_n, y_n, df_n)
    else:
        return regresion_neuronal_category(x, y, x_n, y_n, df_n)
```

Programa principal

In [ ]:
```
x_n = 'GPA'
variables = ["StudyTimeWeekly", "Absences", "ParentalSupport", "Tutoring", "Extracurricular", "Sports", "Music"

results_dict = {}

for y_n in variables:
    x = pd.Series(df[x_n])
    y = pd.Series(df[y_n])

    # Dividir los datos en conjuntos de entrenamiento y prueba
    X_train, X_test, y_train, y_test = train_test_split(x.values.reshape(-1, 1), y.values.reshape(-1, 1), test_

    models = {'Regresión Lineal': regresion_lineal(X_train, X_test, y_train, y_test, x_n, y_n, nameof(df)),
    'Regresión Polinomica': regresion_polinomica(X_train, X_test, y_train, y_test, x_n, y_n, nameof(df)),
    'Regresión Neuronal': regresion_neuronal(x, y, x_n, str(y_n), nameof(df)),
    }
    comparison_df = compare_models(models)

    x = pd.Series(df_standard[x_n])
    y = pd.Series(df_standard[y_n])

    # Dividir los datos en conjuntos de entrenamiento y prueba
    X_train, X_test, y_train, y_test = train_test_split(x.values.reshape(-1, 1), y.values.reshape(-1, 1), test_

    models = {'Regresión Lineal': regresion_lineal(X_train, X_test, y_train, y_test, x_n, y_n, nameof(df_standa
    'Regresión Polinomica': regresion_polinomica(X_train, X_test, y_train, y_test, x_n, y_n, nameof(df_standard
    'Regresión Neuronal': regresion_neuronal(x, y, x_n, str(y_n), nameof(df_standard)),
    }
    comparison_df_standard = compare_models(models)
```

```
    x = pd.Series(df_minmax[x_n])
    y = pd.Series(df_minmax[y_n])

    # Dividir los datos en conjuntos de entrenamiento y prueba
    X_train, X_test, y_train, y_test = train_test_split(x.values.reshape(-1, 1), y.values.reshape(-1, 1), test_

    models = {'Regresión Lineal': regresion_lineal(X_train, X_test, y_train, y_test, x_n, y_n, nameof(df_minmax
    'Regresión Polinomica': regresion_polinomica(X_train, X_test, y_train, y_test, x_n, y_n, nameof(df_minmax))
    'Regresión Neuronal': regresion_neuronal(x, y, x_n, str(y_n), nameof(df_minmax)),
    }
    comparison_df_minmax = compare_models(models)

    results_dict[f'df_{y_n}'] = {
        'comparison_df': comparison_df,
        'comparison_df_standard': comparison_df_standard,
        'comparison_df_minmax': comparison_df_minmax
    }
```

Importar resultados

In [ ]:
```
# Exporta los resultados a un archivo CSV
# Abre un archivo en modo de escritura CSV
with open('resultados.csv', 'w', newline='') as file:
    # Itera sobre todas las entradas en el diccionario
    for variable_name, results in results_dict.items():
        # Escribe los resultados sin normalizar
        file.write(f"Resultados para {variable_name} (sin normalizar):\n")
        results['comparison_df'].to_csv(file, index=False)
        file.write("\n\n")

        # Escribe los resultados estandarizados
        file.write(f"Resultados para {variable_name} (estandarizado):\n")
        results['comparison_df_standard'].to_csv(file, index=False)
        file.write("\n\n")

        # Escribe los resultados normalizados Min-Max
        file.write(f"Resultados para {variable_name} (normalizado Min-Max):\n")
        results['comparison_df_minmax'].to_csv(file, index=False)
        file.write("\n\n" + "="*80 + "\n\n")

print("Los resultados se han exportado a 'resultados.csv'")
```

In [ ]:
```
# Exporta los resultados a un archivo de texto
# Abre un archivo en modo de escritura
with open('resultados.txt', 'w') as file:
    # Itera sobre todas las entradas en el diccionario
    for variable_name, results in results_dict.items():
        # Escribe los resultados en el archivo
        file.write(f"Resultados para {variable_name} (sin normalizar):\n")
        file.write(results['comparison_df'].to_string())
        file.write("\n\n")

        file.write(f"Resultados para {variable_name} (estandarizado):\n")
        file.write(results['comparison_df_standard'].to_string())
        file.write("\n\n")

        file.write(f"Resultados para {variable_name} (normalizado Min-Max):\n")
        file.write(results['comparison_df_minmax'].to_string())
        file.write("\n\n" + "="*80 + "\n\n")

print("Los resultados se han exportado a 'resultados.txt'")
```

Análisis de datos

Los datos presentados permiten extraer diversas inferencias sobre el funcionamiento de modelos de regresión lineal, polinomial y neuronal en conjuntos de datos variados y con distintas técnicas de normalización. A continuación se detallan las observaciones más destacadas:

Conjunto de datos sobre tiempo de estudio semanal:

- Sin Normalizar: El modelo basado en redes neuronales muestra superioridad (Índice de error 30.33, Coeficiente predictivo 0.048).
- Estandarizado: El enfoque basado en redes neuronales mantiene su ventaja (Índice de error 0.95, Coeficiente predictivo 0.048).
- Normalización Min-Max: El método basado en redes neuronales sigue liderando (Índice de error 0.076, Coeficiente predictivo 0.048).

En general, el enfoque basado en redes neuronales demuestra un rendimiento superior en todas las técnicas de preparación de datos para este conjunto.

Conjunto de datos sobre ausencias:

- Sin Normalizar: El modelo polinomial destaca (Índice de error 9.69, Coeficiente predictivo 0.865).
- Estandarizado: El enfoque polinomial mantiene su liderazgo (Índice de error 0.136, Coeficiente predictivo 0.865).
- Normalización Min-Max: El método polinomial sigue siendo el más efectivo (Índice de error 0.0115, Coeficiente predictivo 0.865).

El enfoque polinomial muestra un rendimiento superior en todas las técnicas de preparación de datos para este conjunto.

Conjunto de datos sobre apoyo familiar:

- Sin Normalizar: El modelo polinomial tiene una ligera ventaja (Índice de error 1.226, Coeficiente predictivo 0.0658). El enfoque basado en redes muestra un rendimiento inferior.
- Estandarizado: El método polinomial mantiene su superioridad (Índice de error 0.972, Coeficiente predictivo 0.0658).
- Normalización Min-Max: El enfoque polinomial continúa liderando (Índice de error 0.0766, Coeficiente predictivo 0.0658).

El modelo polinomial demuestra ser consistentemente superior para este conjunto de datos.

Conjunto de datos sobre tutoría:

- Sin Normalizar: El enfoque polinomial muestra el mejor rendimiento (Índice de error 0.205, Coeficiente predictivo 0.0379).
- Estandarizado: El modelo polinomial mantiene su ventaja (Índice de error 0.975, Coeficiente predictivo 0.0379).
- Normalización Min-Max: El método polinomial sigue siendo el más efectivo (Índice de error 0.205, Coeficiente predictivo 0.0379).

El enfoque polinomial demuestra un rendimiento superior en todas las técnicas de preparación de datos para este conjunto.

Conjunto de datos sobre actividades extracurriculares:

- Sin Normalizar: El modelo lineal muestra una leve superioridad (Índice de error 0.239, Coeficiente predictivo 0.0031).
- Estandarizado: El enfoque basado en redes toma la delantera (Índice de error 0.986, Coeficiente predictivo 0.0133).
- Normalización Min-Max: El método lineal recupera su ventaja (Índice de error 0.239, Coeficiente predictivo 0.0031).

Para este conjunto, los enfoques lineal y basado en redes muestran resultados competitivos, con una ligera ventaja del lineal en dos de las tres técnicas de preparación de datos.

Conjunto de datos sobre actividades físicas:

- Sin Normalizar: El modelo lineal muestra una leve superioridad (Índice de error 0.206, Coeficiente predictivo -0.0013).
- Estandarizado: El enfoque basado en redes destaca (Índice de error 0.993, Coeficiente predictivo 0.0068).
- Normalización Min-Max: El método lineal vuelve a destacar (Índice de error 0.206, Coeficiente predictivo -0.0013).

Para este conjunto, los enfoques lineal y basado en redes ofrecen resultados cercanos, con una sutil ventaja del lineal en dos de las tres técnicas de preparación de datos.

Conjunto de datos sobre música:

- Sin Normalizar: El modelo lineal muestra una leve superioridad (Índice de error 0.157, Coeficiente predictivo 0.0097).
- Estandarizado: El enfoque polinomial toma la delantera (Índice de error 0.993, Coeficiente predictivo 0.0095).
- Normalización Min-Max: El método polinomial mantiene su ventaja (Índice de error 0.157, Coeficiente predictivo 0.0095).

Para este conjunto, los enfoques lineal y polinomial muestran resultados competitivos, con una ligera ventaja del polinomial en dos de las tres técnicas de preparación de datos.

Conclusiones

Conclusiones generales:

- Regresión Neuronal: Sobresale en el conjunto de datos de tiempo de estudio semanal.
- Regresión Polinomial: Muestra mejor rendimiento en los conjuntos de ausencias, apoyo parental, tutoría y música.
- Regresión Lineal: Presenta un desempeño competitivo en los conjuntos de actividades extracurriculares y deportes.

Es relevante señalar que la normalización y estandarización de los datos pueden impactar significativamente en el rendimiento de los modelos, y la elección del modelo más adecuado puede variar según el preprocesamiento aplicado a los datos.