Promps usados:

Punto 2:

Se usaron los promps:

**Visualización:**

- genera un codigo en python para realizar el histograma de las posiciones de los jugadores en las columnas posicion vs numeros de jugadores de un data set

**salida:**

Aquí tienes un código en Python utilizando **pandas** y **matplotlib** para generar un histograma que muestra las posiciones de los jugadores frente al número de jugadores:

```python
import pandas as pd
import matplotlib.pyplot as plt

# Asegúrate de cargar tu dataset
# df = pd.read_csv('ruta_al_archivo.csv')

# Asegúrate de que la columna "Posición" esté limpia
df['Posición'] = df['Posición'].str.strip()  # Elimina espacios en blanco si los hay

# Cuenta la cantidad de jugadores por posición
position_counts = df['Posición'].value_counts()

# Crear el histograma
plt.figure(figsize=(10, 6))
plt.bar(position_counts.index, position_counts.values, color='skyblue', edgecolor='black')
plt.title('Número de jugadores por posición', fontsize=14)
plt.xlabel('Posición', fontsize=12)
plt.ylabel('Número de jugadores', fontsize=12)
```

```
plt.xticks(rotation=45)

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()


# Mostrar el gráfico

plt.show()
```

### Explicación:

1. **`df['Posición'].str.strip()`**: Limpia espacios en blanco en los datos de la columna `Posición`.

2. **`df['Posición'].value_counts()`**: Cuenta la frecuencia de cada posición.

3. **`plt.bar()`**: Crea un gráfico de barras donde las posiciones son las categorías del eje X y las cantidades el eje Y.

4. **Configuración adicional**:

   - Se personaliza el gráfico con títulos, etiquetas y colores.

   - Las posiciones en el eje X (`xticks`) tienen una rotación de 45° para que sean más legibles.

Este código es simple y efectivo para visualizar la distribución de posiciones de los jugadores en tu dataset. Si necesitas adaptar algo, avísame.

- haz un código para un histograma para la columna edades y el conteo de las mismas

  **Salida:**

  ```
  import matplotlib.pyplot as plt

  plt.hist(df['Edad'], bins=range(df['Edad'].min(), df['Edad'].max() + 2), edgecolor='black')
  plt.xlabel('Edades')
  plt.ylabel('Frecuencia')
  plt.title('Histograma de Edades')
  plt.show()
  ```

- genera histogramas para  , Número de Dobles y Triples Dobletes por Posición, promedio de puntos por posición, Número de jugadores por posición

'Promedio de puntos fantasy por posición en subgraficos ordenados'

```
# Número de Dobles y Triples Dobletes por Posición

dobles_por_posicion = df.groupby('Posición')['Dobles_Dobles'].sum()

triples_por_posicion = df.groupby('Posición')['Triples_Dobles'].sum()


# Promedio de puntos por posición

avg_points_by_position = df.groupby('Posición')['Puntos'].mean()


# Número de jugadores por posición

position_counts = df['Posición'].value_counts()


# Promedio de puntos fantasy por posición

promedio_fantasy_por_posicion = df.groupby('Posición')['Puntos_Fantasy'].mean()


# Crear figura con subgráficos

fig, axes = plt.subplots(2, 2, figsize=(16, 12))


# Gráfico 1: Número de Dobles Dobletes por Posición

axes[0, 0].bar(dobles_por_posicion.index, dobles_por_posicion.values, color='skyblue',
edgecolor='black')

axes[0, 0].set_title('Número de Dobles Dobletes por Posición')

axes[0, 0].set_xlabel('Posiciones')

axes[0, 0].set_ylabel('Cantidad')

axes[0, 0].tick_params(axis='x', rotation=45)


# Gráfico 2: Número de Triples Dobletes por Posición

axes[0, 1].bar(triples_por_posicion.index, triples_por_posicion.values, color='orange',
edgecolor='black')

axes[0, 1].set_title('Número de Triples Dobletes por Posición')

axes[0, 1].set_xlabel('Posiciones')
```

```
axes[0, 1].set_ylabel('Cantidad')

axes[0, 1].tick_params(axis='x', rotation=45)


# Gráfico 3: Promedio de Puntos por Posición

axes[1, 0].bar(avg_points_by_position.index, avg_points_by_position.values, color='green', edgecolor='black')

axes[1, 0].set_title('Promedio de Puntos por Posición')

axes[1, 0].set_xlabel('Posiciones')

axes[1, 0].set_ylabel('Promedio de Puntos')

axes[1, 0].tick_params(axis='x', rotation=45)


# Gráfico 4: Número de Jugadores por Posición

axes[1, 1].bar(position_counts.index, position_counts.values, color='purple', edgecolor='black')

axes[1, 1].set_title('Número de Jugadores por Posición')

axes[1, 1].set_xlabel('Posiciones')

axes[1, 1].set_ylabel('Número de Jugadores')

axes[1, 1].tick_params(axis='x', rotation=45)


# Ajuste del diseño

plt.tight_layout()

plt.show()


# Gráfico adicional: Promedio de Puntos Fantasy por Posición

plt.bar(promedio_fantasy_por_posicion.index, promedio_fantasy_por_posicion.values, color='cyan', edgecolor='black')

plt.title('Promedio de Puntos Fantasy por Posición')

plt.xlabel('Posición')

plt.ylabel('Promedio Puntos Fantasy')

plt.xticks(rotation=45)

plt.show()
```

- crea un histograma para el conteo de edades

  plt.hist(df['Edad'], bins=range(df['Edad'].min(), df['Edad'].max() + 2), edgecolor='black')
  plt.xlabel('Edades')
  plt.ylabel('Frecuencia')
  plt.title('Histograma de Edades')
  plt.show()
- implementa el esquema de validación cruzada para la regresión lineal de un set de datos ajustando el gidsearch para busqueda de hiperparametros en términos de MSE y con incorporación de hiperparametros y presenta los rendimientos promedios MSE R2 MAPE Y MAE

  **SALIDA**

  Para implementar un esquema de **validación cruzada** con la regresión lineal usando **GridSearchCV** y ajustar los **hiperparámetros** en términos de **MSE**, con la incorporación de los hiperparámetros, te proporcionaré el código completo y detallado.

  **Requisitos:**

1. **Modelo de regresión lineal**: Usaremos LinearRegression o Ridge para este ejemplo.
2. **Búsqueda de hiperparámetros con GridSearchCV**: Estableceremos el grid de hiperparámetros para la regresión.
3. **Validación cruzada**: Usaremos 5 folds en la validación cruzada.
4. **Métricas de evaluación**: Calcularemos las métricas de **MSE**, **R2**, **MAPE** y **MAE**.

   **Código:**

   ```
   # Importar las librerías necesarias
   from sklearn.model_selection import train_test_split, GridSearchCV, cross_validate
   from sklearn.linear_model import LinearRegression, Ridge
   from sklearn.metrics import make_scorer, mean_absolute_error, mean_squared_error, r2_score
   import numpy as np
   import pandas as pd
   import matplotlib.pyplot as plt

   # Dividir los datos en variables predictoras y objetivo
   X = base_datos_alta_correlacion.drop(columns=['Puntos'])
   y = base_datos_alta_correlacion['Puntos']

   # Dividir en conjuntos de entrenamiento y prueba
   X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

   # Inicializar el modelo base (puedes usar LinearRegression o Ridge)
   linear_model = LinearRegression()  # o Ridge() si deseas regularización

   # Definir las métricas de evaluación
   scoring = {
      'MAE': make_scorer(mean_absolute_error),
   ```

```
    'MSE': make_scorer(mean_squared_error),
    'R2': make_scorer(r2_score),
    'MAPE': make_scorer(lambda y_true, y_pred: np.mean(np.abs((y_true - y_pred) /
y_true)) * 100, greater_is_better=True)
}

# Configurar el grid de hiperparámetros para LinearRegression o Ridge (agregar
regularización)
param_grid = {
    'fit_intercept': [True, False],  # Ajuste o no el intercepto
    'normalize': [True, False]  # Normalizar las características (solo para LinearRegression, no
para Ridge)
}

# Realizar búsqueda de hiperparámetros con GridSearchCV
grid_search = GridSearchCV(
    estimator=linear_model,
    param_grid=param_grid,
    scoring='neg_mean_squared_error',  # Evaluar en términos de MSE
    cv=5,  # Validación cruzada con 5 folds
    n_jobs=-1  # Usar todos los núcleos disponibles
)

# Entrenar el modelo con la búsqueda de hiperparámetros
grid_search.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

# Realizar validación cruzada con el mejor modelo
cv_results = cross_validate(best_model, X_train, y_train, cv=5, scoring=scoring,
return_train_score=False)

# Resumir los resultados
```

**Explicación del código:**

1. **Carga de datos**:
    - Estoy usando el conjunto de datos diabetes de sklearn como ejemplo. Reemplaza base_datos_alta_correlacion con tu propio conjunto de datos.
2. **División del conjunto de datos**:
    - Se divide el conjunto de datos en conjuntos de entrenamiento y prueba con **train_test_split**.
3. **Modelo y búsqueda de hiperparámetros**:

    - Se crea el modelo Ridge (o LinearRegression si prefieres) y se realiza una **búsqueda de hiperparámetros** utilizando GridSearchCV. Aquí se buscan los mejores valores de **alpha**, **fit_intercept** y **max_iter**.
4. **Validación cruzada**:
    - Se realiza la validación cruzada con **5 folds** utilizando cross_validate y las métricas **MAE**, **MSE**, **R2**, y **MAPE**.
5. **Resultados de validación cruzada**:
    - Se calculan los promedios y desviaciones estándar de las métricas en los 5 folds.
6. **Evaluación en el conjunto de prueba**:
    - Después de obtener los mejores hiperparámetros, se evalúa el modelo final en el conjunto de prueba y se calculan las mismas métricas (MAE, MSE, R2, MAPE).
7. **Gráfica de resultados**:
    - Finalmente, se genera una gráfica de **predicciones vs. valores reales** para visualizar cómo el modelo ha aprendido los datos.

**Resultados esperados:**

El **output** incluirá:

- **Mejores hiperparámetros** seleccionados por GridSearchCV.
- **Promedios de las métricas** (MSE, R2, MAPE, MAE) en los 5 folds de la validación cruzada.
- **Resultados en el conjunto de prueba**, con las métricas de evaluación.
- **Gráfica** que compara las predicciones contra los valores reales.

---

Este código te permite ajustar un modelo de regresión lineal utilizando validación cruzada, optimizar los hiperparámetros y calcular las métricas de evaluación clave. Si necesitas realizar algún ajuste o ampliación, házmelo saber. 😊

- Para adaptar el código a **Procesos Gaussianos (Gaussian Processes)**, utilizaremos el modelo **GaussianProcessRegressor** de sklearn. Este modelo es ideal para la regresión no lineal y puede manejar tanto la predicción como la estimación de la incertidumbre en los resultados.

  Los **procesos gaussianos** no requieren una matriz de características explícita para la transformación de las características como en los modelos de regresión lineales o de árboles. En su lugar, utilizan un **kernel** para capturar las relaciones entre los puntos de los datos.

  Voy a ajustar tu código para usar **GaussianProcessRegressor**, con algunos **hiperparámetros** comunes que puedes ajustar:

- **kernel**: Controla el tipo de kernel utilizado para modelar las relaciones.
- **alpha**: Controla la varianza del ruido (como una regularización).
- **n_restarts_optimizer**: Número de reinicios para el optimizador de la función de verosimilitud.

  **Código adaptado para Procesos Gaussianos:**

  from sklearn.model_selection import train_test_split, GridSearchCV, cross_validate
  from sklearn.gaussian_process import GaussianProcessRegressor

```
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C, RationalQuadratic as RQ
from sklearn.metrics import make_scorer, mean_absolute_error, mean_squared_error, r2_score
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Dividir los datos en variables predictoras y objetivo
X = base_datos_alta_correlacion.drop(columns=['Puntos'])
y = base_datos_alta_correlacion['Puntos']

# Dividir en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Inicializar el modelo base
kernel = C(1.0, (1e-3, 1e3)) * RBF(length_scale=1.0, length_scale_bounds=(1e-3, 1e3))
gaussian_process_model = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)

# Definir las métricas de evaluación
scoring = {
   'MAE': make_scorer(mean_absolute_error),
   'MSE': make_scorer(mean_squared_error),
   'R2': make_scorer(r2_score),
   'MAPE': make_scorer(lambda y_true, y_pred: np.mean(np.abs((y_true - y_pred) / y_true)) * 100, greater_is_better=True)
}
# Ajustar el kernel y los límites de los parámetros
kernel = C(1.0, (1e-3, 1e4)) * RBF(length_scale=1.0, length_scale_bounds=(1e-3, 1e4))

# Param grid actualizado
param_grid = {
   'kernel': [
      C(1.0, (1e-2, 1e4)) * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e4)),
      C(1.0, (1e-2, 1e4)) * RQ(alpha=1.0, length_scale=1.0),
      C(1.0, (1e-2, 1e4)) * (RBF(length_scale=1.0) + RQ(alpha=1.0, length_scale=1.0))
   ]
}



# Realizar búsqueda de hiperparámetros con GridSearchCV
grid_search = GridSearchCV(
   estimator=gaussian_process_model,
```

```
    param_grid=param_grid,
    scoring='neg_mean_squared_error',
    cv=5,
    n_jobs=-1
)

# Entrenar el modelo con búsqueda de hiperparámetros
grid_search.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

# Realizar validación cruzada con el mejor modelo
cv_results = cross_validate(best_model, X_train, y_train, cv=5, scoring=scoring,
return_train_score=False)

# Resumir resultados de validación cruzada
results_summary = {
    metric: [np.mean(cv_results[f'test_{metric}']), np.std(cv_results[f'test_{metric}'])]
    for metric in scoring.keys()
}

# Convertir a DataFrame
results_df = pd.DataFrame(results_summary, index=["Mean", "Std"]).T

# Imprimir resultados de validación cruzada
print("Resultados de Validación Cruzada con GaussianProcessRegressor (5 folds):")
print("Mejores hiperparámetros:", best_params)
print(results_df)

# Evaluar el modelo final en el conjunto de prueba
y_pred = best_model.predict(X_test)

test_mae = mean_absolute_error(y_test, y_pred)
test_mse = mean_squared_error(y_test, y_pred)
test_r2 = r2_score(y_test, y_pred)
test_mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100

print("\nResultados en el conjunto de prueba:")
print(f"Mejores Hiperparámetros: {best_params}")
print(f"MAE: {test_mae}")
print(f"MSE: {test_mse}")
print(f"R2: {test_r2}")
```

```
print(f"MAPE: {test_mape}")

# Graficar los resultados de predicciones vs valores reales
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.7, color='b', label='Predicciones')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='r', linestyle='--',
linewidth=2, label='Línea ideal')
plt.xlabel('Valores Reales')
plt.ylabel('Predicciones')
plt.title('Gráfica de Predicciones vs Valores Reales para GaussianProcessRegressor')
plt.legend()
plt.grid(True)
plt.show()
```

**Descripción de los cambios:**

1. **Modelo de Proceso Gaussiano**:
   - Se usa GaussianProcessRegressor y se le asigna un **kernel compuesto** de una constante (C) y una función de base radial (RBF), que es el kernel más común en los procesos gaussianos.
2. **Hiperparámetros**:
   - Se utiliza alpha para regularizar la varianza del ruido. Es un parámetro muy importante que controla la cantidad de ruido permitido en las predicciones.
   - **kernel**: He configurado varios **kernels** posibles con diferentes longitudes de escala para optimizar la relación entre las características de entrada.
   - **n_restarts_optimizer**: Esto controla el número de reinicios del optimizador para evitar quedar atrapado en un mínimo local y mejorar la convergencia del modelo.
3. **Búsqueda de Hiperparámetros con GridSearchCV**:
   - La búsqueda de hiperparámetros se realiza sobre varios valores de kernel, alpha y n_restarts_optimizer.
4. **Evaluación**:
   - **Validación cruzada**: Se realiza validación cruzada para calcular las métricas de evaluación.
   - **Métricas de rendimiento**: Se calculan las métricas **MAE**, **MSE**, **R2** y **MAPE** tanto en el conjunto de validación cruzada como en el conjunto de prueba.
5. **Gráfico**:
   - Se genera un gráfico de **predicciones vs valores reales** para verificar visualmente el rendimiento del modelo.

**Hiperparámetros del Kernel:**

En **procesos gaussianos**, el kernel juega un papel crucial. El kernel **RBF** es ampliamente utilizado para capturar relaciones no lineales, y la constante **C** controla la escala de esta relación. Los parámetros de longitud de escala en el kernel **RBF** también afectan cómo el modelo mide la cercanía entre puntos de datos. En este caso, usé una **longitud de escala** de 1.0, pero esto puede ajustarse según el problema.

**Resumen:**

Este código adapta tu flujo de trabajo para usar **Procesos Gaussianos** para regresión, realizando una búsqueda de hiperparámetros, evaluando el modelo con validación cruzada, calculando las métricas y generando una gráfica para comparar las predicciones con los valores reales.

Si tienes más preguntas o necesitas ajustes adicionales, ¡házmelo saber! 😊