
Aprendizaje Automático – 63GIIN

Actividad 2 – Portafolio

Gagliardo Miguel Angel

26 de Diciembre de 2024

Disclaimer: Todas las imágenes de este documento fueron tomadas del repositorio donde está el trabajo: <https://github.com/magliardo/viu-63giin-uc2/>

En el mismo, se puede ver el jupyter notebook con todo el código, librerías, imports y demás.

Cualquier consulta referirse a dicho repo que tiene todos los detalles.

Información del Dataset cargada raw

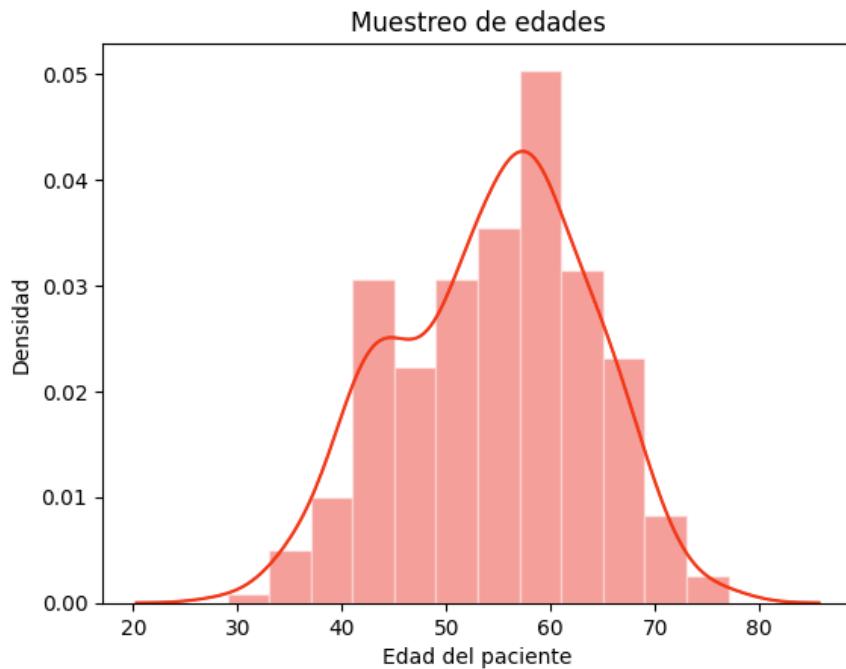
```
# Cargamos el CSV con pandas
data = pd.read_csv("./Infarto.csv")
data.head(10)
```

	age	sex	cp	trtbps	chol	fb	restecg	thalachh	exng	oldpeak	sip	caa	thall	output
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
5	57	1	0	140	192	0	1	148	0	0.4	1	0	1	1
6	56	0	1	140	294	0	0	153	0	1.3	1	0	2	1
7	44	1	1	120	263	0	1	173	0	0.0	2	0	3	1
8	52	1	2	172	199	1	1	162	0	0.5	2	0	3	1
9	57	1	2	150	168	0	1	174	0	1.6	2	0	2	1

Muestreo de edades de los pacientes en el Dataset

```
# Mostramos los datos por edad
edades = sns.histplot(
    data["age"],
    kde=True,
    stat="density",
    kde_kws=dict(cut=3),
    alpha=.4,
    edgecolor=(1, 1, 1, .4),
    color="red"
)
edades.set(xlabel = "Edad del paciente", ylabel = "Densidad")
plt.title("Muestreo de edades")
```

Text(0.5, 1.0, 'Muestreo de edades')



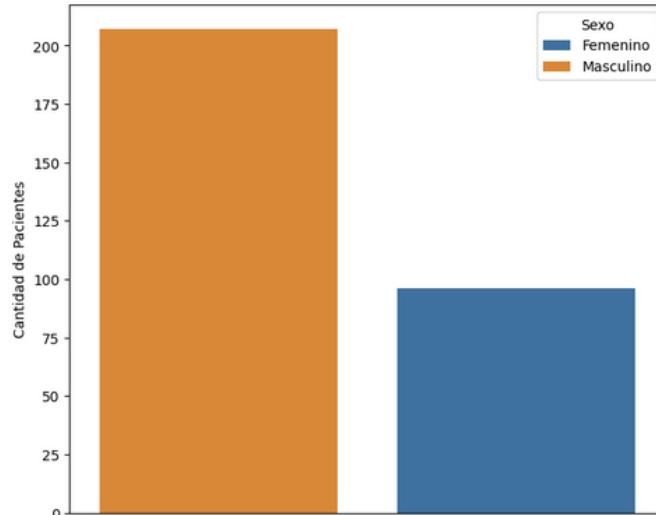
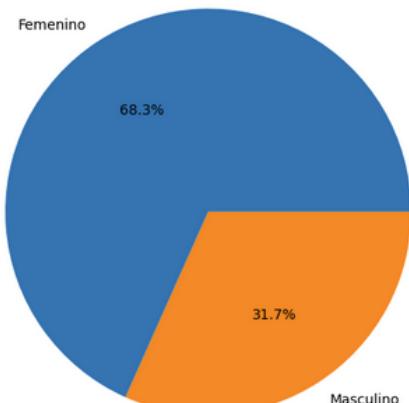
Relacion entre sexo del paciente y enfermedad

```
# Mostramos las relación entre enfermedades cardíacas y el sexo del paciente
labels = ["Femenino", "Masculino"]

# Gráfico de Torta
fig, ax = plt.subplots(1, 2, figsize=(10, 6))
fig.tight_layout()
data["sex"].value_counts().plot.pie(
    ax=ax[0],
    autopct="%1.1f%%",
    labels=labels
)

# Gráfico de Barras
sns.countplot(x="sex", hue = "sex", data = data, ax = ax[1], order = data["sex"].value_counts().index)
fig.suptitle("Enfermedades cardíacas por sexo")
plt.legend(title="Sexo", loc="upper right", labels=labels)
plt.subplots_adjust(left=None, bottom=None, right=1.3, top=0.9, wspace=None, hspace=None)
plt.xlabel("Sexo")
plt.ylabel("Cantidad de Pacientes")
ax[0].get_yaxis().set_visible(False)
ax[1].get_xaxis().set_visible(False)
plt.show()
```

Enfermedades cardíacas por sexo



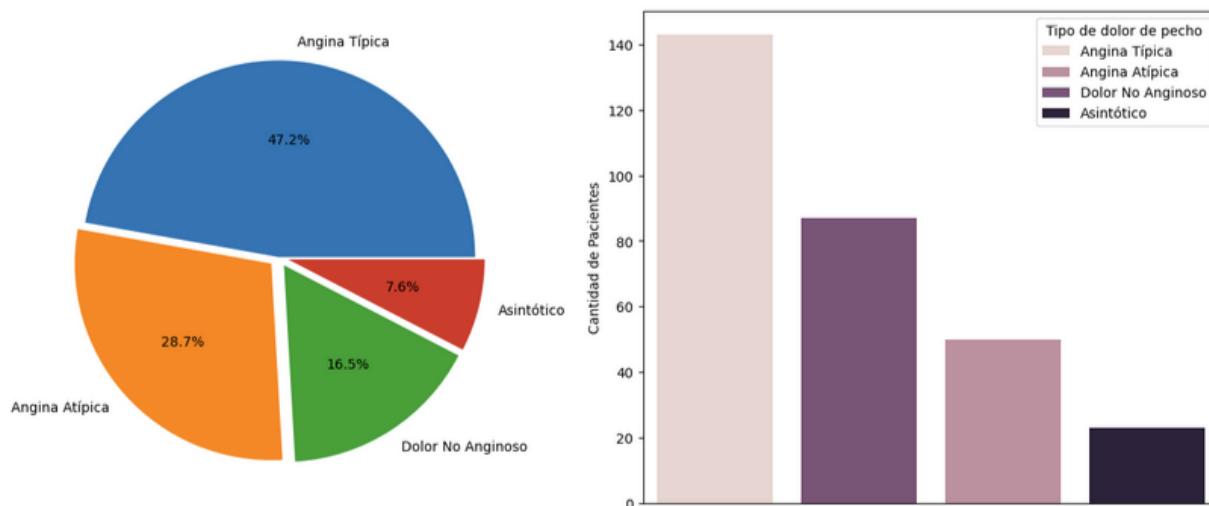
Tipos de enfermedades más comunes

```
# Tipos de dolores de pecho
labels = [
    "Angina Típica",
    "Angina Atípica",
    "Dolor No Anginoso",
    "Asintótico",
]

#
fig , ax = plt.subplots(1, 2, figsize=(10,6))
fig.tight_layout()
data["cp"].value_counts().plot.pie(
    ax=ax[0],
    autopct="%1.1f%%",
    explode=[0, 0.05, 0.05, 0.05],
    labels=labels
)
cp_plot = sns.countplot(x="cp", hue="cp", data=data, ax=ax[1], order=data["cp"].value_counts().index)

# Gráfico de Barras
fig.suptitle("Enfermedades por tipo de dolor de pecho")
plt.legend(
    title="Tipo de dolor de pecho",
    loc="upper right",
    labels=labels
)
ax[0].get_yaxis().set_visible(False)
ax[1].get_xaxis().set_visible(False)
plt.subplots_adjust(left=None, bottom=None, right=1.3, top=0.9, wspace=None, hspace=None)
plt.xlabel("Tipo de dolor")
plt.ylabel("Cantidad de Pacientes")
plt.show()
```

Enfermedades por tipo de dolor de pecho



Primer modelo: RandomForest y sus resultados

Entrenamiento con RandomForest

```

# Ahora tenemos que crear tanto los datos de entrenamiento como las características independientes
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# El modelo que hemos importado es: Random Forest. Usaremos 100 árboles para la estimación.
random_forest = RandomForestClassifier(n_estimators=100)

# Ajustamos el modelo con los parámetros de entrenamiento que creamos
random_forest.fit(X_train, y_train)
y_pred_random_forest = random_forest.predict(X_test)
random_forest.score(X_train, y_train)

: 1.0

: display_markdown("""
### Generamos el informe del modelo RandomForest

- Precision: La proporción de predicciones correctas entre todas las predicciones realizadas
- Recall: La proporción de verdaderos positivos entre todos los ejemplos que realmente pertenecen a la clase positiva
- F1-Score: La media armónica entre la precisión y el recall. Es una medida que combina ambos aspectos en una sola métrica
- Support: El número de ocurrencias de cada clase en los datos reales
""", raw=True)

print(classification_report(y_test, y_pred_random_forest))

display_markdown("""
### Resultados para RandomForest
- El modelo tiene una precisión de 81% para la clase 0 (sin enfermedad) y un recall de 81%
- El modelo tiene una precisión de 85% para la clase 1 (con enfermedad) y un recall de 85%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 84%
""", raw=True)

```

Generamos el informe del modelo RandomForest

- Precision: La proporción de predicciones correctas entre todas las predicciones realizadas
- Recall: La proporción de verdaderos positivos entre todos los ejemplos que realmente pertenecen a la clase positiva
- F1-Score: La media armónica entre la precisión y el recall. Es una medida que combina ambos aspectos en una sola métrica
- Support: El número de ocurrencias de cada clase en los datos reales

	precision	recall	f1-score	support
0	0.88	0.85	0.87	27
1	0.89	0.91	0.90	34
accuracy			0.89	61
macro avg	0.89	0.88	0.88	61
weighted avg	0.89	0.89	0.88	61

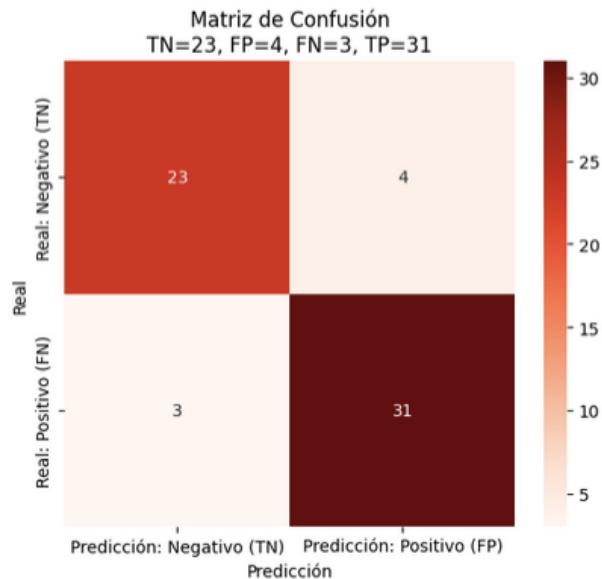
Resultados para RandomForest

- El modelo tiene una precisión de 81% para la clase 0 (sin enfermedad) y un recall de 81%
- El modelo tiene una precisión de 85% para la clase 1 (con enfermedad) y un recall de 85%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 84%

Generamos la matriz de confusión para RandomForest

Una matriz de confusión es una tabla que muestra el número de predicciones correctas e incorrectas clasificadas por clase:

- **Verdaderos positivos (TP):** Casos correctamente clasificados como positivos.
- **Falsos positivos (FP):** Casos incorrectamente clasificados como positivos.
- **Falsos negativos (FN):** Casos incorrectamente clasificados como negativos.
- **Verdaderos negativos (TN):** Casos correctamente clasificados como negativos.



Segundo Modelo: KNN y sus resultados

Entrenamiento con KNN

- Precision: La proporción de predicciones correctas entre todas las predicciones realizadas
- Recall: La proporción de verdaderos positivos entre todos los ejemplos que realmente pertenecen a la clase positiva
- F1-Score: La media armónica entre la precisión y el recall. Es una medida que combina ambos aspectos en una sola métrica
- Support: El número de ocurrencias de cada clase en los datos reales

```
|: from sklearn.neighbors import KNeighborsClassifier
# Creamos el modelo con k = 5. O sea basándonos en las 5 observaciones más cercanas en el conjunto de entrenamiento
knn = KNeighborsClassifier(n_neighbors=5)
# La función fit() ajusta el modelo a los datos de entrenamiento (X_train para las características y y_train para las etiquetas)
knn.fit(X_train, y_train)

# La función predict() realiza las predicciones del modelo utilizando los datos de prueba (X_test).
# El modelo asigna una etiqueta de clase a cada ejemplo del conjunto de prueba, basándose en la proximidad de sus vecinos
knnpredict = knn.predict(X_test)

display_markdown("""
### Generamos el informe del modelo K-Nearest Neighbors (KNN)

- Precision: La proporción de predicciones correctas entre todas las predicciones realizadas
- Recall: La proporción de verdaderos positivos entre todos los ejemplos que realmente pertenecen a la clase positiva
- F1-Score: La media armónica entre la precisión y el recall. Es una medida que combina ambos aspectos en una sola métrica
- Support: El número de ocurrencias de cada clase en los datos reales
""", raw=True)
print(classification_report(y_test, knnpredict))

display_markdown("""
#### Resultados para KNN
- El modelo tiene una precisión de 59% para la clase 0 (sin enfermedad) y un recall de 63%
- El modelo tiene una precisión de 69% para la clase 1 (con enfermedad) y un recall de 65%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 64%
""", raw=True)
```

Generamos el informe del modelo K-Nearest Neighbors (KNN)

- Precision: La proporción de predicciones correctas entre todas las predicciones realizadas
- Recall: La proporción de verdaderos positivos entre todos los ejemplos que realmente pertenecen a la clase positiva
- F1-Score: La media armónica entre la precisión y el recall. Es una medida que combina ambos aspectos en una sola métrica
- Support: El número de ocurrencias de cada clase en los datos reales

	precision	recall	f1-score	support
0	0.59	0.63	0.61	27
1	0.69	0.65	0.67	34
<hr/>				
accuracy			0.64	61
macro avg	0.64	0.64	0.64	61
weighted avg	0.64	0.64	0.64	61

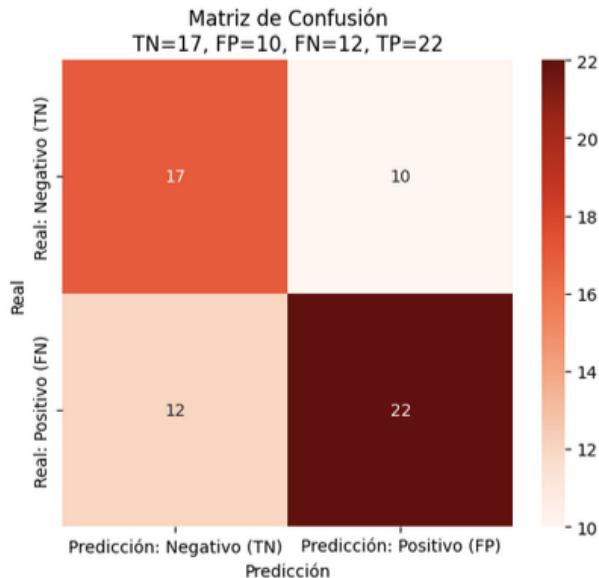
Resultados para KNN

- El modelo tiene una precisión de 59% para la clase 0 (sin enfermedad) y un recall de 63%
- El modelo tiene una precisión de 69% para la clase 1 (con enfermedad) y un recall de 65%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 64%

Generamos la matriz de confusión

Una matriz de confusión es una tabla que muestra el número de predicciones correctas e incorrectas clasificadas por clase:

- **Verdaderos positivos (TP)**: Casos correctamente clasificados como positivos.
- **Falsos positivos (FP)**: Casos incorrectamente clasificados como positivos.
- **Falsos negativos (FN)**: Casos incorrectamente clasificados como negativos.
- **Verdaderos negativos (TN)**: Casos correctamente clasificados como negativos.



Tercer Modelo: Regresión Logística y sus resultados

Entrenamiento con Regresión Logística

```
# penalty: L2 es una regularización de tipo 'Ridge', que penaliza los coeficientes grandes para evitar sobreajuste
# dual: Este parámetro indica que no se utilizará la formulación dual para la optimización. Se usa True solo cuando el problema es linealmente separable
# C: Es el parámetro de regularización que controla la magnitud de la penalización. Un valor más bajo indica una mayor regularización
model = LogisticRegression(fit_intercept=True, penalty='l2', dual=False, C=1.0)
model.fit(X_train, y_train)
model.predict = model.predict(X_test)

# Utilizamos los mismos reportes anteriormente vistos
print(classification_report(y_test, model.predict))
display_markdown(""""

#### Resultados para Regresión Logística
- El modelo tiene una precisión de 85% para la clase 0 (sin enfermedad) y un recall de 81%
- El modelo tiene una precisión de 86% para la clase 1 (con enfermedad) y un recall de 88%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 85%
""", raw=True)
```

	precision	recall	f1-score	support
0	0.85	0.81	0.83	27
1	0.86	0.88	0.87	34
accuracy			0.85	61
macro avg	0.85	0.85	0.85	61
weighted avg	0.85	0.85	0.85	61

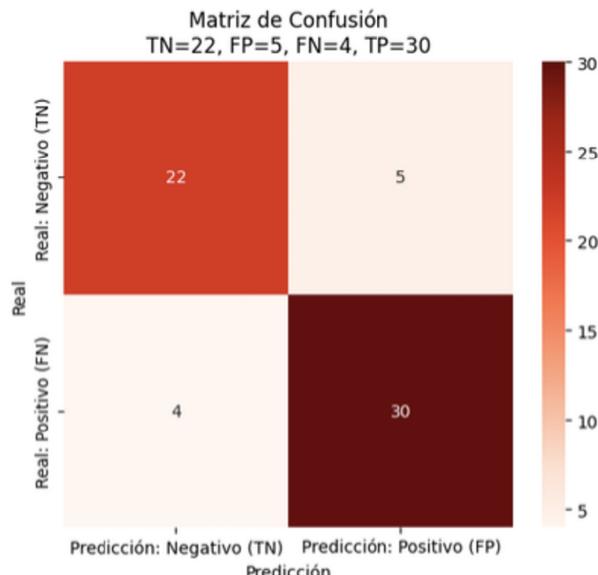
```
/Users/gattes/Documents/repos/github/viu-63giin-uc2/.venv/lib/python3.9/site-packages/sklearn/linear_model/_logistic.py:465: ConvergenceWarning: lbfsgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression
n_iter_i = _check_optimize_result()
```

Resultados para Regresión Logística

- El modelo tiene una precisión de 85% para la clase 0 (sin enfermedad) y un recall de 81%
- El modelo tiene una precisión de 86% para la clase 1 (con enfermedad) y un recall de 88%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 85%

Generamos la matriz de confusión para regresión logística



Primer Conclusión

El modelo de **Random Forest** ha mostrado el mejor rendimiento, con una precisión, recall y F1-score más altos en comparación con KNN y Regresión Logística.

En particular, Random Forest tiene una mayor capacidad para predecir ambas clases (sin enfermedad y con enfermedad) con una precisión global del 87%, mientras que KNN tiene un rendimiento notablemente más bajo (64%). Regresión Logística se encuentra en un punto intermedio, con un 85% de precisión, lo que la hace una opción competitiva, pero aún por debajo de Random Forest.

Optimización

Paso 1: Optimizando nuestro modelo (RandomForest)

Optimización de Hiperparámetros

En Random Forest, los parámetros clave que podemos ajustar son:

- **n_estimators**: Número de árboles en el bosque. Aunque 100 es un valor estándar, vamos a probar con valores mayores (por ejemplo, 200, 300) para ver si mejora la precisión. Sin embargo, más árboles normalmente pueden aumentar el tiempo de cómputo.
- **max_depth**: La profundidad máxima de los árboles. Si los árboles son demasiado profundos, pueden sobreajustar los datos. Limitar la profundidad de los árboles puede ayudar a generalizar mejor.
- **min_samples_split**: El número mínimo de muestras necesarias para dividir un nodo. Aumentar este valor puede hacer que los árboles sean más simples y

evitar el sobreajuste.

- **min_samples_leaf:** El número mínimo de muestras necesarias para estar en una hoja. Aumentarlo puede hacer que el modelo sea más robusto y menos susceptible a las fluctuaciones menores de los datos.
- **max_features:** El número máximo de características que se consideran para dividir cada nodo. Podemos probar con sqrt o log2, que son valores comunes y pueden mejorar el rendimiento al reducir la correlación entre los árboles.
- **bootstrap:** Si se utiliza muestreo con reemplazo para la construcción de los árboles. Normalmente se establece en True, pero es interesante probarlo con False para ver si mejora la generalización.

Métodos para optimizar

Podemos usar Busqueda de Cuadrícula (GridSearch) o Busqueda Aleatoria (RandomizedSearch) para realizar una búsqueda en el espacio de los hiperparámetros y encontrar la mejor combinación.

```

: display_markdown("## Optimizando usando Busqueda de Cuadrícula", raw=True)

# Definimos el espacio de búsqueda para los hiperparámetros
param_grid = {
    'n_estimators': [100, 200, 300],           # Número de árboles
    'max_depth': [None, 10, 20, 30],          # Profundidad máxima de los árboles
    'min_samples_split': [2, 5, 10],          # Mínimo de muestras para dividir un nodo
    'min_samples_leaf': [1, 2, 4],            # Mínimo de muestras en una hoja
    'max_features': ['auto', 'sqrt', 'log2'], # Número de características a considerar
    'bootstrap': [True, False]                # Si se utiliza muestreo con reemplazo
}

# Recreamos el modelo base
random_forest = RandomForestClassifier(random_state=0)

# Configuramos GridSearchCV con validación cruzada de 5
grid_search = GridSearchCV(estimator=random_forest, param_grid=param_grid, cv=5, n_jobs=-1, verbose=0)

# Ajustamos el modelo a los datos de entrenamiento
grid_search.fit(X_train, y_train)

```

Mejores parámetros encontrados con Búsqueda de Cuadricula:

```
{'bootstrap': True,
 'max_depth': None,
 'max_features': 'sqrt',
 'min_samples_leaf': 2,
 'min_samples_split': 10,
 'n_estimators': 100}
```

Precisión del mejor modelo en el conjunto de prueba: 0.8197

```
:
display_markdown("""
## Informe del modelo RandomForest luego de la optimización con Búsqueda de Cuadricula
""", raw=True)

print(classification_report(y_test, y_pred_grid_search))

display_markdown("""
### Resultados para RandomForest
- El modelo tiene una precisión de 83% para la clase 0 (sin enfermedad) y un recall de 74%
- El modelo tiene una precisión de 81% para la clase 1 (con enfermedad) y un recall de 88%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 82%
""", raw=True)

display_markdown("""
## Ahora la matriz de confusión del modelo luego de la optimización con Búsqueda de Cuadricula
""", raw=True)

# Generar la matriz de confusión
cm = confusion_matrix(y_test, y_pred_grid_search)

# Extraer los valores de la matriz de confusión para TP, FP, FN, TN
TN, FP, FN, TP = cm.ravel()

# Crear un mapa de calor para mostrar la matriz de confusión con las etiquetas TP, FP, FN, TN
plt.figure(figsize=(6, 5)) # Tamaño de la figura
sns.heatmap(cm, annot=True, fmt="d", cmap="Reds",
            xticklabels=["Predicción: Negativo (TN)", "Predicción: Positivo (FP)"],
            yticklabels=["Real: Negativo (TN)", "Real: Positivo (FN)"])

# Añadir etiquetas y título
plt.xlabel("Predicción")
plt.ylabel("Real")
plt.title(f"Matriz de Confusión\nTN={TN}, FP={FP}, FN={FN}, TP={TP}")

plt.show()
```

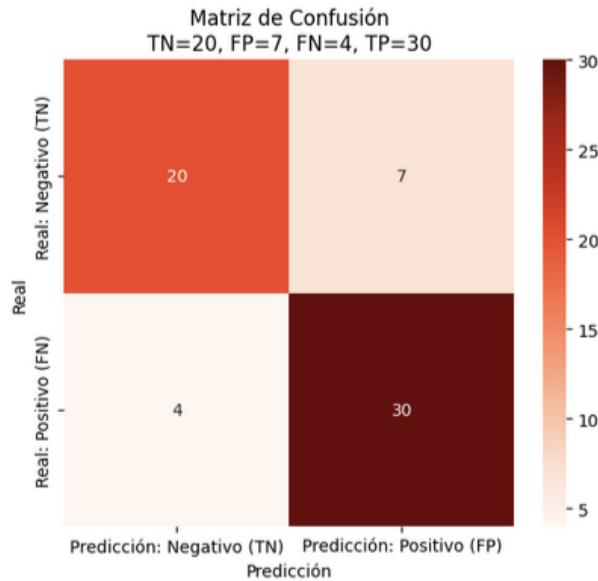
Informe del modelo RandomForest luego de la optimización con Búsqueda de Cuadricula

	precision	recall	f1-score	support
0	0.83	0.74	0.78	27
1	0.81	0.88	0.85	34
accuracy			0.82	61
macro avg	0.82	0.81	0.81	61
weighted avg	0.82	0.82	0.82	61

Resultados para RandomForest

- El modelo tiene una precisión de 83% para la clase 0 (sin enfermedad) y un recall de 74%
- El modelo tiene una precisión de 81% para la clase 1 (con enfermedad) y un recall de 88%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 82%

Ahora la matriz de confusión del modelo luego de la optimización con Búsqueda de Cuadricula



Optimizando usando Busqueda Aleatoria

Si el espacio de hiperparámetros es muy grande, podemos realizar una búsqueda más rápida pero aún efectiva usando Busqueda Aleatoria.

Este método selecciona aleatoriamente combinaciones de hiperparámetros y prueba un número fijo de iteraciones. Esto puede ser útil cuando tenemos una gran cantidad de parámetros.

```
# Definir el espacio de búsqueda para los hiperparámetros
param_dist = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [None, 10, 20, 30, 40],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2'],
    'bootstrap': [True, False]
}

# Recreamos el modelo base
random_forest = RandomForestClassifier(random_state=0)

# Configuramos RandomizedSearchCV con validación cruzada de 5
random_search = RandomizedSearchCV(estimator=random_forest, param_distributions=param_dist, n_iter=100, cv=5, n_jobs=-1)

# Ajustamos el modelo a los datos de entrenamiento
random_search.fit(X_train, y_train)
```

Mejores parámetros encontrados con Búsqueda Aleatoria

```
{'bootstrap': False,
 'max_depth': 20,
 'max_features': 'log2',
 'min_samples_leaf': 1,
 'min_samples_split': 20,
 'n_estimators': 400}
```

Precisión del mejor modelo en el conjunto de prueba: 0.8525

```
|: display_markdown("""
### Informe del modelo RandomForest luego de la optimización con Búsqueda Aleatoria
""", raw=True)

print(classification_report(y_test, y_pred_random_search))

display_markdown("""
### Resultados para RandomForest
- El modelo tiene una precisión de 85% para la clase 0 (sin enfermedad) y un recall de 81%
- El modelo tiene una precisión de 86% para la clase 1 (con enfermedad) y un recall de 88%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 85%
""", raw=True)

display_markdown("""
### Ahora la matriz de confusión del modelo luego de la optimización con Búsqueda Aleatoria
""", raw=True)

# Generar la matriz de confusión
cm = confusion_matrix(y_test, y_pred_random_search)

# Extraer los valores de la matriz de confusión para TP, FP, FN, TN
TN, FP, FN, TP = cm.ravel()

# Crear un mapa de calor para mostrar la matriz de confusión con las etiquetas TP, FP, FN, TN
plt.figure(figsize=(6, 5)) # Tamaño de la figura
sns.heatmap(cm, annot=True, fmt="d", cmap="Reds",
            xticklabels=["Predicción: Negativo (TN)", "Predicción: Positivo (FP)"],
            yticklabels=["Real: Negativo (TN)", "Real: Positivo (FN)"])

# Añadir etiquetas y título
plt.xlabel("Predicción")
plt.ylabel("Real")
plt.title(f"Matriz de Confusión\nTN={TN}, FP={FP}, FN={FN}, TP={TP}")

plt.show()
```

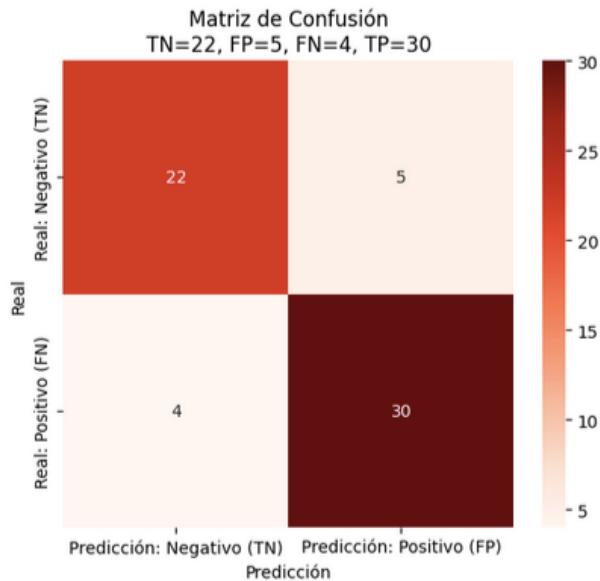
Informe del modelo RandomForest luego de la optimización con Búsqueda Aleatoria

	precision	recall	f1-score	support
0	0.85	0.81	0.83	27
1	0.86	0.88	0.87	34
accuracy			0.85	61
macro avg	0.85	0.85	0.85	61
weighted avg	0.85	0.85	0.85	61

Resultados para RandomForest

- El modelo tiene una precisión de 85% para la clase 0 (sin enfermedad) y un recall de 81%
- El modelo tiene una precisión de 86% para la clase 1 (con enfermedad) y un recall de 88%
- Finalmente entre lo más relevante, vemos que tenemos un accuracy (exactitud) de 85%

Ahora la matriz de confusión del modelo luego de la optimización con Búsqueda Aleatoria



Comparación de resultados post-optimización

La optimización con **RandomizedSearch** proporcionó la mejor mejora en comparación con el modelo original y la optimización por **GridSearch**. Mientras que el **GridSearch** introdujo un sesgo hacia la clase 1 (con enfermedad), sacrificando el rendimiento en la clase 0 (sin enfermedad), **RandomizedSearch** logró un equilibrio entre ambas clases, mejorando tanto la precisión como el recall, lo que resultó en una mayor precisión global.

Este ejercicio demuestra la importancia de una optimización bien equilibrada para mejorar el rendimiento del modelo, sin comprometer demasiado la detección de alguna de las clases.

Pero, por qué pueden mejorar o empeorar los resultados?

Como hemos visto, es posible que los resultados mejoren o empeoren después de aplicar optimizaciones como la búsqueda de cuadrícula o la búsqueda aleatoria, y existen varias razones por las cuales esto puede ocurrir:

- **Sobreajuste (Overfitting):** Cuando optimizamos un modelo, especialmente con técnicas como la búsqueda de cuadrícula, el modelo puede terminar ajustándose demasiado a los datos de entrenamiento, lo que reduce su capacidad para generalizar a datos nuevos (el conjunto de prueba).
- **Búsqueda demasiado exhaustiva o parámetros no adecuados:** Si durante la optimización el rango de parámetros seleccionados no es el adecuado para el modelo o el conjunto de datos, el proceso de optimización podría conducir a configuraciones subóptimas.
- **Evaluación incorrecta durante la optimización:** Dependiendo de cómo se evalúe el modelo durante el proceso de optimización, el rendimiento podría verse afectado por un error en la estrategia de validación o en la evaluación de los parámetros.
- **Hiperparámetros mal sintonizados:** Algunos hiperparámetros pueden ser más sensibles que otros, y pequeños ajustes en estos pueden llevar a un rendimiento mucho peor.
- **Métrica de optimización inadecuada:** Si el proceso de optimización se centra en una métrica que no refleja bien los objetivos del problema, los resultados pueden empeorar

Conclusión

- Al comparar los tres modelos (RandomForest, KNN y Regresión Lineal), el original Random Forest es claramente el más fuerte, con una accuracy de 89% y un F1-score alto para ambas clases.
- Comparado con los otros modelos, como KNN y Regresión Logística, que tuvieron resultados inferiores en cuanto a precision y recall, **Random Forest sigue siendo el modelo más robusto.**
- KNN presentó una accuracy de 64% y Regresión Logística de 85%, lo que demuestra que, en este caso, Random Forest proporciona el mejor equilibrio entre precisión y recall.

Reflexión final y aprendizajes

Finalmente y para coronar esta actividad, experimentamos con técnicas de optimización de hiperparámetros para mejorar el rendimiento:

- Aunque Random Forest mostró un rendimiento inicial sólido, las optimizaciones (búsqueda de cuadrícula y aleatoria) en ocasiones empeoraron los resultados debido a problemas de sobreajuste o una selección subóptima de parámetros, como es el caso de búsqueda por cuadrícula.
- En el caso de Búsqueda Aleatoria y tal como hemos visto, se mostró una mejora (aunque no sustancial) respecto del modelo inicial.
- Esto subraya la importancia de realizar una optimización cuidadosa y no asumir que

cualquier mejora en los hiperparámetros llevará automáticamente a un mejor rendimiento en el conjunto de prueba.

- El aprendizaje clave es que, aunque la optimización puede mejorar ciertos aspectos del modelo, es fundamental evaluar el impacto de los cambios de manera crítica, utilizando métricas adecuadas que se alineen con los objetivos del problema.
 - Además, cada modelo tiene sus fortalezas y limitaciones, y la clave para obtener buenos resultados no solo radica en elegir el modelo correcto, sino también en entender cómo afinar sus parámetros para equilibrar el ajuste y la generalización.