# PROYECTO DE CIENCIA DE DATOS

## Análisis del conjunto de datos de diabetes y medición del rendimiento de modelos aplicados.

## Índice

# 1. Introducción

El Conjunto de Datos de Diabetes de las Mujeres Pima, originario del Instituto Nacional de Diabetes y Enfermedades Digestivas y Renales, contiene información de 768 mujeres de una población cerca de Phoenix, Arizona, EE. UU. La variable objetivo evaluada fue la diabetes, de las cuales 258 dieron positivo y 500 negativo. Por lo tanto, hay una variable objetivo y los siguientes atributos :

- Pregnancies (número de veces embarazada),
- Oral glucose tolerance test (concentración de glucosa en plasma a las dos horas después de 75 g de glucosa anhidra en mg/dl),
- Blood pressure (Presión Arterial Diastólica en mmHg),
- Skin thickness (grosor del pliegue cutáneo del tríceps en mm),
- Insulin (insulina sérica a las 2 horas en mu U/ml),
- BMI (Índice de Masa Corporal en $kg/m^2$),
- Age (años),
- DiabetesPedigreeFunction (función que representa cuán probable es que contraigan la enfermedad extrapolando la historia de sus ancestros).

## 2. Objetivos

Los objetivos son:

1. Hacer un análisis del conjunto de datos y prepararlos para ser usados por diversos algoritmos.

2. Seleccionar el algoritmo con mejor rendimiento sobre el conjunto.

## 3. Análisis exploratorio de datos

Todo lo de esta sección se encuentra en el archivo “EDA.ipynb”.

### Librerías importadas

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import rcParams
```

## 3.1 Vista general

En un rápido vistazo a nuestros datos se observa que tenemos 768 renglones y 9 columnas, lo cual consideramos como un conjunto de datos pequeño, la ultima columna es la variable objetivo

| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 6 | 148 | 72 | 35 | 0 | 33.6 | 0.627 | 50 | 1 |
| 1 | 1 | 85 | 66 | 29 | 0 | 26.6 | 0.351 | 31 | 0 |
| 2 | 8 | 183 | 64 | 0 | 0 | 23.3 | 0.672 | 32 | 1 |
| 3 | 1 | 89 | 66 | 23 | 94 | 28.1 | 0.167 | 21 | 0 |
| 4 | 0 | 137 | 40 | 35 | 168 | 43.1 | 2.288 | 33 | 1 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 763 | 10 | 101 | 76 | 48 | 180 | 32.9 | 0.171 | 63 | 0 |
| 764 | 2 | 122 | 70 | 27 | 0 | 36.8 | 0.340 | 27 | 0 |
| 765 | 5 | 121 | 72 | 23 | 112 | 26.2 | 0.245 | 30 | 0 |
| 766 | 1 | 126 | 60 | 0 | 0 | 30.1 | 0.349 | 47 | 1 |
| 767 | 1 | 93 | 70 | 31 | 0 | 30.4 | 0.315 | 23 | 0 |

768 rows × 9 columns

Información de cada columna:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

Todas las variables son de carácter numérico. Ahora un análisis sencillo

```
data.describe()
```

| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin |
|---|---|---|---|---|---|
| count | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 |
| mean | 3.845052 | 120.894531 | 69.105469 | 20.536458 | 79.799479 |
| std | 3.369578 | 31.972618 | 19.355807 | 15.952218 | 115.244002 |
| min | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 1.000000 | 99.000000 | 62.000000 | 0.000000 | 0.000000 |
| 50% | 3.000000 | 117.000000 | 72.000000 | 23.000000 | 30.500000 |
| 75% | 6.000000 | 140.250000 | 80.000000 | 32.000000 | 127.250000 |
| max | 17.000000 | 199.000000 | 122.000000 | 99.000000 | 846.000000 |

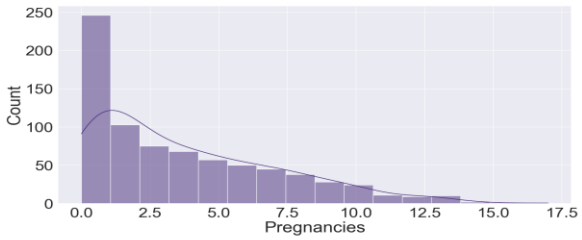
| BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|
| 768.000000 | 768.000000 | 768.000000 | 768.000000 |
| 31.992578 | 0.471876 | 33.240885 | 0.348958 |
| 7.884160 | 0.331329 | 11.760232 | 0.476951 |
| 0.000000 | 0.078000 | 21.000000 | 0.000000 |
| 27.300000 | 0.243750 | 24.000000 | 0.000000 |
| 32.000000 | 0.372500 | 29.000000 | 0.000000 |
| 36.600000 | 0.626250 | 41.000000 | 1.000000 |
| 67.100000 | 2.420000 | 81.000000 | 1.000000 |

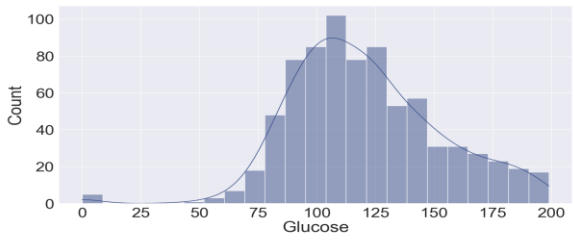
En las columnas de “Insulin” y “SkinThickness” tenemos una varianza muy grande, también se puede notar que hay una cantidad inusual de ceros. Veamos cuantos ceros hay por columna

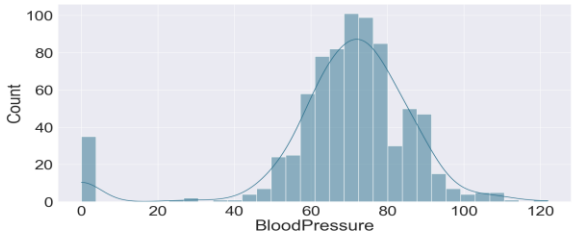
```
ceros = (data == 0).sum()
print(ceros)
```

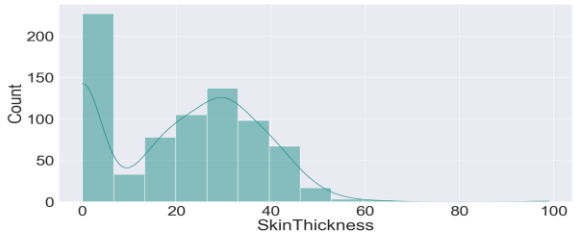
```
Pregnancies                 111
Glucose                       5
BloodPressure                35
SkinThickness               227
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                     500
```

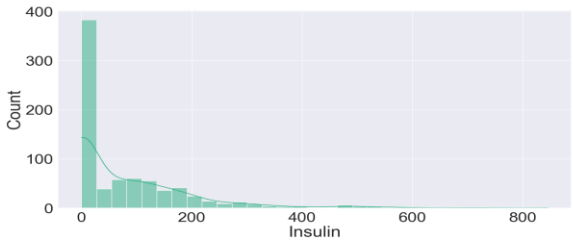
En dos columnas hay una gran cantidad en comparación con la cantidad de datos que tenemos. Revisemos las distribuciones

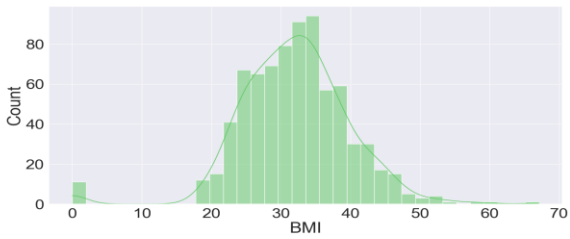


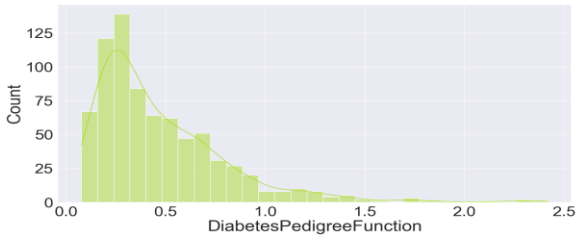



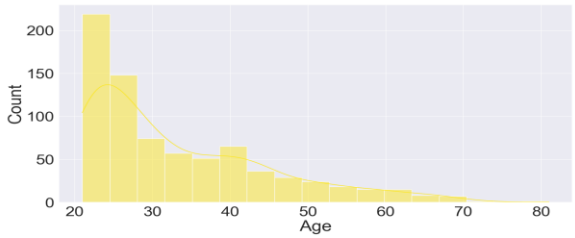



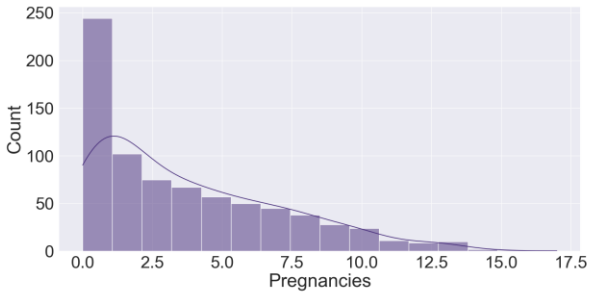
## 33.2 Nans y outliers

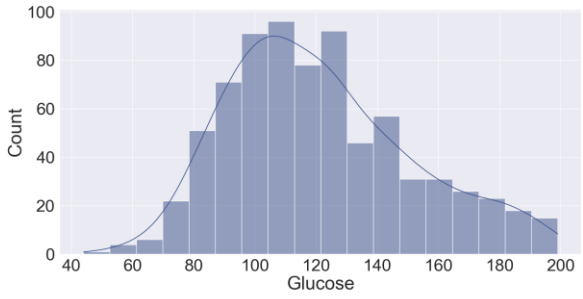
Se observa que en “Insulin, SkinThickness, BloodPressure, BMI y Glucose” los ceros que aparecen no hacen sentido a la distribución que siguen, también se observa que hay algunos outliers en el conjunto. Cambiaremos los ceros que no se ajustan a la distribución por nans y removeremos outliers

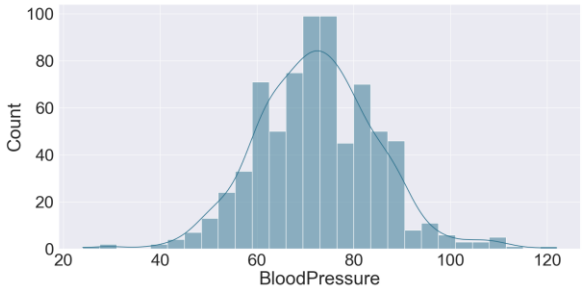
```
columns = ['Glucose', 'BloodPressure', 'SkinThickness','Insulin', 'BMI']
data[columns] = data[columns].replace(0, np.nan)
```

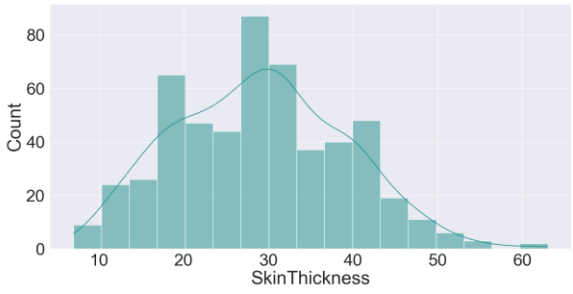
```
def remove_outliers_std(df, col, threshold=4):
    mean = df[col].mean()
    std = df[col].std()

    upper_limit = mean + threshold * std

    outliers = df[(df[col] > upper_limit)].index

    df = df.drop(outliers)

    return df

for col in ['Insulin', 'Pregnancies', 'SkinThickness']:
    data = remove_outliers_std(data, col)
```

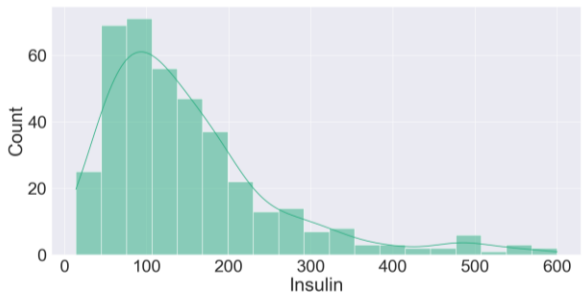
Ahora revisemos las distribuciones

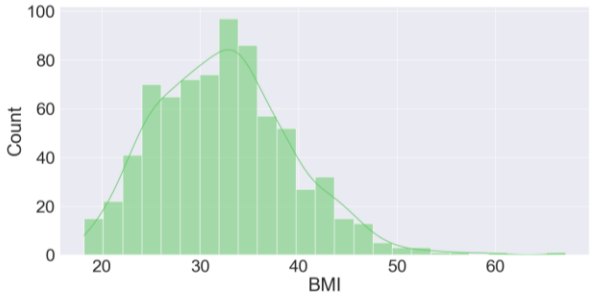


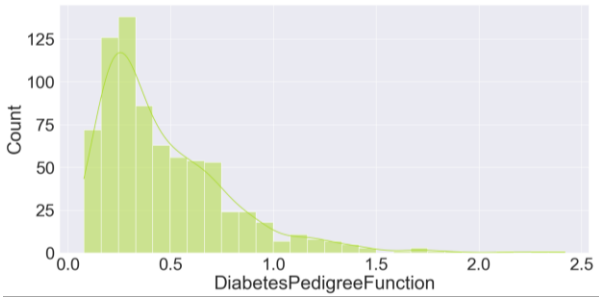



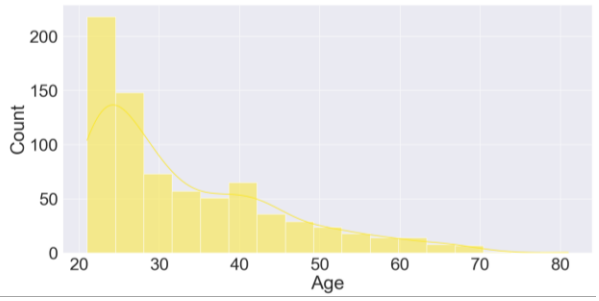




Count
Insulin
0
20
40
60
0
100
200
300
400
500
600

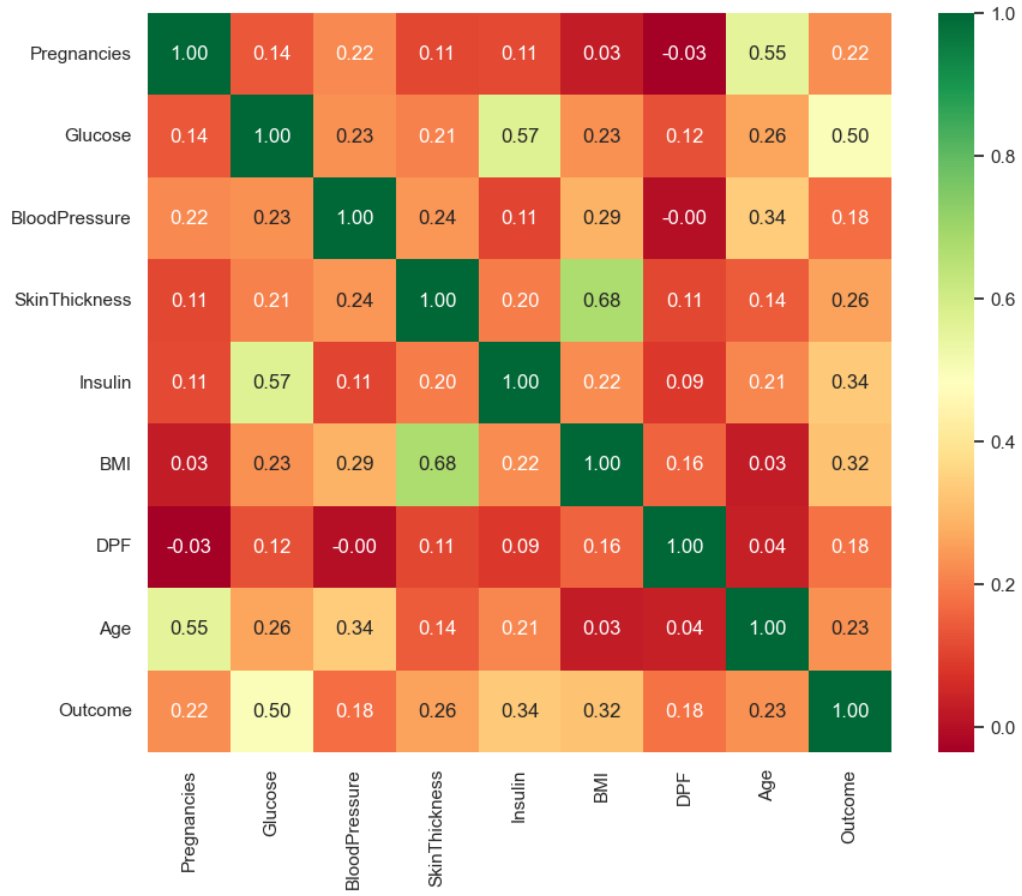

Count
BMI
0
20
40
60
80
100
20
30
40
50
60


Count
DiabetesPedigreeFunction
0
25
50
75
100
125
0.0
0.5
1.0
1.5
2.0
2.5


Count
Age
0
50
100
150
200
20
30
40
50
60
70
80

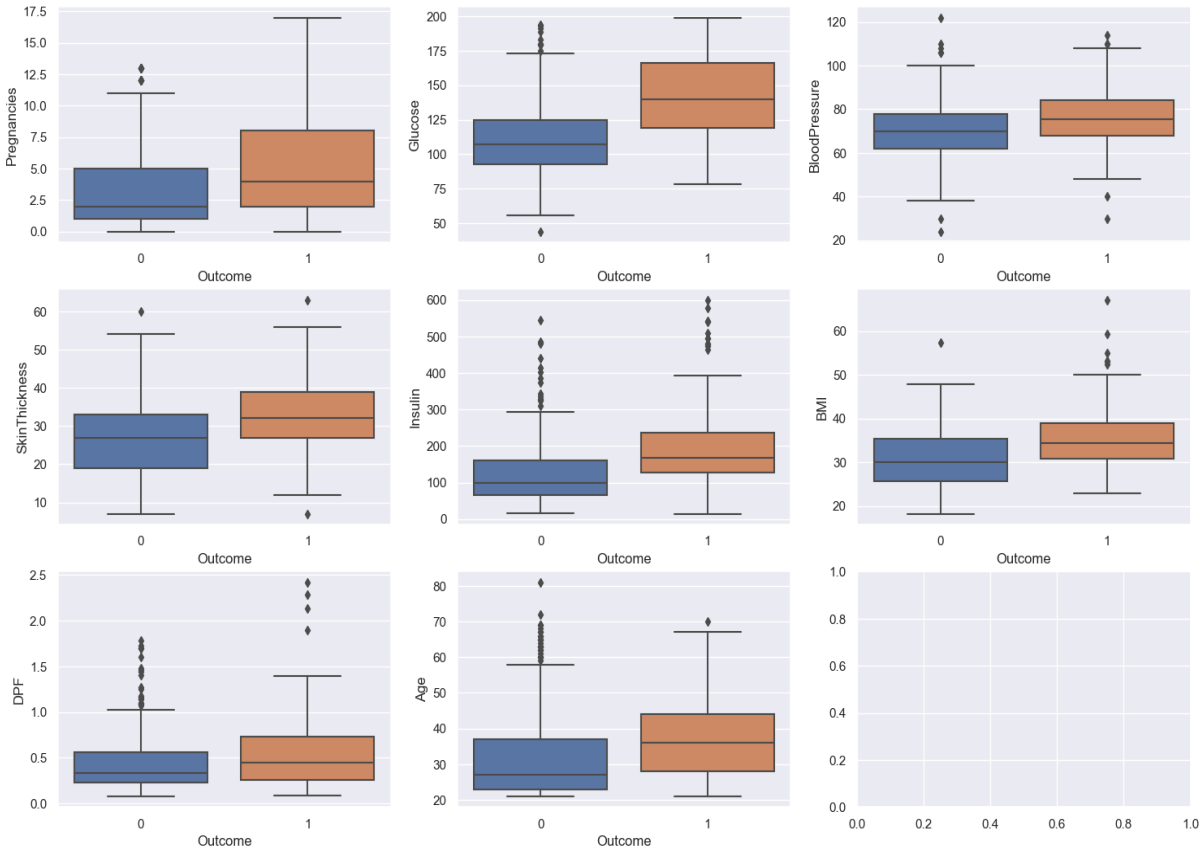
## 3.3 Correlación

Analicemos las correlaciones entre las variables




Observemos la diferencia del reparto de los datos por clase

```
features = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
    'Insulin', 'BMI', 'DPF', 'Age', 'Outcome']
fig, ax = plt.subplots(round(len(features) / 3), 3, figsize = (18, 12))

for i, ax in enumerate(fig.axes):
    if i < len(features) - 1:
        sns.boxplot(y=features[i], x='Outcome', data=data[features], ax=ax)
```

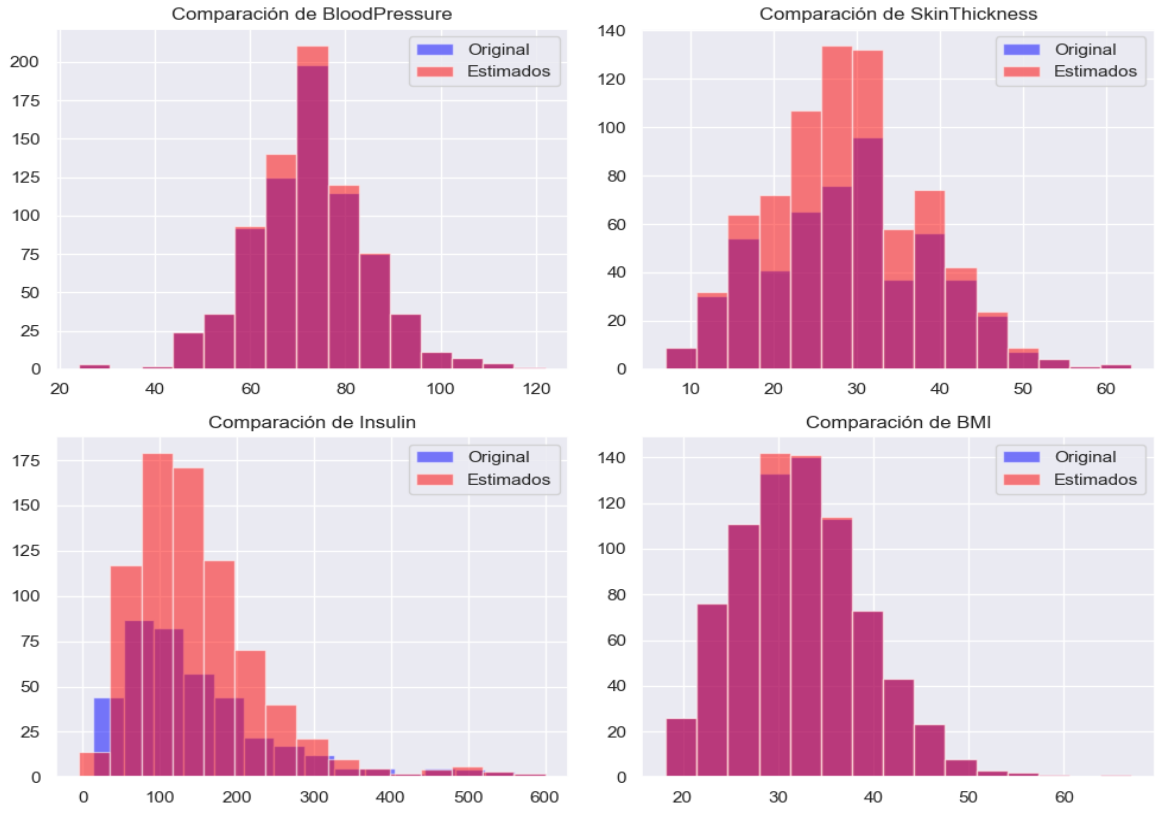



Para la clase 1 todas las categorías tienen un valor mayor como lo indica la matriz de correlación al ser todas correlaciones positivas.

## 3.4 Imputer, análisis e importancia

Hay bastante correlación entre la variable “Insulin” y la variable objetivo, también tiene muchos datos nans así que eliminarlos nos dejaría con muy pocos datos para hacer el entrenamiento, por lo que utilizaremos un imputer para reemplazar los nans. Observemos como son la antigua y nueva distribución de los datos

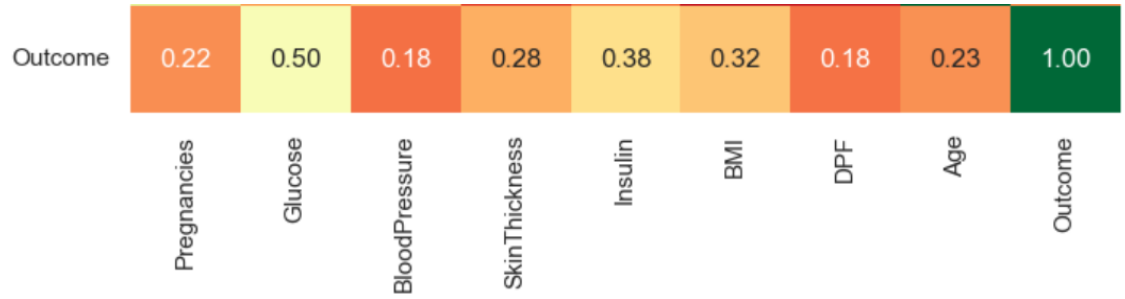
```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

imp = IterativeImputer()
data_imp = imp.fit_transform(data)
data_imp = pd.DataFrame(data_imp, columns=data.columns)

fig, axes = plt.subplots(nrows=len(data.columns), ncols=1, figsize=(8, 20))

for i, col in enumerate(data.columns):
    axes[i].hist(data[col].dropna(), bins=15, alpha=0.5, label='Original', color='blue')

    axes[i].hist(data_imp[col], bins=15, alpha=0.5, label='Estimados', color='red')

    axes[i].set_title(f'Comparación de {col}')
    axes[i].legend()

plt.tight_layout()
plt.show()
```



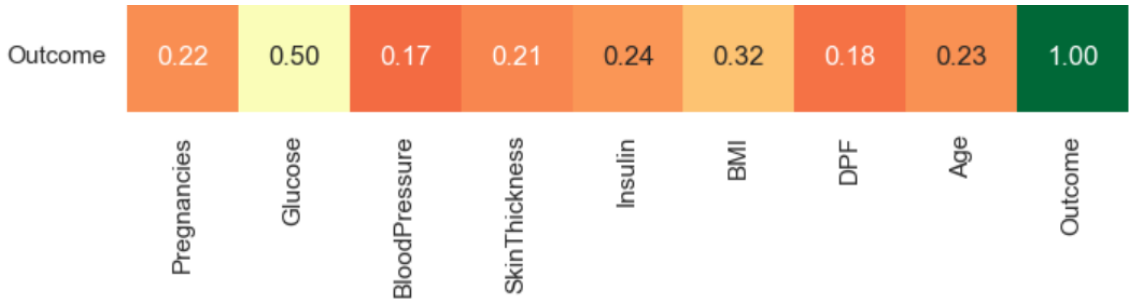
El imputer sigue la distribución de los datos.Ahora los rellenaremos con el promedio de cada variable

```
data_mean = data.fillna(data.mean())
```

Inspeccionemos las correlaciones con la variable objetivo de cada datos imputados, para “Iterative Imputer”




para el promedio

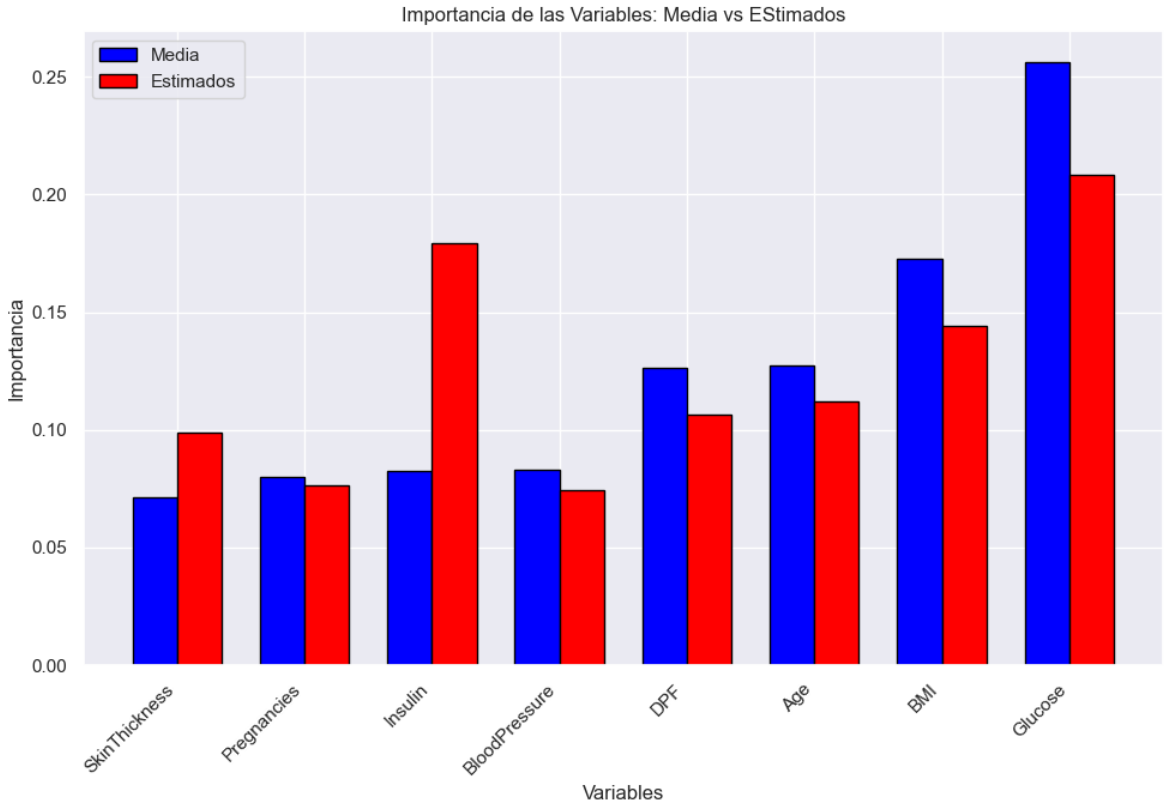



Hay una reducción significativa en los datos rellenados con el promedio en las variables donde hay mas nans, en los datos imputados se mantiene la correlación.

Por ultimo en esta sección analicemos la importancia de cada categoría con un RandomForesst

```
from sklearn.ensemble import RandomForestClassifier

X_imp = data_imp.drop('Outcome', axis=1)
y_imp = data_imp['Outcome']

X_mean = data_mean.drop('Outcome', axis=1)
y_mean = data_mean['Outcome']

model = RandomForestClassifier()
model.fit(X_imp, y_imp)

importances_imp = model.feature_importances_

model = RandomForestClassifier()
model.fit(X_mean, y_mean)
```

```
indices = np.argsort(importances_mean)

plt.figure(figsize=(10, 7))

bar_width = 0.35
r1 = np.arange(X_imp.shape[1])
r2 = r1 + bar_width

plt.bar(r1, importances_mean[indices], color='blue', width=bar_width, edgecolor='black', label='Media')

plt.bar(r2, importances_imp[indices], color='red', width=bar_width, edgecolor='black', label='Estimados')

plt.xlabel('Variables')
plt.ylabel('Importancia')
plt.title('Importancia de las Variables: Media vs EStimados')
plt.xticks(r1 + bar_width / 2, X_imp.columns[indices], rotation=45, ha='right')
plt.legend()

plt.tight_layout()
plt.show()
```




En las variables “Insulin y SkinThickness” tienen más correlación que con el promedio lo cual concuerda con las matrices de correlación.

# 4. Modelo

## Librerías importadas

Todo esto se encuentra en el archivo “Model.ipynb”.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer, KNNImputer
from sklearn.model_selection import train_test_split, GridSearchCV, RepeatedStratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (accuracy_score,
    confusion_matrix,
    f1_score,
    recall_score,
    precision_score)
from sklearn.model_selection import KFold, cross_val_score
from sklearn.ensemble import RandomForestClassifier
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import GradientBoostingClassifier
from xgboost import XGBClassifier
from collections import Counter
```

## 4. 1 Funciones utilizadas

**4.1.1 Training**: Esta función divide los datos en entrenamiento y prueba, luego usa el "IterativeImputer" y "StandardScaler" para rellenar los nans y escalar los datos, el score para decidir el mejor modelo es "f1", entrenamos el modelo con los parámetros que deseemos, este hace un "RepeatedStratifiedKFold" y la función nos devuelve un dataframe con los resultados de todos los entrenamientos (incluido el "train score"), el mejor modelo, los datos de prueba y las diferentes métricas del mejor modelo.

```
def training(model, data, param_grid):
    X = data.drop('Outcome', axis=1)
    y = data['Outcome']

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, stratify=y, random_state= None)

    imputer = IterativeImputer()
    X_train = imputer.fit_transform(X_train)
    X_test = imputer.transform(X_test)

    # smote = SMOTE()
    # X_train, y_train = smote.fit_resample(X_train, y_train)

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    scoring = {
        'accuracy': 'accuracy',
        'f1': 'f1',
        'precision': 'precision',
        'recall': 'recall'
    }
    cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=42)

    grid_search = GridSearchCV(model, param_grid=param_grid, cv=cv, n_jobs=-1,
                               return_train_score=True, scoring=scoring, refit = 'f1')

    grid_search.fit(X_train, y_train)

    cv_results = pd.DataFrame(grid_search.cv_results_)

    print('Los mejores parámetros son:', grid_search.best_params_)
    print('El score de entrenamiento es:', cv_results['mean_train_f1'].iloc[grid_search.best_index_])
    print('El mejor score es:', grid_search.best_score_)

    metrics_df = pd.DataFrame({
    'Mean Test Accuracy': [cv_results['mean_test_accuracy'].iloc[grid_search.best_index_]],
    'Mean Test F1': [cv_results['mean_test_f1'].iloc[grid_search.best_index_]],
    'Mean Test Precision': [cv_results['mean_test_precision'].iloc[grid_search.best_index_]],
    'Mean Test Recall': [cv_results['mean_test_recall'].iloc[grid_search.best_index_]]
    }, index=[model.__class__.__name__])

    return cv_results, grid_search, X_test, y_test, metrics_df
```

**4.1.2 Param_heat:** Nos muestra como varían los resultados del conjunto de entrenamiento y el conjunto de validación en un mapa de calor respecto a dos parámetros que elijamos

```python
def param_heat(results, param_grid, param_x, param_y):

    param_x_r = param_grid[param_x]
    param_y_r = param_grid[param_y]
    scores = results['mean_test_f1'].values.reshape(len(param_y_r), len(param_x_r))

    plt.figure(figsize=(8, 6))
    sns.heatmap(scores, annot=True, xticklabels=param_x_r, yticklabels=param_y_r, cmap="viridis")

    plt.xlabel(str(param_x))
    plt.ylabel(str(param_y))
    plt.title('Heatmap de Resultados')

    plt.show()
```

**4.1.3 Conf_matrix:** Nos devuelve los resultados del mejor modelo con cada métrica y una matriz de confusión de los datos de prueba

```
def conf_matrix(y_test, y_pred):

    recall = recall_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    accuracy = accuracy_score(y_test, y_pred)

    print(f"Recall: {recall:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"Accuracy: {accuracy:.4f}")

    cm = confusion_matrix(y_test, y_pred)

    plt.figure(figsize=(6,6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=['Clase 0', 'Clase 1'], yticklabels=['Clase 0', 'Clase 1'])
    plt.xlabel('Predicción')
    plt.ylabel('Real')
    plt.title('Matriz de Confusión')
    plt.show()
```

**4.1.4 Robust_model_evaluation:** Es parecida a la función “training” solo que divide los datos en varios conjuntos de entrenamiento y prueba (en este caso en 5 distintos), para luego entrenar el modelo en cada uno de ellos, nos devuelve un dataframe con todos los resultados de cada conjunto

```
def robust_model_evaluation(model, data, param_grid):
    X = data.drop('Outcome', axis=1)
    y = data['Outcome']
    random_states = [42, 21, 84, 0, 12]
    all_results = []

    for seed in random_states:
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=seed)
        imputer = IterativeImputer()
        X_train = imputer.fit_transform(X_train)
        X_test = imputer.transform(X_test)

        # smote = SMOTE()
        # X_train, y_train = smote.fit_resample(X_train, y_train)

        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        scoring = {
        'accuracy': 'accuracy',
        'f1': 'f1',
        'precision': 'precision',
        'recall': 'recall'
        }
        cv = 5

        grid_search = GridSearchCV(model, param_grid=param_grid, cv=cv, n_jobs=-1,
                                   return_train_score=True, scoring=scoring, refit = 'f1')

        grid_search.fit(X_train, y_train)

        validation_results = {
            'random_state': seed,
            'mean_test_accuracy': grid_search.cv_results_['mean_test_accuracy'][grid_search.best_index_],
            'mean_test_f1': grid_search.cv_results_['mean_test_f1'][grid_search.best_index_],
            'mean_test_precision': grid_search.cv_results_['mean_test_precision'][grid_search.best_index_],
            'mean_test_recall': grid_search.cv_results_['mean_test_recall'][grid_search.best_index_]
        }

        test_scores = {
            'test_accuracy': accuracy_score(y_test, grid_search.predict(X_test)),
            'test_f1': f1_score(y_test, grid_search.predict(X_test)),
            'test_precision': precision_score(y_test, grid_search.predict(X_test)),
            'test_recall': recall_score(y_test, grid_search.predict(X_test))
        }
        combined_results = {**validation_results, **test_scores}
        all_results.append(combined_results)

    results_df = pd.DataFrame(all_results)

    return results_df
```

**4.1.5 Collect_model_averages:** Junta todos los resultados de cada modelo y cada división de datos, lo junta en un dataframe dando el promedio y la desviación estándar

```
def collect_model_averages(model_dfs, model_names):
    results_list = []

    for model_name, model_df in zip(model_names, model_dfs):
        mean_results = model_df.drop('random_state', axis=1).mean()
        std_results = model_df.drop('random_state', axis=1).std()

        combined_results = pd.DataFrame({
            'valid accuracy': [f"{mean_results['mean_test_accuracy']:.3f} ± {std_results['mean_test_accuracy']:.3f}"],
            'test accuracy': [f"{mean_results['test_accuracy']:.3f} ± {std_results['test_accuracy']:.3f}"],
            'valid f1': [f"{mean_results['mean_test_f1']:.3f} ± {std_results['mean_test_f1']:.3f}"],
            'test f1': [f"{mean_results['test_f1']:.3f} ± {std_results['test_f1']:.3f}"],
            'valid recall': [f"{mean_results['mean_test_recall']:.3f} ± {std_results['mean_test_recall']:.3f}"],
            'test recall': [f"{mean_results['test_recall']:.3f} ± {std_results['test_recall']:.3f}"],
            'valid precision': [f"{mean_results['mean_test_precision']:.3f} ± {std_results['mean_test_precision']:.3f}"],
            'test precision': [f"{mean_results['test_precision']:.3f} ± {std_results['test_precision']:.3f}"],


        })

        combined_results.index = [model_name]
        results_list.append(combined_results)

    results_summary = pd.concat(results_list)

    return results_summary
```

## 4.2 Entrenamiento simple

Haremos un entrenamiento con la función “Training” para cada modelo y veremos el resultado en cada uno.

### 4.2.1 KNeighbors

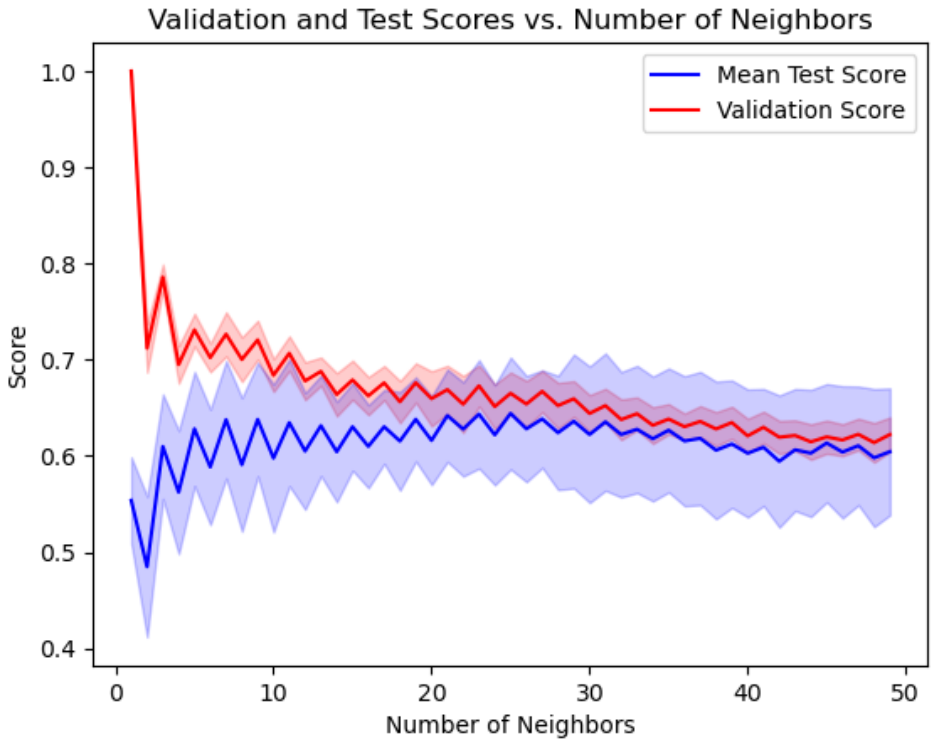
```
knn_params = {'n_neighbors' : range(1, 50)}

knn_results, knn_grid, X_test, y_test, knn_metrics = training(KNeighborsClassifier(), data, knn_params)
```

✓ 20.8s

```
Los mejores parámetros son: {'n_neighbors': 25}
El score de entrenamiento es: 0.66515079973441
El mejor score es: 0.6443047505143138
```

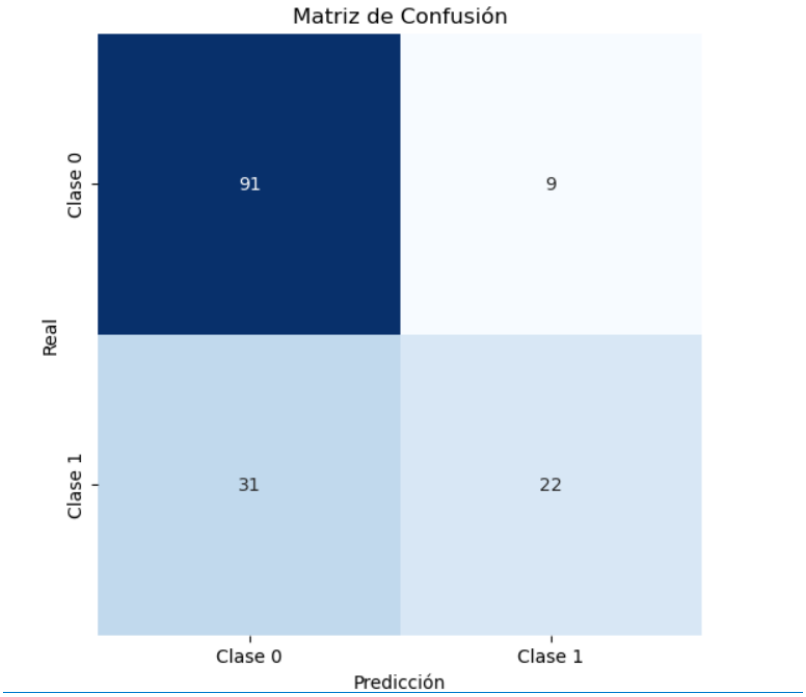
Grafica de como varía los resultados según el numero de vecinos

```
param = 'param_n_neighbors'
score_p = ['mean_test_f1', 'mean_train_f1']
std_p = ['std_test_f1', 'std_train_f1']
knn_results[param] = knn_results[param].astype(float)
fig, ax = plt.subplots()

ax.plot(knn_results[param],
        knn_results[score_p[0]],
        label='Mean Test Score',
        color = 'blue')

ax.plot(knn_results[param],
        knn_results[score_p[1]],
        label='Validation Score',
        color='red')

ax.fill_between(knn_results[param],
                (knn_results[score_p[1]] - knn_results[std_p[1]]),
                (knn_results[score_p[1]] + knn_results[std_p[1]]),
                color='red', alpha=0.2)

ax.fill_between(knn_results[param],
                (knn_results[score_p[0]] - knn_results[std_p[0]]),
                (knn_results[score_p[0]] + knn_results[std_p[0]]),
                color='blue', alpha=0.2)

ax.set_xlabel('Number of Neighbors')
ax.set_ylabel('Score')
ax.set_title('Validation and Test Scores vs. Number of Neighbors')

ax.legend()

plt.show()
```




Resultados en el conjunto de prueba

```
Recall: 0.4151
Precision: 0.7097
F1 Score: 0.5238
Accuracy: 0.7386
```



### 4.2.2 RandomForest

```
rf_param = {
    'n_estimators': [50, 70, 90, 100, 200],
    'max_depth': range(1,9)
}

rf_results, rf, X_test_rf, y_test_rf, rf_metrics = training(RandomForestClassifier(), data, rf_param)
```

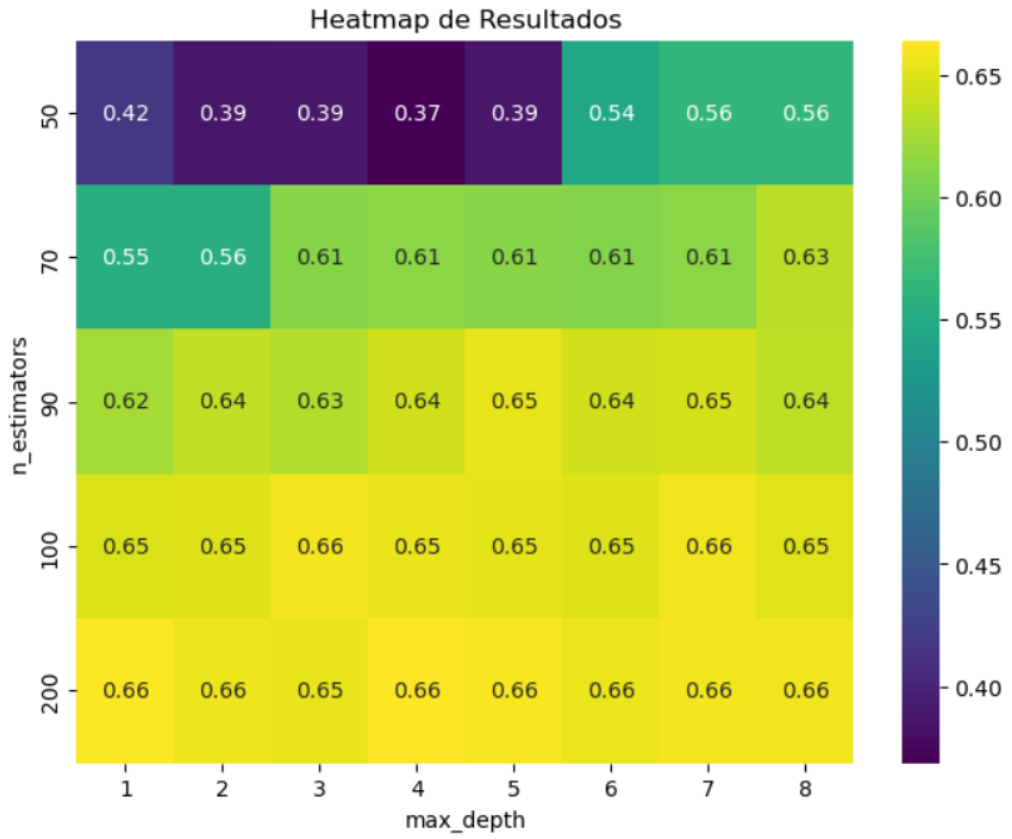
✓ 57.3s

```
Los mejores parámetros son: {'max_depth': 8, 'n_estimators': 50}
El score de entrenamiento es: 0.9670303421868216
El mejor score es: 0.6641994623954344
```

Resultados según el numero de arboles y profundidad

```
param_heat(rf_results, rf_param, 'max_depth', 'n_estimators')
```

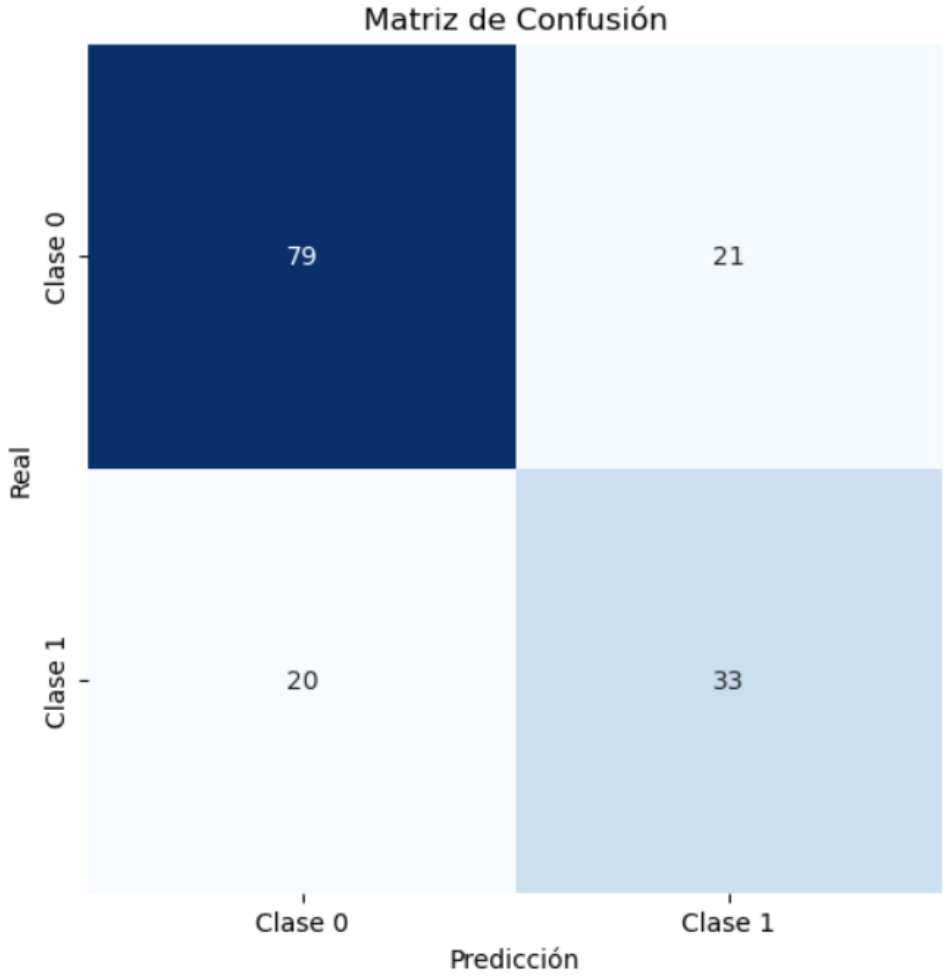
✓ 0.3s




Resultados en el conjunto de prueba

```
y_pred = rf.predict(X_test_rf)
conf_matrix(y_test_rf, y_pred)
```

Recall: 0.6226
Precision: 0.6111
F1 Score: 0.6168
Accuracy: 0.7320



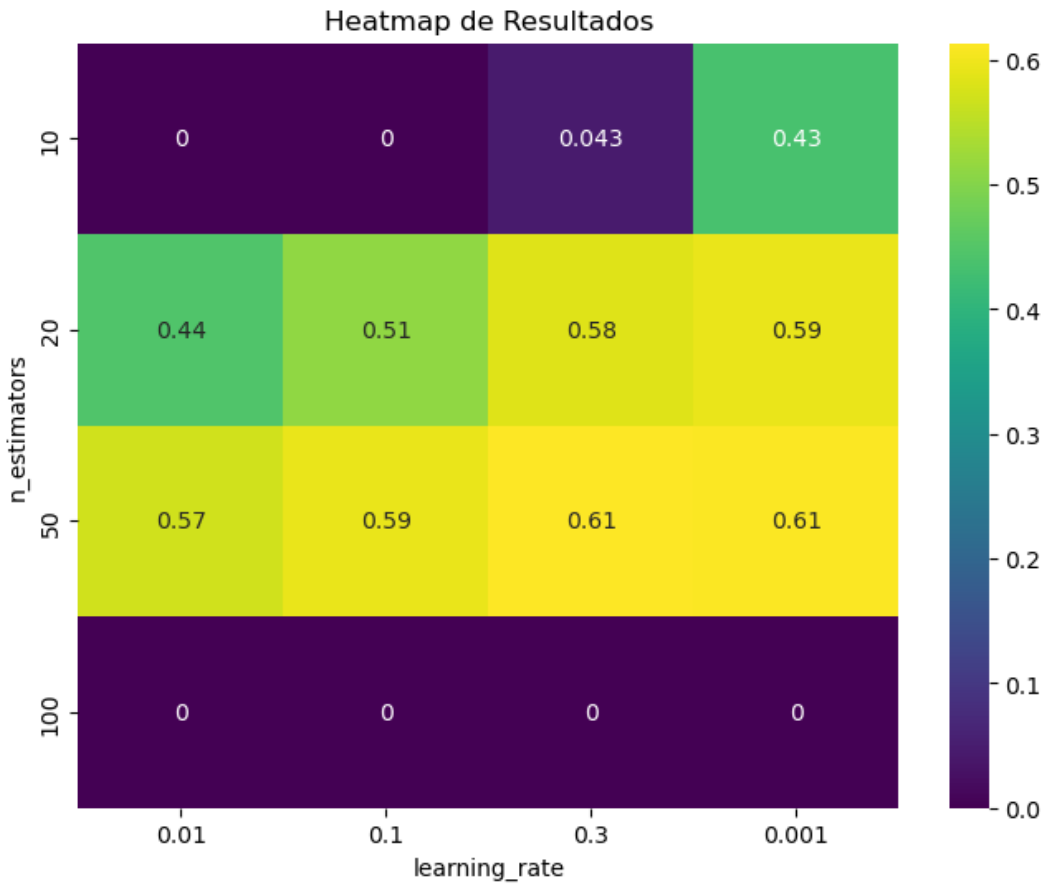
### 4.2.3 GradientBoost

```
gb_param = {
    'n_estimators': [10, 20, 50, 100],
    'max_depth': range(1,5),
    'learning_rate': [0.01, 0.1, 0.3, 0.001]#,
    #'subsample': [0.7, 1]
}
gb_results, gb, X_test, y_test, gb_metrics = training(GradientBoostingClassifier(), data, gb_param)
```

✓ 33.3s

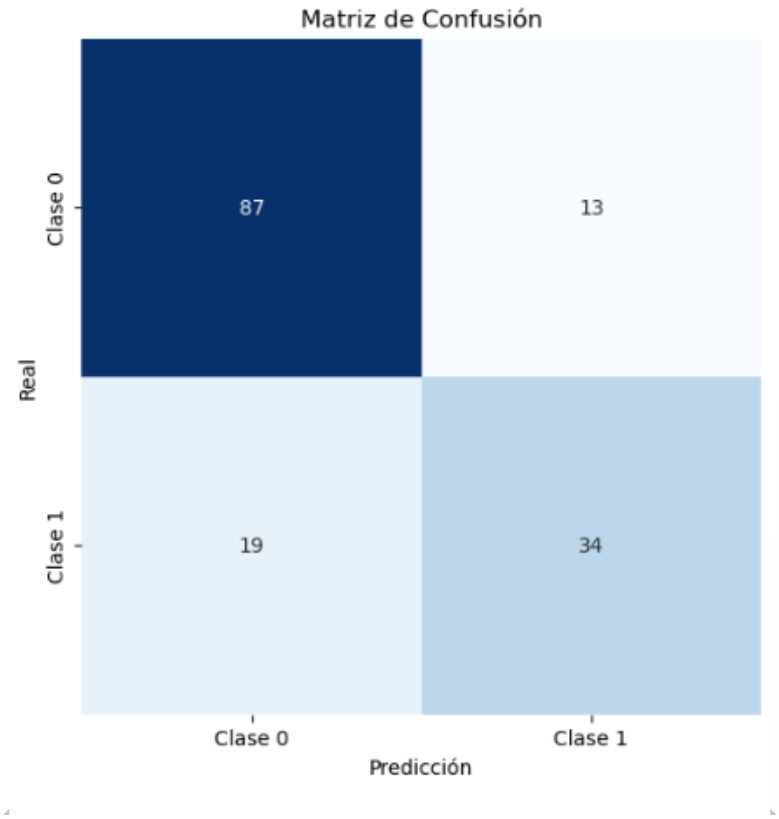
```
Los mejores parámetros son: {'learning_rate': 0.3, 'max_depth': 3, 'n_estimators': 10}
El score de entrenamiento es: 0.7960053908584839
El mejor score es: 0.6207500928982065
```

Se hizo un mapa de calor para cada profundidad (solo mostraremos el de max_depth = 1)




Resultados en conjunto de prueba

Recall: 0.6415
Precision: 0.7234
F1 Score: 0.6800
Accuracy: 0.7908



### 4.2.4 XGBoost

```
xgb_params = {'max_depth' : range(2, 5),
              'learning_rate' : [0.3, 0.1, 0.03, 0.01],
              # 'subsample':[0.5, 0.6],
              # 'colsample_bytree': [0.5, 1],
              # 'reg_alpha': [0, 0.1, 1, 10],
              # 'reg_lambda': [0, 0.1, 1, 10],
              # 'gamma': [0, 0.1, 1, 10],
              'scale_pos_weight' : [1, 1.5, 2, 3]
    }

xgb_results, xgb, X_test, y_test, xgb_metrics = training(XGBClassifier(), data, xgb_params)
```

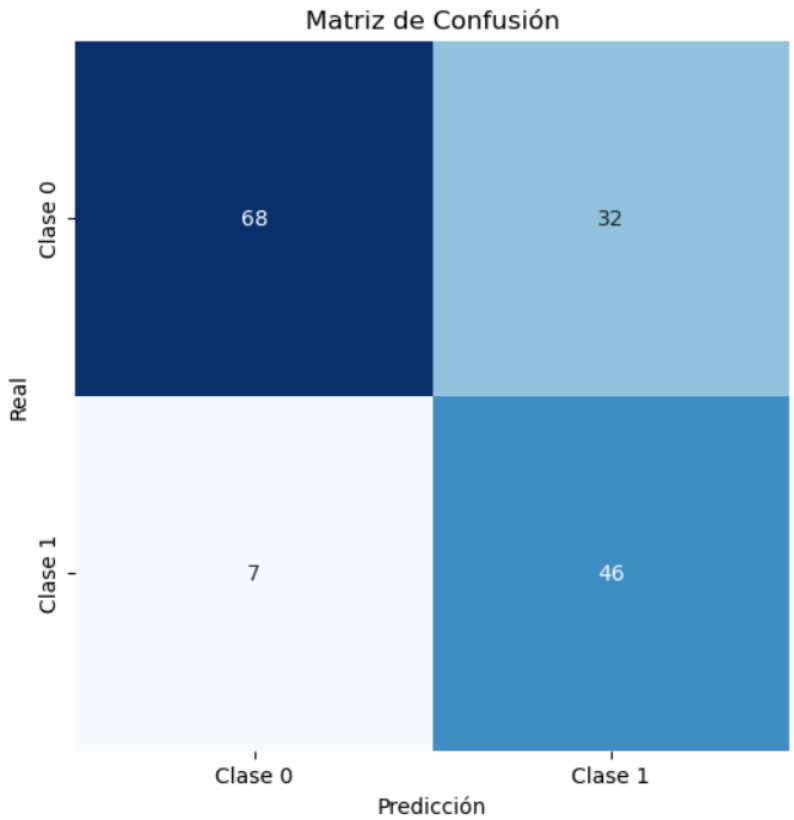
✓ 15.6s

```
Los mejores parámetros son: {'learning_rate': 0.03, 'max_depth': 2, 'scale_pos_weight': 2}
El score de entrenamiento es: 0.7475018996015518
El mejor score es: 0.6860206620248103
```

Resultados conjunto de prueba

```
y_pred = xgb.best_estimator_.predict(X_test)
conf_matrix(y_test, y_pred)
```

```
Recall: 0.8679
Precision: 0.5897
F1 Score: 0.7023
Accuracy: 0.7451
```



## 4.3 Resultados del entrenamiento simple

Ahora veamos el rendimiento de cada algoritmo (en el conjunto de validación)

```
metrics = pd.concat([knn_metrics,rf_metrics, gb_metrics, xgb_metrics])
metrics
```

✓ 0.0s

| | Mean Test Accuracy | Mean Test F1 | Mean Test Precision | Mean Test Recall |
|---|---|---|---|---|
| KNeighborsClassifier | 0.774184 | 0.644305 | 0.716827 | 0.591399 |
| RandomForestClassifier | 0.779626 | 0.664199 | 0.715322 | 0.627132 |
| GradientBoostingClassifier | 0.753987 | 0.620750 | 0.673169 | 0.579144 |
| XGBClassifier | 0.744649 | 0.686021 | 0.601549 | 0.801107 |

XGBoost presenta el mejor valor de f1, a pesar de tener el peor accuracy. Notemos que también tiene un recall muy alto comparado con los otros algoritmos, esta es una métrica importante ya que es más valioso estar seguro de que alguien que tiene diabetes de positivo a que alguien que no tenga de negativo, es decir es más importante reducir los falsos negativos que los falsos positivos.

Algo que podríamos hacer para mejorar el recall, es balancear los datos con un SMOTE, esta función esta comentada en las funciones “Training” y “Robust_model_evaluation” ya que tras probarla da una varianza en los resultados mayor y sobreentrena los modelos. Así que se puede usar pero en este análisis preferimos no hacerlo.

## 4.4 Resultado general

Al tener muy pocos datos, una varianza muy grande y muchos nans aunque la muestra esté estratificada los resultados en todos los modelos varía significativamente, por lo que usaremos la función de “Robust_model_evaluation” para poder generalizar mejor el rendimiento de cada algoritmo (ignoraremos la precisión ya que no es muy relevante en este conjunto de datos y accuracy, f1 y recall son métricas más útiles)

```
Total_xgb = robust_model_evaluation(XGBClassifier(), data, xgb_params)
Total_knn = robust_model_evaluation(KNeighborsClassifier(), data, knn_params)
Total_rf = robust_model_evaluation(RandomForestClassifier(), data, rf_param)
Total_gb = robust_model_evaluation(GradientBoostingClassifier(), data, gb_param)
model_dfs = [Total_xgb, Total_rf, Total_knn, Total_gb]
model_names = ['XGBoost', 'RandomForest', 'KNeighbors', 'GradientBoosting']

results_summary = collect_model_averages(model_dfs, model_names)
results_summary
```

| | valid accuracy | test accuracy | valid f1 | test f1 | valid recall | test recall |
|---|---|---|---|---|---|---|
| XGBoost | 0.765 ± 0.007 | 0.756 ± 0.031 | 0.702 ± 0.010 | 0.692 ± 0.039 | 0.794 ± 0.019 | 0.796 ± 0.090 |
| RandomForest | 0.773 ± 0.006 | 0.745 ± 0.017 | 0.647 ± 0.010 | 0.602 ± 0.061 | 0.601 ± 0.012 | 0.566 ± 0.105 |
| KNeighbors | 0.772 ± 0.007 | 0.740 ± 0.016 | 0.638 ± 0.015 | 0.595 ± 0.041 | 0.581 ± 0.027 | 0.558 ± 0.092 |
| GradientBoosting | 0.769 ± 0.010 | 0.748 ± 0.026 | 0.648 ± 0.015 | 0.614 ± 0.065 | 0.612 ± 0.020 | 0.589 ± 0.109 |

En general, en el conjunto de prueba hay más varianza que en el de validación. En el recall hay una varianza muy grande hasta del 10.9%, lo cual coincide con que el desempeño de los algoritmos depende significativamente de como se distribuyan los datos. El mejor modelo en las métricas f1 y recall es XGBoost que son las métricas más importantes en nuestro estudio.

### 4.4.1 Resultado sin “Insulin”

La categoría con más nans y con mayor varianza es “Insulin”, es interesante ver como afectará quitar esta columna en el rendimiento de los algortimos

```
data_ins = data.drop(['Insulin'], axis = 1)
Total_xgb_i = robust_model_evaluation(XGBClassifier(), data_ins, xgb_params)
Total_knn_i = robust_model_evaluation(KNeighborsClassifier(), data_ins, knn_params)
Total_rf_i = robust_model_evaluation(RandomForestClassifier(), data_ins, rf_param)
Total_gb_i = robust_model_evaluation(GradientBoostingClassifier(), data_ins, gb_param)

model_dfs_i = [Total_xgb_i, Total_rf_i, Total_knn_i, Total_gb_i]

results_summary_i = collect_model_averages(model_dfs_i, model_names)
results_summary_i
```

| | valid accuracy | test accuracy | valid f1 | test f1 | valid recall | test recall |
|---|---|---|---|---|---|---|
| XGBoost | 0.756 ± 0.014 | 0.749 ± 0.015 | 0.698 ± 0.013 | 0.686 ± 0.019 | 0.809 ± 0.018 | 0.792 ± 0.072 |
| RandomForest | 0.777 ± 0.005 | 0.742 ± 0.012 | 0.650 ± 0.010 | 0.590 ± 0.039 | 0.601 ± 0.020 | 0.540 ± 0.083 |
| KNeighbors | 0.775 ± 0.011 | 0.753 ± 0.017 | 0.633 ± 0.018 | 0.604 ± 0.027 | 0.560 ± 0.030 | 0.547 ± 0.074 |
| GradientBoosting | 0.773 ± 0.007 | 0.752 ± 0.015 | 0.643 ± 0.016 | 0.605 ± 0.060 | 0.591 ± 0.024 | 0.558 ± 0.102 |

Se observa que hay un desempeño similar que teniendo la columna pero la varianza de las métricas es ligeramente menor en general.

## 5. Conclusión

En conclusión, por los resultados vistos el algoritmo que mejor rendimiento da en general en este conjunto de datos es XGBoost. Para conseguir un mejor resultado hace falta explorar más hiperparámetros y que tengan un mayo conjunto de selección, o hacer un ensamble de modelos que no es el objetivo de este proyecto.