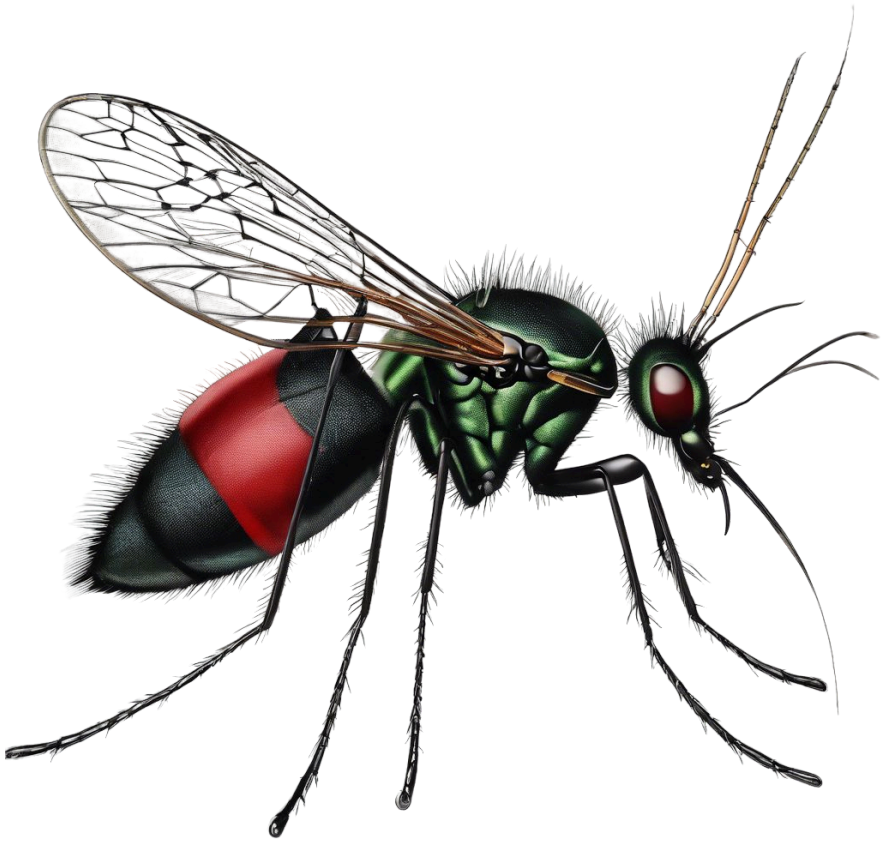
Néstor Batista Díaz

DengAI: predicción de la propagación de enfermedades



ÍNDICE

INTRODUCCIÓN

En este estudio académico, se aborda el desafío de la competición "Predicting Disease Spread" de DrivenData, que busca desarrollar un modelo predictivo eficaz para predecir la incidencia de casos de dengue en diferentes regiones. El objetivo es explorar y comparar varios modelos de aprendizaje automático para identificar el enfoque más efectivo en la predicción de la propagación del dengue. A través de la experimentación y evaluación de diferentes algoritmos, se busca contribuir al desarrollo de herramientas predictivas para mejorar la planificación y respuesta ante brotes de enfermedades como el dengue.

PROCESAMIENTO DE DATOS

Importamos los datasets y los juntamos.

DATASET

```
# Lee los archivos CSV
df1 = pd.read_csv('https://raw.githubusercontent.com/Nestorbd/DengIA-Competition/master/datasets/dengue_features_train.csv')
df2 = pd.read_csv('https://raw.githubusercontent.com/Nestorbd/DengIA-Competition/master/datasets/dengue_labels_train.csv')

# Combina los dos DataFrames usando merge
df_combined = pd.merge(df1, df2, how='outer', on=['city', 'year', 'weekofyear'])

# Guarda el DataFrame combinado en un nuevo archivo CSV
df_combined.to_csv('dengue_train.csv', index=False)
df_combined.shape
```

✓ 0.9s

```
(1456, 25)
```

Transformamos los parámetros de texto en números.

```
df_combined['city'].replace(['iq', 'sj'],[0, 1], inplace=True)

df_combined['week_start_date'] = pd.to_datetime(df_combined['week_start_date'])
df_combined['week_start_date'] = (df_combined['week_start_date'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1D')
```

✓ 0.6s

VISUALIZACIÓN DE DATOS

```
df_combined.describe()
```
✓ 0.6s — Python

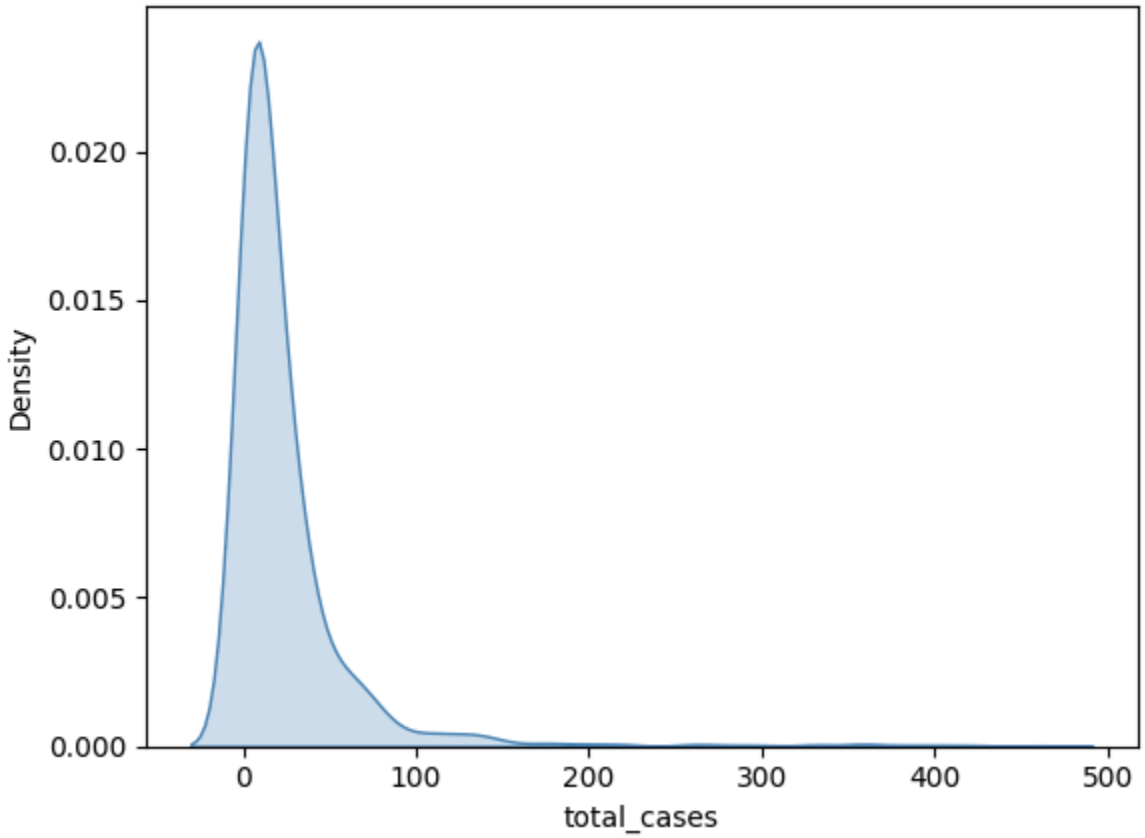
	city	year	weekofyear	week_start_date	ndvi_ne	ndvi_nw	ndvi_se	ndvi_sw	precipitation_amt_mm	reanalysis_air_temp_k	...	reanalysis_relative_humidity_percent	reanalysis_sat_precip_
count	1456.000000	1456.000000	1456.000000	1456.000000	1262.000000	1404.000000	1434.000000	1434.000000	1443.000000	1446.000000	...	1446.000000	144
mean	0.642857	2001.031593	26.503434	11512.667582	0.142294	0.130553	0.203783	0.202305	45.760388	298.701852	...	82.161959	4
std	0.479322	5.408314	15.019437	1970.417074	0.140531	0.119999	0.073860	0.083903	43.715537	1.362420	...	7.153897	4
min	0.000000	1990.000000	1.000000	7424.000000	-0.406250	-0.456100	-0.015533	-0.063457	0.000000	294.635714	...	57.787143	
25%	0.000000	1997.000000	13.750000	9979.250000	0.044950	0.049217	0.155087	0.144209	9.800000	297.658929	...	77.177143	
50%	1.000000	2002.000000	26.500000	11835.000000	0.128817	0.121429	0.196050	0.189450	38.340000	298.646429	...	80.301429	3
75%	1.000000	2005.000000	39.250000	13113.000000	0.248483	0.216600	0.248846	0.246982	70.235000	299.833571	...	86.357857	7
max	1.000000	2010.000000	53.000000	14785.000000	0.508357	0.454429	0.538314	0.546017	390.600000	302.200000	...	98.610000	39

8 rows × 25 columns

```
df_combined.info()
```
✓ 0.7s

```
Output exceeds the size limit. Open the full output data in a text editor
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1456 entries, 0 to 1455
Data columns (total 25 columns):
 #   Column                                Non-Null Count  Dtype
---  ------                                --------------  -----
 0   city                                  1456 non-null   int64
 1   year                                  1456 non-null   int64
 2   weekofyear                            1456 non-null   int64
 3   week_start_date                       1456 non-null   int64
 4   ndvi_ne                               1262 non-null   float64
 5   ndvi_nw                               1404 non-null   float64
 6   ndvi_se                               1434 non-null   float64
 7   ndvi_sw                               1434 non-null   float64
 8   precipitation_amt_mm                  1443 non-null   float64
 9   reanalysis_air_temp_k                 1446 non-null   float64
 10  reanalysis_avg_temp_k                 1446 non-null   float64
 11  reanalysis_dew_point_temp_k           1446 non-null   float64
 12  reanalysis_max_air_temp_k             1446 non-null   float64
 13  reanalysis_min_air_temp_k             1446 non-null   float64
 14  reanalysis_precip_amt_kg_per_m2       1446 non-null   float64
 15  reanalysis_relative_humidity_percent  1446 non-null   float64
 16  reanalysis_sat_precip_amt_mm          1443 non-null   float64
 17  reanalysis_specific_humidity_g_per_kg 1446 non-null   float64
 18  reanalysis_tdtr_k                     1446 non-null   float64
 19  station_avg_temp_c                    1413 non-null   float64
```

Aquí podemos comprobar que hay columnas con bastantes nulos, los trataremos más adelante.



Aquí podemos comprobar que la densidad de los datos está entre 0 y 100 casos totales y llega hasta 500. Esto nos indica que puede haber outliers que nos dificultará el entrenamiento.

PROCESAMIENTO DE DATOS

Lo primero es quitar todas las filas nulas, también se podrían modificar por otro dato estadístico, pero en este caso he preferido quitarlas directamente.

QUITAR NULOS

```
df_combined = df_combined.dropna()
```

✓ 0.6s

Eliminamos también los outliers.

```
# Calcular el rango intercuartílico (IQR) para cada columna
Q1 = df_combined.quantile(0.25)
Q3 = df_combined.quantile(0.75)
IQR = Q3 - Q1
xL=Q1 - k * IQR
xU=Q3 + k * IQR

# Definir un filtro para eliminar outliers
filtro_outliers = ~((df_combined < xL) | (df_combined > xU)).any(axis=1)

# Aplicar el filtro para mantener solo los datos sin outliers
df_sin_outliers = df_combined[filtro_outliers].reset_index(drop=True)

# Visualizar la diferencia de filas
outlaiers = df_combined.shape[0]- df_sin_outliers.shape[0]

# Verificar la forma del DataFrame después de eliminar outliers
print(f'Forma del DataFrame antes de eliminar outliers: {df_combined.shape}')
print(f'Forma del DataFrame después de eliminar outliers: {df_sin_outliers.shape}')
print(f'Se han eliminado {outlaiers} outliers')
```

3] ✓ 0.6s

```
Forma del DataFrame antes de eliminar outliers: (1199, 25)
Forma del DataFrame después de eliminar outliers: (979, 25)
Se han eliminado 220 outliers
```

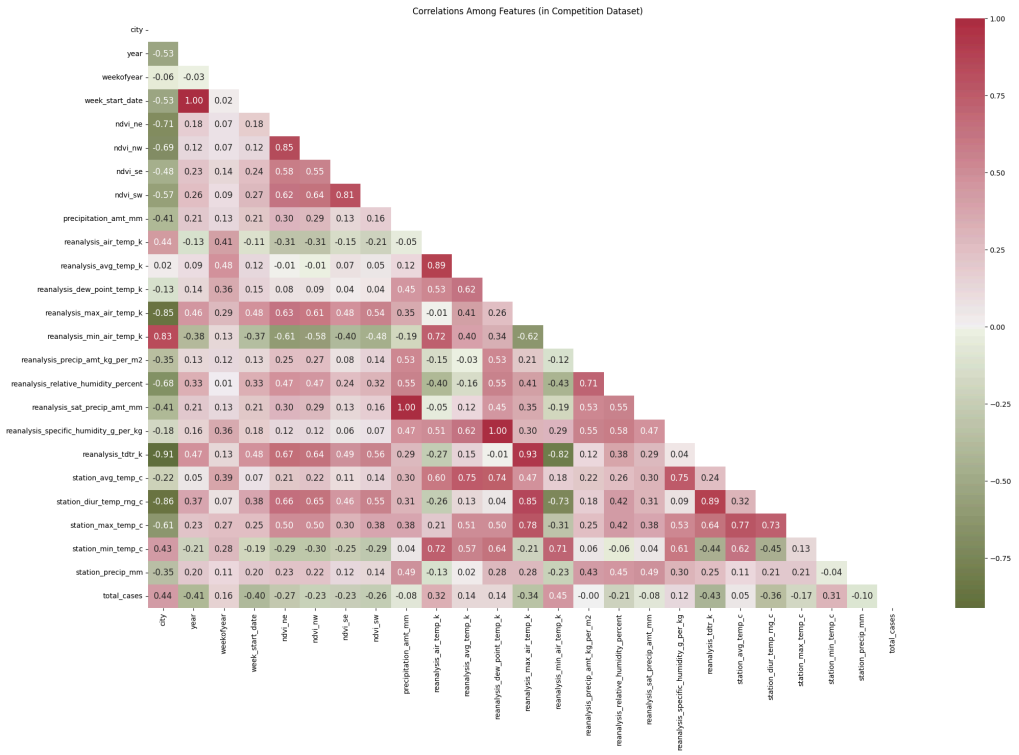
Normalizamos los datos para su posterior entrenamiento.

NORMALIZACIÓN

```
column_names = df_sin_outliers.columns.values
index_to_remove = np.where(column_names == 'total_cases')[0]
column_names = np.delete(column_names, index_to_remove)

scaler = StandardScaler()
scaled_data = scaler.fit_transform(df_sin_outliers.drop('total_cases', axis=1))
scaled_df = pd.DataFrame(scaled_data, columns=column_names)

df_complete = scaled_df.copy()
df_complete["total_cases"] = df_sin_outliers['total_cases']
```

24] ✓ 0.7s

SELECCIÓN DE CARACTERÍSTICAS



Como podemos observar hay muchos datos que se correlacionan entre sí y es muy difícil de determinar a simple vista cuales son los mejores, así que, vamos a usar otra herramienta para seleccionar las mejores características. En este caso usaré **Lasso**.

```
# Inicializar el modelo de regresión LASSO
lasso_model = Lasso(alpha=1.0)

# Entrenar el modelo
lasso_model.fit(X_train, y_train)

# Obtener los coeficientes no nulos (características seleccionadas)
selected_features = X.columns[lasso_model.coef_ != 0]

# Mostrar las características seleccionadas
print("Características seleccionadas:", selected_features)
```

[27] ✓ 0.7s

```
Características seleccionadas: Index(['year', 'weekofyear', 'reanalysis_air_temp_k',
       'reanalysis_min_air_temp_k', 'reanalysis_tdtr_k'],
      dtype='object')
```

Las mejores características son ‘year’, ‘weekofyear’, ‘reanalysis_air_temp_k’, ‘reanalysis_min_air_temp_k’ y ‘reanalysis_tdtr_k’. Si comprobamos estos campos con la matriz de correlación, vemos que tiene sentido que estos parámetros sean los mejores para empezar a entrenar.

Ahora, con estos parámetros dividimos los datos del dataset en train, val y test.

DIVIDIR LOS DATOS

```
X = scaled_df[selected_features]
y = df_sin_outliers['total_cases']

# Dividir los datos 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)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
```

ENTRENAMIENTO

Para este trabajo vamos a usar tres tipos de modelos de entrenamiento y vamos a explorar las diferencias entre seleccionadores de hyperparametros como **GridSearchCV** y **RandomizedSearchCV**.

RANDOM FOREST

RANDOM FOREST

```
# Definir los hiperparámetros a ajustar
params_rf = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10]
}

# Inicializar el modelo Random Forest
rf = RandomForestRegressor()
```

[29] ✓ 0.6s

```
best_mae_rf, best_params_rf, best_technique_rf, best_splits_rf = best_params(model=rf, params=params_rf, CV_technique="all", max_splits=15)
print("Best MAE:",best_mae_rf,", Best params:", best_params_rf,", Best technique:", best_technique_rf,", Best splits:", best_splits_rf)
```

[30] ✓ 8m 7.3s

```
min: 2 max: 13 actual: 3
search: 2 - 8 actual: 7
gridSearchCV Finish
randomizeSearchCV Finish
Best MAE: 7.205214111285539 , Best params: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 300} , Best technique: gridSearchCV , Best splits: 13
```

GridSearchCV

GridSearchCV

+ Código + Markdown

```
# Inicializar GridSearchCV
grid_search_rf = GridSearchCV(estimator=rf, param_grid=params_rf, cv=best_splits_rf, n_jobs=-1, scoring='neg_mean_absolute_error')

# Ajustar el modelo utilizando GridSearchCV
grid_search_rf.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params_grid_search_rf = grid_search_rf.best_params_
best_score_grid_search_rf =  - grid_search_rf.best_score_
print("Mejores hiperparámetros:", best_params_grid_search_rf)
print("MAE: ", best_score_grid_search_rf)
```

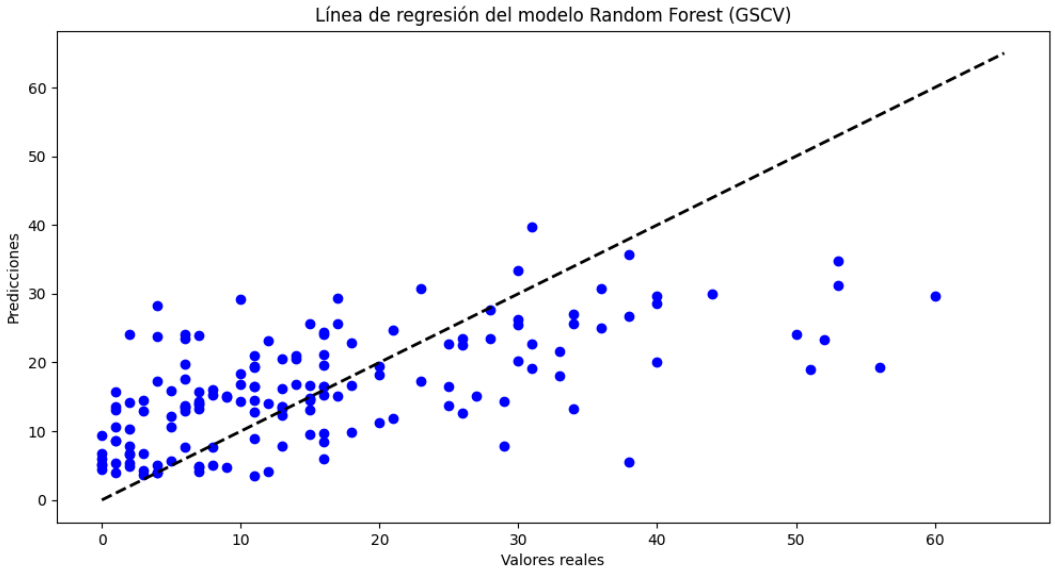
✓ 44.7s

```
Mejores hiperparámetros: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 200}
MAE:  7.211374280481423
```

```
# Realizar la validación cruzada para obtener las predicciones
y_pred_val_gscv_rf = cross_val_predict(grid_search_rf, X_val, y_val, cv=5)

# Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
mae_gscv_rf = mean_absolute_error(y_val, y_pred_val_gscv_rf)

print(f"Mean Absolute Error: {mae_gscv_rf}")
```

32] ✓ 1m 6.5s

```
Mean Absolute Error: 8.159919230849349
```



RandomizedSearchCV

RandomizedSearchCV

```
# Inicializar GridSearchCV
random_search_rf = RandomizedSearchCV(estimator=rf, param_distributions=params_rf, n_iter=10, cv=best_splits_rf, n_jobs=-1, scoring='neg_mean_absolute_error')

# Ajustar el modelo utilizando GridSearchCV
random_search_rf.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params_random_search_rf = random_search_rf.best_params_
best_score_random_search_rf =  - random_search_rf.best_score_
print("Mejores hiperparámetros:", best_params_random_search_rf)
print("MAE: ", best_score_random_search_rf)
```

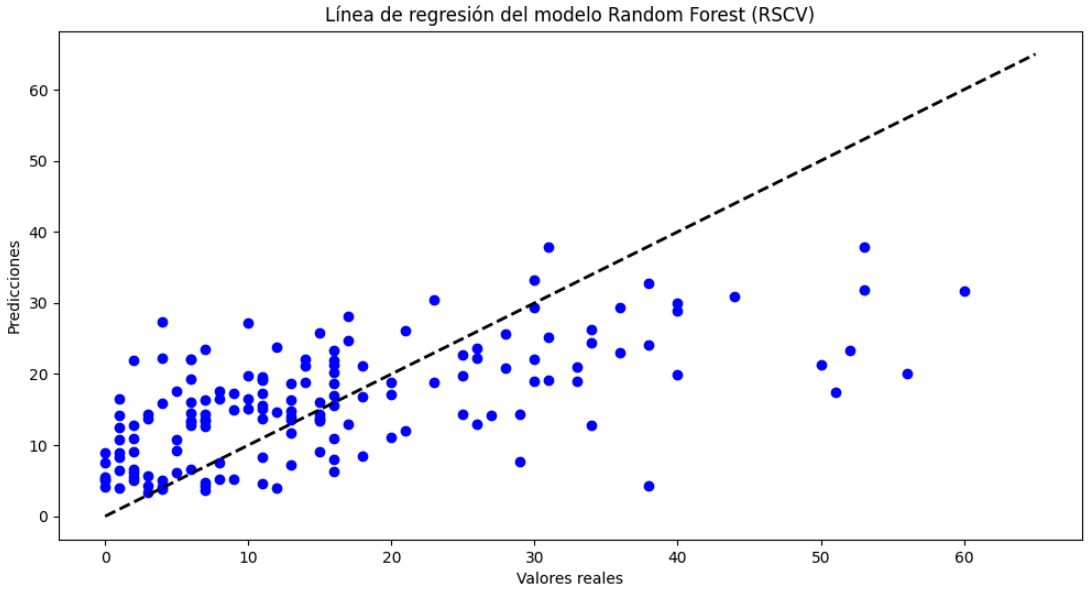
✓ 16.7s

```
Mejores hiperparámetros: {'n_estimators': 100, 'min_samples_split': 2, 'max_depth': None}
MAE:  7.21359497645212
```

```
# Realizar la validación cruzada para obtener las predicciones
y_pred_val_rscv_rf = cross_val_predict(random_search_rf, X_val, y_val, cv=5)

# Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
mae_rscv_rf = mean_absolute_error(y_val, y_pred_val_rscv_rf)

print(f"Mean Absolute Error: {mae_rscv_rf}")
```

[35] ✓ 27.4s

```
Mean Absolute Error: 8.207880936928644
```



CONCLUSIÓN

En este caso, no hay mucha diferencia entre usar **GridSearchCV** y **RandomizedSearchCV**, los dos dan un **MAE** bastante parecido y sus gráficas también son bastante similares. Aunque si

tengo que elegir uno, sería el **GridSearchCV** porque con los datos de validación da mejor resultados (8,16) que **RandomizedSearchCV** (8,20).

GRADIENT BOOSTING

Gradient Boosting

```
# Definir los hiperparámetros a ajustar
params_gb = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.05, 0.1, 0.2]
}

# Inicializar el modelo Gradient Boosting Regressor
gb = GradientBoostingRegressor()
```

[37] ✓ 0.4s

```
best_mae_gb, best_params_gb, best_technique_gb, best_splits_gb = best_params(model=gb, params=params_gb, CV_technique="all", max_splits=15)
print("Best MAE:",best_mae_gb,", Best params:", best_params_gb,", Best technique:", best_technique_gb,", Best splits:", best_splits_gb)
```

[38] ✓ 6m 59.3s

```
min: 5 max: 15 actual: 13
search: 8 - 15 actual: 14
gridSearchCV Finish
randomizeSearchCV Finish
Best MAE: 6.567191408598866 , Best params: {'n_estimators': 200, 'max_depth': 5, 'learning_rate': 0.1} , Best technique: randomizeSearchCV , Best splits: 14
```

GridSearchCV

GridSearchCV

```
# Inicializar RandomizedSearchCV con MAE como métrica
grid_search_gb = GridSearchCV(estimator=gb, param_grid=params_gb, cv=best_splits_gb, n_jobs=-1, scoring='neg_mean_absolute_error')

# Ajustar el modelo utilizando RandomizedSearchCV
grid_search_gb.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params_grid_search_gb = grid_search_gb.best_params_
best_score_grid_search_gb =  - grid_search_gb.best_score_
print("Mejores hiperparámetros:", best_params_grid_search_gb)
print("MAE: ", best_score_grid_search_gb)
```

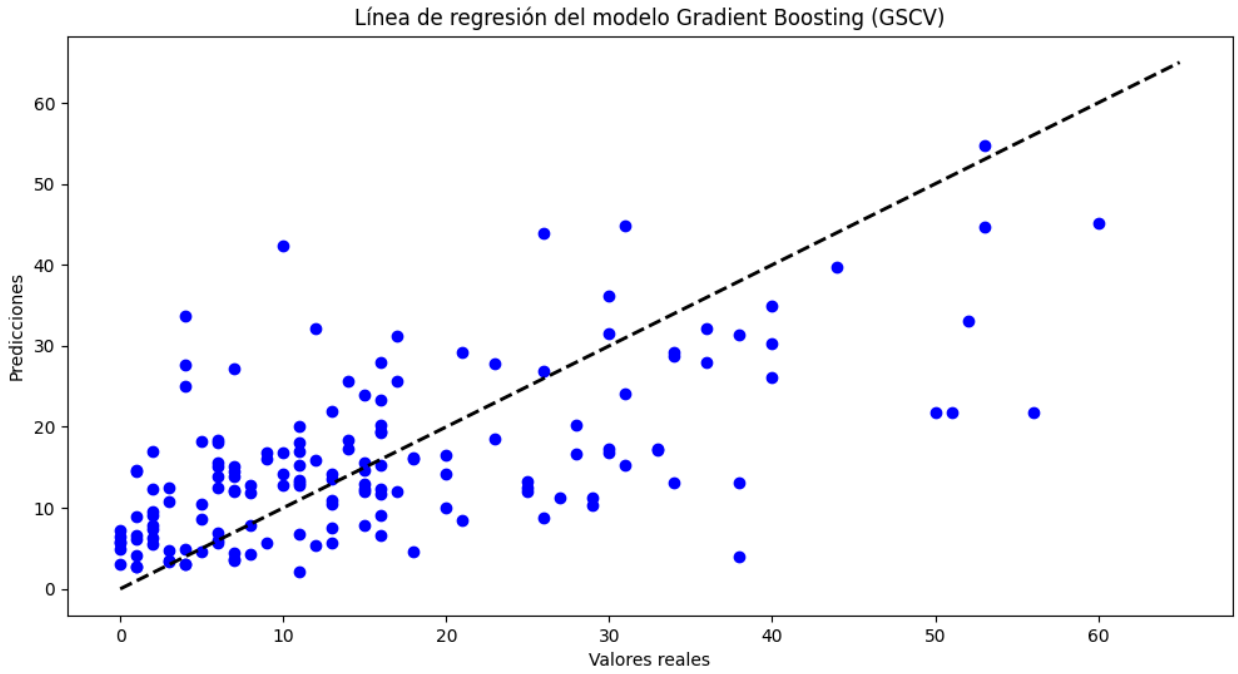
[39] ✓ 45.8s

```
Mejores hiperparámetros: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200}
MAE:  6.585524221646681
```

```
# Realizar la validación cruzada para obtener las predicciones
y_pred_val_gscv_gb = cross_val_predict(grid_search_gb, X_val, y_val, cv=5)

# Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
mae_gscv_gb = mean_absolute_error(y_val, y_pred_val_gscv_gb)

print(f"Mean Absolute Error: {mae_gscv_gb}")
```

[40] ✓ 53.6s

```
Mean Absolute Error: 8.178609747948391
```



RandomizedSearchCV

RandomizedSearchCV

```
# Inicializar RandomizedSearchCV con MAE como métrica
random_search_gb = RandomizedSearchCV(estimator=gb, param_distributions=params_gb, n_iter=10, cv=best_splits_gb, n_jobs=-1, scoring='neg_mean_absolute_error')

# Ajustar el modelo utilizando RandomizedSearchCV
random_search_gb.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params_random_search_gb = random_search_gb.best_params_
best_score_random_search_gb =  - random_search_gb.best_score_
print("Mejores hiperparámetros:", best_params_random_search_gb)
print("MAE: ", best_score_random_search_gb)
```

✓ 17.9s

```
Mejores hiperparámetros: {'n_estimators': 300, 'max_depth': 5, 'learning_rate': 0.1}
MAE:  6.685963812139855
```

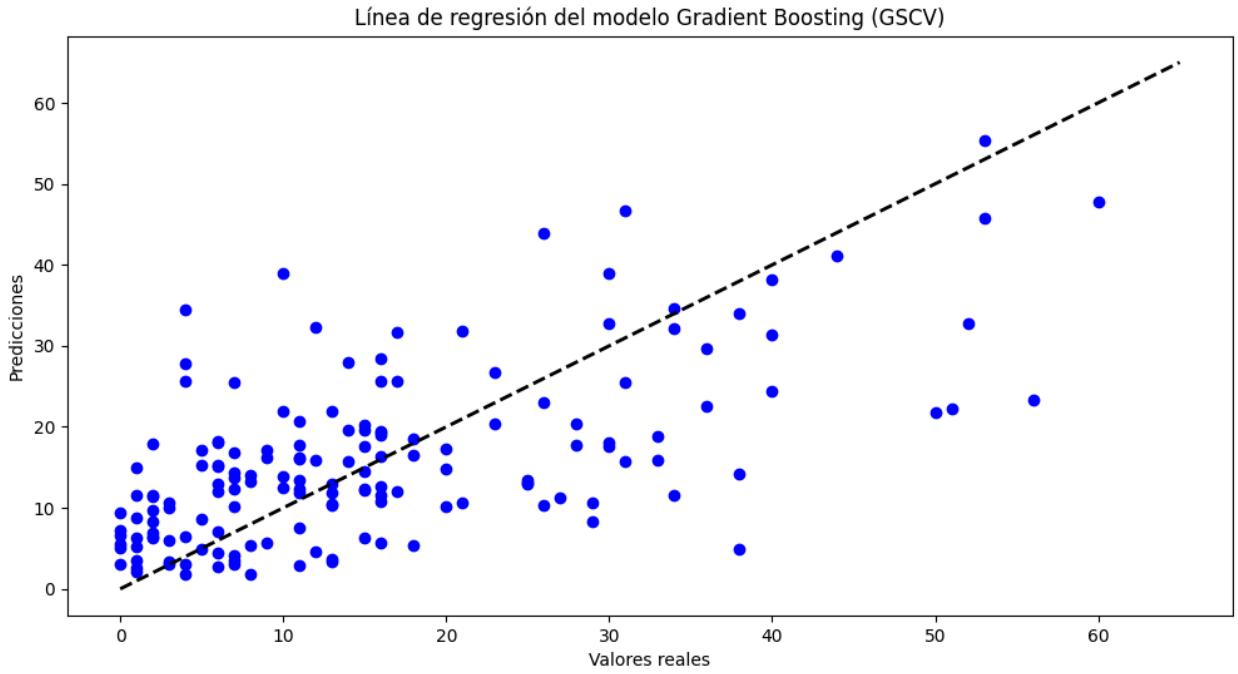
+ Código + Markdown

```
# Realizar la validación cruzada para obtener las predicciones
y_pred_val_rscv_gb = cross_val_predict(random_search_gb, X_val, y_val, cv=5)

# Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
mae_rscv_gb = mean_absolute_error(y_val, y_pred_val_rscv_gb)

print(f"Mean Absolute Error: {mae_rscv_gb}")
```

[43] ✓ 20.9s

```
Mean Absolute Error: 8.257375810819797
```

+ Código + Markdow



CONCLUSIÓN

En este caso, se ve algo más de diferencias, pero tampoco mucha, sigue ganando **GridSearchCV** (8,18) por muy poca ventaja sobre **RandomizedSearchCV** (8,26).

Con respecto al modelo anterior, **Random Forest**, este parece que da mejores resultados en la búsqueda de hyperparametros, pero al compararlo con los datos de validación se acerca pero se queda corto (**Random Forest** 8,16 - **Grading Boosting** 8,18). Por otro lado las gráficas de **Gradient Boosting** son algo mejores que la de **Random Forest**, parece que la línea de regresión está más cerca de todos los puntos en comparación a las gráficas de **Random Forest**. Aun así, creo que me sigo quedando con el modelo de **Random Forest**.

KNN

KNN

```
# Define el espacio de búsqueda de parámetros
params_knn = {'n_neighbors': np.arange(1, 51), 'weights': ['uniform', 'distance']}

# Crea el modelo KNN
knn = KNeighborsRegressor()
```

45] ✓ 0.5s

```
best_mae_knn, best_params_knn, best_technique_knn, best_splits_knn = best_params(model=knn, params=params_knn, CV_technique="all")
print("Best MAE:",best_mae_knn,", Best params:", best_params_knn,", Best technique:", best_technique_knn,", Best splits:", best_splits_knn)
```

46] ✓ 1m 26.6s

```
min: 51 max: 88 actual: 85
search: 80 - 88 actual: 87
gridSearchCV Finish
randomizeSearchCV Finish
Best MAE: 8.43495538825445 , Best params: {'n_neighbors': 10, 'weights': 'distance'} , Best technique: gridSearchCV , Best splits: 85
```

GridSearchCV

GridSearchCV

+ Código + Markdown

```
# Inicializar GridSearchCV con MAE como métrica
grid_search_knn = GridSearchCV(estimator=knn, param_grid=params_knn, cv=best_splits_knn, n_jobs=-1, scoring='neg_mean_absolute_error')

# Ajustar el modelo utilizando GridSearchCV
grid_search_knn.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params_grid_search_knn = grid_search_knn.best_params_
best_score_grid_search_knn =  - grid_search_knn.best_score_
print("Mejores hiperparámetros:", best_params_grid_search_knn)
print("MAE: ", best_score_grid_search_knn)
```

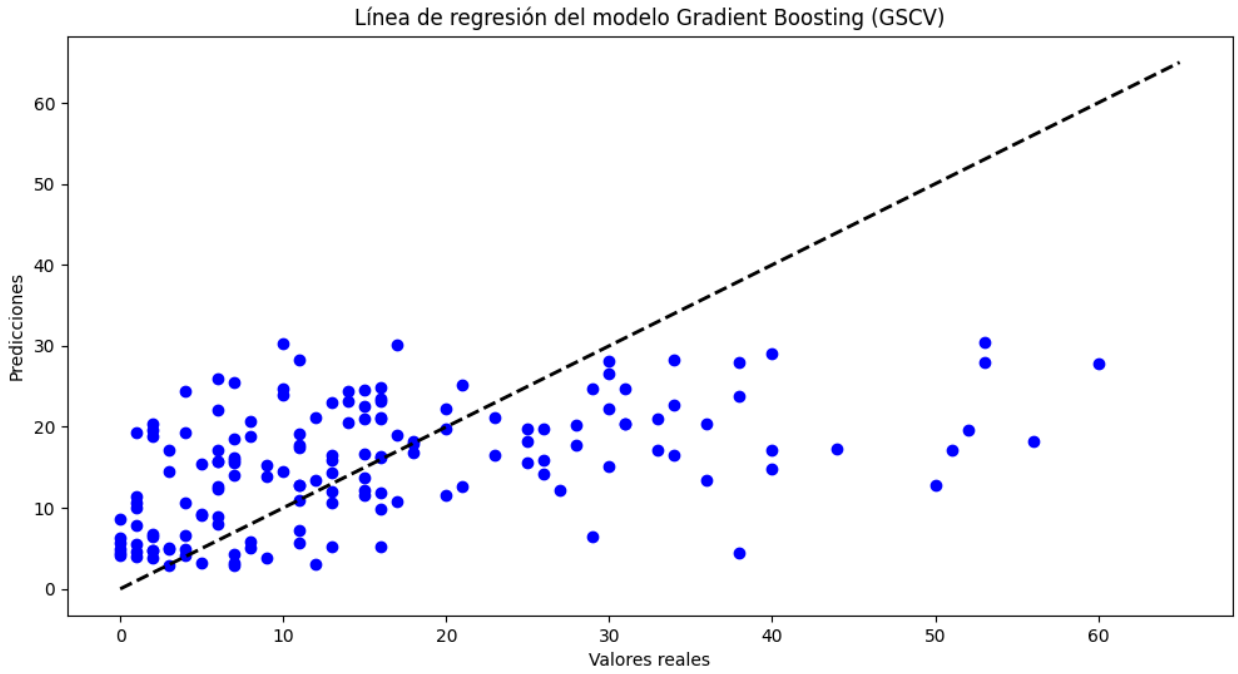
[47] ✓ 3.1s

```
Mejores hiperparámetros: {'n_neighbors': 10, 'weights': 'distance'}
MAE:  8.43495538825445
```

```
# Realizar la validación cruzada para obtener las predicciones
y_pred_val_gscv_knn = cross_val_predict(grid_search_knn, X_val, y_val, cv=5)

# Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
mae_gscv_knn = mean_absolute_error(y_val, y_pred_val_gscv_knn)

print(f"Mean Absolute Error: {mae_gscv_knn}")
```

8] ✓ 14.2s

```
Mean Absolute Error: 9.044992559186305
```



RandomizedSearchCV

RandomizedSearchCV

```
# Inicializar RandomizedSearchCV con MAE como métrica
random_search_knn = RandomizedSearchCV(estimator=knn, param_distributions=params_knn, n_iter=10, cv=best_splits_knn, n_jobs=-1, scoring='neg_mean_absolute_error')

# Ajustar el modelo utilizando RandomizedSearchCV
random_search_knn.fit(X_train, y_train)

# Obtener los mejores hiperparámetros
best_params_random_search_knn = random_search_knn.best_params_
best_score_random_search_knn =  - random_search_knn.best_score_
print("Mejores hiperparámetros:", best_params_random_search_knn)
print("MAE: ", best_score_random_search_knn)
```

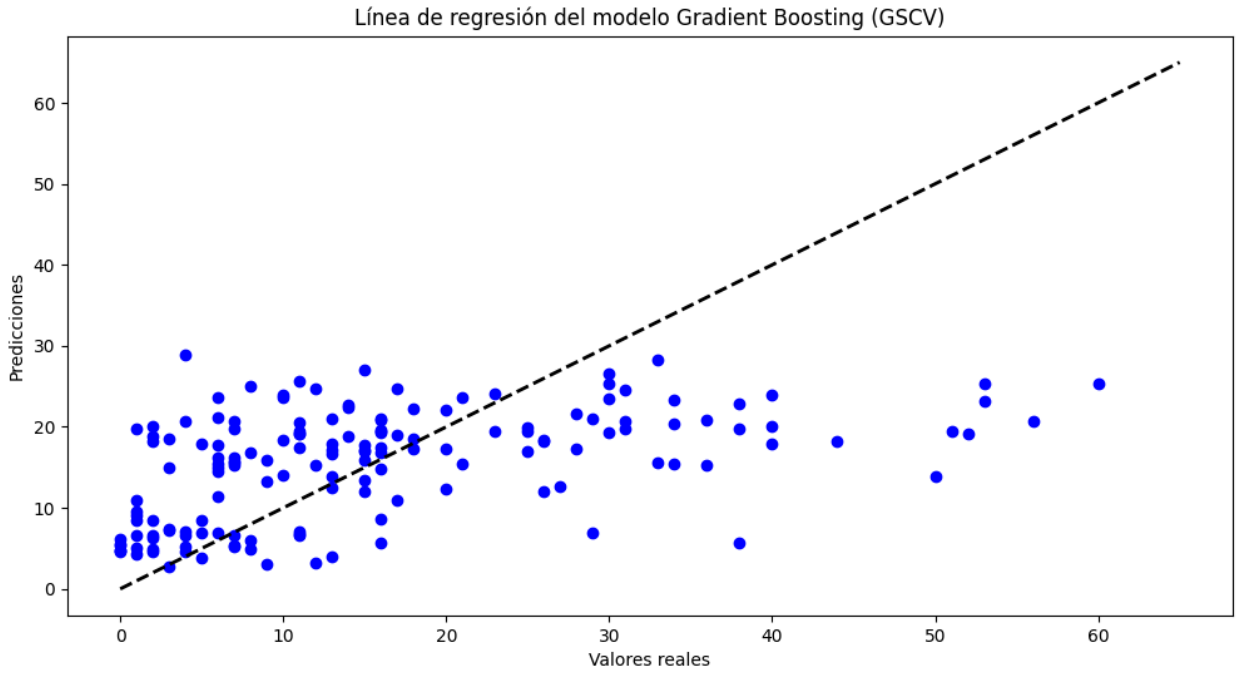
✓ 0.4s

```
Mejores hiperparámetros: {'weights': 'uniform', 'n_neighbors': 10}
MAE:  8.468172268907564
```

```
# Realizar la validación cruzada para obtener las predicciones
y_pred_val_rscv_knn = cross_val_predict(random_search_knn, X_val, y_val, cv=5)

# Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
mae_rscv_knn = mean_absolute_error(y_val, y_pred_val_rscv_knn)

print(f"Mean Absolute Error: {mae_rscv_knn}")
```

51] ✓ 1.9s

```
Mean Absolute Error: 9.014629135113445
```



CONCLUSIÓN

En el caso de **KNN** vemos que si comparamos el **MAE** dado por **GridSearchCV** (8,43) con el **MAE** dado por **RandomizedSearchCV** (8,47), vemos que sigue estando mejor el entrenamiento con **GridSearchCV**, pero si nos fijamos en la predicción con los datos de validación, es lo contrario, **RandomizedSearchCV** (9,01) es mejor que **GridSearchCV** (9,04). En este caso creo que gana **RandomizedSearchCV**.

Ahora, en comparación a los modelos anteriores, **KNN** es el peor de los tres, en el caso de **Random Forest** y **Gradient Boosting** la comparación estaba muy ajustada ya que daban **MAEs** muy parecidos y las gráficas no eran muy dispares. Pero, en el caso de **KNN** pasamos de un **MAE** de 8 de los casos anteriores a un **MAE** de 9 y no solo con eso la línea de regresión parece ser que no está tan cerca de los puntos como en los anteriores modelos.

CONCLUSIONES

Random Forest es el modelo elegido para el entrenamiento final. Aunque como la competición me deja tres intentos diarios voy a realizar una última comparación de los tres modelos con los **MAEs** obtenidos en la competición.

COMPROBACIÓN CON LOS DATOS DE TEST

RANDOM FOREST

Random Forest

```
random_forest = joblib.load('trained_models/dengIA_rf_modelo_entrenado.pkl')
```

132] ✓ 0.1s Group 2

```
# Hacer predicciones con el modelo cargado
predictions_rf = random_forest.predict(X_test)

# Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
mae_rf = mean_absolute_error(y_test, predictions_rf)

print(f"Random Forest MAE: {mae_rf}")
```

133] ✓ 0.4s Group 2

```
Random Forest MAE: 6.771071428571428
```

GRADIENT BOOSTING

Gradient Boosting

```
gradient_boosting = joblib.load('trained_models/dengIA_gb_modelo_entrenado.pkl')
```

134] ✓ 0.6s Group 2

```
# Hacer predicciones con el modelo cargado
predictions_gb = gradient_boosting.predict(X_test)

# Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
mae_gb = mean_absolute_error(y_test, predictions_gb)

print(f"Gradient Boosting MAE: {mae_gb}")
```

135] ✓ 0.2s Group 2

```
Gradient Boosting MAE: 6.646383523748631
```

KNN

```
KNN

    knn_model = joblib.load('trained_models/dengIA_knn_modelo_entrenado.pkl')
6]  ✓ 0.5s  Group 2

    # Hacer predicciones con el modelo cargado
    predictions_knn = knn_model.predict(X_test)

    # Calcular el Mean Absolute Error (MAE) entre las predicciones y los valores reales
    mae_knn = mean_absolute_error(y_test, predictions_knn)

    print(f"KNN MAE: {mae_knn}")
7]  ✓ 0.3s  Group 2

KNN MAE: 7.98265306122449
```

CONCLUSIÓN

Teniendo en cuenta los datos de test, vemos que **KNN** sigue siendo el peor de los tres modelos y que **Random Forest** y **Gradient Boosting** están muy igualados, pero en este caso **Gradient Boosting** (6,65 **MAE**) tiene mejores resultados que **Random Forest** (6,77 **MAE**).

PREDICCIÓN CON LOS DATOS DE LA COMPETICIÓN

Importamos el dataset de test facilitada en la competición.

```
# Cargar el archivo CSV con los datos a predecir
data_to_predict = pd.read_csv('https://raw.githubusercontent.com/Nestorbd/DengIA-Competition/master/datasets/dengue_features_test.csv')
✓ 0.3s  Group 2
```

Le aplicamos los mismos ajustes que al dataset de entrenamiento para poder pasarlo al modelo de entrenamiento. En este caso en vez de eliminar las filas con datos nulos, las cambiamos por la media de los valores de la columna para que tenga el mismo tamaño que el dataset de comparación que tendrá la competición.

```
data_to_predict['city'].replace(['iq', 'sj'],[0, 1], inplace=True)

data_to_predict['week_start_date'] = pd.to_datetime(data_to_predict['week_start_date'])
data_to_predict['week_start_date'] = (data_to_predict['week_start_date'] - pd.Timestamp("1970-01-01")) // pd.Timedelta('1D')

for column in data_to_predict.columns:
    # Verifica si la columna tiene valores nulos
    if data_to_predict[column].isnull().any():
        # Calcula la mediana de cada columna con datos nulos
        media = data_to_predict[column].mean()
        data_to_predict[column].fillna(media, inplace=True)
```

✓ 0.4s Group 2

Normalizamos los datos.

```
column_names_test = data_to_predict.columns.values

scaler_test = StandardScaler()
scaled_data_test = scaler_test.fit_transform(data_to_predict)
scaled_df_test = pd.DataFrame(scaled_data_test, columns=column_names_test)
```

✓ 0.2s Group 2

Y por último seleccionamos las columnas que nos interesan.

```
scaled_df_test = scaled_df_test[selected_features]
```

✓ 0.2s

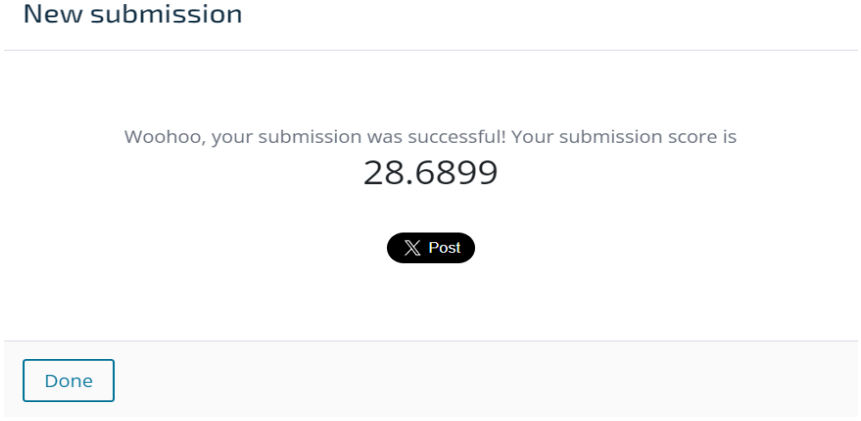
Ahora pasamos a predecir con cada uno de los modelos y ver que resultados obtiene cada uno:

RANDOM FOREST

Random Forest

```
# Realizar las predicciones con el modelo importado
predictions_rf = random_forest.predict(scaled_df_test)

results_rf = data_to_predict.copy()

results_rf["total_cases"] = predictions_rf.astype('int')

results_rf = results_rf[['city', 'year', 'weekofyear', 'total_cases']]

results_rf['city'].replace([0, 1],['iq', 'sj'], inplace=True)

# Guardar las columnas seleccionadas en un archivo CSV
results_rf.to_csv('results/dengIA_rf_results.csv', index=False)
```

✓ 0.4s Group 2

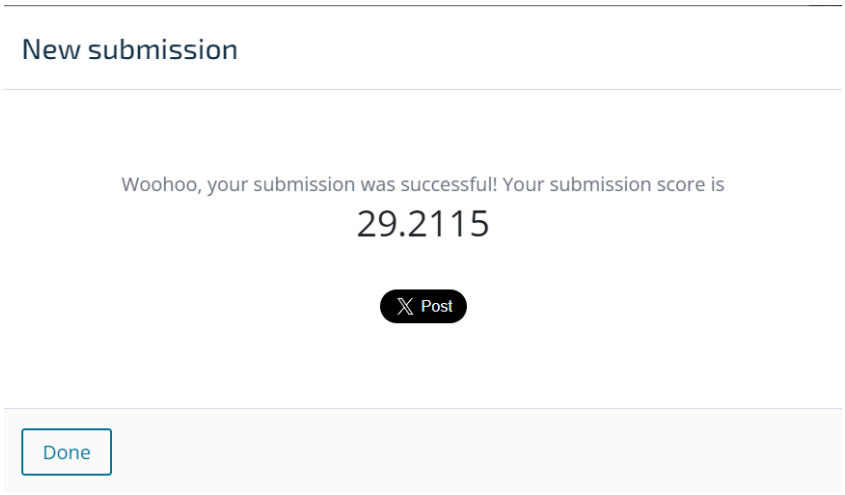


GRADIENT BOOSTING

Gradient Boosting

```
# Realizar las predicciones con el modelo importado
predictions_gb = gradient_boosting.predict(scaled_df_test)

results_gb = data_to_predict.copy()

results_gb["total_cases"] = predictions_gb.astype('int')

results_gb = results_gb[['city', 'year', 'weekofyear', 'total_cases']]

results_gb['city'].replace([0, 1],['iq', 'sj'], inplace=True)

# Guardar las columnas seleccionadas en un archivo CSV
results_gb.to_csv('results/dengIA_gb_results.csv', index=False)
```

✓ 0.3s Group 2

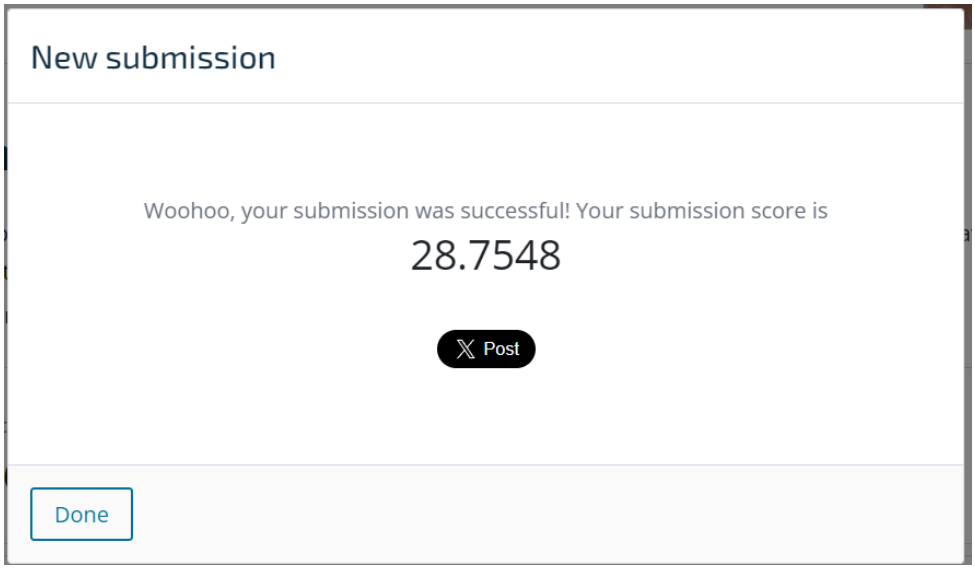


KNN

KNN

```
# Realizar las predicciones con el modelo importado
predictions_knn = knn_model.predict(scaled_df_test)

results_knn = data_to_predict.copy()

results_knn["total_cases"] = predictions_knn.astype('int')

results_knn = results_knn[['city', 'year', 'weekofyear', 'total_cases']]

results_knn['city'].replace([0, 1],['iq', 'sj'], inplace=True)

# Guardar las columnas seleccionadas en un archivo CSV
results_knn.to_csv('results/dengIA_knn_results.csv', index=False)
```

5] ✓ 0.3s Group 2



CONCLUSIÓN

Como podemos observar, al final el modelo que obtuvo mejor resultado fue **Random Forest** con 28,6899, el segundo inesperadamente fue **KNN** y el tercero, con bastante diferencia, **Gradient Boosting.**

CONCLUSIONES

Teníamos claro que **Random Forest** era uno de los mejores modelos entre estos tres en este trabajo, pero habría que analizar porque el modelo **KNN** ha sacado tanta ventaja a **Gradient Boosting** si en nuestra evaluación **KNN** era claramente el peor.

POSICIÓN FINAL