Actividad Guiada 3 (Extr): DengAi Predicting Disease Spread – Optimización

Nombre: Daniel Portugal Revilla

Objetivos: Predicción sobre los casos de dengue en una semana a partir de los datos metereológicos de la misma. La competición mide el error cometido en la predicción.

Técnica a utilizar: Optimización (GridSearch ,Boosting , RandomizedSearch).

Tecnología: Para el reto de DengAI se escogió trabajar con Python y sus diversas librerías como:

Al tener un backgroud de informático, la comodidad y familiaridad de trabajar con python es nativa, así como la ventaja de sus diversas librerías, el poder trabajar en diferentes entornos y plataformas, poder migrar a frameworks como Pyspark con facilidad, etc.

Introducción y definición del problema

La competición le reta a predecir el número de casos de dengue que se notifican cada semana en San Juan (Puerto Rico) e Iquitos (Perú) utilizando los datos ambientales recogidos por diversos organismos del Gobierno Federal de los Estados Unidos. Los datos proporcionados tienen características como la temperatura, la humedad y la precipitación máximas semanales. El aumento de las precipitaciones también debería contribuir al aumento de los mosquitos y, por lo tanto, casos de fiebre del dengue. Con una gran cantidad de datos climáticos y otros factores, vamos a calcular los grandes contribuyentes y predecir los resultados de los datos proporcionados más adelante.

Descripción de los datos

1. Tres conjuntos de datos proporcionados por Driven Data
2. characteristics:
    a. City, year, weekofyear, etc.
    b. Los datos climáticos son los más comunes, es posible eliminar las características correlacionadas

Cargamos los datos

```
train = pd.read_csv('./Data/dengue_features_train.csv', encoding='utf-8',index_col=['city', 'year', 'weekofyear'])
test = pd.read_csv('./Data/dengue_features_test.csv', encoding='utf-8', index_col=['city', 'year', 'weekofyear'])
labels = pd.read_csv('./Data/dengue_labels_train.csv', encoding='utf-8', index_col=['city', 'year', 'weekofyear'])
```

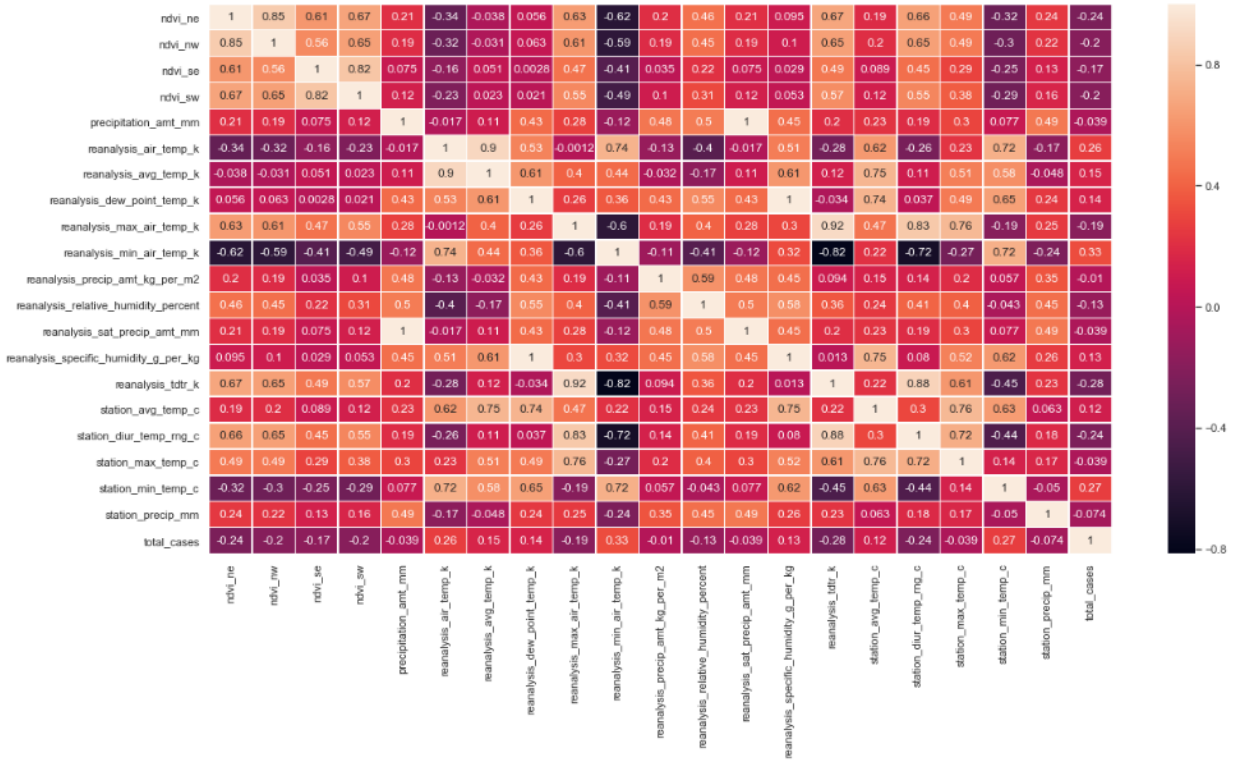
```
train_label = pd.merge(train, labels, on=['city', 'year', 'weekofyear'])
train_label.head(3)
```

			week_start_date	ndvi_ne	ndvi_nw	ndvi_se	ndvi_sw	precipitation_amt_mm	reanalysis_air_temp_k	reanalysis_avg_temp
city	year	weekofyear								
sj	1990	18	1990-04-30	0.12260	0.103725	0.198483	0.177617	12.42	297.572857	297.7428
		19	1990-05-07	0.16990	0.142175	0.162357	0.155486	22.82	298.211429	298.4428
		20	1990-05-14	0.03225	0.172967	0.157200	0.170843	34.54	298.781429	298.8785

Posteriormente seleccionamos las variables con las que trabajar. Es importante seleccionar las características cuando se va a resolver un problema mediante KNN ya que muchas variables pueden distorsionar el resultado del algoritmo que está basado en la distancia. Utilizaremos la correlación entre cada una de las características y la variable a predecir.

```
sns.set()
fig, ax = plt.subplots(figsize=(20,10))
sns.heatmap(train_label.corr(), annot=True, linewidths=.5, ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1f2a19cf240>
```



Seleccionamos aquellas características que, mediante análisis de correlaciones, nos han dado un valor superior a 0.20 o menor -0.20.

Realizamos una limpieza de los datos para eliminar los valores nulos (NaN)

```
if train_label_a.isnull().values.any():
  train_label_a = train_label_a.fillna(train_label_a.mean())

train_label_a.isnull().values.any()
```

```
False
```

```
# Elimnacion de los NaNs con la media
if test.isnull().values.any():
  test = test.fillna(test.mean())

test.isnull().values.any()
```

PARAMETRO DE OPTIMIZACION

Este problema es denominado habitualmente optimización de hiperparámetros, donde los parámetros del algoritmo se denominan hiperparámetros, mientras que los coeficientes encontrados por el propio algoritmo de aprendizaje se denominan parámetros.

El planteamiento será del de buscar aquella parametrización que ofrezca los resultados de mayor calidad (con respecto a las métricas establecidas) y de mayor robustez.

Utilizaremos la librería de machine learning Scikit-learn el cual proporciona diferentes herramientas para que la optimización de estos hiperparámetros pueda ser lo más sencilla posible. En concreto ofrece dos alternativas, la búsqueda en cuadrícula (grid search) y la búsqueda aleatoria (RandomSearch)

GridSearch

La búsqueda en cuadrícula es un enfoque de ajuste de parámetros que permite construir y evaluar metódicamente un modelo para cada combinación de parámetros de algoritmo especificados en una cuadrícula.

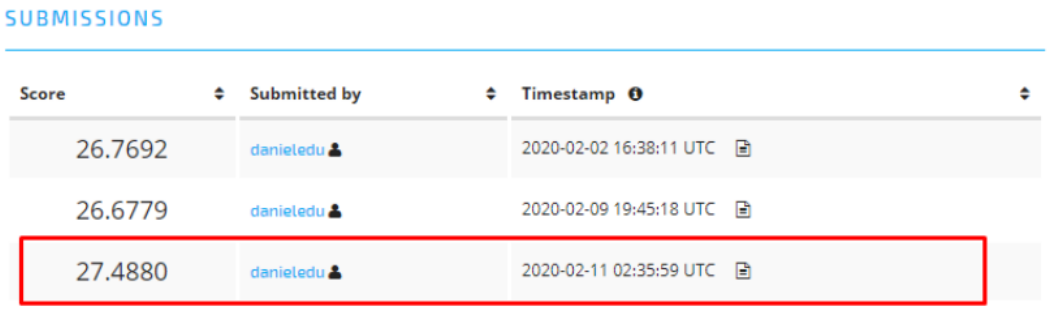
```
# Se ha usado 5 arboles en random forest para mayor precision en la optimizacion
param_dist = {"n_estimators": [4, 8, 16, 32, 64, 128],   # Numero de arboles en random forest
              "max_features": ['auto', 'sqrt'],          # Número de características a considerar en cada división
              "max_depth": [8, 4, 2],                    # Número máximo de niveles en el árbol.
              "min_samples_split": [2, 4, 6],            # Número mínimo de muestras requeridas para dividir un nodo
              "min_samples_leaf": [8, 12, 16],           # Número mínimo de muestras requeridas en cada nodo hoja
              "bootstrap": [True, False]                 # Método de selección de muestras para entrenar cada árbol.
            }
```

```
# Creamos nuestra estructura de busqueda por validacion cruzada. cv = intervalo de distribucion cruzada
# Entre mayor sea el numero de cross validation (cv), mejor es el aprendizaje
grid_regres = GridSearchCV(estimator = regressor, param_grid= param_dist, cv = 10)

# Ajustar el modelo de busqueda aleatorio
grid_regres.fit(X = train_label_a.drop(['total_cases'], axis=1),
                          y = train_label_a['total_cases']
                )
```

```
# Elegir el mejor
best_grid = grid_regres.best_estimator_

# Ajuste y predicción
best_grid.fit(X = train_label_a.drop(['total_cases'], axis=1), y = train_label_a['total_cases'])
y_pred = best_grid.predict(X = test)
```



Boosting

Vamos a utilizar las mismas técnicas para optimizar los párametros de los algortimos basados en Boosting. En primer lugar, Adaboost, con parámetros como el número de árboles, el coeficiente de aprendizaje y la función de pérdida.

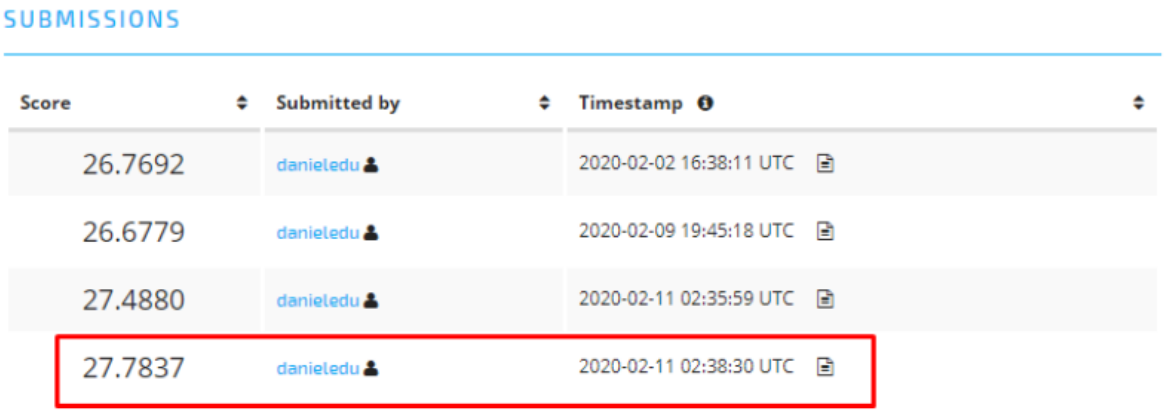
```
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor

param_dist = {
 "n_estimators": [4, 8, 16, 32, 64, 128],
 "learning_rate" : [0.01,0.05,0.1],
 "loss" : ['linear', 'square', 'exponential']
 }
grid_ada = GridSearchCV(AdaBoostRegressor(DecisionTreeRegressor(criterion='mae')),
                        param_grid = param_dist, cv=10)

grid_ada.fit(X = train_label_a.drop(['total_cases'], axis=1),y = train_label_a['total_cases'])

best_ada = grid_ada.best_estimator_
print (best_ada)
# fit and predict
best_ada.fit( X = train_label_a.drop(['total_cases'], axis=1), y = train_label_a['total_cases'])
y_pred = best_ada.predict(X = test)
```

```
AdaBoostRegressor(base_estimator=DecisionTreeRegressor(criterion='mae',
                                                       max_depth=None,
                                                       max_features=None,
                                                       max_leaf_nodes=None,
                                                       min_impurity_decrease=0.0,
                                                       min_impurity_split=None,
                                                       min_samples_leaf=1,
                                                       min_samples_split=2,
                                                       min_weight_fraction_leaf=0.0,
                                                       presort=False,
                                                       random_state=None,
                                                       splitter='best'),
                  learning_rate=0.01, loss='exponential', n_estimators=128,
                  random_state=None)
```



RandomizedSearch

Para utilizar RandomizedSearchCV, primero necesitamos crear el conjunto de parámetros a muestrear durante el proceso de optimización.

```
param_dist = {"n_estimators": [4, 8, 16, 32, 64, 128], # Number of trees in random forest
              "max_features": ['auto', 'sqrt'], # Number of features to consider at every split
              "max_depth": [16, 12,8,4, 2,None], # Maximum number of levels in tree
              "min_samples_split": sp_randint(2, 50), #  Minimum number of samples required to split a node
              "min_samples_leaf": sp_randint(1, 50), # Minimum number of samples required at each leaf node
              "bootstrap": [True, False], # Method of selecting samples for training each tree
              "criterion": ["mse", "mae"]}
```

En cada iteración, el algoritmo elegirá una combinación diferente de las características. Si se probaran de forma exhaustiva todas las características el problema se volvería muy costoso computacionalmente. Al menos al utilizar búsqueda aleatoria se seleccionará al azar una muestra de las mismas para buscar en un reducido pero significativo rango de valores. Los argumentos más importantes en RandomizedSearchCV son n_iter, que controla el número de combinaciones diferentes a probar, y cv, que es el número de cruces a usar para la validación cruzada. Más iteraciones cubrirán un espacio de búsqueda más amplio y más cruces de cv reducen las posibilidades de sobreaprendizaje, pero al aumentar cada una de ellas se incrementará el tiempo de ejecución.

```
rnd_regres = RandomizedSearchCV(estimator = regressor,
                                param_distributions = param_dist,
                                n_iter = 100, cv = 10, random_state=0, n_jobs = -1)
# Fit the random search model
rnd_regres.fit(X = train_label_a.drop(['total_cases'], axis=1),
               y = train_label_a['total_cases'] )
```

Una vez que hemos identificado la mejor parametrización vamos a pasar a hacer una ejecución del modelo y vamos graficar sus resultados. Recordamos que al final del paso 1 hemos dividido en entrenamiento/tuneado y test Posteriormente, vamos a ejecutar el modelo con la mejor parametrización que hayamos obtenido anteriormente

```
# choose the best
best_random = rnd_regres.best_estimator_
# fit and predict
best_random.fit(X = train_label_a.drop(['total_cases'], axis=1), y = train_label_a['total_cases'])
y_pred = best_random.predict(X = test)
```

SUBMISSIONS

Score	Submitted by	Timestamp
26.7692	danieledu	2020-02-02 16:38:11 UTC
26.6779	danieledu	2020-02-09 19:45:18 UTC
27.4880	danieledu	2020-02-11 02:35:59 UTC
27.7837	danieledu	2020-02-11 02:38:30 UTC
28.5529	danieledu	2020-02-11 02:40:19 UTC