Master Universitario en Bioinformática para Ciencias de la Salud
Facultad de Informática
UNIVERSIDADE DA CORUÑA

Práctica 3. Análisis de datos con Spark.

INTELIGENCIA COMPUTACIONAL PARA DATOS DE ALTA DIMENSIONALIDAD.

Cristian Morillo Losada | Pedro Sánchez García

MÁSTER UNIVERSITARIO EN BIOINFORMÁTICA PARA CIENCIAS DE LA SALUD.
PROFESOR: DR. CARLOS EIRAS FRANCO.

INTRODUCCIÓN.

En la anterior práctica de la asignatura, se trabajó principalmente con el API de RDDs en Spark, como conjuntos de elementos de diversos tipos para efectuar numerosas operaciones *map* y *reduce* junto con variantes asociadas. Ahora, se trabaja el API de DataFrame, que conforma un nivel de abstracción donde se distingue un conjunto de filas bajo un esquema determinado, de tal forma que se puede realizar diversas operaciones sobre columnas, filtrados, agrupaciones y consultas en SQL sobre vistas gracias a spark.sql.

Por tanto, la primera pregunta planteada pretende recoger el manejo de las APIs de RDDs y DataFrames, así como la transformación entre ambos tipos. En cambio, las siguientes cuestiones se centran en el aprendizaje automático mediante la librería MLLib de Spark, distinguiendo un entrenamiento de modelos y análisis de su precisión en base al error cuadrático medio.

Para ello, se utiliza el conjunto de datos "Reef Life Survey (RLS): Global reef fish dataset", localizado en el repositorio AOC (Australian Ocean Data Network). Su estructura consta de registros de peces óseos y elasmobranquios recolectados por buceadores de Reef Life Survey (RLS) a lo largo de transectos de 50 metros en arrecifes de coral, localizados alrededor del mundo.

OBJETIVOS.

Se pretende comprobar la posibilidad de utilizar el recuento por especies de peces de una zona para estimar su localización geográfica. Para ello, se manipulan los datos en un formato específico para el algoritmo de aprendizaje, procediendo a su entrenamiento y validación de modelos para estimar latitud o longitud dado un recuento de peces. Se deben adquirir o reforzar las destrezas mostradas:

- Manejo de las APIs de RDDs y DataFrames.
- Transformación entre RDDs y DataFrames.
- Trabajo con la librería MLLib de Spark.
- Seguimiento de un esquema de trabajo de aprendizaje máquina adecuado.

PREGUNTAS.

P1. Carga los datos en Spark y escribe el código necesario para contestar a las siguientes preguntas:

1. ¿Cuántas especies (Taxon) distintos se han identificado?

```
especies = df.groupBy('Taxon').count()

print('Se han identificado {0} especies distintas'.format(especies.count()))
```

Se ha planteado un agrupamiento por taxón, con sus correspondientes recuentos. No obstante, dado que se solicitan especies distintas, se procede al recuento de filas en el Dataframe de "especies", logrando 3.017 como resultado.

2. ¿Cuáles son las 10 familias con mayor recuento (Total)?

```
df = spark.read.csv('/media/sf_PR_ICDAD/Materiales_parte3/RLS.csv', header=True)

df.createOrReplaceTempView('RLS')

spark.sql('select Family as familias, count(*) as recuento from RLS group by familias order by recuento desc').show(10)
```

En este caso, se creó una tabla denominada 'RLS' a partir del DataFrame original, procediendo posteriormente a una consulta en SQL donde se proyectan las familias y el total, agrupando por familias, ordenando el recuento en orden descendente y mostrando los 10 primeros resultados:

```
>>> spark.sql('select Family as familias, count(*) as recuento from RLS group by familias order by recuento desc').show(10)
+--------------+--------+
|      familias|recuento|
+--------------+--------+
|      Labridae|  127803|
| Pomacentridae|   99807|
|Chaetodontidae|   30493|
|  Acanthuridae|   28657|
|      Scaridae|   22827|
|    Serranidae|   21854|
|    Kyphosidae|   21713|
|      Mullidae|   13029|
| Monacanthidae|   12530|
|    Blenniidae|   11280|
+--------------+--------+
only showing top 10 rows
```

Figura 1. Resultado en la pregunta 2, donde se aprecia un mayor recuento total para la familia *Labridae*.

3. ¿En qué "eco regiones" de España se han realizado mediciones?

```
spark.sql('select Ecoregion as eco_regiones from RLS where Country = "Spain" group by eco_regiones order by eco_regiones asc').show()
```

Sobre la tabla 'RLS' creada anteriormente, se realizó la consulta reflejada, de modo que con la cláusula *where* se filtran aquellas eco regiones de España en las que se efectuaron mediciones:

```
>>> spark.sql('select Ecoregion as eco_regiones from RLS where Country = "Spain" group by eco_regiones order by eco_regiones asc').show()
+--------------------+
|        eco_regiones|
+--------------------+
|         Alboran Sea|
|Azores Canaries M...|
|   Saharan Upwelling|
|South European At...|
|Western Mediterra...|
+--------------------+
```

Figura 2. Resultado de consulta en la pregunta 3, donde se muestran las 5 eco regiones de España.

4. ¿Cuántas familias tienen recuentos totales inferiores a 10?

```
familias_10 = df.filter(df.Total < 10).select('Family').distinct().count()

print('Se han identificado {0} familias con recuentos inferiores a 10'.format(familias_10))
```

Sobre el DataFrame, se planteó la consulta reflejada, donde se filtra en primer lugar por aquellos recuentos totales inferiores a 10, proyectando a continuación aquellas familias en base al filtrado y

hallando el recuento de estas. Tal y como se refleja a continuación, existen 183 familias que cumplen la condición:

```
>>> familias_10 = df.filter(df.Total < 10).select('Family').distinct().count()
>>> print('Se han identificado {0} familias con recuentos inferiores a 10'.format(familias_10))
Se han identificado 183 familias con recuentos inferiores a 10
```

Figura 3. Resultado de la pregunta 4, con el recuento alcanzado para la condición establecida.

5. ¿Cuántos individuos se encontraron en la survey que más individuos encontró en una región española?

```
spark.sql('select SurveyID, Country, sum(Total) as individuos from RLS where Country = "Spain"
group by SurveyID, Country order by individuos desc').show(1)
```

Para esta última cuestión, se efectuó esta consulta SQL en RLS, donde se proyecta el identificador de la survey, país y suma de los recuentos totales filtrando por España y agrupando por el identificador y país, de modo que se ordena dicha suma en forma descendente y mostrando el primer resultado, que será aquella survey que cumple con la condición solicitada:

```
>>> spark.sql('select SurveyID, Country, sum(Total) as individuos from RLS where Country = "Spain" group by SurveyID, Country order by individuos desc').show(1)
+---------+-------+----------+
| SurveyID|Country|individuos|
+---------+-------+----------+
|912349520|  Spain|  100143.0|
+---------+-------+----------+
only showing top 1 row
```

Figura 4. Resultado para la pregunta 5, con la survey que posee mayor número de individuos.

Tal y como se refleja en la **Figura 4**, se hallaron 100.143 individuos en la survey 912349520 como identificador.

P2. Adapta tus datos en Spark al formato que necesita MLLib. Deberás conseguir un DataFrame con, al menos, dos columnas: una con la variable que quieres predecir y otra con los datos de entrada de que dispondrá el modelo para hacer sus predicciones (representadas con un objeto de tipo pyspark.ml.linalg.Vectors.DenseVector).

En primer lugar, del DataFrame original, se seleccionaron las variables correspondientes a SurveyID, SiteLat, SiteLong, Family y Total. A continuación, mediante el API de RDDs, se procedió al agrupamiento de las mediciones por survey, con el fin de lograr, para cada una, un listado de tuplas (Family, Total) que indican cuántos individuos de esa familia se determinaron:

```
## 1.) De cada medición, quédate con las variables SurveyID, SiteLat, SiteLong,Family y Total:
df_original = spark.read.csv('/media/sf_PR_ICDAD/Materiales_parte3/RLS.csv', header=True)

df = df_original['SurveyID', 'SiteLat', 'SiteLong', 'Family', 'Total']

## 2.) Usando el API de RDDs, agrupa las mediciones por survey para obtener, para cada survey, un
## listado de tuplas (Family, Total) que representan cuántos individuos de esa familia se encontraron.

### 2.1) Conversión df -> RDD:

rdd1 = df.rdd.map(lambda x:[x[0],float(x[1]),float(x[2]), x[3],x[4]])

### 2.2) Agrupación por survey:

rdd2 = rdd1.map(lambda x: ((x[0],x[1],x[2]), (x[3],x[4]))).groupByKey().mapValues(list)
```

Figura 5. Planteamiento de las dos primeras fases en la adaptación de los datos para MLLib.

Cabe destacar, con respecto al rdd2 reflejado en la **Figura 5**, que el agrupamiento se hizo en base a la survey, la latitud y longitud, distinguiendo, además, el listado de las tuplas (Family, Total) como el otro elemento. A través de la función toVector suministrada, se transformó dicha lista en un vector que conserva la proporción en la que aparecen representadas las familias en ese survey. Finalmente, gracias a la función auxiliar pasaFilaARow, se obtuvo el DataFrame preciso por la transformación de cada elemento del RDD logrado en el último paso en una Row:

```
familias=[linea.rstrip() for linea in open("/media/sf_PR_ICDAD/Materiales_parte3/familias.txt",'r').readlines()]
import numpy as np

def toVector(familyCountList: list) -> list:
    '''
    Transforma una lista de tuplas (nombre_familia, recuento) a un vector que
    indica en qué proporción aparece cada una de las familias de interés
    '''
    counts=np.zeros(len(familias))
    for (f,c) in familyCountList:
        if f in familias:
            counts[familias.index(f)]=float(c)
    total=np.sum(counts)
    if total==0:
        return counts
    return (counts/total).tolist()

rdd3 = rdd2.map(lambda x: (x[0], toVector(x[1])))

## 4.) Obtén el DataFrame requerido transformando cada elemento del RDD en una Row con la función pasaFilaARow s
from pyspark.sql.types import Row
from pyspark.ml.linalg import Vectors

def pasaFilaARow(x:any) -> Row:
    '''
    Recibe un elemento de cualquier tipo y lo transforma en un Row de pyspark
    '''
    # Utilizaremos este diccionario para definir los campos de la Row y sus valores
    d = {}
    d["SurveyID"] = x[0][0]
    d["SiteLat"] = x[0][1]
    d["SiteLong"] = x[0][2]
    d["Vector"] = Vectors.dense(x[1])
    return Row(**d)

pasafilarow = rdd3.map(lambda x: pasaFilaARow(x))
dataframe_p2 = pasafilarow.toDF()
```

Figura 6. Tercera y cuarta fase para la adaptación de los datos en MLLib.

En base a la **Figura 6**, se puede observar el paso de la función sobre rdd2 en el elemento correspondiente a la lista de tuplas, de modo que se alcanza un rdd3 donde se distingue un componente de tuplas con SurveyID, SiteLat, SiteLong y otro con el vector que almacena las proporciones para cada familia de interés. A continuación, con la función auxiliar pasaFilaARow, fue necesario definir campos de la Row y sus valores correspondientes, pasándola al rdd3 para la conversión de cada elemento. Por último, se logró el dataframe_p2 adaptado con la función toDF().

P3. Entrena un modelo Random Forest Regression y calcula su precisión, en términos del error cuadrático medio, a la hora de predecir la latitud de cada survey atendiendo a los recuentos de especies.

Para la realización de esta pregunta, se tomó como punto de partida un dataframe_p3, que se encuentra centrado en el vector de proporciones para cada familia generado anteriormente y la correspondiente etiqueta, es decir, la latitud de cada survey. Como fundamentos principales del modelo Random Forest Regression, se trata de un algoritmo de aprendizaje supervisado de regresión, cuya funcionalidad es la predicción del valor numérico correspondiente a la latitud. En primer lugar, se llevó a cabo la partición de los datos en conjuntos de entrenamiento y test, con 70 y 30% respectivamente. Posteriormente, dado que estamos manejando ensembles de los árboles de decisión, se evaluaron unos parámetros relevantes en la configuración del entrenamiento del modelo: número de árboles (numTrees), máxima profundidad (maxDepth) y una semilla (seed) fijada por defecto en 42. Se llevaron a cabo diversas pruebas modificando el número de árboles y la máxima profundidad con el fin de mejorar precisión:

```
## Entrena un modelo Random Forest Regression y calcula su precisión, en términos del error cuadrático medio, a la
## hora de predecir la latitud de cada survey atendiendo a los recuentos de especies.
dataframe_p3 = dataframe_p2['Vector','SiteLat']

from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import RegressionEvaluator

# Separamos el dataframe en conjuntos de entrenamiento (70%) y test (30%):
(trainingData, testData) = dataframe_p3.randomSplit([0.7, 0.3])

## Pruebas con parámetros modificados para verificar mejoras en precisión:

# Con maxDepth = 5:
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=20, maxDepth=5, seed=42)  # 11.52
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=30, maxDepth=5, seed=42)  # 11.27
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=50, maxDepth=5, seed=42)  # 11.23

# Con maxDepth = 10:
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=30, maxDepth=10, seed=42) # 8.06

# Con maxDepth = 15:
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=40, maxDepth=15, seed=42) # 6.87
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=45, maxDepth=15, seed=42) # 7.01
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=50, maxDepth=15, seed=42) # 6.98

# Con maxDepth = 20:
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=40, maxDepth=20, seed=42) # 6.61
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=45, maxDepth=20, seed=42) # 6.67
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=50, maxDepth=20, seed=42) # 6.68

# Con maxDepth = 30:
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=40, maxDepth=30, seed=42) # 6.51 ****
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=45, maxDepth=30, seed=42) # 6.63
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=50, maxDepth=30, seed=42) # 6.59
rf = RandomForestRegressor(labelCol="SiteLat", featuresCol="Vector",numTrees=60, maxDepth=30, seed=42) # 6.67
```

Figura 7. Configuraciones con parámetros mencionados en el entrenamiento del modelo.

En la parte derecha de cada configuración mostrada en la **Figura 7**, se indican los resultados alcanzados en términos de la raíz del error cuadrático medio (rmse), determinado tras las siguientes fases:

```
model = rf.fit(trainingData)

## Predicciones:
predictions = model.transform(testData)

## Visualizamos 10 primeras filas para apreciar las predicciones con respecto a la latitud:
predictions.show(10)

## Evaluación de la raíz del error cuadrático medio (rmse):
rmse = RegressionEvaluator(labelCol="SiteLat", predictionCol="prediction", metricName="rmse")
rmse_evaluado = rmse.evaluate(predictions)

print("La raíz del error cuadrático medio (RMSE) en conjunto de test es: ", rmse_evaluado)
```

Figura 8. Determinación de la precisión lograda con las configuraciones en términos de raíz del error cuadrático medio.

De acuerdo con la **Figura 7**, se alcanzó el valor más reducido en la raíz del error cuadrático medio, en concreto, con valor de 6.51. Debemos tener en cuenta que esta medida de precisión nos permite conocer la diferencia exacta entre la salida real y esperada. Por tanto, se persigue un error cuadrático medio con un valor próximo a 0, que muestra una tendencia adecuada para el problema tratado. En este caso, a pesar de que es notablemente elevado, se ha logrado reducir de manera drástica frente a otras configuraciones, lo que parece indicar que la máxima profundidad ubicada entre 20 y 30, junto con el ajuste a 40 y 50 en número de árboles, contribuyen al mantenimiento de esa reducción frente a los otros casos realizados.

P4. Entrena algún otro modelo de regresión y compara el rendimiento con el apartado anterior. ¿Es válido para estimar latitud alguno de los modelos que has obtenido? Argumenta tu respuesta.

Se ha optado por el modelo de regresión Decision Tree Regressor, que de nuevo conforma un algoritmo de aprendizaje supervisado que se basa fundamentalmente en reglas de decisión con el conjunto de entrenamiento establecido, que se mantuvo en un 70% al igual que el caso anterior. Se modificó la máxima profundidad (maxDepth) en la configuración del entrenamiento del modelo. Para ello, se probó con valores de 2, 10, 20 y 30, logrando el valor más reducido (9.19) de raíz de error cuadrático medio con 10:

```
## Entrena algún otro modelo de regresión y compara el rendimiento con el apartado anterior.
## ¿Es válido para estimar latitud alguno de los modelos que has obtenido? Argumenta tu respuesta.

# Decision tree regression:

from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.evaluation import RegressionEvaluator

dataframe_p3 = dataframe_p2['Vector','SiteLat']

# Separamos el dataframe en conjuntos de entrenamiento (70%) y test (30%):
(trainingData, testData) = dataframe_p3.randomSplit([0.7, 0.3])

dt = DecisionTreeRegressor(labelCol="SiteLat", featuresCol="Vector") ## 12.48

# Configuraciones de modelo ajustando la profundidad máxima en el árbol:
dt = DecisionTreeRegressor(maxDepth=2, labelCol="SiteLat", featuresCol="Vector") ## 16.16

dt = DecisionTreeRegressor(maxDepth=10, labelCol="SiteLat", featuresCol="Vector") ## 9.19

dt = DecisionTreeRegressor(maxDepth=20, labelCol="SiteLat", featuresCol="Vector") ## 9.29

dt = DecisionTreeRegressor(maxDepth=30, labelCol="SiteLat", featuresCol="Vector") ## 9.43

model_dt = dt.fit(trainingData)

## Predicciones:
predictions_dt = model_dt.transform(testData)

## Visualizamos 10 primeras filas para apreciar las predicciones con respecto a la latitud:
predictions_dt.show(10)

## Evaluación de la raíz del error cuadrático medio (rmse):
rmse_dt = RegressionEvaluator(labelCol="SiteLat", predictionCol="prediction", metricName="rmse")
rmse_evaluado_dt = rmse_dt.evaluate(predictions_dt)

print("La raíz del error cuadrático medio (RMSE) en conjunto de test es: ", rmse_evaluado_dt)
```

Figura 9. Configuración del modelo Decision Tree Regressor con ajustes en profundidad máxima.

De este modo, se aprecia el incremento considerable en el resultado de precisión si se compara con el otro modelo de Random Forest Regression. Esto pone de manifiesto que los ajustes realizados en este último son adecuados de cara a la precisión en la pregunta planteada. Dado que resultan lejanos a 0, ambos modelos no serían válidos para estimar latitud en base al vector. Sin embargo, ambos serían útiles frente a una predicción aleatoria de latitud en base al vector, pues debemos tener en cuenta la influencia de la distribución no uniforme para la latitud en los datos empleados.

P5. ¿Qué 10 familias son las más relevantes a la hora de hacer las predicciones? Atendiendo solo a esas 10 familias, ¿cuánto empeora el modelo?

Se utilizó la funcionalidad *featureImportances* para cada modelo y convirtiendo a un array al finalizar el entrenamiento y determinar la correspondiente medida de precisión en términos del rmse. Se analizaron aquellos valores más elevados en el array obtenido y ordenado, hallando posteriormente su posición en el array accediendo a los índices:

```
>>> print(featureImportances)
[1.72843514e-02 2.18997682e-04 3.65349807e-06 0.00000000e+00
 1.13410981e-08 0.00000000e+00 3.68321215e-08 1.59437706e-06
 2.32702766e-03 8.57809469e-03 1.99589219e-12 4.28406795e-05
 1.17252185e-03 7.15287181e-05 5.96604305e-04 1.44335600e-03
 7.79083902e-03 9.48194760e-06 1.79204837e-04 7.09479048e-05
 7.45429008e-03 6.75534138e-04 1.24384860e-04 4.25041510e-06
 0.00000000e+00 3.79919169e-08 2.04998719e-03 1.04800025e-06
 2.21181490e-04 6.18054089e-04 1.87429838e-03 7.51575873e-04
 8.62266118e-04 1.28512656e-05 0.00000000e+00 5.02532742e-04
 3.36982978e-02 2.47492301e-05 7.14008962e-06 1.28749647e-01
 1.80055021e-04 1.17939340e-04 2.84043998e-03 2.53129284e-04
 1.94393856e-04 8.54438366e-06 5.26960953e-05 2.18047101e-02
 2.58029557e-07 7.61949523e-05 1.06850987e-06 1.07910838e-09
 5.21930182e-04 2.25965635e-03 5.91295989e-05 5.85728422e-06
 0.00000000e+00 1.07004274e-05 1.66520638e-02 7.12391237e-04
 3.24494104e-04 2.57702595e-04 0.00000000e+00 4.36430602e-04
 1.83987006e-02 8.12110567e-05 4.40296560e-07 2.47844329e-03
 1.84213489e-05 5.92018713e-06 3.57557219e-10 6.18686452e-04
 3.25812812e-02 1.07426554e-03 9.22954515e-03 4.38227267e-04
 2.87250962e-06 2.66251209e-05 5.38490607e-05 1.85177912e-02
 7.50665723e-03 1.68674402e-04 3.07808085e-07 0.00000000e+00
 8.87444913e-06 0.00000000e+00 5.52492697e-06 8.51987482e-02
 3.09086719e-02 3.34526373e-03 2.40398806e-05 7.59266187e-05
 1.11813236e-03 4.35095015e-03 3.58415169e-06 1.22735372e-05
 5.14364603e-05 8.76694715e-03 4.13154413e-04 0.00000000e+00
 1.16547395e-03 5.27308485e-06 1.40378412e-10 3.68037768e-02
 1.86947841e-07 1.26006289e-04 1.44767756e-04 3.59871169e-05
 1.04702636e-03 7.85095718e-03 9.57520030e-04 1.21796728e-04
 5.23892930e-03 1.23495964e-03 7.08014378e-04 5.55927141e-02
 7.77494124e-05 4.46619642e-06 3.75386957e-05 8.36769627e-06
 1.07014650e-04 1.91697742e-04 2.60577542e-04 1.55079078e-03
 8.67881464e-04 3.59415375e-06 5.95958152e-05 7.75348992e-06
 6.47460535e-06 3.53187716e-03 2.26463194e-05 1.95752419e-04
 2.74687553e-08 1.54931751e-09 6.38711066e-06 5.38117827e-03
 4.28869456e-05 5.47565981e-07 4.82395838e-03 2.74311253e-04
 3.77818186e-02 6.05752622e-03 1.75022271e-04 1.58844272e-02
 4.19944562e-02 1.46706748e-06 9.30176272e-04 1.58451356e-03
```

```
featureImportances = model.featureImportances.toArray()
a = np.sort(featureImportances)
b = np.featureImportances
indices, = np.where(featureImportances==0.00000000e+00)
print(indices)
```

Figura 10. Ejemplo de *featureImportances* con valores para modelo Random Forest Regressor (izquierda) y fases para la obtención de los índices con el fin de hallar las 10 familias con mayor relevancia (derecha).

Como resultado, por ejemplo, para el caso del modelo Random Forest Regressor, se obtuvieron los siguientes índices: [3 5 24 34 56 62 83 85 99 148]. En consecuencia, se accedió al familias.txt y se seleccionaron las familias que ocupaban las *n+1* posiciones en el array. De este modo, las 10 familias más relevantes para dicho modelo fueron: *Ammodytidae, Anguillidae, Brachionichthyidae, Centrolophidae, Eleotridae, Exocoetidae, Idiosepiidae, Isonidae, Megalopidae* y *Pseudomugilidae*. A continuación, estas se conservaron en un nuevo familias_10_rf.txt, empleado en la fase de uso de la función toVector auxiliar para el nuevo entrenamiento del modelo.

Se siguió el mismo procedimiento en el caso del modelo Decision Tree Regressor, logrando los índices y familias correspondientes: [1 2 4 6 7 10 14 17 21 23], *Agonidae, Ambassidae, Anarhichadidae, Aploactinidae, Aracanidae, Aulorhynchidae, Batrachoididae, Bothidae* y *Brachaeluridae*. Por tanto, se conservó un familias_10_dt.txt para el nuevo entrenamiento del modelo.

En ambos casos, se incrementó notablemente el rmse, con un valor de 21.23 en el modelo Random Forest Regressor y de 20.78 para el Decision Tree Regressor, lo que pone de manifiesto que las predicciones de la latitud en base a dichas familias empeoran de manera drástica. De esta forma, se puede concluir que las 179 familias, a pesar de presentar unos valores con amplias variaciones en *featureImportances*, contribuyen en gran medida a dichas predicciones, lo que se verificó anteriormente con los valores de rmse en las preguntas 3 y 4.