Sistemas de Gestión de Datos y de la Información
Máster en Ingeniería Informática, 2018-19
Práctica 2

Fecha de entrega: miércoles 21 de noviembre de 2018, 16:55h

Objetivos mínimos
Implementar el algoritmo de generación de árboles de clasificación *ID3* (apartado 1). Generar un índice invertido que almacene pesos TF-IDF a partir de una colección de documentos y realizar consultas vectoriales y booleanas tipo *AND* (apartado 3).

Entrega de la práctica
La práctica se entregará en un único fichero **GrupoXX.zip** mediante el Campus Virtual de la asignatura. El fichero contendrá una carpeta por cada uno de los apartados (ID3, SparkML, Recuperacion_vectorial e Indice_completo) conteniendo los ficheros de código fuente (extensión .py) y documentos PDF que requiera cada apartado.

Lenguaje de programación
Python **3.6**.

Ficheros
• Apartado 1: ficheros en formato CSV con cabecera para probar el clasificador ID3 (railway.data, lens.data y car.data). Para este último hay una selección más pequeña (car_small.train y car_small.test).
• Apartado 2: ficheros en formato CSV con cabecera adult.train y adult.test.
• Apartados 3 y 4: colección de mensajes 20news−18828.tar.gz.

Calificación
La generación de árboles de clasificación mediante el algoritmo *ID3* (apartado 1) y la generación de *índices invertidos completos* (apartado 4) tienen un peso del 30 % cada uno. La generación de tuberías SparkML de clasificación (apartado 2) y la generación y uso de índices invertidos vectoriales con TF-IDF (apartado 3) tienen un peso del 20 % cada uno. Además del correcto funcionamiento del código se valorará también su claridad, su documentación y su organización en funciones auxiliares reutilizables.

Declaración de autoría e integridad
Todos los ficheros entregados (tanto código fuente como documentos de texto) contendrán una cabecera en la que se indique la asignatura, la práctica, el grupo y los autores. Esta cabecera también contendrá la siguiente declaración de integridad:

> *NombreAlumno1* y *NombreAlumno2* declaramos que esta solución es fruto exclusivamente de nuestro trabajo personal. No hemos sido ayudados por ninguna otra persona ni hemos obtenido la solución de fuentes externas, y tampoco hemos compartido nuestra solución con nadie. Declaramos además que no hemos realizado de manera deshonesta ninguna otra actividad que pueda mejorar nuestros resultados ni perjudicar los resultados de los demás.

No se corregirá ningún fichero que no venga acompañado de dicha cabecera.

1. Árbol de clasificación ID3 [30 %]

A) [20 %] (Objetivo mínimo)

Extender el esqueleto ID3.py para implementar una clase ID3 que genere árboles de clasificación siguiendo el método ID3 (TDIDT con **ganancia de información**) para la generación de árboles de clasificación. La clase ID3 debe contener al menos los siguientes métodos:

1. __init__(self, fichero)
   El constructor lee el fichero de texto fichero formado por instancias en formato CSV[1] con cabecera donde el último atributo se llama **class** y contiene el valor de clase. Los datos están bien formados, así que todas las líneas tienen el mismo número de campos. Además, supondremos que todos los atributos son categóricos y que los posibles valores para cada atributo (incluido el atributo **class**) aparecerán en el conjunto de entrenamiento.

   El constructor no devuelve nada pero crea y almacena el árbol de clasificación como un atributo del objeto para poder usarlo para clasificar más adelante.

2. clasifica(self, instancia)
   A partir de una instancia (diccionario que contiene los valores para cada atributo {atributo_1: valor_1, ..., atributo_n: valor_n } sin incluir la clase **class**) devuelve una cadena de texto con la clase que predice el árbol de clasificación.

3. test(self, fichero)
   Este método sirve para medir la tasa de aciertos del clasificador previamente entrenado. Para ello recibe un conjunto de test almacenado en fichero, que tiene el mismo formato CSV con cabecera que el fichero usado en el constructor, y clasifica cada instancia de test leída. Se debe mostrar la siguiente información por pantalla:

   - Valores de los atributos de la instancia completa, incluida la clase.
   - Clase que predice el clasificador *ID3*.
   - Una linea indicando si la clasificación acierta o falla.

   El método test devolverá una tupla (aciertos, total, tasa) conteniendo el número total de instancias de test clasificadas correctamente, el número total de instancias de test y la tasa de aciertos (*aciertos/total*).

La clase ID3 se usaría de la siguiente manera:

```
id3 = ID3('railway.train')
clase = id3.clasifica({'season':'winter','rain':'heavy','wind':'high','day':'saturday'})
print('Clase:', clase)
(aciertos, total, tasa) = id3.test('railway.test')
print('Aciertos:', aciertos, 'Total:', total, 'Tasa:', tasa)
```

Considerando un fichero railway.test de 2 instancias mostraría por pantalla algo como:

```
Clase: very late

Test:
{'season': 'winter', 'wind': 'high', 'day': 'weekday', 'rain': 'heavy'}−very late
Clase predicha: very late
    −−> Acierto
```

[1] La biblioteca csv.DictReader os puede ayudar a analizar ficheros CSV con cabecera.

```
digraph tree {
  //nodos
  sexo [label="Sexo",shape="box"];
  f1 [label="Futbol"];
  exp [label="Expediente",shape="box"];
  f2 [label="Futbol"];
  p [label="Padel"];

  //aristas
  sexo -> f1 [label="M"];
  sexo -> exp [label="F"];
  exp -> f2 [label="<A"];
  exp -> p [label=">=A"];
}
```

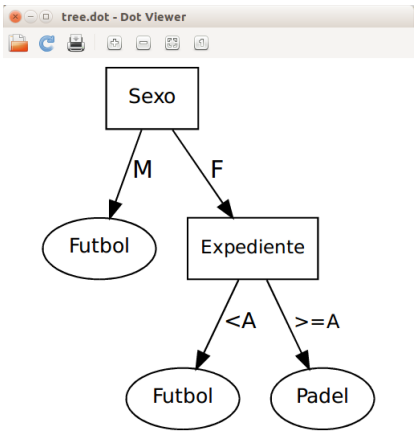

Figura 1: Fichero DOT y su visualización con xdot

```

{'season': 'winter', 'wind': 'high', 'day': 'sunday', 'rain': 'heavy'}−very late
Clase predicha: very late
    −−> Acierto

Aciertos: 2 Total: 2 Tasa: 1.0
```

Notas: Podéis representar el árbol de clasificación como prefiráis, aunque éste se puede construir de manera sencilla utilizando únicamente tuplas anidadas o combinando tuplas y diccionarios. Los nodos internos del árbol de clasificación deben almacenar el nombre del atributo sobre el que realizan la decisión, y cada arista debe estar decorada con el valor concreto del atributo. Si preferís crear vuestra propia clase de árboles también es una buena opción.

B) Visualización del árbol de clasificación [10 %]

Almacenar los árboles de clasificación generados por la función id3 en un fichero en formato *DOT*. DOT es un formato textual muy sencillo para la descripción de grafos, que luego pueden visualizarse con herramientas como `xdot`. Podéis encontrar más información sobre este lenguaje en `https://graphviz.gitlab.io/_pages/doc/info/lang.html` y `https://en.wikipedia.org/wiki/DOT_language`. Un ejemplo muy simple que muestra todos los elementos del lenguaje DOT que necesitáis aparece en la Figura 1.

Para este apartado se pide extender la clase ID3 con un método:

- save_tree(self, fichero)
  Vuelca el árbol de clasificación generado y almacenado en el objeto en un fichero de texto en formato DOT. El método no devuelve nada.

El nuevo método de la clase ID3 se usaría de la siguiente manera:

```
id3 = ID3('railway.train')
id3.save_tree('railway.dot')
```

2. SparkML [20 %]

En este apartado deberéis explorar SparkML (`https://spark.apache.org/docs/latest/ml-guide.html`), la biblioteca de aprendizaje automático de Spark, para construir clasificadores y evaluar su ta-

sa de aciertos. **Es importante que utilicéis la versión de SparkML basada en DataFrames** (spark.ml) y no la que se basa en RDDs, puesto que esa versión está en modo mantenimiento y desaparecerá en un futuro.

Como conjunto de datos tomaremos el *Adult Data Set* obtenido de `https://archive.ics.uci.edu/ml/datasets/Adult`, en el que se persigue predecir cuándo los ingresos anuales serán mayores que $50K a partir de datos del censo. Este conjunto ya dispone de un fichero de entrenamiento (adult.train) y otro de test (adult.test), así que no debemos preocuparnos por esto. He «arreglado» un poco los ficheros que he subido al Campus Virtual añadiendo la cabecera, eliminando espacios molestos y corrigiendo la representación de la clase en el fichero de test (cada línea tenía un punto final inesperado). Sin embargo, todavía tendréis que realizar algunas tareas de preprocesado en los DataFrames que carguéis y en la propia tubería de clasificación.

En este ejercicio deberéis realizar los siguientes pasos:

1. Cargar los ficheros en DataFrames y explorarlos para conocer cada uno de los atributos: rango de valores, tipo de atributo, posibles valores que faltan... Os puede ayudar la explicación detallada en `https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.names`.

2. Limpiar los conjuntos de datos para dejar únicamente las **instancias completas**.

3. Crear varias *tuberías* para clasificar utilizando **al menos 3 clasificadores diferentes** de los disponibles en `https://spark.apache.org/docs/latest/ml-classification-regression.html`. Una tubería (objeto de tipo pyspark.ml.Pipeline) es una secuencia de etapas encadenadas (*transformadores*) que se aplican sobre un DataFrame y que terminan en un *estimador* (en nuestro caso un clasificador). Podéis encontrar más información sobre las tuberías en `https://spark.apache.org/docs/latest/ml-pipeline.html`. Tened en cuenta que los DataFrames de los que partís tienen columnas categóricas (cadenas de texto) y otras continuas (enteros o reales), pero los clasificadores necesitan que toda la información que vais a usar para clasificar una instancia esté almacenada como vector numérico en una única columna. Por ello necesitaréis realizar algunas etapas de extracción/transformación de atributos de las disponibles en `https://spark.apache.org/docs/latest/ml-features.html`.

4. Una vez tengáis varias tuberías de clasificación, debéis entrenarlas con el conjunto de entrenamiento y evaluar su calidad con el conjunto de test. Como métrica de calidad utilizaremos la tasa de aciertos (proporción de instancias correctamente clasificadas de entre todas las probadas). Este cálculo se puede hacer «a mano» sin mucho problema, pero es bastante más cómodo usar un objeto de la clase pyspark.ml.evaluation.MulticlassClassificationEvaluator.

Además del fichero Python con el código fuente, debéis crear un **fichero PDF** en el que expliquéis brevemente las distintas etapas que forman cada una de las tuberías e indiquéis la tasa de acierto obtenida para cada una.

3. Recuperación vectorial [20 %] (Objetivo mínimo)

Completar el esqueleto VectorialIndex.py para implementar la clase VectorialIndex que procesa una colección de documentos, crea un índice invertido con pesos TF-IDF y lo usa para acelerar consultas vectoriales y booleanas (únicamente conjunción) sobre la colección. La clase debe contener al menos los siguientes métodos:

- __init__(self, directorio, stop=[ ])
  Recorre **recursivamente todos los ficheros** que hay en directorio y crea un índice invertido para relacionar las palabras con una lista de parejas (documento, peso). Los pesos se deben calcular

usando la técnica TF-IDF, y los documentos dentro del índice deben representarse como números enteros en lugar de rutas completas para minimizar el espacio que ocupan en memoria. El constructor también acepta una lista de palabras vacías stop que se deben ignorar al procesar los fichero y que por tanto no deben aparecen en el índice invertido. Esta lista debe almacenarse para procesar las consultas más adelante. Para que todos consideremos la misma *definición* de palabra, se debe usar la función extrae_palabras que aparece en el esqueleto para extraer las palabras de una línea de texto. Es importante que el índice invertido almacene la **mínima información necesaria** para minimizar su tamaño.

- consulta_vectorial(self, consulta, n=3).
  Dada una consulta representada como una cadena de palabras no repetidas separadas por espacios, devuelve una lista de parejas (ruta_fichero, relevancia) con los n resultados más relevantes usando el modelo de recuperación vectorial.

- consulta_conjuncion(self, consulta)
  Dada una consulta representada como una cadena de palabras no repetidas separadas por espacios que se entiende como una conjunción, devuelve una lista de nombres de fichero con todos los resultados en los que aparecen **todas las palabras** de la consulta.

En los métodos consulta_vectorial() y consulta_conjuncion() el índice invertido se debe utilizar de la manera más eficiente posible para resolver la consulta. Las palabras de la consulta que aparecen en la lista de palabras vacías usada al crear el índice deben ignorarse.

La clase VectorialIndex se utilizaría de la siguiente manera:

```
stoplist = ['the', 'a', 'an']
i = VectorialIndex('/home/pepe', stoplist)
print(i.consulta_vectorial("patatas fritas", 2))
print(i.consulta_conjuncion("patatas fritas McDonalds"))
```

Y produciría la siguiente salida por pantalla:

```
[('/home/pepe/documentos/patatas.txt', 0.532323), ('/home/pepe/recetas/buenas/fritos.txt',
    0.5172170959488461)]
['/home/pepe/recetas/buenas/fritos.txt', '/home/pepe/documentos/patatas.txt', '/home/pepe/comida.txt', '
    /home/pepe/recetas/mejores.txt']
```

4. Índice invertido completo [30 %]

A) Creación y utilización del índice [10 %]

Completar el esqueleto CompleteIndex.py para implementar la clase CompleteIndex que procesa una colección de documentos, crea un índice invertido completo y lo usa para acelerar consultas de frase sobre la colección. La clase tiene un interfaz similar al índice vectorial y debe contener al menos los siguientes métodos:

- __init__(self, directorio)
  Recorre **recursivamente todos los ficheros** que hay en directorio y crea un índice invertido completo para relacionar cada palabra con una lista de tuplas (numero_doc, l_pos), donde numero_doc es el número de documento y l_pos contiene una lista de **diferencias** entre posiciones. Como granularidad usaremos el número de palabra dentro del documento empezando en 1. Al igual que en el apartado anterior, se debe usar la función extrae_palabras que aparece en el esqueleto para extraer las palabras de una línea de texto.

- consulta_frase(self, frase).
  Dada una consulta representada como una cadena de palabras separadas por espacios, devuelve una lista con los nombres de los ficheros en los que aparece **exactamente** esa frase.

B) Compresión del índice [20 %]

Extender la clase CompleteIndex del apartado anterior para incorporar compresión en la **lista de diferencias entre posiciones** del índice completo. Para ello incluiremos un nuevo parámetro en el constructor indicando qué codificación aplicar y añadiremos un método nuevo:

- __init__(self, directorio, compresion=None)
  El nuevo parámetro compresion contendrá uno de los siguientes valores:

  1. None: No aplicar ninguna compresión, es decir, genera el índice igual que el apartado A.
  2. 'unary': Codificación en formato unario.
  3. 'variable−bytes': Codificación en formato de bytes variables.
  4. 'elias−gamma': Codificación en formato de Elias-γ.
  5. 'elias−delta': Codificación en formato de Elias-δ.

  Cuando se utilice compresión el índice invertido completo no almacenará las diferencias de posiciones en una lista Python sino que codificará la secuencia como una lista de bits usando un objeto bitarray.

- num_bits(self)
  Devuelve el número de bits utilizados para representar todas las listas de diferencias de posiciones en el índice invertido completo. Se puede obtener el número de bits de un bitarray simplemente usando la función **len**. Para el caso de un índice creado con compresion = None el número de bits se calculará considerando que cada elemento de la lista de diferencias de posiciones ocupa 32 bits.

- consulta_frase(self, frase).
  Este método se comportará igual que antes pero tendrá en cuenta la posible compresión utilizada. A la hora de resolver consultas se debe perseguir la eficiencia, es decir, se debe decodificar únicamente aquellas listas de diferencias de posiciones que son relevantes para la consulta.

Además de los cambios en el código, en este apartado es necesario crear un PDF para comparar compresiones. Para ello se creará un índice invertido completo de la colección 20news−18828 con las 5 compresiones posibles (dependiendo de la eficiencia de vuestro código y del ordenador utilizado puede necesitar bastantes minutos) y se calculará el número de bits empleado en representar las listas de diferencias de posiciones. El PDF contendrá el número de bits para cada valor de compresión y expondrá de manera razonada qué compresión sería la más adecuada para la colección.