ITAM Text Miner

Felipe Gerard

2 de noviembre de 2015

0. El problema por resolver

1. Orquestación

Es importante que un sistema de análisis de esta escala sea robusto y confiable. Para ello se optó por utilizar el orquestador luigi, que fue desarrollado por Spotify en el lenguaje `Python`. Este esquema tiene varias ventajas sobre los scripts simples, algunas de las cuales se enumeran a continuación:

1. *Modularidad:* El código se segmenta en pasos bien definidos, lo que facilita su mantenimiento y expansión.
2. *Robustez:* Se requiere que todos los pasos serialicen sus resultados a disco, lo que hace que el sistema sea resistente a fallos y que no tenga que repetir todo el proceso después de un error.
3. *Idempotencia:* Si se programa correctamente, los procesos sólo corren una vez. Incluso si se corre el proceso una segunda vez, sólo corre lo que no haya sido corrido antes.
4. *Paralelismo:* Es bastante sencillo paralelizar los procesos que convenga para utilizar los núcleos de un servidor grande de manera eficiente.

Obtener estas ventajas es relativamente simple con `luigi`. Lo único que se requiere es que los pasos tengan entradas y salidas bien definidas; un proceso únicamente requiere sus dependencias para correr y debe arrojar los resultados que necesite la siguiente sección.

`luigi` corre nativamente en `Python`, por lo que se puede hacer cualquier cosa que se pueda hacer en `Python`. Esto incluye llamar códigos en `shell`, `R` y muchos otros lenguajes. La mayoría de los procesos se hicieron directamente en `Python` por simplicidad, pero algunos se dejaron en otros lenguajes por diversas razones de conveniencia.

A continuación se describe a grandes rasgos los pasos del *pipeline*. Cabe mencionar que esta descripción es para entender el proceso y *no* es una descripción técnica. Internamente hay algunos procesos que se efectúan en un orden ligeramente distinto al aquí presentado, pero resultaría más complicado de entender, de modo que presentamos una versión didáctica.

a. Extracción de texto

En esta sección definimos a grandes rasgos los diversos componentes del *pipeline* de datos, desde los archivos en PDF crudos hasta las salidas necesarias para mostrar en el producto final.

(i) Pegado de PDFs en archivos de 50 MB

–> Petri

(ii) Extracción de texto a partir de PDFs y detección de idioma

Input: PDFs crudos, en el formato descrito abajo.

Output: Libros en formato texto tal cual fue extraido, archivos JSON con extractos de los libros.

Paquetes relevantes: `pdfminer`, `nltk`

El proceso empieza con una carpeta general en la que deben estar todos los PDFs. Adentro de ella debe haber una carpeta por libro que contenga los PDFs de las diversas hojas, por ejemplo: `pdf/libro_de_arte/hoja_i.pdf`

Se extraen los textos utilizando el paquete `pdfminer` de `Python` y se juntan los resultados de todas las hojas en un solo archivo por libro. Posteriormente se detecta el idioma de cada texto utilizando la metodología propuesta por la UNAM, es decir, se observa qué porcentaje de las *stopwords* de cada idioma tiene un texto y se le asigna el idioma del que tenga mayor porcentaje. Después de la detección, los libros en versión texto se meten en una carpeta según su idioma. Adicionalmente, se guardan dos archivos de metadatos, uno con los idiomas registrados y otro con una relación entre los libros y sus idiomas correspondientes.

Por practicidad se optó por extraer los párrafos representativos en este mismo proceso. Para ello se tomaron 3 secciones de aproximadamente 500 letras al 10%, 50% y 90% de avance del libro aproximadamente. El criterio tiene cierta flexibilidad e intenta encontrar párrafos completos (que empiecen en salto de línea y mayúscula), aunque esto en general es poco preciso por la imperfección del OCR efectuado al momento de digitalizar los libros.

Nota 1: Esta extracción se considera que tiene nivel de limpieza "nulo", por lo que los resultados de la extracción se meten a la carpeta `raw` dentro de la carpeta de textos.

Nota 2: Por razones internas del funcionamiento de `luigi`, es imposible checar de antemano a qué carpeta debe ir cada libro, ya que no se puede saber su idioma antes de procesarlo. Para darle la vuelta a esta limitación, se generó una carpeta con un archivo de metadatos por libro, `libro.meta`, que sirve para que `luigi` pueda checar el árbol de dependencias.

(iii) Limpieza de textos

Input: Libros en formato texto crudo.

Parámetros: Idioma(s) a procesar, nivel(es) de limpieza a obtener.

Output: Libros en formato texto limpio, según la especificación.

Paquetes relevantes: `nltk`, `unicodedata`

Según el nivel de limpieza elegido, se genera una estructura similar a la de los textos crudos (`raw`). Los dos niveles son incrementalmente más "limpios":

- Limpieza básica (`clean`):
  - Se quitan los acentos.
  - Se quitan los saltos de página.
  - Se pasa todo a minúsculas.
  - Se quitan los caracteres especiales que queden después de quitar los acentos.
  - Se quitan palabras cortas (de 3 o menos caracteres).
  - Se quitan palabras con algún caracter repetido 3 o más veces.
- Limpieza avanzada (`stopwords`):
  - Todo lo de la limpieza básica.
  - Se quitan las palabras poco informativas (*stopwords*).

Los dos pasos anteriores se generan de manera incremental, lo que significa que si se genera uno más avanzado, siempre se generan todos los anteriores. Cada nivel de limpieza genera una estructura similar a la generada en el paso de extracción (i.e. a la carpeta `raw`), pero con nombre y características según sea el caso.

Nota: Nuevamente se utilizó un esquema de dependencias artificiales como en la extracción.

b. Minería de textos

Una vez habiendo extraido y preparado los textos se puede empezar el proceso de minería. De aquí en adelante los procesos generan varios resultados, uno para cada nivel de limpieza y para cada idioma que se pida al comenzar el programa.

(i) Vectorización de textos

Input: Textos de un idioma dado, con un nivel de limpieza dado.

Output: Un diccionario con los conteos de palabras y el corpus vectorizado (matriz términos - documentos).

Paquetes relevantes: `gensim`

Los algoritmos de minería necesitan entradas numéricas que viven en espacios vectoriales. Por ello el primer paso del análisis es vectorizar los textos. En el paquete `gensim` este proceso se divide en dos etapas: (1) generar un `diccionario` con las palabras válidas dentro de la colección, y (2) generar una matriz términos - documentos, que contiene los conteos de cada término en cada documento. En `gensim` se le llama `corpus` a la matriz términos - documentos.

(ii) Latent Dirichlet Analysis: Clusters por tópicos

Input:

Parámetros:

Output:

Paquetes relevantes: –> Lechuga

(iii) Latent Semantic Indexing: Documentos similares

Input: Un diccionario y un corpus.

Parámetros: Número de dimensiones latentes a utilizar.

Output: Modelo vectorizado transformado con TF-IDF. Listas los libros similares a cada libro, en diversos formatos (JSON, CSV, HTML, XML, red).

Paquetes relevantes: `gensim`, `pandas`, `markdown`, `pandoc` (es de `bash`)

El Análisis de Semántica Latente o LSI se lleva a cabo en varias etapas:

1. Se pondera el corpus con el modelo TF-IDF para dar más relevancia a las palabras poco comunes.
2. Se genera el modelo de semántica latente.
3. Se calculan las similitudes entre los documentos y se guardan las más relevantes.

Al finalizar estos procesos se tiene varios archivos con información de los modelos de `gensim` y las salidas en diversos formatos. Cada uno de ellos tiene un fin distinto. Por ejemplo, el HTML y la red permiten visualizar fácilmente los resultados, con el fin de que un experto elija los parámetros óptimos del modelo, mientras que el JSON y el CSV permiten explotar la información en otros sistemas. El XML sirve para mostrar los resultados en la página web del producto.

c. Paquete

Con el fin de obtener un producto redondeado en lugar de una colección de scripts, se optó por agruparlos en el paquete `itm` (ITAM Text Miner). De este modo la instalación es relativamente sencilla y checa la mayoría de las dependencias de `Python` automáticamente. Además esto nos permite guardar ejecutables en el `PATH` del servidor, lo que permite correr los códigos desde cualquier ruta.

(i) Instalación

La instalación del paquete en Ubuntu es bastante simple. Se requiere instalar `Python`, `R` y algunas pocas librerías de `Python` usando `apt-get` y las demás se instalan solas al instalar la librería `itm`.

```
# Necesario para instalar R >= 3.1
echo "deb http://cran.rstudio.com/bin/linux/ubuntu trusty/" >> /etc/apt/sources.list && \
    sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E084DAB9

# Actualizar índices de apt-get
sudo apt-get update && \
    apt-get upgrade # Necesario para instalar R >= 3.1

# Instalar Python   2.7 y PyPi
sudo apt-get install -y \
    python2.7-dev \
    python-pip

# Instalar R (debe ser >= 3.1)
sudo apt-get install -y \
    r-base \
    r-base-dev && \
    R --version

# Librerías de álgebra lineal, Fortran y compilador de Markdown
sudo apt-get install -y \
    libblas-dev \
    liblapack-dev \
    gfortran \
    pandoc

# Librerías de Python que no se instalan bien solas
sudo apt-get install -y \
    python-numpy \
    python-scipy

# Instalar el paquete itm   y sus dependencias de Python
PACK = itm-0.1 # U otra versión
cd /ruta/a/paquete
echo ${PACK}.tar.gz && \
    cd /tmp && \
    tar xzf ${PACK}.tar.gz && \
    cd ${PACK} && \
    python setup.py install

# Instalar librerías de R y bajar stopwords de nltk (no se bajan al bajar el paquete)
```

```
R -e 'install.packages(c("dplyr", "optparse", "networkD3"), repos = "http://cran.us.r-project.org")'
python -c "import nltk; nltk.download('stopwords')"
```

(ii) Breve guía de uso