



Proyecto 1

Profesor: Cristóbal Rojas
Ayudante: Diego Rodriguez

Introducción

En este proyecto se pondrá a prueba lo aprendido sobre vectores, su manipulación y usos. En particular, se desarrollará la clase `Vec` entregada en el material de *Coding the Matrix* (encontrado en <http://resources.codingthematrix.com/>) que permite modelar vectores generalizados. También se desarrollará la clase `ListVec`, para el modelamiento de vectores como listas. Se utilizarán dichas clases para elaborar un sistema recomendador de películas y para abordar el problema de buscar una aguja en un pajar. Este proyecto podrán realizarlo en grupo, con un **máximo de 3 integrantes por grupo**. El plazo máximo para la entrega del proyecto es el **26 de Octubre a las 23:59 hrs.**

Clase Vec (15 pts.)

El archivo `vec.py`, encontrado en la sección *The Vector: Problems* (del material de *Coding The Matrix*) contiene el esqueleto de una clase `Vec` que, como su nombre lo indica, permite crear y manipular vectores generalizados. Para usar esta clase debes primero importarla (`from vec import Vec`). Para instanciar un objeto `Vec`, debes definir un cierto dominio en forma de conjunto, y una función que mapea elementos del dominio a escalares. Por ejemplo, si D es el dominio $\{ 'a', 'b', 'c', 'd' \}$, entonces el D -vector v asociado a la función

$$\begin{aligned} 'a' &\rightarrow 2 \\ 'b' &\rightarrow 0 \\ 'c' &\rightarrow 1 \\ 'd' &\rightarrow 3 \end{aligned}$$

se define de la forma

$$v = \text{Vec}(D, \{ 'a':2, 'c':1, 'd':3 \})$$

Recuerden que, para ahorrar memoria, usamos la convención de no incluir en el diccionario los pares *llave:valor* cuando el valor es 0.

Tu primera tarea consiste en rellenar los métodos de la clase `Vec` especificados en el archivo `vec.py`. Estos métodos son:

- `getitem(v, k)`
- `setitem(v, k, val)`
- `equal(u, v)`
- `add(u, v)`
- `dot(u, v)`
- `scalar_mul(v, alpha)`
- `neg(v)`

Una descripción detallada de cada uno, junto al retorno exigido, se muestran en los `docstrings` de los métodos (las descripciones que se entregan inmediatamente después de definir cada función). A continuación, se presentan algunas consideraciones que es necesario tener en mente:



- Cada método incluye una línea de tipo **assert**. Dicha expresión se encarga de asegurar que la condición que le sigue se cumpla. Por ejemplo, la expresión **assert u.D == v.D** para dos vectores u y v chequea que los dominios de ambos vectores sean iguales. En caso de que la condición no se cumpla, el programa se caerá y arrojará un error reflejando que los vectores no comparten el dominio. No se probarán casos en los que **assert** incluya una condición falsa.

Se recomienda rellenar el método en las líneas posteriores a la expresión **assert**.

- Al final del archivo **vec.py**, luego de la línea de símbolos **#**, se define la clase **Vec** propiamente tal, que asume que los métodos anteriores han sido creados. Para efectos de esta tarea, pueden considerar esta parte como una caja negra, que consiste básicamente en tomar los métodos rellenos y definir la clase.
- Es importante que te asegures que tus métodos funcionen correctamente, pues los usarás en la segunda parte del proyecto.
- La gran ventaja de los objetos **Vec** es que la sintaxis para operarlos es como la usual para operar números: por ejemplo, para calcular el producto punto entre dos objetos **Vec**, digamos u y v , basta con escribir **u*v**, en vez de **dot(u, v)**, que es la función que tú programarás y que la clase **Vec** usará solo internamente.

Clase ListVec (10 pts.)

En el archivo **listvec.py** que podrás encontrar en el drive del curso, se entrega un esqueleto tentativo para la construcción de la clase **ListVec**, que hereda de la clase **list** de Python. De este modo, para instanciar un objeto **ListVec**, la sintaxis es análoga a la utilizada para crear una lista a través del método **list()**. Por ejemplo, si el vector a instanciar es $v = [1, 2, 3] \in \mathbb{R}^3$, entonces el vector v se define de la forma:

$$v = \text{ListVec}([1, 2, 3])$$

La clase debe contar con los siguientes métodos:

- **vec_sum(self, vec: ListVec)**: Método que permite efectuar la suma vectorial entre dos instancias de la clase **ListVec**. Debe retornar una instancia de la clase **ListVec**.
- **scalar_mul(self, scalar: float)**: Método que permite efectuar la ponderación por escalar para una instancia de la clase **ListVec**. Debe retornar una instancia de la clase **ListVec**.
- **dot(self, vec: ListVec)**: Método que permite efectuar el producto punto entre dos instancias de la clase **ListVec**. Debe retornar un **float**.
- **norm(self)**: Este método debe retornar un **float** correspondiente a la norma euclidiana (o norma 2) del vector.

Debes asegurarte de que tu clase funciona correctamente.

Sistema Recomendador de Películas (25 pts.)

Contexto

Existe una infinidad de aplicaciones que se basan en vectores. Ejemplo de esto son prácticamente todos los algoritmos de *Machine Learning*, que mezclan tópicos como Álgebra Lineal y Estadística, y el área de las Ciencias de Datos que investiga sobre Sistemas Recomendadores como los que se encuentran presentes en las plataformas *Amazon*, *Facebook*, *Netflix*, etc. En general, todo sistema que recomienda contenido en base



a las preferencias de los usuarios (pueden ser recomendaciones globales o personalizadas). Como ejemplo, *Amazon* ofrece productos en base a las preferencias del usuario, las cuales pueden ser extraídas en forma de *rating* explícito (calificación que un usuario haya asignado a un producto), o en forma de *rating* implícito, el cual se basa en el comportamiento de un usuario (por ejemplo, la cantidad de tiempo que visita la página de un determinado producto).

Dentro del área de Sistemas Recomendadores, se utilizan diversas técnicas para predecir las preferencias/gustos de los usuarios. Dentro de las más simples e intuitivas se encuentran los métodos basados en la idea de usar los gustos de usuarios que son *vecinos* del usuario al que se le quieren hacer recomendaciones, en el sentido que sus gustos son *similares*. Esta técnica es ampliamente utilizada en contextos de predicción (no solo en Sistemas Recomendadores), y suele ser denominada KNN, en referencia al término *k-Nearest Neighbors* (o los k vecinos más cercanos).

Pero, ¿cómo se puede saber qué tan similares son los gustos de dos usuarios? Existen múltiples medidas de similitud, basadas en diferentes nociones de *distancia*. Para este proyecto utilizaremos una medida llamada **similitud de coseno**, la cual en términos prácticos mide qué tan “alineados” están dos vectores dados. Formalmente, si u y v son los vectores de gustos de dos usuarios, entonces definimos la *similitud* entre u y v como

$$\text{sim}(u, v) = \frac{u \cdot v}{(\sqrt{u \cdot u})(\sqrt{v \cdot v})} \quad (1)$$

donde $u \cdot v$ es el producto punto entre u y v . Es importante notar que, como en este caso los vectores no contienen componentes negativas, el resultado de esta medida es siempre positivo. Además, el factor en el denominador hace que el valor de esta medida sea como máximo 1 (cuando $u = v$). La idea es que mientras más alineados estén los vectores u y v , más cercano a 1 será el valor de $\text{sim}(u, v)$, lo que interpretamos como una mayor similitud entre u y v .

Ahora, consideremos el contexto de una plataforma que ofrece *streaming* de un total de n películas a m usuarios, y supongamos que dicha plataforma cuenta con datos de la forma (**user id**, **movie id**, **rating**), donde **rating** es un número entre 1 y 5 (como las estrellas de *Netflix*). Luego, para cada usuario i (con $i \in \{1, \dots, m\}$), podemos definir su vector de gustos como $u_i = (\text{rating movie}_1, \dots, \text{rating movie}_n)$. De esta forma, se estima que usuarios que tienen una similitud alta entre sus vectores de gustos, tienen preferencias similares.

MovieLens

MovieLens consiste en un recomendador de películas que se remonta a la década de los '90. Su funcionamiento se basa en *ratings* explícitos, es decir, en recomendar películas en base a los *ratings* que han entregado los usuarios. Para este proyecto, recibirás una pequeña muestra de la base de datos, conformada por 943 usuarios, 1.682 películas y 80.000 *ratings*.

Los archivos de la base de datos son los siguientes:

- **ratings.csv**: Es un archivo separado por comas, que contiene datos de la forma (**user_id**, **movie_id**, **rating**). El primer elemento corresponde al identificador único de un usuario, el segundo al identificador único de una película, y el tercero al **rating** entregado por el usuario a dicha película.
- **movies.csv**: Es un archivo separado por comas, que contiene pares de la forma (**movie_id**, **movie_name**). Al igual que en el archivo anterior, el primer elemento corresponde al identificador único de la película, mientras que el segundo corresponde al nombre de dicha película.



- Recuerda que para leer un archivo `file.csv`, separar el contenido de sus filas y ponerlo todo por ejemplo en una lista llamada `archivo`, puedes usar:

```
archivo=[i.strip().split(',') for i in list(open('file.csv'))]
```

Por desarrollar

En base a lo anterior, se te pide desarrollar los siguientes puntos:

- Para cada usuario, crea su *vector de gustos*, utilizando la clase `Vec`. Para esto, debes definir correctamente el dominio (*ids* de las películas), y la función `movie_id` \rightarrow `rating` (utilizando un diccionario). Si un usuario no ha calificado alguna película, debes asumir que el *rating* que asigna a dicha película es 0 (nota que esto se hace automáticamente gracias a la clase `Vec`). Guarda los vectores en un diccionario llamado `users`, utilizando la sintaxis `users[user_id] = user_vec` de modo que `users[n]` sea el *vector de gustos* del usuario con *id* n . **(5 pts.)**
- De manera equivalente, para cada película crea su vector de la clase `Vec` con los ratings que todos los usuarios le dieron a esta película. Debes definir correctamente el dominio (*ids* de los usuarios), y la función `user_id` \rightarrow `rating`. Nuevamente, si un usuario no ha calificado una película, debes asumir que el *rating* es 0. Guarda los vectores en un diccionario llamado `movies`, utilizando la sintaxis `movies[movie_id] = movies_vec`, de modo que `movies[movie_id]` sea el vector de la película con este *id*. **(5 pts.)**
- Define una función `vecinos(users, user_id, k)` que recibe un diccionario `users` (donde `users[i]` es el vector de gustos del usuario i), el identificador `user_id` de un usuario en el diccionario `users` y un entero positivo k . La función debe, en base a la noción de similaridad de coseno (1), encontrar los k usuarios del diccionario `users` más similares a `user_id` (pero diferentes de este!), y retornarlos en una lista en orden decreciente de similaridad utilizando el formato `[(user_i, similarity_i), ..., (user_k, similarity_k)]` (Nota que es una lista de tuplas). Se asumirá que $k \in \{0, \dots, |\text{users}| - 1\}$ y que el *id* del usuario es válido. **(7 pts.)**

HINT: Recuerda que para ordenar un diccionario `dict` de forma decreciente según sus valores, puedes usar

```
sorted(dict.items(), key = lambda x:x[1], reverse=True)
```

que entrega una lista `[(key,value),...]` en el order deseado.

- Responda: ¿Cómo se puede ocupar la información obtenida en (c) para generar recomendaciones para el usuario? Fundamente su respuesta. **(4 pts.)**
- Reflexione: ¿cuál es la ventaja de enfrentar este problema utilizando la clase `Vec`? ¿Qué complicaciones podría tener enfrentar este mismo problema utilizando la clase `ListVec`? **(4 pts.)**

HINT: Incluya el concepto de *sparsity* en su respuesta.

Una aguja en un pajar (10 pts.)

Otro de los problemas que se pueden abordar haciendo uso de la similaridad del coseno, es la búsqueda de una aguja en un pajar. En la sección 2.9.3 *Measuring similarity* del libro *Coding the Matrix Ed. 1* (páginas 115 - 118) se presenta este problema en el contexto de buscar las apariciones de un pequeño segmento de audio en un audio más grande.

Lea las páginas indicadas del libro y luego desarrolle:



- (a) Incluya en la definición de la clase `ListVec` el método `similarity(self, vec: ListVec)` que le permita calcular la similaridad del coseno para dos instancias de la clase `ListVec`. **(4 pts.)**
- (b) Suponga que ambos segmentos de audio ya se encuentran instanciados como objetos de la clase `ListVec`. Defina la función `needle_search(v: ListVec, u: ListVec)` que recibe dos instancias de la clase `ListVec` y retorna los 3 segmentos de `u` que tienen mayor similaridad con `v` (estos segmentos deben corresponder a instancias de la clase `ListVec`). Puede asumir sin problemas que `len(v) < len(u)` y que `u` es lo suficientemente largo como para cumplir con las exigencias del problema. Explique cómo funciona su método. **(4 pts.)**
- (c) Reflexione ¿Qué tipo de aplicaciones podría tener una función como la descrita en el ítem (b)? **(2 pts.)**

Consideraciones adicionales

1. Debes entregar tu solución tanto de la clase `Vec` como de la clase `ListVec` como un archivo de `Python` con nombre `listvec.py` y `vec.py` respectivamente.
2. El desarrollo de las preguntas debe ser entregado en un archivo `Jupyter Notebook`. Puedes importar la clase `Vec` y `ListVec` desde tu *notebook* sin problemas. Se evaluará el orden del notebook (penalización máxima de **5 pts.** por falta de orden).
3. No está permitido el uso de librerías externas.
4. La fecha de entrega del proyecto es el **26 de Octubre hasta las 23:59 hrs.**