

Realicen otras implementaciones con el objetivo de determinar las ventajas y desventajas de cada implementación y/o librerías.

```
Distancia Manhattan

[135] from scipy.spatial.distance import cityblock
      def Manhattan_lib(x1,x2):
          return cityblock(x1, x2)

Distancia Euclidiana

[163] from scipy.spatial.distance import euclidean
      def Euclidiana_lib(x1,x2):
          return euclidean(x1,x2)

Distancia Correlacion de Pearson

def Pearson_lib(x1,x2):
    return np.corrcoef(x1, x2)[0,1]

Distancia Similitud de coseno

def Coseno_lib(x1,x2):
    return 1 - spatial.distance.cosine(x1,x2)
```

En resumen los resultados fueron los mismos excepto en la correlación de pearson, el tiempo de ejecución que tomó comprobando fue que usar las operaciones numpy en una función propia es más rápido que usar las librerías propuestas por scipy.

Pero la diferencia es que la cantidad de memoria usada es menor.

Si se usa las operaciones usando python con iteraciones y de más es lento en comparación con scipy.

La ventajas de usar librerías en esta implementación (en este caso use la librería scipy)

- Es muy útil para reutilizar data de tipo numpy
- En memoria usar una librería es más útil debido a que las operaciones que se realizan en una función propia almacén valores según en cada operación

La desventajas de usar librerías en esta implementación

- Es el tiempo que tarda la ejecución esto debido a que scipy está escrito en python
- No es posible descartar valores nan y se tiene que colocar a cero para que las distancias estén correctas.
- Al ser implementado en python almacena los objetos de tipo heterogéneos por ello es que demora cuando encuentra un tipo nan porque debe primero verificar que tipo de datos y de ahí operar.

Respecto a la correlación de Pearson en scipy es que pide que por lo menos existan más de dos calificaciones por cada usuario, por ello se utilizó la función `np.corrcoef(x,y)`. Esta función retorna valores redondeados por lo tanto afecta mucho y cambian los valores más cercanos a un usuario.

- Resultados del algoritmo knn usando implementación de la correlación de Pearson

```
start_time = time()
print(Knn(data_ratings,1,Pearson,3))
elapsed_time = time() - start_time
print("Tiempo transcurrido: %.10f milisegundos." % (elapsed_time *1000))
|
```

```
[(550, 1.000000000000000016), (146, 1.000000000000000002), (106, 1.0)]
Tiempo transcurrido: 15578.3267021179 milisegundos.
```

- Resultados del algoritmo knn usando la librería numpy de la correlación de Pearson

```
start_time = time()
print(Knn(data_ratings,1,Pearson_lib,3))
elapsed_time = time() - start_time
print("Tiempo transcurrido: %.10f milisegundos." % (elapsed_time *1000))
```

```
/usr/local/lib/python3.7/dist-packages/numpy/lib/function_base.py:2691: Run
c /= stddev[:, None]
/usr/local/lib/python3.7/dist-packages/numpy/lib/function_base.py:2692: Run
c /= stddev[None, :]
/usr/local/lib/python3.7/dist-packages/numpy/lib/function_base.py:2683: Run
c = cov(x, y, rowvar, dtype=dtype)
/usr/local/lib/python3.7/dist-packages/numpy/lib/function_base.py:2542: Run
c *= np.true_divide(1, fact)
/usr/local/lib/python3.7/dist-packages/numpy/lib/function_base.py:2542: Run
c *= np.true_divide(1, fact)
[(106, 1.0), (146, 1.0), (333, 1.0)]
Tiempo transcurrido: 14742.6805496216 milisegundos.
```

Según los resultados en la correlación de pearson sin librerías es que los 3 vecinos más cercanos son :

`[(550, 1.000000000000000016), (146, 1.000000000000000002), (106, 1.0)]`

dónde el id del usuario más cercano al usuario 1 es 550, sin embargo el otro resultado con la función `np.corrcoef(x,y)` son:

`[(106, 1.0), (146, 1.0), (333, 1.0)]`

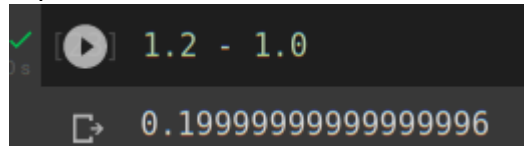
dónde el id del usuario más cercano al usuario 1 es 106, esto debido al redondeo.

Por lo tanto usar `np.corrcoef(x,y)` no es confiable además de que el tiempo de ejecución es mayor debido a que internamente procesa primero una matriz de covarianzas.

Análisis de Decimales.

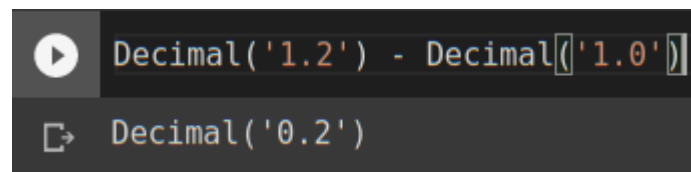
El problema en general es el manejo de los números de punto flotante internamente ya que esto usa un número fijo de dígitos binarios para representar un número decimal. Y por ello conduce a pequeños errores de redondeo.

El tamaño de un flotante es muy grande para almacenar por ejemplo el número 1.2 en binario sería 1.0011001100110011001... y esto se representa como 1.9999999... por lo tanto si el valor flotante opera con el 1.2 en realidad opera con el valor 1.9999999... dándonos un valor inexacto para nosotros.



```
1.2 - 1.0
0.19999999999999996
```

Entonces ante una operación así puede resultar valores errados por ello se probó utilizar la función Decimal.



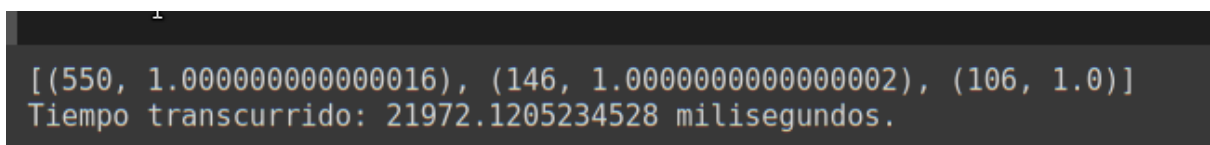
```
Decimal('1.2') - Decimal('1.0')
Decimal('0.2')
```

Y comparando con los resultados utilizando el dataset de 100k son:

- Usando la distancia Manhattan los resultados son similares
- Usando la distancia Euclidiana los resultados son similares

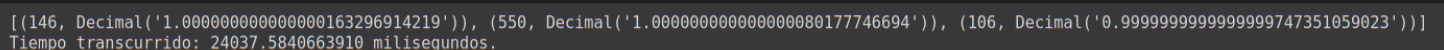
Usando la correlación de Pearson se obtuvieron resultados diferentes:

La siguiente imagen son los resultados usando float, la cual indica que el vecino más cercano es con el id 550



```
[(550, 1.00000000000000016), (146, 1.00000000000000002), (106, 1.0)]
Tiempo transcurrido: 21972.1205234528 milisegundos.
```

La siguiente imagen son los resultados usando la función decimal, la cual indica que el vecino más cercano es el id 146.



```
[(146, Decimal('1.000000000000000163296914219')), (550, Decimal('1.000000000000000080177746694')), (106, Decimal('0.999999999999999747351059023'))]
Tiempo transcurrido: 24037.5840663910 milisegundos.
```

Según estos resultados usando decimales el usuario id 146 es el más cercano mientras que en el otro es el segundo más cercano. Esto porque la cantidad de decimales es mayor y devuelven resultados exactos.

Por lo tanto es preferible usar la función decimal para tener más confianza en los resultados sin embargo tener un gran volumen de datos requiere mayor tiempo de procesamiento al operar números, como se muestra en las imágenes usando la función decimal demora más tiempo que el otro.

La siguiente imagen son los resultados usando float, la cual indica que el vecino más cercano es el id 77

```
[(77, 1.0), (85, 1.0), (184, 1.0)]
Tiempo transcurrido: 21007.7679157257 milisegundos.
```

La siguiente imagen son los resultados usando la función decimal, la cual indica que el vecino más cercano es el id 77.

```
[{77, Decimal('1.000000000000000080177746703')}, {85, Decimal('1')}, {184, Decimal('1')}]
Tiempo transcurrido: 27981.1768531799 milisegundos.
```

Los resultados como el vecino más cercano coincidieron, en este caso el vecino más cercano es el id 77, pero la distancia no. ya que usando la función decimal es mas exacto y contiene mas decimales