Task 1: Entity Resolution

Task 1.1: Dataset Construction

For both datasets, I included their name string, name tokens(tokenized by regex only including words), publish date and year, ISBN13 number, author and page count.

In [ ]:
```
good_ds.generate_dataframe().head(5)
```

	id	name_string	author	page_count	name_tokens	ISBN13	publication_date	publi
0	0	Managing My Life: My Autobiography	Alex Ferguson	531	[managing, my, life, my, autobiography]	9780340728567	2000-08-01	
1	1	I Remember: Sketch for an Autobiography	Boris Pasternak	0	[i, remember, sketch, for, an, autobiography]	9780844627106	None	(It valid
2	2	Betty Boothroyd: Autobiography	Betty Boothroyd	384	[betty, boothroyd, autobiography]	9780712679480	2002-11-01	
3	3	Caddie, A Sydney Barmaid: An Autobiography	Caddie	199	[caddie, a, sydney, barmaid, an, autobiography]	9780725100148	1966-09-15	
4	4	Nureyev: An Autobiography With Pictures	Rudolf Nureyev	160	[nureyev, an, autobiography, with, pictures]	9780340014684	1963-09-15	

Task 1.2: Blocking

Note: In the following mentions, RR will not be introduced for every blocker since the value is way too small.

1. My first blocker is one using name tokens. The original tokenizer does poorly on tokenizing, since it does not strip away punctuations. To make tokens more standardized, I created a tokenizer that uses regex. This blocker alone generates PC = 97%.

In [ ]:
```
name_tokens_block = bg.generate(
    bg.block(good_ds, function_=lambda r: ",".join(r.name_tokens)),
    bg.block(noble_ds, function_=lambda r: ",".join(r.name_tokens))
) #0.50,0.97
```

1. The second blocker is a joined blocker composed of publish year and author name.
    - The reason to use publish year instead of publish date as a whole is because lots of records in Goods are not standardized or missing. Most of its records atleast have a year of publish, so I choose to slice it out. It proves to be way more effective than using date as a whole. This block alone generates PC=80%.

- Using First author name generates PC=79%.
- The reason to join both blockers together instead of union is that it is nearly impossible for two different books to have the same author AND be published the same year: they must be the same book. The PC of the joined blocker is 64%.

In [ ]:
```
year_block = bg.generate(
    bg.block(good_ds, property_='publication_year'),
    bg.block(noble_ds, property_='publication_year')
) #0.23,0.80

author_block = bg.generate(
    bg.block(good_ds, property_='author', base_on=year_block),
    bg.block(noble_ds, property_='author', base_on=year_block)
) #0.69,0.79
```

1. The third blocker uses ISBN13 number. This is a very standardized attribute. Since the equality of ISBN between two books gurantees a matching, this is a perfect but strict blocker. PC=37%, with high Precision=97.

In [ ]:
```
ISBN_block = bg.generate(
    bg.block(good_ds, property_='ISBN13'),
    bg.block(noble_ds, property_='ISBN13')
) #0.12,0.55
```

Ultimately, 3 blockers are unioned together. This is because all of them have very narrow filter. Name tokens only allows those books have exactly the same tokens, creates lots of false negative; publish year is not that strict, but blocking with author name is very strict and combining them filters out a lot of books with no publish date records, also creates lots of false negatives; ISBN is a great blocker, but there are books with different ISBN actually matches, thus also creates lots of false negatives. However, their union greatly retrieves those false negaive cases since it is extremely rare for any pair to be the same book while having different author name, publish year, ISBN number and name tokens.

As a result, the union blocker generates RR=0.0003, PC=0.97 and precision=0.73

Task 1.3: Entity Linking

1. I used jaro similarity in the demo, which is not as good as later token cosine sim. option.

In [ ]:
```
def name_jaro_similarity(r1, r2):
    s1 = r1.name_string
    s2 = r2.name_string

    return rltk.jaro_winkler_similarity(s1, s2)
```

1. Name tokens cosine similarity score measures the cosine similarity of the tokens. This performs great for books with almost the same wordings. This is used as my main measure of similarity. However, this alone generates good F score while very low precision due to lots of false positives.

In [ ]:

```
def cos_similarity(vec1,vec2):
    intersection = set(vec1.keys()) & set(vec2.keys())
    numerator = sum([vec1[x] * vec2[x] for x in intersection])

    sum1 = sum([vec1[x] ** 2 for x in list(vec1.keys())])
    sum2 = sum([vec2[x] ** 2 for x in list(vec2.keys())])
    denominator = math.sqrt(sum1) * math.sqrt(sum2)

    if not denominator:
        return 0.0
    else:
        return float(numerator) / denominator

def name_tokens_cos_similarity(r1, r2):
    vec1 = Counter(r1.name_tokens)
    vec2 = Counter(r2.name_tokens)
    return cos_similarity(vec1,vec2)
```

1. Tried to lower the false negativity using the author names, but did not work out.

In [ ]:

```
def author_tokens_cos_similarity(r1, r2):
    vec1 = Counter(my_tokenizer(r1.author))
    vec2 = Counter(my_tokenizer(r2.author))
    return cos_similarity(vec1,vec2)
```

1. ISBN is always the best gurantee. This is used as the filter of my measure: if a pair passes ISBN equality check, they automatically passes with a confidence of 1.

In [ ]:

```
def ISBN_equality(r1, r2):
    if r1.ISBN13 == r2.ISBN13:
        return 1

    return 0
```

1. To address the high false positivity issue generated by #2, publish year equality is used. This is more like a filter at the very end, to throw away those books with high name similarities but are published in different years.
   - If the books have different publish years, they got a hard 0 for the score.
   - If the books have the same year of one of them do not have valid record, that means there is still chance of 50% that they are matches. Thus, a score of 0.5 is given.

In [ ]:

```
def publish_year_equality(r1, r2):
    y1 = r1.publication_year
    y2 = r2.publication_year
    if y1 == y2:
        return 0.5
    else:
        try:
            _ = int(y1)
            _ = int(y2)
            return 0
        except:
            return 0.5
```

Ultimately, I picked and chose the measure functions and set my parameters by testing out each combination. This following combination generates the best result:

- precison: 0.9180327868852459 recall: 0.835820895522388 f-measure: 0.875
- tp: 0.835821 [56]
- fp: 0.021739 [5]
- tn: 0.978261 [225]
- fn: 0.164179 [11]

In [ ]:
```
MY_TRESH = 0.83

def rule_based_method(r1, r2):
    if ISBN_equality(r1, r2)==1:
        return True,1
    score_1 = publish_year_equality(r1, r2)
    score_2 = name_tokens_cos_similarity(r1, r2)

    total = 0.2 * score_1 + 0.75 * score_2
    return total > MY_TRESH, total
```

Task 2: Knowledge Representation

Task 2.1: Model Construct

Name Space wise I included ISBNDB.com for the URIs of book subjects, schema.org and dbpedia.org for properties and classes, and myns for self created subproperties.

In [ ]:
```
FOAF = Namespace('http://xmlns.com/foaf/0.1/')
SCHEMA = Namespace('https://schema.org/')
ISBN = Namespace('https://isbndb.com/book/')
DBPD = Namespace('https://dbpedia.org/ontology/')
MYNS = Namespace('http://dsci558.org/myfakenamespace#')
```

To construct one subject, first I define its URI using isbndb.com:

In [ ]:
```
subject = URIRef(ISBN[row[4]])
```

If the record doesn's has one, then I create one using its id and its reference origin. The reason not to use only thier id is that there can be occasions when id131 from Goodreads and id131 from Nobles both exist, which will result in confusions.

In [ ]:
```
id = f"good_{row[0]}"
subject = MYNS[id]
```

Then, I use schema.org to set the class of subject to Book:

In [ ]:
```
my_kg.add((subject, RDF.type, SCHEMA.Book))
```

I also set its archived origin using schema:archivedAt

In [ ]:
```
my_kg.add((subject, SCHEMA.archivedAt, Literal("https://www.goodreads.com/")))
my_kg.add((subject, SCHEMA.archivedAt, Literal("https://www.barnesandnoble.com/")))
```

For all the other attributes, I find their corresponding property from my namespaces IF there exists a valid record. I also set the datatype for all the attributes as follows:

In [ ]:

```
if row[3].strip(): my_kg.add((subject, SCHEMA.isbn, Literal(row[3])))
if row[12].strip(): my_kg.add((subject, SCHEMA.publisher, Literal(row[12], datatype=S
if row[13].strip(): my_kg.add((subject, SCHEMA.datePublished, Literal(row[13], dataty
```

I also used blank nodes for defining nested elements like reviews and ratings.

In [ ]:

```
# Rating
        my_kg.add((BNode("rating_"+id), RDF.type, SCHEMA.AggregateRating))
        my_kg.add((BNode("rating_"+id), SCHEMA.ratingValue, Literal(row[9], datatype=
        my_kg.add((BNode("rating_"+id), SCHEMA.ratingCount, Literal(row[10], datatype
        my_kg.add((subject, SCHEMA.aggregateRating, BNode("rating_"+id)))
```

I also created a few self-defined properties for Nobles and Barnes' records, like salesRank and paperbackPrice. I defined its type as property, its subclassOf, range and domain.

In [ ]:

```
my_kg.add((MYNS["salesRank"], RDF.type, RDF.Property))
my_kg.add((MYNS["salesRank"], RDFS.subPropertyOf, DBPD.rank))
my_kg.add((MYNS["salesRank"], RDFS.domain, SCHEMA.Book))
my_kg.add((MYNS["salesRank"], RDFS.range, XSD.integer))
```

The reason why I seperated the matching pairs is because in most cases matching subjects still differs a lot regarding their name, page count, author name and ratings. My design thus persists their original data while matching them using the property RDFS:seeAlso.

In [ ]:

```
my_kg.add((r1_subject, RDFS.seeAlso, r2_subject))
my_kg.add((r2_subject, RDFS.seeAlso, r1_subject))
```

Unfortunately, I found it impossible for the RDG Grapher to graph the whole KG, not even 20 of the entries. As a result, I sampled about 10 of them and try to include all the situations I designed.