Web Scraping and Social Media Scraping Project
Scraping recent Computer Science papers

Adele Ossareh, 436818
Mohammad Saeed Pourjafar, 436817

1 Description

The webpage, arxiv has always been a valuable source for the state-of-the-art papers in different categories of science from well-known researchers and journal publishers. It is essentially an open-access repository of electronic preprints owned by Cornell University and it's good to know that it was first launched on August 14, 1991 (29 years ago).

It consists of scientific papers in the fields of mathematics, physics, astronomy, electrical engineering, computer science, quantitative biology, statistics, mathematical finance and economics, which can be accessed online. As of April 2021, the submission rate is about 16000 articles per month.

Our project is centered on scraping the information for the most recent papers in the field of computer science from arxiv homepage as a starting point for all of our scrapers, namely Beautiful Soup (BS), Scrapy and Selenium. The information we aimed to scrape are the title of the paper, category of the paper, authors, the date that the paper has been submitted on the webpage and finally the link for the PDF format of the paper.

2 Scraper mechanics

Each scraper has a unique way of gathering the required information through its own functionality. Below you can find their specific approaches for retrieving the data:

Beautiful Soup: We begin by the starting webpage http://www.arxiv.org for the URL variable, then by the help of request package, we open the URL and from there by instantiating a beautiful soup library, we read and parse the

HTML of the webpage which we opened earlier.Next by calling the `find_all` method we are able to achieve the anchor tags which their id contains the phrase cs (abbreviated of Computer Science), for this part we used regular expressions to extract the desired ids. At this moment we stored the tags but for extracting the hypertext reference from these tags (href) we need to call `findall` method (from regex) to do it for us and then by appending the 'https://arxiv.org' to the beginning of the extracted links, we now have a links for all computer science categories which we stored and printed out for further use.

Next we are gathering the links for each category based on what we did in the previous step. The process is somehow the same but this time the tags are defined as the anchors which their associated title contains the term "Abstract". We also defined the limit value of 100 pages base on flattening the list of lists (from previous part) which simply means instead of having a nested list for different categories, in order to impose a constraint (100 pages) we need to transform the nested list into a one single list so then we can tell the program to only scrape the 100 elements (pages) off of it. If the limit is set to false, another path will be taken which has no limit for scraping papers.

Finally an empty pandas' dataframe is instantiated to store the information on each paper. By using `bs.find` method in a loop for each paper, we are able to find and store the required data such as title, category, author and date of submission in a dictionary and then append it to our data frame for the output.

Scrapy: The scrapy part works in three separated spiders. In the first spider (arxiv1.py) we create an empty field of links so we can store the links for different categories of CS in it. Then we defined the name of our spider which is "topics" and provide the starting webpage as before. Then we define a parsing function which gets response as an argument. Then we define the xpath in which the id of the anchor tag contains the term "cs." To match the desired xpath, then we instantiated a link class which we created earlier and finally by adding the proper URL to the beginning of the scraped links, we have our first output for categories in computer science. In this first part, since the number of categories in computer science are exactly 40 which is less than 100, therefore we applied the limit on the next spider.

In the next spider (arxiv2.py) we are defining the limit and reading the previous csv file for cs topics, we parse every link for papers (from anchor tags in href) by their xpath in which the title contains the term "Abstract". We also defined a counter for limiting the number of scraped pages. If the limit is set to True (default) we tell the spider to close by using `CloseSpider()` exception and just like before we append the proper link to the beginning of the links but this time, the links are for papers (not topics). If the limit is set to False, the spiders scrape all the links of papers for all topics in computer science.

Finally in the last spider (arxiv3.py) we first create 5 fields then we open the csv

files from the previous step to read the link for each paper. We then defined 5 different xpaths for our 5 fields in our parse function and grabbed their text off of them. There are also some criteria for some xpath (date) to make sure that we end up in the legitimate output (simply to adjust xpath in case the xpath doesn't yield the expected output). Aside from criteria which defined in the "if" statement, we also tried to handle errors with `try/except` chuck of code for PDF since xpath for PDF in some of the papers slightly differs from the rest. In addition we tried to make sure that our xpath gets the CS tags if the paper is interdisciplinary. In the end we finally yield the output for our papers.

Selenium: Since this time we have the advantage of interaction on our side, the approach and therefore the mechanism is different from those two. So it means instead of first scraping categories of CS and then scraping the papers in those categories, we can directly access the papers in all categories via advanced search button from the webpage.

After loading required libraries and locating the proper webdriver (geckodriver since we both used Firefox), we defined the maximum wait for the driver and let the driver get the webpage url ("https://arxiv.org").Since in the performance we want to measure running time for our scrapers, in Selenium instead of explicitly tell the program to wait for some fixed amount of time, we used `presence_of_element_located()` method from `expected_conditions` which has been imported from `selenium.webdriver.support`. After the "Advanced Search" button becomes visible based on its dedicated xpath, we click on it. From there we click on the checkbox for "Computer Science", leaving the Submission date (most recent) as it is since we are only interested in the recent papers in CS. Then we find the xpath for "Search" button and after clicking on it, the results of latest CS papers will appear which in each page we have the results for 50 papers.

Now it's time to initialize the empty lists for our fields in each paper. We defined the scrape function which we need to pass the driver as an argument, and we wait for the title of the last paper to be visible (by the proper xpath). Then we scrape the title by the help of regex and for each paper we append it to the `Title_list`. For the "Authors" and "Category" instead of using regex we find the element by class name which is `authors` and `tags.is-inline-block` respectively. We repeat the process for date and PDF as well.

Since the result per pages is set to 50 by default, therefore instead of programmatically defining the limit by counter and length of our scraped links, we can tell the program to click the Next button and scrape the info until it reaches 100 results (pages) by passing our function to a loop and repeat it 2 times. Finally we created a pandas series and dataframe and pass our lists of information to it as the columns.

3 Description of the output

From the point of information which we scraped, all the output are labeled as papers in the field of Computer Science and they are all the recent papers in this field which means although there are different subcategories in CS (exactly 40), but our aim is to scrape the recent papers in CS no matter in which specific category it belongs to and giving back the scraped information as the output.

From the point of output structure, each of the participants used different ways to sort the information (list, dictionary, pandas series and dataframe) however for the sake of consistency we both agreed that no matter how we stored the data, the final format of the output should be a single csv file (papers.csv) with five columns consisting of the title of the paper, category of paper, authors, dateof submission and the link to the PDF format of the paper.

4 Performance comparison

The performance of the scrapers has been measured by two factors: time and memory. In this regard we computed the elapsed running time for selenium and beautiful soup scrapers to complete the 100 pages scraping (base level) by `timeit()` function. For the scrapy the elapsed time is already printed in the console output. Two things to notice here. First since scraping the web pages also relies on the internet connection speed, we would like to report the results from each participant, since each of us use a different network technology and therefore different connection speed. Another thing to notice is the starting point and the end point of our `timeit()` function. In this regard we start timing the program for scraping the five columns from papers. This way we can tell how many seconds it takes for each scraper to gather the last and target information as it's also the case for scrapy to measure the time for scraping the third part and by this unified starting point we can have a better measure for elapsed time.

For memory usage we used `psutil` library to measure the total memory usage in MB of the scraper while it's running. Please note that for selenium it could be biased since we also need to start a browser and this memory usage won't be captured from the codes, but it's definitely adding up to the total memory usage.

The time and memory efficiency is summarized in the table below. All times are reported in seconds and rounded to two decimal points.

Participant/Memory	Beautiful Soup	Scrapy (arxiv3.py)	Selenium
Adele (4G)	253.32	19.53	25.17
Saeed (ADSL2+ 4Mbps)	437.71	23.84	27.63
Memory (in MB)	75.84	47.98	55.22

Table 1- Comparison of scraper performance

Based on the results from the table above, and since our primary goal was to scrape the info (rather than automating and interaction), clearly the winner is scrapy. Please note that the actual memory usage for Selenium would be higher than the one that reported here since this number didn't account for the browser memory usage (Firefox).

5 Data analysis and further use

Since all our scraped information is non-numerical data (except for date of submission which can be treated as timestamp) for the proof of data analysis for gathered information, we decided to classify the papers based on their categories. For this regard we used a NLP (Natural Language Processing) approach to make predictions on classifying categories for each title. We scraped and used a dataset which contains information on 3000 papers. We used `gensim` and `nltk` libraries to create and vectorize titles and finally build the model with a ridge classifier from `scikit-learn`. Although the accuracy we obtained is low, (probably another idea would be to implement RNN with LSTM) it proves the usability of the scraped data.
For this PDF to be self-contained, we add the codes in the appendix and also the notebook is available in the `Data_analysis_NLP` folder in the repository which also contains the PDF and HTML formats of the `arxiv_cs_papers_classification` file.

6 Participation

In the table below we provided information for participation among the group members. Please note that although each part was meant to be specific for each member and that member did the most of that part, the overlapping and collaboration was inevitable which means we both did our specific tasks and also we are aware of our teammate's task so we helped each other along the way.

Participant	Beautiful Soup	Scrapy	Selenium	NLP	Report
Adele	Most parts	Revision	Xpaths for landing on target	Building model	Scraper mechanics/Output
Saeed	Revision	Most parts	Xpaths to get info from target	EDA/vectorization	Scraper mechanics/Performance

Table 2- Participation

Appendix: arxiv_cs_papers_classification

This is a simple model for classifying the paper category in CS. The data has been gathered from the scraped information on 3000 recent computer science papers from http://arxiv.org

In [1]:

```
import pandas as pd
import sklearn
import numpy as np
import nltk
import re
import matplotlib.pyplot as plt
import seaborn as sns
# nltk.download('stopwords') # Ucomment if it hasn't been downloaded yet
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn import linear_model
import warnings
from sklearn.feature_selection import chi2
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import StratifiedKFold
import gensim
from gensim.models import Word2Vec
```

We want to build the model based on NLP for title and classify it according to it's label (Category). As you can see in the table below, we have shown a random selection of 10 papers in which each title is associated with its defined category on arxiv. The left nuumbers are the indices for papers that goes all the way from 0 to 2999 for a total of 3000 papers.

In [24]:

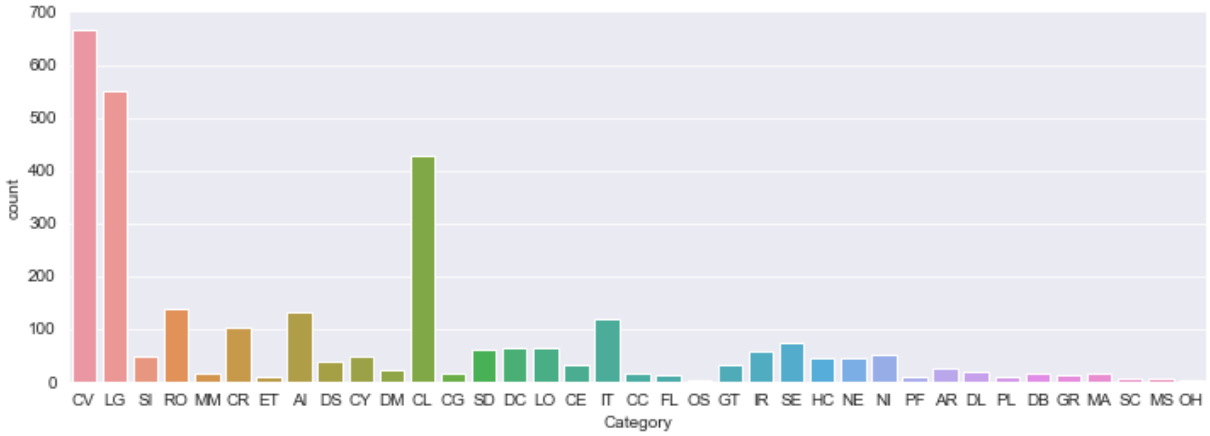
```
# Reading the data from 3000 scraped papers
df = pd.read_csv('papers.csv')
df.shape
pd.set_option('max_colwidth', 900)
df.sample(10).iloc[:,0:2]
```

Out[24]:

	Title	Category
1991	A Two-stage Deep Network for High Dynamic Range Image Reconstruction	CV
217	Continuous Decoding of Daily-Life Hand Movements from Forearm Muscle Activity for Enhanced Myoelectric Control of Hand Prostheses	RO
231	Algorithmic Factors Influencing Bias in Machine Learning	LG
2249	Consistent Accelerated Inference via Confident Adaptive Transformers	CL
958	Attention on Global-Local Embedding Spaces in Recommender Systems	IR
2442	3-Coloring on Regular, Planar, and Ordered Hamiltonian Graphs	CC
2543	Data Augmentation for Voice-Assistant NLU using BERT-based Interchangeable Rephrase	CL
2707	Faithful and Plausible Explanations of Medical Code Predictions	LG
2815	Image Super-Resolution via Iterative Refinement	CV
693	Ideology in Open Source Development	SE

In [18]:

```
plt.figure(figsize=(12,4))
sns.set_style("darkgrid")
sns.countplot(x="Category",data=df);
```



Looks like CV (Computer Vision), LG (machine LearninG) and CL (Computation and Language) were the most frequent categories in recent computer science papers

In [8]:
```
print("Number of unique categories in dataframe:",len(df.Category.unique()))
all_cat = ['AI','CL','CC','CE','CG','GT','CV','CY','CR','DS','DB','DL','DM','DC',
           'ET','FL','GL','GR','AR','HC','IR','IT','LO','LG','MS','MA','MM','NI',
           'NE','NA','OS','OH','PF','PL','RO','SI','SE','SD','SC','SY']
for i in all_cat:
    if i not in df.Category.unique():
        print(i)
titles = df['Title']
labels = df['Category']
```

```
Number of unique categories in dataframe: 37
GL
NA
SY
```

looks like General Literature (GL), Numerical Analysis (NA) and Systems and Control (SY) hasn't been in recent papers

In [9]:
```
# Preprocessing
processed_titles_wordlist = []
processed_titles = []
stops = set(stopwords.words('english'))
for i in range( 0, titles.size):
    words = titles[i].lower().split()
    words = [w.lower() for w in words if not w in stops]
    processed_titles_wordlist.append(words)
    processed_titles.append(" ".join(words))
print(processed_titles[0:5])
print(processed_titles_wordlist[0:5])
```

```
['exemplar-based 3d portrait stylization', 'large-scale study unsupervised spatiotem
poral representation learning', 'learned spatial representations few-shot talking-he
ad synthesis', 'discover unknown biased attribute image classifier', 'mongenet: effi
cient sampler geometric deep learning']
[['exemplar-based', '3d', 'portrait', 'stylization'], ['large-scale', 'study', 'unsu
pervised', 'spatiotemporal', 'representation', 'learning'], ['learned', 'spatial',
'representations', 'few-shot', 'talking-head', 'synthesis'], ['discover', 'unknown',
'biased', 'attribute', 'image', 'classifier'], ['mongenet:', 'efficient', 'sampler',
'geometric', 'deep', 'learning']]
```

Vectorization and building the model

Now that we cleaned our data and removed the stopwords and gathered a processed title and wordlist, it's time to vectorize title by TFIDF (Term Frequency–Inverse Document Frequency)

In [12]:

```
# Vectorizing by TFIDF and building Document Term Matrix (DTM)
vect = TfidfVectorizer()
dtm = vect.fit_transform(processed_titles).toarray()
chisqModel = SelectKBest(chi2,k=5655)
chisqDtm = chisqModel.fit_transform(dtm,labels)

def makeFeatureVec(words, model, num_features):
    feature_vec = np.zeros((num_features,),dtype="float32")
    nwords = 0
    index2word_set = set(model.wv.index2word)
    for word in words:
        if word in index2word_set:
            nwords += 1
            feature_vec = np.add(feature_vec,model.wv[word])

    feature_vec = np.divide(feature_vec,nwords)

    return feature_vec

def getAvgFeatureVecs(title, model, num_features):
    counter = 0
    titleFeatureVecs = np.zeros((len(title), num_features),dtype="float32")
    for t in title:
        titleFeatureVecs[counter] = makeFeatureVec(t, model,num_features)
        counter += 1
    return titleFeatureVecs

word2vec_model = Word2Vec(processed_titles_wordlist, workers=1,
            size=5655, min_count = 1,
            window = 8, sample = 1e-5)
word2vec_model.init_sims(replace=True)
wordVecs = getAvgFeatureVecs(processed_titles_wordlist, word2vec_model, 5655)
combinedFeatures = np.hstack([chisqDtm,wordVecs])
```

Finally we feed the data which has been split into train and test set to our model and evaluate our model accuracy.

In [13]:

```
# Building the model
skf = StratifiedKFold(n_splits=10)
skf.get_n_splits(combinedFeatures, labels)
warnings.filterwarnings("ignore", category=UserWarning)
for train_index, test_index in skf.split(combinedFeatures,labels):
    X_train, X_test = combinedFeatures[train_index], combinedFeatures[test_index]
    y_train, y_test = labels[train_index], labels[test_index]
    model = linear_model.RidgeClassifier().fit(X_train, y_train)
    y_pred = model.predict(X_test)
print("Model accuracy:",round(sklearn.metrics.accuracy_score(y_test, y_pred),2))
```

```
Model accuracy: 0.57
```