



Università degli Studi di Roma La Sapienza
Faculty of Ingegneria dell'Informazione,
Informatica e Statistica

Master of Science in Engineering
in Computer Science

Master's Degree

Financial Services Heuristic Retrieval for Operations and Payments Settlement Directorate of Banca d'Italia

Candidate:
Mauro Papa
ID Number 1530104

Supervisor:
Prof. Ioannis Chatzigiannakis
Co-Supervisor:
Prof. Aris Anagnostopoulos

Academic Year 2019-2020

*The views expressed in this thesis are those of the author
and do not involve the responsibility of Banca d'Italia.*

*I am the vine, you are the branches. Whoever remains in me and I in him
will bear much fruit, because without me you can do nothing.*
John 15:5

CONTENTS

1	INTRODUCTION	3
1.1	Business Case	3
1.2	Project overview	4
1.3	Outline	5
2	DATASET BUILDING AND PRE-PROCESSING	7
2.1	Dataset Building	7
2.1.1	Dataset Identification	7
2.1.2	Data collecting	9
2.2	Dataset pre-processing	18
2.2.1	Pre-processing steps	18
2.2.2	Markdown Syntax Clean-up	19
2.2.3	Part-of-Speech tagging	22
3	WORD EMBEDDING MODEL TRAINING	27
3.1	Introduction to Word Embedding	27
3.2	Training phase	27
3.3	Test set definition and Evaluation technique	31
3.4	Accuracy measurement	33
4	NAMED ENTITY DISAMBIGUATION	37
4.1	Introduction to Named Entity Disambiguation	37
4.2	Choice of the Knowledge Base	37
4.3	Core idea behind NED algorithm for word embeddings	40
4.4	Description of NED algorithm for word embeddings	43
4.4.1	First Batch of HTTP requests	44
4.4.2	Second Batch of HTTP requests	51
4.4.3	Third Batch of HTTP requests	55
4.4.4	Fourth Batch of HTTP requests	58
4.4.5	Selection of the best combination of entities	61
4.4.6	Handling of unresolved and discarded terms	74
4.5	NED results evaluation	74
4.5.1	Accuracy of NED against the manually evaluated test set.	75
4.5.2	Evaluation of the word embedding models removing unmatched terms	77
5	WEB APPLICATION FOR MODEL QUERYING	81
5.1	Overview	81
5.2	Frameworks choice	81
5.2.1	JavaScript front-end framework	81
5.2.2	CSS front-end framework	82
5.2.3	Python back-end framework	84
5.3	Web Application Description	84
5.3.1	Web Application Overview	84

5.3.2	Search-bar	87
5.3.3	Container with details of target word matched entities	94
5.3.4	Container of matched entities details	97
5.3.5	Container of unresolved terms and container of discarded terms	100
5.3.6	D3 Force Graph	103
6	USE-CASE: FINANCIAL DATA PLATFORMS	111
6.1	Searching: bloomberg_terminal	112
6.2	Searching: eikon	113
6.3	Final considerations	114
7	CONCLUSIONS AND FUTURE WORKS	117
	BIBLIOGRAPHY	119

1 | INTRODUCTION

1.1 BUSINESS CASE

The Financial Data Providers industry sells market data and related services to financial institutions, traders and investors. Leading industry vendors aggregate data and contents from stock market feeds, brokers and dealer desks as well as regulatory repositories, to distribute financial news and business information to the investor community. They play a key role in the financial professional's workflow and the demand for such services is constantly growing.

In addition to the big number of products and services available, the Financial & Market Data offering is extremely complex and it is characterized by: low competition, strong expansion of demand and constantly increasing prices [28]. In terms of variety, products range from simple services to complex data feeds and sophisticated data processing platforms, serving multiple business areas of a financial institution (front-middle-back office).

Table 1: Financial information platforms market shares [26]

Vendor	Platform	Number of users	Share
Bloomberg	Bloomberg Terminal	325,000	33.40%
Refinitiv	Eikon	190,000	23.10%
S&P	Capital IQ	Undisclosed	5.60%
FactSet	FactSet	89,000	4.20%
Others	-	-	33.70%

In this scenario, it is extremely difficult to identify small financial services providers, since they only own very little market shares and are therefore often excluded from the market data procurement process. Indeed, as stated in the Gartner Glossary¹: *Procurement is the corporate function that has governance over purchasing decisions for a company. Activities of the procurement function include strategic vendor evaluation and selection, competitive bidding, contract negotiation and purchasing. Effective procurement practices enable organizations to reduce costs and maximize value.*

¹ <https://www.gartner.com/en/sales/glossary/procurement>

In the past, Procurement Automation (aka eProcurement) was mainly focused on the use of ERP management tools to record and examine previous buying decisions and expenditure data [13]. In recent years, machine learning and artificial intelligence were applied to procurement workflows, introducing computation of external or third-party unstructured data, to achieve a higher level of market knowledge and decision automation. This new kind of procurement is often referred to as AI Procurement or Digital Procurement[1].

Although the procurement process of a public institution, like Banca d'Italia, must go through several legal, commercial and financial validation steps, the use of Machine Learning and Natural Language Processing perfectly suits the need of the financial services sourcing task, particularly for below-threshold contracts².

The main requirement of this experimentation is the development of a search engine that allows to search a financial product and outputs a list of potential competitors as comprehensive as possible. A platform of this kind, can potentially speedup and enhance market researches that are generally executed manually or by purchasing expensive third party reports. In detail, the Operations and Payments Settlement Directorate aims to:

1. Identify new suppliers and markets
2. Make more accurate decisions to carry out well-timed analysis and insights based on Big Data
3. Identify new opportunities for spending optimization
4. Automate manual tasks by freeing Full Time Equivalent to involve in other processes
5. Optimize demand management
6. Improve suppliers management (vendor analysis and rating)
7. Have greater transparency about the selection and acquisition of services on the market
8. Discover innovative products never used before in business operations

1.2 PROJECT OVERVIEW

To achieve the aforementioned goals, this project was built as a pipeline of three state-of-the-art Natural Language Processing mod-

² Below-threshold contracts definition:

https://www.codiceappalti.it/Italian_Procurement_Code/Art__36__Below-threshold_contracts/9622

els, to end up with an *hybrid system* in which the output of a model is used as input for the next one. This project can indeed be split into four consequent steps that I'm going to further explain in the next chapters:

1. Dataset building: Collecting a huge corpus of financial textual data from *Reddit*³.
2. Dataset pre-processing: Removing all the markdown syntax and computing a **Part-of-Speech tagging model** over the dataset for better information extraction.
3. Word embedding model training: Train a **Word Embedding model** over preprocessed dataset to exploit search by similarity feature.
4. Named-entity disambiguation and linking: Similarity results disambiguation and matching against a knowledge base (*Wiki-data*) to provide structured information. This custom algorithm uses combinations of **Document Embeddings** generated over Wikipedia pages and **edit distances** between Knowledge Base entity aliases.

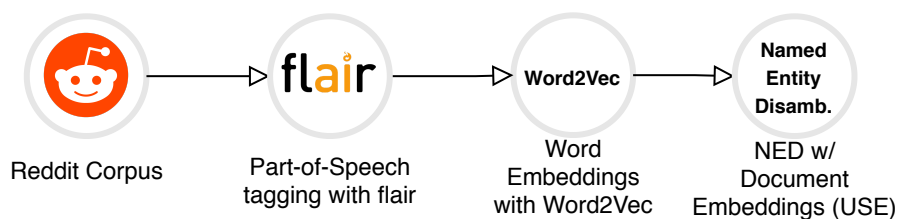


Figure 1: Hybrid System pipeline visualization

1.3 OUTLINE

This thesis is composed of seven chapters, each of them covers one or two of the steps just described. Below you can find a short outline of what each chapter was intended for:

1. **Chapter 1:** The business case and the requirements for this project are presented
2. **Chapter 2:** The methodology for dataset building and pre-processing is shown

³ <https://www.reddit.com/>

3. **Chapter 3:** An in-depth analysis of Word Embedding models for similarity computation among words in the dataset is provided
4. **Chapter 4:** A Named Entity Disambiguation custom algorithm description for additional filtering and matching of Word Embedding results is given
5. **Chapter 5:** Web application overview for user model querying
6. **Chapter 6:** A real use-case is addressed
7. **Chapter 7:** Conclusions and future works

2

DATASET BUILDING AND PRE-PROCESSING

2.1 DATASET BUILDING

2.1.1 Dataset Identification

Data is the most important factor for a project of this kind. The *Dataset of Economic and Financial News by Refinitiv* is a financial text corpus that has already been used in many sentiment-analysis based projects [3][29][31]. This dataset is available in different forms:

1. Historical Data from 1997 to 1998 and from 2008 to 2009 can be downloaded after application to NIST¹.
2. Recent news can be collected using Machine Readable News (MRN) with Elektron Message API (EMA) by Refinitiv².

Refinitiv itself provides straight-forward sentiment analysis tutorials for developers, using financial news from its Eikon trading platform API ³.

Nevertheless, preliminary word embedding tests carried out over this dataset, in partnership with the Italian National Institute of Statistics (ISTAT), showed **discouraging** results. In particular, such experiments highlighted the need for a dataset with the following requirements:

1. Containing (also) information like: **opinions, descriptions and comparisons** of financial products
2. Data size big enough to train a machine learning model, in the order of Gigabytes
3. Freely available (non-mandatory)

The quest for a suitable corpus ended up in *Reddit*. It is an American social news aggregation, web content rating, and discussion website⁴, that covers a very wide range of topics in multiple languages. The key feature of Reddit is that discussions are organized into user-created areas of interest called *subreddits*. This allows for a much

¹ <https://trec.nist.gov/data/reuters/reuters.html>

² <https://www.refinitiv.com/en/products/world-news-data>

³ <https://developers.refinitiv.com/article/introduction-news-sentiment-analysis-eikon-data-apis-python-example>

⁴ <https://en.wikipedia.org/wiki/Reddit>

easier data topic filtering if compared to a social media corpus like Twitter or others.

The selection of the financial subreddits of interest was carried out manually, searching directly into Reddit or exploiting existing non-complete lists available online. In particular, 66 subreddits were selected, each of them having a relevant number of members (at least few hundreds). It is important to specify that only english subreddits were taken into account. The chosen subreddits are listed below:

- | | |
|-----------------------|---------------------------|
| 1. Investing | – r/PersonalFinanceCanada |
| – r/investing | – r/FinancialPlanning |
| – r/RobinHood | – r/CRedit |
| – r/wallstreetbets | – r/finance |
| – r/SecurityAnalysis | – r/FinancialCareers |
| – r/InvestmentClub | – r/CFA |
| – r/StockMarket | – r/portfolios |
| – r/Stock_Picks | – r/Economics |
| – r/Forex | – r/Accounting |
| – r/options | – r/Bogleheads |
| – r/cryptocurrencies | – r/economy |
| – r/CanadianInvestor | – r/AskEconomics |
| – r/stocks | – r/tax |
| – r/ETFs | – r/actuary |
| – r/ausstocks | – r/sales |
| – r/UKInvesting | – r/fican |
| – r/FuturesTrading | – r/fiaustralia |
| – r/TradeVol | – r/FIREUK |
| – r/Commodities | – r/ukpersonalfinance |
| – r/Daytrading | – r/eupersonalfinance |
| – r/Trading | – r/EuropeFIRE |
| – r/phinvest | – r/financialindependence |
| – r/FixedIncome | – r/leanfire |
| – r/forex_trades | – r/fatFIRE |
| – r/pennystocks | – r/Fire |
| – r/IndiaInvestments | – r/UKPersonalFinance |
| 2. Finance in general | – r/PersonalFinanceNZ |
| – r/personalfinance | – r/AusFinance |

- | | |
|---|--|
| <ul style="list-style-type: none"> 3. Economic Data <ul style="list-style-type: none"> – r/econmonitor – r/econometrics – r/academiceconomics – r/datasets 4. Green <ul style="list-style-type: none"> – r/greeninvestor | <ul style="list-style-type: none"> – r/sustainableFinance 5. Algorithms <ul style="list-style-type: none"> – r/algotrading – r/BusinessIntelligence – r/quant – r/algorithmictrading – r/fintech |
|---|--|

2.1.2 Data collecting

Reddit provides official APIs to download and collect data. Many wrapper libraries are also available, the most liked one on Github is PRAW⁵ (Python Reddit API Wrapper). However, we are **not** going to use any library that works with the official API, because a recent update disabled the functionality of filtering posts based on time ranges⁶. This makes the retrieval of threads from the past extremely hard, allowing to get only posts from *last day*, *last month*, etc.

Luckily, a project called *Pushshift*⁷, that is not supported by Reddit, provides an API that serves a copy of all Reddit comments and submissions since 2015. More in general, Pushshift is a big-data storage and analytics project started and maintained by Jason Baumgartner. As of today, Reddit data is copied into Pushshift at the time it is posted to Reddit. As stated in [6], Pushift is composed of the following subsystems:

1. An ingest engine to collect and store raw data.
2. A PostgreSQL database for data advanced querying and meta-data storing.
3. An Elastic Search document store cluster, for indexing and data aggregation.
4. An API to access collected data.

Before technically describing how posts were collected, it is important to note that data in Reddit is split into *submissions* and *comments*:

1. **submissions**: are the posts of a subreddits, containing either a title and a description. Many times they refer to questions asked by users to the community of the subreddit.

⁵ <https://github.com/praw-dev/praw>

⁶ https://www.reddit.com/r/changelog/comments/7tus5f/update_to_search_api/

⁷ <https://pushshift.io/api-parameters/>

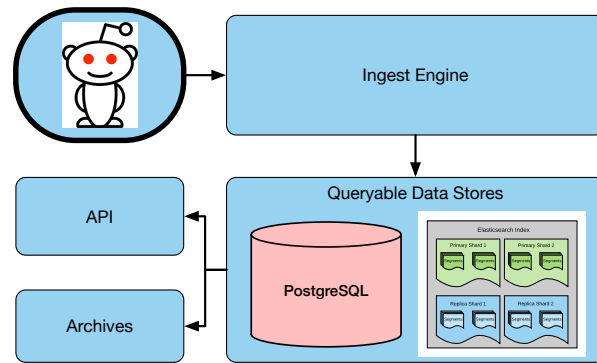


Figure 2: Pushshift's Reddit data collection platform. Source: [6]

2. **comments:** are the replies made by the users of a subreddit to a submission. **All submissions are archived after six months, denying the possibility to add new comments**

Due to this categorization, data provided by Reddit always contains the identification code of a parent element. Hierarchical identification code prefixes are used to better highlight the interconnections between data objects:

- **t1_** = Comment
- **t2_** = Account
- **t3_** = Link
- **t4_** = Message
- **t5_** = Subreddit
- **t6_** = Award

Accounts, links, messages and awards are not useful for this thesis. If you would like to further investigate them please refer to <https://www.reddit.com/dev>

Pushshift also uses the same kind of classification. In particular, data can be downloaded in two ways:

1. Via monthly dumps, with separate files for comments and submissions: <https://files.pushshift.io/reddit/>
2. Via API, with separate endpoints for comments and submissions:
 - a) <https://api.pushshift.io/reddit/submission/search/>
 - b) <https://api.pushshift.io/reddit/comment/search/>

Most of the data for this project was collected filtering subreddits of interest from monthly dumps. These files are very big (up to 16Gb for single dump) and are **jsonlines (aka ndjson) serialized**, i.e. json-like encoding with a separate json object per line. These files can be *lazy-loaded* into computer RAM reading one line at a time, removing any potential risk of memory saturation.

Moreover, they are compressed using one of the following algorithms: bzip2 (.bz2), LZMA2 (.xz) or zstd (.zst). A python class called `GrepDumps` has been implemented to read them, line by line, while still being compressed. The code extracts only json objects that refer to the subreddits taken into account. It can be used for both submissions dumps and comments dumps. All json objects that refer to the same subreddit are merged in a single file. For example, the files for subreddit **r/investing** will be:

- | | |
|------------------------------|------------------------------|
| 1. Submissions files: | 2. Comments files: |
| – RS_2016-01_investing.jsonl | – RC_2016-01_investing.jsonl |
| – RS_2016-02_investing.jsonl | – RC_2016-02_investing.jsonl |
| – ... | – ... |

Listing 1: Methods `grep_xz` and `write_line` of `GrepDumps` class

```
def grep_xz(self, root, pname, ext):
    with lzma.open(os.path.join(root, pname), mode='rt')
        as file:
        for line in file:
            try:
                obj = json.loads(line)
                self.write_line(obj, pname)
            except Exception as e:
                logger.error("Skipped line while working on %s/%s%s"
                    " % (root, pname, ext))
                logger.error(e)
    return

def grep_bzip(self, root, pname, ext):
    [...]

def grep_zst(self, root, pname, ext):
    [...]

def write_line(self, obj, outname):
    if 'subreddit' in obj and obj['subreddit'] in self.
        subreddit_list:
    try:
        self.file_writers[obj['subreddit']].write(obj)
        logger.info('Writing line in %s_%s.jsonl' % (outname
            , obj['subreddit']))
```

```

except Exception as e:
    Path(os.path.join(self.output_directory, obj['
        subreddit'])).mkdir(parents=True, exist_ok=True)
    self.file_writers[obj['subreddit']] = jsonlines.open
        (os.path.join(self.output_directory, obj['
            subreddit'], '%s_%s.jsonl' % (outname, obj['
                subreddit'])), mode='w')
    logger.info('Opened file %s.jsonl for writing' %
        outname)
    self.file_writers[obj['subreddit']].write(obj)
    logger.info('Writing line in %s_%s.jsonl' % (outname
        , obj['subreddit']))

[...]

```

However, these dump files are published with a certain delay in the repository. Hence, it was needed to implement a small `Crawler` class to download and collect data of remaining months through the API. This class is based on a wrapper library called PSAW⁸ (Python Pushshift.io API Wrapper).

Listing 2: Method `extract_reddit_data` of `Crawler` class

```

@staticmethod
def extract_reddit_data(type, month, output_directory,
    subreddit_list):
    api = PushshiftAPI()

    assert type == 'submission' or type == 'comment', "
        Type not allowed"

    start_epoch = int(month.timestamp())
    end_epoch = int((month + relativedelta(months=1)).
        timestamp())

    logger.info('Crawling %ss for %s ( from %s to %s )' %
        (type, month.strftime('%Y-%m'), str(start_epoch),
            str(end_epoch)))

    if type == 'submission':
        file_name = "RS_" + month.strftime('%Y-%m')
    else:
        file_name = "RC_" + month.strftime('%Y-%m')

    for subreddit in subreddit_list:
        logger.info('Crawling %ss for subreddit: %s' % (type,
            subreddit))
        Path(os.path.join(output_directory, subreddit)).mkdir
            (parents=True, exist_ok=True)

```

⁸ <https://github.com/dmarx/psaw>


```

output_file = os.path.join(output_directory,
                             subreddit, file_name + '_' + subreddit + '.jsonl')
with jsonlines.open(output_file, mode='w') as writer:
    if type == 'submission':
        gen = api.search_submissions(after=start_epoch,
                                      before=end_epoch,
                                      subreddit=subreddit,
                                      sort='asc')
    else:
        gen = api.search_comments(after=start_epoch,
                                  before=end_epoch,
                                  subreddit=subreddit,
                                  sort='asc')

    for item in gen:
        item[-1].pop("created")
        writer.write(item[-1])
    logger.info('%ss saved to: %s' % (type, str(
        output_file)))

```

By means of these two classes, a dataset starting from January 2016 to February 2020 was collected. The total size of the data, including json encoding syntax, is 36Gb. Indeed, every post in Reddit comes with a bunch of extra information, but only the following json tags are important for us:

1. Submissions files:
 - **id**: submission identifier without prefix
 - **subreddit**: subreddit name
 - **selftext**: markdown encoded text of the submission
 - **created utc**: UNIX timestamp referring to the time of the submission creation
 - **num comments**: number of comments of the submission
2. Comments files:
 - **id**: comment identifier without prefix
 - **link id**: submission identifier with prefix
 - **parent id**: identifier of the parent of this comment with prefix, i.e. a submission id or another comment id
 - **subreddit**: subreddit name
 - **body**: markdown encoded text of the comment
 - **score**: number of upvotes of the comment minus the number of downvotes
 - **created utc**: UNIX timestamp referring to the time of the comment creation

- **retrieved_utc**: UNIX timestamp referring to the time the comment was crawled by Pushshift
- **num comments**: number of comments of the submission

After downloading both comments and submission files, they were merged together into a single file per month per subreddit:

- RSRC_2016-01_investing.jsonl
- RSRC_2016-02_investing.jsonl
- ...

This allows a much easier data feed in the training phase of a machine learning model. However, there are several issues to face before getting those files merged. In particular, since a submission is archived in six months, if it is created in month n , its comments will be split in files from month n to month $n + 7$ (considering the delay between **created_utc** and **retrieved_utc**). To solve this problem, let's assume to have a directory with all submissions and comments files of a subreddit. Then let's create an ordered list with all the names of comments files that can be read using a window of size 7, which will be shifted to the right every time the algorithm starts reading the submission file of the next month.

Moreover, the comments of a submission should be merged in the same order they are displayed in Reddit, since they are feeded as a single sentence to the word embedding model during the training phase. This is important to achieve maximum information extraction. Indeed, one of the possible algorithms used for training (CBOW) is based on a fixed-size sliding window mechanism for words in a phrase. More about this is presented in 3.2.

There are two major issues to recreate such an order:

1. Comments in Reddit can be nested many times. There is virtually no depth limit in nesting level.
2. Comments in Reddit are sorted by default using a complex algorithm called *BEST*, which applies to all comments of the same level that share the same parent item (comment or submission). Further details about this algorithm can be found in [22].

To address the former issue, we have to read all comments of a submission and create a *tree* by using the **parent_id** tag provided by Pushshift. Then traverse the graph using a *depth-first search* (DFS) algorithm. As cited by Wikipedia: "*The (depth-first search) algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before*

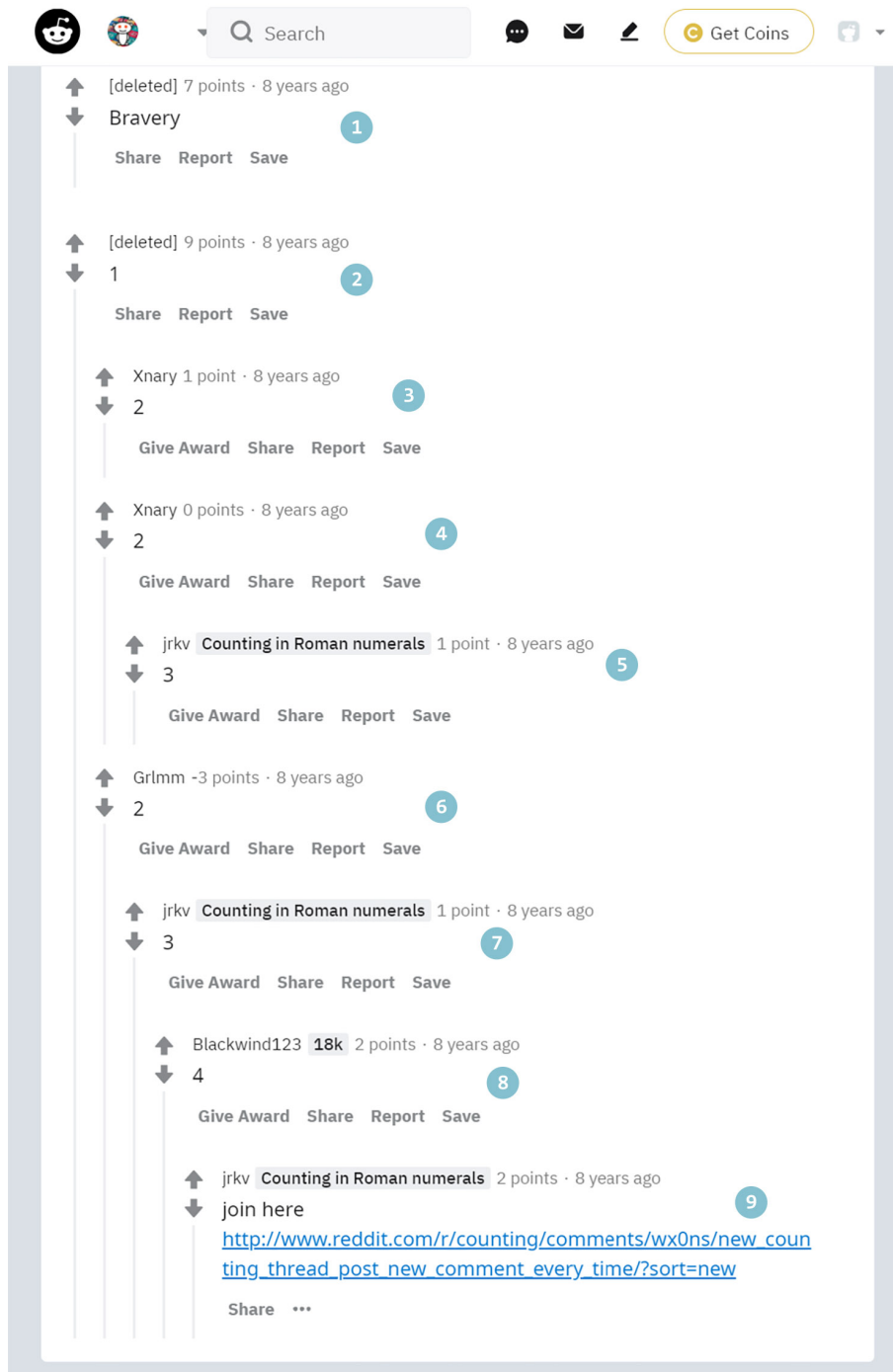


Figure 3: Reddit comments ordering example.

backtracking". In this case, the root node is the submission itself.

There may be some (very rare) cases, though, in which some comments were not retrieved, turning our tree into a *disconnected directed graph*. A disconnected directed graph is a graph in which exist two nodes such that, if we replace directed edges with undirected edges, there is no path having those nodes as endpoints. Hence, we need

to run DFS over every *connected component* in the graph, selecting the root node as the one without any incoming edge.

About the latter issue, we cannot compute the BEST algorithm because we would need to know the number of upvotes and the total number of votes. Unfortunately, **score** tag provided by the API is defined as the number of upvotes of the comment minus the number of downvotes. In this case, the best approximation we can get is to sort by **score** itself.

To develop what described so far, classes `Binder` and `Graph` were implemented. Relevant code snippets are shown below:

Listing 3: Methods `add_to_buffer` and `bind_submission_with_comments` of `Binder` class

```
def add_to_buffer(self, path_to_file):
    in_memory_json = {}
    with jsonlines.open(path_to_file) as reader:
        for obj in reader:
            if obj['link_id'] not in in_memory_json:
                in_memory_json[obj['link_id']] = []
            in_memory_json[obj['link_id']].append(obj)
    self.buffer.append(in_memory_json)

def bind_submission_with_comments(self, nested_bool):
    comments_files = []
    for root, dirs, files in os.walk(self.input_directory):
        :
        for fname in filter(lambda fname: fname.startswith('
            RC_') and fname.endswith('.jsonl'), files):
            comments_files.append(fname.replace('RC_', '').
                replace('-', '').replace('.jsonl', ''))

    comments_files.sort()
    iterations = len(comments_files)

    # Initialization
    if iterations != 0:
        output_dir = self.input_directory + '
            _submissions_comments/'
        Path(output_dir).mkdir(parents=True, exist_ok=True)
        if iterations < 7:
            loop = iterations
        else:
            loop = 7
    else:
        return

    logger.info('Loading buffer of files')
    for i in range(loop):
```

```

rc_file_dir = os.path.join(self.input_directory, 'RC_
' + comments_files[i][:4] + '-' + comments_files[i
][4:] + '.jsonl')
if os.path.isfile(rc_file_dir):
    self.add_to_buffer(rc_file_dir)

for j in range(iterations):
    filename_suffix = comments_files[j][:4] + '-' +
        comments_files[j][4:] + '.jsonl'
    rs_file_dir = os.path.join(self.input_directory, 'RS_
' + filename_suffix)
    if os.path.isfile(rs_file_dir):
        logger.info(' Working on file RS_' + filename_suffix
        )
        with jsonlines.open(os.path.join(output_dir, 'RSRC_'
+ filename_suffix), mode='w') as writer:
            with jsonlines.open(rs_file_dir) as reader:
                for obj in reader:
                    obj['comments'] = []
                    submission_id = 't3_' + obj['id']

                    g = Graph()
                    for i in range(len(self.buffer)):
                        if submission_id in self.buffer[i]:
                            for comment in sorted(self.buffer[i][
                                submission_id], key=lambda k: (k['score'], -
                                k['created_utc')):
                                g.addEdge(comment['parent_id'], 't1_' +
                                    comment['id'])
                                obj['comments'].append(comment)
                    if nested_bool:
                        order_list = g.DFS_iterative(submission_id)
                        obj['comments'] = sorted(obj['comments'], key=
                            lambda x: order_list[1:].index('t1_' + x['id'
                                ]))
                    writer.write(obj)
    if len(self.buffer) > 0:
        self.buffer.pop(0)
    if j + 7 < iterations:
        self.add_to_buffer(os.path.join(self.input_directory
        , 'RC_' + comments_files[j + 7][:4] + '-' +
        comments_files[j + 7][4:] + '.jsonl'))

```

Listing 4: Methods *DFSUtil_iterative* and *DFS_iterative* of *Graph* class

```

def DFSUtil_iterative(self, s, visited, order_list):
    # Create a stack for DFS
    stack = []

    # Push the current source node.
    stack.append(s)
    while (len(stack) != 0):

```

```

s = stack.pop()

if (not visited[s]):
    order_list.append(s)
    visited[s] = True

i = 0
while i < len(self.graph[s]):
    if (not visited[self.graph[s][i]]):
        stack.append(self.graph[s][i])
    i += 1

def DFS_iterative(self, v):
    order_list = []

    visited = {v_key: False for v_key in [item for sublist
        in self.graph.values() for item in sublist]}
    visited.update({v : False})

    self.DFSUtil_iterative(v, visited, order_list)

    while not all(visited.values()):
        logger.info("Broke graph:", v)
        subtree_root = self.find_subtree_root(visited)
        if subtree_root:
            self.DFSUtil_iterative(subtree_root, visited,
                order_list)
        else:
            break

    return order_list

[...]
```

2.2 DATASET PRE-PROCESSING

2.2.1 Pre-processing steps

Next step of this experimentation is to clean and enhance data before training the model. The following pre-processing tasks were planned:

1. **Markdown Syntax Clean-up:** needed to extract plain text of submissions (comments) body from markdown encoded **self-text (body)** json tag.
2. **Part-of-Speech tagging:** to concatenate names made of two or more words into a single one using underscores. More details about this step are in section 2.2.3

2.2.2 Markdown Syntax Clean-up

Pushshift provides the body of submissions and comments as *Markdown* encoded text. Markdown is an effortless lightweight markup that can be easily converted to HTML and other formats. Official markdown syntax can be found at <https://daringfireball.net/projects/markdown/syntax>

Some natural language models require textual documents to be encoded in a specific syntax before being trained. The word embedding model that we are going to use next needs plain-text only, instead. Therefore, all markdown syntax must be removed. In python, this can be achieved by using a Markdown parser that outputs a HTML code (e.g. Mistune v2⁹), and then retrieving decoded text with a further HTML parser (e.g. BeautifulSoup4¹⁰)

Reddit however uses a **custom** Markdown syntax which is slightly different from the official one and is fully documented at <https://www.reddit.com/wiki/markdown>. This makes standard Markdown parsers unusable without writing an ad-hoc extension. However, our requirement is not to *parse* any Markdown, but just to *remove* any encoding. This could be worked around by using a series of *regular expressions*.

Regular expressions (a.k.a. regex) describe a syntax used to define patterns for strings handling. Regex engines are available for the majority of the existing programming languages, making this syntax a de-facto standard for strings manipulation. More about regular expressions can be read in the original 1968 paper by Ken Thompson[30].

A new class, called `Preprocess`, provides method `preprocess_sentence_regex`, to carry out this clean-up process. Markdown links were the most difficult part to handle, the syntax for a hyperlink is the following:

```
[Roma](https://en.wikipedia.org/wiki/Rome)
```

We were interested in removing the url (within round braces) while preserving the title of the link (within square braces). However, the Reddit Markdown documentation states: *'Note that links can only contain parentheses if they are "balanced" — that is, if every "(" is later followed by ")"*. To link to a URL with unbalanced parentheses, either escape

⁹ <https://github.com/lepture/mistune>

¹⁰ <https://www.crummy.com/software/BeautifulSoup/>

the parenthesis with backslash ("\"), or use the alternate linking syntax, enclosing the URL in matched angle brackets, "<" and ">".

Hence, URLs (that are already surrounded by round braces) can contain both balanced or unbalanced inner round parenthesis. In the former case allowing for **multiple nested braces**. This was handled writing a regex called `MD_URL_INNERMOST_PARENTHESIS` to be executed in loop until no matches occur. This regular expression uses *variable-length lookbehind and lookahead* that are not natively supported by the Python regex engine, a custom one called *mrab-regex*¹¹ was used instead.

Code snippet of `preprocess_sentence_regex` method is provided below:

Listing 5: Method `preprocess_sentence_regex` of `Preprocess` class

```
MD_URL_INNERMOST_PARENTHESIS = re.compile(r'
    (?<=\\[.+\]\(^\)] ) \\((^\()]\) \\) (?=[^(\)])', re
    .UNICODE) # v 1.0.0

MARKDOWN_LINKS = re.compile(r'\\([^\[\]]\\)
    \\([^\] \\) \\)', re.UNICODE) # v1.0.0

# Regex copyright https://emailregex.com/ edited by
# Mauro Papa
EMAIL_REGEX = re.compile(r'((mailto:)?[a-zA-Z0-9_+]+@
    [a-zA-Z0-9-]+\.[a-zA-Z0-9-]+)', re.UNICODE)

# Regex copyright Diego Perini https://gist.github.com/
# dperini/729294
# Ported in python by Peter Cheung
# Edited by Mauro Papa to check urls in the middle of a
# sentence
URL_REGEX = re.compile(
    u"(?: (?: (?: https?|ftp|git|steam|irc|news|mumble|ssh|
        ircs|ts3server): ) ? / ) ?"
    u"(?: \S+ (?: : \S ) ? @ ) ?"
    u"(?: "
    u"(?! (?: 10|127) (?: \. \d{1,3}) {3} )"
    u"(?! (?: 169\.254|192\.168) (?: \. \d{1,3}) {2} )"
    u"(?! 172\. (?: 1[6-9]|2\d|3[0-1]) (?: \. \d{1,3}) {2} )"
    u"(?: [1-9]\d?|1\d\d|2[01]\d|22[0-3])"
    u"(?: \. (?: 1?\d{1,2}|2[0-4]\d|25[0-5]) ) {2}"
    u"(?: \. (?: [1-9]\d?|1\d\d|2[0-4]\d|25[0-4]) )"
    u"|"
    u"(?: "
    u"(?: "
    u"[a-zA-Z0-9\u00a1-\uffff]"
    u"[a-zA-Z0-9\u00a1-\uffff_]{0,62}"
```

¹¹ <https://bitbucket.org/mrabarnett/mrab-regex/src/hg>


```

u")?"
u"[a-z0-9\u00a1-\uffff]\."
u")+ "
u"(?:[a-z\u00a1-\uffff]{2,}\.?)"
u")"
u"(?:\d{2,5})?"
u"(?:[/?#\s])?"
, re.UNICODE | re.I
)

[...]

@classmethod
def preprocess_sentence_regex(cls, sentence):
    if not sentence:
        return ''

    # https://www.reddit.com/dev/api/
    pre_replacements = [
        ('&lt;', ''),
        ('&gt;', ''),
        ('&', '\u0026'),
        (r'\\(', '%28'), # Remove escaped
        (r'\\)', '%29'),
        ('\\ ', ''),
        ('~~', ''),
        ('!<', ''),
        ('>!', ''),
        ('\\^', ''),
        (''', ''),
        ('#', ''),
        ('\\\"', ''),
        ('\\n', ''),
        ('\\r', '')
    ]

    for old, new in pre_replacements:
        sentence = re.sub(old, new, sentence)

    # Turn unicode values into unicode characters
    sentence = unicode.unidecode(sentence)

    # Convert html entities
    sentence = html.unescape(sentence)

    nb_rep = 1

    # Remove innermost parenthesis in MD links
    while (nb_rep):
        (sentence, nb_rep) = cls.MD_URL_INNERMOST_PARENTHESIS
            .subn(r'%28\\1%29', sentence)

```

```

sentence = cls.MARKDOWN_LINKS.sub(r'\1', sentence)
sentence = cls.EMAIL_REGEX.sub('', sentence)

# Removes non-MD urls
sentence = cls.URL_REGEX.sub('', sentence)

post_replacements = [
    (r'(?<=[^_\s])_(?=[^_\s])', ' '),
    ('_', ' '),
]

for old, new in post_replacements:
    sentence = re.sub(old, new, sentence)

return sentence

```

2.2.3 Part-of-Speech tagging

As stated in section 1, this dataset will be used to feed a Word Embedding model. This kind of NLP model can turn words into numerical vectors, such that the distance between vectors expresses the semantic similarity between words. But since every word becomes a vector, if a name of a product or a brand is made of two or more words, then the model will turn that name in multiple vectors *splitting the meaning of the original name*.

For example, suppose we have "six" and "six financial". The former refers to a number and the latter to a financial information provider. If we would give those names as input of the Word Embedding model, it would produce two vectors: "six" and "financial". Therefore, we would lose any reference to the original name "six financial". Instead we would like to get a vector for "six" and another one for "six financial", as in picture 5.

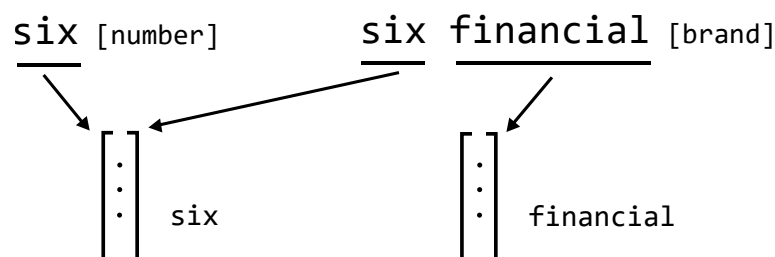


Figure 4: Current Word Embedding output.

This can be achieved just by concatenating those names into a single word, by means of underscores or other special characters, i.e. *six*

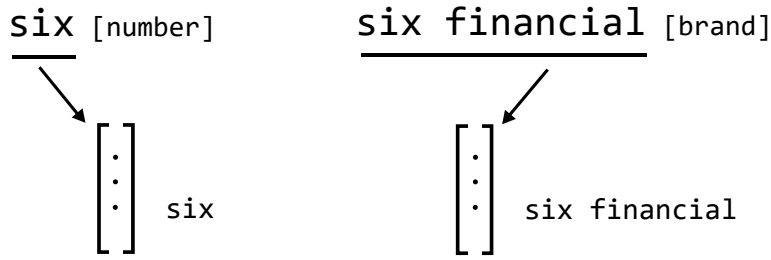


Figure 5: Desired Word Embedding output.

financial would become *six_financial*. Word Embedding model is then forced to treat them as single entities.

Due to the huge size of the collected corpus, this task surely can not be carried out by hand. A Part-of-Speech model, indeed, is an NLP model that computes grammatical tagging over sentences, identifying each word as a noun, verb, adjective, adverb, etc. This can be used to concatenate all names (nouns) close to each other using underscores. There are several Part-of-Speech frameworks out there. Among the others, *Flair*¹², although being notoriously slower if compared to other frameworks like *spaCy*¹³, can provide state-of-the-art results. Further details about Flair can be found in [2].

Flair provides three english PoS models (see table 2). These models are ready-to-use since they are already trained on an ad-hoc dataset called *Ontonotes* [32].

Table 2: Flair PoS English models

ID	Task	Training Dataset	Accuracy
'pos'	PoS Tagging (fine-grained)	Ontonotes	98.19
'upos'	PoS Tagging (universal)	Ontonotes	98.6
'pos-fast'	PoS Tagging (fine-grained)	Ontonotes	98.1

To speed-up the overall inference time, *pos-fast* model was picked. The computation was carried out by both my home desktop computer and by Google Colab, with the following configurations:

Table 3: Home desktop computer configuration

CPU	RAM	GPU
Intel i7 4770k	16 Gb DDR3	Nvidia gtx 750-ti

However, inference time turned out to still be very slow if used to tag one comment or submission at a time. A huge speed improvement can be achieved by passing to the tagger a *list of sentences* and

¹² <https://github.com/flairNLP/flair>

¹³ <https://spacy.io>

Table 4: Google Colab configuration

GPU	GPU Memory	RAM
Nvidia K80 / T4	12 / 16 Gb	25 Gb

enabling *mini-batches* during inference. This means that, before computing PoS, the algorithm has to read all or as many phrases as possible from a file, introducing risk of memory saturation. Once the tagging is done, the algorithm has to rebuild the file preserving the original json encoding.

Moreover, even with a small number of sentences in the input list, the tagger is not able to handle phrases that are too long. Hence, we split them at every occurrence of the *dot* character and defined a `max_sentence_size` threshold. If after the splitting, a period is still longer than `max_sentence_size`, it is split again at the *right-most* space before the `max_sentence_size`-th character (method `split_just_before_limit` of `Preprocess` class). Default value for `max_sentence_size` was set to 2500 characters.

The list of input sentences to pass to the model is in reality a list of instances of the flair `Sentence` class. This class provides a method to return the tagged text after the inference process. Therefore, we can write a child class called `SentencePoS` that inherits from `Sentence` and provides an additional method (`to_concatenated_string`) to return the original string with consecutive nouns concatenated together. It is important to specify that Flair gives a *confidence score* for every tag. The above method only considers noun tags with a confidence greater than 0.65.

The overall tagging process was mainly carried out with the desktop configuration above, however also Google Colab was used to infer some files. This means it is hard to give an exact timing of the whole tagging process, but it can be estimated between 10 and 12 days.

Due to the length of the code described above, relevant snippets only are provided:

Listing 6: Methods `flair_pos_tagging` and `split_just_before_limit` of `Preprocess` class

```
@staticmethod
def flair_pos_tagging(list_of_sentences, pos_model):
    list_of_sentences = [SentencePoS(sentence) for
        sentence in list_of_sentences]
    list_of_sentences = pos_model.predict(
        list_of_sentences, mini_batch_size=16)
    return [sentence.to_concatenated_string() for sentence
        in list_of_sentences]
```

```

[...]

@staticmethod
def split_just_before_limit(sentence, max_sentence_size
=2500):
    list_of_sentencesSplitted = re.findall(r'\s?[^\\s]+',
        sentence)
    list_of_sentences = []
    current_sentence = ''

    for splitted in list_of_sentencesSplitted:
        if len(splitted) + len(current_sentence) >
            max_sentence_size:
            if current_sentence != '':
                list_of_sentences.append(current_sentence)

            if len(splitted) > max_sentence_size:
                list_of_sentences += re.findall(r'.{{1,{
                    max_sentence_size}}}', splitted)
                current_sentence = ''
            else:
                current_sentence = splitted
        else:
            current_sentence += splitted

    if current_sentence != '':
        list_of_sentences.append(current_sentence)

    return list_of_sentences

[...]
```

Listing 7: *SentencePoS* class that inherits from Flair *Sentence* class

```

class SentencePoS(Sentence):

    def is_noun(self, span):
        suitable = ["NOUN", "PROPN"]
        if span.tag in suitable and span.score > 0.65 and len
            (span.text) > 1:
            return True
        else:
            return False

    def process_noun_from(self, noun):
        return re.escape(re.sub(r'\s+', ' ', noun.strip()))

    def process_noun_to(self, noun):
        return re.sub("'", "", re.sub(r'\s+', ' ', noun.strip
            ()))
```

```

def to_concatenated_string(self):
    list_replacements = [[]]

    for span in self.get_spans('pos'):
        # Spans remove leading and trailing spaces but keep
        # \n
        # \n is treated like dot
        if self.is_noun(span):
            if span.text[0] == r"\n" or span.text[0] in string.
                punctuation or (list_replacements[-1] and
                    list_replacements[-1][-1][-1] in string.
                        punctuation):
                list_replacements.append([span.text])
            else:
                list_replacements[-1].append(span.text)
        elif list_replacements[-1] != []:
            list_replacements.append([])

    if list_replacements[-1] == []:
        list_replacements.pop()

    concatenated_string = re.sub(r'\s+', ' ', self.
        to_original_text())
    for noun in list_replacements:
        try:
            noun_from = self.process_noun_from(' '.join(noun))
            noun_to = self.process_noun_to(' '.join(noun))
            concatenated_string = re.sub(
                noun_from, noun_to, concatenated_string)
        except Exception as error:
            logger.error(error)
            logger.error('Bad noun is: %s' % noun)

    return concatenated_string

```

3

WORD EMBEDDING MODEL TRAINING

3.1 INTRODUCTION TO WORD EMBEDDING

Word embedding models are neural networks that provide vector representations for words, i.e. they allow to turn words into numerical vectors, such that the distance between vectors expresses the semantic similarity between words.

The most famous algorithms for word embedding are:

1. **Word2Vec** developed by Tomas Mikolov[20][21]
2. **GloVe** developed by Stanford University[25]
3. **fastText** developed by Facebook's AI Research (FAIR) lab[8][17]

In general, embeddings generated by Word2Vec and GloVe tend to perform very similarly¹. On the other hand, fastText can also handle words that never appeared in the training dataset (OOV, out-of-vocabulary) by splitting each words in smaller n-grams. However, if we compare the results of Word2Vec against fastText, as done in [27], we can see that Word2Vec model seems to perform better on semantic tasks, because information from irrelevant n-grams worsens the embeddings. Instead, fastText embeddings are significantly better at encoding syntactic information.

The main idea behind the search engine that we aim to build, is to discover new financial services by selecting the words that correspond to the neighbours vectors of an input one (i.e. name of the service we are searching competitors for). Since there is no evidence that similar financial services tend to have similar names, Word2Vec seems a logic choice for this project.

3.2 TRAINING PHASE

In order to train Word2Vec, it is needed to write an iterator class (`CorpusIter`) that goes through the whole dataset and feeds the model while training. This class also handles:

1. **Tokenization** of the sentences in the dataset

¹ <https://www.quora.com/How-is-GloVe-different-from-word2vec>

2. Bot detection and filtering

Listing 8: *CorpusIter* class

```

class CorpusIter(object):
    def __init__(self, dirname, bot_file, limit):
        self.dirname = dirname
        self.bots = []
        count = 0
        with jsonlines.open(bot_file) as reader:
            # pylint: disable=not-an-iterable
            for list_line in reader:
                if count >= limit:
                    break

                count += 1
                self.bots.append(list_line[0])

    def preprocess_tokenizer_wrap(self, obj, censor=True):
        document_list = []

        if 'title' in obj :
            document_list.append(obj['title'])

        if 'selftext' in obj :
            document_list.append(obj['selftext'])

        for comment in obj['comments']:
            if comment['author'] not in self.bots and comment['author'][-3:] != 'bot' and comment['body'] not in document_list:
                document_list.append(comment['body'])

        document = ' '.join(document_list)

        if censor:
            document = profanity.censor(document, '')

        return Preprocess.preprocess_tokenizer(document)

    def __iter__(self):
        for r, d, f in os.walk(self.dirname):
            for file in f:
                if '.jsonl' in file:
                    print('Working on file: ' + file)
                    with jsonlines.open(os.path.join(r, file)) as reader:
                        for obj in reader:
                            document = self.preprocess_tokenizer_wrap(obj)
                            if document != []:
                                yield document

```

Tokenization takes a sentence as input and returns a list of lower-case tokens (words), filtering out undesired characters. These tokens are generated by a function of the `Preprocess` class that is based on the use of regular expressions. This regex select every consecutive occurrence of: letters (a-z), underscores and & character. It discards all numbers and every other special character. Please note that:

1. Underscores are preserved due to Part-of-Speech tagging
2. & character is preserved because it is sometimes used in financial vendor names, e.g. "Standards & Poors".
3. **Numbers are discarded because they are most likely tagged with label NUMBERS by the Part-of-Speech and not with NOUNS. This means we are potentially unable to distinguish names containing numbers from names just written before or after a number. Hence method `to_concatenated_string` from class `SentencePos` will never concat a number together with a name (see code listing number 7).**

Listing 9: Methods `tokenize` and `preprocess_tokenizer` of `Preprocess` class

```
PAT_ALPHABETIC_UNDERSCORE = re.compile(r'(((?![\d])[A-Za-z_&])+)', re.UNICODE)

[...]

@classmethod
def tokenize(cls, sentence):
    sentence = deaccent(sentence.lower())
    for match in cls.PAT_ALPHABETIC_UNDERSCORE.finditer(
        sentence):
        yield match.group()

[...]

@classmethod
def preprocess_tokenizer(cls, sentence, min_len=2,
    max_len=50):

    if sentence == '[ deleted ]' or sentence == '[deleted]'
        ' or not sentence:
        return []

    tokens = [
        token for token in cls.tokenize(sentence)
        if min_len <= len(token) <= max_len and not token.
            startswith('_')
    ]

    return tokens
```

Bot detection is carried out measuring the average response time of every user that appeared in the dataset. Most likely bot candidates are the ones with the smallest response delay. In Pushshift Github repository there is a script to be run over the monthly dumps that does exactly this kind of measurement², ordering users by response time in ascending order and providing a bunch of extra useful metrics. By manually evaluating the names of the accounts in the output list of this script, it was decided to discard the comments of the first 250 accounts. Moreover, comments (or submissions) post by users with name ending with "bot" are skipped too.

The above filters are really effective and remove a good amount of bots without building a machine learning classifier for bot detection. Few false positives or false negatives can be allowed for this kind of experimentation.

The training phase of a word embedding model require some hyper parameters to be fine-tuned. For this reason, six different models were trained (see table 5).

Table 5: Word embedding models trained with different hyperparameres

	size Embeddings dimension	min_count Minimum word occur.	sg Skipgram (sg=1) or CBOW (sg=0)
m1_s300_cb	300	1	0
m1_s100_cb	100	1	0
m5_s100_cb	100	5	0
m5_s100_sg	100	5	1
m5_s300_cb	300	5	0
m5_s300_sg	300	5	1

All these models were trained using 5 iterations (epochs) over the corpus, this has been done not to excessively increase the average training time, that is now around two days and six hours. More details are shown in table 8.

The hardware configuration used for training is the same described in section 2.2.3 in table 3.

The choice of `min_count` values has been done looking at the dataset vocabulary. This text file provides *word-occurrences* pairs and was extracted from the first trained model (m1_s300_cb). The goal was to

² <https://github.com/pushshift/Reddit-Bot-Detector>

find a threshold which could remove enough garbage words without removing too many names/brands of products. After a manual inspection `min_count = 5` seemed a fair value, `min_count = 1` was left for completeness.

Please note that CBOW and Skip-Gram are the only algorithms used in literature for this particular model, and they were introduced in the original Word2Vec paper. Both of them describe how the neural network learns the underlying word representation.

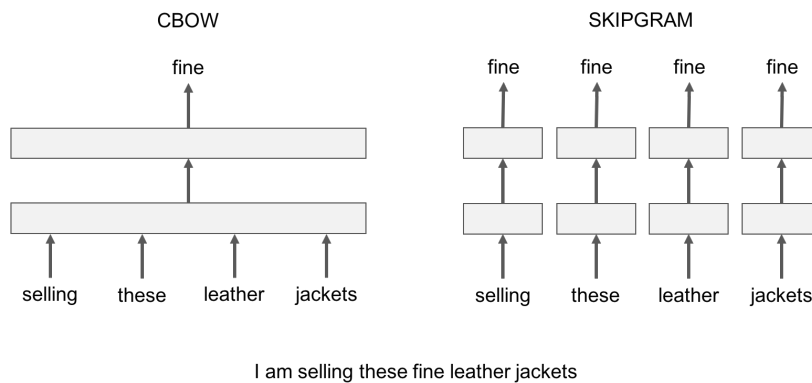


Figure 6: CBOW vs Skipgram. Source: <https://fasttext.cc/>

CBOW (Continuous Bag-of-Words) tries to predict a target word given a context (fixed size window of surrounding words to the target), while Skipgram learns to predict a target word thanks to a nearby word.

For example, given the sentence *I am selling these fine leather jackets* and *fine* as target word, Skipgram model tries to predict the target using a close-by word. CBOW, instead, selects a window of surrounding words and uses the sum of their vectors to predict the target.

Further details can be found in the original Word2Vec paper by Thomas Mikolov[21]. Accuracy measurement of the above models is carried out in section 3.3

3.3 TEST SET DEFINITION AND EVALUATION TECHNIQUE

In order to evaluate the word embedding model, Banca d'Italia business experts defined a set of 22 words, appeared at least once in the dataset, to be searched in the model. Such target words are listed below:

1. bloomberg_terminal
2. dow_jones
3. eikon
4. euro_stoxx
5. factset
6. finastra
7. iboxx
8. intex
9. metastock
10. moodys_analytic
11. morningstar
12. morningstar_ratings
13. morningstar_research
14. msci_emerging_markets
15. pitchbook
16. refinitiv
17. reuters_news
18. stoxx
19. tradeweb
20. vanguard_money_market
21. xetra
22. ycharts

For each of these words, the first 15 most similar terms returned by the word embedding model were used to build a test set. Since six different models were trained, there is a total of $15 \times 22 \times 6 = 1980$ results to evaluate. **Filtering duplicates returned by these w2v models, the number of results in the test set drops to 966.** Banca d'Italia business experts made a **boolean (GOOD/BAD) evaluation** of these results by hand. In particular:

1. If a term was already known, it was immediately classified as GOOD or BAD in relation to the corresponding target word

2. If a term was not known, an additional online search was needed to figure out if it was somehow related to its target word.

This kind of evaluation was time-consuming but allowed us to get the most reliable measurement of the model accuracy. An example of manual evaluation of target word *iboxx*, for model **m5_s300_cb**, is presented in table 6.

Table 6: Manual evaluation of target word *iboxx* for model **m5_s300_cb**

Target term	W2V similar term	Cosine sim.	Label
iboxx	powershares	0.736	GOOD
iboxx	ishares_u	0.735	GOOD
iboxx	ishares_core	0.727	GOOD
iboxx	ishares_core_s	0.723	GOOD
iboxx	xtrackers	0.723	GOOD
iboxx	year_treasury_bond_etf	0.721	BAD
iboxx	core_s	0.720	BAD
iboxx	aggregate_bond	0.707	BAD
iboxx	yr_ucits_etf	0.702	BAD
iboxx	large_cap_etf	0.690	BAD
iboxx	invesco_s	0.692	GOOD
iboxx	vfmv	0.688	GOOD
iboxx	xslv	0.680	GOOD
iboxx	schwab_us	0.680	GOOD
iboxx	ishares_s	0.679	GOOD

Please note that at this stage no *false positives* or *false negatives* are available. Therefore no recall, precision or f1 can be computed for the moment.

3.4 ACCURACY MEASUREMENT

Accuracy for manual evaluation of the test set is defined below:

$$\text{Accuracy} = \frac{\text{Number of elements labeled as GOOD}}{\text{Total number of elements}} \quad (3.1)$$

As presented in table 7, model **m5_s300_cb** has the best accuracy with 80% of good results, while model **m1_s300_cb** follows with 77%.

Hence, skipping words with a number of occurrences that is too low, while training the w2v model, seems to slightly increase the accuracy for this test set.

Table 7: Manually evaluated accuracy for Word embedding models.

Model name	Num. of results per model	Accuracy
m1_s300_cb	22 x 15 = 330	0.778
m1_s100_cb	22 x 15 = 330	0.715
m5_s100_cb	22 x 15 = 330	0.757
m5_s100_sg	22 x 15 = 330	0.712
m5_s300_cb	22 x 15 = 330	0.8
m5_s300_sg	22 x 15 = 330	0.739

Let's also point out that models m5_s300_cb and m5_s300_sg are trained with the same values for `size` and `min_count`. **Therefore, CBOW reaches higher accuracy if compared to Skip-Gram, over this dataset, with this test set.**

Since these models will need to be loaded in-memory, it is also important to compare their size to better understand the hardware (RAM) requirements needed at run-time. These details are provided in the table 8, together with training and loading times.

Table 8: Word embedding models size and loading time.

Model name	Size	Training time (Cython enabled)	Loading time
m1_s300_cb	8.5 GB	2d 9h 12m	3m 57s
m1_s100_cb	2.92 GB	2d 4h 50m	2m 57s
m5_s100_cb	338 MB	2d 3h 44m	18s
m5_s100_sg	338 MB	2d 4h 35m	16s
m5_s300_cb	993 MB	2d 8h 25m	20s
m5_s300_sg	993 MB	2d 9h 14m	22s

The accuracy of m1_s300_cb and m5_s300_cb are very close to each other. By the table above, however, it is clear that m1_s300_cb is also the biggest and slowest model to load. On the other hand, model m5_s300_cb seems to be the best choice, because it provides either a higher accuracy and a smaller size. Please note that "model loading time" refers to the initialization time and not to the time to carry out a single query.

During the evaluation process, Banca d'Italia experts were able to identify new alternative financial services that require further investigation and analysis. Although the number of discovered services depends on Banca d'Italia previous knowledge and it is not representative of the model accuracy, it is an important proof of the goodness of the model built so far.

Table 9: New discovered financial services for Word embedding models.

Model name	Number of discovered financial services
m1_s300_cb	26
m1_s100_cb	32
m5_s100_cb	33
m5_s100_sg	40
m5_s300_cb	26
m5_s300_sg	29

As we can see from table 9, model m5_s100_sg has the highest number of discovered services.

Initially, we thought this could be related to the usage of Skip-gram, because Mikolov himself stated that: *Skip-gram works well with small amount of the training data, represents well even rare words or phrases; CBOW is several times faster to train than the skip-gram, slightly better accuracy for the frequent words*³. However, we would have expected to see a similar spike also in model m5_s300_sg which was trained with Skip-gram too, but this did not happen. Therefore, we do **not** have any strong statistical evidence that Skip-gram performs better to find less-known financial products in this test set.

³ <https://groups.google.com/g/word2vec-toolkit/c/NLvYXU99cAM/m/E5ld8LcDx1AJ>

4

NAMED ENTITY DISAMBIGUATION

4.1 INTRODUCTION TO NAMED ENTITY DISAMBIGUATION

Named-entity disambiguation (NED), also known as Entity-linking or Wikification, is the Natural Language Processing task of mapping words of interest (named-entities) from an input text to corresponding unique entities in a target Knowledge Base. Name-entity disambiguation is different from Named-entity Recognition which instead tries to classify names of interest into pre-defined categories such as *person, place, organization, etc.*

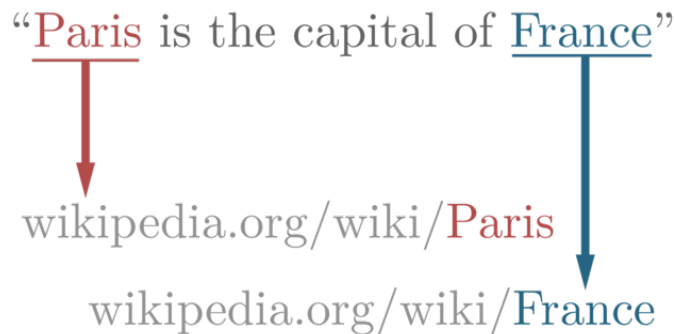


Figure 7: NED example. Source: https://en.wikipedia.org/wiki/Entity_linking

Named entity algorithms available in literature always take a sentence as input and extract the named-entities to disambiguate. In this case, instead, the input of the NED algorithm is the output of the word embedding model, i.e. a list of semantically similar words.

In this chapter, the development of a custom NED algorithm that is capable of dealing with this kind of input will be presented.

4.2 CHOICE OF THE KNOWLEDGE BASE

When developing a NED algorithm, the first choice to make is the Knowledge Base to use for matching. From now on, let's use the term Knowledge **Graph** instead of Knowledge Base. Although there is quite a debate on the real definition of Knowledge Graph, for the purposes of this project we can define it as a *RDF-based Knowledge*

Base.

RDF (aka Resource Description Framework) it's a W3C specification[18] for semantic data modelling. It is based on the use of *triples*, i.e. statements of the form subject–predicate–object. Either the subject, the predicate and the object must have a unique identifier. Such kind of data encoding is easily readable by both humans and machines. RDF, together with ontology languages, is at the base of the so-called *Semantic Web*¹

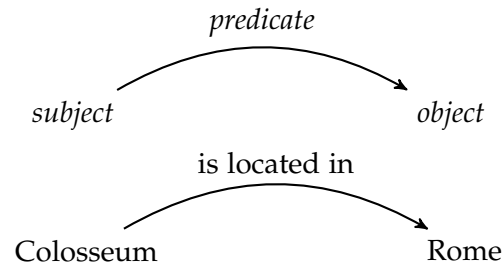


Figure 8: A generic RDF triple (top). An real example of RDF triple (down).

Both Google and Microsoft have private Knowledge Graphs. The former one, called Google Knowledge Graph is freely accessible by means of an API that, at the time of writing, is defined as *not suitable for use as a production-critical service*². The latter, called Bing Knowledge Graph, can be accessed as paid service through the Bing Entity Search API³.

Despite them, there is also a larger body of open Knowledge Graphs, such as DBpedia[7], Wikidata⁴ and YAGO⁵[24]. An overview of how these databases are populated and the interlinking between them can be found at [14]. The choice for the best fitting Knowledge Graph depends on project domain and requirements. In this experimentation, we are probably going to need matching also some *not-well-known entities*, like minor financial products with small market shares. As highlighted by the comparison work among these KGs in [11], Wikidata is the best suited for this kind of use-case. Moreover, it can be edited at any time by the community and it is continuously queryable.

The most important aspect of Wikidata, however, is its bigger number of entities if compared to other open bases. As of today, Wikipedia features 89 millions of content pages⁵ (October 2020) versus the 5 millions entities of DBPedia[14] (Version 2016-10) and the 50 million en-

¹ https://en.wikipedia.org/wiki/Semantic_Web

² <https://developers.google.com/knowledge-graph>

³ <https://azure.microsoft.com/en-gb/services/cognitive-services/bing-entity-search-api/>

⁴ <https://en.wikipedia.org/wiki/Wikidata>

⁵ <https://www.wikidata.org/wiki/Wikidata:Statistics>

titles of YAGO₄⁶

Of course, due to the language-dependent nature of the NLP models used so far, we are only going to disambiguate among English entities.

Wikidata is a document-oriented NoSQL database for entities collection. Each entity represents a topic, concept, or object and has an identifier starting with letter 'Q' (QID). There may be entities named with the same label but each of them has a different QID. Descriptions and aliases are provided too, if available.

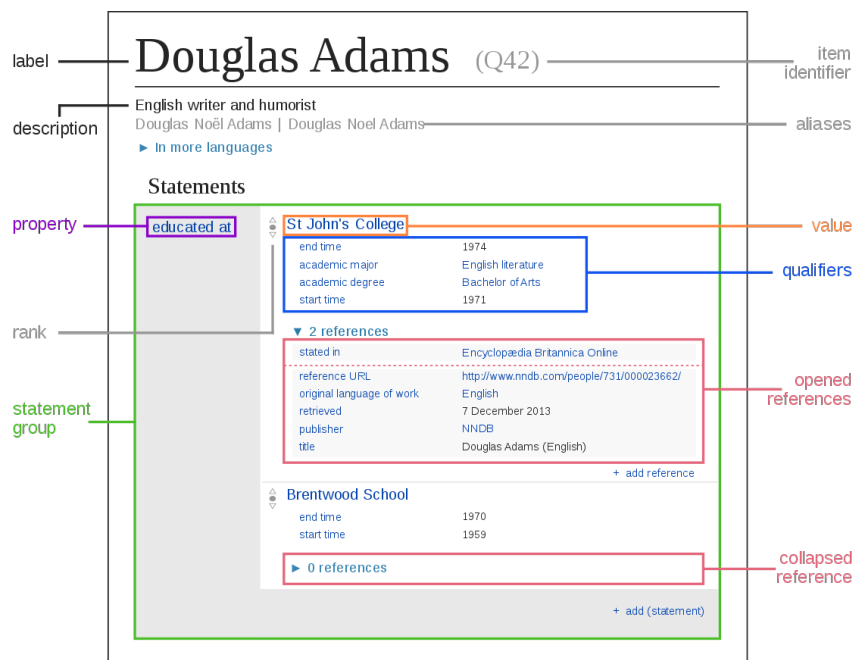


Figure 9: Wikidata page example. Source: <https://en.wikipedia.org/wiki/Wikidata>

Statements in Wikidata are recorded as *property-value* pairs. Every property has an identifier starting with letter 'P' and may have one or more entities as values. For the sake of this project, we are mainly interested in the following properties:

1. **Property P856 "official website"**: The official website of the entity, if available
2. **Property P831 "instance of"**: That class of which the current entity is a particular example and member. Since every entity of Wikidata can be used as a value for properties, we could have millions of potential classes to choose from.

⁶ <https://yago-knowledge.org/downloads/yago-4>

As we will see later, the algorithm also takes in special consideration the Wikipedia page linked to each entity, being the main source for an in-depth description of the entities.

4.3 CORE IDEA BEHIND NED ALGORITHM FOR WORD EMBEDDINGS

Before going through the full description of the NED algorithm developed for this project, it is important to clarify some differences between applying NED on the output of a Word Embedding model instead of on a sentence. Take for example the fifteen most similar words for term **bloomberg_terminal** from model **m5_s300_cb**, together with their cosine similarity:

1. bloomberg_terminals [0.7495272159576416]
2. capital_iq [0.6867133378982544]
3. datastream [0.6670815944671631]
4. capiq [0.6655162572860718]
5. eikon [0.6468931436538696]
6. wrds [0.6486623287200928]
7. capitaliq [0.6128060817718506]
8. yahoo_finance [0.6137667298316956]
9. bloomberg [0.6159719824790955]
10. factset [0.6060885787010193]
11. quandl [0.6075608730316162]
12. bamsec [0.5830370187759399]
13. ycharts [0.5825964212417603]
14. reuters_eikon [0.5766008496284485]

As we can see, there are some particularities that must be pointed out. First, the same product may appear multiple times with slightly different names (e.g. *eikon* and *reuters_eikon*). Second, the same word may appear again due to typing errors (e.g. *capital_iq* and *capitaliq*). Third, the similarity score provided by the word embedding model cannot be trusted too much in detail, i.e. *reuters_eikon* is the same of *eikon*, however, it is at the bottom of the list with the lowest similarity.

Last but not least, it is important to note that most of these words refer to the same context (trading platforms in this case) and this is extra information that must be exploited somehow. **Indeed, we want to disambiguate those words together, instead of one at a time, such that the context information does not get lost.** Let's try to clarify it with an example. Suppose you need to disambiguate the word *Apple*. A quick search on Wikidata outputs two candidate entities: *Apple Inc.* (Q312) and *Apple (fruit)* (Q89). If the word embedding model returned *Apple* as similar word for *Microsoft*, then with high probability we are referring to *Apple Inc.* (Q312). If, instead, the word embedding model returned *Apple* as similar word for *Orange*, then we are probably referring to *Apple (fruit)* (Q89).

In order to achieve this, we can imagine generating a new vector embedding on the description of each candidate entity, such that the more similar the descriptions, the closer the embeddings of those entities. This is somewhat analogous to what the word embedding models already do. However, we now need to encode a full-text document into a single vector and not just a single word.

Suppose we need to disambiguate a list of n terms. We search these words in Wikidata and get a set of candidate entities per term:

$$Q_i = \text{candidate entities for term } t_i; \text{ with } i = 1, \dots, n$$

We can then define the set C of all combinations of candidate entities as follows

$$C = \{c_j \mid c_j = \langle \vec{v}_1, \dots, \vec{v}_i, \dots, \vec{v}_n \rangle \text{ st } \vec{v}_i = e(q_i) \text{ and } q_i \in Q_i\}$$

where $e(q_i) = \vec{v}_i$ represents the formula to turn entity q_i into embedding \vec{v}_i . Of course, the choice of the right method to convert entities into embeddings is extremely important and influences the accuracy of the whole NED algorithm. As explained in section 4.4.5, a third machine learning model is used here.

The core idea of this algorithm is to select the combination such that the distance between its embeddings is the smallest among all other combinations.

Of course, since a combination may have more than two entities, instead of computing the sum of the pairwise cosine similarity between all the embeddings, we can just compute the sum of the similarity of each embedding against the mean vector \vec{m} of the combination itself. Thus we can choose the best combination c_{best} as

$$c_{\text{best}} = \operatorname{argmax}_{c_j \in C} \sum_{\vec{v}_i \in c_j} \text{cs}(\vec{m}_{c_j}, \vec{v}_i) \quad (4.1)$$

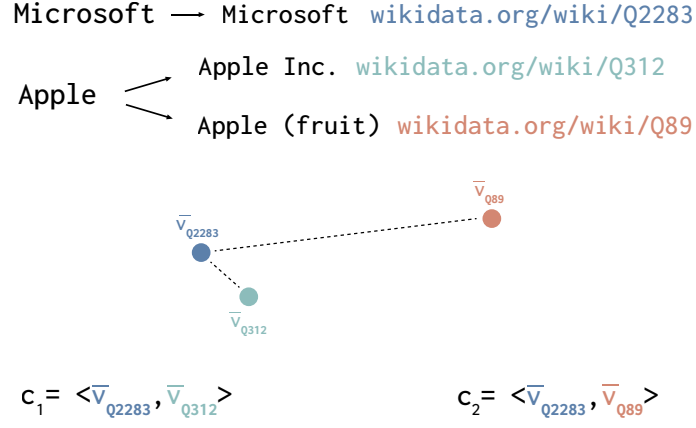


Figure 10: Context as distance between vector embedding of entities

where $cs(\vec{v}_a, \vec{v}_b)$ is the *cosine similarity* formula that returns a value $c \in [0, 1]$. The higher c the closer the argument vectors.

The formula above, however, suffers from a major drawback that may occur when the candidate entities of a searched term are very similar to each other. For example, we want to disambiguate together *Pepsi-cola* and *Sprite*, as in picture 11. Suppose that searching *Pepsi-cola* in Wikidata we get both *Pepsi-Cola* ($Q47719$) and *Coca-Cola* ($Q2813$), while *Sprite* returns *Sprite* ($Q206978$) only. Computing the embeddings for these entities we end up having two possible combinations:

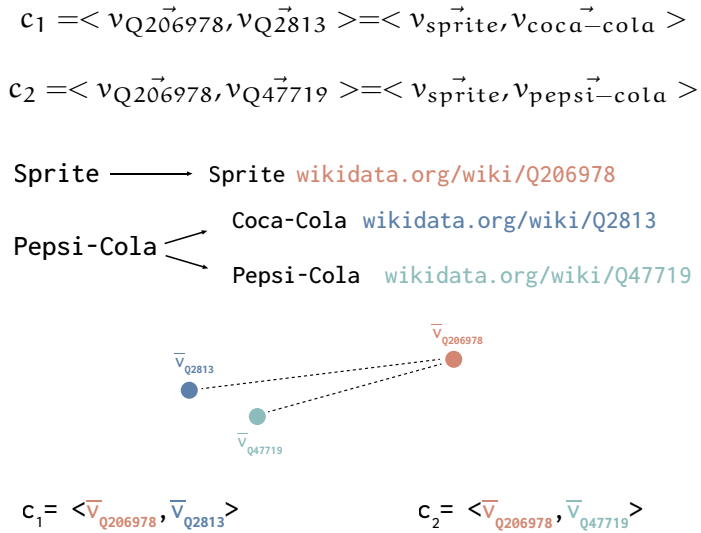


Figure 11: Failure example of NED based on formula 4.1

Either *Pepsi-Cola* ($Q47719$) and *Coca-Cola* ($Q2813$) refers to cola-based carbonated soft drinks and their embeddings will surely be close to each other. However, since both *Coca-Cola* ($Q2813$) and *Sprite* ($Q206978$) are produced by 'The Coca-Cola Company', they will probably be

closer than the distance between *Sprite* (Q206978) and *Pepsi-Cola* (Q47719). Hence, formula number 4.1 would fail choosing combination c_1 instead of combination c_2 .

This issue can be solved noting that the term *Pepsi-Cola* is more similar (equal) to the title of entity Q47719 (*Pepsi-Cola*) than title of entity Q2813 (*Coca-cola*). Hence, we can just compute an *edit distance* similarity $ed(string_1, string_2) \in [0, 1]$ between the searched term and the name of each candidate entity. The greater the value, the more similar the two strings. Formula 4.1 now becomes:

$$c_{best} = \operatorname{argmax}_{c_j \in C} \sum_{\vec{v}_i \in c_j} cs(m\vec{v}_{c_j}, \vec{v}_i) * ed(t_i, title_{q_i}) \quad (4.2)$$

Even if with different implementations, Knowledge Bases and machine learning models used, the idea behind this approach is similar to [23].

4.4 DESCRIPTION OF NED ALGORITHM FOR WORD EMBEDDINGS

This section will go through a detailed description of the NED algorithm developed for this project, that implements the idea described in 4.3.

Since both the word embedding model and this algorithm will be queried at runtime, the latter should be as fast as possible. However, **this project has no real-time requirement**. Therefore, the NED algorithm will not work with any preprocessed in-memory dump of Wikidata, but instead, it will deal with Wikidata (and Wikipedia) API, using *asynchronous* HTTP requests when possible, such that entities displayed to the user are always up-to-date.

To make the whole algorithm faster, it extensively uses *vectorization* instead of looping, i.e. most of the computation is done at optimized c-level. This is mainly achieved using specific python libraries, like: *SciPy*⁷, *NumPy*⁸ and *Pandas*⁹.

The code of this algorithm is composed of four consecutive batches of **asynchronous** HTTP requests to Wikipedia and Wikidata APIs. It can be virtually split into six main steps, that we are going to describe in the next subsections:

⁷ <https://www.scipy.org>

⁸ <https://numpy.org>

⁹ <https://pandas.pydata.org/>

1. **First Batch of HTTP requests** to collect candidate entities searching each term through the Wikidata and Wikipedia search engine via Web API.
2. **Second Batch of HTTP requests** to analyze disambiguation pages retrieved in the previous batch through the Wikipedia API.
3. **Third Batch of HTTP requests** to get property values and other information from Wikidata page of candidate entities.
4. **Fourth Batch of HTTP requests** to retrieve introduction extract from Wikipedia and *meta-description-tag* from entities official websites.
5. **Selection of the best combination of entities**
6. **Handling of unresolved and discarded terms**

4.4.1 First Batch of HTTP requests

The algorithm has been implemented by means of a `NED` class that provides method `disambiguate` for entity linking:

Listing 10: Prototype of method *disambiguate* of *NED* class

```
def disambiguate(self, terms, terms_cosine = [],
                 terms_occ = [], chunk_size = 3, min_true_nn = 1)
```

This method takes the following input parameters:

1. `terms` : a list of the most similar terms returned by the word embedding model when searching a target word. Such target word is assumed to be placed at the beginning of the list.
2. `terms_cosine` : a list of the cosine similarities returned by the word embedding model, in the same order provided for `terms`.
3. `terms_occ` : a list with the number of occurrences of input words in `terms` as counted in the dataset used for training the word embedding model.
4. `chunk_size` : value for combinations splitting to speed-up the computation. More about this is stated in 4.4.5.
5. `min_true_nn` : value for nearest neighbouring for Wikidata property *instance of* (*P31*). More about this is described in 4.4.5.

Before start making any HTTP request, the algorithm does a preliminary check on misspelt input words. As said in section 4.3, very

often the word embedding model, trained on this dataset, returns the same word either spelt correctly and with some minor typos. Consequently, if a pair of very similar words has been detected, the algorithm assumes the good one to be the one with the highest number of occurrences and discards the other. Hence, terms are first ordered by occurrences in descending order, then an *edit distance* metric is computed between each term and the ones with higher occurrences (method `get_st_couples`).

An *edit distance* is a measure that expresses how much similar two strings are. The algorithm used here is the *levenshtein_ratio* that gives a value $l \in [0, 1]$ equal to 1 when two strings are identical. The *levenshtein_ratio* is based on the *levenshtein_distance*, that can be informally defined as the minimum number of atomic edits needed to turn a string into another one. A detailed analysis of the *levenshtein_distance* can be found in the original paper by Vladimir Levenshtein from 1966 [19].

$$\text{levenshtein_ratio} = \frac{\text{levenshtein_distance}(\text{string}_A, \text{string}_B)}{\max(\text{len}(\text{string}_A), \text{len}(\text{string}_B))} \quad (4.3)$$

The levenshtein computation is carried out by the *FuzzyWuzzy*¹⁰ python library (method `max_fuzz_ratio`), that also provides other *edit distance* functions, used later on in this algorithm. At this stage, we only prune misspelt words that are very similar to the good ones, i.e. words with a *levenshtein_ratio* equal or greater than 0.9

Listing 11: Methods `max_fuzz_ratio`, `get_st_couples` and portion of method `disambiguate` of NED class

```
def max_fuzz_ratio(self, search_term, arr_like,
                  scorer_array = [fuzz.WRatio], filter_len = True):

    if len(arr_like) == 0:
        return (np.nan, 0)

    search_term = re.sub('_', ' ', search_term)
    assert search_term != np.nan
    mst_len = 1.5 len(search_term)

    # If there are strings, np.nan get converted to 'nan'
    labels = np.hstack(arr_like).astype('str')
    labels = labels[labels != 'nan']

    if labels.size == 0:
        return (np.nan, 0)
```

¹⁰ <https://github.com/seatgeek/fuzzywuzzy>

```

if filter_len:
    filter_labels = np.array([True if len(x) < mst_len
                              else False for x in labels])
    labels = labels[filter_labels]

max_match = tuple()

for scorer in scorer_array:
    sc = process.extractOne(search_term, np.char.lower(
        labels), scorer = scorer)
    if sc:
        max_match = max_match + sc
    else:
        max_match = max_match + (np.nan, 0)

return max_match

def get_st_couples(self, terms, terms_occ):

    sort_idx = terms_occ.argsort()[::-1]
    terms_sorted = terms[sort_idx]

    # Term with highest occurrences goes immediately in
    terms_pruned = [terms_sorted[0]]
    df_rpl = pd.DataFrame(columns=['st_kept', 'st_rpl'])

    for st in terms_sorted[1:]:
        st_kept, kept_score = process.extractOne(st,
            terms_pruned, scorer = fuzz.ratio)
        logger.debug("St preliminar pruning: %s %s" % (st,
            kept_score))
        if kept_score > 90:
            df_rpl = df_rpl.append({'st_kept': st_kept, 'st_rpl':
                st}, ignore_index=True)
        else:
            terms_pruned.append(st)

    return (terms_pruned, df_rpl)

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    headers = {
        'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X
            10_11_5) AppleWebKit/537.36 (KHTML, like Gecko)
            Chrome/50.0.2661.102 Safari/537.36'}

    min_true_nn = min_true_nn if min_true_nn < 1 else 1

    assert len(terms) == len(terms_cosine) and len(
        terms_cosine) == len(terms_occ), "Different sizes
        in parameters"

```

```

# IMPORTANT target word is assumed to be in first
    position
root_term = terms[0]
terms = np.asarray(terms)
terms_occ = np.asarray(terms_occ).astype(int)
terms_cosine = np.asarray(terms_cosine).astype(float)

# terms_pruned sorted differently from terms
terms_pruned, df_rpl = self.get_st_couples(terms,
    terms_occ)

[...]

```

Once this preliminary pruning is done, the algorithm is ready to search the remaining terms in Wikidata, to find the candidate entities to disambiguate. **Since every Wikipedia page has a corresponding Wikidata entity (a corresponding QID)**, the search process can also be carried out through Wikipedia. This is handled using both Wikipedia and Wikidata web APIs, that return a json-encoded version of the same results we would have using the search feature provided in their online GUI. Please note that, for each term that we want to disambiguate, the algorithm needs to make two HTTP calls (one for Wikidata and another one for Wikipedia). In order to speed-up the whole process, these requests are made *asynchronously*, using the AIO-HTTP¹¹ python framework.

Listing 12: Methods *wikidata_search_entities*, *wikipedia_search_entities*, *fp_search_entities* and a piece of method *disambiguate* of NED class

```

WIKIDATA_ENDPOINT="https://www.wikidata.org/w/api.php"
WIKIPEDIA_ENDPOINT="https://en.wikipedia.org/w/api.php"

WIKIDATA_SEARCH_ENTITIES = {
    'action': 'query',
    'generator': 'search',
    'format': 'json',
    'prop' : 'cirrusbulddoc',
    'gsrlimit' : 10
}

WIKIPEDIA_SEARCH_ENTITIES = {
    'action': 'query',
    'generator': 'search',
    'format': 'json',
    'prop': 'pageprops|description|redirects',
    'gsrlimit' : 10
}

```

¹¹ <https://docs.aiohttp.org/en/stable/>


```

elif 'redirects' in response['query']['pages'][
    page_id]:
    df = df.append({'search_term': search_term, '
        entity_id': response['query']['pages'][
            page_id]['pageprops']['wikibase_item'], '
        wp_title': response['query']['pages'][page_id
            ]['title'], 'redirects': tuple(redirect['
            title'] for redirect in response['query']['
            pages'][page_id]['redirects']), 'dis_page':
            False}, ignore_index=True)
else:
    df = df.append({'search_term': search_term, '
        entity_id': response['query']['pages'][
            page_id]['pageprops']['wikibase_item'], '
        wp_title': response['query']['pages'][page_id
            ]['title'], 'redirects': np.nan, 'dis_page':
            False}, ignore_index=True)

return df

@backoff.on_exception(backoff.expo, aiohttp.ClientError
    , max_tries=5, max_time=100, giveup=fatal_code)
async def wikidata_search_entities(self, search_term):
    df = pd.DataFrame(columns=['search_term', 'entity_id'
        ])
    search_parameter = {
        'gsrsearch': search_term
    }

    async with aiohttp.ClientSession() as session:
        async with session.get(url=self.WIKIDATA_ENDPOINT,
            params={ self.WIKIDATA_SEARCH_ENTITIES,
                search_parameter}, raise_for_status=True) as r:
            response = await r.json()
            # Similar to wikipedia_search_entities
            [...]

def fp_search_entities(self, search_terms=[]):
    wd_group = asyncio.gather([self.
        wikidata_search_entities(term) for term in
        search_terms])
    wp_group = asyncio.gather([self.
        wikipedia_search_entities(term) for term in
        search_terms])
    all_groups = asyncio.gather(wd_group, wp_group)
    return self.loop.run_until_complete(all_groups)

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

#First API call

```

```

wikidata_entities_plus_disamb = self.
    fp_get_wikidata_entities(terms_pruned)
wd_dataframe = pd.concat(wikidata_entities_plus_disamb
    [0])
wp_dataframe = pd.concat(wikidata_entities_plus_disamb
    [1])

#Dataframes merged together
df = wd_dataframe.merge(wp_dataframe, on=['search_term
    ', 'entity_id'], how="outer")

[...]
```

The candidate entities found so far are stored in a *pandas Dataframe*. A *DataFrame* is a 2-dimensional labelled data structure with columns of potentially different types, that allows for faster and easier data manipulation. It is the de-facto standard for Data Scientists.

Table 10 shows an example output *Dataframe* retrieved after the first batch of HTTP calls, searching words *capital_iq* and *eikon*.

Table 10: Example of *pandas Dataframe* after first batch of HTTP calls.

search_term	entity_id	wp_title	redirects	dis_page
eikon	Q12727367	NaN	NaN	NaN
eikon	Q1303835	NaN	NaN	NaN
eikon	Q43763745	Eikon	NaN	False
eikon	Q5323200	EIKON Intern...	NaN	False
eikon	Q5349269	Eikon Basilike	(Icon Basilike,)	False
eikon	Q58628389	NaN	NaN	NaN
eikon	Q96377047	Eikon Exhibi...	NaN	False
eikon	Q96697163	NaN	NaN	NaN
eikon	Q99203699	NaN	NaN	NaN
...
eikon	Q60741469	Refinitiv	NaN	False
eikon	Q17078254	Reuters 3000 Xtra	NaN	False
capital_iq	Q2931538	CIQ	NaN	True
capital_iq	Q5157260	Compustat	(CompuStat,)	False
capital_iq	Q170277	Intelligence quo...	NaN	False
capital_iq	Q6066208	Ira Rennert	NaN	False
capital_iq	Q56283816	List of S&P 600 ...	NaN	False
capital_iq	Q4035851	S&P Capital IQ	NaN	False

Entities retrieved from method `wikipedia_search_entities` have additional information if compared to the ones retrieved by method `wikipedia_search_entities`. In particular:

1. `wp_title`: a Wikipedia page title
2. `redirects`: a tuple of alternative names for Wikipedia pages. They can be used as additional aliases for corresponding Wikidata entity.
3. a boolean flag called `dis_pages`: to highlight all the entities that are *Wikipedia disambiguation pages*

The column `search_term` refers to the input terms we are trying to disambiguate. Please note that `wikidata_search_entities` does not currently provide any title for entities found (it will be added later on). Moreover, **this method skips all disambiguation pages**, because they are already handled by `wikipedia_search_entities` that allows for in-depth research of those pages based on their inner text content.

4.4.2 Second Batch of HTTP requests

All the disambiguation pages found so far (`dis_page = True`) are searched back in Wikipedia, in order to retrieve their text body. They are an useful source for extra information and new candidate entities.

The algorithm parses the content of such pages and adds every new mentioned entity to the Dataframe, as a candidate entity for the input term the disambiguation page was returned for.

Moreover, these entities could have names or descriptions that are different from what can be found in Wikipedia and Wikidata. These can be used as additional information that we want to save for later computation (stored in columns `dis_aliases` and `dis_text`).

The title of the disambiguation page itself can often be used as an alias for all the entities included in the page. Most of the times it refers to an acronym (e.g. page <https://en.wikipedia.org/wiki/CIQ> - we want 'CIQ' to be an alias for 'Capital IQ'). However, there may be cases in which this title is too generic and using it would degrade the result of the whole NED process. This happens when the title of the page is similar to a substring of the title of the entity mentioned in the page text (e.g. page <https://en.wikipedia.org/wiki/Bloomberg> - we do **not** want 'Bloomberg' to be an alias for 'Bloomberg Terminal'). Thus, we can use another *edit distance* function provided by FuzzyWuzzy library, called *token_set_ratio*, that can compare also strings that are of widely differing lengths using the best partial match from a set of matching blocks. More details on this algorithm can be found in [10]. Consequently, the algorithm is **not** going to use the title of the disambiguation page if the *token_set_ratio*

is higher than 0.8. This threshold, of course, was empirically fixed.

The above computation is carried out by two methods:

1. `wikipedia_resolve_disambiguation_text` to parse the text of the Wikipedia Disambiguation pages by means of regular expressions
2. `wikipedia_resolve_disambiguation_entities` to retrieve the Wikidata identifiers of the entities mentioned in the Wikipedia Disambiguation pages and to perform the *edit distance* measurements as explained above.

Listing 13: Methods `wikipedia_resolve_disambiguation_text`, `sp_get_disambiguations` and a piece of method `disambiguate` of NED class

```
WIKIPEDIA_GET_DISAMB_ENTITIES = {
    'action': 'query',
    'generator': 'links',
    'format': 'json',
    'redirects': 1,
    'prop': 'pageprops',
    'gpllimit': 50,
    'ppprop': 'wikibase_item'
}

@backoff.on_exception(backoff.expo, aiohttp.ClientError,
    , max_tries=5, max_time=100, giveup=fatal_code)
async def wikipedia_resolve_disambiguation_entities(
    self, disamb_entity_id, title):
    df = pd.DataFrame(columns=['disamb_entity_id', '
        entity_id', 'wp_title', 'dis_aliases'])
    search_disambiguations = {
        'titles': title
    }

    title_clean = re.sub(' (disambiguation)', '', title)

    async with aiohttp.ClientSession() as session:
        async with session.get(url=self.WIKIPEDIA_ENDPOINT,
            params={ self.WIKIPEDIA_GET_DISAMB_ENTITIES,
                search_disambiguations}, raise_for_status=True) as
            r:
                response = await r.json()

        warning_keys = set(response).difference({'query', '
            batchcomplete'})
        if warning_keys:
            if 'error' in warning_keys:
```



```

logger.error("[%s] [%s]: %s" % (disamb_entity_id,
    title, {key:response[key] for key in
        warning_keys}))
else:
    logger.warning("[%s] [%s]: %s" % (disamb_entity_id
        , title, {key:response[key] for key in
            warning_keys}))

redirects = {}
red_aliases = {}

[...]

redirects[red['to']] = red['from']
red_aliases[red['to']] = [red['from'], red['
    tofragment']] if 'tofragment' in red else [red['
    from']]
# Disambiguation pages could link to other
disambiguation pages
red_aliases[red['to']] = list(filter(self.
    NON_DISAMBIGUATION.search, red_aliases[red['to'
    ]]))

[...]
page_title = response['query']['pages'][page_id]['
    title']
if page_title in red_aliases:
    alias, score = self.max_fuzz_ratio(title_clean,
        [red_aliases[page_title], page_title], [fuzz.
            token_set_ratio], filter_len = False)
else:
    alias, score = self.max_fuzz_ratio(title_clean, [
        page_title], [fuzz.token_set_ratio],
            filter_len = False)

if score <= 80:
    if page_title in red_aliases:
        red_aliases[page_title] = [red_aliases[
            page_title], title_clean]
    else:
        red_aliases[page_title] = [title_clean]

[...]

if 'pageprops' in response['query']['pages'][
    page_id]:
    df = df.append({'disamb_entity_id' :
        disamb_entity_id, 'entity_id':response['query'
        ]['pages'][page_id]['pageprops']['
        wikibase_item'], 'wp_title': redirects[
        page_title] if page_title in redirects else
        page_title, 'dis_aliases': tuple(red_aliases[

```

```

        response['query']['pages'][page_id]['title'])
        if response['query']['pages'][page_id]['title']
        ' in red_aliases else tuple()}, ignore_index=
        True)
    return df

def sp_get_disambiguations(self, dis_pages=[]):
    dis_text = asyncio.gather([self.
        wikipedia_resolve_disambiguation_text(dis[1]) for
        dis in dis_pages])
    dis_entities = asyncio.gather([self.
        wikipedia_resolve_disambiguation_entities(dis) for
        dis in dis_pages])
    all_groups = asyncio.gather(dis_text, dis_entities)
    return self.loop.run_until_complete(all_groups)

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):

    [...]

    extra_text = self.sp_get_disambiguations(dis_pages)
    # Add descriptions found in disambiguation pages
    df_disamb_text = pd.concat(extra_text[0])
    df_disamb_entities = pd.concat(extra_text[1])

    # Add new entities found in disambiguation pages
    # Disambiguation pages could be duplicated
    new_entities = df[df['dis_page'] == True][['
        search_term', 'entity_id']]
    new_entities.columns = ['search_term', '
        disamb_entity_id']
    new_entities = new_entities.merge(df_disamb_entities[['
        disamb_entity_id', 'entity_id']], on=['
        disamb_entity_id'], how="left")
    new_entities = new_entities.drop(columns=['
        disamb_entity_id'])
    full_entities = pd.concat([new_entities, df[['
        search_term', 'entity_id']]]).drop_duplicates()
    df = full_entities.merge(df.drop(columns=['search_term
        '])).drop_duplicates(), on=['entity_id'], how="left"
    )

    dsd_df = df_disamb_text.merge(df_disamb_entities.drop(
        columns=['disamb_entity_id']), on=['wp_title'], how
        ="inner").drop(columns=['wp_title']).
        drop_duplicates('entity_id', keep='last')
    df = df.merge(dsd_df, on=['entity_id'], how='left')

    [...]

```

4.4.3 Third Batch of HTTP requests

Now that we have a complete list of candidate entities for each input term, we want to retrieve as many details as possible from those entities. Searching them by identifier in Wikidata, we can retrieve the following information of interest:

1. The title of the Wikidata entity (`wp_title`)
2. The title of the corresponding Wikipedia page (`wp_page`), if available and if not already retrieved by method `wikipedia_search_entities`
3. A short text description (`description`)
4. A tuple containing all the aliases in addition to the ones already found by means of Wikipedia (`aliases`)
5. A tuple of all the Wikidata entities for which the current entity is *instance of* (`instance_entities`) (Wikidata P31 property). It can be seen as a set of *classes* it belongs to.
6. A boolean flag (`instance_of`) that is True if at least one of the `instance_entities` is in a manually selected set of **70 entities**. This set has been chosen from the most common entities/classes that are instance of the test set used for word embedding evaluation. More about this can be found in the section 4.4.5.
7. A tuple with the official websites of the Wikidata entity (`websites`) (Wikidata P856 property)

The method that handles this retrieval is called `wikidata_fill_prune_entities`. It is mandatory to specify that it does not return any of the Disambiguation pages managed in 4.4.2, since we already took everything needed from them. Since the Dataframe returned by this method will be *inner merged* with the Dataframe containing all the candidate entities, those disambiguation pages will be completely discarded.

At this stage, we have all the information to compute the edit distance that we are going to use to pick the best combination of entities. For every candidate entity we compute this value as the **maximum edit distance** between the `search_term` and one of the following values: `wp_title`, `wd_title`, `redirects`, `aliases` and `dis_aliases`.

This measurement is carried out using FuzzyWuzzy's *WRatio* function, that weights results from other FuzzyWuzzy's functions¹². After empirical comparisons, it turned out *WRatio* returns a good score

¹² <https://github.com/seatgeek/fuzzywuzzy/blob/master/fuzzywuzzy/fuzz.py#L222-L257>

even if the searched term has some adjectives or extra words (eg. `WRatio('frances cac', 'CAC 40') = 0.86`). However, a more permissive function like this could return high scores when a term is measured against a very long string, that has that term inside it. To avoid such cases, the algorithm skips all the strings that are longer than 1.5 times the length of the input term. Please note, every string is lowercased before computing the `WRatio`.

The outcome of the `WRatio` computation is a percentage value stored in a new column of the Dataframe called `fuzz_ratio`. **The candidate entities with a `fuzz_ratio` lower than 0.8 are immediately discarded.** This threshold, called `min_lev`, was empirically fixed after many trials.

	search_term	entity_id	wd_title	...	fuzz_ratio
0	capital_iq	Q868587	S&P Global	...	1.00
3	eikon	Q43763745	Eikon	...	1.00
13	capital_iq	Q4035851	S&P Capital IQ	...	1.00
26	capital_iq	Q5973340	IQ reference chart	...	0.86
27	capital_iq	Q170277	intelligence quotient	...	0.90

Table 11: Candidate entities Dataframe after pruning on `fuzz_ratio`.

Listing 14: Methods `wikipedia_fill_prune`, `tp_fill_prune_entities`, `max_fuzz_ratio` and a piece of method `disambiguate` of NED class

```

WIKIDATA_ENDPOINT = "https://www.wikidata.org/w/api.php"
"

WIKIDATA_GET_PARAMETERS = {
    'action': 'wbgetentities',
    'props': 'claims|sitelinks|labels|aliases|descriptions',
    'format': 'json'
}

@backoff.on_exception(backoff.expo, aiohttp.ClientError,
    max_tries=5, max_time=100, giveup=fatal_code)
async def wikidata_fill_prune_entities(self,
    entity_list):
    #similar to wikidata_search_entities
    [...]
    async with session.get(url=self.WIKIDATA_ENDPOINT,
        params={ self.WIKIDATA_GET_PARAMETERS,
            search_parameter}, raise_for_status=True) as r:
        [...]

def tp_fill_prune_entities(self, entity_list=[]):
    return self.loop.run_until_complete(asyncio.gather([
        self.wikidata_fill_prune_entities(entity_list[i]

```

```

        self.WIKI_MAX_LIMIT:(i + 1) self.WIKI_MAX_LIMIT]])
        for i in range((len(entity_list) + self.
WIKI_MAX_LIMIT - 1) // self.WIKI_MAX_LIMIT ]))

def max_fuzz_ratio(self, search_term, arr_like,
    scorer_array = [fuzz.WRatio], filter_len = True):
    if len(arr_like) == 0:
        return (np.nan, 0)

    search_term = re.sub('_', ' ', search_term)
    assert search_term != np.nan
    mst_len = 1.5 len(search_term)

    labels = np.hstack(arr_like).astype('str')
    labels = labels[labels != 'nan']

    if labels.size == 0:
        return (np.nan, 0)

    if filter_len:
        filter_labels = np.array([True if len(x) < mst_len
            else False for x in labels])
        labels = labels[filter_labels]

    max_match = tuple()

    for scorer in scorer_array:
        sc = process.extractOne(search_term, np.char.lower(
            labels), scorer = scorer)
        if sc:
            max_match = max_match + sc
        else:
            max_match = max_match + (np.nan, 0)

    return max_match

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

    # Third API call
    wb_fill = self.tp_fill_prune_entities(
        candidate_entities)
    df = df.drop(columns=['wp_title'])
    df = df.merge(pd.concat(wb_fill), on=['entity_id'],
        how='inner')

    # Pruning of rows with no names
    df = df.loc[df[['search_term', 'wp_title', 'wd_title',
        'redirects', 'aliases', 'dis_aliases']].dropna(
        thresh=2).index].reset_index(drop=True)

```

```

# max_fuzz_ratio call
df_lev = pd.DataFrame([self.max_fuzz_ratio(tuple[1],
    tuple[2:], [fuzz.WRatio]) for tuple in df[['
    search_term', 'wp_title', 'wd_title', 'redirects',
    'aliases', 'dis_aliases']].itertuples()), columns=[
    'fuzz_best_match', 'fuzz_ratio'])

df_lev['fuzz_ratio'] = df_lev['fuzz_ratio'] / 100
df = pd.concat([df, df_lev], axis=1)
df = df.drop(columns=['redirects', 'aliases', '
    dis_aliases'])

# Pruning of Wratio too low
df = df[df['fuzz_ratio'] > self.min_lev]

[...]
```

4.4.4 Fourth Batch of HTTP requests

The only descriptions available so far for the candidate entities are the Wikidata description and the text retrieved from Disambiguation pages (if available). In this batch of HTTP calls, the algorithm aims to find additional information from:

1. The introduction of the english Wikipedia page linked to the Wikidata entity
2. The *title* and *meta-description* tags in the HTML source code of the official websites of the entity (property P856)

These descriptions are important, because they will be used to feed a *document-embedding* machine learning model, to retrieve a vector-based representation of these entities. More about this is presented in the next subsection.

As already said before, the NLP models used in this project only deal with English-based text. However, the websites retrieved by the Wikidata P856 property could be in other languages. To work around this issue we only preserve websites having meta-description with *alphanumeric characters* that fall into the ASCII standard and remaining characters that fall into the following Unicode classes:

1. Punctuation (P)
2. Separator (Z)
3. Other (C)

4. Mark (M)

5. Symbol (S)

Even if this approach is not very strong, it is very fast and it turned out to be working in most of the cases, without the need for any additional machine learning model for language recognition.

Listing 15: Methods *wikipedia_get_extracts*, *get_meta_titles* and a piece of method *disambiguate* of NED class

```

WIKIDATA_ENDPOINT="https://www.wikidata.org/w/api.php"

[...]

WIKIPEDIA_GET_INTRO = {
    'action': 'query',
    'prop': 'extracts',
    'exintro': 1,
    'explaintext': 1,
    'redirects': 1,
    'format': 'json'
}

[...]

META_TITLE = re.compile(r'(?<=<title[^\>]>)([^\<]+)(?=</title>)', re.UNICODE)
META_DESCRIPTION = re.compile(r'(?<=<meta\s[^\>](property|name)\s?=\s?"([A-Za-z]+:)?(description|DESCRIPTION))"\s[^\>] content\s?=\s?"([^\>"]+)(?="^\>"/?>)', re.UNICODE)

@backoff.on_exception(backoff.expo, aiohttp.ClientError,
    max_tries=5, max_time=100, giveup=fatal_code)
async def wikipedia_get_extracts(self, df_input):
    df = pd.DataFrame(columns=['wp_title', 'text'])
    titles_list = '|'.join(df_input['wp_title'].to_list())

    search_parameter = {
        'titles': titles_list
    }

    async with aiohttp.ClientSession() as session:
        async with session.get(url=self.WIKIPEDIA_ENDPOINT,
            params={ self.WIKIPEDIA_GET_INTRO,
                search_parameter}, raise_for_status=True) as r:
            [...]

    async def get_meta_title(self, entity_id, url, headers
        ={}):
        df = pd.DataFrame(columns=['entity_id', 'meta', 'rank'
            ])
        table = str.maketrans("\n\t\r", " ")

```

```

try:
    async with aiohttp.ClientSession() as session:
        async with session.get(url, headers=headers, timeout=
            =5, allow_redirects=True) as r:
            response = await r.text()
            meta_list = []
            rank = 0

            meta_description = self.META_DESCRIPTION.search(
                response)
            if meta_description:
                meta_description_lower = meta_description.group(0)
                    .lower()
                if 'robot' not in meta_description_lower and '
                    captcha' not in meta_description_lower:
                    meta_des_ascii = self.is_ascii(url,
                        meta_description.group(0))
                    if meta_des_ascii:
                        meta_list.append(html.unescape(meta_des_ascii).
                            translate(table))
                        rank = 1

            meta_title = self.META_TITLE.search(response)
            if meta_title:
                meta_title_lower = meta_title.group(0).lower()
                if 'robot' not in meta_title_lower and 'captcha'
                    not in meta_title_lower:
                    meta_title_ascii = self.is_ascii(url, meta_title.
                        group(0))
                    if meta_title_ascii:
                        meta_list.append(html.unescape(meta_title_ascii)
                            .translate(table))

            meta = ' '.join(meta_list)
            if meta:
                df = df.append({'entity_id' : entity_id, 'meta' :
                    meta, 'rank' : rank}, ignore_index=True)

except Exception as e:
    logger.warning("Unable to scrape meta title for: %s
        [%s]" % (url, e))

return df

def fp_get_wiki_meta(self, entity_wiki_extracts,
    entity_web_pages, headers={}):
    get_extracts = asyncio.gather([self.
        wikipedia_get_extracts(entity_wiki_extracts[i
            self.WIKI_MAX_LIMIT:(i + 1) self.WIKI_MAX_LIMIT])

```



```

        for i in range((len(entity_wiki_extracts) + self.
            WIKI_MAX_LIMIT - 1) // self.WIKI_MAX_LIMIT ))
    get_meta = asyncio.gather([self.get_meta_title(row['
        entity_id'], url, headers) for index, row in
        entity_web_pages.iterrows() for url in row['
        websites']])
    all_groups = asyncio.gather(get_extracts, get_meta)
    return self.loop.run_until_complete(all_groups)

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

    # Merge of Wikipedia extracts
    wiki_web_meta = self.fp_get_wiki_meta(
        entity_wiki_extracts, entity_web_pages, headers)
    df = df.merge(pd.concat(wiki_web_meta[0]), on=['
        entity_id'], how='left')

    # Take meta-descriptions/titles from websites of
    entity only once
    meta_df = pd.concat(wiki_web_meta[1])
    meta_df = meta_df.sort_values(by=['rank'])
    meta_df = meta_df.drop_duplicates('entity_id', keep='
        last')
    df = df.merge(meta_df, on=['entity_id'], how='left')
    df = df.drop(columns=['dis_page', 'rank'])

    [...]

```

4.4.5 Selection of the best combination of entities

As already explained in 4.3, assuming to have the candidate entities grouped by input terms (*search_term*), the main idea of this algorithm is roughly to turn such entities into embeddings and find the best combination using formula 4.2.

Depending on the number of input terms and candidate entities found, the evaluation of all the possible combinations could be computationally infeasible.

Therefore, we may think to split the list of input terms into smaller chunks. Such that the algorithm can evaluate the combinations of a single chunk and when the final entities are chosen, they may be concatenated to the combinations of the next chunk, to preserve the meaning of the context. Let's call this mechanism *context sharing*. The chunk dimension is set using variable *chunk_size*.

However, splitting into chunks may lead the algorithm to undesired behaviours we have to deal with. For example, suppose that we fixed a `chunk_size` equal to **two** and that the word embedding model returned a list of financial indexes. For some reason, the splitting process returns a chunk made of the following words: *spanish ibex* and *ibex*. Both of them refer to the official index IBEX-35 of the Spanish Continuous Exchange. Let's assume the candidate entities for both of these terms are:

1. Spanish ibex www.wikidata.org/wiki/Q549108
2. IBEX-35 www.wikidata.org/wiki/Q938032

The latter is the index itself while the former is a mammal and, unfortunately, it gets a 100% score when computing the WRatio against the input term *spanish ibex*. This means that among the four possible combinations displayed in figure 12, the algorithm chooses the one with entity *Spanish ibex* (Q549108) for both the input terms, while we would expect to get *IBEX-35* (Q938032) instead (see image 12).

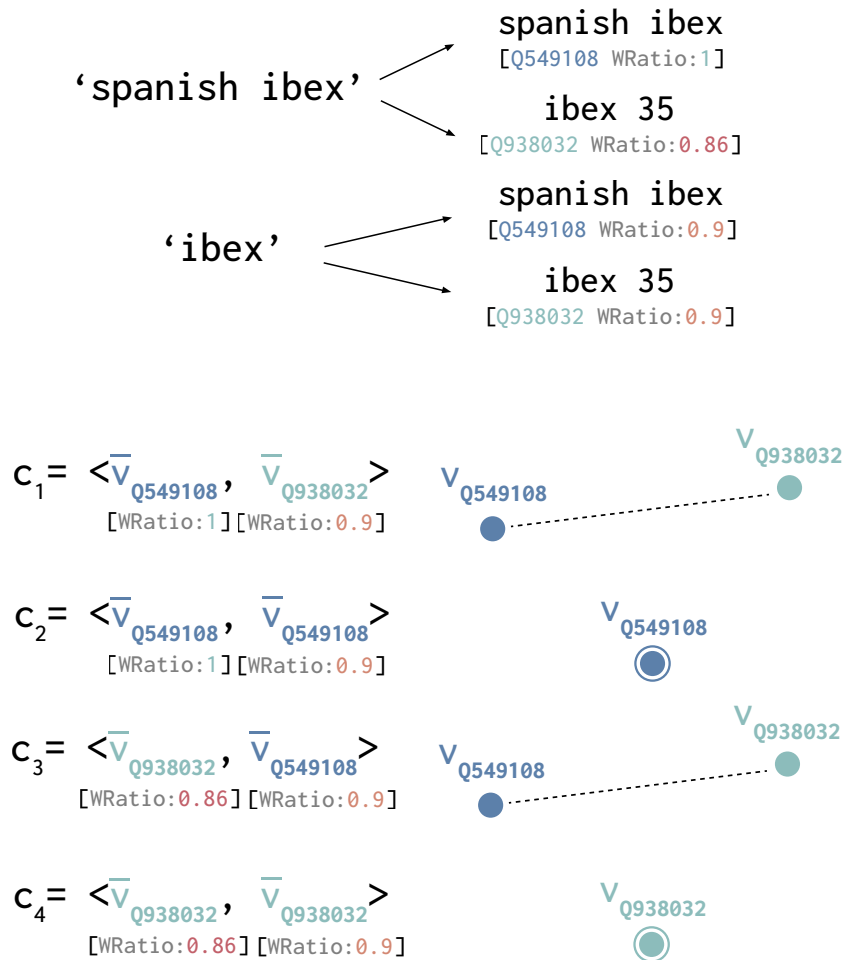


Figure 12: Undesired behaviour of the splitting process

Most of the times, the *context sharing* mechanism already prevent this issue to occur. However, it may still happen in the first chunk for which there is no previous context to use. In general, this problem can be solved by ensuring terms, with similar candidate entities, to be evaluated in different chunks.

This can be achieved computing the pairwise *Jaccard index* [16] on the set of candidate entities of each input term. Assuming to have two non-empty sets A and B, the Jaccard index is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \text{ with } 0 \leq J(A, B) \leq 1. \quad (4.4)$$

With the formula above, we can build a matrix of Jaccard indexes, having the input terms on both columns and rows, like in table 12. For each row, we group together all the input terms with a Jaccard index greater than zero. **We want the entities being in the same group not to be in the same chunk.**

	eikon	capiq	capital_iq	ciq	datastream
eikon	1.0	0.00	0.00	0.00	0.0
capiq	0.0	1.00	0.25	0.25	0.0
capital_iq	0.0	0.25	1.00	0.33	0.0
ciq	0.0	0.25	0.33	1.00	0.0
datastream	0.0	0.00	0.00	0.00	1.0

Table 12: Example of matrix with pairwise jaccard index on input terms.

Of course, it may happen to have both duplicate groups to be removed and different groups with **non-empty** intersection. The latter case can be handled merging those groups into a single one. This can be related to the problem of identifying the *connected components* within a graph, i.e. the subgraphs in which there is a path between any two vertices, that are not connected to any other vertices in the supergraph. *Scipy* library offers a convenient method for connected components identification.

Once the final groups are identified, the algorithm split the input terms into chunks using a *round-robin-based* mechanism, looping between groups and going back to the first one every time a chunk is full.

Listing 16: A piece of method *group_by_search_term* and a piece of method *disambiguate* of NED class

```
def group_by_search_term(self, df, np_entity_ids,
    root_term):
    if df.empty:
```

```

    return df
# Sort by search_term and instance_of = true first
# Targer word must be placed on top
true_search_terms = np.unique(df[df['instance_of'] ==
    True]['search_term'])
df = df.sort_values(by=['search_term'])
df = pd.concat([df[df['search_term'].isin(
    true_search_terms)], df[~df['search_term'].isin(
    true_search_terms)]]
df = pd.concat([df[df['search_term'] == root_term], df
    [df['search_term'] != root_term]])
df.reset_index(drop=True, inplace=True)
df.drop(columns=['instance_of'], inplace=True)

keys, values = df.values.T
_, index = np.unique(keys, True)
index = np.sort(index)
ukeys = keys[index]
arrays = np.split(values, index[1:])
df2 = pd.DataFrame(data = [np.logical_or.reduce(
    np_entity_ids == a.reshape((-1,1))).astype(int) for
    a in arrays], columns = np_entity_ids, index =
    ukeys)
return df2

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

# IMPORTANTE: Reset index before Google USE
np_entity_ids = np.unique(df['entity_id'])
df_pre_jaccard = self.group_by_search_term(df[['
    search_term', 'entity_id', 'instance_of']],
    np_entity_ids, root_term)

df_jaccard = distance.pdist(df_pre_jaccard, 'jaccard')
df_jaccard = 1 - pd.DataFrame(squareform(df_jaccard),
    index=df_pre_jaccard.index, columns= df_pre_jaccard
    .index)

len_st_grp, st_label = connected_components(csgraph=
    csr_matrix(np.where(df_jaccard > 0, 1, 0)),
    directed=False, return_labels=True)

# Take search_terms ordered as in df_jaccard
st_jaccard = df_jaccard.index
df_cc = pd.DataFrame(data = { 'cc_label' : st_label, '
    st_grp' : st_jaccard})
df_cc = self.group_by(df_cc, tuple_flag = False)
st_grp = df_cc['st_grp'].to_list()

```

```

st_rr_flatten = []

# Round robin that restarts when a chunk is full
i = 0
lm_mod = chunk_size

while i < len(st_jaccard) :

    if lm_mod > len_st_grp:
        lm_mod = len_st_grp

    im = i % lm_mod

    if st_grp[im] != []:
        st_rr_flatten.append(st_grp[im].pop(0))
        i += 1
    else:
        st_grp.pop(im)
        len_st_grp -= 1

st_chunk = [st_rr_flatten[i : chunk_size:(i + 1)
                    chunk_size] for i in range((len(st_rr_flatten) +
                    chunk_size - 1) // chunk_size )

[...]
```

It is now finally time to turn entities into embeddings. This conversion can be achieved computing *document embeddings* over a concatenation of the descriptions got so far for each entity:

1. Wikidata description
2. Description from Wikipedia disambiguation pages
3. Wikipedia page introduction
4. Title and meta-description of the official websites

In recent years, many powerful NLP models were released. We chose to carry out this conversion by means of the *Google Universal Sentence Encoder (USE)* [9] model. This model is capable of turning any variable length English text into a 512 dimensional vector. If compared to other models like BERT, ELMo, GPT-3 and XLNet, USE outperforms them in semantic textual similarity (STS), since it was built with this task in mind [15].

Google Tensorflow Hub provides several pre-trained USE models ready for inference. In particular, this project uses the *universal-sentence-encoder-lite v2*¹³, it is only 25MB and allows for smaller RAM

¹³ <https://tfhub.dev/google/universal-sentence-encoder-lite/2>

requirements. Although it returns slightly worse results if compared to the standard version (915MB) [4][5], empirical trials have shown no real difference for the kind of use made by this algorithm.

Listing 17: A piece of method `_init_` and a piece of method `disambiguate` of NED class

```
def __init__(self, event_loop = None, dim_red = False):
    [...]

    # Create graph and finalize (finalizing optional but
    # recommended).
    tf.disable_v2_behavior()
    g = tf.Graph()
    with g.as_default():
        use_module = hub.Module("https://tfhub.dev/google/
            universal-sentence-encoder-lite/2")
        self.input_placeholder = tf.sparse_placeholder(tf.
            int64, shape=[None, None])
        self.encodings = use_module(
            inputs=dict(
                values=self.input_placeholder.values,
                indices=self.input_placeholder.indices,
                dense_shape=self.input_placeholder.dense_shape))
        init_op = tf.group([tf.global_variables_initializer()
            , tf.tables_initializer()])

    # Create session and initialize.
    self.session = tf.Session(graph=g)
    self.session.run(init_op)
    spm_path = self.session.run(use_module(signature="
        spm_path"))
    self.sp = spm.SentencePieceProcessor()
    self.sp.Load(spm_path)
    g.finalize()

    [...]

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

    # Generate document-embeddings with Google USE
    # Duplicate candidate entities are computed only once
    phrases = df.drop_duplicates('entity_id', keep='last')
        [['dis_text', 'description', 'text', 'meta']].
        fillna('').agg(' '.join, axis=1).to_list()
    values, indices, dense_shape = self.
        process_to_IDs_in_sparse_format(phrases)

    np_embeddings = pd.DataFrame(self.session.run(
        self.encodings,
        feed_dict={self.input_placeholder.values: values,
```

```

        self.input_placeholder.indices: indices,
        self.input_placeholder.dense_shape: dense_shape
    )))

map_un_eid = df[['entity_id']].drop_duplicates('
    entity_id', keep='first')

map_un_eid_idx = map_un_eid.index
map_un_eid = map_un_eid.to_numpy()

np_embeddings['entity_id'] = map_un_eid
np_embeddings = df[['entity_id']].merge(np_embeddings,
    on=['entity_id'], how='left')
np_embeddings = np_embeddings.drop(columns=['entity_id
    ']).to_numpy()
df['overview'] = df['text'].fillna(df['meta']).fillna(
    df['dis_text']).fillna(df['description'])
df = df.drop(columns=['dis_text', 'description', 'text
    ', 'meta'])

[...]
```

Having both the USE embeddings and the (maximum) FuzzyWuzzy WRatio score for the candidate entities, the algorithm is ready to compute combinations and apply formula 4.2 to select the best one for each chunk. Let's call the entities in these selected combinations *matched entities*. Relevant piece of code is provided below:

Listing 18: A piece of method *disambiguate* of NED class

```

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

    # Context sharing
    gb = df[['search_term', 'entity_id']].groupby('
        search_term')

    chunk_final_indexes = []
    final_indexes = set()

    for chunk in st_chunk:
        # Compute combinations for every chunk
        gb_chunk = [gb.get_group(label).index for label in
            chunk]
        np_cmb_indexes = np.array(np.meshgrid( gb_chunk)).T.
            reshape(-1, len(chunk))

        if chunk_final_indexes != []:
            np_cmb_indexes = np.concatenate((np_cmb_indexes, np.
                tile(chunk_final_indexes, (len(np_cmb_indexes),1)
                )), axis=1)
```

```

# Auxiliary array to compute the argmax
np_cmb_dist = np.array([distance.cdist(np_embeddings[
    cmb], [np_embeddings[cmb].mean(axis=0)], 'cosine')
    for cmb in np_cmb_indexes])
np_cmb_dist = np.reshape(np_cmb_dist, (np_cmb_indexes
    .shape[0], np_cmb_indexes.shape[1]))
np_cmb_dist = 1 - np_cmb_dist

# Array with maximum levenshtein ratio for every
entity
np_cmb_lev = np.array([df.loc[cmb]['fuzz_ratio'] for
    cmb in np_cmb_indexes])

np_final_score = np.sum(np_cmb_lev * np_cmb_dist,
    axis=1)
chunk_final_indexes = np_cmb_indexes[np.argmax(
    np_final_score)]

final_indexes.update(chunk_final_indexes)

[...]

```

Once the matched entities are finally chosen, the algorithm performs a final pruning over the entities that do not fall into a fixed set of values for the Wikidata property *instance of* (P31). The value of this property can be seen as a set of *classes* the entity belongs to and can be used to filter undesired results. Please note that every entity of Wikidata can be used as a value for this property, therefore we could have millions of potential classes to choose from.

First, a fixed set of 70 classes was selected (WIKIDATA_CLASSES), chosen from the most commons classes appeared in the test set defined in section 3.3. For the reason just explained, this cannot be a complete set. The allowed classes are displayed below:

- | | |
|--------------------------------|---------------------------------|
| 1. Q30242023: money amount | 8. Q223371: stock market index |
| 2. Q179179: interest rate | 9. Q730038: credit institution |
| 3. Q4830453: business | 10. Q22687: bank |
| 4. Q6881511: enterprise | 11. Q73712047: JP Morgan branch |
| 5. Q4201895: investment fund | 12. Q11691: stock exchange |
| 6. Q1738991: index number | 13. Q179076: exchange |
| 7. Q1284784: Bond market index | 14. Q37654: market |

- | | |
|--|--|
| 15. Q159810: economy | 38. Q166142: application |
| 16. Q1331793: media enterprise | 39. Q778043: electronic trading platform |
| 17. Q43371537: trading venue | 40. Q361390: Interbank foreign exchange market |
| 18. Q750458: capital market | 41. Q2659777: Interbank lending market |
| 19. Q208697: financial market | 42. Q650241: financial institution |
| 20. Q845477: Exchange-traded fund | 43. Q1194970: dot-com company |
| 21. Q2620430: financial endowment | 44. Q1386901: overnight rate |
| 22. Q1167393: economic indicator | 45. Q1058914: software company |
| 23. Q205180: interest rate derivative | 46. Q194166: consortium |
| 24. Q1166072: financial transaction | 47. Q1153191: online newspaper |
| 25. Q247506: financial instrument | 48. Q17232649: news website |
| 26. Q15809678: financial product | 49. Q1110794: daily newspaper |
| 27. Q837171: financial services | 50. Q167037: corporation |
| 28. Q169489: security | 51. Q72182881: venture capital firm |
| 29. Q783794: company | 52. Q5001853: business channel |
| 30. Q192283: news agency | 53. Q218616: proprietary software |
| 31. Q21980538: commercial organization | 54. Q438711: alternative trading system |
| 32. Q7397: software | 55. Q5449703: Financial data vendor |
| 33. Q2429814: software system | 56. Q10836209: digital currency |
| 34. Q765517: credit rating agency | 57. Q11032: newspaper |
| 35. Q891723: public company | 58. Q1143635: business school |
| 36. Q35127: website | 59. Q13479982: cryptocurrency |
| 37. Q186165: web portal | 60. Q155271: think tank |

- | | |
|--------------------------------------|---|
| 61. Q1589009: privately held company | 66. Q5418962: private equity firm |
| 62. Q18388277: technology company | 67. Q41298: magazine |
| 63. Q20514253: blockchain | 68. Q8513: database |
| 64. Q2073644: brokerage firm | 69. Q536390: self-regulatory organization |
| 65. Q219577: holding company | 70. Q43229: organization |

Then, the algorithm uses a nearest neighbouring approach trained over the space of **the embeddings of all the candidate entities**. Every matched entity with property P₃₁ outside `WIKIDATA_CLASSES` is preserved only if enough of their nearest neighbours have their value within this set.

Due to the high dimensionality of the embeddings generated with the Universal Sentence Encoder, the algorithm uses an *Approximated Nearest Neighbouring* approach for fast classification of those entities, implemented using the *Spotify Annoy* library¹⁴.

As every KNN classifier, it is mandatory to pick a value for parameter K. However, the number of embeddings used for training depends on the candidates found and indeed it is not fixed. For this reason, instead of trying multiple values for K with a fixed minimum threshold of 50%, K was empirically fixed to 9 (not too small odd number) and the minimum required number of neighbours within `WIKIDATA_CLASSES` was let variate instead. This minimum threshold is a hyperparameter to take care of while computing accuracy measurements in 4.5, let's call it `min_nn`. Of course, since we want the number of good surrounding entities to be greater than the bad ones, the threshold values to test will be in $\{\frac{6}{9}, \frac{7}{9}, \frac{8}{9}, \frac{9}{9}\} = \{0.65, 0.75, 0.85, 1\}$.

Listing 19: A piece of method *disambiguate* of *NED* class

```
def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

    f = 512
    t = AnnoyIndex(f, 'angular') # Length of item vector
                                that will be indexed

    for i in range(len(map_un_eid)):
        # It will allocate memory for max(i)+1 items.
        t.add_item(i, np_embeddings[map_un_eid_idx[i]])
```

¹⁴ <https://github.com/spotify/annoy>

```

t.build(10) # 10 trees

false_instance_of = df.loc[final_indexes_list][df['
    instance_of'] == False]['entity_id']
for falsy_eid in false_instance_of:
    falsy_un_idx, = np.where(map_un_eid == falsy_eid)
    near_neigh = t.get_nns_by_item(falsy_un_idx, 10,
        search_k=-1, include_distances=False)
    near_neigh.pop(near_neigh.index(falsy_un_idx))

    near_neigh_eid = map_un_eid[near_neigh]

    sr_nn = df[df['entity_id'].isin(near_neigh_eid)][['
        entity_id', 'instance_of']].drop_duplicates('
        entity_id', keep='last')['instance_of'].
        value_counts(normalize=True)
    sr_nn.index = sr_nn.index.map(str)

    if 'True' in sr_nn.index and sr_nn.loc['True'] >=
        min_true_nn:
        df.loc[df['entity_id'] == falsy_eid, 'instance_of']
            = True
        logger.debug('Instance of %s converted to True' %
            falsy_eid)

df = df.loc[final_indexes_list]

# Final pruning of entities outside WIKIDATA_CLASSES
df = df[df['instance_of'] == True]

[...]
```

Either the unresolved terms and the misspelled ones removed in `get_st_couples` are now bonded to the closest matching term by the algorithm. This is done performing a pairwise levenshtein ratio between these terms, using ratio threshold equal to 0.75, which is lower than 0.8 used in `get_st_couples`. The output Dataframe is indeed grouped by the QID of the matched entities, as shown in table 13.

Listing 20: A piece of method *disambiguate* of NED class

```

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

    # Try to solve misspelled terms using matched terms
    terms_found = df['search_term'].to_numpy()
    logger.debug("Tutti i search terms (unfiltered) sono :
        %s" % terms)
    missing_search_terms = np.setdiff1d(terms, terms_found
        )
```

entity_id	search_term	wd_title	...	strong_sim
Q43763745	(eikon, reuters_eikon, ...	Eikon	...	1.00
Q2349604	(bloomberg_terminal,)	Bloomberg Ter...	...	0.77
Q1885491	(factset,)	FactSet	...	0.76
Q1268489	(metastock,)	MetaStock	...	0.76
Q98496776	(datastream,)	Datastream	...	0.74
Q86453325	(intrinio,)	Intrinio	...	0.70
Q4035851	(capitaliq, cap_iq, ...	S&P Capital IQ	...	0.69

Table 13: Example output Dataframe

```

missing_df = pd.DataFrame([(mst, ) + self.
    max_fuzz_ratio(mst, terms_found, [fuzz.ratio]) for
    mst in missing_search_terms], columns=['
    missing_search_term', 'search_term', 'mlev_ratio'])

# IMPORTANT We use a lower threshold here than
get_st_couples()
missing_df = missing_df[missing_df['mlev_ratio'] >
    75].drop(columns=['mlev_ratio'])
missing_df['search_term']=missing_df['search_term'].
    astype(str)
logger.debug("Dataframe terms that can be aggregated
    against resolved one")
logger.debug(missing_df)
missing_df = missing_df.merge(df, on=['search_term'],
    how='left').drop(columns=['search_term'])
missing_df.rename(columns={'missing_search_term': '
    search_term'}, inplace=True)
df = pd.concat([df, missing_df])

[...]

has_root_term = False
root_eid = ''

if root_term in df.search_term.values:
    has_root_term = True
    root_eid = df[df['search_term'] == root_term][['
        entity_id']].iat[0,0]
    logger.debug("Root eid %s" % root_eid)

# Group by entity id
df_group = self.group_by(df[['entity_id', '
    search_term']])
df = df.drop(columns=['search_term']).drop_duplicates(
    'entity_id', keep='last')
df = df_group.merge(df, on=['entity_id'], how='left')

```

```

# Among all the terms matched against an entity, we
# provide the highest word embedding similarity
bst_st_cosine = [self.bst_util(terms, terms_cosine,
    terms_occ, tuple_st[1]) for tuple_st in df[['
    search_term']].itertuples())]
bst_st, bst_cosine = zip(*bst_st_cosine)
df['bst_st'] = bst_st
df['wv_similarity'] = bst_cosine

[...]

```

Last, the USE embeddings can be used to provide a **similarity score** for each matched entity. This can be computed as the cosine similarity between the embedding of each input term against the embedding of the target word. Let's call it *STRONG Similarity* as opposed to the (WEAK) similarity provided by the word embedding model. It is much more meaningful since it is computed over in-depth textual descriptions. In section 4.5.1 it is shown how the accuracy of the system increases discarding all matched entities with *STRONG Similarity* under a certain threshold.

Listing 21: A piece of method *disambiguate* of NED class

```

def disambiguate(self, terms, terms_cosine = [],
    terms_occ = [], chunk_size = 3, min_true_nn = 1):
    [...]

    if has_root_term:
        root_index = map_un_eid_idx[np.where(map_un_eid ==
            root_eid)][0]
        map_ordering_df = pd.DataFrame(data = {'entity_id' :
            map_un_eid})
        df = map_ordering_df.merge(df, on = ['entity_id'],
            how = 'inner')
        reids = df['entity_id'].to_numpy() # resolved eids
        rmask = np.in1d(map_un_eid, reids)
        np_sim = distance.cdist(np_embeddings[map_un_eid_idx[
            rmask]], [np_embeddings[root_index]], 'cosine')
        np_sim = np.reshape(np_sim, (np_sim.shape[0], np_sim.
            shape[1]))
        np_sim = 1 - np_sim
        df['use_similarity'] = np_sim

    # Sort by USE similarity
    df = df.sort_values(by='use_similarity', ascending=
        False, ignore_index=False)

    [...]

```

4.4.6 Handling of unresolved and discarded terms

The algorithm described so far only returns to the user a Dataframe containing matched entities. However, it takes also care of input terms that, for some reason, have not been matched. Imagine for example that the word embedding model returns the name of a pertinent financial product that is not registered in Wikidata. The NED will not be able to resolve that term, however, we still want to display it to the final user since it represents useful information that can be easily solved by means of a quick Google search.

Having said this, we want to distinguish between *unresolved* and *discarded* terms. The formers refer to useful input terms that could not be solved due to missing entities in Wikidata. The latters refer to input terms that should correctly be filtered out since they do not represent any useful information.

Let's define the unresolved terms as:

1. Not-matched input terms with no candidate entities at all
2. Not-matched input terms with candidate entities pruned due to WRatio lower than 0.8 (both in and out `WIKIDATA_CLASSES`)

Please note that the unresolved terms do not contain the not matched input terms having candidate entities with WRatio higher than 0.8 but outside `WIKIDATA_CLASSES`. We cannot, indeed, differentiate between the case in which the input term exists in Wikidata but is completely unrelated to the financial world, from the case in which the term is pertinent but Wikidata contains only entities with the same name but that refer to completely different things.

Finally, let's define the discarded terms as the not matched input terms that are not unresolved.

4.5 NED RESULTS EVALUATION

In this section, we are mainly focused on:

1. Measuring the accuracy of the NED algorithm against the manually evaluated test set.
2. Measuring the *overall accuracy*, that is the accuracy of the word embedding model, removing the results filtered out by the NED algorithm

4.5.1 Accuracy of NED against the manually evaluated test set.

To compare the results of the NED algorithm against the results of the manual evaluation of the test set, we first need to clarify that the NED algorithm will never be able to resolve a term, if the corresponding entity is missing from the Knowledge Base. To evaluate the accuracy of the algorithm, all the terms that are unresolvable due to lack of entities in Wikidata must be removed. **Therefore, the number of samples in the test set, defined in 3.3, drops from 966 to 590 elements.**

Moreover, the NED algorithm performs different choices than the manual evaluation carried out by business experts. In particular, we can imagine splitting the results of the word embedding model into two categories:

1. Names or Brands of products
2. Generic words: words that do not include the proper name of a product or vendor.

In the tables below we present the different choices made by either the manual evaluation and the NED algorithm when evaluating these two classes of results:

Table 14: Manual evaluation and NED algorithm behaviours

Manual evaluation	Name of products	Generic words
Finance related	GOOD	GOOD
Finance unrelated	BAD	BAD
NED Algorithm	Name of products	Generic words
Finance related	GOOD	BAD
Finance unrelated	BAD	BAD

In order to make a fair comparison between these two systems, all the financial generic words that were evaluated as *GOOD* by the manual evaluation, need to be labeled as *BAD* for this accuracy measurement.

The NED Algorithm is capable of distinguishing between results related or unrelated to the financial world by means of:

1. Filtering over Wikidata “instance_of” property (P31). This *nearest-neighbouring-based* mechanism was described in 4.4.5 and requires `min_nn` hyperparameter tuning.

2. Pruning results with a too low STRONG Similarity score computed over document embeddings. This minimum similarity is another hyperparameter that influences accuracy measurements, let's call it `min_similarity`.

Another hyperparameter is the size of the chunks (`chunk_size`) used to split the computation of all the candidate entities combinations. For this evaluation, this parameter was empirically fixed to 5.

In order to measure *accuracy*, *f1*, *recall* and *precision* of the NED model, it is mandatory to provide a clear definition of True Positives, True Negatives, etc. as done in table 15.

Table 15: TP, TF, FP, FN definition

Manual Evalutation	NED returns an entity?	Right entity?	Treated as
Good	Yes, $\geq \text{min_similarity}$	Yes	True Positive (TP)
Good	Yes, $\geq \text{min_similarity}$	No	False Positive (FP)
Bad	Yes, $\geq \text{min_similarity}$	Yes	False Positive (FP)
Bad	Yes, $\geq \text{min_similarity}$	No	False Positive (FP)
Good	No	No	False Negative (FN)
Bad	No	No	True Negative (TN)
Good	Yes, $< \text{min_similarity}$	Yes	False Negative (FN)
Good	Yes, $< \text{min_similarity}$	No	False Negative (FN)
Bad	Yes, $< \text{min_similarity}$	No	False Negative (FN)
Bad	Yes, $< \text{min_similarity}$	Yes	True Negative (TN)

Formulas for *accuracy*, *f1*, *recall* and *precision* are provided below:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (4.5)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (4.6)$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (4.7)$$

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4.8)$$

Important: as already said, table 15 does not take into account cases in which the NED algorithm fails to match a word due to the lack of the corresponding entity in Wikidata, since those terms were previously pruned from the test-set.

As can be seen from figures 13, 14, 15 and 16, the maximum accuracy achieved is 0.766 with `min_nn = 1` and `min_similarity = 0.50`. Similarly, the highest F1 score is 0.821 with `min_nn = 1` and `min_similarity = 0.50`. It is interesting to note that accuracy and F1 have similar shape, leading us to think they have the same underlying meaning in this kind of evaluation.

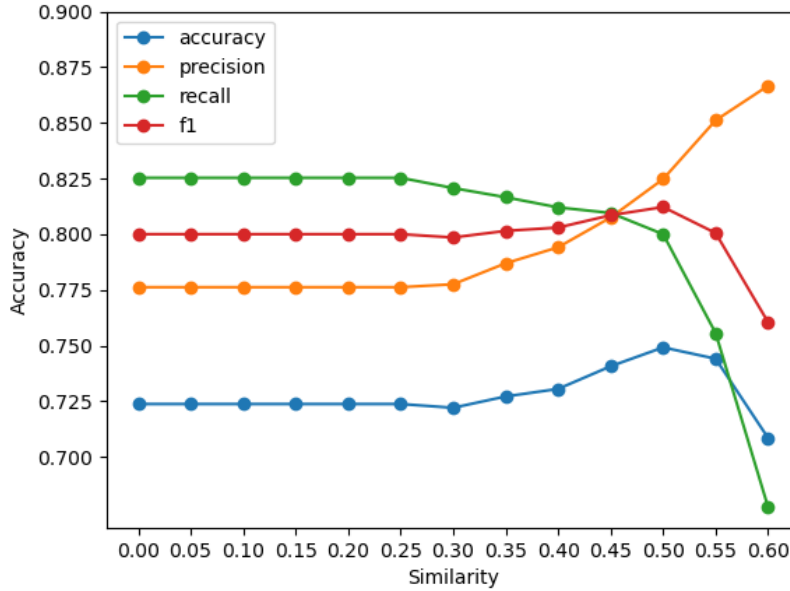


Figure 13: `min_nn = 0.65`; `min_similarity = [0, 0.6]`

4.5.2 Evaluation of the word embedding models removing unmatched terms

The NED Algorithm acts as an additional filtering step over the results of the word embedding model. Hence, the overall accuracy measurement of the full hybrid system, built as a pipeline of the three machine learning models (PoS, W2V, NED), **can be computed as the accuracy of the word embedding models (section 3.4), removing from the test set:**

1. All the unresolvable terms due to missing corresponding entity in Wikidata
2. **All the terms not matched by the NED algorithm or with a similarity lower than `min_similarity`**

Of course, hyperparameters `min_nn` and `min_similarity` were fixed respectively to 1 and 0.50 as found in section 4.5.1. Please note the NED algorithm may return slightly different results at every computation and that for this measurement the number of samples per

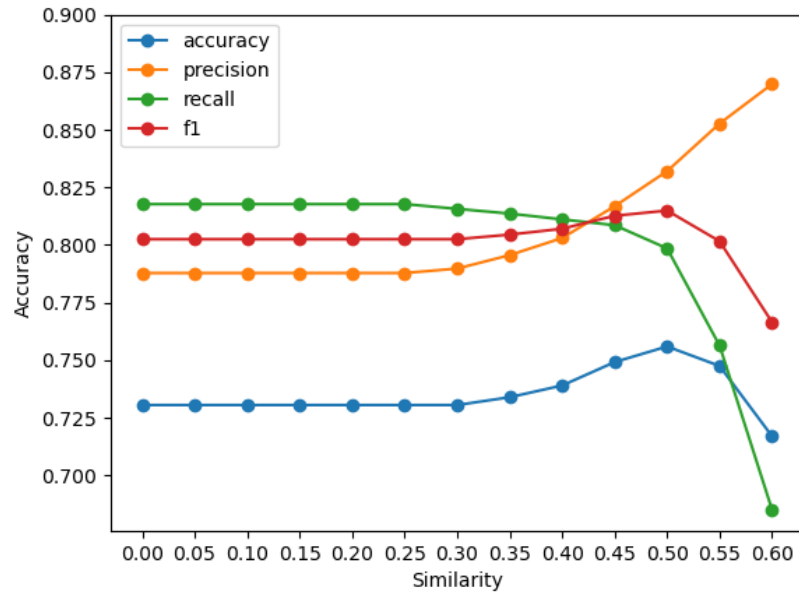


Figure 14: min_nn = 0.75; min_similarity = [0, 0.6]

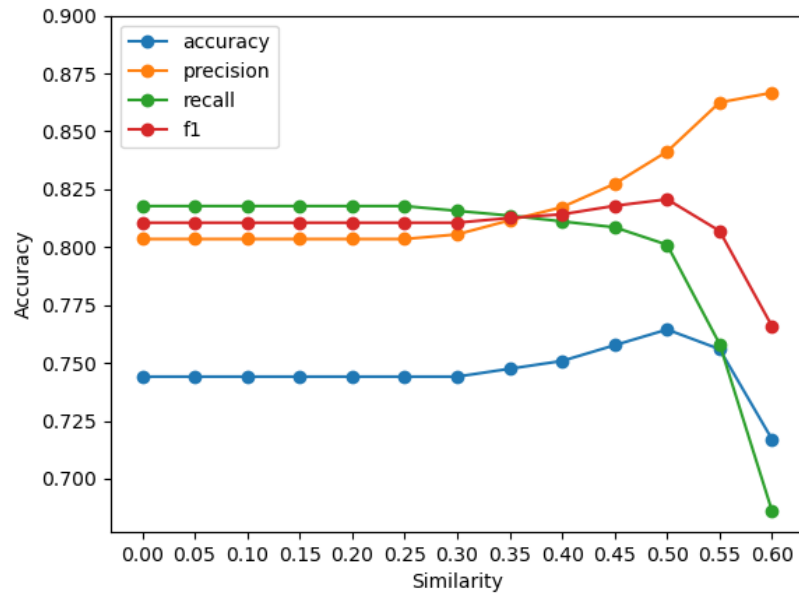


Figure 15: min_nn = 0.85; min_similarity = [0, 0.6]

single model is relatively low. Therefore, the computation was repeated three times and mean values are provided in table 16.

Again, model m5_s300_cb reaches the best accuracy followed by m5_s100_sg and m1_s300_cb. Interestingly, m5_s100_sg has a higher accuracy increment, if compared to the values in table 7, than the

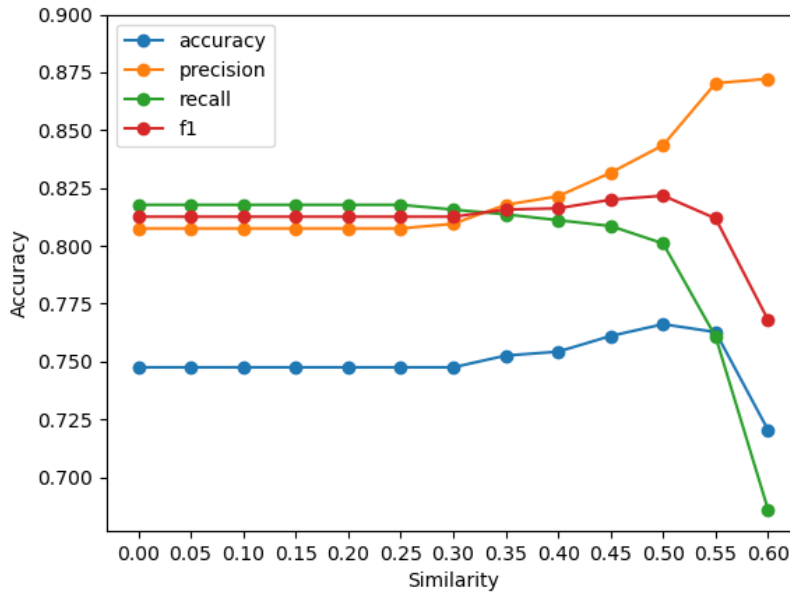


Figure 16: min_nn = 1; min_similarity = [0, 0.6]

Table 16: Accuracy of the word embedding models removing unmatched terms

Model name	Total num. of results after pruning	Accuracy
m1_s300_cb	183	0.927
m1_s100_cb	150	0.88
m5_s100_cb	171	0.867
m5_s100_sg	128	0.935
m5_s300_cb	169	0.940
m5_s300_sg	137	0.910

other models. However, it has only 128 samples versus the 169 of model m5_s300_cb and the 183 of model m1_s300_cb.

5

WEB APPLICATION FOR MODEL QUERYING

5.1 OVERVIEW

This chapter presents the development process of a web application that allow users to inference the models built so far. As described in the next sections, this application uses a Javascript front-end framework and a Python back-end framework. Particular attention has been paid to the UX and UI design, in order to provide a pleasant and fast user experience. The application name is *FiSHeR: Financial Service Heuristic Retrieval*.

5.2 FRAMEWORKS CHOICE

5.2.1 JavaScript front-end framework

As of 2020, several JavaScript-based front-end frameworks are available, the most famous are: *React*¹, *Angular*² and *Vue.js*³.

They allow to build *reactive web applications*. Reactivity is the ability of a web framework to update your view whenever the application state has changed. Reactive programming is a development paradigm based on *declarative state-driven code*, that allows to dissect user interfaces into logical units called *components*. These units interact with each other providing fast cutting-edge user experience. This is generally achieved using techniques like *virtual DOM diffing* that updates the view at *run-time*.

Reactive programming, however, requires ad-hoc syntax and forces the browser to do extra work to convert those declarative structures into DOM operations. These techniques, indeed, cannot apply changes to the real DOM without first comparing the new virtual DOM with the previous snapshot, introducing additional overhead in terms of speed and app size.⁴.

¹ <https://reactjs.org>

² <https://angular.io>

³ <https://vuejs.org>

⁴ <https://svelte.dev/blog/virtual-dom-is-pure-overhead>

This project, however, uses a new JavaScript front-end framework called *Svelte*⁵, that shifts this extra work made by the browser into a compile step that happens at *build-time*, converting components into highly efficient *imperative code* that surgically updates the DOM. Svelte, indeed, is not a real framework but a compiler that preserves the benefits of reactivity while building smaller-size framework-less vanilla JavaScript applications. Consequently, Svelte also delivers a better developer experience thanks to an easier syntax and almost no boilerplate code. Although being quite young (version 3 was released in 2019), Svelte is quickly raising popularity in the JavaScript community, as stated in a 2019 survey carried over 21,717 respondents⁶.

All the code developed by Svelte will be deployed using *nginx*⁷ web server.

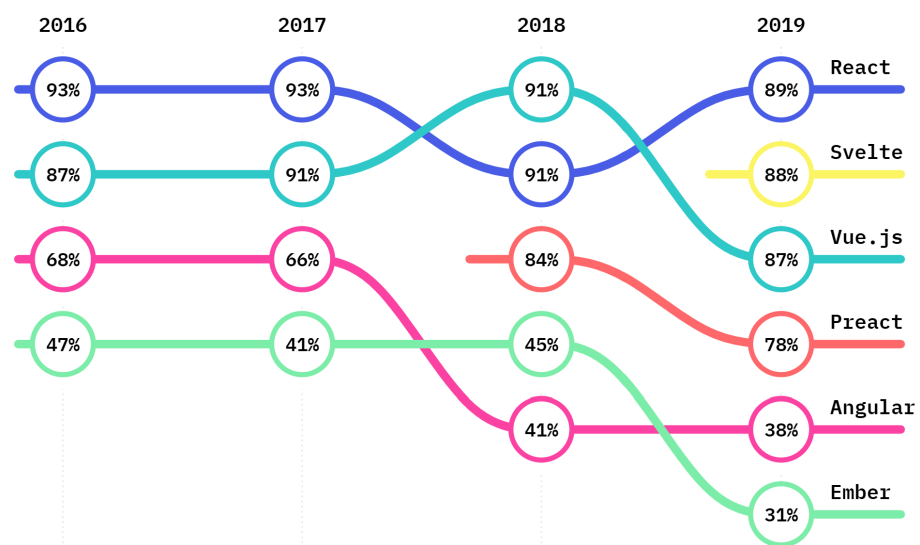


Figure 17: Satisfaction rate for JavaScript front-end frameworks. Source: <https://2019.stateofjs.com/front-end-frameworks/>

5.2.2 CSS front-end framework

CSS Frameworks provide pre-written CSS classes that can be used to reduce the time needed for user interface styling. Most of these frameworks deliver:

1. built-in responsiveness
2. cross-compatibility between browsers

⁵ <https://svelte.dev>

⁶ <https://2019.stateofjs.com/>

⁷ <https://www.nginx.com>

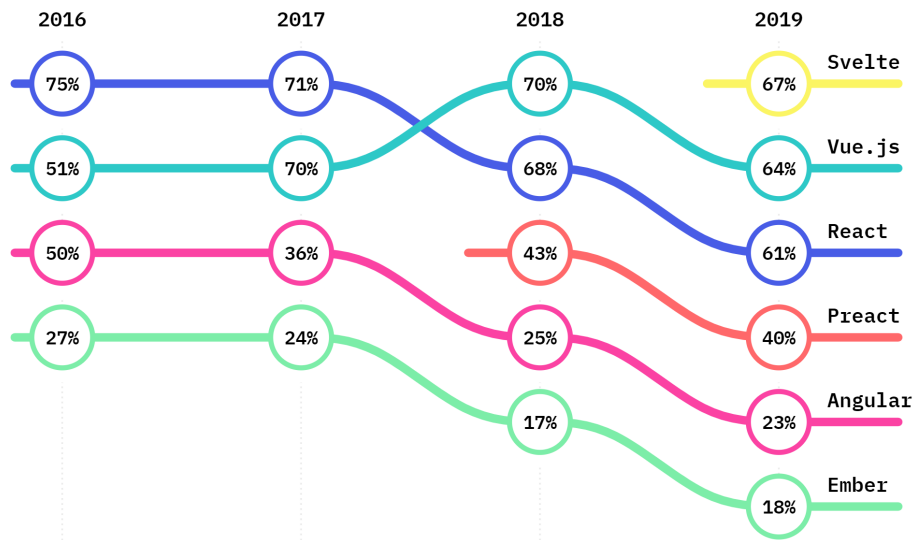


Figure 18: Interest rate for JavaScript front-end frameworks. Source: <https://2019.stateofjs.com/front-end-frameworks/>

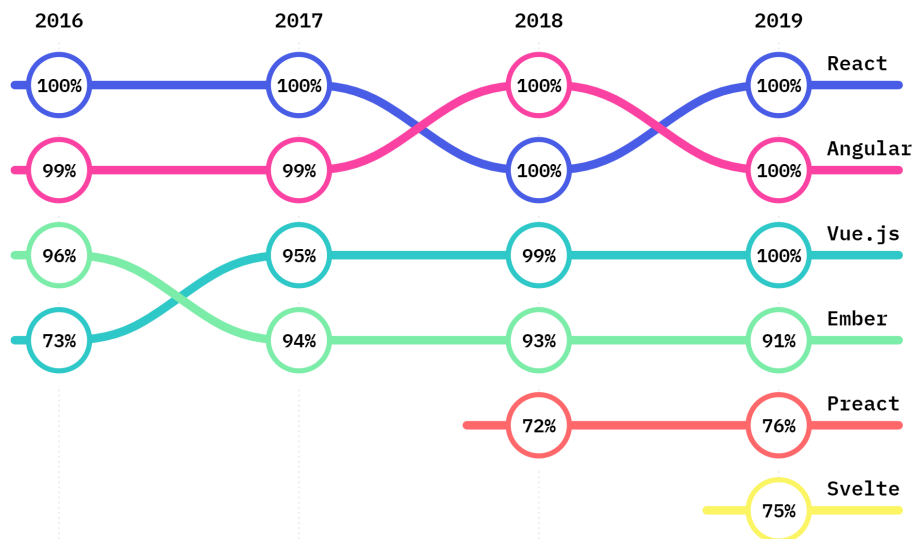


Figure 19: Awareness rate for JavaScript front-end frameworks. Source: <https://2019.stateofjs.com/front-end-frameworks/>

3. usage of pseudo-CSS languages like *Less*⁸ or *Sass*⁹

This application uses *Bulma*¹⁰ framework. Although not being the most adopted, it is very small in size and it has been developed as a pure-CSS framework with no additional JavaScript code. The latter feature goes very well with Svelte, since it handles all the dynamic behaviours on its own.

⁸ <http://lesscss.org/>

⁹ <https://sass-lang.com/>

¹⁰ <https://bulma.io>

5.2.3 Python back-end framework

Back-end is powered by Python *AIOHTTP*¹¹ framework that was already used for asynchronous calls in the NED algorithm described in section 4.4. More in detail, AIOHTTP delivers:

1. Both client and http web server
2. Client and server WebSockets out-of-the-box
3. Middlewares, signals and pluggable routing for web server

Of course, we are only going to use the server-side functionalities of AIOHTTP. The server deployment will be carried out using *Gunicorn*¹² Python WSGI HTTP Server for UNIX.

5.3 WEB APPLICATION DESCRIPTION

5.3.1 Web Application Overview

The description of both the front-end and the back-end code is carried out starting from the main parts of the application Graphic User Interface. Indeed, FiSHER GUI is composed of six main areas that we are going to analyze in next the sections:

1. A search bar
2. A container with details of target word matched entity
3. A container with matched entities details
4. A container with unresolved terms
5. A container with discarded terms
6. A D3 force graph

Before going through the description of the parts above, it is important to show the preliminary initializing steps performed by both the front-end and the back-end code.

On the front-end side, the Svelte application is composed as single page represented by `App.svelte` component. This parent component wraps all other components and orchestrate their behaviours. The structure of the front-end application is:

¹¹ <https://docs.aiohttp.org/en/stable/>

¹² <https://gunicorn.org>

- `App.svelte`
 - `D3ForceGraph.svelte`
 - `EntityCard.svelte`
 - `D3ForceGraph.svelte`
 - `OccBox.svelte`
 - `SearchBar.svelte`
 - `TargetEntity.svelte`
 - Some minor components

On the back-end side, an `app` method is defined to perform the following activities:

1. Creating an AIOHTTP application object instance
2. Creating an instance of class `Server`
3. Defining callable endpoints for the application instance and attaching methods of class `Server` to them

The initializer method of class `Server` is in charge of loading both the Word Embedding model and the Universal Sentence Encoder model (creating an instance of class `NED`). Moreover, it defines a python dictionary (`vocab`) with the occurrences of each word appeared in the former model and a sorted list of words that can be searched in FiSHeR (`orig_words`) (see subsection 5.3.2).

Please note that only words with occurrences in the empirically-fixed range `[15, 150000]` are allowed to be searched. Words with occurrences higher than 150000 are most likely generic words that are of no interest for this application and we want them not to be displayed in the auto-complete functionality of FiSHeR search-bar (the first proper name is *Google* with 149276 occurrences). Words with occurrences lower than 15 generally provide very bad results that are not worth to be displayed. **It is mandatory to specify that this lower-bound does not contradict the `min_count = {1,5}` parameter used to train word embedding models. Indeed, a word with low occurrences can still be returned as a result of the search of a term with higher occurrences.**

Listing 22: Back-end: method `_init_` of class `Server` and method `app`

```
class Server(object):
    def __init__(self, model_name, vocab_name, event_loop)
        :
        self.vocab = {}

    # load the word2vec model from gzipped file
    self.model = gensim.models.KeyedVectors.
        load_word2vec_format(model_name, binary=True)
```

```

self.model.init_sims(replace=True)
try:
    del self.model.syn0
    del self.model.syn1
except:
    pass

try :
    with open(vocab_name, 'r') as dictfile:
        for line in dictfile:
            cells = line.strip().split()
            self.vocab[cells[0]] = int(cells[1])
    self.orig_words = [gensim.utils.to_unicode(word) for
        word in self.model.wv.index2word if self.vocab[
            gensim.utils.to_unicode(word)] >= 15 and self.
            vocab[gensim.utils.to_unicode(word)] <= 150000]
except:
    self.orig_words = [gensim.utils.to_unicode(word) for
        word in self.model.wv.index2word]

indices = [i for i, _ in sorted(enumerate(self.
    orig_words), key=lambda item: item[1])]
self.orig_words = [self.orig_words[i] for i in
    indices]
self.ned = NED(event_loop = event_loop)

[...]

async def app():
    logger.info("running %s" % ' '.join(sys.argv))

    nest_asyncio.apply()

    loop = asyncio.get_event_loop()
    app = aiohttp.web.Application(loop=loop)

    load_dotenv()
    client_url = os.getenv('CLIENT_URL')
    wv_model_path = os.getenv('W2V_MODEL_PATH')
    wv_vocab_path = os.getenv('W2V_VOCAB_PATH')
    host = os.getenv('HOST', 'localhost')
    port = int(os.getenv('PORT', 3000))

    aiohttp_cors.setup(app, defaults={
        client_url : aiohttp_cors.ResourceOptions()
    })

    server_app = Server(wv_model_path, wv_vocab_path, loop
        )

```

```

cors_sgt = app.router.add_route('GET', '/sgt',
                                server_app.suggest)
app['aiohttp_cors'].add(cors_sgt)

app.router.add_route('GET', '/wsms', server_app.
                     ws_most_similar)
aiohttp.web.run_app(app, host=host, port=port)
# return app

if __name__ == '__main__':
    app()

```

5.3.2 Search-bar

The search-bar is a single Svelte component, called `SearchBar.svelte`, that let the user select the target term to search into the model. The word embedding model only allows to search words that appeared at least once in the training dataset. Therefore, this component calls the back-end to retrieve and display a set of searchable words, starting with the text typed into the search-bar, to be chosen by the user from a dropdown list. The component itself is also in charge of denying to query the model if a word not in this list is submitted.

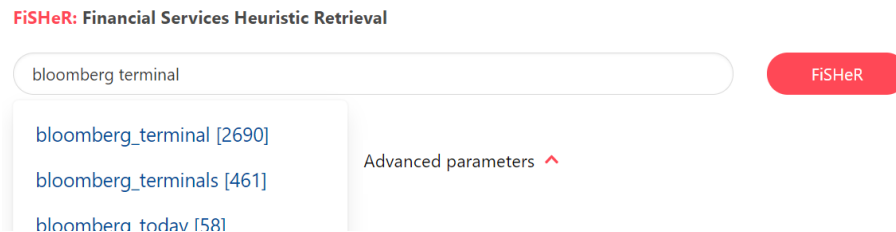


Figure 20: FiSHeR Search bar

In particular, the component makes a GET request to the python back-end endpoint `/sgt?term={target_worg}` passing the string that was typed into the search-bar. This HTTP request calls the back-end `suggest` method of class `Server` that returns the first ten words, from variable `orig_words` described in 5.3.1, starting with GET variable `term`. Moreover, it also returns the number of occurrences of each of these terms.

When the search button is submitted, `App.svelte` calls back-end endpoint `/wsms` and method `ws_most_similar` of class `Server` is triggered. This method queries the word embedding model with target word just searched by the user. The output of this model is then given as input to the NED algorithm, whose results, in turn, are displayed

to the user with some additional computation. `App.svelte` and `ws_most_similar` establish a *WebSocket* connection as a full-duplex TCP communication channel, that will be used to share the results of the Word Embedding and NED models. This is required because the inference of such models requires a delay of a few seconds that a standard HTTP call could not handle. WebSocket protocol was standardized by the IETF as RFC 6455 [12] and was intended to replace older approaches like *Long Polling*.

`ws_most_similar` will be analyzed more in detail later on, however, it returns a json encoded string with the following tags:

1. `st_good`: The json-encoded Dataframe of matched entities resolved by the NED algorithm described in subsection 4.4.5
2. `st_miss`: A list of unresolved word embedding terms described in subsection 4.4.6
3. `st_bad`: A list of discarded word embedding terms described in subsection 4.4.6
4. `wv_sum_occ`: A json-encoded python dictionary with the sum of the occurrences of each word embedding term matched against the same entity, plus the occurrences of the not matched terms.
5. `wv_dist`: A json-encoded python dictionary with the (WEAK) cosine similarities of each word embedding term.
6. `nodes`: A json-encoded Dataframe with nodes of the D3 force graph, more about this is in section 5.3.6
7. `links`: A json-encoded Dataframe with edges of the D3 force graph, more about this can be found in section 5.3.6

All the above information are retrieved by `App.svelte` and passed to its child components. More details in the following subsections.

Last, the `App.svelte` component provides some advanced search filters that are placed just below the search bar. These parameters are:

1. **Number of W2V results**: The number of terms returned by the word embedding model
2. **Chunk size**: The number of terms for each chunk, described in section 4.4.5

The selectable values for these parameters, as can be seen in picture 21, were empirically set.

Listing 23: Front-end: a piece of *App.svelte* component

[...]



Figure 21: FiSHeR advanced search parameters

```

{#if advanced_flag}
<div class="is-flex advanced-container" transition:
  slide>
<div class="field has-addons has-addons-centered mx-3
  ">
  <div class="field-label is-small">
    <label class="label">Number of V2W results</label>
  </div>
  <div class="buttons has-addons is-inline-block">
    <button class="button is-rounded is-small is-light"
      ></button>
    {#each nr_tick as tick, i}
      <button on:click={() => nr_idx = i} class="button
        is-rounded is-small mx-1 {nr_idx == i ? 'is-
          primary' : 'is-light'}">{tick}</button>
    {/each}
    <button class="button is-rounded is-small is-light"
      ></button>
  </div>
</div>
<div class="field has-addons has-addons-centered mx-3
  ">
  <div class="field-label is-small">
    <label class="label">Chunk size</label>
  </div>
  <div class="buttons has-addons is-inline-block">
    <button class="button is-rounded is-small is-light"
      ></button>
    {#each cs_tick as tick, i}
      <button on:click={() => cs_idx = i} class="button
        is-rounded is-small mx-1 {cs_idx == i ? 'is-
          primary' : 'is-light'}">{tick}</button>
    {/each}
    <button class="button is-rounded is-small is-light"
      ></button>
  </div>
</div>
</div>
{#if}

[...]
```

Listing 24: Front-end: *SearchBar.svelte* component

```

<script>
import Icon from 'svelte-awesome';
import { faExclamationTriangle } from '@fortawesome/
  free-solid-svg-icons';
import { createEventDispatcher, tick } from 'svelte';
const dispatch = createEventDispatcher();
let search_term = ''
let current_terms = []
let tmp_occ = []
let all_terms = new Set();
let hover_index = -1
let input_focus = false
let danger_flag = false
export let isWsLoading = false

$: st_regex = search_term.replace(/^[^w\s&]/gi, '').
  replace(/[\s]/g, '_')

const fetch_ws_sgt = async (e) => {
  if (e.keyCode === 13) {
    if(all_terms.has(st_regex)){
      danger_flag = false
      dispatch('startsearch', st_regex)
      current_terms = []
      tmp_occ = []
      hover_index = -1
    } else {
      danger_flag = true
      current_terms = []
      tmp_occ = []
      hover_index = -1
    }
  } else if (e.keyCode === 40) {
    // ArrowDown
    hover_index = (hover_index + 1) < current_terms.
      length ? hover_index + 1 : current_terms.length
      - 1
    search_term = current_terms[hover_index]
  } else if (e.keyCode === 38) {
    // ArrowUp
    hover_index = (hover_index - 1) > 0 ? hover_index -
      1 : 0
    search_term = current_terms[hover_index]
  } else {
    if (st_regex !== ''){
      const res = await fetch('http://SERVER_ENDPOINT/sgt
        ?term='+st_regex, {
        method: 'GET',
        headers: { 'Origin' : window.location.host}

```

```

    })

    let response = await res.json()
    console.log(response)
    current_terms = response['wv_sim']
    tmp_occ = response['wv_occ']
    current_terms.forEach(item => all_terms.add(item))
  } else {
    current_terms = []
    tmp_occ = []
  }
  hover_index = -1
}
}

const select_term = async (e, i) => {
  search_term = current_terms[i]
  // Fix st_regex reactivity
  await tick()
  if(all_terms.has(st_regex)){
    danger_flag = false
    dispatch('startsearch', st_regex)
    current_terms = []
    tmp_occ = []
    hover_index = -1
  } else {
    danger_flag = true
    current_terms = []
    tmp_occ = []
    hover_index = -1
  }
}

const select_submit = () => {
  if(all_terms.has(st_regex)){
    danger_flag = false
    dispatch('startsearch', st_regex)
    current_terms = []
    hover_index = -1
  } else {
    danger_flag = true
    current_terms = []
    hover_index = -1
  }
}

</script>

<div class="columns">
  <div class="column is-10 pb-0">
    <div class="field">
      <div class="control is-expanded has-icons-right">

```

```

<input class="input is--rounded is--small {
  danger_flag ? 'is--danger' : ''} { isWsLoading ?
  'has--text--white' : ''}" autocomplete="off"
  disabled={isWsLoading} type="text" on:focus={()
  => input_focus=true} on:blur={() => input_focus=
  false} id="search-bar" on:keyup={fetch_ws_sgt}
  bind:value={search_term}>
  {#if danger_flag}
    <span class="icon is--small is--right">
      <Icon data={faExclamationTriangle}/>
    </span>
    <p class="help is--danger ml-4">Word is not in
      vocabulary</p>
  {/if}
  <div class="dropdown {current_terms.length > 0 ? '
    is--active' : ''}">
    <div class="dropdown--menu" role="menu">
      <div class="dropdown--content">
        {#each current_terms as candidate, i}
          <a href="#" on:click={(e) => select_term(e, i)}
            on:mouseover={() => hover_index = i} on:
            mouseout={() => hover_index = -1} class="
            dropdown--item has--text--left { i ===
            hover_index ? 'has--background--light' : ''}"
            >
              {candidate} [{tmp_occ[i]}]
            </a>
          {/each}
        </div>
      </div>
    </div>
  </div>
  </div>
  <div class="column is--2" id="sbt">
    <button class="button is--danger is--small is--fullwidth
      is--rounded {isWsLoading ? 'is--loading' : ''}" on
      :click={select_submit} >FiSheR</button>
  </div>
</div>

<style>
@media screen and (max-width: 769px) {
  #sbt{
    padding-top:0px;
    margin-bottom: 20px;
  }
}
</style>

```

5.3.3 Container with details of target word matched entities

This part of the GUI is mainly composed of `TargetEntity.svelte` component. It shows the structured information provided by the matched entity for the target word:

1. The title of the Wikidata Entity
2. A description for the Wikidata Entity
3. Some useful links like: official website, Wikidata and Wikipedia pages
4. All the word embedding matching terms, that are the target word plus every other word embedding term that was matched against the same Wikidata entity
5. The sum of the occurrences of all the above words

The sum of the occurrences is retrieved from tag `wv_sum_occ` while all other information from tag `st_miss` of the output of method `ws_most_similar`. The useful links are displayed as high-contrast buttons to be immediately recognized by the user and to allow further investigation for the entity displayed.

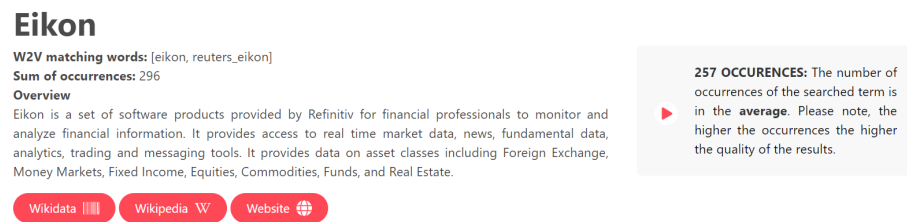


Figure 22: Details component for target word *eikon*

Listing 26: Front-end: *TargetEntity.svelte* component

```
<script>
  export let entityInfo = {}
  export let occ = 0
  import Icon from 'svelte-awesome';
  import { faBarcode, faChevronDown, faGlobe, faMinus }
    from '@fortawesome/free-solid-svg-icons';
  import { faWikipediaW } from '@fortawesome/free-brands
    -svg-icons';
  import { slide } from 'svelte/transition';
  let show_more = false
</script>

<div class="has-text-left">
  <h1 class="is-size-3 has-text-weight-bold">{entityInfo
    ['wd_title']}</h1>
```

```

<p><span class="has-text-weight-bold">W2V matching
  words:</span> [{entityInfo['search_term'].join(", "
    )}]</p>
<p><span class="has-text-weight-bold">{#if entityInfo
  ['search_term'].length > 1}Sum of occurrences:{:
  else}Occurrences:{/if}</span> {occ}</p>
<p class="has-text-weight-bold">Overview</p>
<p class="has-text-justified mb-3">
  {#if entityInfo['overview'].length < 530}
  {entityInfo['overview']}
  {:else}
    <span>{entityInfo['overview'].slice(0, 530)}{#if !
      show_more}...{/if}</span>
    {#if show_more}
      <span transition:slide>{entityInfo['overview'].slice
        (531)}</span>
    {/if}
    <a href="#" class="has-text-primary" on:click={() =>
      show_more = !show_more}>Read {#if !show_more}
      More{:else}Less{/if}</a>
  {/if}
</p>
<a href="https://www.wikidata.org/wiki/{entityInfo['
  entity_id']}" target="_blank" class="button is-
  primary is-small is-rounded">Wikidata &nbsp; <Icon
  data={faBarcode}/></a>
{#if entityInfo['wp_title'] != null }<a href="https://
  en.wikipedia.org/wiki/{entityInfo['wp_title']}"
  target="_blank" class="button is-primary is-small
  is-rounded">Wikipedia &nbsp; <Icon data={
  faWikipediaW}/></a>{/if}
{#if entityInfo['websites'] != null && entityInfo['
  websites'].length > 0 }<a href="{entityInfo['
  websites'][0]}" target="_blank" class="button is-
  primary is-rounded is-small">Website &nbsp; <Icon
  data={faGlobe}/></a>{/if}

</div>

```

Since the number of occurrences of the target word heavily influences the quality of the results displayed by the whole system, an additional component was reserved to print different messages about the goodness of the search based on the value of this parameter. It is displayed as a separate box on the right of the screen. Three different messages are available:

1. **OCCURRENCES < 50:** *The number of occurrences of the searched term is very low. Most likely, poor quality results will be shown.*

2. **$50 \leq \text{OCCURRENCES} < 500$** : *The number of occurrences of the searched term is in the average. Please note, the higher the occurrences the higher the quality of the results.*
3. **$\text{OCCURRENCES} \geq 500$** : *Woah! The number of occurrences of the searched term is very high. You should get amazing results!*

Listing 27: Front-end: *OccBox.svelte* component

```

<script>
import Icon from 'svelte-awesome';
import {faExclamationCircle, faPlayCircle,
      faCheckCircle} from '@fortawesome/free-solid-svg-
      icons';
import lang from '../public/json/lang_en.json'
export let occ = 0
</script>

<article class="media px-4 py-4">
  <div class="media-left">
    {#if occ < 50}
      <Icon data={faExclamationCircle} class="has-text-
        white" scale="1.5" />
    {:else if occ < 500}
      <Icon data={faPlayCircle} class="has-text-white"
        scale="1.5" />
    {:else}
      <Icon data={faCheckCircle} class="has-text-white"
        scale="1.5" />
    {/if}
    <div style="width:15px; height:15px; margin-top:-25
      px; margin-left:5px" class="has-background-
        primary"></div>

  </div>
  <div class="media-content">
    <div class="content">
      <p class="has-text-black has-text-justified"><strong
        >{occ} OCCURRENCES: </strong>
        {#if occ < 50}
          {@html lang['low_occ']}
        {:else if occ < 500}
          {@html lang['med_occ']}
        {:else}
          {@html lang['high_occ']}
        {/if}
      </p>
    </div>
  </div>
</article>

<style>

```

```

    article{
border: 1px solid rgba(0, 0, 0, 0);
border-radius: 5px;
background-color: #f5f5f5b0;
    }

    .media-left{
margin-top:auto;
margin-bottom:auto;
    }
</style>

```

5.3.4 Container of matched entities details

This area of the user interface is built as a list of instances of the component `EntityCard.svelte`. A new instance is created for each entity returned by the NED algorithm, with the exception of the entity matched to the target word. Each one provides the same information displayed by component `TargetEntity.svelte`. Again, useful links are displayed as easily reachable buttons for quicker external information retrieval.

This component also displays a STRONG similarity score computed as the cosine similarity between the USE embeddings of the current entity and the target one (see 4.4.5). If however, the NED algorithm was not able to match the target word, no STRONG similarity can be computed and the (WEAK) Word Embedding one is shown instead. A dropdown infobox was placed above all the instances of this component to clarify this aspect and the meaning of 'Named Entity Disambiguation' (NED) to the user (image 24).

Listing 28: Front-end: *EntityCard.svelte* component

```

<script>
[...]
```

```

</script>

<div class="card">
  <header class="card-header {hover_flag ? 'has-background-white-ter card-hover' : ''}" on:click=
    ={{() => open_flag = ! open_flag}}>
    <p class="card-header-title" on:mouseenter={{() =>
      hover_flag = true}} on:mouseleave={{() => hover_flag
        = false}}>
      {#if has_root}
        {#if similarity >= 0.60}
          <Icon class="has-text-success is-hidden-mobile"
            data={faMinus}/>
        {:else}

```

NED resolved terms [STRONG Similarity] ⓘ






















— Bloomberg Terminal	  	STRONG Similarity 77%	✓
— FactSet	  	STRONG Similarity 77%	✓
— MetaStock	  	STRONG Similarity 76%	^
<p>W2V matching words: [metastock]</p> <p>Occurrences: 41</p> <p>Overview</p> <p>MetaStock is a proprietary computer program originally released by Computer Asset Management in 1985. It is used for charting and technical analysis of stock (and other asset) prices. It has both real-time and end-of-day versions. MetaStock is a product of Innovative Market Analysis.</p>			
— Datastream	 	STRONG Similarity 75%	✓
— Intrinio	 	STRONG Similarity 70%	✓
— S&P Capital IQ	  	STRONG Similarity 70%	✓
— Value Line	  	STRONG Similarity 69%	✓
— Quandl	 	STRONG Similarity 58%	✓

Figure 23: List of *EntityCard* components

NED resolved terms [STRONG Similarity] ⓘ

These terms were resolved by a **Named Entity Disambiguation** algorithm, which is capable of providing structured information as: official websites, Wikipedia pages and more. **If the searched term is resolved too** . the application can provide a stronger similarity measure for resolved terms computed over document embeddings.




— Bloomberg Terminal	  	STRONG Similarity 77%	✓
----------------------	--	-----------------------	---

Figure 24: Infobox close-up

```

{#if similarity >= 0.40}
  <Icon class="has-text-warning is-hidden-mobile"
    data={ faMinus}/>
{:else}
  <Icon class="has-text-danger is-hidden-mobile"
    data={ faMinus}/>
{/if}

```

```

    {/if}
  {:else}
    <Icon class="has-text-light is-hidden-mobile" data
      ={faMinus}/>
  {/if}
  <span>&nbsp; {entityInfo['wd_title']}

```

```

<p><span class="has-text-weight-bold">{#if
  entityInfo['search_term'].length > 1}Sum of
  occurrences:{:else}0occurrences:{/if}</span> {occ
}</p>
<p class="has-text-weight-bold">Overview</p>
<p>
  {#if entityInfo['overview'].length < 530}
  {entityInfo['overview']}
  {:else}
  <span>{entityInfo['overview'].slice(0, 530)}{#if
    !show_more}...{/if}</span>
  {#if show_more}
  <span transition:slide>{entityInfo['overview'].
    slice(531)}</span>
  {/if}
  <a href="#" class="has-text-primary" on:click={()
    => show_more = !show_more}>Read {#if !
    show_more}More{:else}Less{/if}</a>
  {/if}
</p>
</div>
</div>
</div>
{/if}
</div>

<style>
[...]
```

5.3.5 Container of unresolved terms and container of discarded terms

In this case, there is not a dedicated Svelte component to display unresolved and discarded terms returned by method `ws_most_similar` and described in 4.4.6. But they are just injected in the HTML code by means of two dedicated svelte loops in the main `App.svelte` component.

The unresolved terms are displayed as buttons groups, see picture 25:

1. The *left-most button* prints the term string, the number of occurrences and the WEAK similarity. If pressed, it opens a new browser window in which the term is searched in the Google search engine.
2. The *button in the middle* let the user search back the term into FiSHeR.

3. The *right-most button* opens a new browser window with the Wikidata page to add a new entity.

A drop-down infobox is printed to explain the meaning of ‘unresolved terms’ and ‘weak similarity’ to the user.

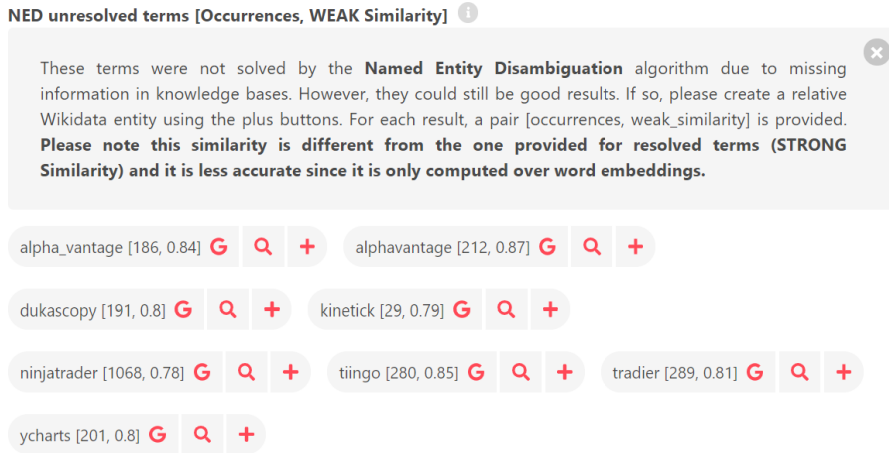


Figure 25: Example of unresolved terms for target word *iqfeed*

The discarded terms are printed as simple HTML anchor tags with link name that contains the term string, the number of occurrences and the WEAK similarity. If the tag is clicked, the term is searched back into FiSheR. Again, an infobox is provided to explain the meaning of ‘discarded terms’ and ‘weak similarity’ (see image 26)

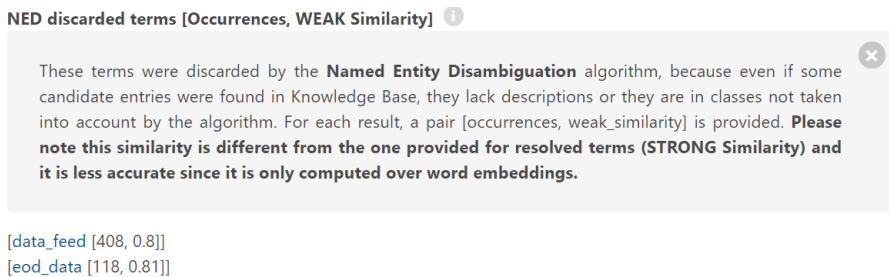


Figure 26: Example of discarded terms for target word *iqfeed*

Listing 29: Front-end: a piece of *App.svelte* component

[...]

```
<div class="column is-6">
  <SmallTitle title={has_root ? 'NED resolved terms [
    STRONG Similarity]' : 'NED resolved terms [WEAK
    Similarity]'} snippet={lang['resolvable_terms']}/>
  {#each unrooted_payload as row, i}
    <div transition:slide|local class="py-3 entity-
      divider">
      <EntityCard has_root={has_root} entityInfo={row} occ
        ={{payload['wv_sum_occ']}[row['bst_st']]}}/>
    </div>
  {/each}</div>
```

```

</div>
{/each}
{#if payload['st_miss'].length > 0}
<div class="has-text-left mt-3">
  <SmallTitle title="NED unresolved terms [Occurrences
    , WEAK Similarity]" snippet={lang['
      unresolvable_terms']}/>
  {#each payload['st_miss'] as uterm}
  {#if search_term != uterm}
  <div class="buttons has-addons mb-2 mr-3 is-inline
    -block">
    <a class="button is-rounded is-small is-light px
      -2" href="http://www.google.com/search?q={
        uterm.replace("_", "+")} target="_blank">{
        uterm}&nbsp;[{payload['wv_sum_occ']][uterm]},&
         [{Math.round(payload['wv_dist'])(uterm)
          100) / 100}]&nbsp;&nbsp; <Icon data={faGoogle}
          class="has-text-primary" scale="0.8"/></a>
    <button class="button is-rounded is-small is-
      light px-2" on:click={ws_send(uterm)}><Icon
        data={faSearch} class="has-text-primary" scale
          ="0.8"/></button>
    <a class="button is-rounded is-small is-light px
      -2" href="https://www.wikidata.org/wiki/
        Special:NewItem?label={firstToUpper(uterm)}"
        target="_blank"> <Icon data={faPlus} scale="
          0.8" class="has-text-primary"/></a>
  </div>
  {/if}
{/each}
</div>
{/if}
</div>

[...]

<!--Discarded results-->
{#if payload['st_bad'].length > 0}
<div class="has-text-left">
  <SmallTitle title="NED discarded terms [Occurrences,
    WEAK Similarity]" snippet={lang['discarded_terms'
    ]}/>
  {#each payload['st_bad'] as uterm}
  {#if search_term != uterm}
  <p><a href="#" on:click={ws_send(uterm)}>{uterm}</
    a>&nbsp;[{payload['wv_sum_occ']][uterm]},&nbsp;[{
      Math.round(payload['wv_dist'])(uterm) 100) /
      100}]]</p>
  {/if}
{/each}
</div>
{/if}

```

[...]

5.3.6 D3 Force Graph

D3.js is a JavaScript library for interactive data visualizations in web browsers. Among the available data representation layouts, a D3-force graph simulates physical forces on particles (nodes) and we are going to use it to display the various interlinks between the terms returned by the word embedding model (figure 27). A D3 force graph takes as input:

1. Nodes as json objects with `id` and `label` fields.
2. Edges as json objects with `target` and `source` fields, whose values are the identifiers of the nodes.

Three kinds of nodes were budgeted for this graph:

1. A *yellow* node that represents the entity matched against the target term. It has `label` equal to the Wikidata title and `id` equal to the word embedding term, matched to this entity, with the highest number of occurrences.
2. *Red* nodes, that represent all other matched entities. Again, `label` is equal to the Wikidata title and `id` is equal to the word embedding term with the highest number of occurrences for each entity.
3. *Grey* nodes represent unresolved terms. In this case both the `id` and the `label` are equal to the term itself.

Nodes can be both *dragged* and *clicked*, in the latter case the `id` field of the node is searched back into FiSHeR. An edge between two nodes means that searching the `id` of one of such nodes into the word embedding model, the `id` of the other node would be returned in the list of similar terms. Therefore, the whole set of edges is found taking all the resolved and unresolved terms and searching them back into the word embedding model. This computation is mainly carried out by back-end method `build_graph` that is called by `ws_most_similar`.

Listing 30: Front-End: *D3ForceGraph.svelte* component

```
<script>
[...]
```

```
let svg;
export let width
export let height
export let nodes
```

D3 Force Graph Visualization

● = searched term ● = resolved terms ● = unresolved terms

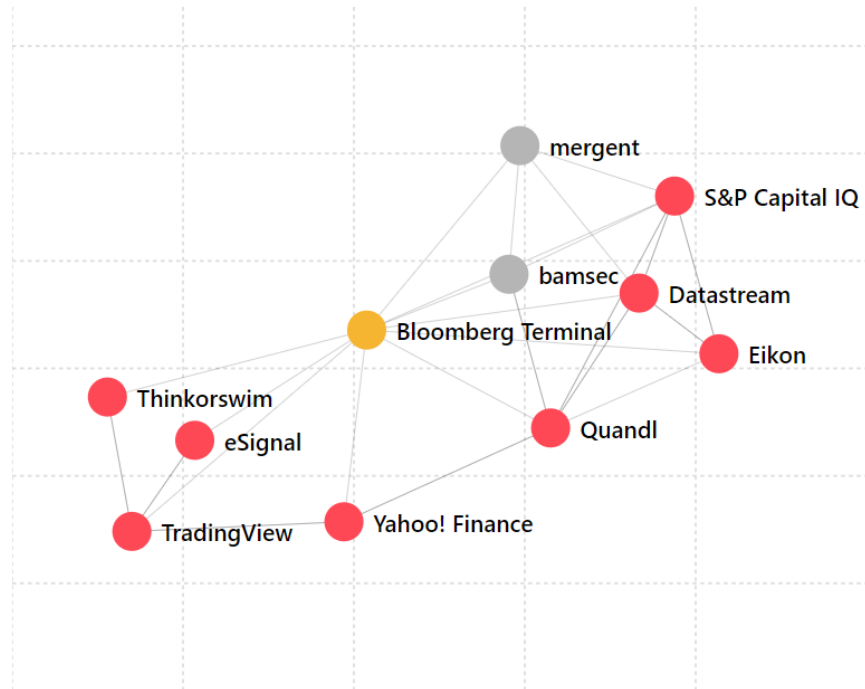


Figure 27: Example of D3 force graph for target word *bloomberg_terminal*

```
export let links
export let has_root
const nodeRadius = 10

const padding = { top: 20, right: 0, bottom: 0, left:
  0 };

let transform = d3.zoomIdentity;
let simulation

$: xTicks = width > 180 ?
  [0, 4, 8, 12, 16, 20] :
  [0, 10, 20];

$: yTicks = height > 180 ?
  [0, 2, 4, 6, 8, 10, 12] :
  [0, 4, 8, 12];

$: xScale = scaleLinear()
  .domain([0, 20])
  .range([padding.left, width - padding.right]);

$: yScale = scaleLinear()
  .domain([0, 12])
  .range([height - padding.bottom, padding.top]);
```

```

$: {
  if (simulation !== null && simulation.force("center")
    !== null){
    simulation.force("center").x(width / 2).y(height /
      2);
    simulation.alpha(0.3).restart();
  }
}

onMount( async () => {
  simulation = d3.forceSimulation(nodes)
    .force("link", d3.forceLink(links).id(d => d.id))
    .force("charge", d3.forceManyBody().strength(-500))
    .on('tick', simulationUpdate);

  d3.select(svg)
    .call(d3.drag()
      .container(svg)
      .subject(dragsubject)
      .on("start", dragstarted)
      .on("drag", dragged)
      .on("end", dragended)).call(d3.zoom()
      .scaleExtent([1 / 10, 8])
      .on('zoom', zoomed));

  await tick()
  simulation.force("center", d3.forceCenter(width / 2,
    height / 2))

});

const fire_search = (search_term) =>{
  dispatch('startsearch', search_term)
}

const fill_color = (d, i) =>{
  return i == 0 && has_root ? "#F4B400" : d.id == d.
    label ? "#b5b5b5" : "#ff4c56"
}

function simulationUpdate () {
  simulation.tick();
  nodes = [...nodes];
  links = [...links];
}

function zoomed() {
  transform = currentEvent.transform;
  simulationUpdate();
}

```

```

function dragsubject() {
  const node = simulation.find(transform.invertX(
    currentEvent.x), transform.invertY(
    currentEvent.y), nodeRadius + 5);
  if (node) {
    node.x = transform.applyX(node.x);
    node.y = transform.applyY(node.y);
  }
  return node;
}

function dragstarted() {
  if (!currentEvent.active) simulation.
    alphaTarget(0.3).restart();
  currentEvent.subject.fx = transform.invertX(
    currentEvent.subject.x);
  currentEvent.subject.fy = transform.invertY(
    currentEvent.subject.y);
}

function dragged() {
  currentEvent.subject.fx = transform.invertX(
    currentEvent.x);
  currentEvent.subject.fy = transform.invertY(
    currentEvent.y);
}

function dragended() {
  if (!currentEvent.active) simulation.
    alphaTarget(0);
  currentEvent.subject.fx = null;
  currentEvent.subject.fy = null;
}

</script>

<svg bind:this={svg} width='{width}' height='{height}'>
  <g class='axis y-axis'>
    {#each yTicks as tick}
      <g class='tick tick-{tick}' transform='translate(0,
        {yScale(tick)})'>
        <line x1={padding.left} x2={xScale(22)}/>
        <!--<text x={padding.left - 8} y='+4'>{tick}</text
        >-->
      </g>
    {/each}
  </g>

  <g class='axis x-axis'>
    {#each xTicks as tick}

```

```

    <g class='tick' transform='translate({xScale(tick)
      },0)''>
      <line y1={yScale(0)} y2={yScale(13)}/>
      <!--<text y={height - padding.bottom + 16}>{tick}</
        text-->
    </g>
  {/each}
</g>

<g class="links">
  {#each links as link}
    <line x1='{link.source.x}' y1='{link.source.y}'
      x2='{link.target.x}' y2='{link.target.y}'
      transform='translate({transform.x} {
        transform.y}) scale({transform.k} {
        transform.k})''>
    </line>
  {/each}
</g>

<g class="nodes">
  {#each nodes as point, i}
    <circle class="hover-node" on:click={fire_search(
      point.id)} r={nodeRadius} fill={fill_color(point,
      i)} cx={point.x} cy={point.y}
      transform='translate({transform.x} {transform.y})
      scale({transform.k} {transform.k})''></circle>
  {/each}
</g>

<g class="is-size-7 has-text-weight-semibold labels">
  {#each nodes as point}
    <text x={point.x + 15} y={point.y + 5} transform='
      translate({transform.x} {transform.y}) scale({
      transform.k} {transform.k})''>{point.label}</text>
  {/each}
</g>

</svg>

<style>
  .tick line {
    stroke: #ddd;
    stroke-dasharray: 2;
  }

  .links line {
    stroke: rgba(0, 0, 0, 0.2);
    stroke-width: 0.5;
  }

  .hover-node: hover{

```

```

        stroke: rgba(0, 0, 0, 0.1);
        stroke-width: 5px;
        cursor: pointer;
    }

</style>

```

Listing 31: Back-end: method *build_graph* and a piece of method *ws_most_similars* of class *Server*

```

async def ws_most_similar(self, request):
    logger.info('Websocket connection starting')
    ws = aiohttp.web.WebSocketResponse()
    await ws.prepare(request)
    logger.info('Websocket connection ready')

    async for msg in ws:
        logger.info("Message received %s" % msg.data)
        print(msg)
        if msg.type == aiohttp.WSMsgType.TEXT:
            try:

                [...]

                # Contains only terms with highest occurrences
                terms_pruned = df['bst_st'].to_numpy()
                # Like terms but sorted differently
                map_terms_pruned = np.asarray(df['search_term'])
                map_terms_pruned_idx = np.array([np.repeat(i, len(
                    map_terms_pruned[i])) for i in range(
                        map_terms_pruned.shape[0])])
                map_terms_pruned_idx = np.hstack(
                    map_terms_pruned_idx)
                map_terms_pruned = np.hstack(map_terms_pruned)

                terms_pruned = np.concatenate([terms_pruned,
                    st_miss, st_bad])
                map_terms_pruned = np.concatenate([map_terms_pruned,
                    st_miss, st_bad])
                map_terms_pruned_idx = np.concatenate([
                    map_terms_pruned_idx, np.arange(
                        map_terms_pruned_idx[-1]+1, terms_pruned.size)])

                wv_sum_occ = {}
                for i in range(map_terms_pruned.size):
                    if map_terms_pruned[i] == st:
                        wv_sum_occ['$root$'] = self.vocab[
                            map_terms_pruned[i]]
                wv_sum_occ[terms_pruned[map_terms_pruned_idx[i]]]
                    = wv_sum_occ[terms_pruned[map_terms_pruned_idx[
                        i]]] + self.vocab[map_terms_pruned[i]] if

```



```

        terms_pruned[map_terms_pruned_idx[i]] in
        wv_sum_occ else self.vocab[map_terms_pruned[i]]

df_nodes, df_links = self.build_graph(df, nr,
    terms_pruned, map_terms_pruned_idx,
    map_terms_pruned, st_miss)
df_nodes = df_nodes.to_json(orient='table', index=
    False)
df_links = df_links.to_json(orient='table', index=
    False)

await ws.send_str('{ "st_good" : %s, "st_miss" : %s,
    "st_bad" : %s, "wv_sum_occ" : %s, "wv_dist": %s
    , "nodes" : %s, "links" : %s}' % (df.to_json(
    orient='table', index=False), json.dumps(st_miss
    .tolist()), json.dumps(st_bad.tolist()), json.
    dumps(wv_sum_occ), json.dumps(wv_dist), df_nodes
    , df_links))
await ws.close()
logger.info('Websocket connection closed')
except Exception as e:
    exc_type, exc, tb = sys.exc_info()
    logger.error("%s - %s" % (e, '\n\n'.join(traceback.
        format_tb(tb, limit=5))))
    await ws.send_str('{ "error": "%s"}' % e)
    await ws.close()
    logger.info('Websocket connection closed')

return ws

def build_graph(self, df, nr, terms_pruned,
    map_terms_pruned_idx, map_terms_pruned, st_miss):
    df_tmp_links = pd.DataFrame([])
    df_links = pd.DataFrame(columns=['target', 'source'])
    df_nodes = df[['bst_st', 'wd_title']]
    df_nodes.columns = ['id', 'label']

    for similar in map_terms_pruned:
        child_similars = self.model.most_similar(positive =
            similar, topn = nr)
        child_np_sim, _ = zip( child_similars)
        # Lo svuoto ogni volta
        df_tmp_links = df_tmp_links.iloc[0:0]
        df_tmp_links['target'] = terms_pruned[
            map_terms_pruned_idx[np.in1d(map_terms_pruned,
            child_np_sim)]]
        idx = np.where(map_terms_pruned == similar)
        df_tmp_links['source'] = terms_pruned[
            map_terms_pruned_idx[idx[0][0]]]
        df_links = pd.concat([df_tmp_links, df_links])

    df_links = df_links.drop_duplicates()

```

```
df_miss_nodes = pd.DataFrame({'id' : st_miss, 'label'  
                             : st_miss})  
df_nodes = pd.concat([df_nodes, df_miss_nodes])  
  
found_bst = df_nodes["id"]  
df_links = df_links[df_links['source'] != df_links['  
    target']]  
df_links = df_links[(df_links['source'].isin(found_bst  
    )) & (df_links['target'].isin(found_bst))]  
  
return (df_nodes, df_links)
```

6

USE-CASE: FINANCIAL DATA PLATFORMS

In this chapter, we want to test FiSHeR to see how it performs in a real-world example. In particular, we want to use this application to find *information data platforms*. The financial data industry is dominated by four main applications:

1. Bloomberg Terminal
2. Eikon, by Refinitiv (ex Thomson Reuters)
3. Capital IQ by Standard & Poor's
4. FactSet

As stated in table 1, together they own 66,3% of the market share. Since the cost for a single user license ranges from \$12000 of FactSet up to \$24000 of Bloomberg Terminal per year [26], organizations can be interested in identifying alternative applications that could lead to huge cost savings. Therefore, we want to see how FiSHeR performs in this scenario.

Before going on, we must clarify that although being similar, these applications may provide quite different features. Some of them are:

1. Real-time market data
2. Equity research and trading
3. Financial news and insights
4. Investment analytics

We consider a result as correct if it supplies one or more of the functionalities above. Of course, the word embedding model chosen for this use-case is `m5_s300_cb`, since it reaches the highest accuracy between the models trained (section 4.5.1).

Searching one of the platforms above into FiSHeR, we expect either to retrieve the name of the other most common platforms, but also to find the names of less-known software that could be used as alternatives. We want to start looking at the number of occurrences of the former names into the collected Reddit dataset.

As we can see from table 17, terms of interest can appear multiple times in the dataset with small differences. For this experimentation, we are going to search into FiSHeR either *bloomberg_terminal*

term	occurrences in Reddit dataset
bloomberg_terminal	2690
bloomberg_terminals	461
eikon	257
factset	1026
capital_iq	167
capiq	314

Table 17: Occurrences for most known financial data platforms

and *eikon*, which have different orders of magnitude in the number of occurrences.

6.1 SEARCHING: BLOOMBERG_TERMINAL

In this first attempt, we want the application to return the first 20 terms that are similar to *bloomberg_terminal*. Among these returned terms, there will be also unresolved and discarded terms. This time however, we will focus only on matched entities, that will probably be less than 20, but are shipped by FiSHeR with additional information, like description and STRONG similarity score.

WD title	terms	sum of occurrences	strong similarity
Bloomberg L.P.	bloomberg	15357	81%
Eikon	eikon, reuters_eikon	296	77%
FactSet	factset	1026	72%
S&P Capital IQ	capiq, capital_iq, cap_iq, capitaliq	596	71%
Value Line	valueline	123	69%
Wharton Research Data Services	wrds	79	65%
Yahoo! Finance	yahoo_finance	7657	64%
DataStream	datastream	70	62%
TradingView	tradingview	4688	57%
Quandl	quandl	1000	48%

Table 18: Matched FiSHeR results searching *bloomberg_terminal*

Table 18 contains the matched results. The first four rows refer to the vendor company of Bloomberg Terminal (which is Bloomberg L.P.) and to the other platforms that we already know. This is a good result since it confirms our previous knowledge acquired analyzing

the market shares of such products [26].

Let's search the remaining terms online to get descriptions to understand if they are of any usefulness. FiSHeR already provides descriptions and quick links for the matched term, in order to facilitate the user in this additional analysis step.

1. Value Line: *Value Line [...] (provides) accurate and insightful investment research on companies, industries, markets and economies. From the latest data, sophisticated tools and proven ranks to expert analysis and guidance.*¹
2. Wharton Research Data Services: *WRDS provides the leading business intelligence, data analytics, and research platform to global institutions*².
3. Yahoo! Finance: *It provides financial news, data and commentary including stock quotes, press releases, financial reports, and original content.*³
4. Datastream: *It is an historical financial database*⁴. As of today, Datastream is also accessible from Eikon itself. Hence this result does not provide any additional information to our previous knowledge.
5. TradingView: *TradingView is a social network for traders and investors on Stock, Futures and Forex markets!*⁵
6. Quandl: *The premier source for financial, economic, and alternative datasets, serving investment professionals*⁶

The results with higher STRONG similarity represent platforms that provide some of the functionalities described at the beginning of this chapter. The ones with lower similarity, like Datastream, TradingView and Quandl refer to databases or financial social networks that are, therefore, different kinds of platforms compared to the ones we are looking for. This proves the goodness of this similarity score, as a metric to have a preliminary comparison of the output results.

6.2 SEARCHING: EIKON

Repeating the same experimentation searching *eikon* we get the results printed in table 19.

¹ <https://www.valueline.com/about/aboutvalueline.aspx>

² <https://wrds-www.wharton.upenn.edu/pages/about/>

³ https://en.wikipedia.org/wiki/Yahoo!_Finance

⁴ <https://www.refinitiv.com/en/products/datastream-macroeconomic-analysis>

⁵ Meta description tag of <https://www.tradingview.com>

⁶ <https://www.quandl.com/>

WD title	terms	sum of occurrences	strong similarity
Bloomberg Terminal	bloomberg_terminal	2690	77%
FactSet	factset	1026	77%
MetaStock	metastock	41	76%
DataStream	datastream	70	75%
Intrinio	intrinio	108	70%
S&P Capital IQ	capiq, capital_iq, cap_iq, capitaliq, p_capital_iq	683	70%
Bloomberg L.P.	bloomberg	15357	67%
Thomson Reuters	thompson_reuters	77	52%

Table 19: Matched FiSHeR results searching *eikon*

Among them, we can still find the name of all the well-known applications previously described. At the bottom of the table, we can find *Thomson Reuters* (now Refinitiv) and *Bloomberg L.P.* which are the vendors respectively of Eikon and Bloomberg Terminal. In the middle of the table we can find two additional terms, *MetaStock* and *Intrinio*, that we want to further analyze along the lines of what was done in the previous section:

1. MetaStock: Market Analysis Tools, in some cases powered by Eikon product Xenith⁷. Similarly to Datastream, this result does not add any valuable information.
2. Intrinio: *Real-time, delayed, intraday, and historical market data for US stocks, ETFs, and options*⁸

6.3 FINAL CONSIDERATIONS

In this chapter, we have shown a real example of the usage of FiSHeR. As we have seen, it outputs results that are consistent with the well-known market players. We also found out some additional terms that could prove to be useful alternatives. However, to fully exploit the capabilities of FiSHeR, we would also have to increase the number of results returned by the platform and take into account unresolved terms. These, indeed, are names that are not in Wikidata but could still be relevant for our purposes.

Due to the fragmentation of the services offered by these applications, the competitors found will probably provide only a partial

⁷ <https://www.metastock.com/>

⁸ <https://intrinio.com/financial-market-data>

overlap in terms of functionality. Therefore, any meaningful comparison of the results returned by FiSHeR, must be out by business experts that deeply know the needs of the final users.

7

CONCLUSIONS AND FUTURE WORKS

In this thesis we have shown that machine learning is a viable solution to find new financial services. Although not being perfect, this project leveraged a huge quantity of data, extracting information that could be difficult to find otherwise. The output of this experimentation is a search engine that enhances the human capability of discovering less-known products.

Most of the problems faced referred to the difficulty of distinguishing useful results from useless ones. Indeed, among the terms returned by the word embedding model, there may be generic or misspelt terms that are not worth being displayed to the user. This problem was addressed matching information against Wikidata that, however, does not provide an adequate classification of items for our purposes.

We could therefore hypothesize the creation of a new Knowledge Graph for financial services in which items are categorized with an ad-hoc schema. This schema should be specific for financial products and standardized among financial institutions. Moreover, it should provide a finite number of classes, such that it would be possible to define a complete list of useful classes.

If we assume this Knowledge Graph to be interlinked with (a portion of) Wikidata, FiSheR could either:

1. Exploit additional classification for entities that are in both databases, to better resolve them
2. Provide wizards to populate the Knowledge Graph migrating information retrieved by Wikidata, for entities matched only in this latter database
3. Provide wizards to allow the creation of new entities from scratch, starting from the unresolved terms of interest

Of course, these migrations wizards should be supervised by the human, that is in charge of carefully choosing the relevant classification items for each new entity, from this domain-specific schema.

In this way, we could create an ecosystem of products in which FiSheR is both a consumer and a producer of information.

ACKNOWLEDGMENTS (IN ITALIAN)

In primis, voglio ringraziare il mio relatore prof. Ioannis Chatzi-giannakis ed il mio correlatore prof. Aris Anagnostopoulos, per avermi concesso d'intraprendere questa sperimentazione come mia personale tesi, avendo voluto dare fiducia ad un progetto le cui premesse iniziali di riuscita sembravano incerte.

Voglio poi ringraziare l'ing. Dario Russo, Capo del Servizio Regolamento operazioni finanziarie e pagamenti della Banca d'Italia, fautore dell'idea alla base di questa ricerca che mi ha garantito la possibilità di occuparmene in prima persona. Un grazie va anche al dott. Gianluca Mura, che con dedizione si è occupato della valutazione manuale dei risultati ottenuti.

È mio desiderio riservare un ringraziamento tutto particolare al dott. Enrico Levrini, a cui mi lega un'amicizia sincera. I suoi consigli sono stati preziosissimi in questo nuovo ambiente di lavoro di cui ora faccio parte.

Devo poi ringraziare i miei genitori, mio fratello, mia sorella ed i miei nipotini. Tutti insieme hanno coltivato il concetto di famiglia che sempre mi accompagna nelle scelte quotidiane. Sono stati essenziali per affrontare le continue prove che questa tesi mi ha posto davanti. Hanno regalato gioia lì dove c'era tristezza, hanno regalato speranza lì dove c'era incertezza.

Per ultima, ma di certo non per importanza, voglio dire grazie a Chiara, la mia ragazza. La cui infinita pazienza non mi ha mai fatto pesare nulla in questi mesi di costante impegno. Nelle giornate peggiori, ho sempre potuto contare sul suo sostegno e sul suo conforto. Grazie, nella maniera più sincera di cui sono capace.

BIBLIOGRAPHY

- [1] Accenture. “Next Generation Digital Procurement”. In: (2017). URL: https://www.accenture.com/t20171023T071231Z__w_/us-en/_acnmedia/PDF-63/Accenture-Next-Generation-Digital-Procurement-POV.pdf (cit. on p. 4).
- [2] Alan Akbik et al. “FLAIR: An Easy-to-Use Framework for State-of-the-Art NLP”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 54–59. DOI: [10.18653/v1/N19-4010](https://doi.org/10.18653/v1/N19-4010). URL: <https://www.aclweb.org/anthology/N19-4010> (cit. on p. 23).
- [3] Dogu Araci. *FinBERT: Financial Sentiment Analysis with Pre-trained Language Models*. 2019. arXiv: [1908.10063](https://arxiv.org/abs/1908.10063) [cs.CL] (cit. on p. 7).
- [4] The TensorFlow Hub Authors. *Pearson correlation coefficient of the Universal Sentence Encoder*. URL: https://www.tensorflow.org/hub/tutorials/semantic_similarity_with_tf_hub_universal_encoder (cit. on p. 66).
- [5] The TensorFlow Hub Authors. *Pearson correlation coefficient of the Universal Sentence Encoder lite*. URL: https://www.tensorflow.org/hub/tutorials/semantic_similarity_with_tf_hub_universal_encoder_lite (cit. on p. 66).
- [6] Jason Baumgartner et al. “The Pushshift Reddit Dataset”. In: 14 (2020), pp. 830–839. URL: <https://www.aaai.org/ojs/index.php/ICWSM/article/view/7347> (cit. on pp. 9, 10).
- [7] Christian Bizer et al. “DBpedia - A crystallization point for the Web of Data”. In: *Journal of Web Semantics* 7.3 (2009). The Web of Data, pp. 154 –165. ISSN: 1570-8268. DOI: <https://doi.org/10.1016/j.websem.2009.07.002>. URL: <http://www.sciencedirect.com/science/article/pii/S1570826809000225> (cit. on p. 38).
- [8] Piotr Bojanowski et al. “Enriching Word Vectors with Subword Information”. In: *CoRR* abs/1607.04606 (2016). arXiv: [1607.04606](https://arxiv.org/abs/1607.04606). URL: <http://arxiv.org/abs/1607.04606> (cit. on p. 27).
- [9] Daniel Cer et al. “Universal Sentence Encoder”. In: *CoRR* () (cit. on p. 65).
- [10] Adam Cohen. “FuzzyWuzzy: Fuzzy string matching in python”. In: *ChairNerd Blog* 22 (2011). URL: <https://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python> (cit. on p. 51).

- [11] Michael Färber and Achim Rettinger. “Which Knowledge Graph Is Best for Me?” In: *CoRR* abs/1809.11099 (2018). arXiv: 1809.11099. URL: <http://arxiv.org/abs/1809.11099> (cit. on p. 38).
- [12] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. <http://www.rfc-editor.org/rfc/rfc6455.txt>. RFC Editor, 2011. URL: <http://www.rfc-editor.org/rfc/rfc6455.txt> (cit. on p. 88).
- [13] Inc. Gartner. *Prepare for the Impact of AI on Procurement*. 2017. URL: <https://www.gartner.com/smarterwithgartner/prepare-for-the-impact-of-ai-on-procurement/> (cit. on p. 4).
- [14] Nicolas Heist et al. *Knowledge Graphs on the Web – an Overview*. 2020. arXiv: 2003.00719 [cs.AI] (cit. on p. 38).
- [15] Cathal Horan. *When not to choose the best NLP model*. <https://blog.floydhub.com/when-the-best-nlp-model-is-not-the-best-choice/>. 2019 (cit. on p. 65).
- [16] Paul Jaccard. “Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines”. In: *Bulletin de la Société Vaudoise des Sciences Naturelles* 37 (1901), pp. 241–272 (cit. on p. 63).
- [17] Armand Joulin et al. “Bag of Tricks for Efficient Text Classification”. In: *CoRR* abs/1607.01759 (2016). arXiv: 1607.01759. URL: <http://arxiv.org/abs/1607.01759> (cit. on p. 27).
- [18] Markus Lanthaler, Richard Cyganiak, and David Wood. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. W3C, Feb. 2014. URL: <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/> (cit. on p. 38).
- [19] Vladimir I Levenshtein. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710 (cit. on p. 45).
- [20] Tomas Mikolov et al. “Distributed Representations of Words and Phrases and their Compositionality”. In: *CoRR* abs/1310.4546 (2013). arXiv: 1310.4546. URL: <http://arxiv.org/abs/1310.4546> (cit. on p. 27).
- [21] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013. URL: <http://arxiv.org/abs/1301.3781> (cit. on pp. 27, 31).
- [22] Evan Miller. *How Not To Sort By Average Rating*. 2016. URL: <http://www.evanmiller.org/how-not-to-sort-by-average-rating.html> (cit. on p. 14).

- [23] Alberto Parravicini et al. “Fast and Accurate Entity Linking via Graph Embedding”. In: *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA)*. GRADES-NDA’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. ISBN: 9781450367899. DOI: [10.1145/3327964.3328499](https://doi.org/10.1145/3327964.3328499). URL: <https://doi.org/10.1145/3327964.3328499> (cit. on p. 43).
- [24] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. “YAGO 4: A Reason-able Knowledge Base”. In: *The Semantic Web - 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31-June 4, 2020, Proceedings*. Vol. 12123. Lecture Notes in Computer Science. Springer, 2020, pp. 583–596. DOI: [10.1007/978-3-030-49461-2_34](https://doi.org/10.1007/978-3-030-49461-2_34). URL: https://doi.org/10.1007/978-3-030-49461-2_34 (cit. on p. 38).
- [25] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation”. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: <http://www.aclweb.org/anthology/D14-1162> (cit. on p. 27).
- [26] Wall Street Prep. *Bloomberg vs. Capital IQ vs. FactSet vs. Thomson Reuters Eikon*. 2018. URL: <https://www.wallstreetprep.com/knowledge/bloomberg-vs-capital-iq-vs-factset-vs-thomson-reuters-eikon/> (cit. on pp. 3, 111, 113).
- [27] Mohit Rathore. *Comparison of FastText and Word2Vec*. URL: [https://markroxor.github.io/gensim/static/notebooks/Word2Vec_FastText_Comparison.html#:~:text=Trainingtimesforgensimare,muchoftheheavylifting\)](https://markroxor.github.io/gensim/static/notebooks/Word2Vec_FastText_Comparison.html#:~:text=Trainingtimesforgensimare,muchoftheheavylifting)). (cit. on p. 27).
- [28] Dario Russo and Gianluca Mura. “An Empirical Taxonomy of Financial Data Services”. In: (2020 - in course of publication) (cit. on p. 3).
- [29] Wataru Souma, Irena Vodenska, and Hideaki Aoyama. “Enhanced news sentiment analysis using deep learning methods”. In: *Journal of Computational Social Science* 2 (Feb. 2019). DOI: [10.1007/s42001-019-00035-x](https://doi.org/10.1007/s42001-019-00035-x) (cit. on p. 7).
- [30] Ken Thompson. “Programming Techniques: Regular Expression Search Algorithm”. In: *Commun. ACM* 11.6 (June 1968), 419–422. ISSN: 0001-0782. DOI: [10.1145/363347.363387](https://doi.org/10.1145/363347.363387). URL: <https://doi.org/10.1145/363347.363387> (cit. on p. 19).
- [31] Matthias W. Uhl. “Reuters Sentiment and Stock Returns”. In: *Journal of Behavioral Finance* 15.4 (2014), pp. 287–298. DOI: [10.1080/15427560.2014.967852](https://doi.org/10.1080/15427560.2014.967852). eprint: <https://doi.org/10.1080/15427560.2014.967852>. URL: <https://doi.org/10.1080/15427560.2014.967852> (cit. on p. 7).