**Hubsites Project**

**Goal:**

The hubsites project is aimed at finding companies’ executive teams, starting from company seeds.

The result will be used in core mainpath projects (such as execs).

**Tools Used:**

First of all, the tools used by the project are:

1. Python
2. Mongo (Heroku)
3. RabbitMQ (Heroku)

And tools used for validation and performance are:

1. RapidMiner (for assessing weights)
2. Jupyter (formerly called iPython)

*(Note: Credentials and user-accounts are supplied at the end of the document).*

*(Note2: Project code is available both on Heroku (credentials supplied at the end of the document, and on “Stash”, under “personal projects” (“hubsites” & “corporateir\_and\_sec”).*

**Process Overview:**

The algorithm is split into several stages, separated by RabbitMQ queues (which allows for scalability). The stages are:

*Stage 0 (distribution)*: Turning seeds saved on a MongoDB collection called into objects in a rabbitMQ queue (so that workers can process them concurrently).

*Stage 1*: Taking messages from the described queue and processing them i*n order to find a companys’ executive page*. (The message consists of information such as company name, company websites, a few of the known executives’ names, etc.)

At the end of this stage, the results are saved in a second mongo collection.

*Stage 1.5 (distribution)*: Populating the second RabbitMQ queue, just taking information out of a Mongo collection, turning it into python objects and populating the queue for the second stage.

*Stage 2*: Taking messages from the second rabbitMQ queue and processing them *in order to find a companys’ leadership/executive team*. The input (rabbitMQ message) already contains the executive page (found in stage 1) and this stage revolves around finding patterns in the pages’ HTML, determining which patterns target the executives and retrieving them.

At the end of this stage, the results are saved in a third mongo collection.

Company\_F1000\_forExecPages

ExecPagesFromYahoo

Mongo

NewContacts

Mongo

ExecPageSearch

RabbitMQ

Hubsites

RabbitMQ

rbmq\_msg\_distributer

exec\_pages

From Yahoo search

rbmq\_msg\_distributer

XPathLearner

**Useful Commands:**

* installation stage:
* Attaching to the git repo: git remote add heroku git@heroku.com:hubsites.git
* Running the process:

heroku run:detached python rbmq\_msg\_distributer.py Company\_F1000\_forExecPages ExecPageSearch 30000 0 company,titles,Names,url

heroku ps:scale yahoo\_exec\_pages=100

heroku run:detached python rbmq\_msg\_distributer.py ExecPagesFromYahoo HubSites 30000 0 company,titles,is\_correct,Names,url,yahoo\_score

heroku ps:scale XPLearner=100

These commands demonstarte how to run the 2-step process on heroku.

A nice point to note is that after running the “rbmq\_msg\_distributer” commands, we can run the code in our own dev environment and dequeue rabbitMQ (whether you run the “ps:scale” function or not - which starts the heroku workers), we can debug, make changes, etc..

In this scenario, you behave as any other worker working on the queue.

**Detailed Description:**

* **Notable Python Classes**

rbmq\_msg\_distributer.py: Recieves a mongo collection and a queue name and simply populates the queue (using *json.dumps* and *json.loads*), needs to be activated before each of the 2 steps of the hubsites algorithm.

Each of the two stages of the hubsites algorithm works by consuming a full rabbit queue.

exec\_page.py: The basic model that’s moved between queues and mongo-collections. Contains data fields and basic methods to compute feature scores, process urls, find important words within its’ fields, etc..

exec\_pages\_from\_yahoo.py: This file contains the code that handles the ***first*** stage of the hubsites algorithm (Transforming seeds (i.e company names & websites) into possible executive page urls).

1. The file starts with some preprocessing/cleaning of the company name and urls and deduping (by turning a list into a set, and back again).

2. After that, there are a few search steps:

2.1 The company name is searched in Yahoo, and the domains are extracted from the results and saved to a variable called “company\_name\_domains”.

**WHY??**

Later in the algorithm, we’ll want to check domains we received in the results and their yahoo ranks. If we’re checking a result that has a domain that’s different than the domain of the website of the company we’re currently checking, than we might chuck out the result.

We’ll review the result only if the current domain comes **before** the first appearance of the current company domain.

Let’s try an example: We’re looking for Oracle execs, but the first result with an oracle.com domain is only the 4th result. This means we’ll check the 1st, 2nd and 3rd results even if the domain is not oracle.com, but ONLY if these sites contain pharses with the word “oracle” in them (such as “Oracle executive team”).

This is very important, since sometimes Yahoo attempts to search for sites while using only parts of our original search term, thus, a search term such as “Oracle executive team” can be reduced to “executive team” and we have to make sure to filter that out.

This is especially important for cases when companies get acquired by bigger companies, and their executive page might move to a different domain.

2.2. Patterns are constructed using the company name and search terms (such as “executive team” or “executive officers”), and these patterns are used in Yahoo searchs (later, when we’ll save the results to MongoDB, we’ll also save the pattern itself, for grouping purposes).

Also, only the first 6 results are fetched per search term. That’s because later results have proven to not be as useful.

3. Each result is fetched from the web (inside the method *“operate”* in the class “exec\_page”), and then scored (inside the method *“scoring”* in the same class). The scoring method uses weights that have been produced using RapidMiner (using a perceptron, meaning linear weights). Also, there is a fixed number which we add to the score (also, generated using RapidMiner), and this is used to calibrate the score to start from zero.

The features themselves are set in the *“setfeatures”* method in the “exec\_pages” class.

4. Next, we fetch pages that Turk users have pointed out to be the executive pages (sup\_url1 and sup\_url2, which is short for supervised\_url) and save them to the data-frame which will be saved to Mongo later.

5. Lastly, we save the result to Mongo, which means we iterate over the data-frame in our exec\_page object and add that to the bulk-insert object for saving. During this process, we also check and validate stuff, such as the domain checking explained in step 2.

XPathLearner.py: The XPathLearner is the 2nd part of the hubsites algorithm - it is the part that given an exec-page, finds the execs and stores them in Mongo. It does this by trying out many different XPaths in the pages’ html and choosing the one that seems to capture the largest amount of people.

It is important to remember, that the input for this class is not just the exec-page, but also the company name, and several execs that we already know of (at least two). The algorithm checks for XPaths that capture the **known** execs/titles, as well as other elements. It will then check that the additional elemets that were captured are in fact names and/or titles (using knowledge bases of names, titles, etc..)

The class uses a library called “lxml” which is an xml-parsing library (which also parses html, of course) and turns the dom into a tree.

The method “LearnXPath” creates an “Hypothesis Table” which is basically a data frame containing all the possible XPaths that can be found (using backtracking - this is done in the helper method “GetXPathComb”).

We search separately for paths leading to *“personname”* elements, and for *“title”* elements - then we check every *personname* XPath against every *title* XPath and check if they contain the same number of captured elements, also we check for a common XPath prefix (if a mutual prefix is found, it raises the couples’ score).

(this is a little like a map-reduce job, where we query everything about ObjectA, and then query everything about ObjectB, and then search for interesting information concerning the two).

After that, img and bio information are also extracted, and the whole thing is saved to Mongo.

ner -> The NER scoring is an important feature that should be described.. NER stands for “Named Entity Recognition” and bascially means that we attempt to classify text elements into pre-defined categories (such as first names, titles, etc). The NER scoring uses knowledge bases that are loaded onto the memory which we check against. Each name in the names knowledge-base also has a confidence score, related to how common the name is (between 0 and 1, so a name like “John” will get a higher score than a name like “Wei”).

The ner score is added (after normalization) to the final score.

Last thing, we check for long titles, and if more than a third of the found titles are longer than 70 characters, we might suspect that something is wrong with our extraction process, and so, currently, if this condition holds - we chuck out the results.

Note: Some parts of the code check that only one company exists per search, this is of historical reasons, since in the past the code supported searching for multiple companies, but that has been deprecated.

Enteties folder: Contains knowledge-bases in CSV form that are used in various parts of the projects.

Some additional notes:

1. Currently, the final\_score/yahoo\_score, calculated in the first stage of the algorithm is somewhat atrophied, since all pages are saved to mongo and fed to the second stage. The reason for this is that the 2nd stage is able to filter false pages rather well, and so precision isn’t affected (while recall may be slightly elevated).

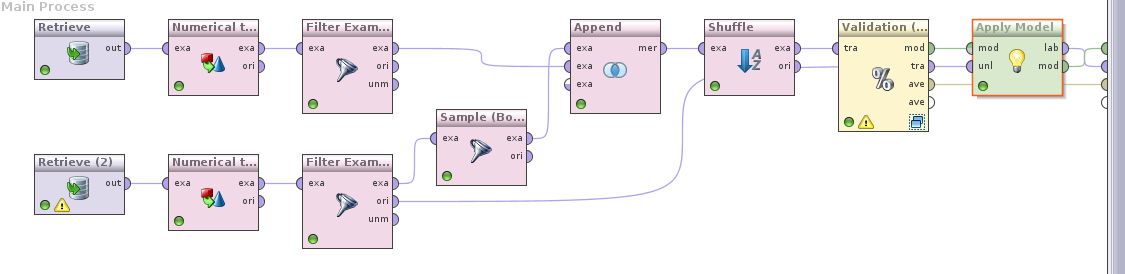
Also, the scores serve as a tool to check performance with.

1. more…?

* **RapidMiner Work:**

Determinig weights for 1st stage features: Scoring is a part of the 1st stage of the hubsites algorithm, it is computed using weights that are assigned to particular features generated by the algorithm.

The weights are determind using a RapidMiner process called “hubsites.rmp” (**TODO:** SAVE IT ON GOOGLE DOCS OR SOME OTHER PUBLIC LOCATION).



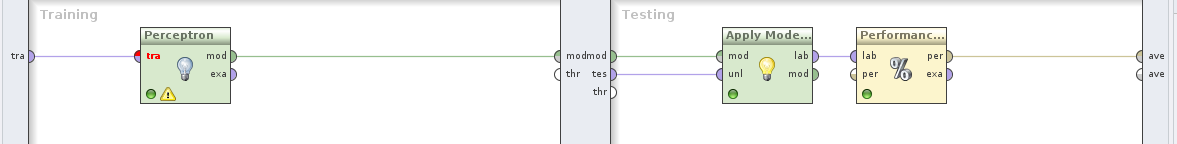
Flow:

1. Load data: The project loads CSV data (has to be **supervised** data) - It loads it twice, and each time it filters it differently: One time it keeps all the *correct* cases (according to Turk) and the other time it keeps all the *wrong* cases in the CSV. The reason for this is that our current dataset has about 5 times as many *wrong* examples as it does *correct* ones, and since RapidMiner wishes to get as many correct answers as possible (true positives or true negatives) we usually end up with a result that puts more emphasis on recall, rather than precision.

That is why we go through step 4.

1. Number to Binomial: Pretty trivial/technical, one of the fields is an integer (with 1’s and 0’s) and we simply cast it as a boolean/binomial.
2. Filter: As explained before, one time we filter correct answers, the other time, false answers.
3. Multiplying the sample (Bootstrapping): We multiply one of the samples in order to have a ration that’s close to 50%-50% between *correct* and *false* answers.
4. Append: Unioning the two samples together.
5. Shuffle: Shuffling is important since learning systems need random data in order to perform at their best.
6. Validation: The validation stage is comprised of two stages which run several times while changing the data used for training and testsing (i.e cross-validation).

The shuffled data is used as input and a model is created (meaning, the perceptron calculates weights to be used). That model is used on the test data and the result is moved to a performance operator to that outputs a confusion matrix.



1. Apply Model: Another apply model, not really sure why… :|

Guess the validation operator didn’t output exactly what we wanted

and apply model tweaks it.

**Note**: No such RapidMiner project exists which tests the recall/precision of the entire hubsites algorithm, only this (which tests only one of two stages). Measuring performance for the entire algorithm is done using Turk as well as manually.

* **iPython/Jupyter Work:**

iPython is an interactive web-based system that allows its’ user to run scripts interactivaly. Alon left behind a bunch of scripts we he used to normalize and handle data “on-the-fly” as well as display data in plots/graphs.

The scripts still need to be reviewed in order to assess what can be used.

Note**:** The scripts are saved in a folder in Google Drive, under the link:

<https://drive.google.com/a/salesforce.com/folderview?id=0B4X_zatDRoB-fkZIay1OY1JkYjgtbkVXdTl1dE1jZVh1cFJrYndySXBhRS1OQUVXUllmSGs&usp=sharing>

**Additional/Related Projects (also in Heroku & Stash):**

1. Yahoo Predicates Project: A project revolving around using numerous “search patterns”, such as “{NAME} is the ceo of”, or “the ceo of {COMPANY}” and extraction of the results.

**TODO**: Expand on this subject..

1. SEC Filings: The Securities and Exchange Committee, which is simiar to the Israeli Rasham HaChavarot (“רשם החברות”) is a formal organization to which companies serve certain forms. One of which is the 14A form, which is publicly available online, and contains executive information.

This project deals with mining that information.

1. CorporateIR: Nasdaq offers a unique service for companies who want to have an executive page, without the hassle of having to build one themselves.

The service allows them to build a webpage on the CorporateIR platform, which has a

pre-determined html structure (that makes it very easy to crawl).

The project finds all companies that have such a webpage, and extracts the information.

Also, a document with helpful “tips & tricks” is located in:

<https://docs.google.com/a/salesforce.com/document/d/1xLdcHkL081hb7l-MRqyjyWBjuNs9JyvX3Vp-KiBI7w0/edit?usp=sharing>

**Mongo Collections:**

* **“hubsites” collections**
* Company\_F1000\_forExecPages: SUPERVISED input collection for the 1st stage of the hubsites project. Contains information for 1000 companies, where each company has an exact company name, website, and a supervised url (of their executive page) supplied by Turk.
* jigsaw\_sample\_with\_website\_1000: SUPERVISED input collection - similar to previously described collection.
* ext\_supervised\_1000\_execpages\_23\_8: SUPERVISED input collection - similar to previously described collections.
* companiesWithWebsite\_100K\_Clean: UNSUPERVISED input - apart from that, similar to the previously described collections.
* company\_domain\_850K\_Clean: SUPERVISED input collection for 850K companies, simiar to the previously described collection.
* Company\_F1000\_forHubsites: (Start only from 2nd stage!)
* SearchResults\_Cache: A yahoo cache in Mongo so that the tests don’t waste as much money… Documents in this collection also include a timestamp so that we can override/ignore data that is older than X.
* **“Sec Filings” collections**
* SEC\_DEF14A: The input collection of the SEC project. The collection was used using a site called “edgar” which crawls the US governments’ site and allows you to download a dump of their data.   
  Additional information can be found in other files/collections, such as (SEC\_Company\_info, SEC\_Company\_locations, etc..)   
  Additional information can include subsidiary companies, locations, and more.
* SEC\_Executives: The output collection of the SEC project. Contains contact/execs.
* **“CorporateIR” collections**
* CorpIRExecPages: The input collection for the CorporateIR project. Basically, a list of urls, each directing to a Corporate-IR page. The urls always have a similar prefix, with some id. (It was created by running a loop that tried to the similar urls (with only the id changing between requests) and checking which requests yielded results.

* corporate-ir-contacts: The output collection for the CorporateIR project. Contains contacts/execs.
* **“Predicates” collections**
* TODO: List relevant collections.

**Credentials:**

* Heroku account:
* username: [atalmor@salesforce.com](mailto:atalmor@salesforce.com)
* password: Salont12
* project: hubsites
* Also, the project uses 3 plugings:
  + Logentries (Free)
  + Compose MongoDB (Free)
  + RabbitMQ Bigwig (20$ / month)
* MongoDB:
* Address: dogen.mongohq.com
* Port: 10042
* Username: heroku
* Password: a4H\_qs7Dt9RAPt1fGq71efe3-u1DakgvypoRvpXanflyfAgxgu7hOu5fTQO5l7yUS8546FpbTLLDp1RjF-mO3Q
* Database: app32007554
* RabbitMQ:
* connection-string: amqp://OCOkInZs:Ru-ZUupM8O24eoJauvROkeFcX2ivL-Pv@scared-silverweed-57.bigwig.lshift.net:10085/HGV2EcJFOW4e

(consists of: user=OCOkInZs, pass=Ru-ZUupM8O24eoJauvROkeFcX2ivL-Pv,

address/host=scared-silverweed-57.bigwig.lshift.net, port=10085,

db=HGV2EcJFOW4e )

* YahooBOSS:
* key: dj0yJmk9Nzk2OXZrZU55NG9yJmQ9WVdrOU1rMDVRbU51TlRBbWNHbzlNVE01TXpVeE9UTTJNZy0tJnM9Y29uc3VtZXJzZWNyZXQmeD1hNQ--
* secret: 2852b12d9ecb9f06203c32a30441208ebd7caeca
* Logentries (heroku plugin):
* token: 37e45677-c2b1-4611-9458-6517b57790ff

**Knowledge that still needs to be passed (w/ Alon):**

1. The process of assessing the recall/precision of the entire hubsites algorithm (specifically 2nd stage). Nowdays, it is done manually and with Turk, but a precise explanation was never given.
2. Go over the Predicate project (in-depth).