Deposit Identification Utility and Visualization Tool

Colorado School of Mines
Field Session Summer 2014

David Alexander
Jeremy Kerr
Luke McPherson

Introduction

Newmont Mining Corporation was founded in 1921. It is one of the global leaders in gold and copper production and is a Fortune 500 company. Headquartered near Denver, Newmont has a goal to become the most valued and respected mining company through its industry-leading performance. The company would like to investigate the process of data mining as a means to classify websites describing various geological deposit types in order to expedite the process of evaluating projects of interest.

Text mining is the process of deriving high quality information from text. It is used to study word frequency, perform pattern recognition, extract information, and create data for visualizations. The goal of text mining is to turn text into relevant data for analysis. Text comparison analysis is a broad field which involves determining similarity between one set of documents and another document by creating a model from the set of documents, and then comparing that model to another document to determine how related to the model the document may be.

The task given for this field session project was to explore several data mining and visualization toolchains in order to construct a web application to assist in identifying projects of interest. The goal was to create an application capable of determining which potentially valuable projects Newmont competitors are working on by exploring their websites and comparing their mining operations to models produced based on the mineral deposits Newmont is interested in. Similarity between the models and websites would be determined by comparing the text content of the website to the models, where similarity is good if sites contain many words included in a model, and similarity is considered excellent if sites also use words in the same context as the model. Context is determined through the construction of topics, where topics are composed of the words that appear frequently together in a given model.

Requirements

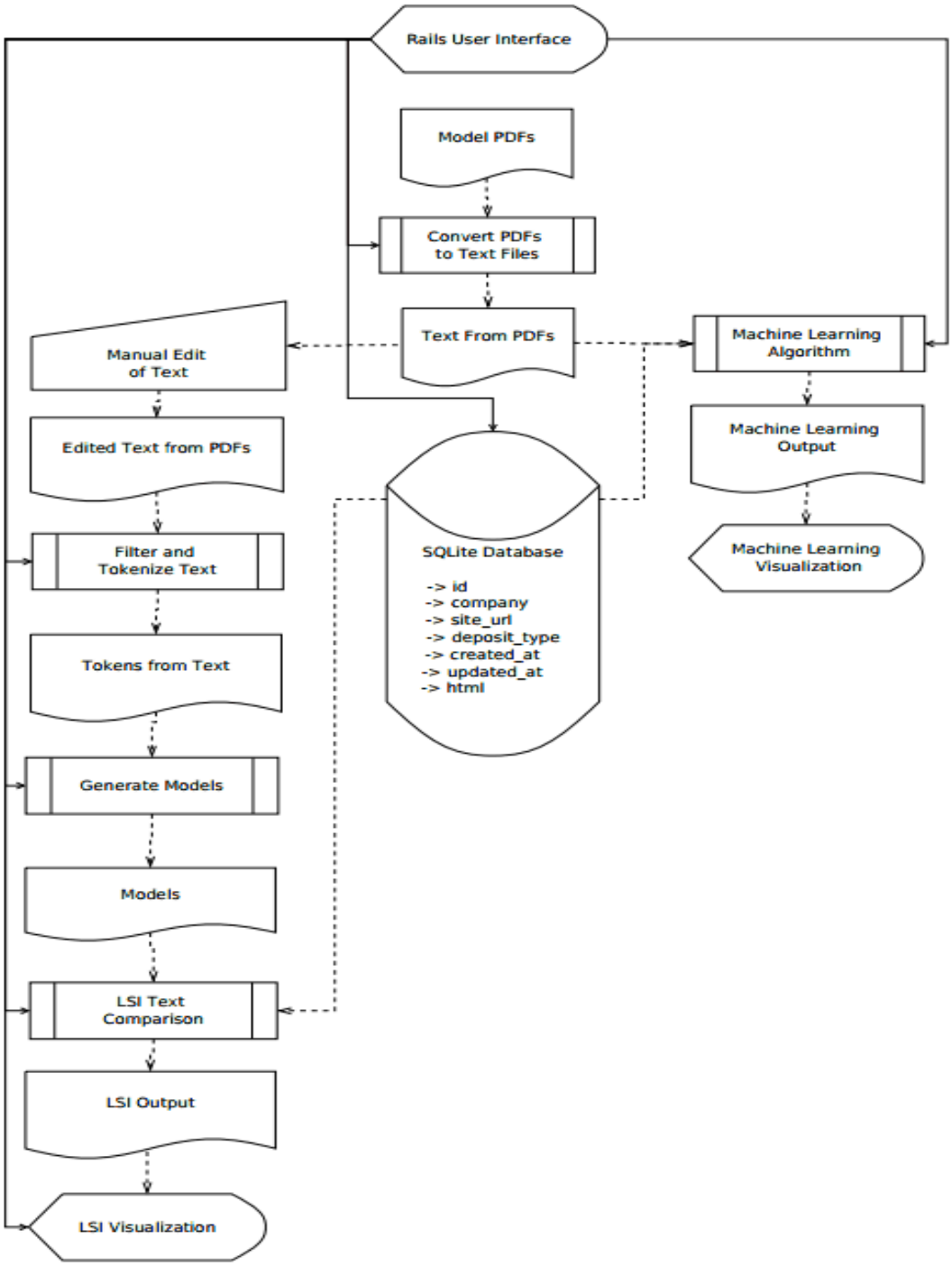
Functional Requirements

- Read in multiple sets of PDF documents and pull out the text for data extraction
- Filter each document derived from the PDFs to remove fractured words and all words that are not nouns, verbs, adjectives, and adverbs
- Generate an associated model for each set of documents derived from the original PDFs
- Allow multiple models to be stored and compared against in the application
- Utilize a web scraper to obtain the text stored on selected websites for later comparison to the models
- Filter the text from each site in the same manner models were filtered
- Compare each model to each site using a latent semantic indexing algorithm and produce a guess for which model each site is most similar to
- Present the output from the latent semantic indexing algorithm in an interactive visual format using the d3.js library
- Train a machine learning algorithm on a set of well defined sample sites with known deposit types
- Test the performance of the machine learning algorithm on a set of sample sites with known deposit types to determine its accuracy in determining the correct site
- Present the information from the machine learning algorithm in an interactive visual format using the d3.js library

Non-functional requirements

- Must function as a web application using Ruby on Rails
- The code implementing functionality must be written in Python, Ruby, and d3.js
- All code must be well commented and include relevant documentation for continued development and suggestions for where to go to improve existing results
- Application must be formatted properly when used on Chrome browser
- Application must run on an Amazon EC2 Ubuntu server
- Code should be developed using a github repository for version control

System Architecture



The application was designed to run on an Amazon EC2 server on the Ubuntu 12.04 OS. It can be accessed online whenever the Ruby on Rails server is launched, however our client requested that rather than permanently launching the application we simply document how to start and stop the Rails server.

The diagram on the previous page shows the organization of our tools and additionally serves as a documentation of dependencies. Dashed lines represent dependencies and output. Solid lines indicate the tools and visualization that can be called from the Rails interface. Rectangles within rectangles show where there are specific scripts that can be run to convert their dependencies into output files. The Rails user interface and the database have no current dependencies.

The top level control structure is the Rails interface. It has tools that allow every script to be run, placed in the correct order of their dependencies on the main page. Additionally it includes a brief description of what each tool does. It does not provide a way to view the output of each script, but it does allow users to access the visualization showing the data produced from the Latent Semantic Indexing algorithm. Running any one tool from the Rails application leaves the whole program in a valid state, provided the tool is allowed to finish running before another is run. This allows users to only call tools when something has changed or updated, and otherwise allows users to access the visualization directly without having to wait for the program to run. Documentation included with the application includes instructions for running each tool from the command line, for further development and any potential debugging.

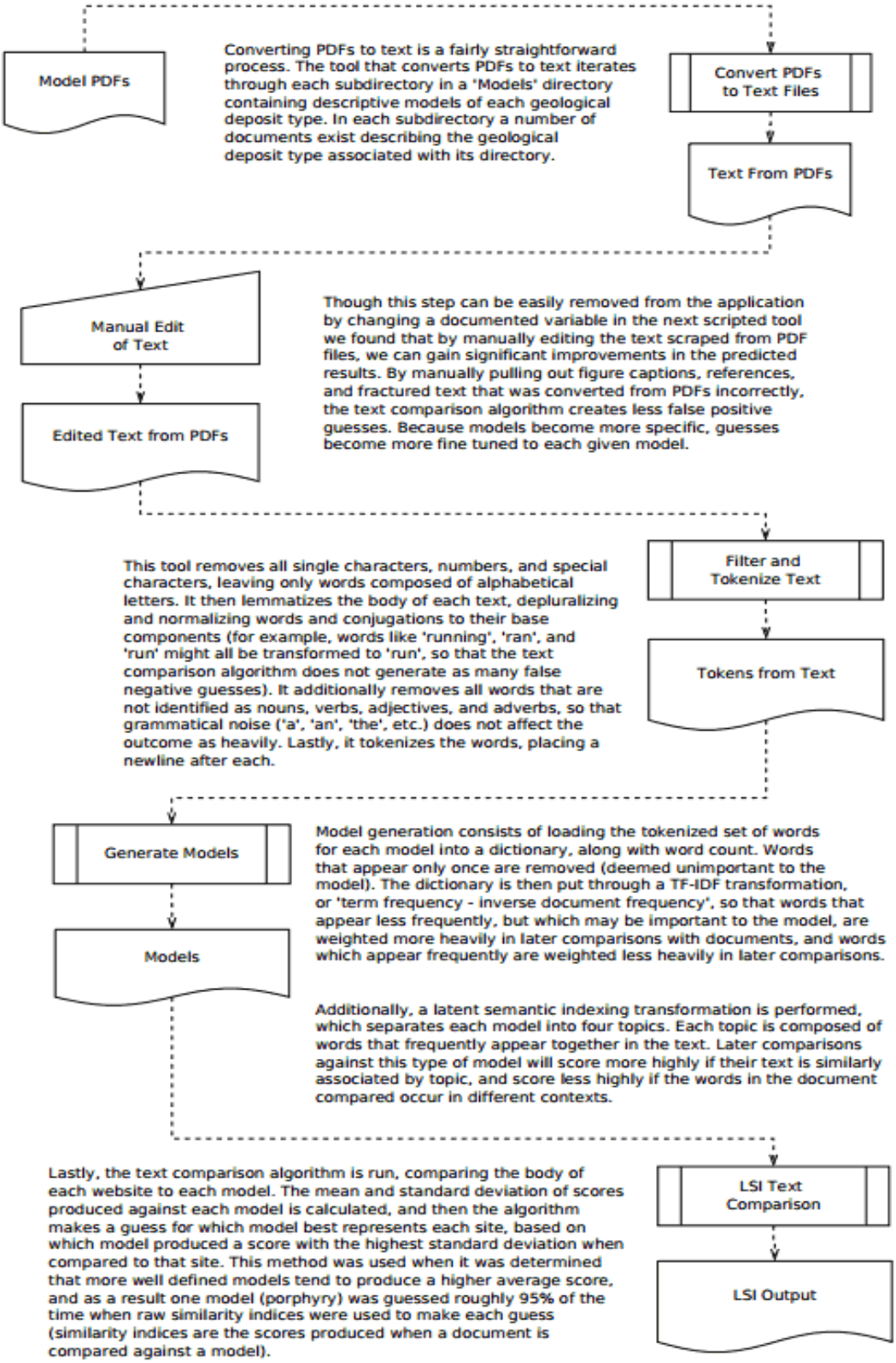
Technical Design

This application can be divided up into four subcomponents: the latent semantic indexing comparison, the machine learning comparison, the graphical user interface, and the data visualizations. Because data mining exploratory applications don't always produce accurate results, we decided to try two different methods to improve our chances of producing an effective product that would meet our client's needs.

Both algorithms were tested the same way. The client provided a list of over one thousand sites to run our algorithms with and a sizable number of these sites included additional information, such as the company name and the actual deposit type. Each tool monitored the guesses it produced and additionally determined the number of correct and incorrect guesses out of the known sites. This allowed us to get some indication of how well the algorithms were working individually as well as to make incremental improvements to each algorithm to improve accuracy.

The machine learning algorithm takes a distinctly different approach from the algorithm utilizing gensim. This algorithm begins by examining all the text pulled from the various PDFs for the different deposit types. After performing various filtering and lemmatizing the text into distinct tokens, it performs a series of comparisons to determine which words are important to a specific deposit type, and each of these words are assigned the same initial point value. Then, it makes a pass through all of the available data and guesses the type of each deposit using the word lists and point values it previously generated. Next, it looks specifically at the deposits types that already have known types and checks if it guessed correctly. If the guess is incorrect, it examines all the words that contributed to the score and attempts to add points to words that were important and found for the correct model but not important to the other models. This process is repeated through many runs of the data in order to generate a best fit model for the data set as a whole that can be used to predict the unknown deposits.

The visualization produced for the latent semantic indexing data displays the results by plotting points corresponding to each website using their standard deviation as the horizontal coordinate and their similarity index as the vertical coordinate. Additionally, the data visualization code was modified for use with the machine learning algorithm by using the length of extracted text for each site as the horizontal coordinate and the similarity index as the vertical coordinate. Hovering over a point in either visualization displays all data collected for that point. Clicking on a data point in either visualization opens the site in a new tab in the default web browser.


Model PDFs
Converting PDFs to text is a fairly straightforward process. The tool that converts PDFs to text iterates through each subdirectory in a 'Models' directory containing descriptive models of each geological deposit type. In each subdirectory a number of documents exist describing the geological deposit type associated with its directory.
Convert PDFs to Text Files
Text From PDFs
Manual Edit of Text
Though this step can be easily removed from the application by changing a documented variable in the next scripted tool we found that by manually editing the text scraped from PDF files, we can gain significant improvements in the predicted results. By manually pulling out figure captions, references, and fractured text that was converted from PDFs incorrectly, the text comparison algorithm creates less false positive guesses. Because models become more specific, guesses become more fine tuned to each given model.
Edited Text from PDFs
Filter and Tokenize Text
This tool removes all single characters, numbers, and special characters, leaving only words composed of alphabetical letters. It then lemmatizes the body of each text, depluralizing and normalizing words and conjugations to their base components (for example, words like 'running', 'ran', and 'run' might all be transformed to 'run', so that the text comparison algorithm does not generate as many false negative guesses). It additionally removes all words that are not identified as nouns, verbs, adjectives, and adverbs, so that grammatical noise ('a', 'an', 'the', etc.) does not affect the outcome as heavily. Lastly, it tokenizes the words, placing a newline after each.
Tokens from Text
Generate Models
Model generation consists of loading the tokenized set of words for each model into a dictionary, along with word count. Words that appear only once are removed (deemed unimportant to the model). The dictionary is then put through a TF-IDF transformation, or 'term frequency - inverse document frequency', so that words that appear less frequently, but which may be important to the model, are weighted more heavily in later comparisons with documents, and words which appear frequently are weighted less heavily in later comparisons.
Models
Additionally, a latent semantic indexing transformation is performed, which separates each model into four topics. Each topic is composed of words that frequently appear together in the text. Later comparisons against this type of model will score more highly if their text is similarly associated by topic, and score less highly if the words in the document compared occur in different contexts.
LSI Text Comparison
Lastly, the text comparison algorithm is run, comparing the body of each website to each model. The mean and standard deviation of scores produced against each model is calculated, and then the algorithm makes a guess for which model best represents each site, based on which model produced a score with the highest standard deviation when compared to that site. This method was used when it was determined that more well defined models tend to produce a higher average score, and as a result one model (porphyry) was guessed roughly 95% of the time when raw similarity indices were used to make each guess (similarity indices are the scores produced when a document is compared against a model).
LSI Output

Design and Implementation Decisions

Dependency Decisions

All tools and libraries used were chosen first and foremost for being well-maintained open source projects. It was advantageous to use projects available under LGPL, BSD/3, MIT/X, and MIT licenses for their freedom for ongoing projects using the provided source code, which include nearly unrestricted use for both commercial and private applications.

- Python 2.7 was used for the primary application functionality due to the availability of open-source tools for data mining in Python 2. Python 3 was not chosen because none of the mainstream data mining tools we explored have been sufficiently developed to reach version 1.0 in Python 3 yet.
- PDFMiner was a tool used to extract text from PDFs. It was chosen and used because of its compatibility with Python 2.7, length of use (it has been in development since 2004), and continued development. We did additional testing before integrating it to be sure it could parse PDF files correctly and with minimal errors and found it to be sufficiently functional with which to suit our needs.
- BeautifulSoup was used for extracting data from web files, in conjunction with the HTML5Lib parser. These were a good fit with the Python 2.7 environment. This pair of tools were chosen for their ease of use and minimal dependencies.
- The gensim library was chosen for the text comparison portion of this project based on its ongoing development, minimal dependencies, and primarily for its optimized implementation of many modern text comparison algorithms, such as Latent Semantic Analysis.
- D3.js was implemented for creating the data visualization in response to the client's suggestion, as well as being an appropriate and powerful JavaScript visualization library for Web implementations.
- Ruby on Rails was chosen as the tool to use for the web interface because of the client's interest in exploring this technology.

Functionality Decisions

- The PDF-to-text conversion tool was given its own script to convert PDFs into text files. The reason it is a standalone tool is because of the length of time it takes to run (which can be up to several hours) and because it only needs to be run when a model is changed or updated. This does allow for any model to be changed or updated with new or more PDFs, or for entirely new models to be added.
- The visualization was designed to include responsive filtering with several filter criteria, allowing the user to view model scores, actual models, or guessed models. It appends all relevant data associated with a highlighted point in the table below the graph for further investigation.

Lessons Learned

- Machine learning and document similarity comparisons are nontrivial programming challenges. Performing each effectively requires a great deal of understanding and planning prior to writing the code.
    - Requires understanding and appropriate use of transformation algorithms such as TF-IDF and Latent Semantic Indexing to generate useful models
    - Large amounts of high quality data are required to produce more effective models. This data should ideally be very similar in structure and language to the unknown data that you are trying to classify
    - More well defined models tend to have a higher average correlation with a given document of unknown classification. Therefore, in the case that some models are better defined than others, steps should be taken to normalize the various similarity scores to an extent.
- Verify the accuracy of test data. Out of the 1150 sites we were given to use as test data, more than half were erroneous. Erroneous data was classified as such if trying to reach the URL linked by the client resulted in page not found or page not valid errors thrown by the servers they were supposedly stored on. Trying to compare sites and visualize data is impossible without appropriately handling corrupted testing data.
- Filter results. We determined that many sites were returning financial or historical information about a given mining site, rather than information about the type of minerals contained within it. Because the text comparison utility we wrote makes a guess about which model a site is best represented by regardless of content, our correct guesses are hidden among useless or incorrect results. To handle this problem, we created a null model to catch results about financial and historical data, so that websites describing mineral deposits can be more accurately represented in our visualization.

Results

The primary goal of this project was to programmatically guess what types of mineral deposits websites describe. Guesses would be made by comparing websites to models composed of text derived from descriptions of deposits of interests. We were able to get every feature up and running, with varying degrees of accuracy.

We wrote two separate text comparison tools. The first compares the text extracted from a website to each model directly, producing a coefficient of similarity. This tool analyzes these similarity coefficients to produce a guess for each site. At this time, it correctly identifies 83 of the 333 sites with known models. However, of the 333 known models, the null model also catches 106 sites. This means that of the valid guesses, 83 out of 227 are correct (or 37%). Additionally, of the known sites, nearly all sites with a similarity coefficient above a certain threshold (specific to each model) is correctly identified, whereas websites identified as certain geologic models that have similarity indices below that threshold tend to be miscategorized. When viewed with the visualization, what this data shows us is that the top guesses for each model tend to have a higher fraction of correct guesses than the mid to low range guesses produced for each model.

The visualization makes it much more visually apparent that, as the standard deviation and simulation scores improve, so, too, do the results of our latent semantic indexing algorithm's prediction accuracy. This is a promising piece of data supportive of the ability of the algorithm to associate Web sites with their appropriate deposit type that compare well. The data tends to favor the most correct guesses with standard deviations above 0 and simulation index scores at or above 0.45. Furthermore, data with standard deviations above 1.7 and simulation index scores above 0.53 are almost always correct. Knowing this threshold is pivotal when analyzing future data on unknown documents and determining if the algorithm appropriately guessed the deposit type.

The separate text comparison tool was originally more exploratory by nature, but it operates as a learning algorithm and has led to promising results thus far. By making multiple passes through known data, it trains itself and modifies similarity weights of words and phrases in order to create a best fit model for the training data. Using a small selection of well defined sites, the algorithm was able to produce a best fit model that correctly recognized 95% of the sites it used as a training set. When the algorithm was run over the entire data set, it was able to create a best fit model correctly recognizing over 40% of the known sites.

Future Work

To improve the quality and accuracy of guesses, we suggest several possible routes. The most significant improvement will likely be made by simply improving and editing the text derived from PDFs manually, in order to more accurately describe the model the text is meant to match. This can have a huge impact because the PDF to text conversion utility is not 100% accurate, and because the PDF descriptions of deposit types are dissimilar from deposits described on websites. Where the PDF articles used describe deposit types from an academic standpoint, websites tend to be more geared toward investors or other persons who are not geologists. Improving models can be done intelligently by analyzing the topics and words that have the greatest impact on scores when a model is generated.

To improve filter accuracy, the null model can be edited and further built upon. Currently, a very minimal approach has been taken to match financial or historical words to test the effectiveness of a null model, but it still lacks much depth. Every model benefits from being as large as possible, while still being able to describe the documents it is intended to catch.

Another possible route that could lead to an improvement in model guessing is modifying the guessing logic itself. We have tried various permutations of guessing based on the similarity index (the score produced when a site is compared against a model, measures absolute similarity using a latent semantic indexing algorithm), including guessing based on the maximum, minimum, and average similarity indexes (Multiple similarity indices are produced for each comparison as a result of each model being composed of multiple documents. We chose to use the average because it gives better results than using either the minimum or maximum). We have gotten the best results so far (by a large margin) by guessing based on which model gives the highest standard deviation for a given site.

Appendices

Further information about the libraries used for this field session project can be found within the links below.

- gensim: Topic modelling for humans
  http://radimrehurek.com/gensim/
  Available under the GNU LGPL
  http://www.gnu.org/licenses/lgpl.html
- BeautifulSoup 4: HTML and XML parser
  http://www.crummy.com/software/BeautifulSoup/
  Available under the MIT License
  http://opensource.org/licenses/MIT
- HTML5Lib: HTML parser and serializer
  https://github.com/html5lib/html5lib-python
  Available under the MIT License
  https://github.com/html5lib/html5lib-python/blob/master/LICENSE
- PDFMiner: Python PDF parser and analyzer
  http://www.unixuser.org/~euske/python/pdfminer/
  Available under the MIT/X License
  http://www.unixuser.org/~euske/python/pdfminer/#license
- D3.js: Javascript library for manipulating documents based on data
  http://d3js.org/
  Available under the BSD License
  http://opensource.org/licenses/BSD-3-Clause
- Ruby on Rails: open-source web framework
  http://rubyonrails.org/
  Available under the MIT License
  http://opensource.org/licenses/mit-license.php