A Tale of Four Libraries

Alejandro Weinstein‡*, Michael Wakin‡

Abstract—This work describes the use some scientific Python tools to solve information gathering problems using Reinforcement Learning. In particular, we focus on the problem of designing an agent able to learn how to gather information in linked datasets. We use four different libraries—RL-Glue, Gensim, NetworkX, and scikit-learn—during different stages of our research. We show that, by using NumPy arrays as the default vector/matrix format, it is possible to integrate these libraries with minimal effort.

Index Terms—reinforcement learning, latent semantic analysis, machine learning

Introduction

In addition to bringing efficient array computing and standard mathematical tools to Python, the NumPy/SciPy libraries provide an ecosystem where multiple libraries can coexist and interact. This work describes a success story where we integrate several libraries, developed by different groups, to solve some of our research problems.

Our research focuses on using Reinforcement Learning (RL) to gather information in domains described by an underlying linked dataset. We are interested in problems such as the following: given a Wikipedia article as a seed, find other articles that are interesting relative to the starting point. Of particular interest is to find articles that are more than one-click away from the seed, since these articles are in general harder to find by a human.

In addition to the staples of scientific Python computing NumPy, SciPy, Matplotlib, and IPython, we use the libraries RL-Glue [Tan09], NetworkX [Hag08], Gensim [Reh10], and scikit-learn [Ped11].

Reinforcement Learning considers the interaction between a given environment and an agent. The objective is to design an agent able to learn a policy that allows it to maximize its total expected reward. We use the RL-Glue library for our RL experiments. This library provides the infrastructure to connect an environment and an agent, each one described by an independent Python program.

We represent the linked datasets we work with as graphs. For this we use NetworkX, which provides data structures to efficiently represent graphs, together with implementations of many classic graph algorithms. We use NetworkX graphs to describe the environments implemented using RL-Glue. We also use these graphs to create, analyze and visualize graphs built from unstructured data.

* *Corresponding author: aweinste@mines.edu*
‡ *the EECS department of the Colorado School of Mines*

One of the contributions of our research is the idea of representing the items in the datasets as vectors belonging to a linear space. To this end, we build a Latent Semantic Analysis (LSA) [Dee90] model to project documents onto a vector space. This allows us, in addition to being able to compute similarities between documents, to leverage a variety of RL techniques that require a vector representation. We use the Gensim library to build the LSA model. This library provides all the machinery to build, among other options, the LSA model. One place where Gensim shines is in its capability to handle big data sets, like the entirety of Wikipedia, that do not fit in memory. We also combine the vector representation of the items as a property of the NetworkX nodes.

Finally, we also use the manifold learning capabilities of sckit-learn, like the ISOMAP algorithm [Ten00], to perform some exploratory data analysis. By reducing the dimensionality of the LSA vectors obtained using Gensim from 400 to 3, we are able to visualize the relative position of the vectors together with their connections.

Source code to reproduce the results shown in this work is available at https://github.com/aweinstein/a_tale.

Reinforcement Learning

The RL paradigm [Sut98] considers an agent that interacts with an environment described by a Markov Decision Process (MDP). Formally, an MDP is defined by a state space $\mathscr{X}$, an action space $\mathscr{A}$, a transition probability function P, and a reward function r. At a given sample time $t = 0, 1, \ldots$ the agent is at state $x_t \in \mathscr{X}$, and it chooses action $a_t \in \mathscr{A}$. Given the current state x and selected action a, the probability that the next state is x' is determined by $P(x, a, x')$. After reaching the next state x', the agent observes an immediate reward $r(x')$. Figure 1 depicts the agent-environment interaction. In an RL problem, the objective is to find a function $\pi : \mathscr{X} \mapsto \mathscr{A}$, called the *policy*, that maximizes the total expected reward

$$R = \mathbf{E}\left[\sum_{t=1}^{\infty} \gamma^t r(x_t)\right],$$

where $\gamma \in (0, 1)$ is a given discount factor. Note that typically the agent does not know the functions P and r, and it must find the optimal policy by interacting with the environment. See Szepesvári [Sze10] for a detailed review of the theory of MDPs and the different algorithms used in RL.

We implement the RL algorithms using the RL-Glue library [Tan09]. The library consists of the *RL-Glue Core* program and a set of codecs for different languages[1] to communicate with the library. To run an instance of a RL problem one needs to write three different programs: the *environment*, the *agent*, and the *experiment*. The environment and the agent programs match exactly the corresponding elements of the RL framework, while

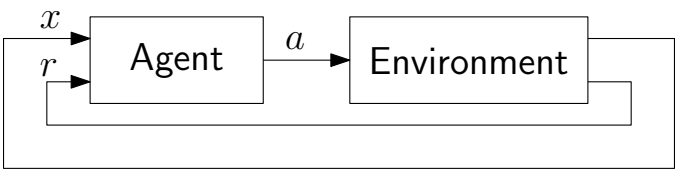

Fig. 1: *The agent-environment interaction. The agent observes the current state x and reward r; then it executes action $\pi(x) = a$.*

the experiment orchestrates the interaction between these two. The following code snippets show the main methods that these three programs must implement:

```
################ environment.py ################
class env(Environment):
    def env_start(self):
        # Set the current state

        return current_state

    def env_step(self, action):
        # Set the new state according to
        # the current state and given action.

        return reward

################## agent.py ##################
class agent(Agent):
    def agent_start(self, state):
        # First step of an experiment

        return action

    def agent_step(self, reward, obs):
        # Execute a step of the RL algorithm

        return action

################ experiment.py ################
RLGlue.init()
RLGlue.RL_start()
RLGlue.RL_episode(100) # Run an episode
```

Note that RL-Glue is only a thin layer among these programs, allowing us to use any construction inside them. In particular, as described in the following sections, we use a NetworkX graph to model the environment.

Computing the Similarity between Documents

To be able to gather information, we need to be able to quantify how relevant an item in the dataset is. When we work with documents, we use the similarity between a given document and the seed to this end. Among the several ways of computing similarities between documents, we choose the Vector Space Model [Man08]. Under this setup, each document is represented by a vector. The similarity between two documents is estimated by the *cosine similarity* of the document vector representations.

The first step in representing a piece of text as a vector is to build a *bag of words* model, where we count the occurrences of each term in the document. These word frequencies become the vector entries, and we denote the *term frequency* of term t in document d by $\mathrm{tf}_{t,d}$. Although this model ignores information related to the order of the words, it is still powerful enough to produce meaningful results.

In the context of a collection of documents, or corpus, word frequency is not enough to asses the importance of a term. For this reason, we introduce the quantity *document frequency* df_t, defined to be the number of documents in the collection that contain term t. We can now define the *inverse document frequency* (idf) as

$$\mathrm{idf}_t = \log \frac{N}{\mathrm{df}_t},$$

where N is the number of documents in the corpus. The idf is a measure of how unusual a term is. We define the tf-idf weight of term t in document d as

$$\text{tf-idf}_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t\,.$$

This quantity is a good indicator of the discriminating power of a term inside a given document. For each document in the corpus we compute a vector of length M, where M is the total number of terms in the corpus. Each entry of this vector is the tf-idf weight for each term (if a term does not exist in the document, the weight is set to 0). We stack all the vectors to build the $M \times N$ *term-document matrix* C.

Note that since typically a document contains only a small fraction of the total number of terms in the corpus, the columns of the term-document matrix are sparse. The method known as Latent Semantic Analysis (LSA) [Dee90] constructs a low-rank approximation C_k of rank at most k of C. The value of k, also known as the *latent dimension*, is a design parameter typically chosen to be in the low hundreds. This low-rank representation induces a projection onto a k-dimensional space. The similarity between the vector representation of the documents is now computed after projecting the vectors onto this subspace. One advantage of LSA is that it deals with the problems of *synonymy*, where different words have the same meaning, and *polysemy*, where one word has different meanings.

Using the Singular Value Decomposition (SVD) of the term-document matrix $C = U\Sigma V^T$, the k-rank approximation of C is given by

$$C_k = U_k \Sigma_k V_k^T,$$

where U_k, Σ_k, and V_k are the matrices formed by the k first columns of U, Σ, and V, respectively. The tf-idf representation of a document q is projected onto the k-dimensional subspace as

$$q_k = \Sigma_k^{-1} U_k^T q.$$

Note that this projection transforms a sparse vector of length M into a dense vector of length k.

In this work we use the *Gensim* library [Reh10] to build the vector space model. To test the library we downloaded the top 100 most popular books from project Gutenberg.[2] After constructing the LSA model with 200 latent dimensions, we computed the similarity between *Moby Dick*, which is in the corpus used to build the model, and 6 other documents (see the results in Table 1). The first document is an excerpt from *Moby Dick*, 393 words long. The second one is an excerpt from the Wikipedia *Moby Dick* article. The third one is an excerpt, 185 words long, of *The Call of the Wild*. The remaining two documents are excerpts from Wikipedia articles not related to *Moby Dick*. The similarity values we obtain validate the model, since we can see high values (above 0.8) for the documents related to *Moby Dick*, and significantly smaller values for the remaining ones.

1. Currently there are codecs for Python, C/C++, Java, Lisp, MATLAB, and Go.

2. As per the April 20, 2011 list, http://www.gutenberg.org/browse/scores/top.

Text description	LSA similarity
Excerpt from Moby Dick	0.87
Excerpt from Wikipedia Moby Dick article	0.83
Excerpt from The Call of the Wild	0.48
Excerpt from Wikipedia Jewish Calendar article	0.40
Excerpt from Wikipedia Oxygen article	0.33

TABLE 1: *Similarity between Moby Dick and other documents.*

Next, we build the LSA model for Wikipedia that allows us to compute the similarity between Wikipedia articles. Although this is a lengthy process that takes more than 20 hours, once the model is built, a similarity computation is very fast (on the order of 10 milliseconds). Results in the next section make use of this model.

Note that although in principle it is simple to compute the LSA model of a given corpus, the size of the datasets we are interested in make doing this a significant challenge. The two main difficulties are that in general (i) we cannot hold the vector representation of the corpus in RAM memory, and (ii) we need to compute the SVD of a matrix whose size is beyond the limits of what standard solvers can handle. Here Gensim does stellar work by being able to handle both these challenges.

Representing the State Space as a Graph

We are interested in the problem of gathering information in domains described by linked datasets. It is natural to describe such domains by graphs. We use the NetworkX library [Hag08] to build the graphs we work with. NetworkX provides data structures to represent different kinds of graphs (undirected, weighted, directed, etc.), together with implementations of many graph algorithms. NetworkX allows one to use any hashable Python object as a node identifier. Also, any Python object can be used as a node, edge, or graph attribute. We exploit this capability by using the LSA vector representation of a Wikipedia article, which is a NumPy array, as a node attribute.

The following code snippet shows a function[3] used to build a directed graph where nodes represent Wikipedia articles, and the edges represent links between articles. Note that we compute the LSA representation of the article (line 11), and that this vector is used as a node attribute (line 13). The function obtains up to `n_max` articles by breadth-first crawling the Wikipedia, starting from the article defined by `page`.

```
1  def crawl(page, n_max):
2      G = nx.DiGraph()
3      n = 0
4      links = [(page, -1, None)]
5      while n < n_max:
6          link = links.pop()
7          page = link[0]
8          dist = link[1] + 1
9          page_text = page.edit().encode('utf-8')
10         # LSI representation of page_text
11         v_lsi = get_lsi(page_text)
12         # Add node to the graph
13         G.add_node(page.name, v=v_lsi)
14         if link[2]:
15             source = link[2]
16             dest = page.name
17             if G.has_edge(source, dest):
18                 # Link already exists
19                 continue
20             else:
21                 sim = get_similarity(page_text)
22                 self.G.add_edge(source,
23                                 dest,
24                                 weight=sim,
25                                 d=dist)
26         new_links = [(l, dist, page.name)
27                      for l in page.links()]
28         links = new_links + links
29         n += 1
30
31     return G
```

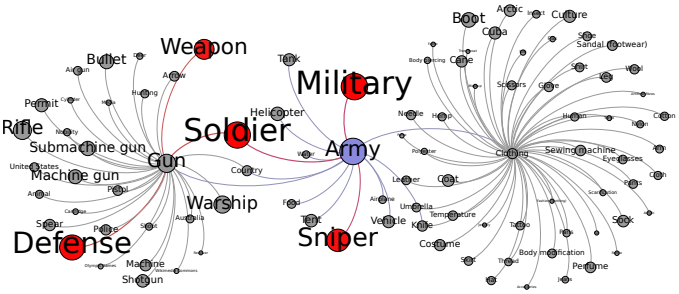

Fig. 2: *Graph for the "Army" article in the simple Wikipedia with 97 nodes and 99 edges. The seed article is in light blue. The size of the nodes (except for the seed node) is proportional to the similarity. In red are all the nodes with similarity greater than 0.5. We found two articles ("Defense" and "Weapon") similar to the seed three links ahead.*

We now show the result of running the code above for two different setups. In the first instance we crawl the *Simple English Wikipedia*[4] using "Army" as the seed article. We set the limit on the number of articles to visit to 100. The result is depicted[5] in Fig. 2, where the node corresponding to the seed article is in light blue and the remaining nodes have a size proportional to the similarity with respect to the seed. Red nodes are the ones with similarity bigger than 0.5. We observe two nodes, "Defense" and "Weapon", with similarities 0.7 and 0.53 respectively, that are three links away from the seed.

In the second instance we crawl Wikipedia using the article "James Gleick"[6] as seed. We set the limit on the number of articles to visit to 2000. We show the result in Fig. 3, where, as in the previous example, the node corresponding to the seed is in light blue and the remaining nodes have a size proportional to the similarity with respect to the seed. The eleven red nodes are the ones with similarity greater than 0.7. Of these, 9 are more than one link away from the seed. We see that the article with the biggest similarity, with a value of 0.8, is about "Robert Wright (journalist)", and it is two links away from the seed (passing through the "Slate magazine" article). Robert Wright writes books about sciences, history and religion. It is very reasonable to consider him an author similar to James Gleick.

Another place where graphs can play an important role in the RL problem is in finding basis functions to approximate the value-

3. The parameter `page` is a mwclient page object. See http://sourceforge.net/apps/mediawiki/mwclient/.

4. The Simple English Wikipedia (http://simple.wikipedia.org) has articles written in *simple English* and has a much smaller number of articles than the standard Wikipedia. We use it because of its simplicity.

5. To generate this figure, we save the NetworkX graph in GEXF format, and create the diagram using Gephi (http://gephi.org/).

6. James Gleick is "an American author, journalist, and biographer, whose books explore the cultural ramifications of science and technology".

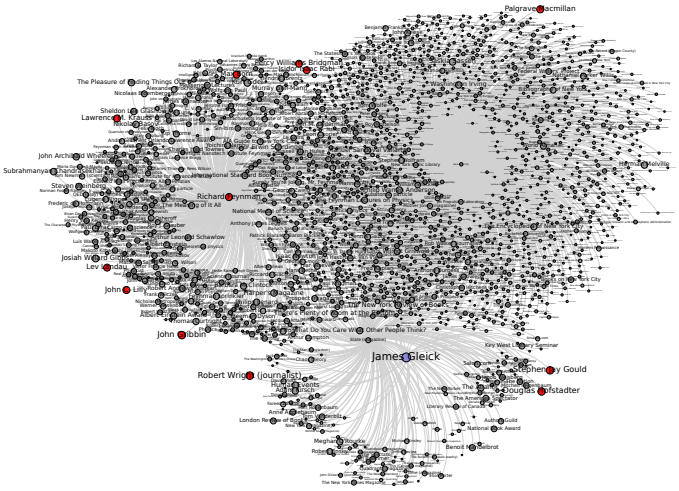

Fig. 3: *Graph for the "James Gleick" Wikipedia article with 1975 nodes and 1999 edges. The seed article is in light blue. The size of the nodes (except for the seed node) is proportional to the similarity. In red are all the nodes with similarity bigger than 0.7. There are several articles with high similarity more than one link ahead.*

function. The value-function is the function $V^\pi : \mathscr{X} \mapsto \mathbb{R}$ defined as

$$V^\pi(x) = \mathbf{E}\left[\sum_{t=1}^{\infty} \gamma^t r(x_t) \mid x_0 = x, a_t = \pi(x_t)\right],$$

and plays a key role in many RL algorithms [Sze10]. When the dimension of $\mathscr{X}$ is significant, it is common to approximate $V^\pi(x)$ by

$$V^\pi \approx \hat{V} = \Phi w,$$

where Φ is an n-by-k matrix whose columns are the basis functions used to approximate the value-function, n is the number of states, and w is a vector of dimension k. Typically, the basis functions are selected by hand, for example, by using polynomials or radial basis functions. Since choosing the right functions can be difficult, Mahadevan and Maggioni [Mah07] proposed a framework where these basis functions are learned from the topology of the state space. The key idea is to represent the state space by a graph and use the k smoothest eigenvectors of the graph laplacian, dubbed *Proto-value* functions, as basis functions. Given the graph that represents the state space, it is very simple to find these basis functions. As an example, consider an environment consisting of three 16×20 grid-like rooms connected in the middle, as shown in Fig. 4. Assuming the graph is stored in `G`, the following code[7] computes the eigenvectors of the laplacian:

```
L = nx.laplacian(G, sorted(G.nodes()))
evalues, evec = np.linalg.eigh(L)
```

Figure 5 shows[8] the second to fourth eigenvectors. Since in general value-functions associated to this environment will exhibit a fast change rate close to the room's boundaries, these eigenvectors provide an efficient approximation basis.

7. We assume that the standard `import numpy as np` and `import networkx as nx` statements were previously executed.

8. The eigenvectors are reshaped from vectors of dimension $3 \times 16 \times 20 = 960$ to a matrix of size 16-by-60. To get meaningful results, it is necessary to build the laplacian using the nodes in the grid in a row major order. This is why the `nx.laplacian` function is called with `sorted(G.nodes())` as the second parameter.

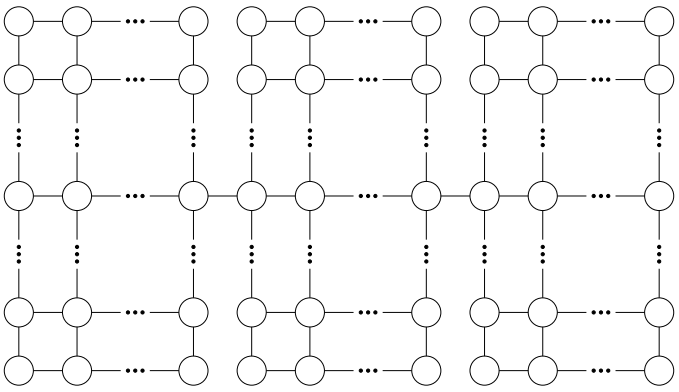

Fig. 4: *Environment described by three* 16×20 *rooms connected through the middle row.*

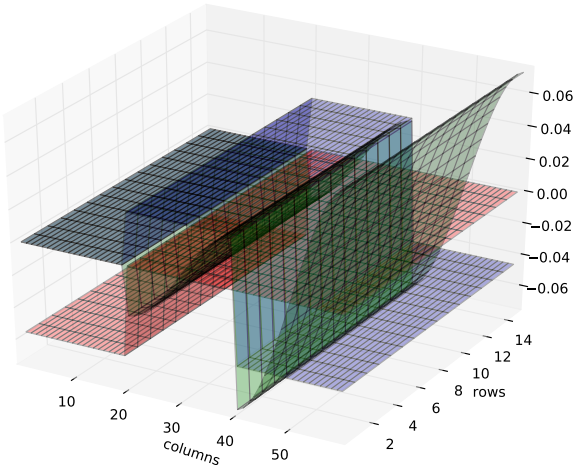

Fig. 5: *Second to fourth eigenvectors of the laplacian of the three rooms graph. Note how the eigendecomposition automatically captures the structure of the environment.*

Visualizing the LSA Space

We believe that being able to work in a vector space will allow us to use a series of RL techniques that otherwise we would not be available to use. For example, when using Proto-value functions, it is possible to use the Nyström approximation to estimate the value of an eigenvector for out-of-sample states [Mah06]; this is only possible if states can be represented as points belonging to a Euclidean space.

How can we embed an entity in Euclidean space? In the previous section we showed that LSA can effectively compute the similarity between documents. We can take this concept one step forward and use LSA not only for computing similarities, but also for embedding documents in Euclidean space.

To evaluate the soundness of this idea, we perform an exploratory analysis of the simple Wikipedia LSA space. In order to be able to visualize the vectors, we use ISOMAP [Ten00] to reduce the dimension of the LSA vectors from 200 to 3 (we use the ISOMAP implementation provided by scikit-learn [Ped11]). We show a typical result in Fig. 6, where each point represents the LSA embedding of an article in $\mathbb{R}^3$, and a line between two points represents a link between two articles. We can see how the points close to the "Water" article are, in effect, semantically related ("Fresh water", "Lake", "Snow", etc.). This result confirms that the

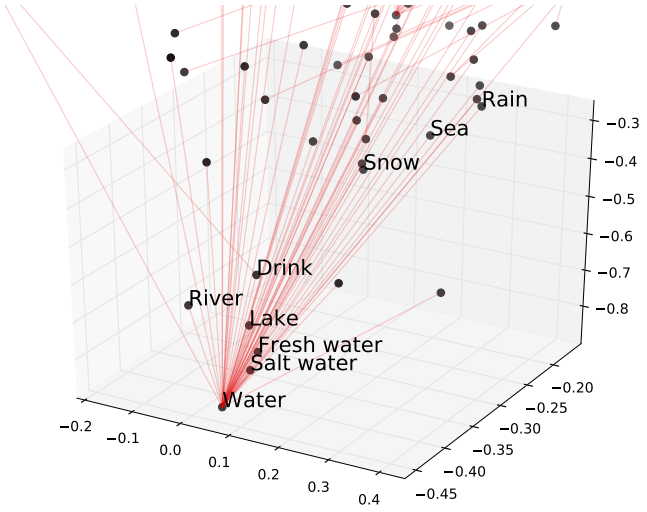

Fig. 6: *ISOMAP projection of the LSA space. Each point represents the LSA vector of a Simple English Wikipedia article projected onto $\mathbb{R}^3$ using ISOMAP. A line is added if there is a link between the corresponding articles. The figure shows a close-up around the "Water" article. We can observe that this point is close to points associated to articles with a similar semantic.*

LSA representation is not only useful for computing similarities between documents, but it is also an effective mechanism for embedding the information entities into a Euclidean space. This result encourages us to propose the use of the LSA representation in the definition of the state.

Once again we emphasize that since Gensim vectors are NumPY arrays, we can use its output as an input to scikit-learn without any effort.

Conclusions

We have presented an example where we use different elements of the scientific Python ecosystem to solve a research problem. Since we use libraries where NumPy arrays are used as the standard vector/matrix format, the integration among these components is transparent. We believe that this work is a good success story that validates Python as a viable scientific programming language.

Our work shows that in many cases it is advantageous to use general purposes languages, like Python, for scientific computing. Although some computational parts of this work might be somewhat simpler to implement in a domain specific language,[9] the breadth of tasks that we work with could make it hard to integrate all of the parts using a domain specific language.

Acknowledgment

This work was partially supported by AFOSR grant FA9550-09-1-0465.

References

[Tan09] B. Tanner and A. White. *RL-Glue: Language-Independent Software for Reinforcement-Learning Experiments*, Journal of Machine Learning Research, 10(Sep):2133-2136, 2009

[Hag08] A. Hagberg, D. Schult and P. Swart, *Exploring Network Structure, Dynamics, and Function using NetworkX*, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11-15, Aug 2008

9. Examples of such languages are MATLAB, Octave, SciLab, etc.

[Ped11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay. *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, 12:2825-2830, 2011

[Reh10] R. Řehůřek and P. Sojka. *Software Framework for Topic Modelling with Large Corpora*, in Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, pp. 45-50 May 2010

[Sze10] C. Szepesvári. *Algorithms for Reinforcement Learning*. San Rafael, CA, Morgan and Claypool Publishers, 2010.

[Sut98] R.S. Sutton and A.G. Barto. *Reinforcement Learning*. Cambridge, Massachusetts, The MIT press, 1998.

[Mah07] S. Mahadevan and M. Maggioni. *Proto-value functions: A Laplacian framework for learning representation and control in Markov decision processes*. Journal of Machine Learning Research, 8:2169-2231, 2007.

[Man08] C.D. Manning, P. Raghavan and H. Schutze. *An introduction to information retrieval*. Cambridge, England. Cambridge University Press, 2008

[Ten00] J.B Tenenbaum, V. de Silva, and J.C. Langford. *A global geometric framework for nonlinear dimensionality reduction* . Science, 290(5500), 2319-2323, 2000

[Mah06] S. Mahadevan, M. Maggioni, K. Ferguson and S.Osentoski. *Learning representation and control in continuous Markov decision processes*. National Conference on Artificial Intelligence, 2006.

[Dee90] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer and R. Harshman, R. (1990). *Indexing by latent semantic analysis.* Journal of the American Society for Information Science, 41(6), 391-407.