Applying Machine Learning to Improve apropos(1)

Abhinav Upadhyay
<*abhinav@NetBSD.org*>

Abstract

In 2011 NetBSD acquired a new implementation of `apropos(1)`. This new implementation is capable of doing full text search using the Sqlite backend.

One of the core components of this new implementation of `apropos(1)` is its ranking algorithm, which allows `apropos(1)` to show more relevant results at the top of the search results. This ranking algorithm uses a term weighting scheme called *tf-idf*. Its performance has largely proven to be quite satisfactory, however, there is still much room of improvement.

Playing around with the ranking model requires a dataset in order to evaluate and compare the performance of various models. This paper discusses the creation of a dataset in order to evaluate the performance of ranking models. It also discusses results of training machine learning models on this dataset with the task of improving the existing ranking algorithm used in `apropos(1)`.

1 Introduction

The classical version of `apropos(1)` [2] worked by just searching the `NAME` section of man pages. This was a fine compromise considering the computing power of the machines when `apropos(1)` was first implemented. However, on present day machines much more interesting things can be achieved. As mentioned in [5], in 2011 NetBSD replaced its old `apropos(1)` with a new implementation. The new implementation [1] was designed to search over the complete body of man pages. With the increased search area, the number of search results also increased in the same proportion and in order to make `apropos(1)` a useful tool, it was important to figure out which of the matches are the most relevant to the user query and show them in the beginning. To do this `apropos(1)` employs a sophisticated ranking algorithm. The algorithm computes a relevance score for each of the search results and `apropos(1)` sorts the results in decreasing order of this relevance score.

The first section of this paper briefly discusses the design of the new `apropos(1)` in NetBSD and the details of its ranking scheme. In later sections it talks about machine learning and how does it plug into the existing ranking algorithm of `apropos(1)` to improve its performance. Namely, it discusses the details of the regression and classification models from machine learning and analyses their impact on the performance of `apropos(1)`.

2 Implementation of NetBSD's apropos(1)

The new `apropos(1)` for *NetBSD* was developed In 2011 as part of *Google Summer of Code 2011*. This new implementation uses *mandoc*'s [11] library interface to traverse the abstract syntax tree representation of the man pages to extract the text out of the man pages, and indexes it using *Sqlite's* FTS module [12]. Listings 1 and 2 show the differences in the outputs of the old `apropos(1)` and the new implementation in NetBSD.

2.1 How the new apropos(1) works

`makemandb(8)` [3] is the tool which was designed to replace `makewhatis(8)` [4]. It is used to parse and index man pages. `makemandb(8)` uses *mandoc's* AST library interface to parse man pages and stores them in an Sqlite table. The schema of this table is shown in table 1.

```
$apropos ls
intro(9) - introduction to kernel internals
rump(7) - The Anykernel and Rump Kernels
NLS(7) - Native Language Support Overview
usermgmt.conf(5) - user management tools configuration file
shells(5) - shell database
protocols(5) - protocol name data base
pkg_install.conf(5) - configuration file for package installation tools
uslsa(4) - USB support for Silicon Labs CP210x series serial adapters
mpt(4) - LSI Fusion-MPT SCSI/Fibre Channel driver
mpls(4) - Multiprotocol Label Switching

$apropos "how to compare two strings"
how to compare two strings: nothing appropriate
```

Listing 1: Old apropos output

```
$apropos ls
ls (1)    list directory contents
nslookup (8)      query Internet name servers interactively
curses_border (3) curses border drawing routines
column (1)        columnate lists
dmctl (8) manipulate device-mapper driver command
getbsize (3)      get user block size
chflags (1)       change file flags
symlink (7)       symbolic link handling
string_to_flags (3)       Stat flags parsing and printing functions
stat (1)  display file status

$apropos ``how to compare two strings''
strcmp (3)        compare strings
memcmp (9)        compare byte string
memcmp (3)        compare byte string
bcmp (3)          compare byte string
bcmp (9)          compare byte string
ldapcompare (1)   LDAP compare tool
strcasecmp (3)    compare strings, ignoring case
wcscasecmp (3)    compare wide-character strings, ignoring case
msgcmp (1)        compare message catalog and template
strcoll (3)       compare strings according to current collation
```

Listing 2: New apropos output

Column	Weight
name (NAME)	2.0
name_desc	2.0
desc (DESCRIPTION)	0.55
library	0.10
return_values	0.001
environment	0.20
files	0.01
exit_status	0.001
diagnostics	2.0
errors	0.05
machine	1.0

Table 1: Sections indexed and their associated weights

The design of the database has been described in detail in [2]. The key part of the design is that the content of the man pages is split across multiple columns in the table. Each column in the table represents a section of the man page. There are columns for prominent sections like `NAME`, `DESCRIPTION`, `RETURN VALUES`, `LIBRARY`, `ENVIRONMENT`, `FILES` and so on. Table-1 shows the list of columns which are part of the database table at this time.

The split of the content into multiple columns is an important part of the design of the ranking algorithm used by `apropos(1)`. Each of these sections is assigned a weight. The higher the weight for a section, the higher is the importance for a match found in that section. Algorithm-1 shows the ranking algorithm as used by `apropos(1)`. It shows how those weights are used while computing the relevance score for a document for a given query. To lookup the details of the ranking algorithm, please refer [5].

The quantities `tf` and `idf` as shown in Algorithm-1 are *term frequency* and *inverse document frequency* respectively [7]. *Term frequency* of a term t in a document d is defined as the number of times the term occurs in that document. Whereas the *document frequency* is defined as the number of documents in which that term occurs at least once. *Inverse document frequency* is an inverse quantity of *document frequency*, it is used to dampen the effect of words which are very common, i.e., have high term frequencies and occur in a large number of documents. Therefore, the product $tf \times idf$ tends to boost matches of the terms having high frequencies within a document but at the same time discouraging matches of the very common words, such as *"and"*, *"the"*, *"or"* etc.

The weights as shown in Table-1 were obtained by manually running a fixed set of queries and inspecting the output produced by `apropos(1)` for various values

Algorithm 1 Compute Relevance Weight of a Document for a Given User Query

Require: User query `q`
Require: Document `d` whose weight needs to be computed
Require: An array `weights` consisting of preassigned weights for different columns of the FTS table
`tf` ← 0.0
`idf` ← 0.0
`k` ← 3.5 ▷ k is an experimentally determined parameter
`doclen` ← length of the current document
`ndoc` ← Total number of documents in the corpus
for each phrase `p` in `q` **do**
  for each column `c` in the FTS table **do**
    $w \leftarrow weights[c]$ ▷ weight for column `c`
    $idf \leftarrow idf + log(\frac{ndoc}{ndocshitcount}) \times w$
    $tf \leftarrow tf + \frac{(nhitcount \times w)}{(nglobalhitcount \times doclen)}$
  end for
end for
$score \leftarrow \frac{(tf \times idf)}{(k+tf)}$
return *score*

of the weights. The set of values of these weights for which `apropos(1)` was producing best results were chosen as the final values. However, these values are not necessarily the best possible values of those weights, they worked well for those fixed set of queries, but they might not necessarily produce the expected results for a different set of queries.

We want to obtain the optimum values of these weights which will work well for all possible set of queries representing the man page corpus in NetBSD. We can use machine learning models to optimise these weights.

3 Machine Learning - how does it help

Machine learning [8] has been defined as the field of study that gives computers the ability to learn without being explicitly programmed. Machine learning models are trained on a set of example inputs (called training data) in order to learn a function which can be used to predict the output of such inputs. These set of input examples is called a dataset, and it consists of the features of the input data, along with the expected output. For example, for the problem of predicting housing prices, the input features could be number of bedrooms, the area of the plot, and the output would be the actual price for those houses. Given enough number of examples of

this data, the machine learning model would try to learn a function which could predict the price of the house.

Machine learning models are themselves divided into two broad categories:

1. **Supervised machine learning** [8] is used for problems where we have labelled dataset. The housing price example mentioned above falls under the category of supervised learning. Similarly, a dataset consisting of images along with labels about whether they contain cats or not, can be used to train a model to recognize cats in images

2. **Unsupervised machine learning** [8] models are used for problems where the output in the dataset is not labelled. Clustering is an example of unsupervised machine learning where the model learns to identify clusters in the input dataset without any labels in the dataset or human supervision. Intrusion detection is also an example which comes under unsupervised learning. The amount of data produced by the devices and appliances running in a production environment is so massive that it is impossible for humans to label it. Instead, unsupervised learning techniques allow building models which can learn to identify events which signify normal operations in the system and events which signify an abnormality.

3.1 Supervised Machine Learning

For the task of learning the weights of the ranking algorithm, we are only concerned with supervised learning models. Supervised machine learning consists of two broad categories of models: **regression** and **classification** [8]. Regression models are used for problems where we want to predict a continuous range of values. Predicting the price of a house, or predicting the temperature of the day are examples of regression problems. Classification models are used for problems where the model needs to learn to predict one of the fixed set of discrete values. Examples of classification problems include predicting whether the image contains a cat or not, or whether a search result is relevant to the query or not.

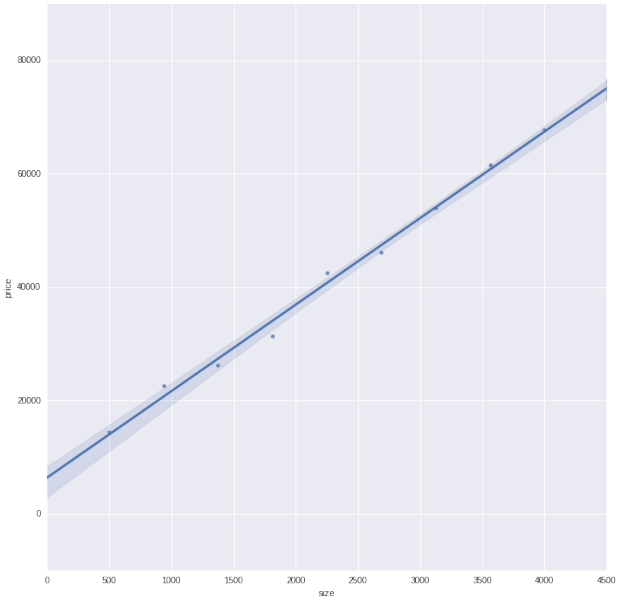
3.2 Understanding Regression

In this section we see in detail how does a regression model work. Table-2 shows a sample training dataset consisting of house sizes in square feet and prices of the houses in $. In order to learn to predict the price of a house given its size, we start by defining a hypothesis

Area (Sqft.)	Price ($)
500	14375
937	22450
1372	26220
1812	31360
2250	42535
2687	46195
3125	54060
3562	61540
4000	67700

Table 2: Sample housing price data set

Figure 1: Area vs Housing Prices



function such as:

$$y = w_0 + w_1 x \tag{1}$$

Where y represents the output of the model (price of the house in this example), x represents the input data (size of the house in case of this example), and w_0 and w_1 are the weights to be learned by the model. Equation (1) can be rewritten as:

$$y = w_0 x_0 + w_1 x_1 \tag{2}$$

where x_0 is always 1, it is just added for notational convenience.

This hypothesis function presented in (2) is good for representing only one input feature x_1. It can easily be extended to represent an arbitrary number of input features, as shown below:

$$y = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_Nx_N \quad (3)$$

The equation in (3) can be compactly represented as:

$$y = \sum_{i=1}^{N} w_i x_i \quad (4)$$

The equation presented in (4) represents the final hypothesis function, which can be used to predict the price of a house given its size as input to the function. In order to evaluate the performance of the model's prediction, an error function is required, which quantifies how far are model's prediction from actual values. The model trains on the training data to minimize this error. One of the most common error function used for regression is the **residual sum of squares** (**RSS**) [8], which is defined below:

$$E = \frac{1}{2}\sum_{i=1}^{N}(y_i^{'} - y_i)^2 \quad (5)$$

Here, $y_i^{'}$ represents the value predicted by the regression model and y_i represents the real value for input i as present in the training data. The quantity $y_i^{'} - y_i$ represents the difference between the output predicted by the model and the actual output, which is essentially the error made by the model. Squaring this error ensures that it will always remain positive. Optimization algorithms such as *gradient descent* [8] are used in order to minimize this error over the input dataset. The values of w at which the error function is minimized represent the optimum values of the model weight parameters.

As shown in Algorithm-1, `apropos(1)` uses a ranking function which is very similar to the hypothesis function shown in (4). The function to calculate the relevance score is shown below:

$$score = \sum_{c=1}^{N} w_c \times tf_c \times idf_c \quad (6)$$

Where c is the column in the Sqlite table, representing the sections of man pages. w_c is the weight associated with column c, tf_c is the term frequency for column c and idf_c is the inverse document frequency for column c. Comparing this to the hypothesis function above, it can be seen that this function can easily be fit into a regression model to learn the optimum value of the weights.

3.3 Generation of a dataset for apropos(1)

In order to train a regression model for learning the weights for the ranking algorithm of `apropos(1)` a training dataset is needed. Lack of an existing dataset makes this a difficult problem. In order to build this dataset, query logs from *man-k.org*[1] [10] have been used.

The query logs of *man-k.org* provided about 1100 unique queries. Those queries were run on `apropos(1)` and for each of those queries the top 5 results were sampled. Along with the top 5 results, the column wise *tf-idf* weights were also produced for each of those results. While generating these *tf-idf* weights it was made sure that those were pure *tf-idf* scores and that they were not multiplied by the existing weights being used in `apropos(1)`. Not using the original weights while generating the scores for this dataset was an important part of the process. If those weights were part of the scores present in the dataset, the machine learning models trained on that dataset would end up learning the same weights again.

Finally, each row of the dataset thus generated was manually labelled, by assigning a relevance score on a scale of 0 to 4, where 0 represents least relevant result and 4 represents most relevant result to the query. At the end of this process, a small dataset of around 2500 records was generated. The dataset is available on the github repository of the project [3].

3.4 Result of Ranking as Regression

For training regression models, the final relevance score (column `w_total` in the dataset) was used as the target feature while the column wise `tf-idf` weights were used as the input features to the model. Before training the models, the value of the `w_total` column for the rows which were labelled with a relevance value of 4 (most relevant to the query), was set to 0.67, which was the maximum value of `w_total` in the complete dataset. While for the rows which were labeled as 0 (least relevant to the query), their `w_total` was set to 0.62, which was the minimum value of `w_total` in the

[1] *man-k.org* is a web based interface to NetBSD's `apropos(1)`, `built by the author`.

dataset. Doing this was necessary because this indicates to the model that it needs to learn weights so that search results with features similar to the rows having relevance score of 4 are pushed up in the ranks, while documents having features similar to that of the rows with relevance score of 0 are pushed down. If this was not done, the model would not have anything to learn and all the weights learned by it would be close to 1. For example, consider a query like *"list files"* and the top two results found by `apropos(1)`: `ls(1)` and `file(1)`. While `file(1)` shows up at rank one and `ls(1)` at two, the model should learn to rank `ls(1)` as number one for this query.

Various regression models were trained and their performance was measured on this task. In order to objectively measure the performance of these models, the dataset was split into three parts - *training set*, *validation set*, and *test set*. Models were trained on the *training set* and the *validation set* was used to tune their hyper parameters. Their final performance was measured on the *test set*.

It is a good practice to not to test the performance of the model on the same data on which it was trained, because instead of learning to generalize from data, the model can simply memorize the training data and give 100% performance, but when tested on unseen data it would perform very badly. For this purpose a *validation set* is used. Based on the performance of the model on the *validation set* the hyper parameters of the model are tuned and it is retrained until the performance on the *validation set* becomes satisfactory. The *test set* is held back for evaluating the final performance of the model.

In order to measure the performance of the regression models, **mean squared error** [8] was used as the metric. Mean squared error is one of the standard metrics for measuring the performance of regression models and is computed as shown below:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i^{'} - y_i)^2 \tag{7}$$

Here again, $y_i^{'}$ is the value predicted by the model for the i^{th} input example and y_i is the actual value of that example. N is the total number of examples in the dataset.

A Ridge regression model gave a mean squared error value of approximately 1.9×10^{-4}. Whereas a random forest model obtained an error of about 1.6×10^{-4}. The random forest model used here was trained using 300 estimators, with a maximum depth of 15. It is important to note here that even though the random forest model outperforms the ridge regression model, the function learned by the random forest model might not be strictly linear and thus those weights may not work well with the linear ranking model used in `apropos(1)`.

4 Ranking as Classification Problem

Ranking can also be treated as classification problem where the task is to classify the search results as either *relevant* or *not-relevant*.

There are a variety of classification models which are available, such as *logistic regression*, *support vector machines*, *neural networks* and so on. However, out of all these models, *logisitc regression* closely resembles the form of the ranking function used by `apropos(1)`, as described in the previous section.

Logistic regression model is essentially an extension of the *linear regression* model described in the previous section. Since the output of a *linear regression* model is continuous, while a classification model needs to predict one of the fixed set of classes as its output, an additional step is required. The output as produced by the regression model is passed to a function called the *sigmoid* function, which scales down this value in the range of $[0, 1]$.

$$sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{8}$$

Where z is the output produced by the *linear regression* model (as shown in (4)).

The scaled down value essentially represents the probability of the input belonging to one of output classes. For a binary classification problem, usually a value > 0.5 is used to classify the input as belonging to class 1, or otherwise the input is classified as belonging to class 2. For a multiclass classification task, something more complicated is needed. Usually a technique called *one-vs-all* is used for doing multiclass classification, where one binary classifier is trained per class in the training data, and each binary classifier learns to classify the input as either belonging to that class or belonging to one of the other classes. If the input data has $k + 1$ classes, then k number of binary classifiers are trained. When doing prediction, the input is passed through each of the classifiers and the classifier predicting the maximum probability is chosen as the winner.

The same dataset as used in the regression model was used for training classification models. The `w_total`

```
$ # output with old weights
$ apropos -n 10 -C fork
fork (2)  create a new process
perlfork (1)      Perls fork() emulation
cpu_lwp_fork (9)  finish a fork operation
pthread_atfork (3)  register handlers to be called when process forks
rlogind (8)       remote login server
rshd (8)  remote shell server
rexecd (8)        remote execution server
script (1)        make typescript of terminal session
moncontrol (3)    control execution profile
vfork (2) spawn new process in a virtual memory efficient way

$ #output with new weights
$ apropos -n 10 -C fork
fork (2) create a new process
perlfork (1) Perls fork() emulation
cpu_lwp_fork (9) finish a fork operation
pthread_atfork (3) register handlers to be called when process forks
vfork (2) spawn new process in a virtual memory efficient way
clone (2) spawn new process with options
daemon (3) run in the background
script (1) make typescript of terminal session
openpty (3) tty utility functions
rlogind (8) remote login server}
```

Listing 3: Comparison of output with old and new weights

```
$ #output with old weights
$ apropos -n 10 create new process
init (8)  process control initialization
fork (2)  create a new process
fork1 (9) create a new process
timer_create (2)  create a per-process timer
getpgrp (2)       get process group
supfilesrv (8)    sup server processes
posix_spawn (3)   spawn a process
master (8)        Postfix master process
popen (3) process I/O
_lwp_create (2)   create a new light-weight process

$ #output with new weights
$ apropos -n 10 create new process
fork (2) create a new process
fork1 (9) create a new process
_lwp_create (2) create a new light-weight process
pthread_create (3) create a new thread
clone (2) spawn new process with options
timer_create (2) create a per-process timer
UI_new (3) New User Interface
init (8) process control initialization
posix_spawn (3) spawn a process
master (8) Postfix master process}
```

Listing 4: Comparison of output with old and new weights

```
$ #output with old weights
apropos -n 10 -C remove packages #old weights
groff_mdoc (7)    reference for groffs mdoc implementation
pkg_add (1)       a utility for installing and upgrading software package distributions
pkg_create (1)    a utility for creating software package distributions
pkg_delete (1)    a utility for deleting previously installed software package distributions
deroff (1)        remove nroff/troff, eqn, pic and tbl constructs
pkg_admin (1)     perform various administrative tasks to the pkg system
groff_tmac (5)    macro files in the roff typesetting system
ci (1)    check in RCS revisions
update-binfmts (8)       maintain registry of executable binary formats
rpc_svc_reg (3)   library routines for registering servers


$ #output with new weights
apropos -n 10 -C remove packages
pkg_create (1) a utility for creating software package distributions
pkg_add (1) a utility for installing and upgrading software package distributions
pkg_delete (1) a utility for deleting previously installed software package distributions
deroff (1) remove nroff/troff, eqn, pic and tbl constructs
groff_mdoc (7) reference for groffs mdoc implementation
groff_tmac (5) macro files in the roff typesetting system
ci (1) check in RCS revisions
pkg_admin (1) perform various administrative tasks to the pkg system
update-binfmts (8) maintain registry of executable binary formats
rpc_svc_reg (3) library routines for registering servers
```

Listing 5: Comparison of output with old and new weights

feature was ignored for the classification models, and instead the `relevance` feature was used as the target output for training the models.

4.1 Performance of Classification Models

Mean accuracy was used as a metric to measure the performance of the *logistic regression* models. On the task of classifying the relevance of the results on a scale of 0 to 4 (with 0 being least relevant while 4 being most relevant), the classification models obtained an accuracy of about 53.06%. On reducing the number of classes to 3, the accuracy improved to about 62.04%. Finally, on reducing the number of classes to 2, namely, *relevant* or *not-relevant*, the accuracy jumped up to 80.41%.

However, the weights learned from classification models, do not show as great improvements in the output of `apropos(1)`, which suggests a disconnect between the problem and model. There are two main reasons behind the poor performance of classification models on this task.

1. While the accuracy on the binary classification task was good, it was too broad a category, with too few examples. The inputs in the dataset having relevance as 1 or 2 were clubbed with inputs having relevance as 0, while inputs having relevance as 3 were relabelled as 4. Reducing the examples from five classes to two classes increases the accuracy, since there are only two classes, even a blind guess has a 50% chance of being correct. But in reality, the small number of examples result in a very confused model. An example which was labelled as 2 in the training data, would be classified as 0, which is not really correct for real search results. On increasing the number of classes in the dataset, a sharp drop in the accuracy of the models was observed, which again suggests that the dataset was too small and not enough number of examples were observed for the model to properly learn to distinguish between different classes.

2. The second reason for the bad performance of the weights learned by *logistic regression* models on the output of `apropos(1)` is that the fact that *logistic regression* needs to learn k number of classifiers if the input data contains $k+1$ classes. And this means that the model ends up learning k sets of weights. But the ranking model used by `apropos(1)` cannot use multiple sets of weights, therefore training the model to predict more than two classes is futile unless suitable changes are made in the ranking algorithm as well. Reducing the number of classes to 2 ensures that the model only learns a single set of weights, but as noted previously, binary classification is a poor fit for the problem, unless more examples are added in the dataset.

5 Comparison of Results With New Weights

Listings 3, 4 and 5 compare the differences in the rankings of the results produced by `apropos(1)` with old weights and with the new weights learned from the regression model as described previously.

Listing 5 shows that for the query *"fork"*, the output with old weights contains results like `rshd(8)` and `rexecd(8)`, while in the output produced with the learned weights, it can be seen that `rshd(8)` disappears from the top 10 results and `rlogind(8)` moves to the bottom. At the same time, more relevant results like `vfork(2)` and `clone(2)` move further up, which is a significant improvement.

Similarly Listing 4 compares the outputs for the query *"create new process"*. It can be seen that the output with old weights contains irrelevant results like `supfileserv(8)` and `init(8)` at the top. However with the new weights the most relevant result to the query, `fork(2)`, moves to the top. More relevant results such as `pthread_create(3)` and `clone(2)` also show up in the top 10, while they were not present in output with the old weights, this is again a great change.

6 Further Work

There is still much scope in this area left to be explored. More data needs to be collected, which covers wider range of queries and man pages, a dataset more representative of the corpus would certainly help get better results.

Only classification and regression models were tried so far. There is a third variety of models, called *ranking models*. These models are specifically developed in order to learn a ranking function. It would be interesting to see the results of training these models on this dataset.

Another open avenue on this front is to try out different ranking algorithms. With the availability of a dataset it is much easier to try out new ranking algorithms and evaluate their performance. Metrics like *precision* and *recall* [9] can be computed to compare the performance of different ranking algorithms and to decide whether a new algorithm improves the search performance or not. Investigating different ranking algorithms in combination with the machine learning models is also an open area.

7 Availability

The datasets generated as part of this project and the associated models are available on the following github repository:
https://github.com/abhinav-upadhyay/man-nlp-experiments

References

[1] *Manual page for new apropos(1) in NetBSD 7.0* `http://netbsd.gw.com/cgi-bin/man-cgi?apropos++NetBSD-7.0`

[2] *Manual page for old apropos(1) in NetBSD 5.0* `http://netbsd.gw.com/cgi-bin/man-cgi?apropos++NetBSD-5.0`

[3] *Manual page for makemandb(8) in NetBSD 7.0* `http://netbsd.gw.com/cgi-bin/man-cgi?makemandb+8+NetBSD-7.0`

[4] *Manual page for makewhatis(8) in NetBSD 5.0* `http://netbsd.gw.com/cgi-bin/man-cgi?makewhatis++NetBSD-5.0`

[5] Upadhyay A.; Sonnenberger J. *Apropos Replacement: Development of a full text search tool for man pages* AsiaBSDCon, Tokyo p. 011-023, 2012.

[6] *The github repository containing datasets and python code for training models* `https://github.com/abhinav-upadhyay/man-nlp-experiments`

[7] Jones; K. S. *A statistical interpretation of term specificity and its application in retrieval* Journal of documentation, 28(1) p 011-021, 1972.

[8] Tom Mitchell *Machine Learning* McGraw Hill, 1997.

[9] Manning, Raghavan, Schütze *Introduction to Information Retrieval* Cambridge University press, 2008.

[10] *A web interface to NetBSD's apropos(1)* `https://man-k.org`

[11] *The mandoc project* `http://mdocml.bsd.lv/`

[12] *The FTS module documentation for Sqlite* `https://www.sqlite.org/fts3.html`