SearchEngine

The SearchEngine is a powerful tool for all kinds of linkages, especially the matching of large scale company databases. This manual will guide you through the installation, the functions of the graphical user interface (GUI) and their respective commands of the integrated script language. It describes exemplary use cases that may help you with designing your own search strategies and how to implement the SearchEngine Machine Learning approach SEML to filter out false positives from your matching results.

Discussion Paper

To prepare yourself for all the technical terms used throughout this document, it is suggested to read the discussion paper "The SearchEngine: A Holistic Approach to Matching" to gain insights about the search algorithm, applied heuristics and the general approach to matching projects. It is part of the Github package and can be found here: doc\HolisticMatching.pdf

A citeable version of the discussion paper is available on the ZEW homepage:

Doherr, Thorsten (2023), The SearchEngine: A Holistic Approach to Matching, ZEW Discussion Paper Nr. 23-001, Mannheim. [ftp.zew.de/pub/zew-docs/dp/dp23001.pdf]

Installation

The SearchEngine is a self-contained, portable Windows application. You do not have to install it. All necessary files can be found in the "SE" directory of the downloaded Github package. Copy the files or the whole "SE" directory to any location. The SeachEngine is always associated with the so called base table. This table will provide the universe of candidates retrieved by a search. This inherent connection should be acknowledged by choosing a location close to the base table, for example in a "SE" sub-directory of the base table directory. If you want to search in a different base table you have to copy the original files into a separate directory because every SearchEngine copy is locked in with one specific base table. As the base table provides the candidates, the search tables provide the search terms. All tables have to be tab-delimited text files with column headers. Preferably, the files should use extended ASCII or Latin-1 but UTF-8 encoding is also supported for Latin, Cyrillic and Greek characters. Before you can use multiprocessing you have to start the SearchEngine.exe application at least once as administrator. In case the SearchEngine complains at startup about missing DLLs, an assortment of potential culprits is available in the "DLL" directory of the Github package to be copied into the respective SearchEngine directory.

Preparation

Before the SearchEngine can link data sources it has to import them into its internal database system, which is the Advanced Visual Foxpro database. Your data has to be provided in the form of tab-delimited text files with column headers. To avoid unnecessary workload, it is recommended to create dedicated matching tables comprising only the fields required for the linkage. In general, the tables should already be part of another data management system to be able to handle the matching tables and the output of the SearchEngine. For that purpose, this system should be capable to import and export tab-delimited text files as this is the exchange format of the SearchEngine. Replace any control characters like tabulators (ASCII 9), line feeds (ASCII 10) and carriage return (ASCII 13) with blanks (ASCII 32) within all character fields before exporting the data. The internal databases of the SearchEngine only allow field names with a maximum length of 10 characters. Longer field names will be abbreviated by the import function.

It is also beneficial to remove redundancies in the form of duplicate entries when they exceed 10% of all observations. The SearchEngine will refer to matches using the record numbers of the matching tables. Not all data management systems allow to directly address records by their position, especially client-server based database mangagement systems (ORCACLE, MySQL, ...). You can attach a unique identifier to your data which can be used by the SearchEngine to address the linkages in its export formats. As the SearchEngine Machine Learning approach SEML will always only resort to record numbers, it is convenient to have the line number as additional field in your data.

Example: You want to match patent data to a company database by the patent applicants, which are usually firms. Let us assume an inefficient database structure with a high redundancy in regard of applicants because one applicant may have filed multiple patents. The first step would be the extraction of the applicant name, the address fields and a composed key consisting of the patent number and the position of the applicant in the patent document (in case there are multiple) into a dedicated table. You may apply filtering by country and data quality at this step. Replace any control characters in this table with blanks. Because you want to match applicants and not patents, it is sensible to create a condensed version of this table for the matching procedure. In SQL, this can be achieved with a "SELECT DISTINCT ..." command on the applicant name and the address fields (without the composed key field). This will leave you with the matching fields as linkage between the original table and the condensed table. Because you have lost the key, create a new one simply based on the line number. Of course, having the matching fields as linkage between the condensed table and the original table is not ideal. If your system has better solutions to remove duplicate entries without losing the connection to the original data feel free to use them. For example, STATA has a "group" command that creates a unique identifier for every entity. By keeping this identifier in the original and the condensed data, both tables can be joined with a simple number. Export the condensed matching table, consisting of a line number, the applicant name and address fields into a tab-delimited text file with column headers.
[importbase]
[importsearch]
[SEML]

Search Strategy

All search strategies for the SearchEngine are driven by the fact that the search algorithm is not commutative respectively symmetric. The search always has a distinct direction. A base table constitutes the heuristic allowing the SearchEngine to efficiently retrieve the records that match a given search term. The retrieved base table records are called candidates as there is still a chance for false positives given the search parameters. The search terms are provided by search tables, which are tables that share mutual data in the form of search fields like names, addresses or other fuzzy criteria. A central feature of this setup is that the retrieval is completely unaffected by surplus words in the candidates not addressed by the search term. In that regard, the SearchEngine algorithm behaves like a web search where a found web page only needs to match the search term words within its whole site. To stress this analogy even more, the SearchEngine also uses frequencies to assign relevance to every single component of a search term (see **discussion paper**).

To link two tables sharing a similar context, for example companies, persons, addresses, it is necessary that the representation of this context is congruent. If one table has multiple fields to represent an address with the sub-contexts street, city and postcode, while the other table has only one sweeping address field, you need to either separate the compound field or to conjoin the separate fields. Usually, having more sub-contexts is better but not always applicable considering the complexity of the underlying structure. When context congruency has been achieved, you have to decide which table should become the base table.

Focused Data Sources

There are two types of data sources. Focused data sources represent the search context in the best way possible. The source is reliable to prevent redundancies, the search context is also the in the focus of the source and the data is in general well maintained and curated. The sources are usually professional data provider. In terms of companies, these are administrative databases or commercial products like Orbis, Compustat, creditreform etc. The main feature of a focused base table is that only one candidate is to be expected for a concise search term due to unambiguous entity representation. Focused base tables allow to implement an **incremental search**, which is the preferred strategy.
[Incremental Search]

Unfocused Data Sources

On the other hand, we have unfocused sources where the search context is only a collateral, the data collection method is unsupervised or general diligence is missing. Patent offices collect company data as a by-product of patent documents and not as dedicated entities thus creating redundancy and variation. They usually do not even have separate identifiiers. Scraped web data does not fulfil uniform standards and amateurishly maintained databases contain noise and redundancies. Even diligent data sources have to be considered unfocused when they maintain subordinate entities to the intended search context, for example departments or establishments vs. firms. The main feature of an unfocused base table is that even for a concise search term multiple candidates have to be expected due to ambiguous entity representation. Unfocused base tables require to implement a **compound search**, which is the less favorable strategy.
[Compound Search]

Noise

Noise occurs mostly in unfocused data sources. Noise are non-relevant components in the search term diluting and diverging the search context. Typical instances are scraped web data where fields are misaligned, user content created without supervision, superfluous additions like marketing statements or sub-ordinate entities requiring additional information like departments of firms. A significant amount of noise may force you to use an unfocused source instead of a focused one because additional components in the candidates have no impact on the search success. Choose always the source with the highest amount of noise as base table. If this is a **focused** source, you can still implement an **incremental search**. Otherwise, implement a **compound search** on the **unfocused** base table. And finally, in case the extent of the noise is difficult to assess, you can always use two installments of the SearchEngine switching base tables and merge the results.
[Incremental Search]
[Compound Search]

Containment

Weak search terms, like a company name consisting of only a legal form in a large city, can generate humonguous candidate lists of false positives. Usually, you have expectations for the maximum of a plausible size of a candidate list depending on the underlying search context. An **incremental search** has smaller candidate lists per search term than a **compound search**. But even for a **compound search**, it is possible to give an educated guess for a realistic number of variations to a searched entity. To keep the results of a search within these boundaries, you can specify a **cutoff** representing this educated guess. In a candidate list, sorted by identity in descending order, the identity at the cutoff position will become a temporary threshold. Because candidate lists of weak search terms have usually no variation, a fact that would render this approach useless, the **cutoff** works together with **feedback** and **activation** to temporarily create variation for an efficient containment of inflated candidate lists. **feedback** is a mechanism that discounts identity according to the relevance of surplus words of the candidates while **activation** is just a trigger to declare the **feedback** as a support feature for the **cutoff**. All these settings are summarized under the command **contain**. Please browse through the documentation of these commands for a better understanding of containment.
[contain]
[cutoff]
[feedback]
[activation]

Creation

No matter the base table, be it focused or unfocused, the SearchEngine has to be created by selecting the fields of the base table to become search fields. Usually, there are search fields that directly define the search context and auxiliary fields complementing the identification of a searched entity. For example, companies are identified by their name and ambiguities eliminated by auxiliary address fields. The SearchEngine operates with search types, which are the combination of search fields and preparers. Preparers are directives guaranteeing uniform normalization,

harmonization and tokenization by imposing them on undiscriminatingly on base and search table fields. In many cases, specific preparers are not necessary as every field will experience basic normalization before the attachment of preparers. Some preparers implement methods from the field of computational linguistics, which destroy, dilute or disperse information to increase robustness towards misspellings. Those preparers are called "destructive" and the search types using them are called destructive search types. They improve the recall at the cost of precision and performance. They should only be attached to relevant search fields and only in addition to a conventional search type for the same field. In most cases, your search strategy will have a search run not relying on auxilary search fields to accommodate matches beyond the confines defined by them. For company data, this would cater for relocations of companies. It would be a waste of resources (your time) to tackle misspellings in those fields, when you include a search run that completely renounces them. You should consider the identity only as a means for retrieval to be tested against a threshold. There are better ways to gauge the quality of a match, namely the SearchEngine Machine Learning approach **SEML**, which will include string metrics tolerant to misspellings anyway. You should concentrate your efforts (and the efforts of the SearchEngine) on retrieval and less on fine tuning of the identity.
[create]
[Preparers]
[SEML]

Incremental Search

The incremental search is based on the notion that for every search record in the search table there is only one candidate in the base table or none at all. The entities in the base table match exactly the purpose of the search and are professionally maintained. In the base source each entity has a designated identifier to prevent redundancies in the form of duplicates or variations. Still, even if the base source is in exemplary condition the context itself may impose ambiguities. Companies may have subsidiaries or branches with slightly different names, persons may have namesakes and so on. Due to these natural ambiguities multiple candidates per search term are to be expected even though the entities are perfectly resolved therefore you should give the **containment** some leeway. For focused data sources, you can skip the aggregation of duplicates during the **preparation** at your own discretion.

The main characteristic of an incremental search is the **darwinian** setting. During retrieval, only the candidates with the highest identity will survive. We do not want the second best candidates. When the base table has an almost perfect entity representation the second best candidates have to be different entities. An incremental search always consists of multiple search runs whereby the first run is the most restrictive one, enforcing high quality standards on the retrieved candidates, while the parametrization is relaxed with every subsequent run. During such a run, the SearchEngine will skip all search records that already have candidates in the result table retrieved in earlier runs with more demanding settings. The initial run should use only conventional search types because destructive ones are less demanding and will be used in later runs. The **threshold** should enforce that next to the main field(s), representing the search context, a good share of the auxiliary search fields have to match. For example, next to the firm name field at least half of the address fields have to match. The following runs keep the **threshold** but replace the conventional search types with destructive, linguistic search types to capture misspellings and typos. Only after that, search runs should relax the requirement of matching auxiliary fields by lowering the **threshold** slightly below the sum of the weights of mandatory fields. In context of firms, we now allow candidates to be at a different location but also grant more leeway for the name at the matching location. By switching weights from conventional to destructive search types we complete our relaxed search with typo-tolerant runs. If there are only few search records left without candidates, you can even dare to conclude with a **zealous** search run ignoring the **threshold**.

The following example shows a complete script of an incremental search of a patent applicant table in a focused company base table with an additional 3-gram search type on the firm name:

```
importbase("c:\orbis\firms.txt")
create("firm NOABBREV, firm NOABBREV GRAM3, street SEPNUM, zip, city")
importsearch("c:\patents\applicants.txt")
result("c:\patents\applicants_result")
join("applicant", "firm")
join("street")
join("postcode", "zip")
join("city")
reset()
contain(5)
darwinian(.t.)
threshold(85)
types("firm 70, firm 0, street 10, zip 10, city 10")
search()
types("firm 0, firm 70 log, street 10, zip 10, city 10")
search(1, 1)
types("firm 0, firm 70, street 10, zip 10, city 10")
search(1, 1)
threshold(65)
types("firm 70, firm 0, street 10, zip 10, city 10")
search(1)
types("firm 0, firm 70 log, street 10, zip 10, city 10")
search(1, 1)
types("firm 0, firm 70, street 10, zip 10, city 10")
search(1, 1)
```

This is a basic setup for an incremental search. You can also experiment with weight shifts toward the auxiliary fields, i.e. address fields, to provide more leeway for the main fields. Use the **statistics** function to control your progress through the search table as any search record with a candidate is solved. Still, even with an incremental search, false positives cannot completely be avoided, especially when the base table does not represent the full search population. In case a full manual verification is not possible, the SearchEngine Machine Learning approach **SEML** is the next logical step.
[Focused Data Sources]
[search]
[darwinian]
[threshold] [statistics]
[SEML]

Compound Search

A compound search is necessary when the base table has no clear representation of entities but is mere collection of occurrences. It is expected that a search term will retrieve multiple candidates, which are all variations of the searched entity. Compared to an **incremental search** on a **focused data source**, the risk of retrieving candidates representing different entities is also much higher due to the general ambiguity. In a sense, a **compound search** is much easier to implement as an **incremental search** as the only decision is about the height of the **threshold**. Because you have to expect a multitude of candidates of variable identities for every search term, the task is to balance the recall of all potential variations with the risk of attracting false positives. You have to adapt your strategy to the deficiencies of the base table, especially if **noise** is involved. In general, you will only perform search runs necessary to capture the issues with the data like partially missing fields but it does not make sense to gradually decrease the retrieval requirements if the results of runs will be merged anyway. The **containment** should be more generous to accommodate the expected variation of valid candidates or skipped completely if that number is difficult to assess. Noisy base tables may have more missing entries in the search fields, e.g. address fields, requiring to add search steps that ignore those fields. To achieve a consistent identity, it is advisable to impose an adjustment with the **research** action afterwards. If the search table is from a focused source, this fact can be exploited by stripping the results retroactively with the **strip** action imposing an inverse darwinian cutoff keeping only the best search term(s) per candidate. Conversely to an **incremental search**, this can only be done after all search runs instead of during the search. Such an approach is only viable with harmonized identities achieved by a final **research** action introducing a **feedback** effect.

The following example script matches a focused company database with the affiliations of a bibliometric database with a lot of **noise** and partially incomplete addresses. This search does not use linguistic search types due to the unfavorable relation between complexity and potential recall gains:

```
importbase("c:\scopus\affiliations.txt")
create("affil NOABBREV, street SEPNUM, postcode, city")
importsearch("c:\orbis\firms.txt")
result("c:\orbis\firms_result")
join("firm", "affil")
join("street")
join("zip", "postcode")
join("city")
reset()
threshold(75)
types("affil 70, street 10, postcode 10, city 10")
search()
types("affil 100 log, street 0, postcode 0, city 0")
threshold(95)
search(2)
types("affil 70 log, street 10, postcode 10, city 10")
save("orbis")
unjoin()
join("firm", "affil")
feedback(20)
research()
unjoin()
join("street")
join("zip", "postcode")
join("city")
feedback(0)
research(4)
load("orbis")
strip(.t.)
```

This strategy example consists of two search steps. One requires at least some matching parts of the address with a threshold of 75% and a second one ignoring the address fields but with a higher threshold of 95%. Of course, an identity of 85% of the first run cannot be compared to the same identity of the second run. To harmonize the identities, we redistribute the search type priorities and apply a feedback of 20% only on the name by unjoining the address fields and calling the research action. To complete the identities in the result table, which now have a maximum of 70%, we unjoin the name and rejoin the address fields. The research action is additive and without feedback. By loading the previously saved settings, the original setup is restored and all fields properly joined again. The strip action conducts the inverse dawinian approach to exploit the fact of having a focused search table. The result table will be stripped of redundant linkages.

The **compound search** retrieves much more false positives than an **incremental search**. For large search projects that cannot be validated manually, it is recommended to process the results with the SearchEngine Machine Learning approach **SEML**.
[Unfocused Data Sources]
[Focused Data Sources]
[Incremental Search]
[Containment]
[Noise]
[search]

[threshold]
[threshold]
[feedback]
[research]
[strip]
[SEML]

Self-referential Search

Of course, the decision, which table should become the base table, is trivial when base and search table are the same. Self-referential searched have the purpose to identify duplicates in a table or to create clusters of representing similar entities. The fact that duplicates or ambiguous entity representation is expected in the data designates it as **unfocused data source**. An **incremental search** would be pointless as every search record will find itself as best candidate. A **compound search**, on the other hand, should be more restrictive as in the normal case of linking two different tables. Because candidates and search terms are intermingled, the resulting tuples of search and base record define edges of an intransitive similarity network. This requires the dissolving of search term and candidate relationships into cluster memberships by traversing the network along the edges to identify cohesive areas. Having to many misleading edges in such a network, due to a less restrictive **compound search** will only bog down this process. Besides the restrictiveness, the search is not different to a normal search. The major difference is the post-search treatment of the results, which have to be clustered with the **exportgrouped** function implementing a rule based clustering algorithm.

In the following example a self-referential search is conducted on noisy company data scraped from webpages with downstream clustering:

```
importbase("c:\big_haul\scaped_companies.txt")  importsearch("c:\big_haul\scaped_companies.txt")
result("c:\big_haul\scraped_result")
create("name NOABBREV, name NOABBREV GRAM3, address SEPNUM")
join("name")
join("address")
contain(20)
threshold(85)
types("name 90, name 0, address 10")
search()
types("name 0, name 90 log, address 10")
search(2, 1)
exportgrouped("c:\big_haul\generous_cluster", "min >= 90 @ 41", .f., .t.)
```

This is a simple example for a clustering rule allowing clusters up to a size of 40 based on intransitive connections, which is higher than the **containment** to include overlap between contained lists. Beyond that limit, cluster have to be based on bi-directional connections with at least 90% identity. Because clusters have to be independent from the starting node, this means a complete re-evaluation of the current cluster. As this is a very generous clustering, the export table will only report clusters with at least two members and the search fields for quality control. Assuming that the first effort was not satisfactory, a more complex ruleset based on the enforcement of bi-directional edges is applied:

```
mirror()
types("name 90 log, name 0, address 10")
research("3")
exportgrouped("c:\big_haul\tight_cluster", "min >= 90 @ 0; min >= 70 @ 0, min >= 70 and p >= 80 or min >= 85
@ 11, min >= 90 @ 21", .f., .t.)
```

The bi-directional edges are enforced with the **mirror** command and supplied with identities by the **research** command. The first ruleset creates a interim network of clusters as hyper-nodes with a high similarity to harmonize

the variation. The second ruleset, after the semicolon, clusters the harmonized hyper-nodes by referencing identities below the threshold introduced by the enforced symmetry and using the percentiles of the absolute identification potential as additional quality measure. This so called **nested cascaded traversal** is an experimental design requiring experience with the data based on try-and-error. For more information, please read the **discussion paper** and the **exportgrouped** command description.
[Discussion Paper]
[Unfocused Data Sources]
[Compound Search]
[exportgrouped]

SEML

The SearchEngine Machine Learning approach facilitates the quality management for large search projects. The SearchEngine is quite capable to linking very large datasets by fuzzy criteria. But, given the underlying search strategy, it may retrieve too many false positives in an effort to avoid false negatives. Although, the balance of recall and precision is a fundamental element of all linkage endeavours, it is very difficult to maintain. Avoiding false positives at any cost would increase the amount of unobserved false negatives due to constrains imposed by an engine specialized in retrieval. Apart from the fact that ex-ante definitions for such a balance are impossible to stipulate, it is significantly less complex to filter false positives from the matches, a job that can be performed manually by browsing through the data guided by a sense for the objective of the match. Of course, for a large search project this is not a feasible approach.

The SearchEngine combines the simplicity of a manual validation with the efficiency of machine learning. The fundamental approach is to label training data, which is a relatively small sample of the full data, to identify an implicit ruleset derived from meta data providing variation correlated with the decision process behind the labeling. The SearchEngine has the function **exportmeta** to export the meta data of a result table. A sample can be drawn with the **exportresult** command. The **exportextended** function presents the results of the sample in an easy to label format. After labeling, the sample is enriched with the corresponding meta information to initiate a machine learning process, usually the training of a neural network. After the training, the neural network is applied on the complete meta data to convey the trained behavior on the matches. The final result is a dataset where true positives are separated from false positives without the need of painful compromises regarding the balance of recall and precision.

The SEML approach is the reason that the **reseach** and **refine** functions are only used in the context of the refinement of candidates retrieved by destructive search types or **self-referential searches** to provide an identity for mirrored connections. Before the introduction of SEML, the identity was the most important indicator for the quality of a match but still too unreliable for the identification of false positives. With the SEML approach, there is no need to readjust a single parameter like "identity" when the meta data provides much more information.

Meta Data

The meta data is exported with the **exportmeta** function. The main component is based on the absolute identification potentials (AIP) of every search type. An aggregate version of this potential can already be found in the result data under the field name "score". The AIPs are reported per search type for the matching words, surplus words of the candidate (exclusive to the candidate) and the words of the search term that did not match (exclusive to the search term). For every search type and category only a given number of AIPs will be reported. This number reflects the most relevant words required to identify an entity by the respective search type on average. For example, firm names may require 5 words, street names can be identified by the 3 most relevant words while a postcode consists of only one word anyway. For destructive search types based on tokenization, like n-grams, we suggest to multiply the corresponding word based number by 3 or 4. Search types not used for retrieval can be omitted. The definition of these numbers is a central parameter of the **exportmeta** function (see the command description for details).

Another setting for the meta export is the aggregation of search runs. Especially for an incremental search, the number of candidates retrieved per run can be extremely skewed. Use the **statistics** function to get an overview of the run distribution. A heavily skewed distribution has implications for the representation in the training sample. You have the option to aggregate search run indicators according to the characteristics of the used search types. Usually, search runs based on conventional search types can be separated from runs based on destructive search types. But also different smoothing methods (see **types**) provide potential aggregation opportunities. The aggregated runs should represent candidates sharing specific characteristics caused by the underlying retrieval, like misspellings or search type weight distribution. As aggregation should only mitigate severe underrepresentation of run indicators, one could argue that it is actually not necessary to put much effort into it. Especially regarding the fact that this is only one component of many in the meta data. Nevertheless, the **exportmeta** function has an optional parameter for this purpose.

The meta data also contains string distances, overlap indicators across fields and co-candidate related statistics. It provides a framework of parameters creating variation that may be correlated with the validity of a match. Additional statistical parameters can be derived from the exported meta data to enrich the variation with standard deviations of string distances over all candidates of a search term (see **Machine Learning**)
[exportmeta]
[statistics]

Training Sample

It requires four steps to export a training sample. First, use the **exportresult** function to export the sample based on an absolute number or a share of the search terms (records) represented by the current result table. A sample of 1000 search records will consist of all candidates retrieved for those records. Second, replace the current result table with the sample by assigning it with the **result** function. These two steps are combined in the GUI window **File>Export>Result Export**. Now, you can export the training sample with the **exportextended** function, which creates a text file in a convenient format. Before we commence with the labeling, in the last step we restore the original result table by replacing the sample with the **result** function. Following this procedure, you can export multiple samples if you intent to distribute the labeling workload. In general, a sample sizes between 1000 and 2000 should suffice. This is the equivalent to one or two lazy afternoons of menial work, which you can transfer onto millions of matches. Of course, the SEML approach is not suitable for small search projects below the size of a robust training sample.

The results may have a very skewed distribution with many candidates allocated to few search terms. You can check the distribution with the **statistics** function. To capture the outliers, it can be beneficial to draw a complementary sample weighted by the number of candidates especially in the case of a **compound search** without **containment**. Be cognizant that those samples are usually much larger and therefore require a smaller sample size pertaining search terms. The provided machine learning scripts can handle multiple training datasets.

Import the exported training sample into any spreadsheet tool of your choice. The file is separated into blocks. The header of a block is the search term followed by the associated candidates. The fields "searched" and "found" refer to the record numbers in the base and search table. The labeling will be carried out in the "equal" field. Enter a "1" for a valid match (true positive) and a "9" for a wrong assignment (false positive). It is strongly discouraged to use a "0" in that context because it is associated with default values. You can reduce the typing by using the "equal" field of the search term line (header) as the default value for the whole block. Exceptions to the default value of the block and be marked in the "equal" field of the respective candidate ("9" default, "1" exception and vice versa). Another way to improve efficiency is to declare a sweeping default value for the whole training data based on the general tendency, i.e. all matches are defaulting to true positive when they are in the majority. Do not forget to realize this implicit rule in the data after labeling. The training script has a convenience setting for that purpose (see following section).

Do not expect wonders from the machine learning. The meta data does not carry any semantic information. It can only derive rules from consistent labeling that does not include too much human intuition into the decision process. If you accept subsidiaries and branches of a company in a firm match but exclude cantinas and other service oriented subsidiaries, there may not be enough information in the meta data too capture this behavior. If you have additional data available about the matched data sets related with the search context, you can avoid to confuse the AI by applying a rule based filtering process after the SEML approach. Remove all "true positives" with specific industry

codes or create a rank among the surviving candidates for a search term that favors your intention. Assigning a patent applicant to the largest company among multiple candidates will also assign all patents to the parent company, which is the legal owner. In short, be as generous with the assignment of true positives as you can afford considering subsequent filtering or ranking options.
[File>Export>Result Export]
[File>Export>Extended Export]
[exportresult]
[result]
[exportextended]
[statistics]
[Compound Search]
[Containment]

Machine Learning

The Github package brings its own machine learning tool. You can find it in the sub-directory "SEML". Unfortunately, it requires STATA, a commercial statistical software. Of course, you can implement your own machine learning approach based on the meta and training sample. In that case, ignore the STATA references and derive the general steps. The "SEML" directory consists of two modules: "seml_train.do" for the machine learning part and "seml_think.do" to transfer what it learned on the whole meta data. The scripts require the "brain" module, which can also be found in the "SEML" directory. To install this module, you can copy its files into the ADO folder of your STATA installation or directly to the script file location. The latter will restrict the usage of the module on this directory. It is also available at the Statistical Software Components archive and can installed with the STATA command "ssc install brain". For more information about the "brain" module, call its help file. Copy the "seml_train.do" and "seml_think.do" scripts into a dedicated directory, usually also called "SEML", together with the meta data text file and the training sample.

The training sample should be exported from the spreadsheet tool as tab-delimited text file with column headers. The training script expects that the file name of the training sample contains the word "sample" and ends with ".txt". You can have multiple training files in the script directory as long as they follow the naming convention, i.e. mysample.txt, sample1.txt, sample2.txt, exportsample.txt and so on. Make sure that no other files complying with this convention are in the directory. The sample file must have the fields "searched" (search table record number), "found" (base table record number) and "equal" (coded 1 or true positive, 9 for false positive). The "seml_train.do" script will automatically parse the labeling convention as described in the **Training Sample** section by imputing empty/missing "equal" fields (zero is considered missing) with the value found under the respective header line for the search term. Of course, you can fill in the "equal" column according to your own conventions as long as they are complete. Lines are dropped when the "equal" field is still not 1 or 9 after that procedure. The script employs a global macro called "default" in the "Settings" section for your convenience. You can set it to "1", when the global default for the sample file(s) is "true positives", and to "9" for "false positives" as default for missing candidate block defaults. The script will impute all missing header assignments accordingly. Leave it at "0", when you have not used a global default, which has to be the same for all sample files.

The meta data has to be (re)named "meta.txt" to be recognized by the "seml_train.do" script. The "seml_train.do" script will first import the "meta.txt" file and calculate some additional parameters which could no be calculated with the **exportmeta** command. These are the standard deviations of the fields with the prefix "csf" and "cfs" containing string distances between search term (searched) and candidate (found) of the search fields. The standard deviations are clustered by the search record field "searched" and imputed with zero where only one candidate exists for a specific "searched" ID. These additional fields are not mandatory and can be skipped in your own approach. The imported file will be saved under "meta.dta". The meta data and the training sample are merged into the training data keeping only the overlap of the sample. In the "equal" variable, the nines (9) are finally transformed into a zeroes (0) as, beyond this point, misinterpretation is of no concern. The "seml_train.do" script retains 10% of the training data by marking it as "ground truth", reserved to test the performance of out-of-sample prediction. This selection will not be used for training/learning. The file "seml_train.dta" will contain the final training data. If changes to the meta data or training sample require a new training, you have to delete the "seml_train.dta" file before running the script. Delete the "meta.dta" file only if changes to the meta data have occurred. The training script can be modified to change the size of the retainment or to aggregate search runs (see comments in the script).

Before the training, the script will perform a conventional, linear probabiliy model (OLS) to get an early impression of the training data coherence and to provide a benchmark for the machine learning. The "seml_train.do" script now iterates through several neural network hidden layer setups alternating between raw and weighted outputs. When the output is weighted, a virtual balance between true and false positives is established. The setups range from 25 hidden neurons to a doubled layered network with 100 hidden neurons each. The script uses a decaying hyper-parameter **eta**, that will be halved very time the last 100 training iterations did not improve the network error, until progress has converged. This may take longer than necessary but is fail proof. After every setup, a confusion matrix with recall, precision and accuracy rates is calculated with the "ground truth" to determine the out-of-sample performance. The best network, according to the accuracy, is saved into a so called brain file with the name "seml.brn".

Call the "seml_think.do" script after checking the log file of the training script to verify the prediction quality. The script will read the transformed meta data file "meta.dta". It marks the records used for training in the variable "train". The neural network module will read the brain file "seml.brn" containing the network structure. The thinking process will predict the "equal" variable for all records in the meta data. According to the convention, a "1" (one) is a true positive while a "9" (nine) is a false positive. The probability for a true positive is saved in the variable "brain". If "brain" is above 0.5 the match between a "searched" and a "found" record is considered a true positive. The output is file called "think.dta" and its tab-delimited equivalent "think.txt" containing the "searched" (search table record number), "found" (base table record number) and the mentioned fields "equal", "brain" and "train". Keep in mind that the record numbers do not include header lines (1 is always the first data line).

GUI

The SearchEngine has a Graphical User Interface you can bring up by simply double-clicking the "SearchEngine.exe" file. Its main screen will always show you the so called structure string, which is a formatted representation of the current SearchEngine settings. This string also serves as storage format for your search projects. While you are in the GUI mode, every change to a setting or any action will be logged in the file "searchengine.log". You can open this file with any text editor to peruse the associated commands. These commands can be issued in the command window (see **File>Command**) or composed into a script to be called in the **Batch Mode** of the SearchEngine. The GUI is the first entry point to learn the script commands for a more efficient handling of the SearchEngine.

The main menu has the following sections:

File

- saving and loading of settings
- export of results, meta data for machine learning, result table extraction and sampling
- command window

Config

- base, search and result table declaration and field association
- search type priorities, weights and smoothing
- search settings
- general preferences

Action

- searching
- identity manipulation with refine and research methods
- stripping the results by retrocatively imposed search requirements

- mirroring of results for self-referential searches
- creating the index files for the search
- registry adjustment

Tools

- interactive quick search
- quality control of results
- result statistics
- misc.

The following chapters will describe the menu items and the purpose of the invoked windows. Because every window is just a template to compose the commands to be called in the background, the descriptions will only provide a context for the command calls. Refer to the associated commands for further details. If items are greyed out in the GUI, they are not applicable on the current state of the SearchEngine, e.g. "Grouped Export" will only be available when search and base table are the same (self-referential search). The different selection lists may not be in the order of the parameters of the associated commands.

File

This menu provides functions allowing you to manage your search project files and to export the result table for manual labeling and quality control. It is also the starting point for machine learning by providing access to the meta data export function and the result table handling to draw samples. Finally, you can open a command window to manage your SearchEngine scripts, directly execute script commands or browse the SearchEngine.log file.

File>Save Settings

This window lists the existing search projects. You can click on a project to overwrite it or enter a new name for a fresh project. The current settings will be saved under the specified name. You should always start a new project by saving the current settings under a new project name to make sure that any changes to the settings do not inadvertently affect another project. You can also load a similar project as a template before saving the setting under a new name. At startup, the SearchEngine will always load the project that was saved last. This is also the project name that will be selected by default when opening the save dialog. You can also delete a selected project except the "Default" project, which contains persistent information about search types.
[save]
[remove]
[list]
[slot]

File>Load Settings

This dialog shows the available search projects. You can load a selected project. A project only consists of the structure string as displayed on the main screen of the GUI. You will not be able to revert any major actions affecting the result table or SearchEngine index tables to a previous state by reloading a project. A project file contains only administrative information about search settings, file paths and preferences. It does not contain tangible data.
[load]
[list]
[slot]

File>Export>Export

Use this option to export the result table into a simple format where every record represents a linkage between a search table record (search term) and a base table record (candidate).
[export]

File>Export>Extended Export

The extended format is intended for manual labeling and quality control of the results. It is organized into blocks with the search term in the headline followed by the matched candidate. A dedicated column is reserved for labeling providing an efficient way to separate false from true positives.
[exportextended]

File>Export>Grouped Export

The grouped format is specialized in the export of self-referential search results where the search and the base table are the same. Because the results of such a search constitute a non-transitive similarity network, the grouped export implements a rule based clustering mechanism.
[exportgrouped]

File>Export>Result Export

With this export function copies of result tables can be created that can be assigned as new result tables for the SearchEngine. Search runs, designating the different runs of a search strategy, can be aggregated and filtered to improve representation in samples, which can also be drawn to provide an easy way to extract training data for the SearchEngine Machine Learning approach (SEML). Just draw a smaller sample of the original result table as interim result table to export the training data in extended format.
[exportresult]

File>Export>Meta Export

You can export meta data about all matches in the result table providing variation potentially explaining the separation of false from true positives. The meta data is also the foundation for the SearchEngine Machine Learning approach called SEML. A neural network or any other machine learning device is trained based on the meta data and a training data set. The meta export has specific features to define a concise data framework to prevent over-specification.
[exportmeta]

File>Command

The command window allows to manage SearchEngine script files by specifying them in the file selection field. The can be loaded or saved using the corresponding buttons. A loaded script can be edited in the main command area. By changing the file name it is possible to save the script under a different name. By starting with a fresh command area and a new file name a new script can be created. The file searchengine.log is reserved for the SearchEngine reporting and cannot be overwritten. Still, this file can be opened in the command window as a template for scripts and saved under a different name. To execute commands in the script, you have to select them with the mouse or with shift plus cursor keys and press the "Run" button or the shortcut key combo (**ctrl + enter**). Any command that is marked, even only partially, will be executed. The command area will switch to the execution area where you can observe the progression of the select script commands and their output. To switch between the command and the execution area without performing any actions make sure that nothing is selected in the command area by simply clicking into it or moving the cursor without the shift key. The command area behaves like any other editor in that regard. To execute a complete script you have to mark all lines, which can be achieved the fastest with the (**ctrl + a**) key combination.

You can access this help document with the **help** command. By specifying a header only the corresponding section will be shown in the execution window. By clicking on links, which are enclosed in [brackets], referenced sections will be displayed.

The command window is not the only way to execute scripts. You can also call the SearchEngine from the command line with a script file as parameter to initiate the **Batch Mode**.
[Script]

[help]
[Batch Mode]

File>Exit

This command closes the SearchEngine application. If you have changed the settings till the last save, it will ask you to save the settings under the current project.

Config

In this section, all parameters for a search can be specified starting with the definition of the base table. Unless the SearchEngine is not created using the base table, all other search related options in this section are not available. You can define the search tables to be searched in the base table and the associated SearchEngine index files and the result table. Every search project should have its own result table. After the file specifications the linkage of the search fields of the base table and the corresponding fields in the search table is another required step in the configuration of a search project. The priority redistribution of the search types is an integral part of any search strategy and the corresponding menu will often be consulted multiple times over the course of a search project. The search settings define the key parameters of a search step like the general threshold for the identity, cutoff points and activation limits for the feedback to contain the inflation of weak search terms and switches for specific SearchEngine heuristics. Finally, you can change general preferences of the technical aspects of the SearchEngine like the search depth, location of temporary files and multi-processing.

Config>File Locations

You should only specify the base table once per SearchEngine installation. The search tables will always be searched within the same base table. Depending on the setup, this may be a singular search project between two tables or a extensive framework catered for a multitude of search projects on a central base table. You should also assign a dedicated result table for every search project aka. search table. You can click on the "Auto" button next to the result table field to create an appropriate file name. The search and base table will be automatically imported into the database format of the SearchEngine if this conversion has not occurred yet. The result table will always be in Foxpro format (.dbf). The import option to truncate outliers can be activated if you consider field lengths beyond 254 characters as data artefacts.
[importbase]
[importsearch]
[result]

Config>Join Search Fields

With this window you can join the search fields, which are the fields of the base table used to create the SearchEngine index files (left list) and the search table fields of the currently assigned search table (right list). Make sure that the joined fields have the same context. Depending on the setup, it may be necessary to split composed fields in the search table, e.g. a sweeping address field into separate street, city and zip fields. It that seems to complicated, a new SearchEngine installation on a modified base table with the same composed field structure as the search table is much easier to implement. Not all search fields of the base table need to be joined. Free search fields have no impact on the search as long as the priorities of the associated search types are zero.
[join]
[unjoin]

Config>Search Types

The search type priorities constitute the search strategy. The priorities will be transformed into percentage weights by normalization. It is good practice to already use percentages as priorities. Keep in mind that you can specify priorities for all search types regardless of them being joined with search table fields or not. Beyond priorities, this is the place where you can specify the different smoothing methods, which are called: offset, log and softmax. A search type is deactivated when its priority is zero. It can be part of a search strategy to alternate between different search types on

the same search field, usually a conventional and a linguistic search type, by switching the zero priority.
[types]

Config>Settings

These settings determine the behavior of the SearchEngine during a search. The paramount setting is the definition of the threshold as the limiting factor for the identity of candidates to be transferred into the result table. The cutoff declares the maximum expected number of candidates retrieved per search term. Feedback is applied when the candidate list meets the activation limit. Feedback discounts the surplus words of the candidate in accordance to their identification potential. When activation and cutoff are both specified, the feedback will only be used to create variation for a close cutoff. This setting combination applies, it is acknowledged as **Containment**. In darwinian mode, the SearchEngine will only consider the best candidates. With the relative switch, the identification potential of missing search table fields can be redistributed on existing fields without loss. An ignorant SearchEngine dismisses unknown words not represented in the registry (base table) as non-existent instead of giving them an average frequency and thereby a loss for the overall achievable identity. A zealous SearchEngine will lower the threshold dynamically to assure matches.
[threshold]
[limit]
[cutoff]
[activation]
[feedback]
[darwinian]
[relative]
[ignorant]
[zealous]
[Containment]
[contain]

Config>Preferences

The preferences set technical aspects of the SearchEngine. The search depth determines maximum frequency of the least frequent word of a search term to be valid. If every single word of a search term occurs more often in the base table, the search term will be skipped. The scope defines the elasticity of the string distance function LRCPD used to refine candidates retrieved by destructive (linguistic) search types. The location of temporary files can be specified in case the default location is too limited. The preferences windows also provides access to the multiprocessing control, a benchmark and the logging of the execution time of actions.
[depth]
[scope]
[configtmp]
[mp]
[benchmark]
[timer]

Action

This main section controls the major actions of the SearchEngine starting with the search itself. This action will retrieve the candidates according the search types and the search settings. The identity of already found candidates in the result table can be adjusted with the research action carrying out a "what-if" scenario based on the current settings. The research action ignores all settings related to controlling the size of candidate lists like cutoff or activation. With refine the identities can be reappraised using the string distance metric LRCPD to replace the frequency based heuristic.
The actions Strip and Mirror affect an existing result table by either stripping unwanted or redundant results or by mirroring a result table after self-referential searches. The latter provides, in conjunction with the research action, the bi-directional edges for the grouped export.

These search related action will be disabled unless the SearchEngine index files, consisting of the registry table and linkage files, have been created. Those files can be recreated if a misconception regarding the search type setup has occurred. In case a search requires a different search type setup, but the existing setup is still useful, a fresh installation into a different directory would be the more sensible procedure. A rarely used action is the expansion of the registry with the frequencies of the search table because the consequences are difficult to assess and control.

Action>Search

This is the main action of the SearchEngine. It will create the result table or expands an existing one with the candidates of a new search run. The general attitude of the SearchEngine can be adjusted in this window. The options to enforce refinement and to readjust the threshold afterwards are rarely used. The separate threshold is only available if refinement is initiated, either by the inclusion of destructive preparers or by enforcing it. The direction of the refinement can be specified if applicable. The most important setting is the interaction of the new results with the existing results. This determines if the search strategy is incremental or compound.
[search]
[Search Strategy]

Action>Research

The research action recalculates the identity according to the current settings without retrieving new candidates. It ignores all settings that restrict the retrieval like threshold or cutoff. The research can be limited on specific runs, i.e. the mirrored run of a self-referential search. There are options to interact the new identity with the existing identity. Because the absolute identification potential, called score, depends on the priority setting of the search types, it is also subject to changes requiring a mode of integration. The research action can be restricted to search types without destructive preparers. This option is intended to be used with the reverse option of the refine action. Because this specific usage of the action is already integrated into the search action, this option is deprecated.
[research]

Action>Refine

The refine action reappraises the identity with the string distance function LRCPD. No new candidates will be retrieved. The action can be restricted on specific runs. You can specify how the distance metric will replace the existing identity. The direction of the comparison can be specified. Because the refine action was used to re-evaluate destructive search types to contain their volatility, there is an option to restrict the refinement on destructive search types. This was a cumbersome process requiring the manipulation of priorities. It is now completely integrated into the search action declaring the option as deprecated.
[refine]

Action>Strip

The strip action allows you to retroactively (after the the search) impose a (higher) threshold or to cutoff on the results. The action can be restricted on specific runs. Complete runs can be removed with an unachievable treshold for selected runs. An inverse cutoff picks only the best search terms per candidate (instead of the best candidate per search term) to exploit the benefit of a curated search table. Run numbers can be changed or aggregated.
[strip]

Action>Mirror

The result table of a self-referential search is comparable to a network definition. Every candidate will also be a search term because search records and candidates share the same table. Because of the threshold and the non-symmetric search algorithm, not every candidate of a search term will retrieve the search term as a candidate. With the mirror action, symmetry can be enforced by creating the missing reverse incidents. All creates candidates have an identity (and score) of zero. To assign an identity to the mirrored cases you have to use the research action specifying the mirror run. This will provide some information about how close the candidate is, especially when using smoothed

search types. This information can be exploited for the cluster rule definition of the Grouped Export.
[mirror]

Action>Create

This action creates the SearchEngine by parsing the base table, collecting the words according to the search types to register them in the registry by counting their frequencies. During this process, linkage tables maintain a tight connection between the registry and the base table records. To define search types you have to assign preparers to base table fields. A preparer is a specific directive declaring how fields are normalized, beyond the basic normalization, and potentially tokenized. By default, all characters are transformed to upper case and special, non-alphanumeric characters are replaced with blanks. The SearchEngine understands most derivatives of European alphabets including nordic languages, greek and cyrillic. A word is everything separated by a blank after normalization. This definition can be changed using linguistic methods implemented by so-called destructive preparers. In general, you will only attach non-destructive preparers to your base table fields and only the most context relevant fields should get a second search type based on destructive (preferably GRAM) preparers. You do not want to waste time handling misspellings in auxiliary fields with a low priority.
[create]
[showpreparer]

Action>Recreate

This action will delete an existing SearchEngine and create a new one. The dialog is identical to the search action. Consider to install a new SearchEngine in a different directory instead, if you just require a different search type setup while the existing SearchEngine is totally fine.
[create]
[erase]
[force]

Action>Expand

This action will use the current search table to reassign the frequencies of mutual words in the registry. Because the consequences of this action a difficult to assess, this feature is of a more experimental nature. You will always be able to undo this action with the "Rebuild Occurrences" option, which will use the internal linkage tables to restore the original frequencies.
[expand]

Tools

You do not need them but you also cannot live without them. They allow to test the SearchEngine setting by manually entering search terms, to easily browse through your search results without exporting, to get valuable statistics about the distribution of candidates over search runs and to browse internal tables like the registry for educational purposes.

Tools>Quick Search

To quickly search for a specific entry in the base table or to test the current search settings, you can specify a search term in this window to be searched. By right-clicking on a result you will open additional windows with information about the found base table record and the involved heuristics.

Tools>Result Checker

This tool allows you to browse through the current result table. It shows a search term of the result table and the found candidates in a list. Originally, it was intended to mark true positives, which explains the additional features in the list to declare a "no hit" or to avoid a decision. The navigation buttons also reflect this purpose, i.e. by providing buttons to jump to the next search record without a marking. You can ignore these features because the "Extended

Export" provides a much faster way to scrutinize the results. Still, the Result Checker is an invaluable tool to get a first impression about the quality of a search strategy. Furthermore, you can right-click on a candidate to invoke the heuristics for educational purposes.

Tools>Statistics

Another way to assess the quality of a search strategy is the Statistics window. Click calculate to get descriptive statistics for every search run. This tool also shows how many search records were supplied with candidates for a specific search run. You may use this information to aggregate similar search runs during Meta Export, which otherwise would be severely underrepresented. This occurs usually for incremental search runs based on destructive preparers, which can be aggregated into one run to separate them from conventional search runs aggregated into another run.
[statistics]

Tools>Notes

Since there is a searchengine.log file recording all activities within the GUI, this feature is rarely used.
[note]

Tools>Browser

With this tool you can open Foxpro files with the extension ".dbf" to browse them.

Tools>About

Shows legal information about the SearchEngine.

Script Language

The SearchEngine has a script language as backbone. Most actions and settings made in the GUI only compose script commands to be executed. Those script commands can be reviewed in the log file **searchengine.log** in the engine directory. To get easy access to the log file call the command window with the menu option **File>Command**. In the command window you can not only directly send commands to the SearchEngine but also manage scripts that define a complete workflow in the SearchEngine from the creation of the engine files to the implementation of a search strategy including the export of the results. To differentiate script files, it is recommended to use the extension ".se". You are not bound to the command window editor to compose your script files nor to execute them. The SearchEngine can be called in **Batch Mode** to execute a script directly from the command line. The following section will explain how to evoke the **Batch Mode**, the **Keywords** and the **Commands** of the SearchEngine script language.

Batch Mode

You can call the SearchEngine directly from the command line with the following syntax to initiate Batch Mode:

```
searchengine.exe script_file [output_file [append]] [para1=content] [para2=content] ...
```

The script_file can be any text file containing script commands. The SearchEngine will open an execution window that shows the progression through the script. Errors will halt the script with a message box. The execution window will be closed after completion. The output of the script can be optionally directed to an *output_file* specified as a second parameter. With the parameter "append" the output file will not be overwritten but expanded. A script can have placeholders representing variable parts of the script. In the script a placeholder name is enclosed in brackets, i.e. **[my_placeholder]**. You can define the content of placeholders by assigning it after the name with an equal sign. The content has to directly follow the equal sign. To include spaces you have to put the complete placeholder definition into quotes, for example:

```
searchengine.exe myscript.se "greetings=Welcome to the SearchEnine" threshold=75
```

You can define up to 22 placeholders this way. You can define placeholders within the script with the command **setpara**.
[setpara]

Keywords

Keywords are a part of the SearchEngine script language. They control the behavior of the SearchEngine during script execution. Normally, they only affect the command that follows them on the same line. Multiple keywords, separated by space, can be combined in a line. The silent and loud keywords have global effects for the script when not followed by a command.

silent

This keyword activates silent mode. By default, an executed command is displayed (preceded by a dot) potentially followed by its output. In silent mode, the command is suppressed and only the output will be shown. This keyword can be put in front of a command to suppress a single command or separately as global setting. This keyword can be used in conjunction with the output command to write information to files without the command call, for example statistics:

```
silent
output("statistics.txt")
statistics(.t.)
output()
```

The silenced **statistics** command does not only displays its results but also writes them to an output file specified with the **output** function. The file is closed afterwards by calling **output** without parameters.
[Keywords]
[output]

loud

This keyword displays executed commands when in silent mode. This keyword can be put in front of a command or separately as global setting to cancel silent mode.
[Keywords]
[silent]

force

The SearchEngine will stop with an error if a command would overwrite an existing file. This keyword grants the command, following it on the same line, the permission to overwrite files or show other reckless behavior for the sake of uninterrupted execution. This is especially useful for export related functions, for example:

```
force exportextended("export.txt")
```

[Keywords]
[catch]

catch

This keyword prevents the script to stop if the following command generated an error. Can be used instead of **force** to assure the existence of a file or condition without recreating it. For example, skipping the creation of an already existing SearchEngine:

```
catch create("name, name gram3, street, zip, city")
```

Be aware that any error will be caught including errors you may not have expected!
[Keywords]

[force]

exit

`exit` terminates the execution of the script.

comments

Comments are initiated with a star * or an ampersand **&** at the beginning of a line. You can initiate an inline comment after a command with two ampersands **&&**:

```
* or a & for a comment at the beginning of the line
say("Hello") && say hello to an inline comment
```

Commands

A script command performs a function of the SearchEngine. Usually, they correspond with a specific action or setting in the GUI. A command always has the shape of a function with parameters in parenthesis's. Parameters come in different types designated by the first, capitial letter of the parameter name:

- *Iactivation* = Integer - can be zero or negative depending on the function of the parameter
- *Nthreshold* = Numeric - potential limitations like intervals are addressed in the description
- *Ldarwin* = Logical (boolean) - use *.t.* for true or yes and *.f.* for false or no
- *Spath* = String - a string of characters can be enclosed in "double" or 'single' quotes

A parameter that does not begin with a capital letter can have multiple types according to its description. Parameters or sequences of parameters enclosed in brackets are optional and can be omitted for the default parameter value. Parameters are always separated by only one comma regardless of omitted parameters. For example, the function definition

```
exportresult(Stable [, Nshuffle [, Lweighted]] [, Nlow, Nhigh] [, Srunfilter] [, Lnewrun])
```

will allow the following function calls

```
exportresult("newresult.dbf", 1000)
exportresult("newresult.dbf", 1000, .f.)
exportresult("newresult.dbf", 1000, .f., .t.)
exportresult("newresult.dbf", 90, 100, "1-3")
exportresult("newresult.dbf", 90, 100, .t.)
```
etc.

The default behavior in regard of omitted parameters can be derived from the function description. Some parameters are mutual exclusive. Those are designated by a pipe | symbol in the syntax diagram, i.e.

```
help([Skeyword | Nlevel], [Lexpand])
```

The file **searchengine.log** in the SearchEngine directory records all actions and settings conducted in the GUI. This text file can be exploited as a source for your own script efforts.

activation

`activation(Iactivation)`
is used to trigger **feedback** only if the number of potential candidates equals of excceeds *Iactivation*. Feedback introduces a negative impact of surplus words of the candidates by gradually transforming the identity into a Jaccard index. A feedback of 10% means that the identity will be composed of 10% Jaccard index (taking additional words into account) and 90% original identity (ignoring all additional words). Because **feedback** calculation is time

consuming, this option allows to apply it sparingly when inflated candidate lists due to weak search terms require the introduction of variation.

Weak search terms, usually composed of few very common words, conjure large lists of mostly false positive candidates. Unfortunately, most of the time these candidates are retrieved with the same identity making it impossible to curtail them with the **cutoff** setting. After introducing variation just for this purpose, inflated lists can be truncated much closer to the **cutoff** point. If a **cutoff** is specified together with an **activation** limit, this variation is only temporary to enable truncation based of the relevance of the noise caused by additional words. The final results will not show a feedback effect. If **cutoff** or **activation** is omitted, the feedback effect remains. Usually, **activation** is set to the same value as **cutoff** namely the expected number of candidates given the quality and context of the base table, i.e. "at max, 10 candidates seem to be plausible".

You can use the command **contain** as a shortcut for setting up a strategy for the **containment** of weak search terms.
[Config>Settings]
[Containment]
[contain]
[feedback]
[cutoff]

benchmark

`benchmark()`
runs the benchmark based on current multiprocessing settings and reports the time in seconds. The benchmark simulates typical SearchEngine tasks with concurrent random file access, parallel writing to different files and CPU usage. It is intended to find the sweet spot for the **mp** setting providing a significant speed boost over a single process without risking performance cannibalization of too many processes. The speed of the SearchEngine mainly depends on the efficiency of the file system and less on raw CPU power.
[Config>Preferences]
[mp]

cancel

`cancel()`
stops the current script.

cls

`cls()`
clears the current screen. This has no effect on a potentially open log file.
[output]

configtmp

`configtmp([Spath])`
defines the default path for temporary files. If *Spath* is omitted, the default directory will be used. This setting will only take effect with the next start of the SearchEngine. By default, temporary files reside deeper in the user directory on the local drive, usually the **C:** drive. You can specify a different path on another drive if the default space is too limited or the SearchEngine has already complained of not having enough space for temporary files. [Config>Preferences]

contain

`contain([Ncutoff [, Nactivation [, Nfeedback]]])`
implements the **containment** of weak search terms. This command is a shortcut for setting up a **cutoff** for inflated candidate lists enabled by the **activation** of **feedback** to create the necessary variation in the otherwise homogenous identity distribution engendered by weak search terms. You only have to declare *Ncutoff* as *Nactivation* follows suit

and *Nfeedback* is defaulting to 10%. It is possible to specify individual values for all three parameters to deviate from the standard setting. By omitting all parameters, they will be set to zero deactivating **containment**.

Examples:
`contain(5)`
is equivalent to
`cutoff(5)`
`activation(5)`
`feedback(10)`
[Containment]
[cutoff]
[activation]
[feedback]

create

`create(`*`Ssearchtypes`*`)`
creates the SearchEngine. Within *Ssearchtypes*, the search types are separated by commas. A search type consists of the name of the search field followed by the associated preparer names, separated by blanks:

search_field [*preparer* ...][, *search_field [*preparer* ...] ...]

A search field is always a column name of the base table. The preparers will be activated in order of assignment from left to right. For example, if you want to only use the leftmost 5 words and apply 3-grams, you have to first use the preparer **MAXWORDS5** followed by **GRAM3**. The other way round, you would have restrained the search type to only five 3-grams. A search field can occur multiple times in *Ssearchtypes* with different preparer combinations, for example:

```
create("firm_name NOABBREV, firm_name NOABBREV GRAM3, street SEPNUM, zip, city")
```

This will create a SearchEngine with two search types for the base table field **firm_name**, one with condensed abbreviations (single letters combined) and one that is additionally fragmented into grams. Destructive preparers like **GRAM3** destroy and diffuse information for the sake of an increased robustness towards typos and misspellings. Search types based on them should only be used sparely because they require much more computational resources than basic (non-destructive) search types. A search type is considered destructive when an exclamation mark appears after one of its preparer in the structure string shown in the main window. Integrate them into your search strategy with dedicated incremental search runs after you have exploited the capacities of the basic search types (see **Search Strategy**).
[Action>Create]
[Search Strategy]
[Creation]
[Preparers]

cutoff

`cutoff([`*`Icutoff`*`])`
declares the expected number of candidates for a plausible match. This number depends on the quality and context of the base table. If the base table is curated in terms of the search context, i.e. a well maintained firm database with no duplicates, one would expect not more than one candidate. Of course, the search terms are in most cases not as concise or there are inherent ambiguities, for example subsidiaries with little variation to the main firm, which require a more lenient assessment. If the base table is not curated in terms of the search context and contains variation for an entity (same entity but different representations in the data), the *Icutoff* should be considerably larger or not set at all (default = 0).

If *Icutoff* is larger than zero, the SearchEngine will pick the identity of the candidate at this position in the sorted list (descending by identity) and use it as the new threshold. This prevents inconsistent truncations of candidate lists but is useless if there is no variation among the candidates. Unfortunately, large candidate lists are caused by weak search terms with an inherent tendency for low variation. A weak search term has few and very common words providing not much opportunity for variance. Variance based on the relevance of surplus words of the candidates can be introduced with **feedback**. To apply this variance only if needed for **cutoff**, set the **activation** parameter to the same value as the **cutoff**. The **activation** parameter triggers **feedback** when the candidate list equals or exceeds its value. The variation will only be temporary to enable a truncation of the ordered list as close to the *Icutoff* position as possible. In general, a low **feedback** of 10% suffices for such an application.

cutoff occurs after all other settings affecting potential candidate lists, like **darwinian** or **zealous**, have been carried out.

You can use the command **contain** as a shortcut for setting up a strategy for the **containment** of weak search terms.
[Config>Settings]
[Containment]
[activation]
[feedback]
[darwinian]
[zealous]

darwinian

`darwinian([Ldarwin])`
forces the SearchEngine to keep only the best candidates that made it over the **threshold**. By default *Ldarwin* is .f. and all candidates for a search term with an identity greater than or equal to the **threshold** are reported. When *Ldarwin* is .t., these selected candidates are further truncated by keeping only the candidates that share the highest identity. This truncation occurs **before** the **cutoff**.

This setting is suited for curated base tables where reporting anything than the best candidate(s) is redundant. It does not improve the match quality if the second best candidates of a well maintained base table - in regard of the search context - are reported. Given the search term, multiple candidates may share the same identity. Therefore, it is not guaranteed that only one candidate survives the darwinian treatment.
[Config>Settings]
[threshold]
[cutoff]

depth

`depth([Idepth])`
sets the search depth. *Idepth* can be a number between 0 and 8388608 (8192K). By default it is set to 0, which will be interpreted as 262144 (256K). The SearchEngine sorts all words of the search term by their identification potential in descending order. Following that order, it collects all candidates sharing the respective word into a buffer defined by the search depth. When the next batch of candidates would exceed the capacity of the buffer this retrieval process stops. In case the first word would exceed the capacity immediately, the SearchEngine will look for a word that fits the buffer but has a lower identification potential due to the weight assigned to its search type. The SearchEngine will skip a search term if no word will fit the buffer. To handle this case, it is possible to increase the search depth.

The deeper the buffer, the slower the search process due to redundant candidates still fitting in the buffer. The buffer size can be increased for specialized searches for classes of candidates instead of individual candidates, i.e. looking for firms with a specific legal form. You can speed up the search with a lower search depth, but this increases the risk of skipping search terms. In general, you should keep the depth at its default value unless you have a good reason to increase it like having only very short search terms of common words in a single search field.
[Config>Preferences]

erase

```
erase()
```

erases the internal SearchEngine index files. Base, search and result table are not affected. Erasing the current SearchEngine is required to be able to create a new one with different search types. Using the keyword **force** on the **create** command has the same effect.
[create]
[force]

expand

```
expand([IexpandMode])
```

expands the SearchEngine by merging a virtual registry of the search table with the original registry adjusting the frequencies. No new entries will be created. The parameter *IexpandMode* defines how the frequencies will be merged:

0 = restore original frequencies (default)
1 = replace with search table frequency
2 = use the maximum of base and search table word frequency
3 = use the minimum of base and search table word frequency
4 = increment base table frequency by search table frequency
5 = use the average of both frequencies

This command is rarely used. It allows to adjust the frequencies of the base table to the idiosyncrasies of a potentially biased search table. For example, the search table contains only wineries while the base table is a general company database. The frequencies of the word "winery" or synonyms in the registry can be adjusted to the frequencies of the search table. Because the effect of such a manipulation is difficult to assess and, in case the base table represents the population, pointless, this command should only be used when both tables are relatively small and a bias towards specific words may have a large effect.
[Action>Expand]

export

```
export(Stable [, Ssearchkey, Sfoundkey] [, Nlow, Nhigh [, Lexclusive]] [, Srunfilter], [Ltext])
```

exports the result table using a direct format. *Stable* specifies the path of the export file. The file format will be tab-delimited with column headers (except you specify the "dbf" extension). In direct format every line represents a linkage between a search record and a candidate record of the base table. The format always has a "searched" column for the identification of the search record and a "found" column to identify the found record in the base table. By default, these columns contain the corresponding record numbers. The optional parameters *Ssearchkey* and *Sfoundkey* specify unique identifiers in the search and base table, which will be used instead of the record numbers. You can also choose to keep the record numbers for one field by specifying an empty string for its corresponding key field. The format also includes the identity, the absolute identification potential called score, the run of the retrieval and an empty "equal" field for free use. It will be sorted by the "searched" key field.

With the parameters *Nlow* and *Nhigh* an interval for the identity of exported candidates can be specified. *Nlow* is included in the interval of valid identities while *Nhigh* is excluded (*Nlow >= identity < Nhigh*). With these parameters, you can split the result table into separate files for specific identity intervals. Attached to the interval is the parameter *Lexclusive*. If this parameter is .t., a search record will not be reported if there are candidates with an identity equal to or higher than *Nhigh*. Every search record will only be reported in the interval with the highest range as long as they are not overlapping.

With *Srunfilter* you can restrict the export on specified search runs. *Srunfilter* is a list of comma separated search runs to be exported. Ranges can be defined with a minus sign, e.g. "1-3, 7, 9".

The format is intended to provide a direct access to the raw results. It is not suited for manual labeling or quality control. Choose the **exportextended** function for such purposes. Still, it is possible to include the search term fields and the candidate data. When the parameter *Ltext* is .t., the fields "searchtxt" and "foundtxt" will be appended to the format containing the concatenated data of the search and candidate fields (separated with the pipe character). Even though, this representation of the results contains much more redundancy due to repeating search term than the extended export format, it may be better suited for post-processing comparisons.

Examples:

`export("D:\se\export.txt")`

exports the result table in direct format using record numbers of search and base table.

`export("D:\se\exportA.txt", "affil", "firm_id")`

uses the field "affil" to identify searched records and "firm_id" for found records.

`export("D:\se\export100.txt", 100, 101)`

exports only candidates with 100% identity.

`export("D:\se\export90.txt", 90, 100, .t.)`

exports only candidates with identities greater equal 90 and lesser than 100 and no 100% candidates for the same search record.

`export("D:\se\export80.txt", 80, 90, .t.)`

exports only candidates with identities greater equal 80 and lesser than 90 and no better candidates for the same search record.

`export("D:\se\export123.txt", "affil", "firm_id", "1-3")`

exports only candidates matched in search runs 1, 2 and 3.

`export("D:\se\exportfull.txt", .t.)`

exports the full search term and candidate data.

[File>Export>Export]

[exportextended]

exportextended

`exportextended(`*`Stable`*` [, `*`Ssearchkey`*`, `*`Sfoundkey`*` [, `*`Ssearchgroupkey`*`, `*`Sfoundgroupkey`*`]] [, `*`Nlow`*`, `*`Nhigh`*` [, `*`Lexclusive`*`]] [, `*`Srunfilter`*`])`

exports the result table using the extended format. *Stable* specifies the path of the export file. The file format will be tab-delimited with column headers (except you specify the "dbf" extension). Extended format is intended to be used for manual checking of candidates, especially to label training data. The format can be easily imported into Excel or any other table management tool. It is divided into blocks separated by blank lines. The first line of every block is reserved for the search term (searched: search table record) followed by the candidates (found: base table records). The format always has a "searched" column for the identification of the search record and a "found" column to identify the found record in the base table. By default, these columns contain the corresponding record numbers. The optional parameters *Ssearchkey* and *Sfoundkey* specify unique identifiers in the search and base table, which will be used instead of the record numbers. You can also choose to keep the record numbers for one field by specifying an empty string for its corresponding key field. With *Ssearchgroupkey* and *Sfoundgroupkey* superordinate fields can be specified additionally to the unique identifiers regardless of them being real fields or record numbers. A superordinate field identifies an entity comprising multiple records representing different variants. A typical example for such an entity would be a firm that went through multiple name and address changes over time. The superordinate key field is the same for all variants because we observe the same entity over time. The SearchEngine will optimize its export output by reducing redundancy on the entity level. If a group key is an empty string or both group keys are omitted every record is considered an entity within the respective table.

With the parameters *Nlow* and *Nhigh* an interval for the identity of exported candidates can be specified. *Nlow* is included in the interval of valid identities while *Nhigh* is excluded (*Nlow* >= *identity* < *Nhigh*). With these parameters, you can split the result table into separate files for specific identity intervals. Attached to the interval is the parameter *Lexclusive*. If this parameter is .t., a search record will not be reported if there are candidates with an identity equal to or higher than *Nhigh*. Every search record will only be reported in the interval with the highest range as long as they are not overlapping.

With *Srunfilter* you can restrict the export on specified search runs. *Srunfilter* is a list of comma separated search runs to be exported. Ranges can be defined with a minus sign, e.g. "1-3, 7, 9".

Examples:

```
exportextended("D:\se\export.txt")
```

exports the full result table in extended format using record numbers of search and base table.

```
exportextended("D:\se\exportA.txt", "affil", "firm_id")
```

uses the field "affil" to identify searched records and "firm_id" for found records.

```
exportextended("D:\se\exportB.txt", "affil", "firm_id", "", "firm_key")
```

exports found records grouped by the entity identification "firm_key".

```
exportextended("D:\se\export100.txt", 100, 101)
```

exports only candidates with 100% identity.

```
exportextended("D:\se\export90.txt", 90, 100, .t.)
```

exports only candidates with identities greater equal 90 and lesser than 100 and no 100% candidates for the same search record.

```
exportextended("D:\se\export80.txt", 80, 90, .t.)
```

exports only candidates with identities greater equal 80 and lesser than 90 and no better candidates for the same search record.

```
exportextended("D:\se\export123.txt", "affil", "firm_id", "1-3")
```

exports only candidates matched in search runs 1, 2 and 3.

When you export a training sample for the SearchEngine Machine Learning (**SEML**) extracted with **exportresult**, you should omit *Ssearchkey* and *Sfoundkey* or specify empty strings. The references have to be based on record numbers because the **meta** export exclusively uses them. There is an option to specify group keys in brackets, i.e. "[firm_key]" to enable implicit entity optimization referring to record numbers. This option is DEPRECATED because it creates a sample bias. It was introduced for completeness sake without further thought.

The format has a field called "equal", which is dedicated for labeling the matches. The convention is to mark valid (true positive) matches with a 1 while invalid (false positive) matches are designated with a 9. Because very block consists of the search term in the first line followed by the found candidates, the first line is not only required as a base for you assessment but can be used to enter a default value for the whole block of candidates. If you enter a 1 into the equal field of the first line of a block, all candidates are considered to be valid. You can mark exemptions to this default value among the candidates with a 9 in the equal field. Of course, this also works in the opposite way with a 9 as default and a 1 as exemption. This convention saves a lot of typing when used in a sensible fashion and a script that fills the gaps according to the default values can easily be implemented in any language (Python, Visual Basic, Stata, ...). The extended format is suited to completely check smaller matches especially if the original keys of the search and base table were assigned. But it is also the best method to label training samples for the **SEML** (SearchEngine Machine Learning) approach as the included machine learning scripts in Stata understand the labeling convention. Importing the export format and parsing the labeling convention with other programming languages and machine learning tools should pose no problem.
[File>Export>Extended Export]
[SEML]
[meta]

exportgrouped

exportgrouped(*Stable* [, *Scascade*] [, *Sbasekey*] [, *Nlow*, *Nhigh* [, *Lexclusive*]] [, *Srunfilter*] [, *Lnotext* [, *Lnosingles*]])

exports the result table using the grouped format. This function is only available when the result table is based on a self-referential search. The flag "mono" has to be set in the structure string (see **show**) of the SearchEngine denoting that the base table and the search table are the same. The result table constitutes a network structure where candidates of a search term appear also as search term with candidates and so on. This is only possible because the searched and the found records stem from the same realm. A self-referential search has the purpose of find clusters of similar entries to identify specific entities in the data. In general, those searches require a high threshold for the identity as retrieval (avoiding false negatives) is not the focus but the creation of a similarity network within the data. The grouped format represents clusters within those similarity networks. The cluster demarcations can be defined by a rule set called "cascade" outlined by the parameter *Scascade*. By default, a cluster is demarcated from another cluster when no connection between the respectively contained nodes exists. These clusters can get quite large because weak search terms act like super-connectors. Another reason for huge clusters is the intransitivity of the network caused by the asymmetry of the search algorithm. The candidate as a search term may retrieve different candidates than its original search term. These intransitive connections may propagate over long distances with completely unrelated entities at their ends.

A cascade is a rule set that restricts the validity of a connection conditional on the intermediate size of the currently traversed cluster. There are four variables that can be addressed in the rules:

min - minimum identity between the two nodes defining the connection
max - maximum identity between the two nodes defining the connection
s - the minimum absolute identification potential of the connected nodes (score)
p - percentile of s (always between 0 (lowest score) and 100 (highest score)

The directed network, is transformed into an undirected network. If the reverse direction does not exist for a connection, the respective identity is zero. This means that the **min** value is zero when the candidate as a search term does not retrieve the original search term. This is the case, when the original search term is weak (consisting of few, common words) while the candidate has more on offer. The network can be completed by enforcing bi-directional connections with the **mirror** and the **research** function, which will provide a **min** value even below the threshold. This is only necessary for very complex rule sets, which are rarely beneficial given the blunt assumptions that have to be made about the data. The **max** value of a connection is always larger than zero because otherwise there would not be a connection. The score related variables **s** and **p** are based on the nodes and not the connection. In most cases, the rules will refer to the **min** variable and rarely to the **max** variable because the latter is redundant as the connection is already constituted by a high threshold. A rule is a logical expression based on the connection parameters attached to an activation limit designated by the **@ ** symbol. For example

```
min >= 90 @ 5
```

will define a restricting rule based on the **min** variable when the currently traversed cluster would exceed 4 nodes. All cluster sizes up to 4 nodes are considered safe because there are no super-connectors involved with such a small cluster limit. Of course, the activation threshold is an arbitrary choice and may require experience with the data and multiple attempts with different rule sets. Multiple logical expressions can be combined with the logical operators **and** and **or**. The priority order of the expression resolution can be adjusted with parenthesis's. The name **cascade** stems from the fact that multiple rule sets can be put in a comma separated sequence. Every time the activation threshold of the following rule is breached, it will become active and the traversal is reset to the respective starting node. For example

```
min >= 90 or min >= 70 and p >= 75 @ 0, min >= 90 @ 11
```

will have an immediately active rule from the beginning of the traversal engaged with a threshold of 0. This rule already enforces a high similarity of 90% in both directions, which will be attenuated when the absolute identification potential of both candidates is in the 75% percentile. The next rule will be activated when the cluster still exceeds the 10 nodes limit. This rule drops the attenuation clause for a stricter re-traversal. It can be quite daunting to assess an activation limit for the perfect cluster, especially if there is a high variation in the data in regard of the representation of an entity. Some entities have short names with less potential for variation while others may have long winded names with lots of opportunities for rearrangement of words and misspellings. To enforce even ground pertaining this issue, the **cascade** can be **nested**. The first cascade defines clusters with a high similarity encompassing all trivial variations of an entity that stem from rearrangements or misspellings. A second cascade allows for so called intransitive transitions requiring a gradual overlap between related entities, i.e. merger, daughter companies, joint-ventures or the shift in topics within abstract. Of course, the second cascade is optional. It will be initiated with a semicolon "**;**" and uses the same syntax than the basic cascade with the one difference that the activation limit refers to the clusters of the first cascade instead of candidate nodes. For example

```
min >= 90 or min >= 70 and p >= 75 @ 0, min >= 90 @ 11; min >= 90 @ 4
```

creates a intermediate network of clusters formed by the first cascade. The connections between those clusters are left-overs and weaker than the rules that constituted the respective clusters. In this example, clusters are combined as long as not more than three of them form a new hyper-cluster. If a fourth is connected, the rule will be applied and re-traversal initiated preventing any further clustering from that starting node. Nested cascades allow for a more daring clustering beyond the confines of simple entity resolution. Please read the **discussion paper** for more details.

Stable specifies the path the grouped export file. The file format will be tab-delimited with column headers (except you specify the "dbf" extension). If the *Scascade* definition is skipped or empty all available connections will be traversed and clustered. *Sbasekey* specifies the unique identifier field of the base table that should be used instead of record numbers. With *Nlow* and *Nhigh* an interval for the identity can be specified to filter the eligible records of the result table. *Nhigh* is excluded from the interval. *Lexclusive* skips the export for result records for search terms with better identities than defined by the interval assuming that they are already exported into another file.

With *Srunfilter* you can restrict the export on specified search runs. *Srunfilter* is a list of comma separated search runs to be exported. Ranges can be defined with a minus sign, e.g. "1-3, 7, 9".

By default search fields will be exported for assessment of the clustering. With *Lnotext* equal .t., the reporting of the search fields will be suppressed.

With *Lnosingles* equal .t., the reporting of single member clusters can be suppressed.

Examples:
```
exportgrouped("D:\patents\abstracts.txt", "min >= 80 @ 11, min >= 90 @ 101")
```
clusters patent abstracts with activation limit 11 and 101 as fallback.
```
exportgrouped("D:\patents\applicants.txt", "min >= 90 @ 0; min >= 80 @ 6, min >= 90 @ 11", "app_id", "4-255",
.f., .t.)
```
clusters patent applicants with a nested cascade using a key field and including any run after the third skipping singles but showing the search fields.
[File>Export>Grouped Export]
[Discussion Paper]
[mirror]
[research]

exportresult

exportresult(*Stable* [, *Nshuffle*, [, *Lweighted*]] [, *Nlow*, *Nhigh*] [, *Srunfilter*] [, *Lnewrun*])
extracts a filtered copy/sample of the result table. The exported *Stable* is always a Foxpro table (extension "dbf"). It is advised to not specify an extension or use ".dbf". With *Nshuffle* specified, a sample of the result table will be drawn. The sample size is a share, when *Nshuffle* is lesser than 1, or it is an absolute number, when *Nshuffle* is greater equal 1. The size relates to the number of search records and not to the number of candidates. When *Nshuffle* is 1000, it means that 1000 search records with all associated candidates will be exported. By default, the sample draw does not take into account the number of candidates for a search term. If *Lweighted* is .t., the search terms are weighted by the number of their candidates. Search terms with more candidates have a higher probability to be drawn. This can be beneficial to compose a training data after **compound searches**, which usually have very skewed canidate distributions due to necessarily weak **containment** (or none at all).

With *Nlow* and *Nhigh* an interval for the identity can be specified to filter the eligible records. *Nhigh* is excluded from the interval. With *Srunfilter* you can restrict the export on specified search runs. *Srunfilter* is a list of comma separated search runs to be filtered for export. Ranges can be defined with a minus sign, e.g. "1-3, 7, 9". You can change the run number for all entries in such a run list by appending the aggregate number after a colon, e.g. "1,5:1". To specify multiple aggregate run numbers, separate those lists with a semicolon. The lists are parsed from left to right and can overwrite each other. An aggregate run number of 0 will exclude the runs from preceding lists. For example: "1-9:2;

4,5:1; 3:0" is equivalent to "4,5:1; 1,2,6-9:2". Be aware that any run not explicitly included in *Srunfilter* will be excluded. If *Lnewrun* is .t., the search runs will be resequenced to bridge gaps.

This function can be used to remove unwanted runs from the result table or to draw a sample for a training dataset for the SearchEngine Machine Learning approach **SEML**. To replace a result table you have to declare the new one with the **result** function. In the context of SEML, the training sample will be exported with **exportextended**. After that, do not forget to reassign the original result table. Otherwise, you will be surprised if you try to draw a second sample from an already drawn sample. In case of removing unwanted runs, you can delete the old result table after assigning the new one. Alternatively, the **strip** function can also be used to remove and renumber runs directly without exporting the result table and the related hassle. As it directly affects the current result table, these actions cannot be reverted.

Examples:
```
exportresult("D:\patents\result_sample", 1000)
```
draws a sample of 1000 search terms with the corresponding candidates.
```
exportresult("D:\pulications\result_sample", 200, .t.)
```
draws a weighted sample maintaining the candidate distribution.
```
exportresult("D:\patents\result_1", "1-3, 5", .t.)
result("D:\patents\result_1")
```
creates a new result table by removing run number 4 from the current result table, closes the gap (run 5 becomes 4) and declares the modified table as new result table. Maybe something was wrong with run 4.
[File>Export>Result Export]
[SEML]
[result]
[exportextended]
[Compound Search]
[containment]
[strip]

exportmeta

```
exportmeta(Stable [, Smeta] [, Lnocomp] [, Nlow, Nhigh] [, Srunfilter])
```
exports meta data of the result table. For every search term and candidate tuple, the meta data codifies the absolute identification potentials for the following sets: the matching words, the words of the search term that did not match and the surplus words of the candidate for every search type. The respective potentials are reported in descending order. This is the main component of the meta data but it further contains a plethora of other data that may be related with the probability of false positives (see **discussion paper**). The meta data is the integral part of the SearchEngine Machine Learning approach **SEML** providing the variance needed to imitate the decision making process behind the labeling of the training data. The training sample is drawn with the **exportresult** and exported into an easy-to-label format using the **exportextended** function. After labeling, which means separating the true from the false positives, the training data can be enriched with meta data. The SearchEngine package provides a Stata tool to train a neural network with the training and the meta data. Both data sets can easily be handled with other programming tools to apply your own machine learning approach not based on Stata. After training, the model of your choice, be it a neural network, a random forest or something else, can be applied to the whole meta data. The meta data refers to the record numbers of the search table (searched) and the record numbers of the candidates in the base table (found). You should keep that in mind when exporting and handling the training data.

Stable defines the path of the meta export file. As long as the extension is not explicitly set to ".dbf", the file will be a tab-delimited text file with column headers. Every meta file will have the fields "searched" and "found" for search table and candidate records numbers as reference. It is advised to either have record numbers in your source files for the base and search table or use database formats supporting record numbers. Not all meta fields are normalized between 0 and 1. Make sure that your ML approach can handle raw numbers or normalize them accordingly.

The *smeta* parameter defines how many absolute identification potentials (AIP) should be exported per set. By default, 5 AIPs are exported per set. Because some search types do not have so much variation, for example postal codes or city names, or require more entries for a meaningful assessment, like search types based on n-grams, you can specify how many AIPs are exported per search type set. The syntax for *smeta* refers to search types by number of definition as shown in the structure string. You can define a list of search types by separating them with commas and/or defining ranges with a minus sign that should be exported with a specific number of AIPs. This number follows the list or single search type number separated with an equal sign. Multiple assignments are separated with semicolons, i.e.: 1, 3-6 = 5; 2 = 15

By default, the meta data includes string similarity metrics bases on the LRCPD algorithm (see **discussion paper**), which is independent of word positions. This is fine for limited search fields like firm or person names or addresses but will take exponentially more time to be calculated for longer texts like abstracts while losing its relevance. Setting the parameter *Lnocomp* to .t. will suppress this part of the meta data.

With *Nlow* and *Nhigh* an interval for the identity can be specified to filter the eligible records. *Nhigh* is excluded from the interval. *Srunfilter* is a list of comma separated search runs to be filtered for export. Ranges can be defined with a minus sign, e.g. "1-3, 7, 9". You can change the run number for all entries in such a run list by appending the aggregate number after a colon, e.g. "1,5:1". To specify multiple aggregate run numbers separate those list with a semicolon. The list are parsed from left to right and can overwrite each other. An aggregate run number of 0 will exclude the runs of the preceding list. For example: "1-9:2; 4,5:1; 3:0" is equivalent to "4,5:1; 1,2,6-9:2". Be aware that any run not explicitly included in *Srunfilter* will be excluded.

One criteria of the meta data is the run number as one-hot dummy vector. You can use *Srunfilter* to aggregate runs using roughly same method of retrieval, i.e. destructive linguistic search types vs. conventional search types, to prevent under-representation of run dummies in the training data due to the typically low number of misspellings. In general, all runs that share the same composition of the identity and are only differentiated by the **threshold**, **cutoff** or other restricting settings without direct impact on the identity can be aggregated without any concerns. If aggregation is necessary at all, can be assessed with the **statistics** function, which shows the distribution of the candidates over the runs. Of course, aggregation of run dummies can also be conducted on exported meta data at a later stage or even completely ignored (as the whole run dummy aggregation is only a pet peeve of the author without a significant impact on the efficiency of the machine learning).

Examples:
```
exportmeta("d:\patents\meta.txt", "1=5; 2=15; 3=3; 4=1; 5=2")
```
exports meta data adjusting for the particularities of the search types: firm, firm GRAM3, street, zip, city.
```
exportmeta("d:\patents\meta.txt", "1=5; 2=15; 3=3; 4=1; 5=2", "1,4:1; 2,3,5,6:2")
```
exports meta data adjusting for the particularities of the search types: firm, firm GRAM3, street, zip, city.
The runs 1 and 4 are using the same priority (weight) setting while the runs 2,3,5 and 6 use a different weight scheme and are condensed under another aggregate run. Alternatively, the following expression could have been used: "1-6:2; 1,4:1"
```
exportmeta("d:\abstracts\meta.txt", "1=10", .t.)
```
exports meta data of a search within patent abstracts skipping the LRCPD similarity metrics.

Besides the meta data defined by *Stable* the function also creates a registry table to account for the word frequencies of the search table. It will use the same search types as the main registry. The search table registry can be found in the same directory as the search table with the postfix "_registry.dbf" attached to the search table name. Keep this file because it will speed up meta exports for the same search and base table setups. Delete/rename this file, if something has changed in this setup. Unfortunately, the SearchEngine will not recognise a change in the base table or the search type configuration to initiate the creation of a new search table registry. You have to force it by deleting the file in such a case.
[File>Export>Meta Export]
[Discussion Paper]
[SEML]
[exportresult]

[exportextended]
[statistics]

feedback

`feedback([`*`Nfeedback`*`])`
sets the *Nfeedback* parameter, which can be a percentage number between 0 (default) and 100. Feedback determines the effect of surplus words of the candidate on the identity. These are words not represented by the search term. By default, additional words of the candidate are ignored. The higher *Nfeedback* is, the higher the negative impact of additional words. At 100% feedback, the identity will be completely transformed into a Jaccard index. You can use feedback to enforce a higher similarity on the candidates beyond the bare retrieval. This can be relevant for a search context with a highly formalized structure like person names. Because feedback influences the identity, it can be applied after retrieval with the **research** command without risking to lose candidates. Since the introduction of the SearchEngine Machine Learning approach SEML, identity readjustment became less important. Still, it is an invaluable component for the **containment** of weak search terms.

The SearchEngine is focused on retrieval of candidates and therefore ignores any additional noise within them. Of course, this may lead to inflated candidate lists for weak search terms, i.e. "London Limited, London". Feedback is one component to fight this inflation by introducing variation among the candidates where by default no variation exists due to the focus on matching the search term words. If **activation** is specified feedback will only applied if the count of candidates equals or exceeds this number. This affects the identity of candidates with more relevant noise stronger than candidates with weak or no noise (additional words). By specifying an additional **cutoff** for the number of candidates, the feedback will only be applied temporarily to create variation for the cutoff.

If **feedback** is used in conjunction with **cutoff** and **activation** to reduce inflated candidate lists due to weak search terms, a low value of 10(%) suffices. **cutoff** and **activation** should be set to the number of expected candidates depending on the quality of the base table.

You can use the command **contain** as a shortcut for setting up a strategy for the **containment** of weak search terms.
[Config>Settings]
[Containment]
[contain]
[activation]
[cutoff]
[research]
[SEML]

ignorant

`ignorant([`*`Lignorant`*`])`
controls how the SearchEngine reacts to words of the search term not represented in the registry respectively base table. By default, the average frequency of a word of the respective search type will be used to calculate the identification potential of an unknown word. The average frequency is derived from the registry. When *Lignorant* is .t., unknown words are ignored. This parameter affects all search types.

Ignoring words is risky because it increases the identification potential of the remaining words which may be nondescript. An ignorant search can lead to inflated candidate lists and is in most cases only suitable for small search projects with subsequent manual control. This is an experimental option.
[Config>Settings]

getpara

`getpara([`*`Spara`*`])`
displays the content of placeholder *Spara* or of all placeholders, if *Spara* is omitted or empty. Placeholders will be substituted before a command is executed. They are enclosed in brackets.

As the name suggests, placeholders will be substituted before the execution of a command. They are enclosed in brackets. For example:
`setpara("mypara", "Hello World")` sets the placeholder *mypara*
`getpara("mypara")` displays "mypara=Hello World"
`say("[mypara]!!!")` displays "Hello World!!!"

Placeholders can also be defined with command line parameters (therefore the "para") of the SearchEngine batch mode call, for example:
`searchengine.exe myscript.se "mypara=Hello World"` (quotes are necessary in this case because spaces are used to separate command line parameters).
[setpara]
[Batch Mode]

help

`help([`*`Skeyword`*` [, `*`Lexpand`*`]] | [`*`Nlevel`*`])`
displays sections of this help file (searchengine.md). *Skeyword* refers to a header within this documentation. If the header exists, the corresponding section will be displayed. You can expand the view to all subordinate sections with *Lexpand* set to .t. (default is .f.). For example: `help("Commands", .t.)` lists not only the paragraphs directly under this header but all commands listed in the chapter "Commands".

Nlevel can be used to get the table of contents up to the specified depth. With 1 showing only the main title, 2 includes the main chapters and so on. If no parameter is specified the full table of contents will be displayed.

The help texts use links to refer to other relevant sections. Links are enclosed in brackets. Clicking on a link in the command window envokes the **help** command to show the referenced section. At the end every section are links to the directly subordinated sections.

An empty string "" or the keyword "all" displays the complete help file. The same can be achieved with
`help("searchengine", .t.)`
as the main title is "SearchEngine".
[File>Command]

importbase

`importbase(`*`Sfile`*` [, `*`Lnomemos`*`])`
imports respectively declares the base table. If the file extension is ".txt", the SearchEngine first checks the existence of an already imported Foxpro table with the same name and the extension ".dbf". An existing table will be declared as the base table otherwise *Sfile* will be imported into a Foxpro table. A text file needs to be tab-delimited with a header line with column names. Usually, the data is exported from a different system into the intermediate text format. The success of the import depends on the flawless structure of the text file. You should blank all critical control characters within the source fields: tabulator (ascii: 9), line feed (ascii: 10) and carriage return (ascii: 13). Export only necessary search fields and one unique identifier to avoid superfluous data transfer. After a successful import the base table will appear as Foxpro table (extension ".dbf").

Lnomemos prevents the usage of memo fields during import. Memo fields are required if text fields exceed the length of 254 characters. In the case of suppressed memo fields a data loss of 0.1% is considered tolerable to handle outliers as truncation is already authorized. Use memo fields if you expect longer texts in the data. Set *Lnomemos* to .t. if text fields longer than 254 characters can be considered unwanted outliers. Memo fields appear as type "M" and characters fields as type "C" in the table structure (see below).

You can import the text file again by deleting the corresponding Foxpro table (everything with the same name and the extensions .dbf, .cdx, .fpt, .idx). This can be necessary if the file was affected by control characters in field columns. You can verify a proper import by checking the table structure of the base table in the structure string (look for [BaseTable]) shown in the main window of the GUI or displayed by the command **show**. The number of records, the

field names and types should be as expected. Otherwise, check the source data for critical control characters.
[Config>File Locations]
[show]

importsearch

```
importsearch(Sfile [, Lnomemos])
```
imports respectively declares the search table. If the file extension is ".txt", the SearchEngine first checks the existence of an already imported Foxpro table with the same name and the extension ".dbf". An existing table will be declared as the base table otherwise *Sfile* will be imported into a Foxpro table. A text file needs to be tab-delimited with a header line with column names. Usually, the data is exported from a different system into the intermediate text format. The success of the import depends on the flawless structure of the text file. You should blank all critical control characters within the source fields: tabulator (ascii: 9), line feed (ascii: 10) and carriage return (ascii: 13). Export only necessary search fields and one unique identifier to avoid superfluous data transfer. After a successful import the search table will appear as Foxpro table (extension ".dbf").

Lnomemos prevents the usage of memo fields during import. Memo fields are required if text fields exceed the length of 254 characters. In the case of suppressed memo fields a data loss of 0.1% is considered tolerable to handle outliers as truncation is already authorized. Use memo fields if you expect longer texts in the data. Set *Lnomemos* to .t. if text fields longer than 254 characters can be considered unwanted outliers. Memo fields appear as type "M" and characters fields as type "C" in the table structure (see below).

You can import the text file again by deleting the corresponding Foxpro table (everything with the same name and the extensions .dbf, .cdx, .fpt, .idx). This can be necessary if the file was affected by control characters in field columns. You can verify a proper import by checking the table structure of the search table in the structure string (look for [SearchTable]) shown in the main window of the GUI or displayed by the command **show**. The number of records, the field names and types should be as expected. Otherwise, check the source data for critical control characters.
[Config>File Locations]
[show]

join

```
join(Sfield [, Ssearchfield])
```
links a field of the search table with the name *Sfield* to the base table field *Ssearchfield*. If both have the same name, *Ssearchfield* can be omitted. Not all available search fields (base table fields assigned to search types, see **create**) need to linked to a search table field, for example, if your search table has no street address field but there is an address search field in the base table. In such cases, the priorities (weights) for those unlinked fields should be zero to declare them as inactive. Make sure that both linked fields share the same context. You should avoid to link composed search table fields to separated search fields of the base table, i.e. a composed address field with street, city, postcode with the isolated street field. If you cannot separate a composed field, it is better to create a new SearchEngine on an adjusted base table with the same structure as the search table by concatenating the respective fields into composed fields. In general, it is simpler to compose fields than to separate them.
[Config>Join Search Fields]
[unjoin] [create]

limit

```
limit(Nlimit)
```
set the threshold for the identity of the candidates. It can be any number between 0 and 100. This function is identical to **threshold**.
[Config>Settings]
[threshold]

list

```
list([Lalpha])
```

lists all SearchEngine save slots. If *Lalpha* is .t., sort order will be by slot name instead of change date.
[File>Load Settings]
[File>Save Settings]

load

```
load([Sslot])
```

loads the specified SearchEngine slot. If *Sslot* is omitted or empty, the current slot will be reloaded. The SearchEngine saves are just copies of the structure string. They describe a search project with all current settings. A SearchEngine is always associated with only one base table with the **create** function via extensive index files in the engine directory. You can replace those files by recreating the SearchEngine with a different base table or search type setting but reloading an older slot based on overwritten index files will not miraculously restore a bunch of index tables of considerable size. Every SearchEngine can only handle one base table. Create separate engine directories for different base tables or search types setups.
[File>Load Settings]

loadpreparer

```
loadpreparer()
```

reloads all **custom preparers** from xml-files in the engine directory beginning with "searchengine", like "searchengine.xml" or "SearchEngine_firma_strasse.xml". Errors in the xml-structure, preparer definitions or missing preparers of current search types will be reported. Use this function to develop custom preparers by testing them with the **prepare** command and browsing through the code with **showpreparer**.
[Preparers]
[Custom Preparers]
[prepare]
[showpreparer]

message

```
message([Stext])
```

opens a message box showing *Stext*. Program halts until confirmation.

mirror

```
mirror([Srunfilter])
```

mirrors the matches without a reverse entry to enforce symmetry for a self-referential search where base and search table are the same. The created entries have an incremented run number and an identity of zero. The run number will always be reserved even if there are no mirrored entries. You have to use the **research** or **refine** functions with a run filter targeted on the new run number to evaluate the identity for the reverse entries. Keep in mind that these are of lower similarity because they have not made it over the threshold. It is advised to apply the **research** function with *log* smoothed search types or the **refine** function to completely waive the frequency driven heuristic. The latter may even lead to identities above the threshold. The **mirror** function is exclusively intended to support the **exportgrouped** function by providing tangible information for the reverse case of lopsided matches. It allows to separate justifiable matches (close to the threshold) from random matches caused by weak search terms (very low identity).

With *Srunfilter* you can restrict the export on specified search runs. *Srunfilter* is a list of comma separated search runs to be filtered for export. Ranges can be defined with a minus sign, e.g. "1-3, 7, 9". This parameter usually does not apply.
[Action>Mirror]
[exportgrouped]

[research]
[refine]

mp

`mp([Icpu])`
dedicates *Icpu* number of CPUs to SearchEngine actions. Negative numbers determine the CPUs not used by the SearchEngine. Omitting *Icpu* returns the current setting and a zero reserves up to 6 CPUs (initial setting). You can deactivate multiprocessing by setting *Icpu* to 1. Because the SearchEngine also causes heavy file access traffic increasing the number of assigned CPUs has diminishing returns and may even have a detrimental effect. Use the **benchmark** function to find the sweet spot for appropriate CPU usage.
[Config>Preferences]
[benchmark]

note

`note([Snote, [Lreplace]])`
appends a new line of text to the notes in the info section of the SearchEngine structure string. You can append multiple lines if *Snote* contains "<br>" tags, which will be translated to line breaks. If *Lreplace* is .t., the previous notes will be overwritten. If called without parameters, the notes will be displayed. You can use this function to write down your search strategy, further proceedings or your shopping list. It became less important since the introduction of the SearchEngine log file and script files.
[Tools>Notes]

output

`output([Slogfile [, Lappend]])`
defines a new log file for the command output. By default, a new file will be created respectively an existing file overwritten. If *Lappend* is .t., the output will be appended to an existing file, which will be created if necessary. If *Slogfile* is omitted, the current output file will be closed. Only one output file can be open at a time. Opening a new file will close the current one. By default, all output, including the commands, will be logged in the current output file as it is displayed in the command window. The keywords **loud** and **silent** control whether the commands are logged or only the output of the commands. If a script is called in **Batch Mode**, the output file can be specified as parameter.

Examples:
`output("d:\se\logfile.txt", .t.)` appends to the existing file logfile.txt
`output()` closes the current log file

Do not confuse the output file with the "SearchEngine.log" file, which is reserved exclusively for logging activities in the GUI.
[Batch Mode
[silent]
[loud]

prepare

`prepare([Spreparerlist [, *Stext]])`
applies the preparers listed in *Spreparerlist* (comma or blank separated) on the string specified with *Stext*. Use this function to test combinations of built-in preparers and custom preparers. You can get a list of all installed preparer by omitting any parameter.

Examples:
`prepare("noabbrev gram3", "B.A.S.F. Aktiengesellschaft")` applies the NOABBREV and GRAM3 preparer in this order on the specified text.
`prepare()` shows a list of all installed preparers.

[Preparers]
[showpreparer]
[loadpreparer]

showpreparer

`showpreparer([Spreparer], [Lcompact])` shows the XML code of the preparer *Spreparer*. If a preparer name is omitted, the code of all installed preparers will be shown. By default, a more spacious format is used. If *Lcompact* is .t., every command of a preparer is displayed in one line.

Examples:
`preparer("Firma", .t.)` shows the compact XML code of custom preparer "Firma".
`preparer()` shows the XML code of all installed preparers.
[Preparers]
[prepare]
[loadpreparer]

refine

`refine([Iidentitymode, Icomparemode], [Srunfilter] , [Ldestructiveonly])`
refines the existing matches in the result table. Refining is a misleading but catchy name for applying a string distance metric on retrieved candidates. Together with **research**, it belongs to the identity manipulation functions applied after retrieval of candidates. While the retrieval of the SearchEngine is exclusively based on the word frequency based heuristic, the LRCPD metric of the string comparison ignores the identification potential of word frequencies and replaces them with the similarity of strings (see **discussion paper** for an extensive explanation of the LRCPD metric). This is helpful in conjunction with destructive, linguistic preparers like n-grams. Those search types retrieve too many false positives because that is their nature. They provide tolerance towards misspellings at the expense of concise matches leading to an increase of false positives. Refining those results is an effective method to reduce the number of false positives with a method that is also typo-tolerant but much more precise in measuring straight similarities. Those methods always require two parameters for the comparison and are therefore not suited for effective index based retrieval.

The **refine** function is part of the **search** command and will automatically be called when destructive linguistic preparers are involved and post-search refinement is specified. The general process for such a search run is a follows:

1. The SearchEngine retrieves potential candidates into a temporary result set for the current run.
2. Threshold, cutoff and feedback are applied as for a normal search.
3. Identities are set to zero for the results of the current run.
4. The SearchEngine calls the **refine** method on the current run. It compares the original fields of the search and base table where the associated search types contain destructive preparers using the LRCPD method. This method returns a string similarity percentage which will be weighted by the priorities for the corresponding search types. This constitutes the "destructive search types" part of the identity.
5. The remaining identity for the involved search types with non-destructive preparers is calulated by a final step using the **research** function, which is based on the frequency heuristic. This constitutes the "non-destructive search types" part of the identity.
6. The composed identity replaces the candidate identity.

Because this process is already integrated into the **search** call, the **refine** function is only used when a string similarity metric should explicitly replace the frequency heuristic. The **refine** function will always compare all search and base table fields that are linked with the **join** function unless you set the *Ldestructiveonly* parameter to .t., which skips all search fields without a destructive search type (implementing a destructive preparer). To exclude search fields from the comparison you can **unjoin** them.

The parameter *Iidentitymode* defines how the refined identity relates to the existing identity for the respective candidate:

1 = replace identity (default)
2 = maximize identity to overrule the existing identity if better
3 = minimize identity to enforce a worst case scenario
4 = add the refined identity to the existing identity
5 = calculate the average between the refined and existing identity

The value 4 (adding) is required if the identity is already only partial due to search field selection via **join** and **unjoin** or switching search types on and off via the priority (weight setting to zero/non-zero). *Identitymode* 2, 3 and 5 are of a more experimental nature. By default, the identity will be replaced (*Identitymode* = 1). This setting provides a high amount of flexibility to design your own composed identity to decouple the initial identity used for retrieval from bespoke similarity measures.

The LRCPD comparison is not commutative (symetric). The LRCPD method like the frequency based heuristic is a directed comparison ignoring surplus words in the target term. The direction of the comparison can be adjusted with *Icomparemode*:

1 = compare searched terms with found target terms (default)
2 = dynamic - compare in both directions choosing the minimum
3 = compare found terms with searched target terms

Option 2 (dynamic) can be used when there is not much leeway in regard of the search context, i.e. person names. Option 3 (found in searched) can be used when more noise is expected among the candidates. This may indicate, that the whole search direction should be switched.

Srunfilter is a list of comma separated search runs to be filtered for refining. Ranges can be defined with a minus sign, e.g "1-3, 7, 9". This parameter allows to target the refinement on specific runs.

Because the LRCPD algorithm compares all combinations of word-tuples, the number of comparisons increases by the power of two with the number of words. Keep this in mind when applying refinement on larger text fields.

In general, you will rarely use the **research** or **refine** commands as their main purpose is the re-evaluation of linguistic search types, which is already integrated into the **search** function.
[Action>Refine]
[research]
[search]
[join]
[unjoin]

relative

```
relative([Lrelative])
```
redistributes the priorities and thereby the weights of empty search fields on the weights of the remaining filled search fields according to their respective priority. The total of the weights will always be 100% regardless of empty fields. By default, *Lrelative* is .f. and the weight of empty search fields will be missing from the maximum identity.

Relative priorities can be applied for sparsely filled search tables especially when the fields are interchangeable or optional, for example synonyms or additional variants. They are less suited for a normal search where search field are not substitutive, for example a firm name with auxiliary address fields. The identity will not be indicative of the quality of the match as a missing search fields does not reduce the achievable identity while an existing but not matching field has this effect.
[Config>Settings]

remove

```
remove(Sslot)
```

removes a save slot with the name specified in *Sslot*. This has no effect on the SearchEngine index files. To delete those, you have to use the **erase** function.
[File>Save]
[load]
[save]
[erase]

research

```
research([Iidentitymode [, Iscoremode]] [, Srunfilter] [, Lnondestructiveonly])
```

reapplies the current settings on existing matches in the result table. All parameters affecting the composition of the identity and the score (absolute identification potential) are re-evaluated. Together with **refine** this function belongs to the identity manipulation functions applied after retrieval of candidates. It uses the same mechanisms and heuristics as the retrieval by the **search** function. It is possible to apply *feedback* or redistribute priorities (weights) retroactively, allowing the separation of retrieval from the final evaluation of candidates. The new identities will not be affected by the *threshold*.

The **research** function is part of the **search** command and will automatically be called when destructive linguistic preparers are involved and post-search refinement is specified. The general process for such a search run is a follows:

1. The SearchEngine retrieves potential candidates into a temporary result set for the current run.
2. Threshold, cutoff and feedback are applied as for a normal search.
3. Identities are set to zero for the results of the current run.
4. The SearchEngine calls the **refine** method on the current run. It compares the original fields of the search and base table where the associated search types contain destructive preparers using the LRCPD method. This method returns a string similarity percentage which will be weighted by the priorities for the corresponding search types. This constitutes the "destructive search types" part of the identity.
5. The remaining identity for the involved search types with non-destructive preparers is calculated by a final step using the **research** function, which is based on the frequency heuristic. This constitutes the "non-destructive search types" part of the identity.
6. The composed identity replaces the candidate identity.

The main purpose of the **research** functionality is to reconstruct the identity after the application of the **refine** function for search types implementing destructive linguistic preparers to capture misspellings. Because only components of an identity associated with destructive search types are replaced with the LRCPD metric (see **refine**) the intermediate identity is limited by the total of the corresponding weights. The remaining non-destructive search types re-evaluated with the **research** function and both results combined define the new identity.

You can **unjoin** search fields to restrict the effect of a **research** call on specific fields without having to readjust the priorities of the associated search types. Of course, this is cumbersome and may also require re-adjustment of priorities in case a search field encompasses search types with and without destructive preparers. By setting the parameter *lnondestructiveonly* to .t., the SearchEngine immediately selects only non-destructive search types for **research**, which works in tandem with the *Ldestructiveonly* parameter of the **refine** function.

The parameter *Iidentitymode* defines how the new identity relates to the existing identity for the respective candidate:

0 = do not update existing identity, i.e. to only update the score
1 = replace identity (default)
2 = maximize identity to overrule the existing identity if better
3 = minimize identity to enforce a worst case scenario

4 = add the researched identity to the existing identity
5 = calculate the average between the researched and existing identity

The value 4 (adding) is required if the identity is already only partial due to search field selection via **join** and **unjoin** or switching search types on and off via the priority (weight setting to zero/non-zero). *Identitymode* 2, 3 and 5 are of a more experimental nature. By default, the identity will be replaced (*Identitymode* = 1). This setting provides a high amount of flexibility to design your own composed identity to decouple the initial identity used for retrieval from bespoke similarity measures.

In contrast to the **refine** function, the **research** function can also re-evaluate the absolute identification potential (score). The parameter *Iscoremode* defines how the new score and the existing score interact:

0 = do not update existing score, , i.e. to only update the identity (default)
1 = always replace score
2 = maximize score by choosing the higher one
3 = minimize score by choosing the lower one

Srunfilter is a list of comma separated search runs to be filtered for researching. Ranges can be defined with a minus sign, e.g "1-3, 7, 9". This parameter allows to target the research on specific runs.

In general, you will rarely use the **research** or **refine** commands as their main purpose is the re-evaluation of linguistic search types, which is already integrated into the **search** function. Still, it is useful to create an order among the candidates without interfering with the retrieval, for example by applying a small **feedback** on specific fields:

`types("firm 70, street 10, zip 10, city 10")` The search type setting may deviate from retrieval.
`unjoin()`
`join("applicant", "firm")` We only want to apply feedback on the firm search field and associated types.
`feedback(5)` applies a small feedback on the firm name to create an order among the candidates.
`research(1)` replaces the identity to a maximum of 70% (no update of the score).
`unjoin()` resets the linkage of the search fields to exclude the firm name.
`join("street")`
`join("postcode", "zip")` You only have to specify the search field if the name is different.
`join("street")`
`research(4)` adds up to 30% to the current partial identity according to the address similarity.
`unjoin()` restoring the order of the search field linkage.
`join("applicant", "firm")`
`join("street")`
`join("postcode", "zip")`
`join("street")`
[Action>Research]
[refine]
[search]
[join]
[unjoin]
[feedback]

reset

```
reset()
```
sets all search parameters to their default value. **reset** is usually called before setting the parameters for a new search to provide a defined state. This command is equivalent to the following command sequence:

```
threshold(90)
cutoff(0)
activation(0)
```

```
feedback(0)
darwinian(.f.)
relative(.f.)
ignorant(.f.)
zealous(.f.)
```

All these settings can be configured in the **Config>Settings** menu of the **GUI**.
[Config>Settings]
[threshold]
[cutoff]
[feedback]
[activation]
[contain]
[darwinian]
[relative]
[ignorant]
[zealous]

result

`result(`*`Sresult`*`)`
sets the result table by specifying the file path in *Sresult*. As the result table will always be a Foxpro file, the default extension is ".dbf". This file will be created when a search is initiated. The **search** command provides multiple settings to determine how the current search run will be integrated into the result table.

It contains the retrieved candidates of previous search runs. Every result table has its own independent run counter reporting the number of search runs performed on it. Every entry consists of a record number field referring the search table called "searched", a record number of the candidate in the base table called "found", the identity, the absolute identification potential of the search term "score" and the "run" number of the first retrieval. Every combination of "searched" and "found" can only be retrieved once. If the same candidate is found at a later run, it will be ignored. Even if a run is without any new candidates, the run counter will be incremented. It is possible to rearrange and compress the runs with the **exportresult** function.

A result table is always more attached to a specific search table than to the base table, which can be attached to many search tables. Therefore, its file path should reflect its association with the search table. This can be done by naming it accordingly and/or by choosing a path close to the search table.
[Config>File Locations]
[search]

run

`run([`*`Spara`*`])`
displays the current run count of the active result table and optionally stores it into placeholder *Spara*.

safemode

`safemode([`*`Lsafemode`*`])`
activates the usage of visible Foxpro workers instead of invisible Parallelfox workers for multiprocessing when *Ssafemode* is .t.. This function is only available in the development environment and has no effect in the application.

save

`save([`*`Sslot`*`])`
saves the current SearchEngine structure string under the specified name in *Sslot*. You can list all available slots with the **list** function. The current slot can be displayed with the **slot** function and it is shown in the structure string, which

can be called with the **show** command. Omitting *Slot* will overwrite the current slot. The slot name has a maximal length of 40 characters. As it represents the name of the associated matching project, make it relevant. SearchEngine will always load the last saved slot at start up.

Remember that saving the current structure file has no effect on the internal SearchEngine index files. Creating a new Searchengine in the same directory will replace the existing index files. You cannot restore these by simply reloading a slot, which only contains paths of the involved tables and settings but for sure not a bunch of index files of considerable size.
[File>Save Settings]

say

`say([Ssomething])`
displays the specified text.

scope

`scope([Iscope])`
defines the width of the LRCPD scope. By default, *Iscope* is 12. This value determines the maximum distance of a wrong character from its original position. Because the error of a wrong (typo, misspelling) character diminishes with the length of the compared strings, it is necessary to restrict the scanning for a match. This also improves performance. If a character cannot be found in the scope range, it accrues the highest error value equivalent to a missing character.

The LRCPD string comparison is used by the **refine** function to impose a visual similarity on candidates retrieved by linguistic methods that may be to generous with that term.*Iscope* defines how tight the algorithm will squint its eyes to establish a sense of visual similarity of two strings. Before changing this parameter, consult the **discussion paper**.
[Config>Preferences]
[Discussion Paper]
[refine]

screen

`screen(Sproperty [, value])`
sets various screen properties determining the look of the execution window of the **Batch Mode** and to some extend of the command window:

`screen("width", 550)` sets the screen with to 550 pixel.
`screen("height",400)` sets the screen height to 400 pixel.
`screen("left",100)` sets the screen coordinate of the top left corner to 100 pixel.
`screen("top",100)` sets the screen coordinate of the top left corner to 100 pixel.
`screen("hide")` hides the screen.
`screen("maximize")` maximizes the screen.
`screen("minimize")` minimizes the screen.
`screen("normal")` switches the screen from maximized/minimized state into normal state .
`screen("backcolor", "0,0,0")` sets the back color of the screen in RGB format.
`screen("forecolor", "25,245,75")` sets the fore color of the screen in RGB format.
`screen("font", "Courier New")` sets the font.
screen("fontsize", 9) sets the font size in points.

Only the font and color related properties are supported by the command window.
[Batch Mode]
[File>Command]

setpara

`setpara(Spara [, value [, Lkeep]])`
sets the value of the placeholder *Spara* to the *value*, which can be of any type (string, numeric, logical). If *value* is omitted the placeholder will be empty. If *Lkeep* is .t. and the placeholder is already defined as command line parameter or by an earlier `setpara` call, the original value will not be changed.

As the name suggests, placeholders will be substituted before the execution of a command. They are enclosed in brackets. For example:
`setpara("mypara", "Hello World")` sets the placeholder *mypara*
`getpara("mypara")` displays "mypara=Hello World"
`say("[mypara]!!!")` displays "Hello World!!!"

Placeholders can also be defined with command line parameters (therefore the "para") of the SearchEngine batch mode call, for example:
`searchengine.exe myscript.se "mypara=Hello World"` (quotes are necessary in this case because spaces are used to separate command line parameters).
[getpara]
[Batch Mode]

search

`search([Iincrement] [, Icomparemode [, Lrefineforce] [, Nrefinelimit]])`
executes a search by sequentially transforming every record in the search table into a search term by transforming the search fields according to the associated search types. The heuristic for a search term is established by consulting the registry to attach frequencies to every word in the context of its search type. The frequencies are translated into relative identification potentials *RIP* weighted by the priority of the corresponding search type to represent a share of up to 100%. The settings defined by **ignorant** and **relative** may further influence the distribution of the *RIP*. The search algorithm retrieves candidates matching enough words with the search term that the total of the respective relative identification potentials meet the **threshold**. If the setting **zealous** is active, the **threshold** will be dynamically lowered to the highest identity of the candidates, if none of them will make it over the regular **threshold**. This initial list of candidates may undergo a **darwinian** selection process, allowing only those with the highest identity to advance.

After that, if the size of the list equals or exceeds the **activation** limit, **feedback** will be applied to partially transform the identity into a Jaccard index by discounting surplus words according to their respective *RIP*. This introduces variation among the candidates to be exploited by the **cutoff** mechanism picking the identity at a given position as the individual *threshold* for the candidate list, which is sorted by the adjusted identity in descending order. If both, **activation** and **cutoff**, have been specified (larger than zero) the **feedback** effect is considered to be only temporary to create variation and will not be reported. The **feedback** will persist when **feedback** is larger than zero and either **activation** or **cutoff** are zero or both, indicating that the variation is intended to be reported. In that case, the candidate list will be re-evaluated a final time according to the **threshold**, **zealous**, **darwinian** and **cutoff** settings.

The *Iincrement* parameter defines the interaction with existing results in the result table:

0 = **replace** all existing results by resetting/creating the result table (default)
1 = **complete** for unmatched search records by skipping search records that have already candidates
2 = **merge** results with candidates not yet found
3 = complete by resuming last run after cancelation (rarely used)
4 = merge by resuming last run after cancelation (rarely used)
-1 = complete by replacing last run (forgot something?)
-2 = merge by replacing last run (forgot something?)

Every call of the **search** method increments the run counter of the result table irrespective of retrieved candidates. All candidates are assigned to the run number of their retrieval. A candidate will never be overwritten except the whole result table will be replaced with *Lincrement* set to 0. If *Lincrement* is set to "complete" (1,3,-1) only search records that have no candidates yet in the result table are processes for the current search run. This is called an incremental search, which is best suited for a **search strategy** exploiting the properties of curated base tables. Merging is required when there is no dedicated best candidate because the base table has an ambiguous representation of entities in regard of the search topic. Merging is slower than completing as the SearchEngine will not skip search records. Existing candidates to a search record are supplemented with new candidates of the current search run.

By default, the SearchEngine does not acknowledge the presence of destructive search types and treats them like all other. This changes when the *Icomparmode* parameter is set to a value larger than zero. If there are any destructive preparers involved with the current search, the SearchEngine will initiate refinement of the new candidates as follows:

0. Continue if destructive search types were involved with the current search, otherwise ignore refinement request.
1. Identities are set to zero for the results of the current run.
2. The SearchEngine calls the **refine** method on the current run. It compares the original fields of the search and base table where the associated search types contain destructive preparers using the LRCPD method. This method returns a string similarity percentage which will be weighted by the priorities for the corresponding search types. This constitutes the "destructive search types" part of the identity.
3. The remaining identity for the involved search types with non-destructive preparers is calculated by a final step using the **research** function, which is based on the frequency heuristic. This constitutes the "non-destructive search types" part of the identity.
4. The composed identity replaces the candidate identity. It has to pass a *threshold* taking into account the **darwinian** and **zealous** settings. The *threshold* can differ from the general **threshold** used for retrieval.

The LRCPD comparison is not symmetric (commutative). The LRCPD method like the frequency based heuristic is a directed comparison ignoring surplus words in the target term. The direction of the comparison can be adjusted with *Icomparemode*:

0 = no refinement will keep the original identities (default)
1 = compare searched terms with found target terms
2 = dynamic - compare in both directions choosing the minimum
3 = compare found terms with searched target terms

Most of the time, we want that the refinement acts like the basic retrieval algorithm by ignoring surplus words in the candidate (*Icomparemode* = 1). Option 2 is suited for cases where additional words can constitute different entities, like person names. It is the equivalent to a high **feedback**. Option 3 can be applied when the search table has more noise and clutter than the base table. Although, the later case may be an indicator for the wrong search strategy.

If there are no destructive preparers involved with the current search, the SearchEngine will ignore the refinement request except it is enforced with *Lrefineforce*. A valid *Icomparemode* is the prerequisite for the following parameters. If *Lrefineforce* is .t., the complete identity will be replaced with a simple string similarity metric for all involved search fields according to the *Icomparemode* setting. If *Nrefinelimit* is specified, it will replace the subsequent *threshold* for the candidates after refinement, which, by default, is the general **threshold** used for retrieval.

Examples:
`search()` initiates a search with a clean result table. Most search strategies begin with this command.
`search(0)` is equivalent to search().
`search(1)` searches incrementally by only regarding search records that have no candidates yet.
`search(1, 1)` searches incrementally with refinement for candidates retrieved with linguistic methods.
`search(0, 1)` initiates a search with a clean result table using refinement.
`search(2)` merges the new search results with the existing ones without replacing.
`search(2, 1, 75)` refines the identities of the new search run before merging enforcing a threshold of 75% on the refined identities.
`search(2, 1, .t.)` refines all search fields regardless of the search type and uses the general threshold.
`search(2, 1, .t., 75)` refines all search fields regardless of the search type and enforces a post-refinement

threshold.
`search(-1)` replaces the last run with an incremental search. Maybe some settings were not right the run before.

Read the chapter about **search strategies** and the **discussion paper** to garner more insights about how to handle the SearchEngine.
[Action>Search]
[Discussion Paper]
[Search Strategy]
[threshold]
[cutoff]
[feedback]
[activation]
[contain]
[darwinian]
[relative]
[ignorant]
[zealous]
[refine]
[research]

show

`show()`
displays the SearchEngine structure string in all its glory.
[GUI]

slot

`slot()`
displays the current save slot name.
[File>Save]
[File>Load]
[save]
[load]
[list]

statistics

`statistics([Lwide])`
calculates and displays statistics about the current result table per run and in total. When *Lwide* is .t., the output will be in horizontal tab-delimited format instead of vertical list format.

You can get export the statistics by calling the statistics command in wide format with the **quiet** keyword in the Command Window after opening a log file with **output**. After closing the log file, you have a tab delimited table with column headers ready to be imported.
[Tools>Statistics]
[quiet]
[output]

strip

`strip([Nthreshold, Icutoff][, Linverse][, *Srunfilter])`
retroactively applies a **threshold** and **cutoff** on an existing result table by deleting the candidates not matching the requirements. This can be useful to trim the result table of unwanted candidates or even complete runs. With *Nthreshold* and *Icutoff* a temporary **threshold** and **cutoff** will be defined. Set the value to zero to dectivate the

corresponding requirement. You can skip both parameters if you do not intend to impose restrictions on the candidates. A *Nthreshold* above 100 will strip all candidates, which can be useful in conjunction with a run filter defined with the *Srunfilter* parameter.

Srunfilter is a list of comma separated search runs to be affected by stripping. Ranges can be defined with a minus sign, e.g. "1-3, 7, 9". You can change the run number for all entries in such a run list by appending the aggregate number after a colon, e.g. "1,5:1". To specify multiple aggregate run numbers, separate those list with a semicolon. The list are parsed from left to right and can overwrite each other. An aggregate run number of 0 will exclude the runs from preceding lists. For example: "1-9:2; 4,5:1; 3:0" is equivalent to "4,5:1; 1,2,6-9:2". Be aware that any run not explicitly included in *Srunfilter* will be excluded. If you only want to renumber runs skip the *Nthreshold* and *Icutoff* parameters. The maximum run number will be changed when actively changed to a larger number or to a number greater equal any existing run number, i.e. after aggregating runs 3, 4 and 5 to run 3 (3-5:3) the maximum run number 5 will be set to 3. The maximum run number is relevant for numbering new search steps.

By default, *Icutoff* defines the position in a list of candidates by search term sorted in descending order by identity. If the identity at this position is larger than the specified *Nthreshold* parameter, it will replace it to curtail the candidates of the respective search term. Of course, **cutoff** will be more efficient when **feedback** has been applied to create variance. Because the usual search strategy does use **feedback** only implicitly to **contain** weak search terms, you may have to introduce an explicit feedback effect with the **research** command that will actually change the identity. A low feedback of 20% should suffice.

The *Linverse* parameter directly refers to the **cutoff**. When it is .t., the *Icutoff* parameter still defines a position in a list, but this time it is sorted within the candidates. The **cutoff** removes search terms from a candidate that have a lower identity than the search term at the cutoff position. When you are forced to use a **compound search** because one table has to become the search table due to excessive noise while the search terms stem from a focused, curated database, you can exploit this feature to impose a retroactive darwinian approach on the results (after applying a feedback effect for variance). This will greatly reduce the number of false positives. If you skip the first two parameters but specify .t. for *Linverse*, a cutoff of 1 will be assumed.

Examples:
`strip(90, 0, "2,3")` strips all candidates of run 2 and 3 below 90%.
`strip(101, 0, "4")` removes run 4 (no identity will be larger than 100%).
`strip(0, 1)` applies a cutoff of 1 for all search terms. Together with a feedback of 10% this will only keep the best candidate(s).
`strip(0, 1, .t.)` applies an inverse cutoff of 1 for all canidates. Together with a feedback of 10% this will only keep the best search terms.
`strip(.t.)` equivalent to the command above.
`strip(90, 5, "3:2")` applies a threshold of 90% if the cutoff at 5 is lower for run 3, which will then be merged with run 2.
`strip("1,4:1; 2,3,5,6:2")` groups runs, i.e. conventional search steps and steps based on 3-gram preparer.
[Action>Strip]
[threshold]
[cutoff]
[research]
[contain]
[feedback]
[Compound Search]

threshold

`threshold(`*`Nthreshold`*`)`
sets the threshold for the identity of the candidates. It can be a number between 0 and 100. During **search** only candidates with identities equal or higher than the **threshold** are eligible to be reported. There are many more settings that affect the eligibility of a candidate (see **search**) but the **threshold** is the main parameter of a search

strategy due to its direct interaction with the weighting scheme of the search **types**.
[Config>Settings]
[search]
[types]

time

`time()`
displays the current date and time.

timer

`timer([`*`Ltimer`*`])`
activates a timer to measure the execution time for every action. If the action takes less than a second, the timer results will not be reported. If *Ltimer* is .t. or omitted, timing will be activated. By setting *Ltimer* to .f. timing can be deactivated.

In interactive mode (GUI), the timing results will be reported as comments in the **searchengine.log** file.
[Config>Preferences]

types

`types([`*`Ssearchtypes|Ldetail`*`])`
determines the search types settings. In the parameter *Ssearchtypes* Search types are separated by commas. A search type definition consists of the search type name, a priority, an optional offset (preceded with a plus or minus sign), an optional softmax parameter (preceded with a hash #) and an optional "log" keyword to enable logarithmic smoothing. It is advised to always specify all search types in order of definition as shown in the structure string (see **show**). If you skip a search type name in *Ssearchtypes*, it will get assigned a priority of zero.

Search types are defined with the **create** command. You can always reference them in the structure string evoked with the command **show** or in the main window of the GUI. To establish a **search strategy** you have to distribute weights over the search types according to their influence on the identification of an entity, your actual search topic. You may have addresses in your search and base table but the firm name is the most significant element in identifying the topic, which are companies. Naturally, the fields directly related to the search topic should get the dominant share of the weights. If your search topic would be addresses, i.e. as part of a geocoding exercise, distributing the weights evenly over the address fields is a good start. Because the weights must always add up to 100%, they are assigned as **priorities** which will be normalized to 100%. The terms weight and priority are used interchangeably throughout this document because priorities are just raw weights before normalization.

Search types are deactivated by assigning a priority of zero. This can be useful for search types associated with search fields that are not available in the search table, i.e. a missing street address in a company database. Deactivating search types can also be a part of a search strategy whereby destructive and non-destructive search types are alternatingly deactivated to separate conventional from linguistic search runs. This is most effective for search strategies employing incremental searches on curated base tables.

Besides the priority, a search type has multiple ways to adjust the calculation of the relative identification potential *RIP* of the words of the associated search field:

The *offset* component directly follows the priority separated by a space. It can be negative. The *offset* of a search type will be added to the word frequency of every word in the search field before the calculation of the *RIP*. This smooths the distribution of the potentials. A negative offset will reduce the frequencies up to a minimum of 1 to curtail the frequency distribution. If a negative *offset* exceeds the highest frequency the *RIP*s become homogenous transforming the frequency based into a word based heuristic. This is the most relevant usage of this otherwise outdated smoothing method.

The *log* component is declared with the keyword "log" at the end of a search type definition. It is a more common approach to smooth the *RIP* distribution of a search type. Before the calculation of the *RIP*, every single frequency will be transformed to its natural logarithm (base e). This is a much more intuitive smoothing method than the arbitrary definition of an additive numeric *offset*. Log smoothing should be part of any search strategy using search types with n-grams, because of the skewed distribution of the frequencies that may favor the fragments containing the typo/misspelling. Searching with and without log smoothing in such a case may yield complementary results.

The *softmax* component of a search type is declared with a preceding hash. This parameter is of experimental nature as it accentuates the differences in the *RIP* distribution instead of smoothing them. It requires the frequencies of the other words to be calculated. Therefore, it cannot be integrated into the **feedback** effect as the other smoothing methods. The feedback will ignore the *softmax* adjustment. The parameter has the following effects: a value below 1 attenuates the distribution, while a value above 3 will accentuate the distribution. The maximum accentuation is reached with a value 30, which will almost negate the existence of other words besides the one with the highest *RIP*. This parameter can be used to highlight the keywords of longer texts like patent titles or abstracts.

Examples: `types("firm 70, firm 0, street 10, zip 10, city 10")`
sets the main focus on the first search type of the search field firm, which is non-destructive, while deactivating the second firm search type, which is destructive. The remaining priorities are evenly distributed over the address search types.
`types("firm 0, firm 7, street 1, zip 1, city 1")`
switches the main priority to the destructive search type for the firm field while deactivating the conventional search type. The priorities do not have to be percentages because they will always be internally normalized into weights representing shares of 100%.
`types("firm 0, firm 70 log, street 10, zip 10, city 10")`
activates log smoothing on the destructive search type of the firm field for a consecutive search run.
`types("firm 40 -9999999, firm 0, street 20, zip 20, city 20")`
subtracts a smothering offset from the frequencies of the firm name field replacing the frequency based with a word based heuristic. The focus is slightly shifted to the address fields.
`types("firm 40 log, firm 0, street 20 log, zip 20, city 20")`
reduces the dominance of rare words in the firm name and the street address. Especially street addresses may benefit from log smoothing because they have less insubstantial noise than firm names. Furthermore, house numbers are skewed towards the lower range as every street has a house number 1 but rarely a number 777 but both numbers have the same identification property. Log smoothing will alleviate this context blindness of the heuristic.
`types("abstract 100 #7")`
enforces an accentuation on keywords in the abstract field with a softmax value of 7.

In principle it is possible - but not advised - to combine different smoothing methods, like "firm 10 -999 #7 log". They will applied in the following order: offset, softmax, log. The ramifications of combined smoothing methods are difficult to assess, especially in the context of a heuristic that necessarily is based on an abstracted simplification of the matching issue.

If you omit the *Ssearchtypes* parameter or set *Ldetail* to .f., the command will show you the current search types as comma separated list. In the command window you can copy this list as template for the *Ssearchtypes* parameter. When *Ldetail* is .t., detailed information about the search types and their position will be shown. Destructive search types are marked with an exclamation mark after a destructive preparer. The position is relevant for the **exportmeta** command.
[Config>Search Types]
[Search Strategy]
[search]
[show]
[feedback]
[exportmeta]

unjoin

`unjoin([Sfield])`
removes the link between a field with the name *Sfield* of the search table and the associated search field. If *Sfield* is omitted, all links will be revoked. It is best practice to first unjoin all links before joining them.
[Config>Join Search Fields]
[join]

version

`version()`
displays the current version.
[show]

wait

`wait([Iseconds])`
halts execution for *Iseconds* seconds or until a button press. If *Iseconds* is zero or omitted, execution will be continued after a button press.

zealous

`zealous([Lzealous])`
defines whether the SearchEngine is dynamically lowering the **threshold** to guarantee matches or is complying by disposing all candidates below it, which is the default behavior. If *Lzealous* is .t., the SearchEngine will use the identity of the best candidate as a the new threshold, if no candidate would make it over the regular **threshold**. As this may lead to considerably inflated candidate list, this setting should be used with care. In general, a zealous SearchEngine is only recommended for small search projects.

In case of an incremental **search strategy**, where the number of searched records dwindles from search run to search run, even a large search project can become relatively small, when there are only few search records without candidates left. This can be determined with the **statistics** function. If this is the case, a last search run can be a zealous search. Usually, this search should employ destructive search types to capture misspellings, which may be the main reason for missing candidates, besides having no valid match anyway.

In case of a merging search strategy that does not skip search records with candidates, the proceedings are quite similar. First, perform the normal search runs without the *zealous* option followed by assessing the number of search records without candidates. If this number seems reasonably small, there is a high overlap between the search and the base table. A final *zealous* search will not be harmful in such an environment. The *zealous* search should be incremental (see **search**) because it will not find new candidates for search records that already have matches given the **threshold** and other search settings will not be changed.

This option has a high risk of collecting false positives when the expected overlap between search and base table is small. For example searching firma usually not involved with patenting, like wineries, in patent data. Furthermore, even a zealous SearchEngine will not guarantee a match for every search term when the physical limitations of the search **depth** are exceeded.
[Config>Settings]
[Search Strategy]
[search]
[statistics]
[threshold]
[depth]

Preparers

Preparers are directives that can be attached to search fields of the base table. Multiple preparers can be attached to one field to form a search type. But even if a search field has no explicity assigned preparer, it will be harmonized by the default preparer, which transforms all characters to upper-case, removes all special, non-alpha-numeric characters and replaces all Umlauts, Apostrophes and other character mutations with their inoffensive ASCII representations. Beginning with this default preparer, the directives will be processed from the first preparer in the list to the last one. A directive can be a harmonization of synonymous words or group of words but also a linguistic tokenization replacing the word based separation. Because preparers are applied to search fields, they affect the base table in the same way as the search table by always guaranteeing a synchronous harmonization, which does not rely on manual preprocessing of data. This helps in reducing the potential error sources caused by manual data preparation. The SearchEngine has an assortment of built-in preparers to not only handle the usual harmonization task but also to provide linguistic methods that would be difficult to implement otherwise. Besides those internal preparers, custom preparers can be defined using simple xml-definitions to implement solutions especially for language related issues, like contracted words. They can be used automatize frequent harmonization tasks before matching.

You can use the **showpreparer** function in the command window to see the underlying xml-script for a preparer. A preparer uses a sequence of simple commands to manipulate the input string. To get a better understanding of specific preparers, use the **prepare** function to try out a single preparer or combinations of them. Finally, you can reload custom preparers with **loadpreparer** during development to test them without restarting the SearchEngine.
[showpreparer]
[loadreparer]
[prepare]

Built-in Preparers

These preparers provide a foundation for simple harmonization or transformation tasks and linguistic approaches independent of language related particularities. Built-in preparers mostly implement a single command with different parametrizations.

NOABBREV

contracts all contiguous single characters into a word: "B.a.s.f. Ölplattform" > "BASF OELPLATTFORM"

NOUMLAUT

replaces all occurences of "OE", "AE" and "UE" with the first letter to replicate a typical error in American spelling of German words: "Ölplattform" > "OLPLATTFORM"

SEPNUM

separates numbers from letters: "B.A.S.F Ölplattform71.3w" > "B A S F OELPLATTFORM 71 3 W"

GRAM2

implements 2-grams: "Ölplattform" > "OE EL LP PL LA AT TT TF FO OR RM"

GRAM3

implements 3-grams: "Ölplattform" > "OEL ELP LPL PLA LAT ATT TTF TFO FOR ORM"

GRAM4

implements 4-grams: "Ölplattform" > "OELP ELPL LPLA PLAT LATT ATTF TTFO TFOR FORM"

GRAM5

implements 5-grams: "Ölplattform" > "OELPL ELPLA LPLAT PLATT LATTF ATTFO TTFOR TFORM"

METAPHONE

implements the linguistic Metaphone transformation: "Ölplattform" > "OLPLTFRM"

SOUNDEX

implements the linguistic Soundex transformation: "Ölplattform" > "O7176298"

COLOGNE

implements the linguistic Cologne transformation according to the "Kölner Phonetik": "Ölplattform" > "05152376"

LINK

joins preparer sequences together. Every sequence is based on the orginal data.
MAXWORDS2 MAXLENGTH1 NOABBREV LINK LASTWORDS1: "Maria Helena Andromachi" > "MH ANDROMACHI"

DESTRUCTIVE

declares the search type destructive regardless of the presense of destructive preparers. Use this preparer to enforce refinement for the associated search type (see **search** command.
[search]

MAXLENGTH1

keeps only the first letter of every harmonized word: "B.a.s.f. Ölplattform" > "B A S F O"
This preparer can be used to create initials.

MAXLENGTH2

keeps only the first 2 letters of every harmonized word. "B.a.s.f. Ölplattform" > "B A S F OE"

MAXLENGTH3

keeps only the first 2 letters of every harmonized word. "B.a.s.f. Ölplattform" > "B A S F OEL"

MAXLENGTH7

keeps only the first 7 letters of every harmonized word. "B.a.s.f. Ölplattform" > "B A S F OELPLAT"

MAXLENGTH10

keeps only the first 2 letters of every harmonized word. "x" is a placeholder and must be replaced with a number between 1 and 15.

MAXLENGTH15

keeps only the first 15 letters of every harmonized word: "Methionylthreonylthreonylglutaminylarginyl..." > "METHIONYLTHREON" This preparer can be used to curtail unnecessarily long words of some languages or expert terminology. By the way, the word in the example describes a protein and has actually 189819 letters.

MAXLENGTH20

keeps only the first 20 letters of every harmonized word. The SearchEngine will automatically truncate words longer than 30 characters.

MAXWORDS1

keeps only the first harmonized word: "Paul Atreides" > "PAUL"

MAXWORDSx

keeps only the first x harmonized words. "x" is a placeholder and must be replaced with a number between 1 and 5.

MAXWORDS10

keeps only the first 10 harmonized words.

MAXWORDS20

keeps only the first 20 harmonized words.

MAXWORDS40

keeps only the first 40 harmonized words.

SKIPWORDS1

skips the first harmonized word: "Paul Atreides" > "ATREIDES"

SKIPWORDSx

skips the first x harmonized words. "x" is a placeholder and must be replaced with a number between 1 and 5.

SKIPWORDS10

skips the first 10 harmonized words.

SKIPWORDS20

skips only the first 20 harmonized words.

SKIPWORDS40

skips only the first 40 harmonized words.

LASTWORDS1

keeps the last harmonized word: "Maria Helena Andromachi" > "ANDROMACHI"

LASTWORDSx

keeps the last x harmonized words. "x" is a placeholder and must be replaced with a number between 1 and 5.

LASTWORDS10

keeps the last 10 harmonized words.

LASTWORDS20

keeps the last 20 harmonized words.

LASTWORDS40

keeps the last 40 harmonized words.

Custom Preparers

Custom preparers are mostly used to standardize and automatize regular harmonization activities before matching. If you use the SearchEngine regularly and caught yourself by repeating the same harmonization steps over and over, you should think about implementing this routine as a preparer. In general, the pre-processing effort depends on the search context and the particularities of the involved languages. For example, the English language is relatively effortless and usually does not require custom preparers. On the other hand, for the German language custom preparers may improve the search success noticeably because it allows the contraction of words. Especially street names are affected by inconsistent usage of this feature. The Github package has a sub-directory "preparer" with a (hopefully) growing collection of preparer definitions. The file "searchengine_firma_strasse.xml" contains preparers to handle German firm and street names. The street name preparer "STRASSE" separates the most common street types, like "Straße", "Allee", "Platz", "Weg" and so on, from any preceding street names. The firm name preparer "FIRMA" harmonizes abbreviations and legal form variations. To make those preparers available, you have to copy the corresponding xml-files into the engine directory.

At startup, the SearchEngine scans the engine directory for xml-files that start with "searchengine", like "SearchEngine.xml", "searchengine_my_preparers.xml" and so on. It will extract all preparers from those files to be used for search type definitions of the **create** command. If an error is found in a definition, it will be reported in the structure string. Every preparer must have a unique name over all xml-files in the engine directory including built-in preparers. After you have used custom preparers to create a SearchEngine, you should not make any changes to the participating preparers.

The preparer definitions in an xml-file have the following structure:

```
<searchengine>
        <preparer>
                <type>preparer_name</type>
        </preparer>
        <command>
                <com>command</com>
                <para1>parameter1</para1>
        </command>
        <command>
                <com>command</com>
                <para1>parameter1</para1>
                <para2>parameter2</para2>
                <para3>parameter3</para3>
        </command>
        <command>
                <com>command</com>
        </command>
        <preparer>
                <type>preparer_name</type>
                <com>command</com>
                <para1>parameter1</para1>
                <para2>parameter2</para2>
        </preparer>
</searchengine>
```

The xml-file is initiated with the <searchengine> tag. A preparer definition starts with the <preparer> tag enclosing the preparer name within the <type> tag. All commands of the preparer follow in separate <command> tags. A command has a name in the <com> tag and may have numbered parameters in corresponding <para#> tags, i.e. <para3> for the 3rd parameter. Parameters are usually numbers or strings. Because any command will only be applied on already harmonized data, the parameters will always be automatically harmonized before they become a part of a preparer definition. You can omitt some parameter including the tags for the default value (see description). If a preparer consists of only one command the preparer definition and the command can be put together into the <preparer> tag (see last preparer definition in the structure example). You can integrate comments by enclosing them with the "<!--" and "-->" tags.

The following commands are available:

blank

`<com>`**`blank`**`</com><para1>`*`characters`*`</para1>`
All characters defined in the 1st parameters are blanked in the data. You can remove numbers or letters from the data with this command.

Example:
`<com>`**`blank`**`</com><para1>ABCDEFGHIJKLMNOPQRSTUVWXYZ</para1>`
"DE 67063 Ludwigshafen" > "67063"

call

`<com>`**`call`**`</com><para1>`*`preparer_name`*`</para1>`
This command calls another installed preparer, which will be executed on the data. This allows to define routines to be included without repeating the definitions or to make additions to existing preparer without copying the definitions into a new preparer.

change

`<com>`**`change`**`</com><para1>`**`left|right|word|free`**`</para1><para2>`*`from_string`*`</para2><para3>`*`to_string`*`</para3>`
The **change** command is the safe version of the **replace** command. The *from_string* will be replaced with the *to_string* as long as the occurrence is aligned according to the 1st parameter: **left** aligned, **right** aligned, **word** identity or **free** of any alignment requirements (default). It this In contrast to the **replace** command, the changed part will be enclosed is brackets to be referenced by other commands, like **split** or **replace**. This is helpful, when longer phrases are abbreviated to a short form with change and you want to include those in subsequent commands without inadvertently changing original content. The brackets marking the changes can be removed with the **cleanup** command.

Example:
```
<command>
        <com>change</com>
        <para1>right</para1>
        <para2>GESELLSCHAFT</para2>
        <para3>GES</para3>
</command>
<command>
        <com>change</com>
        <para1>right</para1>
        <para2>GESELL</para2>
        <para3>GES</para3>
</command>
<command>
        <com>split</com>
        <para1></para1>
        <para2>[GES]</para2>
</command>
```

This sequence will change longer forms of a term into a shorter representation and then separates only changed occurrences of this abbreviation from the preceding word. Of course, the brackets will be removed before the next preparer will start its execution.
[cleanup]
[replace]
[split]

cleanup

`<com>`**`cleanup`**`</com>`
The **change** brackets will be removed. Of course, the brackets will also be removed automatically before the next preparer will start its execution. Alternatively, you can use the "reset" command, which is deprecated because it can be confused with the script function of the same name.
[change]

cockle

`<com>`**`cockle`**`</com>`
The command **Cockle** contracts all contiguous single characters into a word: "B A S F SE" > "BASF SE"

cut

`<com>`**`cut`**`</com><para1>`**`left|right|both`**`</para1><para2>`*`string`*`</para2>`
The **cut** command removes trailing (**left**), leading (**right**) or adjoining (**both**) parts of a word containing the 2nd parameter *string*. This command can be used to truncate country specific variants of the same root word.

Examples:
`<com>`**`cut`**`</com><para1>`**`left`**`</para1><para2>pre</para2>`
"MARKDOWN PREINSTALLED" > "MARKDOWN PRE"
`<com>`**`cut`**`</com><para1>`**`right`**`</para1><para2>down</para2>`
"MARKDOWN PREINSTALLED" > "DOWN PREINSTALLED"
`<com>`**`cut`**`</com><para1>`**`both`**`</para1><para2>install</para2>`
"MARKDOWN PREINSTALLED" > "MARKDOWN INSTALL"
`<com>`**`cut`**`</com><para1>`**`left`**`</para1><para2>universi</para2>`
`<com>`**`replace`**`</com><para1>`**`right`**`</para1><para2>universi</para2><para3>university</para3>`
"LISBOA UNIVERSIDAD" > "LISBOA UNIVERSITY"
"UNIVERSTAET BERLIN" > "UNIVERSITY BERLIN"

destructive

`<com>`**`destructive`**`</com>`
By declaring a preparer **destructive**, search types using it will be treated differently during the refinement step of the **search** command. Apply this tag to any custom preparer that destroys information for the sake of retrieval where a "visual" confirmation during refinement seems appropriate.
[search]

encode

`<com>`**`encode`**`</com><para1>`**`metaphone|soundex|cologne`**`</para1>`
Use the **encode** function to get access to methods of computational linguistics. The methods are: **metaphone**, **soundex** and **cologne**. They all have in common that they are specialized for specific languages. **Soundex** is better suited for English, while **Cologne** has a German background. **Metaphone** is a revised version of the **Soundex** algorithm. Even though these algorithm are quite robust, we suggest to use the **gram** method to handle misspellings because it also can mitigate concatenated or separated words.

Examples: `<com>`**`encode`**`</com><para1>`**`metaphone`**`</para1>`
"OELPLATTFORM" > "OLPLTFRM" `<com>`**`encode`**`</com><para1>`**`soundex`**`</para1>`
"OELPLATTFORM" > "O7176298" `<com>`**`encode`**`</com><para1>`**`cologne`**`</para1>`
"OELPLATTFORM" > "05152376"

gram

`<com>`**`gram`**`</com><para1>`*`length`*`</para1>`
This function fragments all words into so called grams. The size of these overlapping tokens is defined by the 1nd parameter *length*. This is the most powerful linguistic function because it not only provides tolerance for misspellings but also for erroneously contracted or separated words.

Example:
`<com>`**`gram`**`</com><para1>`**`3`**`</para1>`
"SEARCHENGINE IS A POWERFUL TOOL" > "SEA EAR ARC RCH CHE HEN ENG NGI GIN INE IS A POW OWE WER ERF RFU FUL TOO OOL"

join

`<com>`**`join`**`</com>`
With this command you can concatenate independent command sequences respectively based on the full data. Multiple command sequences can be **joined**. If you specify join as the only command of the preparer, it will declare a special join preparer (see built-in "link" preparer).

Example:
```
<command>
        <com>limit</com>
        <para1>count</para1>
        <para2>2</para2>
        <para3>left</para3>
</command>
<command>
        <com>limit</com>
        <para1>length</para1>
        <para2>1</para2>
        <para3>left</para3>
</command>
<command>
        <com>cockle</com>
</command>
<command>
        <com>join</com>
</command>
<command>
        <com>limit</com>
        <para1>count</para1>
        <para2>1</para2>
        <para3>right</para3>
</command>
```
"MARIA HELENA ANDROMACHI" > "MH ANDROMACHI"
[link]

limit

```
<com>limit</com><para1>length|char|count|word</para1><para2>num</para2><para3>left|skip|right</para3>
```
Limits the length of strings when the 1st parameter is **length** and curtails words when it is **count**. Alternatively, you can use the keywords **char** for **length** and **word** instead of **count**. The number of characters or words that are kept is defined by the 2nd parameter *num*. If the characters or words are count from the **left** or **right** is specified by the 3rd parameter. If **skip** is declared the number of characters or words are skipped from the left. The last parameter can be omitted for the default setting: **left**.

Examples:
```
<com>limit</com><para1>length</para1><para2>1</para2>
```
"MARIA HELENA ANDROMACHI" > "M H A"
```
<com>limit</com><para1>length</para1><para2>2</para2><para3>left</para3>
```
"MARIA HELENA ANDROMACHI" > "MA HE AN"
```
<com>limit</com><para1>char</para1><para2>5</para2><para3>skip</para3>
```
"MARIA HELENA ANDROMACHI" > "A MACHI"
```
<com>limit</com><para1>count</para1><para2>1</para2><para3>right</para3>
```
"MARIA HELENA ANDROMACHI" > "ANDROMACHI"
```
<com>limit</com><para1>word</para1><para2>1</para2><para3>skip</para3>
```
"MARIA HELENA ANDROMACHI" > "HELENA"

replace

```
<com>replace</com><para1>left|right|word|free</para1><para2>from_string</para2><para3>to_string</para3>
```
The *from_string* will be replaced with the *to_string* as long as the occurence is aligned according to the 1st parameter: **left** aligned, **right** aligned, **word** identity or **free** of any alignment requirements (default). You will use this command to harmonize different spellings or abbreviations of common terms.

Examples:
```
<com>replace</com><para1>right</para1><para2>GESELLSCHAFT</para2><para3>GES</para3>
```
"BASF AKTIENGESELLSCHAFT" > "BASF AKTIENGES"
```
<com>replace</com><para1>word</para1><para2>AKTIEN GES</para2><para3>SE</para3>
```
"BASF AKTIENGES" > "BASF SE"
```
<com>replace</com><para1>word</para1><para2>AKTIEN GES</para2><para3>SE</para3>
```
"BASF AKTIENGES" > "BASF SE"

separate

```
<com>separate</com><para1>characters1</para1><para2>characters2</para2>
```
This command separates all characters defined by the 1st parameter from all characters defined by the 2nd parameter preventing that any characters of both groups will ever touching each other. It is usually used to separate numbers from letters.

Example:
```
<com>separate</com><para1>0123456789</para1><para2>ABCDEFGHIJKLMNOPQRSTUVWXYZ</para2>
```
"32ND STREET" > "32 ND STREET"

split

```
<com>split</com><para1>left|right|free</para1><para2>separate_string</para2><para3>both|left|right</para3>
```
The command will separate the *separate_string* from the adjoined word when it is aligned according to the 1st parameter: **left** aligned, **right** aligned or not aligned with **free**. The 3rd parameter can be omitted. It dictates where the separating blank will be inserted: **both** separates the string on both sides (default), **left** separates it on the left side and **right** separates it on the right side.

Example:

```
<com>split</com><para1>right</para1><para2>strasse</para2>
```

"HAUPTSTRASSE 17" > "HAUPT STRASSE 17"

```
<com>split</com><para2>schul</para2><para3>left</para3>
```

"HAUPTSCHULVERWALTUNG" > "HAUPT SCHULVERWALTUNG"