Record Linkage

Requires: numpy, matplotlib

Tested on Python 2.7.

Utility functions

The records in a database were perturbed (simulating typographical errors) and written to a second file. A sample of 700 original records were written to a file *recs1.csv*, and a sample of 400 perturbed records were written to a file *recs2.csv*. These are used to demonstrate the use of the code. The number of original records was 1000.

```
>>> import record_linkage
>>> names1, recs1 =
record_linkage.parse_file('/home/duncan/Documents/linkage/RSS/recs1.csv')
>>> names2, recs2 =
record_linkage.parse_file('/home/duncan/Documents/linkage/RSS/recs2.csv')
```

Although the records are already aligned we can demonstrate the use of the *aligned* function to marginalise to the key variables and the unperturbed unique ID field. The ID field is (arbitrarily) placed at the end of each record.

```
>>> recs1 = record_linkage.aligned(recs1, [0,2,3,8,7])
>>> recs2 = record_linkage.aligned(recs2, [0,2,3,8,7])
>>> recs1[0]
['1', '1976', '1', 'alan', '1609883']
```

Let's generate the record pairs.

```
>>> import itertools
>>> pairs = list(itertools.product(recs1, recs2))
```

As we need similarity scores we'll import a couple of functions from the *pseudonymization* library (https://github.com/DuncanSmith147/pseudonymization). Other string similarity measures can be found in the *jellyfish* library (https://pypi.python.org/pypi/jellyfish). We tokenize only the key variables, indexed [0,1,2,3], making sure that we have some way of indicating a missing value. Here we tokenize strings into padded *n*-grams and retain the empty string used in the underlying datasets to indicate missing values.

```
>>> from pseudo import Jaccard
>>> from tokenization import n_grams
```

```
>>> tokenized1 = [[n_grams(val, 2, True) if val else '' for val in rec[:4]]
for rec in recs1]
>>> tokenized2 = [[n_grams(val, 2, True) if val else '' for val in rec[:4]]
for rec in recs2]
>>> tokenized2[8]
[['_1', '1_'], ['_1', '19', '96', '68', '8_'], '', ['_d', 'da', 'av', 'vi',
'id', 'd_']]
```

Having tokenized into padded bigrams we can generate an array of Jaccard similarity scores. We create a list of pairs of tokenized records that corresponds to our pairs of records. We only process the key variables – indexed [0,1,2,3] since we marginalised to the key variables earlier and removed the ID field when we tokenized. We use the Jaccard score for each field, although we could choose distinct similarity measures for distinct fields.

```
>>> scores = record_linkage.simscores(list(itertools.product(tokenized1,
tokenized2)), [0,1,2,3], [Jaccard]*4, missing='')
>>> scores.shape
(280000, 4)
>>> scores[8]
array([  1.00000000e+00,   2.50000000e-01,  -9.99999990e+07,
         0.00000000e+00])
```

scores is an array of shape (*N*, *n*) corresponding to *N* record pairs and *n* variables. Note that when missing values prevent a similarity score from being generated the value -99999999 is inserted into the data array.

The above simply demonstrates the use of utility functions. Subsequent record linkage simply depends on having an array of shape (*N*, *n*) containing similarity scores with (large enough) negative values to denote missing scores.

Linkage

Binary EM

The traditional Fellegi-Sunter approach based on binary comparisons (Fellegi and Sunter, 1969). Values are either equal or unequal. For an appropriate similarity score this comparison would correspond to scores of 1 or 0. The Jaccard score is a set similarity measure, so our data array does not contain exactly this information. (If we had included the full string as a token, then we would have had a 1 to 1 correspondence between equality and scores of 1.) But by choosing an appropriate threshold we can dichotemize the values to demonstrate Binary EM.

```
>>> binary_scores = record_linkage.categorized(scores, [-0.01, 0.9999, 1])
>>> binary_scores.shape
(280000, 2, 4)
```

Scores have been categorized into 2 intervals, [0, 0.9999] and (0.9999, 1]. The list passed to the function contains a lower bound followed by *c* upper bounds. Upper bounds are inclusive, whilst

lower bounds are not. We supplied a small negative value as the first list item – so that the first interval will include 0 whilst not including the negative value used to designate missing values.

We use an EM algorithm (Jaro, 1989) to estimate *u* and *m* probabilities, and *p* as defined below. Missing values are correctly handled under the missing at random assumption. They do not contribute terms in the calculation of likelihoods or expected values. No attempt is made to impute them.

We have $i=1,\ldots,n$ variables and $j=1,\ldots,N$ record pairs. We have a set *M* of correct matches and a set *U* of incorrect matches.

$\gamma_i^j=0$ if attribute *i* differs for record pair *j*, and $\gamma_i^j=1$ if attribute *i* matches for record pair *j*.

$$m_i=Pr(\gamma_i^j=1|r_j\in M)$$

$$u_i=Pr(\gamma_i^j=1|r_j\in U)$$

$$p=\frac{|M|}{|M\cup U|}$$

As the vast majority of record pairs are incorrect matches we can use the data to estimate the *u* probabilities by treating all pairs as incorrect matches.

```
>>> u_probs
array([[ 0.53832298,  0.98358571,  0.91553912,  0.99724643],
       [ 0.46167702,  0.01641429,  0.08446088,  0.00275357]])
```

This is the default starting value for the *u* probabilities for the EM algorithm. Other defaults are selected for the *m* probabilities and *p*. In fact fixed values can be supplied for the *u* probabilities and / or *p* and these quantities will remain fixed. Here we will use the default values, although they are all shown in this first function call for illustrative purposes.

```
>>> u, m, p, g_m, log_likelihoods = record_linkage.EM(binary_scores, u=None,
p=None, u_start=None, m_start=None, p_start=None, u_prior=1, m_prior=1,
p_prior=1, max_its=10000, tol=1e-12)
```

The returned values are the estimates for the *u* probabilities, the *m* probabilities and *p*; as well as the posterior probabilities of a correct match, *g_m*, and the log likelihoods generated by the fitting algorithm.

```
>>> u
array([[  5.40695367e-01,   9.84716329e-01,   9.16832900e-01,
          9.99598511e-01],
       [  4.59304633e-01,   1.52836708e-02,   8.31670999e-02,
```

```
          4.01489457e-04]])
>>> m
array([[ 0.0596773 ,  0.75369262,  0.65260658,  0.51898684],
       [ 0.9403227 ,  0.24630738,  0.34739342,  0.48101316]])
>>> p
0.0048939343591164934
```

We can see that the *u* probabilities have not changed much. But the *m* probabilities for the highest similarity score level look low (which they are, given the level of perturbation applied to the data), and *p* is about 5 times greater than it should be (it is actually 0.001). We can plot the log likelihoods.

```
>>> record_linkage.log_likelihood_plot('binary_scores.jpg', log_likelihoods)
```

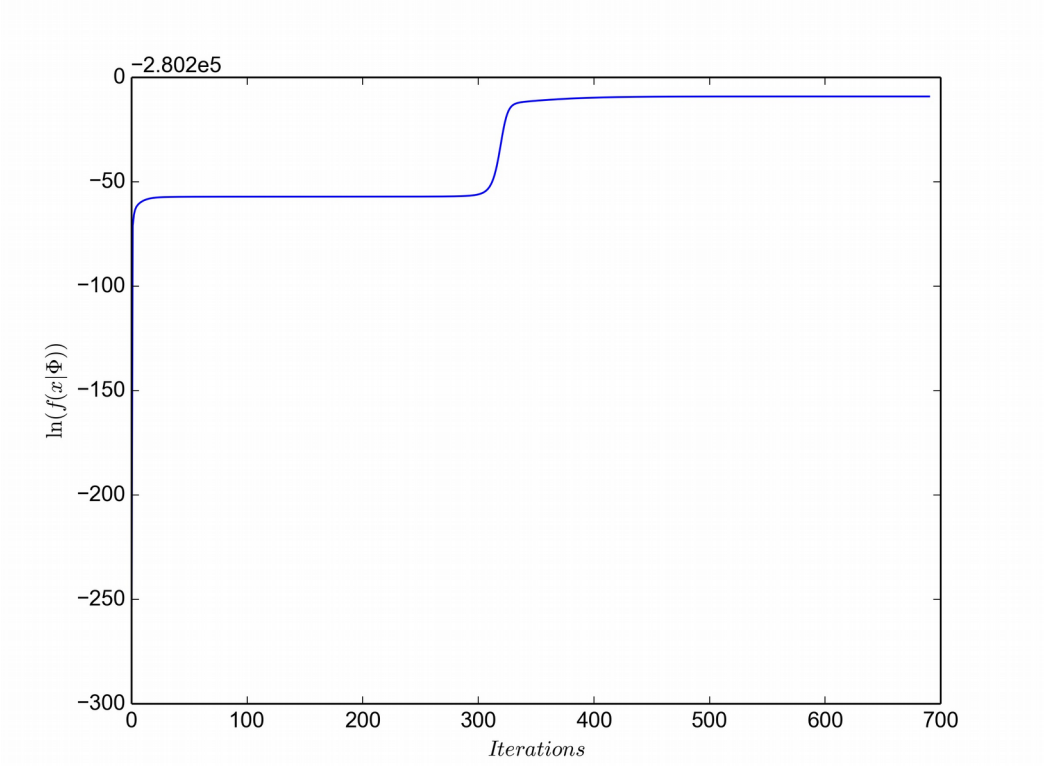

Illustration 1: Log likelihoods for Binary EM

This is all suggestive of over-fitting. There are a number of ways to deal with this. But first we can generate a data series that can be used to generate plots that can be used to assess linkage performance. Note that we require the original record pairs and the index of the ID field to assess true match status, as well as our array of correct match probabilities. (Of course, in real world circumstances we will not be able to do this.)

```
>>> bin1 = record_linkage.series(pairs, 4, g_m)
```

Dealing with over-fitting

One approach that can often deal with over-fitting is to reduce the number of parameters being

estimated by fixing some of them. We generated reasonable estimates for the *u* probabilities earlier. So we can supply them and only estimate the *m* probabilities and *p*.

```
>>> u, m, p, g_m, log_likelihoods = record_linkage.EM(binary_scores,
u=u_probs)
>>> m
array([[ 0.0796533 ,  0.24655305,  0.06201338,  0.26048561],
       [ 0.9203467 ,  0.75344695,  0.93798662,  0.73951439]])
>>> p
0.00099409099533202751
```

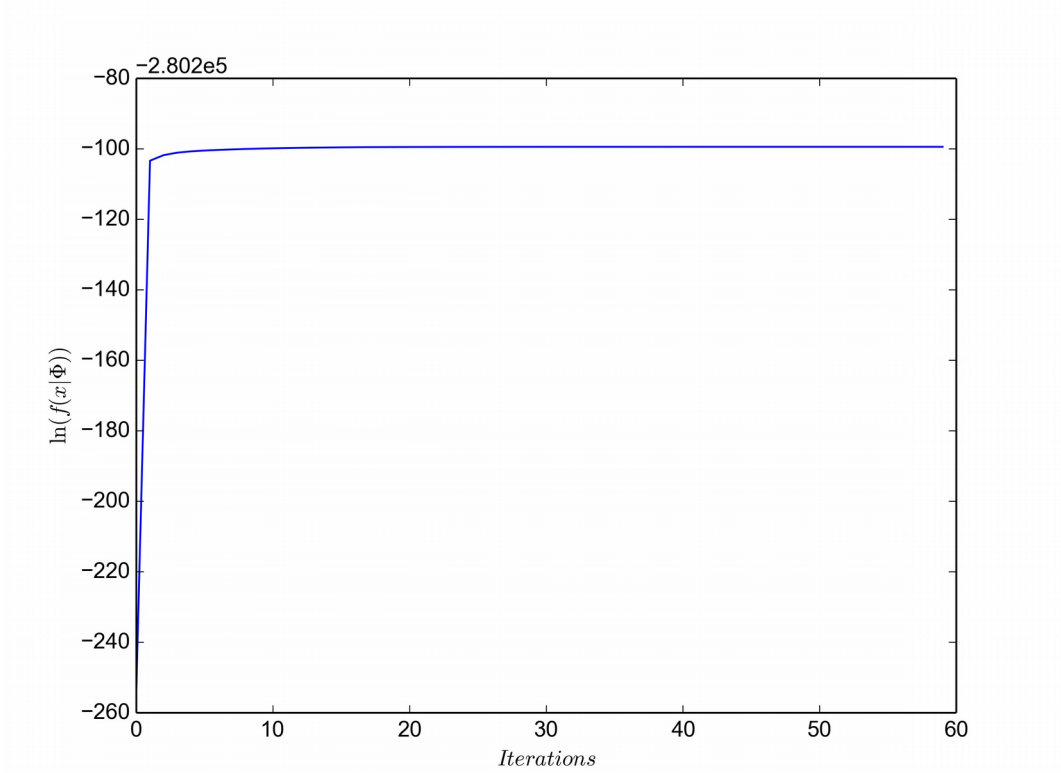

Illustration 2: Log likelihoods for Binary EM with fixed u probabilities

The resulting *m* probabilities look far more plausible, *p* is close to its true value (although we wouldn't generally know what it should be), and the plot looks reasonable. Note also that convergence requires many fewer iterations. Let's create another data series for subsequent plotting before we over-write the returned values.

```
>>> bin2 = record_linkage.series(pairs, 4, g_m)
```

EpiLink

We can also use the EpiLink algorithm for linkage (Contiero et al., 2005). We can supply weights directly, or supply error rates and 'average frequencies'. What Contiero et al. term 'average frequencies' is just the reciprocal of the number of distinct values for a variable. We can use the error rates generated by the binary EM. There is a function for generating the 'average frequencies'.

```
>>> av_freqs = record_linkage.average_frequencies(recs1+recs2, [0,1,2,3],
missing='')
>>> Epi_scores = record_linkage.EpiLink(scores, w=None, error_rates=m[0],
av_freqs=av_freqs)
```

Although EpiLink generates scores, rather than posterior probabilities, we can still use the scores for plots.

```
>>> Epi = record_linkage.series(pairs, 4, Epi_scores)
```

Other options using the results from Binary EM

There are a few options for performing interpolation using the *u* and *m* probabilities returned by Binary EM. The most generally effective of these is due to Winkler (1990).

```
>>> post_probs = record_linkage.Winkler_weighted_post(scores, u, m, p,
c=9/2)
>>> Winkler = record_linkage.series(pairs, 4, post_probs)
```

Other (generally less effective) options can be found in the source code.

Multinomial EM

We can more effectively exploit similarity scores by using more than 2 categories. We simply generate a data array as before, but use several categories. The EM algorithm is used exactly as before.

```
>>> mult_scores = record_linkage.categorized(scores, [-0.01, 0.05, 0.1, 0.2,
0.4, 0.6, 0.8, 0.9, 0.999, 1])
>>> u, m, p, g_m, log_likelihoods = record_linkage.EM(mult_scores)
>>> m
array([[  2.97780860e-02,   5.10778440e-02,   6.83618899e-01,
          1.32614138e-08],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          3.33926270e-01],
       [  0.00000000e+00,   2.50687747e-02,   2.99031216e-02,
          2.34834338e-01],
       [  0.00000000e+00,   6.24293355e-01,   4.74107658e-02,
          1.95273174e-01],
       [  0.00000000e+00,   1.91720686e-01,   0.00000000e+00,
          4.97839860e-02],
       [  6.85279010e-02,   1.40212100e-03,   1.97461216e-02,
          1.16136316e-02],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          1.08200128e-02],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          6.52699581e-04],
       [  9.01694013e-01,   1.06437220e-01,   2.19321092e-01,
```

```
          1.63095875e-01]])
>>> p
0.016415340276378785
>>> mult1 = record_linkage.series(pairs, 4, g_m)
```

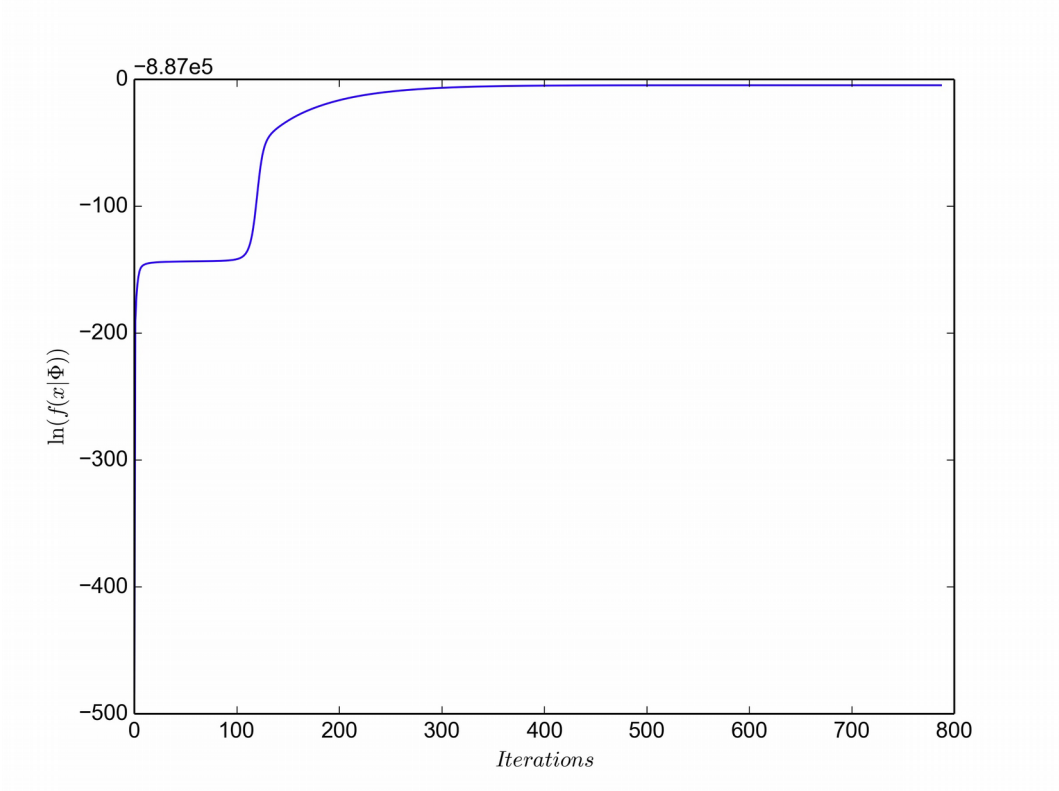

Illustration 3: Log likelihoods for Multinomial EM

Again, the *m* probabilities for the highest similarity score level look unreasonable. The likelihoods plot shows the same 'multi-stage convergence' that we saw in Illustration 1. Let's try using fixed *u* probabilities again.

```
>>> u_probs = record_linkage.u_probs(mult_scores)
>>> u, m, p, g_m, log_likelihoods = record_linkage.EM(mult_scores,
u=u_probs)
>>> m
array([[  2.98938539e-02,   8.28955195e-03,   1.36250872e-02,
          6.13519734e-07],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          1.20539324e-02],
       [  0.00000000e+00,   1.06022657e-02,   7.85930560e-03,
          8.76286624e-03],
       [  0.00000000e+00,   1.49455725e-01,   1.58531033e-02,
          1.28377182e-01],
       [  0.00000000e+00,   8.99477820e-02,   0.00000000e+00,
          1.22929377e-01],
       [  6.94401664e-02,   1.02343302e-02,   3.06361441e-02,
          5.42150588e-02],
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          2.63483114e-02],
```

```
       [  0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
          5.96239510e-03],
       [  9.00665980e-01,   7.31470345e-01,   9.32026360e-01,
          6.41350263e-01]])
>>> p
0.0012016122435814405
>>> mult2 = record_linkage.series(pairs, 4, g_m)
```

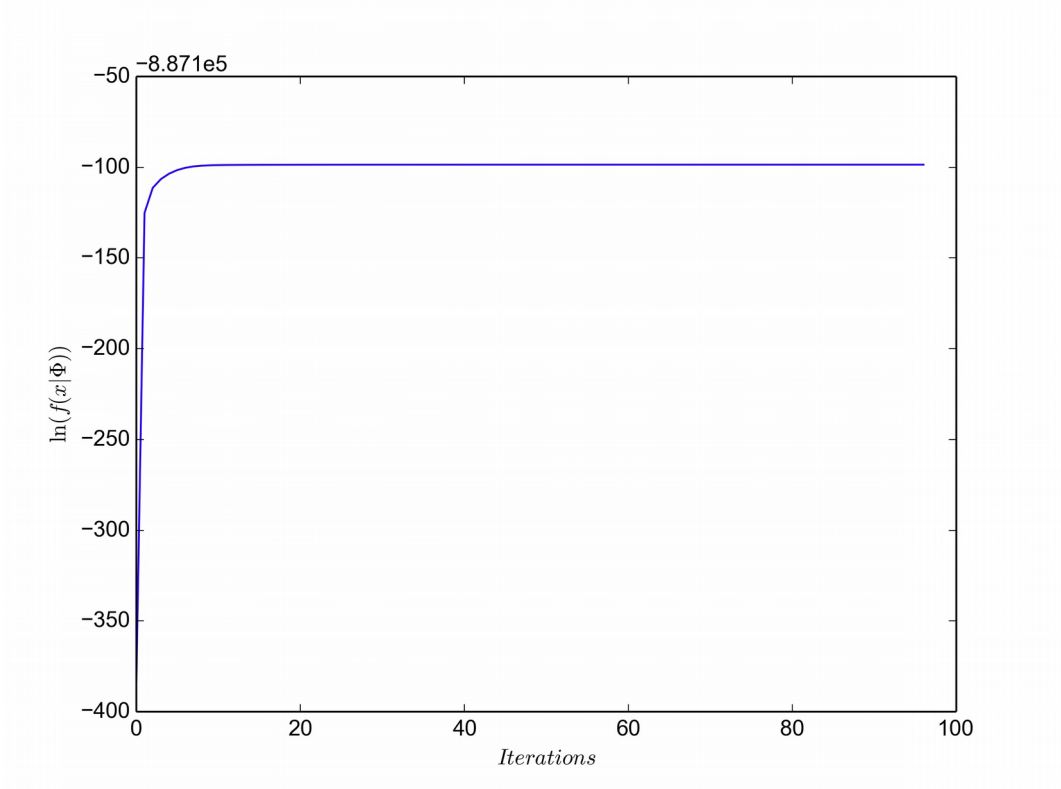

Illustration 4: Log likelihoods for Multinomial EM with fixed u probabilities

Convergence is quicker, smoother and converges to a solution with lower log likelihood.

An alternative way to deal with over-fitting is to place priors on one or more of *u*, *m* and *p*. (We assume prior independence.) Priors can be arrays, or single integers or floats. Here we use a Dirichlet prior for the *m* probabilities with all parameters equal to 5.

```
>>> u, m, p, g_m, log_likelihoods = record_linkage.EM(mult_scores,
m_prior=5)
>>> m
array([[ 0.0363187 ,  0.02610546,  0.0785319 ,  0.03975267],
       [ 0.00906444,  0.00844316,  0.00928524,  0.04367458],
       [ 0.00906444,  0.02401016,  0.03159564,  0.03619444],
       [ 0.00906444,  0.19853186,  0.04042481,  0.12150554],
       [ 0.00906444,  0.11723391,  0.00928524,  0.11211652],
       [ 0.0760655 ,  0.01791594,  0.04460014,  0.05336443],
       [ 0.00906444,  0.00844316,  0.00928524,  0.03329168],
       [ 0.00906444,  0.00844316,  0.00928524,  0.01389584],
       [ 0.83322918,  0.59087318,  0.76770654,  0.54620431]])
>>> p
```

```
0.0015634152366554943
>>> mult3 = record_linkage.series(pairs, 4, g_m)
```

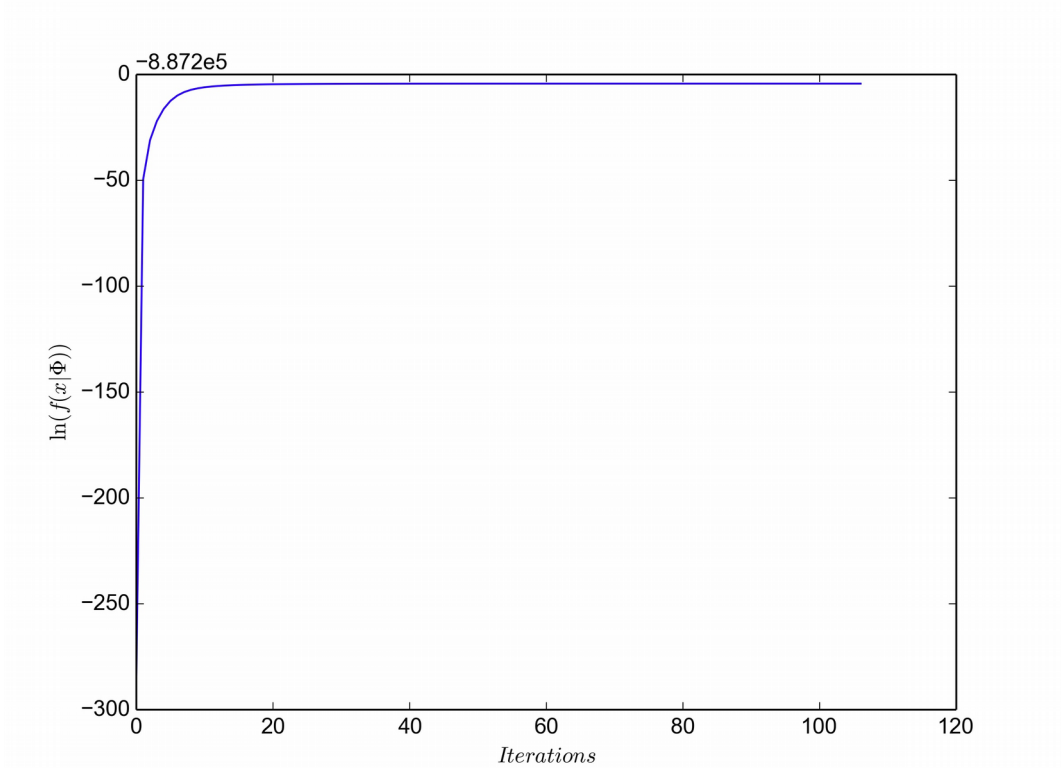

Illustration 5: Log likelihoods for Multinomial EM using MAP estimation

Using fixed *u* probabilities or a prior on the *m* probabilities both result in rapid, smooth convergence to a solution with lower likelihood than when estimating all parameters via maximum likelihood.

We now have three multinomial EM fits. The first suggested a degree of over-fitting. The latter two are harder to distinguish. We could also have tried fixing *u* probabilities as well as specifying a prior for *m*. We could also have specified additional priors or a fixed value for *p* (although in practice we would not generally know the value of *p*).

Assessing performance

We have seen the log likelihood plots and how they can be useful for model diagnostics. When we know the true match statuses we can also plot receiver operating characteristic (ROC) curves or precision recall curves. The latter tend to be more informative. So we can now plot all our data series to compare the record linkage performance of the various approaches.

For any given threshold we will have a number of false positives *fp*, and a number of false negatives *fn*. Similarly we will have number of true positives *tp*, and a number of true negatives *tn*.

$$Precision = \frac{tp}{tp + fp}$$

$$Recall = \frac{tp}{tp + fn}$$

A plot of precision against recall for a large range of thresholds (one threshold for each distinct precision, recall pair) allows the comparison of record linkage approaches. Good approaches will produce curves in the upper right of the plot.

```
>>> record_linkage.precision_recall_curve('precision_recall.jpg', [bin1,
bin2, Epi, Winkler, mult1, mult2, mult3], ['Binary EM', 'Binary EM - fixed
u', 'EpiLink', 'Winkler', 'Multinomial EM', 'Multinomial EM - fixed u',
'Multinomial EM - MAP'])
```

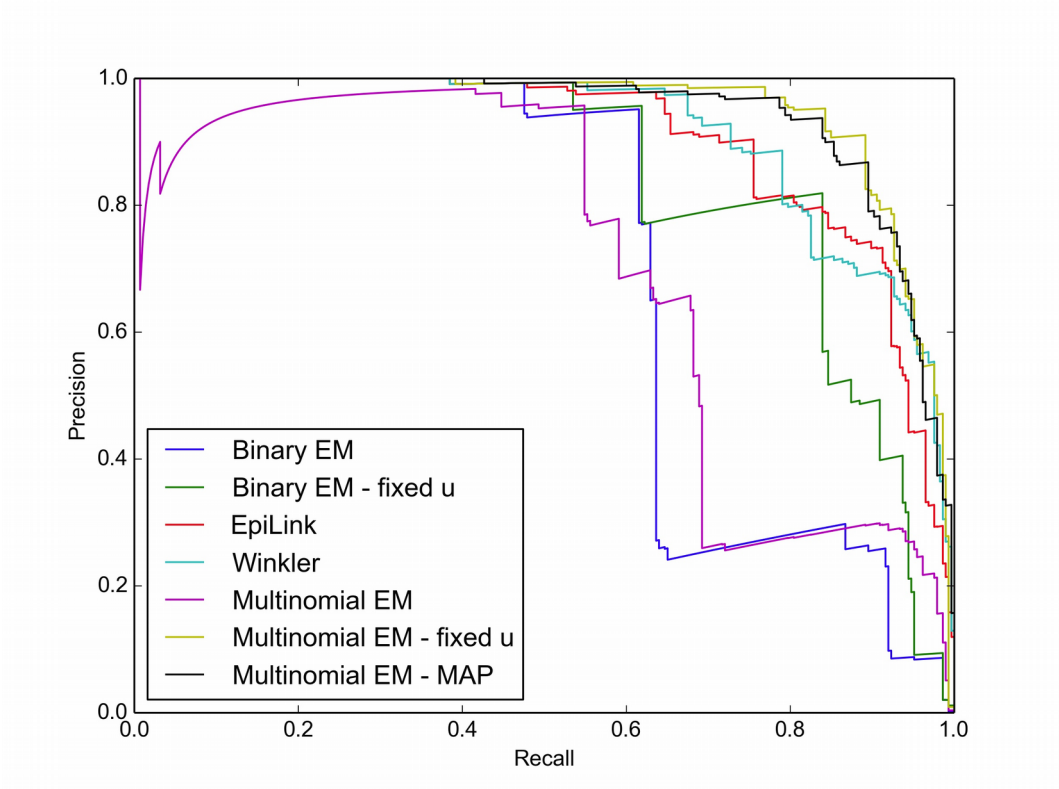

Illustration 6: Precision against recall for all model fits

The traditional Binary EM and the Multinomial EM perform poorly if over-fitting is not dealt with properly. Both can be significantly improved. EpiLink and Winkler both provide improvements over Binary EM. The multinomial EM with fixed *u* probabilities or a prior over the *m* probabilities performs significantly better than all the other approaches.

Note: for the purposes of illustration, samples that result in over-fitting were deliberately chosen. In the many simulations that have been carried out (using the same data and sample sizes) over-fitting was only rarely an issue. Improvements were consistent, and similar to those shown in Illustration 6.

We can also generate a classification table (for our Multinomial MAP fit).

```
>>> matches = record_linkage.matches(pairs, 4)
>>> predictions = record_linkage.predictions(g_m, 0.5)
>>> record_linkage.classification_table(matches, predictions)
array([[  2.60000000e+02,   7.80000000e+01],
       [  2.60000000e+01,   2.79636000e+05]])
```

Here we have 260 true positives, 279636 true negatives, 78 false positives and 26 false negatives for a threshold on the posterior probability of a correct match of 0.5. In theory, this threshold should minimise the total number of misclassified record pairs.

Fellegi-Sunter decision rules

In a separate module, *rules.py*, there is code that implements decision rules (Fellegi and Sunter, 1969). Under a decision rule a record pair will be allocated to one of 3 sets.

A_1 – a set of correct links
A_2 – a set of uncertain links
A_3 – a set of incorrect links

There are two types of error associated with a decision rule,

$$P(A_1|U)=\mu$$

and

$$P(A_3|M)=\lambda \quad .$$

We can generate error rates from thresholds on the posterior probability of a correct match. The returned error rates are ordered (μ,λ) .

```
>>> import rules
>>> rules.error_rates_from_threshold(u, m, p, 0.5, b=None)
(0.00013530660316521032, 0.2493427417847722)
```

So if we use a cut-off of 0.5 on the posterior probability of a correct match, then we should have error rates of about 0.000135 and 0.249. The empirical error rates are actually 78 / (78+279636) = 0.0002788562603230443 and 26 / (26+260) = 0.09090909090909091. So in this case the error rates are not terribly accurate, but perhaps still useful.

We can also generate thresholds from error rates.

```
>>> rules.thresholds_from_error_rates(u, m, p, 0.000135306603165210,
0.24934274178477, b=None)
((0.49691589286495463, 0.99999999998597), (0.50922395107726548,
0.99999999999942846))
```

Note that the error rates supplied as arguments to the function above are slightly lower than those returned by the previous function call. This is to avoid an exception being raised due to the lower threshold being higher than the upper threshold (due to floating point issues). The returned values imply that any record pair with posterior probability less than 0.49691589286495463 should be allocated to A_3, and any record pair with posterior probability equal to 0.49691589286495463 should be allocated to A_3 with probability 0.99999999998597. Similarly, any record pair with posterior probability greater than 0.50922395107726548 should be allocated to A_1, and any record pair with posterior probability equal to 0.50922395107726548 should be allocated to A_1 with probability 0.99999999999942846. There are no record pairs with comparison vectors (vectors of similarity score levels) that produce posterior probabilities between these thresholds. As we reduced the error rates slightly the above function call has returned a rule that will allocate (with very small probability) some record pairs to A_2. Of course, we might deliberately choose error rates to generate such a rule.

We can also generate a useful plot.

```
>>> rules.plot_error_rates('error_rates.jpg', u, m, p, b=None)
```

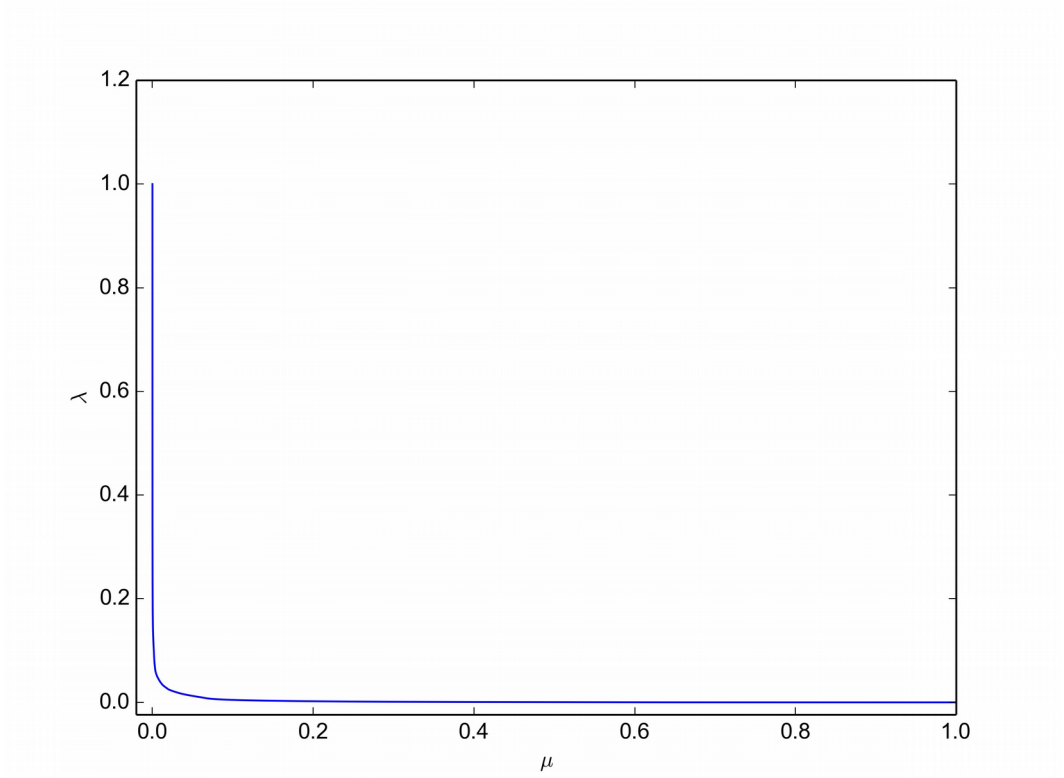

Illustration 7: Error rates for Multinomial MAP fit

In Illustration 7 we can see the relationship between error rates. Any point on the curve represents a rule where A_2 is empty. Points below and to the left of the curve (and with non-negative rates) represent rules with non-empty A_2. For rules with empty A_2 the gradient of the curve is minus the threshold on the likelihood ratio – which can be easily mapped to a threshold on the posterior probability of a correct match.

The optional argument to the previous functions, *b*, allows blocking to be accommodated. It is a dictionary mapping variable indices to similarity level indices (blocking levels). So if we were

considering the impact of blocking on the highest similarity level of the first variable:

```
>>> rules.plot_error_rates('error_rates.jpg', u, m, p, b={0:-1})
```

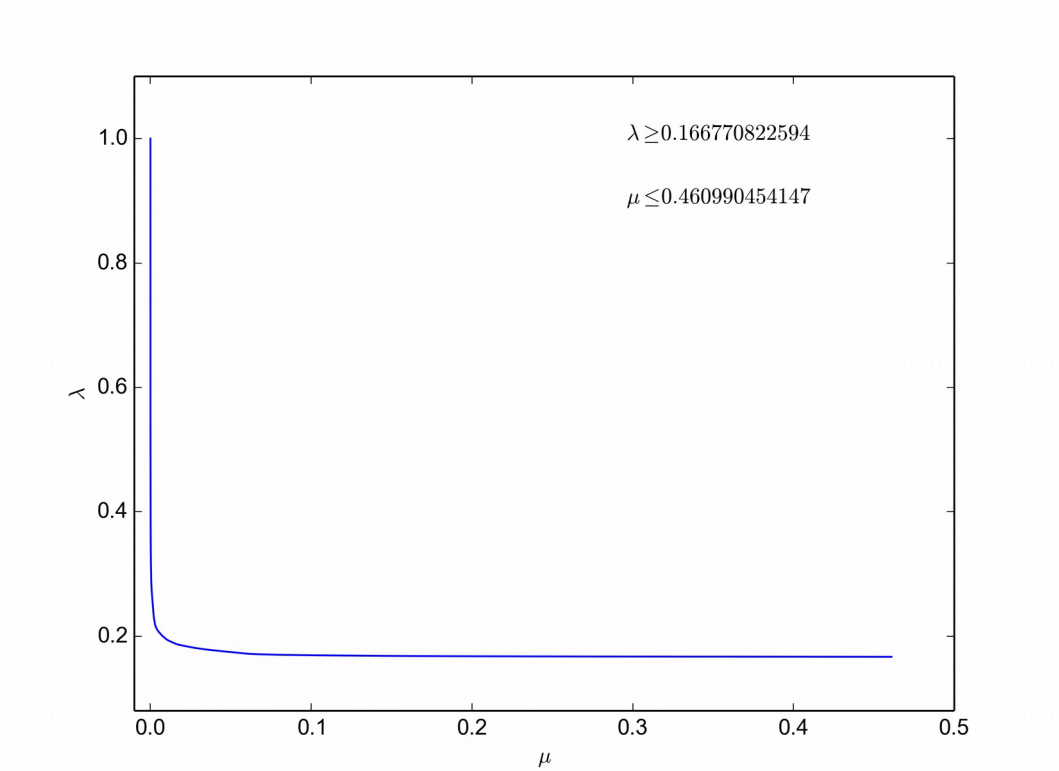

Illustration 8: Error rates for Multinomial MAP fit with blocking

Blocking places constraints on the error rates and these are shown in the plot. Of course, we would normally apply blocking to reduce the computational cost of estimation. But if we were to fit using sub-samples of data, then these plots could be useful for investigating blocking schemes that might be applied to the full samples.

There are a few other functions in *rules.py*. For example, we can generate thresholds that will minimise a cost function,

$$z = C_{MU} P(A_1|U) P(U) + C_{UU} P(A_3|U) P(U) + C_{UM} P(A_3|M) P(M) + C_{MM} P(A_1|M) P(M)$$

where C_{XY} is the cost of allocating a pair to class *X* when the correct classification is class *Y*.

If we assign equal costs to μ and λ and zero costs for correct classification, then we get the threshold that minimises the total number of misclassified record pairs.

```
>>> rules.threshold_from_cost_function(1, 1, C_UU=0, C_MM=0)
0.5
```

Summary

The code contains a few utility functions, and a number of methods for performing record linkage using similarity scores. The most effective method is the Multinomial EM approach. Performance will depend on the suitability of the similarity score and the chosen categorization. As yet there is no functionality for trying to automatically identify a good categorization, but the one used here appears to be reasonably effective for the data used (with various levels of perturbation) and a number of types of score (Jaccard, Dice, Jaro and Jaro-Winkler). There are diminishing returns from adding extra categories, and this increases computational costs. There is an occasional issue with over-fitting using maximum likelihood estimation. Solutions have been presented. It is important to check the plausibility of parameter estimates and the likelihoods plot to identify when over-fitting might have occurred. We won't normally have the luxury of generating precision recall plots.

There are also a number of potentially useful functions relating to the Fellegi-Sunter decision rules.

The code contains more comprehensive documentation, including for functionality that has not been described in this document.

References

Contiero, P., Tittarelli, A., Tagliabue, G., Maghini, A., Fabiano, S., Crosignani, P. and Tessandori, R. (2005) *The EpiLink Record Linkage Software : Presentation and Results of Linkage Test on Cancer Registry Files*. Methods Inf Med, 44, pp.66-71

Fellegi, I.P. and Sunter A.B. (1969) *A theory for record linkage*. JASA Vol. 64, No. 238, pp.1183 1210

Jaro, M.A. (1989). *Advances in record linkage methodology as applied to the 1985 census of Tampa Florida*. Journal of the American Statistical Association 84 (406) pp.414-20

Winkler, W. E. (1990). *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. Proceedings of the Section on Survey Research Methods (American Statistical Association) pp.354-359