Fast Algorithms For The Calculation Of Kendall's Tau

David Christensen

EMB, Link House, Great Shelford, Cambridge, England. CB2 5LT

Summary

Traditional algorithms for the calculation of Kendall's Tau between two datasets of n samples have a calculation time of $O(n^2)$. This paper presents a suite of algorithms with expected calculation time of O(n log n) or better using a combination of sorting and balanced tree data structures. The literature, e.g. (Dwork et al, 2001), has alluded to the existence of O(n log n) algorithms without any analysis: this paper gives an explicit descriptions of such algorithms for general use both for the case with and without duplicate values in the data. Execution times for sample data are reduced from 3.8 hours to around 1-2 seconds for one million data pairs.

Keywords: Kendall's Tau, Algorithm, O(n log n)

1 Introduction

Kendall's Tau is a non-parametric estimator of correlation between two sets of random variables, X and Y, defined as

$$\tau(X,Y) \equiv P\{(X-X^*)(Y-Y^*)>0\} - P\{(X-X^*)(Y-Y^*)<0\} \quad \text{..... (1)}$$

where (X^*,Y^*) is an independent copy of (X,Y). The traditional algorithm (Press, Flannery, Teukolsky, Vetterling, 1993, and other sources) to calculate this considers all pairings of X, Y and classifies each of them as one of:

Concordant (c):	$(X_i - X_j)(Y_i - Y_j) > 0$
Discordant (d):	$(X_i - X_j)(Y_i - Y_j) < 0$
ExtraX (eX)	$X_i \neq X_j, Y_i = Y_j$
ExtraY (eY)	$X_i = X_j, Y_i \neq Y_j$
Spare	$X_i = X_j, Y_i = Y_j$

The value of τ is then calculated from

$$\tau = \frac{c-d}{\sqrt{(c+d+eX)(c+d+eY)}} \quad \text{..... (2)}$$

When each dataset is of size n, the calculation requires the comparison of ${}^nC_2 = n(n-1)/2$ sets of points, and as such is an $O(n^2)$ process. For sample sizes in excess of a few thousand, this rapidly becomes an unattractive proposition.

Recently, however, Kendall's τ has been gaining use with large datasets. Areas of application include analysis of results from different Internet search engines (Dwork, Kumar, Naor, Sivakumar 2001) and in the author's own area of work in the manipulation of stochastic data using elliptical distributions and their copulas, as also addressed by (Lindskog, McNeil, Schmock 2001). When considering stochastic simulations with a million datapoints, the standard $O(n^2)$ algorithm takes hours on even a very fast PC.

The algorithms described in this paper rely on the following pair of observations:

- the value of τ is unaltered if both datasets are reordered using the same set of transpositions; and

- the calculation of τ becomes considerably simpler if one of the datasets is known to be in ascending order.

A further consideration is that many datasets may be known in advance to lack duplicates – for example sets of rankings, or samples from continuous distributions[1]. We consider the simpler case of known uniqueness (Xs unique and, independently, Ys unique) in the data samples first.

2 Algorithm NDTau for no-duplicate datasets

If there are no duplicates, then the pairs divide neatly into concordant and discordant. (2) can then be simplified to (c-d)/(c+d). However since c+d=n(n-1)/2, we can further simplify this, giving:

$$\tau = \frac{4c}{n.(n-1)} - 1 \qquad \text{..... (3)}$$

ND1) Sort the pairs (X_i,Y_i) such that if $i<j$ then $X_i < X_j$.
ND2) Iterate through the pairs $(X_1,Y_1)..(X_n,Y_n)$ in the sorted order. For each Y_i find NumBefore, the number of Ys with $Y<Y_i$. This can be done by inserting the Y_i values into a balanced binary tree and keeping a count of the prior values at each node. For an overview of the use of binary trees in this context see Appendix 1.
ND3) Concordant := Concordant + NumBefore
ND4) Once the iteration is complete, calculate τ from (3)

Since the Xs are sorted, all Y values that are already in our tree must correspond to pairings with X_j less than the current X_i, so those in the tree with $Y_j<Y_i$ are concordant and – because of the uniqueness of the values – the remainder must be discordant. Note also that to avoid duplication (in the same way that the conventional algorithm considers n(n-1)/2 points rather than n^2) we perform are iterations and calculations only between (X_i,Y_i) and (X_j,Y_j) for $j < i$.

[1] Whilst samples from continuous distributions may contain duplicates to the accuracy with which they are represented numerically, the algorithm presented will treat them as logically unequal with (effectively) a random ordering being selected for them. The chances of this occurring in practice on any sensible accuracy of machine representation are sufficiently slight that further analysis of this case has not been performed.

Since steps ND3 and ND4 is performed in constant time for each of the n values of Y_i, the time limiting factor in this algorithm is the sorting within step ND1 and the insertion within step ND2. Generally speaking a standard sort routine (e.g. Quicksort) should perform this step in an expected time of O(n log n) but the implementor should consider the nature of the X_i data as to whether the chosen pivot algorithm might cause performance to degrade into $O(n^2)$. In particular if the data are already sorted (e.g. ranks from a search engine) then the step may not be required, and if the data are already *almost or totally* sorted then a basic quicksort algorithm which does not use a random pivot selection will degrade into $O(n^2)$; if this is a significantly likely event, then either a more sophisticated choice of pivot should be used than the basic algorithm – for example the median of three approach. Alternatively a Heapsort algorithm can be employed which is on average slightly slower than a quicksort, but which remains O(n log n) in the worst case. For a more detailed examination of these possibilities, consult a standard textbook on sorting such as Knuth (1998).

The insertion into the tree can be performed using an AVL Tree (Adel'son-Vel'skii and Landis 1962) which will perform each insertion in O(log n) time. Note that since we only discard the entire tree at one go and do not perform intermediate deletions we do not need to worry about deletion performance of AVL Trees.

Thus the overall performance of the NDTau algorithm is expected to be O(n log n). With a quicksort algorithm this may occasionally degenerate to $O(n^2)$, but in cases where this is a significant risk a heapsort can bring the solution back to a (slightly slower) O(n log n).

3 Algorithm SDTau for some-duplicate datasets

Once we have the possibility of duplicates, the calculations become more complex. We proceed by sorting the data in ascending[2] order of Xs, with ascending values of Ys used where there are multiple pairs with the same X value. When we then re-iterate through these values we are effectively traversing a sparse array of values indexed by X and Y values. At any stage of the process we are only interested in partitioning this array into five parts, where those parts are determined by the relationship between the previous (X,Y) versus the current

[2] The descriptions throughout this document assume ascending sorts are used. Obviously if descending-sorted data are available, the algorithms can be modified accordingly rather than resorting

(X_i,Y_i). For all points with $X<X_i$ we are only interested in the distribution of the Y values into less than (A), equal to (B) or greater than (C) the current Y_i – their X values are irrelevant. When $X=X_i$ we divide the points into those (E) matching the current value and those (D) with Y_i less than the current value. Because we are iterating through a sorted set of data points, the other categories ($X=X_i$ and $Y>Y_i$, and $X>X_i$) will not contain any values.

	$X<X_i$	$X=X_i$	$X>X_i$
$Y<Y_i$	A	D	
$Y=Y_i$	B	E	
$Y>Y_i$	C		

Each new data point added into cell E will then contribute to the parameters for (2) as follows:

The ExtraY count will increase by D
ExtraX will increase by B
Concordant will increase by A
Discordant will increase by C

We can calculate these values by some fairly straightforward tallying as we iterate through the points (variables referred to as ACount..ECount in the algorithm below count the number of prior pairs in each of the boxes above). At each point we work out how the prior distribution of A..E must be modified to take into account the difference between successive pairs:

SD1) Sort the pairs (X_i,Y_i) such that if $i<j$ then $X_i <= X_j$ and if $X_i=X_j$ then $Y_i<=Y_j$.
SD2) Set ECount=0, DCount=0
SD3) Iterate (from here to SD9 inclusive) through the pairs (X_1,Y_1),...,(X_n,Y_n) in the sorted order.
SD4) If $X_i <> X_{i-1}$ then DCount := 0, ECount := 1
Else
if $Y_i := Y_{i-1}$ then
ECount := ECount+1
else
DCount := DCount+ECount; ECount=1
SD5) Insert Y_i into a binary tree, and note NumBefore, the number of values in the tree with $Y<Y_i$ and NumEqual, the number of values in the tree with $Y=Y_i$ (including Y_i).
SD6) ACount := NumBefore – DCount
SD7) BCount := NumEqual – ECount
SD8) CCount := i - (ACount+BCount+DCount+ECount - 1) *[-1 since E includes X_i]*

SD9) ExtraY := ExtraY + DCount; ExtraX := ExtraX + BCount; Concordant := Concordant + ACount; Discordant := Discordant + CCount;

SD10) Once the iteration is complete, calculate τ using equation (2) above.

Once again the two time-consuming steps are the initial sort and the subsequent insertions into the AVL Tree as the additional steps to go from NDTau to SDTau are all performed in constant time for each of the n iterations.

4 MDTau – an algorithm for high numbers of duplicates with unknown values

High numbers of duplicates

As the number of duplicates increases, a change of strategy may be called for. One particular case is when the number of potential values of each of X and Y is known in advance and is small compared to the size of the dataset. If X can take p values and Y can take q values, then the well known (e.g. Press, Flannery, Teukolsky, Vetterling 1993) algorithm can be used whereby a (p x q) counter array is constructed, a single pass made over the data points is made and each point increments the corresponding counter in the p x q array. The calculation of the values required for (2) is then done in one O(pq) pass over the array, so the limiting factor is the initial O(n) bucketing operation.

Moderate numbers of duplicates

Where the (X,Y) data contains large numbers of duplicate values (either jointly, or independently for X and Y), it is intuitive to expect that a combination of the bucketing approach with the sorted tree structure might yield rewards. This section describes such an algorithm, MDTau. It is computationally more complex owing to the data structures required, but is a simple extension of SDTau merely with duplicate data points collapsed into single weighted points.

The algorithm is not to sort the (X,Y) as such, but to insert them into a tree of trees. In the outer tree each node corresponds to an X value and has within it a tree of Y values; the Y value trees have a count of instances of the (X,Y) values corresponding to the X value of the outer tree and the value of the inner tree. This insertion phase should take a time of roughly O(n log (p q)) where n is the number of data points and p and q are the expected number of distinct values of X and Y respectively. We then perform a second pass over the trees taking each X node in (sorted) turn and then each Y node within each. This will generate a sequence of X,Y pairs in the same manner as SDTau, save that each pair has a

weight corresponding to the number of values of the (X,Y) pair. Note that we still need to generate the "global" tree of Y values in order for the calculation to succeed, as we need an amalgamated record of the values so far sorted by Y order.

The calculation then proceeds much as before, except that we are guaranteed that $Y_i <> Y_{i-1}$ every time, and that we need to multiply the amounts we increase our counts by to take account of the number of elements at the current (X_i,Y_i) node.

MD1) Sort the pairs (X_i,Y_i) such that if $i<j$ then $X_i <= X_j$ and if $X_i=X_j$ then $Y_i<=Y_j$.
MD2) Set ECount=0, DCount=0
MD3) Iterate through the pairs $(X_1,Y_1),...,(X_n,Y_n)$ in the sorted order for the steps below down to MD10.
MD4) If $X_i <> X_{i-1}$ then DCount := 0 Else DCount := DCount + ECount
MD5) ECount := CountAtCurrentNode
MD6) Insert Y_i into a binary tree (incrementing the relevant frequency count by CountAtCurrentNode), and note NumBefore, the number of values in the tree with $Y<Y_i$ and NumEqual, the number of values in the tree with $Y=Y_i$ (including those at Y_i).
MD7) ACount := NumBefore – DCount
MD8) BCount := NumEqual – ECount
MD9) CCount := i – (ACount+BCount+DCount)
MD10) ExtraY := ExtraY + DCount*ECount; ExtraX := ExtraX + BCount*ECount; Concordant := Concordant + ACount*ECount; Discordant := Discordant + CCount*ECount;
MD11) Once the iteration is complete, calculate τ using equation (2) above.

5 Timings

Tests were performed on the conventional algorithm, NDTau, SDTau, and MDTau, with varying size data sets (from 4 to 1,000,000 pairs) of two sorts: No-duplicate data, and fixed-number of values dataset (integral values between 0 and 100 regardless of the number of data points). Tests were performed using code written in Delphi 6 (Object Pascal) and timings were performed on an Athlon XP2400+ machine running Windows 2000.

No duplicate data, timings in milliseconds unless otherwise stated:

Number Of Pairs	**Conventional**	**NDTau**	**SDTau**	**MDTau**
250	0.71	0.13	0.14	0.46
1,000	11	0.74	0.75	3.02
30,000	13300	53	54	170
250,000	855600	761	771	2093
1,000,000	**3.8 hours**	**3.86 seconds**	**3.89 seconds**	**11 seconds**

Data sampled from [0,1,...,100], timings in milliseconds unless otherwise stated:

Num. of Pairs	**Conventional**	**SDTau**	**MDTau**
250	0.71	0.13	0.46
1,000	11	0.59	1.28
30,000	13300	28	29
250,000	855600	280	184
1,000,000	**3.8 hours**	**1.7 seconds**	**0.7 seconds**

The above timings are all average timings using random data samples. The conventional algorithm has a run time which is independent of the data values; however the new algorithms have some variability of performance owing to the nature of the quicksort (or heapsort) algorithm chosen. For large non-pathological datasets the variability is quite low, as extreme orderings are required for the quicksort algorithm to break down. The reader is referred to Knuth (1998) for a full treatment of this issue; however with 1,000,000 datapoints the standard deviations (in milliseconds) of run-times for 50 runs of the above random data, using a conventional quicksort algorithm, were:

	NDTau	SDTau	MDTau
No-duplicate data	50	47	91
Data sampled from 0..100	n/a	22	15

6 Conclusion

A number of conclusions can be drawn:

- The conventional algorithm exhibits the expected $O(n^2)$ behaviour. Whilst it is fastest for values of n smaller than about 30, it becomes massively slower than the alternatives for larger n.
- SDTau and NDTau have very similar performance timings, to the extent that it is almost certainly not worth the effort of separately implementing NDTau as it is rarely more than 3% faster than SDTau.
- Where the number of duplicates is not particularly high, MDTau is about $1/3^{rd}$ of the speed of SDTau. When testing with pairs of values in the 0..100 range, MDTau only overtook SDTau when n exceeded about 30000; however for a million pairs of (0..100) points, MDTau is twice as fast as SDTau.

Appendix 1: Use of binary trees

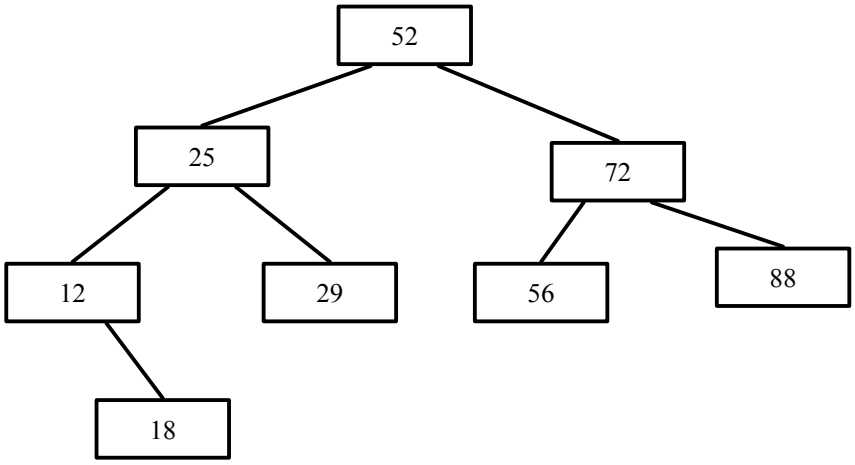
This appendix is provided for those unfamiliar with the use of binary trees to provide O(log n) lookup of data. It presents no new results and can be safely skipped by those familiar with the use of such data structures.

At several locations within the algorithms in the main paper, we wish to find a value of Y_i in a list of those that have already been encountered, and if it is not there, add it in. Furthermore we also wish to maintain a count of the number of values which occur in the sequence (so far) which have $Y < Y_i$.

A simple approach to this would maintain a list or array of values found so far. In the list case the search is O(n) and the insertion is in constant time. In the array case the search can be O(log n) but the insertion is O(n). Either way, the overall search and insert process is O(n). Given that we are performing this n times in the original algorithm, this produces an overall execution time of $O(n^2)$ – precisely what we are trying to avoid.

The standard computer science solution to this problem is to use a binary tree. In this data structure, each item is stored in a "node" which as well as storing the item's value also refers to (up to) two other items, a "left node" and a "right node". These nodes themselves refer to other nodes, so that each node has two "sub trees" under it. The rule for organising such a tree is that the value at a node

is greater than the values at all nodes in its left sub tree, and less than the values in all nodes in its right sub tree. For example:



The efficiency of the data structure comes from the fact that the depth of the tree is at most $1+\log_2(n)$ *provided that the tree is balanced*, i.e. that the routes from root to leaf nodes are all approximately the same length. Ensuring that the tree remains perfectly balanced is not a trivial process, and the normal solution is to use the AVL algorithm described in (G. M. Adel'son-Vel'skii and E. M. Landis 1962) which keeps the tree "near enough balanced" and maintains O(log n) performance for both searching and insertion.

Appendix 2: Code listings

Since SDTau seems to be the best all-round performer, only that algorithm is included here. It also assumes the availability of Quicksort and AVLTree implementations. However, all of the algorithms described and an implementation of AVLTree are available on request from the author at d.christensen@emb.co.uk.

```
function SDTau(var X,Y : array of ElementType) : Real;
var
  i,n : Integer;
  Tree : AVLTree;
  Concordant,Discordant,ExtraX, ExtraY : Integer;
  ACount,BCount,CCount,DCount,ECount : Integer;
  NumBefore : Integer;  // Value returned by FindOrInsert
```

```
  NumEqual : Integer;
  PreviousX, PreviousY : ElementType;
  Tmp, Tmp2 : Real;                          // Used to avoid integer overflows.
begin
  n := Length(X);
  QSort(X,Y,0,n-1);
  Tree := AVLTree.Create;

  // The implementation of AVLTree takes advantage of the fact that we know
  // how many nodes will be inserted to optimise its memory management
  Tree.Initialise(n);

  Concordant:=0;
  Discordant:=0;
  ExtraX:=0;
  ExtraY:=0;
  DCount:=0;
  ECount:=0;
  PreviousX := X[0]-1;  // Ensure different!
  PreviousY := Y[0]-1;  // Ensure different first time through loop

  for i:=0 to n-1 do
  begin
    if X[i]<>PreviousX then
    begin
      DCount := 0;
      Ecount := 1;
    end
    else
    begin
      if Y[i]=PreviousY then
        Inc(ECount)
      else
      begin
        DCount := DCount+ECount;
        ECount := 1;
      end;
    end;
    NumEqual := Tree.FindOrInsert(Y[i],NumBefore).EqualCount ;
    ACount := NumBefore - DCount;
    BCount := NumEqual - ECount;
    CCount := i - (ACount+BCount+DCount+ECount-1);
    ExtraY := ExtraY + DCount;
    ExtraX := ExtraX + BCount;
    Concordant := Concordant + ACount;
    Discordant := Discordant + CCount;
    PreviousX := X[i];
    PreviousY := Y[i];
  end;

  Tmp := Concordant+Discordant+ExtraX;
```

```
  Tmp2 := Concordant+Discordant+ExtraY;

  Result := (Concordant-Discordant)/sqrt(Tmp*Tmp2);
  FreeAndNil(Tree);

end;
```

References

Adel'son-Vel'skii, G. M. and Landis, E. M. (1962). An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259—1262.

Dwork, C, Kumar, R, Naor, M, Sivakumar, D, (2001): Rank Aggregation Revisited. *Proc. 10th International World Wide Web Conference,* pages 613—622.

Knuth, D.E., (1998), *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition.

Lindskog, F., McNeil, A., Schmock, U. (2001): Kendall's τ for Elliptical Distributions. *Working paper from* *http://www.math.ethz.ch/~mcneil/pub_list.html*

Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T. (1993) *Numerical Recipes,* Cambridge University Press.