A Toolkit for Matching Maximum Score Estimation and Point and Set Identified Subsampling Inference*

Jeremy T. Fox
University of Michigan and NBER†

David Santiago
University of Chicago (former)

February 2015

Contents

*Thanks to Theodore Chronis, Denisa Mindruta, Joao Montalvao, and Azeem Shaikh for comments.
†Jeremy T. Fox, University of Michigan, Ann Arbor, MI 48109, *e-mail:* jeremyfox@gmail.com.

1 Introduction

Certain economic situations are well-described by matching games, where a finite number of agents are rivals to match with a limited number of goods or other agents. For example, Bajari and Fox (2005) use a matching model to analyze an FCC spectrum auction (where companies are rivals to assemble complementary packages of licenses), and Akkus (2006) uses a matching model to study the merging decisions of banks. Fox (2006) introduces a maximum score estimator for the parameters of the production function for match output. The estimator is semiparametric as it does not require the full specification of the stochastic structure.

This document serves two purposes. The primary purpose is to document the Match Estimation toolkit, a Mathematica package that eases the use of the pairwise maximum score estimator. Empirical researchers can use the routines in the package for estimation and inference in a variety of models. The package supports a number of variations on the main estimator. A secondary goal of this paper is to describe in detail the implementation of the toolkit, so that other researchers can more easily implement the estimator in different languages. Several key insights that make the estimation process significantly faster are discussed, and should be useful in many different computing environments.

2 Installation

The Match Estimation toolkit consists of software written in Mathematica. Preparing Mathematica to use the software is simply a matter of either putting the software in a place where Mathematica will automatically look for it, or telling Mathematica its location.

2.1 Places Mathematica Checks

The Mathematica documentation gives a complete list of the locations the runtime system checks when Mathematica loads, and the correct one to pick will depend on the specific directories you have access to, and which users should have access to the software.

One location that is usually accessible and convenient is the user's add-on directory. Installing the Match Estimation toolkit involves simply copying the files into the appropriate subdirectory of the user add-on directory.

To determine the user add-on directory, first evaluate the expression

```
$UserAddOnsDirectory
```

Find that location on your filesystem, enter the `Applications` subdirectory, and place the files into a directory called `MatchEstimation`. Thus, the file `MatchEstimation.m` will be in the directory

```
$UserAddOnsDirectory/Applications/MatchEstimation
```

2.2 Telling Mathematica Where To Look

The other option is to tell Mathematica where to look for the files before you load them. If the file `MatchEstimation.m` is not in a standard directory, it must be placed into the current directory before use. An easy way to set the current directory is using the `SetDirectory` command. If the files are in the same directory as the notebook using them, a convenient line to evaluate at the beginning of a notebook is

```
SetDirectory[ToFileName[Extract["FileName" /.
NotebookInformation[EvaluationNotebook[]], {1}, FrontEnd`FileName]]];
```

Following the execution of this line, the toolkit can be loaded with the statement

```
Needs["MatchEstimation`"]
```

3 Review of Maximum Score Estimation

The maximum score estimator introduced in Manski (1975) is an extremum estimator similar to the maximum likelihood estimator. While the maximum likelihood estimator is the parameter vector maximizing the likelihood function, the maximum score estimator is the parameter vector maximizing the score, the number of observations correctly predicted by the discrete choice model. The objective function for the maximum score estimator is generally a summation over an indicator function $1[\cdot]$ that evaluates to one when its argument is true, and zero otherwise:

$$Q_n(\beta) = \sum 1[\cdot].$$

The maximum score estimator is consistent under weak conditions on the distribution of the error term. Specifically, the choice probabilities for a given agent must be rank ordered by the deterministic part of the choice payoff. The maximum score estimator is semiparametric, since it does not require the full parametric specification of the stochastic structure of the model, aside from the rank order property. The semiparametric nature of the estimator, combined with its easily computed objective function, makes maximum score an attractive choice for computationally difficult models.

The maximum score estimator has some drawbacks. The estimator converges at the rate $\sqrt[3]{n}$, instead of the more common $\sqrt{n}$, and its limiting distribution is too complex to use for inference (Kim and Pollard, 1990). Abrevaya and Huang (2005) show that the bootstrap is also inconsistent for maximum score. To work around these difficulties, there are two ways to do inference on a maximum score estimator.

The first, proposed by Horowitz (1992), is called smoothed maximum score (see Section 6.3). Smoothing the objective function results in an estimator that converges almost as fast as $\sqrt{n}$. Horowitz also shows that it is asymptotically normal, with a covariance matrix that can be estimated for use in inference.

The second option is to use the subsampling procedure, which is described in detail in Politis, Romano and Wolf (1999). Subsampling is a resampling procedure similar to the bootstrap. The bootstrap procedure generates an estimate of the asymptotic sampling distribution by generating draws from a distribution similar to the true sampling distribution: the sampling distribution of the estimator when samples of size n are drawn from the dataset with replacement. In contrast, the subsampling procedure estimates the asymptotic sampling distribution by sampling the estimator for samples of size $b \ll n$, but which are drawn from the dataset without replacement. Therefore, the subsampled "datasets" are valid samples of size b from the original data generating process, while the bootstrap samples are not. Because of this, subsampling does not require the smoothness conditions required by the bootstrap for consistency.

Suppose that the maximum score estimator $\hat{\beta}_n(X_1,...,X_n)$ is a point-identified estimator of some parameter β. Let J_n be the sampling distribution of $\tau_n(\hat{\beta}_n - \beta)$ based on sample of size n, where τ_n is a sequence such that the distribution of the statistic is not degenerate, and assume that J_n converges to a limiting distribution J as $n \to \infty$.

The empirical distribution of the estimator can be approximated as

$$L_n(x) = \frac{1}{B}\sum_{i=1}^{B} 1[\tau_b(\hat{\beta}_b(X_{s_{i,1}},...,X_{s_{i,b}}) - \hat{\beta}_n) \leq x],$$

where s_i is a subset of $\{1,...,n\}$ of size $b \ll n$, and there are B such subsets. Politis, Romano and Wolf show that if $b \to \infty$ and $b/n \to 0$ as $n \to \infty$, then $L_n(x) \to J(x)$ in probability. This approximation to the limiting distribution of the estimator allows the calculation of confidence intervals.

In certain settings, the maximum score estimator is not point-identified. For example, in the matching maximum score estimator introduced in Fox (2006) and discussed below, certain assumptions lead to an estimator that is only set-identified. In these situations, inference becomes more difficult, as a confidence region must be generated for an entire set. Chernozhukov, Hong and Tamer (2005) and Shaikh (2005) are two recent papers that provide algorithms for estimating a confidence region for a set-identified estimator. The Match Estimation toolkit uses the latter algorithm to estimate set-identified confidence regions.

4 Data Model

The Match Estimation toolkit can deal with many types of data and many functional forms for the models it estimates. In order to do this it makes certain assumptions about the layout of the data that it will work with. The following sections document the data formats the toolkit uses for either input or output.

4.1 The Matching Problem

In a matching problem, the basic unit of analysis is a "market," which consists of two distinct groups of people or entities, called the "upstream" and "downstream" sides of the market. People can conceivably match with one person in the other group, a group of people in the other group, or agents can form a coalition consisting of people in both groups. These three cases are called one-to-one, one-to-many, and many-to-many matching.

The econometrician observes the matches that agents in a market have formed, as well as certain characteristics of each of the agents in the market. The total payoff for a given matched coalition can be written as a function of the characteristics of the agents in the coalition. The goal of the econometrician is to estimate the unknown parameters of the match payoff function from the matches observed.

The econometrician must make a decision about how the data are generated. The econometrician may have data containing all of the agents in a number of independent markets sampled from a much larger number of such markets. In this case, the logic of the asymptotics for consistency is that the econometrician is expected to observe more and more such markets. Fox (2006) shows the consistency of the matching maximum score estimator in this case.

It is also possible that the data is a sample of some of the coalitions in a single large market. The logic of the asymptotics in this case is that the econometrician expects to observe more and more coalitions from this large market. There has been some previous work on models of this sort. Han (1987) introduced an estimator for single-agent ordered choice in a similar setting, which he called the maximum rank correlation estimator. Sherman (1993) showed that this estimator is $\sqrt{H}$-consistent and asymptotically normal, where H is the number of observed matches. Fox (2006) shows that the matching maximum score estimator is consistent in the single large market setting as well.

4.2 Data

The econometrician is assumed to have data on the agents and their matches in a number of markets. The markets are assumed to be numbered, though the ordering is not important. Each participant on each side of a market is also numbered (again, the ordering is not important). Therefore, each agent in the dataset can be uniquely identified by a market number, a side of the market, and its index for that side of the market.

A match can be specified by a market number and two lists of numbers indicating the agents on each side of the market for a given coalition. For one-to-one matching, this will mean that a match is identified by a tuple $\{m, i, j\}$, indicating the market number, and the indexes of the two agents on each side of the market (upstream then downstream). More generally, a many-to-many coalition can be written as $\{m, \{i_1, ...\}, \{j_1, ...\}\}$. In the special case that a match payoff in a many-to-many model is additively separable across the individual one-to-one matches, it may be more convenient to consider the coalition as a list of $\{m, i, j\}$ tuples whose individual payoffs are then added together.

4.3 Objective Function

Estimation is computationally the maximization of the objective function, so the objective function is the key input into the routines in the Match Estimation package. The code in the package can do a lot of the work involved in estimating matching games, but it cannot know the objective function required for your task. Therefore, the organizing idea of the estimation code is a challenge-response mechanism: when you call a routine, you must supply an objective function taking a parameter vector as an argument. Since this mechanism provides the objective function value for any value of the parameter vector that Match Estimation requires, the code is able to generate an estimate.

The maximum score objective function is the score function, the number of data points predicted correctly by the model. In the simplest case of one-to-one matching, the score function is evaluated at a parameter value β by considering all pairs of matches in each market, and considering the hypothetical match payoff if two upstream members of each match switched downstream members. If the total payoff of the match as observed is higher at β than the payoff function when they switch partners, then the score is increased by one, since the partners have matched themselves as this value of β would predict. This is expressed as

$$Q(\beta) = \sum_{m \in M} \sum_{i \in U_m} \sum_{j \in U_m \setminus i} 1[f_\beta(m,i,\mu_m(i)) + f_\beta(m,j,\mu_m(j)) > f_\beta(m,i,\mu_m(j)) + f_\beta(m,j,\mu_m(i))]$$

where f_β is the payoff function of the match evaluated at the parameter vector β, M is the set of markets observed, U_m is the set of upstream coalition members in market m, and $U_m \setminus i$ is the set of upstream coalition members in market m excluding member i. The function $\mu_m(\cdot)$ is used to reference the downstream partner of the upstream agent in market m. In the Match Estimation toolkit, the function f_β is not limited to any specific functional form.

In more general matching situations, such as many-to-many matching, the objective function is slightly more complex. Since there can be an arbitrary number of members on each side of a match, the total match payoff must involve a sum over the coalition members in each inequality being considered. This is most easily expressed by summing over the set of inequalities being considered.

$$Q(\beta) = \sum_{m \in M} \sum_{\{C^{LHS}, C^{RHS}\} \in I_m} 1[\sum_{\vec{a} \in C^{LHS}} f_\beta(\vec{x_a}) > \sum_{\vec{a} \in C^{RHS}} f_\beta(\vec{x_a})]$$

In this notation, M is a set of markets and I_m is a set of inequalities to be compared. Each element of I_m is a pair of sets of coalitions $\{C^{LHS}, C^{RHS}\}$, one of which is observed and one of which is a hypothetical variation of the other. The total match payoff on each side of the inequality sign is summed over the total payoffs of all of the coalitions being considered, and the payoff functions are written as a function of the covariates $\vec{x_a}$ of each coalition in the set of coalitions being considered. Note that the simpler one-to-one score function above can be written in this form by considering I_m to be the set of all pairs of upstream agents joined with their observed or hypothetical match partners.

The objective function needs to be normalized for any purpose that requires it to have an asymptotic limit. Since there are two ways to generate more data, and thus two different limiting objective functions, there are two different normalizations. If data will be added by observing more markets, then the objective function should be divided by M, the number of markets. On the other hand, if there is one big, static market, and more and more matches from this market will be observed, then the objective function should be normalized by dividing by $H(H-1)$, where H is the number of coalitions observed; note that $H(H-1)/2$ is the number of comparisons that will be performed in the nested sums.

Functions in Match Estimation that require an approximation to the asymptotic objective function will perform the normalization internally.

5 Using the Toolkit

The basic process for using the toolkit for estimation is as follows:

1. Write an objective function that is specified up to unknown parameters
2. Create the data array and data map for the objective function
3. Call the estimation routines with an objective function and data matrix

5.1 Write an Objective Function

In order to enable the full functionality of the Match Estimation toolkit, the package requires not only the value of the objective function for given parameter values, but also the ability to evaluate the objective function on smaller, hypothetical datasets that it generates. For example, to calculate confidence regions the toolkit must generate smaller datasets by sampling from the full dataset, and then calculate an estimate from the smaller datasets. Therefore, the toolkit requires that the data used in estimation be given in a specific form, so that it can correctly generate these smaller datasets.

As described in section 4.3, the objective function is the sum of the indicator function evaluated over a set of inequalities. Removing agents from the sample requires knowing which inequalities correspond to comparisons involving the agents being removed. The approach taken in this software is to require the objective function to accept two arguments: the hypothesized parameter values and a "data array" (described in detail in the following section). Thus, an objective function for the purposes of maximum score estimation is a function that takes a data array and a set of parameter values as arguments and returns an integer representing the number of inequalities in the objective function that were correctly predicted. For example, such a function could be called as

```
objective[dataArray_, b_, g_]
```

The first argument is always the data array parameter, after which any number of arguments can follow. The objective function must not assume the data array is of any specific size.

For example, the following is a simple implementation of the maximum score objective function with one parameter to be estimated:

```
objective[data_,b_] := Module[{total, i},
    total = 0;
    For[i=1, i <=  Length[data], i = i+1,
        total += If[data[[1,i]] + b*data[[2,i]] > 0,1,0];
    ] ;
    total
]
```

The function loops over the data array and adds one to the total score for each inequality that is correctly predicted according to the data. A more efficient implementation of the objective function is described in section 6.2.

5.2 Create a Data Array and Data Map

5.2.1 The Data Array

All of the data used to evaluate the objective function must be packed into a list structure. Since multiple numerical values are required to evaluate a single inequality, a natural form for the i^{th} element of the data array is itself a list, making the data array a list of lists, which can be thought of as a matrix in Mathematica. All data used in the evaluation of inequality i must be in the i^{th} column of this matrix. In the course of estimating confidence regions, the toolkit will remove columns from the matrix and call the objective function with these smaller data arrays. This is why the objective function must not assume the data array is a certain size. Writing an objective function that does not meet the requirement that all data for a given inequality come from the corresponding column of the data array will result in nonsense estimates.

It may be easier to think of the data array as a list of vector-valued variables. If there are four variables that are used in each inequality, say `var1` through `var4`, then these four variables would be vectors containing the data value for each inequality. The data array would be the list

```
{var1, var2, var3, var4}
```

As an example, suppose that the payoff function for a given match has one parameter β, and the payoff function has the form

$$f_\beta(i,j) = x_i y_j + \beta w_i z_j$$

A single inequality of the objective function, let's say the first inequality, will consider the total payoff of two matches against the payoff if they switch their partners:

$$1[f_\beta(i,j) + f_\beta(q,u) > f_\beta(i,u) + f_\beta(q,j)]$$

which is equivalent to

$$1[x_i y_j + \beta w_i z_j + x_q y_u + \beta w_q z_u > x_i y_u + \beta w_i z_u + x_q y_j + \beta w_q z_j]$$

A simple implementation of this inequality would require 8 values:

$$x_i y_j, w_i z_j, x_q y_u, w_q z_u, x_i y_u, w_i z_u, x_q y_j, w_q z_j$$

These 8 quantities would be the elements of the list going into the first element of the data array. Thus, we might have a data array whose transpose looks like:

```
{{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0},
  {... data for inequality 2 ...},
      ...}
```

A later section will suggest a better way to implement this objective function and data array.

5.2.2 The Data Map

An important related issue is how to communicate the contents of each element of the data array to the Match Estimation toolkit. In a typical situation, the toolkit might need to evaluate a subset of the data where agent q (continuing the above example) has been removed. This means that any inequality that references agent q must not be evaluated. To implement this, the toolkit will remove any elements from the data array that reference agent q. The problem is then how to tell the Match Estimation routines which elements of the data array correspond to which agents. This is achieved with a secondary data structure, the data map.

The data map is also a list of lists. The elements of the top-level lists of the two data structures must match, so that the first list in the data map describes the elements of the first list in the data array. The contents of the data map are left entirely to the user, and it can be anything that the user might want the software to select subsamples according to. For example, if the data contains information on multiple separate markets, the user might want the subsampling routine to generate smaller datasets by selecting a sample of markets from the full dataset. In this case, each row of the data map would contain the market number of the agents in the inequality for that row of the data array. Similarly, if the subsampling is to be done agent-by-agent, the data map would contain the identifier numbers of the agents involved in each inequality.

To continue the example from before, suppose the data array's first five elements are data for inequalities between elements in the first nest and asymptotics are in the number of nests observed. Then the data map would take the form

```
{{1}, {1}, {1}, {1}, {1},....}
```

On the other hand, if the asymptotics are in the number of coalitions in one big market, then the data map may take the form

```
{{1,2},{1,3},{1,4},...}
```

In this case, the first inequality is between upstream agents 1 and 2, the second is between upstream agents 1 and 3, and so forth.

5.2.3 A Marriage Example

To illustrate the creation of the data array from a more standard data format, suppose that a dataset contains a collection of 5 men and 5 women. For each person, x_i is person i's years of schooling and y_i is person i's yearly income. The marriage production function for man i and woman j is

$$u(i,j) = x_i x_j + \beta y_i y_j$$

Suppose that in Mathematica, this dataset is represented by three arrays. The first, `x[[i]]`, contains the years of schooling of person i, and the second, `y[[i]]`, contains the yearly income of person i. A third matrix `mate[[i]]` contains the index of person i's partner for those indexes such that i is male. Therefore, `x[[mate[[i]]]]` contains man i's wife's years of schooling. Assume that the men are indexed by $i = \{1,2,3,4,5\}$.

In order to use the toolkit, this dataset must be converted into a data array. The following blocks of code illustrate the intuition behind this process on a simple example.

```
ineqL = {};
Do[
  Do[
    ineqL = Append[ineqL, {i, j}];
  ,{j,i+1,5}],
{i,1,5}];
```

The first block of code creates a list of all possible pairs of men. These pairs of men will hypothetically exchange partners for the evaluation of the maximum score objective function.

```
ob1=Table[x[[ineqL[[ineq, 1]] ]]*x[[mate[[ineqL[[ineq, 1]] ]] ]]
     + x[[ineqL[[ineq, 2]] ]]*x[[mate[[ineqL[[ineq, 2]] ]] ]],
     {ineq, 1, Length[ineqL]}];
ob2=Table[y[[ineqL[[ineq, 1]] ]]*y[[mate[[ineqL[[ineq, 1]] ]] ]]
     + y[[ineqL[[ineq, 2]] ]]*y[[mate[[ineqL[[ineq, 2]] ]] ]],
     {ineq, 1, Length[ineqL]}];
cf1=Table[x[[ineqL[[ineq, 1]] ]]*x[[mate[[ineqL[[ineq, 2]] ]] ]]
     + x[[ineqL[[ineq, 2]] ]]*x[[mate[[ineqL[[ineq, 1]] ]] ]],
     {ineq, 1, Length[ineqL]}];
cf2=Table[y[[ineqL[[ineq, 1]] ]]*y[[mate[[ineqL[[ineq, 2]] ]] ]]
     + y[[ineqL[[ineq, 2]] ]]*y[[mate[[ineqL[[ineq, 1]] ]] ]],
     {ineq, 1, Length[ineqL]}];
dataArray = {ob1, ob2, cf1, cf2};
```

These lines create the actual data array. The data array will have four elements per inequality; `ob1` and `ob2` correspond to the covariates $x_i x_{mate(i)}$ and $y_i y_{mate(i)}$, which are the covariates of the matches observed in the data, while `cf1` and `cf2` are the counterfactual covariates created by switching the mates of men i and j. The code

implements this by looping through the inequality list created in the previous code block and creating these quantities for each inequality.

A simple objective function that takes this data array as an argument and calculates the maximum score objective function based on the match utility function above can be written as

```
objective[data_,b_] := Module[{total, i},
    total = 0;
    For[i=1, i <=  Length[ineqL], i = i+1,
        total += If[data[[1,i]] + b*data[[2,i]]
           > data[[3,i]] + b*data[[4,i]],1,0];
    ] ;
    total
]
```

For each inequality, this function adds one to the score only if the match production function at parameter value `b` is greater for the observed match than at the counterfactual match. Note that the iteration is over the number of inequalities. Section 6.2 explains how this objective function can be written to evaluate much faster, resulting in significant speed gains for estimation and inference. The appendix describes routines, such as `allPairsInequalityList`, and associated data structures that can be used to automate the generation of inequality lists for more general datasets that include observations on multiple markets. Code to convert the output of these routines to a data array is only slightly more complex than the code above.

5.3 Call the Estimation Routine

The main estimation routine is `pairwiseMSE`. It takes three arguments: an objective function, a data array, and a list of symbols naming the unknown parameters. The result of the routine will be a list with two elements, the first being the maximum value of the objective function, the other a list of replacement rules, each giving the estimate for the parameters in the last argument to `pairwiseMSE`. A typical usage might look like:

```
ans = pairwiseMSE[objective, dataArray, {b, g}]
```

where ans was equal to

```
{12167., {b -> 4.93183, g -> 2.80412}}
```

5.4 Generate Confidence Regions

Fox (2006) discusses the conditions necessary for the pairwise maximum score estimator to be point-identified. To achieve point-identification, the production function must have as a parameter some characteristic that has continuous support over $\mathbb{R}$. In the absence of such a characteristic, the estimator will be set-identified. The Match Estimation toolkit provides three functions for generating confidence regions, one that assumes a point-identified estimator, and two that assume a set-identified estimator. The

set-identified functions can be used on point-identified models, but the point-identified estimator takes much less time to run and can produce a less conservative estimate.

The routine to generate a confidence region under point identification is called `pointIdentifiedCR`. This routine takes seven arguments. The first argument specifies the size of each subsample to be generated and the second argument specifies the number of subsamples to use in constructing the approximation to the estimator's distribution. The third argument is a list containing the point estimate from `pairwiseMSE`, the fourth argument is the objective function, and the fifth argument is a list containing the names of the variables being estimated. The final two arguments are the data map and data array described in the preceding section. For example,

```
pointIdentifiedCR[10, 200, estimate, q, {b}, dataMap, dataArray,
     asymptotics->coalitions]
```

In this case, the routine will calculate 200 subsamples, each with 10 elements. The objective function is q, and the name of the variable being estimated is b. By default,`pointIdentifiedCR` produces asymmetric confidence regions. To produce symmetric confidence regions give `pointIdentifiedCR` the option `symmetric->True`.

As explained in section 4.1, there are two types of asymptotics available: asymptotics as the number of nests increases, and asymptotics as the number of coalitions observed in the market increases. For calculating the confidence region with nest asymptotics, the option `asymptotics->nests` should be passed to `pointIdentifiedCR`; otherwise the option `asymptotics->coalitions` should be passed. This means that the meaning of the first parameter depends on the type of asymptotics: if the asymptotics are in the number of nests, then the first argument is the number of nests in a subsample, and if the asymptotics are in the number of coalitions observed then the first argument is the number of coalitions in a subsample. If the `asymptotics` option is not given, the default behavior is nest asymptotics. It goes without saying that the data in the data map should correspond to the correct type of asymptotics for your purposes; if you are using nest asymptotics, you should put the nest number of each inequality in the corresponding slot of the data map.

The routine to generate a set-identified confidence region is called `setIdentifiedCR`. This routine takes the exact same arguments and options as `pointIdentifiedCR`, except that it cannot generate symmetric confidence regions (if the `symmetric` option is given, it is ignored).

A third routine, `sampleSetIdentifiedCR`, provides an alternative way to get information about the set-identified confidence region. Instead of calculating an axis-aligned bounding box for the confidence region, `sampleSetIdentifiedCR` generates random points in a neighborhood of the estimate and returns only those that are found to be inside the confidence region. The number and variance of the trial points are both user-configurable. The arguments to the function are similar to those of the other two routines:

```
sampleSetIdentifiedCR[10, 200, 3000, estimate, q, {b}, dataMap,
dataArray, asymptotics->nests, samplingVariance->10]
```

The first two arguments are the same as for `setIdentifiedCR`. The third argument is the number of trial points to attempt. The rest of the arguments are the same as `setIdentifiedCR`, except that there is a new parameter, `samplingVariance`. Care should be taken when using this routine to ensure that the `samplingVariance` parameter is appropriate to the scale of the estimates.

Because subsampling is a resampling procedure, there is no guarantee that the estimated confidence regions (which are generated from subsets of the input data) contain the point estimates, from the full sample.

The technique used by `pointIdentifiedCR` is subsampling, as described in Politis et al. (1999). The algorithm for generating set-identified confidence regions is the one described in Shaikh (2005). Since both routines for set-identified inference take so long to run, they output the intermediate steps of the optimization routine as they run.

6 Advanced Topics

6.1 Controlling the Maximization Parameters

Optimization of the objective function is performed by Mathematica's `NMaximize` function. By default, the method used is Differential Evolution. Differential Evolution is an excellent method for performing global maximization on a non-smooth objective function, which is the main task in maximum score estimation. Since the Differential Evolution algorithm is stochastic, there can be situations where it is desirable to manually control the algorithm's tuning parameters by setting the options used in the call to `NMaximize`. This can be done by setting the `nMaximizeOptions` option in the call to `pairwiseMSE` or any variant, as well as any of the subsampling routines.

The `nMaximizeOptions` rule should be a rule that evaluates to a list of rules that will be passed directly into a call to `NMaximize`. For example, the default value of the option is:

```
nMaximizeOptions->{Method->{"DifferentialEvolution"}}
```

Note that `Method` is an option that `NMaximize` accepts, and that the option must take the form of a list of replacement rules. It is highly recommended that Differential Evolution be used for maximum score estimation, but the DE algorithm itself has many tuning parameters that can be changed in the options to the `Method` option. For example, suppose that we want to set the number of initial points (one of the parameters of the DE algorithm) to a specific value, and we want to also set the maximum number of iterations (one of the parameters to `NMinimize` itself). We can use the following call to `pairwiseMSE`:

```
pairwiseMSE[q,dataArr, {b,g}, nMaximizeOptions->{Method->
{"DifferentialEvolution", "InitialPoints"->50},
MaxIterations->200}]
```

Note how options for the DE algorithm go in the list of options for the `Method` parameter, while options for `NMaximize` itself are on their own. All of the options for `NMaximize`, as well as all of the global optimization algorithms it can use and all of their options, are explained in detail in the Advanced Documentation section of the documentation for `NMinimize`.

Although the Differential Evolution algorithm is stochastic, by default `NMaximize` uses the same random seed for each invocation. This means that each time the algorithm is run it will return the same answer. In general this is a good default behavior, but sometimes you may wish to run multiple different attempts to maximize the same problem with different random behavior each time. To do this, you should set the `RandomSeed` option in the options to Differential Evolution to something different for each run. For example, to ensure a different random seed every time, we could set the `RandomSeed` option to be the output of a time command. For example:

```
pairwiseMSE[q,dataArr,{b,g}, nMaximizeOptions->{Method->
{"DifferentialEvolution","RandomSeed"->Floor[SessionTime[]]}}]
```

6.2 Performance Tips

Section 5.2 presented an example payoff function and one of the inequalities of the payoff function induced:

$$1[x_i y_j + \beta w_i z_j + x_q y_u + \beta w_q z_u > x_i y_u + \beta w_i z_u + x_q y_j + \beta w_q z_j]$$

Each row of the data array would then contain the eight values

$$x_i y_j, w_i z_j, x_q y_u, w_q z_u, x_i y_u, w_i z_u, x_q y_j, w_q z_j$$

An objective function could access these eight values as the first through eighth elements of a column of the data array, so that for the i^{th} inequality, these would be `dataArray[[1, i]]` through `dataArray[[8, i]]`. A naive implementation of the objective function would be

```
objective[data_,b_] := Module[{total, i},
    total = 0;
    For[i=1, i <=  Length[data], i = i+1,
        total += If[data[[1,i]] + b*data[[2,i]]+data[[3,i]]+b*data[[4,i]] >
                        data[[5,i]]+b*data[[6,i]]+data[[7,i]]+b*data[[8,i]],1
    ] ;
    total
]
```

This is a valid objective function, but it can be made to run significantly faster, cutting estimation time from hours to minutes.

The first thing to notice is that the expression being calculated can be simplified further by collecting terms.

$$
\begin{aligned}
& x_i y_j + \beta w_i z_j + x_q y_u + \beta w_q z_u > x_i y_u + \beta w_i z_u + x_q y_j + \beta w_q z_j \\
\Leftrightarrow \quad & (x_i y_j + x_q y_u - x_i y_u - x_q y_j) + \beta * (w_i z_j + w_q z_u - w_i z_u - w_q z_j) > 0 \qquad (1)
\end{aligned}
$$

A simple optimization would be to compute the values in the parentheses as the elements of the data array, reducing the number of math operations that must be performed on each inequality. The objective function would then become

```
objective[data_,b_] := Module[{total, i},
    total = 0;
    For[i=1, i <=  Length[data], i = i+1,
        total += If[data[[1,i]] + b*data[[2,i]] > 0,1,0];
    ] ;
    total
]
```

This code does significantly less math in the inner loop, which will speed up the evaluation of every routine that depends on the objective function.

The next improvement is to take advantage of Mathematica's vectorization capabilities. When the operation performed inside a loop is relatively quick, as above, the loop overhead (keeping track of iterations, deciding whether to iterate again, updating variables, etc) can be a significant cost in terms of speed. One way to avoid this loop overhead is to rephrase the computation in terms of vectorizable math and matrix operations.

Simple arithmetic operations that are performed on atomic values can also be performed element-by-element on entire vectors. For example, adding together two vectors of numbers produces a vector of the resulting sums:

```
In[1] := {1,2,3} + {9, 8, 7}
Out[1] = {10, 10, 10}
```

All of Mathematica's basic arithmetic operators can be vectorized. The advantage of using vectorization is that the loop overhead is reduced, and the resulting operations can run an order of magnitude more quickly. The objective function can thus be rewritten as

```
objective[data_,b_] := Module[{values, onesorzeros},
    values = data[[1]] + b*data[[2]];
    onesorzeros = values / Abs[values] + 1.0;
    Total[onesorzeros] /2.
]
```

The first line of the objective function creates a vector of the values on the left hand side of the inequality in equation 1. The second line tests each of these values to be greater than or less than zero, and assigns a 2 if they are greater than zero, and a 0

otherwise. The final line sums up the elements of the array of "ones" (really twos) and zeros and divides by two to compensate for the fact that in the previous line we really wanted to assign ones instead of twos, so that the score is half of the sum of the `onesorzeros` variable[1]. The reason for this bizarre procedure is that Mathematica can vectorize division, addition, and the absolute value operation, but it cannot vectorize an `If` statement. Therefore, this is just a clever way to vectorize that operation.

For a second, much more involved example, suppose the production function is

$$f_{\beta_1,\beta_2}(i,j) = r_i s_j/(x_i y_j)^{\beta_1} + \beta_2 w_i z_j$$

where a scale normalization has already been performed. A typical inequality would be

$$1[r_i s_j/(x_i y_j)^{\beta_1} + \beta_2 w_i z_j + r_q s_u/(x_q y_u)^{\beta_1} + \beta_2 w_q z_u > r_i s_u/(x_i y_u)^{\beta_1} + \beta_2 w_i z_u + r_q s_i/(x_q y_i)^{\beta_1} + \beta_2 w_q z_i]$$

In this example, dataArray[[1, i]] through dataArray[[10, i]] would contain

$$\{r_i s_j, x_i y_j, (w_i z_j + w_q z_u), r_q s_u, x_q y_u, r_i s_u, x_i y_u, (w_i z_u + w_q z_i), r_q s_i, x_q y_i\}$$

for each inequality i. A highly optimized objective function would then be

```
objective[data_,b1_,b2_] := Module[{values, onesorzeros},
    values = data[[1]]/(data[[2]]^b1)+b2*data[[3]]
              +data[[4]]/(data[[5]]^b1) + data[[6]]/(data[[7]]^b1)
              +b2*data[[8]]+data[[9]]/(data[[10]]^b1);
    onesorzeros = values / Abs[values] + 1.0;
    Total[onesorzeros] /2.
]
```

This example demonstrates that very general functional forms can be estimated efficiently by the toolkit; there is no limitation to linear functional forms.

One final speed tip is to be sure to use the `Developer`ToPackedArray` function to ensure that the data array and the data map are packed arrays. Packed arrays are restricted to machine-sized integer or floating point numbers, but Mathematica is able to perform operations on them much more quickly. To convert an array into a packed array, all you need to do is execute the line

```
packedArray = Developer`ToPackedArray[array]
```

The Mathematica documentation contains more details about what packed arrays are and how to use them.

[1] Obviously, dividing by two is unnecessary, since the maximum is the same either way, but you might be interested in the score that the maximum occurs at.

6.3 Smoothed Maximum Score

Horowitz (1992) discusses an interesting variation on maximum score. Horowitz's estimator has a smooth objective function (which is easier to optimize numerically), is asymptotically normal, and has a faster rate of convergence than standard maximum score. The smoothed maximum score estimator is implemented by replacing the indicator function $1[\cdot > 0]$ in the maximum score objective function with a kernel $K(\cdot)$. The kernel K has the properties that $K(v)$ is finite-valued for all v, $\lim_{v\to-\infty} K(v) = 0$ and $\lim_{v\to\infty} K(v) = 1$. Unlike in standard kernel estimators, K resembles a CDF instead of a PDF. In the case of pairwise maximum score, this means that inequalities that are not satisfied receive very little weight, and inequalities that are satisfied by a large margin receive the most weight.

Horowitz's estimator is easily implemented in the Match Estimation toolkit by replacing the indicator function with the desired kernel in the user's objective function. In this case, the maximum score objective function will no longer be integer-valued, but the software never makes such an assumption and so estimation will proceed without difficulty. However, estimating confidence regions becomes more difficult, since Horowitz's estimator has a different rate of convergence, which the confidence region routines are not currently written to use. Therefore, the user will have to take the source code from the file `MatchEstimation.m` and change the rate of convergence by hand to that derived by Horowitz. An even better option would be to use a bootstrap, since Horowitz (2002) shows that the bootstrap is consistent for the smoothed maximum score estimator.

References

Abrevaya, Jason and Jian Huang, "On the bootstrap of the maximum score estimator," *Econometrica*, 2005, *73* (4), 1175–1204.

Akkus, Oktay, "The Determinants of Bank Mergers: A Revealed Preference Analysis," 2006.

Bajari, Patrick and Jeremy Fox, "Complementarities and Collusion in an FCC Auction," 2005.

Chernozhukov, Victor, Han Hong, and Elie Tamer, "Parameter Set Inference in a Class of Econometric Models," 2005.

Fox, Jeremy, "Estimating Matching Games with Transfers," 2006.

Han, Aaron K., "Nonparametric Analysis of a Generalized Regression Model," *Journal of Econometrics*, 1987, *35*, 303–316.

Horowitz, Joel, "A Smoothed Maximum Score Estimator for the Binary Response Model," *Econometrica*, 1992, *60* (3), 505–531.

_ , "Bootstrap Critical Values for Tests Based on the Smoothed Maximum Score Estimator," *Journal of Econometrics*, 2002, *111* (2), 141–167.

Kim, Jeankyung and David Pollard, "Cube Root Asymptotics," *The Annals of Statistics*, 1990, *18* (1), 191–219.

Manski, Charles F., "Maximum Score Estimation of the Stochastic Utility Model of Choice," *Journal of Econometrics*, 1975, *3* (3), 205–228.

Politis, Dimitris N., Joseph P. Romano, and Michael Wolf, *Subsampling*, Springer New York, 1999.

Shaikh, Azeem, "Inference for Partially Identified Econometric Models," 2005.

Sherman, Robert P., "The Limiting Distribution of the Maximum Rank Correlation Estimator," *Econometrica*, 1993, *61* (1), 123–137.

Appendix: Function Reference

This section provides a brief description for every routine in the Match Estimation toolkit.

vec

```
vec[mtx_]
Stacks the columns of a matrix into a vector.
```

unvec

```
unvec[v_, rows_]
Takes a vector and "unstacks" it into a matrix with the
number of rows given by the rows parameter.
```

numCoalitions

```
numCoalitions[matchIdxMtx_]
Takes a match matrix and returns the total number of coalitions
across all markets in the match matrix.
```

generateRandomSubsample

```
generateRandomSubsample[ssSize_, groupIDs_, dataArray_]
Generates a subsample of a given size from a data array.

Parameters:
   ssSize - Size of the subsample generated, in terms of the number of
      distinct entities that will be represented in the subsample
      (nests or coalitions).
```

```
   groupIDs - A data map that the routine will use to examine the rows
      of the data array for possible inclusion into the subsample.
   dataArray - A data array structure suitable for passing into the
      objective function.
```

generateAssignmentMatrix

```
generateAssignmentMatrix[payoffs_, quotaU_:1, quotaD_:1,
   options___?OptionQ]
Generates the optimal assignment of matches from the given matrix of
payoffs for each match. In an assignment matrix, each entry (i,j) is
1 if i and j are matched and 0 otherwise.

Parameters:
   payoffs - A list of matrixes, one for each market, where the (i,j)th
      element is the total production from matching i and j.
   quotaU - Maximum number of partners each upstream agent can
      have. Default = 1.
   quotaD - Maximum number of partners each downstream agent can
      have. Default = 1.
```

matchIndexMatrix

```
matchIndexMatrix[assignmentMatrices_, quotaU_:1]
Takes a list of assignment matrixes (see generateAssignmentMatrix) and
generates a list of triply-indexed matrixes with the i'th element
containing the index of i's match. Each element in the list is a list
with two elements, and each of those is a list of lists. The two
elements within each market represent the two sides of the market,
upstream and downstream, in that order. In the upstream half,
the i'th element will be a list with an entry for the i'th coalition
in the market; each of these lists will contain all of the
identifiers of the upstream market participants in that coalition.
The downstream half will contain the same number of entries, each of
which is a list of the downstream members of the coalition.

An example will clarify:

{
  {
    {{1},{2},{3}},
    {{4,5},{6,7},{8,9}}
  },
  {
    {{10,11},{12,13},{14,15}},
```

```
    {{16},{17},{18}}
  }
}

In this very simple example, there are two markets. In the first,
there are three coalitions, each of which has one upstream agent
and two downstream agents. In the second market, there are also
three coalitions, each of which has two upstream agents and one
downstream agent.

If the second argument is given, it makes a match matrix which
has a list of length quotaU in each element. The second for is for
multiple matching; unfortunately, you need to specify this, because it
is not always clear from an assignment matrix alone what the quota was
(it might not have been optimal to take advantage of the possibility of
multiple matching).

Parameters:
   assignmentMatrices - An assignment matrix, as returned from
      generateAssignmentMatrix.
   quotaU - The maximum number of agents an upstream agent
      can match with. Default = 1.
```

allPairsInequalityList

```
allPairsInequalityList[matchMtx_]
Takes a match matrix and returns a list of every possible
pairwise swap, where each one is of the form
{{n, i1, p1}, {n, i2, p2}}, n is a nest index, i1 and i2 are
upstream agents, and p1 and p2 index one of each of their
downstream partners.
```

randomInequalityList

```
randomInequalityList[matchMtx_, inequalitiesPerNest_]
Takes a match matrix and generates a random sample of inequalities
(unlike allPairsInequalityList, which generates all of them).
Parameters:
   matchMtx - A match matrix.
   inequalitiesPerNest - The number of inequalities to pick for each
      nest. Repeated inequalities cannot be ruled out without
      performing the lengthy computation this function is designed to
      avoid, so this parameter should be significantly smaller than
      the number of possible inequalities per nest.
```

pairwiseMSE

```
pairwiseMSE[q_, dataArray_, args_, options___?OptionQ]
Generates an estimate using the pairwise maximum score estimator.

Parameters:
   q - An objective function, which takes a data array and a sequence of
      scalar arguments as parameters.
   dataArray - The data array parameter to use in the objective function.
   args - A list of unique symbols equal in length to the number of
      parameters to estimate. The return value from pairwiseMSE will
      contain a list of replacement rules keyed on the elements of args.
   options - An optional parameter for maximization options.
      The only recognized option is nMaximizeOptions, discussed above.
```

pointIdentifiedCR

```
pointIdentifiedCR[ssSize_, numSubsamples_, pointEstimate_, objFunc_,
   args_, groupIDs_, dataArray_, options___?OptionQ]
Generates a confidence region estimate using subsampling.

Parameters:
   ssSize - The size of each subsample to be estimated.
   numSubsamples - The number of subsamples to use in estimating
      the confidence region.
   pointEstimate - The point estimate to build the confidence region.
   objFunc - The objective function used in pairwiseMSE.
   args - A list of unique symbols used in pairwiseMSE.
   groupIDs - A data map used to generate the subsamples.
   dataArray - The dataArray parameter used in pairwiseMSE.
   options - An optional parameter specifying options. Available options:
      progressUpdate - How often to print progress (0 to disable).
         Default = 0.
      confidenceLevel - The confidence level of the region.
         Default = .95.
      asymptotics - Type of asymptotics to use (nests or coalitions).
         Default = nests.
      subsampleMonitor - An expression to evaluate for each subsample.
         Default = Null.
      symmetric - True or False. If True, the confidence region will
         be symmetric about the estimate.
         Default = False.
```

setIdentifiedCR

```
setIdentifiedCR[ssSize_, numSubsamples_, pointEstimate_, objFunc_,
   args_, groupIDs_, dataArray_, options___?OptionQ]
Generates a confidence region estimate using the set-identified
procedure in Shaikh (2006). The parameters are exactly the same
as pointIdentifiedCR except that symmetric is not an option.
```

sampleSetIdentifiedCR

```
sampleSetIdentifiedCR[ssSize_, numSubsamples_, numSamplePoints_,
   pointEstimate_, objFunc_, args_, groupIDs_,
   dataArray_, options___?OptionQ]
Generates a random set of points that are inside a set-identified
confidence region.

Parameters:
   ssSize - The size of each subsample to be estimated.
   numSubsamples - The number of subsamples to use in estimating the
      confidence region.
   numSamplePoints - The number of points to randomly sample. Some
      subset of these points will hopefully be found to be inside the
      confidence region, and thus returned in the output.
   pointEstimate - The point estimate to build the confidence region
      around (typically the output of pairwiseMSE).
   objFunc - The objective function used in pairwiseMSE.
   args - A list of unique symbols used in pairwiseMSE.
   groupIDs - A data map used to generate the subsamples.
   dataArray - The dataArray parameter used in pairwiseMSE.
   options - An optional parameter specifying options.
      progressUpdate - How often to print progress (0 to disable).
         Default = 0.
      confidenceLevel - The confidence level of the region.
         Default = .95.
      asymptotics - Type of asymptotics to use (nests or coalitions).
         Default = nests.
      subsampleMonitor - An expression to evaluate for each subsample.
         Default = Null.
      samplingVariance - The variance of the distribution used to draw
         the sample points. Default = 20.
```