Online Appendix*

Probabilistic Transitivity in Sports

Johannes Tiwisina Philipp Külpmann

1 Parameter space

In this appendix we explore the effect of the transitivity conditions on the parameter space of winning probabilities to illustrate the limitations enforced by it. To do that we compare the size of the parameter space with transitivity to the space of unrestricted winning probabilities $\overline{S}_n$, e.g. every p_{ij}, p_{ji} fulfilling $p_{ij} + p_{ji} = 1$.
The space of parameters including the transitivity conditions is a subset of this set $\overline{S}_n$. $S_n(R)$ is hereby defined as the size of this space relative to $\overline{S}_n$ only considering the restrictions for $p_{ij} \in R$. The unrestricted parameter space is in this simple case: $\overline{S}_n = [0,1]^{\frac{n(n-1)}{2}}$ which can be easily seen by the fact that every p_{ji} is completely determined by p_{ij}. The restricted space for n players and the transitivity conditions for every $(i,j) \in K_n$ with $K_n = \{(i,j)|i,j \in \{1,2,...,n\}, i < j\}$ is therefore

$$S_n(K_n) = \int_{b_{i+1,j}}^{b_{i,j+1}} S_n(K_n \backslash \{(i,j)\}) \mathrm{d}p_{ij}$$

with

$$S_n((i_0, j_0)) = \int_{b_{i_0+1,j_0}}^{b_{i_0,j_0+1}} \mathrm{d}p_{i_0 j_0}$$

and

$$b_{i,j} := \begin{cases} p_{ij}, & \text{for } (i,j) \in K_n \\ 0.5, & \text{for } i = j \\ 0, & \text{else} \end{cases}$$

*This is the online appendix to Tiwisina and Külpmann (2014).

As this fairly complicated recursive integral may be hard to interpret, Table 1 gives the values for the relative size of the transitive parameter space for up to five teams. It can be seen that the size rapidly shrinks and it is not hard to imagine that for a league comprising e.g. 18 teams the conditions are in this sense very strict.

n	2	3	4	5	6	7
Relative size	1	$\frac{1}{4}$	$\frac{1}{120}$	$\frac{1}{40320}$	$\frac{1}{203212800}$	$\frac{1}{19313344512000}$
Approximation	1	0.25	8.3×10^{-3}	2.5×10^{-5}	4.9×10^{-9}	5.2×10^{-14}

Table 1: Relative size of the transitive parameter space

2 The Linear Ordering Problem

If one is given a complete directed graph $D_n = (V_n, A_n)$ with arc weights c_{ij} for every ordered pair $(i, j) \in V_n \times V_n$, the linear ordering problem consists of finding an acyclic tournament T (which corresponds to a permutation of the set of objects or teams), which maximizes the sum of the arcs which are in agreement with the direction of the arcs from D_n. So the sum $\sum_{(i,j)\in T} c_{ij}$ has to be maximal. Equivalently one could formulate the problem as minimizing the so called remoteness corresponding to minimizing the arc weights pointing in the opposite direction.

A more illustrative representation of the problem is the maximization of the sum of super-diagonal elements in a matrix by manipulating the row/column ordering. This is the so called Triangulation Problem.

The reader might already be able to grasp a sense of similarity here. To establish a direct connection between the LOP and the problem dealt with in this paper, consider a situation where we fix the probabilities of wins and losses at homogeneous values below and above the diagonal of the matrix independently of which teams are in question. This means we set $p_{ijh} = \overline{p}_h$ above diagonal and $p_{ijh} = \underline{p}_h$ below it and analogously for the away probabilities. Let us consider the case where $\overline{p}_h > \underline{p}_h$ and $\overline{p}_a > \underline{p}_a$. Remember that the goal is to maximize

$$\begin{aligned}
&\sum_{(ij)\in E} w_{ijh}\ln(p_{ijh}) + w_{jia}\ln(p_{jia}) + (1 - w_{ijh} - w_{jia})\ln(1 - p_{ijh} - p_{jia}) \\
=&\sum_{(ij)\in \overline{E}} w_{ijh}\ln(\overline{p}_h) + w_{ija}\ln(\overline{p}_a) + t_{ijh}\ln(1 - \overline{p}_h - \underline{p}_a) \\
&+\sum_{(ij)\in \underline{E}} w_{ijh}\ln(\underline{p}_h) + w_{ija}\ln(\underline{p}_a) + t_{ijh}\ln(1 - \underline{p}_h - \overline{p}_a)
\end{aligned}$$

where $\overline{E}$ and $\underline{E}$ represent the sets of elements above and below the diagonals, respectively. $t_{ijh} = t_{jia}$ is the number of times team i ties team j.

The results of a particular team in his two games against a particular opponent makes a certain contribution to the sum. This contribution might be higher because it is multiplied by higher probabilities if the records are superdiagonal. So we are confronted with a triangulation problem just like the one described above. Many Authors suggest an application of the LOP in sports rankings (see e.g. Marti and Reinelt (2011)). And since it indeed seems well suited for our purposes, we will include it in the analysis.

3 Premier League

For the English Premier League, we have data for every season from 1997/98 to 2012/13. Again, we calculate the maximum likelihood p-matrices as well as the objective function values using the "two-points-for-a-win", the "three-points-for-a-win", Elo system and the ranking from the solution to the linear ordering problem. Finally, we apply the Tabu Search algorithm and use the likelihood it finds as a reference value and plot the differences to these likelihoods in a diagram.

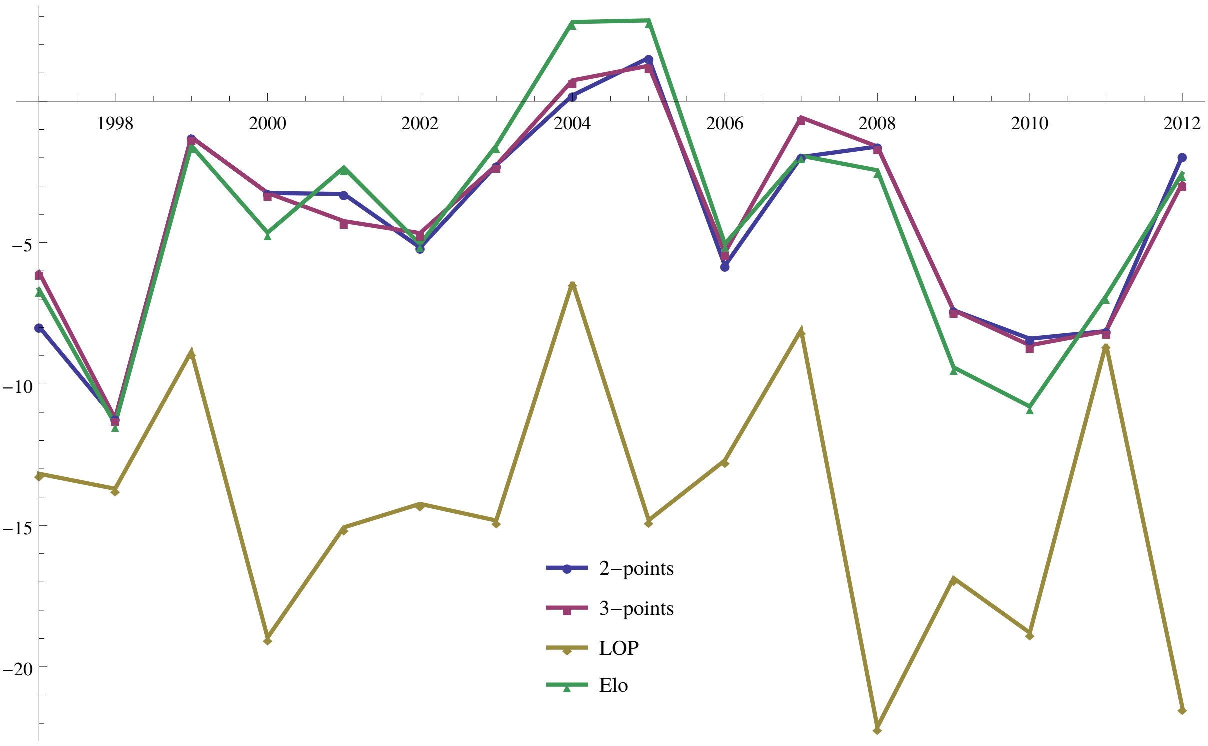

Figure 1: Maximum Likelihoods for Premier League panel data

Figure 1 reveals that the two and three-point systems are very close in their maximum likelihood values. Just like in the Bundesliga, this is because in most cases the rankings determined by the two systems only differ in a few spots and so the likelihood values are not very far apart either. In the Premier League the three-point system has a 5.2% higher explanatory power. The Elo-system also gives us likelihoods in the same range, indicated by the green lines. The ranking resulting from solving the linear ordering problem is never nearly as good as the other ranking systems.

The Tabu Search heuristic is able to improve on every single ranking from the sample, except for the Premier League seasons 04/05 and 05/06. On average it helps to explain the results about 457 times better. Interestingly, the two seasons where Tabu Search was not able to improve upon the 2 and three-point systems are the seasons with the by far best performance of the Elo ranking. There might be a connection here.

4 Code

The first code listing shows the problem definition of an optimization with fixed team ordering, so that the ipopt framework will understand it.

```
#include "nfl_nlp.hpp"

#include <cassert>
#include <iostream>
#include <math.h>

static int t=50;

static int w[3][50][50];
static double p[2][50][50];


using namespace Ipopt;

// constructor
nfl_NLP::nfl_NLP(int myw[][50][50], double* myp[][50][50], double*& ↩
    zielwert, int myt)
{
     zielwert = &zw;
     t=myt;

     for (int h=0; h<3; h++) {
          for (int k=0; k<t; k++) {
               for (int l=0; l<t; l++) {
                    w[ht][k][l]=myw[ht][k][l];
                    myp[ht][k][l]=&p[ht][k][l];
               }
          }
     }

```

```
}

//destructor
nfl_NLP::~nfl_NLP()
{}

// returns the size of the problem
bool nfl_NLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                           Index& nnz_h_lag, IndexStyleEnum& index_style)
{
    // The problem described in nfl_NLP.hpp has 4 variables, x[0] through x[3]
    n = 2*pow(t,2);

    // one equality constraint and one inequality constraint
    m =  pow(t,2) + 4*t*(t-1);

    // in this example the jacobian is dense and contains 8 nonzeros
    nnz_jac_g = 2*m;

    // the hessian is also dense and has 16 total nonzeros, but we
    // only need the lower left corner (since it is symmetric)
    nnz_h_lag = 2*n-4*t;

    // use the C style indexing (0-based)
    index_style = TNLP::C_STYLE;

    return true;
}

// returns the variable bounds
bool nfl_NLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                              Index m, Number* g_l, Number* g_u)
{
    // here, the n and m we gave IPOPT in get_nlp_info are passed back to us.
    // If desired, we could assert to make sure they are what we think they are.

    // the variables have lower bounds of 0
    for (Index i=0; i<2*t*t; i++) {
        x_l[i] = 0.0;
    }

    // the variables have upper bounds of 1
    for (Index i=0; i<2*t*t; i++) {
```

```
        x_u[i] = 1.0;
    }


    Index i = 0;
    for (Index k=0; k<t; k++) {
        for (Index l=0; l<t; l++) {
            g_l[i] = -2e19;
            g_u[i] = 1.0;
            i++;
        }
    }
    for (Index h=0; h<2; h++) {
        for (Index k=0; k<t; k++) {
            for (Index l=0; l<t; l++) {
                if (l<t-1) {
                    g_l[i] = -2e19;
                    g_u[i] = 0.0;
                    i++;
                }
                if (k<t-1) {
                    g_l[i] = -2e19;
                    g_u[i] = 0.0;
                    i++;
                }
            }
        }
    }

    return true;
}

// returns the initial point for the problem
bool nfl_NLP::get_starting_point(Index n, bool init_x, Number* x,
                                 bool init_z, Number* z_L, Number* z_U,
                                 Index m, bool init_lambda,
                                 Number* lambda)
{
    assert(init_x == true);
    assert(init_z == false);
    assert(init_lambda == false);

    // initialize to the given starting point
    for (Index i=0; i<2*t*t; i++) {
        x[i] = 0.4;
    }

```

```
    return true;
}

// returns the value of the objective function
bool nfl_NLP::eval_f(Index n, const Number* x, bool new_x, Number& obj_value)
{
    assert(n == 2*t*t);

    obj_value=0;
    Index i=0;
    for (Index k=0; k<t; k++) {
        for (Index l=0; l<t; l++) {
            if (k!=l) {
                obj_value += (-w[0][k][l]*log(x[t*k+l]+0.000001) - w[1][l][k]*log(x[t*t+t*l+k]+0.000001) - (w[2][k][l]-w[0][k][l]-w[1][l][k])*log(1-x[t*k+l]-x[t*t+t*l+k]+0.000001));

                i++;
            }
        }
    }

    return true;
}

// return the gradient of the objective function grad_{x} f(x)
bool nfl_NLP::eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f)
{
    Index i=0;
    for (Index h=0; h<2; h++) {
        for (Index k=0; k<t; k++) {
            for (Index l=0; l<t; l++) {
                if (k!=l) {
                    if (h==0)
                        grad_f[i] = -w[ht][k][l]/(x[t*t*h+t*k+l]+0.000001) + (w[2][k][l]-w[ht][k][l]-w[1-h][l][k])/(1-x[t*t*h+t*k+l]-x[t*t*(1-h)+t*l+k]+0.000001);
                    else
                        grad_f[i] = -w[ht][k][l]/(x[t*t*h+t*k+l]+0.000001) + (w[2][l][k]-w[ht][k][l]-w[1-h][l][k])/(1-x[t*t*h+t*k+l]-x[t*t*(1-h)+t*l+k]+0.000001);
                }
                else{
```

```
                    grad_f[i] = 0;
                }
                i++;
            }
        }
    }

    return true;
}

// return the value of the constraints: g(x)
bool nfl_NLP::eval_g(Index n, const Number* x, bool new_x, Index m, Number* g)
{
    Index i = 0;
    for (Index k=0; k<t; k++) {
        for (Index l=0; l<t; l++) {
            g[i] = x[t*k+l]+x[t*t+t*l+k];
            i++;
        }
    }
    for (Index h=0; h<2; h++) {
        for (Index k=0; k<t; k++) {
            for (Index l=0; l<t; l++) {
                if (l<t-1) {
                    g[i] = x[t*t*h+t*k+l]-x[t*t*h+t*k+l+1];
                    i++;
                }
                if (k<t-1) {
                    g[i] = x[t*t*h+t*(k+1)+l]-x[t*t*h+t*k+l];
                    i++;
                }
            }
        }
    }
    return true;
}

// return the structure or values of the jacobian
bool nfl_NLP::eval_jac_g(Index n, const Number* x, bool new_x,
                           Index m, Index nele_jac, Index* iRow, Index *jCol,
                           Number* values)
{
    if (values == NULL) {
        // return the structure of the jacobian

```

```
      // this particular jacobian is dense


      Index z=0;
      Index r=0;
      for (Index k=0; k<t; k++) {
          for (Index l=0; l<t; l++) {
              iRow[z] = r;
              jCol[z] = t*k+l;
              z++;
              iRow[z] = r;
              jCol[z] = t*t+t*l+k;
              z++;

              r++;
          }
      }
      for (Index h=0; h<2; h++) {
          for (Index k=0; k<t; k++) {
              for (Index l=0; l<t; l++) {
                  if (l<t-1) {
                      iRow[z] = r;
                      jCol[z] = t*t*h+t*k+l;
                      z++;
                      iRow[z] = r;
                      jCol[z] = t*t*h+t*k+l+1;
                      z++;
                      r++;
                  }
                  if (k<t-1) {
                      iRow[z] = r;
                      jCol[z] = t*t*h+t*k+l;
                      z++;
                      iRow[z] = r;
                      jCol[z] = t*t*h+t*(k+1)+l;
                      z++;
                      r++;
                  }
              }
          }
      }
      assert(z==nele_jac);
  }
  else {
      Index z=0;
      Index r=0;
      for (Index k=0; k<t; k++) {
```

```
            for (Index l=0; l<t; l++) {
                values[z] = 1;
                z++;
                values[z] = 1;
                z++;

                r++;
            }
        }
        for (Index h=0; h<2; h++) {
            for (Index k=0; k<t; k++) {
                for (Index l=0; l<t; l++) {
                    if (l<t-1) {
                        values[z] = 1;
                        z++;
                        values[z] = -1;
                        z++;
                        r++;
                    }
                    if (k<t-1) {
                        values[z] = -1;
                        z++;
                        values[z] = 1;
                        z++;
                        r++;
                    }
                }
            }
        }
        assert(z==nele_jac);
    }

    return true;
}

//return the structure or values of the hessian
bool nfl_NLP::eval_h(Index n, const Number* x, bool new_x,
                       Number obj_factor, Index m, const Number* lambda,
                       bool new_lambda, Index nele_hess, Index* iRow,
                       Index* jCol, Number* values)
{
    if (values == NULL) {
        // return the structure. This is a symmetric matrix, so we fill the lower left
        // triangle only.
        Index i=0;
        for (Index h=0; h<2; h++) {
```

```
            for (Index k=0; k<t; k++) {
                for (Index l=0; l<t; l++) {
                    if (k!=l) {
                        iRow[i] = t*t*h+t*k+l;
                        jCol[i] = t*t*h+t*k+l;
                        i++;

                        iRow[i] = t*t*h+t*k+l;
                        jCol[i] = t*t*(1-h)+t*l+k;
                        i++;
                    }
                }
            }
        }
    }
    else {
        // return the values. This is a symmetric matrix, fill the lower left
        // triangle only

        Index i=0;
        for (Index h=0; h<2; h++) {
            for (Index k=0; k<t; k++) {
                for (Index l=0; l<t; l++) {
                    if (k!=l) {
                        if (h==0){
                            values[i] = obj_factor *( w[ht][k][l]*pow(x[t*t*h+t*k+l]+0.000001, -2) + (w[2][k][l]-w[ht][k][l]-w[1-h][l][k])*pow((1-x[t*t*h+t*k+l]-x[t*t*(1-h)+t*l+k]+0.000001),-2));
                            i++;
                            //std::cout << "values[" << i << "] = " << values[i] << std::endl;
                            values[i] = obj_factor *0.5*( (w[2][k][l]-w[ht][k][l]-w[1-h][l][k])*pow((1-x[t*t*h+t*k+l]-x[t*t*(1-h)+t*l+k]+0.000001),-2));
                        }
                        else{
                            values[i] = obj_factor *( w[ht][k][l]*pow(x[t*t*h+t*k+l]+0.000001, -2) + (w[2][l][k]-w[ht][k][l]-w[1-h][l][k])*pow((1-x[t*t*h+t*k+l]-x[t*t*(1-h)+t*l+k]+0.000001),-2));
                            i++;
                            //std::cout << "values[" << i << "] = " << values[i] << std::endl;
                            values[i] = obj_factor *0.5*( (w[2][l][k]-w[ht][k][l]-w[1-h][l][k])*pow((1-x[t*t*h+t*k
```

```
                                +l]-x[t*t*(1-h)+t*l+k]+0.000001),-2));
                        }
                        i++;
                    }
                }
            }
        }
    }

    return true;
}

void nfl_NLP::finalize_solution(SolverReturn status,
                                  Index n, const Number* x, const Number* z_L, const Number* z_U,
                                  Index m, const Number* g, const Number* lambda,
                                  Number obj_value,
                                  const IpoptData* ip_data,
                                  IpoptCalculatedQuantities* ip_cq)
{
    // here is where we store the solution to variables
    // so we could use the solution.

    for (Index h=0; h<2; h++) {
        for (Index k=0; k<t; k++) {
            for (Index l=0; l<t; l++) {
                //std::cout << x[t*t*h+t*k+l] << "  ";
                p[ht][k][l]=x[t*t*h+t*k+l];
            }
            //std::cout << "  " << std::endl;
        }
        //std::cout << "  " << std::endl;
    }

    zw = obj_value;
}
```

The second code snippet shows how the program reads 5 NFL seasons, puts them in different orders, and then optimizes the probabilities and prints them.

```
int main(int argv, char* argc[])
{

    // Create a new instance of the nlp
```

```
    SmartPtr<TNLP> mynlp;
    SmartPtr<TNLP> mynlp2;

    // Create a new instance of IpoptApplication
    SmartPtr<IpoptApplication> app = IpoptApplicationFactory();

    // Change some options
    app->Options()->SetIntegerValue("print_level", 0);
    app->Options()->SetNumericValue("tol", 1e-4);
    app->Options()->SetStringValue("mu_strategy", "adaptive");
    app->Options()->SetStringValue("output_file", "ipopt.out");

    // Intialize the IpoptApplication and process the options
    ApplicationReturnStatus status;
    status = app->Initialize();
    if (status != Solve_Succeeded) {
      std::cout << std::endl << std::endl << "*** Error during initialization!" << std::endl;
      return (int) status;
    }

    srand(time(NULL));


    double* z;  //Variable for the likelihood
    int tempW50[3][50][50]; //temporary result matrices

    for (Index h=0; h<3; h++)
        for (Index k=0; k<50; k++)
            for (Index l=0; l<50; l++)
                tempW50[ht][k][l]=w050[ht][k][l];

    double l[14][6];

    //for the years 2000 till 2005, the nfl data is read from the files
    for (int jahr=100; jahr<105; jahr++) {
        int t;
        if (jahr<95)
            t=28;
        else if (jahr<99)
            t=30;
        else if (jahr<102)
            t=31;
        else
            t=32;

        t=10;
```

```


        einlesenNfl(jahr,t);

        //order according to the two-point system and run the optimization
        ordneNachPunkteSystem50(2,w50,t);
        mynlp2 = new nfl_NLP(w50,p50,z,t);
        app->OptimizeTNLP(mynlp2);
        l[jahr][1]=-*z;

        //order according to the three-point system and run the optimization
        ordneNachPunkteSystem50(3,w50,t);
        mynlp2 = new nfl_NLP(w50,p50,z,t);
        app->OptimizeTNLP(mynlp2);
        l[jahr][2]=-*z;

        //order according to the LOP system and run the optimization
        ordneNachLOP50(w50,t);
        mynlp2 = new nfl_NLP(w50,p50,z,t);
        app->OptimizeTNLP(mynlp2);
        l[jahr][3]=-*z;

        //order according to the ELO system and run the optimization
        ordneNachSchach50(w50,t);
        mynlp2 = new nfl_NLP(w50,p50,z,t);
        app->OptimizeTNLP(mynlp2);
        l[jahr][4]=-*z;

        //run the Tabu Search for 100 iterations and then run the optimization
        l[jahr][5]=tabuSearch50(100,t);

        //print out the results
        cout<<l[jahr][0]<<" "<<l[jahr][1]<<" "<<l[jahr][2]<<" "<<l[jahr][3]<<" "<<l[jahr][4]<<" "<<l[jahr][5]<<endl;
        cout<<endl;
    }
```

References

Rafael Marti and Gerhard Reinelt. The linear ordering polytope. In *The Linear Ordering Problem*, pages 117–143. Springer Berlin Heidelberg, 2011.

Johannes Tiwisina and Philipp Külpmann. Probabilistic transitivity in sports. *Center for Mathematical Economics Working Paper*, (520), 2014.