Modeling Ockeghemian Meter

An Addendum to "The Other *Missa Prolationum*"

William Watson

Background

This document provides a fuller accounting of the regression modeling that I employed in "The Other *Missa Prolationum*," *Journal of Musicology* 37.3 (2020): [pgs] (hereafter "TOMP"). Relying on a corpus of 100 sections from 4-voice settings of the Ordinary of the Mass composed by Johannes Ockeghem, I trained a logistic model that identifies whether a section of stylistically comparable music is more likely to have been metrically organized according to perfect or to imperfect *tempus* (minor *prolatio* is assumed). By applying this model to the thirteen sections that together make the *Missa Prolationum*, each of which is a mensuration canon, I was able to argue that the mensural *resolutiones* of this mass contained within fascicle XX of the manuscript Vienna, Österreichische Nationalbibliothek 11883 constitute a compelling metric analysis of the music. I provide some details about this model in an appendix to TOMP, but this document goes into much greater depth. I'm assuming that, since you're reading this alongside / instead of that appendix, you care about and are at least somewhat familiar with statistical modeling (in particular, you can read R and you know what logistic regression is). That being the case, I'm going to leave most of the musical particulars out after the following paragraph, and will instead treat this as a pretty straight-ahead exercise in modeling. If you want to know why I use "capta" instead of "data," please read the full article (or at least the critical data studies that I cite).

The Capta

I began by identifying every cadence in each section of music (for more details about how I defined "cadence" and how the corpus was chosen, see TOMP), and recording its position with respect to two global metric grids, one of perfect *tempus*, another of imperfect (i.e., one of three semibreves and one of two). Evaluating the relative frequencies with which cadences occurred in each metric position within a given section gave me five variables to play with: three to record the relative frequencies with which cadences occurred on beat-classes 1, 2, and 3 of a hypothetical global triple meter (let's call these x_1, x_2, and x_3), and two to record the equivalent relative frequencies for a hypothetical global duple meter (y_1 and y_2). If the musical terminology is opaque to you, don't worry about it. All that really matters is that $\{x_n\}$ and $\{y_n\}$ are sets of relative frequencies.

Since $x_1 + x_2 + x_3 = 1 = y_1 + y_2$, we can immediately get rid of one informationally redundant x and one similarly redundant y. So for modeling purposes we're only interesed in x_1, x_2, and y_1. Attach a binary indication of whether a given section of music is really in perfect or imperfect *tempus*, and we have the beginnings of a captaset waiting to be analyzed. The table below gives an excerpt of this captaset, covering six sections (don't worry about the cryptic section labels, although they should be legible if you know Ockeghem's œuvre).

section	triple	x_1	x_2	y_1
Cp_A1	1	0.6666667	0.1666667	0.3333333
Cp_A2	1	1	0	0.6666667
Cp_A3	0	0.5	0.5	1

section	triple	x_1	x_2	y_1
MM_K1	1	0.5	0	0.5
MM_Chr	1	1	0	1
MM_K2	1	1	0	0.6666667
...	...	...	...	...

Exploratory Analysis

With this captaset in hand, let's start playing around with it (as a tibble called `ockeghem`). First, let's get a handle on what the balance of perfect and imperfect *tempus* is like in this corpus.

```
mean(ockeghem$triple)
```

```
## [1] 0.62
```

Since the corpus is not perfectly balanced (about 60% of the corpus is in perfect *tempus*), we should be alert to the danger of constructing a model that is insufficiently sensitive to the less common class (imperfect *tempus*). Now let's quickly look at some correlations, just to get a slightly finer sense for what's happening in this captaset and for what we should expect once we move on to regression below.

```
cor(ockeghem[sapply(ockeghem, is.numeric)])
```

```
##            triple         x1         x2         y1
## triple  1.0000000  0.6839898 -0.6116200 -0.6126196
## x1      0.6839898  1.0000000 -0.8201029 -0.4378775
## x2     -0.6116200 -0.8201029  1.0000000  0.3682637
## y1     -0.6126196 -0.4378775  0.3682637  1.0000000
```

This looks fairly promising. Strongish correlations between `triple` and all the other variables, and the contrasting signs indicate that x_1 and the other variables should usefully push our classifier in different directions. With that in mind, let's construct a preliminary, almost-certainly overfit model.

```
m <- glm(triple ~ x1 + x2 + y1, data = ockeghem, family = binomial)
summary(m)
```

```
##
## Call:
## glm(formula = triple ~ x1 + x2 + y1, family = binomial, data = ockeghem)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4620  -0.1897   0.1645   0.3348   2.5466
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)    2.231      1.938   1.151 0.249795
## x1             5.161      2.025   2.549 0.010791 *
## x2            -4.408      2.824  -1.561 0.118492
## y1            -6.616      1.719  -3.850 0.000118 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 132.813  on 99  degrees of freedom
```

```
## Residual deviance:  50.868  on 96  degrees of freedom
## AIC: 58.868
##
## Number of Fisher Scoring iterations: 6
```

Hmm. This is less than ideal. We have a nice p-value for y_1, but everything else is a little disappointing. It's particularly troubling that the "equal but opposite" relation we hypothesized above doesn't seem to hold, and that there is such a discrepancy between the significances of y_1 and the two x variables. But if we glance back at the correlation matrix printed above, it's pretty easy to identify the culprit: non-trivial correlation between x_1 and x_2. There are a couple of options for dealing with this problem, but I decided to go with the easiest one: get rid of x_2. After all, it has a higher p-value, a smaller coefficient in absolute terms, and a smaller variance than x_1 (0.06 vs. 0.10, in case you were interested). Let's rerun the regression analysis without the potentially confounding x_2 (and without the apparently useless intercept term) and see what we get.

```
m <- glm(triple ~ x1 + y1 - 1, data = ockeghem, family = binomial)
summary(m)
```

```
##
## Call:
## glm(formula = triple ~ x1 + y1 - 1, family = binomial, data = ockeghem)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4728  -0.2780   0.1467   0.3459   2.6423
##
## Coefficients:
##    Estimate Std. Error z value Pr(>|z|)
## x1    7.249      1.410   5.142 2.72e-07 ***
## y1   -6.359      1.343  -4.736 2.18e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 138.629  on 100  degrees of freedom
## Residual deviance:  53.689  on  98  degrees of freedom
## AIC: 57.689
##
## Number of Fisher Scoring iterations: 6
```

Much better. Solid p-values, coefficients that make sense, this is the kind of model that we want to work with. We still have some fine-tuning to do with the coefficients to mitigate the danger of overfitting—remember that we eventually want to use our model to classify some stylistically similar music outside of our curated captaset—but this is still a pretty good place to be. So now let's start to think about how to expand on this initial success with an eye toward what we actually want to use our model to do.

Iterating and Validating

Our goal is to use this model to generate estimated "resolved mensurations" for the *Missa Prolationum*, a piece for which the relation between mensuration and meter is not as obvious as it is for most fifteenth-century music. (I argue in the article that there is, in the end, a relation between signed mensuration and metric organization in this piece, but we aren't going into that here.) That is, the corpus of Ockeghemian music we've been working with is just a training set, and so overfitting is a nontrivial concern here. There are a few options for dealing with this, but once again I'm going with something fairly simple.

1. Build a number of models using resampled training sets.
2. Validate these models on the corresponding resampled validation sets.
3. Use those validated models to construct a final model whose coefficients are weighted averages of the validated models' coefficients (the weights simply being the validated models' success rates).

In case you're following along with TOMP, the final results of this aren't going to be exactly the same as what I present there, partially because I left in the trailing coefficient in each of the iterated models for the article (although the effect from this should be negligible), and partially because pseudorandom resampling is pseudorandom. While there are ways to automate this process in R, it doesn't hurt to slow down and dig into the weeds every once in a while. So let's set up a simple loop.

```
# Proportion of captaset used for the training set in each iteration.
TRAIN_PPN <- 0.75

# Number of iterations
ITER <- 25

# Initialize data storage structure
modelSet <- vector()

# Set size of training and test sets
trainSize <- floor(TRAIN_PPN * nrow(ockeghem))

# To make specific segmentation replicable
# set.seed(1)

# Loop for iterating the model
for(i in 1:ITER)    {

    # Segment data into test and training sets
    trainIndex <- sample(nrow(ockeghem), trainSize, replace = FALSE)
    trainSet <- ockeghem[trainIndex, ]
    valSet <- ockeghem[-trainIndex, ]

    # Build model
    m <- glm(triple ~ x1 + y1 - 1, data = trainSet, family = binomial)

    # Validate the model using the test set, shoehorn predictions into a binary world,
    # compute success rate
    p <- predict(m, newdata = valSet, type = "response")
    for (j in 1:length(p)){
      if (p[j] > 0.5){
        p[j] <- 1
      }
      else{
        p[j] <- 0
      }
    }
    s <- mean(1 - abs(valSet$triple - p))

    # Add coefficients and success rate to storage structure
    modelSet <- rbind(modelSet, cbind(t(m$coefficients), s))

}
```

```
modelSet <- as_tibble(modelSet)
```

This gives us a set of model coefficients and a set of predictions that we can use to validate them. So let's take a look at our average success rate.

```
mean(modelSet$s)
```

```
## [1] 0.8992
```

Not bad at all. So if all has gone as it should, we should expect our final model to have a success rate of about 0.9. Let's see.

```
# Calculate final model coefficients
xCoef <- weighted.mean(modelSet$x1, modelSet$s)
yCoef <- weighted.mean(modelSet$y1, modelSet$s)

# Calculate final modeled probabilities of perfect tempus, shoehorn into categories
ockeghem$pFinal <- plogis(xCoef*ockeghem$x1 + yCoef*ockeghem$y1)
ockeghem$pTriple <- -1
for (i in 1:nrow(ockeghem)){
  if (ockeghem$pFinal[i] > 0.5){
    ockeghem$pTriple[i] <- 1
  }
  else{
    ockeghem$pTriple[i] <- 0
  }
}

# Calculate final success rate
sFinal <- mean(1 - abs(ockeghem$triple - ockeghem$pTriple))
sFinal
```

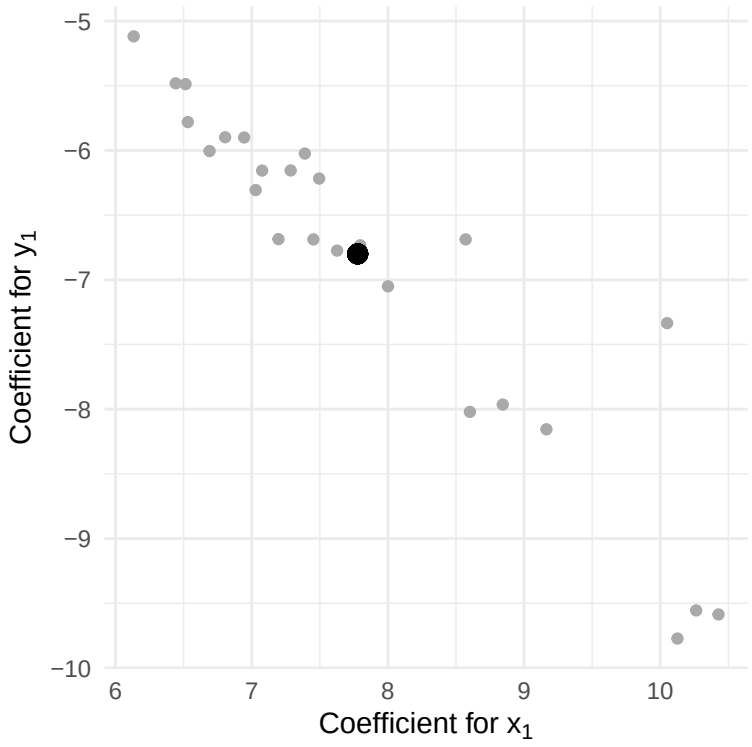
```
## [1] 0.91
```

Excellent.

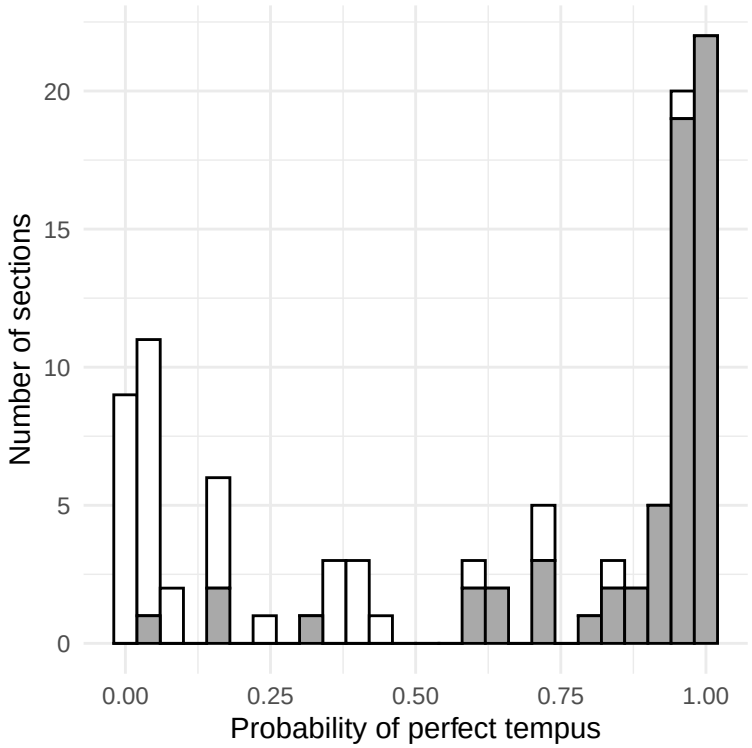
Figures

In case we want to do a little more granular analysis, we can produce a couple of figures (that might be familiar from TOMP) to show the distribution of model coefficients and modeled probabilities of perfect *tempus* that we're talking about here.

This scatterplot shows the coefficients for the various iterations of the models in gray (which should be roughly collinear), and the final, wieghted average coefficients in black.



And this histogram shows the distribution of modeled probablilities of perfect *tempus*, segmented by true *tempus*.



As the histogram makes clear, the errors that the final model makes in the initital Ockeghemian corpus are balanced between the two categories, which is a relief given the unevenness of the corpus that flagged above.

And that's it. Fairly straightforward modeling, just like I said!