Meiosis-induced Alterations in Transcript Architecture and Noncoding RNA Expression in *S. cerevisiae*

Karen S. Kim Guisbert, Yong Zhang, Jared Flatow, Sara Hurtado, Jonathan P. Staley, Simon Lin, Manyuan Long and Erik J. Sontheimer

SUPPLEMENTAL MATERIAL

Novel computational pipelines for the analysis of tiling microarrays

Our dataset contains measurements of probe expression of a high-density genome tiling array. The measurements pertain to samples of yeast taken at regular intervals during meiosis, as well as several other non-time-series conditions. It is not obvious, nor is it known, what is the best way to conduct the analysis of such data. In our case, we are interested in characterizing the alternative expression of different regions along the yeast genome, as well as developing an overall picture of the genomic activity under the varying conditions. Our analysis techniques are guided by these overarching goals, as well as computational considerations.

To achieve our aims, we designed and implemented several novel analysis pipelines that are described below. We provide a justification for each pipeline, pseudo-code for its implementation under a *mapreduce* framework (http://labs.google.com/papers/mapreduce.html). We choose to use mapreduce in anticipation of the fact that a natural extension to improve our results would be to simply increase the number of samples included in the analysis. As the number of samples increases, our pipelines suffer little to no time penalty, as long as the number of nodes in our cluster remains greater than or equal to the number of samples.

Normalization. Normalization allows us to compare probe expression values from one sample to the next. We have normalized the mRNA data for each sample by assuming the probes come from one of two distributions, expressed or unexpressed. We calculate the mean and standard deviation of the unexpressed probes. All probes are scaled such that the background mean falls at 0, and 90% of the probes have a score < 1.

```
def map(samples):
    for sample in samples:
        for probe in sample.probes:
            probe.score = (probe.score - sample.background_mean) / sample.nth_percentile(n=90)
            yield probe
```

Segmentation. The goal of segmentation is to identify regions, called *segments*, where changes in transcription activity may be occurring. Segments give us a basis for comparison across samples, and reduce the complexity of the data. Segments have an intuitive interpretation as the largest contiguous regions along the genome which have the potential to be expressed. In other words, regions not contained within a segment are *never* expressed under any condition in our samples. Probes within the same segment could be part of the same transcript, or part of some

kind of alternative transcript of the same region.

Fragmentation. Fragmentation identifies contiguous regions of expression, within each sample. As shown in the normalization figure, we define the expressed probes to be those that are more than 2 standard deviations above the unexpressed mean. *Fragments* are defined as regions of 5 or more overlapping expressed probes. Due to the tiling array design, a single position in the genome can be covered by up to 5 probes. Using a minimum number of 5 overlapping expressed probes is therefore reasonable in order to lessen the dependence between expression value and probe affinity. If there is a gap in coverage along the genome, we will ignore it if the probes on both sides of the gap are expressed. The rationale for this is that we simply do not have enough evidence to split the fragments on either side of the gap. Essentially, we take a conservative approach by assuming that any part of the genome which does not have probes to cover it, could potentially be expressed.

Fragmentation can be implemented as the map function of segmentation:

```
def map(samples):
    for sample in samples:
        for gap in sample.coverage_gaps:
            if sample.expressed_on_both_sides_of_gap(gap):
               sample.close_gap(gap)

        for region in sample.expressed_regions:
            if len(region.probes) >= 5:
               yield region
```

We define the segments to be the union of the fragments from all of the samples. In this way, segments give global boundaries for the expressed regions across the samples. Forming such a unified segmentation allows us to consider the alternative splicing for each segment across the samples. In order to reduce noise, we only keep segments containing at least 40 probes.

```
segments = efficient_avl_tree()

def reduce(regions):
    for region in regions:
        segments.insert(region)

    for segment in segments:
        if len(segment.probes) >= 40:
           yield segment
```

We choose 40 probes as a minimum in order to eliminate the majority of poorly covered segments which also contain large gaps in coverage. As shown below, there is a population of segments which do not have many probes covering them, but which cover a disproportionately large portion of the genome due to gaps in coverage. The minimum number of probes constraint truncates the segment list such that we only consider higher quality segments.

Factorization. Factorization breaks down each segment into *transcription units*. Transcription

units of each segment form a basis which can be used to approximate the expression of any given sample. We use a non-negative matrix factorization (nmf), since the factors are meant to represent physically realizable transcription units. The implementation we used was derived from the *Matlab* implementation given at http://www.broad.mit.edu/cancer/pub/nmf.

Determining the number of transcription units in a particular segment is key. We want each segment to contain as few transcription units as possible while capturing as much of the expression information as possible. In order to do this we begin by factoring into one transcription unit, and look at the error between the approximation and the real data. We perform a principal component analysis on the error matrix to see if it is random or still contains some structure. If most of the energy of the error matrix is contained within the first principal component, we assume there must be another transcription unit. We repeat this process until there is no structure leftover in the error matrix.

```
def minimal_nmf(V):
    num_transcription_units = 0
    while True:
        num_transcription_units = num_transcription_units + 1
        W, H = nmf(V, num_transcription_units)
        U = V - dot(W, H)
        L, S, Z, k = pca(U, cutoff=.5)
        if k > 1:
           return W, H
```

We can formulate the factorization as a map function that computes this minimal non-negative matrix factorization for each of the segments:

```
def map(segments):
    for segment in segments:
        W, H = minimal_nmf(segment.data)
        for factor, coefficients in zip(W.cols, H.rows):
                yield TranscriptUnit(segment, factor, coefficients)
```