

Problem Set 1

1. **Evolutionary Distance and whole-genome duplication**

a. Basic Implementation of Needleman-Wunsch:

i. Relevant Python Code:

```
def seqalignDP(seq1, seq2, subst_matrix, gap_pen):
    """return the score of the optimal Needleman-Wunsch alignment for seq1 and
       Note: gap_pen should be positive (it is subtracted)
    """
    # initialize the F and TB matrices
    F = [[0 for j in range(len(seq2)+1)] for i in range(len(seq1)+1)]
    TB = [[PTR_NONE for j in range(len(seq2)+1)] for i in range(len(seq1)+1)]

    n = len(seq1) + 1
    m = len(seq2) + 1

    # initialize dynamic programming table for Needleman-Wunsch alignment (Dur
    for i in range(1, n):
        F[i][0] = 0 - i*gap_pen
        TB[i][0] = PTR_GAP2 # indicates a gap in seq2
    for j in range(1, m):
        F[0][j] = 0 - j*gap_pen
        TB[0][j] = PTR_GAP1 # indicates a gap in seq1

    for i in range(1, n):
        for j in range(1, m):

            # the bases corresponding to F[i][j] are actually seq1[i-1] and seq2[j]
            ai = base_idx[seq1[i-1]]
            bj = base_idx[seq2[j-1]]

            match = F[i-1][j-1] + S[ai][bj]
            # careful, gap_pen is positive
            delete = F[i-1][j] - gap_pen
            insert = F[i][j-1] - gap_pen

            F[i][j] = max(match, delete, insert)
            result = F[i][j]

            ptr_for_result = { match: PTR_BASE,
                               delete: PTR_GAP2,
                               insert: PTR_GAP1}

            TB[i][j] = ptr_for_result[result]

    # return the score, the F matrix, and the backpointer matrix
    return F[len(seq1)][len(seq2)], F, TB
```

ii. Optimal Alignment of **AGGTGAT** and **AGTAA**:

Score matrix:

| | A | G | T | A | A | |
|---|-------|------|------|------|------|-------|
| [| [0, | -4, | -8, | -12, | -16, | -20], |
| A | [-4, | 3, | -1, | -5, | -9, | -13], |
| G | [-8, | -1, | 6, | 2, | -2, | -6], |
| G | [-12, | -5, | 2, | 4, | 1, | -3], |
| T | [-16, | -9, | -2, | 5, | 2, | -1], |
| G | [-20, | -13, | -6, | 1, | 4, | 1], |
| A | [-24, | -17, | -10, | -3, | 4, | 7], |
| T | [-28, | -21, | -14, | -7, | 0, | 3]] |

Optimal score: 3

One optimal alignment:

AGGTGAT

AG-TAA-

iii. Human and mouse HoxA13 alignment score: **2971**

- b. Compute the maximum possible score (score of self-alignment) = $\text{len}(\text{seq1}) * S[\text{diag}]$, assuming that seq1 is the longer of the two sequences and that $S[\text{diag}]$ returns the value of S along the diagonal.

$$\text{Distance} = -y * (\text{returned_score} - \text{max_score})$$

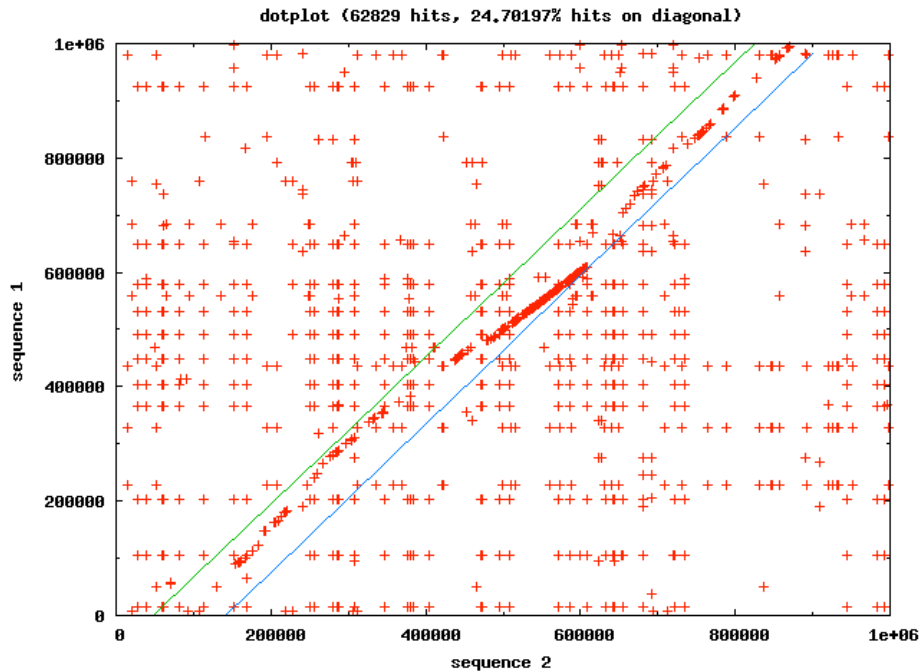
where y is some scalar representing the number of years/ unit score difference.

- c. For the HoxA13 gene, the resulting unscaled distance ($y = 1$) is 530, so I've decided to make $y = 70,000,000 / 530 \sim 132,075$. This naïvely assumes that time of divergence scales linearly with score. We're also assuming here that HoxA13 is a good model for deriving a scale factor, i.e. the time taken for HoxA13 to mutate is representative of mutation times in all regions of all genomes at large, which is patently untrue. Regardless, it's a start.
- d. The modified program estimates that HoxA13 and HoxD13 diverged **332,366,737** years ago. This was determined by taking the average of the unweighted scores comparing human A/D and mouse A/D (2534 and 2499, respectively), and multiplying that average by our naïve scaling factor (132,075).

This estimate, surprisingly, agrees decently with the current predominant estimate of the divergence, which pegs the duplication event at ~ 450 MYa.¹

2. Sequence hashing and dotplot visualization

- a. Below is the result of running the unmodified script on human-hoxa-region.fa (sequence 1) and mouse-hoxa-region.fa (sequence 2):



There are 62,829 hits, $\sim 24.7\%$ of which fall on the diagonal.

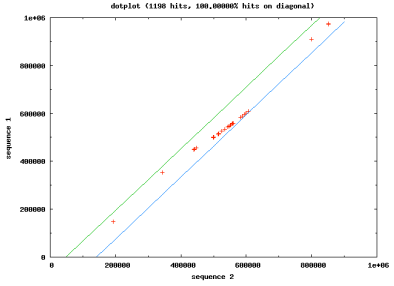
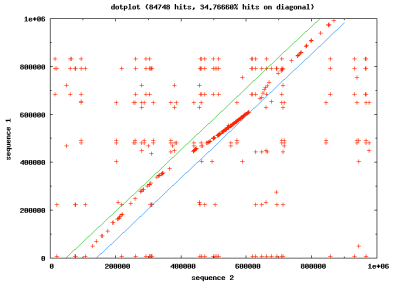
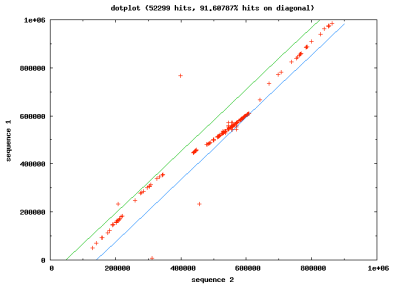
The off-diagonal hits are surprisingly tabular, with repeats occurring every 20,000 bases in some regions. These bits are likely common motifs, like promoters or ribosome binding sites. The highest region of conservation occurs between 400 and 600 thousand base pairs: here lies a massive, unbroken cluster of matches along the diagonal. Such matches are more likely to be orthologous because not only is the subsequence conserved: it also occurs at the same location across both sequences.

b. Modifications:

- i. Exact 100-mer matches: made kmerlen = 100
Very few hits now, but all lie on the diagonal.
- ii. 60-mers matching (at a minimum) every *other* base:
- iii. 90-mers matching (at a minimum) every *third* base:
- iv. 120-mers matching (...) every *forth* base:

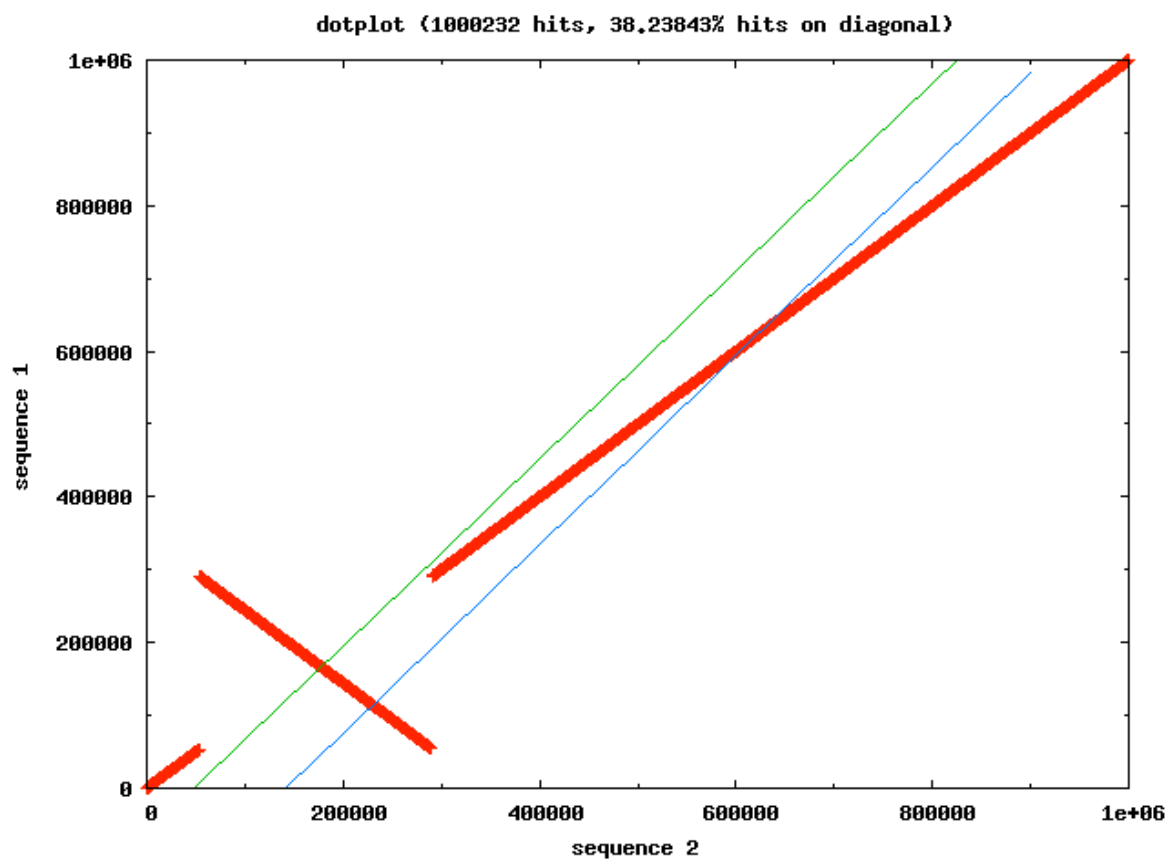
¹ This number comes from: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1197285/>

- v. 100-mers with *third* base mismatches allowed (the third base is the most mutable since it seldom changes the resulting amino acid, assuming we're in the correct reading frame).

| Modification | Description | Chart Thumbnail |
|---|--|---|
| i. Exact 100-mer matches: just made kmerlen = 100 | Very few points, but <i>all</i> lie along the diagonal. Largest cluster between 500 and 600 KBa. 1198 hits 100% on diagonal |  |
| ii. 60-mers matching every other base: made kmerlen = 60, hashed substrings of keys with characters indexed mod 2. | Far less noisy than the initial 30-mer case, but has more data points than exact 60-mer matching. Very distinct bands ~ every 200 KBa. 84748 hits 34.77% on diagonal |  |
| iii. 90-mers matching every third base same implementation as ii: kmerlen = 90, hashed substrings mod 3 | Highest hit count to diagonal point ratio so far. Very few points not on the diagonal, but many matches returned. 52,299 hits 91.61% on diagonal |  |

| Modification | Description | Chart Thumbnail |
|--|---|---|
| iv. 120-mers matching every fourth base same imp. as ii and iii: kmerlen = 120, hashed mod 4 | Similar to the 90-mer graph, with slightly more data points at the cost of specificity. Strangely, almost all non-diagonal matches occur around 800 Kba in humans. 56,788 hits 79.83% on diagonal | <p>dotplot (56788 hits, 79,837.7% hits on diagonal)</p> <p>The dotplot shows sequence 1 on the y-axis and sequence 2 on the x-axis, both ranging from 0 to 1e+06. Red dots represent matches. A green diagonal line and a blue line are shown. Most matches are concentrated along the diagonal, with some scattered points above and below it.</p> |
| v. 100-mers allowing third base mismatches similar to the previous methods, but this time hashing the concatenation of the first two bases in a codon for each key | This is fantastic. Here we've doubled our hit count from the original 100-mer match but maintained 100% specificity. 2772 hits 100% on diagonal | <p>dotplot (2772 hits, 100,000% hits on diagonal)</p> <p>The dotplot shows sequence 1 on the y-axis and sequence 2 on the x-axis, both ranging from 0 to 1e+06. Red dots represent matches. A green diagonal line and a blue line are shown. All matches are concentrated along the diagonal, with no scattered points.</p> |

- c. b.iii (90-mers matching every third base) is far more specific to the diagonal than b.ii and b.iv. This is likely because it maintains alignment with the reading frame and two thirds of the keys correspond to bases which seldom change in exons (the first and second base of a codon).
- d. Generally speaking, increasing the mer unit size increases specificity while decreasing sensitivity (see exact 30-mers vs. exact 100-mers). We can amend this somewhat by allowing for mismatches, in which case the best choices seem to be third base mismatches or every third base alignment. These selections manage to simultaneously increase sensitivity while maintaining (or even increasing) specificity.
- e. Here's the dotplot output for a comparison between human-hoxa-region and human-hoxa-region-modified after adding support for inversion detection to the code. I've set the mer-length to 1000:



3. Motif finding using evolutionary signatures

```
import re
from collections import Counter
from operator import itemgetter

def find_motifs():
    inter_data = ""
    motifs = Counter()
    conserved_motifs = Counter()
    conservation_ratios = {}

    with open("genes/allinter") as f:
        inter_data = f.read()

    motifs.update([inter_data[i:(i+6)] for i in range(len(inter_data)-6)])

    print "Total number of hexamers: {0}".format(len(motifs))
    print "50 most frequent motifs:"
    print motifs.most_common(50)

    with open("genes/allintercons") as f:
        cons_data = f.read()
        # use finditer
        for match in re.finditer('\s\{6\}\s', cons_data):
            motif = inter_data[match.start(1):match.end(1)]
            conserved_motifs.update([motif])

    for motif in conserved_motifs:
        conservation_ratios[motif] = float(conserved_motifs[motif]) / motifs[motif]

    print "Total number of conserved hexamers: {0}".format(
        sum(conserved_motifs.values()))
    print "50 most *frequent* conserved motifs:"
    print conserved_motifs.most_common(50)

    print "Top 50 most *conserved* motifs (conserved / total):"
    print sorted(conservation_ratios.items(),
        key=itemgetter(1), reverse=True)[:50]
```

And here's the output from running the program on the provided yeast data:

Total number of hexamers: 21400

50 most frequent motifs:

```
[('-----', 61588), ('AAAAAA', 16743), ('TTTTTT', 16354), ('TATATA',
8616), ('ATATAT', 8255), ('AAAAAT', 6053), ('ATTTTT', 5911), ('GAAAAA',
5528), ('TTTTTC', 5202), ('AAAATA', 5106), ('TAAAAA', 5001), ('AATAAA',
4990), ('ATAAAA', 4981), ('TATTTT', 4972), ('AAGAAA', 4953), ('AAATAA',
4876), ('CTTTTT', 4871), ('AGAAAA', 4808), ('TTTCTT', 4770), ('TTATTT',
4665), ('AAAAAG', 4663), ('TTTTTA', 4647), ('TTTTAT', 4641), ('TTTATT',
4637), ('TTTTCT', 4592), ('AAAGAA', 4522), ('ATAATA', 4499), ('TTCTTT',
4428), ('ATATAA', 4290), ('TATTAT', 4288), ('AATATA', 4286), ('TCTTTT',
4261), ('AAATAT', 4257), ('ATATTT', 4252), ('AAAAGA', 4208), ('TTATAT',
4176), ('AATTTT', 4166), ('TATATT', 4141), ('AAAATT', 4077), ('AATAAT',
3996), ('AATATT', 3990), ('TAAATA', 3863), ('ATTATT', 3854), ('AAATTT',
3826), ('TATAAA', 3818), ('TAATAA', 3758), ('A-----', 3752), ('ATATTA',
3727), ('TATTTA', 3720), ('TAATAT', 3706)]
```

Total number of conserved hexamers: 6965

50 most *frequent* conserved motifs:

```
[('TTTTTT', 188), ('AAAAAA', 178), ('TATATA', 114), ('-----', 111),
('GAAAAA', 53), ('TAAAAA', 35), ('ATATAT', 34), ('ATTTTT', 33), ('CTTTTT',
33), ('TTTATA', 32), ('TTTTTC', 31), ('AATAAA', 28), ('TAAATA', 25),
('AAGAAA', 22), ('TATAAA', 21), ('TTTATT', 20), ('TTTCTT', 19), ('TATTTT',
19), ('TTCTTT', 19), ('ACCCGG', 19), ('AGAAAA', 19), ('AAATAA', 19),
('TTTTCA', 19), ('ATAAAA', 18), ('TCTTTT', 17), ('TATGTA', 17), ('TATTTA',
17), ('TATACA', 16), ('TAAAAA', 16), ('AAAAAG', 16), ('ACTTTT', 16),
('ATAATA', 15), ('TACATA', 15), ('AAAAAT', 15), ('GGAAAA', 15), ('AAAGAA',
15), ('TTTTTA', 15), ('GTTTTT', 14), ('TTTTAT', 14), ('TTTTCT', 14),
('TGTTTT', 13), ('TGTATA', 13), ('AATATA', 13), ('CACCCA', 12), ('ATTGTT',
12), ('CCTTTT', 12), ('TATATT', 12), ('AAACAA', 12), ('CAAAAA', 12),
('TTATTT', 12)]
```

Top 50 most *conserved* motifs (conserved / total):

```
[('ACCCGG', 0.06666666666666667), ('ACGCGT', 0.03594771241830065),
('CAGGGG', 0.03468208092485549), ('CACGTG', 0.03206997084548105),
('CCGGGT', 0.03070175438596491), ('AGGCAC', 0.02967359050445104),
('GCCCCG', 0.02564102564102564), ('CACCCA', 0.023121387283236993),
('CGGGCC', 0.021897810218978103), ('CCAGCC', 0.020618556701030927),
('CGGGTA', 0.020512820512820513), ('TACCCG', 0.01991150442477876),
('CACCGG', 0.0196078431372549), ('CACGAG', 0.018604651162790697),
('GTGCCT', 0.01812688821752266), ('GGCCGG', 0.018072289156626505),
('CCCCGC', 0.016853932584269662), ('CTCGAG', 0.01639344262295082),
('ACGGCG', 0.016042780748663103), ('GGGTAA', 0.01585014409221902),
('CATCGG', 0.015151515151515152), ('CTCATC', 0.01509433962264151),
('TCGCGT', 0.014814814814814815), ('CGGACC', 0.014598540145985401),
('CGTGAG', 0.014285714285714285), ('GTGCAT', 0.014209591474245116),
('TCCGTG', 0.014035087719298246), ('ACCGGG', 0.013986013986013986),
('CCGATA', 0.013966480446927373), ('GACGCG', 0.013888888888888888),
('GACGAC', 0.013824884792626729), ('GCTCGT', 0.013761467889908258),
('GAGCCG', 0.013636363636363636), ('GGCATC', 0.013333333333333334),
('TTACCC', 0.013237063778580024), ('TATATA', 0.013231197771587743),
('TAGCTA', 0.013182674199623353), ('TGCGGT', 0.013071895424836602),
('CAACGG', 0.013029315960912053), ('GTCGAC', 0.012903225806451613),
('ACTCCG', 0.012658227848101266), ('TGGGTG', 0.012371134020618556),
('CGCGCA', 0.012345679012345678), ('AGGCGC', 0.012295081967213115),
('GGTGGC', 0.012096774193548387), ('AAGGGG', 0.012048192771084338),
('CGGAGC', 0.012048192771084338), ('CTGGGG', 0.011764705882352941),
('GGGCTG', 0.011764705882352941), ('TTTTTT', 0.011495658554482084)]
```


Going through the data, we see that the most frequent motifs consist almost exclusively of combinations of A and T, with only 5 or so exceptions present in the top 50, and even these only contain one non-A/T base. The hexamers with the highest concentration ratios, on the other hand, contain a more balanced overall ratio of bases. In the top 50 most conserved bases, the ratio breakdown is: {'G': 111, 'C': 98, 'A': 46, 'T': 45}. So Gs and Cs actually turn out to be the majority constituents.

I'm guessing that the most *conserved* motifs are more likely the biologically significant ones. All of the A and T repeats are likely the result of historic retrotransposon activity.

In my lists, I recognize two motifs. TATAAA is one of the 50 most frequent motifs, and this hexamer generally corresponds to the promoter region of a gene (TATA box). This may not be the hexamer's function in this case, though, since we're looking at intergenic DNA. Of the known motifs provided, I've only found one in common: **SWI6** (ACGCGT). This is the second most conserved motif from the data. According to the table of known functions, Swi6 is involved in the cell cycle. I also notice that 6 of the 7 bases for **REB1** are among the top 50 conserved (TTACCC). Another, **CCGATA**, contains Met4,28, which is apparently involved in starvation response. **CACGTG**, the inner bit of **CBF1** is the 5th most conserved hexamer: CBF1 is also somehow involved in the cell cycle. And I'm sure there are a few more that I missed.

4. See [SherbondyEthan_Profile.pdf/doc](#)
5. See [scribing_notes.html](#)