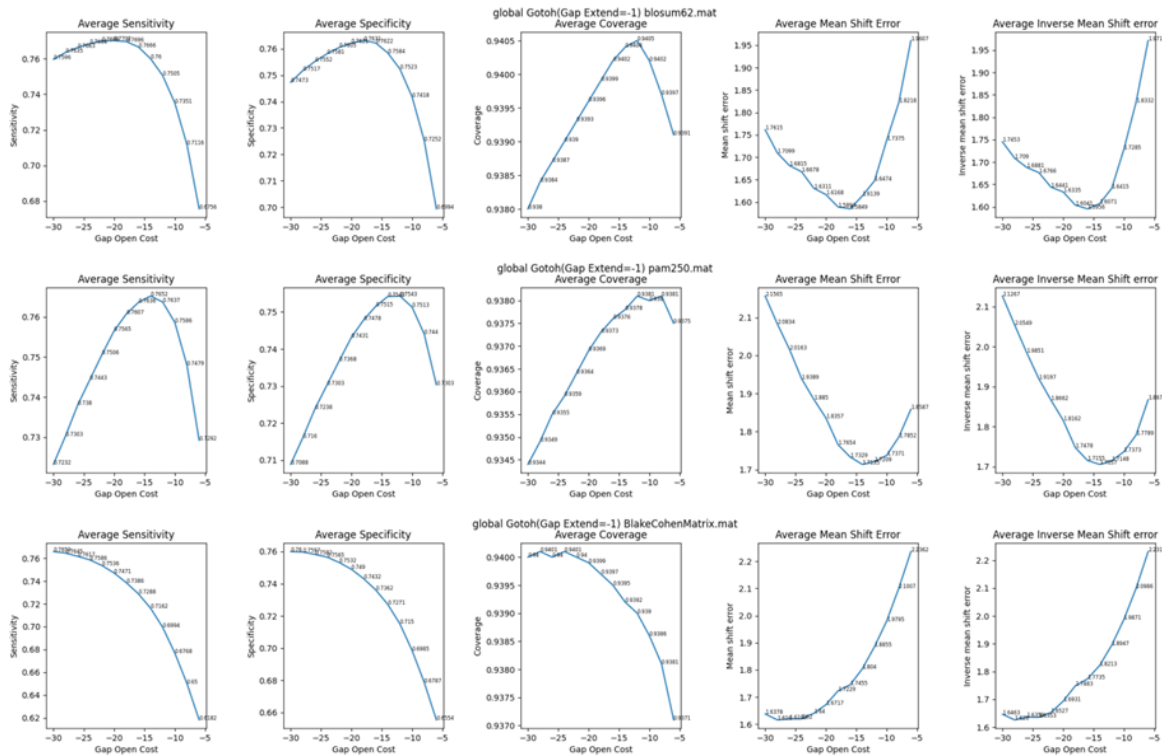
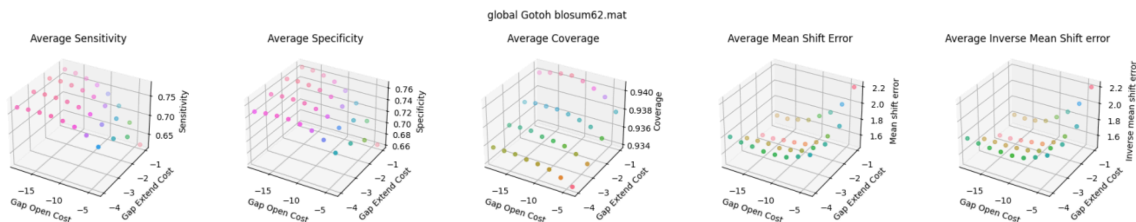
Correlations of validation metrics

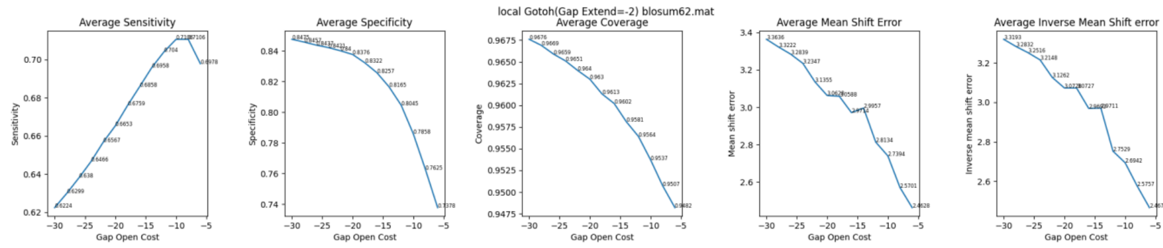
We validated our algorithms against the reference alignments from the **HOMSTRAD** database. The graphs offer valuable insights into **optimal parameter combinations** for our algorithms to achieve peak performance. Given that both reference and candidate alignments have the same length, global alignments are most suitable for validation in our scenario. These graphs can guide parameter tuning to meet specific needs. When using substitution matrices such as PAM250, BLOSUM62, and Dayhoff and gap extend costs are set to -1, opting for gap open costs around -15 proves most advantageous across all validation metrics.



The **lowest Avg. Mean Shift Error** can be attained by using the global Gotoh alignment and using the BLOSUM62 and setting the gap-open penalty to -15 and the gap-extend cost to -2.



Local Gotoh alignments reveal a distinct trend where **specificity and sensitivity diverge**.



Further observations include:

- **Global Gotoh** algorithms deliver the **best validation results** given our reference database and the general advantage of the Gotoh algorithm over the simple nw and sw algorithms.
- Gotoh Local and Gotoh Freeshift alignments generally excel at lower gap costs, except for BlakeCohen, which is an outlier, performing well even at higher gap costs.
- Across all alignment types and substitution matrices, there's a consistent **correlation between specificity and sensitivity, as well as between average mean shift error and average inverse mean shift error** (the exception to this trend is Local Gotoh).
- It's important to note that these examples may not apply to all scenarios, and each input parameter can significantly impact the alignment quality.

Benchmark: Dynamic vs Recursive Approach

Before we started implementing our version of the 6 dynamic programming algorithms, we investigated the difference between the dynamic approach and the recursive approach to calculating A global alignment score. Our initial expectations going into our investigation were that the dynamic algorithms would greatly outperform the recursive approach at higher sequence lengths. We constructed two basic implementations of the two algorithms with constant values for a match score, gap cost, and a non match score. The data we collected from our tests further reinforced our initial expectations.

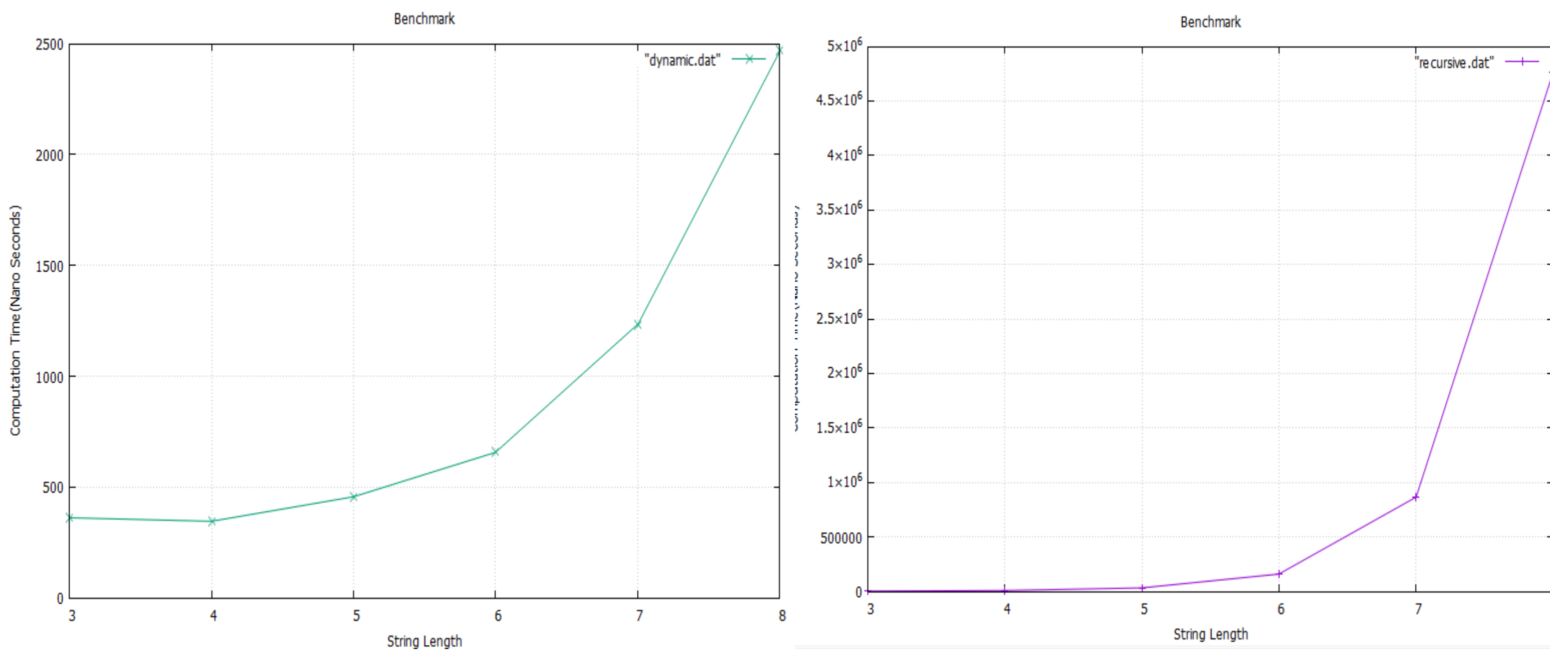


Figure 1.1 Dynamic program computation time (left) and recursive program computation time (right): both graphs were made using the same set of 10,000 randomly generated strings of n-lenght (x-axis) and were measured on average computation time in nanoseconds (y-axis)dz

In both graphs, a relatively similar shape arises, with the recursive program having a much higher rate of growth in comparison to the dynamic graph. By a string length of 8 the recursive function had an average computation time of 4.57×10^6 nanosecond while the dynamic algorithm was averaging 2337 nanoseconds. This inefficacy in the recursive function can be mainly attributed to inability to store values of previously calculated scores, having to recalculate the value for the same combination of two prefixes multiple times. This drawback is resolved in the dynamic approach with the addition of the dynamic matrix, allowing the program to

systematically calculate an alignment score of two sequences using previously calculated prefixes. The dynamic matrix in the dynamic method allows it to reach a computation time of $O(n^2)$, majorly surpassing the recursive method's $O(3^n)$. While this is the recursive function's main flaw, it can be rectified with something like a hash map, allowing for the computation time to near $O(n^2)$ by longer sequences. A shortcoming that can't be solved however, is a recursive function's intrinsic limitation in stack space. By particularly long sequences, the recursive function will have a stack overflow error before reaching the base case for the first time.