

## A HYPERPARAMETER SETTINGS

For all models, we used the following hyperparameter settings:

- top\_k=50
- top\_p=0.95
- temperature=0.7
- early\_stopping=True
- num\_return\_sequences=1
- do\_sample=True

## B FUNCTION STATISTICS

Below are our distributions of comment lines, token lengths, and the number of parameters per function across our entire dataset. Generally, our number of parameters and comments are left-skewed, while our prompt lengths, while left-skewed, had a much more even distribution.

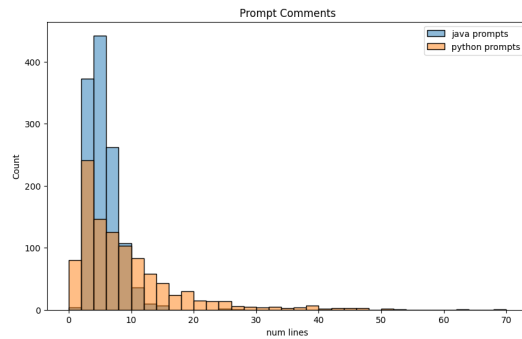


Figure 4: Comment lines per Function Distribution

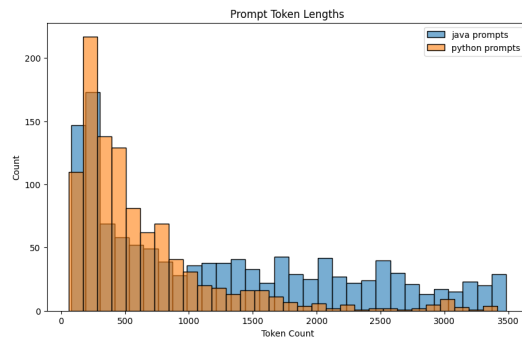


Figure 5: Prompt Token Length Distribution

Below is the distribution of the number of output tokens, averaged across either Python or Java. The vast concentration of the generations is concentrated at 0-250, as those are the minimum reserved tokens for generations if the prompts are too large. Across each language, we utilized nine models to generate outputs.

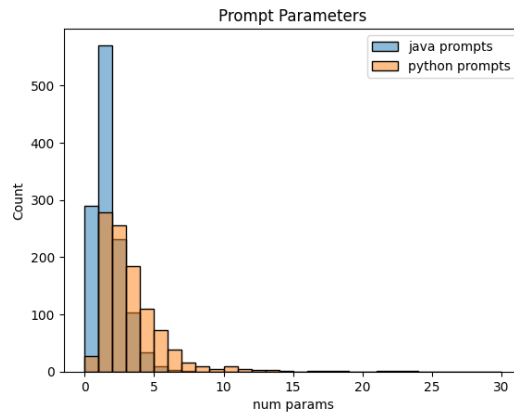
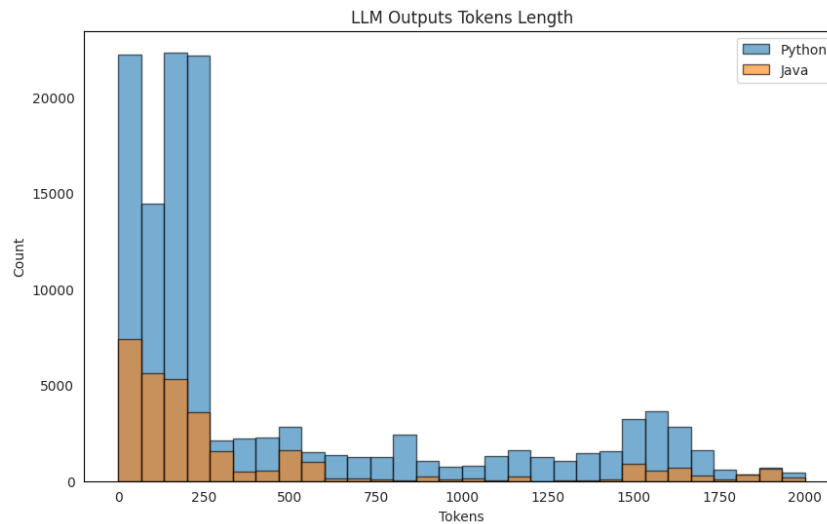


Figure 6: Parameters per Function Distribution

Figure 7: The distribution of the number of models' output tokens.



## C PYTHON EXAMPLES

### C.1 PYTHON EXAMPLE 1

#### C.1.1 FULL PROMPT

```

1 #This is in python
2 #Write a function called "unpipe_name" that takes in a string parameter
  called "name". The function takes a string containing multiple gene
  names separated by pipes, and returns a single gene name devoid of
  duplicates and pipe characters. The function checks if there are any
  duplicates, and removes any meaningless target names. If there are
  multiple gene names present, the function takes the longest name as
  the final name. If there are any ambiguous gene names, the function
  logs a warning and selects the longest name as the final name. The
  function should also import "cnvlib.params".
3 #
4 #def unpipe_name(name):
5 #
6 #Here are the imports:
7 #import numpy as np

```

```

8 #import logging
9 #from . import params
10 #from skgenome import tabio
11 #Here are the global variables:
12 #MIN_REF_COVERAGE = -5.0
13 #MAX_REF_SPREAD = 1.0
14 #NULL_LOG2_COVERAGE = -20.0
15 #GC_MIN_FRACTION = 0.3
16 #GC_MAX_FRACTION = 0.7
17 #INSERT_SIZE = 250
18 #IGNORE_GENE_NAMES = '-', '.', 'CGH'
19 #ANTITARGET_NAME = 'Antitarget'
20 #ANTITARGET_ALIASES = ANTITARGET_NAME, 'Background'
21 #Here are the class declarations:
22 #Here are the additional function declarations:
23 #def do_import_picard(fname, too_many_no_coverage):
24 #    summary: Reads a file in 'picardhs' format, processes the data, and
25 #    returns a modified dataframe.
26 #    param: fname (string) - the file name/path to be read in 'picardhs'
27 #    format.
28 #    param: too_many_no_coverage (int) - if the number of bins with no
29 #    coverage is greater than this value, a warning message is logged.
30 #    Default is 100.
31 #    return: garr (pandas dataframe) - a modified dataframe with added
32 #    columns 'gene' and 'log2' based on the original dataframe read from
33 #    the input file.
34 #def unpipe_name(name):
35 #    summary: Remove duplicate gene names and pipe characters from a given
36 #    string.
37 #    param: name (str) - the string containing the gene names.
38 #    return: new_name (str) - the cleaned gene name string.
39 #def do_import_theta(segarr, theta_results_fname, ploidy):
40 #    summary: A function for importing theta results and estimating copy
41 #    number and log2 ratios of segments.
42 #    param: segarr (numpy array) - array of segments
43 #    param: theta_results_fname (str) - name of theta results file
44 #    param: ploidy (int) - ploidy of genome (default is 2)
45 #    return: generator of numpy arrays - array of segments with estimated
46 #    copy number and log2 ratios.
47 #def parse_theta_results(fname):
48 #    summary: Parses THetA results into a data structure with NLL, mu, C,
49 #    and p* columns.
50 #    param: fname (str) - name of the file to parse the results from
51 #    return: (dict) - a dictionary containing the NLL, mu_normal,
52 #    mu_tumors, C, and p* values
53 #Here are the comments and the specs:
54 #Write a function called "unpipe_name" that takes in a string parameter
55 #called "name". The function takes a string containing multiple gene
56 #names separated by pipes, and returns a single gene name devoid of
57 #duplicates and pipe characters. The function checks if there are any
58 #duplicates, and removes any meaningless target names. If there are
59 #multiple gene names present, the function takes the longest name as
60 #the final name. If there are any ambiguous gene names, the function
61 #logs a warning and selects the longest name as the final name. The
62 #function should also import "cnvlib.params".
63 #def unpipe_name(name):

```

### C.1.2 SMALL PROMPT

```

1 Write a function with the following specs:
2 --specs begin here--
3 #Write a function called "unpipe_name" that takes in a string parameter
4 #called "name". The function takes a string containing multiple gene
5 #names separated by pipes, and returns a single gene name devoid of

```

```

duplicates and pipe characters. The function checks if there are any
duplicates, and removes any meaningless target names. If there are
multiple gene names present, the function takes the longest name as
the final name. If there are any ambiguous gene names, the function
logs a warning and selects the longest name as the final name. The
function should also import "cnvlib.params".
4 param: name (str) - the string containing the gene names.
5 return: new_name (str) - the cleaned gene name string.
6 --specs end here--
7 Note the function will be embedded in the following context
8 --context begins here--
9 import random
10 import hashlib
11 import numpy as np
12 import skimage
13 import skimage.measure
14 import scipy.ndimage
15 import os
16 import logging
17 from functools import wraps
18 from scipy import stats
19 import sys
20 import math
21 IGNORE_GENE_NAMES = ("-", ".", "CGH")
22 <<insert solution here>>
23 def main():
24     string1 = <|string|>
25     string2 = 'CGH'
26     name=f'{string1}|{string2}'
27     print(unpipe_name(name))
28 // context continues
29 --context ends here--
30 Make sure to only generate the function and not any of the context. Make
    sure you are generating valid, runnable code. Begin your solution
    with:
31 def unpipe_name(name):
32 MAKE SURE TO INDENT THE BODY OF YOUR FUNCTION BY A TAB

```

### C.1.3 NO COMMENT PROMPT

```

1 This is in python
2 Write a function called "unpipe_name" that takes in a string parameter
    called "name". The function takes a string containing multiple gene
    names separated by pipes, and returns a single gene name devoid of
    duplicates and pipe characters. The function checks if there are any
    duplicates, and removes any meaningless target names. If there are
    multiple gene names present, the function takes the longest name as
    the final name. If there are any ambiguous gene names, the function
    logs a warning and selects the longest name as the final name. The
    function should also import "cnvlib.params".
3
4 def unpipe_name(name):
5
6 Here are the imports:
7 import numpy as np
8 import logging
9 from . import params
10 from skgenome import tabio
11 Here are the global variables:
12 MIN_REF_COVERAGE = -5.0
13 MAX_REF_SPREAD = 1.0
14 NULL_LOG2_COVERAGE = -20.0
15 GC_MIN_FRACTION = 0.3
16 GC_MAX_FRACTION = 0.7

```

```

17 INSERT_SIZE = 250
18 IGNORE_GENE_NAMES = '-', '.', 'CGH'
19 ANTITARGET_NAME = 'Antitarget'
20 ANTITARGET_ALIASES = ANTITARGET_NAME, 'Background'
21 Here are the class declarations:
22 Here are the additional function declarations:
23 def do_import_picard(fname, too_many_no_coverage):
24     summary: Reads a file in 'picardhs' format, processes the data, and
25     returns a modified dataframe.
26     param: fname (string) - the file name/path to be read in 'picardhs'
27     format.
28     param: too_many_no_coverage (int) - if the number of bins with no
29     coverage is greater than this value, a warning message is logged.
30     Default is 100.
31     return: garr (pandas dataframe) - a modified dataframe with added
32     columns 'gene' and 'log2' based on the original dataframe read from
33     the input file.
34 def unpipe_name(name):
35     summary: Remove duplicate gene names and pipe characters from a given
36     string.
37     param: name (str) - the string containing the gene names.
38     return: new_name (str) - the cleaned gene name string.
39 def do_import_theta(segarr, theta_results_fname, ploidy):
40     summary: A function for importing theta results and estimating copy
41     number and log2 ratios of segments.
42     param: segarr (numpy array) - array of segments
43     param: theta_results_fname (str) - name of theta results file
44     param: ploidy (int) - ploidy of genome (default is 2)
45     return: generator of numpy arrays - array of segments with estimated
46     copy number and log2 ratios.
47 def parse_theta_results(fname):
48     summary: Parses THetA results into a data structure with NLL, mu, C,
49     and p* columns.
50     param: fname (str) - name of the file to parse the results from
51     return: (dict) - a dictionary containing the NLL, mu_normal,
52     mu_tumors, C, and p* values
53 Here are the comments and the specs:
54 Write a function called "unpipe_name" that takes in a string parameter
55 called "name". The function takes a string containing multiple gene
56 names separated by pipes, and returns a single gene name devoid of
57 duplicates and pipe characters. The function checks if there are any
58 duplicates, and removes any meaningless target names. If there are
59 multiple gene names present, the function takes the longest name as
60 the final name. If there are any ambiguous gene names, the function
61 logs a warning and selects the longest name as the final name. The
62 function should also import "cnvlib.params".
63 def unpipe_name(name):

```

#### C.1.4 REFERENCE CODE

```

1 def unpipe_name(name):
2     """Fix the duplicated gene names Picard spits out.
3
4     Return a string containing the single gene name, sans duplications
5     and pipe
6     characters.
7
8     Picard CalculateHsMetrics combines the labels of overlapping
9     intervals
10    by joining all labels with '|', e.g. 'BRAF|BRAF' -- no two distinct
11    targeted genes actually overlap, though, so these dupes are redundant
12    .
13    Meaningless target names are dropped, e.g. 'CGH|FOO|-' resolves as '
14    FOO'.
```

```

11     In case of ambiguity, the longest name is taken, e.g. "TERT|TERT
    Promoter"
12     resolves as "TERT Promoter".
13     """
14     if '|' not in name:
15         return name
16     gene_names = set(name.split('|'))
17     if len(gene_names) == 1:
18         return gene_names.pop()
19     cleaned_names = gene_names.difference(IGNORE_GENE_NAMES)
20     if cleaned_names:
21         gene_names = cleaned_names
22     new_name = sorted(gene_names, key=len, reverse=True)[0]
23     if len(gene_names) > 1:
24         logging.warning('WARNING: Ambiguous gene name %r; using %r', name
25             ,
26             new_name)
27     return new_name

```

### C.1.5 ANNOTATION PROMPTS

```

1 #This is in python
2 #Write a function called "unpipe_name" that takes in a string parameter
    called "name". The function takes a string containing multiple gene
    names separated by pipes, and returns a single gene name devoid of
    duplicates and pipe characters. The function checks if there are any
    duplicates, and removes any meaningless target names. If there are
    multiple gene names present, the function takes the longest name as
    the final name. If there are any ambiguous gene names, the function
    logs a warning and selects the longest name as the final name. The
    function should also import "cnvlib.params".
3 #
4 #def unpipe_name(name):

```

### C.1.6 CONTEXT

```

1 import random
2 import hashlib
3 import numpy as np
4 import skimage
5 import skimage.measure
6 import scipy.ndimage
7 import os
8 import logging
9 from functools import wraps
10 from scipy import stats
11 import sys
12 import math
13 IGNORE_GENE_NAMES = ("-.", ".", "CGH")
14 <<insert solution here>>
15 def main():
16     string1 = <|string|>
17     string2 = 'CGH'
18     name=f'{string1}|{string2}'
19     print(unpipe_name(name))
20 if __name__ == "__main__":
21     main()

```

## C.2 PYTHON EXAMPLE 2

### C.2.1 FULL PROMPT

```

1 #This is in python
2 #write a function called "UnifyLevels" that takes three parameters:
   baseLevel (a constant array of integers representing the base levels)
   , addonLevel (a constant array of integers representing the
   additional levels), and windowSize (an integer representing the
   window size). The function should merge the two lists of breakpoints,
   but drop addonLevel values that are within windowSize of baseLevel
   values. The function should return an array of integers called
   joinedLevel. If addonLevel is an empty array, the function should
   return baseLevel. The output should be sorted and of dtype=np.int_.
3 #
4 #def UnifyLevels(baseLevel, addonLevel, windowSize):
5 #
6 #Here are the imports:
7 #from scipy import stats
8 #import math
9 #import numpy as np
10 #import logging
11 #import pandas as pd
12 #Here are the global variables:
13 #Here are the class declarations:
14 #Here are the additional function declarations:
15 #def segment_haar(cnarr, fdr_q):
16 #   summary: Segment CNVkit data using HaarSeg algorithm
17 #   param: cnarr (CopyNumArray) - binned, normalized copy ratios
18 #   param: fdr_q (float) - false discovery rate q-value
19 #   return: segarr (CopyNumArray) - CBS data table as a CNVkit object
20 #def one_chrom(cnarr, fdr_q, chrom):
21 #   summary: This function segments copy number data for a single
   chromosome using the HaarSeg algorithm.
22 #   param: cnarr (pandas.DataFrame) - a dataframe with columns '
   chromosome', 'start', 'end', 'log2', and 'probes'.
23 #   param: fdr_q (float) - the false discovery rate threshold for
   segmenting the copy number data.
24 #   param: chrom (str) - the chromosome to segment.
25 #   return: table (pandas.DataFrame) - a dataframe with columns '
   chromosome', 'start', 'end', 'log2', 'gene', and 'probes',
   representing the segmented copy number data.
26 #def variants_in_segment(varr, segment, fdr_q):
27 #   summary: Generates a table of variant segments based on allele
   frequencies
28 #   param: varr (object) - variant data
29 #   param: segment (object) - genomic segment data
30 #   param: fdr_q (float) - false discovery rate threshold
31 #   return: table (object) - pandas DataFrame with segmented data
32 #def haarSeg(I, breaksFdrQ, W, rawI, haarStartLevel, haarEndLevel):
33 #   summary: Perform segmentation on a 1D array of log-ratio values
   according to the HaarSeg algorithm.
34 #   param: I (array) - A 1D array of log-ratio values, sorted according
   to their genomic location.
35 #   param: W (array) - Weight matrix, corresponding to quality of
   measurement, with values :math:'1/(\sigma^2)'. Must have the same
   size as I.
36 #   param: rawI (array) - The minimum between the raw test-sample and
   control-sample coverages (before applying log ratio, but after any
   background reduction and/or normalization). These raw red / green
   measurments are used to detect low-value probes, which are more
   sensitive to noise. Used for the non-stationary variance compensation
   . Must have the same size as I.
37 #   param: breaksFdrQ (float) - The FDR q parameter. This value should
   lie between 0 and 0.5.
38 #   param: haarStartLevel (int) - The detail subband from which we start
   to detect peaks.
39 #   param: haarEndLevel (int) - The detail subband until which we use to
   detect peaks.

```

```

40 # return: dict - Returns a dictionary containing the start and end
    points of each segment and the mean value of each segment.
41 #def FDRThres(x, q, stdev):
42 # summary: Calculates the False Discovery Rate (FDR) threshold.
43 # param: x (unknown type) - array of values.
44 # param: q (unknown type) - a floating-point number.
45 # param: stdev (unknown type) - a floating-point number representing
    the standard deviation.
46 # return: T (unknown type) - a floating-point number representing the
    FDR threshold.
47 #def SegmentByPeaks(data, peaks, weights):
48 # summary: Average the values of the probes within each segment.
49 # param: data (array) - the probe array values
50 # param: peaks (array) - Positions of copy number breakpoints in the
    original array
51 # param: weights (None or array) - optional array of weights of same
    length as the data array
52 # return: segs (array) - array of segment values obtained by averaging
    the values of the probes within each segment.
53 #def HaarConv(signal, weight, stepHalfSize):
54 # summary: Convolve haar wavelet function with a signal, applying
    circular padding.
55 # param: signal (array of floats) - signal to be convolved.
56 # param: weight (array of floats) - optional weights for the steps of
    the convolution.
57 # param: stepHalfSize (int) - half size of the step to be used in the
    convolution.
58 # return: array (array of floats) - of floats, representing the
    convolved signal.
59 #def FindLocalPeaks(signal):
60 # summary: Finds local maxima on positive values, local minima on
    negative values.
61 # param: signal (const array of floats): an array of floating point
    numbers
62 # return: peakLoc (array of ints): Locations of extrema in 'signal'
63 #def UnifyLevels(baseLevel, addonLevel, windowSize):
64 # summary: Merge two lists of breakpoints and drop addonLevel values
    that are too close to baseLevel values.
65 # param: baseLevel (const array of ints) - a list of integers
    representing the base level.
66 # param: addonLevel (const array of ints) - a list of integers
    representing the addon level.
67 # param: windowSize (int) - an integer representing the maximum
    distance between addonLevel and baseLevel values.
68 # return: joinedLevel (array of ints) - a sorted array of integers
    representing the merged list of breakpoints.
69 #def PulseConv(signal, pulseSize):
70 # summary: Convolve a pulse function with a signal applying circular
    padding to the signal for non-stationary variance compensation.
71 # param: signal (const array of floats) - the signal to be convolved.
72 # param: pulseSize (int) - the size of the pulse function.
73 # return: array of floats - the convolved signal.
74 #def AdjustBreaks(signal, peakLoc):
75 # summary: Improve localization of breaks in a signal by adjusting peak
    locations.
76 # param: signal (const array of floats) - the signal to adjust the peak
    locations in.
77 # param: peakLoc (const array of ints) - the current peak locations in
    the signal.
78 # return: newPeakLoc (array of ints) - the adjusted peak locations in
    the signal.
79 #def table2coords(seg_table):
80 # summary: Returns x and y arrays for plotting with the help of the
    input segment Table

```



```

81 # param: seg_table (list) - a list of tuples containing start, size,
    and value.
82 # return: x (list) - a list of x-coordinates for plotting.
83 # return: y (list) - a list of y-coordinates for plotting.
84 #Here are the comments and the specs:
85 #write a function called "UnifyLevels" that takes three parameters:
    baseLevel (a constant array of integers representing the base levels)
    , addonLevel (a constant array of integers representing the
    additional levels), and windowSize (an integer representing the
    window size). The function should merge the two lists of breakpoints,
    but drop addonLevel values that are within windowSize of baseLevel
    values. The function should return an array of integers called
    joinedLevel. If addonLevel is an empty array, the function should
    return baseLevel. The output should be sorted and of dtype=np.int_.
86 #def UnifyLevels(baseLevel, addonLevel, windowSize):

```

### C.2.2 SMALL PROMPT

```

1 Write a function with the following specs:
2 --specs begin here--
3 #write a function called "UnifyLevels" that takes three parameters:
    baseLevel (a constant array of integers representing the base levels)
    , addonLevel (a constant array of integers representing the
    additional levels), and windowSize (an integer representing the
    window size). The function should merge the two lists of breakpoints,
    but drop addonLevel values that are within windowSize of baseLevel
    values. The function should return an array of integers called
    joinedLevel. If addonLevel is an empty array, the function should
    return baseLevel. The output should be sorted and of dtype=np.int_.
4 param: baseLevel (const array of ints) - a list of integers representing
    the base level.
5 param: addonLevel (const array of ints) - a list of integers representing
    the addon level.
6 param: windowSize (int) - an integer representing the maximum distance
    between addonLevel and baseLevel values.
7 return: joinedLevel (array of ints) - a sorted array of integers
    representing the merged list of breakpoints.
8 --specs end here--
9 Note the function will be embedded in the following context
10 --context begins here--
11 import random
12 import hashlib
13 import numpy as np
14 import skimage
15 import skimage.measure
16 import scipy.ndimage
17 import os
18 import logging
19 from functools import wraps
20 from scipy import stats
21 import sys
22 import math
23 <<insert solution here>>
24 def main():
25     np.random.seed(<|int;range=0,1000|>)
26     baseLevel = np.random.randint(20, size=(10))
27     np.random.seed(<|int;range=0,1000|>)
28     addonLevel = np.random.randint(20, size=(10))
29     print(UnifyLevels(baseLevel, addonLevel, 3))
30 if __name__ == "__main__":
31 // context continues
32 --context ends here--

```

```

33 Make sure to only generate the function and not any of the context. Make
    sure you are generating valid, runnable code. Begin your solution
    with:
34 def UnifyLevels(baseLevel, addonLevel, windowSize):
35 MAKE SURE TO INDENT THE BODY OF YOUR FUNCTION BY A TAB

```

### C.2.3 NO COMMENT PROMPT

```

1 This is in python
2 write a function called "UnifyLevels" that takes three parameters:
    baseLevel (a constant array of integers representing the base levels)
    , addonLevel (a constant array of integers representing the
    additional levels), and windowSize (an integer representing the
    window size). The function should merge the two lists of breakpoints,
    but drop addonLevel values that are within windowSize of baseLevel
    values. The function should return an array of integers called
    joinedLevel. If addonLevel is an empty array, the function should
    return baseLevel. The output should be sorted and of dtype=np.int_.
3
4 def UnifyLevels(baseLevel, addonLevel, windowSize):
5
6 Here are the imports:
7 from scipy import stats
8 import math
9 import numpy as np
10 import logging
11 import pandas as pd
12 Here are the global variables:
13 Here are the class declarations:
14 Here are the additional function declarations:
15 def segment_haar(cnarr, fdr_q):
16     summary: Segment CNVkit data using HaarSeg algorithm
17     param: cnarr (CopyNumArray) - binned, normalized copy ratios
18     param: fdr_q (float) - false discovery rate q-value
19     return: segarr (CopyNumArray) - CBS data table as a CNVkit object
20 def one_chrom(cnarr, fdr_q, chrom):
21     summary: This function segments copy number data for a single
    chromosome using the HaarSeg algorithm.
22     param: cnarr (pandas.DataFrame) - a dataframe with columns '
    chromosome', 'start', 'end', 'log2', and 'probes'.
23     param: fdr_q (float) - the false discovery rate threshold for
    segmenting the copy number data.
24     param: chrom (str) - the chromosome to segment.
25     return: table (pandas.DataFrame) - a dataframe with columns '
    chromosome', 'start', 'end', 'log2', 'gene', and 'probes',
    representing the segmented copy number data.
26 def variants_in_segment(varr, segment, fdr_q):
27     summary: Generates a table of variant segments based on allele
    frequencies
28     param: varr (object) - variant data
29     param: segment (object) - genomic segment data
30     param: fdr_q (float) - false discovery rate threshold
31     return: table (object) - pandas DataFrame with segmented data
32 def haarSeg(I, breaksFdrQ, W, rawI, haarStartLevel, haarEndLevel):
33     summary: Perform segmentation on a 1D array of log-ratio values
    according to the HaarSeg algorithm.
34     param: I (array) - A 1D array of log-ratio values, sorted according
    to their genomic location.
35     param: W (array) - Weight matrix, corresponding to quality of
    measurement, with values :math:'1/(\sigma^2)'. Must have the same
    size as I.
36     param: rawI (array) - The minimum between the raw test-sample and
    control-sample coverages (before applying log ratio, but after any
    background reduction and/or normalization). These raw red / green

```

```

measurements are used to detect low-value probes, which are more
sensitive to noise. Used for the non-stationary variance compensation
. Must have the same size as I.
37 param: breaksFdrQ (float) - The FDR q parameter. This value should
lie between 0 and 0.5.
38 param: haarStartLevel (int) - The detail subband from which we start
to detect peaks.
39 param: haarEndLevel (int) - The detail subband until which we use to
detect peaks.
40 return: dict - Returns a dictionary containing the start and end
points of each segment and the mean value of each segment.
41 def FDRThres(x, q, stdev):
42     summary: Calculates the False Discovery Rate (FDR) threshold.
43     param: x (unknown type) - array of values.
44     param: q (unknown type) - a floating-point number.
45     param: stdev (unknown type) - a floating-point number representing
the standard deviation.
46     return: T (unknown type) - a floating-point number representing the
FDR threshold.
47 def SegmentByPeaks(data, peaks, weights):
48     summary: Average the values of the probes within each segment.
49     param: data (array) - the probe array values
50     param: peaks (array) - Positions of copy number breakpoints in the
original array
51     param: weights (None or array) - optional array of weights of same
length as the data array
52     return: segs (array) - array of segment values obtained by averaging
the values of the probes within each segment.
53 def HaarConv(signal, weight, stepHalfSize):
54     summary: Convolve haar wavelet function with a signal, applying
circular padding.
55     param: signal (array of floats) - signal to be convolved.
56     param: weight (array of floats) - optional weights for the steps of
the convolution.
57     param: stepHalfSize (int) - half size of the step to be used in the
convolution.
58     return: array (array of floats) - of floats, representing the
convolved signal.
59 def FindLocalPeaks(signal):
60     summary: Finds local maxima on positive values, local minima on
negative values.
61     param: signal (const array of floats): an array of floating point
numbers
62     return: peakLoc (array of ints): Locations of extrema in 'signal'
63 def UnifyLevels(baseLevel, addonLevel, windowSize):
64     summary: Merge two lists of breakpoints and drop addonLevel values
that are too close to baseLevel values.
65     param: baseLevel (const array of ints) - a list of integers
representing the base level.
66     param: addonLevel (const array of ints) - a list of integers
representing the addon level.
67     param: windowSize (int) - an integer representing the maximum
distance between addonLevel and baseLevel values.
68     return: joinedLevel (array of ints) - a sorted array of integers
representing the merged list of breakpoints.
69 def PulseConv(signal, pulseSize):
70     summary: Convolve a pulse function with a signal applying circular
padding to the signal for non-stationary variance compensation.
71     param: signal (const array of floats) - the signal to be convolved.
72     param: pulseSize (int) - the size of the pulse function.
73     return: array of floats - the convolved signal.
74 def AdjustBreaks(signal, peakLoc):
75     summary: Improve localization of breaks in a signal by adjusting peak
locations.

```

```

76     param: signal (const array of floats) - the signal to adjust the peak
       locations in.
77     param: peakLoc (const array of ints) - the current peak locations in
       the signal.
78     return: newPeakLoc (array of ints) - the adjusted peak locations in
       the signal.
79 def table2coords(seg_table):
80     summary: Returns x and y arrays for plotting with the help of the
       input segment Table
81     param: seg_table (list) - a list of tuples containing start, size,
       and value.
82     return: x (list) - a list of x-coordinates for plotting.
83     return: y (list) - a list of y-coordinates for plotting.
84 Here are the comments and the specs:
85 write a function called "UnifyLevels" that takes three parameters:
       baseLevel (a constant array of integers representing the base levels)
       , addonLevel (a constant array of integers representing the
       additional levels), and windowSize (an integer representing the
       window size). The function should merge the two lists of breakpoints,
       but drop addonLevel values that are within windowSize of baseLevel
       values. The function should return an array of integers called
       joinedLevel. If addonLevel is an empty array, the function should
       return baseLevel. The output should be sorted and of dtype=np.int_.
86 def UnifyLevels(baseLevel, addonLevel, windowSize):

```

#### C.2.4 REFERENCE CODE

```

1 def UnifyLevels(baseLevel, addonLevel, windowSize):
2     """Unify several decomposition levels.
3
4     Merge the two lists of breakpoints, but drop addonLevel values that
       are too
5     close to baseLevel values.
6
7     Parameters
8     -----
9     baseLevel : const array of ints
10    addonLevel : const array of ints
11    windowSize : int
12
13    Returns
14    -----
15    joinedLevel : array of ints
16
17    Source: HaarSeg.c
18    """
19    if not len(addonLevel):
20        return baseLevel
21    joinedLevel = []
22    addon_idx = 0
23    for base_elem in baseLevel:
24        while addon_idx < len(addonLevel):
25            addon_elem = addonLevel[addon_idx]
26            if addon_elem < base_elem - windowSize:
27                joinedLevel.append(addon_elem)
28                addon_idx += 1
29            elif base_elem - windowSize <= addon_elem <= base_elem +
       windowSize:
30                addon_idx += 1
31            else:
32                assert base_elem + windowSize < addon_elem
33                break
34        joinedLevel.append(base_elem)
35    last_pos = baseLevel[-1] + windowSize if len(baseLevel) else -1

```

```
36 while addon_idx < len(addonLevel) and addonLevel[addon_idx] <=
    last_pos:
37     addon_idx += 1
38     if addon_idx < len(addonLevel):
39         joinedLevel.extend(addonLevel[addon_idx:])
40     return np.array(sorted(joinedLevel), dtype=np.int_)
```

### C.2.5 CONTEXT

```
1 import random
2 import hashlib
3 import numpy as np
4 import skimage
5 import skimage.measure
6 import scipy.ndimage
7 import os
8 import logging
9 from functools import wraps
10 from scipy import stats
11 import sys
12 import math
13 <<insert solution here>>
14 def main():
15     np.random.seed(<|int;range=0,1000|>)
16     baseLevel = np.random.randint(20, size=(10))
17     np.random.seed(<|int;range=0,1000|>)
18     addonLevel = np.random.randint(20, size=(10))
19     print(UnifyLevels(baseLevel, addonLevel, 3))
20 if __name__ == "__main__":
21     main()
```

## D ROSALIND EXAMPLES

The Rosalind dataset consists of 253 problems from the bioinformatics training website [rosalind.info](http://rosalind.info). For each of the 253 problems that make up the Rosalind portion of BIOCORDER, we created both a prompt that we fed into the LLMs as well as a golden code solution that we either wrote ourselves or found on public GitHub repositories.

For each problem, to create the prompt, we first scraped the problem’s page on the Rosalind website to obtain a problem description. This problem description would then make up the first section of the prompt. For instance, for the Rosalind problem ba1a, here is the scraped problem description:

```

1 # This is the first problem in a collection of "code challenges" to
  accompany Bioinformatics Algorithms: An Active-Learning Approach by
  Phillip Compeau & Pavel Pevzner.
2 # A k-mer is a string of length k.
3 # We define Count(Text, Pattern) as the number of times that a k-mer
  Pattern appears as a substring of Text.
4 # For example,
5 #  $\text{Count}(\text{ACA}\color{green}\text{ACTAT}\color{black}\text{GCAT}\color{green}\text{ACTAT}\color{black}\text{CGGGA}\color{green}\text{ACTAT}\color{black}\text{CCT}, \color{green}\text{ACTAT}) = 3$ .
6 # We note that  $\text{Count}(\text{CG}\color{green}\text{ATATA}\color{black}\text{TCC}\color{green}\text{ATA}\color{black}\text{G})$ ,  $\color{green}\text{ATA}$  is equal to 3 (not 2) since we should account
  for overlapping occurrences of Pattern in Text.
7 # To compute Count(Text, Pattern), our plan is to "slide a window" down
  Text, checking whether each k-mer substring of Text matches Pattern.
  We will therefore refer to the k-mer starting at position i of Text
  as Text(i, k). Throughout this book, we will often use 0-based
  indexing, meaning that we count starting at 0 instead of 1. In this
  case, Text begins at position 0 and ends at position |Text| - 1 (|
  Text| denotes the number of symbols in Text). For example, if Text =
  GACCATACTG,
8 # then Text(4, 3) = ATA. Note that the last k-mer of Text begins at
  position |Text| - k, e.g., the last 3-mer of GACCATACTG starts at
  position 10 - 3 = 7. This discussion results in the following
  pseudocode for computing Count(Text, Pattern).
9 # PatternCount(Text, Pattern)
10 # count ← 0
11 # for i ← 0 to |Text| - |Pattern|
12 #   if Text(i, |Pattern|) = Pattern
13 #     count ← count + 1
14 # return count
15 # Implement PatternCount
16 # Given: {{DNA strings}} Text and Pattern.
17 # Return: Count(Text, Pattern).
18 # Sample Dataset
19 # GCGCG
20 # GCG
21 # Sample Output
22 # 2

```

From the sample problem description above you can see that a problem description generally consists of an explanation of the problem at hand, as well as a sample test case. The second section of the prompt consists of a custom block of text that prompts the LLM to generate its code to fit the context that generated Rosalind code will be run in. For instance, for the same Rosalind problem ba1a, here is the custom block of text that makes up the second section of the prompt:

```

1 #write the solve() function to solve the above problem
2 #Do NOT print anything
3 #Do NOT make comments
4 #Do NOT call the main() function.
5 #Use 4 spaces for tabs.
6 #input_str: a string

```

```

7 #return output: another string
8
9
10 def main():
11     with open("input.txt", "r") as f:
12         output = solve(f.read())
13         print(output)
14
15 # Write your solution here
16 # Begin with: def solve(input_str):

```

From the example above, you can see that the custom block of text consists of a prompt telling the LLM to generate a solve function as well as the context that the generated code will be run in, namely the main function that is included in the custom text. Each prompt consists of these two sections, i.e., the first section of the prompt will include a problem description consisting of an explanation of the problem as well as a sample test case, and the second section of the prompt will consist of custom prompting text that tells the LLM to generate a solve function and provides the context in which the Rosalind code will be run.

For each of the 253 problems that make up the Rosalind portion of BIOCORDER, we also either found online or wrote custom golden code solutions. Each golden code solution is meant to be run in the custom context, following the main and solve structure that is described in every Rosalind prompt. For instance, here is the golden code solution for the Rosalind problem ba1a:

```

1 # Compute the Number of Times a Pattern Appears in a Text
2
3 def substrings(text, size):
4     for i in range(len(text) - size + 1):
5         yield text[i : i + size]
6
7 def pattern_count(text, pattern):
8     return sum(pattern == x for x in substrings(text, len(pattern)))
9
10 def solve(input_str):
11     text, pattern = input_str.splitlines()
12
13     return pattern_count(text, pattern)

```

As you can see, a golden code solution consists of the solve function requested by the corresponding prompt, as well as any additional helper functions that may be called (possibly recursively) by the solve function. For more information on the Rosalind portion of the BIOCORDER dataset, three detailed examples of Rosalind prompts and golden code are provided below to further illustrate the differences between the code and the prompt.

#### D.1 EXAMPLE 1

The given input is a problem description in the field of bioinformatics, namely the Reverse Complement Problem. The problem is related to the DNA sequences, which are strings containing the characters 'A', 'T', 'G', and 'C'. Each of these characters represents a nucleotide in the DNA.

The problem requires model to find the reverse complement of a given DNA string. The reverse complement of a DNA string is obtained by replacing each nucleotide in the string with its complement ('A' with 'T', 'T' with 'A', 'C' with 'G', and 'G' with 'C'), and then reversing the entire string. For example, the reverse complement of "GTCA" is "TGAC."

The problem description includes a sample dataset (DNA string) and its corresponding output, which is its reverse complement. The problem also specifies the format for the solution: a Python function named 'solve', which takes a DNA string as an input and returns its reverse complement as output. The function should not print anything or include any comments.

```

1
2 Reverse Complement Problem:
3 In DNA strings, symbols 'A' and 'T' are complements of each other, as are
  'C' and 'G'.

```

```

4 Given a nucleotide p, we denote its complementary nucleotide as p. The
  reverse complement of a DNA string Pattern = p1â€¦pn is the string
  Pattern = pn â€¦ p1 formed by taking the complement of each
  nucleotide in Pattern, then reversing the resulting string.
5
6 For example, the reverse complement of Pattern = "GTCA" is Pattern = "
  TGAC".
7
8
9 Problem:
10 Find the reverse complement of a DNA string.
11 Given: A DNA string Pattern.
12 Return: Pattern, the reverse complement of Pattern.
13
14 Sample Dataset:
15 AAAACCCGGT
16
17 Sample Output:
18 ACCGGGTTTT
19
20 write the solve() function to solve the above problem.
21
22 Do NOT print anything
23 Do NOT make comments
24 Do NOT call the main() function.
25 Use 4 spaces for tabs.
26
27 input_str: a string
28 return output: another string
29
30 def main():
31     with open("input.txt", "r") as f:
32         output = solve(f.read())
33         print(output)
34
35
36 Write your solution here:
37 Begin with: def solve(input_str):

```

The given sample solution to the problem is written in Python. It defines two functions: 'revcomp' and 'main'. The 'revcomp' function computes the reverse complement of a DNA string by first reversing the string and then replacing each nucleotide with its complement using Python's 'str.maketrans' method. The 'main' function reads a DNA string from a file, computes its reverse complement using the 'revcomp' function, and then prints the result.

The ground-truth code is a solution to the problem presented in the 'Rosalind' platform, a platform that provides a collection of bioinformatics problems. This platform is known for helping students and researchers learn bioinformatics by solving the computational problems it presents.

```

1 Find the Reverse Complement of a String
2
3
4 def revcomp(seq):
5     return seq[::-1].translate(str.maketrans("ACGT", "TGCA"))
6
7
8 def main(file):
9     seq = open(file).read().splitlines()[0]
10    print(revcomp(seq))

```

## D.2 EXAMPLE 2

The input here consists of several parts.



The initial portion discusses the Hardy Weinberg Principle and Mendel's laws, which are used in genetics to predict how genes will distribute throughout a population over time. The Hardy Weinberg Principle assumes that a population is large and remains in genetic equilibrium, meaning the frequency of each allele (a version of a gene) remains constant across generations, under certain conditions.

The problem statement is presented: if we know the proportion of homozygous recessive individuals (those with two copies of the recessive allele) for each of several genetic factors in a population, can we predict the probability that a randomly selected individual carries at least one copy of the recessive allele for each factor? The task is to write a Python function named 'solve' to solve this problem.

A sample dataset is given, represented as an array of floating-point numbers, each of which indicates the proportion of homozygous recessive individuals for a specific genetic factor in the population. The 'solve' function will receive this array as a string and should return another string representing the solution array.

The final portion of the input is a sample Python code that reads the dataset from a file, calls the 'solve' function to compute the solution, and then prints the solution. However, the user is instructed not to print anything, not to write comments, and not to call this 'main' function in your solution. The user is also instructed to use four spaces for indentation.

```
1 Genetic Drift and the Hardy-Weinberg Principle Mendel's laws of
  segregation and independent assortment are excellent for the study of
  individual organisms and their progeny, but they say nothing about
  how alleles move through a population over time.
2
3 Our first question is: when can we assume that the ratio of an allele in
  a population, called the allele frequency, is stable? G. H. Hardy and
  Wilhelm Weinberg independently considered this question at the turn
  of the 20th Century, shortly after Mendel's ideas had been
  rediscovered. They concluded that the percentage of an allele in a
  population of individuals is in genetic equilibrium when five
  conditions are satisfied: The population is so large that random
  changes in the allele frequency are negligible. No new mutations are
  affecting the gene of interest; The gene does not influence survival
  or reproduction, so natural selection is not occurring; Gene flow, or
  the change in allele frequency due to migration into and out of the
  population, is negligible. Mating occurs randomly with respect to the
  gene of interest. The Hardy-Weinberg principle states that if a
  population is in genetic equilibrium for a given allele, then its
  frequency will remain constant and evenly distributed through the
  population. Unless the gene in question is important to survival or
  reproduction, Hardy-Weinberg usually offers a reasonable enough model
  of population genetics. One of the many benefits of the Mendelian
  theory of inheritance and simplifying models like Hardy-Weinberg is
  that they help us predict the probability with which genetic diseases
  will be inherited, so as to take appropriate preventative measures.
  Genetic diseases are usually caused by mutations to chromosomes,
  which are passed on to subsequent generations. The simplest and most
  widespread case of a genetic disease is a single gene disorder, which
  is caused by a single mutated gene. Over 4,000 such human diseases
  have been identified, including cystic fibrosis and sickle-cell
  anemia. In both of these cases, the individual must possess two
  recessive alleles for a gene in order to contract the disease. Thus,
  carriers can live their entire lives without knowing that they can
  pass the disease on to their children. The above introduction to
  genetic equilibrium leaves us with a basic and yet very practical
  question regarding gene disorders: if we know the number of people
  who have a disease encoded by a recessive allele, can we predict the
  number of carriers in the population?
4
5 Problem:
6
7 To model the Hardy-Weinberg principle, assume that we have a population
  of  $N$  diploid individuals. If an allele is in genetic equilibrium,
  then because mating is random, we may view the  $2N$  chromosomes as
```

```

receiving their alleles uniformly. In other words, if there are  $m$ 
dominant alleles, then the probability of a selected chromosome
exhibiting the dominant allele is simply  $p = \frac{m}{2N}$ . Because
the first assumption of genetic equilibrium states that the
population is so large as to be ignored, we will assume that  $N$  is
infinite, so that we only need to concern ourselves with the value of
 $p$ .
8
9 Given: An array  $A$  for which  $A[k]$  represents the proportion of
homozygous recessive individuals for the  $k$ -th Mendelian factor in a
diploid population. Assume that the population is in genetic
equilibrium for all factors.
10
11 Return: An array  $B$  having the same length as  $A$  in which  $B[k]$ 
represents the probability that a randomly selected individual
carries at least one copy of the recessive allele for the  $k$ -th
factor.
12
13 Sample Dataset
14 0.1 0.25 0.5
15
16 write the solve() function to solve the above problem
17
18 Do NOT print anything
19 Do NOT make comments
20 Do NOT call the main() function.
21
22 Use 4 spaces for tabs.
23
24 input_str: a string
25 return output: another string
26
27 def main():
28     with open("input.txt", "r") as f:
29         output = solve(f.read())
30         print(output)
31
32 Write your solution here
33 Begin with: def solve(input_str):

```

In the output portion, a Python function `afrq` is presented, which takes an array of proportions of homozygous recessive individuals for each genetic factor in a population and returns an array of probabilities that a randomly selected individual carries at least one copy of the recessive allele for each factor. The main function uses this `afrq` function to solve the problem based on the input data. It takes the input data from a file, applies the `afrq` function to compute the solution, and then prints the solution in the form of a space-separated string of floating-point numbers, each rounded to three decimal places.

```

1 # Counting Disease Carriers
2
3 from math import sqrt
4 from .helpers import Parser
5
6 def afrq(a):
7     """Counting Disease Carriers"""
8     return [2 * sqrt(x) - x for x in a]
9
10 def main(file):
11     b = afrq(Parser(file).floats())
12     print(*[round(x, 3) for x in b])

```

## D.3 EXAMPLE 3

This Python script is designed to find the most frequent k-mers (substrings of length k) in a given DNA sequence (a string), while allowing for a specified number of mismatches. This problem is referred to as the “Frequent Words with Mismatches Problem”.

The input to this problem is a string that contains two distinct parts separated by a newline:

The DNA sequence is represented as a string of characters ‘A’, ‘C’, ‘G’, and ‘T’. Two integers, k and d, are separated by a space. The integer k refers to the length of the substrings (k-mers) of interest, and d specifies the maximum number of mismatches that are allowed in a k-mer.

```

1
2 We defined a mismatch in ‘‘Compute the Hamming Distance Between Two
  Strings’’ . We now generalize ‘‘Find the Most Frequent Words in a
  String’’ to incorporate mismatches as well. Given strings Text and
  Pattern as well as an integer d, we define Countd(Text, Pattern) as
  the total number of occurrences of Pattern in Text with at most d
  mismatches. For example, Count1(AACAAGCTGATAAACATTTAAAGAG, AAAAA) = 4
  because AAAAA appears four times in this string with at most one
  mismatch: AACAA, ATAAA, AAACA, and AAAGA. Note that two of these
  occurrences overlap.
3
4 A most frequent k-mer with up to d mismatches in Text is simply a string
  Pattern maximizing Countd(Text, Pattern) among all k-mers. Note that
  Pattern does not need to actually appear as a substring of Text; for
  example, AAAAA is the most frequent 5-mer with 1 mismatch in
  AACAAGCTGATAAACATTTAAAGAG, even though AAAAA does not appear exactly
  in this string. Keep this in mind while solving the following problem
  .
5
6 Frequent Words with Mismatches Problem
7
8 Find the most frequent k-mers with mismatches in a string. Given: A
  string Text as well as integers k and d. Return: All most frequent k-
  mers with up to d mismatches in Text.
9
10 Sample Dataset
11 ACGTTGCATGTCGCATGATGCATGAGAGCT
12 4 1
13
14 Sample Output
15 GATG ATGC ATGT
16
17 write the solve() function to solve the above problem
18 Do NOT print anything
19 Do NOT make comments
20 Do NOT call the main() function.
21 Use 4 spaces for tabs.
22 input_str: a string
23 return output: another string
24
25 def main():
26     with open("input.txt", "r") as f:
27         output = solve(f.read())
28         print(output)
29
30 Write your solution here
31 Begin with: def solve(input_str):

```

Here is the corresponding reference code:

```

1 # Find the Most Frequent Words with Mismatches in a String
2
3 from .ba1g import hamming

```

```
4 from .ba1b import count_kmers, most_frequent
5 from itertools import product
6
7 # Note, the best kmer might not be observed in our sequence. The
  simplistic
8 # method here simply checks all possible kmers (which is ~17M for k = 12)
9
10 def generate_kmers(k):
11     return ("".join(x) for x in product(["A", "C", "G", "T"], repeat=k))
12
13 def count_hamming_kmers(kmers, d, k):
14     for x in generate_kmers(k):
15         count = sum(kmers[y] for y in kmers if hamming(x, y) <= d)
16         if count > 0:
17             yield [x, count]
18
19 def main(file):
20     seq, ints = open(file).read().splitlines()
21     k, d = list(map(int, ints.split()))
22     kmers = count_kmers(seq, k)
23     hkms = dict(count_hamming_kmers(kmers, d, k))
24     print(*most_frequent(hkms))
```

## E ADDITIONAL MODEL INFORMATION

Model	Parameters
InCoder (Fried et al., 2023)	6B
SantaCoder (Allal et al., 2023)	1.1B
StarCoder (Li et al., 2023)	15.5B
StarCoderPlus (Li et al., 2023)	15.5B
InstructCodeT5+ (Wang et al., 2023a)	16B
CodeGen (Nijkamp et al., 2023b)	6B-mono*
CodeGen (Nijkamp et al., 2023b)	16B-mono*
CodeGen2 (Nijkamp et al., 2023a)	7B*
GPT-3.5-Turbo	Unknown
GPT-4	Unknown

Table 8: Size of models.

We were not able to run the 16B models due to issues with scaling and sharding on the A100s.

We found that the following parameters produced the best results across all models.

- top\_k=50
- top\_p=0.95
- temperature=0.7
- early\_stopping=True
- num\_return\_sequences=1
- do\_sample=True

We utilized similar parameters to make model testing consistent across all tested models. This approach allowed for a more unbiased comparison of how each model performed on our benchmark. We determined these weights by averaging the recommended parameters for the above models and then used Bayesian optimization to determine the most optimal parameters.

Furthermore, note that we used the version of GPT-3.5-Turbo hosted on Azure OpenAI Service (gpt-35-turbo-0301). There may be some minor differences compared with the OpenAI Platform version.

## F UTILIZING CHATGPT FOR PROMPT GENERATION

ChatGPT served as a pivotal component in our prompt generation, enabling us to assemble comprehensive annotations for the ground-truth functions and their dependencies. This eliminates the need for extensive manual labor that would otherwise be spent on annotation. Utilizing GPT-3.5 Turbo calls further ensures consistency across annotations, mitigating the variability that might arise from human annotators.

The prompt was constructed as follows:

```
1 Generate a prompt for the following function, such that a programmer
   could reproduce it based solely on the description. Emphasize crucial
   components like the return statement, complex function calls, and
   ambiguous variables. However, avoid line-by-line descriptions; aim to
   provide a high-level overview. The response should be in plain text,
   free from any formatting such as Markdown. Keep the length under 200
   words or 15 lines, whichever is shorter.
2 Here is the function for you to describe:
3 <function>
4 Here are the additional dependencies that may or may not be used within
   the function:
5 <dependencies>
6 Again, limit your responses to no more than 200 words or 15 lines,
   whichever comes first.
7 Start your prompt with "write a function".
```

## G MODEL CONTEXT LIMITS

Model	Context limit
InCoder (Fried et al., 2023)	2048
SantaCoder (Allal et al., 2023)	2048
StarCoder (Li et al., 2023)	8192
StarCoderPlus (Li et al., 2023)	8192
InstructCodeT5+ (Wang et al., 2023a)	2048
CodeGen (Nijkamp et al., 2023b)	2048
CodeGen2 (Nijkamp et al., 2023a)	2048
GPT-3.5-Turbo	8192
GPT-4	8192

Table 9: Length limits of different models.

Since the limit includes both input and output tokens, we reserved 256 tokens for the output, therefore reducing the input length limit by 256. In our best effort to continue testing with these context limits, instead of treating a failing test case immediately, we attempted to truncate the prompts such that they fit barely below the context limit of each respective model. For all models, we also tested a version with the function summary at the top. Note that for all models, the context limit was identical regardless of the parameter count.

## H CODEREVAL COMPARISON

To validate the performance of code LLMs, multiple existing benchmarks are proposed, including only cases of generating a standalone function, i.e., a function that invokes or accesses only built-in functions and standard libraries. However, standalone functions constitute only about 30% of the functions from real open-source projects. To assess a model’s performance for pragmatic code generation (i.e., code generation for real settings of open source or proprietary code), CoderEval (Yu et al. 2023) proposes a benchmark named CoderEval of pragmatic code generation with generative pre-trained models.

In comparison to CoderEval, our approach is more focused on bioinformatics, as we ensure each function requires a certain level of domain knowledge in bioinformatics, as opposed to entry-level data science or software engineering tasks.

Moreover, we employ a more advanced parser, implementing a comprehensive parsing of the respective GitHub repositories. This includes the imported third-party packages and the classes that have been imported from another actual file.

We also conduct function testing at the file level, rather than the repository level, manually extracting the context at the end and running it through a framework that catches more errors than CoderEval.

Additionally, our dataset is larger than CoderEval’s, which only consists of 230 functions from 43 Python projects and 230 methods from 10 Java projects, while we have data from more than two thousand sources.

CoderEval classifies tasks into six levels according to the source of dependency outside the function, while we directly use the most complex dependencies.

CoderEval employs a human-labeled version description from 13 experienced engineers, while we leverage ChatGPT to generate function descriptions. Similarly, for test cases, our use of fuzz testing allows us to scale to large data volumes without the need for intensive manual annotation.

Yet, we share some similarities with CoderEval. Both BIOCODER and CoderEval can be used to assess the performance of models beyond merely generating standalone functions. Given the need to address dependencies with context, we both establish testing based on Docker, cloning GitHub repositories and their associated import dependencies. With ‘k’ candidate outputs generated by different models (e.g., 20), we simulate scenarios in actual IDEs.



## I PROMPT EXPLANATIONS

Below is an explanation of the prompt types:

1. **Summary Only** We begin the prompt from these prompts, where only the summary and the function signature are contained in the prompt. The summary is added as a comment directly above the signature, which is uncommented. The summary includes nearly complete details about the task; however, it intentionally does not thoroughly explain what the context is. Therefore, this result is best treated as a baseline when compared with other prompt types. An example is shown below:

```

1 #This is in python
2 #Write a function named "planewise_morphology_closing" that
  accepts two parameters, "x_data" and "structuring_element".
  Inside the function, create a new variable named "y_data" that
  will hold the result of the morphological closing operation.
  Loop through each item in "x_data" and perform the same
  operation on each. Do this by using the "enumerate" function to
  get both the index and plane for each item. Use the "closing"
  function from the "skimage.morphology" module to perform the
  morphological closing operation on each plane using the "
  structuring_element" parameter. Finally, return the newly
  created "y_data" array that holds the result of the operation.
  Ensure that the function uses the "numpy" and "skimage.
  morphology" modules, as they are dependencies for the function.
3 #
4 #def planewise_morphology_closing(x_data, structuring_element):
5
6

```

2. **Uncommented** We add the summary back into all of the parsed context. Therefore, the entire prompt contains all imports, global variables, classes, internal class functions, etc., that were imported into the file originally. For functions over ten lines, we summarize the parameters, return type, and logic instead of including the actual function code in order to reduce the number of tokens used and eliminate data that the model would not need. Note that, by design, if these extraneous functions were used, the test would have counted as failed, because it would be extremely inefficient code. An example is shown below:

```

1 This is in python
2 Here are the imports:
3 from cellprofiler.library.modules import closing
4 from ._help import HELP_FOR_STREL
5 from cellprofiler_core.module import ImageProcessing
6 from cellprofiler_core.setting import StructuringElement
7 Here are the global variables:
8 Here are the class declarations:
9 class Closing(ImageProcessing):
10     attributes: self.structuring_element, self.function
11     methods:
12     def create_settings(self):
13         summary: creates settings for the Closing function
14         param: self (unknown) - instance of the Closing function
15         return: None
16     def settings(self):
17         summary: Returns settings from superclass with self.
18         structuring_element added.
19         param: None
20         return: list - __settings__ with self.structuring_element
21         appended.
22     def visible_settings(self):
23         summary: This function returns the visible settings of a
24         Closing object.
25         param: self (Closing) - the Closing object whose settings
26         are to be returned.

```

```

23     return: list - the visible settings of the Closing object.
24     def run(self, workspace):
25         summary: Applies morphological closing to an image in a
           workspace.
26         param: workspace (object) - the workspace containing the
           image.
27         return: None
28 Here are the additional function declarations:
29 def planewise_morphology_closing(x_data, structuring_element):
30     summary: Applies morphology closing operation to each plane in
           x_data using structuring_element and returns the resulting
           y_data.
31     param: x_data (numpy array) - 3D array containing the input
           data.
32     param: structuring_element (numpy array) - Structuring element
           used to perform the closing operation.
33     return: y_data (numpy array) - 3D array containing the result
           of closure operation on each plane in x_data.
34 Here are the comments and the specs:
35 Write a function named "planewise_morphology_closing" that accepts
           two parameters, "x_data" and "structuring_element". Inside the
           function, create a new variable named "y_data" that will hold
           the result of the morphological closing operation. Loop through
           each item in "x_data" and perform the same operation on each.
           Do this by using the "enumerate" function to get both the index
           and plane for each item. Use the "closing" function from the
           "skimage.morphology" module to perform the morphological closing
           operation on each plane using the "structuring_element"
           parameter. Finally, return the newly created "y_data" array
           that holds the result of the operation. Ensure that the
           function uses the "numpy" and "skimage.morphology" modules, as
           they are dependencies for the function.
36 def planewise_morphology_closing(x_data, structuring_element):
37
38

```

Note that it is nearly certain that each prompt will contain extraneous data that does not need to be used for the function. The goal is for the LLM to ensure it uses the correct context for the function. Note that for prompts that exceed the model's context limit, this prompt likely fails.

- Summary at Bottom** We generate these using the previous prompt (uncommented), and comment out all the context. Note that for prompts that exceed the model's context limit, this prompt likely fails. In addition, note that there are no results for "summary at bottom" for Java due to an incompatibility with Java syntax. We could not generate this type of prompt for Java similar to how we generated it for Python.

```

1 #Here are the imports:
2 #from cellprofiler.library.modules import closing
3 #from ._help import HELP_FOR_STREL
4 #from cellprofiler_core.module import ImageProcessing
5 #from cellprofiler_core.setting import StructuringElement
6 #Here are the global variables:
7 #Here are the class declarations:
8 #class Closing(ImageProcessing):
9 #     attributes: self structuring_element, self.function
10 #     methods:
11 #     def create_settings(self):
12 #         summary: creates settings for the Closing function
13 #         param: self (unknown) - instance of the Closing function
14 #         return: None
15 #     def settings(self):
16 #         summary: Returns settings from superclass with self.
           structuring_element added.
17 #         param: None

```

```

18 #     return: list - __settings__ with self.structuring_element
    appended.
19 #     def visible_settings(self):
20 #         summary: This function returns the visible settings of a
    Closing object.
21 #         param: self (Closing) - the Closing object whose settings
    are to be returned.
22 #         return: list - the visible settings of the Closing object.
23 #     def run(self, workspace):
24 #         summary: Applies morphological closing to an image in a
    workspace.
25 #         param: workspace (object) - the workspace containing the
    image.
26 #         return: None
27 #Here are the additional function declarations:
28 #def planewise_morphology_closing(x_data, structuring_element):
29 #     summary: Applies morphology closing operation to each plane in
    x_data using structuring_element and returns the resulting
    y_data.
30 #     param: x_data (numpy array) - 3D array containing the input
    data.
31 #     param: structuring_element (numpy array) - Structuring element
    used to perform the closing operation.
32 #     return: y_data (numpy array) - 3D array containing the result
    of closure operation on each plane in x_data.
33 #Here are the comments and the specs:
34 #Write a function named "planewise_morphology_closing" that
    accepts two parameters, "x_data" and "structuring_element".
    Inside the function, create a new variable named "y_data" that
    will hold the result of the morphological closing operation.
    Loop through each item in "x_data" and perform the same
    operation on each. Do this by using the "enumerate" function to
    get both the index and plane for each item. Use the "closing"
    function from the "skimage.morphology" module to perform the
    morphological closing operation on each plane using the "
    structuring_element" parameter. Finally, return the newly
    created "y_data" array that holds the result of the operation.
    Ensure that the function uses the "numpy" and "skimage.
    morphology" modules, as they are dependencies for the function.
35 def planewise_morphology_closing(x_data, structuring_element):
36

```

4. **Summary at Top** These prompts are generated from the previous prompts (summary at bottom); however, the summary is copied at the top (moved for Java). This is intended for models with shorter context lengths, as when we truncated the prompt, the summary would still be intact, along with a portion of the context.

```

1 #This is in python
2 #Write a function named "planewise_morphology_closing" that
    accepts two parameters, "x_data" and "structuring_element".
    Inside the function, create a new variable named "y_data" that
    will hold the result of the morphological closing operation.
    Loop through each item in "x_data" and perform the same
    operation on each. Do this by using the "enumerate" function to
    get both the index and plane for each item. Use the "closing"
    function from the "skimage.morphology" module to perform the
    morphological closing operation on each plane using the "
    structuring_element" parameter. Finally, return the newly
    created "y_data" array that holds the result of the operation.
    Ensure that the function uses the "numpy" and "skimage.
    morphology" modules, as they are dependencies for the function.
3 #
4 #def planewise_morphology_closing(x_data, structuring_element):
5 #
6 #Here are the imports:

```

```

7 #from cellprofiler.library.modules import closing
8 #from ._help import HELP_FOR_STREL
9 #from cellprofiler_core.module import ImageProcessing
10 #from cellprofiler_core.setting import StructuringElement
11 #Here are the global variables:
12 #Here are the class declarations:
13 #class Closing(ImageProcessing):
14 #     attributes: self.structuring_element, self.function
15 #     methods:
16 #     def create_settings(self):
17 #         summary: creates settings for the Closing function
18 #         param: self (unknown) - instance of the Closing function
19 #         return: None
20 #     def settings(self):
21 #         summary: Returns settings from superclass with self.
22 #         structuring_element added.
23 #         param: None
24 #         return: list - __settings__ with self.structuring_element
25 #         appended.
26 #     def visible_settings(self):
27 #         summary: This function returns the visible settings of a
28 #         Closing object.
29 #         param: self (Closing) - the Closing object whose settings
30 #         are to be returned.
31 #         return: list - the visible settings of the Closing object.
32 #     def run(self, workspace):
33 #         summary: Applies morphological closing to an image in a
34 #         workspace.
35 #         param: workspace (object) - the workspace containing the
36 #         image.
37 #         return: None
38 #Here are the additional function declarations:
39 #def planewise_morphology_closing(x_data, structuring_element):
40 #     summary: Applies morphology closing operation to each plane in
41 #     x_data using structuring_element and returns the resulting
42 #     y_data.
43 #     param: x_data (numpy array) - 3D array containing the input
44 #     data.
45 #     param: structuring_element (numpy array) - Structuring element
46 #     used to perform the closing operation.
47 #     return: y_data (numpy array) - 3D array containing the result
48 #     of closure operation on each plane in x_data.
49 #Here are the comments and the specs:
50 #Write a function named "planewise_morphology_closing" that
51 #accepts two parameters, "x_data" and "structuring_element".
52 #Inside the function, create a new variable named "y_data" that
53 #will hold the result of the morphological closing operation.
54 #Loop through each item in "x_data" and perform the same
55 #operation on each. Do this by using the "enumerate" function to
56 #get both the index and plane for each item. Use the "closing"
57 #function from the "skimage.morphology" module to perform the
58 #morphological closing operation on each plane using the "
59 #structuring_element" parameter. Finally, return the newly
60 #created "y_data" array that holds the result of the operation.
61 #Ensure that the function uses the "numpy" and "skimage.
62 #morphology" modules, as they are dependencies for the function.
63 #def planewise_morphology_closing(x_data, structuring_element):

```

5. **Necessary Only** We use a mixture of our syntax solving algorithm and hand annotation to determine exactly which objects of the code are necessary for the function to execute, and then use them in place of the original context. Note that this is very similar to the environment used for testing the functions.

```
1 Write a function with the following specs:
```

```
2 --specs begin here--
3 #Write a function named "planewise_morphology_closing" that
  accepts two parameters, "x_data" and "structuring_element".
  Inside the function, create a new variable named "y_data" that
  will hold the result of the morphological closing operation.
  Loop through each item in "x_data" and perform the same
  operation on each. Do this by using the "enumerate" function to
  get both the index and plane for each item. Use the "closing"
  function from the "skimage.morphology" module to perform the
  morphological closing operation on each plane using the "
  structuring_element" parameter. Finally, return the newly
  created "y_data" array that holds the result of the operation.
  Ensure that the function uses the "numpy" and "skimage.
  morphology" modules, as they are dependencies for the function.
4 param: x_data (numpy array) - 3D array containing the input data.
5 param: structuring_element (numpy array) - Structuring element
  used to perform the closing operation.
6 return: y_data (numpy array) - 3D array containing the result of
  closure operation on each plane in x_data.
7 --specs end here--
8 Note the function will be embedded in the following context
9 --context begins here--
10 import numpy
11 import skimage.morphology
12 import os
13 <<insert solution here>>
14 def main():
15     numpy.random.seed(<|int;range=0,100|>)
16     pixels = numpy.random.randint(2, size=(10, 10, 10))
17     structuring_element = skimage.morphology.square(3)
18     print(planewise_morphology_closing(pixels, structuring_element
19 ))
20 if __name__ == "__main__":
21     main()
22 --context ends here--
23 Make sure to only generate the function and not any of the context
  . Make sure you are generating valid, runnable code. Begin your
  solution with:
24 def planewise_morphology_closing(x_data, structuring_element):
```

## J MODEL ABLATION STUDY

For the ablation study, we considered two representative functions, one for Java and one for Python, and determined how the various models performed in generating the following functions. The Java function we selected is the following:

```

1 public static String getReverseComplementedSequence(SAMRecord record, int
  startIndex, int length) {
2     if (startIndex < 0) {
3         startIndex = record.getReadLength() + startIndex;
4     }
5     byte[] rangeBytes = Arrays.copyOfRange(record.getReadBases(),
  startIndex, startIndex + length);
6     SequenceUtil.reverseComplement(rangeBytes);
7     return new String(rangeBytes);
8 }

```

and the (abridged) context that the function `getReverseComplementedSequence` had access to is the following:

```

1 import org.apache.commons.math3.distribution.HypergeometricDistribution;
2 import java.text.DecimalFormat;
3 import java.util.*;
4 import java.util.function.Function;
5 import htsjdk.samtools.util.SequenceUtil;
6 import java.util.Objects;
7
8 class SAMRecord {
9     public byte[] arr2;
10
11     public SAMRecord(byte[] arr) {
12         arr2 = arr;
13     }
14
15     public byte[] getReadBases() {
16         return arr2;
17     }
18
19     public int getReadLength() {
20         return arr2.length;
21     }
22 }
23
24 final class Utils {
25     /**
26      * Method returns reverse complemented sequence for the part of the
27      * record. Can work with 3' and 5' ends
28      * (if start index < 0, then it will found the index in the end of
29      * sequence by adding the length of record).
30      * @param record read from SAM file to process
31      * @param startIndex index where start the sequence
32      * @param length length of part of sequence
33      * @return reverse complemented part of record
34      */
35     <<insert solution here>>
36
37     public static String reverse(String string) {
38         return new StringBuffer(string).reverse().toString();
39     }
40
41     public static String complement(String string) {
42         final byte[] bases = htsjdk.samtools.util.StringUtil.
  stringToBytes(string);
43         complement(bases);
44         return htsjdk.samtools.util.StringUtil.bytesToString(bases);
45     }
46 }

```

```

43     }
44 }

```

We chose this as the representative Java function because it is highly bioinformatics related and the solution will require an understanding of the context around it, namely the custom SAMRecord class. Furthermore, the model will have to understand that there are many helpful utility functions available in the Utils class that can be used to help complete the function, such as the reverse and complement functions.

The Python function that we selected for this ablation study is the following:

```

1 def absolute_clonal(cnarr, ploidy, purity, is_reference_male,
2   is_sample_female
3   ):
4   """Calculate absolute copy number values from segment or bin log2
5   ratios."""
6   absolutes = np.zeros(len(cnarr), dtype=np.float_)
7   for i, row in enumerate(cnarr):
8     ref_copies, expect_copies = _reference_expect_copies(row.
9     chromosome,
10    ploidy, is_sample_female, is_reference_male)
11    absolutes[i] = _log2_ratio_to_absolute(row.log2, ref_copies,
12    expect_copies, purity)
13  return absolutes

```

and the (abridged) context that the function absolute\_clonal had access to is the following:

```

1 import random
2 import hashlib
3 import numpy as np
4 import os
5
6 def _reference_expect_copies(chrom, ploidy, is_sample_female,
7   is_reference_male
8   ):
9   """Determine the number copies of a chromosome expected and in
10  reference.
11  For sex chromosomes, these values may not be the same ploidy as the
12  autosomes. The "reference" number is the chromosome's ploidy in the
13  CNVkit reference, while "expect" is the chromosome's neutral ploidy
14  in the
15  given sample, based on the specified sex of each. E.g., given a
16  female
17  sample and a male reference, on chromosome X the "reference" value is
18  1 but
19  "expect" is 2.
20  Returns
21  -----
22  tuple
23  A pair of integers: number of copies in the reference, and
24  expected in
25  the sample.
26  """
27  chrom = chrom.lower()
28  if chrom in ['chrX', 'x']:
29    ref_copies = ploidy // 2 if is_reference_male else ploidy
30    exp_copies = ploidy if is_sample_female else ploidy // 2
31  elif chrom in ['chrY', 'y']:
32    ref_copies = ploidy // 2
33    exp_copies = 0 if is_sample_female else ploidy // 2
34  else:
35    ref_copies = exp_copies = ploidy
36  return ref_copies, exp_copies

```

```

32 def _log2_ratio_to_absolute_pure(log2_ratio, ref_copies):
33     """Transform a log2 ratio to absolute linear scale (for a pure sample
34     ).
35     Purity adjustment is skipped. This is appropriate if the sample is
36     germline
37     or if scaling for tumor heterogeneity was done beforehand.
38     .. math :: n = r*2^v
39     """
40     ncopies = ref_copies * 2 ** log2_ratio
41     return ncopies
42
43 def _log2_ratio_to_absolute(log2_ratio, ref_copies, expect_copies, purity
44 =None
45 ):
46     """Transform a log2 ratio to absolute linear scale (for an impure
47     sample).
48     Does not round to an integer absolute value here.
49     Math::
50     log2_ratio = log2(ncopies / ploidy)
51     2^log2_ratio = ncopies / ploidy
52     ncopies = ploidy * 2^log2_ratio
53     With rescaling for purity::
54     let v = log2 ratio value, p = tumor purity,
55         r = reference ploidy, x = expected ploidy,
56         n = tumor ploidy ("ncopies" above);
57     v = log_2(p*n/r + (1-p)*x/r)
58     2^v = p*n/r + (1-p)*x/r
59     n*p/r = 2^v - (1-p)*x/r
60     n = (r*2^v - x*(1-p)) / p
61     If purity adjustment is skipped (p=1; e.g. if germline or if scaling
62     for
63     heterogeneity was done beforehand)::
64     n = r*2^v
65     """
66     if purity and purity < 1.0:
67         ncopies = (ref_copies * 2 ** log2_ratio - expect_copies * (1 -
68         purity)
69         ) / purity
70     else:
71         ncopies = _log2_ratio_to_absolute_pure(log2_ratio, ref_copies)
72     return ncopies
73
74 <<insert solution here>>

```

We chose this as the representative Python function because, like the Java function that we chose, it is both highly bioinformatics related and the solution will require an understanding of the context around it, namely the custom `_reference_expect_copies` and `_log2_ratio_to_absolute` functions, as well as their problem, will require the use of some of the general imports such as the `numpy` package.

1. **InCoder**: Similar to many other models, InCoder has very poor Java performance, due to the fact that it was trained and fine-tuned on a Python corpus. Therefore, InCoder achieves 0% Pass@K for the Java prompts. The main issue that InCoder runs into is that for Java, it seems to struggle to understand the objective in the prompt and fails to generate the actual function; instead, it seems to regurgitate the prompt, which will obviously result in a fail. For instance here is a sample InCoder output for the function `getReverseComplementedSequence`:

```

1     # Do not write any comments in your code.
2 #This is in java
3 #Write a function called "getReverseComplementedSequence" that
4     takes in a SAMRecord object, an integer called "startIndex",
5     and an integer called "length". The function should check if
6     startIndex is less than 0, and if it is, set startIndex to the
7     SAMRecord object's read length plus startIndex. The function
8     should then create a byte array called "rangeBytes" that is a

```



```

        copy of a portion of the SAMRecord object's read bases starting
        from the startIndex and ending at startIndex + length. The
        function should then call SequenceUtil.reverseComplement on
        rangeBytes, and finally return a new String created from the
        rangeBytes byte array.
4 #Here are the imports:
5 #import org.apache.commons.math3.distribution.
    HypergeometricDistribution;
6 #import java.text.DecimalFormat;
7 #import java.util.*;
8 #import java.util.function.Function;
9 #import htsjdk.samtools.util.SequenceUtil;
10 #import java.util.Objects;
11 #Here are the class declarations:
12 #class SAMRecord {
13 #     public byte[] arr2;
14 #     public SAMRecord(byte[] arr) {
15 #         arr2 = arr;
16 #     }
17 #     public byte[] getReadBases() {
18 #         return arr2;
19 #     }
20 #     public int getReadLength() {
21 #         return arr2.length;
22 #     }
23 #}
24 #class Region {
25 #     /**
26 #      * Chromosome name
27 #      */
28 #     public final String chr;
29 #     /**
30 #      * Region start position
31 #      */
32 #     public final int start;
33 #     /**
34 #      * Region end position
35 #      */
36

```

Although InCoder performs comparatively better on Python, it still struggles greatly to understand the objective of the prompt and actually generate the function. Instead, for Python code, the InCoder models seem to simply regurgitate the summary of the Python function that can be found in the prompt. For instance, here is a sample InCoder output for the function `absolute_clonal`:

```

1     # summary: Returns the number of copies of a chromosome
    in reference and in sample.
2 # param: chrom (string) - the chromosome to be evaluated
3 # param: ploidy (int) - the ploidy of the chromosome
4 # param: is_reference_male (bool) - indicates if the reference
    is male or not
5 # return: tuple - a pair of integers indicating the number of
    copies in the reference and in the sample.
6 #def _log2_ratio_to_absolute(cnarr, ref_copies, expect_copies,
    purity):
7 # summary: Converts log2 ratio values into absolute copy number
    values.
8 # param: cnarr (array-like) - Copy number segments or bins
9 # param: ref_copies (array-like) - An array of integers
    indicating the number of copies in the reference for each
    segment or bin.
10 # param: expect_copies (array-like) - An array of integers
    indicating the number of copies in the expected for each
    segment or bin.

```

```

11 # param: purity (float) - estimated tumor pur
12

```

Overall, it seems that InCoder struggles to understand longer and more complex prompts, and t struggles to understand the objective of the prompts. Often it simply defaults to regurgitating parts of the prompt that it has encountered, without generating the desired function.

## 2. SantaCoder:

Similar to many other models, SantaCoder also has poor Java performance, due to the fact that it was trained and fine-tuned on a Python corpus. Therefore, SantaCoder achieves 0% Pass@K for the Java prompts. The main issue that SantaCoder runs into is that Java seems to struggle to understand the objective in the prompt and fails to generate the full function, only generating parts of the function. This may also be due to the smaller context limit of SantaCoder, resulting in it being unable to fully generate the functions. For instance, here is a sample SantaCoder output for the function `getReverseComplementSequence`:

```

1      #    double x = perc;
2 #    int a = (int) x;
3 #    int b = (int) (x - a);
4 #    int c = (int) (b - a);
5 #    if(c==0){
6 #        return 0;
7 #    }else if(c==1){
8 #        return 1;
9 #    }else{
10 #        return 2;
11 #    }
12 #    //return getColor(value, perc);
13
14 #    int c1 = (int) ((perc - a)/b);
15 #    int c2 = (int) ((perc - a - b)/a);
16 #    return 1 + c1 * 3 + c2 * 5;
17
18 #    int a1 = (int) (c1 / 2);
19 #    int a2 = (int) ((c1 - a1) / 2);
20 #    int a3 = (int) (c2 / 2);
21 #    int a4 = (int) ((c2 - a3) / 2);
22 #    int a5 = (int) (c1 +
23

```

Although SantaCoder performs comparatively better on Python, especially considering its small model size, it still struggles greatly to generate many of the functions. Often due to the small context size of the SantaCoder model, thus for more complicated functions, SantaCoder is unable to generate the model. Overall, it seems that SantaCoder struggles to understand the longer and more complex prompts, and it often struggles to understand the objective of the prompts. Furthermore, the small context limit of SantaCoder limits its ability to generate more complex and more context-rich code.

## 3. StarCoder:

Similar to many of the other models, StarCoder has very poor Java performance, due to the fact that it was trained and finetuned on a Python corpus. Therefore, StarCoder achieves 0% Pass@K for the Java prompts. However, unlike some of the other models, it seems to understand the objective of the prompt and uses the context appropriately. For instance, using the `reverseComplement` helper function correctly when generating the `getReverseComplementedSequence` function, as can be seen below:

```

1      def getReverseComplementedSequence(samRecord, startIndex,
2      length):
3          if startIndex < 0:
4              startIndex = samRecord.getReadLength() + startIndex
5              rangeBytes = samRecord.getReadBases().getBytes()[startIndex :
6              startIndex + length]
7              reverseComplemented = SequenceUtil.reverseComplement(
8              rangeBytes)

```

```

6     return reverseComplemented.toString()
7

```

However, StarCoder, being finetuned on Python, seems to have a hard time generating proper Java syntax, and instead generates a Python function. Although StarCoder performs comparatively better on Python, it still struggles greatly to generate many of the functions. Python seems to understand the objective in the prompt and generates proper syntax, however, some of its logic is incorrect, resulting in some failed tests. For instance, here is a sample StarCoder output for the function `absolute_clonal`:

```

1     #Write your function here!
2 def absolute_clonal(cnarr, ploidy, purity, is_reference_male,
3     is_sample_female):
4     absolutes = np.zeros(len(cnarr), dtype=np.float_)
5     for i, row in enumerate(cnarr):
6         chromosome, log2_value = row
7         ref_copies, expect_copies = _reference_expect_copies(
8             chromosome, ploidy, is_sample_female, is_reference_male)
9         absolute_value = _log2_ratio_to_absolute(log2_value,
10            ref_copies, expect_copies, purity)
11         absolutes[i] = absolute_value
12     return absolutes

```

4. **StarCoder+**: Unlike many of the other models, StarCoder+ has nonzero Pass@K performance on Java. This is likely because StarCoder+ is trained and finetuned on a larger corpus that consists of more languages than just Python. It seems to understand the objective of the prompt and uses the context appropriately, for instance using the `reverseComplement` helper function correctly when generating the `getReverseComplementedSequence` function, as can be seen below:

```

1     import htsjdk.samtools.util.SequenceUtil;
2 public class getReverseComplementedSequence
3 {
4 public static String getReverseComplementedSequence(SAMRecord
5     samRecord, int startIndex, int length)
6 {
7 if(startIndex < 0)
8 {
9 startIndex = samRecord.getReadLength() + startIndex;
10 }
11 byte[] rangeBytes = new byte[length];
12 samRecord.getReadBases(rangeBytes, startIndex, length);
13 byte[] rcBytes = SequenceUtil.reverseComplement(rangeBytes);
14 return new String(rcBytes);
15 }
16 }

```

Unlike StarCoder, it seems to understand and be able to generate Java syntax.

5. **CodeGen**: The performance of CodeGen1 was not great. This could be due to a variety of factors, including the fact that CodeGen models are not intended for full multi-line function generation, or the fact that the dataset used to train CodeGen models does not contain much Java code. This is demonstrated in some of the generations, as CodeGen ended up generating Python code, despite us mentioning that this prompt is in Java, and using Java code in the context. Sometimes, it also generates the comment again, such as below:

```

1 #Note: This is not a string method.
2

```

6. **CodeGen2**: CodeGen2 outperformed CodeGen as expected in our normal test cases; however, it was unexpectedly slightly outperformed in the Rosalind benchmark.
7. **GPT-3.5**: GPT clearly performs the best out of all models tested, on both Java and Python and across all prompt types. The version we used on Azure had an 8k context size, so it had

an advantage over some of the models due to being able to understand more of the context. Furthermore, the fact that some of the function descriptions were generated by GPT might mean that it had an easier time generating code from its own description. Furthermore, given the size of the model, it is possible that some of the functions were memorized, although we tried to combat this by changing the function signature slightly in some cases. Here is an example of the GPT output:

```

1 def FindLocalPeaks(signal):
2     peakLoc = []
3     maxSuspect = None
4     minSuspect = None
5     for i in range(1, len(signal)-1):
6         if signal[i] > 0:
7             if signal[i] > signal[i-1] and signal[i] == signal[i
8 +1]:
9                 maxSuspect = i
10            elif signal[i] == signal[i-1] and signal[i] > signal[i
11 +1]:
12                if maxSuspect is not None:
13                    peakLoc.append(maxSuspect)
14                    maxSuspect = None
15                elif signal[i] > signal[i-1] and signal[i] > signal[i
16 +1]:
17                    peakLoc.append(i)
18            elif signal[i] < 0:
19                if signal[i] < signal[i-1] and signal[i] == signal[i
20 +1]:
21                    minSuspect = i
22                elif signal[i] == signal[i-1] and signal[i] < signal[i
23 +1]:
24                    if minSuspect is not None:
25                        peakLoc.append(minSuspect)
26                        minSuspect = None
27                    elif signal[i] < signal[i-1] and signal[i] < signal[i
28 +1]:
29                        peakLoc.append(i)
30            return np.array(peakLoc, dtype=int)

```

It is also important to note that in every prompt, GPT-3.5 generated usable, cleaner code without comments as directed. Therefore, most test cases were run without syntax errors, but mostly logic errors.

Next, we performed a horizontal analysis on a single prompt to compare the outputs of each of the models with each other in order to understand the differences between the models and gain insight into what parts of the prompts the various models seem to understand. We applied the following problem for the horizontal analysis:

```

1 @property
2 def outlines(self):
3     """Get a mask of all the points on the border of objects"""
4     if self._outlines is None:
5         for i, labels in enumerate(self.labels):
6             if i == 0:
7                 self._outlines = centrosome.outline.outline(labels) != 0
8             else:
9                 self._outlines |= centrosome.outline.outline(labels) != 0
10        if self.line_width is not None and self.line_width > 1:
11            hw = float(self.line_width) / 2
12            d = scipy.ndimage.distance_transform_edt(~self._outlines)
13            dti, dtj = numpy.where((d < hw + 0.5) & ~self._outlines)
14            self._outlines = self._outlines.astype(numpy.float32)
15            self._outlines[dti, dtj] = numpy.minimum(1, hw + 0.5 - d[dti,
16 dtj])
17        return self._outlines.astype(numpy.float32)

```

The reasons we chose this function for the horizontal analysis are the following:

1. It is a Python function, and all of the model’s generally seem to be fine-tuned on Python code and thus will perform better on Python code; therefore, it would be a more fair comparison to use a Python function for this analysis.
2. Furthermore, this is also a relatively tricky function to generate as it has many moving parts and relies on many different levels of understanding of the context. First, it is important to note that this has a `@property` decorator and is a function that is part of a class; thus, the models must understand this and generate the `@property` decorator accordingly. Furthermore, the model must understand and correctly use the `self._outlines`, `self._line_width`, and `self.labels` instance variables. Finally, the model must understand and use correctly the highly bioinformatics-specific external library that is `centrosome`.

The prompt that we used for the horizontal analysis is the following “Summary Only” prompt:

```

1 #This is in python
2 #Write a function called "get_border_mask" that takes in a class object
  as an argument and returns a binary mask of all the points on the
  border of objects. The function should check if the instance variable
  "_outlines" is None and if it is, then use the labels attribute of
  the class object to generate the outlines mask using "centrosome.
  outline.outline(labels)" method. If the "_outlines" variable is not
  None, then it should iterate over "centrosome.outline.outline(labels)
  " for each i, and update the "_outlines" variable using the bitwise
  OR operator with the new mask. If the line_width attribute of the
  class object is not None and greater than 1, the function should
  calculate the distance transform using the "~self._outlines" mask and
  compute the binary mask for borders with the given line width (hw)
  using numpy.where(). Finally, the function should return the mask as
  a float32 numpy array. Make sure to import the necessary dependencies
  including "centrosome.outline" and "numpy" and also initialize the "
  _outlines" variable in the class.
3 #
4 #The function is located in the class OutlinesMixin
5 #
6 #@property
7 #def get_border_mask(self):

```

We chose the “Summary Only” format of the prompt because generally, the models performed best when the prompt was written in the “Summary Only” format. The outputs of the various models for this prompt are as follows:

### 1. InCoder:

```

1 def get_border_mask(self):
2     #If outlines is none, generate outlines using centrosome.
    outline.outline(labels) method and set outlines to the
    generated mask.
3     if self.outlines is None:
4         self.outlines = self.outline.outline(labels=self.labels)
5
6     # Get the mask using numpy
7     border_mask = np.where(self.outlines > 0, 1, 0 )
8
9     # Check if line width is greater than 1
10    if self.line_width > 1:
11        # Get distance transform
12        dt = np.abs(self.outlines - border_mask) * 255
13        dt[dt==0] = 255
14
15    # Use np.where to create border mask
16    border_mask = np.where(
17        dt > 0,

```

```

18         1,
19         0
20     )
21
22     # Return border mask as float32 numpy array
23     return border_mask.astype(np.float32)
24

```

## 2. SantaCoder:

```

1 def get_border_mask(self):
2     if self._outlines is None:
3         self._outlines = centrosome.outline.outline(self.labels)
4         outlines_mask = np.where(self._outlines, self._outlines, self.
5             labels)
6         #outlines_mask = np.where(self._outlines, self._outlines, self
7             ._outlines)
8         #outlines_mask = np.where(outlines_mask > 0, outlines_mask, 0)
9         return outlines_mask

```

## 3. StarCoder:

```

1 def get_border_mask(self):
2     if self._outlines is None:
3         self._outlines = centrosome.outline.outline(self.labels)
4     else:
5         for i in range(self._outlines.shape[0]):
6             self._outlines[i] = self._outlines[i] | centrosome.
7                 outline.outline(self.labels[i])
8         if self.line_width is not None and self.line_width > 1:
9             dist_trans = ndimage.distance_transform_edt(~self.
10                 _outlines)
11             border_mask = np.where(dist_trans <= self.line_width, 1,
12                 0)
13         else:
14             border_mask = self._outlines
15         return border_mask.astype(np.float32)

```

## 4. StarCoder+:

```

1 @property
2 def outlines(self):
3     if not hasattr(self, '_outlines'):
4         self._outlines = centrosome.outline.outline(self.labels)
5         return self._outlines
6     else:
7         for i in range(len(self.labels)):
8             self._outlines[i] = centrosome.outline.outline(self.
9                 labels)
10             self._outlines = self._outlines | self._outlines
11         return self._outlines

```

## 5. InstructCodeT5+:

```

1 def get_border_mask(self, labels):
2
3     if self._outlines is None:
4         self._outlines = outline(labels)
5
6     mask = self._outlines
7
8     if self.line_width is not None:
9         hw = self.line_width

```

```

10
11     if self.line_width>1:
12

```

## 6. CodeGen:

```

1 # Do not write any comments in your code.
2 #This is in python
3 #Write a function called "get_border_mask" that takes in a class
  object as an argument and returns a binary mask of all the
  points on the border of objects. The function should check if
  the instance variable "_outlines" is None and if it is, then
  use the labels attribute of the class object to generate the
  outlines mask using "centrosome.outline.outline(labels)" method
  . If the "_outlines" variable is not None, then it should
  iterate over "centrosome.outline.outline(labels)" for each i,
  and update the "_outlines" variable using the bitwise OR
  operator with the new mask. If the line_width attribute of the
  class object is not None and greater than 1, the function
  should calculate the distance transform using the "~self.
  _outlines" mask and compute the binary mask for borders with
  the given line width (hw) using numpy.where(). Finally, the
  function should return the mask as a float32 numpy array. Make
  sure to import the necessary dependencies including "centrosome
  .outline" and "numpy" and also initialize the "_outlines"
  variable in the class.
4 #
5 #The function is located in the class OutlinesMixin
6 #
7 #@property
8 #def get_border_mask(self):
9

```

## 7. CodeGen2:

```

1 def get_border_mask(self):
2     """
3     Computes the mask for the outlines of a morphology.
4
5     Parameters
6     -----
7     self : morphology object
8
9     Returns
10    -----
11    border_mask :
12

```

## 8. GPT-3.5:

```

1 @property
2 def get_border_mask(self):
3     if self._outlines is None:
4         self._outlines = outline(self.labels)
5     else:
6         for i in range(1, np.max(self.labels)):
7             self._outlines |= outline(self.labels == i)
8
9     if self.line_width is not None and self.line_width > 1:
10        dt = np.array(self._outlines, dtype=np.uint8)
11        dt = np.where(dt < 0.5, 1, 0)
12        dt = ndimage.distance_transform_edt(dt)
13        dt = (dt < self.line_width).astype(float)
14        return dt
15    else:
16        return np.array(self._outlines, dtype=np.float32)
17

```

## 9. GPT-4:

```

1 def outlines(self):
2     if self._outlines is None:
3         self._outlines = centrosome.outline.outline(self.labels)
4     else:
5         for i in range(len(self.labels)):
6             self._outlines |= centrosome.outline.outline(self.
7                 labels[i])
8
9     if self.line_width is not None and self.line_width > 1:
10        distances = scipy.ndimage.distance_transform_edt(~self.
11            _outlines)
12        hw = self.line_width / 2
13        self._outlines = numpy.where(distances <= hw, 1, 0)
14
15    return self._outlines.astype(numpy.float32)

```

In comparing the various model’s outputs to the chosen prompt, it is clear that there are various levels of understanding of the prompt between the models. By far the worst performing models are the InstructCodeT5+, CodeGen, and CodeGen2 models. All of these models failed to generate a full function. This is likely due to these models being trained to do one-line completions rather than full-function generation. InstructCodeT5+ did the best out of the three, as it at least generated part of a function and showed an understanding of the `self._outlines` instance variable unlike both the CodeGen and CodeGen2 models. However, InstructCodeT5+ also got the function signature wrong, showing no understanding in the prompt of what the structure of the function it was trying to generate was. The CodeGen2 model did not generate any function body; however, it did generate the function signature correctly and generated part of a useful docstring for the function. Out of these three models, the original CodeGen model performed by far the worst, as it simply regurgitated the prompt, not generating any new code.

For the remaining five models, there were different levels of understanding. The two models that demonstrated the deepest levels of understanding of the prompt were the StarCoder+ and GPT-3.5 models, as they were the only two models to recognize that the `get_border_mask` function has an `@property` function decorator. Furthermore, they both showed some understanding of the external function calls that were necessary to complete the function, with StarCoder+ calling the `centrosome.outline.outline` function correctly and GPT-3.5 using the `ndimage.distance_transform_edt` function, although not completely correctly, as the correct function call would be `scipy.ndimage.distance_transform_edt`. However, the logic that each of these models use to perform the `get_border_mask` function is correct, with GPT-3.5 getting closer to the intended logic. For instance, GPT-3.5 makes the `self.line_width is None` check.

The remaining three models not yet mentioned in this analysis are InCoder, SantaCoder, and StarCoder. These three models generated complete functions unlike InstructCodeT5+, CodeGen, and CodeGen2; however, they did not include the `@property` function decorator, unlike the StarCoder+ and GPT-3.5 models. Out of these three “middle-of-the-pack” models, StarCoder performs especially well, as it shows understanding of the three necessary instance variables, namely `self._outlines`, `self.labels`, and `self.line_width`, and uses both the `centrosome.outline.outline` and `ndimage.distance_transform_edt` external function calls. However, like GPT-3.5, it does not quite make the `ndimage` external call correctly. Furthermore, the structural logic of StarCoder’s code is similar to the golden code, on the same level as GPT-3.5. As for InCoder and SantaCoder, although they both generated full functions, their functions were somewhat off in their structural logic. SantaCoder performs decently, using the `centrosome.outline.outline` external package correctly; however, the function it generated is overly simple and does not meet the prompt description guidelines. As for InCoder, it uses a nonexistent `self.outlines` instance variable, instead of the intended `self._outlines` instance variable. Furthermore, it calls the nonexistent function `self.outline.outline` instead of the intended `centrosome.outline.outline` function. By contrast, InCoder writes a more involved function than SantaCoder which more closely mirrors the guidelines provided by the prompt.



## K PROMPT LENGTH STUDY

As shown by the following scatterplots, it seems that on models with an average Pass@K score of at least 2%, there is an inverse relationship between the number of tokens in the prompt and the Pass@K score. Furthermore, for models such as SantaCoder and GPT, the performance fell sharply after around 500 tokens. This could be due to the massive amount of context “confusing” the models. When we look at the *Necessary Only* prompts, a similar trend occurs for longer contexts, but the phenomenon is not as pronounced, as on average these prompts are significantly shorter.

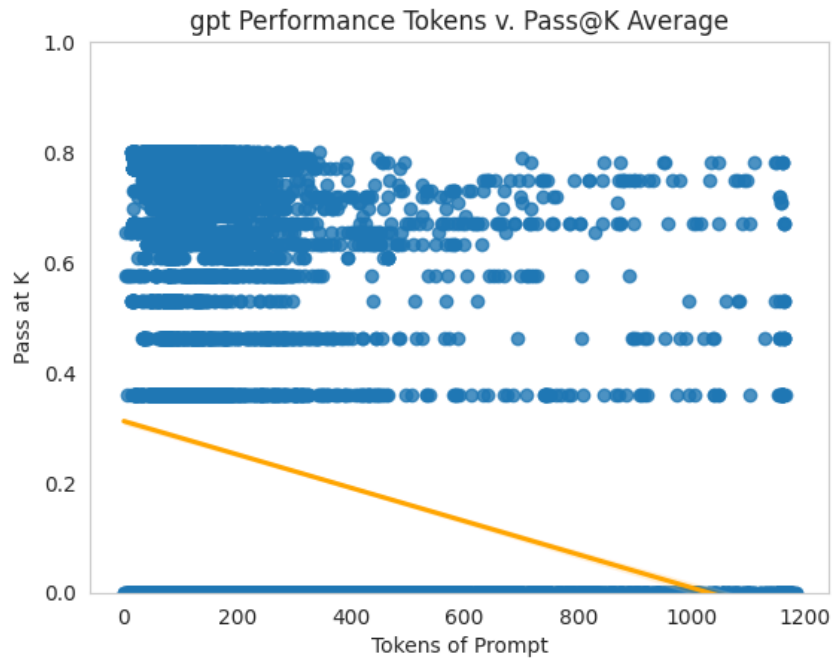


Figure 8: The scatterplots of the correlation of Pass@K and tokens of prompt. Statistics for GPT-4.

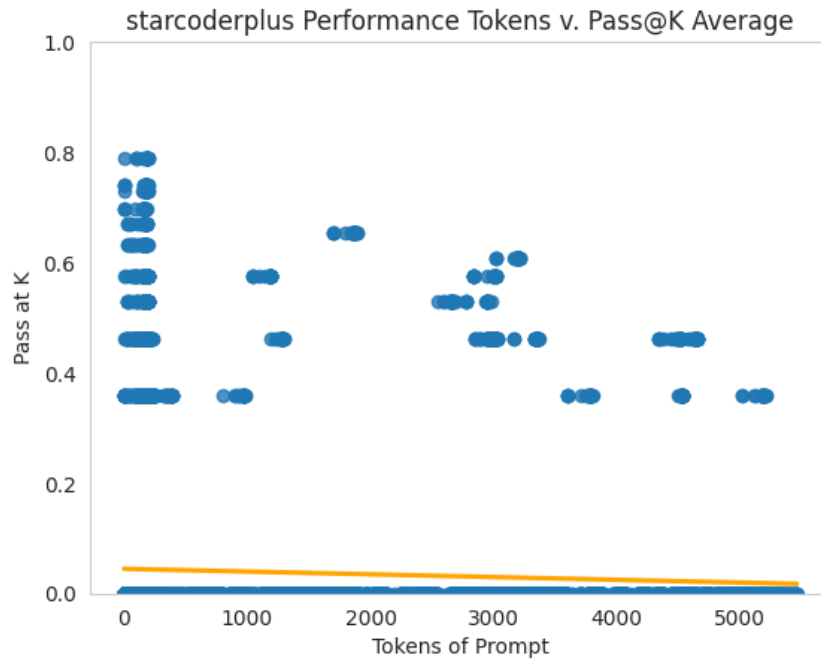


Figure 9: Statistics for Starcoderplus

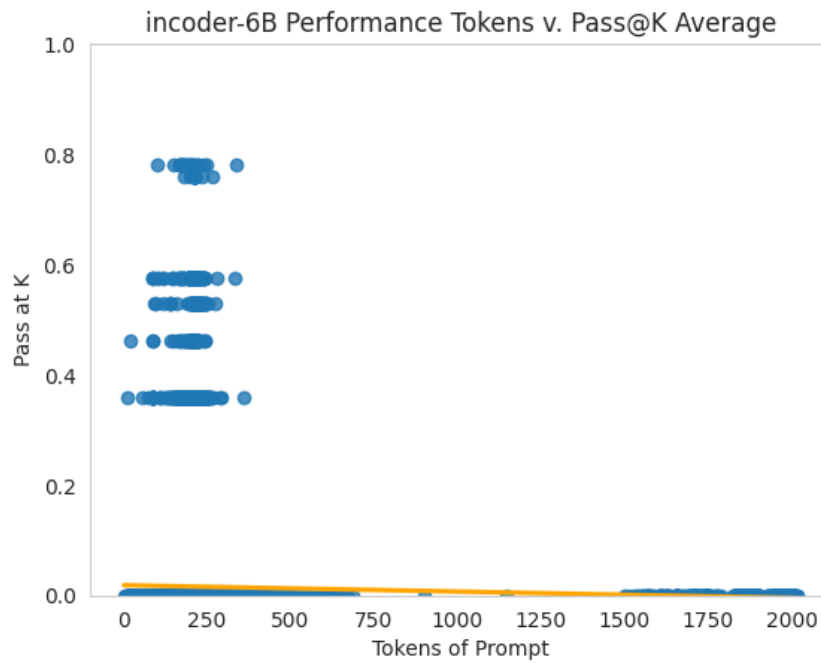


Figure 10: Statistics for Incoder

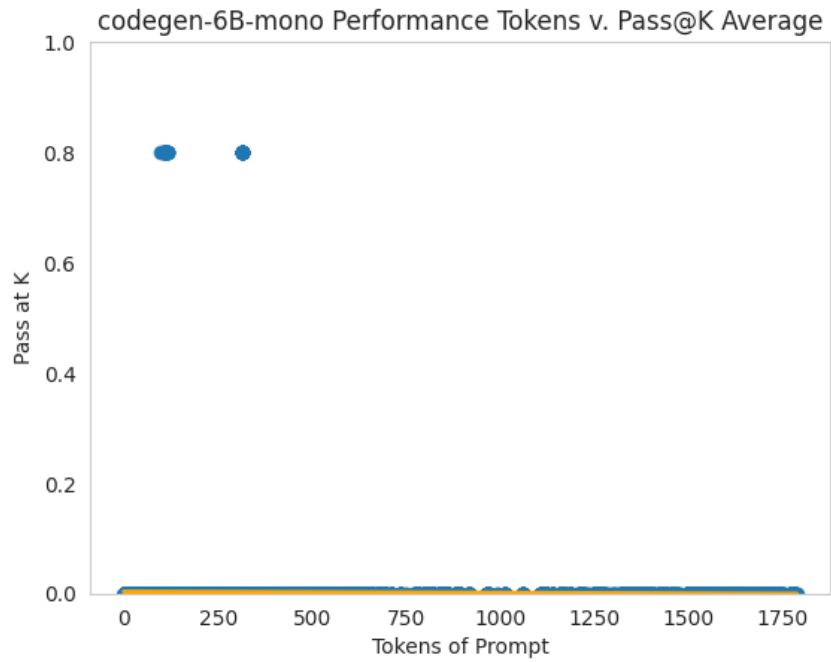


Figure 11: Statistics for CodeGen

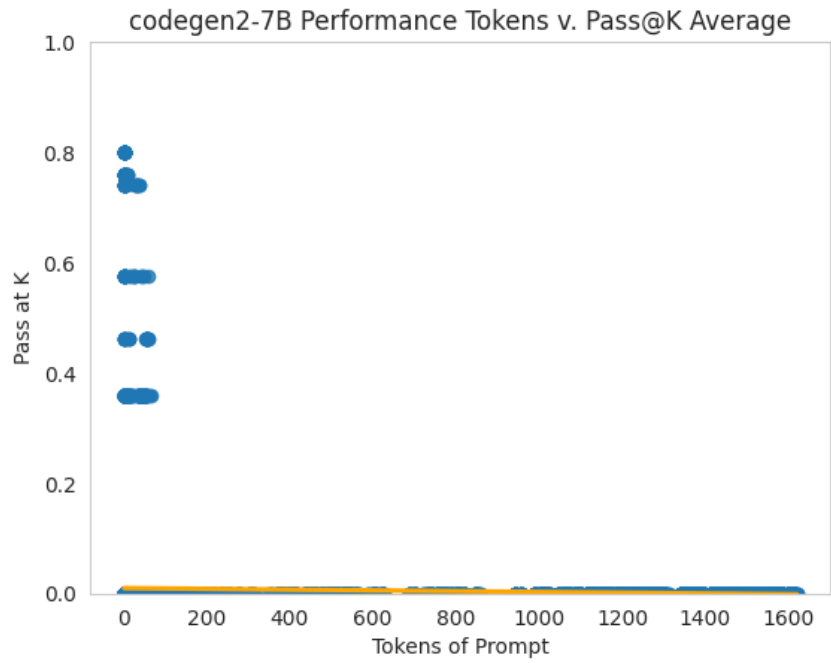


Figure 12: Statistics for CodeGen2

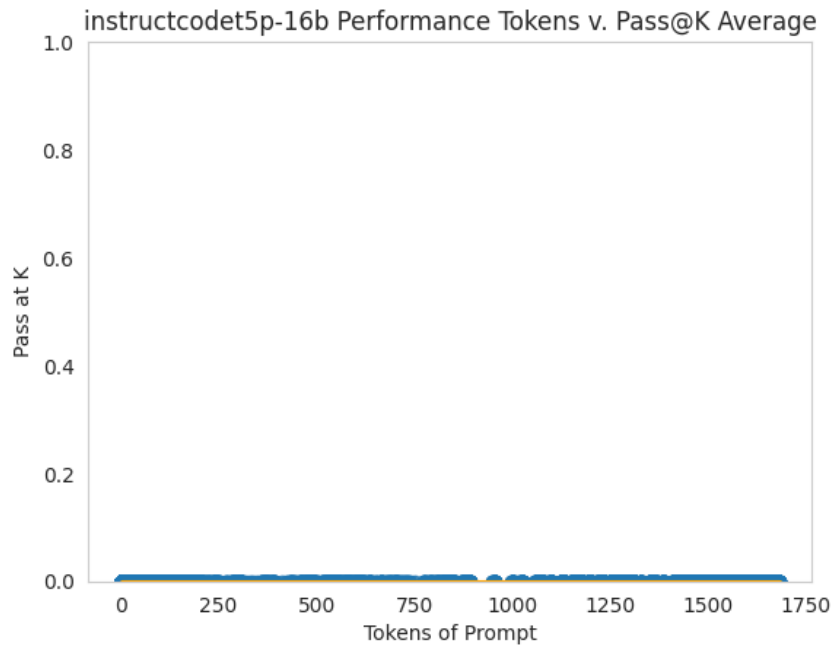


Figure 13: Statistics for InstructCodeT5+

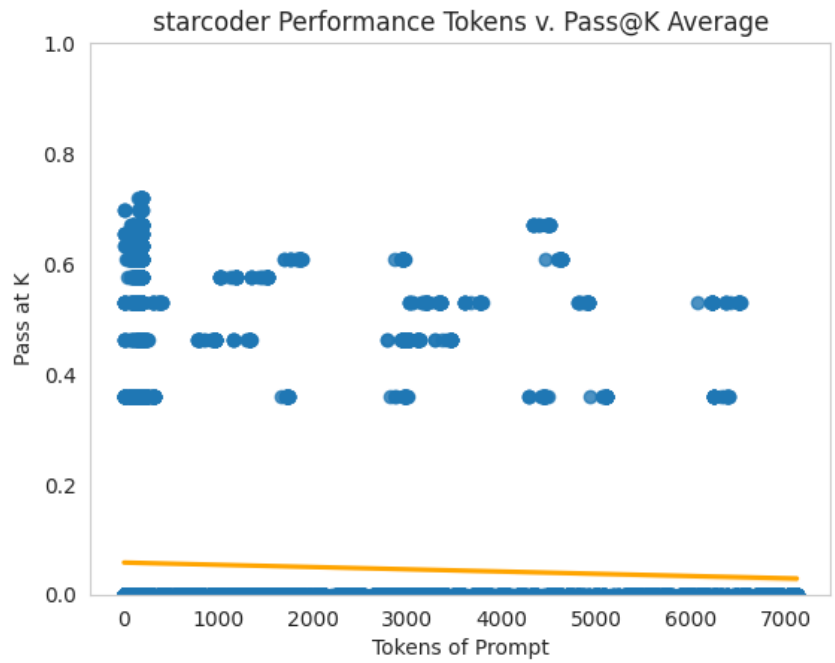


Figure 14: Statistics for Starcoder

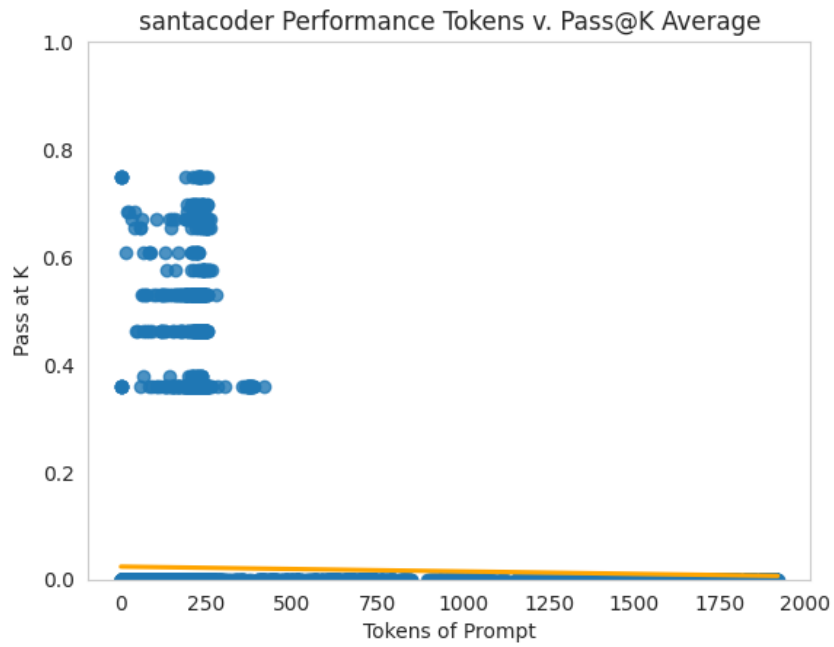


Figure 15: Statistics for Santacoder

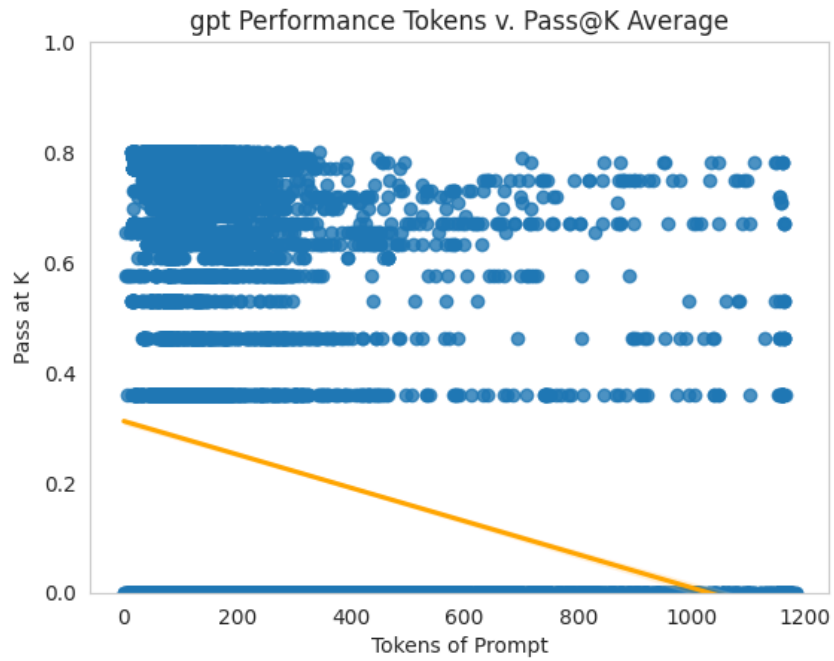


Figure 16: Statistics for GPT-3.5

## L TESTING FRAMEWORK

An outline of our testing frame is as follows:

1. **Preprocessing:** Each generated output is cleaned and minor corrections are applied in accordance with the above steps. We append a call to the `main()` function at the end of the file.
2. **Container Preparation:** The output from LLMs, alongside the environment files and sample test cases, are copied into the container.
3. **Fuzzing Execution:** The test cases are executed with a memory cap of 7GB and a time limit of 60 seconds. A test is considered passed if the actual output precisely matches the expected output, for all cases tested.
4. **Analysis:** We gather and analyze the results from these executions.

Through this approach, we provide a secure and efficient testing framework, promising robustness in the evaluation of generated code.

### L.1 MOTIVATION BEHIND FUZZ TESTING

We decided to utilize concepts from fuzz testing, as fuzz testing is widely used in the industry to capture bugs, crashes, security vulnerabilities, etc. in functions. However, in these cases, they do not have access to a "correct" version of the function; instead, they are merely creating inputs to intentionally try to crash the program, find out of bounds memory accesses, etc. Our situation is unique because we have the "golden code", or the ground truth version of the code, so given an input, we definitely know what the expected output should be, which is not something that's usually available in typical fuzz testing frameworks. Therefore, our situation could be considered a mixture of both unit testing and fuzz testing.

Given this requirement, and the goal of large-scale prompt generation, we decided to implement our own framework. We set out to accomplish two things: make the annotation process a lot easier for human editors, and support our feature set that combines both elements from unit testing and elements from fuzz testing. We believe that our resulting pipeline is more intuitive than piecing together other testing frameworks, and in our annotation process it proved to make things efficient, enabling larger-scale annotation, as the goal of the paper.

Furthermore, note that while handwritten test cases would likely target edge cases of a program (eg. branch coverage, conditional coverage), the probability of our fuzz testing framework to hit all of the same edge cases is high given 1000 iterations of randomly generated inputs. This means that we can save a significant amount of time building the dataset, as we only need to write an outline of a test case, and let the framework handle the rest. Therefore, we can think of the framework as thousands of "random unit tests," with a high probability that these unit tests would include handwritten test cases, if we had written them.

In terms of variable generation, we replace the `<|var_type;parameter|>` syntax with random values each iteration, for an unlimited number of iterations. These parameters are modifiable, and we implemented this system to be flexible, so that we can target specific scopes for fuzz testing. We check correctness by substituting the exact same variables in the original code, and checking if the outputs of the two functions match. This indicates identical functionality with the original code.

Here is an example of random integer and numpy array generation:

```

1 import numpy
2 import skimage.morphology
3 import os
4 <<insert solution here>>
5 def main():
6     numpy.random.seed(<|int;range=0,100|>)
7     labels = numpy.random.randint(2, size=(3, 3))
8     diameter = <|int;range=2,10|>
9     print(fill_object_holes(labels, diameter))
10 if __name__ == "__main__":
11     main()

```

Example of random string generation:

```

1 import random
2 [IMPORTS REDACTED FOR CONCISENESS]
3 import warnings
4 from textwrap import wrap
5 import string
6 import zlib
7 import io
8 from os.path import isfile
9 class GenipeError(Exception):
10     pass
11 _CHECK_STRING = b'GENIPE INDEX FILE'
12 def dosage_from_probs(homo_probs, hetero_probs, scale=2):
13     """Computes dosage from probability matrix (for the minor allele).
14     Args:
15         homo_probs (numpy.array): the probabilities for the homozygous
16         genotype
17         hetero_probs (numpy.array): the probabilities for the
18         heterozygous
19         genotype
20         scale (int): the scale value
21     Returns:
22         numpy.array: the dosage computed from the probabilities
23     """
24     return (homo_probs + (hetero_probs / 2)) * scale
25 <<insert solution here>>
26 def main():
27     np.random.seed(<|int;range=0,100|>)
28     prob_matrix = np.random.rand(10, 10)
29     a1 = <|string|>
30     a2 = <|string|>
31     print(maf_dosage_from_probs(prob_matrix, a1, a2))
32 if __name__ == "__main__":
33     main()

```

<|int|> denotes an integer. If left without parameters, then in one iteration of the program, this will be replaced with a random integer between INT\_MIN and INT\_MAX before compile time (or in this case, before the Python file is executed). There are parameters that can be passed in, that include range, even/odd, etc. Similarly, for <|string|> this generates a random ASCII string of any type. It can be further narrowed down into ASCII strings only, lowercase only, specific characters only, etc. by passing in the relevant parameters. These random inserted values can be manipulated to become part of a larger data structure, for example, a Numpy array, or a mock Python object.

When these files are executed, we replace «insert solution here» with the golden code on one iteration, and the error corrected generated code on a corresponding iteration. The fuzzing framework is designed so that the same inputs will be passed to this pair of iterations, meaning that we should be getting the same output (none of the functions have a non-deterministic component to them). Therefore this supports one aspect of the "secure" testing framework, as we have created an environment where all else is equal, except for the generated/golden code.

## M PROMPT LENGTH ABLATION STUDY

We have conducted additional experiments to illustrate the effect of different prompts.

As a result, our study revealed that more detailed text descriptions improve code generation, particularly for Python, but they have less impact on Java. However, irrespective of the level of detail, the core structure of the generated code remains similar. Hence, ambiguity in descriptions will not dramatically influence the outcome.

Specifically, in response to concerns about the ambiguity of the Python text descriptions, we conducted a study to examine the effects of ambiguous prompts on code generation. Using ChatGPT, we generated both “short” and “long” summaries, with the former being more concise and potentially omitting some details, and the latter providing more detail to reduce ambiguity.

For the “short” summaries, we used the following prompt:

```
1 Please write a clear and succinct prompt in one paragraph directing the
  user to write the following function. In the prompt make sure to
  include sufficient details on what the following function does.
```

For the “long” summaries, we used the following prompt:

```
1 Please write a highly detailed prompt in one paragraph directing the user
  to write the following function. In the one paragraph prompt make
  sure to include all the details on what the following function does.
  If you are unsure of some. of the variable and function references
  make an educated guess.
```

Passing the above prompts into ChatGPT, we generated “short” and “long” summaries. Then, we used these summaries to construct “short” and “long” prompts.

We calculated Pass@K results on ChatGPT generations for both “short” and “long” prompts. The results are as follows:

Prompt	Java				Python			
	Pass@1	Pass@5	Pass@10	Pass@20	Pass@1	Pass@5	Pass@10	Pass@20
Short Summary	34.700	42.313	43.000	43.800	25.960	36.371	39.825	42.484
Long Summary	36.800	46.412	49.842	54.000	38.567	52.419	56.439	60.510

Below are examples of “short” and “long” prompts:

```
1 SHORT PROMPT EXAMPLE
2 #This is in python
3 # Write a Python function called "UnifyLevels" that takes in three
  parameters: "baseLevel" (a constant array of integers), "addonLevel"
  (a constant array of integers), and "windowSize" (an integer). The
  function merges the two lists of breakpoints, but drops addonLevel
  values that are too close to baseLevel values. The function then
  returns the merged list of breakpoints as an array of integers. If
  the "addonLevel" list is empty, the function should return the "
  baseLevel" as is.
4 # Return only the code in the completion. I don't want any other comments
  . Don't say "here is your code" or similar remarks.
5 # def UnifyLevels(baseLevel, addonLevel, windowSize):
```

```
1 LONG PROMPT EXAMPLE
2 #This is in python
3 # Write a function called "UnifyLevels" that takes in three parameters: "
  baseLevel" (a list of integers), "addonLevel" (a list of integers),
  and "windowSize" (an integer). The purpose of this function is to
  merge the two lists of breakpoints (baseLevel and addonLevel), but
  drop any values from addonLevel that are too close to values in
  baseLevel. The function should first check if addonLevel is empty. If
  it is, the function should simply return baseLevel. Otherwise, the
```



```

function should create an empty list called "joinedLevel" and an
integer variable called "addon_idx" initialized to 0. The function
should then iterate over each element in baseLevel using a for loop.
Inside the loop, there should be a while loop that continues as long
as addon_idx is less than the length of addonLevel. Within the while
loop, the function should retrieve the element at index addon_idx
from addonLevel and assign it to a variable called "addon_elem". Next
, there should be three cases to handle:
4 # - If addon_elem is less than base_elem minus windowSize, the function
should append addon_elem to joinedLevel, increment addon_idx by 1,
and continue to the next iteration of the while loop.
5 # - If addon_elem is between base_elem minus windowSize and base_elem
plus windowSize (inclusive), the function should increment addon_idx
by 1 and continue to the next iteration of the while loop.
6 # - If addon_elem is greater than base_elem plus windowSize, the function
should assert that base_elem plus windowSize is less than addon_elem
, break out of the while loop, and move to the next iteration of the
for loop.
7 After each iteration of the for loop, the function should append
base_elem to joinedLevel. Once the for loop is complete, the function
should check if there are any remaining elements in addonLevel
starting from addon_idx. If there are, the function should extend
joinedLevel with those elements. Finally, the function should return
a numpy array containing the sorted elements of joinedLevel, with the
data type set to np.int_.
8 # Return only the code in the completion. I don't want any other comments
. Don't say "here is your code" or similar remarks.
9 # def UnifyLevels(baseLevel, addonLevel, windowSize):

```

According to our study, using more detailed prompts improved the performance, especially for Python generations. Python Pass@1 performance notably increased from 26.0% to 38.6% with more detailed prompts. However, the improvement for Java was less pronounced, with only a minor increase from 34.7% to 36.8%. Despite the difference in performance, the structural similarity between code generated from both short and long prompts remained high. Therefore, even if a prompt is ambiguous or trivial, it will not significantly affect the fundamental structure of the generated code. Consider the following short prompt and its generated example:

```

1
2 SHORT PROMPT
3
4
5 #This is in python
6 # Write a function called "FindLocalPeaks" that takes in a constant array
of floats called "signal" as a parameter. The function should find
the locations of local maxima on positive values and local minima on
negative values in the "signal" array. The first and last index of
the array should not be considered as extrema. The function should
iterate through the array and compare each element with its adjacent
elements to determine if it is a local peak. If a local peak is found
, its index should be stored in an array called "peakLoc". The
function should return the "peakLoc" array as the output. Make sure
to include the necessary import statement for numpy.
7 # Return only the code in the completion. I don't want any other comments
. Don't say "here is your code" or similar remarks.
8 # def FindLocalPeaks(signal):

```

```

1
2 SAMPLE GENERATION
3 import numpy as np
4 def FindLocalPeaks(signal):
5     peakLoc = []
6     for i in range(1, len(signal)-1):
7         if signal[i-1] < signal[i] > signal[i+1] and signal[i] > 0:
8             peakLoc.append(i)

```

```

9     elif signal[i-1] > signal[i] < signal[i+1] and signal[i] < 0:
10         peakLoc.append(i)
11     return peakLoc

```

and here is the corresponding long prompt and its sample generated example:

```

1 LONG PROMPT
2 #This is in python
3 # Write a function called "FindLocalPeaks" that takes in one parameter, "
  signal", which is a constant array of floats. The function is used to
  find local maxima on positive values and local minima on negative
  values. The function excludes the first and last indices of the "
  signal" array as possible extrema. The function should return an
  array of integers, "peakLoc", which represents the locations of these
  extrema in the "signal" array. The function should follow the logic
  provided in the source code provided below, which includes conditions
  for various scenarios when comparing adjacent elements in the "
  signal" array. When a local maxima or minima is found, its location
  should be appended to the "peakLoc" array. If you encounter equal
  values while comparing adjacent elements, there are additional
  conditions that need to be followed. Remember to import the necessary
  modules and define any necessary variables.
4 # Return only the code in the completion. I don't want any other comments
  . Don't say "here is your code" or similar remarks.
5 # def FindLocalPeaks(signal):

```

```

1
2 SAMPLE GENERATION
3 def FindLocalPeaks(signal):
4     import numpy as np
5     peakLoc = []
6     for i in range(1, len(signal)-1):
7         if signal[i] > 0:
8             if signal[i] > signal[i-1] and signal[i] > signal[i+1]:
9                 peakLoc.append(i)
10        elif signal[i] < 0:
11            if signal[i] < signal[i-1] and signal[i] < signal[i+1]:
12                peakLoc.append(i)
13    return np.array(peakLoc)

```

As can be seen from the above example, using a more detailed prompt (10 lines of description vs. 7 lines of description) does not have an outsized effect on the structure of the sample generations, as the functions generated from the two different prompts are structurally similar. Therefore, the improved performance likely stems from the small details that might be omitted in the concise prompts that are relevant to functional performance.

## N OBFUSCATION ABLATION STUDY

We conducted a study using “obfuscated prompts”. The framework for this “obfuscated prompts” ablation study drew inspiration from [Lachaux et al. \(2020\)](#); [Yang et al. \(2023\)](#). The prompts used were primarily “short” GPT summaries in their original form, with function or parameter names substituted with placeholder terminology deliberate adjustment to hinder a model’s ability to memorize portions of the training data.

Our analysis of the results shows that there is a performance decrease with obfuscated prompts compared to their non-obfuscated counterparts. However, this difference doesn’t reach a magnitude to justify attributing ChatGPT’s performance exclusively to the memorization of training data.

Below we provide an example showing a comparison between an obfuscated prompt with a non-obfuscated prompt.

```

1 SAMPLE ORIGINAL PROMPT
2 #This is in python
3 # Write a Python function called "unpipe_name" that takes in a string
  argument called "name" and returns a new string that is the single
  gene name after removing any duplications and pipe characters. This
  function is designed to fix the duplicated gene names produced by
  Picard CalculateHsMetrics, which combines the labels of overlapping
  intervals by joining all labels with '|'. If the name only consists
  of a single gene name, then that name is returned. If there are
  duplicated gene names and a meaningless target name, such as '-' or
  any name in the IGNORE_GENE_NAMES set, then they are removed. If
  there are still duplicated gene names, then the longest name is taken
  . If there are multiple equally long gene names, then one of them is
  arbitrarily selected and a warning is logged.
4
5 # Return only the code in the completion. I don't want any other comments
  . Don't say "here is your code" or similar remarks.
6 # def unpipe_name(name):

```

```

1 SAMPLE OBFUSCATED PROMPT
2 # This is in python
3 # Write a Python function called "FUNCTION" that takes in a string
  argument called "VAR0" and returns a new string that is the single
  gene VAR0 after removing any duplications and pipe characters. This
  function is designed to fix the duplicated gene names produced by
  Picard CalculateHsMetrics, which combines the labels of overlapping
  intervals by joining all labels with '|'. If the VAR0 only consists
  of a single gene VAR0, then that VAR0 is returned. If there are
  duplicated gene names and a meaningless target VAR0, such as '-' or
  any VAR0 in the IGNORE_GENE_NAMES set, then they are removed. If
  there are still duplicated gene names, then the longest VAR0 is taken
  . If there are multiple equally long gene names, then one of them is
  arbitrarily selected and a warning is logged.
4
5 # Return only the code in the completion. I don't want any other comments
  . Don't say "here is your code" or similar remarks.
6 # def FUNCTION(VAR0):

```

Notice that, by replacing specific function names and parameters with generic monikers like FUNCTION and VAR0 in our obfuscated prompts, we sought to limit the potential influence of prior training data memorization in ChatGPT’s code generation. Overall, this process aimed to increase the difficulty for the model to rely on memorization as a primary means of accurate code generation, thus ensuring that the model’s performance is based on interpreting the structure and requirement of the task, not on recollecting learned patterns.

We carried out Pass@K tests on the obfuscated prompts. The results were comparable to those achieved with non-obfuscated versions, although they were slightly lower, showing that obfuscation of the prompts does not significantly impede code generation performance. This implies that GPT 3.5’s performance on our dataset can not be attributed solely to memorization.

Prompt	Java				Python			
	Pass@1	Pass@5	Pass@10	Pass@20	Pass@1	Pass@5	Pass@10	Pass@20
Short Summary	34.700	42.313	43.000	43.800	25.960	36.371	39.825	42.484
Obfuscated Summary	30.700	40.684	44.250	47.900	22.771	32.733	35.872	38.790

Our experiment indeed shows that obfuscation of the prompts does not significantly impede code generation performance. Obfuscated prompts marginally underperformed in comparison to non-obfuscated ones, suggesting that memorization might play a role in ChatGPT’s performance. Yet, the proximity of the results demonstrates that the model’s understanding of the task described in the prompt is the primary driver of its performance.

Moreover, the fact that obfuscated and non-obfuscated short prompts yield similar generated examples serves as further evidence, testifying to the model’s capacity to grasp the task’s requirements and generate relevant codes even when specific cues from the prompt, such as function or parameter names, are withheld.

This finding supports the view that language models like ChatGPT generate code predominantly based on their understanding and interpretation of the task presented to them and that memorization, while present to an extent, does not appear to be the dominant factor.

Here is an example to further bolster our case, again showing that there is no data leakage. This example shows more detail on the generation:

```

1 SHORT PROMPT
2 #This is in python
3 # Write a function called ‘_neighbors‘ that takes in a two-or-three
  dimensional image and returns the number of neighbor pixels for each
  pixel of the image. The function should first pad the image with
  zeros, then create a mask where any pixel value greater than zero is
  considered a neighbor. Next, it should compute the response using a
  uniform filter and multiply it by a scaling factor. The labels are
  then computed by multiplying the response with the mask and slicing
  the padded image. Finally, the function should return the labels as
  an array of unsigned integers.
4 # Return only the code in the completion. I don’t want any other comments
  . Don’t say "here is your code" or similar remarks.
5 # def _neighbors(image):

```

```

1
2
3 SAMPLE GENERATION with ORIGINAL PROMPT
4
5
6 def _neighbors(image):
7     from skimage.filters import uniform_filter
8     from scipy import ndimage
9     scaling_factor = 10
10    mask = (image > 0).astype(int)
11    padded_image = np.pad(image, ((1,1), (1,1), (0,0)), 'constant',
  constant_values=0)
12    response = uniform_filter(padded_image, size=3)
13    labels = (response * scaling_factor * mask)[1:-1, 1:-1, :]
14    labels = ndimage.label(labels)[0]
15    return labels.astype(np.uint16)
16
17

```

```

1 SAMPLE OBFUSCATED PROMPT
2
3
4 # This is in python
5 # Write a function called ‘FUNCTION‘ that takes in a two-or-three
  dimensional VAR0 and returns the number of neighbor pixels for each

```

```
pixel of the VAR0. The function should first pad the VAR0 with zeros,
then create a mask where any pixel value greater than zero is
considered a neighbor. Next, it should compute the response using a
uniform filter and multiply it by a scaling factor. The labels are
then computed by multiplying the response with the mask and slicing
the padded VAR0. Finally, the function should return the labels as an
array of unsigned integers.
6 # Return only the code in the completion. I don't want any other comments
  . Don't say "here is your code" or similar remarks.
7 # def FUNCTION(VAR0):
8
9
10

1
2 SAMPLE GENERATION with OBFUSCATED PROMPT
3 def FUNCTION(VAR0):
4     import numpy as np
5     from scipy.ndimage import uniform_filter
6
7     VAR0 = np.pad(VAR0, ((1, 1), (1, 1)), mode='constant')
8     mask = VAR0 > 0
9     response = uniform_filter(mask, size=3)
10    scaling_factor = 10
11    labels = response * scaling_factor * mask * VAR0[1:-1, 1:-1]
12    return labels.astype(np.uint16)
```

The striking similarity between functions generated from both obfuscated and non-obfuscated prompts reaffirms that the model's ability to generate code is primarily based on its understanding and interpretation of the prompt and task, rather than the memorization of the training data.

Important to note, however, is the impressive resilience of the model's performance even after the obfuscation of specific functions or parameter names. This resilience indicates that the model does not rely heavily on these specific cues to generate appropriate code and can still produce functionally comparable results even when details are abstracted out.

## O TOPIC ANALYSIS

The repositories that we used for this study came from a larger set of 1,720 bioinformatics repositories that were constructed in [Russell et al. \(2018\)](#). We manually selected 28 high-quality repositories from this set. Each repository is a codebase of one bioinformatics journal article. We used latent Dirichlet allocation (LDA) to infer topics for abstracts of the articles citing each repository in the main dataset.

From the LDA model, we identified terms that were primarily associated with a single topic. We chose a model with eight topics due to its maximal coherence of concepts within the top topic-specialized terms. We manually assigned a label to each of the eight topics that capture a summary of the top terms. We then classified each article abstract into one or more topics. The eight topics in the LDA model are described below.

1. Cancer and epigenetics: Cancer and epigenetics refer to the study of heritable changes in gene expression that do not involve changes to the underlying DNA sequence but can lead to the development and progression of cancer. These epigenetic modifications, which include DNA methylation, histone modifications, and small RNA-associated gene silencing, can turn genes on or off, influencing cancer susceptibility, initiation, progression, and response to treatment. Understanding these processes can aid in developing targeted therapies and better diagnostic tools for cancer patients.

2. Proteomics and microscopy: Proteomics and microscopy refer to the integration of proteomic analyses (the large-scale study of proteins, their structures, and functions) with advanced microscopy techniques. This integration allows scientists to visualize and quantify the spatial and temporal distribution of proteins within cells and tissues. By combining the detailed molecular information from proteomics with the high-resolution imaging from microscopy, researchers can gain deeper insights into cellular processes, protein-protein interactions, and the structural organization of cellular components.

3. Variant calling: Variant calling is a process in bioinformatics where sequence data (often from next-generation sequencing) are analyzed to identify differences, or variants, between a given sample and a reference genome. These variants can include single-nucleotide polymorphisms, insertions, deletions, and more. Identifying these variants is crucial for understanding genetic diversity, disease susceptibility, and personalizing medical treatments based on an individual's genetic makeup.

4. Genetics and population analysis: Genetics and population analysis in bioinformatics refers to the study of the genetic composition of populations and how it changes over time. This involves analyzing large datasets of genetic information to understand patterns of genetic variation, inheritance, and evolution within and between populations. Such analyses can provide insights into population structure, migration, adaptation, and the history of species or populations, as well as help identify genetic factors associated with diseases or traits in specific populations.

5. Structure and molecular interaction: Structure and molecular interaction in bioinformatics pertains to the study and prediction of the three-dimensional structures of biomolecules (like proteins and nucleic acids) and how they interact with one another. By using computational methods to model and analyze these structures, scientists can gain insights into the molecular mechanisms underlying biological processes. This understanding can be pivotal for drug design, predicting protein functions, and understanding the effects of genetic mutations on molecular interactions and stability.

6. Web and graphical applications: Web and graphical applications in bioinformatics refer to the development and use of online tools and software with user-friendly graphical interfaces to analyze, visualize, and interpret biological data. These applications allow both experts and non-experts to interact with complex datasets, facilitating tasks like sequence alignment, gene annotation, pathway analysis, and more. By providing accessible platforms, these applications help democratize the analysis and understanding of vast and intricate bioinformatics data.

7. Assembly and sequence analysis: Assembly and sequence analysis in bioinformatics involves the process of taking raw sequence data, typically from next-generation sequencing technologies, and reconstructing the original DNA or RNA sequences. Assembly might mean piecing together short DNA reads into longer sequences or entire genomes. Once assembled, sequence analysis is used to identify genes, predict their functions, detect variations, and compare sequences across different organisms or individuals. This foundational process is key to many downstream analyses in genomics and personalized medicine.

8. Transcription and RNA-seq: Transcription and RNA-seq (RNA sequencing) in bioinformatics relate to the study of the transcriptome—the complete set of RNA molecules expressed from genes in a cell or tissue. RNA-seq is a technique that uses next-generation sequencing to capture and quantify the RNA in a sample, providing insights into which genes are active (or “expressed”) under specific conditions. By analyzing RNA-seq data, researchers can uncover patterns of gene expression, identify novel RNA species, and study regulatory pathways, aiding in understanding cellular responses, disease mechanisms, and developmental processes.

All eight topics were present across our subset of 28 repositories from which we created the benchmark dataset. A detailed breakdown of which topics corresponded to which repositories is as follows:

<b>Repository</b>	<b>Topic Areas</b>
AdmiralenOla/Scoary	Genetics and population analysis, Assembly and sequence analysis
biocore/deblur	Transcription and RNA-seq
CellProfiler/CellProfiler	Cancer and epigenetics, Proteomics and microscopy
CGATOxford/UMI-tools	Genetics and population analysis
choderlab/ensembl	Structure and molecular interaction
etal/cnvkit	Variant calling
gem-pasteur/macsfinder	Structure and molecular interaction, Web and graphical applications
hangelwen/miR-PREFeR	Transcription and RNA-seq
jnktsj/DNApi	Web and graphical applications, Transcription and RNA-seq
juliema/aTRAM	Genetics and population analysis, Assembly and sequence analysis
karel-brinda/rnftools	Variant calling, Structure and molecular interaction
mad-lab/transit	Web and graphical applications
makson96/Dynamics	Structure and molecular interaction
MikkelSchubert/paleomix	Variant calling, Assembly and sequence analysis
msproteomicstools/msproteomicstools	Proteomics and microscopy
ODonnell-Lipidomics/LipidFinder	Structure and molecular interaction
pgxcentre/genipe	Proteomics and microscopy, Genetics and population analysis
ratschlab/spladder	Transcription and RNA-seq
SamStudio8/goldilocks	Web and graphical applications
simonvh/fluff	Web and graphical applications
sjspielman/pyvolve	Structure and molecular interaction, Web and graphical applications
SpatialTranscriptomicsResearch/st pipeline	Proteomics and microscopy, Web and graphical applications, Transcription and RNA-seq
ursgal/ursgal	Proteomics and microscopy
vals/umis	Transcription and RNA-seq
williamgilpin/pypdb	Proteomics and microscopy, Web and graphical applications
zhanglab/psamm	Structure and molecular interaction, Web and graphical applications
zstephens/neat-genreads	Variant calling, Assembly and sequence analysis
samtools/htsjdk	Assembly and sequence analysis, Transcription and RNA-seq

## P MODEL ERROR DISTRIBUTIONS

Models	Failure: Syntax Error	Failure: Runtime Error	Failure: Timeout Error	Failure: Output Dis- agreement	Passed Tests
CodeGen-6B-Mono	11268	8176	1	148	105
CodeGen2-7B	12687	6718	0	211	79
GPT-3.5 Turbo	9231	10603	0	5624	6643
InCoder-6B	11268	8176	1	148	105
InstructCodeT5P-16B	19667	33	0	0	0
SantaCoder	14391	4601	1	555	139
StarCoder	26233	10688	0	1660	808



## Q LANGUAGE ANALYSIS

Our decision to include Java and Python was based on an empirical investigation into the prevalence of different programming languages across bioinformatics repositories. We computed the total byte sizes of various languages across all repositories we surveyed. Out of the 13 languages analyzed (Python, Bourne Shell, R, Perl, Cpp, C, Java, Bourne\_Again\_Shell, MATLAB, m4, SQL, Ruby, PHP), Java was the most prevalent, with a byte size of 242,318,285 bytes, followed closely by Python at 192,324,018 bytes. The presence of Java as the most extensive language indicates that it plays a crucial role in the bioinformatics community, perhaps in the form of underlying tools or frameworks that support high-level bioinformatics operations. A detailed breakdown of the byte size of the various languages in our repository is as follows:

<b>Programming Language</b>	<b>Number of Bytes</b>
Java	242,318,285
Python	192,324,018
C	184,967,144
C++	184,694,473
Perl	129,213,485
R	40,708,273
Bourne Shell	35,495,605
PHP	33,876,889
MATLAB	28,889,990
SQL	15,630,061
Ruby	8,935,640
m4	7,956,980
Bourne Again Shell	2,851,620

## R TEST CASE EXAMPLE

<pre>import random import hashlib import numpy import skimage import skimage.measure import scipy.ndimage import os import logging F_ANGLE = 'Angle' LOGGER = logging.getLogger(__name__) class UntangleWorms:     def ncontrol_points(self):         return &lt; int;range=3,10 &gt;         &lt;&lt;insert solution here&gt;&gt; def main():     print(UntangleWorms().angle_features()) if __name__ == "__main__":     main()</pre>	<b>Context file</b>	<pre>def angle_features(self):     """Return a list of angle feature names"""     try:         return ['_'.join((F_ANGLE, str(n)))                 for n in range(1, self.                     ncontrol_points() - 1)]     except:         LOGGER.error(             'Failed to get # of control             points from training file. Unknown number of             angle measurements')         , exc_info=True)         return []</pre>	<b>Golden Code</b>
		<pre>def angle_features(self):     # Define angle feature names     angle_feature_names = ['Feature1',         'Feature2', 'Feature3']     return angle_feature_names</pre>	<b>Generated Code</b>

Figure 17: Test case for UntangleWorms example. The context file includes various import dependencies and a class definition with a method placeholder for the solution. The UntangleWorms class comes from a GitHub repository file (<https://github.com/CellProfiler/CellProfiler/blob/master/cellprofiler/modules/untangleworms.py>) that was scraped in our study. UntangleWorms is an image analysis tool that was initially part of the paper “An image analysis toolbox for high-throughput *C. elegans* assays.”

## S ERROR DISTRIBUTION CHART

The errors include: ‘different output’ where the generated code’s output did not match the golden code’s output; ‘invalid syntax’ where syntax errors in the generated code prevented code execution; ‘function timed out’ where code execution exceeded time limits; and ‘runtime error’ where the generated code compiled successfully but failed to run. The vast majority of the generated code samples tested encountered a sort syntax or runtime error without resulting in an output. See Figure 18 for more detail.

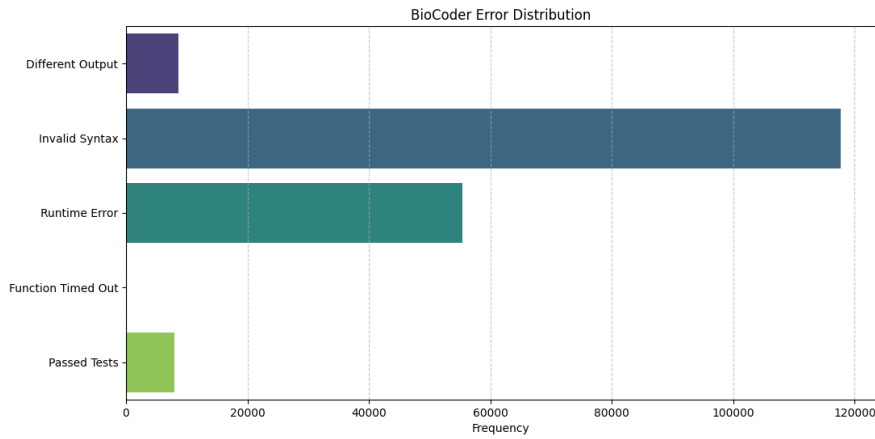


Figure 18: Error distribution aggregated over all models

## T FUNCTION FILTERING PROCESS

When filtering the function, we were given a large baseline of functions. In order to decrease the manual workload of filtering through these functions, we initiated two rounds of automatic filtering. In the first round, every function and its associated comments underwent keyword filtering. The keywords were scraped from bioinformatics Wikipedia articles, with each function requiring at least 10 keyword matches to be considered relevant to bioinformatics. This round was critical to avoid any unnecessary OpenAI API calls, which are expensive. The remaining functions advanced to the second round, where we inputted each function into GPT-3.5, which assessed bioinformatics relevance. With the workload significantly reduced, we manually sorted through the remaining functions, leaving us with a final count of 1,026 Python functions and 1,243 Java functions (see Table 2). Note that in Java, there are no global variables, because all variables are encapsulated in classes. In Table 2, the “similar data” set includes an additional 157 Python problems and 50 Java problems, maintaining the same 253 Rosalind problem tally, thus reflecting the composition of the public data. The issues in this set were cherry-picked based on their L2 similarity norm of calculated statistics, ensuring close alignment to the mean distribution of the public data.

Indeed, knowledge of bioinformatics was an integral part of constructing this dataset and benchmark. We stressed how we used extensive bioinformatics-specific keywords and relied on the expertise of four students specializing in Computer Science and Computational Biology. Our team manually sifted through potential functions to build our final dataset.

It is important to note that our selections were made with a focus on bioinformatics-related coding problems. From the bioinformatics-related repositories, we obtained a vast number of functions. However, among these functions: some were highly specific to bioinformatics (like algorithms), while others were not directly relevant (like web page scripts).

Therefore, a key task in our process was to judiciously filter these functions. We had to strike a balance to ensure that our final dataset comprises functions that are truly bioinformatics-focused and applicable to our study. This filtering process was thorough and meticulous, undertaken by experts with knowledge in bioinformatics, thereby reaffirming the indispensable role of a bioinformatics understanding in this work.

To ensure relevance, we executed a recursive search in scraping the entire Wikipedia corpus using 100 seed words that are known to be relevant to bioinformatics. Eventually, we manually filtered all the collected keywords to ensure relevance, which resulted in a dictionary of bioinformatics/adjacent words.

We then employed a two-level filtering process: (1) In the first round, each function and corresponding comments passed through a keyword filter, requiring at least 10 keyword matches related to bioinformatics. (2) In the following round, we leveraged GPT3.5 to generate associated keywords for each function, which underwent manual review. This strategy minimized the manual workload and helped us retain high-quality, relevant functions for our dataset.

Thus, our effort to benchmark code generation, although general in nature, is deeply rooted in the context of bioinformatics, making use of carefully curated and filtered datasets based on bioinformatics problems. While an understanding of bioinformatics and biology may not be essential for using the benchmark, the benchmark, and the dataset were built to reflect the complexity and domain specifics of bioinformatics.

## U SUMMARY AT BOTTOM RESULTS

Summary At Bottom results were omitted from Table 4 due to brevity reasons, and so are included here. More details about how Summary At Bottom prompts are constructed can be found in Appendix [11](#).

Model	Pass@1	Pass@5	Pass@10	Pass@20
InCoder-6B	0.610	2.587	4.303	6.274
SantaCoder-1.1B	0.510	1.949	3.013	4.459
StarCoder-15.5B	6.465	13.824	16.746	19.076
StarCoder-15.5B (finetuned)	\	\	\	\
StarCoder+	4.172	11.772	14.933	17.197
CodeGen-6B-mono	2.070	4.535	5.896	7.006
CodeGen-16B-mono	2.166	5.137	6.022	6.369
CodeGen2-7B	0.510	1.019	1.207	1.274
GPT-3.5-Turbo	13.439	20.040	22.460	25.478
GPT-4	6.316	15.272	20.223	24.823

Table 11: Performance on "Summary at Bottom" rows for Python columns

## V ERROR DISTRIBUTION GIVEN OVER ALL MODELS

The failure modes are similar between the models but not exactly the same.

Models	Syntax Error	Runtime Error	Timeout Error	Output Disagreement	Passed
Overall	117665	55351	4	8661	7982
CodeGen	11268	8176	1	148	105
CodeGen2	12687	6718	0	211	79
GPT3.5	9231	10603	0	5624	6643
InCoder	11268	8176	1	148	105
InstructCodeT5P	19667	33	0	0	0
SantaCoder	14391	4601	1	555	139
StarCoder	26233	10688	0	1660	808

Table 12: Error distribution per model.

Below is the failure breakdown for each model:

```

1
2 CodeGen-6B-Mono
3
4 Failure: Syntax Error = 11268
5 Failure: Runtime Error = 8176
6 Failure: Timeout Error = 1
7 Failure: Output Disagreement = 148
8 Passed Tests = 105
9
10 CodeGen2-7B
11
12 Failure: Syntax Error = 12687
13 Failure: Runtime Error = 6718
14 Failure: Output Disagreement = 211
15 Passed Tests = 79
16
17 GPT3.5-Turbo
18
19 Failure: Syntax Error = 9231
20 Failure: Runtime Error = 10603
21 Failure: Output Disagreement = 5624
22 Passed Tests = 6643
23
24 InCoder-6B
25
26 Failure: Syntax Error = 12777
27 Failure: Runtime Error = 6502
28 Failure: Timeout Error = 2
29 Failure: Output Disagreement = 309
30 Passed Tests = 100
31
32 InstructCodeT5P-16B
33
34 Failure: Syntax Error = 19667
35 Failure: Runtime Error = 33
36
37 SantaCoder
38
39 Failure: Syntax Error = 14391
40 Failure: Runtime Error = 4604
41 Failure: Timeout Error = 1
42 Failure: Output Disagreement = 555
43 Passed Tests = 139
44
45 StarCoder

```

```
46  
47 Failure: Syntax Error = 26233  
48 Failure: Runtime Error = 10688  
49 Failure: Output Disagreement = 1660  
50 Passed Tests = 808
```

## W MORE MODELS

Here we aim to answer the question of why we didn't mention ChatGPT/Codex/Github copilot as baselines.

ChatGPT/Codex/Github copilot are all powered by a GPT model. In our paper, we provide results for GPT-3.5 and GPT-4, which was the model that ChatGPT and Github Copilot both used. As for Codex, it uses a weaker GPT 3.0 model and was deprecated by OpenAI at the time of testing, so we did not include it in the benchmark.



## X IMPORTANCE OF THE CONTEXT

Imports and classes are predefined and are included in the context because, as we are testing function-level code generation, we are not prompting the model nor expecting the model to generate the classes it needs to pass the tests. Instead, we are testing the model's ability to extract the pertinent imports and classes from the context to use in the generated function.

To further illustrate how the prompt needs to contain information on the context to correctly generate the golden code, we provide an example below. You'll observe that the golden-code function uses the global variable "IGNORE\_GENE\_NAMES". We include this information in the prompt because it embeds the task's necessary foundations. Essentially, we ensure the prompt has enough context for an informed human to reproduce the function, implying that the function generated ought to incorporate the environment's existing resources (imports & classes) rather than generating new ones.

```

1 PROMPT:
2
3 This is in python.
4 Write a function called "unpipe_name" that takes in a string parameter
  called "name". The function takes a string containing multiple gene
  names separated by pipes, and returns a single gene name devoid of
  duplicates and pipe characters. The function checks if there are any
  duplicates, and removes any meaningless target names. If there are
  multiple gene names present, the function takes the longest name as
  the final name. If there are any ambiguous gene names, the function
  logs a warning and selects the longest name as the final name. The
  function should also import "cnvlib.params".
5
6 def unpipe_name(name):
7
8 Here are the imports:
9 import numpy as np
10 import logging
11 from . import params
12 from skgenome import tabio
13 Here are the global variables:
14 MIN_REF_COVERAGE = -5.0
15 MAX_REF_SPREAD = 1.0
16 NULL_LOG2_COVERAGE = -20.0
17 GC_MIN_FRACTION = 0.3
18 GC_MAX_FRACTION = 0.7
19 INSERT_SIZE = 250
20 IGNORE_GENE_NAMES = '-', '.', 'CGH'
21 ANTITARGET_NAME = 'Antitarget'
22 ANTITARGET_ALIASES = ANTITARGET_NAME, 'Background'
23 Here are the class declarations:
24 Here are the additional function declarations:
25 def do_import_picard(fname, too_many_no_coverage):
26     summary: Reads a file in 'picardhs' format, processes the data, and
  returns a modified dataframe.
27     param: fname (string) - the file name/path to be read in 'picardhs'
  format.
28     param: too_many_no_coverage (int) - if the number of bins with no
  coverage is greater than this value, a warning message is logged.
  Default is 100.
29     return: garr (pandas dataframe) - a modified dataframe with added
  columns 'gene' and 'log2' based on the original dataframe read from
  the input file.
30 def unpipe_name(name):
31     summary: Remove duplicate gene names and pipe characters from a given
  string.
32     param: name (str) - the string containing the gene names.
33     return: new_name (str) - the cleaned gene name string.
34 def do_import_theta(segarr, theta_results_fname, ploidy):

```

```

35     summary: A function for importing theta results and estimating copy
36     number and log2 ratios of segments.
37     param: segarr (numpy array) - array of segments
38     param: theta_results_fname (str) - name of theta results file
39     param: ploidy (int) - ploidy of genome (default is 2)
40     return: generator of numpy arrays - array of segments with estimated
41     copy number and log2 ratios.
42 def parse_theta_results(fname):
43     summary: Parses THetA results into a data structure with NLL, mu, C,
44     and p* columns.
45     param: fname (str) - name of the file to parse the results from
46     return: (dict) - a dictionary containing the NLL, mu_normal,
47     mu_tumors, C, and p* values
48 Here are the comments and the specs:
49 Write a function called "unpipe_name" that takes in a string parameter
50 called "name". The function takes a string containing multiple gene
51 names separated by pipes, and returns a single gene name devoid of
52 duplicates and pipe characters. The function checks if there are any
53 duplicates, and removes any meaningless target names. If there are
54 multiple gene names present, the function takes the longest name as
55 the final name. If there are any ambiguous gene names, the function
56 logs a warning and selects the longest name as the final name. The
57 function should also import "cnvlib.params".
58
59 def unpipe_name(name):
60     GOLDEN CODE:
61
62     def unpipe_name(name):
63         """Fix the duplicated gene names Picard spits out.
64
65         Return a string containing the single gene name, sans duplications
66         and pipe
67         characters.
68
69         Picard CalculateHsMetrics combines the labels of overlapping
70         intervals
71         by joining all labels with '|', e.g. 'BRAF|BRAF' -- no two distinct
72         targeted genes actually overlap, though, so these dupes are redundant
73         .
74         Meaningless target names are dropped, e.g. 'CGH|FOO|-' resolves as '
75         FOO'.
76         In case of ambiguity, the longest name is taken, e.g. "TERT|TERT
77         Promoter"
78         resolves as "TERT Promoter".
79         """
80         if '|' not in name:
81             return name
82         gene_names = set(name.split('|'))
83         if len(gene_names) == 1:
84             return gene_names.pop()
85         cleaned_names = gene_names.difference(IGNORE_GENE_NAMES)
86         if cleaned_names:
87             gene_names = cleaned_names
88         new_name = sorted(gene_names, key=len, reverse=True)[0]
89         if len(gene_names) > 1:
90             logging.warning('WARNING: Ambiguous gene name %r; using %r', name
91             ,
92             new_name)
93         return new_name

```

## Y POTENTIAL CONCERNS ABOUT CHANGES IN PACKAGES

In addressing potential concerns about code generation issues due to changes in packages, we have taken steps to ensure stability and consistency in our testing environment. Our testing environment utilizes standardized and version-locked packages that guarantee stable results for our "golden" code samples. As such, we do not need to worry about package updates or feature deprecation.

Furthermore, when giving prompts, we have taken care to specifically mention the packages that need to be used, which guides the LLM to correctly utilize the expected versions of these packages. Any remaining discrepancies will be highlighted in our error analysis for the respective LLMs.

In the prompts we provide, we clearly specify which packages and functions should be used, including any external dependencies. This way, the LLM knows exactly which packages to use.

Take, for example, this dataset-generated prompt in Python:

```
1 #This is in python
2
3 #write a function called "binary_erosion" that takes in two parameters: "
  x_data" and "structuring_element". The function should first check if
  the "structuring_element" is a 2D array or not. If it is, and "
  x_data" is not a 2D array, the function should create a new array
  called "y_data" that is the same size as "x_data" and contains only
  zeros. The function should then loop through each slice of "x_data"
  and apply the skimage.morphology.binary_erosion function using the "
  structuring_element" as the structuring element. The result of each
  slice should be added to the corresponding slice of "y_data". Finally
  , the function should return "y_data". If the "structuring_element"
  is not 2D and "x_data" is 2D, the function should raise a
  NotImplementedError with the message "A 3D structuring element cannot
  be applied to a 2D image." Otherwise, the function should simply
  apply the skimage.morphology.binary_erosion function to "x_data"
  using "structuring_element" as the structuring element and return the
  result.
4 #
5 #def binary_erosion(x_data, structuring_element):
```

You can observe how the prompt meticulously guides the LLM to employ the "binary\_erosion" function derived from the "skimage.morphology package". Any inaccuracies in using the specified packages, likely due to the model training on outdated or more advanced versions, would deem its output incorrect, leading us to categorize it as a failed generation. A paramount facet we are evaluating is the proficiency of the LLM in accurately implementing specified external packages as mentioned in the prompts.

## Z DIFFERENCES BETWEEN PUBLIC, HIDDEN, AND SIMILAR SETS

While constructing the datasets, we observed the need for a categorization that simultaneously caters to present-day language learning models (LLMs) and future, more capable versions. Consequently, we delineated our public and hidden datasets, which admittedly, resulted in a considerable divergence between their summary statistics.

The public test set encapsulates relatively moderate challenges an LLM might encounter, keeping in mind the token limits of current-generation code LLMs, which range from 2,048 to 4,096 tokens. This constraint dictated that the public test set should comprise smaller and simpler functions.

Conversely, the hidden dataset was configured to house a broader, more challenging set of problems, aiming to benchmark future models, for instance, GPT4 with its anticipated 32K token limit. This two-tiered framework ensures the longevity of this benchmark as advancements unfold in the field of LLMs and facilitates additional investigations into domain-specific generative models.

Moreover, we have devised a "Similar Dataset", which is essentially a subset of the hidden dataset harboring statistics comparable to the public dataset. This new dataset ensures direct, fair comparisons between the public and the hidden tests.

The "public data" represents datasets with crafted specific contexts and corresponding test cases. The "hidden data" encompasses a wider array of intricate issues. Furthermore, the "similar data" is a subset of the hidden data, curated to mimic the overall distribution of the public data. This set includes an additional 157 Python problems and 50 Java problems, maintaining the same 253 Rosalind problem tally, thus reflecting the composition of the public data. The issues in this set were cherry-picked based on their L2 similarity norm of calculated statistics, ensuring close alignment to the mean distribution of the public data.

In addition, the reason why the number of hidden tests is much larger than public tests is that we aim to ensure the integrity of our benchmark. Thus, we divided our dataset into a much larger hidden set rather than a public set. We believe that access to too many public test cases might lead to potential overfitting while fine-tuning models, which would fundamentally undermine the benchmark.

## AA PROMPT STRUCTURE ANALYSIS

As demonstrated by the scatterplots in Appendix [K](#), we can see that there is a general negative correlation between the length of prompt and the performance of the model. We can also plot the number of models that perform the best for each prompt type as follows:

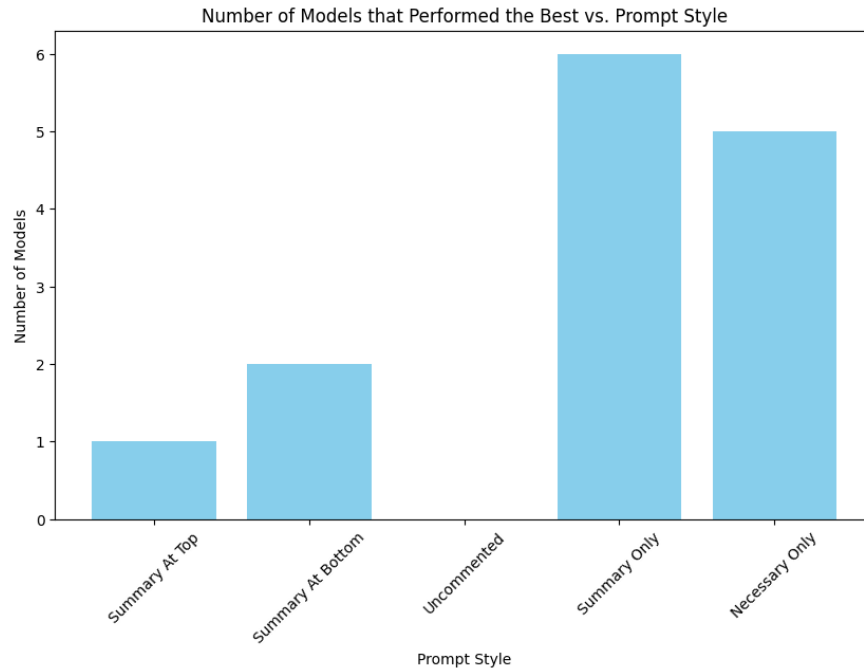


Figure 19: Number of best performing models over all prompt styles

Judging from Figure [19](#), we can see that overall, the prompts with the fewest tokens, i.e. the "Summary Only" and "Necessary Only" prompt styles, had the most models that performed the best, however there are some models that performed better with the longer "Summary At Top" and "Summary At Bottom" prompts. In this section, we will analyze some of the possible reasons for these discrepancies and variations in the results by looking at prompt structure. Consider the following "Summary Only" and "Necessary Only" prompts for the following "binary\_erosion" function:

```

1 def binary_erosion(x_data, structuring_element):
2     is_strel_2d = structuring_element.ndim == 2
3     is_img_2d = x_data.ndim == 2
4     if is_strel_2d and not is_img_2d:
5         y_data = numpy.zeros_like(x_data)
6         for index, plane in enumerate(x_data):
7             y_data[index] = skimage.morphology.binary_erosion(plane,
8                 structuring_element)
9         return y_data
10    if not is_strel_2d and is_img_2d:
11        raise NotImplementedError(
12            'A 3D structuring element cannot be applied to a 2D image.')
13    y_data = skimage.morphology.binary_erosion(x_data,
14        structuring_element)
15    return y_data

```

Here is the "Summary Only" prompt:

```

1 #This is in python
2 #write a function called "binary_erosion" that takes in two parameters: "
  x_data" and "structuring_element". The function should first check if
  the "structuring_element" is a 2D array or not. If it is, and "

```

```

x_data" is not a 2D array, the function should create a new array
called "y_data" that is the same size as "x_data" and contains only
zeros. The function should then loop through each slice of "x_data"
and apply the skimage.morphology.binary_erosion function using the "
structuring_element" as the structuring element. The result of each
slice should be added to the corresponding slice of "y_data". Finally
, the function should return "y_data". If the "structuring_element"
is not 2D and "x_data" is 2D, the function should raise a
NotImplementedError with the message "A 3D structuring element cannot
be applied to a 2D image." Otherwise, the function should simply
apply the skimage.morphology.binary_erosion function to "x_data"
using "structuring_element" as the structuring element and return the
result.
3 #
4 #def binary_erosion(x_data, structuring_element):

```

and here is the "Necessary Only" prompt

```

1 Write a function with the following specs:
2 --specs begin here--
3 #write a function called "binary_erosion" that takes in two parameters: "
  x_data" and "structuring_element". The function should first check if
  the "structuring_element" is a 2D array or not. If it is, and "
  x_data" is not a 2D array, the function should create a new array
  called "y_data" that is the same size as "x_data" and contains only
  zeros. The function should then loop through each slice of "x_data"
  and apply the skimage.morphology.binary_erosion function using the "
  structuring_element" as the structuring element. The result of each
  slice should be added to the corresponding slice of "y_data". Finally
  , the function should return "y_data". If the "structuring_element"
  is not 2D and "x_data" is 2D, the function should raise a
  NotImplementedError with the message "A 3D structuring element cannot
  be applied to a 2D image." Otherwise, the function should simply
  apply the skimage.morphology.binary_erosion function to "x_data"
  using "structuring_element" as the structuring element and return the
  result.
4 param: x_data (numpy.ndarray) - input data to be eroded.
5 param: structuring_element (numpy.ndarray) - structuring element for
  erosion.
6 return: y_data (numpy.ndarray) - eroded data.
7 --specs end here--
8 Note the function will be embedded in the following context
9 --context begins here--
10 import numpy
11 import skimage.morphology
12 import os
13 numpy.random.seed(<|int;range=0,100|>)
14 <<insert solution here>>
15 def main():
16     x_data = numpy.random.randint(2, size=(10, 10))
17     structuring_element = skimage.morphology.square(3)
18     print(binary_erosion(x_data, structuring_element))
19 if __name__ == "__main__":
20     main()
21 --context ends here--
22 Make sure to only generate the function and not any of the context. Make
  sure you are generating valid, runnable code. Begin your solution
  with:
23 def binary_erosion(x_data, structuring_element):
24 MAKE SURE TO INDENT THE BODY OF YOUR FUNCTION BY A TAB

```

From the structure of these prompts, we can hypothesize why some of the simpler models perform better with the "Summary Only" prompts, while the GPT models perform better generally with the "Necessary Only" prompts. Since the "Necessary Only" prompts includes the GPT-generated summary of the function embedded inside the prompt, while including additional context, it is

reasonable to hypothesize that the simpler models such as InCoder, SantaCoder, StarCoder, or CodeGen are simply unable to digest the context information embedded inside the "Necessary Only" prompts, and instead the context information ends up being regarded as noise that just confuses the simpler models resulting in poorer performance. However, the larger models, such as the GPT-3.5 and GPT-4 models released by OpenAI, are able to properly incorporate the context information provided in the "Necessary Only" prompts, and as a result, they perform significantly better than their smaller counterparts with the additional context information.

It is also worth looking at the structure of the "Uncommented" prompt style, to see why these prompts perform so poorly when compared to the commented version of the prompts. For instance here is the "Uncommented" prompt style for the "binary\_erosion" problem.

```

1 This is in python
2 write a function called "binary_erosion" that takes in two parameters: "
  x_data" and "structuring_element". The function should first check if
  the "structuring_element" is a 2D array or not. If it is, and "
  x_data" is not a 2D array, the function should create a new array
  called "y_data" that is the same size as "x_data" and contains only
  zeros. The function should then loop through each slice of "x_data"
  and apply the skimage.morphology.binary_erosion function using the "
  structuring_element" as the structuring element. The result of each
  slice should be added to the corresponding slice of "y_data". Finally
  , the function should return "y_data". If the "structuring_element"
  is not 2D and "x_data" is 2D, the function should raise a
  NotImplementedError with the message "A 3D structuring element cannot
  be applied to a 2D image." Otherwise, the function should simply
  apply the skimage.morphology.binary_erosion function to "x_data"
  using "structuring_element" as the structuring element and return the
  result.
3
4 def binary_erosion(x_data, structuring_element):
5
6 Here are the imports:
7 import skimage.morphology
8 import scipy.ndimage
9 import numpy
10 Here are the global variables:
11 Here are the class declarations:
12 Here are the additional function declarations:
13 def dilation(x_data, structuring_element):
14     summary: Performs dilation on input image data with a structuring
        element
15     param: x_data (numpy array) - input image data to perform dilation on
16     param: structuring_element (numpy array) - structuring element for
        the dilation operation
17     return: y_data (numpy array) - resulting dilated image data
18 def erosion(x_data, structuring_element):
19     summary: Performs erosion operation on input data using a structuring
        element.
20     param: x_data (numpy.ndarray) - input data to apply erosion on.
21     param: structuring_element (numpy.ndarray) - structuring element used
        for erosion operation.
22     return: y_data (numpy.ndarray) - erosion result as a 2D or 3D numpy
        array.
23 def binary_erosion(x_data, structuring_element):
24     summary: Performs binary erosion on input data using a structuring
        element.
25     param: x_data (numpy.ndarray) - input data to be eroded.
26     param: structuring_element (numpy.ndarray) - structuring element for
        erosion.
27     return: y_data (numpy.ndarray) - eroded data.
28 def morphological_gradient(x_data, structuring_element):
29     summary: Computes the morphological gradient of an image.
30     param: x_data (numpy array) - input data.

```

```

31     param: structuring_element (numpy array) - structuring element used
      for the operation.
32     return: y_data (numpy array) - output data.
33 Here are the comments and the specs:
34 write a function called "binary_erosion" that takes in two parameters: "
      x_data" and "structuring_element". The function should first check if
      the "structuring_element" is a 2D array or not. If it is, and "
      x_data" is not a 2D array, the function should create a new array
      called "y_data" that is the same size as "x_data" and contains only
      zeros. The function should then loop through each slice of "x_data"
      and apply the skimage.morphology.binary_erosion function using the "
      structuring_element" as the structuring element. The result of each
      slice should be added to the corresponding slice of "y_data". Finally
      , the function should return "y_data". If the "structuring_element"
      is not 2D and "x_data" is 2D, the function should raise a
      NotImplementedError with the message "A 3D structuring element cannot
      be applied to a 2D image." Otherwise, the function should simply
      apply the skimage.morphology.binary_erosion function to "x_data"
      using "structuring_element" as the structuring element and return the
      result.
35 def binary_erosion(x_data, structuring_element):

```

and here is the corresponding commented version of the prompt:

```

1 #This is in python
2 #write a function called "binary_erosion" that takes in two parameters: "
      x_data" and "structuring_element". The function should first check if
      the "structuring_element" is a 2D array or not. If it is, and "
      x_data" is not a 2D array, the function should create a new array
      called "y_data" that is the same size as "x_data" and contains only
      zeros. The function should then loop through each slice of "x_data"
      and apply the skimage.morphology.binary_erosion function using the "
      structuring_element" as the structuring element. The result of each
      slice should be added to the corresponding slice of "y_data". Finally
      , the function should return "y_data". If the "structuring_element"
      is not 2D and "x_data" is 2D, the function should raise a
      NotImplementedError with the message "A 3D structuring element cannot
      be applied to a 2D image." Otherwise, the function should simply
      apply the skimage.morphology.binary_erosion function to "x_data"
      using "structuring_element" as the structuring element and return the
      result.
3 #
4 #def binary_erosion(x_data, structuring_element):
5 #
6 #Here are the imports:
7 #import skimage.morphology
8 #import scipy.ndimage
9 #import numpy
10 #Here are the global variables:
11 #Here are the class declarations:
12 #Here are the additional function declarations:
13 #def dilation(x_data, structuring_element):
14 #     summary: Performs dilation on input image data with a structuring
      element
15 #     param: x_data (numpy array) - input image data to perform dilation on
16 #     param: structuring_element (numpy array) - structuring element for
      the dilation operation
17 #     return: y_data (numpy array) - resulting dilated image data
18 #def erosion(x_data, structuring_element):
19 #     summary: Performs erosion operation on input data using a structuring
      element.
20 #     param: x_data (numpy.ndarray) - input data to apply erosion on.
21 #     param: structuring_element (numpy.ndarray) - structuring element used
      for erosion operation.

```



```

22 # return: y_data (numpy.ndarray) - erosion result as a 2D or 3D numpy
    array.
23 #def binary_erosion(x_data, structuring_element):
24 # summary: Performs binary erosion on input data using a structuring
    element.
25 # param: x_data (numpy.ndarray) - input data to be eroded.
26 # param: structuring_element (numpy.ndarray) - structuring element for
    erosion.
27 # return: y_data (numpy.ndarray) - eroded data.
28 #def morphological_gradient(x_data, structuring_element):
29 # summary: Computes the morphological gradient of an image.
30 # param: x_data (numpy array) - input data.
31 # param: structuring_element (numpy array) - structuring element used
    for the operation.
32 # return: y_data (numpy array) - output data.
33 #Here are the comments and the specs:
34 #write a function called "binary_erosion" that takes in two parameters: "
    x_data" and "structuring_element". The function should first check if
    the "structuring_element" is a 2D array or not. If it is, and "
    x_data" is not a 2D array, the function should create a new array
    called "y_data" that is the same size as "x_data" and contains only
    zeros. The function should then loop through each slice of "x_data"
    and apply the skimage.morphology.binary_erosion function using the "
    structuring_element" as the structuring element. The result of each
    slice should be added to the corresponding slice of "y_data". Finally
    , the function should return "y_data". If the "structuring_element"
    is not 2D and "x_data" is 2D, the function should raise a
    NotImplementedError with the message "A 3D structuring element cannot
    be applied to a 2D image." Otherwise, the function should simply
    apply the skimage.morphology.binary_erosion function to "x_data"
    using "structuring_element" as the structuring element and return the
    result.
35 #def binary_erosion(x_data, structuring_element):

```

Comparing the two prompt styles, we can see the only difference is the inclusion of the "#" symbol in front of each line in the commented version of the line. These "#" symbols seem to have a large effect on LLM code generation. For instance, here is the a sample ChatGPT code generation after being passed in a "Uncommented" prompt:

```

1 Here is the Python function as you described:
2
3 '''Python
4 import numpy as np
5 from skimage.morphology import binary_erosion
6
7 def binary_erosion(x_data, structuring_element):
8
9     # Check if structuring_element is 2D
10    if len(structuring_element.shape) == 2:
11        # Check if x_data is not 2D
12        if len(x_data.shape) != 2:
13            # Create y_data (same shape as x_data, filled with zeros)
14            y_data = np.zeros_like(x_data)
15
16            # Apply binary_erosion slice by slice
17            for idx in range(x_data.shape[0]):
18                y_data[idx] = binary_erosion(x_data[idx],
19                structuring_element)
20
21            return y_data
22
23    else:
24        # Apply binary_erosion directly
25        return binary_erosion(x_data, structuring_element)

```

```

26     else:
27         # Check if x_data is 2D
28         if len(x_data.shape) == 2:
29             # Raise error
30             raise NotImplementedError("A 3D structuring element cannot be
applied to a 2D image.")
31
32         else:
33             # Apply binary_erosion directly
34             return binary_erosion(x_data, structuring_element)
35 """
36 If 2D structuring element is provided and x_data isn't 2D, function
applies the skimage.morphology.binary_erosion function to each slice
of x_data separately. Then adds result to y_data. Otherwise, if
structuring_element isn't 2D and x_data is 2D, it raises an error. If
both conditions don't apply, it directly applies the binary_erosion
function to x_data.

```

and here is a sample ChatGPT code generation after being passed in a corresponding commented prompt:

```

1 import skimage.morphology
2 import numpy as np
3
4 def binary_erosion(x_data, structuring_element):
5     if len(structuring_element.shape) == 2:
6         if len(x_data.shape) != 2:
7             y_data = np.zeros_like(x_data)
8             for i in range(x_data.shape[0]):
9                 y_data[i] = skimage.morphology.binary_erosion(x_data[i],
structuring_element)
10            return y_data
11
12        elif len(structuring_element.shape) != 2 and len(x_data.shape) == 2:
13            raise NotImplementedError("A 3D structuring element cannot be
applied to a 2D image.")
14
15        else:
16            return skimage.morphology.binary_erosion(x_data,
structuring_element)

```

We can see that the models respond far better to the commented prompt, only returning runnable Python code without wrapping it unnecessary text that may result in runtime errors in the testing environment. Overall, it is not just prompt length that has a tangible effect on LLM code generation performance. For instance, the slightly longer "Necessary Only" prompts can outperform the shorter "Summary Only" prompts for larger more complex models that can properly digest and understand the given code context. Furthermore, the inclusion of certain symbols, such as the presence of a "#" at the beginning of every line of the commented prompts, can help prompt models to generate more desirable code outputs.