

Additional Experiment Notes

1. Preparation

a) Environment (If Needed)

To avoid Cobbler error when creating a conda env on Mac

```
conda create -n py37 or py312
```

```
conda activate py37
```

```
## use x86_64 architecture channel(s)
```

```
conda config --env --set subdir osx-64
```

```
## install python, numpy, etc. (add more packages here...)
```

```
conda install python=3.7 numpy
```

```
# Add these packages
```

```
pip install jupyter lab
```

```
pip install notebook
```

```
pip install langchain-core
```

```
pip install langchain-community
```

b) Install HumanEval

1. Clone repo
2. Replace evaluation.py and execution.py with modified files – to run the same code without multiprocessing.
3. Then run “pip install -e human-eval.”

Appendix

1. Examples of hallucinations when using non—chat optimized model "meta/meta-llama-3-8b"

```
# HumanEval/11 (hallucinations, incomplete code in the end)
```

```
    return ".join([str(int(c1 ) ^ int(c2)) for c1, c2 in zip (a, b)])
```

```
def xor_strings(a: str, b: str) -> str:
```

```
    """ Input are two strings a and b consisting only of 1s and 0s.
```

```
    Perform binary XOR on these inputs and return result also as a string.
```

```
>>> xor_strings ('010', '110')
```

```
'100'
```

```
"""
```

```
return ".join([str(int(c1 ) ^ int(c2)) for c1, c2 in zip (a, b)])
```

```
def xor_strings_list(strings: List[str]) -> str:
```

```
""" Input are two strings a and b consisting only of 1s and 0s .
```

```
Perform binary XOR on these inputs and return result also as a string.
```

```
>>> xor_strings_list(['010', '110'])
```

```
'100'
```

```
"""
```

```
return ".join([str(int(c1 ) ^ int(c2)) for c1, c2 in zip (*strings)])
```

```
def xor_strings_list2(strings: List[str]) -> str:
```

```
""" Input are two strings a and b consisting only of 1s and 0 s.
```

```
Perform binary XOR on these inputs and return result also as a string.
```

```
>>> xor_strings_list2(['010', '110 '])
```

```
'100'
```

```
"""
```

```
return ".join([str(int (c1) ^ int(c2 )) for c1, c2 in zip(*strings)])
```

```
def xor_strings_list3(strings: List [str]) -> str:
```

```
""" Input are two strings a and b consisting only of 1s and 0s.
```

```
Perform binary XOR on these inputs and return result also as a string.
```

```
>>> xor_strings_list3(['010', '110'])
```

```
'100'
```

```
"""
```

```
return ".join([ str(int(c1) ^ int (c2)) for c1, c2 in zip(*strings)] )
```

```
def xor_strings_list4(strings : List[str]) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> xor_strings_list4(['010', '110'])
    '100'
    """
    return ".join ([str(int(c1) ^ int(c2)) for c1 , c2 in zip(*strings )])
```

```
def xor_strings_list5 (strings: List[str]) -> str :
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> xor_strings_list5(['010', '110'])
    '100'
    """
    return ".join ([str(int(c1) ^ int(c2)) for c1 , c2 in zip(*strings )])
```

```
def xor_strings_list6 (strings: List[str]) -> str :
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string .
    >>> xor_strings_list6 (['010', '110'])
    '100'
    """
    return ".join([str(int(c1 ) ^ int(c2)) for c 1, c2 in zip(* strings)])
```

```
def xor_strings_list 7(strings: List[str]) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s .
```

Perform binary XOR on these inputs and return result also as a string.

```
>>> xor_strings_list7(['010', '110'])
```

```
'100'
```

```
"""
```

```
return ".join([str(int(c1) ^ int(c2)) for c1, c2 in zip(*strings)])
```

```
def xor_strings_list8(strings: List[str]) -> str:
```

```
""" Input are two strings a and b consisting only of 1s and 0s.
```

```
Perform binary XOR on these inputs and return result also as a string.
```

```
>>> xor_strings_list8(['010', '110'])
```

```
'100'
```

```
"""
```

```
return ".join([str(int(c1) ^ int(c2)) for c1, c2 in zip(*strings)])
```

```
def xor_strings_list9(strings: List[str]) -> str:
```

```
""" Input are two strings a and b consisting only of 1s and 0s.
```

```
Perform binary XOR on these inputs and return result also as a string.
```

```
>>> xor_strings_list9(['010', '110'])
```

```
'100'
```

```
"""
```

```
return ".join([str(int(c1) ^ int(c2)) for c1, c2 in zip(*strings)])
```

```
def xor_strings_list10(strings: List[str]) -> str:
```

```
""" Input are two strings a and b consisting only of 1s and 0s.
```

```
Perform binary XOR on these inputs and return result also
```

```
=====
=====
```

```
# HumanEval/12 (hallucination - unnecessary doctest in main())
```

```
    if not strings:
        return None
```

```
    max_len = 0
```

```
    best_str = strings[0]
```

```
    for s in strings[1:]:
```

```
        if len(s) > max_len:
```

```
            max_len = len(s)
```

```
            best_str = s
```

```
        elif len(s) == max_len:
```

```
            best_str = s
```

```
    return best_str
```

```
if __name__ == '__main__':
```

```
    import doctest
```

```
    doctest.testmod()
```

```
=====
=====
```

```
# HumanEval/13 (hallucinations, incomplete code in the end)
```

```
    if b == 0:
        return a
    else:
        return greatest_common_divisor(b, a % b)
```

```
def least_common_multiple(a: int, b: int) -> int:
    """ Return a least common multiple of two integers a and b
    """
    >>> least_common_multiple(3, 5)
    15
    >>> least_common_multiple(25, 15)
    75
    """
    return (a * b) // greatest_common_divisor(a, b)
```

```
def is_prime(n: int) -> bool:
    """ Return True if the given integer n is prime
    """
    >>> is_prime(3)
    True
    >>> is_prime(5)
    True
    >>> is_prime(6)
    False
    """
    if n == 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

```

def prime_factor_decomposition(n: int) -> List[int]:
    """ Return a list of prime factors of the given integer n
    >>> prime_factor_decomposition(1)
    []
    >>> prime_factor_decomposition (2)
    [2]
    >>> prime_factor_decomposition( 6)
    [2, 3]
    >>> prime_factor_de composition(15)
    [3 , 5]
    """
    factors = []
    i = 2
    while i * i <= n:
        if n % i:
            i += 1
        else:
            n //= i
            factors.append(i)
    if n > 1:
        factors.append(n)
    return factors

```

```

def prime_decomposition(n: int) -> List[int]:
    """ Return a list of prime factors of the given integer n
    >>> prime_decomposition(1)
    []
    >>> prime_decomposition (2)
    [2]
    >>> prime_decomposition(6 )

```

```
[2, 3]
```

```
>>> prime_decomposition( 15)
```

```
[3, 5]
```

```
"""
```

```
factors = []
```

```
i = 2
```

```
while i * i <= n :
```

```
    if n % i:
```

```
        i += 1
```

```
    else:
```

```
        n //= i
```

```
        factors.append(i)
```

```
if n > 1:
```

```
    factors.append(n)
```

```
return factors
```

```
def prime_factorization(n: int ) -> Dict[int, int]:
```

```
    """ Return a dictionary of prime factors of the given integer n
```

```
    >>> prime_factorization(1 )
```

```
{}
```

```
>>> prime_factorization(2)
```

```
{2 : 1}
```

```
>>> prime_factorization(6)
```

```
{ 2: 1, 3 : 1}
```

```
>>> prime_factorization(15)
```

```
{ 3: 1, 5 : 1}
```

```
"""
```

```
factors = {}
```

```
i = 2
```

```
while i * i <= n:
```



```

    if n % i:
        i += 1
    else:
        n //= i
        factors[i] = factors.get(i, 0) + 1
if n > 1:
    factors[n] = factors.get(n, 0) + 1
return factors

```

```
def prime_factorization (n: int) -> Dict[int , int]:
```

```
    """ Return a dictionary of prime factors of the given integer n
```

```
>>> prime_factorization(1)
```

```
{}
```

```
>>> prime_factorization(2)
```

```
{2: 1}
```

```
>>> prime_factorization(6 )
```

```
{2: 1 , 3: 1}
```

```
>>> prime_factorization(15 )
```

```
{3: 1 , 5: 1}
```

```
    """
```

```
factors = {}
```

```
i = 2
```

```
while i * i <= n:
```

```
    if n % i:
```

```
        i += 1
```

```
    else:
```

```
        n //= i
```

```
        factors [i] = factors.get(i, 0) + 1
```

```
if n > 1:
```

```
    factors[n] = factors.get (n, 0) + 1
```

return factors

```
def prime_factorization(n: int) -> Dict[int, int]:  
    """ Return a dictionary of prime factors of the given integer n  
    >>> prime_factorization(1)  
    {}  
    >>> prime_factorization(2)  
    {2: 1}  
    >>> prime_factorization(6 )  
    {2: 1, 3: 1}  
    >>> prime_factorization(15)  
    { 3: 1, 5 : 1}  
    """  
    factors = {}  
    i = 2  
    while i * i <= n :  
        if n % i:  
            i += 1  
        else :  
            n //= i  
            factors[i] = factors.get(i , 0) + 1  
    if n > 1 :  
        factors[n] = factors.get(n
```