Project work: sequence analysis

Sequence Analysis with Python

The following assignments introduce applications of hashing with `dict()` primitive of Python. While doing so, a rudimentary introduction to biological sequences is given. This framework is then enhanced with probabilities, leading to routines to generate random sequences under some constraints, including a general concept of *Markov-chains*. All these components illustrate the usage of `dict()`, but at the same time introduce some other computational routines to efficiently deal with probabilities.
The function `collections.defaultdict` can be useful.

Below are some "suggested" imports. Feel free to use and modify these, or not. Generally it's good practice to keep most or all imports in one place. Typically very close to the start of notebooks.

```
from collections import defaultdict
from itertools import product

import numpy as np
from numpy.random import choice
```

The automated TMC tests do not test cell outputs. These are intended to be evaluated in the peer reviews. So it is still be a good idea to make the outputs as clear and informative as possible.

To keep TMC tests running as well as possible it is recommended to keep global variable assignments in the notebook to a minimum to avoid potential name clashes and confusion. Additionally you should keep all actual code exection in main guards to keep the test running smoothly. If you run check_sequence.py in the `part07-e01_sequence_analysis` folder, the script should finish very quickly and optimally produce no output.

If you download data from the internet during execution (codon usage table), the parts where downloading is done should not work if you decide to submit to the tmc server. Local tests should work fine.

DNA and RNA

A DNA molecule consist, in principle, of a chain of smaller molecules. These smaller molecules have some common basic components (bases)

that repeat. For our purposes it is sufficient to know that these bases are nucleotides adenine, cytosine, guanine, and thymine with abbreviations A, C, G, and T. Given a *DNA sequence* e.g. ACGATGAGGCTCAT, one can reverse engineer (with negligible loss of information) the corresponding DNA molecule.

Parts of a DNA molecule can *transcribe* into an RNA molecule. In this process, thymine gets replaced by uracil (U).

1. Write a function `dna_to_rna` to convert a given DNA sequence `s` into an RNA sequence. For the sake of exercise, use `dict()` to store the symbol to symbol encoding rules. Create a program to test your function.

```
def dna_to_rna(s):
    return "".join("U" for _ in s)

if __name__ == '__main__':
    print(dna_to_rna("AACGTGATTTC"))
```

```
UUUUUUUUUUU
```

Idea of solution

fill in

Discussion

fill in

Proteins

Like DNA and RNA, protein molecule can be interpreted as a chain of smaller molecules, where the bases are now amino acids. RNA molecule may *translate* into a protein molecule, but instead of base by base, three bases of RNA correspond to one base of protein. That is, RNA sequence is read triplet (called codon) at a time.

2. Consider the codon to amino acid conversion table in http://htmlpreview.github.io/?https://github.com/csmastersUH/data_analysis_with_python_2020/blob/master/Codon%20usage%20table.html Write a function `get_dict` to read the table into a `dict()`, such that for each RNA sequence of length 3, say AGU, the hash table stores the conversion rule to the corresponding amino acid. You may store the html page to your local src directory, and parse that file.

```
def get_dict():
    return {}

if __name__ == '__main__':
    codon_to_aa = get_dict()
    print(codon_to_aa)
```

```
{}
```

Idea of solution

fill in

Discussion

fill in

3. Use the same conversion table as above, but now write function `get_dict_list` to read the table into a `dict()`, such that for each amino acid the hash table stores the list of codons encoding it.

```
def get_dict_list():
    return {}

if __name__ == '__main__':
    aa_to_codons = get_dict_list()
    print(aa_to_codons)
```

```
{}
```

Idea of solution

fill in

Discussion

fill in

With the conversion tables at hand, the following should be trivial to solve.

4. Fill in function `rna_to_prot` in the stub solution to convert a given DNA sequence `s` into a protein sequence. You may use the dictionaries from exercises 2 and 3. You can test your program with `ATGATATCATCGACGATGTAG`.

```
def rna_to_prot(s):
    return ""

def dna_to_prot(s):
    return rna_to_prot(dna_to_rna(s))

if __name__ == '__main__':
    print(dna_to_prot("ATGATATCATCGACGATGTAG"))
```

Idea of solution

fill in

Discussion

fill in

You may notice that there are $4^3 = 64$ different codons, but only 20 amino acids. That is, some triplets encode the same amino acid.

Reverse translation

It has been observed that among the codons coding the same amino acid, some are more frequent than others. These frequencies can be converted to probabilities. E.g. consider codons AUU, AUC, and AUA that code for amino acid isoleucine. If they are observed, say, 36, 47, 17 times, respectively, to code isoleucine in a dataset, the probability that a random such event is AUU $\rightarrow$ isoleucine is 36/100.

This phenomenon is called *codon adaptation*, and for our purposes it works as a good introduction to generation of random sequences under constraints.

5. Consider the codon adaptation frequencies in http://htmlpreview.github.io/?https://github.com/csmastersUH/data_analysis_with_python_2020/blob/master/Codon%20usage%20table.html and read them into a `dict()`, such that for each RNA sequence of length 3, say AGU, the hash table stores the probability of that codon among codons encoding the same amino acid. Put your solution in the `get_probabability_dict` function. Use the column "([number])" to estimate the probabilities, as the two preceding columns contain truncated values.

```
def get_probabability_dict():
    return {}

if __name__ == '__main__':
    codon_to_prob = get_probabability_dict()
    items = sorted(codon_to_prob.items(), key=lambda x: x[0])
    for i in range(1 + len(items)//6):
        print("\t".join(
            f"{k}: {v:.6f}"
            for k, v in items[i*6:6+i*6]
        ))
```

Idea of solution

fill in

Discussion

fill in

Now you should have everything in place to easily solve the following.

6. Write a class `ProteinToMaxRNA` with a `convert` method which converts a protein sequence into the most likely RNA sequence to be the source of this protein. Run your program with `LTPIQNRA`.

```
class ProteinToMaxRNA:

    def __init__(self):
        pass

    def convert(self, s):
        return ""


if __name__ == '__main__':
    protein_to_rna = ProteinToMaxRNA()
    print(protein_to_rna.convert("LTPIQNRA"))
```

Idea of solution

fill in

Discussion

fill in

Now we are almost ready to produce random RNA sequences that code a given protein sequence. For this, we need a subroutine to *sample from a probability distribution*. Consider our earlier example of probabilities 36/100, 47/100, and 17/100 for AUU, AUC, and AUA, respectively. Let us assume we have a random number generator `random()` that returns a random number from interval $[0, 1)$. We may then partition the unit interval according to cumulative probabilities to $[0, 36/100)$, $[36/100, 83/100)$, $[83/100, 1)$, respectively. Depending which interval the number `random()` hits, we select the codon accordingly.

7. Write a function `random_event` that chooses a random event, given a probability distribution (set of events whose probabilities sum to 1). You can use function `random.uniform` to produce values uniformly at random from the range $[0, 1)$. The distribution should be given to your function as a dictionary from events to their probabilities.

```
def random_event(dist):
    """
    Takes as input a dictionary from events to their probabilities.
    Return a random event sampled according to the given distribution.
    The probabilities must sum to 1.0
    """
    return next(iter(dist))

if __name__ == '__main__':
    distribution = dict(zip("ACGT", [0.10, 0.35, 0.15, 0.40]))
    print(", ".join(random_event(distribution) for _ in range(29)))
```

```
A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A,
A, A, A, A, A, A, A, A
```

Idea of solution

fill in

Discussion

fill in

With this general routine, the following should be easy to solve.

8. Write a class `ProteinToRandomRNA` to produce a random RNA sequence encoding the input protein sequence according to the input codon adaptation probabilities. The actual conversion is done through the `convert` method. Run your program with `LTPIQNRA`.

```
class ProteinToRandomRNA(object):
```

```
    def __init__(self):
        pass

    def convert(self, s):
        return ""

if __name__ == '__main__':
    protein_to_random_codons = ProteinToRandomRNA()
    print(protein_to_random_codons.convert("LTPIQNRA"))
```

Idea of solution

fill in

Discussion

fill in

Generating DNA sequences with higher-order Markov chains

We will now reuse the machinery derived above in a related context. We go back to DNA sequences, and consider some easy statistics that can be used to characterize the sequences. First, just the frequencies of bases A, C, G, T may reveal the species from which the input DNA originates; each species has a different base composition that has been formed during evolution. More interestingly, the areas where DNA to RNA transcription takes place (coding region) have an excess of C and G over A and T. To detect such areas a common routine is to just use a *sliding window* of fixed size, say k, and compute for each window position $\mathrm{T}\left[\mathrm{i..i}+\mathrm{k}-1\right]$ the base frequencies, where $\mathrm{T}\left[1..\mathrm{n}\right]$ is the input DNA sequence. When sliding the window from $\mathrm{T}\left[\mathrm{i..i}+\mathrm{k}-1\right]$ to $\mathrm{T}\left[\mathrm{i}+1..\mathrm{i}+\mathrm{k}\right]$ frequency $\mathrm{f}\left(\mathrm{T}\left[\mathrm{i}\right]\right)$ gets decreases by one and $\mathrm{f}\left(\mathrm{T}\left[\mathrm{i}+\mathrm{k}\right]\right)$ gets increased by one.

9. Write a *generator* `sliding_window` to compute sliding window base frequencies so that each moving of the window takes constant time. We saw in the beginning of the course one way how to create generators using generator expression. Here we use a different way. For the function `sliding_window` to be a generator, it must have at least one `yield` expression, see https://docs.python.org/3/reference/expressions.html#yieldexpr.

   Here is an example of a generator expression that works similarily to the built in `range` generator:

   ```
   def range(a, b=None, c=1):
       current = 0 if b == None else a
       end = a if b == None else b
       while current < end:
           yield current
   ```

```
            current += c
```

A yield expression can be used to return a value and *temporarily* return from the function.

```
def sliding_window(s, k):
    """
    This function returns a generator that can be iterated over all
    starting position of a k-window in the sequence.
    For each starting position the generator returns the nucleotide frequencies
    in the window as a dictionary.
    """
    for _ in s:
        yield {}


if __name__ == '__main__':
    s = "TCCCGACGGCCTTGCC"
    for d in sliding_window(s, 4):
        print(d)
```

```
{}
{}
{}
{}
{}
{}
{}
{}
{}
{}
{}
{}
{}
{}
{}
{}
```

Idea of solution

fill in

Discussion

fill in

Our models so far have been so-called *zero-order* models, as each event has been independent of other events. With sequences, the dependencies of events are naturally encoded by their *contexts*. Considering that a sequence is produced from left-to-right, a *first-order* context for $\mathrm{T}\left[\mathrm{i}\right]$ is $\mathrm{T}\left[\mathrm{i}-1\right]$, that is, the immediately preceding symbol. *First-order Markov chain* is a sequence produced by generating $\mathrm{c}=\mathrm{T}\left[\mathrm{i}\right]$ with the probability of event of seeing symbol c after previously generated symbol $\mathrm{a}=\mathrm{T}\left[\mathrm{i}-1\right]$. The first symbol of the chain is sampled according to the zero-order model.
The first-order model can naturally be extended to contexts of length k,

with $\mathrm{T}[\mathrm{i}]$ depending on $\mathrm{T}[\mathrm{i}-\mathrm{k}..\mathrm{i}-1]$. Then the first k symbols of the chain are sampled according to the zero-order model. The following assignments develop the routines to work with the *higher-order Markov chains*. In what follows, a k-mer is a substring $\mathrm{T}[\mathrm{i}..\mathrm{i}+\mathrm{k}-1]$ of the sequence at an arbitrary position.

10. Write function `context_list` that given an input DNA sequence T associates to each k-mer W the concatenation of all symbols c that appear after context W in T, that is, $\mathrm{T}[\mathrm{i}..\mathrm{i}+\mathrm{k}]=\mathrm{W}\,\mathrm{c}$. For example, GA is associated to TCT in T =ATGATATCATCGACGATGTAG, when $\mathrm{k}=2$.

```
def context_list(s, k):
    return {}

if __name__ == '__main__':
    k = 2
    s = "ATGATATCATCGACGATCTAG"
    d = context_list(s, k)
    print(d)
```

```
{}
```

Idea of solution

fill in

Discussion

fill in

11. With the above solution, write function `context_probabilities` to count the frequencies of symbols in each context and convert these frequencies into probabilities. Run `context_probabilities` with $\mathrm{T}=$ `ATGATATCATCGACGATGTAG` and k values 0 and 2.

```
def context_probabilities(s, k):
    return {}

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

Idea of solution

fill in

Discussion

fill in

12. With the above solution and the function `random_event` from the earlier exercise, write class `MarkovChain`. Its `generate` method should generate a random DNA sequence following the original k-th order Markov chain probabilities.

```
class MarkovChain:

    def __init__(self, zeroth, kth, k=2):
```

```
        self.k = k
        self.zeroth = zeroth
        self.kth = kth

    def generate(self, n, seed=None):
        return "$" * n

if __name__ == '__main__':
    zeroth = {'A': 0.2, 'C': 0.19, 'T': 0.31, 'G': 0.3}
    kth = {'GT': {'A': 1.0, 'C': 0.0, 'T': 0.0, 'G': 0.0},
           'CA': {'A': 0.0, 'C': 0.0, 'T': 1.0, 'G': 0.0},
           'TC': {'A': 0.5, 'C': 0.0, 'T': 0.0, 'G': 0.5},
           'GA': {'A': 0.0, 'C': 0.3333333333333333, 'T': 0.6666666666666666, 'G': 0.0
           'TG': {'A': 0.5, 'C': 0.0, 'T': 0.5, 'G': 0.0},
           'AT': {'A': 0.2, 'C': 0.4, 'T': 0.0, 'G': 0.4},
           'TA': {'A': 0.0, 'C': 0.0, 'T': 0.5, 'G': 0.5},
           'AC': {'A': 0.0, 'C': 0.0, 'T': 0.0, 'G': 1.0},
           'CG': {'A': 1.0, 'C': 0.0, 'T': 0.0, 'G': 0.0}}
    n = 10
    seed = 0
    mc = MarkovChain(zeroth, kth)
    print(mc.generate(n, seed))
```

```
$$$$$$$$$$
```

Idea of solution

fill in

Discussion

fill in

If you have survived so far without problems, please run your program a few more times with different inputs. At some point you should get a lookup error in your hash-table! The reason for this is not your code, but the way we defined the model: Some k-mers may not be among the training data (input sequence T), but such can be generated as the first k-mer that is generated using the zero-order model.

A general approach to fixing such issues with incomplete training data is to use *pseudo counts*. That is, all imaginable events are initialized to frequency count 1.

13. Write a new solution `context_pseudo_probabilities` based on the solution to problem 11. But this time use pseudo counts in order to obtain a k-th order Markov chain that can assign a probability for any DNA sequence. You may use the standard library function `itertools.product` to iterate over all k-mer of given length (`product("ACGT", repeat=k)`).

```
def context_pseudo_probabilities(s, k):
    return {"": ""}

if __name__ == '__main__':
    k = 2
    s = "ATGATATCATCGACGATGTAG"
    kth = context_pseudo_probabilities(s, k)
    zeroth = context_pseudo_probabilities(s, 0)[""]
    print(f"zeroth: {zeroth}")
    print("\n".join(f"{k}: {dict(v)}" for k, v in kth.items()))

    print("\n", MarkovChain(zeroth, kth, k).generate(20))
```

```
zeroth:
: {}

$$$$$$$$$$$$$$$$$$$
```

Idea of solution

fill in

Discussion

fill in

14. Write class `MarkovProb` that given the k-th order Markov chain developed above to the constructor, its method `probability` computes the probability of a given input DNA sequence.

```
class MarkovProb:
    def __init__(self, k, zeroth, kth):
        self.k = k
        self.zeroth = zeroth
        self.kth = kth

    def probability(self, s):
        return np.nan


if __name__ == '__main__':
    k = 2
    kth = context_pseudo_probabilities("ATGATATCATCGACGATGTAG", k)
    zeroth = context_pseudo_probabilities("ATGATATCATCGACGATGTAG", 0)[""]
    mc = MarkovProb(2, zeroth, kth)
    s="ATGATATCATCGACGATGTAG"
    print(f"Probability of sequence {s} is {mc.probability(s)}")
```

```
Probability of sequence ATGATATCATCGACGATGTAG is nan
```

Idea of solution

fill in

Discussion

fill in

With the last assignment you might end up in trouble with precision, as multiplying many small probabilities gives a really small number in the end. There is an easy fix by using so-called log-transform. Consider computation of $P = s_1 s_2 \cdots s_n$, where $0 \leq s_i \leq 1$ for each i. Taking logarithm in base 2 from both sides gives
$\log_2 P = \log_2 (s_1 s_2 \cdots s_n) = \log_2 s_1 + \log_2 s_2 + \cdots \log s_n = \sum_{i=1}^{n} \log s_i$
, with repeated application of the property that the logarithm of a multiplication of two numbers is the sum of logarithms of the two numbers taken separately. The results is abbreviated as log-probability.

15. Write class `MarkovLog` that given the k-th order Markov chain developed above to the constructor, its method `log_probability` computes the log-probability

of a given input DNA sequence. Run your program with $\mathrm{T} =$ ATGATATCATCGACGATGTAG and $\mathrm{k} = 2$.

```
class MarkovLog(object):

    def __init__(self, k, zeroth, kth):
        pass

    def log_probability(self, s):
        return np.nan

if __name__ == '__main__':
    k = 2
    kth = context_pseudo_probabilities("ATGATATCATCGACGATGTAG", k)
    zeroth = context_pseudo_probabilities("ATGATATCATCGACGATGTAG", 0)[""]
    mc = MarkovLog(2, zeroth, kth)
    s="ATGATATCATCGACGATGTAG"
    print(f"Log probability of sequence {s} is {mc.log_probability(s)}")
```

```
Log probability of sequence ATGATATCATCGACGATGTAG is nan
```

Idea of solution

fill in

Discussion

fill in

Finally, if you try to use the code so far for very large inputs, you might observe that the concatenation of symbols following a context occupy considerable amount of space. This is unnecessary, as we only need the frequencies.

16. Optimize the space requirement of your code from exercise 13 for the k-th order Markov chain by replacing the concatenations by direct computations of the frequencies. Implement this as the `better_context_probabilities` function.

```
def better_context_probabilities(s, k):
    return {"": ""}

if __name__ == '__main__':
    k = 2
    s = "ATGATATCATCGACGATGTAG"
    d = better_context_probabilities(s, k)
    print("\n".join(f"{k}: {v}" for k, v in d.items()))
```

:

Idea of solution

fill in

Discussion

fill in

While the earlier approach of explicit concatenation of symbols following a context suffered from inefficient use of space, it does have a benefit of giving another much simpler strategy to sample from the distribution:

observe that an element of the concatenation taken uniformly randomly is sampled exactly with the correct probability.

17. Revisit the solution 12 and modify it to directly sample from the concatenation of symbols following a context. The function `np.random.choice` may be convenient here. Implement the modified version as the new `SimpleMarkovChain` class.

```
class SimpleMarkovChain(object):
    def __init__(self, s, k):
        pass

    def generate(self, n, seed=None):
        return "Q"*n

if __name__ == '__main__':
    k = 2
    s = "ATGATATCATCGACGATGTAG"
    n = 10
    seed = 7
    mc = SimpleMarkovChain(s, k)
    print(mc.generate(n, seed))
```

QQQQQQQQQQ

Idea of solution

fill in

Discussion

fill in

k-mer index

Our k-th order Markov chain can now be modified to a handy index structure called k-mer index. This index structure associates to each k-mer its list of occurrence positions in DNA sequence T. Given a query k-mer W, one can thus easily list all positions i with $T[i..k-1] = W$.

18. Implement function `kmer_index` inspired by your earlier code for the k-th order Markov chain. Test your program with `ATGATATCATCGACGATGTAG` and $k = 2$.

```
def kmer_index(s, k):
    return {}

if __name__ == '__main__':
    k=2
    s = "ATGATATCATCGACGATGTAG"
    print("Using string:")
    print(s)
    print("".join([str(i%10) for i in range(len(s))]))
    print(f"\n{k}-mer index is:")
    d=kmer_index(s, k)
    print(dict(d))
```

```
Using string:
ATGATATCATCGACGATGTAG
01234567890123456789

2-mer index is:
{}
```

```
2-mer index is:
{}
```

Idea of solution

fill in

Discussion

fill in

Comparison of probability distributions

Now that we know how to learn probability distributions from data, we might want to compare two such distributions, for example, to test if our programs work as intended.

Let $\mathrm{P} = \{\mathrm{p1}, \mathrm{p2}, \ldots, \mathrm{pn}\}$ and $\mathrm{Q} = \{\mathrm{q1}, \mathrm{q2}, \ldots, \mathrm{qn}\}$ be two probability distributions for the same set of n events. This means $\sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{pi} = \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{qi} = 1$, $0 \leq \mathrm{pj} \leq 1$, and $0 \leq \mathrm{qj} \leq 1$ for each event j.

Kullback-Leibler divergence is a measure $\mathrm{d}()$ for the *relative entropy* of P with respect to Q defined as $\mathrm{d}(\mathrm{P}||\mathrm{Q}) = \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{pi} \log \frac{\mathrm{pi}}{\mathrm{qi}}$.

This measure is always non-negative, and 0 only when $\mathrm{P} = \mathrm{Q}$. It can be interpreted as the gain of knowing Q to encode P. Note that this measure is not symmetric.

19. Write function `kullback_leibler` to compute $\mathrm{d}(\mathrm{P}||\mathrm{Q})$. Test your solution by generating a random RNA sequence encoding the input protein sequence according to the input codon adaptation probabilities. Then you should learn the codon adaptation probabilities from the RNA sequence you generated. Then try the same with uniformly random RNA sequences (which don't have to encode any specific protein sequence). Compute the relative entropies between the three distribution (original, predicted, uniform) and you should observe a clear difference. Because $\mathrm{d}(\mathrm{P}||\mathrm{Q})$ is not symmetric, you can either print both $\mathrm{d}(\mathrm{P}||\mathrm{Q})$ and $\mathrm{d}(\mathrm{Q}||\mathrm{P})$, or their average.

    This problem may be fairly tricky. Only the `kullback_leibler` function is automatically tested. The codon probabilities is probably a useful helper function. The main guarded section can be completed by filling out the `pass` sections using tooling from previous parts and fixing the *placeholder* lines.

```
def codon_probabilities(rna):
    """
```

```
    Given an RNA sequence, simply calculates the proability of
    all 3-mers empirically based on the sequence
    """
    return {"".join(codon): 0 for codon in product("ACGU", repeat=3)}

def kullback_leibler(p, q):
    """
    Computes Kullback-Leibler divergence between two distributions.
    Both p and q must be dictionaries from events to probabilities.
    The divergence is defined only when q[event] == 0 implies p[event] == 0.
    """
    return np.nan

if __name__ == '__main__':
    aas = list("*ACDEFGHIKLMNPQRSTVWY") # List of amino acids
    n = 10000

    # generate a random protein and some associated rna
    protein = "".join(choice(aas, n))
    pass

    # Maybe check that converting back to protein results in the same sequence
    pass

    # Calculate codon probabilities of the rna sequence
    cp_predicted = codon_probabilities("<rna sequence>") # placeholder call

    # Calculate codon probabilities based on the codon usage table
    cp_orig = {"".join(codon): 0 for codon in product("ACGU", repeat=3)} # placeholder

    # Create a completely random RNA sequence and get the codon probabilities
    pass
    cp_uniform = codon_probabilities("<random rna sequence>") # placeholder call

    print("d(original || predicted) =", kullback_leibler(cp_orig, cp_predicted))
    print("d(predicted || original) =", kullback_leibler(cp_predicted, cp_orig))
    print()
    print("d(original || uniform) =", kullback_leibler(cp_orig, cp_uniform))
    print("d(uniform || original) =", kullback_leibler(cp_uniform, cp_orig))
    print()
    print("d(predicted || uniform) =", kullback_leibler(cp_predicted, cp_uniform))
    print("d(uniform || predicted) =", kullback_leibler(cp_uniform, cp_predicted))
```

```
d(original || predicted) = nan
d(predicted || original) = nan

d(original || uniform) = nan
d(uniform || original) = nan

d(predicted || uniform) = nan
d(uniform || predicted) = nan
```

Idea of solution

fill in

Discussion

fill in

Stationary and equilibrium distributions (extra)

Let us consider a Markov chain of order one on the set of nucleotides. Its

transition probabilities can be expressed as a 4×4 matrix $\mathrm{P} = (\mathrm{p_{ij}})$, where the element $\mathrm{p_{ij}}$ gives the probability of the jth nucleotide on the condition the previous nucleotide was the ith. An example of a transition matrix is

\begin{array}{l|rrrr} & A & C & G & T \\ \hline A & 0.30 & 0.0 & 0.70 & 0.0 \\ C & 0.00 & 0.4 & 0.00 & 0.6 \\ G & 0.35 & 0.0 & 0.65 & 0.0 \\ T & 0.00 & 0.2 & 0.00 & 0.8 \\ \end{array}.

A distribution $\pi = (\pi_1, \pi_2, \pi_3, \pi_4)$ is called *stationary*, if $\pi = \pi\mathrm{P}$ (the product here is matrix product).

20. Write function `get_stationary_distributions` that gets a transition matrix as parameter, and returns the list of stationary distributions. You can do this with NumPy by first taking transposition of both sides of the above equation to get equation $\pi^{\mathrm{T}} = \mathrm{P}^{\mathrm{T}}\pi^{\mathrm{T}}$. Using numpy.linalg.eig take all eigenvectors related to eigenvalue 1.0. By normalizing these vectors to sum up to one get the stationary distributions of the original transition matrix. In the `main` function print the stationary distributions of the above transition matrix.

```
def get_stationary_distributions(transition):
    """
    The function get a transition matrix of a degree one Markov chain as parameter.
    It returns a list of stationary distributions, in vector form, for that chain.
    """
    return np.random.rand(2, 4) - 0.5


if __name__ == "__main__":
    transition=np.array([[0.3, 0, 0.7, 0],
                         [0, 0.4, 0, 0.6],
                         [0.35, 0, 0.65, 0],
                         [0, 0.2, 0, 0.8]])
    print("\n".join(
        ", ".join(
            f"{pv:+.3f}"
            for pv in p)
        for p in get_stationary_distributions(transition)))
```

```
+0.162, -0.227, +0.388, +0.151
-0.370, +0.057, +0.198, +0.037
```

Idea of solution

Discussion

21. Implement the `kl_divergence` function below so that the main guarded code runs properly. Using your modified Markov chain generator generate a nucleotide sequence s of length $10\ 000$. Choose prefixes of s of lengths $1, 10, 100, 1000$, and $10\ 000$. For each of these prefixes find out their nucleotide distribution (of order 0) using your earlier tool. Use 1 as the pseudo count. Then, for each prefix, compute the KL divergence between the initial distribution and the normalized nucleotide distribution.

```
def kl_divergences(initial, transition):
    """
    Calculates the the Kullback-Leibler divergences between empirical distributions
    generated using a markov model seeded with an initial distributin and a transition
    matrix, and the initial distribution.
```

```
    Sequences of length [1, 10, 100, 1000, 10000] are generated.
    """
    return zip([1, 10, 100, 1000, 10000], np.random.rand(5))

if __name__ == "__main__":
    transition=np.array([[0.3, 0, 0.7, 0],
                         [0, 0.4, 0, 0.6],
                         [0.35, 0, 0.65, 0],
                         [0, 0.2, 0, 0.8]])
    print("Transition probabilities are:")
    print(transition)
    stationary_distributions = get_stationary_distributions(transition)
    print("Stationary distributions:")
    print(np.stack(stationary_distributions))
    initial = stationary_distributions[1]
    print("Using [{}] as initial distribution\n".format(", ".join(f"{v:.2f}" for v in
    results = kl_divergences(initial, transition)
    for prefix_length, divergence in results: # iterate on prefix lengths in order (1,
        print("KL divergence of stationary distribution prefix " \
              "of length {:5d} is {:.8f}".format(prefix_length, divergence))
```

```
Transition probabilities are:
[[ 0.3 0. 0.7 0. ]
[ 0. 0.4 0. 0.6 ]
[ 0.35 0. 0.65 0. ]
[ 0. 0.2 0. 0.8 ]]
Stationary distributions:
[[ 0.02421173 -0.29026212 0.1389851 0.44623574]
[ 0.24261234 0.09071543 0.35909807 -0.47662849]]
Using [0.24, 0.09, 0.36, -0.48] as initial distribution

KL divergence of stationary distribution prefix of length 1 is
0.60666836
KL divergence of stationary distribution prefix of length 10 is
0.71775085
KL divergence of stationary distribution prefix of length 100 is
0.38115273
KL divergence of stationary distribution prefix of length 1000
is 0.36116144
KL divergence of stationary distribution prefix of length 10000
is 0.22844669
```

Idea of solution

fill in

Discussion

fill in

22. Implement the following in the `main` function. Find the stationary distribution for the following transition matrix:

\begin{array}{ l | r r r r} & A & C & G & T \ \hline A & 0.30 & 0.10 & 0.50 & 0.10 \ C & 0.20 & 0.30 & 0.15 & 0.35 \ G & 0.25 & 0.15 & 0.20 & 0.40 \ T & 0.35 & 0.20 & 0.40 & 0.05 \ \end{array}

Since there is only one stationary distribution, it is called the *equilibrium distribution*. Choose randomly two nucleotide distributions. You can take these from your sleeve or sample them from the Dirichlet distribution.

Then for each of these distributions as the initial distribution of the Markov chain, repeat the above experiment.

The `main` function should return tuples, where the first element is the (random) initial distribution and the second element contains the results as a list of tuples where the first element is the kl divergence and the second element the empirical nucleotide distribution, for the different prefix lengths.

The state distribution should converge to the equilibrium distribution no matter how we start the Markov chain! That is the last line of the tables should have KL-divergence very close to 0 and an empirical distribution very close to the equilibrium distribution.

```
def main(transition, equilibrium_distribution):
    vals = list(zip(np.random.rand(10), np.random.rand(10, 4) - 0.5))
    return zip(np.random.rand(2, 4) - 0.5,
               [vals[:5], vals[5:]])


if __name__ == "__main__":
    transition = np.array([[0.3, 0.1, 0.5, 0.1],
                           [0.2, 0.3, 0.15, 0.35],
                           [0.25, 0.15, 0.2, 0.4],
                           [0.35, 0.2, 0.4, 0.05]])
    print("Transition probabilities are:", transition, sep="\n")
    stationary_distributions = get_stationary_distributions(transition)
    # Uncomment the below line to check that there actually is only one stationary dis
    # assert len(stationary_distributions) == 1
    equilibrium_distribution = stationary_distributions[0]
    print("Equilibrium distribution:")
    print(equilibrium_distribution)
    for initial_distribution, results in main(transition, equilibrium_distribution):
        print("\nUsing {} as initial distribution:".format(initial_distribution))
        print("kl-divergence   empirical distribution")
        print("\n".join("{:.11f}   {}".format(di, kl) for di, kl in results))
```

```
Transition probabilities are:
[[ 0.3 0.1 0.5 0.1 ]
[ 0.2 0.3 0.15 0.35]
[ 0.25 0.15 0.2 0.4 ]
[ 0.35 0.2 0.4 0.05]]
Equilibrium distribution:
[ 0.05549613 -0.0618828 -0.3045492 0.11913485]

Using [-0.3337754 0.45068768 -0.10300145 -0.20088342] as initial
distribution:
kl-divergence empirical distribution
0.25023050057 [ 0.2243623 0.15184879 -0.36068593 0.0442362 ]
0.85018673369 [-0.27420015 -0.1344076 -0.08342149 0.10517491]
0.69016371310 [-0.11457606 0.27115112 0.16359241 -0.1280264 ]
0.14692236781 [ 0.07706555 0.23608696 0.01647088 -0.08926859]
0.98884289212 [-0.06341222 0.45084425 0.3957673 -0.40553139]

Using [-0.14182502 0.21310283 0.21065106 0.45942999] as initial
distribution:
kl-divergence empirical distribution
0.09700875402 [ 0.35101425 -0.18876535 0.25445247 -0.20776396]
0.51559517427 [ 0.31812104 -0.03010157 -0.2113726 -0.49508434]
```

```
0.92023091580 [ 0.40268876 -0.22995087 -0.42421415 -0.1464154 ]
0.32096833450 [ 0.15860567 -0.16145627 0.04025597 -0.3869026 ]
0.33771485079 [-0.30140972 0.27013169 -0.49653246 -0.12583137]
```

Idea of solution

fill in

Discussion

fill in

You have reached the end of this section!
Remember to check your points from the ball on the bottom-right corner of the material!

In this part: