Sequence Evolution and Statistics

How do we decide whether a sequence similarity is *real*, how much change has *actually* happened between two sequences, and whether a gene is under *selection*? This lecture uses three simple tools you can write yourself in Python:

1. **Shuffling** - build your own "null" distribution to get a p-value
2. **Simulation** - evolve sequences on the computer where you *know* the true answer, and see how well our measurements recover it
3. **Ka/Ks (dN/dS)** - compare nonsynonymous and synonymous changes in coding sequences to detect selection

and finishes with what to do when you run the same test on thousands of genes.

The examples need Biopython and SciPy. On the HPCC cluster or your laptop:

```
# e.g. in a conda environment or with pip
pip install biopython scipy
```

Is this similarity better than chance? Shuffling tests

A BLAST or alignment score by itself doesn't tell you whether two sequences are related - two random sequences will still align a little. What we want to know is: *how often would I see a score this good if the sequences were unrelated?* That is the definition of a **p-value**.

BLAST answers this with statistical theory (the **E-value** is the number of hits this good you expect by chance in a search of that database size). We can answer it directly by **shuffling**: keep the amino acid composition of one sequence the same but scramble its order. Any real homology is destroyed, but things like length and composition - which also affect scores - are kept. Align many shuffled copies and see where the real score falls.

This compares yeast ubiquitin to yeast SUMO (Smt3). The two proteins have the same 3D fold and are thought to share an ancestor, but their sequences have diverged a lot.

```
#!/usr/bin/env python3
# Is the alignment score between two proteins better than chance?
import random
from Bio import Align
from Bio.Align import substitution_matrices

# yeast ubiquitin (UBI4 repeat) and yeast SUMO (SMT3)
ubiquitin = "MQIFVKTLTGKTITLEVESSDTIDNVKSKIQDKEGIPPDQQRLIFAGKQLEDGRTLSDYNIQKESTLHLVLRLRGG"
sumo = ("MSDSEVNQEAKPEVKPEVKPETHINLKVSDGSSEIFFKIKKTTPLRRLMEAFAKRQGKEMDSLRFLYDGIRIQ"
        "ADQTPEDLDMEDNDIIEAHREQIGG")

aligner = Align.PairwiseAligner()
aligner.mode = "local"
aligner.substitution_matrix = substitution_matrices.load("BLOSUM62")
aligner.open_gap_score = -11
aligner.extend_gap_score = -1

observed = aligner.score(ubiquitin, sumo)
print("observed score:", observed)

random.seed(42)          # so we all get the same answer
n_shuffles = 1000
scores = []
for i in range(n_shuffles):
    letters = list(sumo)
    random.shuffle(letters)   # same composition, order destroyed
```

```
    scores.append(aligner.score(ubiquitin, "".join(letters)))

n_as_good = sum(1 for s in scores if s >= observed)
pvalue = (n_as_good + 1) / (n_shuffles + 1)
print("mean shuffled score: %.1f  max shuffled score: %.1f" % (sum(scores) / n_shuffles, max(scores)))
print("shuffles scoring >= observed:", n_as_good)
print("empirical p-value: %.4f" % pvalue)
```

```
observed score: 37.0
mean shuffled score: 26.9  max shuffled score: 46.0
shuffles scoring >= observed: 52
empirical p-value: 0.0529
```

About 5% of the shuffled sequences score as well as the real one, so p is about 0.05. That's borderline, even though we know from their structures that these proteins are related. This is the **twilight zone** of sequence similarity (below ~25% identity for proteins): homologs can be too diverged to detect reliably from sequence alone. A failure to find a significant hit does **not** prove two genes are unrelated.

Things to notice:

- `random.seed()` makes the "random" results repeatable - important for reproducible analyses.
- The p-value is `(k + 1) / (n + 1)` rather than `k / n`, so it can never be exactly 0. With 1000 shuffles the smallest possible p-value is ~0.001; to claim smaller p-values you need more shuffles.
- The same idea - a **permutation test** - works for many questions: shuffle sample labels to test for a difference between groups, shuffle gene positions to ask whether genes cluster on a chromosome, etc.

Try it: compare `ubiquitin` to a copy of itself with a few changes, and to a completely unrelated protein. How do the p-values change? What happens to the p-value if you use 100 shuffles instead of 1000?

How much change really happened? Simulating evolution

When we count the differences between two DNA sequences (the **p-distance**, the fraction of sites that differ) we *underestimate* how many mutations happened. A site can mutate twice (A to G to T looks like one change) or mutate back (A to G to A looks like none). The longer two sequences have been diverging, the worse this gets.

We can see this by simulation: start from a random sequence, make a known number of substitutions, and compare what we *observe* to what we *did*.

```
#!/usr/bin/env python3
# Simulate DNA evolution and compare observed differences to real substitutions
import math
import random

random.seed(1)
BASES = "ACGT"
length = 1000

def mutate(seq, n_subs):
    """Make n_subs random substitutions; a site can be hit more than once."""
    seq = list(seq)
    for i in range(n_subs):
        pos = random.randrange(len(seq))
        choices = [b for b in BASES if b != seq[pos]]
        seq[pos] = random.choice(choices)
    return "".join(seq)

def p_distance(s1, s2):
    diffs = sum(1 for a, b in zip(s1, s2) if a != b)
    return diffs / len(s1)
```

```python
def jukes_cantor(p):
    if p >= 0.75:
        return float("inf")   # saturated: no information left
    return -0.75 * math.log(1 - (4.0 / 3.0) * p)

ancestor = "".join(random.choice(BASES) for i in range(length))

print("true_subs_per_site\tp_distance\tJC_distance")
for n_subs in [50, 100, 250, 500, 1000, 1500, 2000, 3000]:
    descendant = mutate(ancestor, n_subs)
    p = p_distance(ancestor, descendant)
    print("%.2f\t%.3f\t%.3f" % (n_subs / length, p, jukes_cantor(p)))
```

```
true_subs_per_site  p_distance  JC_distance
0.05    0.048   0.050
0.10    0.094   0.100
0.25    0.221   0.262
0.50    0.343   0.458
1.00    0.560   1.030
1.50    0.629   1.368
2.00    0.694   1.946
3.00    0.733   2.840
```

- For small amounts of change, p-distance is about the same as the true number of substitutions.
- As change accumulates, p-distance levels off near **0.75**: two unrelated random DNA sequences still match at 1 in 4 sites by chance. This is **saturation**.
- The **Jukes-Cantor (JC69)** correction, `d = -3/4 ln(1 - 4/3 p)`, estimates the true number of substitutions per site from p. It works well until the sequences are close to saturation, where small differences in p give big changes in d (and a lot of uncertainty).

The JC model assumes all bases are equally common and all changes are equally likely. Real DNA has more transitions (A-G and C-T changes) than transversions, and uneven base composition, so phylogenetics programs (e.g. IQ-TREE's model finder, used in the Phylogeny lecture) use more realistic models - but they are fixing exactly this problem.

Why simulate? Whenever you use a method, you can test it on data where you know the answer. If a method can't recover the truth from simulated data, you shouldn't trust it on real data.

Is a gene under selection? Ka/Ks (dN/dS)

In a protein coding gene, a DNA change is either:

- **synonymous** (silent) - the codon changes but the amino acid does not, e.g. `CTT` to `CTC` are both leucine
- **nonsynonymous** (replacement) - the amino acid changes, e.g. `CTT` (Leu) to `CCT` (Pro)

Synonymous changes are close to neutral, so their rate is a baseline for the mutation rate. Comparing the two rates tells us what selection has done to the protein:

dN/dS (Ka/Ks)	Interpretation
much less than 1	**purifying selection** - most amino acid changes were harmful and removed. This is most genes.
about 1	**neutral** - amino acid changes are neither removed nor favored (e.g. a pseudogene)
greater than 1	**positive (diversifying) selection** - amino acid changes were favored (e.g. immune genes, pathogen effectors, arms races)

dN (or Ka) is the number of nonsynonymous substitutions per nonsynonymous *site*, and **dS** (Ks) is synonymous substitutions per synonymous site. We divide by sites because most random mutations would be nonsynonymous (roughly 3/4 of sites), so raw counts would be misleading.

Let's simulate a gene evolving under no selection and under strong purifying selection, then estimate dN/dS with Biopython. The simulation proposes random mutations; synonymous ones are always kept, but nonsynonymous ones are only kept 10% of the time under purifying selection.

```
#!/usr/bin/env python3
# Simulate a gene under different kinds of selection and estimate dN/dS (Ka/Ks)
import random
import warnings
from Bio import BiopythonExperimentalWarning
from Bio.Seq import Seq

with warnings.catch_warnings():
    warnings.simplefilter("ignore", BiopythonExperimentalWarning)
    from Bio.codonalign.codonseq import CodonSeq, cal_dn_ds

random.seed(7)
BASES = "ACGT"
STOPS = {"TAA", "TAG", "TGA"}

def random_gene(n_codons):
    codons = ["ATG"]
    while len(codons) < n_codons:
        codon = "".join(random.choice(BASES) for i in range(3))
        if codon not in STOPS:
            codons.append(codon)
    return "".join(codons)

def evolve(gene, n_proposed, keep_nonsyn):
    """Propose n_proposed random point mutations. Synonymous ones are always kept;
    nonsynonymous ones are kept with probability keep_nonsyn
    (1.0 = neutral, 0.1 = strong purifying selection)."""
    gene = list(gene)
    proposed = 0
    while proposed < n_proposed:
        pos = random.randrange(3, len(gene))        # leave the start codon alone
        start = pos - pos % 3
        old_codon = "".join(gene[start:start + 3])
        new_base = random.choice([b for b in BASES if b != gene[pos]])
        new_codon = list(old_codon)
        new_codon[pos % 3] = new_base
        new_codon = "".join(new_codon)
        if new_codon in STOPS:
            continue                                 # nonsense mutations are lost
        synonymous = Seq(old_codon).translate() == Seq(new_codon).translate()
        if synonymous or random.random() < keep_nonsyn:
            gene[pos] = new_base
        proposed += 1
    return "".join(gene)

ancestor = random_gene(300)
for label, keep in [("neutral", 1.0), ("purifying", 0.1)]:
    descendant = evolve(ancestor, 60, keep)
```

```
    dn, ds = cal_dn_ds(CodonSeq(ancestor), CodonSeq(descendant), method="NG86")
    print("%-10s dN=%.3f dS=%.3f dN/dS=%.2f" % (label, dn, ds, dn / ds))
```

```
neutral    dN=0.066 dS=0.065 dN/dS=1.02
purifying  dN=0.007 dS=0.065 dN/dS=0.11
```

The estimates recover what we built into the simulation: dN/dS of about 1 for the neutral gene and about 0.1 when 90% of amino acid changes are removed. (`NG86` is the Nei-Gojobori 1986 counting method; Biopython also has `LWL85`, `YN00`, and `ML`.)

Using real genes

For real data you need:

1. **Orthologous** coding sequences (see the Orthology lecture) - comparing paralogs or unrelated genes is meaningless.
2. A **codon alignment**: align the *proteins*, then thread the DNA back onto the protein alignment so that codons stay in frame (tools: PAL2NAL, `Bio.codonalign`, or MACSE). Aligning the DNA directly will put gaps inside codons.
3. A program to estimate dN and dS. Besides Biopython, the standard tool is **PAML** (`yn00` for pairs, `codeml` for tests on a tree); check `module avail paml` on the cluster.

Things to watch out for:

- **Saturation of dS**: synonymous sites change fast. If dS is above ~2, the estimate is unreliable (the same problem as the p-distance plateau above), so compare species that are not too distant.
- dN/dS for a *whole gene* is an average. A gene with a few positively selected sites in a mostly conserved protein will still have dN/dS < 1 overall - site models in `codeml` are designed to find these.
- Very similar sequences (dS near 0) give unstable ratios - dividing by a tiny number.

Testing many genes: multiple testing

If you test 5,000 genes for selection at $p < 0.05$, you expect ~250 "significant" results **even if nothing is going on**. In genomics we almost always run many tests, so we control the **false discovery rate (FDR)** - the expected fraction of our significant results that are false positives - with the Benjamini-Hochberg method. The adjusted p-values are often called **q-values**. RNASeq tools like DESeq2 report these as `padj`.

```
from scipy.stats import false_discovery_control

pvalues = [0.001, 0.008, 0.039, 0.041, 0.042, 0.06, 0.074, 0.205, 0.212, 0.216]
qvalues = false_discovery_control(pvalues)   # Benjamini-Hochberg by default
for p, q in zip(pvalues, qvalues):
    print("p=%.3f  q=%.3f  %s" % (p, q, "significant" if q < 0.05 else ""))
```

```
p=0.001  q=0.010  significant
p=0.008  q=0.040  significant
p=0.039  q=0.084
p=0.041  q=0.084
p=0.042  q=0.084
p=0.060  q=0.100
p=0.074  q=0.106
p=0.205  q=0.216
p=0.212  q=0.216
p=0.216  q=0.216
```

Five of these tests have $p < 0.05$, but only two remain significant after FDR correction.

Exercises

1. Modify `shuffle_test.py` to read two protein sequences from a FASTA file (use `Bio.SeqIO`) and report the empirical p-value. Run it on a pair of orthologs from the Orthology lecture data.
2. Change `simulate_jc.py` so that transitions happen twice as often as transversions. Does the Jukes-Cantor correction still recover the true distance? When does it fail?
3. In `kaks_sim.py`, add a "positive selection" case where nonsynonymous changes are kept *more* often than synonymous ones (hint: make synonymous changes the ones that are sometimes rejected). What dN/dS do you get?
4. Run `kaks_sim.py` 100 times with different seeds for the neutral case and make a histogram of dN/dS. How variable is the estimate for a 300 codon gene? For 1000 codons?

Further reading

- Jukes TH, Cantor CR (1969) and the chapter on substitution models in *Molecular Evolution and Phylogenetics* (Nei & Kumar)
- Yang Z. PAML
- Kryazhimskiy S, Plotkin JB (2008) The population genetics of dN/dS. *PLoS Genetics* doi:10.1371/journal.pgen.1000304
- Pearson WR (2013) An introduction to sequence similarity ("homology") searching. *Curr Protoc Bioinformatics* doi:10.1002/0471250953.bi0301s42 - explains E-values, shuffling, and why statistics matter