testing proxy vs push method

In [127…
```
import tsinfer
import tskit
import msprime
import tsdate
import numpy as np
import pandas as pd
import datetime as dt
import time
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.linear_model import LinearRegression
from itertools import combinations
```

In [128…
```
import logging
logger = logging.getLogger("tsinfer")

def insert_proxy_samples(
        self,
        variant_data,
        *,
        sample_ids=None,
        epsilon=None,
        keep_ancestor_times=None,
        allow_mutation=None,  # deprecated alias
        **kwargs,
):
        """
        Take a set of samples from a :class:`.VariantData` instance and create additional
        "proxy sample ancestors" from them, returning a new :class:`.AncestorData`
        instance including both the current ancestors and the additional ancestors
        at the appropriate time points.

        A *proxy sample ancestor* is an ancestor based upon a known sample. At
        sites used in the full inference process, the haplotype of this ancestor
        is identical to that of the sample on which it is based. The time of the
        ancestor is taken to be a fraction ``epsilon`` older than the sample on
        which it is based.

        A common use of this function is to provide ancestral nodes for anchoring
        historical samples at the correct time when matching them into a tree
        sequence during the :func:`tsinfer.match_samples` stage of inference.
        For this reason, by default, the samples chosen from ``sample_data``
        are those associated with historical (i.e. non-contemporary)
        :ref:`individuals <sec_inference_data_model_individual>`. This can be
        altered by using the ``sample_ids`` parameter.

        .. note::

            The proxy sample ancestors inserted here will correspond to extra nodes
            in the inferred tree sequence. At sites which are not used in the full
            inference process (e.g. sites unique to a single historical sample),
            these proxy sample ancestor nodes may have a different genotype from
            their corresponding sample.

        :param VariantData variant_data: The `VariantData` instance
            from which to select the samples used to create extra ancestors.
        :param list(int) sample_ids: A list of sample ids in the ``variant_data``
            instance that will be selected to create the extra ancestors. If
            ``None`` (default) select all the historical samples, i.e. those
            associated with an :ref:`sec_inference_data_model_individual` whose
            time is greater than zero. The order of ids is ignored, as are
            duplicate ids.
        :param list(float) epsilon: An list of small time increments
            determining how much older each proxy sample ancestor is than the
            corresponding sample listed in ``sample_ids``. A single value is also
            allowed, in which case it is used as the time increment for all selected
            proxy sample ancestors. If None (default) find :math:`{\\delta}t`, the
            smallest time difference between the sample times and the next
            oldest ancestor in the current :class:`.AncestorData` instance, setting
            ``epsilon`` = :math:`{\\delta}t / 100` (or, if all selected samples
            are at least as old as the oldest ancestor, take :math:`{\\delta}t`
            to be the smallest non-zero time difference between existing ancestors).
        :param bool keep_ancestor_times: If ``False`` (the default), the existing
            times of the ancestors in the current :class:`.AncestorData` instance
            may be increased so that derived states in the inserted proxy samples.
            can have an ancestor with a mutation to that site above them (i.e. the
            infinite sites assumption is maintained). This is useful when sites
```

```
        times have been approximated by using their frequency. Alternatively,
        if ``keep_ancestor_times`` is ``True``, existing ancestor times are
        preserved, and inserted proxy sample ancestors are allowed to
        possess derived alleles at sites where there are no pre-existing
        mutations in older ancestors. This can lead to a de-novo mutation at a
        site that also has a mutation elsewhere (i.e. breaking the infinite sites
        assumption).
    :param bool allow_mutation: Deprecated alias for `keep_ancestor_times`.
    :param \\**kwargs: Further arguments passed to the constructor when creating
        the new :class:`AncestorData` instance which will be returned.

    :return: A new :class:`.AncestorData` object.
    :rtype: AncestorData
    """
    if allow_mutation is not None:
        if keep_ancestor_times is not None:
            raise ValueError(
                "Cannot specify both `allow_mutation` and `keep_ancestor_times`"
            )
        keep_ancestor_times = allow_mutation
    self._check_finalised()
    variant_data._check_finalised()
    if self.sequence_length != variant_data.sequence_length:
        raise ValueError("variant_data does not have the correct sequence length")
    used_sites = np.isin(variant_data.sites_position[:], self.sites_position[:])
    if np.sum(used_sites) != self.num_sites:
        raise ValueError("Genome positions in ancestors missing from variant_data")

    if sample_ids is None:
        sample_ids = []
        for i in variant_data.individuals():
            if i.time > 0:
                sample_ids += i.samples
    # sort by ID and make unique for quick haplotype access
    sample_ids, unique_indices = np.unique(np.array(sample_ids), return_index=True)

    sample_times = np.zeros(len(sample_ids), dtype=self.ancestors_time.dtype)
    for i, s in enumerate(sample_ids):
        sample = variant_data.sample(s)
        if sample.individual != tskit.NULL:
            sample_times[i] = variant_data.individual(sample.individual).time

    if epsilon is not None:
        epsilons = np.atleast_1d(epsilon)
        if len(epsilons) == 1:
            # all get the same epsilon
            epsilons = np.repeat(epsilons, len(sample_ids))
        else:
            if len(epsilons) != len(unique_indices):
                raise ValueError(
                    "The number of epsilon values must equal the number of "
                    f"sample_ids ({len(sample_ids)})"
                )
            epsilons = epsilons[unique_indices]

    else:
        anc_times = self.ancestors_time[:][::-1]  # find ascending time order
        older_index = np.searchsorted(anc_times, sample_times, side="right")
        # Don't include times older than the oldest ancestor
        allowed = older_index < self.num_ancestors
        if np.sum(allowed) > 0:
            delta_t = anc_times[older_index[allowed]] - sample_times[allowed]
        else:
            # All samples have times equal to or older than the oldest curr ancestor
            time_diffs = np.diff(anc_times)
            delta_t = np.min(time_diffs[time_diffs > 0])
        epsilons = np.repeat(np.min(delta_t) / 100.0, len(sample_ids))

    proxy_times = sample_times + epsilons
    time_sorted_indexes = np.argsort(proxy_times)
    reverse_time_sorted_indexes = time_sorted_indexes[::-1]
    # In cases where we have more than a handful of samples to use as proxies, it is
    # inefficient to access the haplotypes out of order, so we iterate and cache
    # (caution: the haplotypes list may be quite large in this case)
    haplotypes = [
        h[1] for h in variant_data.haplotypes(
            samples=sample_ids, sites=used_sites, recode_ancestral=True
        )
    ]

    new_anc_times = self.ancestors_time[:]  # this is a copy
    if not keep_ancestor_times:
        assert np.all(np.diff(self.ancestors_time) <= 0)
```

```
            # Find the youngest (max) ancestor ID constrained by each sample haplotype
            site_ancestor = -np.ones(self.num_sites, dtype=int)
            anc_min_time = np.zeros(self.num_ancestors, dtype=self.ancestors_time.dtype)
            # If (unusually) there are multiple ancestors for the same focal site, we
            # can take the youngest
            for ancestor_id, focal_sites in enumerate(self.ancestors_focal_sites):
                site_ancestor[focal_sites] = ancestor_id
            for hap_id in time_sorted_indexes:
                derived_sites = haplotypes[hap_id] > 0
                if np.sum(derived_sites) == 0:
                    root = 0  # no derived sites, so only needs to be below the root
                    for i, focal_sites in enumerate(self.ancestors_focal_sites):
                        if len(focal_sites) > 0:
                            if i > 0:
                                root = i - 1
                            anc_min_time[root] = proxy_times[hap_id] + epsilons[hap_id]
                            break
                else:
                    max_anc_id = np.max(site_ancestor[derived_sites])  # youngest ancstr
                    if max_anc_id >= 0:
                        anc_min_time[max_anc_id] = proxy_times[hap_id] + epsilons[hap_id]
            # Go from youngest to oldest, pushing up the times of the ancestors to
            # achieve compatibility with infinite sites.
            # TODO - replace with something mre efficient that uses time_diffs
            for anc_id in range(self.num_ancestors - 1, -1, -1):
                current_time = new_anc_times[anc_id]
                if anc_min_time[anc_id] > current_time:
                    new_anc_times[:(anc_id + 1)] += anc_min_time[anc_id] - current_time
            assert new_anc_times[1] > np.max(sample_times)  # root ancestor

        with self.__class__(  # Create new AncestorData instance to return
            variant_data.sites_position[:][used_sites],
            variant_data.sequence_length,
            **kwargs,
        ) as other:
            mutated_sites = set()  # To check if mutations have occurred yet
            ancestors_iter = self.ancestors()
            anc = next(ancestors_iter, None)
            for i in reverse_time_sorted_indexes:
                proxy_time = proxy_times[i]
                sample_id = sample_ids[i]
                haplotype = haplotypes[i]
                derived_sites = set(np.where(haplotype > 0)[0])
                while anc is not None and new_anc_times[anc.id] > proxy_time:
                    anc_time = new_anc_times[anc.id]
                    other.add_ancestor(
                        anc.start, anc.end, anc_time, anc.focal_sites, anc.haplotype)
                    mutated_sites.update(anc.focal_sites)
                    anc = next(ancestors_iter, None)
                if not derived_sites.issubset(mutated_sites):
                    assert not keep_ancestor_times
                    logging.info(
                        f"Infinite sites assumption deliberately broken: {sample_id}"
                        "contains an allele which requires a novel mutation."
                    )
                logger.debug(
                    f"Inserting proxy ancestor: sample {sample_id} at time {proxy_time}"
                )
                other.add_ancestor(
                    start=0,
                    end=self.num_sites,
                    time=proxy_time,
                    focal_sites=[],
                    haplotype=haplotype,
                )
            # Add any ancestors remaining in the current instance
            while anc is not None:
                anc_time = new_anc_times[anc.id]
                other.add_ancestor(
                    anc.start, anc.end, anc_time, anc.focal_sites, anc.haplotype,
                )
                anc = next(ancestors_iter, None)

            other.clear_provenances()
            for timestamp, record in self.provenances():
                other.add_provenance(timestamp, record)
            other.record_provenance(command="insert_proxy_samples", **kwargs)

        assert other.num_ancestors == self.num_ancestors + len(sample_ids)
        return other
```

simulation

In [129…
```
ts = msprime.sim_ancestry(
    samples = [
        msprime.SampleSet(10, time=0, ploidy = 1),
        msprime.SampleSet(10, time=10, ploidy = 1),
        msprime.SampleSet(5, time=50, ploidy = 1)
    ],
    population_size=1e4,
    ploidy = 1,
    sequence_length=1e6,
    random_seed=42
)


ts = msprime.sim_mutations(ts, rate = 1e-7, random_seed = 42)

sampling_times = ts.nodes_time[0:25]
```

(export vcf and) import as vcz

In [130…
```
# with open(f"sims/sim.vcf", "w") as fh:
#         ts.write_vcf(output = fh)
```

In [131…
```
# get ancestral states
ancestral_states = []

for site in ts.sites():
    if site.ancestral_state is None:
        ancestral_states.append("N")
    else:
        ancestral_states.append(str(site.ancestral_state))

ancestral_states = np.array(ancestral_states)
```

In [132…
```
vdata = tsinfer.VariantData("sims/sim.vcz",
                            ancestral_state=ancestral_states,
                           individuals_time=sampling_times)
```

In [133…
```
dated_ts, fit = tsdate.date(ts,
                        mutation_rate=1e-7,
                        time_units="years",
                        return_fit=True,
                        match_segregating_sites = True,
                        )
```

In [134…
```
samples = list(dated_ts.samples())
```

inference using proxy and push methods

1. insert_proxy_samples

i.e. insert proxy ancestral node for each old tip. proxy ancestors have identical haplotypes to their child tip. proxy ancestor node times fall slightly before their respective tips. since node times are frequency based, 'real' internal nodes are shifted upwards to accomodate the proxy nodes. now, 'real' internal nodes cannot be younger than any tips.

In [135…
```
tsinfer.AncestorData.insert_proxy_samples = insert_proxy_samples
```

In [136…
```
def run_proxy(vdata, rr_value = None):

    if rr_value != None:
        rr = rr_value
        mm = 1
    else:
        rr = None
        mm = None


    anc = tsinfer.generate_ancestors(vdata)
    anc_proxy = anc.insert_proxy_samples(vdata)
    anc_proxy_ts = tsinfer.match_ancestors(vdata, anc_proxy, recombination_rate = rr, mismatch_ratio = mm)
    ts_proxy = tsinfer.match_samples(vdata, anc_proxy_ts, force_sample_times=True, recombination_rate = rr)
    simplified_proxy = tsdate.preprocess_ts(ts_proxy, erase_flanks=False)

    dated_proxy = tsdate.date(simplified_proxy,
                        mutation_rate=1e-7,
                        time_units="years",
                        #return_fit=True,
                        match_segregating_sites = True,
```

```
                    )

    return dated_proxy
```

2. push internal nodes backwards

i.e. adding a bunch of time to internal node times in order to make space for old tips. these internal node times gets recalibrated to more reasonable times via tsdate

In [137…
```
def run_push(vdata, rr_value = None):

    if rr_value != None:
        rr = rr_value
        mm = 1
    else:
        rr = None
        mm = None

    anc = tsinfer.generate_ancestors(vdata)
    anc_ts = tsinfer.match_ancestors(vdata, anc, recombination_rate = rr, mismatch_ratio = mm)
    tables = anc_ts.dump_tables()
    tables.nodes.time += 1e6  # add a million on  to the ancestor times
    anc_ts = tables.tree_sequence()
    ts_push = tsinfer.match_samples(vdata, anc_ts, force_sample_times=True, recombination_rate = rr)
    simplified_push = tsdate.preprocess_ts(ts_push, erase_flanks = True)

    dated_push = tsdate.date(simplified_push,
                      mutation_rate=1e-7,
                      time_units="years",
                      #return_fit=True,
                      match_segregating_sites = True,
                      )

    return dated_push
```

validation testing

get intervals

In [138…
```
def get_intervals(dated_proxy):

    data = []

    for tree in dated_proxy.trees():
        left, right = tree.interval
        data.append({
            "tree_index": tree.index,
            "left": left,
            "right": right,
        })

    intervals_proxy = pd.DataFrame(data)

    return intervals_proxy
```

get pairwise mrcas for each sample in simulated geneology & corresponding mrcas in the inferred tree sequence

In [139…
```
def pairwise_mrcas(dated_proxy, dated_ts, samples):
    res = []

    for a, b in combinations(samples, 2):
        for i in range(1, dated_proxy.get_num_trees()-1):
            t = dated_proxy.at_index(i).tmrca(a, b)
            s = dated_ts.at_index(0).tmrca(a,b) # sim
            w = dated_proxy.at_index(i).interval.right - dated_proxy.at_index(i).interval.left
            res.append({"index": i, "sample_a": a, "sample_b": b, "proxy": t, "sim": s, "width": w})

    mrcas = pd.DataFrame(res)
    return mrcas
```

vary recombination rate during inference

In [140…
```
def rr(vdata):
```

```
    rates = [10**x for x in range(-8, 0, 1)] ## recombination_rate 1e-8 to 0.1
    rates.append(None)

    grid_num_trees = np.zeros(shape=(11, 1))
    seqs = [] #save each ts here

    count = 0

    for rr_idx, rr_value in enumerate(rates):
        count+=1

        ip = run_proxy(vdata, rr_value)

        grid_num_trees[rr_idx] = ip.num_trees

        seqs.append(ip)

        print(f"Finished inference {count}/{len(rates)}. Rate: {rr_value}, num trees: {ip.num_trees}")

    return seqs
```

In [141…

```
def rr_push(vdata):

    rates = [10**x for x in range(-8, 0, 1)] ## recombination_rate 1e-8 to 0.1
    rates.append(None)

    grid_num_trees = np.zeros(shape=(11, 1))
    seqs = [] #save each ts here

    count = 0

    for rr_idx, rr_value in enumerate(rates):
        count+=1

        ip = run_push(vdata, rr_value)

        grid_num_trees[rr_idx] = ip.num_trees

        seqs.append(ip)

        print(f"Finished inference {count}/{len(rates)}. Rate: {rr_value}, num trees: {ip.num_trees}")

    return seqs
```

linear regression

In [142…

```
def lg(mrcas):
    X = mrcas["proxy"].values.reshape(-1, 1)
    y = mrcas["sim"].values
    weights = mrcas["width"]/1e6
    reg = LinearRegression().fit(X, y, weights)
    return X, y, reg, weights
```

proxy method

run inference

In [143…

```
seqs = rr(vdata)
```

```
Finished inference 1/9. Rate: 1e-08, num trees: 3
Finished inference 2/9. Rate: 1e-07, num trees: 3
Finished inference 3/9. Rate: 1e-06, num trees: 3
Finished inference 4/9. Rate: 1e-05, num trees: 3
Finished inference 5/9. Rate: 0.0001, num trees: 3
Finished inference 6/9. Rate: 0.001, num trees: 3
Finished inference 7/9. Rate: 0.01, num trees: 3
Finished inference 8/9. Rate: 0.1, num trees: 3
Finished inference 9/9. Rate: None, num trees: 25
```

I think this is a good sign. Once I allow recombination, the number of trees shrinks down to 3. When we consider that the flanks aren't real (i.e. no mutations), tsinfer is correctly identifying that there is only 1 tree in the simulated geneology. Now we need to see if its estimating pairwise MRCAs correctly, too.

get mrca dfs for each tree

In [144…

```
rates = [10**x for x in range(-8, 0, 1)]
```

```
rates.append("None")
```

In [145…
```
mrcas_list = []

for id, ts in enumerate(seqs):
    df = pairwise_mrcas(ts, dated_ts, samples)
    df['rate'] = str(rates[id])
    mrcas_list.append(df)
```

do LR

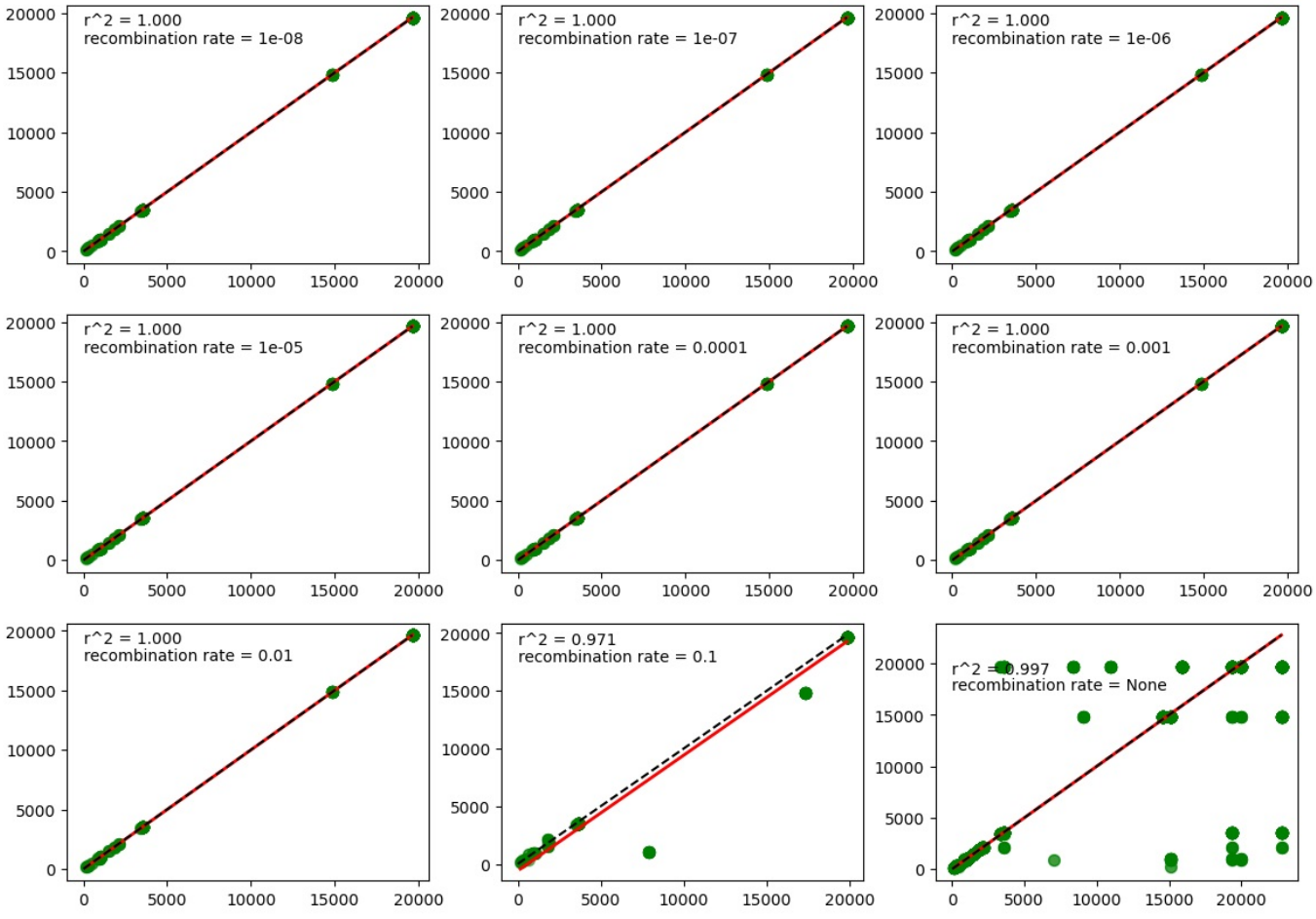
In [146…
```
regs = []

for id, df in enumerate(mrcas_list):
    X, y, reg, weights = lg(df)
    regs.append([df['rate'].unique()[0], X, y, reg, weights])
```

plot logistic regression of MRCAs for [proxy] tree sequence compared to simulated geneology.

weighted by tree width

In [147…
```
fig, arr = plt.subplots(3,3,figsize=(14,10))

for i, (rate, X, y, reg, weights) in enumerate(regs):

    row = i // 3
    col = i % 3

    arr[row][col].scatter(X, y, alpha=0.5, s=50, color = 'green')

    # regression line
    x_range = np.linspace(X.min(), X.max(), 500).reshape(-1, 1)
    arr[row][col].plot(x_range, reg.predict(x_range), color='red', linewidth=2)
    arr[row][col].annotate("r^2 = {:.3f}".format(reg.score(X, y, sample_weight=weights)), (1, 19000))
    arr[row][col].annotate(f'recombination rate = {rate}', (1, 17500))

    # diagonal
    max_val = max(X.max(), y.max())
    arr[row][col].plot([0, max_val], [0, max_val], 'k--')
fig.show()
```

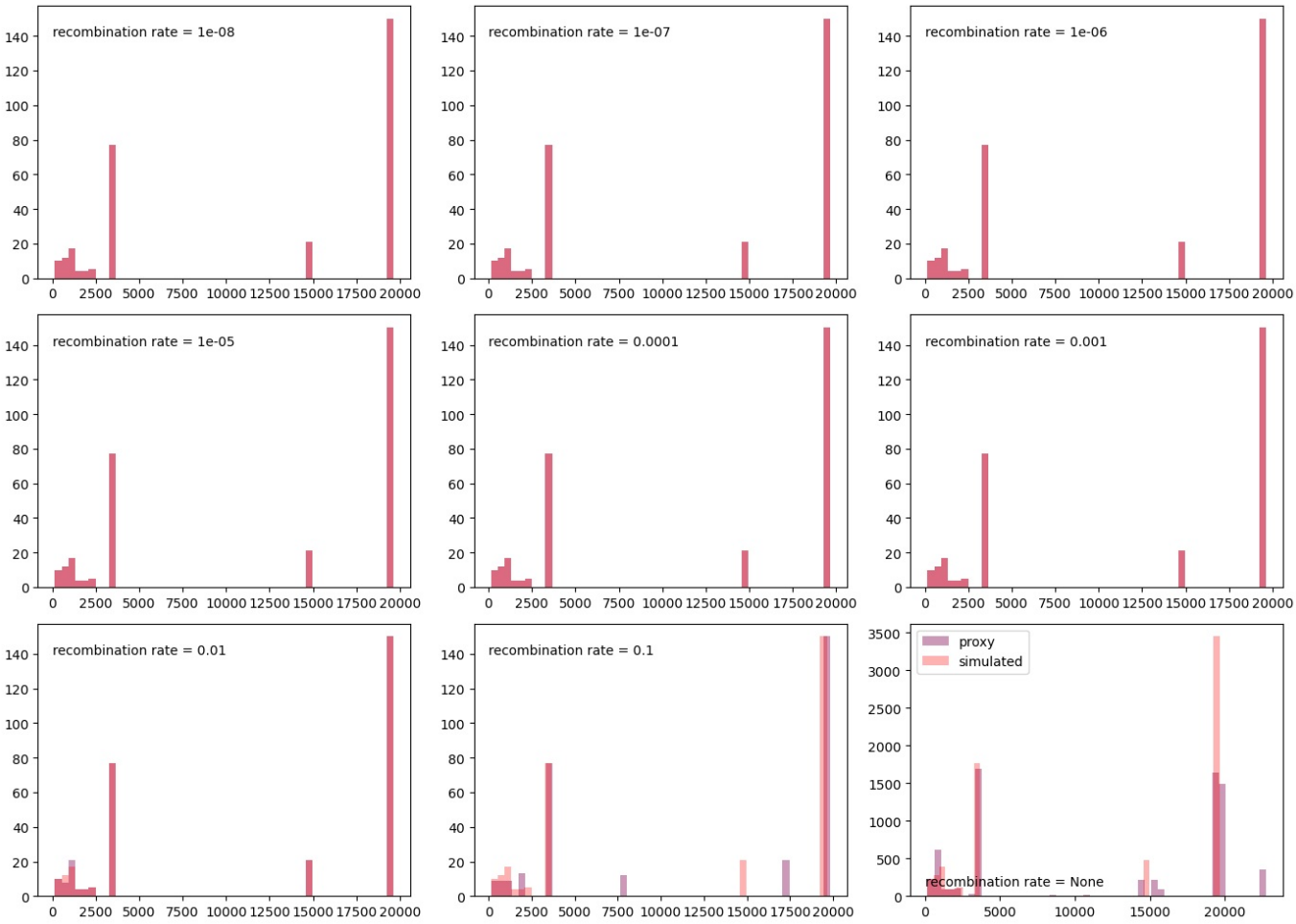
```
/loc/scratch/26305682/ipykernel_4337/189145873.py:19: UserWarning: Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



plot distribution of MRCAs

In [148…
```
fig, arr = plt.subplots(3,3,figsize=(14,10))

for i, df in enumerate(mrcas_list):

    row = i // 3
    col = i % 3

    rate = df['rate'].unique()[0]

    #plt.figure(figsize=(8, 5))

    arr[row][col].hist(
        df["proxy"],
        bins=50,
        alpha=0.6,
        color="#aa5589",
        label="proxy"
    )

    arr[row][col].hist(
        df["sim"],
        bins=50,
        alpha=0.3,
        color="red",
        label="simulated"
    )

    arr[row][col].annotate(f'recombination rate = {(rate)}', (1, 140))

    # plt.xlabel("time (years)")
    # plt.ylabel("frequency")
    # plt.title("distribution of MRCAs")
    plt.legend()
    plt.tight_layout()
fig.show()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
/loc/scratch/26305682/ipykernel_4337/2375097062.py:34: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
/loc/scratch/26305682/ipykernel_4337/2375097062.py:35: UserWarning: Matplotlib is currently using module://matpl
otlib_inline.backend_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



push method

In [149…
```
seqs = rr_push(vdata)
```

```
Finished inference 1/9. Rate: 1e-08, num trees: 3
Finished inference 2/9. Rate: 1e-07, num trees: 3
Finished inference 3/9. Rate: 1e-06, num trees: 3
Finished inference 4/9. Rate: 1e-05, num trees: 3
Finished inference 5/9. Rate: 0.0001, num trees: 3
Finished inference 6/9. Rate: 0.001, num trees: 3
Finished inference 7/9. Rate: 0.01, num trees: 3
Finished inference 8/9. Rate: 0.1, num trees: 3
Finished inference 9/9. Rate: None, num trees: 24
```

get mrca dfs for each tree

In [150…
```
mrcas_list = []

for id, ts in enumerate(seqs):
```

```
    df = pairwise_mrcas(ts, dated_ts, samples)
    df['rate'] = str(rates[id])
    mrcas_list.append(df)
```

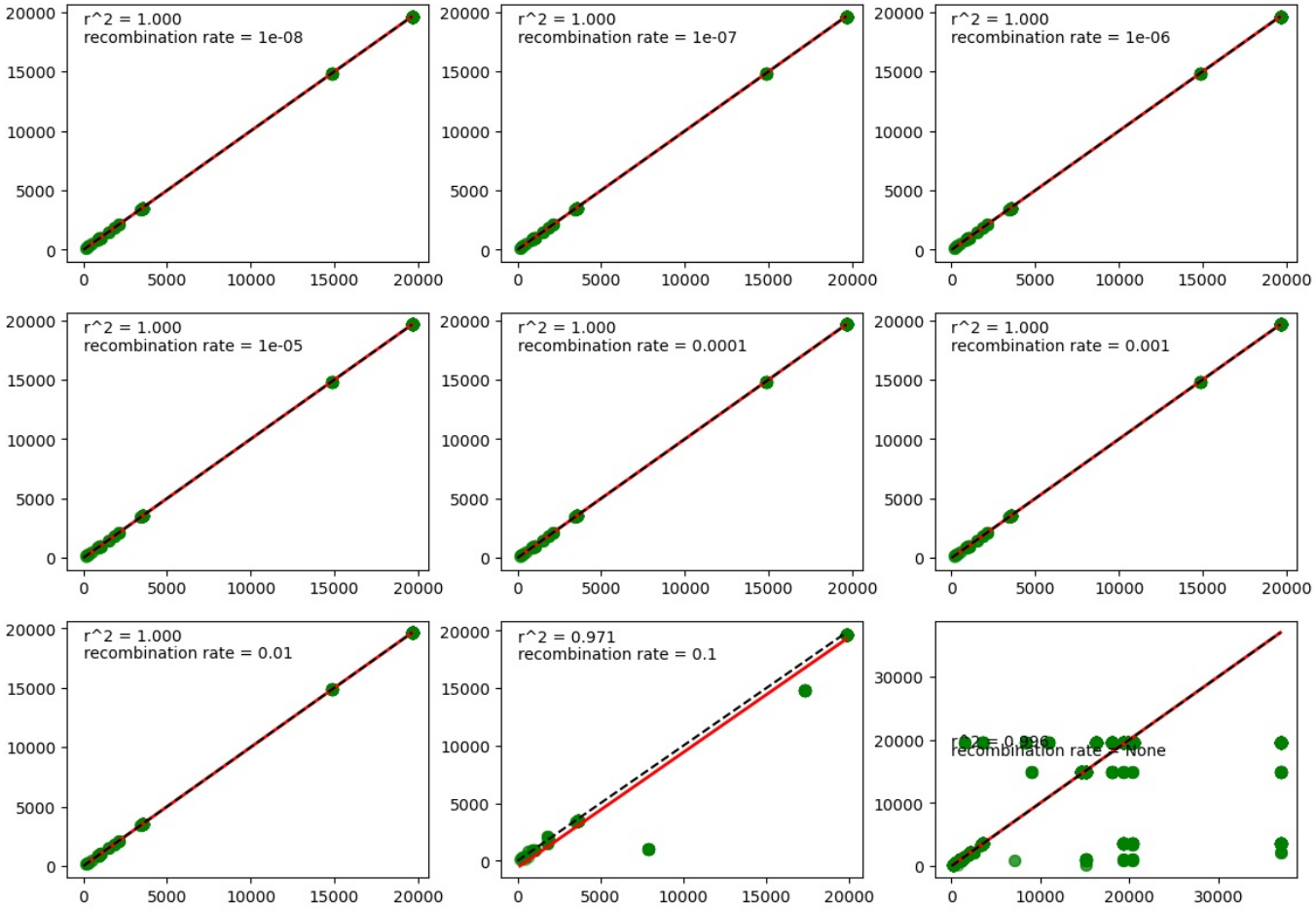
do LR

In [151…
```
regs = []

for id, df in enumerate(mrcas_list):
    X, y, reg, weights = lg(df)
    regs.append([df['rate'].unique()[0], X, y, reg, weights])
```

plot logistic regression of MRCAs for [push] tree sequence compared to simulated geneology.

In [152…
```
fig, arr = plt.subplots(3,3,figsize=(14,10))

for i, (rate, X, y, reg, weights) in enumerate(regs):

    row = i // 3
    col = i % 3

    arr[row][col].scatter(X, y, alpha=0.5, s=50, color = 'green')

    # regression line
    x_range = np.linspace(X.min(), X.max(), 500).reshape(-1, 1)
    arr[row][col].plot(x_range, reg.predict(x_range), color='red', linewidth=2)
    arr[row][col].annotate("r^2 = {:.3f}".format(reg.score(X, y, sample_weight=weights)), (1, 19000))
    arr[row][col].annotate(f'recombination rate = {rate}', (1, 17500))

    # diagonal
    max_val = max(X.max(), y.max())
    arr[row][col].plot([0, max_val], [0, max_val], 'k--')
fig.show()
```

```
/loc/scratch/26305682/ipykernel_4337/189145873.py:19: UserWarning: Matplotlib is currently using module://matplo
tlib_inline.backend_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```

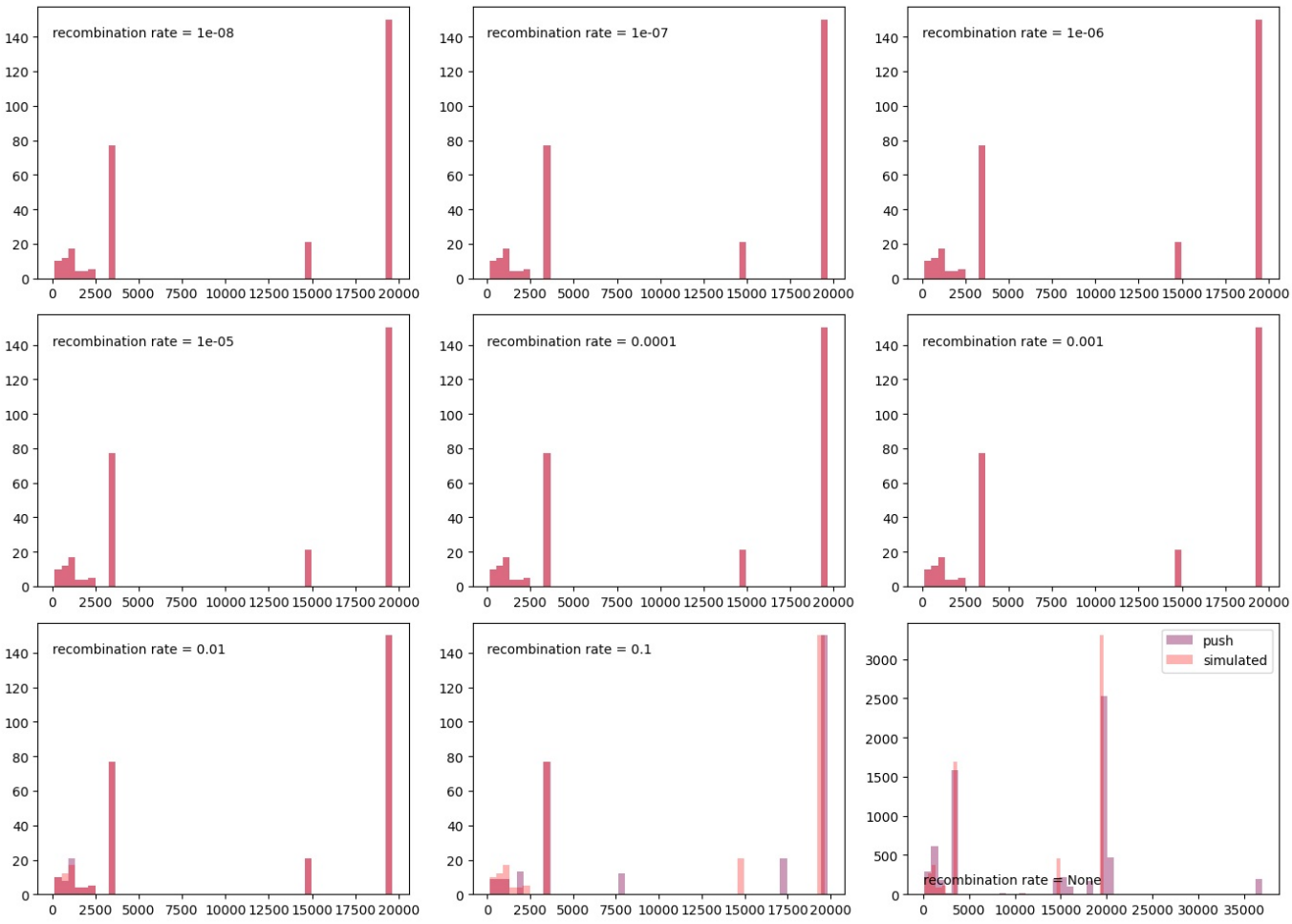


plot distribution of MRCAs

In [153…
```
fig, arr = plt.subplots(3,3,figsize=(14,10))

for i, df in enumerate(mrcas_list):
```

```python
    row = i // 3
    col = i % 3

    rate = df['rate'].unique()[0]

    #plt.figure(figsize=(8, 5))

    arr[row][col].hist(
        df["proxy"],
        bins=50,
        alpha=0.6,
        color="#aa5589",
        label="push"
    )

    arr[row][col].hist(
        df["sim"],
        bins=50,
        alpha=0.3,
        color="red",
        label="simulated"
    )

    arr[row][col].annotate(f'recombination rate = {(rate)}', (1, 140))

    # plt.xlabel("time (years)")
    # plt.ylabel("frequency")
    # plt.title("distribution of MRCAs")
    plt.legend()
    plt.tight_layout()
fig.show()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
/loc/scratch/26305682/ipykernel_4337/354673553.py:34: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose label start wi
th an underscore are ignored when legend() is called with no argument.
/loc/scratch/26305682/ipykernel_4337/354673553.py:35: UserWarning: Matplotlib is currently using module://matplo
tlib_inline.backend_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
```



taking a look at the last plot, with recombination_rate = None

For both proxy and push methods, num_trees ~25, many of which cover very short segments of the genome.

In [165…

```
seqs[8] #tree sequence for recombination_rate = None
```

Out[165…

ts kit **Tree Sequence**

Trees	24
Sequence Length	1,000,000.0
Time Units	years
Sample Nodes	25
Total Size	1.0 MiB
Metadata	▼ dict

Table	Rows	Size	Has Metadata
Edges	129	4.0 KiB	
Individuals	25	2.1 KiB	✅
Migrations	0	8 Bytes	
Mutations	7,855	648.7 KiB	✅
Nodes	73	5.9 KiB	✅
Populations	0	24 Bytes	
Provenances	5	3.8 KiB	
Sites	7,843	391.6 KiB	✅

Provenance Timestamp	Software Name	Version	Command	Full record
15 July, 2025 at 12:28:46 PM	tsdate	0.2.3	variational_gamma	▶ Details
15 July, 2025 at 12:28:46 PM	tsdate	0.2.3	preprocess_ts	▶ Details
15 July, 2025 at 12:28:46 PM	tsinfer	0.4.1	match_samples	▶ Details
15 July, 2025 at 12:28:45 PM	tsinfer	0.4.1	match_ancestors	▶ Details
15 July, 2025 at 12:28:45 PM	tsinfer	0.4.1	generate_ancestors	▶ Details

In [162…
```
mrcas_list[8] #column entitled 'proxy' actually refers to push-method MRCAs
```

Out[162…

	index	sample_a	sample_b	proxy	sim	width	rate
0	1	0	1	1829.507946	1834.813713	179511.0	None
1	2	0	1	1829.507946	1834.813713	91161.0	None
2	3	0	1	1829.507946	1834.813713	52265.0	None
3	4	0	1	1829.507946	1834.813713	1274.0	None
4	5	0	1	1829.507946	1834.813713	161.0	None
…	…	…	…	…	…	…	…
6595	18	23	24	19326.891216	19645.552454	14986.0	None
6596	19	23	24	19326.891216	19645.552454	7595.0	None
6597	20	23	24	19326.891216	19645.552454	268.0	None
6598	21	23	24	19326.891216	19645.552454	24741.0	None
6599	22	23	24	19326.891216	19645.552454	492508.0	None

6600 rows × 7 columns

Comparing MRCAs b/w simulated tree and each of the 24 inferred trees

In [161…
```
# vary alpha for each tree interval

import seaborn as sns

widths = np.sort(mrcas_list[8]["width"].unique())
palette = sns.color_palette("light:#5A9", n_colors=len(widths))
color_map = dict(zip(widths, palette))

# map interval colors to points
colors = mrcas_list[8]["width"].map(color_map)

plt.figure(figsize=(9, 6))

plt.scatter(
    mrcas_list[8]["proxy"],
   mrcas_list[8]["sim"],
    c=colors,
    alpha=0.2,
    s=50,
    label="PUSH vs SIM"
)

# diagonal
max_val = max(mrcas_list[8]["proxy"].max(), mrcas_list[8]["sim"].max())
plt.plot([0, max_val], [0, max_val], 'k--', label="y = x")
```

```
plt.xlabel("push method")
plt.ylabel("simulated geneology")
plt.title("mrca(a,b)")

# legend for intervals
handles = [
    plt.Line2D([0], [0], marker='o', color='w', label=width,
               markerfacecolor=color_map[width], markersize=6)
    for width in widths
]
plt.legend(handles=handles, title="tree width", bbox_to_anchor=(1.05, 1), loc="upper left")

plt.tight_layout()
plt.show()
```