Федеральное государственное автономное образовательное учреждение высшего профессионального образования

"Национальный исследовательский университет "Высшая школа экономики"

Московский институт электроники и математики Национального исследовательского университета "Высшая школа экономики"

Департамент прикладной математики

Отчет по проектной работе на тему:

"Определение происхождения особи методами машинного обучения"

Работу выполнили студенты группы БПМ 202:

Захаров Фёдор Алексеевич
Молоканов Руслан Алексеевич

Руководитель проекта:
Щур Владимир Львович

Москва 2021

Содержание

1 Введение и цели

В настоящее время проблемы популяционной геномики привлекают все больше внимания научно-исследовательских масс. Особенно интересны задачи на стыке генетических исследований и машинного обучения. В данном проекте перед нами встала задача предсказания происхождения особи методами машинного обучения. Сперва нами была создана демографическая модель, состоящая из двух популяций. Затем были построены деревья симуляций потомства. Далее нами была получена матрица генотипов, которая и послужила датасетом для решаемой задачи. Условно нашу задачу можно разбить на два этапа: генерация данных и их анализ.

2 Генерация данных

Теория объединения - это модель того, как варианты генов, отобранные из популяции, могли происходить от общего предка. В простейшем случае теория слияния предполагает отсутствие рекомбинации (которая всё-таки пресутсвует у нас), естественного отбора , потока генов (который реализован у нас) или популяционной структуры, а это означает, что каждый вариант с одинаковой вероятностью передавался от одного поколения к другому. Модель смотрит назад во времени,

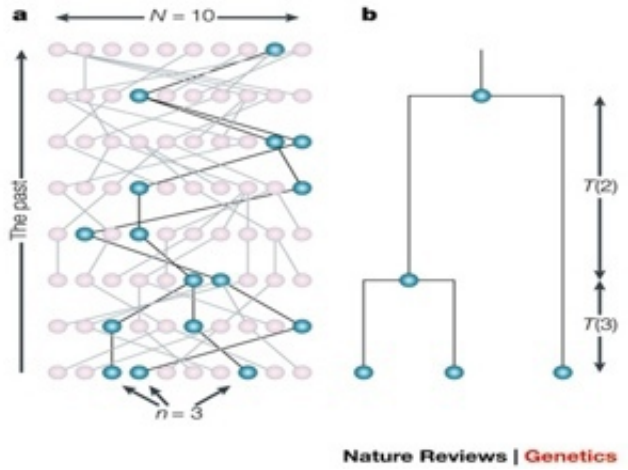


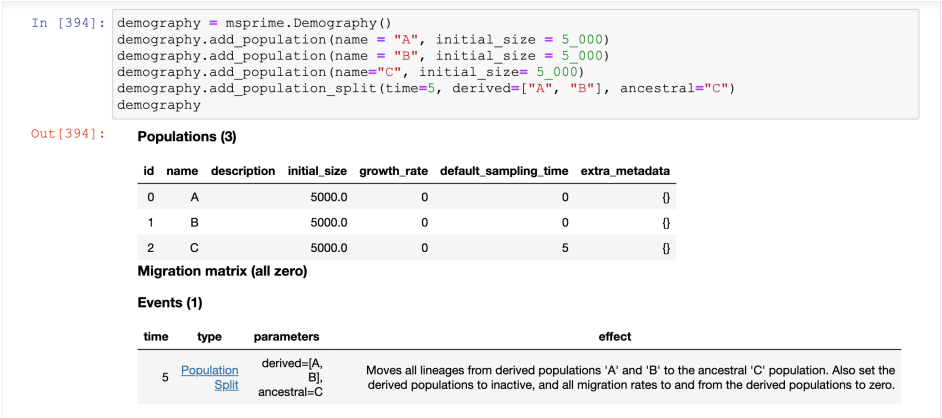
Рис. 1: Древа поколений, размещённого назад во времени

объединяя аллели в единую предковую копию в соответствии со случайным процессом в событиях слияния. Различия в модели возникают как из-за случайной передачи аллелей от одного поколения к другому, так и из-за случайного появления мутаций в этих аллелях (которые так же реализованны в нашей работе).

Нашей задачей было определить происхождение особи исходя из этой модели. Такая задача в свою очередь решается в несколько этапов.

2.1 Демография. Структура

На этом этапе мы задаём особенности демографической структуры нашей модели. Она включает в себя: собственно популяции, события определённые для них и



```
In [394]: demography = msprime.Demography()
demography.add_population(name = "A", initial_size = 5_000)
demography.add_population(name = "B", initial_size = 5_000)
demography.add_population(name="C", initial_size= 5_000)
demography.add_population_split(time=5, derived=["A", "B"], ancestral="C")
demography
```

Out[394]:

Populations (3)

id	name	description	initial_size	growth_rate	default_sampling_time	extra_metadata
0	A		5000.0	0	0	{}
1	B		5000.0	0	0	{}
2	C		5000.0	0	5	{}

Migration matrix (all zero)

Events (1)

time	type	parameters	effect
5	Population Split	derived=[A, B], ancestral=C	Moves all lineages from derived populations 'A' and 'B' to the ancestral 'C' population. Also set the derived populations to inactive, and all migration rates to and from the derived populations to zero.

Рис. 2: Моделиривование демографии

матрицы миграции (некие параметры их перемещений). Здесь первой строчкой мы определяем объект типа демография, далее в нём определяем объекты класса популяция (A ,B и C) с соответствующими размерами в нулевой момент времени. Последней строкой задаём событие "разделение популяции"с характерными для него параметрами: временем который он занимает и структурой расщипления.

2.2 Поток генов (Миграция)

```
rate = demography.migration_matrix[0, 1]
demography.set_symmetric_migration_rate(demography, rate )
```

Рис. 3: Реализация Миграция

Следующим шагом зададим миграцию в нашей системе. Для этого первым шагом обозначим между какими популяциями она будет происходить. Это делается через конструкцию migration matrix в первой строке. Далее зададим непосредственно процесс симметричной миграции (такой при которой особи пермещаются между популяциями в обе стороны), обозначив в какой демографической модели это происходит.

2.3 Родовое древо

Этот раздел состоит всего из двух строк кода, но при этом является ключевым для понимания. Класс tskit.TreeSequence представляет собой последовательность коррелированных эволюционных деревьев вдоль генома. Класс tskit.Tree представляет одно дерево в этой последовательности. Эти классы являются интерфейсами, используемыми для взаимодействия с деревьями и мутационной информацией, хранящейся в древовидной последовательности. Эта библиотека также предоставляет методы для загрузки сохраненных последовательностей дерева, например, с помощью tskit.load(). В нашем случае дерево генерируется методом sim ancestry, для заранее заданного числа особей.

```
n_samples = 5
ts = msprime.sim_ancestry({"A": n_samples, "B": n_samples},
                          demography = demography,
                          sequence_length=500, random_seed=18)
```

Рис. 4: Реализация Родового Дерева

2.4 Мутации

```
gmatrix = np.array([])

flag = 0
for i in range(500):
    mts = msprime.sim_mutations(ts, rate=1e-4)
    for variant in mts.variants():
        gmatrix = np.append(gmatrix, np.expand_dims(variant.genotypes, axis=0), axis=0)
        if flag else np.expand_dims(variant.genotypes, axis=0)
        flag = 1
        break
```

Рис. 5: Реализация Мутации

Объект класса Variant представляет собой наблюдаемую вариацию между образцами для данного участка. Объект состоит из ссылки на рассматриваемый Site-экземпляр, аллелей, которые могут наблюдаться в образцах для этого сайта, и генотипов, сопоставляющих идентификаторы образцов с наблюдаемыми аллелями. Это позволяет создать модель учитывающую мутации.

3 Машинное обучение

3.1 Предобработка сгенерированных данных

После генерации матрицы генотипов мы произвели разбиение датасета на обучающую и тестовую выборки. Для этого была использована функция train test split из модуля sklearn.model selection. Долю тестовых образцов от общего числа данных мы выбрали равной 0,33.

3.2 Полносвязная нейронная сеть

В качестве первого алгоритма для предсказания метки популяции, к которой принадлежит особь, мы выбрали полносвязную нейронную сеть. Перед тем, как перейти к пониманию работы нейронной сети, разберемся с тем, как устроен искусственный нейрон. Искусственный нейрон — это очень отдаленное подобие биологического нейрона. По сути он представляет из себя математическую функцию. У нее есть входы, каждый умножается на некие веса, дальше всё суммируется, прогоняется через какую-то нелинейную функцию, результат выдается на выход. Искусственная нейросеть — это способ собрать нейроны в сеть, которая будет решать определенную задачу, например, задачу классификации. Нейроны собираются по слоям. Есть входной слой, куда подается входной сигнал, есть выходной слой, откуда снимается результат работы нейросети. Между ними располагаются скрытые слои. Их может быть сколь угодно много. Если скрытых слоев больше, чем один, нейросеть считается глубокой, если всего один, то неглубокой.

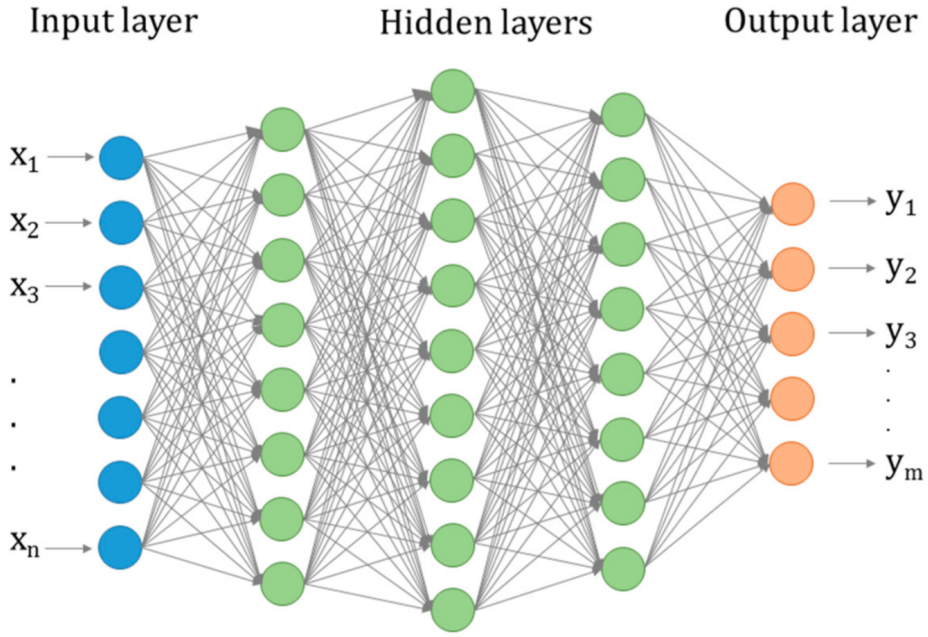


Рис. 6: структура нейронной сети

Для решения нашей задачи мы построили нейронную сеть следующей архитектуры:

```
class GenNet(torch.nn.Module):
  def __init__(self, n_hidden_neurons):
    super(GenNet, self).__init__()

    self.fc1 = torch.nn.Linear(20, n_hidden_neurons)
    self.act1 = torch.nn.Tanh()
    self.fc2 = torch.nn.Linear(n_hidden_neurons, n_hidden_neurons)
    self.act2 = torch.nn.Tanh()
    self.fc3 = torch.nn.Linear(n_hidden_neurons, 1)
    self.act3 = torch.nn.Sigmoid()

  def forward(self, x):
    x = self.fc1(x)
    x = self.act1(x)
    x = self.fc2(x)
    x = self.act2(x)
    x = self.fc3(x)
    x = self.act3(x)
    return x
```

Рис. 7: архитектура сети

Сеть GenNet принимает на вход признаковое описание объекта в виде вектора из двадцати компонент. За входным слоем следуют два скрытых полносвязных слоя. На выходном слое находится один нейрон. После каждого слоя (за исключением последнего) данные проходят через нелинейную активацию – функцию гиперболического тангенса. На выходе из сети мы разместили сигмоидную функцию активации, чтобы результат прогона данных по слоям можно было интерпретировать как вероятность принадлежности объекта к популяции. Теперь приступим к обучению нейронной сети. Для этого нам понадобится подобрать функцию потерь. С помощью нее осуществляется настройка параметров. Наша цель - минимизировать потери для нейронной сети путем оптимизации ее параметров (весов). Потери рассчитываются с использованием функции потерь путем сопоставления целевого (фактического) значения и прогнозируемого значения нейронной сетью. Затем мы используем метод градиентного спуска для обновления весов нейронной сети таким образом, чтобы потери были сведены к минимуму. В качестве функции потерь была выбрана бинарная кросс-энтропия:

В качестве оптимизатора сети мы выбрали алгоритм модифицированного градиентного спуска – Adam (Adaptive Moment Estimation). Adam bявляется продолжением градиентного спуска и естественным преемником таких методов, как AdaGrad и RMSProp, который автоматически адаптирует скорость обучения для каждой входной переменной для целевой функции и дополнительно сглаживает процесс поиска, используя экспоненциально убывающее скользящее среднее градиента для обновления переменных. Сам процесс обучения нейронной сети был реализован нами в двух различных вариациях (двух различных тренировочных циклах. В первом цикле мы формируем батчи вручную внутри каждой из эпох:

```
batch_size = 10
accuracies = []

for epoch in range(5000):
  order = np.random.permutation(len(X_train))
  for start_index in range(0, len(X_train), batch_size):
    optimizer.zero_grad()
    batch_indexes = order[start_index : start_index + batch_size]

    x_batch = X_train[batch_indexes].to(device)
    y_batch = y_train[batch_indexes].to(device)

    preds = model.forward(x_batch)

    loss_value = loss(preds, y_batch.reshape(-1, 1).type(torch.FloatTensor).to(device))
    loss_value.backward()

    optimizer.step()

  if epoch % 500 == 0:
      test_preds = torch.round(model.forward(X_test))
      acc = (test_preds == y_test).float().mean()
      print(f'Train loss: {loss_value}, Test loss: {loss(model.forward(X_test), y_test.reshape(-1, 1).type(torch.FloatTensor).to(device))}')
      accuracies.append(acc)
```

Рис. 8: train loop 1

Во второй вариации тренировочного цикла для побатчевой загрузки данных в сеть мы воспользовались специальными объектами TensorDataset и DataLoader из модуля torch.utils.data.

```
from torch.utils.data import TensorDataset, DataLoader

train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_test, y_test)


train_dataloader = DataLoader(train_dataset, batch_size=16)
test_dataloader = DataLoader(val_dataset, batch_size=16)
```

Рис. 9: objects from torch.utils.data

```
losses = []
accur = []
for i in range(50):
  for j, (X_train, y_train) in enumerate(train_dataloader):

    #calculate output
    output = model.forward(X_train)

    #calculate loss
    loss_value = loss(output, y_train.reshape(-1,1).type(torch.FloatTensor).to(device))

    #accuracy
    predicted = model.forward(torch.tensor(X_train, dtype=torch.float32))
    acc = torch.mean((torch.round(predicted) == y_train).type(torch.FloatTensor))

    #backprop
    optimizer.zero_grad()
    loss_value.backward()
    optimizer.step()

  if i % 5 == 0:
    losses.append(loss)
    accur.append(acc)
    print("epoch {}\tloss : {}\t accuracy : {}".format(i, loss_value, acc))
```

Рис. 10: train loop 2

В первом цикле размер батча составляет 10 элементов, во втором – 16. Так же в обоих случаях (при запуске кода в среде Google Colaboratory) мы переносим вычисления на GPU.

```
Train loss: 0.6986396908760071, Test loss: 0.6943711638450623
Train loss: 0.5065546035766602, Test loss: 1.2324128150939941
Train loss: 0.3927857577800751, Test loss: 2.9605772495269775
Train loss: 0.22393552958965302, Test loss: 3.697453260421753
Train loss: 0.6243859529495239, Test loss: 5.026788711547852
Train loss: 0.22848919034004211, Test loss: 5.378793716430664
Train loss: 0.23388849198818207, Test loss: 5.606553077697754
Train loss: 0.2646511197090149, Test loss: 6.470783710479736
Train loss: 0.2795165777206421, Test loss: 6.53266716003418
Train loss: 0.0043118721805512905, Test loss: 6.629312038421631
```

Рис. 11: функция потерь нейронной сети

Выведем функцию потерь нейронной сети на тренировочных данных и тестовых данных. Вывод осуществляем каждые 500 эпох.

3.3 Случайный лес

Помимо нейронной сети мы решили протестировать еще несколько алгоритмоы обучения для решения нашей задачи. Одна из моделей – случайный лес. Случайный лес — один из самых потрясающих алгоритмов машинного обучения, придуманные Лео Брейманом и Адель Катлер ещё в прошлом веке. Он дошёл до нас в

«первозданном виде» (никакие эвристики не смогли его существенно улучшить) и является одним из немногих универсальных алгоритмов. RF (random forest) — это множество решающих деревьев. В задаче регрессии их ответы усредняются, в задаче классификации принимается решение голосованием по большинству. Все деревья строятся независимо по следующей схеме:

- Выбирается подвыборка обучающей выборки размера samplesize (м.б. с возвращением) – по ней строится дерево (для каждого дерева — своя подвыборка)
- Для построения каждого расщепления в дереве просматриваем max features случайных признаков (для каждого нового расщепления — свои случайные признаки)
- Выбираем наилучшие признак и расщепление по нему (по заранее заданному критерию). Дерево строится, как правило, до исчерпания выборки (пока в листьях не останутся представители только одного класса), но в современных реализациях есть параметры, которые ограничивают высоту дерева, число объектов в листьях и число объектов в подвыборке, при котором проводится расщепление.

В качестве гиперпараметров для модели случайного леса мы выбрали к-во базовых алгоритмов, равное 100, и максимальную глубину деревьев, равную 5.

На следующем рисунке показано, как отрабатывает случайный лес на нашей задаче в зависимости от к-ва объектов в обучающей выборке

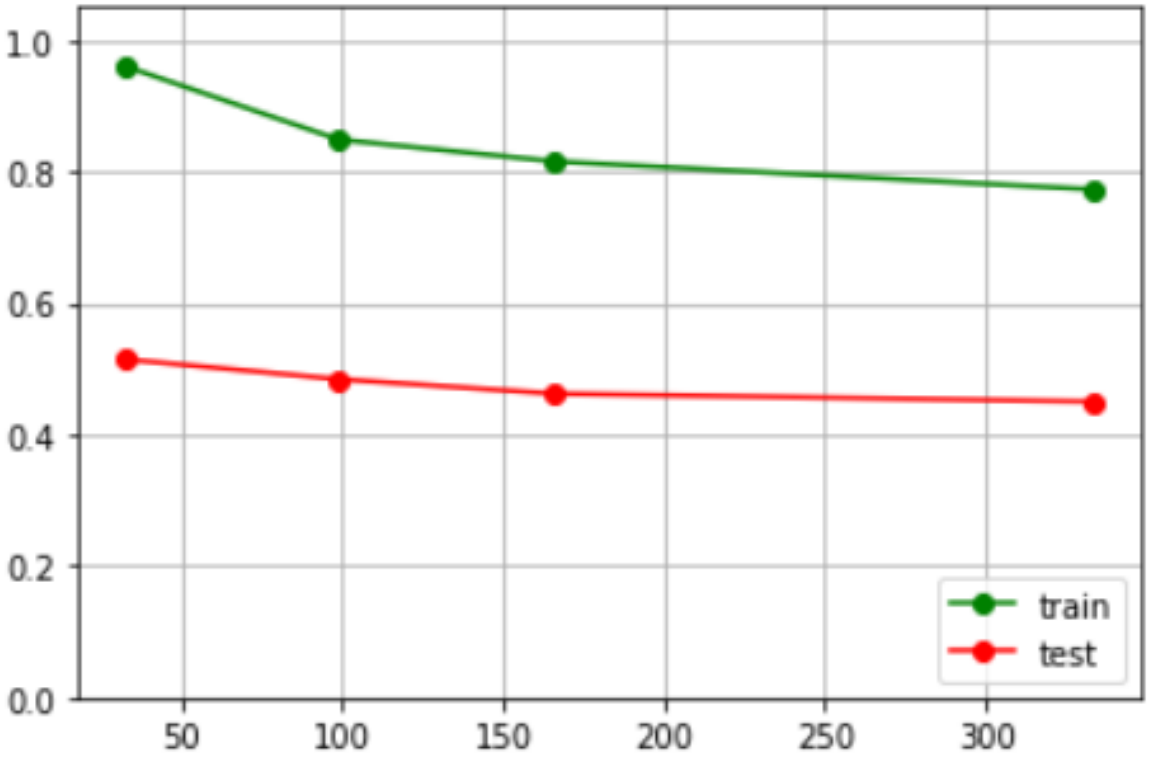


Рис. 12: learning curve RF

3.4 Градиентный бустинг

Последней моделью, которой мы воспользовались, стал градиентный бустинг над случайными деревьями. Бустинг — это техника построения ансамблей, в которой предсказатели построены не независимо, а последовательно. Техника использует идею о том, что следующая модель будет учится на ошибках предыдущей. Они имеют неравную вероятность появления в последующих моделях, и чаще появятся те, что дают наибольшую ошибку. Предсказатели могут быть выбраны из широкого ассортимента моделей. Из-за того, что предсказатели обучаются на ошибках, совершенных предыдущими, требуется меньше времени для того, чтобы добраться до реального ответа. Но мы должны выбирать критерий остановки с осторожностью, иначе это может привести к переобучению. Градиентный бустинг — это пример бустинга. Градиентный бустинг — это техника машинного обучения для задач классификации и регрессии, которая строит модель предсказания в форме ансамбля слабых предсказывающих моделей, обычно деревьев решений. В качестве гиперпараметров для модели градиентного бустинга мы выбрали скорость обучения, равную 0.01, и максимальную глубину деревьев, равную 5. На рисунке ниже изображен график качества работы градиентного бустинга на нашей задаче в зависимости от количества базовых алгоритмов в ансамбле.

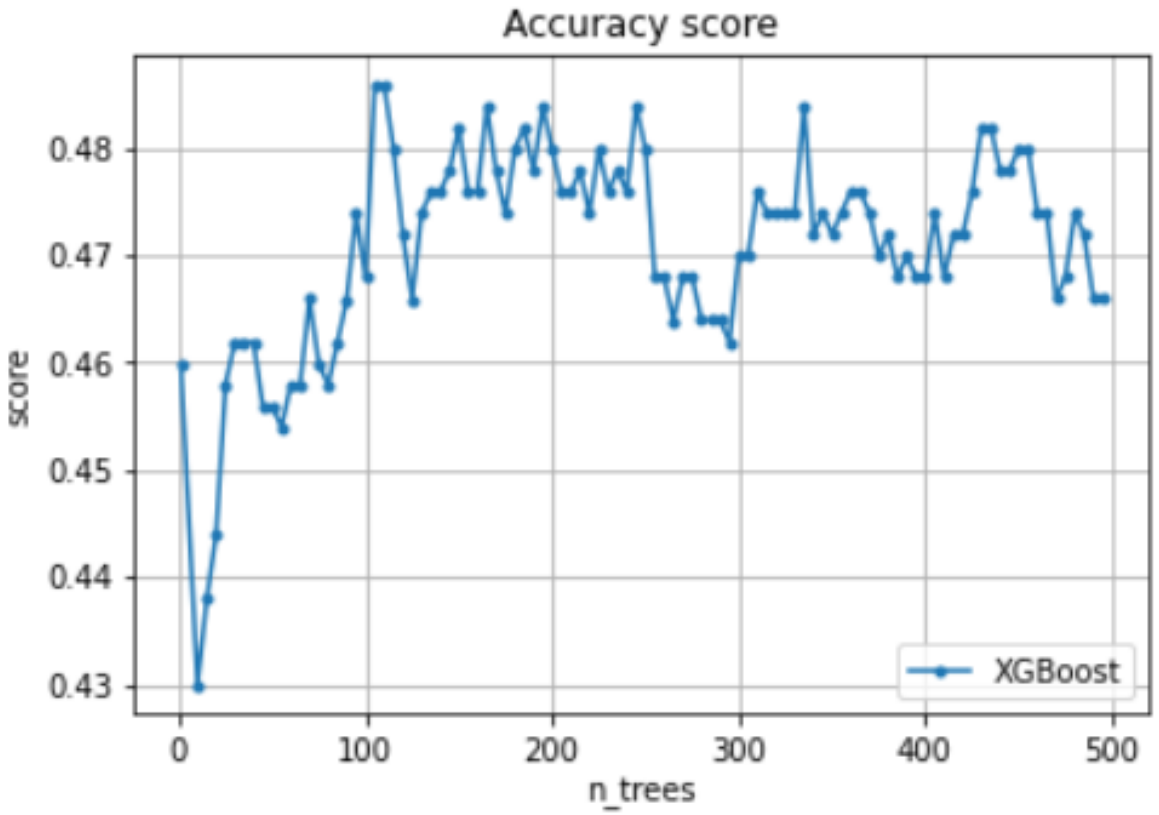


Рис. 13: Boosting accuracy

4 Итог

Итак, мы применили для решения задачи определения генетического происхождения особи несколько методов машинного обучения (в том числе и глубокое обучение). Но высокого качества предсказаний не удалось добиться ни одним методом. Отсюда можно сделать вывод, что зависимость между генотипами особей, которые мы сгенерировали, и принадлежностью особей к конкретным популяциям довольно сложно восстанавливается.