MSc in Computer Science 2020-21

Project Dissertation Code

Project Dissertation title: Deep learning for tabular data in healthcare

Term and year of submission: Trinity Term 2021

Candidate Number: 1047662

Code

All code was run in Python v3.6. All package dependencies of models, normalising flows and regularization techniques are specified in the original / existing implementations of them which this study built on, which are referenced in each section.

Part 1: Synthetic tabular data generation

CTGAN and TVAE implementations in the Synthetic Data Vault (SDV, 2021) were used. The vanilla TVAE implementation was extended with normalising flows based on Huang (2021), using the code below:

```
import numpy as np
import torch
from torch.nn import Linear, Module, Parameter, ReLU, Sequential, Softplus,
Tanh, ModuleList
from torch.nn.functional import cross_entropy
from torch.optim import Adam
from torch.utils.data import DataLoader, TensorDataset

from ctgan.data_transformer import DataTransformer


class EncoderFlows(Module):
    def __init__(self, input_dim, hidden_dims, embed_dim):
        super(EncoderFlows, self).__init__()

        dim = input_dim
        layers = []
        for hidden_dim in list(hidden_dims):
            layers += [Linear(dim, hidden_dim), ReLU()]
            dim = hidden_dim
        self.layers = Sequential(*layers)
        self.fc1 = Linear(dim, embed_dim)
        self.fc2 = Sequential(Linear(dim, embed_dim), Softplus())

    def forward(self, x):
        out = self.layers(x)
        mu = self.fc1(out)
```

```
        var = self.fc2(out)
        std = var.sqrt()

        return mu, std, var, out


class Decoder(Module):
    def __init__(self, embed_dim, hidden_dims, input_dim):
        super(Decoder, self).__init__()

        dim = embed_dim
        layers = []
        for hidden_dim in list(hidden_dims):
            layers += [Linear(dim, hidden_dim), ReLU()]
            dim = hidden_dim

        layers.append(Linear(dim, input_dim))
        self.layers = Sequential(*layers)
        self.sigma = Parameter(torch.ones(input_dim) * 0.1)

    def forward(self, input):
        return self.seq(input), self.sigma


def distribution(x, mean, logvar, dim):
    y = -0.5 * (logvar + (x - mean).pow(2) * logvar.exp().reciprocal())

    return torch.sum(y, dim)


def loss_function_flows(recon_x, x, sigmas, mu, var, z0, zk, log_jacob_det,
embed_dim, output_info, factor):
    st = 0
    loss = []
    for column_info in output_info:
        for span_info in column_info:
            if span_info.activation_fn != "softmax":
                ed = st + span_info.dim
                std = sigmas[st]
                loss.append(((x[:, st] - torch.tanh(recon_x[:, st])) ** 2 / 2
/ (std ** 2)).sum())
                loss.append(torch.log(std) * x.size()[0])
                st = ed

            else:
                ed = st + span_info.dim
                loss.append(cross_entropy(
                    recon_x[:, st:ed], torch.argmax(x[:, st:ed], dim=-1),
reduction='sum'))
                st = ed

    assert st == recon_x.size()[1]
    log_pzk = distribution(zk, 0, torch.zeros(x.size(0), embed_dim), 1)
    log_qz0 = distribution(z0, mu, var.log(), 1)
    KLD = torch.sum(log_qz0 - log_pzk) - torch.sum(log_jacob_det)

    return sum(loss) * factor / x.size()[0], KLD / x.size()[0]
```

```
class Planar(Module):
    def __init__(self):
        super(Planar, self).__init__()

        self.h = Tanh()

    def forward(self, z, u, w, b):
        z = z.unsqueeze(2)
        fz = z + u * self.h(torch.bmm(w, z) + b)
        fz = fz.squeeze(2)

        dfdz = w * (1 - self.h(torch.bmm(w, z) + b) ** 2)
        log_jacob_det = torch.log(torch.abs(1 + torch.bmm(dfdz, u)))
        log_jacob_det = log_jacob_det.squeeze(2).squeeze(1)

        return fz, log_jacob_det


class Sylvester(Module):
    def __init__(self):
        super(Sylvester, self).__init__()

        self.h = Tanh()

    def forward(self, z, Q, R, R_tilde, b):
        z = z.unsqueeze(2)
        b = b.unsqueeze(2)
        QR = torch.bmm(Q, R)
        RQT = torch.bmm(R_tilde, Q.transpose(2, 1))

        fz = z + torch.bmm(QR, self.h(torch.bmm(RQT, z) + b))
        fz = fz.squeeze(2)

        dhdz = (1 - self.h(torch.bmm(RQT, z) + b) ** 2).squeeze(2)
        R_diag = torch.diagonal(R, dim1=1, dim2=2)
        R_tilde_diag = torch.diagonal(R_tilde, dim1=1, dim2=2)
        RR_diag = R_diag * R_tilde_diag
        dfdz = dhdz * RR_diag
        log_jacob_det_diag = (1 + dfdz).abs().log()
        log_jacob_det = log_jacob_det_diag.sum(-1)

        return fz, log_jacob_det


class NICECoupling(Module):
    def __init__(self, in_out_dim, hidden_dim, num_layers):
        super(NICECoupling, self).__init__()

        self.in = Sequential(Linear(in_out_dim // 2, hidden_dim), ReLU())
        self.hidden = ModuleList([Sequential(Linear(hidden_dim, hidden_dim),
                                  ReLU()) for _ in range(num_layers - 1)])
        self.out = Linear(hidden_dim, in_out_dim // 2)
        self.perm = torch.eye(in_out_dim)[torch.randperm(in_out_dim), :]
        self.permT = self.perm.t()
```

```
    def forward(self, x):
        [B, W] = list(x.size())
        x = x @ self.perm
        x = x.reshape((B, W // 2, 2))
        z1, z2 = x[:, :, 0], x[:, :, 1]

        z1_ = self.in(z1)
        for layer in range(len(self.hidden)):
            z1_ = self.hidden[layer](z1_)
        mz1 = self.out(z1_)

        z2 = z2 + mz1

        x = torch.stack((z1, z2), dim=2)
        x = x.reshape((B, W))
        x = x @ self.permT

        return x


class NICEScaling(Module):
    def __init__(self, dim):
        super(NICEScaling, self).__init__()

        self.scale = Parameter(torch.zeros((1, dim)), requires_grad=True)

    def forward(self, x):
        x = x * torch.exp(self.scale)
        log_jacob_det = torch.sum(self.scale, dim=1)

        return x, log_jacob_det


class RealNVP(Module):
    def __init__(self, in_out_dim, hidden_dim, mask):
        super(RealNVP, self).__init__()
        self.mask = mask

        self.scale = Sequential(Linear(in_out_dim // 2, hidden_dim), Tanh(),
Linear(hidden_dim, hidden_dim), Tanh(), Linear(hidden_dim, hidden_dim),
Tanh(), Linear(hidden_dim, hidden_dim), Tanh(), Linear(hidden_dim,
hidden_dim), Tanh(), Linear(hidden_dim, in_out_dim // 2))
        self.translate = Sequential(Linear(in_out_dim // 2, hidden_dim),
Tanh(), Linear(hidden_dim, hidden_dim), Tanh(), Linear(hidden_dim,
hidden_dim), Tanh(), Linear(hidden_dim, hidden_dim), Tanh(),
Linear(hidden_dim, hidden_dim), Tanh(), Linear(hidden_dim, in_out_dim // 2))

    def forward(self, x):
        z1, z2 = x[:, ::2], x[:, 1::2]

        if self.mask:
            z1, z2 = z2, z1

        sz1 = self.scale(z1)
        tz1 = self.translate(z1)

        z2 = z2 * torch.exp(sz1) + tz1
```

```
        if self.mask:
            z1, z2 = z2, z1

        x = torch.cat((z1, z2), dim=1)
        log_jacob_det = sz1.sum(-1)

        return x, log_jacob_det


class TVAESynthesizerPlanarFlow(Module):

    def __init__(
            self,
            hidden_dims=(128, 128),
            embed_dim=32,
            num_flows=5,
            l2scale=1e-5,
            loss_factor=2,
            batch_size=256,
            epochs=300,
            cuda=True
    ):
        super(TVAESynthesizerPlanarFlow, self).__init__()

        self.hidden_dims = hidden_dims
        self.embed_dim = embed_dim
        self.num_flows = num_flows
        self.l2scale = l2scale
        self.loss_factor = loss_factor
        self.batch_size = batch_size
        self.epochs = epochs

        if not cuda or not torch.cuda.is_available():
            device = 'cpu'
        elif isinstance(cuda, str):
            device = cuda
        else:
            device = 'cuda'

        self._device = torch.device(device)

        self.log_jacob_det = 0.

        self.amor_u = Linear(list(self.hidden_dims)[-1], self.num_flows *
self.embed_dim)
        self.amor_w = Linear(list(self.hidden_dims)[-1], self.num_flows *
self.embed_dim)
        self.amor_b = Linear(list(self.hidden_dims)[-1], self.num_flows)

        for k in range(self.num_flows):
            flow_k = Planar()
            self.add_module('flow_' + str(k), flow_k)

    def fit(self, train_data, discrete_columns=tuple()):
        self.transformer = DataTransformer()
        self.transformer.fit(train_data, discrete_columns)
```

```
        train_data = self.transformer.transform(train_data)
        dataset =
TensorDataset(torch.from_numpy(train_data.astype('float32')).to(self._device)
)
        loader = DataLoader(dataset, batch_size=self.batch_size,
shuffle=True, drop_last=False)

        input_dim = self.transformer.output_dimensions
        encoder = EncoderFlows(input_dim, self.hidden_dims,
self.embed_dim).to(self._device)
        self.decoder = Decoder(self.embed_dim, self.hidden_dims,
input_dim).to(self._device)
        optimizer = Adam(list(encoder.parameters()) +
list(self.decoder.parameters()), weight_decay=self.l2scale)

        for i in range(self.epochs):
            for id_, data in enumerate(loader):
                optimizer.zero_grad()
                real = data[0].to(self._device)
                mu, std, var, out = encoder(real)
                u = self.amor_u(out).view(real.size(0), self.num_flows,
self.embed_dim, 1)
                w = self.amor_w(out).view(real.size(0), self.num_flows, 1,
self.embed_dim)
                b = self.amor_b(out).view(real.size(0), self.num_flows, 1, 1)
                eps = torch.randn_like(std)
                z_0 = eps * std + mu
                z = [z_0]
                self.log_jacob_det = torch.zeros([real.shape[0]])
                for k in range(self.num_flows):
                    flow_k = getattr(self, 'flow_' + str(k))
                    z_k, log_jacob_det = flow_k(z[k], u[:, k, :, :], w[:, k,
:, :], b[:, k, :, :])
                    z.append(z_k)
                    self.log_jacob_det += log_jacob_det
                rec, sigmas = self.decoder(z[-1])
                loss_1, loss_2 = loss_function_flows(rec, real, sigmas, mu,
var, z[0], z[-1], self.log_jacob_det, self.embed_dim,
self.transformer.output_info_list, self.loss_factor)
                loss = loss_1 + loss_2
                loss.backward()
                optimizer.step()
                self.decoder.sigma.data.clamp_(0.01, 1.0)

    def sample(self, samples):
        self.decoder.eval()

        steps = samples // self.batch_size + 1
        data = []
        for _ in range(steps):
            mean = torch.zeros(self.batch_size, self.embed_dim)
            std = mean + 1
            noise = torch.normal(mean=mean, std=std).to(self._device)
            fake, sigmas = self.decoder(noise)
            fake = torch.tanh(fake)
            data.append(fake.detach().cpu().numpy())
```

```
        data = np.concatenate(data, axis=0)
        data = data[:samples]
        return self.transformer.inverse_transform(data,
sigmas.detach().cpu().numpy())

    def set_device(self, device):
        self._device = device
        self.decoder.to(self._device)

    def save(self, path):
        device_backup = self._device
        self.set_device(torch.device("cpu"))
        torch.save(self, path)
        self.set_device(device_backup)

    def load(cls, path):
        device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
        model = torch.load(path)
        model.set_device(device)
        return model


class TVAESynthesizerSylvesterFlow(Module):

    def __init__(
            self,
            hidden_dims=(128, 128),
            embed_dim=32,
            num_flows=5,
            num_ortho_vec=4,
            invert=False,
            l2scale=1e-5,
            loss_factor=2,
            batch_size=256,
            epochs=300,
            cuda=True
    ):
        super(TVAESynthesizerSylvesterFlow, self).__init__()

        self.hidden_dims = hidden_dims
        self.embed_dim = embed_dim
        self.num_flows = num_flows
        self.M = num_ortho_vec
        self.invert = invert
        self.l2scale = l2scale
        self.loss_factor = loss_factor
        self.batch_size = batch_size
        self.epochs = epochs

        if not cuda or not torch.cuda.is_available():
            device = 'cpu'
        elif isinstance(cuda, str):
            device = cuda
        else:
            device = 'cuda'
```

```
        self._device = torch.device(device)

        self.epsilon = 1e-6
        self.steps = 100
        self.identity = torch.eye(self.M, self.M).unsqueeze(0)

        self.upper_triang = torch.triu(torch.ones(self.M, self.M))
        self.upper_triang = self.upper_triang.unsqueeze(0).unsqueeze(3)

        self.amor_Q = Linear(list(self.hidden_dims)[-1], self.num_flows *
self.embed_dim * self.M)
        self.amor_b = Linear(list(self.hidden_dims)[-1], self.num_flows *
self.M)
        self.amor_R = Linear(list(self.hidden_dims)[-1], self.num_flows *
self.M * self.M)
        self.amor_R_tilde = Linear(list(self.hidden_dims)[-1], self.num_flows
* self.M * self.M)

        if self.invert:
            self.amor_diag1 = Sequential(Linear(list(self.hidden_dims)[-1],
self.num_flows * self.M), Tanh())
            self.amor_diag2 = Sequential(Linear(list(self.hidden_dims)[-1],
self.num_flows * self.M), Tanh())

        for k in range(self.num_flows):
            flow_k = Sylvester()
            self.add_module('flow_' + str(k), flow_k)

    def maintain_Q_orthogonal(self, Q):
        Q = Q.view(-1, self.embed_dim, self.M)
        norm = torch.norm(Q, p=2, dim=[1, 2], keepdim=True)
        Q_k = torch.div(Q, norm)

        for k in range(self.steps):
            QkTQk = torch.bmm(Q_k.transpose(2, 1), Q_k)
            Q_k = torch.bmm(Q_k, (self.identity + 0.5 * (self.identity -
QkTQk)))

            QkTQk = torch.bmm(Q_k.transpose(2, 1), Q_k)
            norm = torch.sqrt(torch.sum(torch.norm((QkTQk - self.identity),
p='fro', dim=2) ** 2, dim=1))
            max_norm = torch.max(norm)
            if max_norm <= self.epsilon:
                break
        if max_norm > self.epsilon:
            print('orthogonalisation not converged')

        Q_k = Q_k.view(-1, self.num_flows, self.embed_dim, self.M)

        return Q_k.transpose(1, 0)

    def fit(self, train_data, discrete_columns=tuple()):
        self.transformer = DataTransformer()
        self.transformer.fit(train_data, discrete_columns)
        train_data = self.transformer.transform(train_data)
        dataset =
TensorDataset(torch.from_numpy(train_data.astype('float32')).to(self._device)
```

```
)
        loader = DataLoader(dataset, batch_size=self.batch_size,
shuffle=True, drop_last=False)

        input_dim = self.transformer.output_dimensions
        encoder = EncoderFlows(input_dim, self.hidden_dims,
self.embed_dim).to(self._device)
        self.decoder = Decoder(self.embed_dim, self.hidden_dims,
input_dim).to(self._device)
        optimizer = Adam(list(encoder.parameters()) +
list(self.decoder.parameters()), weight_decay=self.l2scale)

        for i in range(self.epochs):
            for id_, data in enumerate(loader):
                optimizer.zero_grad()
                real = data[0].to(self._device)
                mu, std, var, out = encoder(real)
                Q = self.amor_Q(out)
                b = self.amor_b(out).view(real.size(0), self.M,
self.num_flows)
                R = (self.amor_R(out).view(real.size(0), self.M, self.M,
self.num_flows)) * self.upper_triang
                R_tilde = (self.amor_R_tilde(out).view(real.size(0), self.M,
self.M, self.num_flows)) * self.upper_triang
                if self.invert:
                    diag1 = self.amor_diag1(out)
                    diag2 = self.amor_diag2(out)
                    diag1 = diag1.view(real.size(0), self.M, self.num_flows)
                    diag2 = diag2.view(real.size(0), self.M, self.num_flows)
                    R[:, range(self.M), range(self.M), :] = diag1
                    R_tilde[:, range(self.M), range(self.M), :] = diag2
                eps = torch.randn_like(std)
                z_0 = eps * std + mu
                z = [z_0]
                self.log_jacob_det = torch.zeros([real.shape[0]])
                Q_orthogonal = self.maintain_Q_orthogonal(Q)
                for k in range(self.num_flows):
                    flow_k = getattr(self, 'flow_' + str(k))
                    z_k, log_jacob_det = flow_k(z[k], Q_orthogonal[k, :, :,
:], R[:, :, :, k], R_tilde[:, :, :, k], b[:, :, k])
                    z.append(z_k)
                    self.log_jacob_det += log_jacob_det
                rec, sigmas = self.decoder(z[-1])
                loss_1, loss_2 = loss_function_flows(rec, real, sigmas, mu,
var, z[0], z[-1], self.log_jacob_det, self.embed_dim,
self.transformer.output_info_list, self.loss_factor)
                loss = loss_1 + loss_2
                loss.backward()
                optimizer.step()
                self.decoder.sigma.data.clamp_(0.01, 1.0)

    def sample(self, samples):
        self.decoder.eval()

        steps = samples // self.batch_size + 1
        data = []
        for _ in range(steps):
```

```
            mean = torch.zeros(self.batch_size, self.embed_dim)
            std = mean + 1
            noise = torch.normal(mean=mean, std=std).to(self._device)
            fake, sigmas = self.decoder(noise)
            fake = torch.tanh(fake)
            data.append(fake.detach().cpu().numpy())

        data = np.concatenate(data, axis=0)
        data = data[:samples]
        return self.transformer.inverse_transform(data,
sigmas.detach().cpu().numpy())

    def set_device(self, device):
        self._device = device
        self.decoder.to(self._device)

    def save(self, path):
        device_backup = self._device
        self.set_device(torch.device("cpu"))
        torch.save(self, path)
        self.set_device(device_backup)

    def load(cls, path):
        device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
        model = torch.load(path)
        model.set_device(device)
        return model


class TVAESynthesizerNICEFlow(Module):

    def __init__(
            self,
            hidden_dims=(128, 128),
            embed_dim=32,
            num_flows=5,
            m_hidden_dim=80,
            m_num_layers=4,
            l2scale=1e-5,
            loss_factor=2,
            batch_size=256,
            epochs=300,
            cuda=True
    ):
        super(TVAESynthesizerNICEFlow, self).__init__()

        self.hidden_dims = hidden_dims
        self.embed_dim = embed_dim
        self.num_flows = num_flows
        self.m_hidden_dim = m_hidden_dim
        self.m_num_layers = m_num_layers
        self.l2scale = l2scale
        self.loss_factor = loss_factor
        self.batch_size = batch_size
        self.epochs = epochs
```

```
        if not cuda or not torch.cuda.is_available():
            device = 'cpu'
        elif isinstance(cuda, str):
            device = cuda
        else:
            device = 'cuda'

        self._device = torch.device(device)

        self.log_jacob_det = 0.

        for k in range(self.num_flows):
            flow_k = NICECoupling(self.embed_dim, self.m_hidden_dim,
self.m_num_layers)
            self.add_module('flow_' + str(k), flow_k)

        self.scale = NICEScaling(self.embed_dim)

    def fit(self, train_data, discrete_columns=tuple()):
        self.transformer = DataTransformer()
        self.transformer.fit(train_data, discrete_columns)
        train_data = self.transformer.transform(train_data)
        dataset =
TensorDataset(torch.from_numpy(train_data.astype('float32')).to(self._device)
)
        loader = DataLoader(dataset, batch_size=self.batch_size,
shuffle=True, drop_last=False)

        input_dim = self.transformer.output_dimensions
        encoder = EncoderFlows(input_dim, self.hidden_dims,
self.embed_dim).to(self._device)
        self.decoder = Decoder(self.embed_dim, self.hidden_dims,
input_dim).to(self._device)
        optimizer = Adam(list(encoder.parameters()) +
list(self.decoder.parameters()), weight_decay=self.l2scale)

        for i in range(self.epochs):
            for id_, data in enumerate(loader):
                optimizer.zero_grad()
                real = data[0].to(self._device)
                mu, std, var, out = encoder(real)
                eps = torch.randn_like(std)
                z_0 = eps * std + mu
                z = [z_0]
                self.log_jacob_det = torch.zeros([real.shape[0]])
                for k in range(self.num_flows):
                    flow_k = getattr(self, 'flow_' + str(k))
                    z_k = flow_k(z[k])
                    z.append(z_k)
                z_k, log_jacob_det = self.scale(z_k)
                z.append(z_k)
                self.log_jacob_det += log_jacob_det
                rec, sigmas = self.decoder(z[-1])
                loss_1, loss_2 = loss_function_flows(rec, real, sigmas, mu,
var, z[0], z[-1], self.log_jacob_det, self.embed_dim,
self.transformer.output_info_list, self.loss_factor)
                loss = loss_1 + loss_2
```

```
                loss.backward()
                optimizer.step()
                self.decoder.sigma.data.clamp_(0.01, 1.0)

    def sample(self, samples):
        self.decoder.eval()

        steps = samples // self.batch_size + 1
        data = []
        for _ in range(steps):
            mean = torch.zeros(self.batch_size, self.embed_dim)
            std = mean + 1
            noise = torch.normal(mean=mean, std=std).to(self._device)
            fake, sigmas = self.decoder(noise)
            fake = torch.tanh(fake)
            data.append(fake.detach().cpu().numpy())

        data = np.concatenate(data, axis=0)
        data = data[:samples]
        return self.transformer.inverse_transform(data,
sigmas.detach().cpu().numpy())

    def set_device(self, device):
        self._device = device
        self.decoder.to(self._device)

    def save(self, path):
        device_backup = self._device
        self.set_device(torch.device("cpu"))
        torch.save(self, path)
        self.set_device(device_backup)

    def load(cls, path):
        device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
        model = torch.load(path)
        model.set_device(device)
        return model


class TVAESynthesizerRealNVPFlow(Module):

    def __init__(
            self,
            hidden_dims=(128, 128),
            embed_dim=32,
            num_flows=5,
            st_hidden_dim=80,
            l2scale=1e-5,
            loss_factor=2,
            batch_size=256,
            epochs=300,
            cuda=True
    ):
        super(TVAESynthesizerRealNVPFlow, self).__init__()
```

```
        self.hidden_dims = hidden_dims
        self.embed_dim = embed_dim
        self.num_flows = num_flows
        self.st_hidden_dim = st_hidden_dim
        self.l2scale = l2scale
        self.loss_factor = loss_factor
        self.batch_size = batch_size
        self.epochs = epochs

        if not cuda or not torch.cuda.is_available():
            device = 'cpu'
        elif isinstance(cuda, str):
            device = cuda
        else:
            device = 'cuda'

        self._device = torch.device(device)

        self.log_jacob_det = 0.

        for k in range(self.num_flows):
            mask = 0 if k % 2 == 0 else 1
            flow_k = RealNVP(self.embed_dim, self.st_hidden_dim, mask)
            self.add_module('flow_' + str(k), flow_k)

    def fit(self, train_data, discrete_columns=tuple()):
        self.transformer = DataTransformer()
        self.transformer.fit(train_data, discrete_columns)
        train_data = self.transformer.transform(train_data)
        dataset =
TensorDataset(torch.from_numpy(train_data.astype('float32')).to(self._device)
)
        loader = DataLoader(dataset, batch_size=self.batch_size,
shuffle=True, drop_last=False)

        input_dim = self.transformer.output_dimensions
        encoder = EncoderFlows(input_dim, self.hidden_dims,
self.embed_dim).to(self._device)
        self.decoder = Decoder(self.embed_dim, self.hidden_dims,
input_dim).to(self._device)
        optimizer = Adam(list(encoder.parameters()) +
list(self.decoder.parameters()), weight_decay=self.l2scale)

        for i in range(self.epochs):
            for id_, data in enumerate(loader):
                optimizer.zero_grad()
                real = data[0].to(self._device)
                mu, std, var, out = encoder(real)
                eps = torch.randn_like(std)
                z_0 = eps * std + mu
                z = [z_0]
                self.log_jacob_det = torch.zeros([real.shape[0]])
                for k in range(self.num_flows):
                    flow_k = getattr(self, 'flow_' + str(k))
                    z_k, log_jacob_det = flow_k(z[k])
                    z.append(z_k)
                    self.log_jacob_det += log_jacob_det
```

```
                rec, sigmas = self.decoder(z[-1])
                loss_1, loss_2 = loss_function_flows(rec, real, sigmas, mu,
var, z[0], z[-1], self.log_jacob_det, self.embed_dim,
self.transformer.output_info_list, self.loss_factor)
                loss = loss_1 + loss_2
                loss.backward()
                optimizer.step()
                self.decoder.sigma.data.clamp_(0.01, 1.0)

    def sample(self, samples):
        self.decoder.eval()

        steps = samples // self.batch_size + 1
        data = []
        for _ in range(steps):
            mean = torch.zeros(self.batch_size, self.embed_dim)
            std = mean + 1
            noise = torch.normal(mean=mean, std=std).to(self._device)
            fake, sigmas = self.decoder(noise)
            fake = torch.tanh(fake)
            data.append(fake.detach().cpu().numpy())

        data = np.concatenate(data, axis=0)
        data = data[:samples]
        return self.transformer.inverse_transform(data,
sigmas.detach().cpu().numpy())

    def set_device(self, device):
        self._device = device
        self.decoder.to(self._device)

    def save(self, path):
        device_backup = self._device
        self.set_device(torch.device("cpu"))
        torch.save(self, path)
        self.set_device(device_backup)

    def load(cls, path):
        device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
        model = torch.load(path)
        model.set_device(device)
        return model
```

In the Synthetic Data Vault (SDV, 2021), CTGAN and TVAE models are constructed by using wrappers around the synthesisers for the models, so these were also created for TVAE with normalising flow models, using the code below:

```
import numpy as np
```

```
from ctgan import TVAESynthesizerPlanarFlow, TVAESynthesizerSylvesterFlow,
TVAESynthesizerNICEFlow, TVAESynthesizerRealNVPFlow,
from sdv.tabular.base import BaseTabularModel


class TVAEPlanarFlow(CTGANModel):

    _MODEL_CLASS = TVAESynthesizerPlanarFlow

    def __init__(self, field_names=None, field_types=None,
field_transformers=None, anonymize_fields=None, primary_key=None,
constraints=None, table_metadata=None, hidden_dims=(128, 128), embed_dim=32,
num_flows=5, l2scale=1e-5, loss_factor=2, batch_size=256, epochs=300,
cuda=True):
        super().__init__(
            field_names=field_names,
            field_types=field_types,
            field_transformers=field_transformers,
            anonymize_fields=anonymize_fields,
            primary_key=primary_key,
            constraints=constraints,
            table_metadata=table_metadata
        )

        self._model_kwargs = {
            'hidden_dims': hidden_dims,
            'embed_dim': embed_dim,
            'num_flows': num_flows,
            'l2scale': l2scale,
            'loss_factor': loss_factor,
            'batch_size': batch_size,
            'epochs': epochs,
            'cuda': cuda
        }


class TVAESylvesterFlow(CTGANModel):

    _MODEL_CLASS = TVAESynthesizerSylvesterFlow

    def __init__(self, field_names=None, field_types=None,
field_transformers=None, anonymize_fields=None, primary_key=None,
constraints=None, table_metadata=None, hidden_dims=(128, 128), embed_dim=32,
num_flows=5, num_ortho_vec=4, invert=False, l2scale=1e-5, loss_factor=2,
batch_size=256, epochs=300, cuda=True):
        super().__init__(
            field_names=field_names,
            field_types=field_types,
            field_transformers=field_transformers,
            anonymize_fields=anonymize_fields,
            primary_key=primary_key,
            constraints=constraints,
            table_metadata=table_metadata
        )

        self._model_kwargs = {
```

```
            'hidden_dims': hidden_dims,
            'embed_dim': embed_dim,
            'num_flows': num_flows,
            'num_ortho_vec': num_ortho_vec,
            'invert': invert,
            'l2scale': l2scale,
            'loss_factor': loss_factor,
            'batch_size': batch_size,
            'epochs': epochs,
            'cuda': cuda
        }


class TVAENICEFlow(CTGANModel):

    _MODEL_CLASS = TVAESynthesizerNICEFlow

    def __init__(self, field_names=None, field_types=None,
field_transformers=None, anonymize_fields=None, primary_key=None,
constraints=None, table_metadata=None, hidden_dims=(128, 128), embed_dim=32,
num_flows=5, m_hidden_dim=80, m_num_layers=4, l2scale=1e-5, loss_factor=2,
batch_size=256, epochs=300, cuda=True):
        super().__init__(
            field_names=field_names,
            field_types=field_types,
            field_transformers=field_transformers,
            anonymize_fields=anonymize_fields,
            primary_key=primary_key,
            constraints=constraints,
            table_metadata=table_metadata
        )

        self._model_kwargs = {
            'hidden_dims': hidden_dims,
            'embed_dim': embed_dim,
            'num_flows': num_flows,
            'm_hidden_dim': m_hidden_dim,
            'm_num_layers': m_num_layers,
            'l2scale': l2scale,
            'loss_factor': loss_factor,
            'batch_size': batch_size,
            'epochs': epochs,
            'cuda': cuda
        }


class TVAERealNVPFlow(CTGANModel):

    _MODEL_CLASS = TVAESynthesizerRealNVPFlow

    def __init__(self, field_names=None, field_types=None,
field_transformers=None, anonymize_fields=None, primary_key=None,
constraints=None, table_metadata=None, hidden_dims=(128, 128), embed_dim=32,
num_flows=5, st_hidden_dim=80, l2scale=1e-5, loss_factor=2, batch_size=256,
epochs=300, cuda=True):
        super().__init__(
            field_names=field_names,
```

```
            field_types=field_types,
            field_transformers=field_transformers,
            anonymize_fields=anonymize_fields,
            primary_key=primary_key,
            constraints=constraints,
            table_metadata=table_metadata
        )

        self._model_kwargs = {
            'hidden_dims': hidden_dims,
            'embed_dim': embed_dim,
            'num_flows': num_flows,
            'st_hidden_dim': st_hidden_dim,
            'l2scale': l2scale,
            'loss_factor': loss_factor,
            'batch_size': batch_size,
            'epochs': epochs,
            'cuda': cuda
        }
```

TVAE with normalising flow models were trained and sampled from using the code below, similar to CTGAN and vanilla TVAE in the Synthetic Data Vault (SDV, 2021) (example for TVAE with planar flow):

```
import pandas as pd

from sdv.tabular import TVAEPlanarFlow, TVAESylvesterFlow, TVAENICEFlow,
TVAERealNVPFlow
from sdv.metrics.tabular import CSTest, KSTest, LogisticDetection,
SVCDetection, BinaryDecisionTreeClassifier, BinaryAdaBoostClassifier,
BinaryLogisticRegression, BinaryMLPClassifier

real_data = pd.read_csv('portsmouth_balanced.csv')

model = TVAEPlanarFlow(hidden_dims=(128, 128), embed_dim=32, num_flows=5,
batch_size=256, epochs=300)
model.fit(real_data)

synthetic_data = model.sample(4318)

cs = CSTest.compute(real_data, synthetic_data)
ks = KSTest.compute(real_data, synthetic_data)
logistic = LogisticDetection.compute(real_data, synthetic_data)
svc = SVCDetection.compute(real_data, synthetic_data)
dt = BinaryDecisionTreeClassifier.compute(real_data, synthetic_data,
target='Covid-19 Positive')
ada = BinaryAdaBoostClassifier.compute(real_data, synthetic_data,
target='Covid-19 Positive')
lr = BinaryLogisticRegression.compute(real_data, synthetic_data,
target='Covid-19 Positive')
```

```
mlp = BinaryMLPClassifier.compute(real_data, synthetic_data, target='Covid-19
Positive')
```

Part 2: Deep learning for prediction on tabular data

Differentiable tree-based models

NODE

The NODE implementation in the Pytorch Tabular library (PyTorch Tabular, 2021) was used.

The implementation was extended with weight decay (Pytorch-optimizer, 2021b), dropout

(PyTorch, 2021c) and batch normalisation (PyTorch, 2021b), using the code below.

The following code was used to create NODE layers:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from .odst import ODST


class DenseODSTBlock(nn.Sequential):
    def __init__(
        self,
        input_dim,
        num_trees,
        num_layers,
        tree_output_dim=3,
        max_features=None,
        input_dropout=0.1,
        flatten_output=False,
        Module=ODST,
        **kwargs
    ):
        layers = []
        for i in range(num_layers):
            oddt = Module(
                input_dim,
                num_trees,
                tree_output_dim=tree_output_dim,
                flatten_output=True,
                **kwargs
            )
            input_dim = min(input_dim + num_trees * tree_output_dim,
```

```
max_features or float("inf"))
            layers.append(oddt)

        super().__init__(*layers)
        self.num_layers, self.layer_dim, self.tree_dim = (num_layers,
num_trees, tree_output_dim)
        self.max_features, self.flatten_output = max_features, flatten_output
        self.input_dropout = input_dropout

    def forward(self, x):
        initial_features = x.shape[-1]
        for layer in self:
            layer_inp = x
            if self.max_features is not None:
                tail_features = (
                    min(self.max_features, layer_inp.shape[-1]) -
initial_features
                )
                if tail_features != 0:
                    layer_inp = torch.cat(
                        [layer_inp[..., :initial_features],
                         layer_inp[..., -tail_features:]],
                        dim=-1,
                    )
            if self.training and self.input_dropout:
                layer_inp = F.dropout(layer_inp, self.input_dropout)
            h = layer(layer_inp)
            h = F.batch_norm(h, running_mean=None, running_var=None,
training=True)
            x = torch.cat([x, h], dim=-1)

        outputs = x[..., initial_features:]
        if not self.flatten_output:
            outputs = outputs.view(*outputs.shape[:-1], self.num_layers *
self.layer_dim, self.tree_dim)

        return outputs
```

The following code was used to train and test the NODE model:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from torch_optimizer import QHAdam

from pytorch_tabular import TabularModel
from pytorch_tabular.models import NodeConfig
from pytorch_tabular.config import DataConfig, OptimizerConfig,
TrainerConfig, ExperimentConfig
```

```
data = pd.read_csv('portsmouth_balanced.csv')
data2 = pd.read_csv('bedford.csv')
data3 = pd.read_csv('oxford.csv')
data4 = pd.read_csv('birmingham.csv')

cont_cols = ['Age', 'Vital_Sign Heart Rate', 'Vital_Sign Respiratory Rate',
'Vital_Sign Systolic Blood Pressure', 'Vital_Sign Diastolic Blood Pressure',
'Vital_Sign Oxygen Saturation', 'Vital_Sign Temperature Tympanic',
'Blood_Test HAEMOGLOBIN', 'Blood_Test HAEMATOCRIT', 'Blood_Test MEAN CELL
VOL.', 'Blood_Test WHITE CELLS', 'Blood_Test NEUTROPHILS', 'Blood_Test
LYMPHOCYTES', 'Blood_Test MONOCYTES', 'Blood_Test EOSINOPHILS', 'Blood_Test
BASOPHILS', 'Blood_Test PLATELETS', 'Blood_Test SODIUM', 'Blood_Test
POTASSIUM', 'Blood_Test UREA', 'Blood_Test CREATININE', 'Blood_Test eGFR',
'Blood_Test ALT', 'Blood_Test ALK.PHOSPHATASE', 'Blood_Test BILIRUBIN',
'Blood_Test ALBUMIN', 'Blood_Test CRP']

cat_cols = ['Gender', 'Ethnicity']

train, test = train_test_split(data, random_state=42)
train, val = train_test_split(train, random_state=42)

data_config = DataConfig(
    target=['Covid-19 Positive'],
    continuous_cols=cont_cols,
    categorical_cols=cat_cols,
    continuous_feature_transform="quantile_normal")

trainer_config = TrainerConfig(
    batch_size=256,
    max_epochs=30,
    auto_lr_find=False)

optimizer_config = OptimizerConfig()

model_config = NodeConfig(
    task="classification",
    learning_rate=0.001,
    num_layers=2,
    num_trees=512,
    depth=6,
    choice_function="entmax15",
    bin_function="entmoid15",
    input_dropout=0.1,
    embed_categorical=False)

node = TabularModel(
    data_config=data_config,
    model_config=model_config,
    optimizer_config=optimizer_config,
    trainer_config=trainer_config)

node.fit(train=train,
         validation=val,
         optimizer=QHAdam,
         optimizer_params={"nus": (0.7, 1.0), "betas": (0.95, 0.998),
"weight_decay": 0.0001})
```

```
def calculate_auc(y_true, y_pred):
    if isinstance(y_true, pd.DataFrame) or isinstance(y_true, pd.Series):
        y_true = y_true.values
    if isinstance(y_pred, pd.DataFrame) or isinstance(y_pred, pd.Series):
        y_pred = y_pred.values
    if y_true.ndim>1:
        y_true=y_true.ravel()
    if y_pred.ndim>1:
        y_pred=y_pred.ravel()
    auc = roc_auc_score(y_true, y_pred)

    return auc

pred = node.predict(test)
pred2 = node.predict(data2)
pred3 = node.predict(data3)
pred4 = node.predict(data4)

Portsmouth_test_auc = calculate_auc(test['Covid-19 Positive'],
pred["prediction"])
Bedford_valid_auc = calculate_auc(data2['Covid-19 Positive'],
pred2["prediction"])
Oxford_valid_auc = calculate_auc(data3['Covid-19 Positive'],
pred3["prediction"])
Birmingham_valid_auc = calculate_auc(data4['Covid-19 Positive'],
pred4["prediction"])
```

Quantum Forest

The Quantum Forest implementation in Chen (2020) was used. The implementation was extended with weight decay (Pytorch-optimizer, 2021b), dropout (PyTorch, 2021c), batch normalisation (PyTorch, 2021b) and Lookahead (Pytorch-optimizer, 2021a), using the code below.

The following code was used to create Quantum Forest layers and model:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from .DifferentiableTree import *

class DecisionBlock(nn.Sequential):
    def __init__(self, input_dim, config, hasBN=False, flatten_output=True,
feat_info=None, **kwargs):
        super(DecisionBlock, self).__init__()
```

```
        self.config = config
        layers = []
        Module = config.tree_module
        num_trees = config.nTree
        response_dim=config.response_dim
        for i in range(config.num_layers):
            if hasBN:
                layer = nn.BatchNorm1d(input_dim)
                layers.append(layer)
            layer = Module(input_dim, num_trees, config,
flatten_output=True,feat_info=feat_info, **kwargs)
            input_dim = min(input_dim + num_trees * response_dim,
config.max_features or float('inf'))
            layers.append(layer)

        super().__init__(*layers)
        self.num_layers, self.layer_dim, self.response_dim =
config.num_layers, num_trees, response_dim
        self.max_features, self.flatten_output = config.max_features,
flatten_output
        self.input_dropout = config.input_dropout

    def get_variables(self, var_dic):
        for layer in self:
            if isinstance(layer, DeTree):
                if "attention" in var_dic:
                    var_dic["attention"].append(layer.feat_attention)
                if "gate_values" in var_dic:
                    var_dic["gate_values"].append(layer.gate_values)

        return var_dic

    def freeze_some_params(self, freeze_info):
        requires_grad = freeze_info["requires_grad"]
        for layer in self:
            if isinstance(layer, DeTree):
                layer.feat_attention.requires_grad = requires_grad
                layer.feature_thresholds.requires_grad = requires_grad
                layer.log_temperatures.requires_grad = requires_grad

    def forward_dense(self, x):
        initial_features = x.shape[-1]
        for layer in self:
            layer_inp = x
            if self.max_features is not None:
                tail_features = min(self.max_features, layer_inp.shape[-1]) -
initial_features
                if tail_features != 0:
                    layer_inp = torch.cat([layer_inp[..., :initial_features],
layer_inp[..., -tail_features:]], dim=-1)
            if self.training and self.input_dropout:
                layer_inp = F.dropout(layer_inp, self.input_dropout)
            h = layer(layer_inp)
            x = torch.cat([x, h], dim=-1)
        outputs = x[..., initial_features:]
        if not self.flatten_output:
            outputs = outputs.view(*outputs.shape[:-1], self.num_layers *
```

```
self.layer_dim, self.response_dim)
        if self.config.max_out:
            outputs = torch.max(outputs, -1).values
        else:
            outputs = outputs[..., 0]

        return outputs

    def forward(self, x):
        if self.training and self.input_dropout:
            x = F.dropout(x, self.input_dropout)
        for layer in self:
            layer_inp = x
            x = layer(layer_inp)
        outputs = x
        if self.config.leaf_output == "leaf_distri":
            if not self.flatten_output:
                outputs = outputs.view(*outputs.shape[:-1], self.num_layers *
self.layer_dim, self.response_dim)
            if self.config.max_out:
                if self.config.problem()=="classification":
                    pass
                elif self.config.problem()=="regression_N":
                    outputs = outputs.mean(dim=-2)
                else:
                    outputs = torch.max(outputs, -1).values
            else:
                outputs = outputs[..., 0]

        return outputs

    def AfterEpoch(self, epoch=0):
        for layer in self:
            if isinstance(layer,DeTree):
                layer.AfterEpoch(epoch)


import numpy as np
import torch
import torch.nn as nn

from .DecisionBlock import *

class QF_Net(nn.Module):
    def __init__(self, in_features, config, feat_info=None, visual=None):
        super(QF_Net, self).__init__()
        config.feat_info = feat_info
        self.layers = nn.ModuleList()
        self.nTree = config.nTree
        self.config = config
        self.reg_L1 = 0.
        self.L_gate = 0.

        self.embeddings = None
        nFeat = in_features

        for i in range(config.nLayers):
```

```
            if i > 0:
                nFeat = config.nTree
                feat_info = None
            hasBN = config.data_normal == "BN" and i > 0
            self.layers.append(DecisionBlock(nFeat, config, hasBN=hasBN,
flatten_output=False, feat_info=feat_info))

        self.pooling = None
        self.visual = visual

        dump_model_params(self)

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)

        self.Regularization()

        assert len(x.shape)==3
        x = x.mean(dim=-2)

        return x

    def Regularization(self):
        dict_val = self.get_variables({"attention":[],"gate_values":[]})
        reg, l1, l2 = 0, 0, 0
        for att in dict_val["attention"]:
            a = torch.sum(torch.abs(att))/att.numel()
            l1 = l1 + a
        self.reg_L1 = l1
        reg = self.reg_L1*self.config.reg_L1
        for gate_values in dict_val["gate_values"]:
            if self.config.support_vector=="0":
                a = torch.sum(torch.pow(gate_values, 2))/gate_values.numel()
            else:
                a = torch.sum(gate_values)/gate_values.numel()
            l2 = l2 + a
        self.L_gate = l2

        return reg

    def get_variables(self, var_dic):
        for layer in self.layers:
            var_dic = layer.get_variables(var_dic)

        return var_dic

    def freeze_some_params(self, freeze_info):
        for layer in self.layers:
            layer.freeze_some_params(freeze_info)

    def AfterEpoch(self, isBetter=False, epoch=0, accu=0):
        attentions=[]

        dict_val = self.get_variables({"attention":[]})
        for att in dict_val["attention"]:
            attentions.append(att.data.detach().cpu().numpy())
```

```
        attention = np.concatenate(attentions)
        self.nAtt = attention.size
        self.nzAtt = self.nAtt - np.count_nonzero(attention)

        nFeat, nCol = attention.shape[0], attention.shape[1]
        nCol = min(nFeat*3, attention.shape[1])
        cols = random.choices(population = list(range(attention.shape[1])), k
= nCol)
        type="" if self.config.feat_info is None else "sparse"

        if self.visual is not None:
            path = f"{self.config.data_set}_{type}_{epoch}"
            params = {'title':f"{epoch} - {accu:.4f}",'cmap':sns.cm.rocket}
            self.visual.image(path,attention[:,cols], params=params)
        else:
            pass

        return
```

The following code was used to train and test the Quantum Forest model:

```
import os
import numpy as np
import pandas as pd
import pickle
from sklearn.model_selection import train_test_split
from category_encoders import LeaveOneOutEncoder

DATASETS = {'COVID': fetch_COVID}

def fetch_COVID(path, valid_size=0.2):
    pkl_path = f'{path}/covid_.pickle'
    if os.path.isfile(pkl_path):
        with open(pkl_path, "rb") as fp:
            data_dict = pickle.load(fp)
    else:
        csv_path = os.path.join(path, 'portsmouth_balanced.csv')
        csv_path_2 = os.path.join(path, 'bedford.csv')
        csv_path_3 = os.path.join(path, 'oxford.csv')
        csv_path_4 = os.path.join(path, 'birmingham.csv')

        data = pd.read_csv(csv_path, index_col=0)
        data2 = pd.read_csv(csv_path_2, index_col=0)
        data3 = pd.read_csv(csv_path_3, index_col=0)
        data4 = pd.read_csv(csv_path_4, index_col=0)

        X, y = data.drop(columns=['Covid-19 Positive']), data['Covid-19
Positive']
        X_2, y_2 = data2.drop(columns=['Covid-19 Positive']), data2['Covid-19
Positive']
        X_3, y_3 = data3.drop(columns=['Covid-19 Positive']), data3['Covid-19
```

```
Positive']
        X_4, y_4 = data4.drop(columns=['Covid-19 Positive']), data4['Covid-19
Positive']
        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=valid_size, random_state=42)

        y_train = (y_train.values.reshape(-1) == 1).astype('int64')
        y_test = (y_test.values.reshape(-1) == 1).astype('int64')
        y_2 = (y_2.values.reshape(-1) == 1).astype('int64')
        y_3 = (y_3.values.reshape(-1) == 1).astype('int64')
        y_4 = (y_4.values.reshape(-1) == 1).astype('int64')

        cat_features = ['Gender', 'Ethnicity']

        X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=valid_size, random_state=42)

        num_features = X_train.shape[1]
        num_classes = len(set(y_train))

        cat_encoder = LeaveOneOutEncoder()
        cat_encoder.fit(X_train[cat_features], y_train)
        X_train[cat_features] = cat_encoder.transform(X_train[cat_features])
        X_val[cat_features] = cat_encoder.transform(X_val[cat_features])
        X_test[cat_features] = cat_encoder.transform(X_test[cat_features])
        X_2[cat_features] = cat_encoder.transform(X_2[cat_features])
        X_3[cat_features] = cat_encoder.transform(X_3[cat_features])
        X_4[cat_features] = cat_encoder.transform(X_4[cat_features])

        data_dict = dict(
            X_train=X_train.values.astype('float32'), y_train=y_train,
            X_valid=X_val.values.astype('float32'), y_valid=y_val,
            X_test=X_test.values.astype('float32'), y_test=y_test,
            X_2=X_2.values.astype('float32'), y_2=y_2,
            X_3=X_3.values.astype('float32'), y_3=y_3,
            X_4=X_4.values.astype('float32'), y_4=y_4,
            num_features=num_features,
            num_classes=num_classes
        )

        with open(pkl_path, "wb") as fp:
            pickle.dump(data_dict, fp)

    return data_dict


class QForest_config:
    def __init__(self, data, lr_base, nLayer=1, choice_func="r_0.5",
feat_info = None, random_seed=42):
        self.model = "QForest"
        self.data = data
        self.data_set = data.name
        self.lr_base = lr_base
        self.nLayer = nLayer
        self.choice_func = choice_func
        self.seed = random_seed
        self.feat_info = feat_info
```

```
        self.max_features = None
        self.input_dropout = 0.1
        self.num_layers = 1
        self.flatten_output = True
        self.max_out = True
        self.data_normal = "BN"
        self.leaf_output = "leaf_distri"
        self.reg_L1 = 0.01
        self.reg_Gate = 0.1
        self.support_vector = "1"
        self.back_bone = 'resnet18_x'
        self.bagging_fraction = 1
        self.trainer = None
        self.task = "train"
        self.attention_alg = "eca_response"
        self.nMostEpochs = 10
        self.batch_size = 256
        self.depth = 5
        self.nTree = 1024
        self.nLayers = 1
        self.response_dim = data.nClasses


import os
import time
import numpy as np
from sklearn.metrics import roc_auc_score
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch_optimizer
from collections import OrderedDict
from tensorboardX import SummaryWriter

from .some_utils import get_latest_file, check_numpy, model_params

class Experiment(nn.Module):
    def __init__(self, config, data, model, experiment_name=None,
warm_start=False, Optimizer=torch.optim.Adam, optimizer_params={},
verbose=False, n_last_checkpoints=1, **kwargs):
        super().__init__()
        self.data = data
        self.model = model
        self.Optimizer = Optimizer
        self.opt_parameters = optimizer_params
        if hasattr(self.model,"parameters"):
            self.opt = self.Optimizer(filter(lambda p: p.requires_grad,
self.model.parameters()),**self.opt_parameters )
        self.step = 0
        self.verbose = verbose
        self.n_last_checkpoints = n_last_checkpoints
        self.isFirstBackward = True

        if experiment_name is None:
            experiment_name =
'untitled_{}.{:0>2d}.{:0>2d}_{:0>2d}_{:0>2d}'.format(*time.gmtime()[:5])
```

```
        self.experiment_path = os.path.join('logs/', experiment_name)
        if not warm_start and experiment_name != 'debug':
            if os.path.exists(self.experiment_path):
                import shutil
                print(f'experiment {config.experiment} already exists, DELETE
it!!!')
                shutil.rmtree(self.experiment_path)
            assert not os.path.exists(self.experiment_path), 'experiment {}
already exists'.format(experiment_name)

        if hasattr(config, "no_inner_writer"):
            self.writer = None
        else:
            self.writer = SummaryWriter(self.experiment_path,
comment=experiment_name)

        if warm_start:
            self.load_checkpoint()

        self.nFeat4Train = self.data.nFeature

    def SetLearner(self, wLearner):
        self.model = wLearner
        self.opt = self.Optimizer(filter(lambda p: p.requires_grad,
self.model.parameters()),**self.opt_parameters )
        self.opt = torch_optimizer.Lookahead(self.opt, k=5, alpha=0.5)

    def save_checkpoint(self, tag=None, path=None, mkdir=True, **kwargs):
        assert tag is None or path is None, "please provide either tag or
path or nothing, not both"
        if tag is None and path is None:
            tag = "temp_{}".format(self.step)
        if path is None:
            path = os.path.join(self.experiment_path,
"checkpoint_{}.pth".format(tag))
        if mkdir:
            os.makedirs(os.path.dirname(path), exist_ok=True)
        torch.save(OrderedDict([
            ('model', self.state_dict(**kwargs)),
            ('opt', self.opt.state_dict()),
            ('step', self.step)
        ]), path)

        return path

    def load_checkpoint(self, tag=None, path=None, **kwargs):
        if self.model.config.task=="predict":
            checkpoint = torch.load(path)
        else:
            assert tag is None or path is None, "please provide either tag or
path or nothing, not both"
            if tag is None and path is None:
                path = get_latest_file(os.path.join(self.experiment_path,
'checkpoint_temp_[0-9]*.pth'))
            elif tag is not None and path is None:
                path = os.path.join(self.experiment_path,
"checkpoint_{}.pth".format(tag))
```

```
        checkpoint = torch.load(path)

        self.load_state_dict(checkpoint['model'], **kwargs)
        self.opt.load_state_dict(checkpoint['opt'])
        self.step = int(checkpoint['step'])

        return self

    def loss_on_batch(self, y_output, y_batch):
        loss = F.cross_entropy(y_output, y_batch.long())
        loss = loss.mean()

        loss = loss + self.model.reg_L1*self.model.config.reg_L1 +
self.model.L_gate*self.model.config.reg_Gate

        return loss

    def train_on_batch(self, *batch, device):
        x_batch, y_batch = batch
        x_batch = torch.as_tensor(x_batch, device=device)
        y_batch = torch.as_tensor(y_batch, device=device)
        self.y_batch = y_batch

        self.model.train()
        self.opt.zero_grad()
        y_output=self.model(x_batch)
        loss = self.loss_on_batch(y_output,y_batch)
        loss.backward()
        self.isFirstBackward = False
        self.opt.step()
        self.step += 1
        self.y_batch = None
        if self.writer is not None:
            self.writer.add_scalar('train loss', loss.item(), self.step)
        self.metrics = {'loss': loss.item()}
        del loss

        return self.metrics

    def evaluate_auc(self, X_test, y_test, device, batch_size=256):
        X_test = torch.as_tensor(X_test, device=device)
        y_test = check_numpy(y_test)
        self.model.train(False)
        with torch.no_grad():
            logits, dict_info = process_in_chunks(self.model, X_test,
batch_size=batch_size)
            logits = F.softmax(logits, dim=1)
            logits = check_numpy(logits)
            y_pred = np.argmax(logits, axis=1)
            auc = roc_auc_score(y_test, y_pred)

        return auc

    def AfterEpoch(self, epoch, XX_, YY_, best_auc, isTest=False,
isPredict=False):
        t0=time.time()
```

```
        config = self.model.config
        self.y_batch = None
        if isTest:
            self.load_checkpoint(tag='best_auc')

        auc  = self.evaluate_auc(XX_, YY_, device=config.device,
batch_size=config.eval_batch_size)

        self.model.AfterEpoch(isBetter=auc > best_auc, epoch=epoch, accu=auc)
        loss_step = self.metrics['loss']
        if isTest:
            print(f'====== Best step: {self.step} test={XX_.shape}
AUC@Test={auc:.5f} \ttime={time.time()-t0:.2f}')
        else:
            print(f"\n{self.step}\tnP={model_params(self.model)},
nF4={self.nFeat4Train}\t{loss_step:.5f}\tT={time.time()-t0:.2f}"\f"\tVal
AUC:{auc:.4f}" )

        return auc


import numpy as np
import gc
import torch
import torch.nn as nn
from qhoptim.pyt import QHAdam
import seaborn as sns

from .experiment import *
from .some_utils import *

class QuantumForest(object):
    def __init__(self, config, data, feat_info=None, visual=None):
        super(QuantumForest, self).__init__()
        self.config = config
        self.problem = None
        self.preprocess = None
        self.Learners=[]
        in_features = config.in_features
        qForest = QF_Net(in_features, config, feat_info=feat_info,
visual=visual).to(config.device)
        self.Learners.append(qForest)
        self._n_classes = None
        self.best_iteration_ = 0
        self.best_iteration = 0
        self.best_score = 0
        self.visual = visual

        self.optimizer=QHAdam
        self.optimizer_params = { 'nus':(0.7, 1.0), 'betas':(0.95,
0.998),'lr':config.lr_base,'weight_decay': 1e-8 }

        config.eval_batch_size = 256

        wLearner=self.Learners[-1]
```

```
        self.trainer = Experiment(
            config,
            data,
            model=wLearner,
            experiment_name=config.experiment,
            warm_start=False,
            Optimizer=self.optimizer,
            optimizer_params=self.optimizer_params,
            verbose=True,
            n_last_checkpoints=5,
        )

        config.trainer = self.trainer
        self.trainer.SetLearner(wLearner)

    def fit(self, X_train_0, y_train, eval_set, categorical_feature=None,
discrete_feature=None, flags={'report_frequency':1000}):
        assert len(eval_set)==1
        X_eval_0, y_valid = eval_set[0]
        config = self.config
        gc.collect()
        self.feat_info={"categorical":categorical_feature,
"discrete":discrete_feature}

        loss_history, auc_history = [], []
        best_auc, best_epoch_ = float('inf'), float('inf')
        early_stopping_rounds = 3000
        early_stopping_epochs = 10
        trainer = self.trainer
        data = self.trainer.data
        wLearner=self.Learners[-1]

        wLearner.AfterEpoch(isBetter=True, epoch=0)
        epoch, t0=0, time.time()
        nBatch =
(int)(data.__len__()*config.bagging_fraction)//config.batch_size
        nBatch = min(1024,nBatch)

        for epoch in range(config.nMostEpochs):
            for batch in data.yield_batch(config.batch_size,
subsample=config.bagging_fraction, shuffle=True, nMostBatch=nBatch):
                metrics = trainer.train_on_batch(*batch,
device=config.device)
                loss_history.append(metrics['loss'])
                if trainer.step%10==0 or (trainer.step)%nBatch==0:
                    print(f"\r============ {trainer.step} {trainer.step-
epoch*nBatch}/{nBatch}\t{metrics['loss']:.5f}\tL1=[{wLearner.reg_L1:.4g}*{con
fig.reg_L1}]"f"\tL2=[{wLearner.L_gate:.4g}*{config.reg_Gate}]\ttime={time.tim
e()-t0:.2f}\t",end="")

            if torch.cuda.is_available():
                torch.cuda.empty_cache()
            auc = trainer.AfterEpoch(epoch, data.X_valid, y_valid, best_auc)
            if auc > best_auc:
                best_auc, best_epoch_ = auc, epoch
                trainer.save_checkpoint(tag='best_auc')
            auc_history.append(auc)
```

```
            print(f"Time: [{time.time()-t0:.6f}]   [{epoch}]  valid auc:
{auc}")

            if epoch > best_epoch_ + early_stopping_epochs:
                print('BREAK. There is no improvement for {}
steps'.format(early_stopping_rounds))
                print(f"\tBest epoch: {best_epoch_} Best Val AUC:
{best_auc:.5f}")
                break

        self.best_score = best_auc

        return self

    def predict(self, X_, pred_leaf=False, pred_contrib=False,
raw_score=False, flag=0x0):
        self.profile.Snapshot("PRED_0")
        Y_ = self.predict_raw(X_, pred_leaf, pred_contrib, raw_score,
flag=flag)
        Y_ = self.problem.AfterPredict(X_, Y_)
        Y_ = self.problem.OnResult(Y_, pred_leaf, pred_contrib, raw_score)

        return Y_


import sys
import argparse
import time
import numpy as np
import torch
import torch.nn.functional as F

import quantum_forest

def QF_test(data, config, visual=None, feat_info=None):
    YY_train, YY_valid, YY_test, YY_2, YY_3, YY_4 = data.y_train,
data.y_valid, data.y_test, data.y_2, data.y_3, data.y_4

    data.Y_mean, data.Y_std = YY_train.mean(), YY_train.std()

    in_features = data.X_train.shape[1]
    config.in_features = in_features
    config.tree_module = quantum_forest.DeTree

    learner = quantum_forest.QuantumForest(config, data, feat_info=feat_info,
visual=visual).fit(data.X_train, YY_train, eval_set=[(data.X_valid,
YY_valid)])
    trainer, best_auc = learner.trainer,learner.best_score

    if data.X_test is not None:
       portsmouth_auc = trainer.evaluate_auc(data.X_test, YY_test,
device=config.device, batch_size=config.eval_batch_size)
       bedford_auc = trainer.evaluate_auc(data.X_2, YY_2,
device=config.device, batch_size=config.eval_batch_size)
       oxford_auc = trainer.evaluate_auc(data.X_3, YY_3,
device=config.device, batch_size=config.eval_batch_size)
       birmingham_auc = trainer.evaluate_auc(data.X_4, YY_4,
```

```
device=config.device, batch_size=config.eval_batch_size)

    trainer.save_checkpoint(tag=f'last_{best_auc:.6f}')

    return best_auc, portsmouth_auc, bedford_auc, oxford_auc, birmingham_auc

def Fold_learning(fold_n, data, config, visual):
    t0 = time.time()
    if config.feat_info == "importance":
        feat_info = get_feature_info(data,fold_n)
    else:
        feat_info = None
    validation_auc, portsmouth_test_auc, bedford_valid_auc, oxford_valid_auc,
birmingham_valid_auc = QF_test(data, config, visual, feat_info)

    return

parser = argparse.ArgumentParser()
parser.add_argument('--dataset', default='COVID')
parser.add_argument('--data_root',required=True)
parser.add_argument('--model', default="QForest")
parser.add_argument('--epochs', default=10, type=int)
parser.add_argument('--learning_rate', default="0.01", type=float)
parser.add_argument('--subsample', default="1", type=float)
parser.add_argument('--attention', default="eca_response", type=str)

args = parser.parse_args()
dataset = args.dataset
data = quantum_forest.TabularDataset(dataset, data_path=args.data_root,
random_state=42, quantile_transform=True, quantile_noise=1e-3)

config = quantum_forest.QForest_config(data, 0.001)
random_state = 42
config.device = quantum_forest.OnInitInstance(random_state)
config.model = args.model
config.dataset = args.dataset
config.lr_base = args.learning_rate
config.batch_size = 256
config.nMostEpochs = args.epochs
config.bagging_fraction = args.subsample
config.attention_alg = args.attention
config.depth, config.nTree = 5, 1024

config, visual = quantum_forest.InitExperiment(config, 0)
data.onFold(0, config,
pkl_path=f"{args.data_root}/{dataset}/FOLD_Quantile_{config.model}.pickle")
Fold_learning(0, data, config, visual)
```

DNDT

The DNDT implementation in wOOL (2018) was used. The implementation was extended with weight decay (PyTorch, 2021a) and Lookahead (Pytorch-optimizer, 2021a), using the code below.

The following code was used to create the DNDT model:

```
import torch
from functools import reduce


def binning(x, cut_points, temperature=0.1):
    D = cut_points.shape[0]
    W = torch.reshape(torch.linspace(1.0, D + 1.0, D + 1), [1, -1])
    cut_points, _ = torch.sort(cut_points)
    b = torch.cumsum(torch.cat([torch.zeros([1]), -cut_points], 0),0)
    h = torch.matmul(x, W) + b
    res = torch.exp(h-torch.max(h))
    res = res/torch.sum(res, dim=-1, keepdim=True)

    return h

def kronecker_product(a, b):
    res = torch.einsum('ij,ik->ijk', [a, b])
    res = torch.reshape(res, [-1, np.prod(res.shape[1:])])

    return res

def DNDT(x, cut_points_list, leaf_score, temperature=0.1):
    leaf = reduce(kronecker_product, map(lambda z: binning(x[:, z[0]:z[0] +
1], z[1], temperature), enumerate(cut_points_list)))

    return torch.matmul(leaf, leaf_score)
```

The following code was used to pre-process the datasets and divide each into 5 subsets of 7 features:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
import torch
```

```
import torch_optimizer


data = pd.read_csv('portsmouth_balanced.csv')
data2 = pd.read_csv('bedford.csv')
data3 = pd.read_csv('oxford.csv')
data4 = pd.read_csv('birmingham.csv')

cleanup_gender = {"Gender": {"M": 0, "F": 1}}

data = data.replace(cleanup_gender)
data2 = data2.replace(cleanup_gender)
data3 = data3.replace(cleanup_gender)
data4 = data4.replace(cleanup_gender)

one_hot = pd.get_dummies(data['Ethnicity'])
data = data.drop('Ethnicity', axis = 1)
data = data.join(one_hot)

one_hot2 = pd.get_dummies(data2['Ethnicity'])
data2 = data2.drop('Ethnicity', axis = 1)
data2 = data2.join(one_hot2)

one_hot3 = pd.get_dummies(data3['Ethnicity'])
data3 = data3.drop('Ethnicity', axis = 1)
data3 = data3.join(one_hot3)

one_hot4 = pd.get_dummies(data4['Ethnicity'])
data4 = data4.drop('Ethnicity', axis = 1)
data4 = data4.join(one_hot4)

features = ['Age', 'Vital_Sign Heart Rate', 'Vital_Sign Respiratory Rate',
'Vital_Sign Systolic Blood Pressure', 'Vital_Sign Diastolic Blood Pressure',
'Vital_Sign Oxygen Saturation', 'Vital_Sign Temperature Tympanic',
'Blood_Test HAEMOGLOBIN', 'Blood_Test HAEMATOCRIT', 'Blood_Test MEAN CELL
VOL.', 'Blood_Test WHITE CELLS', 'Blood_Test NEUTROPHILS', 'Blood_Test
LYMPHOCYTES', 'Blood_Test MONOCYTES', 'Blood_Test EOSINOPHILS', 'Blood_Test
BASOPHILS', 'Blood_Test PLATELETS', 'Blood_Test SODIUM', 'Blood_Test
POTASSIUM', 'Blood_Test UREA', 'Blood_Test CREATININE', 'Blood_Test eGFR',
'Blood_Test ALT', 'Blood_Test ALK.PHOSPHATASE', 'Blood_Test BILIRUBIN',
'Blood_Test ALBUMIN', 'Blood_Test CRP']

X_cont = data[features]
X_categ = data[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y = data["Covid-19 Positive"]

X_2_cont = data2[features]
X_2_categ = data2[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_2 = data2["Covid-19 Positive"]

X_3_cont = data3[features]
X_3_categ = data3[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_3 = data3["Covid-19 Positive"]
```

```
X_4_cont = data4[features]
X_4_categ = data4[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_4 = data4["Covid-19 Positive"]

x_train_cont, x_test_cont, x_train_categ, x_test_categ, y_train, y_test =
train_test_split(X_cont, X_categ, y, test_size=0.2, random_state=42)

x_train_categ = x_train_categ.to_numpy()
x_test_categ = x_test_categ.to_numpy()
x_train_cont = x_train_cont.to_numpy()
x_test_cont = x_test_cont.to_numpy()
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()

X_2_categ = X_2_categ.to_numpy()
X_2_cont = X_2_cont.to_numpy()
y_2 = y_2.to_numpy()

X_3_categ = X_3_categ.to_numpy()
X_3_cont = X_3_cont.to_numpy()
y_3 = y_3.to_numpy()

X_4_categ = X_4_categ.to_numpy()
X_4_cont = X_4_cont.to_numpy()
y_4 = y_4.to_numpy()

scaler = StandardScaler()
x_train_cont = scaler.fit_transform(x_train_cont, y_train)
x_test_cont = scaler.transform(x_test_cont)
X_2_cont = scaler.transform(X_2_cont)
X_3_cont = scaler.transform(X_3_cont)
X_4_cont = scaler.transform(X_4_cont)

x_train = np.concatenate((x_train_cont, x_train_categ), axis=1)
x_test = np.concatenate((x_test_cont, x_test_categ), axis=1)
x_2 = np.concatenate((X_2_cont, X_2_categ), axis=1)
x_3 = np.concatenate((X_3_cont, X_3_categ), axis=1)
x_4 = np.concatenate((X_4_cont, X_4_categ), axis=1)

x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train,
test_size=0.2, random_state=42)

def select(array, columns):
    return array[:, columns]

def splitx(array):
    a = select(array, [0, 5, 10, 15, 20, 25, 30])
    b = select(array, [1, 6, 11, 16, 21, 26, 31])
    c = select(array, [2, 7, 12, 17, 22, 27, 32])
    d = select(array, [3, 8, 13, 18, 23, 28, 33])
    e = select(array, [4, 9, 14, 19, 24, 29, 34])
    return a, b, c, d, e

def nptotensor(arrays):
    return torch.from_numpy(arrays[0].astype(np.float32)),
torch.from_numpy(arrays[1].astype(np.float32)),
```

```
torch.from_numpy(arrays[2].astype(np.float32)),
torch.from_numpy(arrays[3].astype(np.float32)),
torch.from_numpy(arrays[4].astype(np.float32))

x_train_1, x_train_2, x_train_3, x_train_4, x_train_5 =
nptotensor(splitx(x_train))
x_valid_1, x_valid_2, x_valid_3, x_valid_4, x_valid_5 =
nptotensor(splitx(x_valid))
x_test_1, x_test_2, x_test_3, x_test_4, x_test_5 = nptotensor(splitx(x_test))
x_2_1, x_2_2, x_2_3, x_2_4, x_2_5 = nptotensor(splitx(x_2))
x_3_1, x_3_2, x_3_3, x_3_4, x_3_5 = nptotensor(splitx(x_3))
x_4_1, x_4_2, x_4_3, x_4_4, x_4_5 = nptotensor(splitx(x_4))

y_train = torch.from_numpy(y_train.astype(np.float32)).type(torch.LongTensor)
```

The following code was used to train and test the DNDT model (example for one subset of 7 features):

```
num_cut = [1, 1, 1, 1, 1, 1, 1]
cut_points_list = [torch.rand([i], requires_grad=True) for i in num_cut]
num_leaf = np.prod(np.array(num_cut) + 1)
num_class = 2
leaf_score = torch.rand([num_leaf, num_class], requires_grad=True)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(cut_points_list + [leaf_score], lr=0.1,
weight_decay=0.0001)
optimizer = torch_optimizer.Lookahead(optimizer, k=5, alpha=0.5)
epochs = 2000

for i in range(epochs):
    optimizer.zero_grad()
    y_pred = DNDT(x_train_1, cut_points_list, leaf_score, temperature=0.1)
    loss = loss_function(y_pred, y_train)
    loss.backward()
    optimizer.step()
    if i % 100 == 0:
        print("Epoch:", i, loss.detach().numpy())

y_valid_pred_1 = DNDT(x_valid_1, cut_points_list, leaf_score,
temperature=0.1)
y_valid_pred_1 = np.argmax(y_valid_pred_1.detach().numpy(), axis=1)

y_test_pred_1 = DNDT(x_test_1, cut_points_list, leaf_score, temperature=0.1)
y_test_pred_1 = np.argmax(y_test_pred_1.detach().numpy(), axis=1)

y_2_pred_1 = DNDT(x_2_1, cut_points_list, leaf_score, temperature=0.1)
y_2_pred_1 = np.argmax(y_2_pred_1.detach().numpy(), axis=1)

y_3_pred_1 = DNDT(x_3_1, cut_points_list, leaf_score, temperature=0.1)
y_3_pred_1 = np.argmax(y_3_pred_1.detach().numpy(), axis=1)
```

```
y_4_pred_1 = DNDT(x_4_1, cut_points_list, leaf_score, temperature=0.1)
y_4_pred_1 = np.argmax(y_4_pred_1.detach().numpy(), axis=1)
```

The following code was used to combine the predictions of models trained on different subsets of features by majority voting:

```
def ypred(ypred1, ypred2, ypred3, ypred4, ypred5):
    y_pred = ypred1 + ypred2 + ypred3 + ypred4 + ypred5
    y_pred = np.where(y_pred > 2, 1, 0)
    return y_pred

y_valid_pred = ypred(y_valid_pred_1, y_valid_pred_2, y_valid_pred_3,
y_valid_pred_4, y_valid_pred_5)
auc = roc_auc_score(y_valid, y_valid_pred)
print("Validation AUC:", auc)

y_test_pred = ypred(y_test_pred_1, y_test_pred_2, y_test_pred_3,
y_test_pred_4, y_test_pred_5)
Portsmouth_test_auc = roc_auc_score(y_test, y_test_pred)

y_2_pred = ypred(y_2_pred_1, y_2_pred_2, y_2_pred_3, y_2_pred_4, y_2_pred_5)
Bedford_valid_auc = roc_auc_score(y_2, y_2_pred)

y_3_pred = ypred(y_3_pred_1, y_3_pred_2, y_3_pred_3, y_3_pred_4, y_3_pred_5)
Oxford_valid_auc = roc_auc_score(y_3, y_3_pred)

y_4_pred = ypred(y_4_pred_1, y_4_pred_2, y_4_pred_3, y_4_pred_4, y_4_pred_5)
Birmingham_valid_auc = roc_auc_score(y_4, y_4_pred)
```

Attention-based models

TabNet

The TabNet implementation in the Pytorch Tabular library (PyTorch Tabular, 2021) was used. The implementation was extended with weight decay (PyTorch, 2021a), dropout (PyTorch, 2021c) and batch normalisation (PyTorch, 2021b), using the code below.

The following code was used to create the TabNet encoder:

```
import numpy as np
```

```
import torch
from torch.nn import Linear, BatchNorm1d, ReLU, Dropout

from pytorch_tabnet import sparsemax


def initialize_glu(module, input_dim, output_dim):
    gain_value = np.sqrt((input_dim + output_dim) / np.sqrt(input_dim))
    torch.nn.init.xavier_normal_(module.weight, gain=gain_value)

    return

def initialize_non_glu(module, input_dim, output_dim):
    gain_value = np.sqrt((input_dim + output_dim) / np.sqrt(4 * input_dim))
    torch.nn.init.xavier_normal_(module.weight, gain=gain_value)

    return


class GBN(torch.nn.Module):
    def __init__(self, input_dim, virtual_batch_size=32, momentum=0.9):
        super(GBN, self).__init__()

        self.input_dim = input_dim
        self.virtual_batch_size = virtual_batch_size
        self.bn = BatchNorm1d(self.input_dim, momentum=momentum)

    def forward(self, x):
        chunks = x.chunk(int(np.ceil(x.shape[0] / self.virtual_batch_size)),
0)
        res = [self.bn(x_) for x_ in chunks]

        return torch.cat(res, dim=0)


class GLU_Layer(torch.nn.Module):
    def __init__(self, input_dim, output_dim, fc=None, virtual_batch_size=32,
momentum=0.9):
        super(GLU_Layer, self).__init__()

        self.output_dim = output_dim
        if fc:
            self.fc = fc
        else:
            self.fc = Linear(input_dim, 2 * output_dim, bias=False)
        initialize_glu(self.fc, input_dim, 2 * output_dim)
        self.bn = GBN(2 * output_dim, virtual_batch_size=virtual_batch_size,
momentum=momentum)
        self.dropout = Dropout(p=0.1)

    def forward(self, x):
        x = self.fc(x)
        x = self.bn(x)
        x = self.dropout(x)
        out = torch.mul(x[:, : self.output_dim], torch.sigmoid(x[:,
self.output_dim :]))
```

```
        return out


class GLU_Block(torch.nn.Module):
    def __init__(
        self,
        input_dim,
        output_dim,
        n_glu=2,
        first=False,
        shared_layers=None,
        virtual_batch_size=32,
        momentum=0.9,
    ):
        super(GLU_Block, self).__init__()

        self.first = first
        self.shared_layers = shared_layers
        self.n_glu = n_glu
        self.glu_layers = torch.nn.ModuleList()

        params = {"virtual_batch_size": virtual_batch_size, "momentum":
momentum}

        fc = shared_layers[0] if shared_layers else None
        self.glu_layers.append(GLU_Layer(input_dim, output_dim, fc=fc,
**params))
        for glu_id in range(1, self.n_glu):
            fc = shared_layers[glu_id] if shared_layers else None
            self.glu_layers.append(GLU_Layer(output_dim, output_dim, fc=fc,
**params))

    def forward(self, x):
        scale = torch.sqrt(torch.FloatTensor([0.5]).to(x.device))
        if self.first:
            x = self.glu_layers[0](x)
            layers_left = range(1, self.n_glu)
        else:
            layers_left = range(self.n_glu)

        for glu_id in layers_left:
            x = torch.add(x, self.glu_layers[glu_id](x))
            x = x * scale

        return x


class FeatTransformer(torch.nn.Module):
    def __init__(
        self,
        input_dim,
        output_dim,
        shared_layers,
        n_glu_independent,
        virtual_batch_size=32,
        momentum=0.9,
```

```
    ):
        super(FeatTransformer, self).__init__()

        params = {"n_glu": n_glu_independent, "virtual_batch_size":
virtual_batch_size, "momentum": momentum}

        if shared_layers is None:
            self.shared = torch.nn.Identity()
            is_first = True
        else:
            self.shared = GLU_Block(
                input_dim,
                output_dim,
                first=True,
                shared_layers=shared_layers,
                n_glu=len(shared_layers),
                virtual_batch_size=virtual_batch_size,
                momentum=momentum,
            )
            is_first = False

        if n_glu_independent == 0:
            self.specifics = torch.nn.Identity()
        else:
            spec_input_dim = input_dim if is_first else output_dim
            self.specifics = GLU_Block(
                spec_input_dim, output_dim, first=is_first, **params
            )

    def forward(self, x):
        x = self.shared(x)
        x = self.specifics(x)

        return x


class AttentiveTransformer(torch.nn.Module):
    def __init__(
        self,
        input_dim,
        output_dim,
        virtual_batch_size=128,
        momentum=0.02,
        mask_type="sparsemax",
    ):
        super(AttentiveTransformer, self).__init__()

        self.fc = Linear(input_dim, output_dim, bias=False)
        initialize_non_glu(self.fc, input_dim, output_dim)
        self.bn = GBN(output_dim, virtual_batch_size=virtual_batch_size,
momentum=momentum)
        self.dropout = Dropout(p=0.1)

        if mask_type == "sparsemax":
            self.selector = sparsemax.Sparsemax(dim=-1)
        elif mask_type == "entmax":
            self.selector = sparsemax.Entmax15(dim=-1)
```

```
        else:
            raise NotImplementedError(
                "Please choose either sparsemax" + "or entmax as masktype")

    def forward(self, priors, processed_feat):
        x = self.fc(processed_feat)
        x = self.bn(x)
        x = self.dropout(x)
        x = torch.mul(x, priors)
        x = self.selector(x)

        return x


class TabNetEncoder(torch.nn.Module):
    def __init__(
        self,
        input_dim,
        output_dim,
        n_d=256,
        n_a=256,
        n_steps=3,
        gamma=1,
        n_independent=2,
        n_shared=2,
        epsilon=1e-15,
        virtual_batch_size=32,
        momentum=0.9,
        mask_type="sparsemax",
    ):
        super(TabNetEncoder, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.is_multi_task = isinstance(output_dim, list)
        self.n_d = n_d
        self.n_a = n_a
        self.n_steps = n_steps
        self.gamma = gamma
        self.epsilon = epsilon
        self.n_independent = n_independent
        self.n_shared = n_shared
        self.virtual_batch_size = virtual_batch_size
        self.mask_type = mask_type
        self.initial_bn = BatchNorm1d(self.input_dim, momentum=0.9)

        if self.n_shared > 0:
            shared_feat_transform = torch.nn.ModuleList()
            for i in range(self.n_shared):
                if i == 0:
                    shared_feat_transform.append(
                        Linear(self.input_dim, 2 * (n_d + n_a), bias=False)
                    )
                else:
                    shared_feat_transform.append(
                        Linear(n_d + n_a, 2 * (n_d + n_a), bias=False)
                    )
```

```
        else:
            shared_feat_transform = None

        self.initial_splitter = FeatTransformer(
            self.input_dim,
            n_d + n_a,
            shared_feat_transform,
            n_glu_independent=self.n_independent,
            virtual_batch_size=self.virtual_batch_size,
            momentum=momentum,
        )

        self.feat_transformers = torch.nn.ModuleList()
        self.att_transformers = torch.nn.ModuleList()

        for step in range(n_steps):
            transformer = FeatTransformer(
                self.input_dim,
                n_d + n_a,
                shared_feat_transform,
                n_glu_independent=self.n_independent,
                virtual_batch_size=self.virtual_batch_size,
                momentum=momentum,
            )
            attention = AttentiveTransformer(
                n_a,
                self.input_dim,
                virtual_batch_size=self.virtual_batch_size,
                momentum=momentum,
                mask_type=self.mask_type,
            )
            self.feat_transformers.append(transformer)
            self.att_transformers.append(attention)

    def forward(self, x, prior=None):
        x = self.initial_bn(x)

        if prior is None:
            prior = torch.ones(x.shape).to(x.device)

        M_loss = 0
        att = self.initial_splitter(x)[:, self.n_d :]

        steps_output = []
        for step in range(self.n_steps):
            M = self.att_transformers[step](prior, att)
            M_loss += torch.mean(
                torch.sum(torch.mul(M, torch.log(M + self.epsilon)), dim=1)
            )
            prior = torch.mul(self.gamma - M, prior)
            masked_x = torch.mul(M, x)
            out = self.feat_transformers[step](masked_x)
            d = ReLU()(out[:, : self.n_d])
            steps_output.append(d)
            att = out[:, self.n_d :]
```

```
        M_loss /= self.n_steps

        return steps_output, M_loss
```

The following code was used to train and test the TabNet model:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

from pytorch_tabular import TabularModel
from pytorch_tabular.models import TabNetModelConfig
from pytorch_tabular.config import DataConfig, OptimizerConfig,
TrainerConfig, ExperimentConfig


data = pd.read_csv('portsmouth_balanced.csv')
data2 = pd.read_csv('bedford.csv')
data3 = pd.read_csv('oxford.csv')
data4 = pd.read_csv('birmingham.csv')

cont_cols = ['Age', 'Vital_Sign Heart Rate', 'Vital_Sign Respiratory Rate',
'Vital_Sign Systolic Blood Pressure', 'Vital_Sign Diastolic Blood Pressure',
'Vital_Sign Oxygen Saturation', 'Vital_Sign Temperature Tympanic',
'Blood_Test HAEMOGLOBIN', 'Blood_Test HAEMATOCRIT', 'Blood_Test MEAN CELL
VOL.', 'Blood_Test WHITE CELLS', 'Blood_Test NEUTROPHILS', 'Blood_Test
LYMPHOCYTES', 'Blood_Test MONOCYTES', 'Blood_Test EOSINOPHILS', 'Blood_Test
BASOPHILS', 'Blood_Test PLATELETS', 'Blood_Test SODIUM', 'Blood_Test
POTASSIUM', 'Blood_Test UREA', 'Blood_Test CREATININE', 'Blood_Test eGFR',
'Blood_Test ALT', 'Blood_Test ALK.PHOSPHATASE', 'Blood_Test BILIRUBIN',
'Blood_Test ALBUMIN', 'Blood_Test CRP']

cat_cols = ['Gender', 'Ethnicity']

train, test = train_test_split(data, random_state=42)
train, val = train_test_split(train, random_state=42)

data_config = DataConfig(
    target=['Covid-19 Positive'],
    continuous_cols=cont_cols,
    categorical_cols=cat_cols)

trainer_config = TrainerConfig(
    batch_size=256,
    max_epochs=50,
    auto_lr_find=False)

optimizer_config = OptimizerConfig(
    optimizer='AdamW',
    optimizer_params={'lr': 0.001, 'weight_decay': 0.0001})
```

```
model_config = TabNetModelConfig(
    task="classification",
    n_d=256,
    n_a=256,
    n_steps=3,
    gamma=1,
    virtual_batch_size=32)

tabnet = TabularModel(
    data_config=data_config,
    model_config=model_config,
    optimizer_config=optimizer_config,
    trainer_config=trainer_config)

tabnet.fit(train=train, validation=val)

def calculate_auc(y_true, y_pred):
    if isinstance(y_true, pd.DataFrame) or isinstance(y_true, pd.Series):
        y_true = y_true.values
    if isinstance(y_pred, pd.DataFrame) or isinstance(y_pred, pd.Series):
        y_pred = y_pred.values
    if y_true.ndim>1:
        y_true=y_true.ravel()
    if y_pred.ndim>1:
        y_pred=y_pred.ravel()
    auc = roc_auc_score(y_true, y_pred)

    return auc

pred = node.predict(test)
pred2 = node.predict(data2)
pred3 = node.predict(data3)
pred4 = node.predict(data4)

Portsmouth_test_auc = calculate_auc(test['Covid-19 Positive'],
pred["prediction"])
Bedford_valid_auc = calculate_auc(data2['Covid-19 Positive'],
pred2["prediction"])
Oxford_valid_auc = calculate_auc(data3['Covid-19 Positive'],
pred3["prediction"])
Birmingham_valid_auc = calculate_auc(data4['Covid-19 Positive'],
pred4["prediction"])
```

TabTransformer

The TabTransformer implementation of Wang (2020) was used. The implementation was extended with weight decay (PyTorch, 2021a), dropout (PyTorch, 2021c), batch normalisation (PyTorch, 2021b), stochastic weight averaging (PyTorch, 2021d) and Lookahead (Pytorch-optimizer, 2021a), using the code below.

The following code was used to create the TabTransformer model:

```
import torch
import torch.nn.functional as F
from torch import nn, einsum
from einops import rearrange


class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn

    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)


class GEGLU(nn.Module):
    def forward(self, x):
        x, gates = x.chunk(2, dim = -1)

        return x * F.gelu(gates)


class FeedForward(nn.Module):
    def __init__(self, dim, mult = 4, dropout = 0.1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, dim * mult * 2),
            GEGLU(),
            nn.Dropout(dropout),
            nn.Linear(dim * mult, dim)
        )

    def forward(self, x, **kwargs):
        return self.net(x)


class Attention(nn.Module):
    def __init__(
        self,
        dim,
        heads = 8,
        dim_head = 16,
        dropout = 0.1
    ):
        super().__init__()
        inner_dim = dim_head * heads
        self.heads = heads
        self.scale = dim_head ** -0.5
```

```
        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)
        self.to_out = nn.Linear(inner_dim, dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        h = self.heads
        q, k, v = self.to_qkv(x).chunk(3, dim = -1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h = h),
(q, k, v))
        sim = einsum('b h i d, b h j d -> b h i j', q, k) * self.scale

        attn = sim.softmax(dim = -1)
        attn = self.dropout(attn)

        out = einsum('b h i j, b h j d -> b h i d', attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)', h = h)

        return self.to_out(out)


class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x, **kwargs):
        return self.fn(x, **kwargs) + x


class Transformer(nn.Module):
    def __init__(self, num_tokens, dim, depth, heads, dim_head, attn_dropout,
ff_dropout):
        super().__init__()
        self.embeds = nn.Embedding(num_tokens, dim)
        self.layers = nn.ModuleList([])

        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                Residual(PreNorm(dim, Attention(dim, heads = heads, dim_head
= dim_head, dropout = attn_dropout))),
                Residual(PreNorm(dim, FeedForward(dim, dropout =
ff_dropout))),
            ]))

    def forward(self, x):
        x = self.embeds(x)

        for attn, ff in self.layers:
            x = attn(x)
            x = ff(x)

        return x


class MLP(nn.Module):
    def __init__(self, dims, act = nn.ReLU()):
```

```
        super().__init__()
        dims_pairs = list(zip(dims[:-1], dims[1:]))
        layers = []
        for ind, (dim_in, dim_out) in enumerate(dims_pairs):
            is_last = ind >= (len(dims) - 1)
            linear = nn.Linear(dim_in, dim_out)
            layers.append(linear)
            bn = nn.BatchNorm1d(dim_out)
            layers.append(bn)

            if is_last:
                continue

            layers.append(act)
            dropout = nn.Dropout(p=0.1)
            layers.append(dropout)

        self.mlp = nn.Sequential(*layers)

    def forward(self, x):
        return self.mlp(x)


class TabTransformer(nn.Module):
    def __init__(
        self,
        *,
        categories,
        num_continuous,
        dim,
        depth,
        heads,
        dim_head = 16,
        dim_out = 1,
        mlp_hidden_mults = (4, 2),
        mlp_act = nn.ReLU(),
        num_special_tokens = 2,
        continuous_mean_std = None,
        attn_dropout = 0.1,
        ff_dropout = 0.1
    ):
        super().__init__()
        assert all(map(lambda n: n > 0, categories)), 'number of each
category must be positive'

        self.num_categories = len(categories)
        self.num_unique_categories = sum(categories)

        self.num_special_tokens = num_special_tokens
        total_tokens = self.num_unique_categories + num_special_tokens

        categories_offset = F.pad(torch.tensor(list(categories)), (1, 0),
value = num_special_tokens)
        categories_offset = categories_offset.cumsum(dim = -1)[:-1]
        self.register_buffer('categories_offset', categories_offset)

        if continuous_mean_std is not None:
```

```
            assert continuous_mean_std.shape == (num_continuous, 2),
f'continuous_mean_std must have a shape of ({num_continuous}, 2) where the
last dimension contains the mean and variance respectively'
        self.register_buffer('continuous_mean_std', continuous_mean_std)

        self.norm = nn.LayerNorm(num_continuous)
        self.num_continuous = num_continuous

        self.transformer = Transformer(
            num_tokens = total_tokens,
            dim = dim,
            depth = depth,
            heads = heads,
            dim_head = dim_head,
            attn_dropout = attn_dropout,
            ff_dropout = ff_dropout
        )

        input_size = (dim * self.num_categories) + num_continuous
        l = input_size // 8

        hidden_dimensions = list(map(lambda t: l * t, mlp_hidden_mults))
        all_dimensions = [input_size, *hidden_dimensions, dim_out]

        self.mlp = MLP(all_dimensions, act = mlp_act)

    def forward(self, x_categ, x_cont):
        assert x_categ.shape[-1] == self.num_categories, f'you must pass in
{self.num_categories} values for your categories input'
        x_categ += self.categories_offset

        x = self.transformer(x_categ)

        flat_categ = x.flatten(1)

        assert x_cont.shape[1] == self.num_continuous, f'you must pass in
{self.num_continuous} values for your continuous input'

        if self.continuous_mean_std is not None:
            mean, std = self.continuous_mean_std.unbind(dim = -1)
            x_cont = (x_cont - mean) / std

        normed_cont = self.norm(x_cont)

        x = torch.cat((flat_categ, normed_cont), dim = -1)
        return self.mlp(x)
```

The following code was used to train and test the TabTransformer model:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
import torch
import torch.nn as nn
import torch.utils.data as data_utils
from torch.utils.data import DataLoader
import torch_optimizer


data = pd.read_csv('portsmouth_balanced.csv')
data2 = pd.read_csv('bedford.csv')
data3 = pd.read_csv('oxford.csv')
data4 = pd.read_csv('birmingham.csv')

cleanup_gender = {"Gender": {"M": 0, "F": 1}}

data = data.replace(cleanup_gender)
data2 = data2.replace(cleanup_gender)
data3 = data3.replace(cleanup_gender)
data4 = data4.replace(cleanup_gender)

cleanup_ethnicity = {"Ethnicity": {'White': 0, 'South Asian': 1, 'Black': 2,
'Chinese': 3, 'Mixed': 4, 'Other': 5, 'Unknown': 6}}

data = data.replace(cleanup_ethnicity)
data2 = data2.replace(cleanup_ethnicity)
data3 = data3.replace(cleanup_ethnicity)
data4 = data4.replace(cleanup_ethnicity)

features = ['Age', 'Vital_Sign Heart Rate', 'Vital_Sign Respiratory Rate',
'Vital_Sign Systolic Blood Pressure', 'Vital_Sign Diastolic Blood Pressure',
'Vital_Sign Oxygen Saturation', 'Vital_Sign Temperature Tympanic',
'Blood_Test HAEMOGLOBIN', 'Blood_Test HAEMATOCRIT', 'Blood_Test MEAN CELL
VOL.', 'Blood_Test WHITE CELLS', 'Blood_Test NEUTROPHILS', 'Blood_Test
LYMPHOCYTES', 'Blood_Test MONOCYTES', 'Blood_Test EOSINOPHILS', 'Blood_Test
BASOPHILS', 'Blood_Test PLATELETS', 'Blood_Test SODIUM', 'Blood_Test
POTASSIUM', 'Blood_Test UREA', 'Blood_Test CREATININE', 'Blood_Test eGFR',
'Blood_Test ALT', 'Blood_Test ALK.PHOSPHATASE', 'Blood_Test BILIRUBIN',
'Blood_Test ALBUMIN', 'Blood_Test CRP']

X_cont = data[features]
X_categ = data[['Gender', 'Ethnicity']]
y = data["Covid-19 Positive"]

X_2_cont = data2[features]
X_2_categ = data2[['Gender', 'Ethnicity']]
y_2 = data2["Covid-19 Positive"]

X_3_cont = data3[features]
X_3_categ = data3[['Gender', 'Ethnicity']]
y_3 = data3["Covid-19 Positive"]

X_4_cont = data4[features]
X_4_categ = data4[['Gender', 'Ethnicity']]
y_4 = data4["Covid-19 Positive"]
```

```
X_train_cont, X_test_cont, X_train_categ, X_test_categ, y_train, y_test =
train_test_split(X_cont, X_categ, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_cont = scaler.fit_transform(X_train_cont, y_train)
X_test_cont = scaler.transform(X_test_cont)
X_2_cont = scaler.transform(X_2_cont)
X_3_cont = scaler.transform(X_3_cont)
X_4_cont = scaler.transform(X_4_cont)

X_train_cont = pd.DataFrame(X_train_cont, columns = features)
X_test_cont = pd.DataFrame(X_test_cont, columns = features)
X_2_cont = pd.DataFrame(X_2_cont, columns = features)
X_3_cont = pd.DataFrame(X_3_cont, columns = features)
X_4_cont = pd.DataFrame(X_4_cont, columns = features)

X_train_cont, X_valid_cont, X_train_categ, X_valid_categ, y_train, y_valid =
train_test_split(X_train_cont, X_train_categ, y_train, test_size=0.2,
random_state=42)

X_train_cont = torch.tensor(X_train_cont.values.astype(np.float32))
X_train_categ = torch.tensor(X_train_categ.values.astype(np.float32))
y_train = torch.tensor(y_train.values.astype(np.float32))

X_valid_cont = torch.tensor(X_valid_cont.values.astype(np.float32))
X_valid_categ = torch.tensor(X_valid_categ.values.astype(np.float32))
y_valid = torch.tensor(y_valid.values.astype(np.float32))

X_test_cont = torch.tensor(X_test_cont.values.astype(np.float32))
X_test_categ = torch.tensor(X_test_categ.values.astype(np.float32))
y_test = torch.tensor(y_test.values.astype(np.float32))

X_2_cont = torch.tensor(X_2_cont.values.astype(np.float32))
X_2_categ = torch.tensor(X_2_categ.values.astype(np.float32))
y_2 = torch.tensor(y_2.values.astype(np.float32))

X_3_cont = torch.tensor(X_3_cont.values.astype(np.float32))
X_3_categ = torch.tensor(X_3_categ.values.astype(np.float32))
y_3 = torch.tensor(y_3.values.astype(np.float32))

X_4_cont = torch.tensor(X_4_cont.values.astype(np.float32))
X_4_categ = torch.tensor(X_4_categ.values.astype(np.float32))
y_4 = torch.tensor(y_4.values.astype(np.float32))

train_tensor = data_utils.TensorDataset(X_train_cont, X_train_categ, y_train)
valid_tensor = data_utils.TensorDataset(X_valid_cont, X_valid_categ, y_valid)
test_tensor = data_utils.TensorDataset(X_test_cont, X_test_categ, y_test)
bedford_tensor = data_utils.TensorDataset(X_2_cont, X_2_categ, y_2)
oxford_tensor = data_utils.TensorDataset(X_3_cont, X_3_categ, y_3)
birmingham_tensor = data_utils.TensorDataset(X_4_cont, X_4_categ, y_4)

trainloader = DataLoader(train_tensor, batch_size=128, shuffle=True,
num_workers=2)
valloader = DataLoader(valid_tensor, batch_size=len(valid_tensor),
shuffle=True, num_workers=2)
testloader = DataLoader(test_tensor, batch_size=len(test_tensor),
shuffle=False, num_workers=2)
```

```
bedfordloader = DataLoader(bedford_tensor, batch_size=len(bedford_tensor),
shuffle=False, num_workers=2)
oxfordloader = DataLoader(oxford_tensor, batch_size=len(oxford_tensor),
shuffle=False, num_workers=2)
birminghamloader = DataLoader(birmingham_tensor,
batch_size=len(birmingham_tensor), shuffle=False, num_workers=2)


model = TabTransformer(
    categories = (2, 7),
    num_continuous = 27,
    dim = 32,
    dim_out = 1,
    depth = 6,
    heads = 8,
    attn_dropout = 0.1,
    ff_dropout = 0.1,
    mlp_hidden_mults = (4, 2),
    mlp_act = nn.ReLU(),
)

criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.AdamW(params=model.parameters(), lr=1e-3,
weight_decay=0.0001)
optimizer = torch_optimizer.Lookahead(optimizer, k=5, alpha=0.5)
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer,
T_0=15, T_mult=2)
swa_model = torch.optim.swa_utils.AveragedModel(model)
swa_start = 50
swa_scheduler = torch.optim.swa_utils.SWALR(optimizer, anneal_epochs=5,
swa_lr=0.01)

for epoch in range(100):
    train_loss = 0
    for i, data in enumerate(trainloader, 0):
        cont, categ, labels = data
        categ = categ.type(torch.LongTensor)
        labels = torch.unsqueeze(labels, 1)
        optimizer.zero_grad()
        outputs = model(categ, cont)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
    if epoch > swa_start:
        swa_model.update_parameters(model)
        swa_scheduler.step()
    else:
        scheduler.step()
    if epoch % 5 == 0:
        for data in valloader:
            cont, categ, labels = data
            categ = categ.type(torch.LongTensor)
            labels1 = torch.unsqueeze(labels, 1)
            outputs = swa_model(categ, cont)
            val_loss = criterion(outputs, labels1)
            outputs = torch.sigmoid(outputs)
```

```
            outputs = torch.where(outputs > 0.5, 1, 0)
            labels = labels.detach().numpy()
            outputs = outputs.detach().numpy()
            outputs = np.squeeze(outputs)
            val_auc = roc_auc_score(labels, outputs)
        print("Epoch:", epoch, "Train loss:", train_loss / len(trainloader),
"Val loss:", val_loss.item() / len(valloader), "Val AUC:", val_auc)

with torch.no_grad():
    for data in testloader:
        cont, categ, labels = data
        categ = categ.type(torch.LongTensor)
        outputs = swa_model(categ, cont)
        outputs = torch.sigmoid(outputs)
        outputs = torch.where(outputs > 0.5, 1, 0)
        labels = labels.detach().numpy()
        outputs = outputs.detach().numpy()
        outputs = np.squeeze(outputs)
        Portsmouth_test_auc = roc_auc_score(labels, outputs)

with torch.no_grad():
    for data in bedfordloader:
        cont, categ, labels = data
        categ = categ.type(torch.LongTensor)
        outputs = swa_model(categ, cont)
        outputs = torch.sigmoid(outputs)
        outputs = torch.where(outputs > 0.5, 1, 0)
        labels = labels.detach().numpy()
        outputs = outputs.detach().numpy()
        outputs = np.squeeze(outputs)
        Bedford_valid_auc = roc_auc_score(labels, outputs)

with torch.no_grad():
    for data in oxfordloader:
        cont, categ, labels = data
        categ = categ.type(torch.LongTensor)
        outputs = swa_model(categ, cont)
        outputs = torch.sigmoid(outputs)
        outputs = torch.where(outputs > 0.5, 1, 0)
        labels = labels.detach().numpy()
        outputs = outputs.detach().numpy()
        outputs = np.squeeze(outputs)
        Oxford_valid_auc = roc_auc_score(labels, outputs)

with torch.no_grad():
    for data in birminghamloader:
        cont, categ, labels = data
        categ = categ.type(torch.LongTensor)
        outputs = swa_model(categ, cont)
        outputs = torch.sigmoid(outputs)
        outputs = torch.where(outputs > 0.5, 1, 0)
        labels = labels.detach().numpy()
        outputs = outputs.detach().numpy()
        outputs = np.squeeze(outputs)
        Birmingham_valid_auc = roc_auc_score(labels, outputs)
```

SAINT

The SAINT implementation of Somepalli (2021) was used. The implementation was extended with weight decay (PyTorch, 2021a), dropout (PyTorch, 2021c), batch normalisation (PyTorch, 2021b) and Lookahead (Pytorch-optimizer, 2021a), using the code below.

The following code was used to create the SAINT model:

```
import torch
import torch.nn.functional as F
from torch import nn, einsum
from einops import rearrange


class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn

    def forward(self, x, **kwargs):
        return self.fn(x, **kwargs)


class GEGLU(nn.Module):
    def forward(self, x):
        x, gates = x.chunk(2, dim = -1)

        return x * F.gelu(gates)


class FeedForward(nn.Module):
    def __init__(self, dim, mult = 4, dropout = 0.8):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, dim * mult * 2),
            GEGLU(),
            nn.Dropout(dropout),
            nn.Linear(dim * mult, dim)
        )

    def forward(self, x, **kwargs):
        return self.net(x)


class Attention(nn.Module):
    def __init__(
        self,
        dim,
        heads = 8,
```

```
        dim_head = 16,
        dropout = 0.1
    ):
        super().__init__()
        inner_dim = dim_head * heads
        self.heads = heads
        self.scale = dim_head ** -0.5

        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)
        self.to_out = nn.Linear(inner_dim, dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        h = self.heads
        q, k, v = self.to_qkv(x).chunk(3, dim = -1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h = h),
(q, k, v))
        sim = einsum('b h i d, b h j d -> b h i j', q, k) * self.scale
        attn = sim.softmax(dim = -1)
        attn = self.dropout(attn)
        out = einsum('b h i j, b h j d -> b h i d', attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)', h = h)

        return self.to_out(out)


class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x, **kwargs):
        return self.fn(x, **kwargs) + x


class RowColTransformer(nn.Module):
    def __init__(self, num_tokens, dim, nfeats, depth, heads, dim_head,
attn_dropout, ff_dropout):
        super().__init__()
        self.embeds = nn.Embedding(num_tokens, dim)
        self.layers = nn.ModuleList([])
        self.mask_embed =  nn.Embedding(nfeats, dim)

        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                PreNorm(dim, Residual(Attention(dim, heads=heads,
dim_head=dim_head, dropout=attn_dropout))),
                PreNorm(dim, Residual(FeedForward(dim, dropout=ff_dropout))),
                PreNorm(dim * nfeats, Residual(Attention(dim * nfeats,
heads=heads, dim_head=64, dropout=attn_dropout))),
                PreNorm(dim * nfeats, Residual(FeedForward(dim * nfeats,
dropout=ff_dropout))),
                ]))

    def forward(self, x, x_cont=None, mask = None):
        if x_cont is not None:
```

```
            x = torch.cat((x,x_cont),dim=1)
        _, n, _ = x.shape
        for attn1, ff1, attn2, ff2 in self.layers:
            x = attn1(x)
            x = ff1(x)
            x = rearrange(x, 'b n d -> 1 b (n d)')
            x = attn2(x)
            x = ff2(x)
            x = rearrange(x, '1 b (n d) -> b n d', n = n)

        return x


class MLP(nn.Module):
    def __init__(self, dims, act = nn.ReLU()):
        super().__init__()
        dims_pairs = list(zip(dims[:-1], dims[1:]))
        layers = []
        for ind, (dim_in, dim_out) in enumerate(dims_pairs):
            is_last = ind >= (len(dims) - 1)
            linear = nn.Linear(dim_in, dim_out)
            layers.append(linear)
            bn = nn.BatchNorm1d(dim_out)
            layers.append(bn)

            if is_last:
                continue

            layers.append(act)
            dropout = nn.Dropout(p=0.1)
            layers.append(dropout)

        self.mlp = nn.Sequential(*layers)

    def forward(self, x):
        return self.mlp(x)


class simple_MLP(nn.Module):
    def __init__(self,dims):
        super(simple_MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(dims[0], dims[1]),
            nn.ReLU(),
            nn.Linear(dims[1], dims[2])
        )

    def forward(self, x):
        if len(x.shape)==1:
            x = x.view(x.size(0), -1)
        x = self.layers(x)

        return x


class sep_MLP(nn.Module):
    def __init__(self,dim,len_feats,categories):
```

```
        super(sep_MLP, self).__init__()
        self.len_feats = len_feats
        self.layers = nn.ModuleList([])
        for i in range(len_feats):
            self.layers.append(simple_MLP([dim,5*dim, categories[i]]))

    def forward(self, x):
        y_pred = list([])
        for i in range(self.len_feats):
            x_i = x[:,i,:]
            pred = self.layers[i](x_i)
            y_pred.append(pred)

        return y_pred


class SAINT(nn.Module):
    def __init__(
        self,
        *,
        categories,
        num_continuous,
        dim,
        depth,
        heads,
        dim_head = 16,
        dim_out = 1,
        mlp_hidden_mults = (4, 2),
        mlp_act = nn.ReLU(),
        num_special_tokens = 0,
        attn_dropout = 0.1,
        ff_dropout = 0.8,
        cont_embeddings = 'MLP',
        final_mlp_style = 'sep',
        y_dim = 2
        ):
        super().__init__()
        assert all(map(lambda n: n > 0, categories)), 'number of each
category must be positive'

        self.num_categories = len(categories)
        self.num_unique_categories = sum(categories)

        self.num_special_tokens = num_special_tokens
        self.total_tokens = self.num_unique_categories + num_special_tokens

        categories_offset = F.pad(torch.tensor(list(categories)), (1, 0),
value = num_special_tokens)
        categories_offset = categories_offset.cumsum(dim = -1)[:-1]
        self.register_buffer('categories_offset', categories_offset)

        self.norm = nn.LayerNorm(num_continuous)
        self.num_continuous = num_continuous
        self.dim = dim
        self.cont_embeddings = cont_embeddings
        self.final_mlp_style = final_mlp_style
```

```
        if self.cont_embeddings == 'MLP':
            self.simple_MLP = nn.ModuleList([simple_MLP([1,100,self.dim]) for
_ in range(self.num_continuous)])
            input_size = (dim * self.num_categories)  + (dim *
num_continuous)
            nfeats = self.num_categories + num_continuous
        elif self.cont_embeddings == 'pos_singleMLP':
            self.simple_MLP = nn.ModuleList([simple_MLP([1,100,self.dim]) for
_ in range(1)])
            input_size = (dim * self.num_categories)  + (dim *
num_continuous)
            nfeats = self.num_categories + num_continuous
        else:
            print('Continous features are not passed through attention')
            input_size = (dim * self.num_categories) + num_continuous
            nfeats = self.num_categories

        self.transformer = RowColTransformer(
            num_tokens = self.total_tokens,
            dim = dim,
            nfeats= nfeats,
            depth = depth,
            heads = heads,
            dim_head = dim_head,
            attn_dropout = attn_dropout,
            ff_dropout = ff_dropout,
        )

        l = input_size // 8
        hidden_dimensions = list(map(lambda t: l * t, mlp_hidden_mults))
        all_dimensions = [input_size, *hidden_dimensions, dim_out]

        self.mlp = MLP(all_dimensions, act = mlp_act)
        self.embeds = nn.Embedding(self.total_tokens, self.dim)

        cat_mask_offset =
F.pad(torch.Tensor(self.num_categories).fill_(2).type(torch.int8), (1, 0),
value = 0)
        cat_mask_offset = cat_mask_offset.cumsum(dim = -1)[:-1]

        con_mask_offset =
F.pad(torch.Tensor(self.num_continuous).fill_(2).type(torch.int8), (1, 0),
value = 0)
        con_mask_offset = con_mask_offset.cumsum(dim = -1)[:-1]

        self.register_buffer('cat_mask_offset', cat_mask_offset)
        self.register_buffer('con_mask_offset', con_mask_offset)

        self.mask_embeds_cat = nn.Embedding(self.num_categories*2, self.dim)
        self.mask_embeds_cont = nn.Embedding(self.num_continuous*2, self.dim)
        self.single_mask = nn.Embedding(2, self.dim)
        self.pos_encodings = nn.Embedding(self.num_categories+
self.num_continuous, self.dim)

        if self.final_mlp_style == 'common':
            self.mlp1 = simple_MLP([dim,(self.total_tokens)*2,
self.total_tokens])
```

```
            self.mlp2 = simple_MLP([dim ,(self.num_continuous), 1])
        else:
            self.mlp1 = sep_MLP(dim,self.num_categories,categories)
            self.mlp2 =
sep_MLP(dim,self.num_continuous,np.ones(self.num_continuous).astype(int))

        self.mlpfory = simple_MLP([dim ,1000, y_dim])

    def forward(self, x_categ, x_cont):
        x = self.transformer(x_categ, x_cont)
        cat_outs = self.mlp1(x[:,:self.num_categories,:])
        con_outs = self.mlp2(x[:,self.num_categories:,:])

        return cat_outs, con_outs
```

The following code was used to train and test the SAINT model:

```
import os
import argparse
import numpy as np
import torch
from torch import nn
import torch.optim as optim
from torch.utils.data import DataLoader
import torch_optimizer

from models import SAINT
from data import data_prep,DataSetCatCon
from augmentations import embed_data_mask
from utils import count_parameters, imputations_acc_justy


parser = argparse.ArgumentParser()

parser.add_argument('--dataset', default='portsmouth_balanced', type=str)
parser.add_argument('--embedding_size', default=32, type=int)
parser.add_argument('--transformer_depth', default=6, type=int)
parser.add_argument('--attention_heads', default=8, type=int)
parser.add_argument('--attention_dropout', default=0.1, type=float)
parser.add_argument('--ff_dropout', default=0.8, type=float)
parser.add_argument('--cont_embeddings', default='MLP', type=str,choices =
['MLP','Noemb','pos_singleMLP'])
parser.add_argument('--final_mlp_style', default='sep', type=str,choices =
['common','sep'])
parser.add_argument('--lr', default=0.001, type=float)
parser.add_argument('--batchsize', default=256, type=int)
parser.add_argument('--epochs', default=10, type=int)
parser.add_argument('--savemodelroot', default='./bestmodels', type=str)
parser.add_argument('--run_name', default='testrun', type=str)
parser.add_argument('--set_seed', default=42, type=int)
parser.add_argument('--train_mask_prob', default=0, type=float)
```

```
opt = parser.parse_args()

mask_params = {
    "mask_prob":opt.train_mask_prob,
    "avail_train_y": 0,
    "test_mask":opt.train_mask_prob
}

cat_dims, cat_idxs, con_idxs, X_train, y_train, X_valid, y_valid, X_test,
y_test, train_mean, train_std = data_prep(opt.dataset, opt.set_seed,
mask_params)
continuous_mean_std = np.array([train_mean, train_std]).astype(np.float32)

train_ds = DataSetCatCon(X_train, y_train, cat_idxs, continuous_mean_std)
trainloader = DataLoader(train_ds, batch_size=opt.batchsize, shuffle=True,
num_workers=4)

valid_ds = DataSetCatCon(X_valid, y_valid, cat_idxs, continuous_mean_std)
validloader = DataLoader(valid_ds, batch_size=opt.batchsize, shuffle=False,
num_workers=4)

test_ds = DataSetCatCon(X_test, y_test, cat_idxs, continuous_mean_std)
testloader = DataLoader(test_ds, batch_size=opt.batchsize, shuffle=False,
num_workers=4)

cat_dims_2, cat_idxs_2, con_idxs_2, X_train_2, y_train_2, X_valid_2,
y_valid_2, X_test_2, y_test_2, train_mean_2, train_std_2 =
data_prep('bedford', opt.set_seed, mask_params)
continuous_mean_std_2 = np.array([train_mean_2,
train_std_2]).astype(np.float32)

cat_dims_3, cat_idxs_3, con_idxs_3, X_train_3, y_train_3, X_valid_3,
y_valid_3, X_test_3, y_test_3, train_mean_3, train_std_3 =
data_prep('oxford', opt.set_seed, mask_params)
continuous_mean_std_3 = np.array([train_mean_3,
train_std_3]).astype(np.float32)

cat_dims_4, cat_idxs_4, con_idxs_4, X_train_4, y_train_4, X_valid_4,
y_valid_4, X_test_4, y_test_4, train_mean_4, train_std_4 =
data_prep('birmingham', opt.set_seed, mask_params)
continuous_mean_std_4 = np.array([train_mean_4,
train_std_4]).astype(np.float32)

test_ds_2 = DataSetCatCon(X_test_2, y_test_2, cat_idxs_2,
continuous_mean_std_2)
testloader_2 = DataLoader(test_ds_2, batch_size=opt.batchsize, shuffle=False,
num_workers=4)

test_ds_3 = DataSetCatCon(X_test_3, y_test_3, cat_idxs_3,
continuous_mean_std_3)
testloader_3 = DataLoader(test_ds_3, batch_size=opt.batchsize, shuffle=False,
num_workers=4)

test_ds_4 = DataSetCatCon(X_test_4, y_test_4, cat_idxs_4,
continuous_mean_std_4)
testloader_4 = DataLoader(test_ds_4, batch_size=opt.batchsize, shuffle=False,
num_workers=4)
```

```
cat_dims = np.append(np.array(cat_dims),np.array([2])).astype(int)

model = SAINT(
categories = tuple(cat_dims),
num_continuous = len(con_idxs),
dim = opt.embedding_size,
depth = opt.transformer_depth,
heads = opt.attention_heads,
dim_out = 1,
mlp_hidden_mults = (4, 2),
attn_dropout = opt.attention_dropout,
ff_dropout = opt.ff_dropout,
cont_embeddings = opt.cont_embeddings,
final_mlp_style = opt.final_mlp_style,
y_dim = 2
)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)

criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.AdamW(model.parameters(),lr=opt.lr, weight_decay=0.0001)
optimizer = torch_optimizer.Lookahead(optimizer, k=5, alpha=0.5)

best_valid_auroc = 0
modelsave_path = os.path.join(os.getcwd(), opt.savemodelroot, opt.dataset,
opt.run_name)
os.makedirs(modelsave_path, exist_ok=True)

for epoch in range(opt.epochs):
    model.train()
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        optimizer.zero_grad()
        x_categ, x_cont, cat_mask, con_mask = data[0].to(device),
data[1].to(device),data[2].to(device),data[3].to(device)
        _ , x_categ_enc, x_cont_enc = embed_data_mask(x_categ, x_cont,
cat_mask, con_mask,model, False)
        reps = model.transformer(x_categ_enc, x_cont_enc)
        y_reps = reps[:,len(cat_dims)-1,:]
        y_outs = model.mlpfory(y_reps)
        loss = criterion(y_outs,x_categ[:,len(cat_dims)-1])
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    if epoch % 5 == 0:
        model.eval()
        with torch.no_grad():
            accuracy, auroc = imputations_acc_justy(model, validloader,
device)

            print('[EPOCH %d] VALID AUROC: %.3f' %
                  (epoch + 1, auroc))
```

```
            if auroc > best_valid_auroc:
                best_valid_auroc = auroc
                torch.save(model.state_dict(), '%s/bestmodel.pth' %
(modelsave_path))
        model.train()

Portsmouth_test_accuracy, Portsmouth_test_auroc =
imputations_acc_justy(model, testloader, device)
Bedford_valid_accuracy, Bedford_valid_auroc = imputations_acc_justy(model,
testloader_2, device)
Oxford_valid_accuracy, Oxford_valid_auroc = imputations_acc_justy(model,
testloader_3, device)
Birmingham_valid_accuracy, Birmingham_valid_auroc =
imputations_acc_justy(model, testloader_4, device)
```

Feature interaction-based models

Wide and Deep, DeepFM, DCN, xDeepFM and PNN implementations in the DeepTables library (DeepTables, 2021) were used. These implementations were extended with weight decay (TensorFlow, 2021c), dropout (TensorFlow, 2021b), snapshot ensembles (Majumdar, 2021), batch normalisation (TensorFlow, 2021a), stochastic weight averaging (TensorFlow Addons, 2021b), and Lookahead (TensorFlow Addons, 2021a), using the code below.

The following code was used to create the FM layer in DeepFM, cross layer in cross network of DCN, CIN layer in CIN network of xDeepFM and inner and outer product layer in PNN:

```
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense, Dropout,
BatchNormalization, Activation, Concatenate, Flatten, Input, Embedding,
Lambda, Add, Conv2D, MaxPooling2D, SpatialDropout1D
from tensorflow.keras import backend as K
from tensorflow.keras import initializers, regularizers, losses, constraints


class FM(Layer):
    def __init__(self, **kwargs):
        super(FM, self).__init__(**kwargs)

    def call(self, x, **kwargs):
        if K.ndim(x) != 3:
            raise ValueError(f'Wrong dimensions of inputs, expected 3 but
input {K.ndim(x)}.')
        square_of_sum = tf.square(tf.reduce_sum(x, axis=1, keepdims=True))
```

```
        sum_of_square = tf.reduce_sum(x * x, axis=1, keepdims=True)
        cross = square_of_sum - sum_of_square
        cross = 0.5 * tf.reduce_sum(cross, axis=2, keepdims=False)

        return cross


class Cross(Layer):
    def __init__(self, params, **kwargs):
        self.params = params
        self.num_cross_layer = params.get('num_cross_layer', 6)
        super(Cross, self).__init__(**kwargs)

    def build(self, input):
        num_dims = input[-1]
        self.kernels = []
        self.bias = []
        for i in range(self.num_cross_layer):
            self.kernels.append(
                self.add_weight(name='kernels_' + str(i), shape=(num_dims,
1), initializer='glorot_uniform',
regularizer=tf.keras.regularizers.L2(0.0001), trainable=True))
            self.bias.append(
                self.add_weight(name='bias_' + str(i), shape=(num_dims, 1),
initializer='zeros', trainable=True))

    def call(self, x, **kwargs):
        if K.ndim(x) != 2:
            raise ValueError(f'Wrong dimensions of x, expected 2 but input
{K.ndim(x)}.')
        x_f = tf.expand_dims(x, axis=-1)
        x_n = x_f
        for i in range(self.num_cross_layer):
            x_n = tf.matmul(x_f, tf.tensordot(x_n, self.kernels[i], axes=(1,
0))) + x_n + self.bias[i]
            x_n = Dropout(0.1)(x_n)
        x_n = tf.reshape(x_n, (-1, x_f.shape[1]))

        return x_n

    def get_config(self, ):
        config = {'params': self.params}
        base_config = super(Cross, self).get_config()

        return dict(list(base_config.items()) + list(config.items()))


class CIN(Layer):
    def __init__(self, params, **kwargs):
        self.params = params
        self.cross_layer_size = params.get('cross_layer_size', (100, 100,
100,))
        self.activation = params.get('activation', 'relu')
        self.use_residual = params.get('use_residual', False)
        self.use_bias = params.get('use_bias', False)
        self.direct = params.get('direct', False)
        self.reduce_D = params.get('reduce_D', False)
```

```
        if len(self.cross_layer_size) == 0:
            raise ValueError(
                "cross_layer_size must be a list(tuple) of length greater
than 1")
        super(CIN, self).__init__(**kwargs)

    def build(self, input_shape):
        if len(input_shape) != 3:
            raise ValueError(
                "Unexpected inputs dimensions %d, expect to be 3 dimensions"
% (len(input_shape)))

        self.field_nums = [int(input_shape[1])]
        embed_dim = input_shape[-1]
        self.f_ = []
        self.f0_ = []
        self.f__ = []
        self.bias = []
        for i, layer_size in enumerate(self.cross_layer_size):
            if self.reduce_D:
                self.f0_.append(
                    self.add_weight(name=f'f0_{i}', shape=[1, layer_size,
self.field_nums[0], embed_dim], dtype=tf.float32, initializer='he_uniform'))
                self.f__.append(self.add_weight(name=f'f__{i}', shape=[1,
layer_size, embed_dim, self.field_nums[-1]], dtype=tf.float32,
initializer='he_uniform', regularizer=tf.keras.regularizers.L2(0.0001)))
            else:
                self.f_.append(self.add_weight(name=f'f_{i}', shape=[1,
self.field_nums[-1] * self.field_nums[0], layer_size], dtype=tf.float32,
initializer='he_uniform', regularizer=tf.keras.regularizers.L2(0.0001)))
            if self.use_bias:
                self.bias.append(self.add_weight(name=f'bias{i}',
shape=[layer_size], dtype=tf.float32, initializer='zeros'))

            if self.direct:
                self.field_nums.append(layer_size)
            else:
                if i != len(self.cross_layer_size) - 1 and layer_size % 2 >
0:
                    raise ValueError(
                        "cross_layer_size must be even number except for the
last layer when direct=True")
                self.field_nums.append(layer_size // 2)
        self.activation_layers = [Activation(self.activation) for _ in
self.cross_layer_size]
        if self.use_residual:
            self.exFM_out0 = Dense(self.cross_layer_size[-1],
activation=self.activation, kernel_initializer='he_uniform')
            self.exFM_out = Dense(1, activation=None)
        else:
            self.exFM_out = Dense(1, activation=None)
        super(CIN, self).build(input_shape)

    def call(self, x, **kwargs):
        if K.ndim(x) != 3:
            raise ValueError(f'Wrong dimensions of inputs, expected 3 but
```

```
input {K.ndim(x)}.')

        dim = int(x.get_shape()[-1])
        hidden_nn_layers = [x]
        final_result = []

        split_tensor0 = tf.split(hidden_nn_layers[0], dim * [1], 2)
        for idx, layer_size in enumerate(self.cross_layer_size):
            split_tensor = tf.split(hidden_nn_layers[-1], dim * [1], 2)
            dot_result_m = tf.matmul(split_tensor0, split_tensor,
transpose_b=True)
            dot_result_o = tf.reshape(dot_result_m, shape=[dim, -1,
self.field_nums[0] * self.field_nums[idx]])
            dot_result = tf.transpose(dot_result_o, perm=[1, 0, 2])

            if self.reduce_D:
                f0_ = self.f0_[idx]
                f__ = self.f__[idx]
                f_m = tf.matmul(f0_, f__)
                f_o = tf.reshape(f_m, shape=[1, layer_size,
self.field_nums[0] * self.field_nums[idx]])
                filters = tf.transpose(f_o, perm=[0, 2, 1])
            else:
                filters = self.f_[idx]
            curr_out = tf.nn.conv1d(dot_result, filters=filters, stride=1,
padding='VALID')
            if self.use_bias:
                curr_out = tf.nn.bias_add(curr_out, self.bias[idx])

            curr_out = self.activation_layers[idx](curr_out)
            curr_out = Dropout(0.1)(curr_out)
            curr_out = tf.transpose(curr_out, perm=[0, 2, 1])

            if self.direct:
                direct_connect = curr_out
                next_hidden = curr_out
            else:
                if idx != len(self.cross_layer_size) - 1:
                    next_hidden, direct_connect = tf.split(curr_out, 2 *
[layer_size // 2], 1)
                else:
                    direct_connect = curr_out
                    next_hidden = 0

            final_result.append(direct_connect)
            hidden_nn_layers.append(next_hidden)

        result = tf.concat(final_result, axis=1)
        result = tf.reduce_sum(result, -1)

        if self.use_residual:
            exFM_out0 = self.exFM_out0(result)
            exFM_in = tf.concat([exFM_out0, result], axis=1)
            exFM_out = self.exFM_out(exFM_in)

        else:
            exFM_out = self.exFM_out(result)
```

```
        return exFM_out

    def get_config(self, ):
        config = {'params': self.params}
        base_config = super(CIN, self).get_config()

        return dict(list(base_config.items()) + list(config.items()))


class InnerProduct(Layer):
    def __init__(self, **kwargs):
        super(InnerProduct, self).__init__(**kwargs)

    def call(self, x, **kwargs):
        if K.ndim(x[0]) != 3:
            raise ValueError(f'Wrong dimensions of inputs, expected 3 but
input {K.ndim(x[0])}.')
        num_inputs = len(x)
        num_pairs = int(num_inputs * (num_inputs - 1) / 2)
        row = []
        col = []
        for i in range(num_inputs - 1):
            for j in range(i + 1, num_inputs):
                row.append(i)
                col.append(j)
        p = tf.concat([x[i] for i in row], axis=1)
        q = tf.concat([x[j] for j in col], axis=1)
        ip = tf.reshape(tf.reduce_sum(p * q, [-1]), [-1, num_pairs])

        return ip

    def get_config(self):
        return super(InnerProduct, self).get_config()


class OuterProduct(Layer):
    def __init__(self, params, **kwargs):
        self.params = params
        self.kernel_type = params.get('outer_product_kernel_type', 'mat')
        if self.kernel_type not in ['mat', 'vec', 'num']:
            raise ValueError("kernel_type must be mat,vec or num")
        super(OuterProduct, self).__init__(**kwargs)

    def build(self, input):
        num_inputs = len(input)
        num_pairs = int(num_inputs * (num_inputs - 1) / 2)
        embed_size = input[0][-1]
        if self.kernel_type == 'mat':
            self.kernel = self.add_weight(shape=(embed_size, num_pairs,
embed_size), name='kernel', regularizer=tf.keras.regularizers.L2(0.0001))
        elif self.kernel_type == 'vec':
            self.kernel = self.add_weight(shape=(num_pairs, embed_size,),
name='kernel', regularizer=tf.keras.regularizers.L2(0.0001))
        elif self.kernel_type == 'num':
            self.kernel = self.add_weight(shape=(num_pairs, 1),
name='kernel', regularizer=tf.keras.regularizers.L2(0.0001))
```

```
        super(OuterProduct, self).build(input)

    def call(self, x, **kwargs):
        if K.ndim(x[0]) != 3:
            raise ValueError(f'Wrong dimensions of inputs, expected 3 but
input {K.ndim(x[0])}.')
        row = []
        col = []
        num_inputs = len(x)
        for i in range(num_inputs - 1):
            for j in range(i + 1, num_inputs):
                row.append(i)
                col.append(j)
        p = tf.concat([x[i] for i in row], axis=1)
        q = tf.concat([x[i] for i in col], axis=1)

        if self.kernel_type == 'mat':
            p = tf.expand_dims(p, 1)
            kp = tf.reduce_sum(
                tf.multiply(
                    tf.transpose(
                        tf.reduce_sum(
                            tf.multiply(
                                p, self.kernel),
                            -1),
                        [0, 2, 1]),
                    q),
                -1)
        else:
            k = tf.expand_dims(self.kernel, 0)
            kp = tf.reduce_sum(p * q * k, -1)

        return kp

    def get_config(self, ):
        config = {'params': self.params}
        base_config = super(OuterProduct, self).get_config()

        return dict(list(base_config.items()) + list(config.items()))
```

The following code was used to create the linear model in Wide and Deep, DeepFM and xDeepFM, deep neural network in all models, cross network in DCN, CIN network in xDeepFM and product based neural network in PNN:

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Concatenate, Flatten,
BatchNormalization, Activation, Dropout

from . import layers
```

```
WideDeep = ['linear', 'dnn_nets']
DeepFM = ['linear', 'fm_nets', 'dnn_nets']
DCN = ['dcn_nets']
xDeepFM = ['linear', 'cin_nets', 'dnn_nets']
PNN = ['pnn_nets']


def _concat_embeddings(embeddings, concat_layer_name):
    if embeddings is not None:
        len_embeddings = len(embeddings)
        if len_embeddings > 1:
            return Concatenate(axis=1, name=concat_layer_name)(embeddings)
        elif len_embeddings == 1:
            return embeddings[0]
        else:
            return None
    else:
        return None


def linear(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense,
config, model_desc):
    x = None
    x_emb = None
    concat_embeddings = _concat_embeddings(embeddings,
'concat_linear_embedding')
    if concat_embeddings is not None:
        x_emb = tf.reduce_sum(concat_embeddings, axis=-1,
name='linear_reduce_sum')

    if x_emb is not None and dense_layer is not None:
        x = Concatenate(name='concat_linear_emb_dense')([x_emb, dense_layer])
        x = BatchNormalization(name='bn_linear_emb_dense')(x)
    elif x_emb is not None:
        x = x_emb
    elif dense_layer is not None:
        x = dense_layer
    else:
        raise ValueError('No input layer exists.')

    input_shape = x.shape
    x = Dense(1, activation=None, use_bias=False, name='linear_logit',
kernel_regularizer=tf.keras.regularizers.L2(0.0001))(x)
    model_desc.add_net('linear', input_shape, x.shape)

    return x


def dnn(x, params, cellname='dnn'):
    custom_dnn_fn = params.get('custom_dnn_fn')
    if custom_dnn_fn is not None:
        return custom_dnn_fn(x, params, cellname + '_custom')

    hidden_units = params.get('hidden_units', ((1024, 0.5, True), (512, 0.5,
True) , (256, 0.5, True)))
    activation = params.get('activation', 'relu')
```

```
    kernel_initializer = params.get('kernel_initializer', 'he_uniform')
    kernel_regularizer = params.get('kernel_regularizer',
tf.keras.regularizers.L2(0.0001))
    activity_regularizer = params.get('activity_regularizer')

    if len(hidden_units) <= 0:
        raise ValueError('[hidden_units] must be a list of
tuple([units],[dropout_rate],[use_bn]) and at least one tuple.')

    index = 1
    for units, dropout, batch_norm in hidden_units:
        x = Dense(units, use_bias=not batch_norm,
name=f'{cellname}_dense_{index}', kernel_initializer=kernel_initializer,
kernel_regularizer=kernel_regularizer,
activity_regularizer=activity_regularizer,
                  )(x)

        if batch_norm:
            x = BatchNormalization(name=f'{cellname}_bn_{index}')(x)
        x = Activation(activation=activation,
name=f'{cellname}_activation_{index}')(x)

        if dropout > 0:
            x = Dropout(dropout, name=f'{cellname}_dropout_{index}')(x)
        index += 1

    return x

def dnn_nets(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense,
config, model_desc):
    x_dnn = dnn(concat_emb_dense, config.dnn_params)
    model_desc.add_net('dnn', concat_emb_dense.shape, x_dnn.shape)

    return x_dnn


def fm_nets(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense,
config, model_desc):
    concat_embeddings_layer = _concat_embeddings(embeddings,
'concat_fm_embedding')

    if concat_embeddings_layer is None :
        model_desc.add_net('fm', (None), (None))
        return None

    fm_output = layers.FM(name='fm_layer')(concat_embeddings_layer)
    model_desc.add_net('fm', concat_embeddings_layer.shape, fm_output.shape)

    return fm_output


def dcn_nets(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense,
config, model_desc):
    x = concat_emb_dense
    cross_out = layers.Cross(params=config.cross_params,
name='dcn_cross_layer')(x)
    model_desc.add_net('dcn-widecross', x.shape, cross_out.shape)
```

```
    dnn_out = dnn(x, config.dnn_params, cellname='dcn')
    model_desc.add_net('dcn-dnn2', x.shape, dnn_out.shape)

    stack_out = Concatenate(name='concat_cross_dnn')([cross_out, dnn_out])
    model_desc.add_net('dcn', x.shape, stack_out.shape)

    return stack_out


def cin_nets(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense,
config, model_desc):
    cin_concat = _concat_embeddings(embeddings, 'concat_cin_embedding')
    if cin_concat is None:
        model_desc.add_net('cin', (None), (None))
        return None

    cin_output = layers.CIN(params=config.cin_params)(cin_concat)
    model_desc.add_net('cin', cin_concat.shape, cin_output.shape)

    return cin_output


def pnn_nets(embeddings, flatten_emb_layer, dense_layer, concat_emb_dense,
config, model_desc):
    if embeddings is None or len(embeddings) < 2:
        return None

    ip = layers.InnerProduct(name='pnn_inner_product_layer')(embeddings)
    model_desc.add_net('pnn-inner_product', f'list({len(embeddings)})',
ip.shape)

    op = layers.OuterProduct(params=config.pnn_params,
name='pnn_outer_product_layer')(embeddings)
    model_desc.add_net('pnn-outer_product', f'list({len(embeddings)})',
op.shape)

    concat_all = Concatenate(name='concat_pnn_all')([ip, op,
concat_emb_dense])
    x_dnn = dnn(concat_all, config.dnn_params, cellname='pnn')
    model_desc.add_net('pnn-dnn', concat_all.shape, x_dnn.shape)

    return x_dnn
```

The following code was used to define the functions for the models, including training (with stochastic weight averaging and Lookahead):

```
from typing import List
import collections
from collections import OrderedDict
```

```
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Dense, Concatenate, Flatten, Input, Add,
BatchNormalization, Dropout
from tensorflow.keras.models import Model, load_model, save_model
from tensorflow.keras.utils import to_categorical
import tensorflow_addons as tfa

from . import deepnets
from .layers import MultiColumnEmbedding, dt_custom_objects,
VarLenColumnEmbedding
from deeptables.models.metainfo import CategoricalColumn
from .metainfo import VarLenCategoricalColumn
from ..utils import dt_logging, consts, gpu
import numpy as np


logger = dt_logging.get_logger()

class DeepModel:
    def __init__(self,
                 task,
                 num_classes,
                 config,
                 categorical_columns,
                 continuous_columns,
                 model_file=None,
                 var_categorical_len_columns=None, ):

        if config.gpu_usage_strategy == consts.GPU_USAGE_STRATEGY_GROWTH:
            gpu.set_memory_growth()
        self.model_desc = ModelDesc()
        self.categorical_columns = categorical_columns
        self.continuous_columns = continuous_columns
        self.var_len_categorical_columns = var_categorical_len_columns
        self.task = task
        self.num_classes = num_classes
        self.config = config
        self.model_file = model_file
        self.model = None
        if model_file is not None:
            self.model = load_model(model_file, dt_custom_objects)

    def fit(self, X=None, y=None, batch_size=128, epochs=1, verbose=1,
callbacks=None, validation_split=0.2, validation_data=None, shuffle=True,
class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None,
validation_steps=None, validation_freq=1, max_queue_size=10, workers=1,
use_multiprocessing=False):
        if validation_data is None:
            X, X_val, y, y_val = train_test_split(X, y,
test_size=validation_split, random_state=42)
        else:
            if len(validation_data) != 2:
                raise ValueError(f'Unexpected validation_data length,
expected 2 but {len(validation_data)}.')
```

```
            X_val, y_val = validation_data[0], validation_data[1]

        X_train_input = self.__get_model_input(X)
        X_val_input = self.__get_model_input(X_val)
        y = np.array(y)
        y_val = np.array(y_val)
        if self.task == consts.TASK_MULTICLASS:
            y = to_categorical(y, num_classes=self.num_classes)
            y_val = to_categorical(y_val, num_classes=self.num_classes)

        if self.config.distribute_strategy is not None:
            from tensorflow.python.distribute.distribute_lib import Strategy
            if not isinstance(self.config.distribute_strategy, Strategy):
                raise ValueError(f'[distribute_strategy] in ModelConfig must
be an instance of tensorflow.python.distribute.distribute_lib.Strategy.')
            with self.config.distribute_strategy.scope():
                self.model = self.__build_model(task=self.task,
                                                 num_classes=self.num_classes,
                                                 nets=self.config.nets,
categorical_columns=self.categorical_columns,
continuous_columns=self.continuous_columns,
var_len_categorical_columns=self.var_len_categorical_columns,
config=self.config)
        else:
            self.model = self.__build_model(task=self.task,
                                             num_classes=self.num_classes,
                                             nets=self.config.nets,
categorical_columns=self.categorical_columns,
continuous_columns=self.continuous_columns,
var_len_categorical_columns=self.var_len_categorical_columns,
config=self.config)

        logger.info(f'training...')
        history = self.model.fit(X_train_input,
                                 y,
                                 batch_size=batch_size,
                                 epochs=epochs,
                                 verbose=verbose,
                                 validation_data=(X_val_input, y_val),
                                 shuffle=shuffle,
                                 callbacks=callbacks,
                                 class_weight=class_weight,
                                 sample_weight=sample_weight,
                                 initial_epoch=initial_epoch,
                                 steps_per_epoch=steps_per_epoch,
                                 validation_steps=validation_steps,
                                 validation_freq=validation_freq,
                                 max_queue_size=max_queue_size,
                                 workers=workers,
                                 use_multiprocessing=use_multiprocessing,
                                 )
        logger.info(f'Training finished.')
        history.history = IgnoreCaseDict(history.history)

        return history

    def predict(self, X, batch_size=128, verbose=0):
```

```
        return self.__predict(self.model, X, batch_size=batch_size,
verbose=verbose)

    def __predict(self, model, X, batch_size=128, verbose=0):
        logger.info("Performing predictions...")
        X_input = self.__get_model_input(X)

        return model.predict(X_input, batch_size=batch_size, verbose=verbose)

    def apply(self, X, output_layers=[], concat_outputs=False,
batch_size=128,
              verbose=0, transformer=None):
        model = self.__buld_proxy_model(self.model, output_layers,
concat_outputs)
        output = self.__predict(model, X, batch_size=batch_size,
verbose=verbose)

        if transformer is None:
            return output
        else:
            if isinstance(output, list):
                output_t = []
                for i, x_o in enumerate(output):
                    if len(x_o.shape) > 2:
                        x_o = x_o.reshape((x_o.shape[0], -1))
                    print(f'Performing transformation on [{output_layers[i]}]
by "{str(transformer)}", input shape:{x_o.shape}.')
                    output_t.append(transformer.fit_transform(x_o))
                return output_t
            else:
                return transformer.fit_transform(output)

    def evaluate(self, X_test, y_test, batch_size=128, verbose=0,
return_dict=True):
        logger.info("Performing evaluation...")
        X_input = self.__get_model_input(X_test)
        y_t = np.array(y_test)

        if self.task == consts.TASK_MULTICLASS:
            y_t = to_categorical(y_t, num_classes=self.num_classes)

        result = self.model.evaluate(X_input, y_t, batch_size=batch_size,
verbose=verbose)
        if return_dict:
            result = {k: v for k, v in zip(self.model.metrics_names, result)}
            return IgnoreCaseDict(result)
        else:
            return result

    def save(self, filepath):
        save_model(self.model, filepath, save_format='h5')

    def release(self):
        del self.model
        self.model = None
        K.clear_session()
```

```
    def __get_model_input(self, X):
        train_data = {}

        if self.categorical_columns is not None and
len(self.categorical_columns) > 0:
            train_data['input_categorical_vars_all'] = X[[c.name for c in
self.categorical_columns]].values.astype(consts.DATATYPE_TENSOR_FLOAT)

        if self.continuous_columns is not None and
len(self.continuous_columns) > 0:
            for c in self.continuous_columns:
                train_data[c.name] =
X[c.column_names].values.astype(consts.DATATYPE_TENSOR_FLOAT)

        if self.var_len_categorical_columns is not None and
len(self.var_len_categorical_columns) > 0:
            for col in self.var_len_categorical_columns:
                train_data[col.name] = np.array(X[col.name].tolist())

        return train_data

    def __build_proxy_model(self, model, output_layers=[],
concat_output=False):
        model.trainable = False
        if len(output_layers) <= 0:
            raise ValueError('"output_layers" at least 1 element.')
        outputs = [model.get_layer(l).output for l in output_layers]

        if len(outputs) <= 0:
            raise ValueError(f'No layer found in the model:{output_layers}')
        if len(outputs) > 1 and concat_output:
            outputs = Concatenate()(outputs)
        proxy = Model(inputs=model.input, outputs=outputs)
        proxy.compile(optimizer=model.optimizer, loss=model.loss)

        return proxy

    def __build_model(self, task, num_classes, nets, categorical_columns,
continuous_columns, var_len_categorical_columns, config):
        logger.info(f'Building model...')
        self.model_desc = ModelDesc()
        categorical_inputs, continuous_inputs, var_len_categorical_inputs =
self.__build_inputs(categorical_columns, continuous_columns,
var_len_categorical_columns)
        embeddings = self.__build_embeddings(categorical_columns,
categorical_inputs, var_len_categorical_columns, var_len_categorical_inputs,
config.embedding_dropout)
        dense_layer = self.__build_denses(continuous_columns,
continuous_inputs, config.dense_dropout)

        flatten_emb_layer = None
        if len(embeddings) > 0:
            if len(embeddings) == 1:
                flatten_emb_layer =
Flatten(name='flatten_embeddings')(embeddings[0])
            else:
                flatten_emb_layer = Flatten(name='flatten_embeddings')(
```

```
                    Concatenate(name='concat_embeddings_axis_0')(embeddings))

        self.model_desc.nets = nets
        self.model_desc.stacking = config.stacking_op
        concat_emb_dense = self.__concat_emb_dense(flatten_emb_layer,
dense_layer)

        outs = {}
        for net in nets:
            logit = deepnets.get(net)
            out = logit(embeddings, flatten_emb_layer, dense_layer,
concat_emb_dense, self.config, self.model_desc)
            if out is not None:
                outs[net] = out
        if len(outs) > 1:
            logits = []
            for name, out in outs.items():
                if len(out.shape) > 2:
                    out = Flatten(name=f'flatten_{name}_out')(out)
                if out.shape[-1] > 1:
                    logit = Dense(1, use_bias=False, activation=None,
name=f'dense_logit_{name}')(out)
                else:
                    logit = out
                logits.append(logit)
            if config.stacking_op == consts.STACKING_OP_ADD:
                x = Add(name='add_logits')(logits)
            elif config.stacking_op == consts.STACKING_OP_CONCAT:
                x = Concatenate(name='concat_logits')(logits)
            else:
                raise ValueError(f'Unsupported
stacking_op:{config.stacking_op}.')
        elif (len(outs) == 1):
            name, out = outs.popitem()

            if len(out.shape) > 2:
                out = Flatten(name=f'flatten_{name}_out')(out)
            x = out
        else:
            raise ValueError(f'Unexcepted logit output.{outs}')
        all_inputs = list(categorical_inputs.values()) +
list(var_len_categorical_inputs.values()) + list(continuous_inputs.values())
        output = self.__output_layer(x, task, num_classes,
use_bias=self.config.output_use_bias)
        model = Model(inputs=all_inputs, outputs=output)
        model = self.__compile_model(model, task, num_classes,
config.optimizer, config.loss, config.metrics)
        print(self.model_desc)

        return model

    def __compile_model(self, model, task, num_classes, optimizer, loss,
metrics):
        if optimizer == 'auto':
            optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
            optimizer = tfa.optimizers.Lookahead(optimizer, sync_period=5,
slow_step_size=0.5)
```

```
            optimizer = tfa.optimizers.SWA(optimizer, start_averaging=10,
average_period=5)

        if loss == 'auto':
            if task == consts.TASK_BINARY or task == consts.TASK_MULTILABEL:
                loss = 'binary_crossentropy'
            elif task == consts.TASK_REGRESSION:
                loss = 'mse'
            elif task == consts.TASK_MULTICLASS:
                if num_classes == 2:
                    loss = 'binary_crossentropy'
                else:
                    loss = 'categorical_crossentropy'
        self.model_desc.optimizer = optimizer
        self.model_desc.loss = loss
        model.compile(optimizer, loss, metrics=metrics)

        return model

    def __concat_emb_dense(self, flatten_emb_layer, dense_layer):
        x = None
        if flatten_emb_layer is not None and dense_layer is not None:
            x =
Concatenate(name='concat_embedding_dense')([flatten_emb_layer, dense_layer])
        elif flatten_emb_layer is not None:
            x = flatten_emb_layer
        elif dense_layer is not None:
            x = dense_layer
        else:
            raise ValueError('No input layer exists.')

        x = BatchNormalization(name='bn_concat_emb_dense')(x)
        self.model_desc.set_concat_embed_dense(x.shape)

        return x

    def __build_inputs(self, categorical_columns: List[CategoricalColumn],
continuous_columns, var_len_categorical_columns:
List[VarLenCategoricalColumn]=None):
        categorical_inputs = OrderedDict()
        var_len_categorical_inputs = OrderedDict()
        continuous_inputs = OrderedDict()

        if categorical_columns is not None and  len(categorical_columns) > 0:
            categorical_inputs['all_categorical_vars'] =
Input(shape=(len(categorical_columns),), name='input_categorical_vars_all')
            self.model_desc.add_input('all_categorical_vars',
len(categorical_columns))

        if var_len_categorical_columns is not None and
len(var_len_categorical_columns) > 0:
            for col in var_len_categorical_columns:
                var_len_categorical_inputs[col.name] =
Input(shape=(col.max_elements_length, ), name=col.name)
                self.model_desc.add_input(col.name, col.max_elements_length)

        for column in continuous_columns:
```

```
            continuous_inputs[column.name] = Input(shape=(column.input_dim,),
name=column.name, dtype=column.dtype)
            self.model_desc.add_input(column.name, column.input_dim)

        return categorical_inputs, continuous_inputs,
var_len_categorical_inputs

    def __construct_var_len_embedding(self, column: VarLenCategoricalColumn,
var_len_inputs, embedding_dropout):
        input_layer = var_len_inputs[column.name]
        var_len_embeddings =
VarLenColumnEmbedding(pooling_strategy=column.pooling_strategy,
input_dim=column.vocabulary_size, output_dim=column.embeddings_output_dim,
dropout_rate=embedding_dropout, name=consts.LAYER_PREFIX_EMBEDDING +
column.name, embeddings_initializer=self.config.embeddings_initializer,
embeddings_regularizer=self.config.embeddings_regularizer,
activity_regularizer=self.config.embeddings_activity_regularizer)(input_layer
)

        return var_len_embeddings

    def __build_embeddings(self, categorical_columns, categorical_inputs,
var_len_categorical_columns: List[VarLenCategoricalColumn], var_len_inputs,
embedding_dropout):
        if 'all_categorical_vars' in categorical_inputs:
            input_layer = categorical_inputs['all_categorical_vars']
            input_dims = [column.vocabulary_size for column in
categorical_columns]
            output_dims = [column.embeddings_output_dim for column in
categorical_columns]
            embeddings = MultiColumnEmbedding(input_dims, output_dims,
embedding_dropout, name=consts.LAYER_PREFIX_EMBEDDING +
'categorical_vars_all',
embeddings_initializer=self.config.embeddings_initializer,
embeddings_regularizer=self.config.embeddings_regularizer,
activity_regularizer=self.config.embeddings_activity_regularizer)(input_layer
)
            self.model_desc.set_embeddings(input_dims, output_dims,
embedding_dropout)
        else:
            embeddings = []

        if var_len_categorical_columns is not None and
len(var_len_categorical_columns) > 0:
            for c in var_len_categorical_columns:
                var_len_embedding = self.__construct_var_len_embedding(c,
var_len_inputs, embedding_dropout)
                embeddings.append(var_len_embedding)

        return embeddings

    def __build_denses(self, continuous_columns, continuous_inputs,
dense_dropout, use_batchnormalization=True):
        dense_layer = None
        if continuous_inputs:
            if len(continuous_inputs) > 1:
                dense_layer =
```

```
Concatenate(name=consts.LAYER_NAME_CONCAT_CONT_INPUTS)(list(continuous_inputs
.values()))
            else:
                dense_layer = list(continuous_inputs.values())[0]
        if dense_dropout > 0:
            dense_layer = Dropout(dense_dropout,
name='dropout_dense_input')(dense_layer)
        if use_batchnormalization:
            dense_layer =
BatchNormalization(name=consts.LAYER_NAME_BN_DENSE_ALL)(dense_layer)
        self.model_desc.set_dense(dense_dropout, use_batchnormalization)

        return dense_layer

    def __output_layer(self, x, task, num_classes, use_bias=True):
        if task == consts.TASK_BINARY:
            activation = 'sigmoid'
            output_dim = 1
        elif task == consts.TASK_REGRESSION:
            activation = None
            output_dim = 1
        elif task == consts.TASK_MULTICLASS:
            if num_classes:
                activation = 'softmax'
                output_dim = num_classes
            else:
                raise ValueError('"config.multiclass_classes" value must be
provided for multi-class task.')
        elif task == consts.TASK_MULTILABEL:
            activation = 'sigmoid'
            output_dim = num_classes
        else:
            raise ValueError(f'Unknown task type:{task}')

        output = Dense(output_dim, activation=activation, name='task_output',
use_bias=use_bias)(x)
        self.model_desc.set_output(activation, output.shape, use_bias)

        return output
```

The following code was used for snapshot ensembling:

```
import os
import numpy as np

import keras.callbacks as callbacks
from keras.callbacks import Callback


class SnapshotModelCheckpoint(Callback):
    def __init__(self, num_epochs, num_snapshots, fn_prefix='Model'):
        super(SnapshotModelCheckpoint, self).__init__()
```

```
        self.check = num_epochs // num_snapshots
        self.fn_prefix = fn_prefix

    def on_epoch_end(self, epoch, logs={}):
        if epoch != 0 and (epoch + 1) % self.check == 0:
            filepath = self.fn_prefix + "-%d.h5" % ((epoch + 1) //
self.check)
            self.model.save_weights(filepath, overwrite=True)


class SnapshotCallbackBuilder:
    def __init__(self, num_epochs, num_snapshots, initial_lr=0.2):
        self.num_epochs = num_epochs
        self.num_snapshots = num_snapshots
        self.initial_lr = initial_lr

    def get_callbacks(self, model_prefix='Model_'):
        if not os.path.exists('weights/'):
            os.makedirs('weights/')

        callback_list = [callbacks.ModelCheckpoint("weights/%s-Best.h5" %
model_prefix, monitor="val_auc", save_best_only=True,
save_weights_only=True),
callbacks.LearningRateScheduler(schedule=self.cosine_annealing),
SnapshotModelCheckpoint(self.num_epochs, self.num_snapshots,
fn_prefix='weights/%s' % model_prefix)]

        return callback_list

    def cosine_annealing(self, t):
        cos = np.pi * (t % (self.num_epochs // self.num_snapshots)) /
(self.num_epochs // self.num_snapshots)
        lr = float(self.initial_lr / 2 * (np.cos(cos) + 1))

        return lr
```

The following code was used to train and test the models (example for Wide and Deep):

```
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras import layers

from deeptables.models.deeptable import DeepTable, ModelConfig
from deeptables.models import deepnets
from deeptables.models.deepnets import WideDeep, DeepFM, DCN, xDeepFM, PNN


data = pd.read_csv('portsmouth_balanced.csv')
valid_data = pd.read_csv('bedford.csv')
valid_data_2 = pd.read_csv('oxford.csv')
```

```
valid_data_3 = pd.read_csv('birmingham.csv')

features = ['Age', 'Gender', 'Ethnicity', 'Vital_Sign Heart Rate',
'Vital_Sign Respiratory Rate', 'Vital_Sign Systolic Blood Pressure',
'Vital_Sign Diastolic Blood Pressure', 'Vital_Sign Oxygen Saturation',
'Vital_Sign Temperature Tympanic', 'Blood_Test HAEMOGLOBIN', 'Blood_Test
HAEMATOCRIT', 'Blood_Test MEAN CELL VOL.', 'Blood_Test WHITE CELLS',
'Blood_Test NEUTROPHILS', 'Blood_Test LYMPHOCYTES', 'Blood_Test MONOCYTES',
'Blood_Test EOSINOPHILS', 'Blood_Test BASOPHILS', 'Blood_Test PLATELETS',
'Blood_Test SODIUM', 'Blood_Test POTASSIUM', 'Blood_Test UREA', 'Blood_Test
CREATININE', 'Blood_Test eGFR', 'Blood_Test ALT', 'Blood_Test
ALK.PHOSPHATASE', 'Blood_Test BILIRUBIN', 'Blood_Test ALBUMIN', 'Blood_Test
CRP']

y = data["Covid-19 Positive"]
X = data[features]

valid_y = valid_data["Covid-19 Positive"]
valid_X = valid_data[features]

valid_y_2 = valid_data_2["Covid-19 Positive"]
valid_X_2 = valid_data_2[features]

valid_y_3 = valid_data_3["Covid-19 Positive"]
valid_X_3 = valid_data_3[features]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

conf = ModelConfig(
    nets=WideDeep,
    categorical_columns=['Gender', 'Ethnicity'],
    pos_label=1,
    metrics=['AUC', 'accuracy'],
    embeddings_output_dim=32,
    embeddings_regularizer=None,
    dense_dropout=0,
    embedding_dropout=0,
    earlystopping_patience=10,
    dnn_params={'hidden_units': ((1024, 0.5, True), (512, 0.5, True), (256,
0.5, True)), "kernel_regularizer": tf.keras.regularizers.L2(0.0001)}
)

dt = DeepTable(config=conf)

model, history = dt.fit(X_train, y_train, epochs=100, callbacks=
SnapshotCallbackBuilder(100, 5, 0.2).get_callbacks(model_prefix='Model_'))

portsmouth_test_auc = dt.evaluate(X_test, y_test)
bedford_valid_auc = dt.evaluate(valid_X, valid_y)
oxford_valid_auc = dt.evaluate(valid_X_2, valid_y_2)
birmingham_valid_auc = dt.evaluate(valid_X_3, valid_y_3)
```

The following code was used to configure DeepFM, DCN, xDeepFM and PNN (the code to train and test these models is the same as Wide and Deep above):

```
conf = ModelConfig(
    nets=DeepFM,
    categorical_columns=['Gender', 'Ethnicity'],
    pos_label=1,
    metrics=['AUC', 'accuracy'],
    embeddings_output_dim=32,
    embeddings_regularizer=None,
    dense_dropout=0,
    embedding_dropout=0,
    earlystopping_patience=10,
    dnn_params={'hidden_units': ((400, 0.5, True), (400, 0.5, True), (400,
0.5, True)), "kernel_regularizer": tf.keras.regularizers.L2(0.0001)}
    )


conf = ModelConfig(
    nets=DCN,
    categorical_columns=['Gender', 'Ethnicity'],
    pos_label=1,
    metrics=['AUC', 'accuracy'],
    embeddings_output_dim=32,
    embeddings_regularizer=None,
    dense_dropout=0,
    embedding_dropout=0,
    earlystopping_patience=10,
    dnn_params={'hidden_units': ((1024, 0.5, True), (1024, 0.5, True), (1024,
0.5, True)), "kernel_regularizer": tf.keras.regularizers.L2(0.0001)},
    cross_params={'num_cross_layer': 6}
    )


conf = ModelConfig(
    nets=xDeepFM,
    categorical_columns=['Gender', 'Ethnicity'],
    pos_label=1,
    metrics=['AUC', 'accuracy'],
    embeddings_output_dim=32,
    embeddings_regularizer=None,
    dense_dropout=0,
    embedding_dropout=0,
    earlystopping_patience=10,
    dnn_params={'hidden_units': ((400, 0.5, True), (400, 0.5, True), (400,
0.5, True)), "kernel_regularizer": tf.keras.regularizers.L2(0.0001)},
    cin_params={'cross_layer_size': (100, 100, 100)}
    )


conf = ModelConfig(
    nets=PNN,
    categorical_columns=['Gender', 'Ethnicity'],
```

```
    pos_label=1,
    metrics=['AUC', 'accuracy'],
    embeddings_output_dim=32,
    embeddings_regularizer=None,
    dense_dropout=0,
    embedding_dropout=0,
    earlystopping_patience=10,
    dnn_params={'hidden_units': ((400, 0.5, True), (400, 0.5, True), (400,
0.5, True)), 'activation': "tanh", "kernel_regularizer":
tf.keras.regularizers.L2(0.0001)}
    )
```

Regularisation-based models

RLN

The RLN implementation of Shavitt (2018) was used. The implementation was extended with stochastic weight averaging (TensorFlow Addons, 2021b), using the code below.

The following code was used to create the RLN model:

```
import numpy as np
from pandas import DataFrame
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import BinaryCrossentropy
from keras.callbacks import Callback
from keras import backend as K
import tensorflow_addons as tfa


class RLNCallback(Callback):
    def __init__(self, layer, norm=1, avg_reg=-10, learning_rate=1e6):
        super(RLNCallback, self).__init__()
        self._kernel = layer.kernel
        self._prev_weights, self._weights, self._prev_regularization = [None]
* 3
        self._avg_reg = avg_reg
        self._shape = K.transpose(self._kernel).get_shape().as_list()
        self._lambdas = DataFrame(np.ones(self._shape) * self._avg_reg)
        self._lr = learning_rate
        assert norm in [1, 2]
        self.norm = norm

    def on_train_begin(self, logs=None):
        self._update_values()
```

```
    def on_batch_end(self, batch, logs=None):
        self._prev_weights = self._weights
        self._update_values()
        gradients = self._weights - self._prev_weights

        if self.norm == 1:
            norms_derivative = np.sign(self._weights)
        else:
            norms_derivative = self._weights * 2

        if self._prev_regularization is not None:
            lambda_gradients = gradients.multiply(self._prev_regularization)
            self._lambdas -= self._lr * lambda_gradients

            translation = (self._avg_reg - self._lambdas.mean().mean())
            self._lambdas += translation

        max_lambda_values = np.log(np.abs(self._weights /
norms_derivative)).fillna(np.inf)
        self._lambdas = self._lambdas.clip(upper=max_lambda_values)

        regularization = norms_derivative.multiply(np.exp(self._lambdas))
        self._weights -= regularization
        K.set_value(self._kernel, self._weights.values.T)
        self._prev_regularization = regularization

    def _update_values(self):
        self._weights = DataFrame(K.eval(self._kernel).T)


def base_model(layers=4, input_dim):
    assert layers > 1

    def build_fn():
        model = Sequential()
        dim = input_dim
        for width in np.exp(np.log(input_dim) * np.arange(layers - 1, 0, -1)
/ layers):
            width = int(np.round(width))
            model.add(Dense(width, input_dim=dim,
kernel_initializer='glorot_normal', activation='relu'))
            dim = width

        model.add(Dense(1, kernel_initializer='glorot_normal'))

        optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
        optimizer = tfa.optimizers.SWA(optimizer, start_averaging=50,
average_period=5)

        model.compile(optimizer=optimizer,
loss=BinaryCrossentropy(from_logits=True))

        return model

    return build_fn
```

```
def RLN(layers=4, input_dim, **rln_kwargs):
    def build_fn():
        model = base_model(layers=layers, input_dim=input_dim)()

        rln_callback = RLNCallback(model.layers[0], **rln_kwargs)

        orig_fit = model.fit

        def rln_fit(*args, **fit_kwargs):
            orig_callbacks = fit_kwargs.get('callbacks', [])
            rln_callbacks = orig_callbacks + [rln_callback]

            return orig_fit(*args, callbacks=rln_callbacks, **fit_kwargs)

        model.fit = rln_fit

        return model

    return build_fn
```

The following code was used to create the train and test the RLN model:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
from keras.wrappers.scikit_learn import KerasClassifier


data = pd.read_csv('portsmouth_balanced.csv')
data2 = pd.read_csv('bedford.csv')
data3 = pd.read_csv('oxford.csv')
data4 = pd.read_csv('birmingham.csv')

cleanup_gender = {"Gender": {"M": 0, "F": 1}}

data = data.replace(cleanup_gender)
data2 = data2.replace(cleanup_gender)
data3 = data3.replace(cleanup_gender)
data4 = data4.replace(cleanup_gender)

one_hot = pd.get_dummies(data['Ethnicity'])
data = data.drop('Ethnicity', axis = 1)
data = data.join(one_hot)

one_hot2 = pd.get_dummies(data2['Ethnicity'])
data2 = data2.drop('Ethnicity', axis = 1)
data2 = data2.join(one_hot2)

one_hot3 = pd.get_dummies(data3['Ethnicity'])
```

```
data3 = data3.drop('Ethnicity', axis = 1)
data3 = data3.join(one_hot3)

one_hot4 = pd.get_dummies(data4['Ethnicity'])
data4 = data4.drop('Ethnicity', axis = 1)
data4 = data4.join(one_hot4)

features = ['Age', 'Vital_Sign Heart Rate', 'Vital_Sign Respiratory Rate',
'Vital_Sign Systolic Blood Pressure', 'Vital_Sign Diastolic Blood Pressure',
'Vital_Sign Oxygen Saturation', 'Vital_Sign Temperature Tympanic',
'Blood_Test HAEMOGLOBIN', 'Blood_Test HAEMATOCRIT', 'Blood_Test MEAN CELL
VOL.', 'Blood_Test WHITE CELLS', 'Blood_Test NEUTROPHILS', 'Blood_Test
LYMPHOCYTES', 'Blood_Test MONOCYTES', 'Blood_Test EOSINOPHILS', 'Blood_Test
BASOPHILS', 'Blood_Test PLATELETS', 'Blood_Test SODIUM', 'Blood_Test
POTASSIUM', 'Blood_Test UREA', 'Blood_Test CREATININE', 'Blood_Test eGFR',
'Blood_Test ALT', 'Blood_Test ALK.PHOSPHATASE', 'Blood_Test BILIRUBIN',
'Blood_Test ALBUMIN', 'Blood_Test CRP']

X_cont = data[features]
X_categ = data[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y = data["Covid-19 Positive"]

X_2_cont = data2[features]
X_2_categ = data2[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_2 = data2["Covid-19 Positive"]

X_3_cont = data3[features]
X_3_categ = data3[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_3 = data3["Covid-19 Positive"]

X_4_cont = data4[features]
X_4_categ = data4[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_4 = data4["Covid-19 Positive"]

x_train_cont, x_test_cont, x_train_categ, x_test_categ, y_train, y_test =
train_test_split(X_cont, X_categ, y, test_size=0.2, random_state=42)

x_train_categ = x_train_categ.to_numpy()
x_test_categ = x_test_categ.to_numpy()
x_train_cont = x_train_cont.to_numpy()
x_test_cont = x_test_cont.to_numpy()
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()

X_2_categ = X_2_categ.to_numpy()
X_2_cont = X_2_cont.to_numpy()
y_2 = y_2.to_numpy()

X_3_categ = X_3_categ.to_numpy()
X_3_cont = X_3_cont.to_numpy()
y_3 = y_3.to_numpy()

X_4_categ = X_4_categ.to_numpy()
```

```
X_4_cont = X_4_cont.to_numpy()
y_4 = y_4.to_numpy()

scaler = StandardScaler()
x_train_cont = scaler.fit_transform(x_train_cont, y_train)
x_test_cont = scaler.transform(x_test_cont)
X_2_cont = scaler.transform(X_2_cont)
X_3_cont = scaler.transform(X_3_cont)
X_4_cont = scaler.transform(X_4_cont)

x_train = np.concatenate((x_train_cont, x_train_categ), axis=1)
x_test = np.concatenate((x_test_cont, x_test_categ), axis=1)
x_2 = np.concatenate((X_2_cont, X_2_categ), axis=1)
x_3 = np.concatenate((X_3_cont, X_3_categ), axis=1)
x_4 = np.concatenate((X_4_cont, X_4_categ), axis=1)

x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train,
test_size=0.2, random_state=42)


def test_model(build_fn):
    reg = KerasClassifier(build_fn=build_fn, epochs=100, batch_size=128)
    reg.fit(x_train, y_train)
    y_pred = reg.predict(x_valid)
    y_pred = np.squeeze(y_pred)
    auc = roc_auc_score(y_valid, y_pred)

    return auc, reg

auc, reg = test_model(RLN(layers=4, input_dim=x_train.shape[1], norm=1,
avg_reg=-10, learning_rate=np.power(10, 6)))
print("Validation AUC:", auc)

y_pred = reg.predict(x_test)
y_pred = np.squeeze(y_pred)
Portsmouth_test_auc = roc_auc_score(y_test, y_pred)

y_pred2 = reg.predict(x_2)
y_pred2 = np.squeeze(y_pred2)
Bedford_valid_auc = roc_auc_score(y_2, y_pred2)

y_pred3 = reg.predict(x_3)
y_pred3 = np.squeeze(y_pred3)
Oxford_valid_auc = roc_auc_score(y_3, y_pred3)

y_pred4 = reg.predict(x_4)
y_pred4 = np.squeeze(y_pred4)
Birmingham_valid_auc = roc_auc_score(y_4, y_pred4)
```

MLR

The MLR implementation of anonymousNeurIPS2021submission5254 (2021) was used. The implementation was extended with Lookahead (Pytorch-optimizer, 2021a), using the code below.

The following code was used to create the MLR model:

```
import numpy as np
import pandas as pd
import sklearn as sk
from sklearn.base import BaseEstimator, ClassifierMixin, RegressorMixin
from sklearn.base import is_classifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, roc_auc_score, accuracy_score
from sklearn.utils import check_random_state
from sklearn.utils import shuffle
from scipy.special import expit as logistic_func
import torch as torch
import torch.nn.functional as F
import torch_optimizer as optim
from abc import ABCMeta, abstractmethod
import time


CUDA = True
if torch.cuda.is_available() and CUDA: dev = "cuda:0" else: dev = "cpu"
device = torch.device(dev)

MAX_GPU_MATRIX_WIDTH = int(4096)
MAX_GPU_NETWORK_DEPTH = int(6)
MAX_TRAINING_SAMPLES = int(1e4)
MAX_TREE_SIZE = 50


def n2t(array):
    if type(array) == torch.Tensor: return array
    return torch.tensor(array).to(device).float()
def t2n(tensor):
    if type(tensor) == np.ndarray: return tensor
    return tensor.detach().to("cpu").numpy()
def n2f(array):
    if type(array) == torch.Tensor: array = t2n(array)
    if type(array) == float: return array
    else: return np.array(array).reshape(-1)[0]
def f2s(value, length = 8, delimiter = " |"):
    if type(value) == str:
        return " " * (length - len(value)) + value[:length] + delimiter
    else:
        return ("%." +str(length-6)+"e") % float(value) + delimiter
```

```
class BaseMLRNN(BaseEstimator, metaclass=ABCMeta):
    @abstractmethod
    def __init__(self,
        depth,
        width,
        activation_function,
        loss_function,
        optimizer,
        learning_rate,
        batch_size,
        max_iter,
        max_runtime,
        validation_fraction,
        should_stratify,
        early_stopping_criterion,
        convergence_tol,
        divergence_tol,
        ridge_init,
        n_permut,
        label_noise_scale,
        target_rotation_scale,
        random_state,
        verbose
        ):

        self.depth  = depth
        self.width  = width
        self.activation_function  = activation_function
        self.loss_function = loss_function
        self.optimizer  = optimizer
        self.learning_rate  = learning_rate
        self.batch_size  = batch_size
        self.max_iter  = max_iter
        self.max_runtime  = max_runtime
        self.validation_fraction  = validation_fraction
        self.should_stratify = should_stratify
        self.early_stopping_criterion  = early_stopping_criterion
        self.convergence_tol  = convergence_tol
        self.divergence_tol  = divergence_tol
        self.ridge_init  = ridge_init
        self.n_permut  = n_permut
        self.label_noise_scale  = label_noise_scale
        self.target_rotation_scale = target_rotation_scale
        self.random_state  = random_state
        self.verbose  = verbose

    def _init_valid(self, X, y):

        if self.validation_fraction in [False, None]:
            self.validation = False
            return X, None, y, None
        else:
            def roc_with_proba(y, probas): return roc_auc_score(y, probas[:,-
1])
```

```
            self.valid_metric = roc_with_proba if is_classifier(self) else
r2_score
            self.valid_func = self.predict_proba if is_classifier(self) else
self.predict
            self.validation = True
            if self.should_stratify:
                if is_classifier(self): stratify = y
                else:
                    stratify = self._stratify_continuous_target(y)
            else: stratify = None
            X, X_valid, y, y_valid = train_test_split(
                X, y, random_state=self._random_state,
                test_size=self.validation_fraction,
                stratify=stratify)

            return X, n2t(X_valid), y, y_valid

    def _stratify_continuous_target(self, y):
        from sklearn.tree import DecisionTreeRegressor as tree_binarizer
        tree_binarizer_params = {"criterion":'friedman_mse',
                  "splitter":'best',
                  "max_depth":None,
                  "min_samples_split":2,
                  "min_weight_fraction_leaf":0.0,
                  "max_features":None,
                  "min_impurity_decrease":0.2,
                  "min_impurity_split":None,
                  "ccp_alpha":0.0}

        tree_size = min(int(np.sqrt(len(y))), MAX_TREE_SIZE)
        fit_sample_size = min(len(y), MAX_TRAINING_SAMPLES)
        tree_binarizer_params["random_state"] = self._random_state
        tree_binarizer_params["max_leaf_nodes"] = tree_size
        tree_binarizer_params["min_samples_leaf"] = tree_size

        return
tree_binarizer(**tree_binarizer_params).fit(y[:fit_sample_size].reshape((-
1,1)), y[:fit_sample_size]).apply(y.reshape((-1,1)))

    def _valid_score(self, ):
        return self.valid_metric(self.y_valid, self.valid_func(self.X_valid))

    def _init_batch(self, n_samples):
        if type(self.batch_size) == float:
            self.batch_length = min(MAX_GPU_MATRIX_WIDTH, int(n_samples *
self.batch_size))
            self.batch_learning = True
        elif type(self.batch_size) == int:
            self.batch_length = min(MAX_GPU_MATRIX_WIDTH, self.batch_size,
n_samples)
            self.batch_learning = True
        elif n_samples > MAX_GPU_MATRIX_WIDTH:
            self.batch_length = int(MAX_GPU_MATRIX_WIDTH)
            self.batch_learning = True
        else:
            self.batch_learning = False
        if self.batch_learning:
```

```
            self.n_batches = int(n_samples / self.batch_length)

    def _init_termination_criterion(self,):
        self.check_convergence = self.convergence_tol not in [False, None]
        self.check_divergence = self.divergence_tol not in [False, None]
        self.early_stopping = self.early_stopping_criterion not in [False,
None]
        self.sign_criterion = -1. if self.early_stopping_criterion ==
"validation" else 1.

    def _init_MLR(self, ):
        self.ridge_output = self.ridge_init not in [False, None]
        self.add_MLR = self.ridge_output and self.n_permut not in [False,
None, 0, 0.]

    def _init_label_noise(self, ):
        self.add_label_noise = self.label_noise_scale not in [False, None, 0,
0.]

    def _init_target_rotation(self, ):
        self.rotate_target = self.target_rotation_scale not in [False, None,
0, 0.]

    def _init_hidden_weights(self, ):
        self.hidden_layers = []
        for layer in range(self.depth + 1 - int(self.ridge_output)):
            fan_in = self.n_features if layer == 0 else self.width
            fan_out = 1 if layer == self.depth else self.width
            factor = 2 if self.activation_function == "logistic" else 6
            if self.activation_function == "relu": factor = factor * 2
            init_bound = np.sqrt(factor/(fan_in + fan_out))
            W = init_bound * (torch.rand((fan_in, fan_out), generator =
self.torch_random_state, device = device)* 2. - 1.)
            B = torch.zeros(size = (1,fan_out), device = device)
            self.hidden_layers.append([torch.nn.Parameter(weights) for
weights in (W,B)])
            del W,B
        params = [weight for layer in self.hidden_layers for weight in layer]
        if self.ridge_output: self.hidden_layers.append([None,
torch.zeros(size = (1,), device = device)])

        return params

    def _grid_search_ridge_coef(self, datas):
        GRID_START, GRID_END, GRID_SIZE = 1e-1, 1e4, 11
        candidates, losses = np.geomspace(GRID_START, GRID_END,
GRID_SIZE),np.zeros(GRID_SIZE)
        with torch.no_grad():
            activation = self._forward_pass(datas["input"])
            target = self._add_noise(datas["target"])
            activation_dot_target = activation.transpose(1,0) @
self._scale_target(target)
            activation_dot_target_permuted = None
            if self.add_MLR:
                target_permuted = self._add_noise(datas["target_permuted"])
                activation_dot_target_permuted = activation.transpose(1,0) @
self._scale_target(target_permuted)
```

```
            for i,candidate in enumerate(candidates):
                activation_dot_inv_mat = activation @
self._get_inv_mat(activation, n2t(np.log(candidate)), only_inversion = True)
                loss = self._compute_loss(activation_dot_inv_mat @
activation_dot_target, target)
                if self.add_MLR: loss +=
self._compute_MLR_penalty(activation_dot_inv_mat @
activation_dot_target_permuted, target_permuted)
                losses[i] = n2f(t2n(loss))

        if self.ridge_init == "max_variation":
            return np.geomspace(GRID_START, GRID_END, GRID_SIZE-
1)[np.argmax(losses[1:] - losses[:-1])]
        else:
            return candidates[np.argmin(losses)]

    def _init_ridge_coef(self, datas):
        if self.ridge_output:

            if type(self.ridge_init) in [float, int] or
isinstance(self.ridge_init, np.number):
                ridge_coef = self.ridge_init

            elif self.ridge_init in ["min_value", "max_variation"]:
                ridge_coef = self._grid_search_ridge_coef(datas)
            else:
                ridge_coef = 1.

            self.ridge_coef = torch.nn.Parameter(n2t(np.log(ridge_coef)))
            return [self.ridge_coef]
        else:
            return []

    def _init_record(self, ):
        self.record = {}
        self.record["loss"] = []
        self.record["time"] = []
        if self.validation:
            self.record["validation"] = []
        if self.ridge_output:
            self.record["lambda"] = []
        if self.add_MLR:
            self.record["mlr"] = []

    def _init_intercept(self, target):
        if self.add_MLR:
            with torch.no_grad():
                pred = target.mean() * torch.ones(target.shape, device =
device)
                self.intercept = self._compute_loss(self._scale_target(pred),
target).detach()

    def _initialize(self, X, y):
        self.init_time = time.time()
        self._random_state = check_random_state(self.random_state)
```

```
        self.torch_random_state =
torch.Generator(device=device).manual_seed(int(self._random_state.uniform(0,2
**31)))
        self.print_record  = self.verbose not in [0, False]
        X, self.X_valid, y, self.y_valid = self._init_valid(X, y)
        n_samples, self.n_features = X.shape
        self._init_termination_criterion()
        self._init_batch(n_samples)
        self._init_MLR()
        self._init_record()
        self._init_label_noise(), self._init_target_rotation()
        self.act_func = getattr(torch, self.activation_function)
        self.loss_func =
getattr(torch.nn,self.loss_function)(reduction='none')

        datas = self._init_data(X,y)
        self._init_intercept(datas["target"])
        params = self._init_hidden_weights()
        params = params + self._init_ridge_coef(datas)
        self.optimizer_instance  = getattr(torch.optim, self.optimizer)(lr =
self.learning_rate, params = params)
        self.optimizer_instance = optim.Lookahead(self.optimizer_instance,
k=5, alpha=0.5)
        del params
        self.current_iter = 0
        self.init_time = time.time() - self.init_time

        return X, y, datas

    def _reduce_loss(self, point_wise_loss):
        if self.loss_function == "MSE":
            return torch.sqrt(point_wise_loss.mean(dim = 0))
        else: return point_wise_loss.mean(dim = 0)

    def _init_data(self, X,y):
        datas = {"X":X, "y":y}
        if self.add_MLR:
            datas["y_permuted"] = self._permut_label(y)

        if self.batch_learning:
            self._generate_batch(datas["X"].shape[0])
            datas = self._update_data(datas)
        else:
            datas["input"] = n2t(datas["X"])
            datas["target"] = n2t(datas["y"])
            if self.add_MLR: datas["target_permuted"] =
n2t(datas["y_permuted"])

        return datas

    def _generate_batch(self, n_samples):
        shuffled_indexes =
shuffle(np.arange(n_samples),random_state=self._random_state)
        self.batches = [shuffled_indexes[batch_i * self.batch_length:
(batch_i + 1) * self.batch_length] for batch_i in range(self.n_batches)]

    def _update_data(self, datas):
```

```
        if self.batch_learning:
            if len(self.batches) == 0:
                self._generate_batch(datas["X"].shape[0])
            batch_indexes = self.batches.pop(0)
            datas["input"] = n2t(datas["X"][batch_indexes])
            datas["target"] = n2t(datas["y"][batch_indexes])
            if self.add_MLR: datas["target_permuted"] =
n2t(datas["y_permuted"][batch_indexes])

        return datas

    def _permut_label(self, y):
        y_index = np.arange(len(y))

        return
np.concatenate([y[shuffle(y_index,random_state=self._random_state)].reshape((
-1,1)) for permut in range(self.n_permut)], axis=1)

    def _forward_propagate(self, datas):
        activation = datas["input"]
        target = self._add_noise(datas["target"])
        activation = self._forward_pass(activation)

        if self.ridge_output:
            inv_mat = self._get_inv_mat(activation, self.ridge_coef)
            beta = inv_mat @ self._scale_target(target)
            self.hidden_layers[-1][0] = beta
            pred = activation @ beta
            if self.rotate_target:
                projection = activation @ inv_mat
                pred, target = self._rotate_pred(projection, pred, target)
        else:
            pred = activation.reshape(-1)
        loss = self._compute_loss(pred, target)
        self.record["loss"].append(n2f(t2n(loss)))
        if self.add_MLR:
            target_permutation = self._add_noise(datas["target_permuted"])
            if not self.rotate_target: projection = activation @ inv_mat
            pred_permutation = projection @
self._scale_target(target_permutation)
            if self.rotate_target:
                pred_permutation, target_permutation =
self._rotate_pred(projection, pred_permutation, target_permutation)
            permut_loss = self._compute_MLR_penalty(pred_permutation,
target_permutation)
            self.record["mlr"].append(n2f(t2n(permut_loss)))
            loss += permut_loss
        if self.validation:
self.record["validation"].append(self._valid_score())
        if self.early_stopping: self._save_weights()
        if self.ridge_output:
            self.record["lambda"].append(np.exp(n2f(t2n(self.ridge_coef))))
        self.record["time"].append(time.time() - self.current_time)
        self.current_time = time.time()
        if self.print_record:
            if self.current_iter == 0:
                print("| "+f2s("iter"), *map(f2s, self.record.keys()))
```

```
            if self.current_iter % int(self.verbose) == 0:
                print("| "+f2s(str(self.current_iter)), *map(lambda value :
f2s(value[-1]), self.record.values()))

        return loss

    def _add_noise(self, target):
        if self.add_label_noise:
            return target  + torch.normal(0., self.label_noise_scale, size =
target.shape, generator = self.torch_random_state, device = device)
        else:
            return target

    def _rotate_pred(self, projection, prediction, target):
        if self.rotate_target:
            epsilon = torch.normal(0.,self.target_rotation_scale, size =
target.shape ,generator = self.torch_random_state, device = device)
            complementary = epsilon - projection @ epsilon
            return (prediction + target)/2 + complementary, target
        else:
            return prediction, target

    def _scale_target(self, target):
        if is_classifier(self):
            return target * 2. - 1.
        else:
            return target

    def _forward_pass(self, activation):
        for layer,(W,B) in enumerate(self.hidden_layers[:self.depth + 1 -
int(self.ridge_output)]):
            activation = activation @ W + B
            if layer < self.depth: activation = self.act_func(activation)

        return activation

    def _get_inv_mat(self, activation, ridge_coef, only_inversion = False):
        diag = torch.diag(torch.ones(activation.shape[1],device = device) *
torch.exp(ridge_coef))
        inversed = torch.inverse((activation.transpose(1,0) @ activation) +
diag)
        if only_inversion: return inversed
        else: return inversed @ activation.transpose(1,0)

    def _compute_loss(self, pred, target):
        return self._reduce_loss(self.loss_func(pred, target)).mean()

    def _compute_MLR_penalty(self, pred, target):
        return torch.abs(self.intercept -
self._reduce_loss(self.loss_func(pred, target))).mean()

    def _backward_propagate(self, loss):
        loss.backward()
        self.optimizer_instance.step()
        self.optimizer_instance.zero_grad()
        self.current_iter = self.current_iter + 1
```

```
    def _check_termination(self,):
        return self._check_convergence() or self._check_divergence() or
self._check_timeout() or self.current_iter >= self.max_iter

    def _check_convergence(self, ):
        if self.check_convergence:
            return np.abs(np.min(self.record["loss"][:-1]) -
self.record["loss"][-1]) < self.convergence_tol
        else: return False

    def _check_divergence(self, ):
        if self.check_divergence:
            return self.record["loss"][-1] > self.divergence_tol
        else: return False

    def _check_timeout(self, ):
        return self.max_runtime < self.init_time +
np.sum(self.record["time"])

    def _save_weights(self,):
        if self.current_iter == 0:
            self.best_iter = self.current_iter
        elif self.record[self.early_stopping_criterion][-1] *
self.sign_criterion <
self.record[self.early_stopping_criterion][self.best_iter] *
self.sign_criterion:
            self.best_iter = self.current_iter
            del self.saved_hidden_layers
        if self.current_iter == self.best_iter:
            saved_weights_device = device if self.depth <=
MAX_GPU_NETWORK_DEPTH/2 else torch.device("cpu")
            self.saved_hidden_layers =
[[torch.clone(weights).detach().to(saved_weights_device) for weights in
couple] for couple in self.hidden_layers]

    def _load_weights(self,):
        self.hidden_layers = [[weights.to(device) for weights in couple] for
couple in self.saved_hidden_layers]
        del self.saved_hidden_layers

    def _release_train_memory(self):
        if self.validation: del self.X_valid, self.y_valid
        if self.ridge_output: del self.ridge_coef
        del self.optimizer_instance
        torch.cuda.empty_cache()

    def _forward_pass_fast(self, activation):
        with torch.no_grad():
            for layer,(W,B) in enumerate(self.hidden_layers):
                activation = activation @ W + B
                if layer < self.depth: activation = self.act_func(activation)
            del W,B

        return activation

    def _predict_hidden(self, X):
        if X.shape[0] <= MAX_GPU_MATRIX_WIDTH:
```

```
            return t2n(self._forward_pass_fast(n2t(X)))
        else:
            return np.concatenate([
self._predict_hidden(X[:MAX_GPU_MATRIX_WIDTH]),
self._predict_hidden(X[MAX_GPU_MATRIX_WIDTH:])])


    def _fit(self, X, y, incremental=False):
        if incremental:
            datas = self._init_data(X,y)
        else:
            X, y, datas = self._initialize(X, y)

        self.current_time = time.time()
        loss = self._forward_propagate(datas)
        while not self._check_termination():
            self._backward_propagate(loss)
            del loss
            datas = self._update_data(datas)
            loss = self._forward_propagate(datas)

        del loss
        if self.early_stopping: self._load_weights()
        self._release_train_memory()
        return self

    def fit(self, X, y):
        return self._fit(X,y, incremental=False)

    def delete_model_weights(self):
        del self.hidden_layers
        torch.cuda.empty_cache()

    def partial_fit(self, X, y):
        return self._fit(X,y, incremental=True)


class MLRNNClassifier(ClassifierMixin, BaseMLRNN):
    def __init__(self, *,
        depth = 3,
        width = 1024,
        activation_function = "relu",
        optimizer = "Adam",
        learning_rate = 1e-3,
        batch_size = False,
        max_iter = 100,
        max_runtime = 600,
        validation_fraction = 0.2,
        should_stratify = True,
        early_stopping_criterion = "validation",
        convergence_tol = False,
        divergence_tol = False,
        ridge_init = "max_variation",
        n_permut = 16,
        label_noise_scale = None,
        target_rotation_scale = 1.,
        random_state = None,
```

```
        verbose = False
                ):
        super().__init__(
        depth  = depth,
        width  = width,
        activation_function  = activation_function,
        loss_function = 'BCEWithLogitsLoss',
        optimizer  = optimizer,
        learning_rate  = learning_rate,
        batch_size  = batch_size,
        max_iter  = max_iter,
        max_runtime  = max_runtime,
        validation_fraction  = validation_fraction,
        should_stratify= should_stratify,
        early_stopping_criterion  = early_stopping_criterion,
        convergence_tol  = convergence_tol,
        divergence_tol  = divergence_tol,
        ridge_init  = ridge_init,
        n_permut  = n_permut,
        label_noise_scale  = label_noise_scale,
        target_rotation_scale =  target_rotation_scale,
        random_state  = random_state,
        verbose  = verbose
        )

    def predict(self, X):
        return self._predict_hidden(X) >= 0.

    def decision_function(self, X):
        return self._predict_hidden(X)

    def predict_proba(self, X):
        proba = logistic_func(self._predict_hidden(X))
        return np.c_[1.-proba, proba]
```

The following code was used to create the train and test the MLR model:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


data = pd.read_csv('portsmouth_balanced.csv')
data2 = pd.read_csv('bedford.csv')
data3 = pd.read_csv('oxford.csv')
data4 = pd.read_csv('birmingham.csv')

cleanup_gender = {"Gender": {"M": 0, "F": 1}}

data = data.replace(cleanup_gender)
data2 = data2.replace(cleanup_gender)
```

```
data3 = data3.replace(cleanup_gender)
data4 = data4.replace(cleanup_gender)

one_hot = pd.get_dummies(data['Ethnicity'])
data = data.drop('Ethnicity', axis = 1)
data = data.join(one_hot)

one_hot2 = pd.get_dummies(data2['Ethnicity'])
data2 = data2.drop('Ethnicity', axis = 1)
data2 = data2.join(one_hot2)

one_hot3 = pd.get_dummies(data3['Ethnicity'])
data3 = data3.drop('Ethnicity', axis = 1)
data3 = data3.join(one_hot3)

one_hot4 = pd.get_dummies(data4['Ethnicity'])
data4 = data4.drop('Ethnicity', axis = 1)
data4 = data4.join(one_hot4)

features = ['Age', 'Vital_Sign Heart Rate', 'Vital_Sign Respiratory Rate',
'Vital_Sign Systolic Blood Pressure', 'Vital_Sign Diastolic Blood Pressure',
'Vital_Sign Oxygen Saturation', 'Vital_Sign Temperature Tympanic',
'Blood_Test HAEMOGLOBIN', 'Blood_Test HAEMATOCRIT', 'Blood_Test MEAN CELL
VOL.', 'Blood_Test WHITE CELLS', 'Blood_Test NEUTROPHILS', 'Blood_Test
LYMPHOCYTES', 'Blood_Test MONOCYTES', 'Blood_Test EOSINOPHILS', 'Blood_Test
BASOPHILS', 'Blood_Test PLATELETS', 'Blood_Test SODIUM', 'Blood_Test
POTASSIUM', 'Blood_Test UREA', 'Blood_Test CREATININE', 'Blood_Test eGFR',
'Blood_Test ALT', 'Blood_Test ALK.PHOSPHATASE', 'Blood_Test BILIRUBIN',
'Blood_Test ALBUMIN', 'Blood_Test CRP']

X_cont = data[features]
X_categ = data[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y = data["Covid-19 Positive"]

X_2_cont = data2[features]
X_2_categ = data2[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_2 = data2["Covid-19 Positive"]

X_3_cont = data3[features]
X_3_categ = data3[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_3 = data3["Covid-19 Positive"]

X_4_cont = data4[features]
X_4_categ = data4[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_4 = data4["Covid-19 Positive"]

x_train_cont, x_test_cont, x_train_categ, x_test_categ, y_train, y_test =
train_test_split(X_cont, X_categ, y, test_size=0.2, random_state=42)

x_train_categ = x_train_categ.to_numpy()
x_test_categ = x_test_categ.to_numpy()
x_train_cont = x_train_cont.to_numpy()
x_test_cont = x_test_cont.to_numpy()
```

```
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()

X_2_categ = X_2_categ.to_numpy()
X_2_cont = X_2_cont.to_numpy()
y_2 = y_2.to_numpy()

X_3_categ = X_3_categ.to_numpy()
X_3_cont = X_3_cont.to_numpy()
y_3 = y_3.to_numpy()

X_4_categ = X_4_categ.to_numpy()
X_4_cont = X_4_cont.to_numpy()
y_4 = y_4.to_numpy()

scaler = StandardScaler()
x_train_cont = scaler.fit_transform(x_train_cont, y_train)
x_test_cont = scaler.transform(x_test_cont)
X_2_cont = scaler.transform(X_2_cont)
X_3_cont = scaler.transform(X_3_cont)
X_4_cont = scaler.transform(X_4_cont)

x_train = np.concatenate((x_train_cont, x_train_categ), axis=1)
x_test = np.concatenate((x_test_cont, x_test_categ), axis=1)
x_2 = np.concatenate((X_2_cont, X_2_categ), axis=1)
x_3 = np.concatenate((X_3_cont, X_3_categ), axis=1)
x_4 = np.concatenate((X_4_cont, X_4_categ), axis=1)


reg = MLRNNClassifier(depth = 3, width = 1024, learning_rate = 1e-3,
batch_size=128, random_state=42)
reg.fit(x_train, y_train)

y_pred = reg.predict(x_test)
y_pred = y_pred.astype(float)
Portsmouth_test_auc = roc_auc_score(y_test, y_pred)

y_pred2 = reg.predict(x_2)
y_pred2 = y_pred2.astype(float)
Bedford_valid_auc = roc_auc_score(y_2, y_pred2)

y_pred3 = reg.predict(x_3)
y_pred3 = y_pred3.astype(float)
Oxford_valid_auc = roc_auc_score(y_3, y_pred3)

y_pred4 = reg.predict(x_4)
y_pred4 = y_pred4.astype(float)
Birmingham_valid_auc = roc_auc_score(y_4, y_pred4)
```

SNN

The SNN implementation of Institute of Bioinformatics Johannes Kepler University Linz (2017) was used. The implementation was extended with weight decay (PyTorch, 2021a), dropout

(PyTorch, 2021c), stochastic weight averaging (PyTorch, 2021d) and Lookahead (Pytorch-optimizer, 2021a), using the code below.

The following code was used to create the SNN model:

```
import torch
import torch.nn as nn


class SNN(nn.Module):
    def __init__(self, in_features, out_features, p_drop=0.05):
        super(SNN, self).__init__()

        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=in_features, out_features=1024),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=1024, out_features=1024),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=1024, out_features=1024),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=1024, out_features=512),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=512, out_features=512),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=512, out_features=512),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=512, out_features=256),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=256, out_features=256),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=256, out_features=256),
            nn.SELU(),
            nn.AlphaDropout(p=p_drop),
            nn.Linear(in_features=256, out_features=out_features)
        )

        for param in self.net.parameters():
            if len(param.shape) == 1:
                nn.init.constant_(param, 0)
            else:
                nn.init.kaiming_normal_(param, mode='fan_in',
nonlinearity='linear')
```

```
    def forward(self, x):
        return self.net(x)
```

The following code was used to create the train and test the SNN model:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
import torch
import torch.nn as nn
import torch.utils.data as data_utils
from torch.utils.data import DataLoader
import torch_optimizer as optim


data = pd.read_csv('portsmouth_balanced.csv')
data2 = pd.read_csv('bedford.csv')
data3 = pd.read_csv('oxford.csv')
data4 = pd.read_csv('birmingham.csv')

cleanup_gender = {"Gender": {"M": 0, "F": 1}}

data = data.replace(cleanup_gender)
data2 = data2.replace(cleanup_gender)
data3 = data3.replace(cleanup_gender)
data4 = data4.replace(cleanup_gender)

one_hot = pd.get_dummies(data['Ethnicity'])
data = data.drop('Ethnicity', axis = 1)
data = data.join(one_hot)

one_hot2 = pd.get_dummies(data2['Ethnicity'])
data2 = data2.drop('Ethnicity', axis = 1)
data2 = data2.join(one_hot2)

one_hot3 = pd.get_dummies(data3['Ethnicity'])
data3 = data3.drop('Ethnicity', axis = 1)
data3 = data3.join(one_hot3)

one_hot4 = pd.get_dummies(data4['Ethnicity'])
data4 = data4.drop('Ethnicity', axis = 1)
data4 = data4.join(one_hot4)

features = ['Age', 'Vital_Sign Heart Rate', 'Vital_Sign Respiratory Rate',
'Vital_Sign Systolic Blood Pressure', 'Vital_Sign Diastolic Blood Pressure',
'Vital_Sign Oxygen Saturation', 'Vital_Sign Temperature Tympanic',
'Blood_Test HAEMOGLOBIN', 'Blood_Test HAEMATOCRIT', 'Blood_Test MEAN CELL
VOL.', 'Blood_Test WHITE CELLS', 'Blood_Test NEUTROPHILS', 'Blood_Test
LYMPHOCYTES', 'Blood_Test MONOCYTES', 'Blood_Test EOSINOPHILS', 'Blood_Test
```

```
BASOPHILS', 'Blood_Test PLATELETS', 'Blood_Test SODIUM', 'Blood_Test
POTASSIUM', 'Blood_Test UREA', 'Blood_Test CREATININE', 'Blood_Test eGFR',
'Blood_Test ALT', 'Blood_Test ALK.PHOSPHATASE', 'Blood_Test BILIRUBIN',
'Blood_Test ALBUMIN', 'Blood_Test CRP']

X_cont = data[features]
X_categ = data[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y = data["Covid-19 Positive"]

X_2_cont = data2[features]
X_2_categ = data2[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_2 = data2["Covid-19 Positive"]

X_3_cont = data3[features]
X_3_categ = data3[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_3 = data3["Covid-19 Positive"]

X_4_cont = data4[features]
X_4_categ = data4[['Gender', 'White', 'South Asian', 'Black', 'Chinese',
'Mixed', 'Other', 'Unknown']]
y_4 = data4["Covid-19 Positive"]

X_train_cont, X_test_cont, X_train_categ, X_test_categ, y_train, y_test =
train_test_split(X_cont, X_categ, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_cont = scaler.fit_transform(X_train_cont, y_train)
X_test_cont = scaler.transform(X_test_cont)
X_2_cont = scaler.transform(X_2_cont)
X_3_cont = scaler.transform(X_3_cont)
X_4_cont = scaler.transform(X_4_cont)

X_train_cont = pd.DataFrame(X_train_cont, columns = features)
X_test_cont = pd.DataFrame(X_test_cont, columns = features)
X_2_cont = pd.DataFrame(X_2_cont, columns = features)
X_3_cont = pd.DataFrame(X_3_cont, columns = features)
X_4_cont = pd.DataFrame(X_4_cont, columns = features)

X_train = pd.concat([X_train_cont.reset_index(drop=True),
X_train_categ.reset_index(drop=True)], axis=1)
X_test = pd.concat([X_test_cont.reset_index(drop=True),
X_test_categ.reset_index(drop=True)], axis=1)
X_2 = pd.concat([X_2_cont.reset_index(drop=True),
X_2_categ.reset_index(drop=True)], axis=1)
X_3 = pd.concat([X_3_cont.reset_index(drop=True),
X_3_categ.reset_index(drop=True)], axis=1)
X_4 = pd.concat([X_4_cont.reset_index(drop=True),
X_4_categ.reset_index(drop=True)], axis=1)

X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train,
test_size=0.2, random_state=42)

X_train = torch.tensor(X_train.values.astype(np.float32))
y_train = torch.tensor(y_train.values.astype(np.float32))
```

```
X_valid = torch.tensor(X_valid.values.astype(np.float32))
y_valid = torch.tensor(y_valid.values.astype(np.float32))
X_test = torch.tensor(X_test.values.astype(np.float32))
y_test = torch.tensor(y_test.values.astype(np.float32))

X_2 = torch.tensor(X_2.values.astype(np.float32))
y_2 = torch.tensor(y_2.values.astype(np.float32))

X_3 = torch.tensor(X_3.values.astype(np.float32))
y_3 = torch.tensor(y_3.values.astype(np.float32))

X_4 = torch.tensor(X_4.values.astype(np.float32))
y_4 = torch.tensor(y_4.values.astype(np.float32))

train_tensor = data_utils.TensorDataset(X_train, y_train)
valid_tensor = data_utils.TensorDataset(X_valid, y_valid)
test_tensor = data_utils.TensorDataset(X_test, y_test)
bedford_tensor = data_utils.TensorDataset(X_2, y_2)
oxford_tensor = data_utils.TensorDataset(X_3, y_3)
birmingham_tensor = data_utils.TensorDataset(X_4, y_4)

trainloader = DataLoader(train_tensor, batch_size=128, shuffle=True,
num_workers=2)
valloader = DataLoader(valid_tensor, batch_size=len(valid_tensor),
shuffle=True, num_workers=2)
testloader = DataLoader(test_tensor, batch_size=len(test_tensor),
shuffle=False, num_workers=2)
bedfordloader = DataLoader(bedford_tensor, batch_size=len(bedford_tensor),
shuffle=False, num_workers=2)
oxfordloader = DataLoader(oxford_tensor, batch_size=len(oxford_tensor),
shuffle=False, num_workers=2)
birminghamloader = DataLoader(birmingham_tensor,
batch_size=len(birmingham_tensor), shuffle=False, num_workers=2)


device = "cuda" if torch.cuda.is_available() else "cpu"
network = SNN(in_features=35, out_features=1, p_drop=0.05).to(device)
criterion = nn.BCEWithLogitsLoss()

optimizer = torch.optim.AdamW(params=network.parameters(), lr=1e-3,
weight_decay=0.0001)
optimizer = optim.Lookahead(optimizer, k=5, alpha=0.5)
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer,
T_0=15, T_mult=2)
swa_model = torch.optim.swa_utils.AveragedModel(network)
swa_start = 10
swa_scheduler = torch.optim.swa_utils.SWALR(optimizer, anneal_epochs=5,
swa_lr=0.01)
epochs = 20

for epoch in range(epochs):
    train_loss = 0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        labels = torch.unsqueeze(labels, 1)
        optimizer.zero_grad()
        outputs = network(inputs)
```

```
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
    if epoch > swa_start:
        swa_model.update_parameters(network)
        swa_scheduler.step()
    else:
        scheduler.step()
    if epoch % 2 == 0:
        for data in valloader:
            inputs, labels = data
            labels1 = torch.unsqueeze(labels, 1)
            outputs = swa_model(inputs)
            val_loss = criterion(outputs, labels1)
            outputs = torch.sigmoid(outputs)
            outputs = torch.where(outputs > 0.5, 1, 0)
            labels = labels.detach().numpy()
            outputs = outputs.detach().numpy()
            outputs = np.squeeze(outputs)
            val_auc = roc_auc_score(labels, outputs)
        print("Epoch:", epoch, "Train loss:", train_loss / len(trainloader),
"Val loss:", val_loss.item() / len(valloader), "Val AUC:", val_auc)

with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        outputs = swa_model(inputs)
        outputs = torch.sigmoid(outputs)
        outputs = torch.where(outputs > 0.5, 1, 0)
        labels = labels.detach().numpy()
        outputs = outputs.detach().numpy()
        outputs = np.squeeze(outputs)
        Portsmouth_test_auc = roc_auc_score(labels, outputs)

with torch.no_grad():
    for data in bedfordloader:
        inputs, labels = data
        outputs = swa_model(inputs)
        outputs = torch.sigmoid(outputs)
        outputs = torch.where(outputs > 0.5, 1, 0)
        labels = labels.detach().numpy()
        outputs = outputs.detach().numpy()
        outputs = np.squeeze(outputs)
        Bedford_valid_auc = roc_auc_score(labels, outputs)

with torch.no_grad():
    for data in oxfordloader:
        inputs, labels = data
        outputs = swa_model(inputs)
        outputs = torch.sigmoid(outputs)
        outputs = torch.where(outputs > 0.5, 1, 0)
        labels = labels.detach().numpy()
        outputs = outputs.detach().numpy()
        outputs = np.squeeze(outputs)
        Oxford_valid_auc = roc_auc_score(labels, outputs)
```

```
with torch.no_grad():
    for data in birminghamloader:
        inputs, labels = data
        outputs = swa_model(inputs)
        outputs = torch.sigmoid(outputs)
        outputs = torch.where(outputs > 0.5, 1, 0)
        labels = labels.detach().numpy()
        outputs = outputs.detach().numpy()
        outputs = np.squeeze(outputs)
        Birmingham_valid_auc = roc_auc_score(labels, outputs)
```

References

1. anonymousNeurIPS2021submission5254. (2021). *Supplementary Material.* GitHub repository, https://github.com/anonymousNeurIPS2021submission5254/SupplementaryMaterial.
2. Chen, Y.S. (2020). *Quantum Forest.* GitHub repository, https://github.com/closest-git/QuantumForest/tree/7f8bbd19e129d9d34849603bf8075054b5bbbd42.
3. DeepTables. (2021). *DeepTables.* https://deeptables.readthedocs.io/en/latest/.
4. Institute of Bioinformatics Johannes Kepler University Linz. (2017). *SNNs.* GitHub repository, https://github.com/bioinf-jku/SNNs.
5. Huang, H. (2021). *Normalizing-flows-reproduce.* GitHub repository, https://github.com/andrehuang/normalizing-flows-reproduce.
6. Majumdar, S. (2017). *Snapshot-Ensembles.* GitHub repository, https://github.com/titu1994/Snapshot-Ensembles.
7. PyTorch. (2021a). *AdamW.* https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html#torch.optim.AdamW.
8. PyTorch. (2021b). *Batchnorm1d.* https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html.
9. PyTorch. (2021c). *Dropout.* https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html.
10. PyTorch. (2021d). *torch.optim.* https://pytorch.org/docs/stable/optim.html.
11. Pytorch-optimizer. (2021a). *Lookahead.* https://pytorch-optimizer.readthedocs.io/en/latest/.
12. Pytorch-optimizer. (2021b). *QHAdam.* https://pytorch-optimizer.readthedocs.io/en/latest/api.html#qhadam.
13. PyTorch Tabular. (2021). *PyTorch Tabular.* https://pytorch-tabular.readthedocs.io/en/latest/.
14. SDV. (2021). *Single Table Metrics.* https://sdv.dev/SDV/user_guides/evaluation/single_table_metrics.html.
15. Shavitt, I. (2018). *Regularization_learning_networks.* GitHub repository, https://github.com/irashavitt/regularization_learning_networks.
16. Somepalli, G. (2021). *Saint.* GitHub repository, https://github.com/somepago/saint.
17. TensorFlow. (2021a). *tf.keras.layers.BatchNormalization*. https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization.

18. TensorFlow. (2021b). *tf.keras.layers.Dropout*. https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout.
19. TensorFlow. (2021c). *tf.keras.regularizers.L2*. https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/L2.
20. TensorFlow Addons. (2021a). *tfa.optimizers.Lookahead*. https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers/Lookahead.
21. TensorFlow Addons. (2021b). *tfa.optimizers.SWA*. https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers/SWA.
22. Wang, P. (2020). *Tab-transformer-pytorch.* GitHub repository, https://github.com/lucidrains/tab-transformer-pytorch.
23. wOOL. (2018). *DNDT.* GitHub repository, https://github.com/wOOL/DNDT.