Supplementary material

## Random Forest and BorutaShap for Feature Selection

from BorutaShap import BorutaShap

from xgboost import XGBClassifier

from catboost import CatBoostClassifier

from lightgbm import LGBMClassifier

from sklearn.ensemble import RandomForestClassifier

device = ('gpu' if torch.cuda.is\_available() else 'cpu')

def split\_data(X, y, random\_state, test\_size=0.1, use\_stratified\_kfold=False, n\_splits=10, n\_repeats=1):

    if use\_stratified\_kfold:

      skf = RepeatedStratifiedKFold(n\_splits=n\_splits, n\_repeats=n\_repeats, random\_state=random\_state)

        for train\_index, val\_index in skf.split(X, y):

            X\_train\_split, X\_val\_split = X.iloc[train\_index], X.iloc[val\_index]

            y\_train\_split, y\_val\_split = y[train\_index], y[val\_index]

            print(y\_train\_split)

            yield X\_train\_split, X\_val\_split, y\_train\_split, y\_val\_split

    else:

      X\_train\_split, X\_val\_split, y\_train\_split, y\_val\_split = train\_test\_split(X, y, test\_size=test\_size, random\_state=random\_state)

        yield X\_train\_split, X\_val\_split, y\_train\_split, y\_val\_split

model = RandomForestClassifier()

Feature\_Selector = BorutaShap(model=model,importance\_measure='shap', classification=True)

for X\_train\_, X\_val, y\_train\_, y\_val in split\_data(X\_train, y\_train, random\_state=42, test\_size=0.1, use\_stratified\_kfold=True, n\_splits=2, n\_repeats=n\_repeats):

    Feature\_Selector.fit(X=X\_train\_, y=y\_train\_, n\_trials=50, random\_state=42)

## LightGBM and BorutaShap for Feature Selection

model = LGBMClassifier()

Feature\_Selector = BorutaShap(model=model,importance\_measure='shap', classification=True)

for X\_train\_, X\_val, y\_train\_, y\_val in split\_data(X\_train, y\_train, random\_state=42, test\_size=0.1, use\_stratified\_kfold=True, n\_splits=2, n\_repeats=n\_repeats):

    Feature\_Selector.fit(X=X\_train\_, y=y\_train\_, n\_trials=50, random\_state=42)

## CatBoost and BorutaShap for Feature Selection

model = CatBoostClassifier()

Feature\_Selector = BorutaShap(model=model,importance\_measure='shap', classification=True)

for X\_train\_, X\_val, y\_train\_, y\_val in split\_data(X\_train, y\_train, random\_state=42, test\_size=0.1, use\_stratified\_kfold=True, n\_splits=2, n\_repeats=n\_repeats):

    Feature\_Selector.fit(X=X\_train\_, y=y\_train\_, n\_trials=50, random\_state=42)

## XGBoost and BorutaShap for Feature Selection

model = XGBoostClassifier()

Feature\_Selector = BorutaShap(model=model,importance\_measure='shap', classification=True)

for X\_train\_, X\_val, y\_train\_, y\_val in split\_data(X\_train, y\_train, random\_state=42, test\_size=0.1, use\_stratified\_kfold=True, n\_splits=2, n\_repeats=n\_repeats):

    Feature\_Selector.fit(X=X\_train\_, y=y\_train\_, n\_trials=50, random\_state=42)

## Tuning Random Forest Hyperparameters

def objective(trial):

    params = {

        "n\_estimators": trial.suggest\_int("n\_estimators", 1, 500),

        "max\_depth": trial.suggest\_int("max\_depth", 1, 50),

        "min\_samples\_split": trial.suggest\_int("min\_samples\_split", 2, 10, log=True),

        "min\_samples\_leaf": trial.suggest\_int("min\_samples\_leaf", 1, 5, log=True),

        'random\_state': 42,

    }

    model = RandomForestClassifier(\*\*params)

    model.fit(X\_train, y\_train)

    predictions = model.predict(X\_val)

    acc = accuracy\_score(y\_val, predictions)

return acc

## Tuning LightGBM Hyperparameters

def objective(trial):

    tree\_method = ['exact','approx','hist']

    boosting\_list = ['gbtree', 'gblinear']

    objective\_list\_reg = ['reg:linear', 'reg:gamma', 'reg:tweedie']

    params = {

        'lambda\_l1': trial.suggest\_uniform('lambda\_l1', 0.0, 10.0),

        'lambda\_l2': trial.suggest\_uniform('lambda\_l2', 0.0, 1.0),

 'num\_leaves': trial.suggest\_int('num\_leaves', 2, 1024, step=1, log=True)

 'feature\_fraction': trial.suggest\_uniform('feature\_fragtion', 0.1, 1.0),

        'bagging\_fraction': trial.suggest\_uniform('bagging\_fraction', 0., 1.0),

'bagging\_freq': trial.suggest\_int('bagging\_freq', 0.0, 10),

'min\_child\_sample': trial.suggest\_int('min\_child\_sample', 0.0, 100.0),

        'random\_state': 42,}

    model = LGBMClassifier(\*\*params)

    model.fit(X\_train, y\_train)

    predictions = model.predict(X\_val)

    acc = accuracy\_score(y\_val, predictions)

return acc

## Tuning XGBoost Hyperparameters

def objective(trial):

    objective\_list\_reg = ['depthwise', 'lossguide']

    params = {

          'max\_depth':trial.suggest\_int('max\_depth', 0, 25),

          'alpha':trial.suggest\_uniform ('reg\_alpha', 0, 1),

          'lambda':trial.suggest\_uniform ('reg\_lambda', 0, 1),

          'eta':trial. suggest\_uniform ('reg\_eta', 0, 1),

          'gamma':trial. suggest\_uniform('gamma', 0, 1),

          'grow\_policy':trial.suggest\_categorical('grow\_policy', objective\_list\_reg),

          'random\_seed': 42,

          'verbose':0

    }

    model = XGBClassifier(\*\*params)

    model.fit(X\_train, y\_train)

    predictions = model.predict(X\_val)

    acc = accuracy\_score(y\_val, predictions)

return acc

## Tuning CatBoost Hyperparameters

import optuna

def objective(trial):

    params = {

        "objective": trial.suggest\_categorical("objective", ["Logloss", "CrossEntropy"]),

        "colsample\_bylevel": trial.suggest\_float("colsample\_bylevel", 0.01, 0.1),

        "depth": trial.suggest\_int("depth", 1, 12),

        "boosting\_type": trial.suggest\_categorical("boosting\_type", ["Ordered", "Plain"]),

        "bootstrap\_type": trial.suggest\_categorical(

            "bootstrap\_type", ["Bayesian", "Bernoulli", "MVS"]

        )

    }

    if params["bootstrap\_type"] == "Bayesian":

       params["bagging\_temperature"] = trial.suggest\_float("bagging\_temperature", 0, 10)

    elif params["bootstrap\_type"] == "Bernoulli":

        params["subsample"] = trial.suggest\_float("subsample", 0.1, 1)

    model = CatBoostClassifier(\*\*params)

    model.fit(X\_train, y\_train)

    predictions = model.predict(X\_val)

    acc = accuracy\_score(y\_val, predictions)

return acc

## Hyperparameter Optimization with Optuna

 study = optuna.create\_study(direction='maximize')

study.optimize(lambda trial: objective(trial), n\_trials=100)

## Tuning TabNet Hyperparameters

import scipy

from pytorch\_tabnet.tab\_model import TabNetClassifier

def objective(trial):

    n\_d\_a = trial.suggest\_int('n\_d\_a', 8, 256)

    n\_steps = trial.suggest\_int('n\_steps', 3, 50)

    gamma = trial.suggest\_float('gamma', 1.0, 3.0)

    lr = trial.suggest\_float('lr', 1e-5, 1e-2)

    cat\_emb\_dim = trial.suggest\_int('cat\_emb\_dim', 1, 4)

    tabnet\_params = {"cat\_idxs": cat\_column\_index,

                     "cat\_dims": cat\_cardinalities,

                     "cat\_emb\_dim": 1,

                     "optimizer\_fn": torch.optim.Adam,

                     "optimizer\_params": dict(lr=lr),

                     "scheduler\_fn": None,

                     "mask\_type": 'sparsemax',

                     "device\_name": 'cuda',

                     "n\_d": n\_d\_a,

                     "n\_a": n\_d\_a,

                     "n\_steps": n\_steps,

                     "gamma": gamma,

                     "verbose": 0,

                     "seed": 42}

    clf = TabNetClassifier(\*\*tabnet\_params)

    max\_epochs = 1000

    clf.fit(X\_train=X\_train.values, y\_train=y\_train,

            eval\_set=[(X\_train.values, y\_train), (X\_val.values, y\_val)],

            eval\_name=['train', 'val'],

            eval\_metric=['accuracy', 'logloss','f1\_score'],

            max\_epochs=max\_epochs,

            patience=100,

            batch\_size=1024,

            virtual\_batch\_size=128,)

    # return minimun loss

    return clf.best\_cost

study = optuna.create\_study(direction='minimize')

study.optimize(lambda trial: objective(trial), n\_trials=100)

## Training TabNet Model

from pytorch\_tabnet.tab\_model import TabNetClassifier

tabnet\_params = {"cat\_idxs": cat\_column\_index,

                 "cat\_dims": cat\_cardinalities,

                 "optimizer\_fn": torch.optim.AdamW,

                 "scheduler\_fn": None,

                 "mask\_type": 'sparsemax',

                 "device\_name": 'cuda',

                 "optimizer\_params": dict(lr=0.005990598257128396),

                 "n\_d": 101,

                 "n\_a": 101,

                 "n\_steps": 48,

                 "gamma": 2.46398788362281,

                 'cat\_emb\_dim': 1,

                 "seed": 42}

clf = TabNetClassifier(\*\*tabnet\_params)

max\_epochs = 1000

# Fitting the model

clf.fit(X\_train=X\_train.values, y\_train=y\_train,

        eval\_set=[(X\_train.values, y\_train), (X\_val.values, y\_val)],

        eval\_name=['train', 'val'],

        eval\_metric=['accuracy', 'logloss','f1\_score'],

        max\_epochs=max\_epochs,

        patience=100,

        batch\_size=1024,

        virtual\_batch\_size=128,)

preds = clf.predict(X\_test.values)

print('Accuracy score:',accuracy\_score(y\_test, preds))

print('Precision score:',precision\_score(y\_test, preds))

print('Recall score:',recall\_score(y\_test, preds))

print('F1 score:',f1\_score(y\_test, preds))

print('AUC score:',roc\_auc\_score(y\_test, preds))

## FT-Transformer Model Setup

# Dataset with num / cat features (for rtdl)

class TensorData(Dataset):

    def \_\_init\_\_(self, num, cat, label):

        self.num = num

        self.cat = cat

        self.label = label

        self.len = self.label.shape[0]

    def \_\_getitem\_\_(self, index):

        return self.num[index],self.cat[index], self.label[index]

    def \_\_len\_\_(self):

        return self.len

def setting\_rtdl(data, label):

    '''

    DataFrame, np.array -> torch.Tensor

    ResNet: model(X\_num, X\_cat) / split X -> X\_num, X\_cat

    '''

    cat\_index = cat\_features

    num\_index = num\_features

    X = {'train': {},

         'val': {},

         'test': {}}

    y = {'train': {},

         'val': {},

         'test': {}}

    X['train']['num'] = torch.tensor(data['train'][num\_index].values, device=device)

    X['train']['cat'] = torch.tensor(data['train'][cat\_index].values, device=device)

    X['val']['num'] = torch.tensor(data['val'][num\_index].values, device=device)

    X['val']['cat'] = torch.tensor(data['val'][cat\_index].values, device=device)

    X['test']['num'] = torch.tensor(data['test'][num\_index].values, device=device)

    X['test']['cat'] = torch.tensor(data['test'][cat\_index].values, device=device)

    y['train'] = torch.tensor(label['train'], dtype=torch.float, device=device)

    y['val'] = torch.tensor(label['val'], dtype=torch.float, device=device)

    y['test'] = torch.tensor(label['test'], dtype=torch.float, device=device)

return X, y

def read\_split\_data():

    X = {'train': {},

         'val': {},

         'test': {}}

    y = {'train': {},

         'val': {},

         'test': {}}

    X['train'], X['test'], y['train'], y['test'] = train\_test\_split(X\_split, y\_split, test\_size = 0.1, random\_state=42,stratify=y\_split)

    X['train'], X['val'], y['train'], y['val'] = train\_test\_split(X['train'], y['train'], test\_size = 0.1111, random\_state=42,stratify=y['train'])

    return X, y

def model\_train(model, data\_loader, criterion, optimizer, device, scheduler=None):

    model.train()

    running\_loss = 0

    corr = 0

    # for rtdl

    for x\_num, x\_cat, label in tqdm(data\_loader):

        optimizer.zero\_grad()

        x\_num, x\_cat, label = x\_num.to(device), x\_cat.to(device), label.to(device)

        output = model(x\_num, x\_cat)

        output = torch.sigmoid(output)

        loss = criterion(output, label)

        loss.backward()

        optimizer.step()

        pred = output >= torch.FloatTensor([0.5]).to(device)

        corr += pred.eq(label).sum().item()

        running\_loss += loss.item() \* x\_num.size(0)

    if scheduler:

        scheduler.step()

    # Average accuracy & loss

    accuracy = corr / len(data\_loader.dataset)

    loss = running\_loss / len(data\_loader.dataset)

    history['train\_loss'].append(loss)

    history['train\_accuracy'].append(accuracy)

    return loss, accuracy

def model\_evaluate(model, data\_loader, criterion, device):

    model.eval()

    with torch.no\_grad():

        running\_loss = 0

        corr = 0

        total\_pred, total\_label = torch.tensor([]).to(device), torch.tensor([]).to(device)

        for x\_num, x\_cat, label in data\_loader:

            x\_num, x\_cat, label = x\_num.to(device), x\_cat.to(device), label.to(device)

            output = model(x\_num, x\_cat)

            output = torch.sigmoid(output)

            pred = output >= torch.FloatTensor([0.5]).to(device)

            corr += pred.eq(label).sum().item()

            running\_loss += criterion(output, label).item() \* x\_num.size(0)

            total\_pred = torch.cat((total\_pred, pred), dim=0)

            total\_label = torch.cat((total\_label, label), dim=0)

        accuracy = corr / len(data\_loader.dataset)

        loss = running\_loss / len(data\_loader.dataset)

        precision = precision\_score(total\_label.cpu().numpy(), total\_pred.cpu().numpy())

        f1 = f1\_score(total\_label.cpu().numpy(), total\_pred.cpu().numpy())

        recall = recall\_score(total\_label.cpu().numpy(), total\_pred.cpu().numpy())

        auc = roc\_auc\_score(total\_label.cpu().numpy(), total\_pred.cpu().numpy())

        print(total\_label.cpu().numpy().flatten())

        print( total\_pred.cpu().numpy().flatten())

        return loss, accuracy, precision, recall, f1, auc,total\_pred.cpu().numpy().flatten()

def ftt\_model\_predict(model, data\_input, device):

    (shape,\_\_) = data\_input.shape

    data = []

    numerical\_data = [data\_input[i, num\_index] for i in range(shape)]

    categorical\_data = [data\_input[i, cat\_index] for i in range(shape)]

    numerical\_data = torch.tensor(numerical\_data).to(device).long()

    categorical\_data = torch.tensor(categorical\_data).to(device).long()

    data = TensorData(numerical\_data, categorical\_data, np.zeros(shape))

    data\_loader = DataLoader(data, batch\_size=32, shuffle=False)

    model.eval()

    with torch.no\_grad():

        total\_pred = torch.tensor([]).to(device)

        for x\_num, x\_cat,\_ in data\_loader:

            x\_num, x\_cat = x\_num.to(device), x\_cat.to(device)

            output = model(x\_num, x\_cat)

            output = torch.sigmoid(output)

            pred = output >= torch.FloatTensor([0.5]).to(device)

            total\_pred = torch.cat((total\_pred, pred), dim=0)

        return total\_pred.cpu().numpy().flatten()

def model\_tune(model, train\_loader, val\_loader, criterion, optimizer, device):

    model.train()

    # train\_loader

    for x\_num, x\_cat, label in train\_loader:

        optimizer.zero\_grad()

        x\_num, x\_cat, label = x\_num.to(device), x\_cat.to(device), label.to(device)

        output = model(x\_num, x\_cat)

        output = torch.sigmoid(output)

        loss = criterion(output, label)

        loss.backward()

        optimizer.step()

    # val\_loader

    model.eval()

    with torch.no\_grad():

        running\_loss = 0

        corr = 0

        for x\_num, x\_cat, label in val\_loader:

            x\_num, x\_cat, label = x\_num.to(device), x\_cat.to(device), label.to(device)

            output = model(x\_num, x\_cat)

            output = torch.sigmoid(output)

            pred = output >= torch.FloatTensor([0.5]).to(device)

            corr += pred.eq(label).sum().item()

            running\_loss += criterion(output, label).item() \* x\_num.size(0)

        val\_accuracy = corr / len(val\_loader.dataset)

        val\_loss = running\_loss / len(val\_loader.dataset)

        return val\_loss, val\_accuracy

## Data Preprocessing and Oversampling with SMOTE

from imblearn.over\_sampling import SMOTE

def ready\_data():

    # data setting

    X, y = read\_split\_data()

        sm = SMOTE(random\_state=42)

X['train'], y['train'] = sm.fit\_resample(X['train'], y['train'].ravel())

    X['train'][num\_features] = X['train'][num\_features].apply(lambda x: x.astype('float32'))

    X['val'][num\_features] = X['val'][num\_features].apply(lambda x: x.astype('float32'))

    X['test'][num\_features] = X['test'][num\_features].apply(lambda x: x.astype('float32'))

    y['train'] = y['train'].reshape(-1, 1)

    y['val'] = y['val'].reshape(-1, 1)

    y['test'] = y['test'].reshape(-1, 1)

    cardinalities = []

    for col in cat\_features:

        max\_cat = np.max([np.max(X['train'][col]),

                        np.max(X['val'][col]),

                        np.max(X['test'][col])]) + 1

        cardinalities.append(max\_cat)

        X, y = setting\_rtdl(X, y)

    # dataset, dataloader

    train\_data = TensorData(X['train']['num'], X['train']['cat'], y['train'])

    val\_data = TensorData(X['val']['num'], X['val']['cat'], y['val'])

    test\_data = TensorData(X['test']['num'], X['test']['cat'], y['test'])

    train\_loader = DataLoader(train\_data, batch\_size=32, shuffle=False)

    val\_loader = DataLoader(val\_data, batch\_size=32, shuffle=False)

    test\_loader = DataLoader(test\_data, batch\_size=32, shuffle=False)

    return X, y,cardinalities, train\_loader, val\_loader, test\_loader

X, y,cardinalities, train\_loader, val\_loader, test\_loader = ready\_data()

## Tuning FT-Transformer Hyperparameters

def objective(trial, train\_loader, val\_loader):

    d\_token = trial.suggest\_categorical('d\_token', [64, 128, 256, 512])

    n\_blocks = trial.suggest\_int('n\_blocks', 1, 4, log=True)

    attention\_dropout = trial.suggest\_float('attention\_dropout', 0, 0.5)

    ffn\_d\_hidden = trial.suggest\_int('ffn\_d\_hidden', 64, 1028)

    ffn\_dropout = trial.suggest\_float('ffn\_dropout', 0, 0.5)

    residual\_dropout = trial.suggest\_float('residual\_dropout', 0, 0.2)

    lr = trial.suggest\_float('lr', 1e-5, 1e-2, log=True)

    weight\_decay = trial.suggest\_float('weigth\_decay', 1e-6, 1e-3, log=True)

    model = rtdl.FTTransformer.make\_baseline(n\_num\_features=X['train']['num'].shape[1],

                                             cat\_cardinalities=cardinalities,

                                             d\_token=d\_token,

                                             n\_blocks=n\_blocks,

                                             attention\_dropout=attention\_dropout,

                                             ffn\_d\_hidden=ffn\_d\_hidden,

                                             ffn\_dropout=ffn\_dropout,

                                             residual\_dropout=residual\_dropout,

                                             d\_out=1).to(device)

    criterion = nn.BCELoss()

    optimizer = optim.AdamW(model.parameters(), lr=lr, weight\_decay=weight\_decay)

    EPOCHS = 100

    min\_loss = np.inf

    for epoch in range(EPOCHS):

        val\_loss, val\_acc = model\_tune(model, train\_loader, val\_loader, criterion, optimizer, device)

        if val\_loss < min\_loss:

            min\_loss = val\_loss

    # minimize minimun loss

    return min\_loss

study = optuna.create\_study(study\_name='FT-Transformer', direction='minimize', sampler=TPESampler(seed=42))

study.optimize(lambda trial: objective(trial, train\_loader, val\_loader), n\_trials=30)

print()

print("Best Score:", study.best\_value)

print("Best trial:", study.best\_trial.params)

## FT-Transformer Training and Validation

ft\_t = rtdl.FTTransformer.make\_baseline(n\_num\_features=X['train']['num'].shape[1],

                                        cat\_cardinalities=cardinalities,

                                        d\_token=128,

                                        n\_blocks=1,

                                        attention\_dropout=0.07799726016810132,

                                        ffn\_d\_hidden=120,

                                        ffn\_dropout=0.4330880728874676,

                                        residual\_dropout=0.12022300234864176,

                                        d\_out=1).to(device)

criterion = nn.BCELoss()

optimizer = optim.AdamW(ft\_t.parameters(), lr = 0.001331121608073689, weight\_decay=1.1527987128232402e-06)

history = {'train\_loss' : [],

           'val\_loss': [],

           'train\_accuracy': [],

           'val\_accuracy': []}

EPOCHS = 100

max\_loss = np.inf

for epoch in range(EPOCHS):

    train\_loss, train\_acc = model\_train(ft\_t, train\_loader, criterion, optimizer, device, None)

    val\_loss, val\_acc,precision, recall, f1, auc = model\_evaluate(ft\_t, val\_loader, criterion, device)

    if val\_loss < max\_loss:

        print(f'[INFO] val\_loss has been improved from {max\_loss:.5f} to {val\_loss:.5f}. Save model.')

        max\_loss = val\_loss

        torch.save(ft\_t.state\_dict(), 'FT-Transformer\_Best.pth')

    print(f'epoch {epoch+1:02d}, loss: {train\_loss:.5f}, accuracy: {train\_acc:.5f}, val\_loss: {val\_loss:.5f}, val\_accuracy: {val\_acc:.5f} \n')

## Improving Performance with Stacking Ensemble Learning

params\_xgb = {'lambda': 0.0001755235232960383, 'alpha': 0.7560958082981974, 'max\_depth': 2, 'eta': 0.36236452158852345, 'gamma': 0.00045468425755523697, 'grow\_policy': 'depthwise'}

model\_xgb = XGBClassifier(\*\*params\_xgb, random\_state=42)

params\_lgbm = {'lambda\_l1': 9.221322080940283, 'lambda\_l2': 0.5874525186545668, 'num\_leaves': 253, 'feature\_fraction': 0.7589395550825638, 'bagging\_fraction': 0.9891427010078347, 'bagging\_freq': 4, 'min\_child\_samples': 99}

model\_lgbm = LGBMClassifier(\*\*params\_lgbm, random\_state=42)

params\_rf = {'n\_estimators': 300, 'max\_depth': 50, 'min\_samples\_split': 6, 'min\_samples\_leaf': 2}

final\_model = RandomForestClassifier(\*\*params\_rf,random\_state=42)

ft\_t.load\_state\_dict(torch.load('FT-Transformer\_Best.pth'))

def model\_initial():

 model\_lgbm.fit(X\_train,y\_train)

    model\_xgb.fit(X\_train,y\_train)

 preds\_lgbm = model\_lgbm.predict(X\_val)

    preds\_xgb = model\_xgb.predict(X\_val)

    preds\_ftt = ftt\_model\_predict(ft\_t, X\_val.values, device)

train\_stack = np.column\_stack((preds\_lgbm,preds\_xgb,preds\_ftt))

final\_model.fit(train\_stack,y\_val)

def ensemble\_model(input\_data):

    input\_preds\_lgbm = model\_lgbm.predict(input\_data)

    input\_preds\_xgb = model\_.predict(input\_data)

    input\_preds\_ftt = ftt\_model\_predict(ft\_t, input\_data, device)

    input\_data\_stack = np.column\_stack((input\_preds\_lgbm, input\_preds\_xgb, input\_data\_stack))

    final\_predictions = final\_model.predict(input\_data\_stack)

    return final\_predictions

preds = ensemble\_model(X\_test.values)

print('Accuracy score:',accuracy\_score(y\_test, preds))

print('Precision score:',precision\_score(y\_test, preds))

print('Recall score:',recall\_score(y\_test, preds))

print('F1 score:',f1\_score(y\_test, preds))

print('AUC score:',roc\_auc\_score(y\_test, preds))

## Understanding Model Behavior with SHAP Interpretations

import shap

explainer = shap.KernelExplainer(ensemble\_model, X\_test)

shap\_values = explainer(X\_test)

shap\_values = explainer.shap\_values(X\_test)

shap.summary\_plot(shap\_values, X\_test.values)

shap.plots.waterfall(shap\_values[0])

shap.plots.beeswarm(shap\_values[0])

Supplemenytary Table 1. Optimized hyperparameter for standalone classification models.

|  |  |
| --- | --- |
| Model | Optimized Hyperparameter |
| RF | {“n\_estimators”: 300, “max\_depth”: 50, “min\_samples\_split”: 6, “min\_samples\_leaf”: 2} |
| LightGBM | {“lambda\_l1”: 9.2213, “lambda\_l2”: 0.5875, “num\_leaves”: 253, “feature\_fraction”: 0.7589, “bagging\_fraction”: 0.9891, “bagging\_freq”: 4, “min\_child\_samples”: 99} |
| XGBoost | {“lambda”: 0.0002, “alpha”: 0.7561, “max\_depth”: 2, “eta”: 0.3624, “gamma”: 0.0005, “grow\_policy”: “depthwise”} |
| CatBoost | {“objective”: “logloss”, “colsample\_bylevel”: 0.0348, “depth”: 2, “boosting\_type”: “plain”, “ bootstrap\_type”: “bayesian”, “bagging\_temperature”: 3.5794, “lr”: 0.0013} |
| FTT | {“d\_token”: 128, “n\_blocks”: 1, “attention\_dropout”: 0.0778, “ffn\_d\_hidden”: 120, “ffn\_dropout”: 0.4330, “residual\_dropout”: 0.1202, “lr”: 0.0013, “weigth\_decay”: 1.1528e-06} |
| TabNet | {“n\_d\_a”: 101, “n\_steps”: 48, “gamma”: 2.4639, “lr”: 0.0060, “cat\_emb\_dim”: 1} |

XGBoost: XGBoost Classifier; CatBoost: CatBoost Classifier; LightGBM: Light Gradient Boosting Classifier; TabNet: TabNet Classifier; FTT: (Feature Tokenizer + Transformer) Classifier; RF: Random Forest Classifier.