Statistical Machine Learning

Assessed Practical

P244 P272 P352

Week 8, Hilary Term 2018

Abstract

In this report, we consider the problem of determining the gender of an individual in a picture. More precisely, using a set of features derived from the original images, we try to fit a machine learning model that can accurately classify individuals as male and female, and represent its confidence in the classification.

The performance of the model was evaluated through a Kaggle competition, in which we submitted predictions as team "The Poisson Fishermen".

Contents

1 The data

1.1 Exploratory Data Analysis

The data available are pre-processed data from male and female pictures. Each picture is represented by a 128-numbers long vector of features and a label: 0 for male, 1 for female.

The labelled training set is composed of 15 000 observations, about half of which are of female individuals. The recorded features are all roughly centred and have standard deviations close to 0.9.

The exploratory data analysis does not show any feature particularly standing out. After a PCA, even the first principal component carries only around 3% of the total variance, which suggests variability is widespread across features. Likewise, the Spearman correlations between individual features and the labels are rather low (all between -0.3 and 0.3 with most much closer to 0).

Despite this lack of interpretability of the features, the data shows good separability. A simple logistic regression gives 92% accuracy when predicting the labels. Likewise, using T-SNE (t-distributed stochastic neighbour embedding), an unsupervised method that give a 2D representation of the dataset, clearly separates most males and females (Figure 1). This is promising for the task at hand, since it means that separation of males and females is quite feasible.

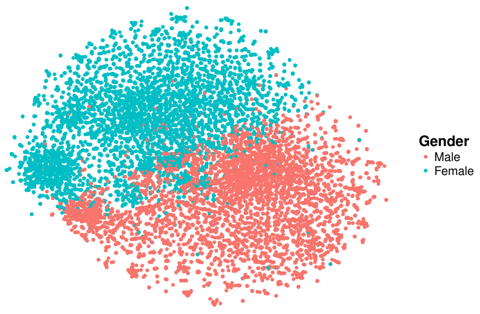

Figure 1: Unsupervised 2-dimensional embedding of the data

1.2 Evaluating of the model

To evaluate the performances of our models, the log loss, aka logistic loss or cross-entropy loss, is used:

$$-\log(y|\hat{y}) = -(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

with y the true label and $\hat{y}$ the predicted one. A particularity of this loss is that high confidence in a wrong prediction is very heavily penalized. This means that not only classification should be accurate, but predictions must also be conservative when there is some doubt on the label.

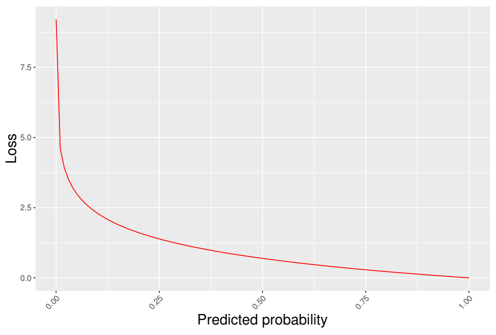

Figure 2: Log loss when the true label is 1

2 Building the model

2.1 Basic classifiers

Given the good separability of the data, we started by experimenting with very simple algorithms. Our first successful attempt was the Quadratic Discriminant Analysis (QDA), which both ran almost instantaneously and yielded an accuracy over 98%. k-nearest neighbours (k-nn) with 10 neighbours had similar precision.

However, in both cases the log loss remained higher than even the simple logistic regression (even though its classification accuracy was lower). This is because the previous algorithms are not well-calibrated: the exact value that they give when predicting a probability is not representative of their true classification performance. This causes them to be over- or under-confident in their probability estimates, and they are thus heavily penalized by the log loss metric. Although some methods exist to calibrate such algorithms (using an isotonic regression for instance), we chose to move on to algorithms that could directly optimize the right metric.

This is the case of the `xgboost` package, which can perform gradient boosting on trees using a log loss. This method showed promising results but failed to consistently bring the average loss below 0.1.

Having gone through those and a number of other standard methods, we turned to deep learning for comparison. After minimal tuning, a Multi-Layer Perceptron (MLP) with one hidden layers gave an average log loss around 0.08. Considering the wide gap between the performance of this algorithm and the others, we chose to focus the rest of our study on neural models.

Before moving on to more complex neural network, we tried to make use of the work mentioned above to derive some useful features for our final model.

2.2 Generating new features

To try to derive interesting features from the data, one possibility is to use the predictions (i.e. the probability of a female picture) from lower-performing models as new covariates for the final model. We chose to use the outputs from the k-nn and QDA for that purpose: since they are not linearly derived from the data, they may provide some information that would take a lot of computation for the neural network to find on its own.

We also tried to obtain condensed features with a simple autoencoder implemented with a multi-layers perceptron. It consists of three layers: the input and the output are composed of 128 nodes, while the hidden layer is composed of 32 neurons. By training the NN with identical input and output, the weights are such that the output of the hidden layer is an encoding that minimises the loss of information. Theoretically, this allows for a more informative representation since the compressed representation from the hidden layer will reduce the noise of the data and keep its most informative, so that the output layer can decode this representation and reconstruct the initial data.

The third path we explored to generate new features was to use non-linear embeddings of the data in lower-dimensional space. As previously mentioned, although linear methods such as the PCA and LDA yielded disappointing separation, more advanced ones such as spectral embedding and T-SNE. We experimented with those, adding the projected coordinates as new covariates for the neural model.

All in all, the most successful addition was that of the QDA, which gave the model a noticeable improvement. Some other artificial features appeared to give a slight boost to the classification but since the difference was extremely small and the computation of these features quite costly, we chose to keep only the QDA.

3 The final model

3.1 Improving the neural network

To improve on the simple MLP model's performances, we worked on building a more advanced neural network using the dedicated PyTorch framework. In particular a gradient descent with momentum (Adam algorithm) significantly improved the model.

Quite quickly, it became apparent that using more than one hidden layer was counter-productive and resulted in strong overfitting. More generally, the main difficulty was to find ways to improve the fit of the model without harming its generalization. When training the model on part of the data and validating on a new set of rows, it became apparent that the performance of the model depended heavily on the split between the training and validation data. This high variability in the performance, associated with the large difference between the training and validation error, called for regularization.

The final neural network has a single hidden layer of 150 units.

The end result of our efforts was to bring our public leaderboard score do 0.074. In local validation, performance varied wildly, from 0.065 to almost 0.08. This is what led us to try to find a way to reduce variability further.

3.2 Preventing overfitting

The main problem of our base network was large overfitting to the training data. To prevent this, we tested multiple possible solutions.

The first, obvious method, is to adjust the optimizer and its *learning rate*. The Adam method is very common in classification networks, but we also tested a simple stochastic gradient descent. We determined the optimal learning rate by cross-validation, taking into account the number of epochs to train the network. But the learning rate is not the only relevant hyperparameter: we can also introduce other regularization methods, such as momentum for SGD and L2 regularisation *(weight decay)* for Adam. Weight decay has given excellent results to limit overfitting, and has improved the neural network results both in overall performance and in variability.

The second important regularization method is to add a *dropout layer* after the dense hidden layer. In our final configuration (determined by cross validation), 50% of the hidden units are randomly set to zero during training. This helps a lot to control the variability of the output.

Another method frequently used to control overfitting is *batch normalisation*. It allows to normalise each minibatch during training. As we normalise the entire dataset before training, it performs a similar operation after the hidden layer. However, neither the performance nor the variability of the log loss was significantly impacted by this additional layer.

3.3 Ensemble of Neural Networks

To limit variability, we also used *ensembling methods*. The general idea is to train several simple neural networks instead of a single large one, and averaging their outputs.

In our model, we train each network on a separate part of the training set. That is, for each network, the training set is itself subdivided into a training and testing set. We can then assess the performance of each network, and the outputs of the networks are averaged to form a final prediction. The final output is the tested on the overall validation set.

Note that the outputs of the networks are averaged before applying the sigmoid activation function. We also use the validation error from the test set of each network: the weight given to each network when averaging is inversely proportional to its validation error. The double cross validation scheme allows us to make the best out of the ensemble method.

3.4 Computational considerations

All the algorithms that we used can be parallelised over multiple processors, which leads to a huge speedup, especially in the ensembling model. We have also tried training the neural networks with a GPU, but the speedup is not significant. The small size of the networks may not lead to a large advantage compared to the cost of moving the data to the GPU before each computation.

4 Conclusion

The results from the ensemble are still quite variable. On average, we achieve a validation error around 0.065, although on the leaderboard the best we achieved was 0.070 — most probably due to overfitting and the variability in the performance of our model. Several other teams surpassed us by a very small margin (about a dozen teams between 0.068 and 0.071) which can be attributed to randomness; but a few did significantly better, enough to prove that some progress is still feasible.

One lead that we could still explore would be to ensemble the results from many kind of models, in order to leverage each one's strengths. Another possibility is to train a model on the residuals of our current model. The role of this secondary model would be to predict the cases when our model is most wrong, and rectify those occurrences.

All in all, besides the technical feat, for most purposes it is plausible that there would be no strong incentive to keep making the model more complex. Indeed, a very small and fast neural network already achieves over 97% accuracy and 0.99 AUC, and less than 0.08 log loss with a very short training time. It seems that we will not get below 0.05 at best with the available data, and it would require significantly more computing for what is a marginal improvement. Apart from cases when this very small advantage proves useful, one will be best served using a basic neural network, or even the Quadratic Discriminant analysis, whose computation is almost instantaneous for very good results.

Python code

deep.py

```python
#!/usr/bin/env python3

"""Main script for the neural network ensembling model.

When run, this script will:
1. Import the data
2. Split it into training and validation sets
3. Preprocess the data using the `preprocessing` module
4. Define the base class for the neural network
5. Train several neural networks using cross-validation
6. Compute a weighted average of the neural networks on the validation
and submission sets

"""

import time

import numpy as np
import pandas as pd

import torch
from torch.utils.data import TensorDataset, DataLoader
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
# import torchvision

from sklearn.model_selection import train_test_split, StratifiedShuffleSplit
from sklearn.metrics import roc_auc_score
# from sklearn.manifold import TSNE
# from MulticoreTSNE import MulticoreTSNE as TSNE

from preprocessing import preprocess


GPU = False  # Set to True if training on GPU is required

start_time = time.time()

print("# Loading data...", end=" ", flush=True)
X = pd.read_csv("data/train.data.csv")
y = pd.read_csv("data/train.labels.csv")
test_data = pd.read_csv("data/test.data.csv")
X = X.values  # Convert the data to a numpy array
y = y.values.ravel()
test_data = test_data.values
print("done.")

print("# Preprocessing")

# print("# t-SNE...", end=" ", flush=True)
# proj = TSNE(n_components=2, n_jobs=4, n_iter=1000,
#             perplexity=10, early_exaggeration=50, learning_rate=100)
# X_proj = proj.fit_transform(X)
# X = np.c_[X, X_proj]
# print("done.")

# Separate the data into training and validation sets
X, X_val, y, y_val = train_test_split(X, y, test_size=0.2, stratify=y)

# Run the preprocessing function
X, y, X_val, test_data = preprocess(X, y, X_val, test_data, verbose=True)


# Neural net definition
class Net(nn.Module):
    """Main neural network for gender classification"""

    def __init__(self, input_size, hidden_size=150):
        """Initialize the network, by creating the required layers"""
        super(Net, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Linear(input_size, hidden_size),  # Dense layer
            nn.BatchNorm1d(hidden_size, momentum=0.2),  # Batch normalization
            nn.Dropout(0.5),  # Dropout 50%
            nn.ReLU()  # Activation function
        )
        # self.layer2 = nn.Sequential(
        #     nn.Linear(150, 64),
        #     nn.BatchNorm1d(64),
        #     nn.Dropout(0.5),
        #     nn.ReLU()
        # )
        self.output = nn.Sequential(
            nn.Linear(hidden_size, 1)  # Output layer: Dense
        )

    def forward(self, x):
        """Evaluate the network on batch x"""
        x = self.layer1(x)
        # x = self.layer2(x)
        x = self.output(x)
        return x.view(-1)


# Training
print("# Training the Neural Networks...", flush=True)
nets = list(range(20))  # Ensemble of NNs
losses = np.zeros(len(nets))  # Loss of each neural net
# Train-test split of the training dataset for each network
skf = StratifiedShuffleSplit(n_splits=len(nets), test_size=0.15)
for k, (train, test) in enumerate(skf.split(X, y)):
    X_train = X[train, :]
    y_train = y[train]
    X_test = X[test, :]
    y_test = y[test]

    print(f"## Training neural net {k}")
    # SUbdivide the training and testing data into batches
    trainset = TensorDataset(torch.Tensor(X_train), torch.Tensor(y_train))
    trainloader = DataLoader(trainset, batch_size=300,
                             shuffle=True, num_workers=2)
    testset = TensorDataset(torch.Tensor(X_test), torch.Tensor(y_test))
    testloader = DataLoader(testset, batch_size=300,
                            shuffle=True, num_workers=2)
    net = Net(X_train.shape[1])  # Create the k-th neural net
    if GPU:
        net.cuda()
    # print(net)
    # Loss function: sigmoid activation + Binary Cross Entropy = log loss
    criterion = nn.BCEWithLogitsLoss()
    # Optimizer: Adam, with appropriate learning rate and L2 regularization
    optimizer = optim.Adam(net.parameters(), lr=0.001, weight_decay=1e-3)
    if GPU:
        criterion.cuda()

    for epoch in range(20):  # Train the network for 20 epochs
        running_loss = 0.0
        running_correct = 0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            if GPU:
                inputs = Variable(inputs.cuda())
                labels = Variable(labels.cuda())
            else:
                inputs = Variable(inputs)
                labels = Variable(labels)
            # Zero out the gradients in the neural network
            optimizer.zero_grad()
            outputs = net(inputs)  # Evaluate the neural net on the inputs
            loss = criterion(outputs, labels.float())  # Compute the loss
            loss.backward()  # Propagate the gradients
            optimizer.step()  # Gradient descent

            running_loss += loss.data[0]
            # Apply activation function (sigmoid) on the output of the NN
            pred = F.sigmoid(outputs).cpu().data.numpy() > .5
            running_correct += np.sum(pred == labels.cpu().data.numpy())
            # if i % 20 == 19:
            #     print(f"[{epoch:2},{i+1:3}] Loss: {running_loss/20:.3f}, "
            #           f"Accuracy: "
            #           f"{100*running_correct/(20*len(outputs)):.1f}%")
            #     running_loss = 0.0
            #     running_correct = 0
        if epoch % 10 == 9:  # Log the training loss and accuracy
            print(f"   [{epoch+1:2}] "
                  f"Loss: {running_loss/((i+1)):.3f}, Accuracy: "
                  f"{100*running_correct/((i+1)*trainloader.batch_size):.1f}%")
            running_loss = 0.0
            running_correct = 0
    val_loss = 0
    correct = 0
    # Compute the log loss and accuracy on the testing set
    for i, data in enumerate(testloader, 0):
        inputs, labels = data
        inputs, labels = Variable(inputs, volatile=True), Variable(labels)
        if GPU:
            inputs.cuda()
            labels.cuda()
        net.eval()
        output = net(inputs)  # Evaluate the neural net on the test input
        # Compute the log loss (sigmoid activation + BCE)
        val_loss += F.binary_cross_entropy_with_logits(
            output, labels.float()).cpu().data[0]
        sigma_output = F.sigmoid(output)
        pred = sigma_output.cpu().data.numpy() > .5
```

```
            correct += np.sum(pred == labels.cpu().data.numpy())
        val_loss /= len(testloader)
        print(f"   -> Test set: Average loss: {val_loss:.4f}, "
              f"Accuracy: {correct}/{((i+1)*testloader.batch_size)} "
              f"({100. * correct/((i+1)*testloader.batch_size):.1f}%)")

        nets[k] = net
        losses[k] = val_loss

    # Validation
    X_val = Variable(torch.Tensor(X_val), volatile=True)
    y_val = Variable(torch.Tensor(y_val))
    if GPU:
        X_val.cuda()
        y_val.cuda()
    output = 0
    # Compute a weighted average of the predictions using the inverse of
    # the log-loss as weights:
    for k, net in enumerate(nets):
        net.eval()
        output += net(X_val) * 1/losses[k]
    output /= np.sum(1/losses)
    val_loss = F.binary_cross_entropy_with_logits(output, y_val).cpu().data[0]
    sigma_output = F.sigmoid(output)
    pred = sigma_output.cpu().data.numpy() > .5
    correct = np.sum(pred == y_val.cpu().data.numpy())
    auc = roc_auc_score(y_val.data, sigma_output.data)
    print(f"\n=> Validation set: Average loss: {val_loss:.4f}, "
          f"ROC AUC: {auc:.4f}, "
          f"Accuracy: {correct}/{len(y_val)} "
          f"({100. * correct/len(y_val):.1f}%)\n")

    # Evaluate on the test data for submission on Kaggle
    test_data = Variable(torch.Tensor(test_data))
    if GPU:
        test_data.cuda()
    output = 0
    for net in nets:
        net.eval()
        output += net(test_data) * 1/losses[k]
    output /= np.sum(1/losses)
    sigma_output = F.sigmoid(output)
    print(sigma_output.cpu().data)

    # Create a Pandas dataframe and save it as a CSV
    submission = pd.DataFrame({'Id': range(1, 15001),
                               'ProbFemale': sigma_output.cpu().data})
    submission = submission[['Id', 'ProbFemale']]
    submission.to_csv("submission.csv", index=False)

    # Log total elapsed time
    time_elapsed = time.time() - start_time
    print(time.strftime("Timing: %Hh %Mm %Ss", time.gmtime(time_elapsed)))
```

autoencoder.py

```python
#!/usr/bin/env python3

"""Autoencoder module.

Defines an autoencoder neural network, along with a function that
trains an autoencoder on the given data and returns it.

"""

import pandas as pd

import torch
from torch.utils.data import TensorDataset, DataLoader
from torch.autograd import Variable
import torch.nn as nn
import torch.optim as optim


class Autoencoder(nn.Module):
    """Autoencoder network.

    We want to extract the hidden layer.
    """
    def __init__(self, input_size, hidden_size):
        super(Autoencoder, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.Threshold(1e-5, 0),  # Set units < 1e-5 to zero
            nn.ReLU()  # Activation function
        )
        self.output = nn.Sequential(
            nn.Linear(hidden_size, input_size)
        )

    def forward(self, x):
        x = self.layer1(x)
        x = self.output(x)
        return x


def train_autoencoder(X, size=32, epochs=30, verbose=0):
    """Train an autoencoder

    Arguments:
        X (array-like): data
        size (int): the size of the hidden layer
        epochs (int): the number of epochs for training
        verbose (int): log level

    Returns:
        The trained Autoencoder object.
    """
    ae_trainset = TensorDataset(torch.Tensor(X), torch.Tensor(X))
    ae_trainloader = DataLoader(ae_trainset, batch_size=256,
                                shuffle=True, num_workers=2)
    autoencoder = Autoencoder(X.shape[1], 32)
    criterion = nn.MSELoss()  # Loss: Mean square error
    # Optimizer: Adadelta with L2 regularization
    optimizer = optim.Adadelta(autoencoder.parameters(),
                                  lr=1.0, rho=0.95, weight_decay=1e-5)
    if verbose == 1:
        print("epoch #", end=" ", flush=True)
    for epoch in range(epochs):
        if verbose > 1:
            running_loss = 0.0
        for i, data in enumerate(ae_trainloader, 0):
            inputs, labels = data
            inputs, labels = Variable(inputs), Variable(labels)
            optimizer.zero_grad()  # Zero out the gradients
            outputs = autoencoder(inputs)  # Evaluate the autoencoder
            loss = criterion(outputs, labels)  # Compute the loss
            loss.backward()  # Propagate the gradients
            optimizer.step()  # Gradient descent
            if verbose > 1:  # Log the training loss
                running_loss += loss.data[0]
                if i % 50 == 49:
                    print(f"[{epoch:3},{i+1:3}] Loss: {running_loss/50:.3f}")
                    running_loss = 0.0
        if verbose == 1:
            print(epoch+1, end=" ", flush=True)
    return autoencoder


if __name__ == "__main__":
    print("# Loading data...", end=" ", flush=True)
    X = pd.read_csv("data/train.data.csv")
    y = pd.read_csv("data/train.labels.csv")
    test_data = pd.read_csv("data/test.data.csv")
    X = X.values
    y = y.values.ravel()
    test_data = test_data.values
    print("done.")

    # Train an autoencoder and returns the values on the hidden layer
    net = train_autoencoder(X, verbose=2)
    print(net.layer1(Variable(torch.Tensor(X))))
```

preprocessing.py

```python
#!/usr/bin/env python3

"""Preprocessing module.

Exports a function to preprocess the data with several possible
methods, including:
- scaling
- autoencoder (using the `autoencoder` module)
- quadratic discriminant analysis
- k-nearest neighbours
- gradient boosting trees (XGBoost)

"""

import numpy as np
import torch
from torch.autograd import Variable
from sklearn.preprocessing import StandardScaler
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.neighbors import KNeighborsClassifier
from xgboost import XGBClassifier

from autoencoder import train_autoencoder


def preprocess(X, y, X_val, test_data, verbose=True, scale=True,
               autoencoder=True, qda=True, knn=False, xgb=False):
    """Preprocess the data by adding features and scaling it.

    For each method, we train the model on the training data using the
    corresponding labels, then apply the same transformation to
    validation and test data.

    Args:
        X (numpy ndarray): Training data
        y (numpy ndarray): Training labels
        X_val (numpy ndarray): Validation data
        test_data (numpy ndarray): Test data for submission
        verbose (bool): log level
        scale (bool): scale the data
        autoencoder (bool): use autoencoder feature
        qda (bool): use Quadratic Discriminant Analysis feature
        knn (bool): use k-nearest neighbours feature
        xgb (bool): use XGBoost feature

    Returns:
        The dataset appropriately transformed by the selected methods.
    """
    if autoencoder:
        if verbose:
            print("## Autoencoder")
            print("### Train...", end=" ", flush=True)
            ae = train_autoencoder(X, size=32, epochs=20, verbose=1)
        else:
            ae = train_autoencoder(X, size=32, epochs=20, verbose=0)
        if verbose:
            print("done.")
            print("### Evaluate...", end=" ", flush=True)
        ae.eval()
```

```
            X_ae = ae.layer1(Variable(torch.Tensor(X))).data
            X = np.c_[X, X_ae]
            X_val_ae = ae.layer1(Variable(torch.Tensor(X_val))).data
            X_val = np.c_[X_val, X_val_ae]
            test_data_ae = ae.layer1(Variable(torch.Tensor(test_data))).data
            test_data = np.c_[test_data, test_data_ae]
            if verbose:
                print("done.")

        if qda:
            if verbose:
                print("## Quadratic Discriminant Analysis...", end=" ", flush=True)
            qdaclf = QuadraticDiscriminantAnalysis(reg_param=0.02)
            qdaclf.fit(X, y)
            X_qda = qdaclf.predict_proba(X)
            X = np.c_[X, X_qda[:, 1]]
            X_val_qda = qdaclf.predict_proba(X_val)
            X_val = np.c_[X_val, X_val_qda[:, 1]]
            test_data_qda = qdaclf.predict_proba(test_data)
            test_data = np.c_[test_data, test_data_qda[:, 1]]
            if verbose:
                print("done.")

        if knn:
            print("## K-Nearest Neighbours...", end=" ", flush=True)
            knnclf = KNeighborsClassifier(n_neighbors=10, p=2, n_jobs=-1)
            knnclf.fit(X, y)
            X_knn = knnclf.predict_proba(X)
            X = np.c_[X, X_knn[:, 1]]
            X_val_knn = knnclf.predict_proba(X_val)
            X_val = np.c_[X_val, X_val_knn[:, 1]]
            test_data_knn = knnclf.predict_proba(test_data)
            test_data = np.c_[test_data, test_data_knn[:, 1]]
            print("done.")

        if xgb:
            print("## XGBoost...", end=" ", flush=True)
            xgbclf = XGBClassifier(max_depth=3, learning_rate=0.1,
                                   n_estimators=1000,
                                   gamma=10, min_child_weight=10,
                                   objective='binary:logistic', n_jobs=4)
            xgbclf.fit(X, y)
            X_xgb = xgbclf.predict_proba(X)
            X_val_xgb = xgbclf.predict_proba(X_val)
            X = np.c_[X, X_xgb[:, 1]]
            X_val = np.c_[X_val, X_val_xgb[:, 1]]
            test_data_xgb = xgbclf.predict_proba(test_data)
            test_data = np.c_[test_data, test_data_xgb[:, 1]]
            print("done.")

        if scale:
            if verbose:
                print("## Scaling...", end=" ", flush=True)
            scaler = StandardScaler()
            X = scaler.fit_transform(X)
            X_val = scaler.transform(X_val)
            test_data = scaler.transform(test_data)
            if verbose:
                print("done.")

        return X, y, X_val, test_data
```