1. Yifan Zhang 51684474

(a)

$$C_1: \begin{cases} +1 & x > a \\ -1 & \text{other} \end{cases} \quad C_2: \begin{cases} +1 & x < b \\ -1 & \text{other} \end{cases} \quad C_3: \{ +1, \ x < +\infty$$

$a < b$, we need to judge the value of $0.5C_3(x) - C_1(x) - C_2(x)$.

① $x \le a$

$0.5C_3(x) - C_1(x) - C_2(x)$

$= 0.5 - (-1) - 1$

$= 0.5$

② $a \le x \le b$

$0.5C_3(x) - C_1(x) - C_2(x)$

$= 0.5 - 1 - 1$

$= -1.5$

③ $x \ge b$

$0.5C_3(x) - C_1(x) - C_2(x)$

$= 0.5 - 1 - (-1)$

$= 0.5$

when $x \le a$, or $x \ge b$, the $f(x)$ is positive.

$f(x)$ is a threshold classifier, because the $f(x)$ can return $+1$ or -1 by the value of x #

(b)

This is because OOB error is a part of bagging method. So we don't need extra codes. And OOB error uses every data to do model training and performance evaluation, the OOB error method doesn't need to split training data and testing data #

2. (a)

Question 2

```
In 10  1  #(a)
       2  from sklearn.datasets import fetch_california_housing
       3  from sklearn.model_selection import train_test_split
       4
       5  data = fetch_california_housing()
       6  print(data.DESCR)
       7
       8  X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,
       9                                                      test_size=0.3,
      10                                                      random_state=50)
```

Executed at 2023.10.14 12:26:28 in 73ms

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

    :Number of Instances: 20640

    :Number of Attributes: 8 numeric, predictive attributes and the target

    :Attribute Information:

The value of d is 8 in this dataset
```

#

(b)

In 11
```
#(b)
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

rf = RandomForestRegressor(n_estimators=100, random_state=50)

rf.fit(X_train, y_train)

y_train_pred = rf.predict(X_train)
y_test_pred = rf.predict(X_test)

train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

print(f'Training MSE: {train_mse}')
print(f'Test MSE: {test_mse}')
```
Executed at 2023.10.14 12:26:49 in 13s 704ms

```
Training MSE: 0.03561237258828922
Test MSE: 0.2643704261029829
```

I hope this won't cause you any inconvenience. Looking forward to your response.The value of m is 8, because we use default hyperparameters, so the max_features is 8.

(C)

In 12
```
#(c)
trees_mse = []
for tree in rf.estimators_:
    y_pred = tree.predict(X_test)
    trees_mse.append(mean_squared_error(y_test, y_pred))

print("MSE for Each Tree in the Test Set: ", trees_mse)
print("MSE for the Entire Forest on the Test Set: ", test_mse)
```
Executed at 2023.10.14 12:27:04 in 210ms

```
MSE for Each Tree in the Test Set:  [0.5492299401658591, 0.5302397605532784, 0.6049794481668281, 0.602028885702487, 0.5947437957668927,
0.6430340320716892, 0.5653728307504038, 0.6013802910773901, 0.5450129434605135, 0.5501374440352228, 0.5672762009702519, 0.5622726671523094,
0.5258212810607558, 0.5702052210381137, 0.5208931283826711, 0.625478728014664, 0.5581362527012274, 0.5805645501952842, 0.594155706095898,
0.5871157760478197, 0.6043121053505168, 0.5761633583569928, 0.607930533690003, 0.6093420094099483, 0.5430173128630813, 0.5709032669198805,
0.5813048339838987, 0.5401538015579296, 0.5882778607182977, 0.6104711977498384, 0.539875976260788, 0.5816350887257105, 0.5848523531307331,
0.5571811815448159, 0.5573483312449774, 0.557944039910998, 0.5419462107537951, 0.5462771541001453, 0.570012139390859, 0.5548241841713663,
0.5260020031309 43, 0.5752917458474484, 0.5670126634930394, 0.5813917092937984, 0.5760583168619186, 0.5499357693941375, 0.5736766030568636,
0.5451472552500322, 0.5780992297623546, 0.561606576464438, 0.5977621740194767, 0.6007376482569928, 0.5894707276436046, 0.5816620031989663,
0.554349996038049, 0.6262230803051033, 0.6056559472432815, 0.5929205906766473, 0.5886772080406977, 0.5703550322495154, 0.5453610392779231,
0.5971695462964631, 0.5952409924689761, 0.6086908845751453, 0.5603233383765504, 0.5742959016875646, 0.547251770634496, 0.5740553610058623,
0.5438541189030524, 0.5814659174075903, 0.5316046526984496, 0.5608903980372738, 0.5985906824239018, 0.5415347226368702, 0.5482515852564599,
0.5868951662474967, 0.6271884934541828, 0.5486584532405685, 0.5849934046366116, 0.5417149131185238, 0.5925992621331717, 0.5523712344748384,
0.5873022618941053, 0.5326184679602551, 0.5503006222430555, 0.5722009066195091, 0.5687434442099968, 0.5528136558831396, 0.6009660923685884,
0.6060363620942021, 0.5635578882474968, 0.54802112889677, 0.5847901830987886, 0.5716677477661014, 0.6093378005148093, 0.5558999161929263,
0.5702055045131945, 0.5834549520683623, 0.5692988724718991, 0.5786533481697351]
MSE for the Entire Forest on the Test Set:  0.2643704261029829
```

(d)

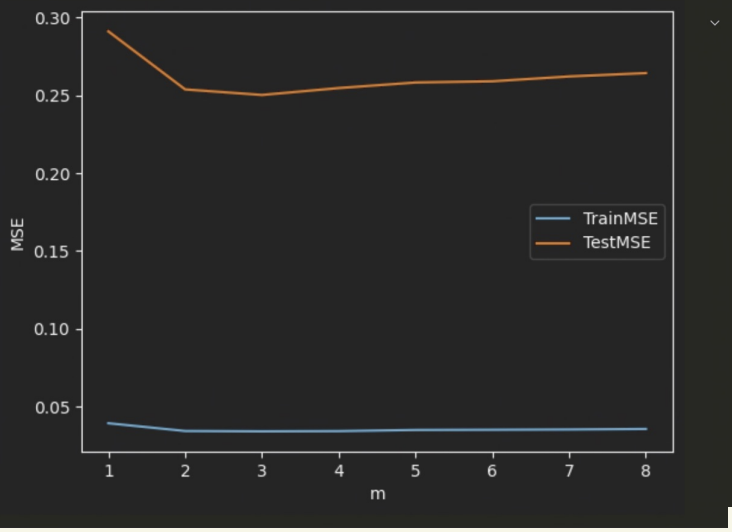
In 13
```
#(d)
import numpy as np
from scipy.stats import pearsonr

predictions = []
for tree in rf.estimators_:
    predictions.append(tree.predict(X_test))

correlations = []
for i in range(len(predictions)):
    for j in range(i+1, len(predictions)):
        corr, _ = pearsonr(predictions[i], predictions[j])
        correlations.append(corr)

mean_corr = np.mean(correlations)
print("the average of all pairwise correlations: ", mean_corr)
```
Executed at 2023.10.14 12:27:18 in 5s 85ms

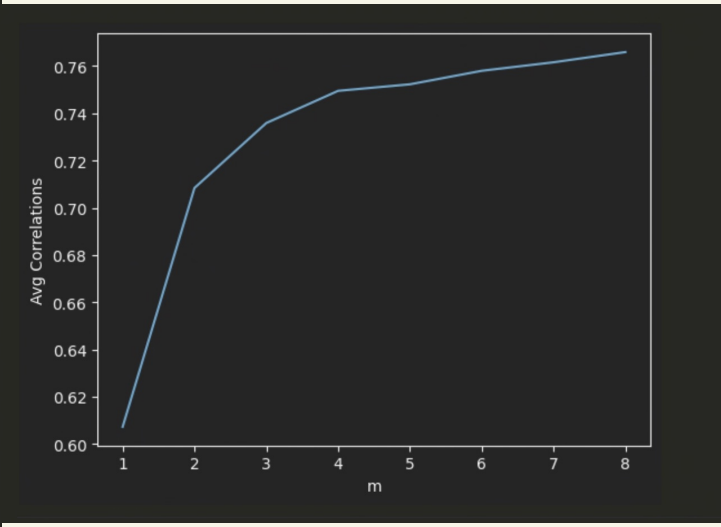
```
the average of all pairwise correlations:  0.7659047740859255
```

(e)

In 14
```python
#(e)
import matplotlib.pyplot as plt
d = 8
mses_train = []
mses_test = []
mean_corrs = []

for m in range(1, d+1):
    rf = RandomForestRegressor(n_estimators=100, max_features=m,
                               random_state=50)
    rf.fit(X_train, y_train)
    y_train_pred = rf.predict(X_train)
    y_test_pred = rf.predict(X_test)
    mse_train = mean_squared_error(y_train, y_train_pred)
    mse_test = mean_squared_error(y_test, y_test_pred)
    mses_train.append(mse_train)
    mses_test.append(mse_test)
    predictions = []
    for tree in rf.estimators_:
        predictions.append(tree.predict(X_test))
    correlations = []
    for i in range(len(predictions)):
        for j in range(i+1, len(predictions)):
            corr, _ = pearsonr(predictions[i], predictions[j])
            correlations.append(corr)
    mean_corr = np.mean(correlations)
    mean_corrs.append(mean_corr)

print("m\tTrain MSE\tTest MSE\tAvg Correlations")
for m, mse_train, mse_test, avg_corr in zip(range(1, d+1), mses_train,
                                            mses_test, mean_corrs):
    print(f"{m}\t{mse_train:.4f}\t\t{mse_test:.4f}\t\t{avg_corr:.4f}")

plt.figure()
plt.plot(range(1, d+1), mses_train, label='TrainMSE')
plt.plot(range(1, d+1), mses_test, label='TestMSE')
plt.xlabel('m')
plt.ylabel('MSE')
plt.legend()
plt.show()


plt.figure()
plt.plot(range(1, d+1), mean_corrs)
plt.xlabel('m')
plt.ylabel('Avg Correlations')
plt.show()
```
Executed at 2023.10.14 12:28:57 in 1m 33s 998ms

m	Train MSE	Test MSE	Avg Correlations
1	0.0393	0.2912	0.6073
2	0.0343	0.2539	0.7084
3	0.0341	0.2503	0.7359
4	0.0342	0.2548	0.7495
5	0.0350	0.2583	0.7523
6	0.0351	0.2591	0.7580
7	0.0353	0.2622	0.7616
8	0.0356	0.2644	0.7659





#

(f)

(f)
The average correlation increases as m increases, this is because in a random forest, the m determines number of features in each splitting node. When m increases the correlation becomes bigger, it means their predictions increases.

(g)

(g)
False, When we choose m, we need to consider the m that corresponds to the smallest MSE value, rather than choosing the smallest m value directly.
Only choosing the m value with the smallest MSE will indicate the best fit.

#

3.

We have $L_\lambda(w) = L(w) + \frac{1}{2}\lambda\|w\|^2$. we need to calculate gradient of $L_\lambda(w)$

$\nabla L_\lambda(w) = \nabla L(w) + \nabla \frac{1}{2}\lambda\|w\|^2$

$= \nabla L(w) + \lambda w$

the gradient of $L(w)$ is $\nabla L(w)$

We can get $\nabla L_\lambda(w) - \nabla L(w) = \lambda w$

$w_{new} = w - \eta \nabla L_\lambda(w)$

$= w - \eta(\nabla L(w) + \lambda w)$

$= w - \eta \nabla L(w) - \eta\lambda w$

$= (1 - \eta\lambda)w - \eta \nabla L(w)$

by $w_{new} = (1-\eta\lambda)w - \eta \nabla L(w)$

we can prof it.

#

4.

(b)

```python
def predict_proba(self, X):
    '''
    Predict the class distributions for given input examples.

    Parameters
    ----------
    X: input examples, represented as an input array of shape (n_sample,
       n_features).

    Returns
    -------
    y: predicted class distributions, represented as an array of shape (n_sample,
       n_classes)
    '''

    # Modify the code below to implement the idea for avoiding numerical flow as described in Q4 (b)
    scores = X @ self.coef_.T + self.intercept_
    scores -= np.max(scores, axis=1).reshape(-1, 1)
    scores = np.exp(scores)
    return scores / scores.sum(axis=1).reshape(-1, 1)
```

```python
def predict(self, X):
    '''
    Predict the classes for given input examples.

    Parameters
    ----------
    X: input examples, represented as an input array of shape (n_sample,
       n_features).

    Returns
    -------
    y: predicted class labels, represented as an array of shape (n_sample,)
    '''

    # replace pass with your code
    proba = self.predict_proba(X)
    return self.classes_[np.argmax(proba, axis=1)]
```

#

(c)

```python
    def fit(self, X, y, lr=0.1, momentum=0, niter=100):
        '''
        Train a multiclass logistic regression model on the given training set.

        Parameters
        ----------
        X: training examples, represented as an input array of shape (n_sample,
           n_features).
        y: labels of training examples, represented as an array of shape
           (n_sample,) containing the classes for the input examples
        lr: learning rate for gradient descent
        niter: number of gradient descent updates
        momentum: the momentum constant (see assignment task sheet for an explanation)

        Returns
        -------
        self: fitted model
        '''
        self.classes_ = np.unique(y)
        self.class2int = dict((c, i) for i, c in enumerate(self.classes_))
        y = np.array([self.class2int[c] for c in y])

        n_features = X.shape[1]
        n_classes = len(self.classes_)

        self.intercept_ = np.zeros(n_classes)
        self.coef_ = np.zeros((n_classes, n_features))

        # Implement your gradient descent training code here; uncomment the code below to do "random training"
        # self.intercept_ = np.random.randn(*self.intercept_.shape)
        # self.coef_ = np.random.randn(*self.coef_.shape)
        for iteration in range(niter):
            softmax = self.predict_proba(X)
            # Compute the gradient of the log-loss
            error = softmax - np.eye(n_classes)[y]
            grad_coef = error.T @ X
            grad_intercept = np.sum(error, axis=0)
            # Update model parameters using gradient descent
            self.coef_ -= lr * grad_coef
            self.intercept_ -= lr * grad_intercept
        return self
```

```python
if __name__ == '__main__':
    X, y = fetch_covtype(return_X_y=True)
    X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, test_size=0.3, random_state=42)

    clf = LogisticRegression()
    clf.fit(X_tr, y_tr)
    print(accuracy_score(y_tr, clf.predict(X_tr)))
    print(accuracy_score(y_ts, clf.predict(X_ts)))
```

Executed at 2023.10.14 12:29:33 in 15s 99ms

```
0.479938924240487
0.4800463557921792
```

#

(d)

```python
if __name__ == '__main__':
    X, y = fetch_covtype(return_X_y=True)
    X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, train_size=0.3,
                                              test_size=0.5, random_state=42)

    learning_rates = [0.0001,0.001, 0.01, 0.1]
    iterations = [50, 100, 500, 800]

    best_log_loss = float('inf')
    best_lr = None
    best_niter = None
    best_model = None

    for lr in learning_rates:
        for niter in iterations:
            clf = LogisticRegression()
            clf, log_losses = clf.fit(X_tr, y_tr, lr=lr, niter=niter)
            avg_log_loss = np.mean(log_losses)
            if avg_log_loss < best_log_loss:
                best_log_loss = avg_log_loss
                best_lr = lr
                best_niter = niter
                best_model = clf

    print(f"Best learning rate: {best_lr}")
    print(f"Best number of iterations: {best_niter}")
    print(f"Best average log-loss: {best_log_loss:.4f}")
    print(f"Training accuracy: {accuracy_score(y_tr, best_model.predict(X_tr)):.4f}")
    print(f"Test accuracy: {accuracy_score(y_ts, best_model.predict(X_ts)):.4f}")
```

Executed at 2023.10.14 12:39:05 in 9m 26s 919ms

```
Best learning rate: 0.0001
Best number of iterations: 800
Best average log-loss: 20.7053
Training accuracy: 0.4753
Test accuracy: 0.4745
```

In order to find the optimal pair of hyperparameters, I have experimented with four different learning rates and four different numbers of iterations. This approach involves varying both the learning rate and the number of iterations to identify the best combination.

$\beta(-\eta g_1)$

(e)

when $t=2$, $w_{2+1} = w_3 = w_2 - \eta g_2 + \beta(w_2 - w_1)$

$= w_2 - \eta g_2 - \beta\eta g_1$

$= w_2 - \eta(g_2 + \beta g_1)$

we assume $w_{t+1} = w_t - \eta(g_t + \beta g_{t-1} + \cdots + \beta^{t-1} g_1)$

$w_{t+2} = w_{t+1} - \eta g_{t+1} + \beta(w_{t+1} - w_t)$

$= \left[w_t - \eta(g_t + \beta g_{t-1} + \cdots + \beta^{t-1} g_1)\right] - \eta g_{t+1} + \beta\left([w_t - \eta(g_t + \beta g_{t-1} + \cdots + \beta^{t-1} g_1)] - w_t\right)$

$= w_t - \eta(g_t + \beta_{t-1} + \cdots + \beta^{t-1} g_1) - \eta g_{t+1} - \beta\eta(g_t + \beta g_{t-1} + \cdots + \beta^{t-1} g_1)$

$= w_{t+1} - \eta(g_{t+1} + \beta(g_t + \cdots + \beta^{t-1} g_1))$

$= w_{t+1} - \eta(g_{t+1} + \beta g_t + \cdots + \beta^t g_1)$

by mathematical induction we can prof $w_{t+1} = w_t - \eta(g_t + \beta g_{t-1} + \cdots + \beta^{t-1} g_1$ #.

(f)

```python
    def fit(self, X, y, lr=0.0001, momentum=0.9, niter=800):
        best_log_loss = float('inf')

        log_loss_list = []

        self.classes_ = np.unique(y)
        self.class2int = dict((c, i) for i, c in enumerate(self.classes_))
        y = np.array([self.class2int[c] for c in y])

        n_features = X.shape[1]
        n_classes = len(self.classes_)

        self.intercept_ = np.zeros(n_classes)
        self.coef_ = np.zeros((n_classes, n_features))

        X_tensor = torch.tensor(X, dtype=torch.float32)
        y_tensor = torch.tensor(y, dtype=torch.long)

        self.model = torch.nn.Linear(n_features, n_classes)

        optimizer = optim.SGD(self.model.parameters(), lr=lr, momentum=momentum)

        loss_fn = torch.nn.CrossEntropyLoss()

        for iteration in range(niter):
            optimizer.zero_grad()
            output = self.model(X_tensor)
            loss = loss_fn(output, y_tensor)
            loss.backward()
            optimizer.step()

            with torch.no_grad():
                y_pred_proba = torch.softmax(output, dim=1)
                training_log_loss = log_loss(y_tensor.numpy(), y_pred_proba.numpy(), labels=np.arange(n_classes))
                log_loss_list.append(training_log_loss)

            if training_log_loss < best_log_loss:
                best_log_loss = training_log_loss

        avg_log_loss = sum(log_loss_list) / len(log_loss_list)

        with torch.no_grad():
            self.intercept_ = self.model.bias.numpy()
            self.coef_ = self.model.weight.numpy()

        self.training_log_loss = avg_log_loss
        return momentum, avg_log_loss
```

```python
if __name__ == '__main__':
    X, y = fetch_covtype(return_X_y=True)
    X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, test_size=0.3, random_state=42)

    best_overall_momentum = None
    best_overall_log_loss = float('inf')
    best_overall_train_acc = 0.0
    best_overall_test_acc = 0.0

    clf = LogisticRegression()

    # Loop through different momentum values
    for momentum in [0.1, 0.5, 0.9, 0.99]:  # Example momentum values
        best_momentum, best_log_loss = clf.fit(X_tr, y_tr, lr=0.0001, momentum=momentum, niter=800)

        if best_log_loss < best_overall_log_loss:
            best_overall_log_loss = best_log_loss
            best_overall_momentum = best_momentum

        # Compute train and test accuracies
        train_preds = clf.predict(X_tr)
        test_preds = clf.predict(X_ts)
        train_acc = accuracy_score(y_tr, train_preds)
        test_acc = accuracy_score(y_ts, test_preds)

        if train_acc > best_overall_train_acc:
            best_overall_train_acc = train_acc
        if test_acc > best_overall_test_acc:
            best_overall_test_acc = test_acc

        print(f"Momentum: {momentum}, Minimum Training Log-Loss: {best_log_loss}")
        print(f"Train Accuracy: {train_acc}, Test Accuracy: {test_acc}")

    print(f"Best Momentum: {best_overall_momentum}, Overall Minimum Training Log-Loss: {best_overall_log_loss}")
    print(f"Overall Best Train Accuracy: {best_overall_train_acc}, Overall Best Test Accuracy: {best_overall_test_acc}")
```

Executed at 2023.10.14 18:48:11 in 8m 23s 280ms

```
Momentum: 0.1, Minimum Training Log-Loss: 9.490934452604215
Train Accuracy: 0.4761524238520019, Test Accuracy: 0.47596727556453094
```

```
Momentum: 0.5, Minimum Training Log-Loss: 9.454841827561578
Train Accuracy: 0.36372286751182664, Test Accuracy: 0.36407655590233157
```

```
Momentum: 0.9, Minimum Training Log-Loss: 9.333311641419847
Train Accuracy: 0.33188675905071946, Test Accuracy: 0.33361253901230037
```

```
Momentum: 0.99, Minimum Training Log-Loss: 8.70195239409016
Train Accuracy: 0.5938978333349725, Test Accuracy: 0.5937557371029925
Best Momentum: 0.99, Overall Minimum Training Log-Loss: 8.70195239409016
Overall Best Train Accuracy: 0.5938978333349725, Overall Best Test Accuracy: 0.5937557371029925
```

By (d) when got the best learning rate and iterater number So we just use different momentum to generate result. #

(g)

```python
    def fit(self, X, y, lr=0.1, momentum=0, niter=100):
        '''
        Train a multiclass logistic regression model on the given training set.

        Parameters
        ----------
        X: training examples, represented as an input array of shape (n_sample,
           n_features).
        y: labels of training examples, represented as an array of shape
           (n_sample,) containing the classes for the input examples
        lr: learning rate for gradient descent
        niter: number of gradient descent updates
        momentum: the momentum constant (see assignment task sheet for an explanation)

        Returns
        -------
        self: fitted model
        '''
        np.random.seed(50)
        self.classes_ = np.unique(y)
        self.class2int = dict((c, i) for i, c in enumerate(self.classes_))
        y = np.array([self.class2int[c] for c in y])

        n_features = X.shape[1]
        n_classes = len(self.classes_)

        self.intercept_ = np.zeros(n_classes)
        self.coef_ = np.zeros((n_classes, n_features))

        log_losses = []
        # Implement your gradient descent training code here; uncomment the code below to do "random training"
        self.intercept_ = np.random.randn(*self.intercept_.shape)
        self.coef_ = np.random.randn(*self.coef_.shape)
        for iteration in range(niter):
            softmax = self.predict_proba(X)



            error = softmax - np.eye(n_classes)[y]
            grad_coef = error.T @ X
            grad_intercept = np.sum(error, axis=0)

            adjusted_softmax = softmax + 1e-15
            log_loss = -np.log(adjusted_softmax[range(len(y)), y]).mean()
            log_losses.append(log_loss)

            self.coef_ -= lr * grad_coef
            self.intercept_ -= lr * grad_intercept
        self.log_losses = log_losses
        return self, self.log_losses
```

```python
if __name__ == '__main__':
    X, y = fetch_covtype(return_X_y=True)



    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, train_size=0.3,
                                              test_size=0.5, random_state=42)

    learning_rates = [0.001, 0.01, 0.1, 1]
    iterations = [50, 100, 500, 1000]

    best_log_loss = float('inf')
    best_lr = None
    best_niter = None
    best_model = None

    for lr in learning_rates:
        for niter in iterations:
            clf = LogisticRegression()
            clf, log_losses = clf.fit(X_tr, y_tr, lr=lr, niter=niter)
            avg_log_loss = np.mean(log_losses)
            if avg_log_loss < best_log_loss:
                best_log_loss = avg_log_loss
                best_lr = lr
                best_niter = niter
                best_model = clf

    print(f"Best learning rate: {best_lr}")
    print(f"Best number of iterations: {best_niter}")
    print(f"Best average log-loss: {best_log_loss:.4f}")
    print(f"Training accuracy: {accuracy_score(y_tr, best_model.predict(X_tr)):.4f}")
    print(f"Test accuracy: {accuracy_score(y_ts, best_model.predict(X_ts)):.4f}")
```

```
Executed at 2023.10.14 12.49.21 in 10m 13s 857ms

Best learning rate: 0.001
Best number of iterations: 1000
Best average log-loss: 12.3414
Training accuracy: 0.5955
Test accuracy: 0.5921
```

It is evident that by standardizing the features, i.e., scaling them to a mean of 0 and a standard deviation of 1, the accuracy of training and testing is significantly improved. The best models trained on data that have been normalized for features exhibit significantly lower mean log-loss compared to models trained on un-normalized data.