DATA7703, Assignment 2

2022 Semester 2

Question 1

(a) $c_1(x) = 1\ only\ when\ x > a$

$c_2(x) = 1\ only\ when\ x < b$

$c_3(x) = 1\ only\ when\ x < +\infty$

To make $f(x) = I(0.1c_3(x) - c_1(x) - c_2(x) > 0)$ classified as positive, we should ensure that$0.1c_3(x) - c_1(x) - c_2(x) > 0$ is true.

Since $0.1c_3(x)$ would always equal to 0.1, we could simply divide this inequation into 3 situations:

- Both $c_1(x)$ and $c_2(x)$ are equal -1: $b \le x \le a$
- $c_1(x) = 1$ and $c_2(x) = -1$: $x > a\ \&\ x \ge b$
- $c_1(x) = -1$ and $c_2(x) = 1$: $x < b\ \&\ x \le a$

All of the above 3 situations are depends on the value of a and b.

(b) Validation set method requires additional data, and cross-validation is computationally expensive. The OOB error is calculated by all basis models trained without it and compute a prediction on it using these models, while cross-validation would be using all models together.

(c) i. False. Essentially speaking, wagging is a variant of bagging algorithm. In principle bagging is performed to reduce variance of fitted values as it increases the stability of the fitted values. Bagging allows us to approximate relative complex response surfaces by practically smoothing over the learners' decision boundaries. the magnitudes of the bias are roughly the same for the bagged and the original procedure. So, we cannot assert that wagging has a smaller bias than bagging.

ii. I don't know.

Question 2

(a) Answer: d = 8

Code:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

dataset = fetch_california_housing()
d = dataset.data.shape[1]
X = dataset.data
y = dataset. Target
X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, test_size=
0.3, random_state= 617)
```

(b) Answer:

Training set MSE is 0.03624832665289494

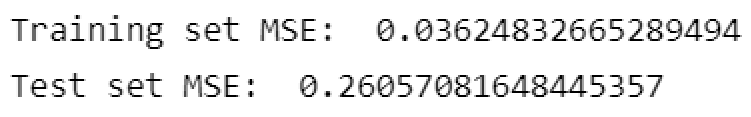
Test set MSE is 0.26057081648445357

Default hyperparameter value is d, so m = d = 8

Code:

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

default_model = RandomForestRegressor(n_estimators= 100,
random_state= 617)
default_model.fit(X_tr, y_tr)
y_training_pred = default_model.predict(X_tr)
training_MSE = mean_squared_error(y_tr, y_training_pred)
print("Training set MSE: ", training_MSE)

y_test_pred = default_model.predict(X_ts)
test_MSE = mean_squared_error(y_ts, y_test_pred)
print("Test set MSE: ", test_MSE)
```

Output:



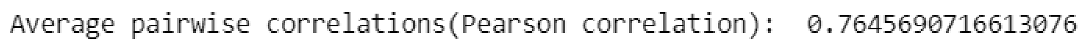
```
Training set MSE:  0.03624832665289494
Test set MSE:  0.26057081648445357
```

(c) Answer: Average of all these pairwise correlations: 0.7645690716613076

Code:

```
   import numpy as np
   from scipy.stats import pearsonr
   estimators = default_model.estimators_
   pearson_correlation = []
   for i in range(len(estimators)):
       for j in range(i + 1, len(estimators)):
           pred_1 = estimators[i].predict(X_ts)
           pred_2 = estimators[j].predict(X_ts)

           pccs = pearsonr(pred_1, pred_2)[0]
           pearson_correlation. append(pccs)
   print("Average pairwise correlations(Pearson correlation): ", np.
mean(pearson_correlation))
```

Output:



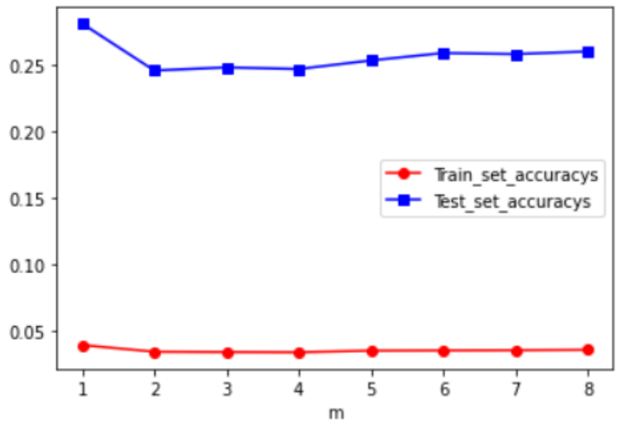
```
Average pairwise correlations(Pearson correlation):  0.7645690716613076
```

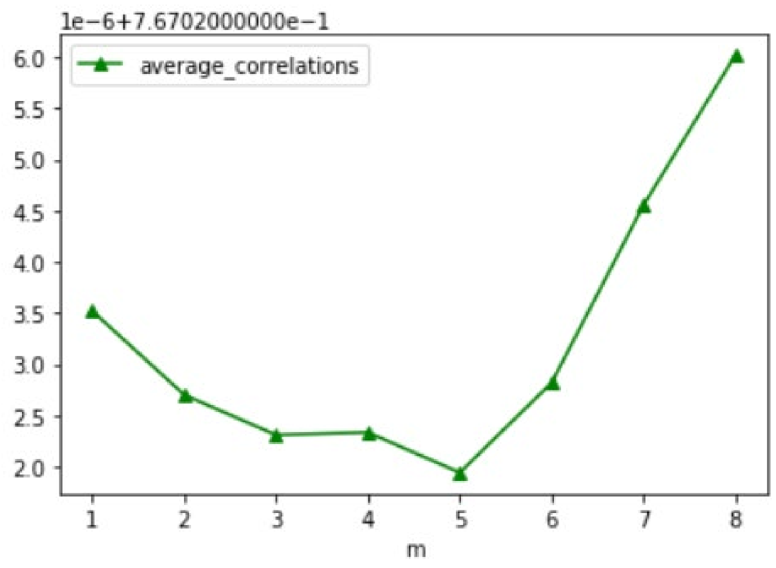
(d) Code:

```
from tabulate import tabulate
import matplotlib.pyplot as plt
training_MSEs = []
test_MSEs = []
average_correlations = []
table_value = []
col_names = ["m", "Training set MSE", "Test set MSE", "Average
correlation"]
for i in range(1, X. shape[1] + 1):
    RF_model = RandomForestRegressor(n_estimators=100,
random_state=617, max_feature=i)
    RF_model.fit(X_tr, y_tr)
    y_training_pred = RF_model.predict(X_tr)
    training_MSE = mean_squared_error(y_tr, y_training_pred)
    training_MSEs.append(training_MSE)

    y_test_pred = RF_model.predict(X_ts)
    test_MSE = mean_squared_error(y_ts, y_test_pred)
    test_MSEs.append(test_MSE)

    for j in range(len(estimators)):
        for k in range(i + 1, len(estimators)):
           pred_1 = estimators[j].predict(X_ts)
           pred_2 = estimators[k].predict(X_ts)
           pccs = pearsonr(pred_1, pred_2)[0]
           pearson_correlation.append(pccs)
    average_correlation = np.mean(pearson_correlation)
    average_correlations.append(average_correlation)
    table_value.append([i, training_MSE, test_MSE,
average_correlation])
print(tabulate(table_value, headers=col_names))

plt.plot(range(1, X.shape[1] + 1), training_MSEs, 'ro-',
label="Trainint set MSEs")
plt.plot(range(1, X.shape[1] + 1), test_MSEs, 'bs-', label="Test
set MSEs")
plt.legend(loc='best')
plt.xlabel('m')
plt.show()

plt.plot(range(1, X.shape[1] + 1), average_correlations, 'g^-',
label="average_correlations”)
plt.legend(loc='best')
plt.xlabel('m')
plt.show()
```

Output:

```
  m    Training set MSE    Test set MSE    Average correlation
---  ------------------  --------------  ---------------------
  1           0.0398142        0.281533               0.767024
  2           0.0348184        0.246248               0.767023
  3           0.0345965        0.248642               0.767022
  4           0.0344288        0.247401               0.767022
  5           0.0356193        0.253843               0.767022
  6           0.0357838        0.259419               0.767023
  7           0.035924         0.258629               0.767025
  8           0.0362483        0.260571               0.767026
```





(e) The average correlation increases as m increases. When m is small, many decision trees use different features to train the model, which leads to low correlation. As m increases, more and more decision trees use same features to train the model, and then their correlations will logically increase.

(f) False. Recall the definition of Prediction error: Prediction errors can be decomposed into two main subcomponents of interest: error from bias, and error from variance. Although a smaller m will make the variance smaller, the bias will increase due to the Bias-variance tradeoff, so we will get a high prediction error.

Question 3

(a) $\lambda w_{t+1} = \lambda w_t - \eta_t \nabla L_\lambda(w_t) = \lambda w_t - \eta_t \nabla(\mathrm{L}(w_t) + \frac{1}{2}\lambda * ||w_t||^2)$

Since: $\nabla\left(\frac{1}{2}\lambda * \left||w_t|\right|^2\right) = \frac{1}{2}\lambda * 2w_1 + \frac{1}{2}\lambda * 2w_2 + \cdots + \frac{1}{2}\lambda * 2w_n = \lambda * w$

So: $\nabla L_\lambda(w_t) = \nabla\left(L(\mathrm{w}) + \frac{1}{2}\lambda * \left||w|\right|^2\right) = \nabla L(w_t) + \lambda * w$

$$\lambda w_{t+1} = \lambda w_t - \eta_t \nabla L_\lambda(w_t) = \lambda w_t - \eta_t(\nabla \mathrm{L}(w_t) + \lambda * \mathrm{w})$$

(b) Assume o_c is the largest output of an output vector $(o_1, o_2, \dots, o_c)$, when we use scaled softmax function, the probability of o_c is

$$\frac{e^{\beta o_c}}{e^{\beta o_1} + e^{\beta o_2} + \cdots + e^{\beta o_c}} = \frac{e^{\beta o_c}}{\sum_{i=1}^{c} e^{\beta o_i}}$$

Suppose $\beta_1 > \beta_2 > 0$, we have: $p_1 = \frac{e^{\beta_1 o_c}}{\sum_{i=1}^{c} e^{\beta_1 o_i}}$ and $p_2 = \frac{e^{\beta_2 o_c}}{\sum_{i=1}^{c} e^{\beta_2 o_i}}$

Multiply the numerator and denominator of p2 by $e^{(\beta_1-\beta_2)o_c}$:

$$p_2 = \frac{e^{\beta_1 o_c}}{\sum_{i=1}^{c} e^{\beta_1 o_c + \beta_2 (o_i - o_c)}}$$

Since o_c is the largest output, compare P1 to the denominator of P2:

$\beta_1(o_i - o_c) < \beta_2(o_i - o_c)$, thus $p_2 = \frac{e^{\beta_1 o_c}}{\sum_{i=1}^{c} e^{\beta_1 o_c + \beta_2 (o_i - o_c)}} < \frac{e^{\beta_1 o_c}}{\sum_{i=1}^{c} e^{\beta_1 o_i}} = p_1$

We can conclude that: when β increases, the probability of the class with the largest output value increases.

Question 4

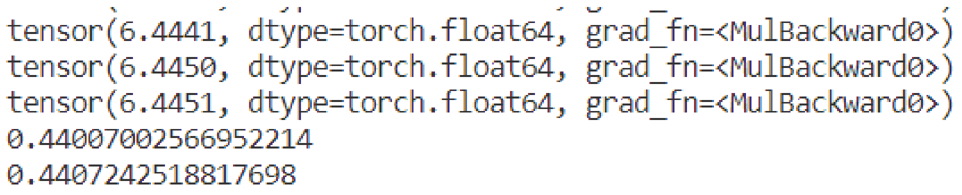
(b) Code:

```
def predict_proba(self, X):
        X = torch.from_numpy(X)
        outputs = X @ self.w.T + self.b
        output = outputs - torch.max(outputs, 1)[0].reshape(-1, 1)
        exp = output.exp()
        sum_exp = exp.sum(dim=1, keepdim=True)
        softmax = exp / sum_exp
        return softmax

def predict(self, X):
        probs = self.predict_proba(X)
        labels = []
        for i in range(len(probs)):
            x = torch.argmax(probs[i]).item() + 1
            labels.append(x)

        return labels
```

(c) Code:

```
def fit(self, X, y, lr=0.1, momentum=0, niter=100):
        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 = X.shape[0]
        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)

        w = torch.from_numpy(self.coef_)
        b = torch.from_numpy(self.intercept_)
        w.requires_grad = True
        b.requires_grad = True
        self.w = w
        self.b = b

        X = torch.from_numpy(X)
        Y_onehot = onehot_encoder.fit_transform(y.reshape(-1, 1))
        Y_onehot = torch.from_numpy(Y_onehot)

        for i in range(niter):
            scores = X @ w.T + b
            score = scores - torch.max(scores, 1)[0].reshape(-1, 1)

            exp = score.exp()
            sum_exp = exp.sum(dim=1, keepdim=True)
            softmax = exp / sum_exp

            loss = -(1 / n) * torch.sum(torch.log(softmax + 1e-5) *
Y_onehot)

            if w.grad is not None:
                w.grad.zero_()
            if b.grad is not None:
                b.grad.zero_()

            loss.backward()

            w.data.add_(-lr * w.grad.data)
            b.data.add_(-lr * b.grad.data)

            print(loss)
        return self
```

(d) When learning rate is 0.1 and learning iteration is 100, the log-loss of the model is:

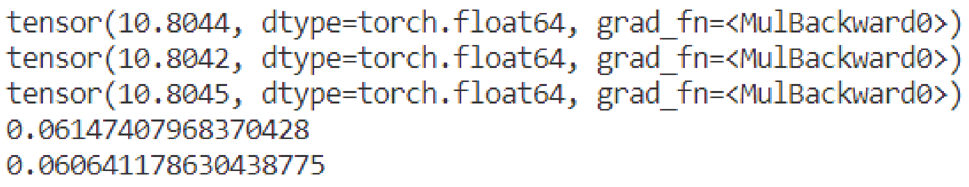


```
tensor(6.4441, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(6.4450, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(6.4451, dtype=torch.float64, grad_fn=<MulBackward0>)
0.44007002566952214
0.4407242518817698
```

The training set accuracy is 0.44007002566952214

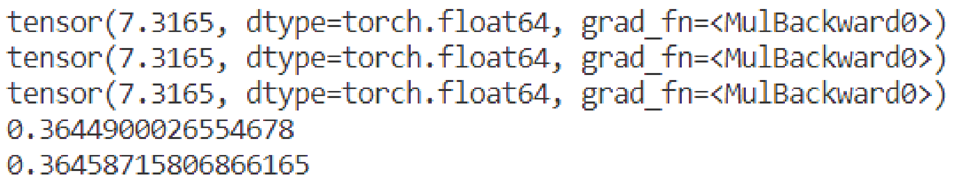
The test set accuracy is 0.4407242518817698

Let's change learning rate to 0.15 and keep learning iteration unchanged, the log-loss of the model is:



```
tensor(10.8044, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(10.8042, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(10.8045, dtype=torch.float64, grad_fn=<MulBackward0>)
0.06147407968370428
0.060641178630438775
```

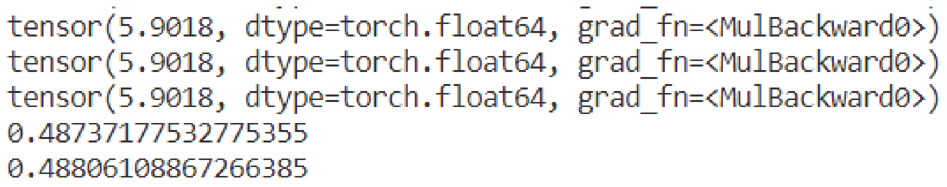
Situation (a) log-loss unchanged



```
tensor(7.3165, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(7.3165, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(7.3165, dtype=torch.float64, grad_fn=<MulBackward0>)
0.3644900026554678
0.36458715806866165
```

Situation (b) log-loss still high

Let's change learning rate to 0.05 and change the learning iteration to 200 to make sure we could get a converge result



```
tensor(5.9018, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(5.9018, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(5.9018, dtype=torch.float64, grad_fn=<MulBackward0>)
0.48737177532775355
0.48806108867266385
```

The training set accuracy is 0.48737177532775355

The test set accuracy is 0.48806108867266385

With the above different log-loss of different hyperparameter, we find that when learning rate is greater than 0.1, log-loss either remains unchanged or quickly converges to a local optimum (but still larger than when learning rate is 0.1). When the learning rate is less than 0.1, the log-loss becomes smaller, but we should keep in mind that the learning iterations should be increase at the same time.

(e) Because $w_1 = 0$

So $w_2 = w_1 - \eta g_1 = -\eta g_1$

Since $w_{t+1} = w_t - \eta g_t + \beta(w_t - w_{t-1})$

We have $w_3 = w_2 - \eta g_2 + \beta(w_2 - w_1) = w_2 - \eta g_2 - \beta\eta g_1$

$$w_4 = w_3 - \eta g_3 + \beta(w_3 - w_2) = w_3 - \eta g_3 - \beta\eta g_2 - \beta^2\eta g_1$$

By the same token, we can obtain that: for any $t \geq 2$, we have

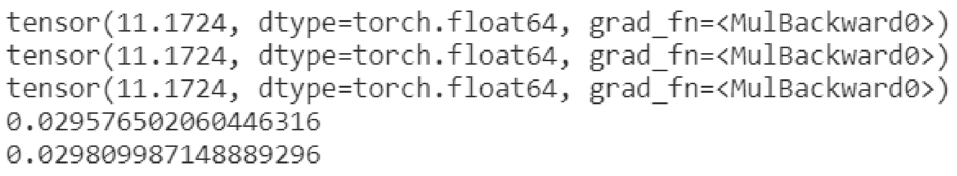
$$w_{t+1} = w_t - \eta(g_t + \beta g_{t-1} + \cdots + \beta^{t-1} g_1)$$

(f) Code:

```
def fit(self, X, y, lr=0.05, momentum=0.1, niter=200):
        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 = X.shape[0]
        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)

        w = torch.from_numpy(self.coef_)
        b = torch.from_numpy(self.intercept_)
        w.requires_grad = True
        b.requires_grad = True
        self.w = w
        self.b = b

        X = torch.from_numpy(X)
        Y_onehot = onehot_encoder.fit_transform(y.reshape(-1, 1))
        Y_onehot = torch.from_numpy(Y_onehot)
        # subquestion (f)
        optimizer = optim.SGD([w, b], lr=lr, momentum=momentum)
        for i in range(niter):
            scores = X @ w.T + b
            score = scores - torch.max(scores, 1)[0].reshape(-1, 1)

            exp = score.exp()
            sum_exp = exp.sum(dim=1, keepdim=True)
            softmax = exp / sum_exp

            loss = -(1 / n) * torch.sum(torch.log(softmax + 1e-5) *
Y_onehot)

            if w.grad is not None:
                w.grad.zero_()
            if b.grad is not None:
                b.grad.zero_()

            # subquestion (f)
            optimizer.zero_grad()

            loss.backward()

            w.data.add_(-lr * w.grad.data)
            b.data.add_(-lr * b.grad.data)

            # subquestion (f)
            optimizer.step()
            print(loss)

        return self
```

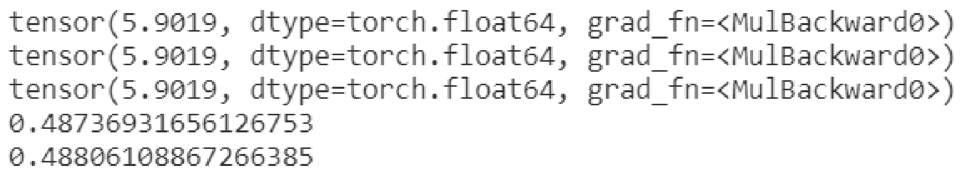
Output:

When momentum is 0, learning rate is 0.05, learning iteration is 200, the log-loss, train and test accuracy are as below:



```
tensor(11.1724, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(11.1724, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(11.1724, dtype=torch.float64, grad_fn=<MulBackward0>)
0.029576502060446316
0.029809987148889296
```

Change the momentum to 0.1, the log-loss, train and test accuracy are as below:



```
tensor(5.9019, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(5.9019, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(5.9019, dtype=torch.float64, grad_fn=<MulBackward0>)
0.48736931656126753
0.48806108867266385
```

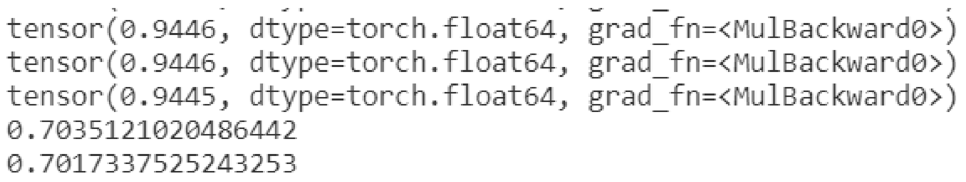
We could find that: the greater the momentum, the more stable convergence while the training and test accuracy remain the same with did not apply momentum.

(g) Code:

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

    # normalize
    X = StandardScaler().fit_transform(X)
    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)))
```

Output:

After apply normalization, log-loss decreases faster and converges to a quite small log-loss. Let's change the learning rate to 0.4, momentum to 0.1, learning iteration to 500, the log-loss, training and test accuracy of this model are as below:



```
tensor(0.9446, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(0.9446, dtype=torch.float64, grad_fn=<MulBackward0>)
tensor(0.9445, dtype=torch.float64, grad_fn=<MulBackward0>)
0.7035121020486442
0.7017337525243253
```