Xuhong Ye

46173957

1.

(a) (10 marks) Consider a regression MLP that uses identity activation functions for all neurons. A data scientist trains the neural network to minimize the MSE, and he observes a much smaller training set MSE for the MLP as compared to that for OLS. Is this possible? Justify your answer.

No, this is not possible. If a regression MLP that uses identity activation functions, this is actually a linear regression. Thus, its effect is equivalent to OLS. Because MLP and OLS both do linear regression, their effects are exactly the same.

(a)

MLP with identity activation functions: $f(x) = \Sigma(w_0 + wx)$

OLS: $g(x) = w_0 + wx$

Therefore, their MSE is: $MSE = (y - \hat{y})^2$

$= (y - (w_0 + wx))^2$

After taking the derivation of MSE, I find that the results of MLP and OLS are the same,

as $\frac{\partial MSE}{\partial w} = 2yx + 2w_0x + 2x^2w$

(b) (5 marks) Show that a gradient descent step on L_λ (w) is equivalent to first multiplying w by a constant, and then moving along the negative gradient direction of the original MSE
loss L(w).

(b)

the gradient descent direction of original $L(w)$ is:

$$\nabla L(w) = \frac{\partial L(w)}{\partial w}$$

After regularizing $L(w)$, we get $L_\lambda(w)$, so the gradient descent direction of it is:

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

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

$$= \frac{\partial L(w)}{\partial w} + \lambda \|w\|$$

Because $\nabla L(w) = \frac{\partial L(w)}{\partial w}$, $\nabla L_\lambda(w) = * \nabla L(w) + \lambda \|w\|$

(c) (10 marks) Show that when β increases, the probability of the class with the largest output value increases.

(c)

I assume that the class with largest output value is O_{max} and the other output is O_n.

And then, putting O_{max} into the scaled softmax function:

$$\text{softmax}_\beta = \frac{e^{\beta O_{max}}}{z_\beta}$$

Putting O_n into the scaled softmax function:

$$\text{other softmax}_\beta = \frac{e^{\beta O_n}}{z_\beta}$$

Therefore, $$\frac{\text{softmax}_\beta}{\text{other softmax}_\beta} = \frac{e^{\beta O_{max}}}{e^{\beta O_n}} = e^{\beta (O_{max} - O_n)}$$

$$\ln\left(\frac{\text{softmax}_\beta}{\text{other softmax}_\beta}\right) = \beta (O_{max} - O_n)$$

From the formula I listed above, we could find that when β increases, $\beta(O_{max} - O_n)$ increases.

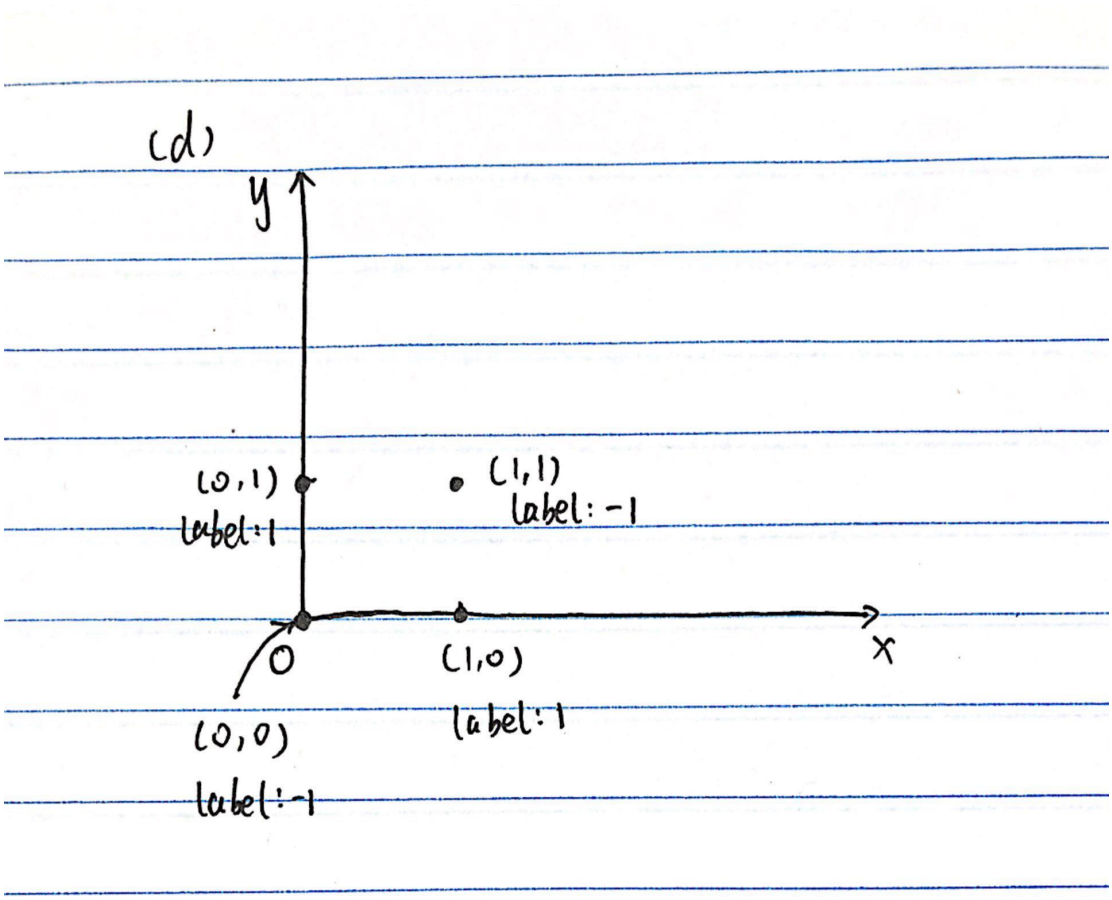
This means $\frac{\text{softmax}_\beta}{\text{other softmax}_\beta}$ increases.

Therefore, when β increases, the probability of the class with the largest output value increases.

(d) (5 marks) Can you find a perceptron to correctly classify all these examples? Can you find w1,w2 correctly classifies all the four examples? Justify your answer.

I can't find a perceptron to correctly classify all these examples and I also can't find w1,w2 correctly classifies all the four examples.

I drew these four examples in the coordinate system and labeled the corresponding points. The picture I have listed below:



Because the formula of perceptron is a linear equation, but from the picture I listed above, we can't use a straight line to clearly classify two labels. This means that we can't find a linear perceptron to correctly classify all these examples. Therefore, I can't find w1,w2 correctly classifies all the four examples.

2.
(b) (5 marks) Implement the predict function and the predict_proba in logistic_regression.py.

```
    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 lables, represened as an array of shape (n_sample,
            n_classes)
        '''

        # replace pass with your code
        oi_value = np.dot(X, self.coef_.T) + self.intercept_    #linear regression
        max_oi_value = np.max(oi_value, axis=1, keepdims=True)
        oi_value -= max_oi_value

        oi_value = np.exp(oi_value) # e^{o_y}

        z = np.sum(oi_value, axis=1, keepdims=True) # \sum_{y'} e^{o_{y'}}

        oi_value += 1e-15 # take 15 decimal places and make it as small as possible
but not equal to 0
        prob = oi_value / z
        return prob

    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 lables, represened as an array of shape (n_sample,)
        '''

        # replace pass with your code
        probability = self.predict_proba(X)
        pred = np.argmax(probability, axis=1) + 1
        return pred
```

(c) (10 marks) Implement the fit function in logistic_regression.py to support train ing a logistic regression model by using gradient-descent to minimize the log-loss.

```
    def fit(self, X, y, lr=0.1, momentum=0, niter=100):
        '''
```

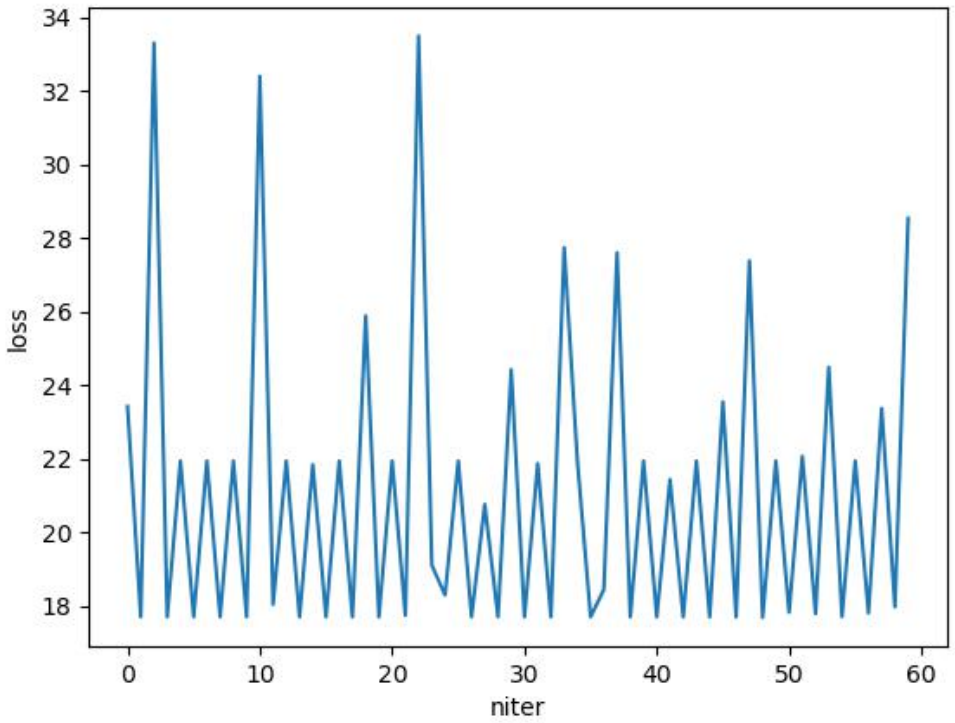
```
        Train a multiclass logistic regression model on the given training set.

        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)

        sample_number = X.shape[0]
        one_hot = np.zeros((sample_number, n_classes))
        one_hot[np.arange(sample_number), y] = 1 # Use the position of the 1 to
represent the class of the corresponding number

        self.loss = []

        for i in range(niter):
            prob = self.predict_proba(X) # probability of forward propagation

            log_loss = one_hot * np.log(prob)
```

```
                log_loss = -1 / sample_number * np.sum(log_loss)
                self.loss.append(log_loss)

            # The gradient of the weight vector and the bias vector to the loss
        function is calculated.
            # ∇_{W_k} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} X^{(i)} [\hat{p}_k^{(i)} - y_k^{(i)}]  (derivation formula of softmax)
                deviation_coef = -1 / sample_number * np.dot(X.T, one_hot - prob)
                deviation_intercept = -1 / sample_number * np.dot(np.ones((1,
sample_number)), one_hot - prob)
            # for each category, update its weight and bias values
            # W_k = W_k - \eta \nabla_{W_k} J
                self.coef_ -= lr * deviation_coef.T

            # b_k = b_k - \eta \nabla_{b_k} J
                self.intercept_ -= lr * deviation_intercept[0]
```

(I found the derivation formula of softmax on this website: https://www.ziiai.com/blog/183)

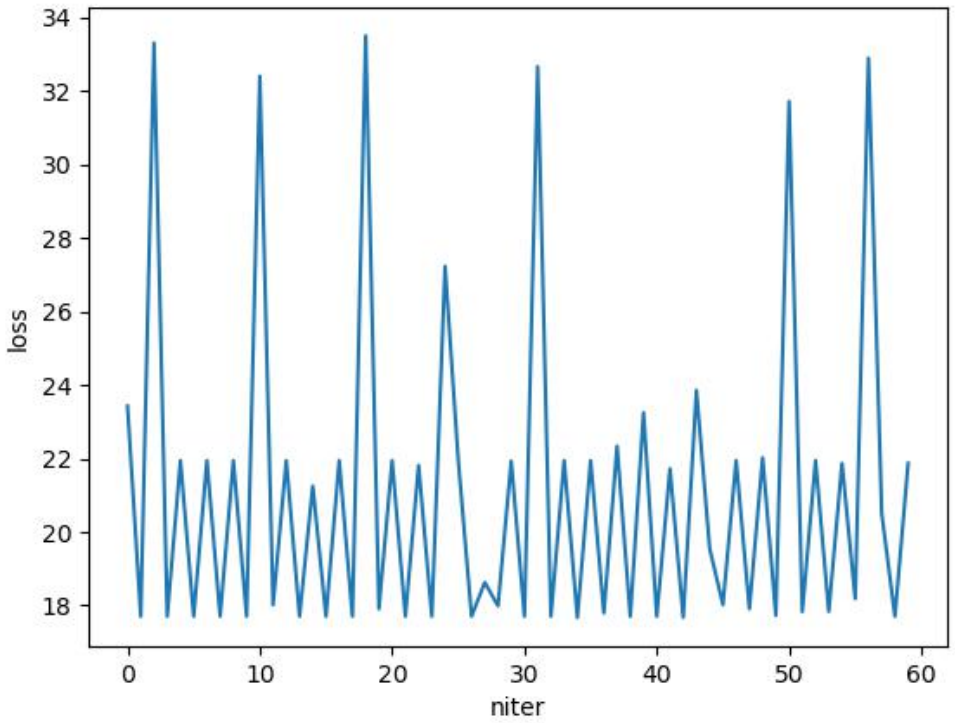
(d) (5 marks) Tune the learning rate and number of learning iterations to minimize the log-loss as much as possible. Describe how you do this. Report the training log-loss of the model that you obtain, and the training and test accuracies.

(1) When lr=0.1, niter = 60
The plot of the training log-loss of the model is listed below:



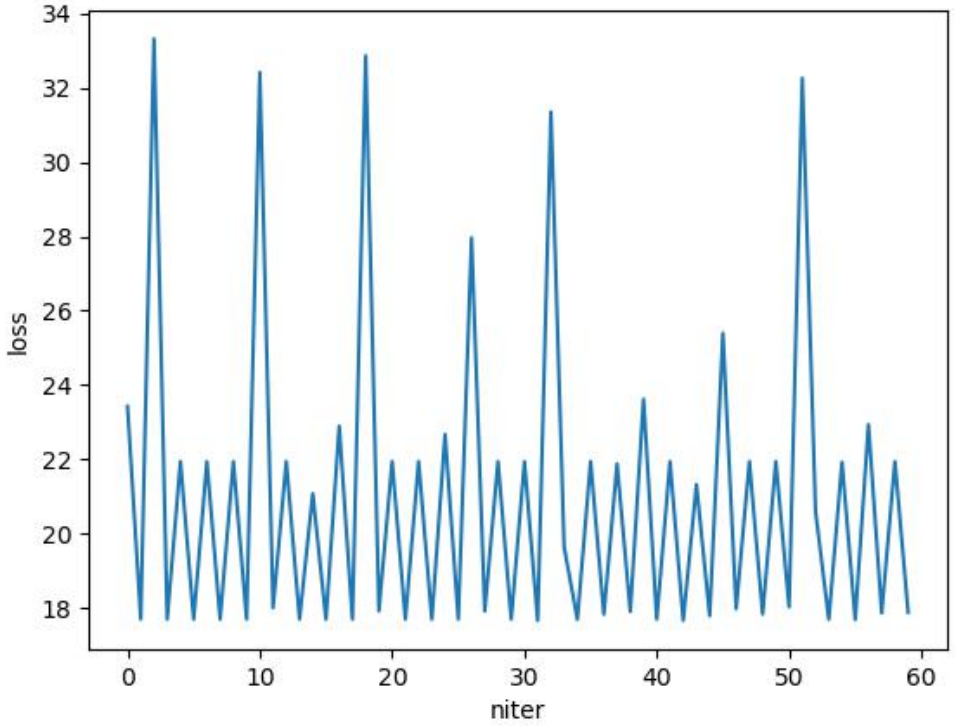
The training accuracy is:0.46568791368746126
The test accuracy is:0.4644242243436754

(2) When lr=0.2, niter = 60
The plot of the training log-loss of the model is listed below:



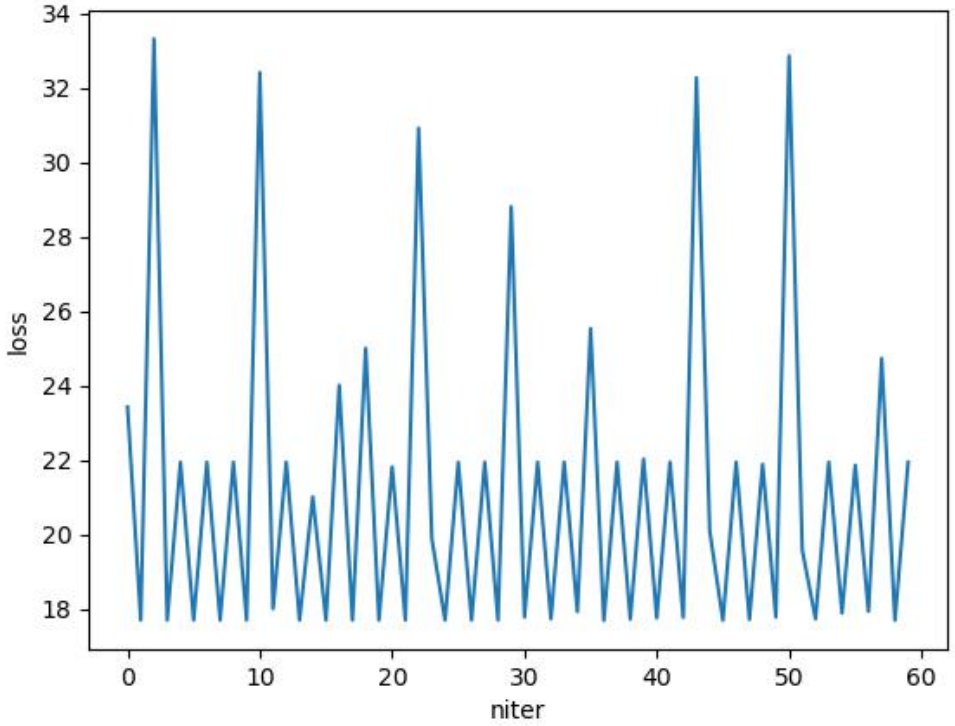
The training accuracy is:0.48811678157302046
The test accuracy is:0.48875527813475306

(3) When lr=0.4, niter = 60
The plot of the training log-loss of the model is listed below:



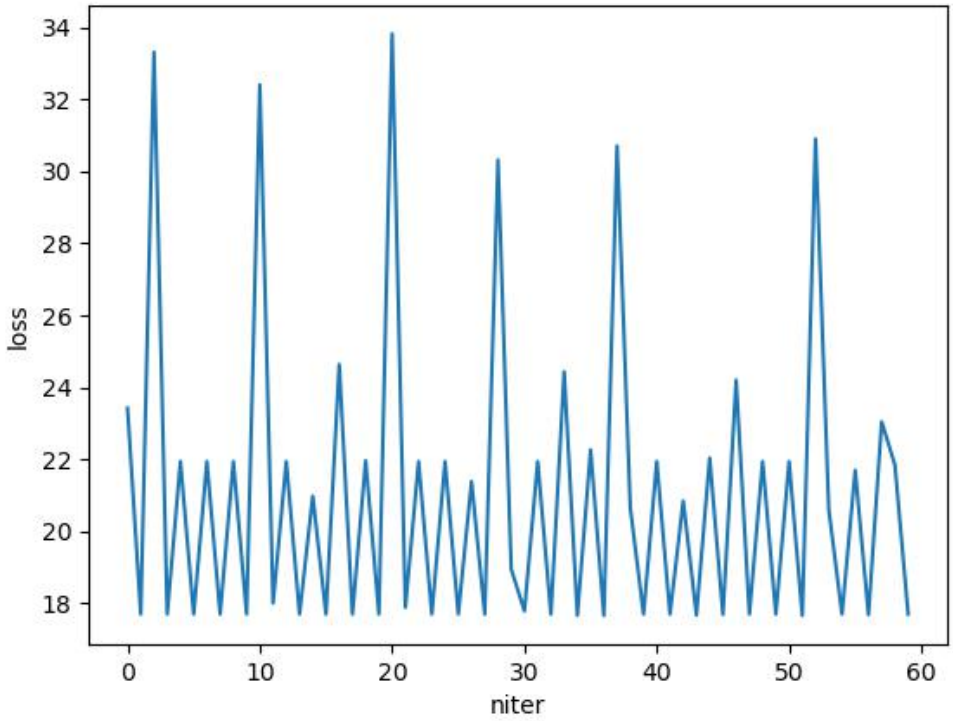
The training accuracy is:0.3674282286062728
The test accuracy is:0.36728359647512393

(4) When lr=0.6, niter = 60
The plot of the training log-loss of the model is listed below:



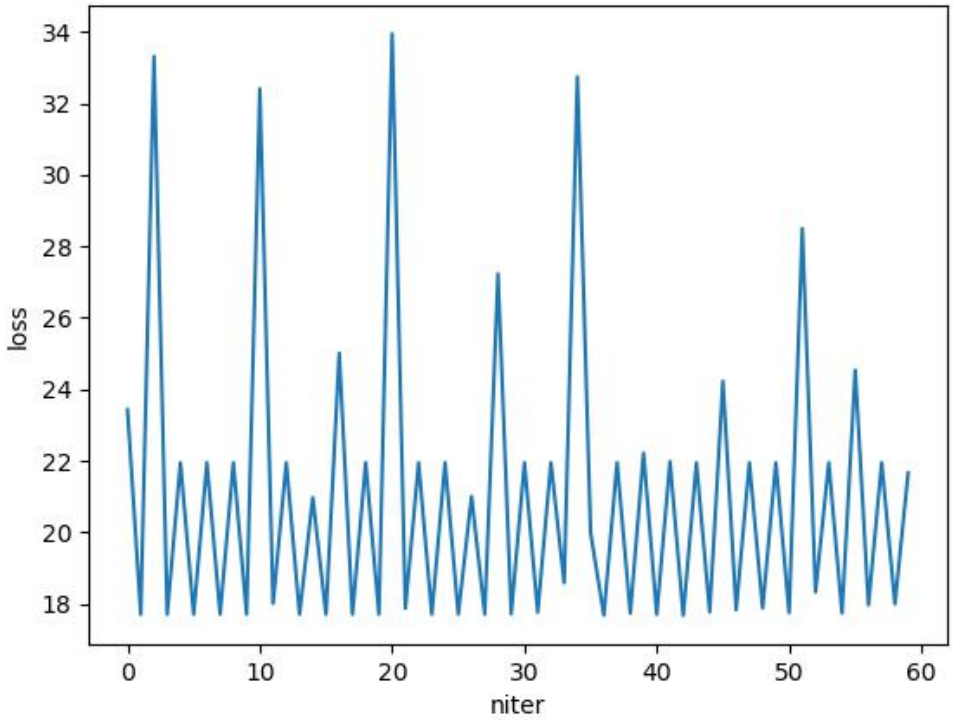
The training accuracy is: 0.48730047109965874
The test accuracy is:0.4880897741876262

(5) When lr=0.8, niter = 60
The plot of the training log-loss of the model is listed below:



The training accuracy is: 0.2427294275008114
The test accuracy is:0.24191642188360565

(6) When lr=1, niter = 60
The plot of the training log-loss of the model is listed below:



The training accuracy is: 0.4804823116338995
The test accuracy is:0.4803389480447953

From the plots and training and test accuracies with different learning rate after changing number of learning iterations to 60, we could find that the log-loss tends to greatly fluctuate as the number of learning iterations increases. Besides, training and test accuracies are also fluctuates with the increase of learning rate, but they all less than 0.5.

(e) (5 marks)

```
        # According to momentum method, the next momentum value is based on
    the current and previous one
        # when i=0, we can't find the previous one, so I initialize the first value to 0
          previous_coef = [0, 0]
          previous_intercept = [0, 0]

          for i in range(niter):
                prob = self.predict_proba(X) # probability of forward propagation

                log_loss = one_hot * np.log(prob)
                log_loss = -1 / sample_number * np.sum(log_loss)
                self.loss.append(log_loss)

                # for each category, update its weight and bias values
```

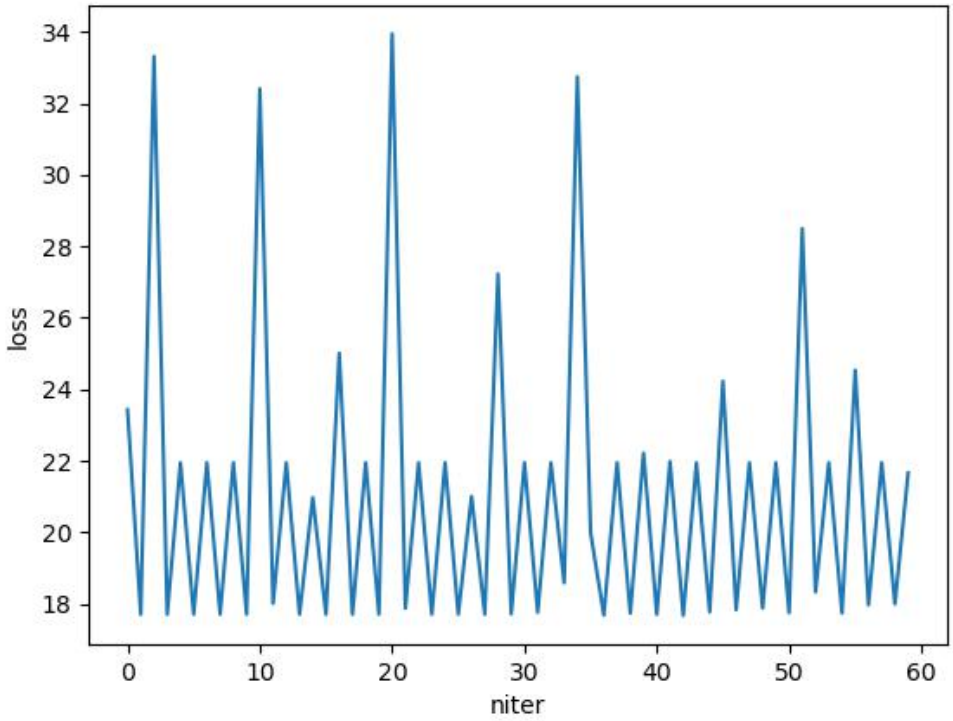
```
                deviation_coef = -1 / sample_number * np.dot(X.T, one_hot - prob)
                deviation_intercept = -1 / sample_number * np.dot(np.ones((1,
sample_number)), one_hot - prob)

                if i == 0:

                    #W_k = W_k - η∇_{W_k} J
                        self.coef_ -= lr * deviation_coef.T
                    #b_k = b_k - η∇_{b_k} J
                        self.intercept_ -= lr * deviation_intercept[0]

                    # update previous value
                        previous_coef[1] = self.coef_
                        previous_intercept[1] = self.intercept_

                else:
                    #  W_{t+1} = W_t - ηg_t + β(W_t - W_{t-1})
                        self.coef_ -= lr * deviation_coef.T + momentum *
(previous_coef[1] - previous_coef[0])
                        self.intercept_ -= lr * deviation_intercept[0] + momentum *
(previous_intercept[1] - previous_intercept[0])

                    #update the previous value and current value
                        previous_coef[0] = previous_coef[1]
                        previous_coef[1] = self.coef_

                        previous_intercept[0] = previous_intercept[1]
                        previous_intercept[1] = self.intercept_
```

(f) (5 marks) Modify your fit function to further support the momentum trick and tune

the momemtum constant β to try to make the log-loss as small as possible.

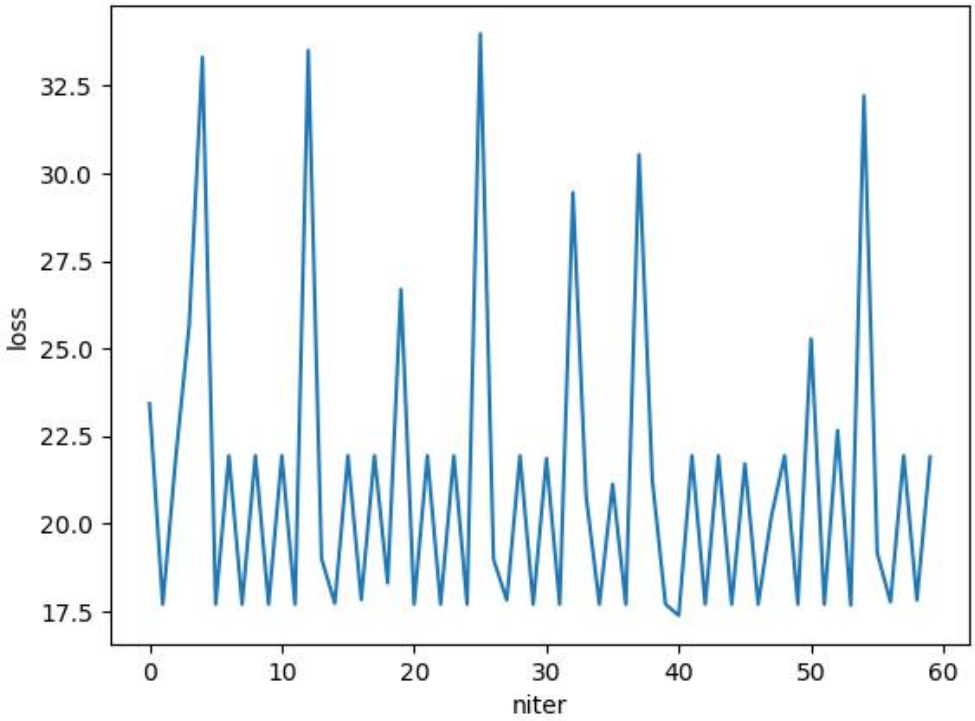
Describe
how you do this. Report the training log-loss of the model that you obtain, and the training and test accuracies.

(1) When lr= 1, niter = 60, momentum = 0
The plot of the training log-loss of the model is listed below:



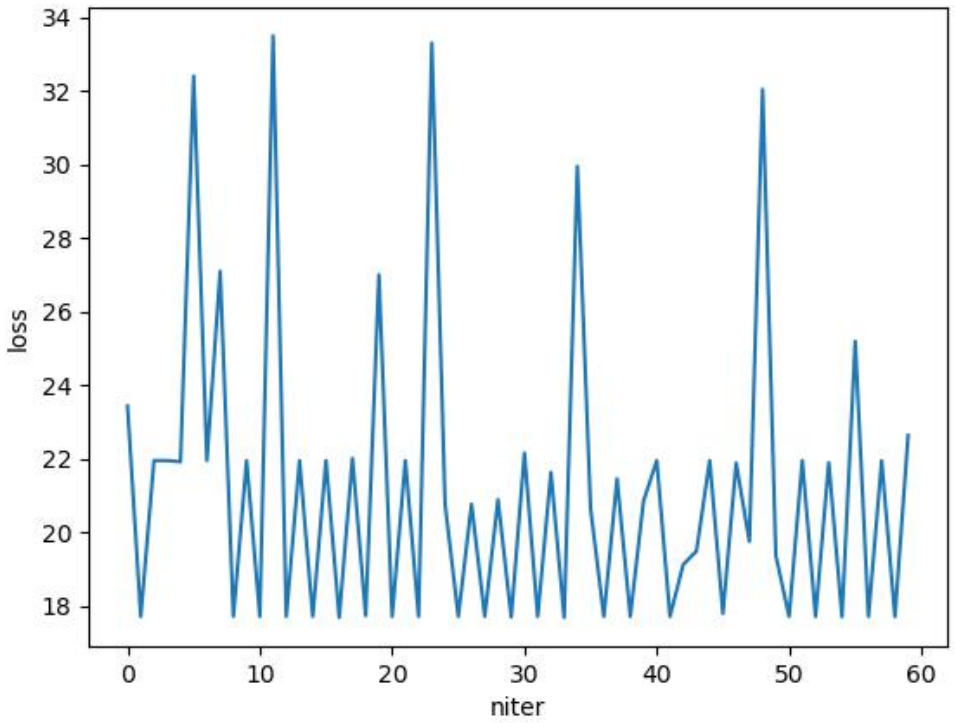
The training accuracy is: 0.4804823116338995
The test accuracy is:0.4803389480447953

(2) When lr= 1, niter = 60, momentum = 2
The plot of the training log-loss of the model is listed below:



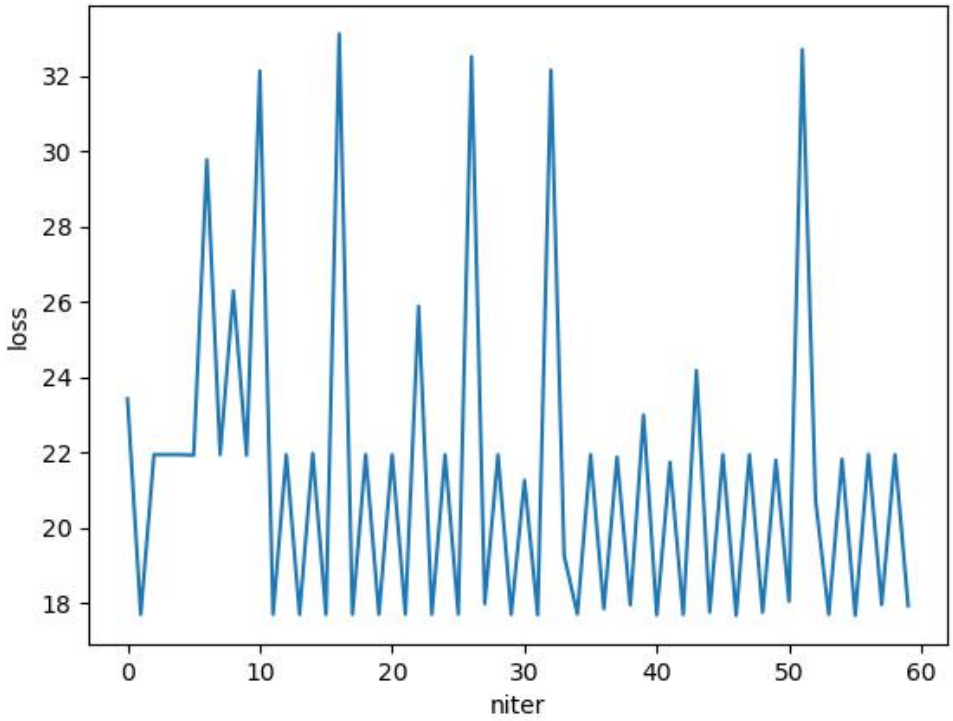
The training accuracy is: 0.4792111293606224
The test accuracy is:0.47889319809069214

(3) When lr= 1, niter = 60, momentum = 5
The plot of the training log-loss of the model is listed below:



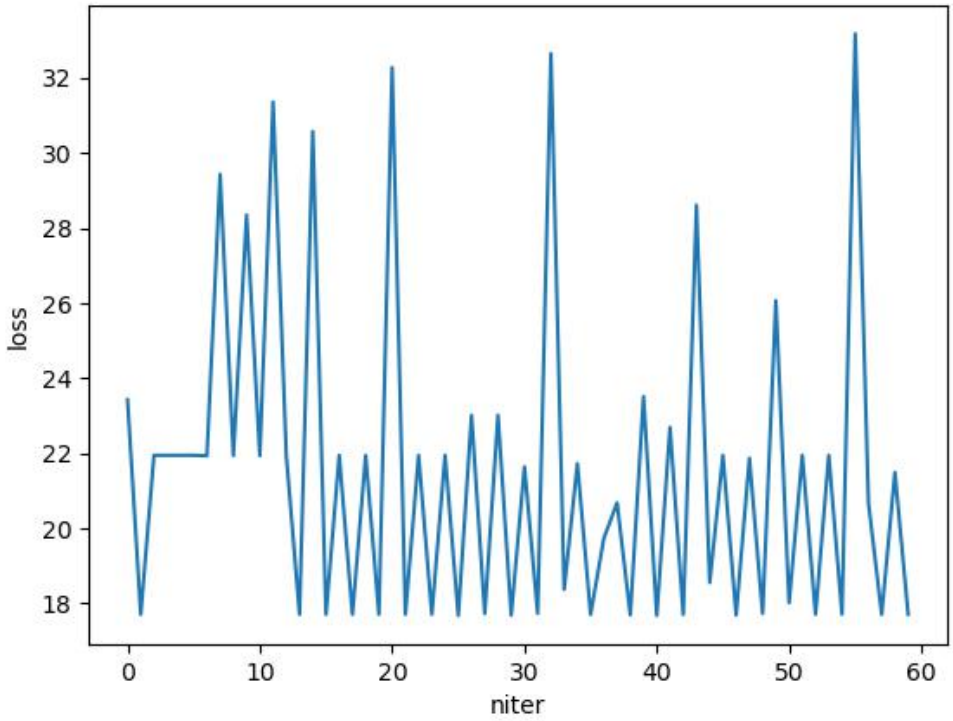
The training accuracy is: 0.48829381276001455
The test accuracy is:0.4888585459886176

(4) When lr= 1, niter = 60, momentum = 8
The plot of the training log-loss of the model is listed below:



The training accuracy is: 0.3649202867905229
The test accuracy is:0.36501170369010466

(5) When lr= 1, niter = 60, momentum = 10
The plot of the training log-loss of the model is listed below:

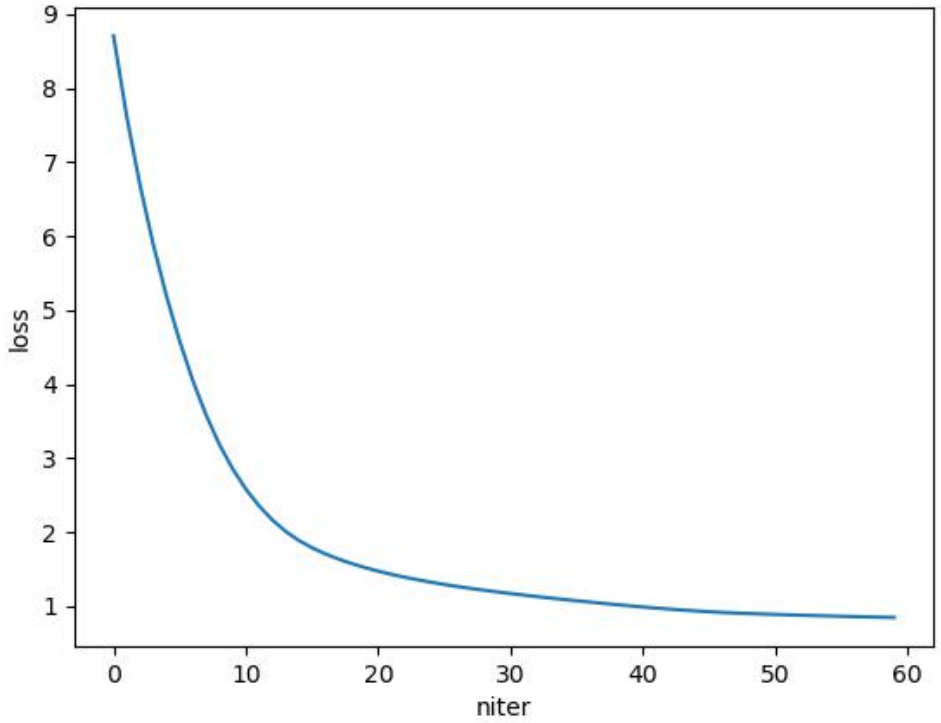


The training accuracy is: 0.42058184250125397
The test accuracy is:0.420128052138792

From the plots and training and test accuracies with different momentum constant β after changing number of learning iterations to 60 and learning rate to 1, we could find that the log-loss tends to greatly fluctuate as the number of learning iterations increases. Besides, training and test accuracies are also fluctuates with the increase of learning rate, but they all less than 0.5.

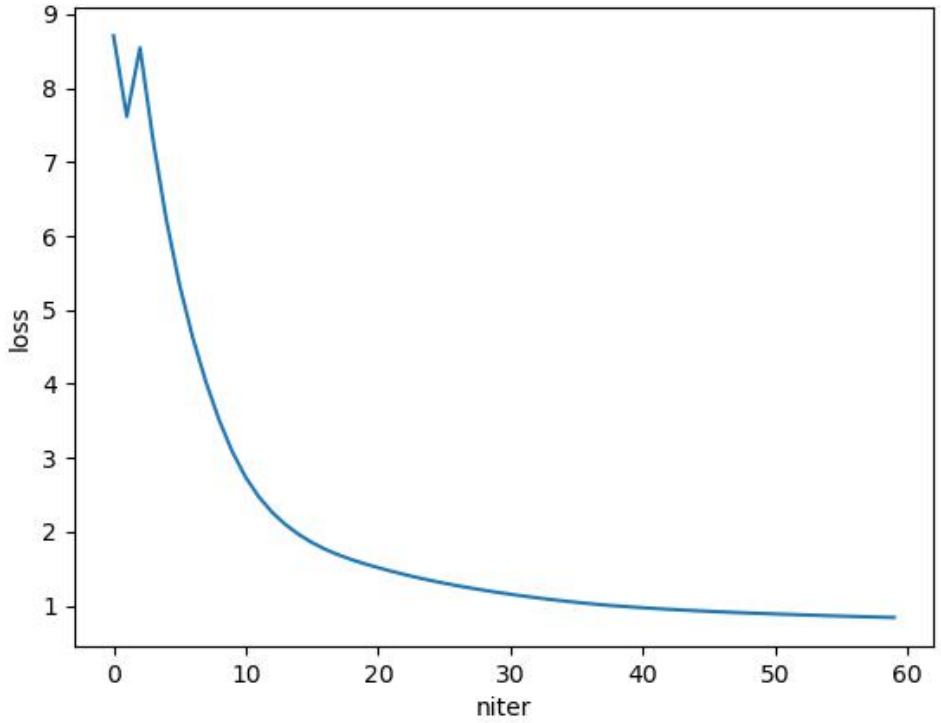
(g) (5 marks) Another useful trick to speed up convergence is to normalize all features to have mean 0 and unit variance first. Implement this, and repeat (d) to try to use gradient descent to find a good model. Comment on the effectiveness of this trick.

(1) When lr= 1, niter = 60, momentum = 0
The plot of the training log-loss of the model is listed below:



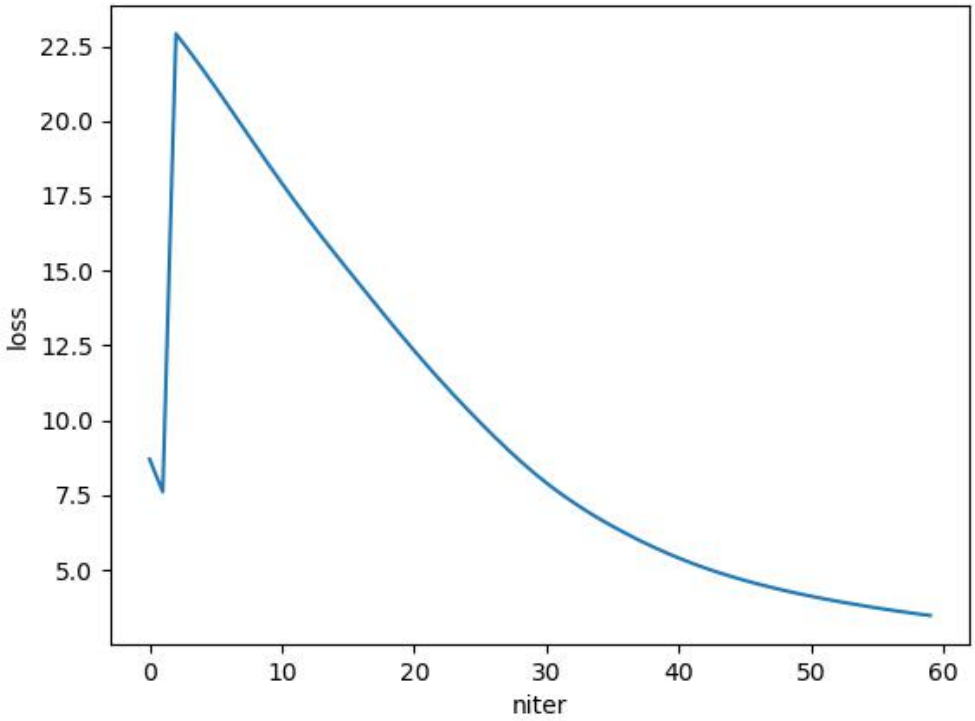
The training accuracy is: 0.7062487091475949
The test accuracy is:0.7043728199008629

(2) When lr= 1, niter = 60, momentum = 2
The plot of the training log-loss of the model is listed below:



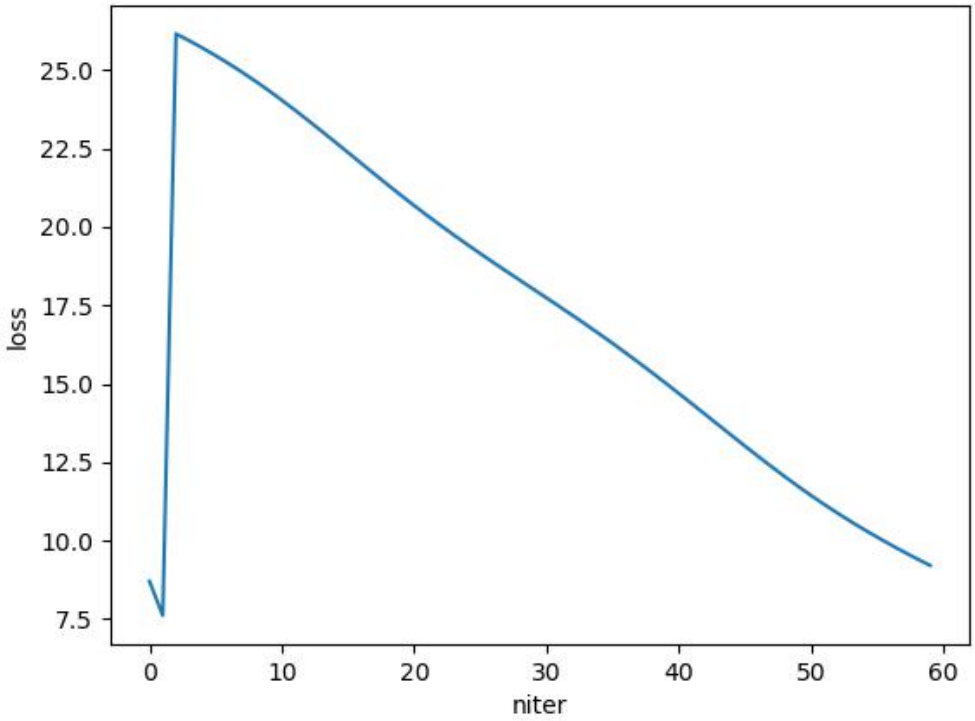
The training accuracy is: 0.7066322767194154
The test accuracy is:0.7057038277951165

(3) When lr= 1, niter = 60, momentum = 5
The plot of the training log-loss of the model is listed below:



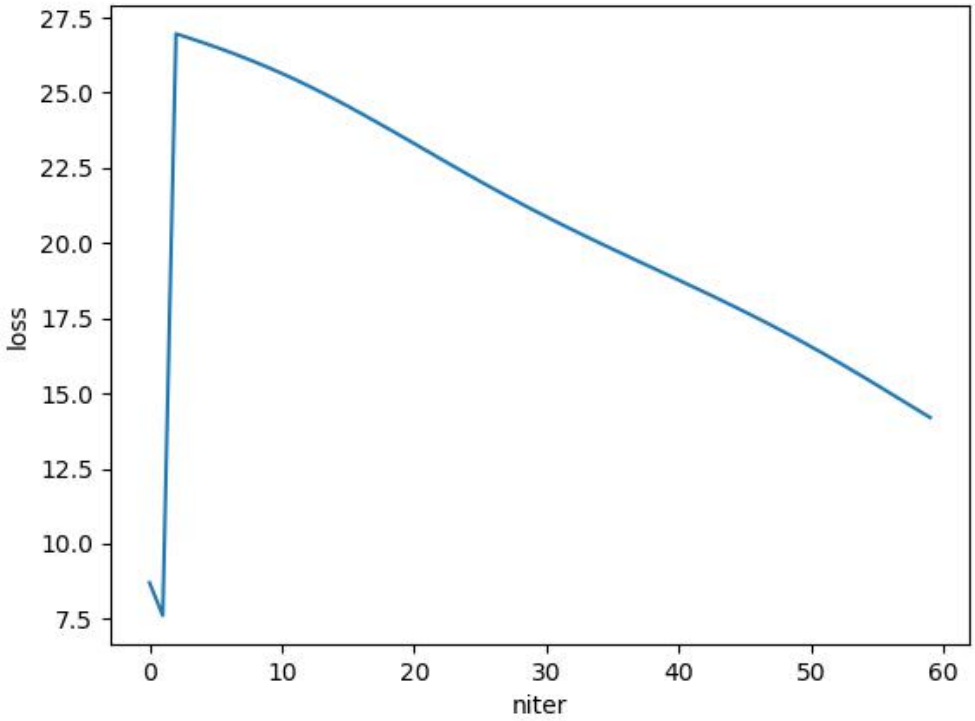
The training accuracy is: 0.60275431021765
The test accuracy is:0.6031244262897008

(4) When lr= 1, niter = 60, momentum = 8
The plot of the training log-loss of the model is listed below:



The training accuracy is: 0.45144182066740757
The test accuracy is:0.4511428309161006

(5) When lr= 1, niter = 60, momentum = 10
The plot of the training log-loss of the model is listed below:



The training accuracy is: 0.3633589700718943
The test accuracy is:0.36292913530383697

After using StandardScaler() function of sklearn.preprocessing.StandardScaler to normalize the model, from the plots and training and test accuracies with different momentum constant β after changing number of learning iterations to 60 and learning rate to 1, we could find that the trend of log-loss is smoother than question (f) and the log-loss decreases as the number of learning iterations increases. Besides, training and test accuracies are much higher than question (f) when momentum from 1 to 6, which are higher than 0.6. However, when momentum is greater than 8, training and test accuracies decline rapidly, which are less than 0.5.

Therefore, the better momentum for this model is from 0 to 2, because their training and test accuracies are all above 0.7.

3.
(b) (5 marks) Implement ENOLS's training algorithm as described above in the fit function of the ENOLS class.

```
def fit(self, X, y, random_state=None):
        '''
        Train ENOLS on the given training set.
```

```
        Parameters
        ----------
        X: an input array of shape (n_sample, n_features)
        y: an array of shape (n_sample,) containing the values for the input
examples

        Return
        ------
        self: the fitted model
        '''

        # use random instead of np.random to sample random numbers below
        random = check_random_state(random_state)

        # add all the trained OLS models to this list
        self.estimators_ = []

        # write your training code below. your code should support the
        # n_estimators and sample_size hyper-parameters described in the
        # documentation for the __init__ function
        sample_number = len(y)

        for i in range(self.n_estimators):
            if self.sample_size == "auto":
                subset_size = X.shape[1]+1
            elif isinstance(self.sample_size, int):
                subset_size = self.sample_size
            elif isinstance(self.sample_size, float):
                subset_size = np.ceil(self.sample_size*sample_number)

            subset_NO = random.choice(sample_number, subset_size) #the index
of sample in subset
            lr = LinearRegression()
            lr.fit(X[subset_NO], y[subset_NO])
            self.estimators_.append(lr)

        return self
```

(c) (5 marks) Implement ENOLS's two prediction strategies as described above in the predict function of the ENOLS class.

```
    def predict(self, X, method='average'):
        '''
```

```
        Parameters
        ----------
        X: an input array of shape (n_sample, n_features)
        method: 'median' or 'average', corresponding to predicting median and
            mean of the OLS models' predictions respectively.

        Returns
        -------
        y: an array of shape (n_samples,) containig the predicted values
        '''

        predicted_values = []

        for estimator in self.estimators_:
            predict_y = estimator.predict(X)
            predicted_values.append(predict_y)

        predicted_values = np.array(predicted_values) #shape(100,152)

        if method == 'average':
            predicted_values = np.mean(predicted_values, axis=0)
        elif method == 'median':
            predicted_values = np.median(predicted_values, axis=0)

        return predicted_values
```

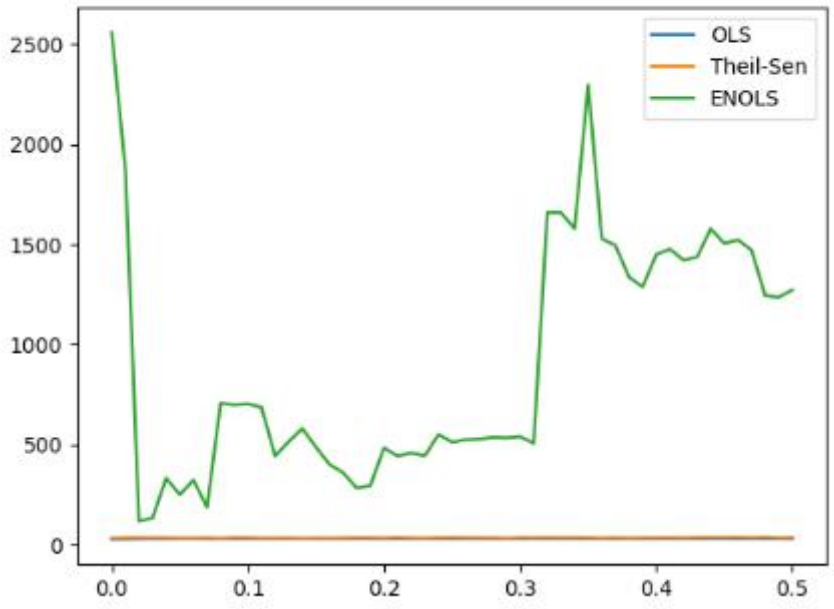
(d) (5 marks) For each proportion p from 0, 0.01, 0.02, . . . to 0.5, use the corrupt function to generate a corrupted training set by making a proportion p of the examples out liers, then train an OLS model, a Theil-Sen estimator, and a ENOLS model (using the default hyper-parameters), and measure their test MSEs. Plot the three models' test MSEs against p, and compare their performance.

I found that question (e), (f) and (g) are all repeat (d), the only difference is that question (e) needs to change method, question (f)
Hence, I create a function called "models_comparison", it could be called conveniently in the following questions.

```
def models_comparison(fig_name, n_estimators=100, method='average', sample_size='auto'):
    proportion = [0.01 * i for i in range(51)]
    OLS_MSE = []
    Theil_Sen_MSE = []
```

```
    ENOLS_MSE = []

    for p in proportion:
        W, z = corrupt(X_tr, y_tr, outlier_ratio=p, random_state=111)

        OLS_model = LinearRegression()
        OLS_model.fit(W, z)

        Theil_Sen_estimator = TheilSenRegressor()
        Theil_Sen_estimator.fit(W, z)

        ENOLS_model = ENOLS(n_estimators=n_estimators,
sample_size=sample_size)
        ENOLS_model.fit(W, z, random_state=111)

        OLS_MSE.append(mean_squared_error(y_ts, OLS_model.predict(X_ts)))
        Theil_Sen_MSE.append(mean_squared_error(y_ts,
Theil_Sen_estimator.predict(X_ts)))
        ENOLS_MSE.append(mean_squared_error(y_ts,
ENOLS_model.predict(X_ts, method=method)))

    plt.plot(proportion, OLS_MSE, label='OLS')
    plt.plot(proportion, Theil_Sen_MSE, label='Theil-Sen')
    plt.plot(proportion, ENOLS_MSE, label='ENOLS')
    plt.legend()
    plt.savefig(fig_name)
   plt.cla()

models_comparison('Question d.png')
```

The output is a plot of the three models' test MSEs against p, which I listed below:



From this plot, we could find that for each proportion p from 0, 0.01, 0.02, . . . to 0.5, test MSEs of OLS model and Theil-Sen estimator are both almost 0, which means that the prediction of these two models is relatively good and the prediction results are very stable, which is basically unaffected by outlier.

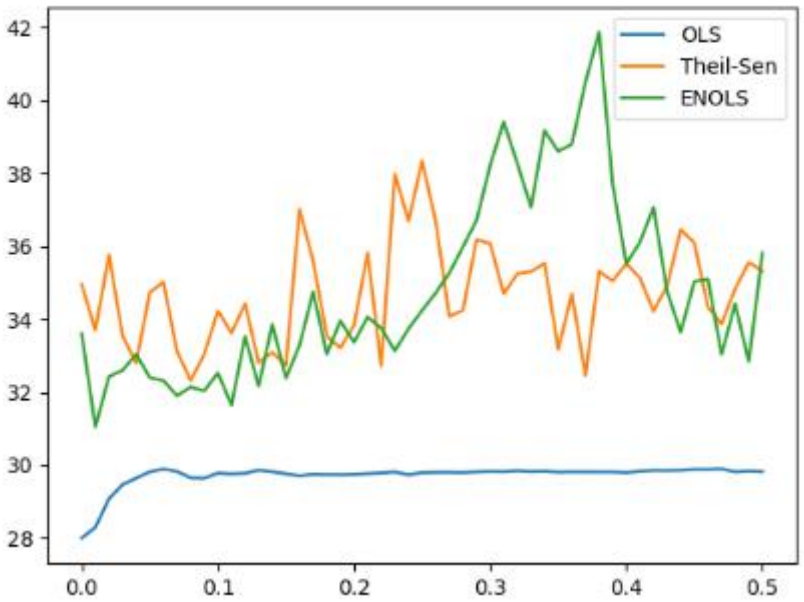
However, test MSEs of ENOLS model vary with the change of proportion p and it fluctuates a lot. This means that the prediction results are strongly influenced by outlier.

The reason for this situation may be that method is “average”.

(e) (5 marks) Repeat (d), but set method="median" when using ENOLS for prediction. How does the performance of ENOLS change as compared to (d)? Explain why.

models_comparison('Question e.png', method='median')

The output is:



From this plot, we could find that for each proportion p from 0, 0.01, 0.02, . . . to 0.5, test MSEs of ENOLS model vary with the change of proportion p and it still fluctuates a lot, although the test MSEs fluctuation of it is much lower than question (d). This means that the prediction results are still affected by outlier.

The test MSEs of Theil-Sen estimator is lower than ENOLS model, they are still vary with the change of proportion p. This means that the prediction of this model is not good and the prediction results are also influenced by outlier.

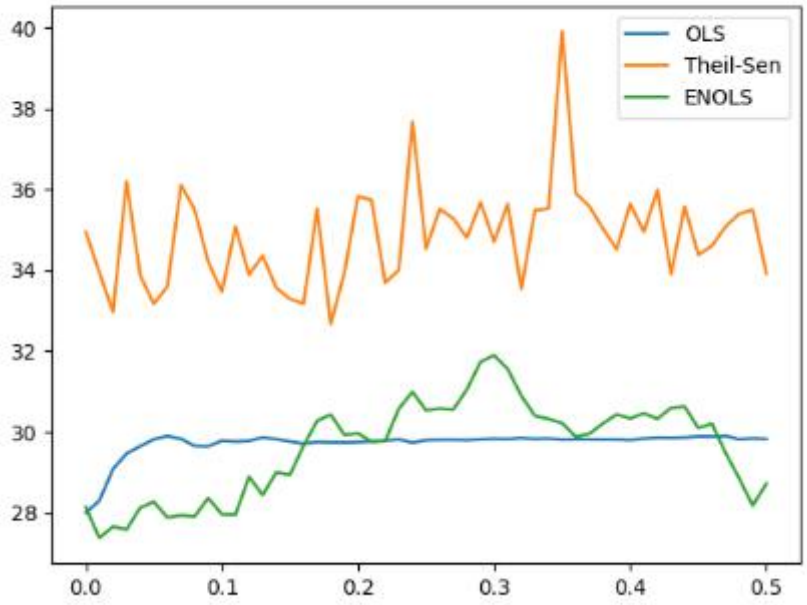
However, The test MSEs of OLS model is relatively lower than other two models and they are very stable after p=0.05, which means that the prediction of this model is relatively good and the prediction results are very stable, which is less affected by outlier.

The reason for this situation may be that method is “median”.

(f) (5 marks) Repeat (e), but set n_estimators=500 when constructing an ENOLS model. How does the performance of ENOLS change as compared to (e)? Explain why.

models_comparison('Question f.png', n_estimators=500, method='median')

The output is:



From this plot, we could find that for each proportion p from 0, 0.01, 0.02, . . . to 0.5, test MSEs of ENOLS model is lower than Theil-Sen estimator, they are still vary with the change of proportion p and it fluctuates a lot, although test MSEs of it is lower than question (e). This means that the prediction of this model is not good and the prediction results are also influenced by outlier.

Test MSEs of Theil-Sen estimator are higher than other two models, which means that the prediction of this model is not good and the prediction results are strongly affected by outlier.

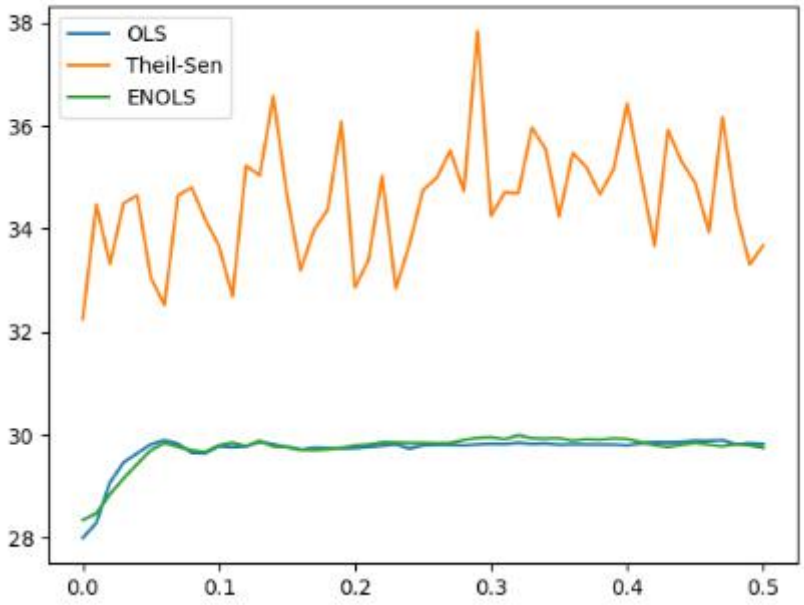
However, test MSEs of OLS model is relatively lower than other two models and they are very stable after p=0.05, which means that the prediction of this model is relatively good and the prediction results are very stable, which is less affected by outlier.

The reason for this situation may be that we increase the number of estimator.

(g) (5 marks) Repeat (f), but set sample_size=42. How does the performance of ENOLS change as compared to (f)? Explain why.

models_comparison('Question g.png', n_estimators=500, method='median', sample_size=42)

The output is:



From this plot, we could find that for each proportion p from 0, 0.01, 0.02, . . . to 0.5, test MSEs of ENOLS model is lower than question (f) with smaller fluctuations and smoother trends.

We could find that test MSEs of OLS model and ENOLS model are both lower than Theil-Sen estimator and the trend of test MSEs of these two models are almost overlapping, which means that the prediction of these two models is relatively good and the prediction results are very stable after p=0.05, which is less affected by outlier.

However, test MSEs of Theil-Sen estimator vary with the change of proportion p and it fluctuates a lot. This means that the prediction results are strongly influenced by outlier.

The reason for this situation may be that the size of sample is 42.

(h) (5 marks) Show that the expected proportion of subsets that does not contain an outlier can be arbitrarily close to 0 for large n.

I discussed this question with Runqi Lin to get the idea of solution.

(h)

I assume that the number of data is n.

From the question, we could know that there are np outliers, so there are $(n-np)$ normal data.

When subset size $m = nq$, there are two situation of subset size:

(1) $nq < n-np$, which means the probability of us to get a normal data is 100% when we take a sample

$\Rightarrow p < 1-q$

$\Rightarrow p+q < 1$

(2) $nq > n-np$, which means the probability of us to get an outlier is 100% when we take a sample

$\Rightarrow p+q > 1$

And then, the probability that we get a normal sample is :

$$\frac{C_{n-np}^{nq}}{C_n^{nq}} = \frac{\frac{(n-np)!}{(n-np)!(n-np-nq)!}}{\frac{n!}{n!(n-nq)!}} = \frac{(n-np)!(n-nq)!}{n!(n-np-nq)!}$$

Therefore, for large n, as $n \to \infty$, $\frac{C_{n-np}^{nq}}{C_n^{nq}} = \frac{(n-np)!(n-nq)!}{n!(n-np-nq)!} \approx$ close to 0

Hence, we could find that the expected proportion of subsets that doesn't contain an outlier can be arbitrarily close to 0 for large n.