Q2.2

October 17, 2019

0.1 Q2.2

[36]:
```
import numpy as np
from check_grad import check_grad
from utils import *
from logistic import *
import matplotlib.pyplot as plt
```

[3]:
```
def plot_losses(train_losses, valid_losses):
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 2, 1.5])

    ax.set_title("error curve")

    x = range(1, hyperparameters['num_iterations'] + 1)
    # Plot the train ce.
    ax.plot(x, train_losses, label='train loss')
    # Plot the validation set ce.
    ax.plot(x, valid_losses, label='validation loss')

    plt.xlabel('# iteration')
    plt.ylabel('ce')
    plt.legend()
```

[4]:
```
def run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, hyperparameters):

    N, M = train_inputs.shape
    # Logistic regression weights
    # TODO:Initialize to random weights here.
    weights = np.random.rand(M + 1, 1)

    # Store all cross entropy losses of each iteration.
    valid_losses = []
    train_losses = []

    # Begin learning with gradient descent
```

```
    for t in xrange(hyperparameters['num_iterations']):
        # TODO: you may need to modify this loop to create plots, etc.

        # Find the negative log likelihood and its derivatives w.r.t. the␣
↪weights.
        f, df, predictions = logistic_pen(weights, train_inputs, train_targets,␣
↪hyperparameters)

        # Evaluate the prediction.
        cross_entropy_train, frac_correct_train = evaluate(train_targets,␣
↪predictions)
        train_losses.append(cross_entropy_train)

        if np.isnan(f) or np.isinf(f):
            raise ValueError("nan/inf error")

        # update parameters
        weights = weights - hyperparameters['learning_rate'] * df / N

        # Make a prediction on the valid_inputs.
        predictions_valid = logistic_predict(weights, valid_inputs)

        # Evaluate the prediction.
        cross_entropy_valid, frac_correct_valid = evaluate(valid_targets,␣
↪predictions_valid)
        valid_losses.append(cross_entropy_valid)

        # print some stats
#         print ("ITERATION:{:4d}  TRAIN NLOGL:{:4.2f}  TRAIN CE:{:.6f} "
#                "TRAIN FRAC:{:2.2f}  VALID CE:{:.6f}  VALID FRAC:{:2.2f}").
↪format(
#                    t+1, f / N, cross_entropy_train, frac_correct_train*100,
#                    cross_entropy_valid, frac_correct_valid*100)

    return weights, train_losses, valid_losses
```

[5]:
```
def run_check_grad(hyperparameters):
    """Performs gradient check on logistic function.
    """

    # This creates small random data with 20 examples and
    # 10 dimensions and checks the gradient on that data.
    num_examples = 20
    num_dimensions = 10

    weights = np.random.randn(num_dimensions+1, 1)
    data    = np.random.randn(num_examples, num_dimensions)
```

```
    targets = np.random.rand(num_examples, 1)

    diff = check_grad(logistic,      # function to check
                      weights,
                      0.001,         # perturbation
                      data,
                      targets,
                      hyperparameters)

    print "diff =", diff
```

[6]:
```
def report_errors(weights, X_train, y_train, X_valid, y_valid, X_test, y_test):
    train = evaluate(logistic_predict(weights, X_train), y_train)
    valid = evaluate(logistic_predict(weights, X_valid), y_valid)
    test = evaluate(logistic_predict(weights, X_test), y_test)
    print("CE_train: {0}, MISSFRAC_train: {1}".format(train[0], 1 - train[1]))
    print("CE_valid: {0}, MISSFRAC_valid: {1}".format(valid[0], 1 - valid[1]))
    print("CE_test: {0}, MISSFRAC_test: {1}".format(test[0], 1 - test[1]))
```

[7]:
```
import itertools
```

0.1.1 Function for performing grid search on possible hyperparameters

[8]:
```
def grid_params(params):
    expand = []
    comb_num = 1
    template = {}
    for param in params:
        template[param] = None
        expand.append(params[param])
    combs = itertools.product(*expand)
    for comb in combs:
        i = 0
        for param in template:
            template[param] = comb[i]
            i += 1
        yield dict(template)
```

[9]:
```
valid_inputs, valid_targets = load_valid()
# Load test data set.
test_inputs, test_targets = load_train()

paramgrid = {
    'learning_rate': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    'weight_regularization': [0.0],
    'num_iterations': [500]
```

```
}
```

0.1.2 Check gradient of the logistic learning without regularizer

[70]:
```
# Verify that your logistic function produces the right gradient.
# diff should be very close to 0.
run_check_grad({'weight_regularizaiton': 10})
```

```
[[ 2.69750521  2.69750433]
 [-3.29662739 -3.2966259 ]
 [ 0.64565057  0.64565057]
 [ 3.17626395  3.17626171]
 [-0.09908095 -0.09908052]
 [-2.07892727 -2.07892598]
 [ 2.25025391  2.25025262]
 [ 1.77186854  1.77186682]
 [ 0.76289127  0.76289158]
 [ 1.06685496  1.06685319]
 [-1.94352892 -1.94352726]]
diff = 3.309969600731357e-07
```

0.1.3 Run logistic regression on the large trainning set, perform grid search to find the best hyperparameter set, and report the best hyperparameters setting.

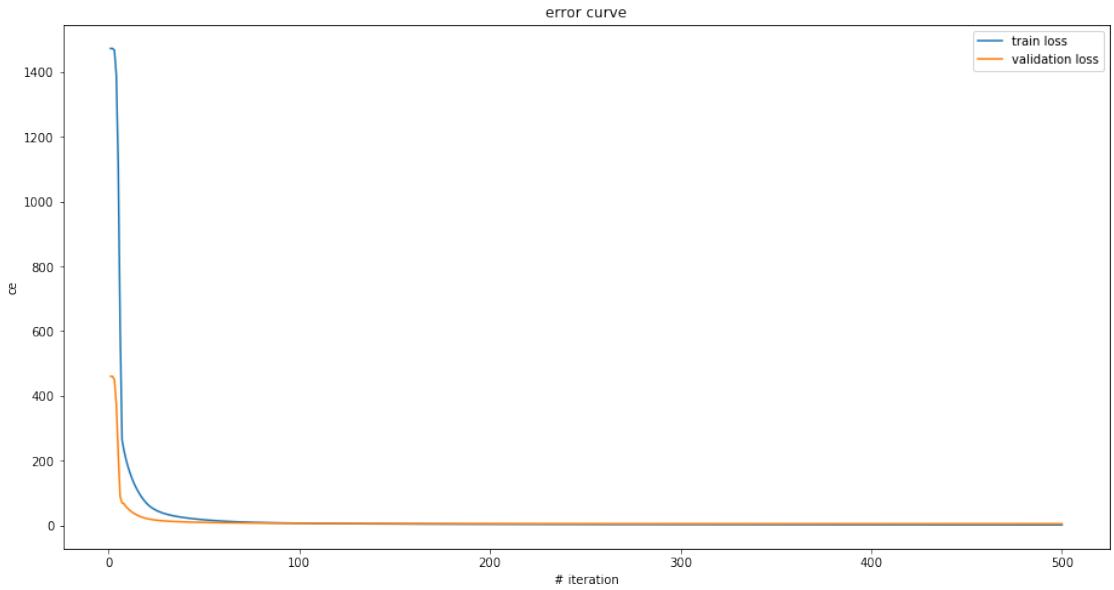
[114]:
```
# Run logistic regression on the large trainning set.
train_inputs, train_targets = load_train()

best_result = None
for hyperparameters in grid_params(paramgrid):
    weights, train_losses, valid_losses = \
        run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, hyperparameters)
    if best_result is None or best_result[2][-1] > valid_losses[-1]:
        best_result = (weights, train_losses, valid_losses, hyperparameters)

print("Best hyperparameters setting:")
print(best_result[3])
%matplotlib inline
plot_losses(best_result[1], best_result[2])
report_errors(best_result[0], train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, test_inputs, test_targets)
```

```
Best hyperparameters setting:
{'num_iterations': 500, 'learning_rate': 0.4, 'weight_regularization': 0.0}
CE_train: 25.0179785967, MISSFRAC_train: 0.0
```

```
CE_valid: 69.1897168959, MISSFRAC_valid: 0.04
CE_test: 25.0179785967, MISSFRAC_test: 0.0
```



0.1.4 Run logistic regression on the small trainning set, perform grid search to find the best hyperparameter set, and report the best hyperparameters setting.

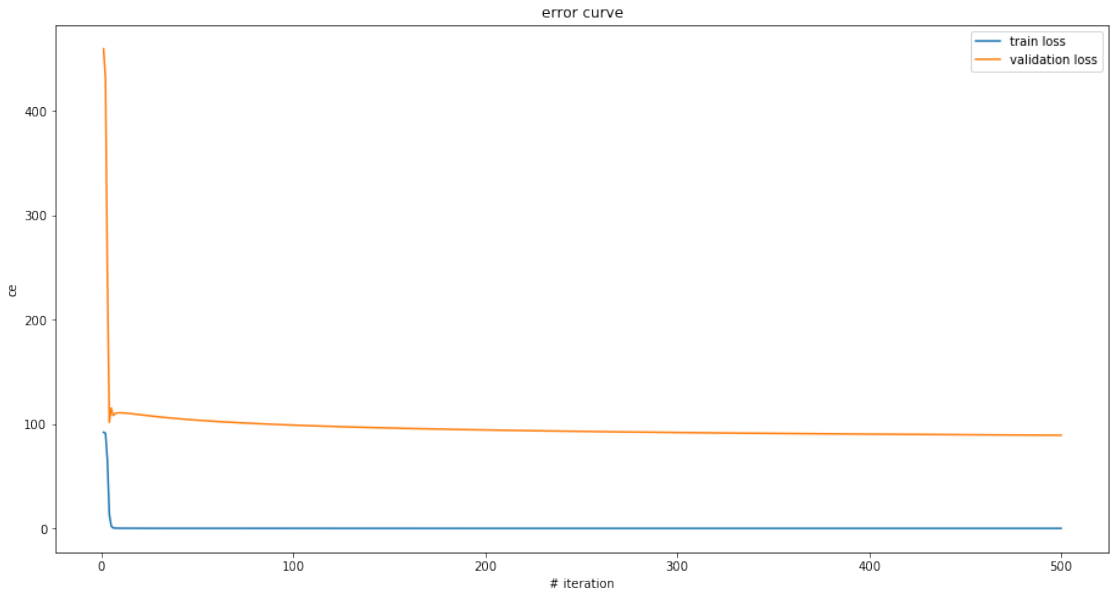
[115]:
```
# Run logistic regression on the large trainning set.
train_inputs, train_targets = load_train_small()

best_result = None
for hyperparameters in grid_params(paramgrid):
    weights, train_losses, valid_losses = \
        run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, hyperparameters)
    if best_result is None or best_result[2][-1] > valid_losses[-1]:
        best_result = (weights, train_losses, valid_losses, hyperparameters)

print("Best hyperparameters setting:")
print(best_result[3])
%matplotlib inline
plot_losses(best_result[1], best_result[2])
report_errors(best_result[0], train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, test_inputs, test_targets)
```

```
Best hyperparameters setting:
{'num_iterations': 500, 'learning_rate': 0.8, 'weight_regularization': 0.0}
CE_train: 0.066998124944, MISSFRAC_train: 0.0
```

```
CE_valid: 326.732616018, MISSFRAC_valid: 0.38
CE_test: 931.614366121, MISSFRAC_test: 0.3
```



0.1.5 Comment on 2.2

In this experiment, I first perform gird search to compare the perfomance of every possible combinations in the given parameter space. The hyperparameters that give the smallest validation error after trainning is selected as the best configuration.

However, I notice that the best configuration is always changing when repeating the experiment. To select the best hyperparameter setting, I should run each configuration multiple times and calculate the average error. Then choose the configuration which yields the lowest average error. This is what I did in Q2.3.

0.2 Q2.3

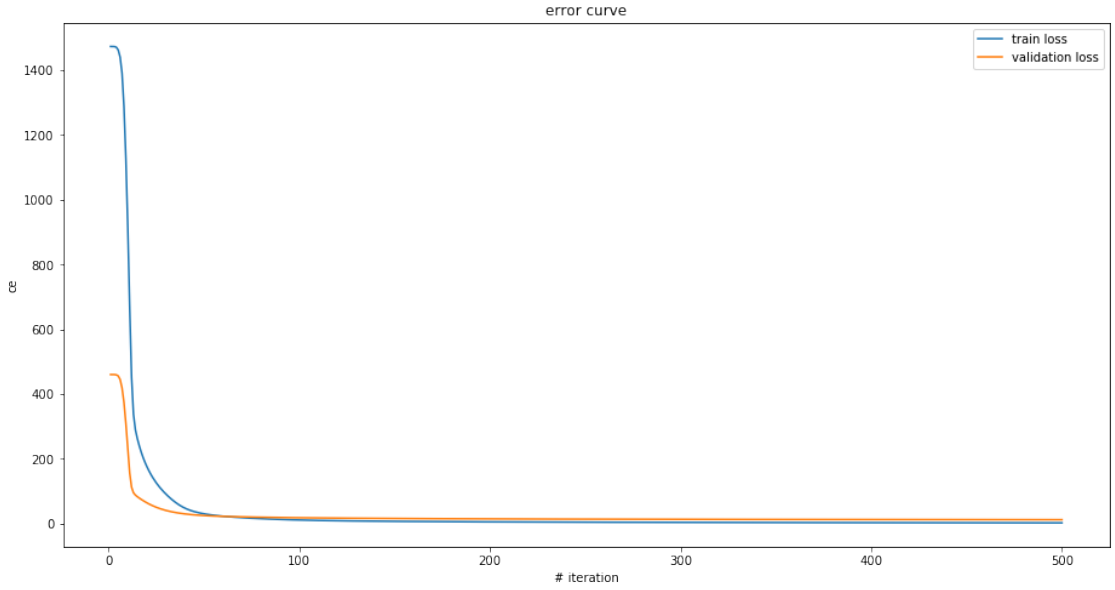
0.2.1 Setup all possible lambda values

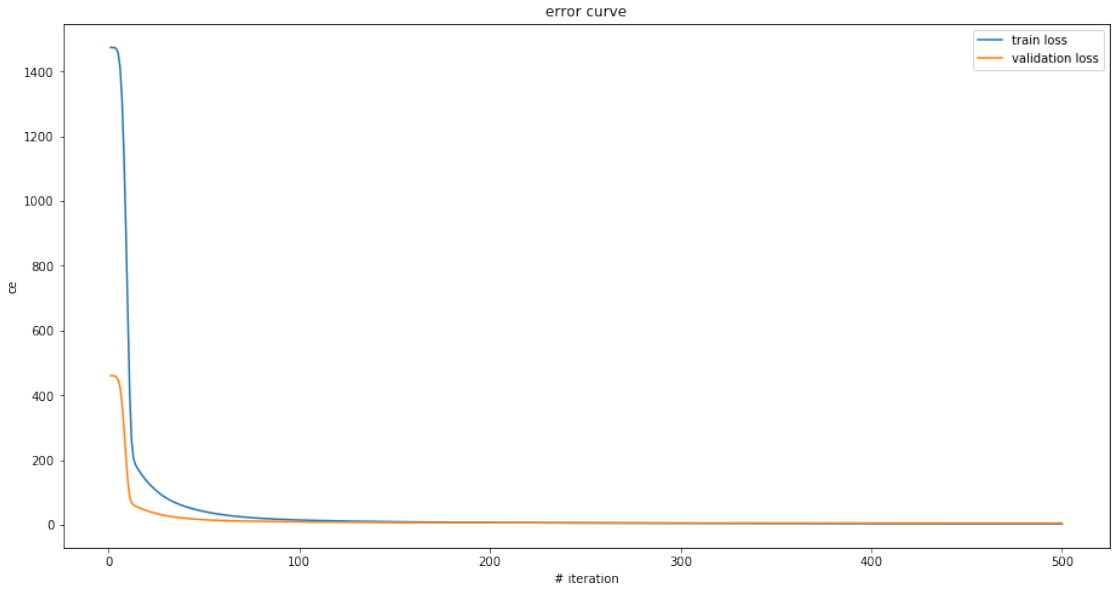
[37]:
```
hyperparameters = {
    'learning_rate': 0.2,
    'weight_regularization': 0,
    'num_iterations': 500
}
lambdas = [0, 0.001, 0.01, 0.1, 1.0]
```

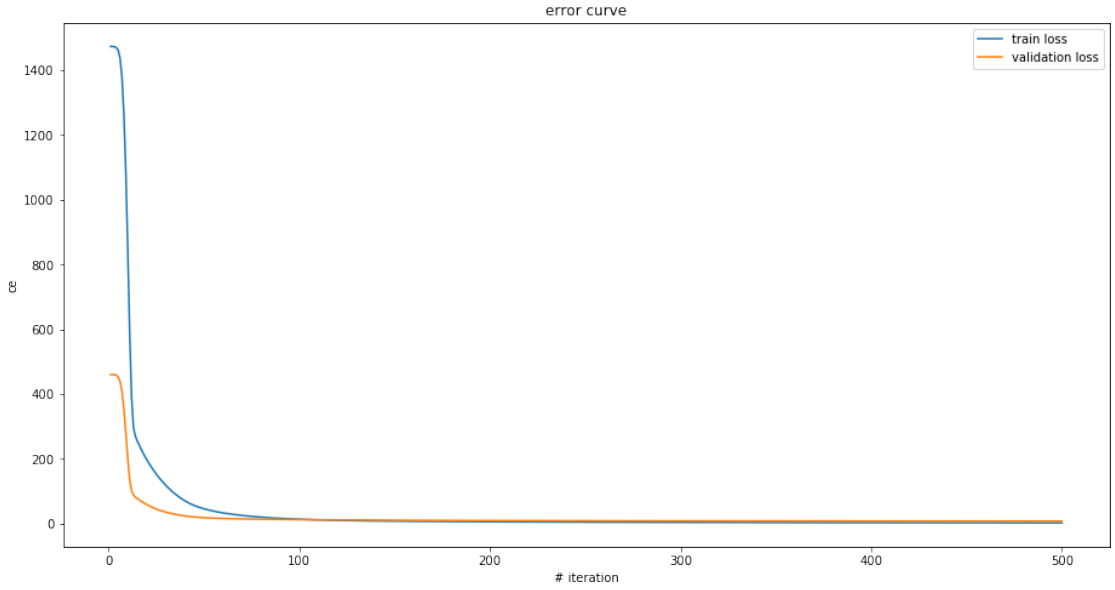
0.2.2 Run penalized logistic regression on large dataset

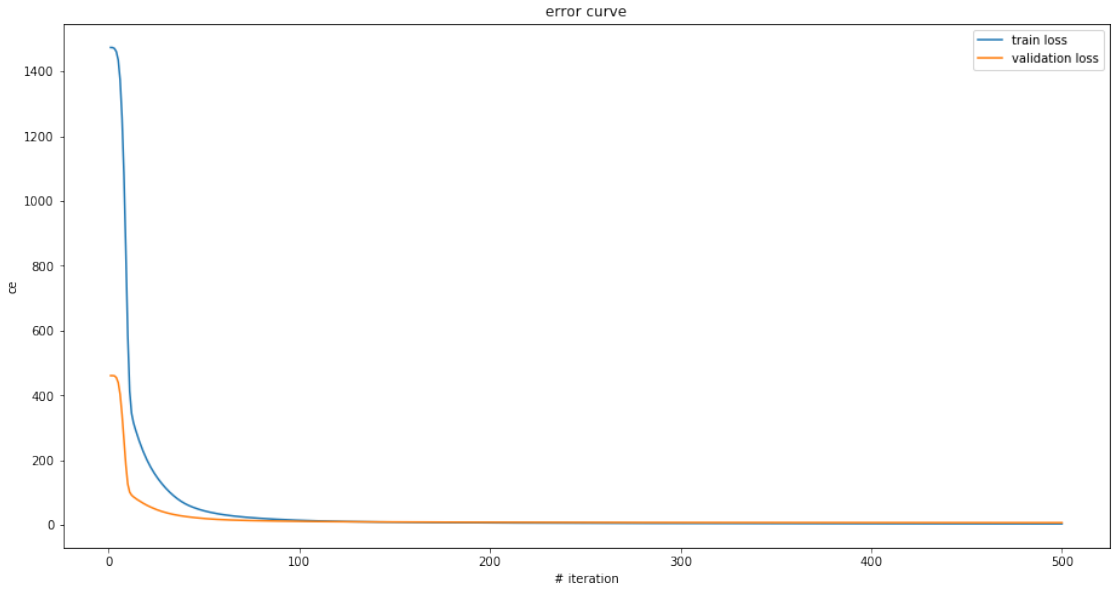
[74]:
```
train_inputs, train_targets = load_train()
```

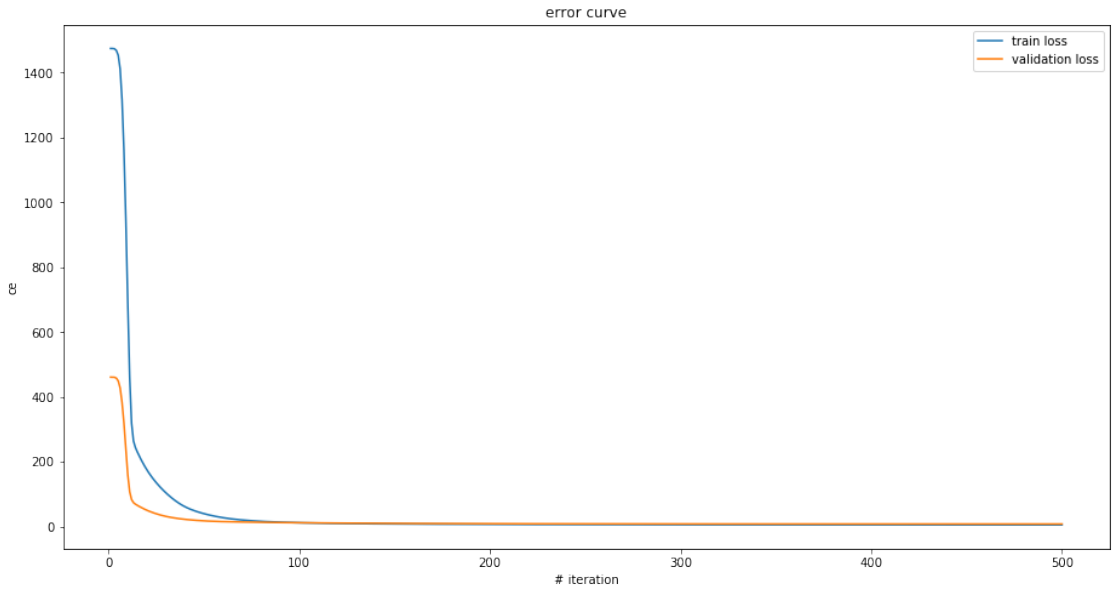
[78]:
```
%matplotlib inline
train_ce_avgs = []
valid_ce_avgs = []
train_fe_avgs = []
valid_fe_avgs = []
weightss = []
for lambd in lambdas:
    hyperparameters['weight_regularization'] = lambd
    train_ce_sum, train_fe_sum = 0, 0
    valid_ce_sum, valid_fe_sum = 0, 0
    weights_arr = []
    for i in range(5):
        weights, train_losses, valid_losses = \
            run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
↪valid_targets, hyperparameters)
        weights_arr.append(weights)
        # Predict with the weights
        y_train = logistic_predict(weights, train_inputs)
        y_valid = logistic_predict(weights, valid_inputs)
        # Compute the cross entrophy error and fraction error
        train_ce, train_fe = evaluate(train_targets, y_train)
        valid_ce, valid_fe = evaluate(valid_targets, y_valid)
        # Compute misclassification rate
        train_fe, valid_fe = 1.0 - train_fe, 1.0 - valid_fe
        # Compute the sum of errors in 5 turns
        train_ce_sum, train_fe_sum = train_ce_sum + train_ce, train_fe_sum +␣
↪train_fe
        valid_ce_sum, valid_fe_sum = valid_ce_sum + valid_ce, valid_fe_sum +␣
↪valid_fe
        # Always print the plot of the last run of each lambda
        if i == 4:
            plot_losses(train_losses, valid_losses)
    # Append the avgs
    train_ce_avgs.append(train_ce_sum / 5.0)
    valid_ce_avgs.append(valid_ce_sum / 5.0)
    train_fe_avgs.append(train_fe_sum / 5.0)
    valid_fe_avgs.append(valid_fe_sum / 5.0)
    weightss.append(weights_arr)
```


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500
0
200
400
600
800
1000
1200
1400

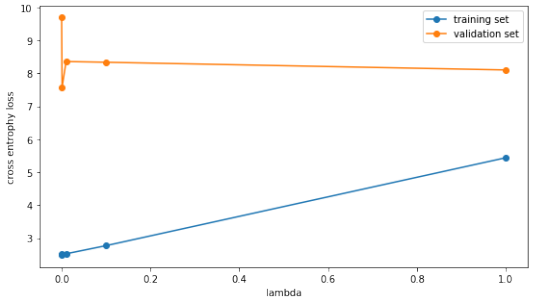


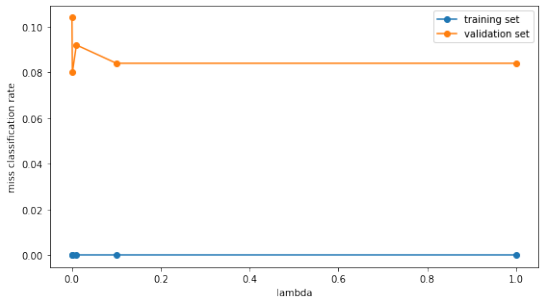
0.2.3 Plot the graphs

[79]:
```
plt.figure(figsize=(20, 5))
plt.subplot(121)
plt.plot(X, train_ce_avgs, '-o', label="training set")
plt.plot(X, valid_ce_avgs, '-o', label="validation set")
plt.xlabel("lambda")
plt.ylabel("cross entrophy loss")
plt.legend()

plt.subplot(122)
plt.plot(X, train_fe_avgs, '-o', label="training set")
plt.plot(X, valid_fe_avgs, '-o', label="validation set")
plt.xlabel("lambda")
plt.ylabel("miss classification rate")
plt.legend()
```

[79]:
```
<matplotlib.legend.Legend at 0x7fb20af66990>
```





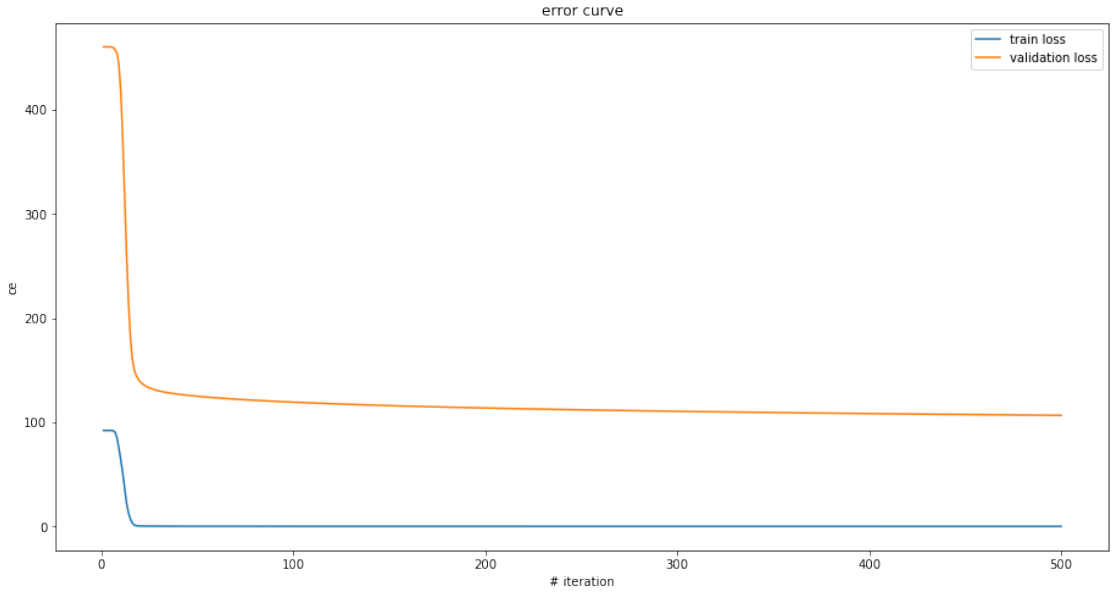
[80]:
```
# Report the best hyperparemeter setting
argmin = np.argmin(valid_ce_avgs)
print("Best lambda setting:")
print(lambdas[argmin])
for i in range(5):
    print("Iteration {0}: ".format(i + 1))
    report_errors(weightss[argmin][i], train_inputs, train_targets,␣
 ↪valid_inputs, valid_targets, test_inputs, test_targets)
```

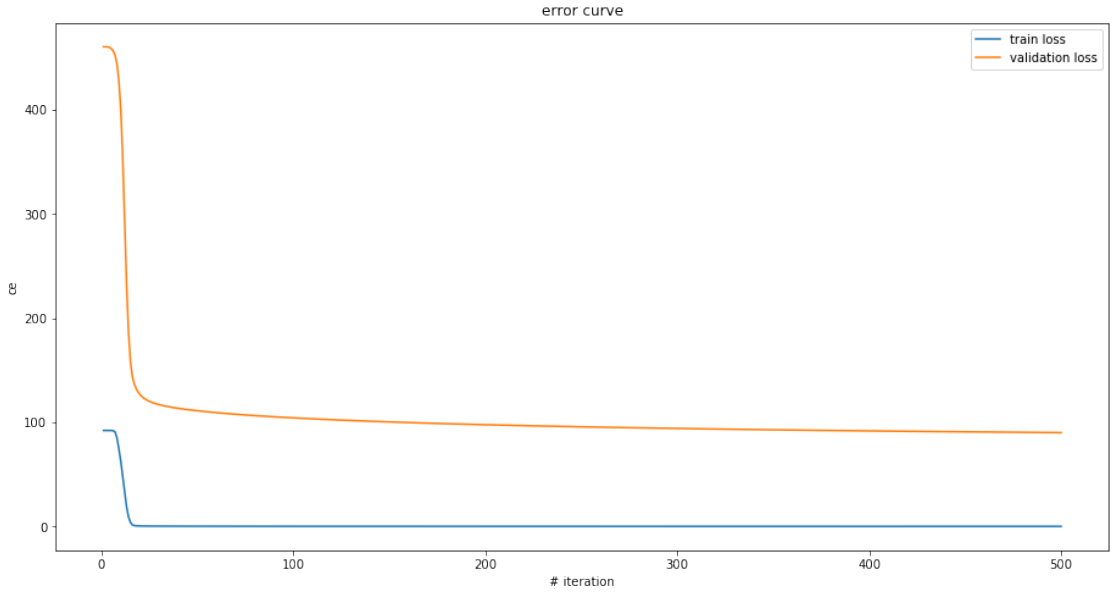
```
Best lambda setting:
0.001
Iteration 1:
CE_train: 44.7075404222, MISSFRAC_train: 0.0
CE_valid: 103.467789306, MISSFRAC_valid: 0.1
CE_test: 44.7075404222, MISSFRAC_test: 0.0
Iteration 2:
CE_train: 45.9161922943, MISSFRAC_train: 0.0
CE_valid: 81.6256268254, MISSFRAC_valid: 0.08
CE_test: 45.9161922943, MISSFRAC_test: 0.0
Iteration 3:
CE_train: 41.0620973085, MISSFRAC_train: 0.0
CE_valid: 96.2279269937, MISSFRAC_valid: 0.12
CE_test: 41.0620973085, MISSFRAC_test: 0.0
Iteration 4:
CE_train: 42.5784271343, MISSFRAC_train: 0.0
CE_valid: 84.9683294723, MISSFRAC_valid: 0.06
CE_test: 42.5784271343, MISSFRAC_test: 0.0
Iteration 5:
CE_train: 49.8034126724, MISSFRAC_train: 0.0
CE_valid: 74.6499798367, MISSFRAC_valid: 0.04
CE_test: 49.8034126724, MISSFRAC_test: 0.0
```

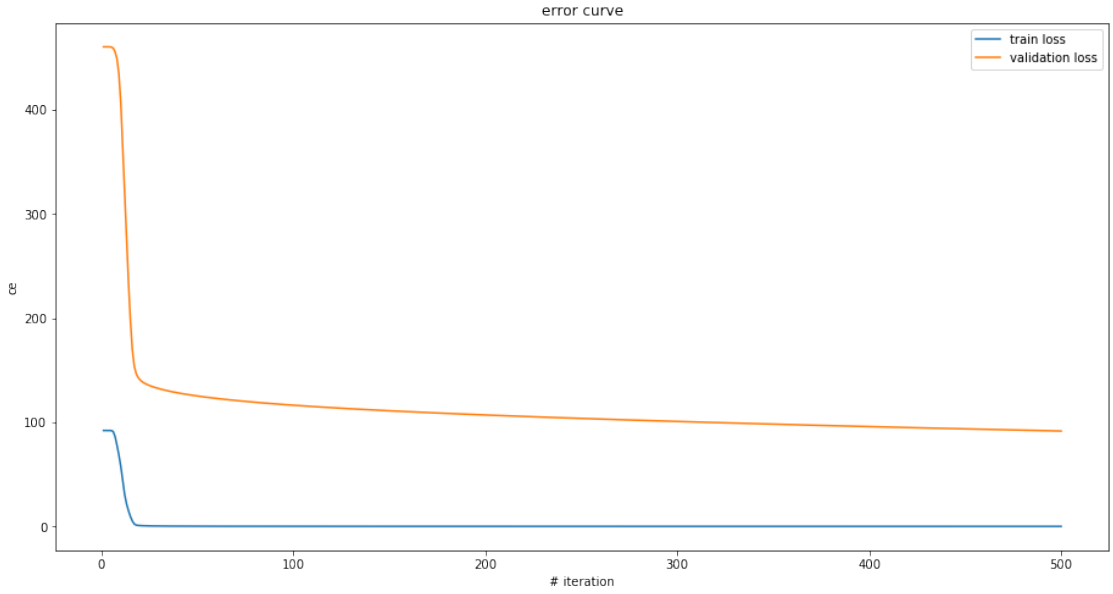
0.2.4 Run penalized logistic regression on small dataset

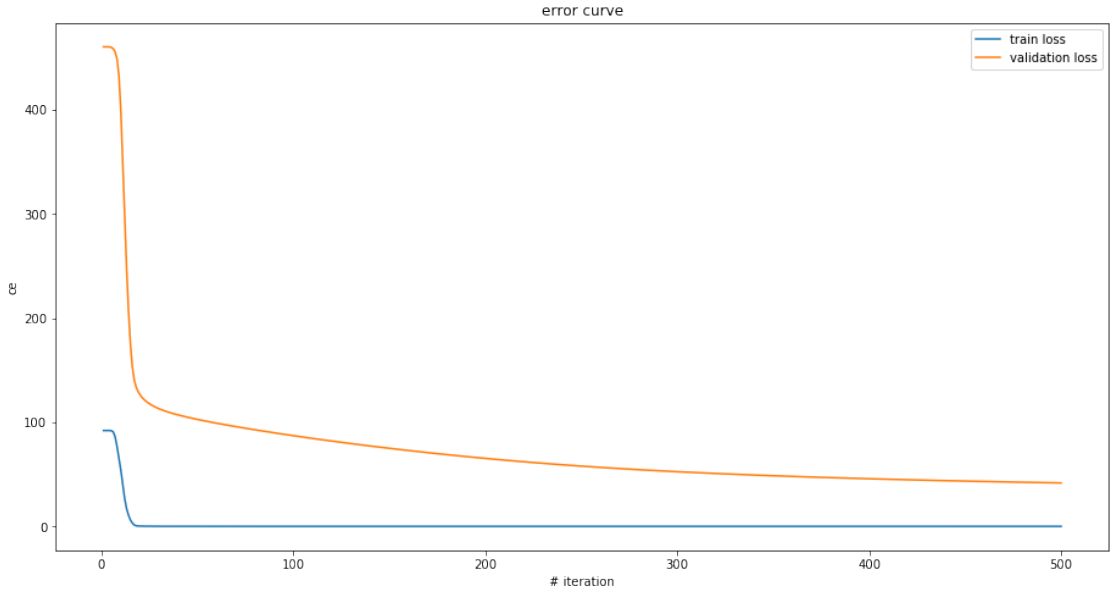
[88]:
```
train_inputs, train_targets = load_train_small()
```

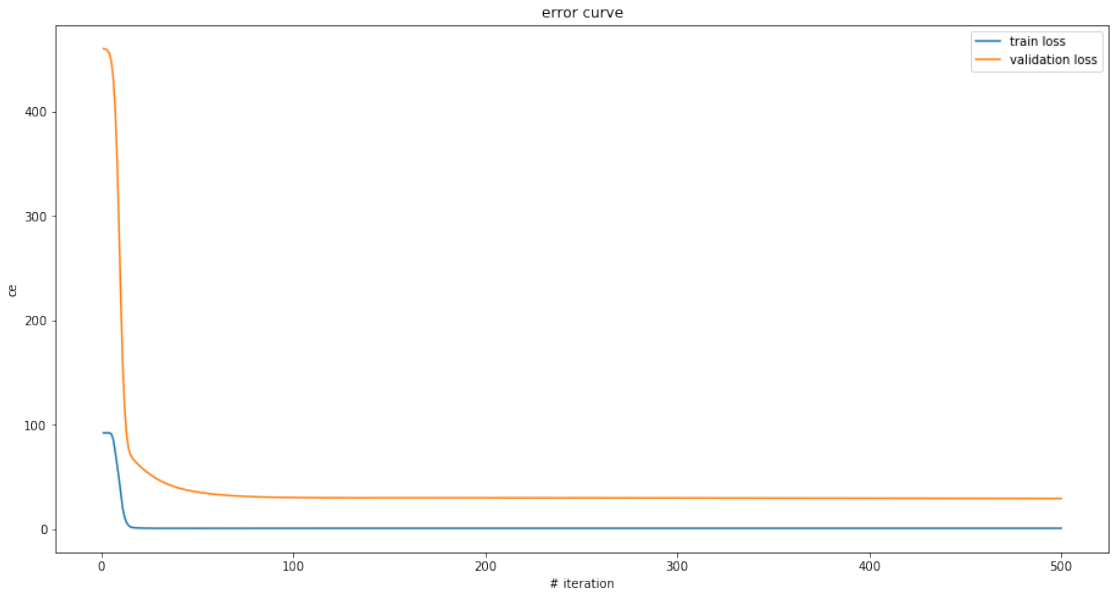
[89]:
```
%matplotlib inline
train_ce_avgs = []
valid_ce_avgs = []
train_fe_avgs = []
valid_fe_avgs = []
lambdas = [0, 0.001, 0.01, 0.1, 1.0]
weightss = []
for lambd in lambdas:
    hyperparameters['weight_regularization'] = lambd
    train_ce_sum, train_fe_sum = 0, 0
    valid_ce_sum, valid_fe_sum = 0, 0
    for i in range(5):
        weights, train_losses, valid_losses = \
            run_logistic_regression(train_inputs, train_targets, valid_inputs,␣
↪valid_targets, hyperparameters)
        # Predict with the weights
        y_train = logistic_predict(weights, train_inputs)
        y_valid = logistic_predict(weights, valid_inputs)
        # Compute the cross entrophy error and fraction error
        train_ce, train_fe = evaluate(train_targets, y_train)
        valid_ce, valid_fe = evaluate(valid_targets, y_valid)
        # Compute misclassification rate
        train_fe, valid_fe = 1.0 - train_fe, 1.0 - valid_fe
        # Compute the sum of errors in 5 turns
        train_ce_sum, train_fe_sum = train_ce_sum + train_ce, train_fe_sum +␣
↪train_fe
        valid_ce_sum, valid_fe_sum = valid_ce_sum + valid_ce, valid_fe_sum +␣
↪valid_fe
        if i == 4:
            plot_losses(train_losses, valid_losses)
    # Append the avgs
    train_ce_avgs.append(train_ce_sum / 5.0)
    valid_ce_avgs.append(valid_ce_sum / 5.0)
    train_fe_avgs.append(train_fe_sum / 5.0)
    valid_fe_avgs.append(valid_fe_sum / 5.0)
    weightss.append(weights)
```


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500


error curve
train loss
validation loss
ce
iteration
0
100
200
300
400
500

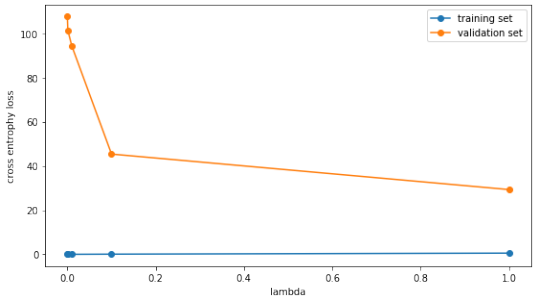


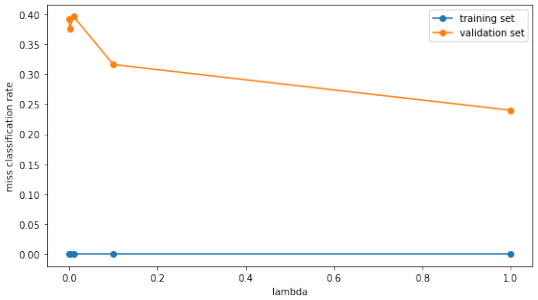
0.2.5 Plot the graphs

[90]:
```
plt.figure(figsize=(20, 5))
plt.subplot(121)
plt.plot(lambdas, train_ce_avgs, '-o', label="training set")
plt.plot(lambdas, valid_ce_avgs, '-o', label="validation set")
plt.xlabel("lambda")
plt.ylabel("cross entrophy loss")
plt.legend()

plt.subplot(122)
plt.plot(lambdas, train_fe_avgs, '-o', label="training set")
plt.plot(lambdas, valid_fe_avgs, '-o', label="validation set")
plt.xlabel("lambda")
plt.ylabel("miss classification rate")
plt.legend()
```

[90]:
```
<matplotlib.legend.Legend at 0x7fb20b0def10>
```





[91]:
```
# Report the best hyperparemeter setting
argmin = np.argmin(valid_ce_avgs)
print("Best lambda setting:")
print(lambdas[argmin])
for i in range(5):
    print("Iteration {0}: ".format(i + 1))
    report_errors(weightss[i], train_inputs, train_targets, valid_inputs,␣
 ↪valid_targets, test_inputs, test_targets)
```

```
Best lambda setting:
1.0
Iteration 1:
CE_train: 0.238065732443, MISSFRAC_train: 0.0
CE_valid: 351.811191892, MISSFRAC_valid: 0.42
CE_test: 1053.15105128, MISSFRAC_test: 0.3625
Iteration 2:
CE_train: 0.277737797047, MISSFRAC_train: 0.0
CE_valid: 378.571513844, MISSFRAC_valid: 0.38
CE_test: 1125.71133289, MISSFRAC_test: 0.3875
Iteration 3:
CE_train: 0.300291217389, MISSFRAC_train: 0.0
CE_valid: 376.784600517, MISSFRAC_valid: 0.44
CE_test: 968.124116102, MISSFRAC_test: 0.325
Iteration 4:
CE_train: 1.65767820029, MISSFRAC_train: 0.0
CE_valid: 317.607770185, MISSFRAC_valid: 0.32
CE_test: 891.038081042, MISSFRAC_test: 0.29375
Iteration 5:
CE_train: 9.74684898525, MISSFRAC_train: 0.0
CE_valid: 342.076474772, MISSFRAC_valid: 0.24
CE_test: 1001.9836519, MISSFRAC_test: 0.25625
```

0.2.6 Comment:

The cross entropy of the validation set is fluctuating in a small range when training on the large dataset, but keep reducing when training on the small dataset as lambda increases.

The reason for this is, large datasize has more sample. Therefore the trainning model is more generalized and less likely to overfit. However, when the datasize is small, it is more likely to overfit when training. Therefore, the regularizer penalized the score when the model is overfitting. That's why the change on lambda has more impact on the small dataset than the large dataset.

In conclusion, for large dataset, no penalization is better. For small dataset, lambda = 1 performs better.

[ ]: