The Experiment Report of Machine Learning

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author:
邢浩

Supervisor:
Mingkui Tan

Student ID：201530613221

Grade:
Undergraduate

December 9, 2017

Experimental Study on Stochastic Gradient Descent for Solving Classification Problems

Abstract—

I. INTRODUCTION

A. Motivation of Experiment

1. Compare and understand the difference between gradient descent and stochastic gradient descent.
2. Compare and understand the differences and relationships between Logistic regression and linear classification.
3. Further understand the principles of SVM and practice on larger data.

B. Dataset

Experiment uses a9a of LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features. Please download the training set and validation set.

II. METHODS AND THEORY

A. Experiment Step

The experimental code and drawing are completed on jupyter.

Logistic Regression and Stochastic Gradient Descent

1. Load the training set and validation set.
2. Initalize logistic regression model parameters, you can consider initalizing zeros, random numbers or normal distribution.
3. Select the loss function and calculate its derivation, find more detail in PPT.
4. Calculate gradient toward loss function from **partial samples**.
5. **Update model parameters using different optimized methods(NAG，RMSProp，AdaDelta and Adam).**
6. Select the appropriate threshold, mark the sample whose predict scores **greater than the threshold as positive, on the contrary as negative**. Predict under validation set and get the different optimized method loss，，and .
7. Repeate step 4 to 6 for several times, and **drawing graph of，，and with the number of iterations**.

Linear Classification and Stochastic Gradient Descent

1. Load the training set and validation set.
2. Initalize SVM model parameters, you can consider initalizing zeros, random numbers or normal distribution.
3. Select the loss function and calculate its derivation, find more detail in PPT.
4. Calculate gradient toward loss function from **partial samples**.
5. **Update model parameters using different optimized methods(NAG，RMSProp，AdaDelta and Adam).**
6. Select the appropriate threshold, mark the sample whose predict scores **greater than the threshold as positive, on the contrary as negative**. Predict under validation set and get the different optimized method loss，，and .
7. Repeate step 4 to 6 for several times, and **drawing graph of，，and with the number of iterations**.

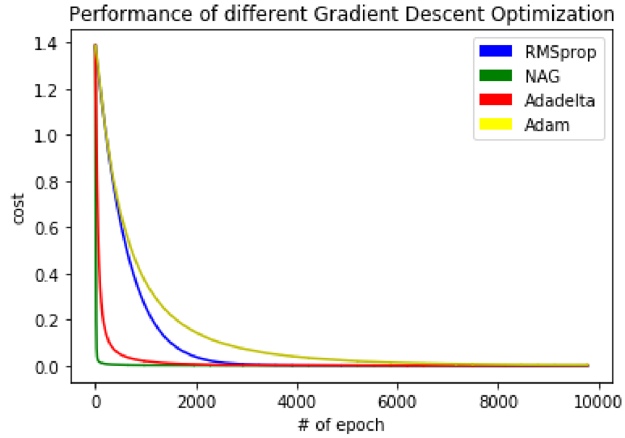
III. EXPERIMENT

Code of Regression

```
class logisticregression:
    def __init__(self, num_input):
        self.num_input = num_input
        self.num_output = 2

    def train(self, x, y, opt_algo, num_epoch=30, mini_batch=100, lambda_=0.01):
        if not opt_algo in opt_algo_set:
            print >> sys.stderr, 'opt_algo not in %s' % opt_algo_set
            return
        print >> sys.stderr, 'optimization with [%s]' % opt_algo

        num_params = 1 * (self.num_input + 1) + self.num_output * 2
        w = np.matrix(0.005 * np.random.random([num_params, 1]))
        data = np.column_stack([x, y])

        gamma = 0.9
        epsilon = 1e-8
```

```
        if opt_algo == 'RMSprop' or opt_algo == 'Adam':
            eta = 0.001
        else:
            eta = 0.05

        v = np.matrix(np.zeros(w.shape))
        m = np.matrix(np.zeros(w.shape))

        # Adam params
        beta1 = 0.9
        beta2 = 0.999
        beta1_exp = 1.0
        beta2_exp = 1.0

        # Adagrad params
        grad_sum_square = np.matrix(np.zeros(w.shape))

        # Adadelta & RMSprop params
        grad_expect = np.matrix(np.zeros(w.shape))
        delta_expect = np.matrix(np.zeros(w.shape))

        first_run = True
        for epoch in range(num_epoch):
            np.random.shuffle(data)
            k = 0
            cost_array = list()
            while k < len(data):
                x = data[k: k + mini_batch, 0: -1]
                y = np.matrix(data[k: k + mini_batch, -1], dtype='int32')
                if opt_algo == 'SGD':
                    # Stochastic gradient descent
                    cost, grad = self.gradient(x, y, lambda_, w)
                    w = w - eta * grad

                elif opt_algo == 'Momentum':
                    # Momentum
                    cost, grad = self.gradient(x, y, lambda_, w)
                    v = gamma * v + eta * grad
                    w = w - v

                elif opt_algo == 'NAG':
                    # Nesterov accelerated gradient
                    cost, grad = self.gradient(x, y, lambda_, w - gamma * v)
                    v = gamma * v + eta * grad
                    w = w - v

                elif opt_algo == 'Adagrad':
                    # Adagrad
                    cost, grad = self.gradient(x, y, lambda_, w)
                    grad_sum_square += np.square(grad)
                    delta = - eta * grad / np.sqrt(grad_sum_square + epsilon)
                    w = w + delta

                elif opt_algo == 'Adadelta':
                    # Adadelta
                    cost, grad = self.gradient(x, y, lambda_, w)
                    grad_expect = gamma * grad_expect + (1.0 - gamma) * np.square(grad)
                    # when first run, use sgd
                    if first_run == True:
                        delta = - eta * grad
                    else:
                        delta = - np.multiply(np.sqrt(delta_expect + epsilon) / np.sqrt(grad_expect + epsilon), grad)
                    w = w + delta
                    delta_expect = gamma * delta_expect + (1.0 - gamma) * np.square(delta)

                elif opt_algo == 'RMSprop':
                    # RMSprop
                    cost, grad = self.gradient(x, y, lambda_, w)
                    grad_expect = gamma * grad_expect + (1.0 - gamma) * np.square(grad)
                    w = w - eta * grad / np.sqrt(grad_expect + epsilon)

                elif opt_algo == 'Adam':
                    # Adam
                    cost, grad = self.gradient(x, y, lambda_, w)
                    m = beta1 * m + (1.0 - beta1) * grad
                    v = beta2 * v + (1.0 - beta2) * np.square(grad)
                    beta1_exp *= beta1
                    beta2_exp *= beta2
                    w = w - eta * (m / (1.0 - beta1_exp)) / (np.sqrt(v / (1.0 - beta2_exp)) + epsilon)

                k += mini_batch
                cost_array.append(cost)
                if first_run == True: first_run = False

            if not opt_algo in plt_dict:
                plt_dict[opt_algo] = list()
            plt_dict[opt_algo].extend(cost_array)
            print >> sys.stderr, 'epoch: [%04d], cost: [%08.4f]' % (epoch, sum(cost_array) / len(cost_array))

        self.w1 = w[0: (self.num_input + 1)].reshape(1, self.num_input + 1)
        self.w2 = w[(self.num_input + 1):].reshape(self.num_output, 2)

    def gradient(self, x, y, lambda_, w):
        # x = data[:, 0: -1]
        # y = np.matrix(data[:, -1], dtype='int32')
        num_sample = len(x)

        w1 = w[0: (self.num_input + 1)].reshape(1, self.num_input + 1)
        w2 = w[(self.num_input + 1):].reshape(self.num_output, 2)
        b = np.matrix(np.ones([num_sample, 1]))
```

```
        a1 = np.column_stack([x, b])
        s2 = sigmoid(a1 * w1.T)
        a2 = np.column_stack([s2, b])
        a3 = sigmoid(a2 * w2.T)

        y_one_hot = np.matrix(np.zeros([num_sample,
self.num_output]))
        y_one_hot[(np.matrix(range(num_sample)), y.T)] = 1

        cost = (1.0 / num_sample) * (
        - np.multiply(y_one_hot, np.log(a3)) - np.multiply(1.0 -
y_one_hot, np.log(1.0 - a3))).sum()
        cost += (lambda_ / (2.0 * num_sample)) *
(np.square(w1[:, 0: -1]).sum() + np.square(w2[:, 0: -1]).sum())

        delta3 = a3 - y_one_hot
        delta2 = np.multiply(delta3 * w2[:, 0: -1],
np.multiply(s2, 1.0 - s2))
        l1_grad = delta2.T * a1
        l2_grad = delta3.T * a2

        r1_grad = np.column_stack([w1[:, 0: -1],
np.matrix(np.zeros([1, 1]))])
        r2_grad = np.column_stack([w2[:, 0: -1],
np.matrix(np.zeros([self.num_output, 1]))])

        w1_grad = (1.0 / num_sample) * l1_grad + (1.0 *
lambda_ / num_sample) * r1_grad
        w2_grad = (1.0 / num_sample) * l2_grad + (1.0 *
lambda_ / num_sample) * r2_grad
        w_grad = np.row_stack([w1_grad.reshape(-1, 1),
w2_grad.reshape(-1, 1)])

        return cost, w_grad

    def predict(self, x):
        num_sample = len(x)
        b = np.matrix(np.ones([num_sample, 1]))
        h1 = sigmoid(np.column_stack([x, b]) * self.w1.T)
        h2 = sigmoid(np.column_stack([h1, b]) * self.w2.T)
        return np.argmax(h2, 1)

    def test(self, x, y):
        num_sample = len(x)
        y_pred = self.predict(x)
        y_one_hot = (np.zeros(y.shape))
        y_one_hot[np.where(y_pred == y)[0]] = 1
        acc = 1.0 * y_one_hot.sum() / num_sample
        return acc
```



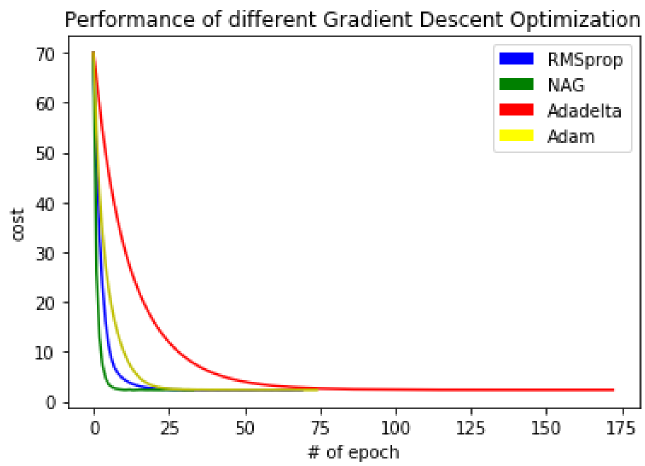
Code of Classification

```
def HingeLoss(y, y_true):

    if not (y.shape[0] == y_true.shape[0]):
        print 'Mismatching of input ndarray input shapes in
function "HingeLoss", line:', cline()
        print 'y.shape =', y.shape, 'y_true.shape =', y.shape
        sys.exit()
    #output=max(0,1-y*y_true)
    output = 1.0 - (y * y_true)
    output *= np.asarray((output > 0.0),dtype=float)

    return output

def Accuracy(y, y_true):

    if not (y.shape[0] == y_true.shape[0]):
        print 'Mismatching of input ndarray input shapes in
function "Accuracy", line:', cline()
        print 'y.shape =', y.shape, 'y_true.shape =', y.shape
        sys.exit()

    output = np.sum(y_true == y) / float(y.shape[0])

    return output

class SGD:
    def __init__(self, model, x, y,opt_algo='SGD',
batch_size=1, lambda_=1.0,
                learning_rate=0.0001, loss_type='HingeLoss',
                regularization='L2'):

        self._loss_type = loss_type
        self._regularization = regularization
        self._lambda = lambda_
        self._lrate = learning_rate
        self._bsize = batch_size
        self._model = model
        self.x = x
        self.y = y
        self.used_samples = np.zeros(self.x.shape[0])
        self.opt_algo=opt_algo
```

```
        self.first_run = True

        self.v = np.zeros(self._model.features_size + 1)
        self.m = np.zeros(self._model.features_size + 1)
        self.grad_expect = np.zeros(self._model.features_size +
1)
        self.beta1_exp = 1.0
        self.beta2_exp = 1.0
        self.delta_expect = np.zeros(self._model.features_size
+ 1)


    def get_batch(self): # start name of func with _

        idxs = np.random.choice(np.where(self.used_samples
== 0)[0], self._bsize)
        batch_x = self.x[idxs]
        batch_y = self.y[idxs]
        self.used_samples[idxs] = 1.0

        return batch_x, batch_y

    def calc_loss(self, batch_x, batch_y, loss=None):

        if not (batch_x.shape[0] == batch_y.shape[0]):
            print 'Input arrays shapes mismatching in \
                    SGD.calc_loss(), line', cline()
            sys.exit()
        if loss is None:
            loss = self._loss_type
        output_loss = 0.0

        # HingeLoss
        if loss == 'HingeLoss':
            batch_y_predicted = self._model.predict(batch_x,
binar=False)
            output_loss =
np.mean(HingeLoss(batch_y_predicted, batch_y))

        # Accuracy
        elif loss == 'Accuracy':
            batch_y_predicted = self._model.predict(batch_x)
            output_loss = Accuracy(batch_y_predicted, batch_y)
        else:
            print 'Unknown loss type in SGD.calc_loss, line',
cline()
            sys.exit()



        if self._regularization == None:
            return output_loss
        elif self._regularization == 'L2':
            if not self._loss_type == 'Accuracy':
                output_loss += 0.5 * self._lambda *
np.sum(self._model.wb[:-1]**2)
            return output_loss
        else:
            print 'Unknown regularization type in SGD.calc_loss,
line', cline()
            sys.exit()

    def grad(self, batch_x, batch_y):
        if not (batch_x.shape[0] == batch_y.shape[0]):
            print 'Input arrays shapes mismatching in \
                    SGD.grad(), line', cline()
            sys.exit()

        grad_matr = np.zeros([self._bsize,
self._model.features_size + 1])
        output = np.zeros(self._model.features_size + 1)
        if self._loss_type == 'HingeLoss':
            batch_y_predicted = self._model.predict(batch_x,
binar=False)
            current_loss = HingeLoss(batch_y_predicted,
batch_y)
            for i in range(self._bsize):
                if current_loss[i] == 0.0:
                    continue
                else:
                    for j in range(self._model.features_size):
                        grad_matr[i, j] = -batch_y[i] * batch_x[i, j]
                    grad_matr[i,self._model.features_size] = -
batch_y[i]

            output = np.sum(grad_matr, axis=0)

        if self._regularization == None:
            return output
        elif self._regularization == 'L2':
            return output + self._lambda * self._model.wb

    def step(self, batch_x, batch_y):
        gamma = 0.9
        epsilon = 1e-8
        if self.opt_algo == 'RMSprop' or self.opt_algo ==
'Adam':
            self._lrate = 0.001
        else:
            self._lrate = 0.0001

        # Adam params
        beta1 = 0.9
        beta2 = 0.999

        if self.opt_algo=='SGD':
            self._model.wb -= self._lrate * self.grad(batch_x,
batch_y)
        elif self.opt_algo=='NAG':
            self.v = gamma * self.v + self._lrate *
self.grad(batch_x, batch_y)
            self._model.wb = self._model.wb - self.v
        elif self.opt_algo == 'Adadelta':
            self.grad_expect = gamma * self.grad_expect + (1.0 -
```

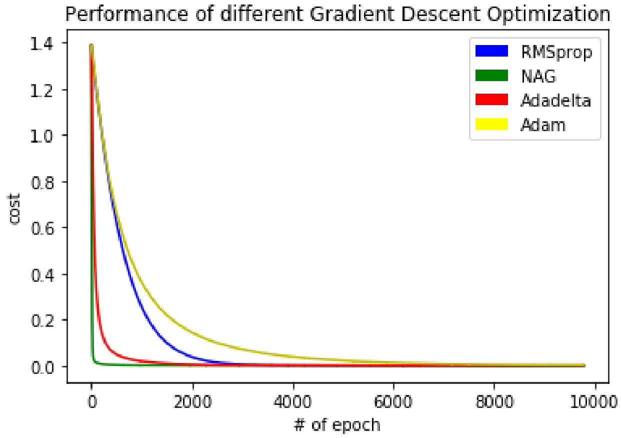
```
gamma) * np.square(self.grad(batch_x, batch_y))
            if self.first_run == True:
               delta = - self._lrate * self.grad(batch_x, batch_y)
            else:
               delta = - np.multiply(np.sqrt(self.delta_expect + epsilon) / np.sqrt(self.grad_expect + epsilon), self.grad(batch_x, batch_y))
            self._model.wb = self._model.wb + delta
            self.delta_expect = gamma * self.delta_expect + (1.0 - gamma) * np.square(delta)
         elif self.opt_algo == 'RMSprop':
            # RMSprop
            grad = self.grad(batch_x, batch_y)
            self.grad_expect = gamma * self.grad_expect + (1.0 - gamma) * np.square(grad)
            self._model.wb = self._model.wb - self._lrate * grad / np.sqrt(self.grad_expect + epsilon)
         elif self.opt_algo == 'Adam':
            # Adam
            grad = self.grad(batch_x, batch_y)
            self.m = beta1 * self.m + (1.0 - beta1) * grad
            self.v = beta2 * self.v + (1.0 - beta2) * np.square(grad)
            self.beta1_exp *= beta1
            self.beta2_exp *= beta2
            self._model.wb = self._model.wb - self._lrate * (self.m / (1.0 - self.beta1_exp)) / (np.sqrt(self.v / (1.0 - self.beta2_exp)) + epsilon)

         if self.first_run == True: first_run = False

      def make_epoch(self):
         while np.sum(self.used_samples) < self.used_samples.shape[0]:
            batch_x, batch_y = self.get_batch()
            self.step(batch_x, batch_y)

         #back to initial statement
         self.used_samples = np.zeros(self.x.shape[0])

   VERY_BIG_NUMBER = 70.0

   class SVM:
      def __init__(self, loss='HingeLoss', optimizer='SGD',
               chronicle_loss_history=True, chronicle_model_history=False,earlystop=20):
         #! make comments in correct form
         # Here is defined model parameters as one array, where
         # 'b' is represented as the last element in wb
         self.istrained = False
         self.loss = loss
         self.optimizer = optimizer

         self.wb = None

         self.epoch_learned = 0
         self.train_loss = VERY_BIG_NUMBER
         self.test_loss = VERY_BIG_NUMBER
         self.early_stop=earlystop


         if chronicle_loss_history:
            self.train_loss_history  = []
         else:
            self.train_loss_history  = None

         if chronicle_model_history:
            self.train_model_history = []
         else:
            self.train_model_history = None

      #! fit doesn't use X_test, I need new func for evaluation
      def fit(self, X_train, y_train, opt_algo,X_test=None, y_test=None, batch_size=15,
            n_epoch=1, learning_rate=0.0001, verbose=True):

         time_start = time.time()
         if not self.optimizer == 'SGD':
            print 'Unknown optimizer type! SVM.fit, line', cline()
            sys.exit()

         if not X_train.ndim == 2:
            print 'X_train has bad shapes in SVM.fit, line', cline()
            sys.exit()

         if not (X_train.shape[0] == y_train.shape[0]):
            print 'Input arrays shapes mismatching in SVM.fit, line', cline()
            sys.exit()

         if not self.istrained:
            self.features_size = X_train.shape[1]
            self.wb = np.random.randn(self.features_size + 1) # 采用随机初始化
            # self.wb = np.zeros(self.features_size + 1)
            if not (self.train_loss_history is None):
               self.train_loss_history.append([self.train_loss, self.test_loss])
            if not (self.train_model_history is None):
               self.train_model_history.append(list(self.wb))

         if self.optimizer == 'SGD':
            if self.epoch_learned == 0:
               solver = SGD(self, X_train, y_train, loss_type=self.loss,
                        batch_size=batch_size, learning_rate=learning_rate,opt_algo=opt_algo)

            min_loss = VERY_BIG_NUMBER
            best_epoch=VERY_BIG_NUMBER
            min_loss_train=VERY_BIG_NUMBER
            without_updates = 0
```

```
            while self.epoch_learned < n_epoch:
                solver.make_epoch()
                self.epoch_learned += 1
                self.train_loss = solver.calc_loss(X_train, y_train)

                if not (X_test is None):
                    self.test_loss = solver.calc_loss(X_test, y_test)

                if verbose:
                    print 'epoch %d, train_loss %1.3lf, test_loss %1.3lf'%\
                        (self.epoch_learned, self.train_loss, self.test_loss)
                self.istrained = True
                if self.test_loss < min_loss:
                    min_loss = self.test_loss
                    min_loss_train=self.train_loss
                    best_epoch=self.epoch_learned
                    without_updates = 0
                else:
                    without_updates += 1
                    if without_updates > self.early_stop:
                        print 'Loss on the test set stops decreasing for',self.early_stop,'times,triggered early stop'
                        break
                if not (self.train_loss_history is None):
                    self.train_loss_history.append([self.train_loss, self.test_loss])
                if not (self.train_model_history is None):
                    self.train_model_history.append(list(self.wb))

        time_finish = time.time()
        print 'best epoch %d, time %1.2lfs, best train_loss %1.3lf, for test_loss %1.3lf'%\
            (best_epoch, time_finish-time_start, min_loss_train, min_loss)

    def predict(self, X, binar=True):
        # Required type(X) equal to np.ndarray with dim = 2
        if not type(X) == np.ndarray:
            print 'Wrong input type in function "SVM.predict", line:', cline()
            sys.exit()
        if not X.ndim == 2:
            print 'Wrong input ndarray size in function "SVM.predict", line:', cline()
            sys.exit()

        if binar:#二分类
            return np.sign(np.dot(X, self.wb[:-1]) + self.wb[-1])
        else:
            return np.dot(X, self.wb[:-1]) + self.wb[-1]

    def get_accuracy(self, X, y_true):

        if not (X.shape[0] == y_true.shape[0]):
            print 'Input arrays shapes mismatching in SVM.get_accuracy, line', cline()

        y = self.predict(X)

        return np.sum(y == y_true) / float(y.shape[0])

    def export(self):
        pass
```



IV. CONCLUSION

1. Both of the experiments chose stay out method as assessment method
2. The initialization method is clf = logisticregression(num_feature) for regression.
   And clf = SVM(earlystop=250) for classification.
3. Regression choose function mse as its loss function.
   Classifiction choose function hinge as its loss function.
4. Parameter choose
   Regression: η =0.001(RMSprop，Adam),0.05（Adadelta,NAG）,epoch=30
   Classification: η =0.001(RMSprop，Adam),0.05（Adadelta,NAG）,epoch=200,early_stop=20
5. Loss Graph
   Regression:



Classification: