華南理工大學

South China University of Technology

The Experiment Report of *Machine Learning*

College	Software College
Subject	Software Engineering
Members	
Student ID	201530613719
E-mail	g2369969039@gmail.com
Tutor	Mingkui Tan
Date submitted	2017.12.15

1. Topic:

Logistic Regression, Linear Classification, and Stochastic Gradient Descent.

2. Time:

December 9, 2017

3. Reporter:

Ziwei Zhang

4. Purposes:

1) Compare and understand the difference between gradient descent and stochastic gradient descent.
2) Compare and understand different optimization algorithm used in updating model parameter.
3) Compare and understand the differences and relationships between Logistic regression and linear classification.
4) Further understand the principles of SVM and practice on larger data.

5. Data sets and data analysis:

Experiment uses a9a of LIBSVM Data, including 32561/16281(testing) samples and each sample has 123/123 (testing) features together with one label.

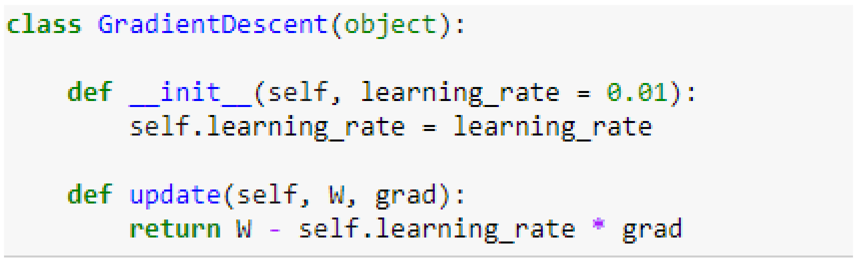
6. Experimental steps:

1) Load and preprocess data.
2) Implement 5 optimization algorithm: Gradient Descent, NAG, RMSProp, AdaDelta, Adam.
3) Implement Logistic Regression model and SVM model.
4) Compare Gradient Descent and Stochastic Gradient Descent.
5) Compare different optimization algorithm.
6) Tune the model and compute the accuracy on test data.
7) Analyze and record the results.

7. Code:

1) Optimization algorithm:

- Gradient Descent



```python
class GradientDescent(object):

    def __init__(self, learning_rate = 0.01):
        self.learning_rate = learning_rate

    def update(self, W, grad):
        return W - self.learning_rate * grad
```

- NAG

```
class NAG(object):

    def __init__(self, learning_rate = 0.01, mu = 0.9):
        self.v = None
        self.learning_rate = learning_rate
        self.mu = mu

    def update(self, W, grad):
        if self.v is None:
            self.v = np.zeros(W.shape)
        v_prev = self.v
        self.v = self.mu * self.v + self.learning_rate * grad
        W -= self.mu * v_prev + (1 + self.mu) * self.v
        return W
```

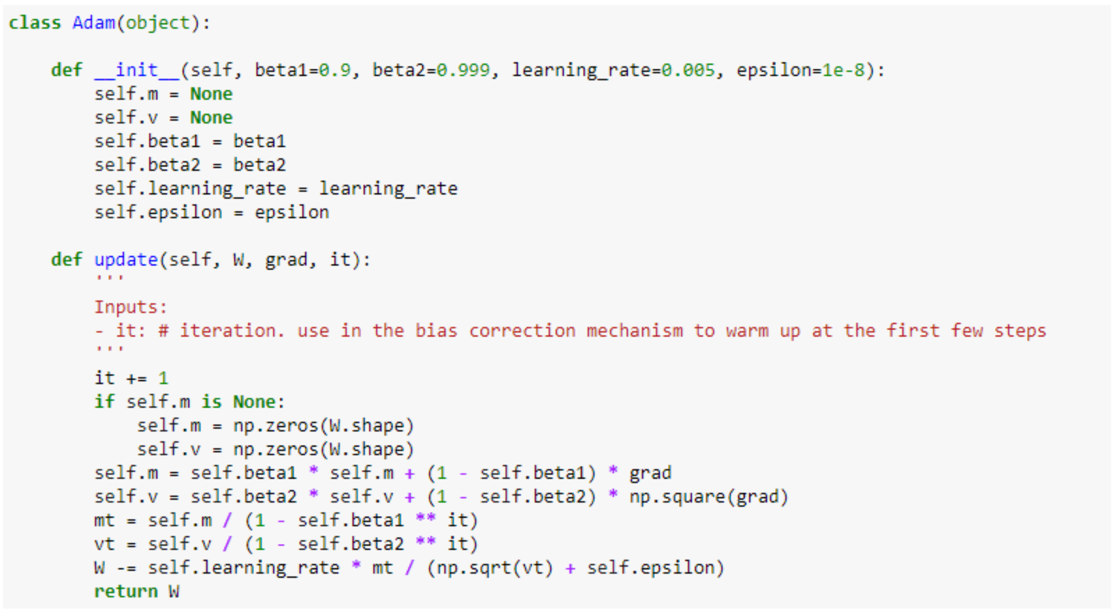
- RMSProp

```
class RMSProp(object):

    def __init__(self, learning_rate = 0.01, decay_rate = 0.9):
        self.cache = None
        self.learning_rate = learning_rate
        self.decay_rate = decay_rate

    def update(self, W, grad):
        if self.cache is None:
            self.cache = np.zeros(W.shape)
        self.cache = self.decay_rate * self.cache + (1 - self.decay_rate) * np.square(grad)
        W -= self.learning_rate * grad / (np.sqrt(self.cache) + 1e-5)
        return W
```

- AdaDelta

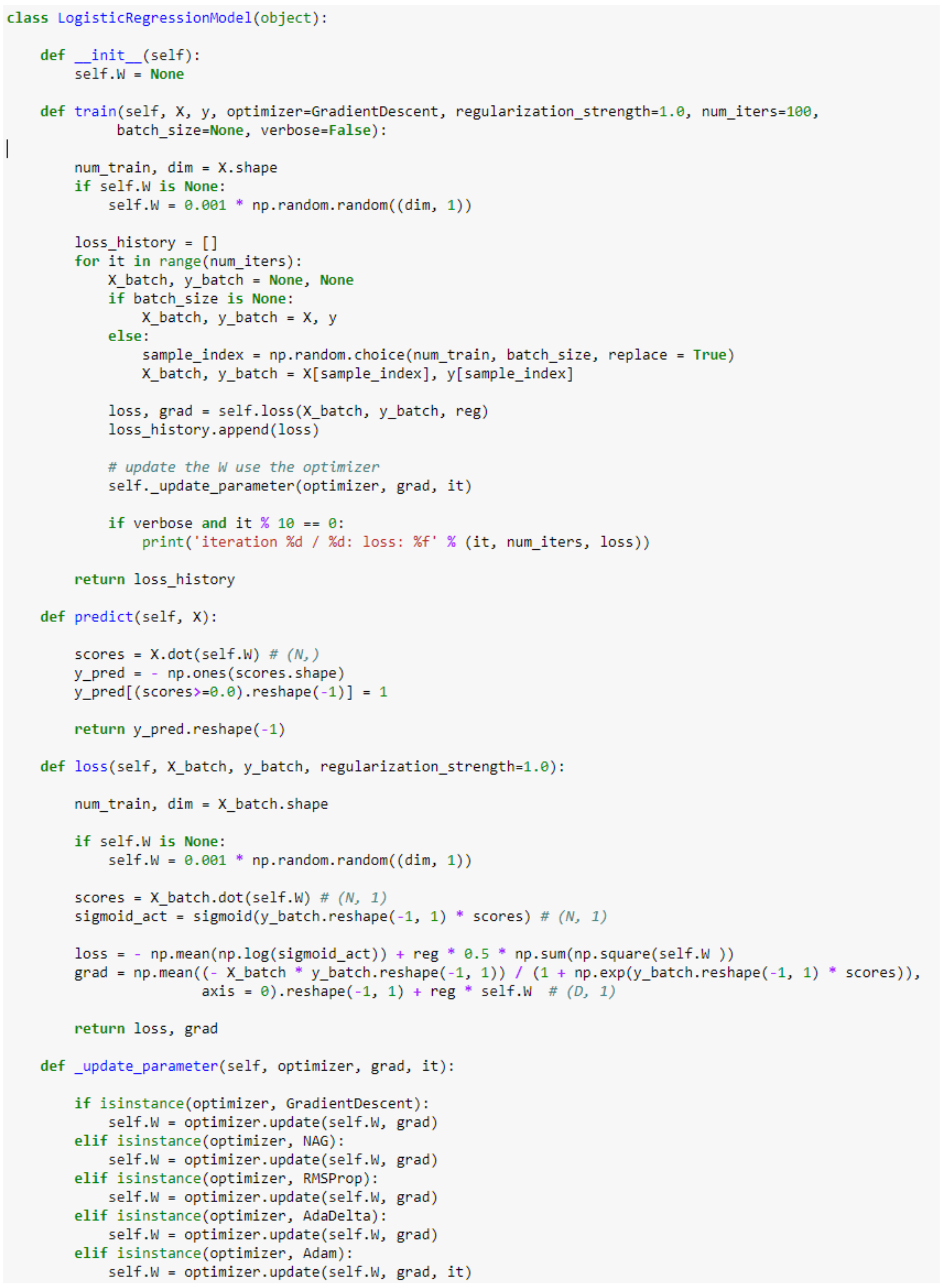
```
class AdaDelta(object):

    def __init__(self, decay_rate = 0.9):
        self.accumulate_grad = None
        self.accumulate_update = None
        self.decay_rate = decay_rate

    def update(self, W, grad):
        if self.accumulate_grad is None:
            self.accumulate_grad = np.zeros(W.shape)
            self.accumulate_update = np.zeros(W.shape)
        self.accumulate_grad = self.decay_rate * self.accumulate_grad \
                               + (1 - self.decay_rate) * np.square(grad)
        delta = - (np.sqrt(self.accumulate_update + 1e-5) / np.sqrt(self.accumulate_grad + 1e-5)) * grad
        W += delta
        self.accumulate_update = self.decay_rate * self.accumulate_update \
                               + (1 - self.decay_rate) * np.square(delta)
        return W
```

- Adam



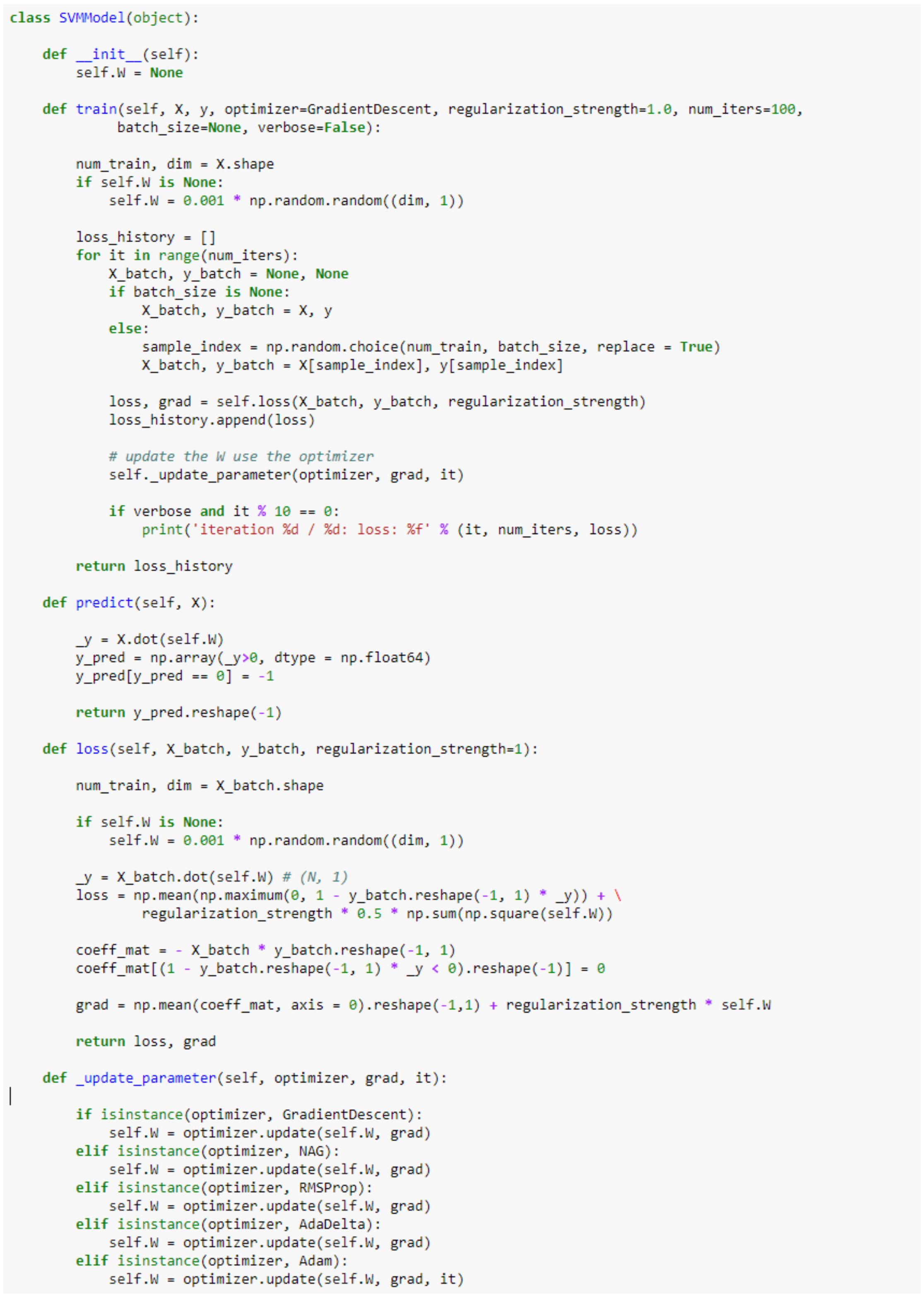
```python
class Adam(object):

    def __init__(self, beta1=0.9, beta2=0.999, learning_rate=0.005, epsilon=1e-8):
        self.m = None
        self.v = None
        self.beta1 = beta1
        self.beta2 = beta2
        self.learning_rate = learning_rate
        self.epsilon = epsilon

    def update(self, W, grad, it):
        '''
        Inputs:
        - it: # iteration. use in the bias correction mechanism to warm up at the first few steps
        '''
        it += 1
        if self.m is None:
            self.m = np.zeros(W.shape)
            self.v = np.zeros(W.shape)
        self.m = self.beta1 * self.m + (1 - self.beta1) * grad
        self.v = self.beta2 * self.v + (1 - self.beta2) * np.square(grad)
        mt = self.m / (1 - self.beta1 ** it)
        vt = self.v / (1 - self.beta2 ** it)
        W -= self.learning_rate * mt / (np.sqrt(vt) + self.epsilon)
        return W
```

2) Logistic Regression model



```python
class LogisticRegressionModel(object):

    def __init__(self):
        self.W = None

    def train(self, X, y, optimizer=GradientDescent, regularization_strength=1.0, num_iters=100,
             batch_size=None, verbose=False):

        num_train, dim = X.shape
        if self.W is None:
            self.W = 0.001 * np.random.random((dim, 1))

        loss_history = []
        for it in range(num_iters):
            X_batch, y_batch = None, None
            if batch_size is None:
                X_batch, y_batch = X, y
            else:
                sample_index = np.random.choice(num_train, batch_size, replace = True)
                X_batch, y_batch = X[sample_index], y[sample_index]

            loss, grad = self.loss(X_batch, y_batch, reg)
            loss_history.append(loss)

            # update the W use the optimizer
            self._update_parameter(optimizer, grad, it)

            if verbose and it % 10 == 0:
                print('iteration %d / %d: loss: %f' % (it, num_iters, loss))

        return loss_history

    def predict(self, X):

        scores = X.dot(self.W) # (N,)
        y_pred = - np.ones(scores.shape)
        y_pred[(scores>=0.0).reshape(-1)] = 1

        return y_pred.reshape(-1)

    def loss(self, X_batch, y_batch, regularization_strength=1.0):

        num_train, dim = X_batch.shape

        if self.W is None:
            self.W = 0.001 * np.random.random((dim, 1))

        scores = X_batch.dot(self.W) # (N, 1)
        sigmoid_act = sigmoid(y_batch.reshape(-1, 1) * scores) # (N, 1)

        loss = - np.mean(np.log(sigmoid_act)) + reg * 0.5 * np.sum(np.square(self.W ))
        grad = np.mean((- X_batch * y_batch.reshape(-1, 1)) / (1 + np.exp(y_batch.reshape(-1, 1) * scores)),
                       axis = 0).reshape(-1, 1) + reg * self.W  # (D, 1)

        return loss, grad

    def _update_parameter(self, optimizer, grad, it):

        if isinstance(optimizer, GradientDescent):
            self.W = optimizer.update(self.W, grad)
        elif isinstance(optimizer, NAG):
            self.W = optimizer.update(self.W, grad)
        elif isinstance(optimizer, RMSProp):
            self.W = optimizer.update(self.W, grad)
        elif isinstance(optimizer, AdaDelta):
            self.W = optimizer.update(self.W, grad)
        elif isinstance(optimizer, Adam):
            self.W = optimizer.update(self.W, grad, it)
```

3) SVM model



```python
class SVMModel(object):

    def __init__(self):
        self.W = None

    def train(self, X, y, optimizer=GradientDescent, regularization_strength=1.0, num_iters=100,
             batch_size=None, verbose=False):

        num_train, dim = X.shape
        if self.W is None:
            self.W = 0.001 * np.random.random((dim, 1))

        loss_history = []
        for it in range(num_iters):
            X_batch, y_batch = None, None
            if batch_size is None:
                X_batch, y_batch = X, y
            else:
                sample_index = np.random.choice(num_train, batch_size, replace = True)
                X_batch, y_batch = X[sample_index], y[sample_index]

            loss, grad = self.loss(X_batch, y_batch, regularization_strength)
            loss_history.append(loss)

            # update the W use the optimizer
            self._update_parameter(optimizer, grad, it)

            if verbose and it % 10 == 0:
                print('iteration %d / %d: loss: %f' % (it, num_iters, loss))

        return loss_history

    def predict(self, X):

        _y = X.dot(self.W)
        y_pred = np.array(_y>0, dtype = np.float64)
        y_pred[y_pred == 0] = -1

        return y_pred.reshape(-1)

    def loss(self, X_batch, y_batch, regularization_strength=1):

        num_train, dim = X_batch.shape

        if self.W is None:
            self.W = 0.001 * np.random.random((dim, 1))

        _y = X_batch.dot(self.W) # (N, 1)
        loss = np.mean(np.maximum(0, 1 - y_batch.reshape(-1, 1) * _y)) + \
                regularization_strength * 0.5 * np.sum(np.square(self.W))

        coeff_mat = - X_batch * y_batch.reshape(-1, 1)
        coeff_mat[(1 - y_batch.reshape(-1, 1) * _y < 0).reshape(-1)] = 0

        grad = np.mean(coeff_mat, axis = 0).reshape(-1,1) + regularization_strength * self.W

        return loss, grad

    def _update_parameter(self, optimizer, grad, it):

        if isinstance(optimizer, GradientDescent):
            self.W = optimizer.update(self.W, grad)
        elif isinstance(optimizer, NAG):
            self.W = optimizer.update(self.W, grad)
        elif isinstance(optimizer, RMSProp):
            self.W = optimizer.update(self.W, grad)
        elif isinstance(optimizer, AdaDelta):
            self.W = optimizer.update(self.W, grad)
        elif isinstance(optimizer, Adam):
            self.W = optimizer.update(self.W, grad, it)
```

8. The initialization method of model parameters:

Random initialization with small value.

9. The selected loss function and its derivatives:

1) Loss function and gradient of Logistic Regression

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} log(1 + e^{-y_i w^T x_i}) + \frac{\lambda}{2}\|w\|_2^2$$

$$\frac{\partial \mathcal{L}}{\partial w} = -\frac{1}{n}\sum_{i=1}^{n}\frac{y_i x_i}{1 + e^{y_i w^T x_i}} + \lambda w$$

2) Loss function and gradient function of svm classification

$$\mathcal{L} = \frac{\lambda\|w\|_2^2}{2} + \frac{1}{n}\sum_{i=1}^{n} max(0,1 - yi(w^T xi))$$

$$g_w(xi) = -yixi \quad 1 - yi(w^T xi) \geq 0$$

$$g_w(xi) = 0 \quad 1 - yi(w^T xi) < 0$$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{n}\sum_{i=1}^{n} g_w(xi) + \lambda w$$

10. Experimental results and curve

1) Logistic Regression

- Hyper-parameter selection

  Regularization strength $\lambda = 0$

  Batch size batch_size $=$ 4096

Optimization algorithm	hyperparameter
SGD	Learning rate lr $= 0.005$

NAG	Learning rate $\mathrm{lr} = 0.005$, decay rate $\mathrm{mu} = 0.9$
RMSProp	Learning rate $\mathrm{lr} = 0.005$, decay rate $\mathrm{decay\_rate} = 0.9$
AdaDelta	decay rate $\mathrm{decay\_rate} = 0.9$
Adam	Learning rate $\mathrm{lr} = 0.01$, Decay rate-1 $\mathrm{beta1} = 0.9$, Decay rate-2 $\mathrm{beta1} = 0.999$

- Predicted results

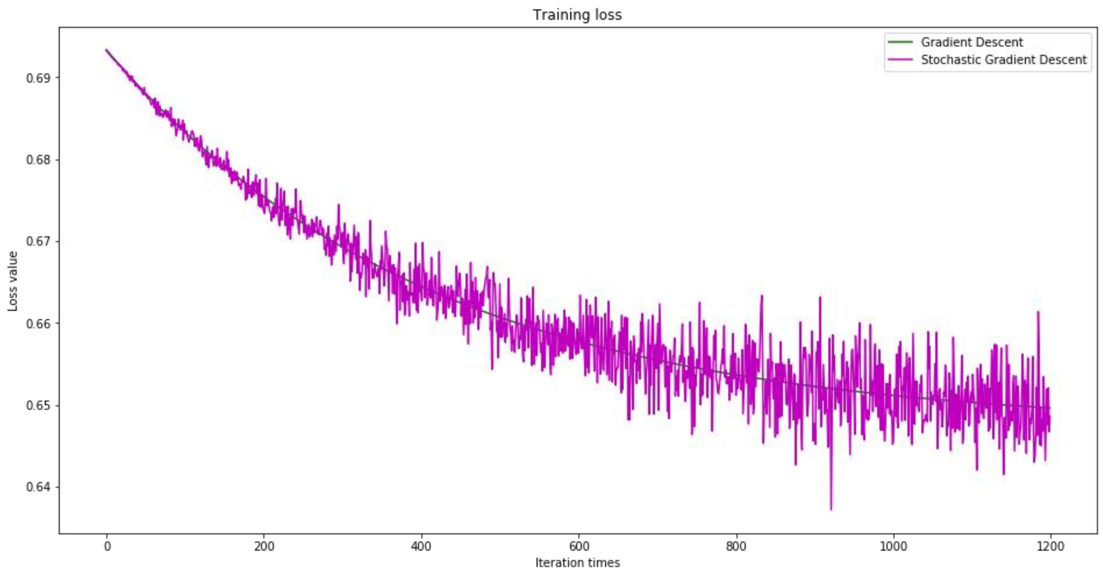
  (With regularization strength equal 0)

  Train accuracy: 84.865%

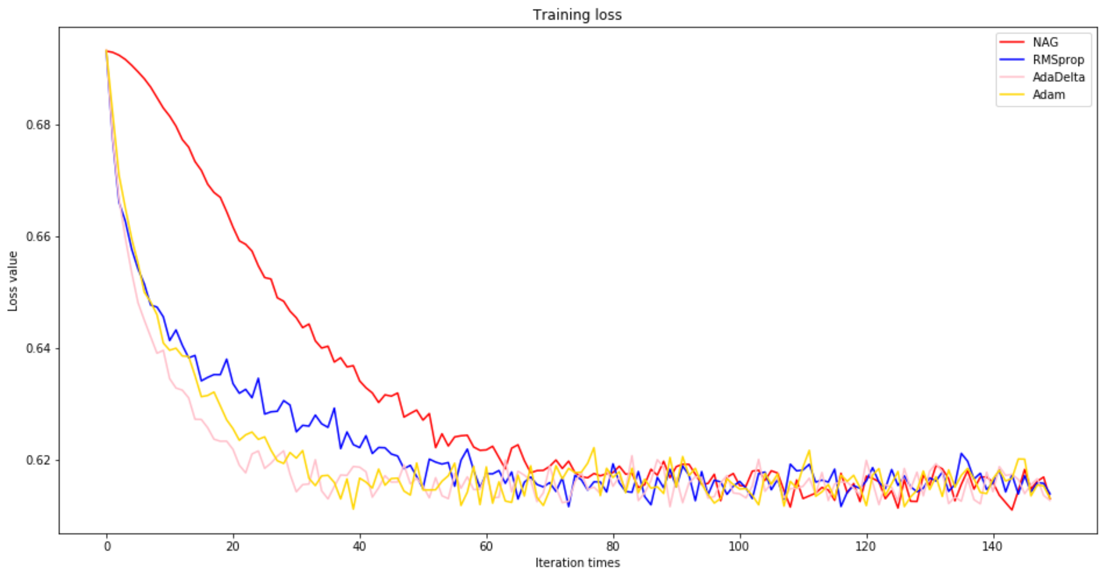
  Test accuracy: 84.804%

- Curve-1 Gradient Descent vs. Stochastic Gradient Descent

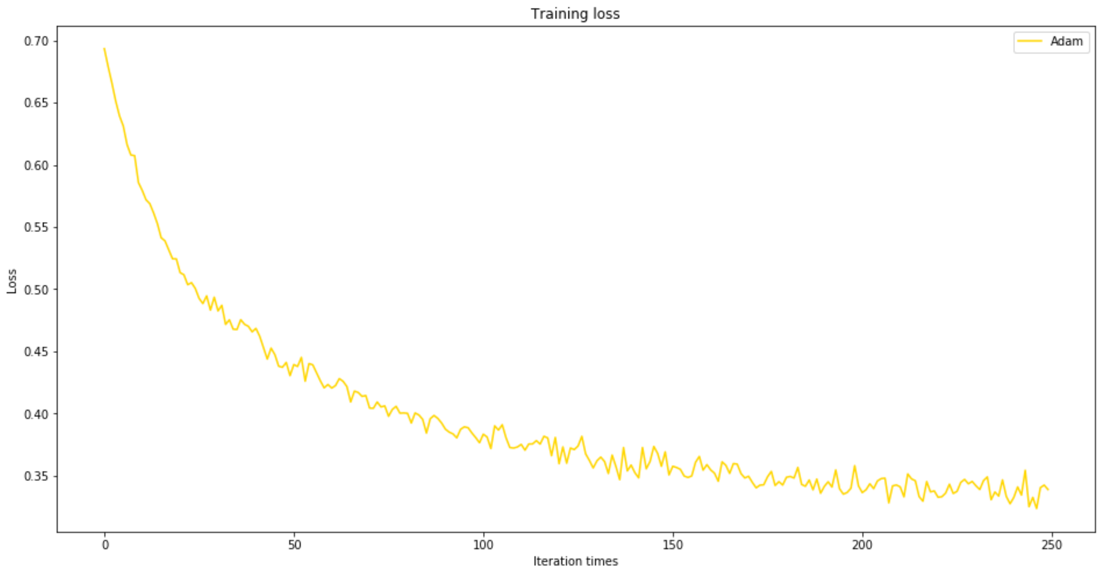
  (with batch size equal to 256)



- Curve-2 NAG & RMSProp & AdaDelta & Adam



- Curve-3 Train model with Adam



2) SVM Classification

- Hyper-parameter selection

  Regularization strength $\lambda = 0$

  Batch size batch_size = 4096

Optimization algorithm	hyperparameter
SGD	Learning rate lr = 0.005
NAG	Learning rate lr = 0.005, decay rate mu = 0.9
RMSProp	Learning rate lr = 0.005, decay rate decay_rate = 0.9
AdaDelta	decay rate decay_rate = 0.9
Adam	Learning rate lr = 0.01, Decay rate-1 beta1 = 0.9,

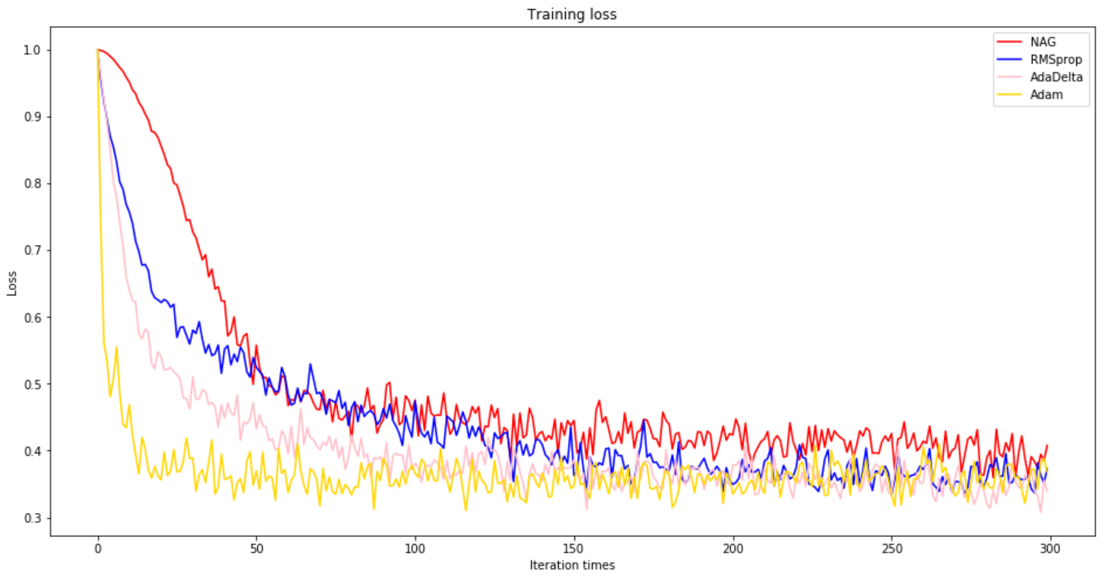
	Decay rate-2 beta1 = 0.999

- Predicted results

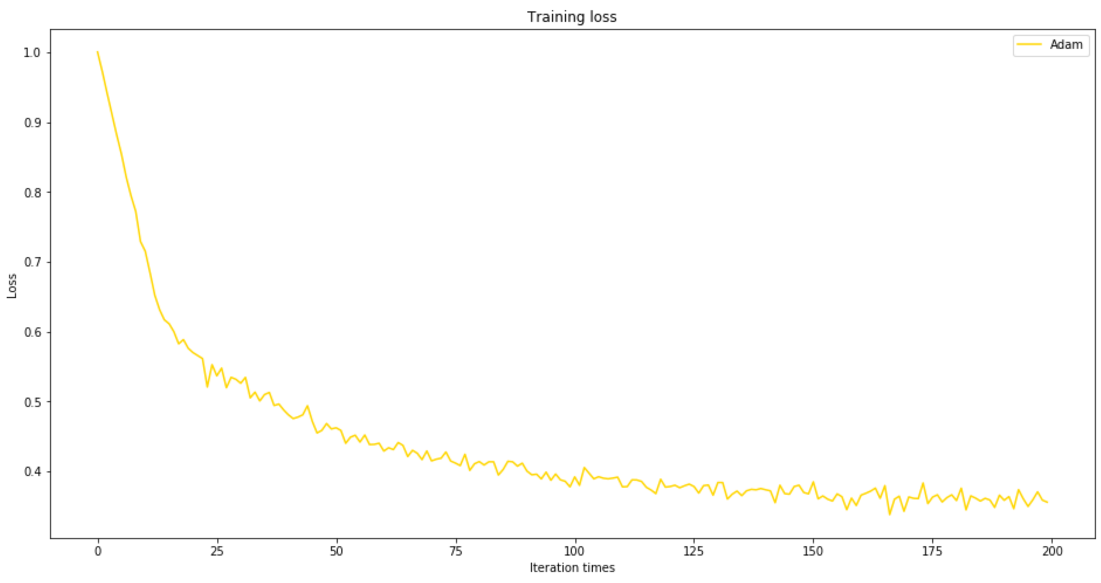
  Train accuracy: 84.755%

  Test accuracy: 84.823%

- Curve-1 NAG & RMSProp & AdaDelta & Adam



- Curve-2 Train model with Adam



11. Results analysis:

1) Stochastic Gradient Descent with batch size 256 (cost 1.24606s for 1200 iterations) is about 35 times faster then Gradient Descent (cost 44.45148s for 1200 iterations). And they achieve approximate accuracy on the test data set. (SGD: 82.15% vs GD: 82.21%)
2) Batch size have an impact on the training progress. With larger batch size, the training progress is more stable. With smaller batch size, the training is faster.
3) Different optimization algorithm behave different on the training progress but can achieve an approximate accuracy on this simple problem.

- The original implementation of Gradient Descent guarantee to make non-negative progress on the loss function.
- The RMSProp update adjusts the Adagrad method using a

moving average of squared gradients to reduce its aggressive, monotonically decreasing learning rate.

- AdaDelta is an adaptive learning rate method. It adjust the learning of per parameter base on their accumulated gradient and their accumulated update.
- Adam is the most sophisticated method and perform pretty well on both logistic regression and SVM classification.

4) On this specified problem, it is strange to find out that setting regularization strength of the model to zero can achieve best test accuracy.

12. Similarities and differences between logistic regression and linear classification：

1) Both logistic regression and linear classification can solve simple linear separable problems.
2) Both logistic regression and linear classification perform a linear transformation on the raw data and then do some interpretation based on the output of the transformation.

13. Summary:

Both logistic regression and SVM can solve simple classification problem. Give priority to SGD + Adam when update model parameters.