SGDSVM

March 24, 2021

If you plan to run the assignment locally: You can download the assignments and run them locally, but please be aware that as much as we would like our code to be universal, computer platform differences may lead to incorrectly reported errors even on correct solutions. Therefore, we encourage you to validate your solution in Coursera whenever this may be happening. If you decide to run the assignment locally, please: 1. Try to download the necessary data files from your home directory one at a time, 2. Don't update anything other than this Jupyter notebook back to Coursera's servers, and 3. Make sure this notebook maintains its original name after you upload it back to Coursera.

Note: You need to submit the assignment to be graded, and passing the validation button's test does not grade the assignment. The validation button's functionality is exactly the same as running all cells.

[2]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from aml_utils import test_case_checker, perform_computation
```

Libraries such as `math` are neither as accurate nor as efficient as `numpy`.

Note: Do not import or use any other libraries other than what is already imported above.

1 *Assignment Summary

The UCI collection includes a set about authenticating banknotes donated by Volker Lohweg and Helene Doerksen at http://archive.ics.uci.edu/ml/datasets/banknote+authentication . This is a low-dimensional data set including only 4 features without any missing data points. To make things more interesting, you can adverserially augment the data with a large amount of noise features (say 5000 noise features), and see how SVM and different regularization schemes perform.

Write a program to train a support vector machine on this data using stochastic gradient descent. These are some constraints to consider:

- Do not use a package to train the classifier (that's the point), but your own code.
- Scale the variables so that each has unit variance.
- Search for an appropriate value of the regularization constant, trying at least the values [1e-3, 1e-2, 1e-1, 1]. Use the validation set for this search.

- Use at least 50 epochs of at least 300 steps each. In each epoch, you should separate out 50 training examples at random for evaluation (call this the set held out for the epoch).
- Compute the accuracy of the current classifier on the set held out for the epoch every 30 steps.

You should produce:

- A plot of the accuracy every 30 steps, for each value of the regularization constant.
- A plot of the magnitude of the coefficient vector every 30 steps, for each value of the regularization constant.
- Your estimate of the best value of the regularization constant, together with a brief description of why you believe that is a good value.
- Your estimate of the accuracy of the best classifier on the test dataset data

References:

- Textbook sections 2.1 - 2.2: https://link-springer-com.proxy2.library.illinois.edu/chapter/10.1007/978-3-030-18114-7_2

2 0. Normalizing the Features

3 Task 1

Write a function `normalize_feats` that takes the following arguments:

1. `train_features`: A numpy array with the shape (N_{train}, d), where d is the number of features and N_{train} is the number of training samples.
2. `some_features`: A numpy array with the shape (N_{some}, d), where d is the number of features and N_{some} is the number of samples to be normalized.
   - Do not assume anything about the values of $N_{\text{train}}, N_{\text{some}}, d$; they could be anything in the test cases.

and does the following:

1. Find the μ_{train}, which is the training set average of the features. μ_{train} Should have d elements.
2. Find the σ_{train}, which is the training set standard deviation of the features. σ_{train} Should have d elements.
3. For each row x in `some_features`, define the equivalent row $\hat{x}$ in `some_features_normalized` to be $\hat{x} = \frac{x-\mu_{\text{train}}}{\sigma_{\text{train}}}$ with the subtraction and division operation defined in an element-wise manner.

The function should return the numpy array `some_features_normalized` whose shape is (N_{some}, d).

[8]:
```python
def normalize_feats(train_features, some_features):
    """
    Normalizes the sample data features.

    Parameters
    ----------
    train_features: A numpy array with the shape (N_train, d), where d is the␣
↪number of features and N_train is the number of training samples.
    some_features: A numpy array with the shape (N_some, d), where d is the␣
↪number of features and N_some is the number of samples to be normalized.

    Returns
    -------
    some_features_normalized: A numpy array with shape (N_some, d).
    """

    # your code here
    mean = np.mean(train_features, 0)
    std = np.std(train_features, 0)
    some_features_normalized = (some_features - mean) / std

    return some_features_normalized
```

[9]:
```
# Performing sanity checks on your implementation
X_train = (np.arange(35).reshape(5,7) ** 13) % 20
X_some = np.arange(7).reshape(1,7) * 10
X_norm_some = normalize_feats(X_train, X_some)
assert np.array_equal(X_norm_some.round(3), np.array([[-1.265,  0.04 ,  2.298, ␣
↪3.24 ,  9.247,  6.798,  8.056]]))

# Checking against the pre-computed test database
test_results = test_case_checker(normalize_feats, task_id=1)
assert test_results['passed'], test_results['message']
```

[10]:
```
# Task 1 Test Cell
```

4 1. The Support Vector Machine

4.1 1.1 Implementing the Utility Functions for SVM

5 Task 2

Write a function `e_term` that takes the following arguments:

1. `x_batch`: A numpy array with the shape (N, d), where d is the number of features and N is the batch size.
2. `y_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size.
3. `a`: A numpy array with the shape $(d, 1)$, where d is the number of features. This is the weight vector.
4. `b`: A scalar.

- Do not assume anything about the values of N, d; they could be anything in the test cases.

and returns the numpy array `e_batch` whose shape is $(N, 1)$ and is defined as the following:

- If the k^{th} row of `x_batch`, `y_batch`, and `e_batch` were to be denoted as $x^{(k)}$, y_k and e_k, respectively, then we have $e_k = 1 - y_k(a \cdot x^{(k)} + b)$, where $a \cdot x^{(k)}$ is the dot product of the vectors a and $x^{(k)}$.

It may be a good thought exercise to implement this function without utilizing `for` loops. In fact, it's easier, more efficient, and possible in a single line.

[11]:
```
def e_term(x_batch, y_batch, a, b):
    """
    Computes the margin of the data points.

    Parameters
    ----------
    x_batch: A numpy array with the shape (N, d), where d is the number of␣
 ↪features and N is the batch size.
    y_batch: A numpy array with the shape (N, 1), where N is the batch size.
    a: A numpy array with the shape (d, 1), where d is the number of features.␣
 ↪This is the weight vector.
    b: A scalar.

    Returns
    -------
    e_batch: A numpy array with shape (N, 1).
    """
    shape = np.shape(y_batch)
    e_batch = np.zeros(shape)
    for k in range(len(y_batch)):
        e_batch[k] = 1 - y_batch[k]*(np.dot(x_batch[k],a)+b)

    return e_batch
```

[12]:
```
# Performing sanity checks on your implementation
x_batch_ = ((np.arange(35).reshape(5,7) ** 13) % 20) / 7.
y_batch_ = (2. * (np.arange(5)>2) - 1.).reshape(-1,1)
a_ = (np.arange(7)* 0.2).reshape(-1,1)
b_ = 0.1
e_batch_ = e_term(x_batch_, y_batch_, a_, b_)

assert np.array_equal(e_batch_.round(3), np.array([[ 5.986],[ 7.043],[ 7.
 ↪529],[-4.014],[-1.7  ]]))

# Checking against the pre-computed test database
test_results = test_case_checker(e_term, task_id=2)
assert test_results['passed'], test_results['message']
```

```
[13]: # Task 2 Test Cell
```

6 Task 3

Write a function `loss_terms_ridge` that computes the hinge and ridge regularization losses. The `loss_terms_ridge` functions should take the following arguments:

1. `e_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size. This is the output of the `e_term` function you wrote previously, and its $k^{(th)}$ element is $e_k = 1 - y_k(a \cdot x^{(k)} + b)$.
2. `a`: A numpy array with the shape $(d, 1)$, where d is the number of features. This is the weight vector.
3. `lam`: A scalar representing the regularization coefficient λ.
   - Do not assume anything about the values of N, d; they could be anything in the test cases.

and return the following two scalars:

1. `hinge_loss`: This hinge loss is defined as $l_{\text{hinge}} = \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i(a \cdot x^{(i)} + b))$. This can easily be written as a function of `e_batch`.
2. `ridge_loss`: This ridge regularization loss is defined as $l_{\text{ridge}} = \frac{\lambda}{2} \|a\|_2^2 = \frac{\lambda}{2} a^T a$.

You should produce both `hinge_loss` and `ridge_loss`. * Make sure that both of them are scalars and not multi-element arrays. * It may be a good thought exercise to implement this function without utilizing `for` loops. You only need a single line for each term.

```
[20]: def loss_terms_ridge(e_batch, a, lam):
          """
          Computes the hinge and ridge regularization losses.

          Parameters
          ----------
          e_batch: A numpy array with the shape (N, 1), where N is the batch size.␣
      ↪This is the output of the e_term function you wrote previously, and its kth␣
      ↪element is e_k = 1 - y_k(a*x_k+b).
          a: A numpy array with the shape (d, 1), where d is the number of features.␣
      ↪This is the weight vector.
          lam: A scalar representing the regularization coefficient  .

          Returns
          -------
          hinge_loss: The hinge regularization loss defined in the above cell.
          ridge_loss: The ridge regularization loss defined in the above cell.
          """
          hinge_loss = np.mean(np.maximum(0, e_batch))
          ridge_loss = (np.divide(lam, 2) * np.transpose(a).dot(a)).item()
          return np.array((hinge_loss, ridge_loss))
          return np.array((hinge_loss, ridge_loss))
```

[21]:
```
# Performing sanity checks on your implementation
e_batch_ = ((np.arange(35).reshape(-1,1) ** 13) % 20) / 7.
a_ = (np.arange(7)* 0.2).reshape(-1,1)
lam_ = 10.

hinge_loss_1, reg_loss_1 = tuple(loss_terms_ridge(e_batch_, a_, lam_))
assert np.round(hinge_loss_1,3) == 1.114 and np.round(reg_loss_1,3) == 18.2

hinge_loss_2, reg_loss_2 = tuple(loss_terms_ridge(e_batch_-1., a_, lam_))
assert np.round(hinge_loss_2,3) == 0.412 and np.round(reg_loss_2,3) == 18.2

# Checking against the pre-computed test database
test_results = test_case_checker(loss_terms_ridge, task_id=3)
assert test_results['passed'], test_results['message']
```

[22]:
```
# Task 3 Test Cell
```

7 Task 4

Write a function `a_gradient_ridge` that computes the ridge-regularized loss gradient with respect to the weights vector and takes the following arguments:

1. `x_batch`: A numpy array with the shape (N, d), where d is the number of features and N is the batch size.
2. `y_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size.
3. `e_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size. This is the output of the `e_term` function you wrote previously, and its $k^{(th)}$ element is $e_k = 1 - y_k(a \cdot x^{(k)} + b)$.
4. `a`: A numpy array with the shape $(d, 1)$, where d is the number of features. This is the weight vector.
5. `lam`: A scalar representing the regularization coefficient λ.
   - Do not assume anything about the values of N, d; they could be anything in the test cases.

`a_gradient_ridge` should return the numpy array `grad_a` whose shape is $(d, 1)$ and is defined as $\nabla_a l := \lambda a + \frac{1}{N}\sum_{i=1}^{N} \nabla_a \max(0, 1 - y_i(a \cdot x^{(i)} + b))$. You may need to revisit the textbook for information on how to compute the hinge loss gradient.

- **Important Note**: To be consistent, be careful about the inequality operators and make sure you are following the textbook; $\geq$ is different from $>$.

It may be a good thought exercise to implement this function without utilizing `for` loops. In fact, it's easier, more efficient, and possible in a single line.

[23]:
```
def a_gradient_ridge(x_batch, y_batch, e_batch, a, lam):
    """
    Computes the ridge_regularized loss gradient w.r.t the weights vector.
    Parameters
    ----------
    x_batch: A numpy array with the shape (N, d), where d is the number of␣
↪features and N is the batch size.
    y_batch: A numpy array with the shape (N, 1), where N is the batch size.
    e_batch: A numpy array with the shape (N, 1), where N is the batch size.␣
↪This is the output of the e_term function you wrote previously, and its kth␣
↪element is e_k = 1 - y_k(a*x_k+b).
    a: A numpy array with the shape (d, 1), where d is the number of features.␣
↪This is the weight vector.
    lam: A scalar representing the regularization coefficient  .

    Returns
    -------
    grad_a: A numpy array with shape (d, 1) and defined as the gradient of the␣
↪ridge regularized loss function.
    """

    # your code here
    return lam * a + (np.mean(np.where(e_batch < 0, 0, -y_batch*x_batch),␣
↪axis=0)).reshape(-1,1)
```

[24]:
```
# Performing sanity checks on your implementation
x_batch_ = ((np.arange(35).reshape(5,7) ** 13) % 20) / 7.
y_batch_ = (2. * (np.arange(5)>2) - 1.).reshape(-1,1)
a_ = (np.arange(7)* 0.2).reshape(-1,1)
b_ = 0.1
lam_ = 10.
e_batch_ = e_term(x_batch_, y_batch_, a_, b_)

grad_a_ = a_gradient_ridge(x_batch_, y_batch_, e_batch_, a_, lam_)

assert np.array_equal(grad_a_.round(3), np.array([[ 0.314],[ 2.686],[ 5.057],[␣
↪6.571],[ 8.657],[11.029],[12.829]]))

# Checking against the pre-computed test database
test_results = test_case_checker(a_gradient_ridge, task_id=4)
assert test_results['passed'], test_results['message']
```

[25]:
```
# Task 4 Test Cell
```

8 Task 5

Write a function `b_derivative` that computes the loss gradient with respect to the bias parameter and takes the following arguments:

1. `y_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size.
2. `e_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size. This is the output of the `e_term` function you wrote previously, and its $k^{(th)}$ element is $e_k = 1 - y_k(a \cdot x^{(k)} + b)$.
    - Do not assume anything about the values of N, d; they could be anything in the test cases.

and returns the numpy array `der_b` which is a scalar just like `b` (Do not return a numpy array). `der_b` is defined as $\frac{\partial}{\partial b} l := \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial b} \max(0, 1 - y_i(a \cdot x^{(i)} + b))$. You may need to revisit the textbook for information on how to compute the hinge loss derivative.

- **Important Note**: To be consistent, be careful about the inequality operators and make sure you are following the textbook; $\geq$ is different from $>$.

It may be a good thought exercise to implement this function without utilizing `for` loops. In fact, it's easier, more efficient, and possible in a single line.

[26]:
```
def b_derivative(y_batch, e_batch):
    """
    Computes the loss gradient with respect to the bias parameter b.

    Parameters
    ----------
    y_batch: A numpy array with the shape (N, 1), where N is the batch size.
    e_batch: A numpy array with the shape (N, 1), where N is the batch size.␣
↪This is the output of the e_term function you wrote previously, and its kth␣
↪element is e_k = 1 - y_k(a*x_k+b).

    Returns
    -------
    der_b: A scalar defined as the gradient of the hinge loss w.r.t the bias␣
↪parameter b.
    """

    # your code here
    return np.mean(np.where(e_batch < 0, 0, -y_batch), axis=0).item()
```

[27]:
```
# Performing sanity checks on your implementation
x_batch_ = ((np.arange(35).reshape(5,7) ** 13) % 20) / 7.
y_batch_ = (2. * (np.arange(5)>2) - 1.).reshape(-1,1)
a_ = (np.arange(7)* 0.2).reshape(-1,1)
b_ = -5.
e_batch_ = e_term(x_batch_, y_batch_, a_, b_)

grad_b_ = b_derivative(y_batch_, e_batch_)

assert np.round(grad_b_, 3) == 0.2

# Checking against the pre-computed test database
```

```
test_results = test_case_checker(b_derivative, task_id=5)
assert test_results['passed'], test_results['message']
```

[28]:
```
# Task 5 Test Cell
```

8.1 1.1 Lasso Regularized SVM

In the textbook, you learned about SVM model with Ridge (a.k.a. L2-norm) regularization and how to use Stochastic Gradient Descent to compute the gradient for the following model

$$l_{ridge} = \lambda \sum_{j=1}^{d} \frac{1}{2} a_j^2 + \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i(a \cdot x^{(i)} + b))$$

One can modify the regularization function of this model to obtain the Lasso (a.k.a. L1-norm) regularized SVM:

$$l_{lasso} = \lambda \sum_{j=1}^{d} |a_j| + \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i(a \cdot x^{(i)} + b))$$

It's fair to say that minimizing Ridge regularization will prioritize minimizing larger weights and puts less emphasis on smaller weights. In other words, **Ridge** regularized models will **supress larger weights more** and care less about smaller weights **than Lasso** regularized models.

You will learn about the differences between Ridge and Lasso regularization later in the course. It's okay if you don't know their differences. The assignment and task descriptions will walk you through what you need for implementation.

9 Task 6

Similar to the `loss_terms_ridge` you previously wrote, write a function `loss_terms_lasso` that computes the hinge and lasso regularization losses. The `loss_terms_lasso` functions should take the following arguments:

1. `e_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size. This is the output of the `e_term` function you wrote previously, and its $k^{(th)}$ element is $e_k = 1 - y_k(a \cdot x^{(k)} + b)$.
2. `a`: A numpy array with the shape $(d, 1)$, where d is the number of features. This is the weight vector.
3. `lam`: A scalar representing the regularization coefficient λ.
   - Do not assume anything about the values of N, d; they could be anything in the test cases.

and return the following two scalars:

1. `hinge_loss`: This hinge loss is defined as $l_{\text{hinge}} = \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i(a \cdot x^{(i)} + b))$. This can easily be written as a function of `e_batch`.
2. `lasso_loss`: This lasso regularization loss is defined as $l_{\text{lasso}} = \lambda \|a\|_1^1 = \lambda \sum_{j=1}^{d} |a_j|$.

You should produce both `hinge_loss` and `lasso_loss`. * Make sure that both of them are scalars and not multi-element arrays. * It may be a good thought exercise to implement this function without utilizing `for` loops. You only need a single line for each term.

```
[67]: def loss_terms_lasso(e_batch, a, lam):
          """
          Computes the hinge and lasso regularization losses.

          Parameters
          ----------
          e_batch: A numpy array with the shape (N, 1), where N is the batch size.␣
       ↪This is the output of the e_term function you wrote previously, and its kth␣
       ↪element is e_k = 1 - y_k(a*x_k+b).
          a: A numpy array with the shape (d, 1), where d is the number of features.␣
       ↪This is the weight vector.
          lam: A scalar representing the regularization coefficient  .

          Returns
          -------
          hinge_loss: The hinge loss scalar as defined in the cell above.
          lasso_loss: The lasso loss scalar as defined in the cell above.
          """

          # your code here
          return np.array((np.mean(np.maximum(0, e_batch)), lam * np.sum(np.abs(a))))
```

```
[68]: # Performing sanity checks on your implementation
      e_batch_ = ((np.arange(35).reshape(-1,1) ** 13) % 20) / 7.
      a_ = (np.arange(7)* 0.2).reshape(-1,1)
      lam_ = 10.

      hinge_loss_1, reg_loss_1 = tuple(loss_terms_lasso(e_batch_, a_, lam_))
      assert np.round(hinge_loss_1,3) == 1.114 and np.round(reg_loss_1,3) == 42.0, np.
       ↪round(reg_loss_1,3)

      hinge_loss_2, reg_loss_2 = tuple(loss_terms_lasso(e_batch_-1., a_, lam_))
      assert np.round(hinge_loss_2,3) == 0.412 and np.round(reg_loss_2,3) == 42.0, np.
       ↪round(reg_loss_2,3)

      # Checking against the pre-computed test database
      test_results = test_case_checker(loss_terms_lasso, task_id=6)
      assert test_results['passed'], test_results['message']
```

```
[69]: # Task 6 Test Cell
```

10 Task 7

Similar to the `a_gradient_ridge` function you previously wrote, write a function `a_gradient_lasso` that computes the lasso-regularized loss sub-gradient with respect to the weights vector and takes the following arguments:

1. `x_batch`: A numpy array with the shape (N, d), where d is the number of features and N is the batch size.
2. `y_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size.
3. `e_batch`: A numpy array with the shape $(N, 1)$, where N is the batch size. This is the output of the `e_term` function you wrote previously, and its $k^{(th)}$ element is $e_k = 1 - y_k(a \cdot x^{(k)} + b)$.
4. `a`: A numpy array with the shape $(d, 1)$, where d is the number of features. This is the weight vector.
5. `lam`: A scalar representing the regularization coefficient λ.
    - Do not assume anything about the values of N, d; they could be anything in the test cases.

`a_gradient_lasso` should return the numpy array `grad_a` whose shape is $(d, 1)$ and is defined as $\nabla_a l := \lambda \nabla_a \|a\|_1^1 + \frac{1}{N}\sum_{i=1}^{N} \nabla_a \max(0, 1 - y_i(a \cdot x^{(i)} + b))$.

- It's fairly easy to compute the lasso sub-gradient:

$$\frac{\partial}{\partial a_k}\|a\|_1^1 = \frac{\partial}{\partial a_k}\sum_{j=1}^{d}|a_j| = \frac{\partial}{\partial a_k}|a_k| = \text{sign}(a_k)$$

, where the scalar sign function is defined as

$$\text{sign}(a_k) := \begin{cases} 1 & a_k > 0 \\ -1 & a_k < 0 \\ 0 & a_k = 0 \end{cases}$$

Now, remember that the gradients are essentially partial derivative vectors for each dimension. Therefore, you can define the sub-gradient as

$$\nabla_a \|a\|_1^1 == \begin{bmatrix} \frac{\partial}{\partial a_1}\|a\|_1^1 \\ \frac{\partial}{\partial a_2}\|a\|_1^1 \\ \cdots \\ \frac{\partial}{\partial a_d}\|a\|_1^1 \end{bmatrix} = \begin{bmatrix} \text{sign}(a_1) \\ \text{sign}(a_2) \\ \cdots \\ \text{sign}(a_d) \end{bmatrix} = \text{sign}(a)$$

It's worth mentioning that the absolute value function's derivative at zero is undefined, and this is why our calculation isn't exactly reporting the gradient (instead we call it a sub-gradient to make this subtle distinction). We actually made an assumption of convenience that at $a_k = 0$ the lasso-derivative is zero.

- **Important Note**: To be consistent, be careful about the inequality operators and make sure you are following the textbook; $\geq$ is different from $>$.

It may be a good thought exercise to implement this function without utilizing `for` loops. In fact, it's easier, more efficient, and possible in a single line.

[70]:
```
def a_gradient_lasso(x_batch, y_batch, e_batch, a, lam):
    """
    Computes the lasso-regularized loss sub-gradient w.r.t the weights vector

    Parameters
    ----------
    x_batch: A numpy array with the shape (N, d), where d is the number of␣
↪features and N is the batch size.
    y_batch: A numpy array with the shape (N, 1), where N is the batch size.
    e_batch: A numpy array with the shape (N, 1), where N is the batch size.␣
↪This is the output of the e_term function you wrote previously, and its kth␣
↪element is e_k = 1 - y_k(a*x_k+b).
    a: A numpy array with the shape (d, 1), where d is the number of features.␣
↪This is the weight vector.
    lam: A scalar representing the regularization coefficient  .

    Returns
    -------
    grad_a: A numpy array with shape (d, 1) and defined as the gradient of the␣
↪lasso-regularized loss function w.r.t the weights vector.
    """

    # your code here
    temp_sum = np.sum(np.where(e_batch<0, 0, -y_batch*x_batch), 0).
↪reshape(-1,1) / int(e_batch.shape[0])
    grad_a = lam * np.where(a>0, 1,(np.where(a==0, 0, -1))) + temp_sum

    return grad_a
```

[71]:
```
# Performing sanity checks on your implementation
x_batch_ = ((np.arange(35).reshape(5,7) ** 13) % 20) / 7.
y_batch_ = (2. * (np.arange(5)>2) - 1.).reshape(-1,1)
a_ = (np.arange(7)* 0.2).reshape(-1,1)
b_ = 0.1
lam_ = 10.
e_batch_ = e_term(x_batch_, y_batch_, a_, b_)

grad_a_lasso_ = a_gradient_lasso(x_batch_, y_batch_, e_batch_, a_, lam_)

assert np.array_equal(grad_a_lasso_.round(3), np.array([[ 0.314], [10.686], [11.
↪057],[10.571], [10.657], [11.029], [10.829]]))
```

```
# Checking against the pre-computed test database
test_results = test_case_checker(a_gradient_lasso, task_id=7)
assert test_results['passed'], test_results['message']
```

[72]:
```
# Task 7 Test Cell
```

10.1 1.3 Training the Support Vector Machine

[73]:
```
def get_acc(a, b, feats_nomalized, labels):
    pred = (feats_nomalized @ a + b) >= 0.
    pred = 2 * pred - 1
    acc = (pred.reshape(-1) == labels.reshape(-1)).mean()
    return acc

def svm_trainer(train_features, train_labels, val_features, val_labels,␣
 ↪heldout_size=50,
                batch_size=1, num_epochs=100, num_steps=300, eval_interval = 30,
                lambda_list=[1e-3, 1e-2, 1e-1, 2e-1], eta_tuner=lambda epoch: 1.
 ↪/(0.01 * epoch + 500.),
                regularization = 'ridge'):

    train_features_normalized = normalize_feats(train_features, train_features)
    val_features_normalized = normalize_feats(train_features, val_features)

    np_random = np.random.RandomState(12345)
    if regularization == 'ridge':
        a_gradient = a_gradient_ridge
        loss_terms = loss_terms_ridge
    elif regularization == 'lasso':
        a_gradient = a_gradient_lasso
        loss_terms = loss_terms_lasso
    else:
        raise Exception(f'Unknown regularization {regularization}')

    train_progress = np.arange(0., num_epochs, eval_interval/num_steps)
    heldout_accs = np.zeros((len(lambda_list), train_progress.size))
    weight_magnitudes = np.zeros((len(lambda_list), train_progress.size))
    hinge_losses = np.zeros((len(lambda_list), train_progress.size))
    reg_losses = np.zeros((len(lambda_list), train_progress.size))
    val_accs = np.zeros(len(lambda_list))

    all_a = np.zeros((len(lambda_list), train_features_normalized.shape[1]))
    all_b = np.zeros(len(lambda_list))

    for lam_idx, lam in enumerate(lambda_list):
        a = np.zeros((train_features_normalized.shape[1], 1))
        b = 0.

        eval_idx = 0
        for epoch in range(num_epochs):
            eta = eta_tuner(epoch)

            # Picking the heldout indices
            # We will avoid the use of np_random.choice due to performance␣
↪reasons
            heldout_size = min(heldout_size, train_features_normalized.shape[0]/
↪/2)

            heldout_indicator = np.arange(train_features_normalized.shape[0]) <␣
↪heldout_size
            np_random.shuffle(heldout_indicator)

            heldout_feats = train_features_normalized[heldout_indicator,:]
            heldout_labels = train_labels[heldout_indicator]

            non_heldout_feats =␣
↪train_features_normalized[heldout_indicator==False,:]
            non_heldout_labels = train_labels[heldout_indicator==False]

            batch_size = min(batch_size, non_heldout_feats.shape[0])

            for step in range(num_steps):
                rand_unifs = np_random.uniform(0, 1, size=non_heldout_feats.
↪shape[0])
                batch_thresh = np.percentile(rand_unifs, 100. * batch_size /␣
↪non_heldout_feats.shape[0])
                batch_indices = (rand_unifs<=batch_thresh)
                x_batch = non_heldout_feats[batch_indices,:]
                y_batch = non_heldout_labels[batch_indices].reshape(-1,1)

                e_batch = e_term(x_batch, y_batch, a, b)
                hinge_loss, reg_loss = loss_terms(e_batch, a, lam)
                grad_a = a_gradient(x_batch, y_batch, e_batch, a, lam)
                grad_b = b_derivative(y_batch, e_batch)

                if step % eval_interval == 0:
                    heldout_acc = get_acc(a, b, heldout_feats, heldout_labels)
                    heldout_accs[lam_idx, eval_idx] = heldout_acc
                    if regularization == 'ridge':
                        weight_magnitudes[lam_idx, eval_idx] = np.sum(a**2)
                    elif regularization == 'lasso':
                        weight_magnitudes[lam_idx, eval_idx] = np.sum(np.abs(a))
                    hinge_losses[lam_idx, eval_idx] = hinge_loss
                    reg_losses[lam_idx, eval_idx] = reg_loss
                    eval_idx += 1
                    if step % (5 * eval_interval) == 0:
                        print('.', end='')

                a = a - eta * grad_a
                b = b - eta * grad_b

        val_acc = get_acc(a, b, val_features_normalized, val_labels)
        val_accs[lam_idx] = val_acc
        all_a[lam_idx, :] = a.reshape(-1)
        all_b[lam_idx] = b
        print((f'\nlambda={lam} yielded a validation accuracy of %.3f '%(100. *␣
 ↪val_acc)) + '%')

    return_dict = dict(all_a=all_a, all_b=all_b, train_progress=train_progress,␣
 ↪regularization=regularization,
                       train_features=train_features, train_labels=train_labels,
                       val_features=val_features, val_labels=val_labels,
                       lambda_list=lambda_list, val_accs=val_accs,␣
 ↪hinge_losses=hinge_losses,
                       weight_magnitudes=weight_magnitudes,␣
 ↪heldout_accs=heldout_accs)
    return return_dict

def get_test_accuracy(test_features, test_labels, training_info):
    train_features = training_info['train_features']
    val_accs = training_info['val_accs']
    all_a = training_info['all_a']
    all_b = training_info['all_b']
    lambda_list = training_info['lambda_list']
    test_features_normalized = normalize_feats(train_features, test_features)
    best_lam_idx = np.argmax(val_accs)
    best_a = all_a[best_lam_idx, :].reshape(-1,1)
    best_b = all_b[best_lam_idx]
    test_acc = get_acc(best_a, best_b, test_features_normalized, test_labels)
    print(f'Best lambda was chosen to be {lambda_list[best_lam_idx]}')
    print((f'The resulting test accuracy was %.3f '%(100. * test_acc)) + '%')

def print_weights(training_info, rounding=20):
    import warnings
    warnings.simplefilter(action='ignore', category=FutureWarning)
    pd.set_option("display.precision", rounding)
    print('Here are the learned weights for each regularization coefficient:')
    print('  * Each row represents a single regularization coefficient.')
    print('  * The last two columns represent the weight vector magnitudes.')
    print('  * Each of the other columns represent a feature weight. ')
    all_a = training_info['all_a']
    lambda_list = training_info['lambda_list']
    all_a_rounded = all_a.round(rounding)
    w_df = pd.DataFrame(all_a_rounded, columns=['$a_{%d}$'%(col+1) for col in␣
 ↪range(all_a.shape[1])])

    if all_a.shape[1] > 7:
        w_df = w_df.drop(w_df.columns[8:(all_a.shape[1]-3)], axis=1)
        w_df.insert(8, '$\cdots$', ['$\cdots$' for _ in lambda_list])

    w_df.insert(0, '$\lambda$', lambda_list)
    w_df['$\|a\|_2^2$'] = np.sum(all_a**2, axis=1).round(rounding)
    w_df['$\|a\|_1^1$'] = np.sum(np.abs(all_a), axis=1).round(rounding)
    max_l_norm = w_df[['$\|a\|_2^2$', '$\|a\|_1^1$']].values.max(axis=1)
    w_df.reset_index(drop=True, inplace=True)
    w_df = w_df.set_index('$\lambda$')
    w_df_style = w_df.style.apply(lambda x: ['font-weight: bold' if v in␣
 ↪max_l_norm else '' for v in x])
    return w_df_style
```

10.2 1.4 Training Plots

[74]:
```
def ema(vec, alpha=0.99):
    # Exponential Moving Average Filter
    # This filter is useful for smoothing noisy training statistics such as the␣
 ↪(stochastic) loss.
    # alpha is the smoothing factor; larger smoothing factors can remove more␣
 ↪noise,
    # but will induce more delay when following the original signal.
    out = [vec[0]]
    last_val = vec[0]
    for val in vec[1:]:
        last_val = val*(1-alpha) + alpha*last_val
        out.append(last_val)
    return np.array(out)

def generate_plots(training_info, heldout_acc_smoothing=0.99, loss_smoothing=0.
 ↪99, weight_smoothing=0.99):
    assert 0 <= heldout_acc_smoothing < 1
    assert 0 <= loss_smoothing < 1
    assert 0 <= weight_smoothing < 1
    all_a = training_info['all_a']
    all_b = training_info['all_b']
    train_progress = training_info['train_progress']
    lambda_list = training_info['lambda_list']
    val_accs = training_info['val_accs']
    hinge_losses = training_info['hinge_losses']
    weight_magnitudes = training_info['weight_magnitudes']
    heldout_accs = training_info['heldout_accs']
    regularization = training_info['regularization']

    fig, axes = plt.subplots(2, 2, figsize=(12,10), dpi=90)
    ax = axes[0,0]
    for lam_idx, lam in enumerate(lambda_list):
        ax.plot(train_progress, ema(weight_magnitudes[lam_idx,:],␣
↪weight_smoothing), label=f'lambda={lam}')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Weight Magnitude')
    if weight_smoothing:
        ax.set_title('(Moving Average of) Weight Magnitudes During Training')
    else:
        ax.set_title('Weight Magnitudes During Training')

    ax.legend()

    ax = axes[0,1]
    for lam_idx, lam in enumerate(lambda_list):
        ax.plot(train_progress, ema(heldout_accs[lam_idx,:],␣
↪heldout_acc_smoothing), label=f'lambda={lam}')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Held-out Accuracy')
    if heldout_acc_smoothing:
        ax.set_title('(Moving Average of) Held-out Accuracy During Training')
    else:
        ax.set_title('Held-out Accuracy During Training')
    _ = ax.legend()

    ax = axes[1,0]
    ax.plot(lambda_list, val_accs)
    ax.set_xscale('log')
    ax.set_xlabel('Regularization coefficient')
    ax.set_ylabel('Validation Accuracy')
    _ = ax.set_title('Validation Accuracy')

    ax = axes[1,1]
    for lam_idx, lam in enumerate(lambda_list):
        ax.plot(train_progress, ema(hinge_losses[lam_idx,:], loss_smoothing),␣
↪label=f'lambda={lam}')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')
    if loss_smoothing:
        ax.set_title('(Moving Average of) Hinge Loss During Training')
    else:
        ax.set_title('Hinge Loss During Training')
    _ = ax.legend()
```

11 2. Bank Note Authentication Problem

11.1 2.1 Description

The UC Irvine Machine Learning Data Repository hosts a collection on banknote authentication, donated by Volker Lohweg and Helene Doerksen at http://archive.ics.uci.edu/ml/datasets/banknote+authentication .

11.2 2.2 Information Summary

- **Input/Output**: This data has a set of 4 attributes; the variance, the skewness, and the curtosis of Wavelet Transformed image along with the entropy of the image. The last column indicates whether the image was taken from a genuine or a forged banknote, and acts as a binary label.
- **Missing Data**: There are no missing data points.
- **Final Goal**: We want to build an SVM classifier that can predict whether a note is authentic or not.

11.3 2.3 Loading

[75]:
```
df_raw = pd.read_csv('../SGDSVM-lib/data/data_banknote_authentication.txt',
                     names=['variance', 'skewness', 'curtosis', 'entropy',␣
↪'class'],
                     header=None)
```

[76]:
```
df_raw.head()
```

[76]:
```
   variance  skewness  curtosis  entropy  class
0    3.6216    8.6661   -2.8073  -0.4470      0
1    4.5459    8.1674   -2.4586  -1.4621      0
2    3.8660   -2.6383    1.9242   0.1065      0
3    3.4566    9.5228   -4.0112  -3.5944      0
4    0.3292   -4.4552    4.5718  -0.9888      0
```

[77]:
```
df_raw.describe()
```

[77]:
```
        variance   skewness   curtosis    entropy      class
count  1372.0000  1372.0000  1372.0000  1372.0000  1372.0000
mean      0.4337     1.9224     1.3976    -1.1917     0.4446
std       2.8428     5.8690     4.3100     2.1010     0.4971
min      -7.0421   -13.7731    -5.2861    -8.5482     0.0000
```

```
25%      -1.7730    -1.7082    -1.5750    -2.4135     0.0000
50%       0.4962     2.3197     0.6166    -0.5867     0.0000
75%       2.8215     6.8146     3.1792     0.3948     1.0000
max       6.8248    12.9516    17.9274     2.4495     1.0000
```

11.4 2.4 Pre-processing

[78]:
```
df = df_raw.copy(deep=True)
df = df.dropna()

label_col_name = 'class'
feature_col_names = [col for col in df_raw.columns.tolist() if col!
 ↪=label_col_name]

class_names = df[label_col_name].unique().tolist()
assert len(class_names) == 2
df[label_col_name] = 2. * np.array([class_names.index(x) for x in␣
 ↪df[label_col_name]]) - 1.

df.head()
```

[78]:
```
   variance  skewness  curtosis  entropy  class
0    3.6216    8.6661   -2.8073  -0.4470   -1.0
1    4.5459    8.1674   -2.4586  -1.4621   -1.0
2    3.8660   -2.6383    1.9242   0.1065   -1.0
3    3.4566    9.5228   -4.0112  -3.5944   -1.0
4    0.3292   -4.4552    4.5718  -0.9888   -1.0
```

11.5 2.5 Splitting the data

[79]:
```
np_random = np.random.RandomState(12345)
# Splitting the data
df_shuffled = df.sample(frac=1, random_state=np_random).reset_index(drop=True)
all_features = df_shuffled.loc[:, df_shuffled.columns != label_col_name].values
all_labels = df_shuffled[label_col_name].values

valid_cols = []
for col_idx in range(all_features.shape[1]):
    if np.unique(all_features[:,col_idx].reshape(-1)).size > 5:
        valid_cols.append(col_idx)
all_features = all_features[:, valid_cols]
```

[80]:
```
def train_val_test_split(all_features, all_labels, train_frac=0.4, val_frac=0.
 ↪3):
    assert train_frac + val_frac <= 1
    assert train_frac > 0
    assert val_frac > 0
    train_cnt = int(train_frac * all_features.shape[0])
    val_cnt = int(val_frac * all_features.shape[0])
    train_features, train_labels = all_features[:train_cnt, :], all_labels[:
 ↪train_cnt]
    val_features, val_labels = all_features[train_cnt:(train_cnt+val_cnt), :],␣
 ↪all_labels[train_cnt:(train_cnt+val_cnt)]
    test_features, test_labels = all_features[(train_cnt+val_cnt):, :],␣
 ↪all_labels[(train_cnt+val_cnt):]
    return train_features, train_labels, val_features, val_labels,␣
 ↪test_features, test_labels
```

[81]:
```
splitted_data = train_val_test_split(all_features, all_labels, train_frac=0.4,␣
 ↪val_frac=0.3)
train_features, train_labels, val_features, val_labels, test_features,␣
 ↪test_labels = splitted_data

train_features.shape, train_labels.shape, val_features.shape, val_labels.shape,␣
 ↪test_features.shape, test_labels.shape
```

[81]:
```
((548, 4), (548,), (411, 4), (411,), (413, 4), (413,))
```

[82]:
```
print(f'Negative Samples = {np.sum(all_labels==-1)} --> %.2f' %(100 * np.
 ↪mean(all_labels==-1)) + "% of total samples")
print(f'Positive Samples = {np.sum(all_labels==1)} --> %.2f' %(100 * np.
 ↪mean(all_labels==1)) + '% of total samples')
```

```
Negative Samples = 762 --> 55.54% of total samples
Positive Samples = 610 --> 44.46% of total samples
```

11.6 2.6 Training and Testing on Plain Data

11.6.1 2.6.1 Ridge Regularization

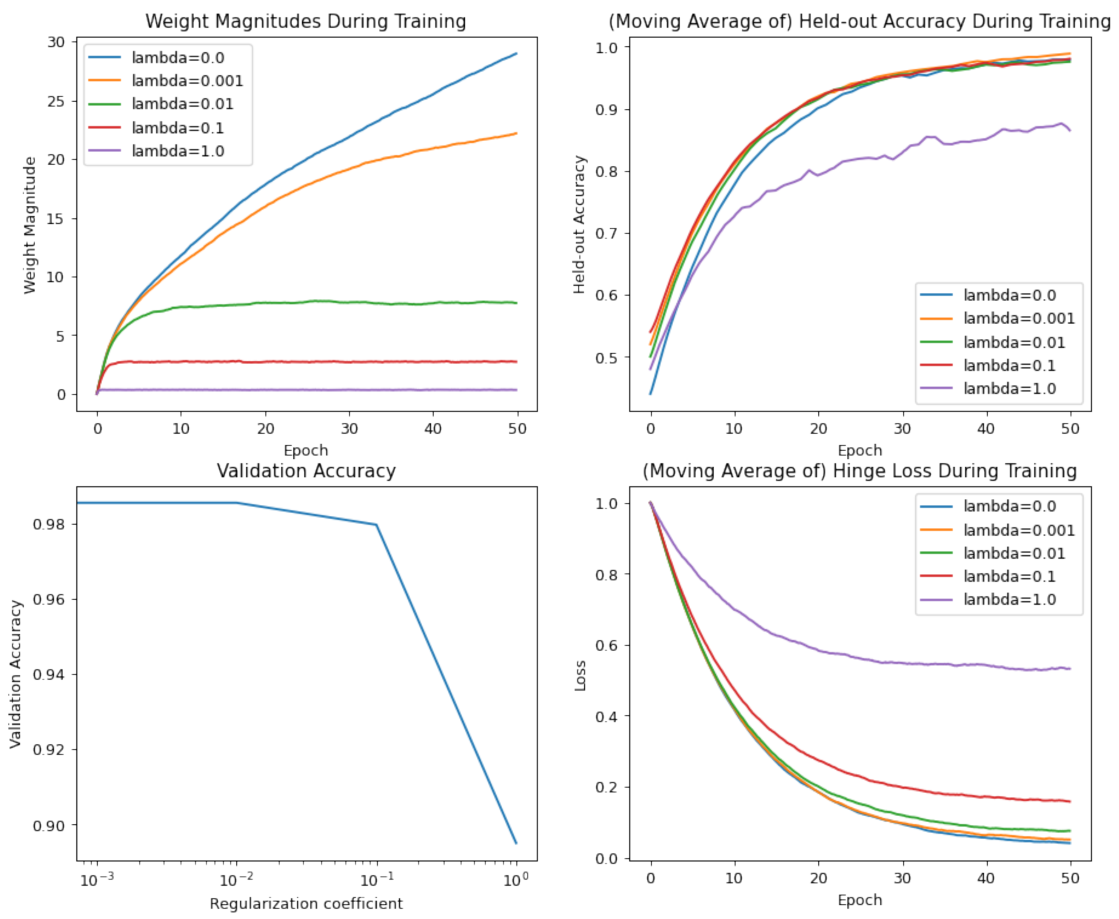
[83]:
```
if perform_computation:
    splitted_data = train_val_test_split(all_features, all_labels, train_frac=0.
 ↪5, val_frac=0.25)

    # The "_pr" variable postfix is short for "plain ridge".
    train_features_pr, train_labels_pr, val_features_pr, val_labels_pr,␣
 ↪test_features_pr, test_labels_pr = splitted_data

    training_info_plain_ridge = svm_trainer(train_features_pr, train_labels_pr,␣
 ↪val_features_pr, val_labels_pr,
                                            heldout_size=50, batch_size=32,␣
 ↪num_epochs=50, num_steps=300,
                                                eval_interval = 30, eta_tuner =␣
↪lambda epoch: 1./(0.01 * epoch + 50.),
                                                lambda_list=[0., 1e-3, 1e-2, 1e-1,␣
↪1e0],
                                                regularization='ridge')
```

```
…
…
lambda=0.0 yielded a validation accuracy of 98.542 %
…
…
lambda=0.001 yielded a validation accuracy of 98.542 %
…
…
lambda=0.01 yielded a validation accuracy of 98.542 %
…
…
lambda=0.1 yielded a validation accuracy of 97.959 %
…
…
lambda=1.0 yielded a validation accuracy of 89.504 %
```

[84]:
```
if perform_computation:
    # The noise in the plots were smoothed-out by an exponential moving average␣
↪filter.
    # If you'd rather see the original plots, you can see the disable noise␣
↪removal smoothing filters
    # for each subplot by setting the corresponding smoothing factor to zero.
    generate_plots(training_info_plain_ridge, heldout_acc_smoothing=0.99,␣
↪loss_smoothing=0.99, weight_smoothing=0.)
```



```
[85]: w_df = None
      if perform_computation:
          w_df = print_weights(training_info_plain_ridge, rounding=4)
      w_df
```

```
Here are the learned weights for each regularization coefficient:
  * Each row represents a single regularization coefficient.
  * The last two columns represent the weight vector magnitudes.
  * Each of the other columns represent a feature weight.
```

```
[85]: <pandas.io.formats.style.Styler at 0x7f0adc1a0ee0>
```

```
[86]: if perform_computation:
          get_test_accuracy(test_features_pr, test_labels_pr,␣
       ↪training_info_plain_ridge)
```

```
Best lambda was chosen to be 0.0
The resulting test accuracy was 98.251 %
```

The weight magnitude plot in this problem is showing $||a||_2^2 = \sum_{j=1}^{d} a_j^2$ since we're using Ridge regularization. Here are some questions you may want to think about as food for thought.

1. Does regularization help at all?
2. If regularization did not help, why do you think the reason was?

- **Hint**: How many features do you have in this problem?

11.6.2 2.6.2 Lasso Regularization

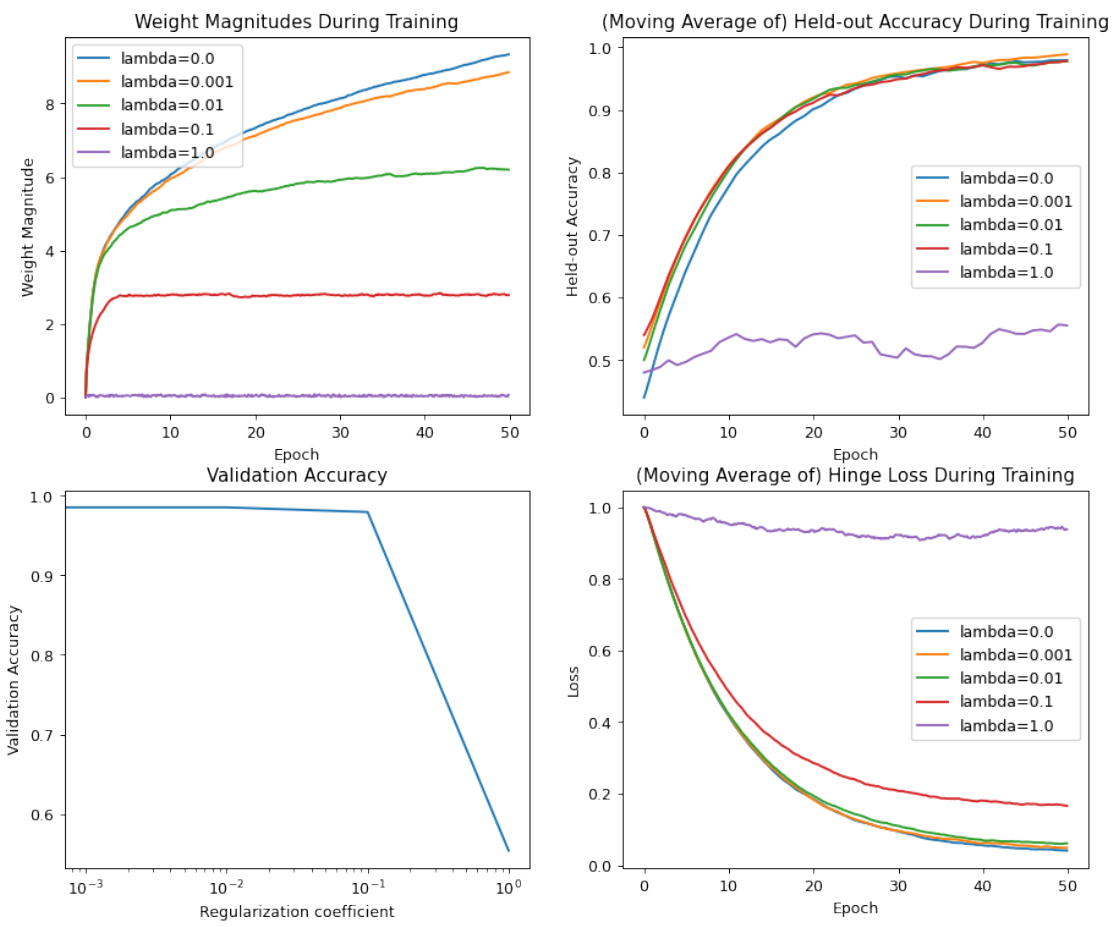
[87]:
```
if perform_computation:
    splitted_data = train_val_test_split(all_features, all_labels, train_frac=0.
 ↪5, val_frac=0.25)

    # The "_pl" variable postfix is short for "plain lasso".
    train_features_pl, train_labels_pl, val_features_pl, val_labels_pl,␣
 ↪test_features_pl, test_labels_pl = splitted_data

    training_info_plain_lasso = svm_trainer(train_features_pl, train_labels_pl,␣
 ↪val_features_pl, val_labels_pl,
                                            heldout_size=50,batch_size=32,␣
 ↪num_epochs=50, num_steps=300,
                                            eval_interval = 30, eta_tuner =␣
 ↪lambda epoch: 1./(0.01 * epoch + 50.),
                                            lambda_list=[0., 1e-3, 1e-2, 1e-1,␣
 ↪1e0],
                                            regularization='lasso')
```

```
…
…
lambda=0.0 yielded a validation accuracy of 98.542 %
…
…
lambda=0.001 yielded a validation accuracy of 98.542 %
…
…
lambda=0.01 yielded a validation accuracy of 98.542 %
…
…
lambda=0.1 yielded a validation accuracy of 97.959 %
…
…
lambda=1.0 yielded a validation accuracy of 55.394 %
```

[88]:
```
if perform_computation:
    generate_plots(training_info_plain_lasso, heldout_acc_smoothing=0.99,␣
 ↪loss_smoothing=0.99, weight_smoothing=0.)
```



```
[89]: if perform_computation:
          get_test_accuracy(test_features_pl, test_labels_pl,␣
       ↪training_info_plain_lasso)
```

```
Best lambda was chosen to be 0.0
The resulting test accuracy was 98.251 %
```

```
[90]: w_df = None
      if perform_computation:
          w_df = print_weights(training_info_plain_lasso, rounding=4)
      w_df
```

```
Here are the learned weights for each regularization coefficient:
  * Each row represents a single regularization coefficient.
  * The last two columns represent the weight vector magnitudes.
  * Each of the other columns represent a feature weight.
```

```
[90]: <pandas.io.formats.style.Styler at 0x7f0ad419f520>
```

Some other questions that can be some food for thought:

1. Did Lasso regularization help at all in this setting?
2. Did small lasso regularization offer any benefir compared to small ridge regularization?
3. Did large lasso regularization do better than large ridge regularization?

- **Hint**: Why do you think $\lambda = 1$ acted more "aggressively" (i.e., supressed the weights more) with Lasso regularization than Ridge regularization?

12 3. Bank Note Authentication Problem With Induced Noise

In many applied learning tasks, the data set includes a lot of raw measurements (i.e., features), which may not be releavant to the final goal of classification. Trying to identify and exclude such "useless" features may not be an easy job, and would be task-specific, may require a lot of expert knowledge for identification, and would not be easy to automate (which is contrary to the whole point of machine learning).

To see in practice the phenomenon that irrelevant features make the problem more high-dimensional and therefore challenging, we will add a lot of "useless" features to the bank note authentication dataset we had. More precisely, 5000 random Gaussian Noise features will be concatenated to the original data set to "confuse" the learning model. Philosophically speaking, putting any weight other than zero towards these noise features when making decisions would hurt the overall testing performance of the model. However, the models are still free to put non-zero weights on such features.

Notice that this is not an additive noise to the features, and the "useful" information for making decisions will be untouched (i.e., the 4 original features are still included plainly without any modification). Therefore, one would expect a robust classifier to still be able to produce the same classification quality and ignore the noise features.

12.1 3.1 Ridge Regularization

```
[ ]: if perform_computation:
         all_features_noised = np.concatenate([all_features, np_random.
     ↪randn(all_features.shape[0], 5000)], axis=1)

         splitted_data = train_val_test_split(all_features_noised, all_labels,␣
     ↪train_frac=0.5, val_frac=0.25)

         # The "_nr" variable postfix is short for "noisy ridge".
         train_features_nr, train_labels_nr, val_features_nr, val_labels_nr,␣
     ↪test_features_nr, test_labels_nr = splitted_data

         training_info_noisy_ridge = svm_trainer(train_features_nr, train_labels_nr,␣
     ↪val_features_nr, val_labels_nr,
                                                 batch_size=32, num_epochs=50,␣
     ↪num_steps=300, eval_interval=30,
                                                 eta_tuner = lambda epoch: 1./(0.2 *␣
     ↪epoch + 1000.),
                                                 lambda_list = [0., 1e-3, 1e-2,␣
     ↪1e-1, 1e0],
                                                 regularization='ridge')
```

```
…
…
lambda=0.0 yielded a validation accuracy of 60.350 %
…
…
lambda=0.001 yielded a validation accuracy of 60.641 %
…
```

[ ]:
```
if perform_computation:
    generate_plots(training_info_noisy_ridge, heldout_acc_smoothing=0.99,␣
↪loss_smoothing=0.99, weight_smoothing=0.)
```

[ ]:
```
w_df = None
if perform_computation:
    w_df = print_weights(training_info_noisy_ridge, rounding=4)
    print('  * The non-noise features are the first four features.')
w_df
```

[ ]:
```
if perform_computation:
    get_test_accuracy(test_features_nr, test_labels_nr,␣
↪training_info_noisy_ridge)
```

Some observational questions:

1. Does SVM without any regularization work on this problem?

- In other words, how different is SVM performing from a random coin toss?

2. Did Ridge Regularization help?

- If so, by what margin? Is it significant?

12.2 3.2 Lasso Regularization

[ ]:
```
if perform_computation:
    all_features_noised = np.concatenate([all_features, np.random.
↪randn(all_features.shape[0], 5000)], axis=1)

    splitted_data = train_val_test_split(all_features_noised, all_labels,␣
↪train_frac=0.5, val_frac=0.25)

    # The "_nl" variable postfix is short for "noisy lasso".
    train_features_nl, train_labels_nl, val_features_nl, val_labels_nl,␣
↪test_features_nl, test_labels_nl = splitted_data

    training_info_noisy_lasso = svm_trainer(train_features_nl, train_labels_nl,␣
↪val_features_nl, val_labels_nl,
                                            batch_size=32, num_epochs=50,␣
↪num_steps=300, eval_interval=30,
                                            eta_tuner = lambda epoch: 1./(0.2 *␣
↪epoch + 1000.),
                                            lambda_list = [0., 1e-3, 1e-2,␣
↪1e-1, 1e0],
                                            regularization='lasso')
```

[ ]:
```
if perform_computation:
    generate_plots(training_info_noisy_lasso, heldout_acc_smoothing=0.99,␣
↪loss_smoothing=0.99, weight_smoothing=0.95)
```

[ ]:
```
w_df = None
if perform_computation:
    w_df = print_weights(training_info_noisy_lasso, rounding=4)
    print('  * The non-noise features are the first four features.')
w_df
```

[ ]:
```
if perform_computation:
    get_test_accuracy(test_features_nl, test_labels_nl,␣
↪training_info_noisy_lasso)
```

1. Did Lasso regularization help in this scenario?
2. If so, why did Lasso help with noisy features? Why didn't Ridge do so?

- **Hint**: In the weights table above, a look at the weights, the corresponding regularization norms, and then making a comparison between the last two trainings (i.e., Ridge vs. Lasso regularization trainings) is useful.
  - Try and justify what you observe.
  - Which regularization coefficient/scheme produced the most "sensible" set of weights? Why?

12.3 3.3 SVMLight

In the noisy features scenario, to get the SVM model to train properly using stochastic gradient descent, the learning rate sequences were modified from the previous part.

Ususally tunning such hyper-parameters can play a significant role in the method's final performance. This hyper-parameter tunning process can be extremely difficult and require a lot of expert-intervention.

SVMLight has some nice default hyper-parameters and search procedures set that were benchmarked and thoughtfully picked by its designers. If you'd rather not spending any time tunning hyper-parameters for whatever reason, and just want to get a feeling about a reasonable baseline, checking such carefully written libraries may be a good idea.

For the sake of comparison, let's see how SVMLight is doing without any intervention.

[ ]:
```
if perform_computation:
    from sklearn.datasets import dump_svmlight_file
    dump_svmlight_file(train_features_nl, train_labels_nl, 'training_feats.
 ↪data',
                       zero_based=False, comment=None, query_id=None,␣
 ↪multilabel=False)

    !chmod +x ./svmlight/svm_learn
    from subprocess import Popen, PIPE
    process = Popen(["../SGDSVM-lib/svmlight/svm_learn", "./training_feats.
 ↪data", "svm_model.txt"], stdout=PIPE, stderr=PIPE)
    stdout, stderr = process.communicate()
    print(stdout.decode("utf-8"))
```

[ ]:
```
if perform_computation:
    from svm2weight import get_svmlight_weights

    def svmlight_classifier(train_features):
        return (train_features @ svm_weights - thresh).reshape(-1) >= 0.

    svm_weights, thresh = get_svmlight_weights('svm_model.txt',␣
 ↪printOutput=False)

    train_pred = 2*(svmlight_classifier(train_features_nl))-1
    test_pred = 2*(svmlight_classifier(test_features_nl))-1

    train_acc = (train_pred==train_labels_nl).mean()
    test_acc = (test_pred==test_labels_nl).mean()
    print(f'The training accuracy of your trained model is %.
 ↪3f'%(train_acc*100) + '%')
    print(f'The testing accuracy of your trained model is %.3f'%(test_acc*100)␣
 ↪+ '%')
```

[ ]:
```
if perform_computation:
    import os
    for file in ['svm_model.txt', 'training_feats.data']:
        if os.path.exists(file):
            os.remove(file)
```