(https://colab.research.google.com/github/christianhbye/bayesian-analysis/blob/main/homeworks/HW9_288.ipynb)

Homework 9

Linear Regression, Regularization, and Logistic & Softmax Regression

This notebook is arranged in cells. Texts are usually written in the markdown cells, and here you can use html tags (make it bold, italic, colored, etc). You can double click on this cell to see the formatting.

The ellipsis (...) are provided where you are expected to write your solution but feel free to change the template (not over much) in case this style is not to your taste.

Hit "Shift-Enter" on a code cell to evaluate it. Double click a Markdown cell to edit.

Imports

In [1]:
```
import numpy as np
from scipy.integrate import quad
#For plotting
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
```

Mounting Google Drive locally

Mount your Google Drive on your runtime using an authorization code.

Note: When using the 'Mount Drive' button in the file browser, no authentication codes are necessary for notebooks that have only been edited by the current user.

In [2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Problem 1 - Ising Model

In earlier HW, we did a simple ML analysis by fitting datasets generated by polynomials in the presence of noise, and this highlighted the fundamental tension common to all ML models between how well we fit the training dataset and predictions on new data.

Here, we consider the problem of learning the Hamiltonian for the Ising model (https://en.wikipedia.org/wiki/Ising_model (https://en.wikipedia.org/wiki/Ising_model)) using the linear regression. This is a lattice model proposed to explain ferromagnetism in materials. In other physics courses, you learned that elementary particles have an intrinsic property called spin, which carries magnetic moments. The magnetism of a bulk material is made up of the magnetic dipole moments of the atomic spins inside the material. The classical Ising model postulates a lattice with a spin S on each site.

Now consider the 1D Ising model with nearest-neighbor interactions

$$H[\boldsymbol{S}] = -J \sum_{j=1}^{L} S_j S_{j+1}$$

on a chain of length L with periodic boundary conditions and $S_j = \pm 1$ Ising spin variables. J is the nearest-neighbor spin interaction

With $J = 1$, we draw a large number of spin configurations. We can draw them n number of times: we have n number of $\boldsymbol{S}^i$, which is a vector of length L. Hence, $\boldsymbol{S}$ is a matrix of $n \times L$.

1. You are given 1000 random Ising states with $L = 40$. (i.e. this state matrix $\boldsymbol{S}$ should have the dimension 1000×40, and its array elements are either 1 or -1.) Define a function which computes the energies H given $\boldsymbol{S}$. Calculate the energies of the first 10 states.

Hint: Each state $\boldsymbol{S}^i$ has its own energy, so $H[\boldsymbol{S}]$ is a vector of length $n = 1000$.

We adopt the periodic boundary conditions, so when $j = L$, $j + 1 = 1$.

In [3]:
```
S = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW9/state.tx
print( np.shape(S) )
print( S )
```

```
(1000, 40)
[[-1.  1.  1. ... -1.  1. -1.]
 [-1.  1.  1. ...  1. -1. -1.]
 [-1. -1.  1. ... -1. -1.  1.]
 ...
 [-1.  1. -1. ... -1. -1.  1.]
 [-1. -1.  1. ... -1. -1.  1.]
 [-1.  1.  1. ...  1.  1. -1.]]
```

In [4]:
```
def H_func(S, J=1):
  """
  Compute H given S and J
  """

  T = np.roll(S, -1, axis=1)  # T[i] = S[i+1]
  return -J * np.sum(S * T, axis=1)
```

In [5]:
```
H = H_func(S)

print("Energies of first 10 states")
print(H[:10])
```

```
Energies of first 10 states
[ 8.  4. -0. -0. 12. -4.  4.  8. -4. -4.]
```

Now, suppose you do not have the knowledge of the above Hamiltonian. Instead, you are given a data set of $i = 1 \dots n$ points of the form $\{(H[\boldsymbol{S}^i], \boldsymbol{S}^i)\}$. Your task is to learn the Hamiltonian using Linear regression techniques.

In the absence of any prior knowledge, one sensible choice is the all-to-all Ising model

$$H_{\text{model}}[\boldsymbol{S}^i] = -\sum_{j=1}^{L}\sum_{k=1}^{L} J_{j,k} S_j^i S_k^i.$$

Notice that this model is uniquely defined by the non-local coupling strengths J_{jk} which we want to learn. Importantly, this model is linear in $\mathbf{J}$ which makes it possible to use linear regression.

To apply linear regression, we would like to recast this model in the form

$$H_{\text{model}}^i \equiv \mathbf{X}^i \cdot \mathbf{J},$$

where the vectors $\mathbf{X}^i$ represent all two-body interactions $\{S_j^i S_k^i\}_{j,k=1}^{L}$, and the index i runs over the samples in the data set. To make the analogy complete, we can also represent the dot product by a single index $p = \{j, k\}$, i.e. $\mathbf{X}^i \cdot \mathbf{J} = X_p^i J_p$. Note that the regression model does not include the minus sign, so we expect to learn negative J's.

2. Create the matrix $\mathbf{X}$. Print $\mathbf{X}$.

Hint: For each state i, we have the state vector $\boldsymbol{S}^i$. $\mathbf{X}^i$ = $\boldsymbol{S}^i_{.T} \otimes \boldsymbol{S}^i_{.T}$, where $\otimes$ is the outer product. (https://en.wikipedia.org/wiki/Outer_product (https://en.wikipedia.org/wiki/Outer_product))

The dimension of $\mathbf{X}^i$ is $L \times L$. Hence, $\mathbf{X}$ has the diemension $n \times L \times L$. Reshape it so that it has the dimension $n \times L * L$ (1000×1600).

You can either use the for-loop or use np.einsum to do the outer product (https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.einsum.html (https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.einsum.html)).

In [6]:
```
def X_func(S):
  n, L = S.shape
  Xmat = np.empty((n, L, L))
  for i in range(n):
    Xmat[i] = np.outer(S[i], S[i])
  return Xmat.reshape(n, L*L)
```

In [7]:
```
X = X_func(S)
print(X)
```

```
[[ 1. -1. -1. ...  1. -1.  1.]
 [ 1. -1. -1. ... -1.  1.  1.]
 [ 1.  1. -1. ... -1. -1.  1.]
 ...
 [ 1. -1.  1. ... -1. -1.  1.]
 [ 1.  1. -1. ... -1. -1.  1.]
 [ 1. -1. -1. ... -1. -1.  1.]]
```

We can now do the linear regression.

$$H^i_{\text{model}} \equiv \mathbf{X}^i \cdot \mathbf{J},$$

Hence, you have data ($\mathbf{X}$, H)

3. Split the data into training and test samples. We choose that the first 70% of n states are training samples, the remaining 30% test samples. No need to shuffle the data because we are already given the random set of states. Print the diemension of training and test samples.

Hint: Here, H means $H[\boldsymbol{S}]$ or $H[\mathbf{X}]$ we calculated in Part 1.

In [8]:
```
ntrain = int(0.7 * len(X))
train_X, test_X = X[:ntrain], X[ntrain:]
train_H, test_H = H[:ntrain], H[ntrain:]

print("Dimensions of training set")
print(f"H: {train_H.shape}")
print(f"X: {train_X.shape}")

print("Dimensions of test set")
print(f"H: {test_H.shape}")
print(f"X: {test_X.shape}")
```

```
Dimensions of training set
H: (700,)
X: (700, 1600)
Dimensions of test set
H: (300,)
X: (300, 1600)
```

In earlier HW, you used "linear_model.LinearRegression()" from scikit-learn to do the linear regression and found that using a complex model can result in overfitting. To resolve such issues, we use regularization in machine learning. A regression model that uses L_1 regularization technique is called Lasso Regression (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html)) and model which uses L_2 is called Ridge Regression (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html)).

First, set up Lasso and Ridge regression models.

In [9]:
```
from sklearn import linear_model
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
%matplotlib inline

ridge = linear_model.Ridge()
lasso = linear_model.Lasso()
```

For each regression model, do the following:

1. Choose the regularization parameter λ.
2. Set the parameter using .set_params()
   e.g. lambda = 1; ridge.set_params(alpha=lambda)
3. Fit the model
   e.g. ridge.fit(training X samples, training H samples)
4. Compute the coefficient of determination R^2 of the prediction. This quantifies the performance of prediction.

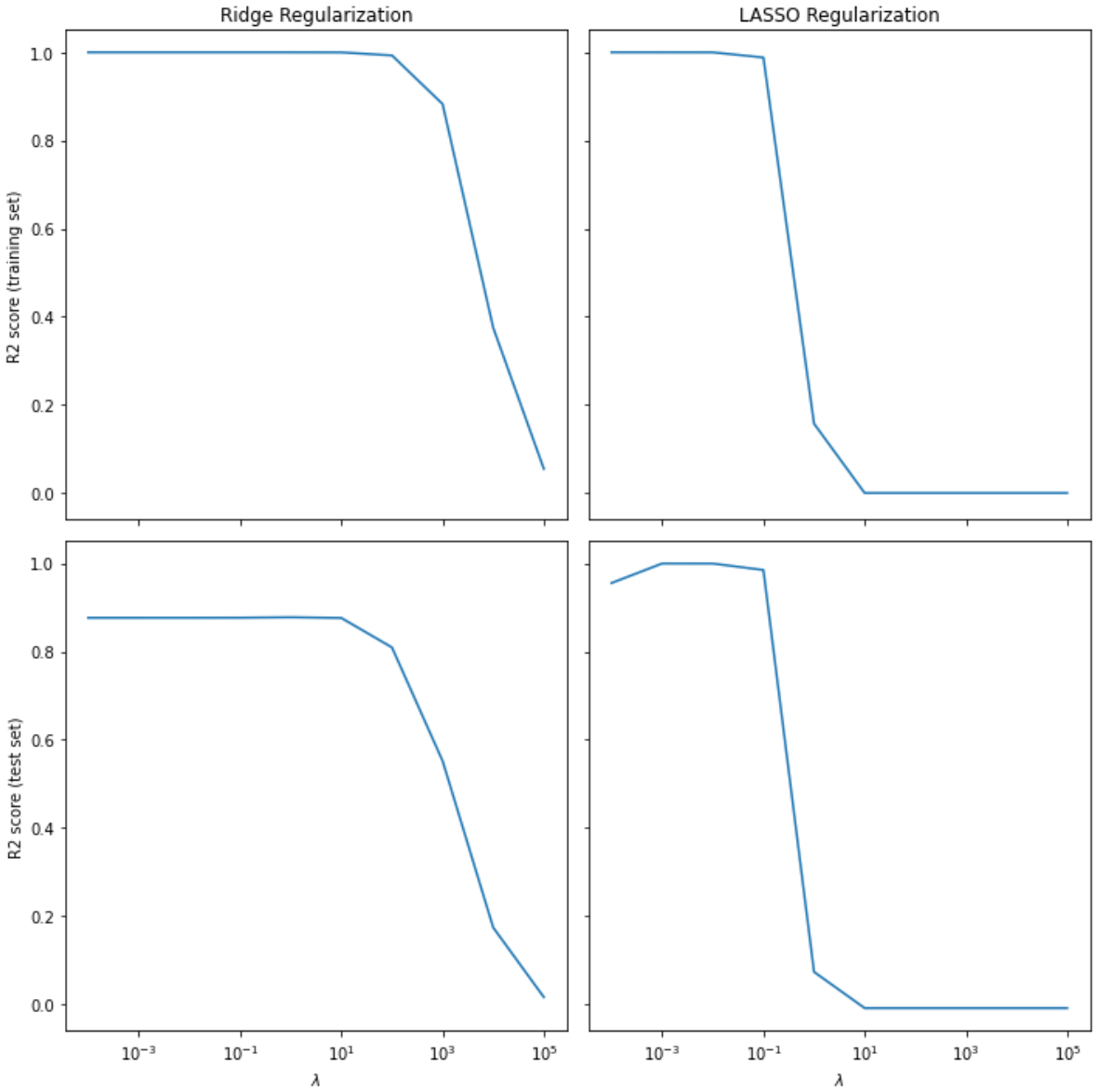
$$R^2 = 1 - \frac{\sum_{i=1}^{n} \left| y_i^{\text{true}} - y_i^{\text{pred}} \right|^2}{\sum_{i=1}^{n} \left| y_i^{\text{true}} - \frac{1}{n} \sum_{i=1}^{n} y_i^{\text{pred}} \right|^2}.$$

   e.g. ridge.score(training or test X samples, training or test H samples)

4. Let lambda = np.logspace(-4, 5, 10). Compute R^2 score for each lambda value and plot it as a function of lambda. Do both Ridge and LASSO regression. Also, make sure to show results for both training and test samples. (4 plots)

In [10]:
```
Lambda = np.logspace(-4, 5, 10)

r2_train = np.empty((len(Lambda), 2))  # 0th col is for ridge, 1st is
r2_test = np.empty((len(Lambda), 2))

for i, l in enumerate(Lambda):
  ridge.set_params(alpha=l)
  ridge.fit(train_X, train_H)
  r2_train[i, 0] = ridge.score(train_X, train_H)
  r2_test[i, 0] = ridge.score(test_X, test_H)

  lasso.set_params(alpha=l)
  lasso.fit(train_X, train_H)
  r2_train[i, 1] = lasso.score(train_X, train_H)
  r2_test[i, 1] = lasso.score(test_X, test_H)
```

In [11]:
```
fig, axs = plt.subplots(figsize=(10,10), nrows=2, ncols=2, sharex=True
for i in range(2):
  axs[0, i].plot(Lambda, r2_train[:, i])
  axs[1, i].plot(Lambda, r2_test[:, i])
axs[0, 0].set_title("Ridge Regularization")
axs[0, 1].set_title("LASSO Regularization")
axs[0, 0].set_ylabel("R2 score (training set)")
axs[1, 0].set_ylabel("R2 score (test set)")
plt.setp(axs[1], xlabel="$\\lambda$", xscale="log")
plt.tight_layout()
plt.show()
```



You should find that the regularization parameter λ affects the Ridge and LASSO regressions at scales, separated by a few orders of magnitude. Therefore, it is considered good practice to always check the performance for the given model and data with λ.

At $\lambda \to 0$ and $\lambda \to \infty$, both models overfit the data, as can be seen from the deviation of the test errors from unity (dashed lines), while the training curves stay at unity.

While the Ridge regression test curves are monotonic, the LASSO test curve is not -- suggesting the optimal LASSO regularization parameter is $\lambda \approx 10^{-2}$. At this sweet spot, the Ising interaction weights $\mathbf{J}$ contain only nearest-neighbor terms (as did the model the data was generated from).

So far we have focused on learning from datasets for which there is a continuous output. Classification problems, however, are concerned with outcomes taking the form of discrete variables (i.e. categories). Here, given a spin configuration of, say, the 2D Ising model, we'd like to identify its phase (e.g. ordered/disordered).

Onsager proved that this model undergoes a thermal phase transition in the thermodynamic limit from an ordered ferromagnet with all spins aligned to a disordered phase at the critical temperature $T_c/J = 2/\log(1+\sqrt{2}) \approx 2.26$.

An interesting question to ask is whether one can train a statistical model to distinguish between the two phases of the Ising model. If successful, this can be used to locate the position of the critical point in more complicated models where an exact analytical solution has so far remained elusive.

In other words, given an Ising state, we would like to classify whether it belongs to the ordered or the disordered phase, without any additional information other than the spin configuration itself.

To this end, we consider the 2D Ising model on a 40×40 square lattice, and use Monte-Carlo (MC) sampling to prepare 10^4 states at every fixed temperature T out of a pre-defined set. Using Onsager's criterion, we can assign a label to each state according to its phase: 0 if the state is disordered, and 1 if it is ordered. Our goal is to predict the phase of a sample given the spin configuration.

First, load the data for the following three types of phases: ordered ($T/J < 2.0$), critical ($2.0 \leq T/J \leq 2.5$) and disordered ($T/J > 2.5$).

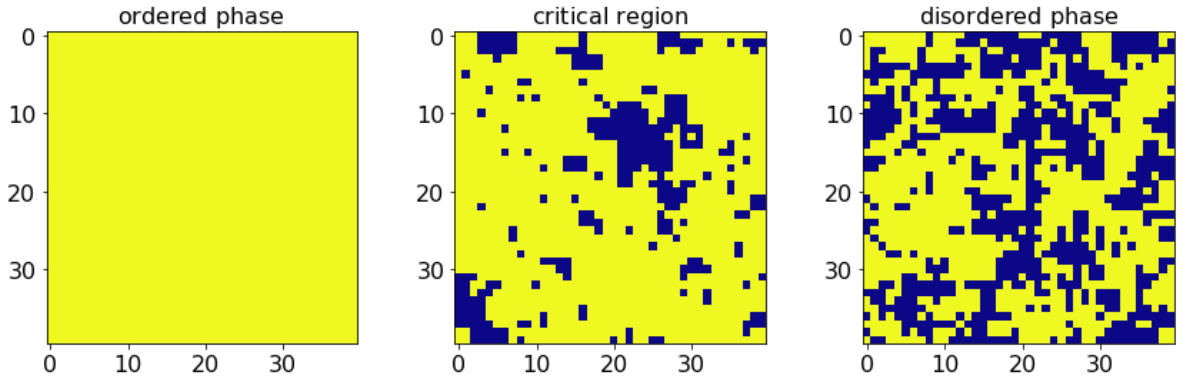
We are given data for $T/J = 1.0$, $T/J = 2.25$, and $T/J = 3.0$.

In [12]:
```
import pickle,os
from sklearn.model_selection import train_test_split

# load data

# state vector
file_name = "/content/drive/My Drive/P188_288/P188_288_HW9/Ising2DFM_r
state_vector = pickle.load(open(file_name,'rb'))

# ordered phases
file_name = "/content/drive/My Drive/P188_288/P188_288_HW9/Ising2DFM_r
data = pickle.load(open(file_name,'rb'))
data = np.unpackbits(data).reshape(-1, 1600)
data=data.astype('int')
data[np.where(data==0)]=-1 # map 0 state to -1 (Ising variable can tak

X_ordered=data[::20]
Y_ordered=state_vector[30000:40000][::20]

# critical phases
file_name = "/content/drive/My Drive/P188_288/P188_288_HW9/Ising2DFM_r
data = pickle.load(open(file_name,'rb'))
data = np.unpackbits(data).reshape(-1, 1600)
data=data.astype('int')
data[np.where(data==0)]=-1

X_critical=data[::20]
Y_critical=state_vector[80000:90000][::20]

# disordered phases
file_name = "/content/drive/My Drive/P188_288/P188_288_HW9/Ising2DFM_r
data = pickle.load(open(file_name,'rb'))
data = np.unpackbits(data).reshape(-1, 1600)
data=data.astype('int')
data[np.where(data==0)]=-1

X_disordered=data[::20]
Y_disordered=state_vector[110000:120000][::20]

L = 40
```

You have $\mathbf{X}$ for ordered, critical and disordered phases and corresponding state vector Y. For each phase (ordered, critical or disordered), we have 500 different 40×40 square lattices. So $\mathbf{X}$ has the dimension $500 \times 40 \times 40$. We reshape it into $500 \times 40 * 40 = 500 \times 1600$. The state vector is a vector of length 500.

Run the below cell to plot examples of typical states of the 2D Ising model for three different temperatures.

In [13]:
```
# plot few Ising states
from mpl_toolkits.axes_grid1 import make_axes_locatable

cmap_args=dict(cmap='plasma_r')

fig, axarr = plt.subplots(nrows=1, ncols=3)

axarr[0].imshow(X_ordered[100].reshape(L,L),**cmap_args)
axarr[0].set_title('$\\mathrm{ordered\\ phase}$',fontsize=16)
axarr[0].tick_params(labelsize=16)

axarr[1].imshow(X_critical[100].reshape(L,L),**cmap_args)
axarr[1].set_title('$\\mathrm{critical\\ region}$',fontsize=16)
axarr[1].tick_params(labelsize=16)

im=axarr[2].imshow(X_disordered[100].reshape(L,L),**cmap_args)
axarr[2].set_title('$\\mathrm{disordered\\ phase}$',fontsize=16)
axarr[2].tick_params(labelsize=16)

fig.subplots_adjust(right=2.0)

plt.show()
```



5. Combine ordered phase samples and disordered phase samples using np.concatenate. Using train_test_split (https://scikit-learn.org/stable/modules/generated /sklearn.model_selection.train_test_split.html), split it into training and test samples. Set train_size = 0.5. (50% of X is our training samples.) Print the dimension of training and test samples.

Using logistic regression, we will investigate how accurately we can distinguish between ordered and disordered phases.

In [14]:
```
X = np.concatenate((X_ordered, X_critical, X_disordered))
Y = np.concatenate((Y_ordered, Y_critical, Y_disordered))

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.
```

Here, we compare the performance of two different optimization routines: a liblinear (the default one for scikit's logistic regression, https://scikit-learn.org/stable/modules/generated /sklearn.linear_model.LogisticRegression.html (https://scikit-learn.org/stable/modules/generated /sklearn.linear_model.LogisticRegression.html)), and stochastic gradient descent (SGD, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html)). It is important to note that all these methods have built-in regularizers, and doing regularization is crucial in order to prevent overfitting.

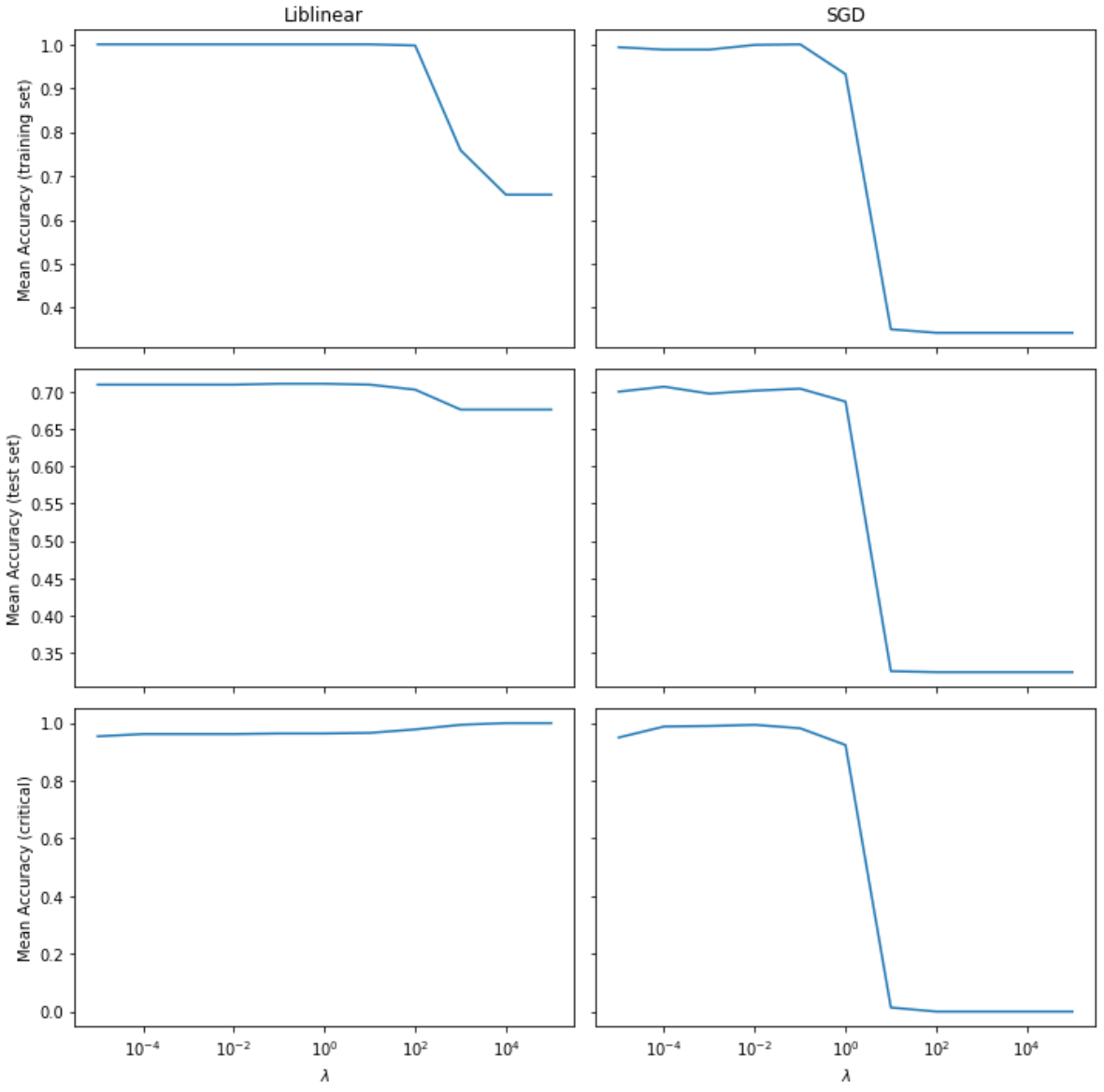
For each optimization routine, do the following:

1. Choose the regularization parameter λ.
2. Define the logistic regressor
   e.g. **liblinear**:
   logreg=linear_model.LogisticRegression(C=1.0/lambda,random_state=1,verbose=0,max_ite
   e.g. **SGD**: logreg_SGD = linear_model.SGDClassifier(loss='log', penalty='l2',
   alpha=lmbda, max_iter=100, shuffle=True, random_state=1, learning_rate='optimal')
   Use the above parameters, but you can play with them if you wish.
3. Fit the model
   e.g. logreg.fit(training X samples, training H samples)
4. Compute the mean accuracy on the given data. e.g. logreg.score(training or test X samples, training or test H samples)

4. Let lambda = np.logspace(-5,5,11). Compute the mean accuracy for each lambda value and plot it as a function of lambda. Do both liblinear and SGD. Also, show results for both training, test samples, and critical phase samples. (6 plots) What do you find?

In [15]:
```
Lambda = np.logspace(-5, 5, 11)

mean_train = np.empty((len(Lambda), 2))
mean_test = np.empty((len(Lambda), 2))
mean_crit = np.empty((len(Lambda), 2))

for i, l in enumerate(Lambda):
  logreg = linear_model.LogisticRegression(
      C=1.0/l, random_state=1, verbose=0, max_iter=1e3, tol=1e-5
  )
  logreg.fit(X_train, Y_train)
  mean_train[i, 0] = logreg.score(X_train, Y_train)
  mean_test[i, 0] = logreg.score(X_test, Y_test)
  mean_crit[i, 0] = logreg.score(X_critical, Y_critical)

  logreg_SGD = linear_model.SGDClassifier(
      loss="log", penalty="l2", alpha=l, max_iter=100, shuffle=True,
      random_state=1, learning_rate="optimal",
  )
  logreg_SGD.fit(X_train, Y_train)
  mean_train[i, 1] = logreg_SGD.score(X_train, Y_train)
  mean_test[i, 1] = logreg_SGD.score(X_test, Y_test)
  mean_crit[i, 1] = logreg_SGD.score(X_critical, Y_critical)
```

In [16]:
```
fig, axs = plt.subplots(figsize=(10,10), nrows=3, ncols=2, sharex=True
for i in range(2):
  axs[0, i].plot(Lambda, mean_train[:, i])
  axs[1, i].plot(Lambda, mean_test[:, i])
  axs[2, i].plot(Lambda, mean_crit[:, i])
axs[0, 0].set_title("Liblinear")
axs[0, 1].set_title("SGD")
axs[0, 0].set_ylabel("Mean Accuracy (training set)")
axs[1, 0].set_ylabel("Mean Accuracy (test set)")
axs[2, 0].set_ylabel("Mean Accuracy (critical)")
plt.setp(axs[2], xlabel="$\\lambda$", xscale="log")
plt.tight_layout()
plt.show()
```



The regression model perform well on the critical samples. Most of the models perform worse when λ becomes too large (typically when it is greater than 1). As expected, the training set accuracy is better than the test set accuracy, but they follow the same trend with λ. The Liblinear accuracy generally suffers less for large values of λ, wheres the SGD is more sensitive and seems to require careful finetuning of λ.

Problem 2 - Back to MNIST

Now, we generalize logistic regression to the case of multiple categories which is called

Softmax regression. A paradigmatic example of SoftMax regression is the MNIST classification problem. The goal is to find a statistical model which recognizes the ten handwritten digits. There are numerous practical applications of such a task, pretty much anywhere one can imagine dealing with large quantities of numbers.

In [17]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import check_random_state

# Load MNIST data
X = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW9/mnistX.d
Y = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW9/mnistY.d
```

"X" contains information about the given MNIST digits. We have a 28x28 pixel grid, so each image is a vector of length 784; we have 3800 images (digits), so X is a 3800x784 matrix. "Y" is a label (0-9; the category to which each image belongs) vector of length 3800.

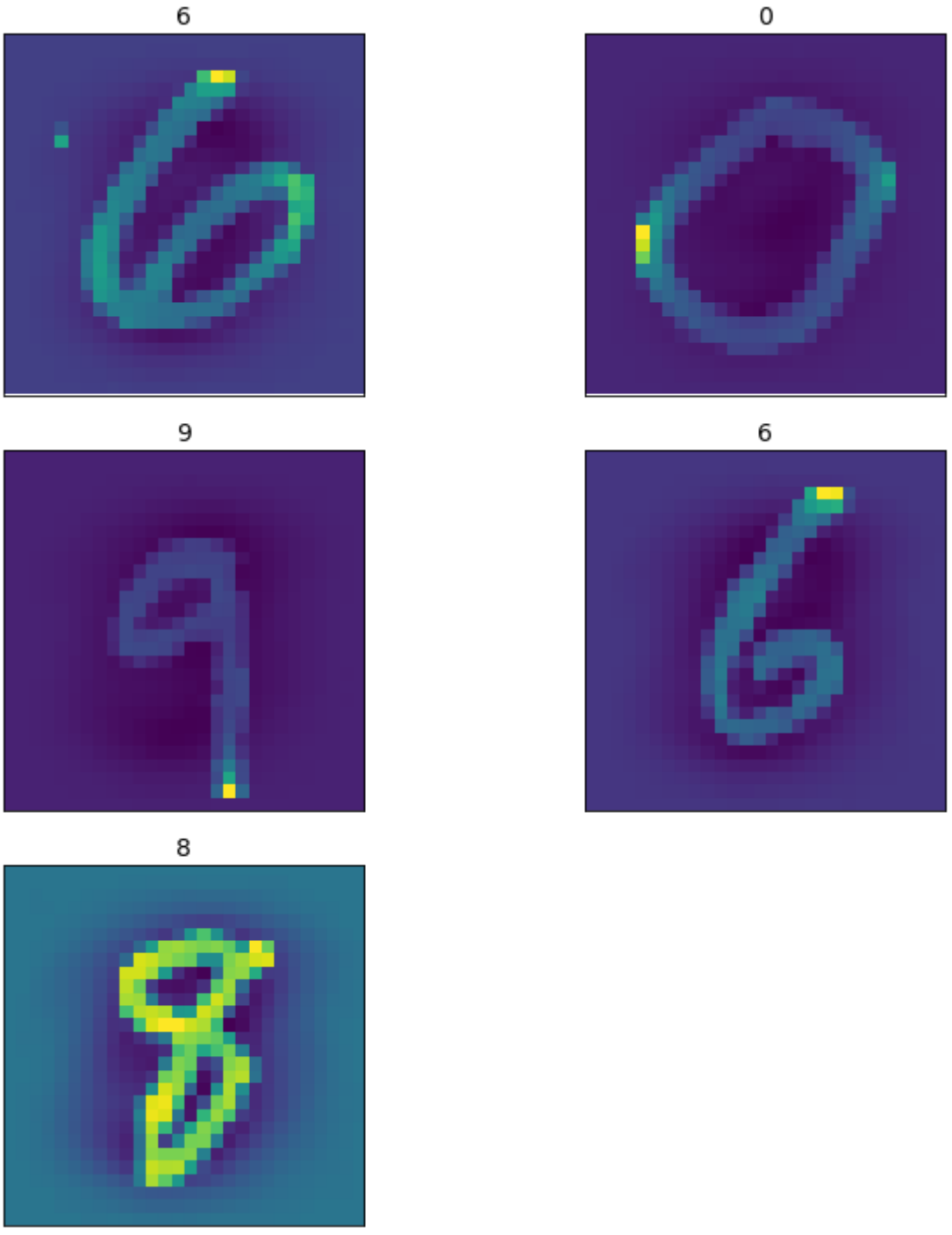
1. Randomly shuffle data and split them into training and test samples using train_test_split. Let train_size = 0.8. Print the dimension of training and test samples.

In [18]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, train_size=0
print("Dimensions:")
print("Training set")
print(f"X: {X_train.shape}")
print(f"Y: {Y_train.shape}")
print("Test set")
print(f"X: {X_test.shape}")
print(f"Y: {Y_test.shape}")
```

```
Dimensions:
Training set
X: (3040, 784)
Y: (3040,)
Test set
X: (760, 784)
Y: (760,)
```

2. Choose any five images and show what the images look like. Print the corresponding labels.

In [19]:
```
fig = plt.figure(figsize=(10, 10))
for i in range(5):
  ax = fig.add_subplot(3, 2, i+1)
  ax.imshow(X_train[i].reshape(28, 28))
  ax.set_title(int(Y_train[i]), fontsize=14)
  ax.set_yticks([])
  ax.set_xticks([])
plt.tight_layout()
plt.show()
```



Now, do logistic regression in the following way:

1. Scale data to have zero mean and unit variance (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html))
   (You can refer back to HW4)
2. Make an instance of the model using LogisticRegression (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)). Try "liblinear" and "sag" optimization algorithms.

liblinear: Use solver='liblinear' and use L1 norm in the penalization (penalty='l1'). Also set C=1e5 and tol=.3
sag: Use solver='sag' and use L2 norm in the penalization (penalty='l2'). Also set C=1e5 and tol=.1
e.g. model = LogisticRegression(...)

3. Train the model on the data
4. Predict the labels of test data.
5. Compute the accuracy

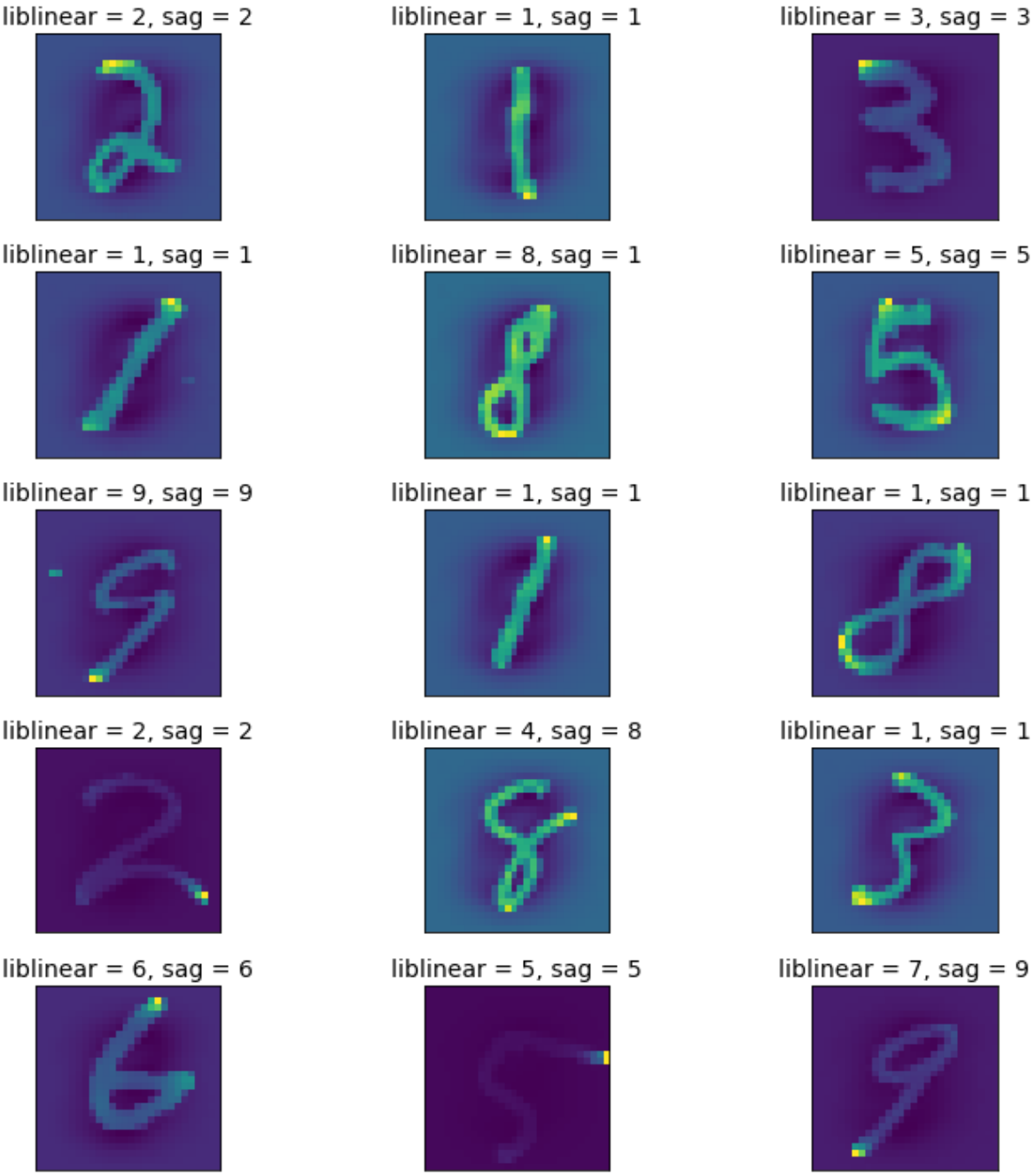
3. Using both liblinear and sag solvers, compute the accuracy of the test samples. Also, measure the training time (how long it takes to train the model on the data) using time.time().

In [20]:
```
import time

# scale all datassets by the same amount (so only fit to the training 
scaler = StandardScaler()
scaler.fit(X_train)
X_tr_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# liblinear
lib_model = LogisticRegression(solver="liblinear", penalty="l1", C=1e5
t0 = time.time()
lib_model.fit(X_tr_scaled, Y_train)
dt = time.time() - t0
print(f"Liblinear trained in {dt} seconds.")
lib_pred = lib_model.predict(X_test_scaled)
lib_acc = len(np.where(lib_pred == Y_test)[0]) / len(Y_test)
print(f"Liblinear accuracy: {lib_acc*100:.3f} %")

print("\n--------\n")
# sag
sag_model = LogisticRegression(solver="sag", penalty="l2", C=1e5, tol=
t0 = time.time()
sag_model.fit(X_tr_scaled, Y_train)
dt = time.time() - t0
print(f"Sag model trained in {dt} seconds.")
sag_pred = sag_model.predict(X_test_scaled)
sag_acc = len(np.where(sag_pred == Y_test)[0]) / len(Y_test)
print(f"Sag accuracy: {sag_acc*100:.3f} %")
```

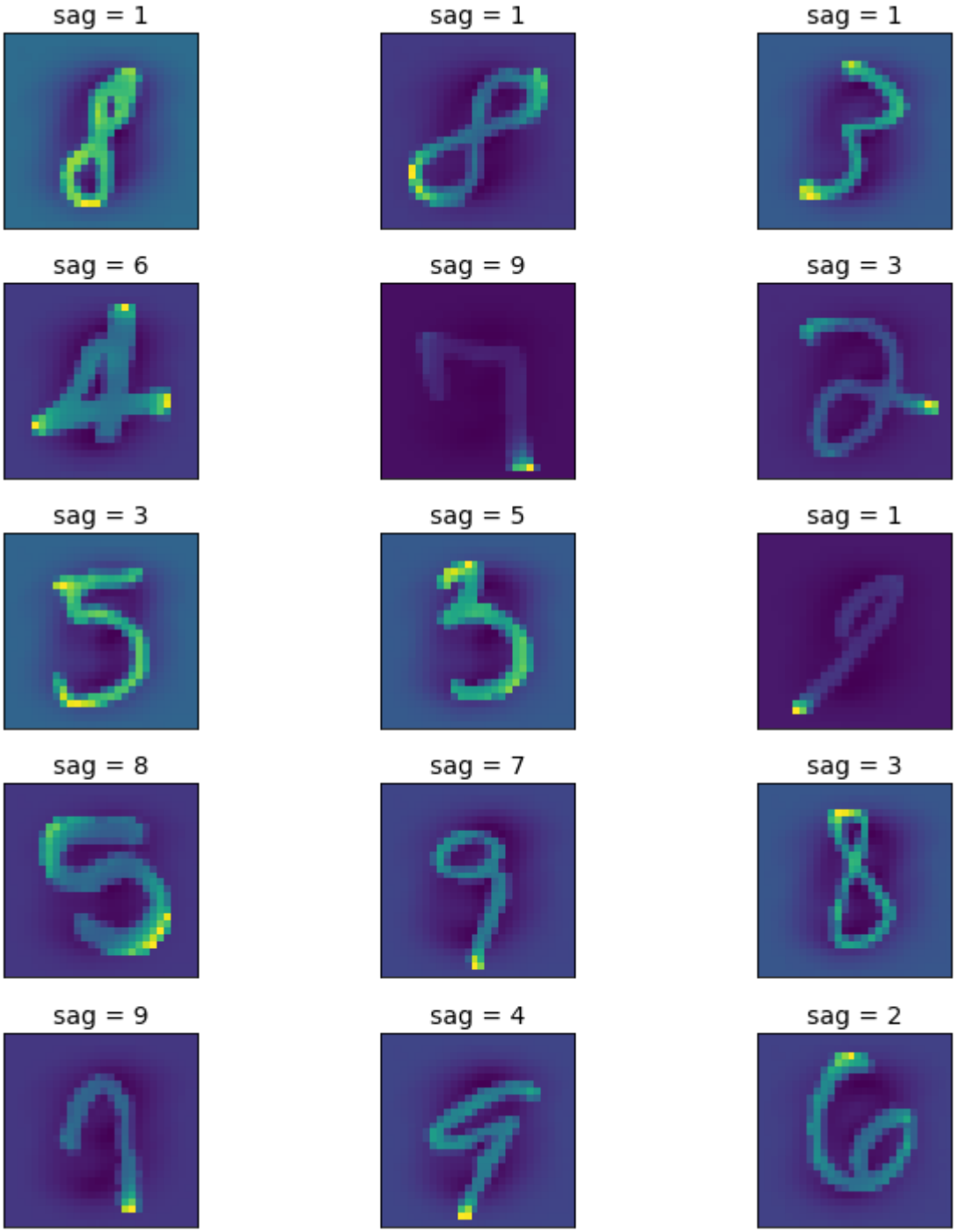
```
Liblinear trained in 1.6968967914581299 seconds.
Liblinear accuracy: 83.421 %

--------

Sag model trained in 1.3799433708190918 seconds.
Sag accuracy: 86.316 %
```

4. Choose any 15 images and show what the images look like. What are the predicted labels corresponding to them? Take a look at the misclassified samples. Use the sag solver.

In [22]:
```
fig = plt.figure(figsize=(10, 10))
for i in range(15):
  ax = fig.add_subplot(5, 3, i+1)
  ax.imshow(X_test[i].reshape(28, 28))
  ax.set_title(f"liblinear = {int(lib_pred[i])}, sag = {int(sag_pred[i
               fontsize=14)
  ax.set_yticks([])
  ax.set_xticks([])
plt.tight_layout()
plt.show()
```



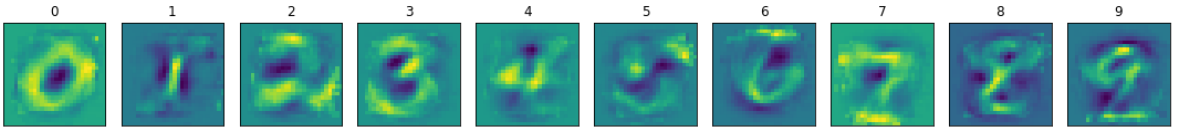
In [23]:
```
# misclassified with sag
mis = np.where(sag_pred != Y_test)[0]

fig = plt.figure(figsize=(10, 10))
for i in range(15):
  ix = mis[i]  # index of misclassified
  ax = fig.add_subplot(5, 3, i+1)
  ax.imshow(X_test[ix].reshape(28, 28))
  ax.set_title(f"sag = {int(sag_pred[ix])}", fontsize=14)
  ax.set_yticks([])
  ax.set_xticks([])
plt.tight_layout()
plt.show()
```



Here, we have 10 classes and 784 features. Using "coef_", we can get the coefficient of the features in the decision function. These are basically classication "weights."

5. Obtain the coefficient of the features for the model using the sag solver. (coef = sag.coef_) This is a 10x784 matrix. (number of classes x number of features) Reshape it into 28x28 and make a plot for each class. How do they look? Can you recognize the digits?

In [29]:
```
coef = sag_model.coef_.reshape(10, 28, 28)

fig, axs = plt.subplots(figsize=(15, 15), ncols=10)
for i, ax in enumerate(axs.ravel()):
  ax.imshow(coef[i])
  ax.set_title(i)
  ax.set_yticks([])
  ax.set_xticks([])
plt.tight_layout()
plt.show()
```



Most digits are easily recongizable except for 5.