Homework 8

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.

Link Okpy

In [ ]:

```
from client.api.notebook import Notebook
ok = Notebook('hw8_U.ok')
_ = ok.auth(inline = True)
```

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')
```

Problem 1 - Ising Model

In HW4, 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

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

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 S^i, which is a vector of length L. Hence, S is a matrix of $n \times L$.

1. You are given 1000 random Ising states with $L = 40$. (i.e. this state matrix 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 S. Calculate the energies of the first 10 states.

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

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

In [2]:

```
S = np.loadtxt("state.txt")
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 [ ]:

```
...
```

Now, suppose you do not have the knowledge of the above Hamiltonian. Instead, you are given a data set of $i = 1 \ldots n$ points of the form $\{(H[S^i], S^i)\}$. 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

$$H_{\rm model}[S^i] = -\sum_{j=1}^{L}\sum_{k=1}^{L} J_{j,k} S^i_j S^i_k.$$

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

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

where the vectors $\mathbf{X}^i$ represent all two-body interactions $\{S^i_j S^i_k\}_{j,k=1}^{L}$, 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. $\mathbf{X}^i \cdot \mathbf{J} = X^i_p J_p$. Note that the regression model does not include the minus sign, so we expect to learn negative J's.

2. Create the matrix X. Print X.

Hint: For each state i, we have the state vector S^i. $\mathbf{X}^i = S^i_{.T} \otimes S^i_{.T}$, 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 [ ]:

```
...
```

We can now do the linear regression.

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

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 $H[S]$ or $H[\mathbf{X}]$ we calculated in Part 1.

In [ ]:

```
...
```

In HW4, 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 [ ]:

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

```
...
```

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 $(2.0 \le T/J \le 2.5)$ 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 [5]:

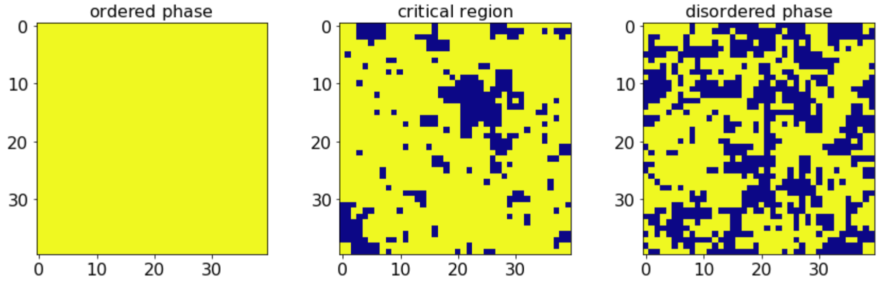
```python
import pickle,os
from sklearn.model_selection import train_test_split

# load data

# state vector
file_name = "Ising2DFM_reSample_L40_T=All_labels.pkl"
state_vector = pickle.load(open(file_name,'rb'))

# ordered phases
file_name = "Ising2DFM_reSample_L40_T=1.00.pkl"
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 take values +/-1)

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

# critical phases
file_name = "Ising2DFM_reSample_L40_T=2.25.pkl"
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 = "Ising2DFM_reSample_L40_T=3.00.pkl"
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 [6]:

```python
# 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 (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 [ ]:

```python
...
```

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_iter=1E3,tol=1E-5)
   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 [ ]:

```
...
```

Problem 2 - 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.

Yann LeCun and collaborators collected and processed 70000 handwritten digits to produce what became known as the most widely used database in ML, called MNIST. Here we will take a subsample of it: 3800 digits. Each handwritten digit comes in a square image, divided into a 28×28 pixel grid. Every pixel can take on 256 nuances of the gray color, interpolating between white and black, and hence each the data point assumes any value in the set {0,1,…,255} . Since there are 10 categories in the problem, corresponding to the ten digits, this problem is a generic SoftMax regression task.

Ever since, the MNIST problem has become an important standard for benchmarking the performance of more sophisticated Machine Learning models. Often times, there are contests for finding a new constellation of hyperparameters and/or model architecture which results in a better accuracy for correctly classifying the digits.

In [7]:

```
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("mnistX.dat")
Y = np.loadtxt("mnistY.dat")
```

"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 [ ]:

```
...
```

2. Choose any five images and show what the images look like. What are the labels corresponding to them?

Hint: each image is a vector of length 784. So reshape it into a 28x28 matrix.
  X_0 = X_train[0]
  X_0 = X_0.reshape((28, 28))
Then, make a plot using imshow.
  plt.imshow(X_0, cmap=plt.cm.gray)

In [ ]:

```
...
```

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))
   scaler = StandardScaler()
   X_train = scaler.fit_transform(X_train)
   X_test = scaler.transform(X_test)
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
   e.g. model.fit(training X sample, training Y samples)
4. Predict the labels of test data.
   e.g. digit_predict = model.predict(test X samples)
5. Compute the accuracy
   e.g. model.score(test X samples, test Y samples)

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 [ ]:

```
...

import time

# liblinear
t0 = time.time()
...
run_time = time.time() - t0

print('liblinear:')
print('Run time: %.3f s' % run_time)
print('accuracy: %.3f' % score)

# sag
t0 = time.time()
...
run_time = time.time() - t0

print('sag:')
print('Run time: %.3f s' % run_time)
print('accuracy: %.3f' % score)
```

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 [ ]:

```
...
```

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 [ ]:

```
...
```

To Submit

Execute the following cell to submit. If you make changes, execute the cell again to resubmit the final copy of the notebook, they do not get updated automatically.
We recommend that all the above cells should be executed (their output visible) in the notebook at the time of submission.
Only the final submission before the deadline will be graded.

In [ ]:

```
_ = ok.submit()
```