Assignment (5)

Artificial Intelligence and Machine Learning (24-787 Fall 2020)

Due Date: 12/04/2020 (Friday) @ 11:59 pm EST
All the files must be put into one directory **andrewID-HW5/**. Convert the jupyter notebook to a pdf file. Ensure that the submitted notebooks have been run and the cell outputs are visible - **Hint:** Restart and Run All option in the Kernel menu. Zip the directory **andrewID-HW5/** and submit the zip on canvas. You can refer to Numpy documentation while working on this assignment. Any deviations from the submission structure shown below would attract penalty to the assignment score. Please use Piazza for any questions on the assignment.

```
andrewID-HW5/
             q1.ipynb
             q1.pdf
             q2.ipynb
             q2.pdf
             q3.ipynb
             q3.pdf
```

Submission file structure

PROBLEM 1

Support Vector Machines **[30 points]**

In this problem, you'll practice to solve a classification problem using SVM. In class, we have seen how to formulate SVM as solving a constrained quadratic optimization problem. Now, you will implement an SVM in the primal form. Conveniently, the **cvxopt** module in python provides a solver for constrained quadratic optimization problem, which does essentially all of the work for you. This solver can solve arbitrary constrained quadratic optimization problems of the following form:

$$\begin{aligned} \arg\min_{\mathbf{z}} \quad & \frac{1}{2}\mathbf{z}^T\mathbf{Q}\mathbf{z} + \mathbf{p}^T\mathbf{z} \\ \text{s.t.} \quad & \mathbf{G}\mathbf{z} \leq \mathbf{h} \end{aligned} \tag{1}$$

a) (Programming problem) You are given a data file **clean_lin.txt**. The first two columns are coordinates of points, while the third column is label.

Now let's implement the following quadratic optimization problem:

$$\begin{aligned} \arg\min_{\mathbf{w},b} \quad & \frac{1}{2}||\mathbf{w}||^2 \\ \text{s.t.} \quad & y_i(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1, \quad (i = 1, 2, \cdots) \end{aligned} \tag{2}$$

To do this, you have to find how to turn this constrained optimization problem into the standard form shown in (1). Things you should keep in mind: which variable(s) does **z** need to represent? What do you need to construct **Q**, **p**, **G** and **h**?

Hint: **z** should be 3×1. **G** should be $n \times 3$, where n is the number of training samples.

Train the linear SVM on the data using ***cvxopt.solvers.qp(Q,p,G,h)***. Plot the decision boundary and margin boundaries. You should have a plot similar to Fig. 1

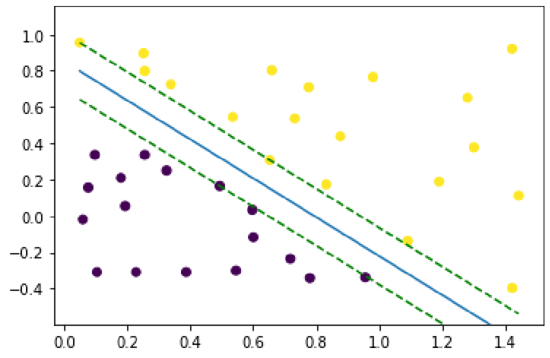


Fig. 1: Expected decision boundary for the linearly separable dataset.

b) (Programming problem) Now let's go ahead to solve a linearly non-separable case using SVM. Load the training data in **dirty_nonlin.txt**.

As discussed in the lecture, introducing slack variables for each point to have a soft-margin SVM can be used for non-separable data. The soft-margin SVM, which has following form:

$$\begin{aligned}
\underset{\mathbf{w},b,\xi}{\text{arg min}} \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_i \xi_i \\
\text{s.t.} \quad & y_i(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i, \quad (i = 1, 2, \cdots) \\
& \xi_i \geq 0, \quad (i = 1, 2, \cdots)
\end{aligned} \tag{3}$$

At this point, use $C = 0.05$ in your code, but keep that as a variable because you will be asked to vary C in the subsequent questions. Again, you want to think about what the terms in the standard formulation represent now.

Hint: The problem is still quadratic in the design variables. So your solution, if found, will be the global minimum. **z** should be $(n + 3) \times 1$. **G** should be $2n \times (n + 3)$. If you construct your design vector as $[w_1, w_2, b, \xi_1, \cdots, \xi_n]$, you shall see that **G** can be constructed by putting together four sub matrices (upper left 3×3, lower right $n \times n$ etc.).

Finally, plot the decision boundary and margin boundaries. You should expect to have a plot similar to Fig. 2.

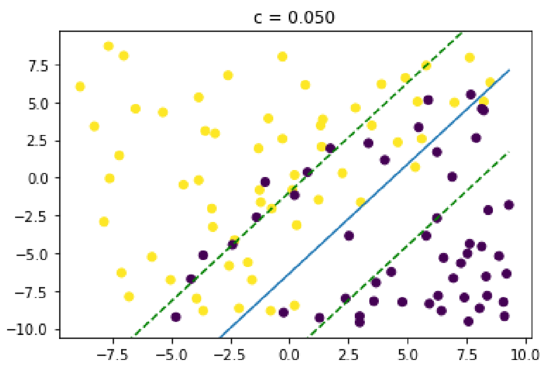


Fig. 2: Decision boundary for the linearly non-separable dataset.

c) (Programming problem) Use your code in **b)** to draw 4 plots, each corresponding to different values of $C = [0.1, 1, 100, 1000000]$. Discussion your observations of the decision margins.

(Problem 2 on next page)

PROBLEM 2

House price prediction **[40 points]**

In this problem, you'll get some hands-on experience on a real-world dataset. Assume you already have a model that could predict the house price, you are going to predict the residual error of this model. Specifically, your target is the error in logarithmic scale (**logerror**), and your inputs are some features that may contribute to the house price, like the number of bathrooms, location, etc.

Data Description

You are given three .csv files, namely **train.csv**, **properties.csv** and **data_dictionary.csv**. The **train.csv** includes transaction id, log error, and transaction date. The **properties.csv** includes the transaction id and a list of features. The **data_dictionary.csv** gives some explanation of what each feature means.

a) (Programming problem) Firstly, let's load and visualize data.

1. Use pandas to load **train.csv** and **properties.csv** into two DataFrame. Merge them into a new DataFrame based on the transaction id.
2. Since there are some outliers at both ends of log error, replace these outliers with proper maximum/minimum value. Specifically, replace log error from 0% to 1% with value at 1%, and 99% to 100% with value at 99%. (HINT: You can use np.percentile)
3. Make a scatter plot and histogram of **logerror**. (HINT: You should find logerror follows a nice normal distribution)

b) (Programming problem) Usually, a dataset will have a lot of missing values (NaN), so we'll first do data cleaning before moving to the next part.

1. Build a new DataFrame that has two columns: **"column_name"** and **"missing_count"**. The **"column_name"** contains every column in merged DataFrame. The **"missing_count"** counts how many missing data that certain column has.
2. Adding a new column called **"missing_ratio"** to this new DataFrame. This new column stores the ratio of number of missing data to the number of total data.
3. Fill the missing data of each feature in merged DataFrame (the df you got from **a**) by its mean value.

c) (Programming problem) At this point, we can do some univariate analysis. For this problem, we will look at the correlation coefficient of each of these variables.

1. For each variable, compute the correlation coefficient with logerror. After that, sort and make a bar chart of these coefficients.
2. If your bar chart is right, there are few variables at the top of this chart without any correlation values. Explain why.

d) (Programming problem) Now it's time to apply some non-linear regression models.

1. To simply, in this problem we only use float value features. Therefore, drop the categorical features, "id" and "transactiondate" in your merged dataset.
2. Split your data into train and test following the 70/30 ratio. Use **sklearn.ensemble.RandomForestRegressor** to train your data.
3. Report the importance of each feature using bar chart. Also, report the Mean Square Error (MSE) of test set.

e) (Programming problem) Cross-validation is a useful way to avoid overfitting. In problem **e**, only use the first 500 samples of your dataset in **d.2**.

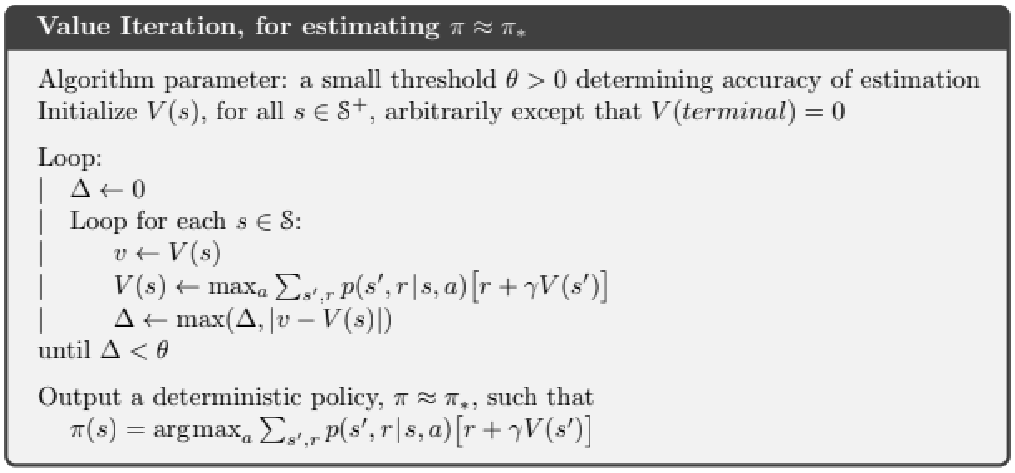
1. Use **sklearn.model_selection.KFold** to implement KFold cross validation with fold=5. Print the overall MSE and compare with your result in **d.2**.
2. Run your algorithm in **d.2** 100 times with random seeds from 0 to 99. Use the same train/test ratio. Report the MSE of each prediction. Explain your findings and what's the advantage of cross-validation. (HINT: Randomly running your algorithm in d.2 means you should pass random seed into both your "train_test_split" process and random forest model)

(Problem 3 on next page)

PROBLEM 3

Policy and Value Iteration Recap

Policy iteration computes value function for a fixed policy, and then continuously improves policy π using new value function(see Fig 4). Notice argmax(a) is essentially a greedy policy with regard to the current value function. Greedy policy might remain the same for a particular state if there is no better action. In practice, it converges in few iterations compared to value iteration and is finite since MDP process only have finite amount of policies. Recall that value iterations turning the Bellman equation into an update rule, and each of its iterations involves policy evaluation (see Fig 4). Value iteration requires multiple sweeping through the state set. When converges, we terminate once the value function changes by only a small amount in a sweep.



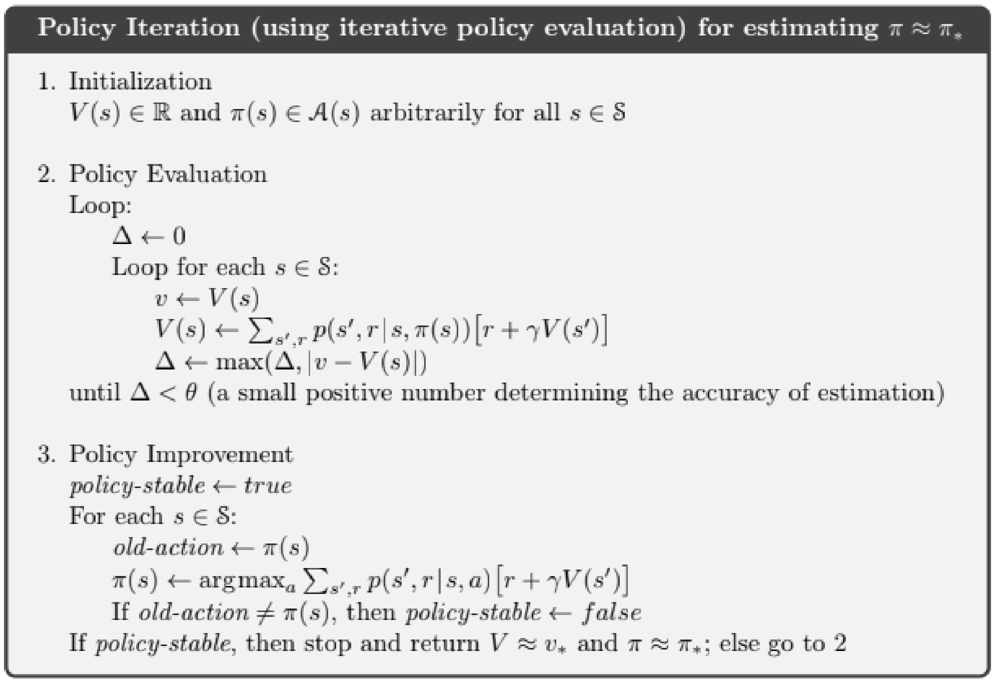
Value Iteration, for estimating $\pi \approx \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
```

Fig. 3: Value Iteration, taken from Section 4.4 of Sutton and Barto's RL Book (2018)



Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in \mathcal{S}$:
       old-action $\leftarrow \pi(s)$
       $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
       If old-action $\neq \pi(s)$, then policy-stable $\leftarrow$ false
   If policy-stable, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Fig. 4: Policy Iteration, taken from Section 4.3 of Sutton and Barto's RL Book (2018)

Frozen Lake with Policy Iteration **[50 points]**

In this problem, you will implement **policy iteration** only. We will be working with the deterministic version of the OpenAI Gym FrozenLake environment2, defined in the source code provided. Note that you do not need to install OpenAI Gym. We have created a custom environment that you can directly import into your code. In this environment, the agent starts at a fixed starting position, marked with "S". The agent can move up, down, left, and right. In the deterministic environments, the up action will always move the agent up, the left will always move left, etc. We have provided a 4×4 map:

$$\begin{array}{c} SFFF \\ FHFH \\ FFFH \\ HFFG \end{array}$$

There are four different tile types: Start (S), Frozen (F), Hole (H), and Goal (G).

- The agent starts in the Start tile at the beginning of each episode.
- When the agent lands on a Frozen or Start tile, it receives 0 reward.
- When the agent lands on a Hole tile, it receives 0 reward and the episode ends.
- When the agent lands on the Goal tile, it receives +1 reward and the episode ends.

States are represented as integers numbered from left to right, top to bottom starting at zero. For example in a 4×4 map, the upper-left corner is state 0 and the bottom-right corner is state 15.

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{array}$$

Note: Be careful when implementing policy evaluation. Keep in mind that in this environment, the reward function depends on the current state, the current action, and the **next state**. Also, terminal states are slightly different.

A starter code has been included in **q3.ipynb**, and it contains a section on how to use the Frozen lake Environment.

a) 40 points: Find the optimal policy using synchronous policy iteration. Specifically, you will implement **policy_iteration_sync()** in **q3.ipynb**, writing the policy evaluation and policy improvement steps in **evaluate_policy_sync()** and **improve_policy()**, respectively. In addition, you would also implement **print_policy()** to print the policy for given value functions of the environment (Refer to the jupyter note-book for further information). Record for 4×4:

- the time taken for execution for both maps.
- the number of policy improvement steps.
- the total number of policy evaluation steps.

b) 10 points: Plot a graph showing the value function of all states as a function of number of policy iterations for 4×4 maps.