Project 5: Password Guessing

This project is due on **April 30, 2023** at **11:59 pm CST**. We strongly recommend that you get started early. You will get 80% of your total points if you turn in up to 72 hours after the due date. Please let the TAs know if you decide to take this option (email or private campuswire post). Late work will not be accepted after 72 hours past the due date.

The project is split into two parts. Checkpoint 1 focuses on the rules we will use to guess the passwords. The recommended deadline for checkpoint 1 is **April 20, 2023**. We strongly recommend that you finish the first checkpoint before the recommended deadline. However, you do NOT need to make a separate submission for each checkpoint. That is, you need to submit your answers for all checkpoints in a single folder before the project due date on **April 30, 2023** at **11:59 pm CST**. Detailed submission requirements are listed at the end of the document.

This is an individual project; you SHOULD work **individually**.

The code and other answers you submit must be entirely your own work, and you are bound by the Student Code. You MAY consult with other students about the conceptualization of the project and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone else. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions MUST be submitted electronically in your git directory, following the submission checklist given at the end of each checkpoint. Details on the filename and submission guideline is listed at the end of the document.

Release date: April 10, 2023 at 10:00 am CST
Checkpoint 1 Recommended Due date: April 20, 2023
Checkpoint 2 Due date: April 30, 2023 at 11:59 pm CST

Course repo setup: `https://edu.cs.illinois.edu/create-gh-repo/sp23_cs463`
Course assignment repo: `https://github.com/illinois-cs-coursework/sp23_cs463_.release`
Your submission repo: `https://github.com/illinois-cs-coursework/sp23_cs463_[NetID]`

Introduction

If users create their passwords by reusing or slightly modifying existing passwords, they will face serious security and privacy threat caused by leaked passwords. In this MP, we will build a password-guessing algorithm based on the paper: "The Next Domino to Fall: Empirical Analysis of User Passwords across Online Services (CODASPY 2018)", so that we can have a quantitative understanding of this threat. Given one of a user's passwords, our task is to successfully guess the other password the user created within a certain number of attempts.

This assignment is individual. Your implementation must be compatible with the provided Python3 code skeleton and interfaces it defined. ***Please read the assignment carefully before starting the implementation.***

Objectives

- Analyze the patterns of users' password creation procedure.
- Be able to guess users' passwords within a certain number of attempts successfully.

Checkpoints

This machine problem is split into 2 checkpoints.

In Checkpoint 1, you will understand the rules of password transformation and preprocess the dataset to see what kind of rules/transformations are used frequently. You will need to implement functions to find rules and transformations and statistically analyze their frequency based on the given code skeleton.

In Checkpoint 2, you will understand how to convert a string password into a numerical vector and then use the Naive Bayes model for multi-classification. You will also need to understand a naive way of password guessing which we call sequential guessing.

Code Skeleton

In the distributed folder, you should find the following folders and files:

- dataset/raw_data/train.txt, dataset/raw_data/test.txt

  Those two files are in the same format. The first value of each row is the hashed user ID, and the following values are the passwords collected from various sources belonging to the same user.

- dataset/raw_data/qwerty_graph.txt

This file is only useful for the "seqkey" rule. It saves a dictionary that maps from each possible character to all characters around it on the keyboard.

- cp1/rules/

  This folder includes 7 rule-related files. Each file is related to one rule mentioned in the paper (except the combination rule). Some of the rule files have been fully implemented, and you will be asked to implement the remaining ones. It will be better for you to also understand the rules before implementing them.

- cp1/rule_process.py/

  This folder contains functions that help to process the raw data. One part of the functions helps to categorize the pairs of passwords and check their transformation. The other part helps to statistically analyze the frequency of transformation.

- cp1/learn_transformation.py/

  This file leverages the functions from the previous files to save preprocessed data files in the dataset/processed_data/ folder. After running with `python cp1/learn_transformation.py` , you can check your intermediate results in this folder.

- cp2/model.py

  This file contains the definition of the Naive Bayes multi-classifier. We have implemented it for you.

- cp2/pipeline.py

  This file contains the implementation of the whole password-guessing procedure. From cp1 you have implemented guessing rules and transformations. Now it is time for wrapping everything up together. You need to implement several helper functions in this file, including the vectorization function and the sequential guess generator. The pipeline.py will load the cp1-generated data and process the train/test split. Then it will train the NB model and do the sequential guessing. Finally, it will report the guess success rate.

Grading Environment and Runtime

Please read this part carefully because it will tell you how your work will be graded on our server. This time we decide to make everything on your local and our server the same.

The max running time is 8 mins. We have set the testing set scale for you, you will see "DONT MODIFY THIS" in the code skeleton. Don't modify those parameters otherwise you may fail to run in 8 min.

We have imported all the packages you may need in the skeletons, which are numpy, tqdm, json, sys, random, sklearn.naive_bayes.GaussianNB, itertools. You should not import anything else since the whole MP is simplified and easy to implement.
The environment will use python 3.8.8 and will only use the packages in the requirements.txt (and also the built-in packages).

The cp2/pipeline.py will be the grading script, which means what you get in your local by running this file is exactly the same as what we get in grading env. Please notice that you need to ensure the seq-key graph, transformation matrix, and train/test raw data are in the position as we indicate in the pipeline.py by variables "graph_path", "mat_path", "train_raw_path" and "test_raw_path". If you strictly follow the code skeleton then they will be in the right place (if you just fill in those TODO functions).

The training and testing data will be exactly the same as we provide. One thing we have to mention is, since all the data are publicized, don't try to cheat or hardcode anything.

After running pipeline.py, you will get a success rate which will be used for grading. More than 45% success rate should be graded full points. Since you don't need to design anything yourself like MP1, there is no need for worrying about success rate, just ensure you follow exactly to the handout and code skeleton.

Checkpoint 1 (50 points)

It is a common behavior for users to modify an existing password and sign up for a new service with the newly modified password. To measure password modification behavior, the first task is to define the commonly used rules (rule_capitalization, rule_substring, etc.). Checkpoint 1 is designed to help you understand the rules and the transformation expressions related to each rule defined in the paper, and then use those rules and transformations to identify and categorize the modification behavior for each pair of passwords from the same user.

In this MP, we will use 7 rules for our password-guessing algorithm. 4 of them have been fully implemented (rule_identical, rule_reverse, rule_seqkey, and rule_CSS). And you will be asked to implement the remaining 3 rules (rule_substring, rule_capt, and rule_leet).

1.1 Define Rules (12 points)

Your first task in this machine problem is to define the 3 remaining rules. You should implement the first function in each rule file respectively (3 functions in total). For each function, given a pair of passwords (strings), you should output a value (boolean) to indicate if this pair of passwords is belonging to the specific rule. The detailed descriptions are shown below:

- cp1/rules/rule_substring.py: check_substring()
  - Substring rule means, either input password can be considered as a substring of the other.
- cp1/rules/rule_capt.py: check_capt()
  - Capt rule means, pw1 can be transformed to pw2 by (de-)capitalizing any number of characters.
- cp1/rules/rule_leet.py: check_leet()
  - Leet transformation refers to replacing certain characters with other similar-looking ones. In the code skeleton, it means, pw1 can be transformed to pw2 by replacing any number of characters with the characters in the same set in the leet list provided (note that one character can appear in multiple sets in the default leet list).
  - The default leet_list contains the most common leet transformations. And implementing functions with this list is enough for this MP. A more complete leet_list (comment out currently in the code skeleton) is also provided as a reference.

1.2 Define and Find Transformations (12 points)

For each rule we will use in this MP, we seek to learn a list of possible transformations during the training phase. After checking a pair of passwords belonging to a certain rule, we want to extract the transformation from the given pair. Note that the implementation of transformation in this MP will be different from the transformation descriptions in the original paper. You should implement the second function in each rule file respectively (3 functions in total). For each function, given a pair of passwords (strings), you should output a transformation (string) to represent the way that the pair of passwords are transformed from one to the other. The detailed descriptions are shown below:

- cp1/rules/rule_substring.py: check_substring_transformation()
  - Substring transformation is characterized by a string as
    "head\t[Additional Str in Head]\ttail\t[Additional Str in Tail]".
    The substring transformation representation records the string remains in the head part and the string remains in the tail part after removing the substring password from the other. Some specific examples are shown in the code skeleton. "\t" is used to separate each term in the output.
- cp1/rules/rule_capt.py: check_capt_transformation()

  - Capt transformation is characterized by a string as "**[head if head char needs (de-)capitalized]\t[tail if tail char needs (de-)capitalized]\t[total # of chars need (de-)capitalized]**".
    The capt transformation representation records, to transform one password to the other, if the head/tail chars need to be (de-)capitalized and the # of chars need to be (de-)capitalized. Some specific examples are shown in the code skeleton. "\t" is used to separate each term in the output.

- cp1/rules/rule_leet.py: check_leet_transformation()

  - Leet transformation is characterized by the following steps
    1. From pw1 to pw2, collect pairs of chars transformed from one to the other. Each pair: "**[char in pw1][char in pw2]**".
    2. Sort all pairs in lexicographic order and connect all of them with the separator "\t".
    The capt transformation representation records, to transform pw1 to pw2, which pairs of characters are used with the leet rule.

1.3 Apply Transformations (16 points)

Now we have defined all rules and their transformation representations. Implemented functions (e.g. generate_train_data()) in cp1/rules_process.py now can use the previous functions to generate intermediate results in dataset/processed_data/. However, we also need to know how to apply the transformation to a given password and generate possible guessing passwords for the future guessing part. You should implement the remaining functions in each rule file respectively (4 functions in total). For each function, given an input password (string) and a transformation (string) defined previously, you should output a list of all possible guessing passwords (list of strings) after applying the transformation on the input password. The detailed descriptions are shown below:

- cp1/rules/rule_substring.py: guess_target_as_substring()

  - This function is used to cover the case where the target guessing password is the substring of the input password. Intuitively, a substring password may be created by deleting certain characters from either the head or the tail. Thus, guessing passwords will be possible substring that removes characters from the head of the input password and those removes from the tail.
  - E.g. input password is "abcd", possible substrings (removes chars from the head) are ["bcd", "cd", "d"], possible substrings (removes chars from the head) are ["abc", "ab", "a"]. The final output should be ["bcd", "cd", "d", "abc", "ab", "a"].

- cp1/rules/rule_substring.py: apply_substring_transformation()

  - The above function (guess_target_as_substring()) will be used here. We will give a name for this kind of transformation as "special_trans_as_substring". That means, if the input transformation is "special_trans_as_substring", this function should output the return value of guess_target_as_substring().

  - For other normal transformation strings in the substring rule, it will record the string remains in the head part and the string remains in the tail part after removing the substring password from the other as mentioned before. So the return list will contain one possible guessing string by adding the remaining head string to the head and the remaining tail string to the tail.

- cp1/rules/rule_capt.py: apply_capt_transformation()
  - The capt transformation string records, if the head/tail chars need to be (de-)capitalized and the # of chars need to be (de-)capitalized. Thus, the output list should contain all possible strings that fit with this transformation. That means, compared with the input string, any string in the output list should match the (de-)capitalized head/tail char and the number of (de-)capitalized chars specified by the transformation.

- cp1/rules/rule_leet.py: apply_leet_transformation()
  - The leet transformation string records the pairs of characters replaced. The possible guessing passwords should be created by applying at least once and only once for each pair of characters in the transformation on the input password.
  - Both directions should be considered. E.g. transformation is "3e\ta@". The forward direction is "3"→"e" and "a"→"@". The backward direction is "e"→"3" and "@"→"a".

1.4 Statistically Record Transformations (10 points)

Before applying the transformations to make guesses, we need to first sort the transformation for each rule with their frequency. Intuitively, the more times a transformation appeared in the training set, the more likely this transformation will be applied in the testing set. You should implement the following function for this part:

- cp1/rules_process.py: transformation_stat()
  - This function takes two input values. "processed_data_file" is the path of intermediate results (as JSON file) of functions including "generate_train_data()". "output_file" is the path of the file where you will save your final result dictionary (as JSON file).
  - The "processed_data_file" saves a nested list as an intermediate result after processing the raw data. The element of the list is
    [[pw1 (string), pw2 (string)], [rule belongs to (string), transformation (string)]].
  - You will save a dictionary in a JSON file at the end. The keys are all the rules (strings) and the values are the lists of transformations belonging to the corresponding rule (sorted by their frequency).

- Two special cases need to be handled. One for the seqkey rule, the transformation recorded in the value list will be the walk found. E.g. input transformation is "qwert\t1q2w3e" ("qwert" and "1q2w3e" are independent walks), then "qwert" will count as appeared once, and "1q2w3e" will count as appeared once respectively. The other is for the substring rule mentioned previously. We need to add the special transformation "special_trans_as_substring" at the front of the value list related to the substring rule.

1.5 Run Checkpoint1 Code Together (0 points)

We have prepared all the codes needed to process the raw data and learn the transformation now. Please run the following code and check your intermediate results.

```
python learn_transformation.py
```

Your processed training data will be in "dataset/processed_data/train_pairs.json"
Your learned rules and transformations will be in "dataset/processed_data/rule_transformation_in_order.json"

Checkpoint 2 (40 points)

In this checkpoint, you will learn how to convert a string password into a numerical vector and how to implement a simple password-guessing method called sequential guessing. You will learn how to extract features from a password. What are the important characteristics of a string password? How to quantify them and generate a reasonable input for the Naive Bayes multi-classifier? How to guess the password?

2.1 Implement the vectorization procedure (15 points)

For this part, you need to implement the "vectorize(password)" function in pipeline.py. The detailed instruction and tips are written as comments in the provided code skeleton. We extract 18 different numerical features from a string password. For one of them, we have provided the code since it is slightly complicated. You need to implement the other 17 features and return a numpy array with shape (1,18).
Please read the comments in pipeline.py carefully which will make the task much easier.

2.2 Implement the sequential guessing generator (25 points)

For this part, you need to implement the "sequential_guessor(password)" function in pipeline.py. The detailed instruction and tips are written as comments in the provided code skeleton.

We have provided an example in code skeleton comments to explain what is sequential guessing and how to write the code based on our setting.
The testing set is a bunch of pairs. Each pair contains two passwords, we want to guess the second by the first one.
The procedure of sequential guessing is as follows:
1. Vectorize the first password in pair using the vectorized() function.
2. Using the vector as input to Naive Bayes model to get the output probabilities. The output will be a dim-7 vector, each element is the probability the password belongs to that rule (for the definition of rule, please refer to CP1).
3. Sort the rules from high probability to low probability
4. In the transformation matrix (define by the variable "transformation_mat" in pipeline.py, we have already defined the order of transformations for each rule (the order they appear in the list which is the value of the Python Dict)
5. Then start guessing. You will be given a maximum number of guessing attempts. Start from the highest probability rule, guessing until running out of all transformations, then switch to the next rule (the second highest one), guessing all transformations...
6. If exceeding the maximum number of guessing attempts or finding the correct password, stop and switch to the next pair.
Notice, the transformation may output more than one password, each of them should count as one guess. The order of them should be exactly the same as the transformation outputs.

2.3 Run Checkpoint2 Code Together (0 points)

We have provided a well-structured and fully-commented code skeleton (model.py and pipeline.py). As we mentioned before, after reading this handout please read the comments carefully. After finishing TODOs in the code skeleton, to test your code please simply run as follows:

```
python pipeline.py
```

There will be a printed success rate in your terminal which indicates how many pairs have been guessed correctly.

Please remember to do CP1 before CP2, since you need the results (rule_transformation_in_order.json and train_pairs.json) from CP1. You also need the "apply_transformation" functions for each rule.

Submission Instructions

There will be no credits from Coursera for this MP. The assignment for MP5 on Coursera is only a placeholder. The mandatory submission for this MP is only the GitHub coding part. The newly created files will not be used in the grading environment (for the intermediate .json files, we will run the ".py" files locally and check the output ".json" files). Please use the following commands to push your changes as your submission for grading (do this often for better version control). The grading will be based on the latest submission before the deadline.

```
git add *
git commit -m "REPLACE THIS WITH YOUR COMMIT MESSAGE"
git push origin main
```