

## Project 5: Password Guessing

This project is due on **Dec 03, 2024 at 11:59 pm CST**. We recommend starting early. You have 3 grace days total for the 5 MPs, giving you a 24-hour extension on deadlines if needed. You can use up to 2 grace days per MP, but no more than 3 in total. If you submit after the due date and beyond the grace period, you'll lose 20% of the total points for each extra late day.

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 **Nov 20, 2024**. We highly recommend finishing this checkpoint before the recommended deadline. Please make sure your code is submitted to your personal **GitHub** repository by **December 03, 2024, at 11:59 PM CST**.

In PrairieLearn, you are required to submit a JSON file containing the following key-value pairs:

- "netid": Your NetID as a lowercase string.
- "github\_token": The fine-grained GitHub access token for your CS 463 repository.

This JSON file is essential for each checkpoint submission, as it verifies your identity and access rights. Before submitting, please refer to the instructions on how to generate the access token at the following link: (<https://github.com/illinois-cs-coursework/generating-access-tokens>). Ensure that you create the token correctly to avoid any submission issues.

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

All code and answers submitted must be solely your own work, in accordance with the Student Code. You may discuss project concepts and question interpretations with peers, but you must not view or collaborate on any part of another student's solution. Published references may be consulted, provided they are properly cited (e.g., in program comments), similar to academic citation practices

**This semester, our autograder has anti-cheating checks. Plagiarism will be regularly monitored, and any form of cheating—whether giving or receiving help—is not allowed. If cheating is detected, you'll receive one warning; further offenses will lead to a zero on the assignment and a report to the relevant authorities.**

Solutions **MUST** be submitted electronically in Github repository, 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:** Nov 12, 2024

**Checkpoint 1 Recommended Due date:** Nov 20, 2024

**Checkpoint 2 Due date:** Dec 03, 2024 at 11:59 pm CST

---

# 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 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.

- `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.

- `rule_process.py`

This file 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.

- `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 learn_transformation.py`, you can check your intermediate results in this folder.

- `model.py`

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

- `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.

## Checkpoint 1 (40 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:

- rules/rule\_substring.py: check\_substring()
  - Substring rule means, either input password can be considered as a substring of the other.
- rules/rule\_capt.py: check\_capt()
  - Capt rule means, pw1 can be transformed to pw2 by (de-)capitalizing any number of characters.
- 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:

- 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.
- 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.
- 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 (12 points)

Now we have defined all rules and their transformation representations. Implemented functions (e.g. generate\_train\_data()) in 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:

- 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 tail) are ["abc", "ab", "a"]. The final output should be ["bcd", "cd", "d", "abc", "ab", "a"].
- 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.
- 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.
- 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 (0 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. We have implemented the following function for this part for you (you don't need to write any code, but remember to run this function before cp2 when testing on your local):

- `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 (4 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"

### What to submit

- `rules/rule_capt.py`
- `rules/rule_substring.py`
- `rules/rule_leet.py`
- `dataset/processed_data/train_pairs.json`
- `dataset/processed_data/rule_transformation_in_order.json`

## Checkpoint 2 (60 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?

There is no partial credit for checkpoint2. We will test based on your final accuracy. Larger than 0.44 will get full points (60) otherwise 0. Please run the auto-grader after finishing both 2.1 and 2.2.

### 2.1 Implement the vectorization procedure (25 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 (35 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.

### What to submit

- pipeline.py

## Autograder

We have set up an autograder on PrairieLearn. To use it, you must first enroll in the course CS 463: Computer Security II by visiting <https://us.prairielearn.com/pl>.

After enrolling, navigate to the **Assessments** section within CS 463: Computer Security II, where you will find **MP5** here:

[https://us.prairielearn.com/pl/course\\_instance/163006/assessment/2478152](https://us.prairielearn.com/pl/course_instance/163006/assessment/2478152).

In PrairieLearn, you are required to submit a JSON file containing the following key-value pairs:

- "netid": Your NetID as a lowercase string.
- "github\_token": The fine-grained GitHub access token for your CS 463 repository.

This JSON file is essential for each checkpoint submission, as it verifies your identity and access rights.

Before submitting, please refer to the instructions on how to generate the access token at the following link: (<https://github.com/illinois-cs-coursework/generating-access-tokens>). Ensure that you create the token correctly to avoid any submission issues.

# Submission Instructions

You need to complete two tasks to submit your work:

## 1. Upload to GitHub

Check your repository to confirm that all files have been uploaded successfully at the end of each checkpoint. We will also perform anti-cheating checks to detect any plagiarism. Please note that any form of code plagiarism is strictly prohibited, including copying parts of someone else's code or code from previous students.

### Folder Structure

- Make sure there is a folder "mp5" inside the root folder of your Github repository in the main branch. Your score will be deducted if you fail to do so.

Below is the folder structure for MP5, showing the files you need to submit:

```
mp5
├── pipeline.py
├── dataset
│   ├── processed_data
│   │   ├── train_pairs.json
│   │   └── rule_transformation_in_order.json
└── rules
    ├── rules_capt.py
    ├── rules_leet.py
    └── rules_substring.py
```

## 2. Upload to PrairieLearn

To facilitate the grading of your submissions, it is essential that you submit a JSON file containing: your NetID and your GitHub token.

Make sure you follow the below steps to ensure that your submission process runs smoothly.

### 1. JSON File Requirements:

- Your submission must be in JSON format, which should include the following keys:
  - "netid": Your NetID as a lowercase string.
  - "github\_token": The GitHub access token generated for your account.

### 2. Token Visibility:

- Please remember to save the generated token securely, as it will only be displayed temporarily after creation. Once you navigate away from the page, you will not be able to retrieve it again.

### 3. Access Permissions:

- Ensure that you have provided Read-Only access to the contents of your repository.

### Submission Limit

Please note that you are limited to a maximum of 40 submissions for each checkpoint on PrairieLearn. Be mindful of each submission you make, as your submission count is limited.

### Git Upload

Use the following commands to push your submission for grading (it is advisable to do this frequently for better version control). Grading will be based on the latest submission to the repository before the deadline. Please push to your main branch rather than creating a separate branch.

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

### Important Note

1. Your highest grade and grading report will be automatically recorded in our server
2. You must submit through **PrairieLearn** to our auto-grader at least once before the deadline to get a grade. We don't grade your code manually, the only way is to use our auto-grader.
3. Try auto-grader as early as possible to avoid the heavy traffic before the deadline. **DON'T DO EVERYTHING IN THE LAST MINUTE!**
4. If the auto-grader is experiencing downtime or behaving unexpectedly, don't worry. Reach out to us on Campuswire, and we will address the issue and provide an extension if needed.
5. We have anti-cheating mechanisms, **NEVER CHEAT OR HELP OTHERS CHEAT!**
6. Please use Python version 3.12. Checkpoint 2 execution may take some time as you are building a model. Please be patient and allow the process to complete.
7. Never abuse the auto-grader. You should only submit your own code. Please do not abuse the auto-grader in any form. Abusing the autograder is considered a form of violation of the student code of conduct, and the corresponding evidence will be gathered and reported.
8. There is no partial credit for cp2. Please finishing both 2.1 and 2.2 before running the auto-grader.