Mathematical Modeling for Computational Science

AM115/AM215 Fall 2025

Harvard John A. Paulson School of Engineering and Applied Sciences

Michael P. Brenner
Chris Gumb

Homework 1

Modeling Tournaments; Bash & Git

Issued:	September 10, 2025
Due:	September 26, 2025

Note that this is roughly a 2 week exercise. *Do not procrastinate the work.*

This problem set consists of two parts: **Modeling**, and **Software Engineering**. The modeling questions provide motivation for *why* we need high quality software engineering. Our intent is to set modeling problems of increasing difficulty so that you will need the use the software engineering methods to solve them at high quality.

Software Engineering

Problem 1: Setup Your Class Git Repository

In this problem you setup your class Git repository. This repository will reside inside a directory on your laptop. You are entirely free how you manage your directory structure locally. One possibility might be to have a `classes` directory below which you have directories for your classes and within them class related data including your private class repository for AM215. For example:

```
classes/
|-- AM215
|   |-- git
|   |   |-- myrepo <- your private Git repository
|   |   \-- main
|   \-- AM215_other_data
\-- CS2050 (some other class in this directory)
```

The following steps are necessary for successful assignment submissions in AM215. *We will not be able to grade your work in this class if you have not performed the following steps.*

a) Sign in on https://code.harvard.edu. This is a GitHub Enterprise instance managed by Harvard University. The username you see in your profile (something similar to `abc123`) is your Harvard NetID.

Material and work for this class is managed through the AM215 organization on that platform. The next steps will not work if you don't complete this step first.

You need access to the AM215 organization in order to proceed with the next subtasks. Access will be granted by the Teaching Staff.

b) Once you have been added to the AM215 organization, proceed with the remaining steps in this tutorial:

This will create your repository on the *remote*. Now you need to set it up on your computer. You can clone the repository by repeating the following steps on other computers as well.

i) Create the directory for your *local* repository. If you follow the suggestion above, this command could look like this

```
$ mkdir -p classes/AM215/git/myrepo
$ cd classes/AM215/git/myrepo
```

The second command changes into the newly created directory.

ii) The next step is to *initialize* a new local repository. If you have not used Git before, make sure you issue the following two commands

```
$ git config --global user.name "Your Name"
$ git config --global user.email your@email.edu
```

As you guessed it, they will tell Git a little something about you. This information is necessary whenever you create commits such that people know who is responsible for the work in the commit. You may not necessarily use your Harvard email if you prefer to use another one instead. You can now initialize your local repository with

```
$ git init
```

iii) Now you can add the remote repository such that Git knows where to push changes to

```
$ git remote add origin git@code.harvard.edu:AM215/abc123.git
```

This assumes that your NetID is `abc123`. You can obtain the link to your remote from inside your repository by clicking on the green button in the top right that says "Code".[1] It is suggested you pick the SSH version but HTTPS would also work.

Note the general convention is to name the main remote repository `origin`. You can name it whatever you like but consistency is good practice when you name things (especially when you work with many Git repositories). Your local repository is now ready. You can always list the configured remotes using

[1] A remote in Git may not necessarily be an URL, it may just as well be a path to another repository on your local file system.

```
$ git remote -v
```

At this point you can perform some customization if you like. By default, Git names the *default branch* `master` (deprecated behavior). This information will be propagated to the remote with your first push. If you like to change the name of the branch you can do so by

```
$ git branch -M main
```

to rename it `main`.[2] Note that `master` is subject to change. You can configure Git with your own name for a default branch using

```
$ git config --global init.defaultBranch <name>
```

Common names are `main`, `trunk` or `development`.

c) Before you push anything to your remote repository, you need to add some data.

i) Create a `README.md` file in the root of your private repository with

```
$ echo '# My AM215 Repository' > README.md
```

ii) It is very important to keep your history clean from polluting files that are not needed to execute the code. Examples of such files are object files of compiled code, editor backup files or the `.DS_Store` file on MacOSX. Such files do not contribute to the value of software and should be *ignored* in your repository. You can achieve this by setting up a `.gitignore` file. It is customary to have a global `.gitignore` on the repository root. You can additional `.gitignore` files in any (sub-)directory you like.
You can create one with the following commands

```
$ echo -e "**/__pycache__\n*.pyc\n.DS_Store\n" > .gitignore
$ echo -e "*~\n*.bak\n*.swp\n*.zip\n" >> .gitignore
$ echo -e ".bash_history\n**/.cache\n**/.local" >> .gitignore
```

You are free to add more patterns you like to ignore now or at any moment in time. Note that this will happily ignore the specified file patterns.
In case you need to add and commit one of those files at some point, you can always force add them using the `-f` option in `git add -f`.
You can check the status of your repository with `git status` *(This is an important command and you should become accustomed to using it often).*

iii) Add the *untracked* data to the *staging area* with

```
$ git add .gitignore README.md
```

and create your first commit with

[2] This is the name of the branch and is not related to the AM215 `main` repository found here

```
$ git commit -m 'Initial commit'
```

iv) Now you can issue your first push to the remote repository with

```
$ git push -u origin master
```

Depending whether you have chosen different naming, please replace `origin` (the remote) and `main` (the branch to push) accordingly. You only need the `-u` option with the two arguments whenever you push a new branch to a remote that *is not tracked* by Git.
If you have chosen to link your remote to the HTTPS URL you will need to setup a personal access token in your GitHub profile. If you have chosen the SSH URL you will need to setup a SSH key and upload the public key to GitHub. You can follow the tutorials on the webpage.

These steps will become second nature to you after this homework and the next few lectures. What you have done is create a `README.md` file using a common Unix/Linux command, and followed the usual Git workflow of staging (via `git add`), commiting changes to your local repo (via `git commit`), and pushing to the remote repository (`git push`). Note that the `-m` option to `git commit` allows you to specify a commit message directly from the command line rather than via your default text editor (e.g. `vim`).

d) Homework is solved on specific Git branches and submitted via pull requests that target your default branch. See the homework tutorial for the details. The following steps should guide you through for this homework. You are expected to do this on your own for the following homework.

i) Be sure you are on your default branch (e.g. `main`). You can create the required branch for this homework with

```
$ git checkout -b hw1
```

Alternatively if your Git version is 2.23 or newer you can use

```
$ git switch -c hw1
```

We will only be grading the work you do on this branch. You will lose *5 points* if you do not solve the homework on the `hw1` branch. As a token of good will, we will not take these 5 points off for this first homework.

You are free to make other branches off of `hw1` or your default branch, say for individual problems, but you are responsible to merge this work into your active homework branch if you want the work graded.

ii) Create your `hw1` directory for this homework

```
$ mkdir -p hw1/submission
```

Note that this will also create the `submission` sub-directory in which you are expected to put your homework solution for grading.
Optional: instead of creating your `hw1` directory manually, you could add the main class repository as another remote and checkout the relevant homework material using Git

```
$ git remote add class git@code.harvard.edu:AM215/main_2025.git
$ git fetch class
```

The name `class` for this remote is again arbitrary. The `git fetch` command updates this remote locally. You can then checkout this (or future) homework material with

```
% $ git checkout class/master -- Homework_Files/Homework1/
```

assuming you are located at the root of your private repository. This is useful if there are additional or auxiliary files distributed with the homework, you can get it all in one go. If you perform a `git status` you will observe that new files have been added in the *staging area*. It is good practice to commit them before you continue

```
$ git commit -m 'Checkout hw1'
```

You can check your current history with

```
$ git log --oneline
96f55f8 (HEAD -> hw1) Checkout hw1
4ae4ade (origin/master, master) Initial commit
```

It tells you that there are two commits and you are currently on the `hw1` branch (`HEAD` is pointing to `hw1`).

iii) Finally, you can create a pull request at this point or wait until you are done with this homework. This step is optional at this point *but is required to submit your homework before the deadline*. You can follow the tutorial on the class website.
Up to here, your private repository should look at least similar to

```
myrepo
|-- hw1
|   \-- submission
\-- README.md
```

Problem 2: Git (*10 points*)

Deliverables: for this problem, copy the `hw1/sandbox` directory from the forked repository discussed below into your `hw1/submission` directory in your private Git repository. The following contents should be in the `sandbox` directory:

1. `README.md`
2. `P2_fork.png`
3. `P2_merge.png`
4. `P2_remote.png`

You must also create a *separate* pull request for this problem as described at the very end of the exercise.

An important part of software development is being able to control and revise a *history of changes* made to the source from possibly different contributors. Version Control Systems (VCS) are the primary tools used to keep track of actual changes to code. Git is the most common VCS used and has established to an industry standard. It is what we will use for this course.

a) ***10 points***

In this problem we are working with another Git repository that can be found in the class organization `https://code.harvard.edu/AM215/sandbox`. You can clone this repository using the `git clone <URL>` command. You can use the URL provided above or visit the website in your browser and click on the green "Code" button in the top right to learn about other options.

It is important that you do not clone this repository inside your private Git repository or any other Git repository for that matter. This would create *nested* repositories which you want to avoid. If you have followed the suggested directory structure mentioned earlier, you could change into the `git` directory and issue the `git clone` command in there to arrive at something that looks like the following

```
classes/
|-- AM215
|   |-- git
|   |   |-- myrepo <- your private Git repository
|   |   |-- sandbox <- the cloned sandbox for this problem
|   |   \-- main
|   \-- AM215_other_data
\-- CS2050 (some other class in this directory)
```

Cloning the repository will automatically *checkout* the default branch and you will be able to see the files associated with that branch locally. The problem is that you only have read access for this repository and you can therefore only clone or pull from it but not contribute your own changes via push for example. There are two options to resolve this:

1. Have the repository owner give you push access. This is often not desired if you are not a main contributor or developer for the project.
2. Create your own *fork* of the repository and make a pull request (PR) back to the original repository when you have made all the changes. This is the approach that you will take in this assignment. It is the workflow used on platforms like GitHub and you are using it to submit your homework as well. The difference for your homework is that you create the PR *within your own repository*, while in practice PR's are *between* repositories (either branches with the same name or differently named branches). You should be familiar with this workflow.

In order to proceed with a *fork* of `https://code.harvard.edu/AM215/sandbox` you must remove your clone from before first. Visit the URL above and click on the "fork" button in the top right corner. Fork the repository to your own account and create a screenshot called `P2_fork.png` of the landing page (you should see "forked from AM215/sandbox" in the top left corner).

Clone your *forked* repository in the same manner as you did before for the read-only `AM215/sandbox` repository (which you have deleted again). Save the screenshot in `hw1/sandbox` in the forked repository, add it using `git add`, commit it using `git commit -m "Add forked screenshot"` and push the commit with `git push`.

b) ***25 points***

One of the most fundamental concepts in Git is the concept of a branch. A branch is simply a sequence of commits, where the branch is referenced by the most recent commit in this sequence. Branches *can therefore change their reference* over time, while the reference for a commit is *immutable*.

Branches are the place where you develop and throw things around. You should never do that in your default branch which is usually accessible to the public. Creation of branches is *cheap* in Git and at its core philosophy.

i) You can check a list of branches with

```
$ git branch -a
```

You can see there is a `test` branch in the forked repository. You can switch branches in Git using

```
$ git checkout # works also in Git 2.23 and later
$ git switch   # only for Git 2.23 and later
```

Checkout the `test` branch. You can inspect the *differences* on this branch using the command

```
$ git diff main..HEAD
```

where here `main` is the default branch (the one you were before) and `HEAD` is a shorthand for the currently checked out commit (you could also omit it in the command above). Spend some time figuring out what the differences are, you can also use other commands to navigate around and list files, for example `ls`. It

is perfectly valid that branches may contain very different files or even have files missing. Most often branches will look somewhat similar except for the features they implement.

ii) Next you will create your own branch. First switch back to the default branch with

```
$ git switch main
```

We did this step because we want to create the new branch based off of the default branch. Naturally, this can be any valid branch. Name the new branch "hw1p2". You can create the branch with

```
$ git branch hw1p2
```

and switch to it with

```
$ git switch hw1p2
```

Because this is such a common procedure, the two operations can be combined with either

```
$ git checkout -b hw1p2
```

or

```
$ git switch -c hw1p2
```

Create one more branch named hw1p2_tmp and switch to it.

iii) You should now be on the branch hw1p2_tmp. Change into the hw1/sandbox directory and edit the README.md file. Add your name at the very bottom of the file. Save and close the file. You can use your text editor of choice.
To check modifications on the current branch you use the `git status` command. *This is a very important command and you should use it often for orientation.* After your modifications you should see

```
$ git status # executed on repository root
On branch hw1p2_tmp
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   hw1/sandbox/README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

The changes have not been *staged* yet. This process adds the changes that you want to *commit* next to the so called *index*. You can *add* changes to the index using the `git add` command. Add the changes you made in README.md to the index. After this operation `git status` should report

```
$ git status # executed on repository root
On branch hw1p2_tmp
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   hw1/sandbox/README.md
```

You can continue to make changes to files (including `README.md`) and repeat the process if you want it to be committed next. Only the currently staged changes will be committed. Which is what you do now. Type `git commit`.
This will open up the default editor (`vim` on most systems) to edit the *commit message*. You can set the `EDITOR` (globally) or `GIT_EDITOR` (only for Git) environment variables if you want to change the editor for future sessions. You are most likely in `vim` now. To enter your commit message type "`i`" to enter *insert mode*. Now you can type your commit message. Press the "`esc`" key to enter *normal mode* again where you can save and exit by typing "`:wq`" (`:` is to enter *command mode*, `w` is for *write* and `q` is for *quit*).
So far we have used `git commit -m 'Commit message'` to bypass the editor if the commit message *is only small*, i. e., small changes. In general, your commit messages must be concise and contain enough information to describe the changes in the commit. Writing is generally a hard task, so is writing good commit messages.
If you now type `git status` again you will see that there are no other changes and you can proceed with the next logical unit that should eventually become a commit. Note that you can also add *partial* changes in a file to the index. See the `--patch` option in `git help add`.

iv) You now have a local commit that has not been pushed to the remote repository of your sandbox fork. You can accomplish this with `git push`. This command will fail because you are working on the `hw1p2_tmp` branch and Git does not know what to do with it because you have not specified enough argument to the push command. You must be more specific by telling Git which *remote* and which *branch* you want to push

```
$ git push origin hw1p2_tmp
```

If you know you will push frequently on this branch because it is a hot development, add the `-u` option such that Git will remember what to do when you are on this branch. If you add this option you can just use `git push` later on. If you want to know how local branches relate to remote branches you can do so with

```
$ git branch -vv
```

v) Recall what you did up to this point. You have modified the `README.md` file by adding your name at the bottom. We now simulate something that can not be avoided in distributed version control and therefore must be expected when working with Git. Although Git is good at resolving this issue autonomously, it will ask for your help when it gets stuck. Switch back to your `hw1p2` branch that you have created earlier. Open again the `README.md` file. You will notice that your name

has disappeared from the file because you have not added it on this *branch*. Now update this `README.md` file in *the same place* where you added your name before (at the bottom of the file). Write something creative other than your name, then add and commit these changes similar to before (you do not need to push).
At some point in the development you want to *merge* branches together again (e. g. merge a working feature into the default branch). For this you use the `git merge` command. We now want to merge our changes on the `hw1p2_tmp` into our current branch (`hw1p2`). To do this type

```
$ git merge hw1p2_tmp
```

This merge will result in a merge conflict because the two `README.md` files contain *different* changes at the same place and Git does not know which it should favor. You should see the following error

```
$ git merge hw1p2_tmp
Auto-merging hw1/sandbox/README.md
CONFLICT (content): Merge conflict in hw1/sandbox/README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Check the status with `git status`. You have to resolve the conflict manually in this case. Open the conflicting `README.md` file in your editor and take a screenshot. Save the screenshot in the same directory as the `README.md` file and name it `P2_merge.png`. You should see the following

```
# AM215 Homework 1 Sandbox

Fall 2025: Fork, pull requests, merge conflicts
<<<<<<< HEAD

This text will cause a conflict
||||||| e55df4c
=======

Ignacio
>>>>>>> hw1p2_tmp
```

Git has marked the conflicting sections. The markers mean the following

- "`<<<<<<< HEAD`" marks the beginning of the conflicting section in the *current branch* (`HEAD`)
- "`||||||| e55df4c`" marks the end of the conflicting section in current branch and marks the beginning of the content in the *common ancestor* of the two branches that are merged. The reference of the common ancestor is further indicated.
- "`=======`" marks the end of the content in the common ancestor and the beginning of the conflicting section in the branch that is being *merged in*. Note that the content in the common ancestor is empty because none of the code we added on either branch existed (we appended at the end of the file).

- "`>>>>>>> hw1p2_tmp`" marks the end of the conflicting section on the other branch.

By manually resolving a merge conflict you must read and understand the changes on both sides and possibly contact the author of either code to further consolidate. You are free to edit this file. You can pick either change or write something completely new. For example

```
# AM215 Homework 1 Sandbox

Fall 2025: Fork, pull requests, merge conflicts

OK, the merge conflict is resolved now.
```

Save the changes and exit the editor. To finalize the merge conflict you still need to *add* your resolution to the index (check with `git status`).

```
$ git add hw1/sandbox/README.md
```

and finally type `git commit` to conclude the merge. You will notice that Git has automatically added a commit message for you. You can use this suggestion or edit further for more content. Save and exit the editor. The merge conflict is now resolved. After merging a branch, it is good practice to *delete* the branch as it is now fully contained in the target branch. You can use `git branch -d hw1p2_tmp` to do this. You can delete the *remote* branch using `git push origin --delete hw1p2_tmp`.

You can also add and commit the `P2_merge.png` file if you have not done so yet. Push the changes to the remote.

vi) You should still be on the `hw1p2` branch. Make one more change to the `README.md` file (e. g. add another line of Markdown) then add, commit and push the change. Since commits are *immutable* you cannot undo that commit.

You can however *revert* a commit using the `git revert` command. In order to do this you need the reference to the commit (its SHA1 hash). You can find this information using `git log` for example. The hash is a number of the form `d037e425`. You only need as many digits as necessary for Git to uniquely identify the commit. Since we want to revert the most recent commit, we could also use `HEAD` as a shortcut. You can inspect where `HEAD` is pointing to with

```
$ cat .git/HEAD
ref: refs/heads/hw1p2
$ cat .git/refs/heads/hw1p2
d037e425b44d779331fd86c23ba423520b649d3c
```

To revert the previous commit (that was already pushed to the remote) use

```
$ git revert HEAD
```

This will create a new commit with the changes undone. Check `git log` to inspect the new commit. Finally push the commit to the remote.
Note that reverting a commit necessarily creates a new commit in the history because commits are *immutable*. This was necessary in this case because the commit we reverted has *already been pushed to the remote*. This means the history is shared with other developers working with this remote. If you had not pushed, the commit would still be *local* to the history of the repository on your laptop. In that case you can actually remove the commit such that the "deletion" is not visible in the history using `git reset`. This process is also related to an *interactive rebase* which will be discussed further in the lecture.

c) ***15 points*** In the previous section we were working with branches. Because we have forked the repository, there is a relation ship between the fork and the original repository. These are *two different* repositories, however, and a relationship between the two has to be established through *remotes*. With this, you can update a local branch using commits on a possibly different branch maintained on the other remote (i. e. in the other repository).
You also want to maintain this relationship to ensure a smooth merging process when you request to pull a branch from your fork into the original repository. You could follow a similar strategy between your private Git repository and the main class repository, where new material is posted. This would allow to easily update your private repository with new material from the other remote.
In this task we want to add a new remote that points to the original sandbox repository in the AM215 organization, https://code.harvard.edu/AM215/sandbox. Since this repository is read-only, you can only fetch/pull from it. You can check your currently configured remotes with

```
$ git remote -v
origin  git@code.harvard.edu:<your NetID>/sandbox.git (fetch)
origin  git@code.harvard.edu:<your NetID>/sandbox.git (push)
```

You can add a new remote called `upstream` (you can name it whatever you like, `upstream` is often used) invoke the following command

```
$ git remote add upstream https://code.harvard.edu/AM215/sandbox
```

If you check your remotes you can verify that it was indeed added

```
$ git remote -v
origin  git@code.harvard.edu:<your NetID>/sandbox.git (fetch)
origin  git@code.harvard.edu:<your NetID>/sandbox.git (push)
upstream        https://code.harvard.edu/AM215/sandbox (fetch)
upstream        https://code.harvard.edu/AM215/sandbox (push)
```

Take a screenshot of this terminal output and name it P2_remote.png. Save the screenshot in hw1/sandbox while still on the `hw1p2` branch. Commit and push this addition.

You have now configured two remotes `origin` (obtained by cloning the forked repository) and `upstream` which points to the original repository. You can push and pull from the former but only pull from the latter. In order to push to `upstream` you must create a pull request. As discussed earlier, you can specify the remote and branch to `git push` and `git pull` to specify a specific destination or source.

You are almost done! The final step is to create a pull request to merge your `hw1p2` branch from your forked repository into the `main` branch in the original repository https://code.harvard.edu/AM215/sandbox. You can achieve this on the webpage of your forked repository. Make sure your PR contains all the steps we went through in this problem. If you did it correct, you can see your PR appear in the https://code.harvard.edu/AM215/sandbox repository.

For completeness, copy your hw1/sandbox directory from this problem into the hw1/submission directory in your *private Git repository* for which you must issue another PR as described in the class syllabus.

Mathematical Modeling

Problem 3: Maximum Likelihood Estimation (*10 points*)

Deliverables:

1. A filename `P3a.md` or `P3a.pdf`
2. `P3b.sh` script with the following lines (exact order)
    i. Command for i)
    ii. Command for ii)
3. `P3c.py` script
4. `P3di.png` and `P3dii.png` plots
5. `P3d.txt` text file

In the following problem, suppose the World Series were best of 3 games. You will use your newly aquired software development techniques to help you solve this problem. Each file has to be commited and pushed (in the correct folder) to your remote repository in the AM215 organization.

Remember to commit frequently. You can always go back to a prevous commit if needed.

Assume that each game is a Bernoulli trial with probability p. Also, assume that we see n_0 events where the better team wins 0 games, n_1 events where the better team wins 1 game and n_2 events where the better team wins 2 games.

a) Following the derivation in class, derive the Likelihood of our observations. Using this result, analytically compute the maximum likelihood for p. Save your derivation in the files `P3a.md` or `P3a.pdf`.

   You must submit your answer using Markdown, Latex, or similar software. Images of handwritten notes, or other type of file will not be graded.

   Note: The algebra for this becomes complicated. You can solve this with any program you like, including Mathematica or Wolfram Alpha. Alternatively you will get full credit by simply setting up the problem in the correct way and detailing the main logical steps, without carrying out the algebra. Part of Applied Mathematics is knowing WHEN a calculation is to hard to do expeditiously!

b) The `p1-HW-WS_LoserWins.csv` file is provided in the `Homework_Files/Homework1` directory of the homework handout. It contains data that we have generated for 44 synthetic competitions. Each competition reports the number of games the losing team wins.

   Before proceeding with modelling, a careful Exploratory Data Analysis (EDA) must be made. This will guarantee that any derived models will have a solid foundation on reliable data.

This process can be performed on a Jupyter notebook or a Python script. In the abscense of any of that, you can always do serious work using the Linux Terminal. You will use only Terminal commands for the next question.

i) Count the number of lines containing relevant data and write the result to an output file called `out.txt`. Your command should assume that the file is in the current directory. For example, if the provided text file contained:

```
a
1
b
0
```

Then the resulting command should produce the following file.

```
$ cat out.txt
2
```

ii) Count the number of lines containing wins and the number of lines containing looses. Write the result to an output file called `out.txt`. For example, if the provided text file contained:

```
a
1
b
0
1
```

Then the resulting command should produce the following file.

```
$ cat out.txt
0 1
1 2
```

c) A critical aspect of any simulation is the comparison with a baseline.

i) From the analyzed data, use the derivation obtained in a) to compute your estimate for p.

ii) Following the code we used in class, carry out a numerical calculation of the maximum likelihood estimate for p and compare to the analytical derivation. Write your code on `P3c.py`. This script must accept the *relative path* of the data as first argument.
The program must output the numerical estimation of p, the value for the analytical derivation, and the percentage error. Assuming the analytical value is the ground truth.

d) Now you should be able to run a simulation with any dataset that complies with the input of your program.

i) Run **a** ~~the~~ **synthetic** 44-game World Series and plot the distribution of times that the losing team wins using Mosteller's number $p = 0.65$. **Submit your plot as P3di.png.**

ii) Use the same program and run it 4400 times. Plot the same distribution of the previous question. **Submit your plot as P3dii.png.**

iii) Does the fraction of games that the losing team won change significantly by changing the number of Series played? Were Mosteller's measurements for the first fifty years of the twentieth century lucky? Answer each question on `P3d.txt`.

Problem 4: Adding more features to compute p (*10 points*)

Deliverables:

1. A filename `P4a.txt`
2. A filename `P4c.png`
3. A filename `P4d.txt`

In Problem 3, you estimated the probability p of the better team winning the series. This value is key to the success of the model. Let's explore how additional features in the data could enhance the accuracy of p's prediction.

In Lecture 2, we introduced a dataset from Major League Baseball that includes a set of features about teams that go beyond games won and lost, including strikeouts, double plays and so forth. The goal of this problem is to create a simple model that predicts the probability a team wins the world series.

a) Explain what the code in the notebook does in `P4a.txt`. Include in your explanation the following concepts:

   1. What is the point of the dataset split into train and test?
   2. Why is it important to handle missing values from the daata?
   3. What is the logistic regression fit predicting? Why is an appropiate model for this problem?
   4. Explain the metrix used in the notebook: accuracy, confusion matrix, precision, and recall?

b) The model inside the Jupyter Notebooks performs adequately. But, it is ready to be moved to a production environment? Move all the code into a Python script. The code should be composed of functions that you can call inside your Jupyter notebook. You can do multiple commits while working. But each commit must modify only one function.

c) A solid model and quality source code is a good starting point. Before iterating to the next and more complex model, you should learn as much as possible from the current model.

   In order to do this, it is always paramount to analyze the feature importance, to correlate it to our knowledge or the one established on the literature. Create and explain the feature importance plot, identifying the features that influence the probability of the better team winning. Save it into `P4c.png`. Does it make sense?

d) With a trained model and a solid understanding of it, now you are in good position to improve on it. One option is to design new features in order to improve the model's performance. This is called feature engineering. Invent new features and find out if they are more informative for predictions.

   Hint: *think about some of the stats you hear about when people discuss baseball, e.g. batting average, winning streaks, run differentials, etc.*

   It is always critical to avoid modifying the existing working code/models. For this part, you will have to create a new branch `dev`, make your modifications/improvements, merge

the code back into the existing branch and then delete `dev`. Your commit structure should reflect the merge.

In `P4d.txt`, explain why you've chosen these features and transformations and see if the turn out to be important and how could you use this type of analysis to improve Mostellar's model. Implement your ideas following a similar work structure of this question.

Problem 5: Simulating a tournament (*10 points*)

Deliverables:

1. Filename `P5a.py`
2. Filename `P5b.csv`
3. Filename `P5d.txt`

Consider a tournament with 128 teams. You are given a draw. You are also given the previous games that the eight teams have played against each other in the file

`game_results_final_10000_N128.csv`

a) Write a Python function to estimate p_{ij} from the previous games. Commit it as `P5a.py`. Where p represents the probability that team i will beat team j in a given game.

b) Given your estimates of p_{ij}, carry out simulations of the draw. For each team, predict the probability that it reaches the round of 64, the round of 32, the round of 16, the quarter-finals, semi-finals; the finals and is the tournament champion. Thus you should create a matrix W_{ij}, where the first index labels the team, and the second index is the probability the team makes the requisite round: W_{i0} gives the round of 64; W_{i1} gives the round of 32; W_{i2} gives the round of 16; W_{i3} gives the quarter-finals; W_{i4} gives the semi finals; W_{i5} is the probability that the team makes the finals and W_{i6} is the probability that the team makes the finals. Save and commit your results in `P5b.csv`.

c) Upload your predictions onto our AM215 Kaggle competition 1 website. We will compare your predictions to the true answer and rank the entries. Note that in this case we know the true answer because we *synthesized* the games for an assumed P_{ij} matrix, and we can thus simulate the tournament probabilities with this matrix.

d) In the first project, one of the directions you can choose is to repeat this exercise but for a REAL sports competition with data – eg the US Open, March Madness, NFL Season, and so forth. Write a brief summary of what you might do when you do this for real, and list some of the challenges. Commit it as `P5d.txt`