Indian Institute of Technology Kanpur
CS771 Introduction to Machine Learning

Instructor: Purushottam Kar
Date: September 28, 2022
Total: 100 marks

ASSIGNMENT 2

1 What should I submit, where should I submit and by when?

Your submission for this assignment will be one PDF (.pdf) file and one zip (.zip) file. Instructions on how to prepare and submit these files is given below.

Assignment Package:
`https://web.cse.iitk.ac.in/users/purushot/courses/ml/2022-23-a/material/assn2.zip`
Deadline for all submissions: 22 October, 2022 9:59PM IST
Code Submission: `https://forms.gle/idnd4QvHwUBPq3596`
Report Submission: on Gradescope
There is no provision for "late submission" for this assignment

1.1 How to submit the PDF report file

1. The PDF file must be submitted using Gradescope in the *group submission mode*. Note that this means that auditors may not make submissions to this assignment.
2. Make only one submission per assignment group on Gradescope, not one submission per student. Gradescope allows you to submit in groups - please use this feature to make a group submission.
3. Link all group members in each group submission you make. If you miss out on a group member while submitting, that member may end up getting a zero since Gradescope will think that person never submitted anything.
4. You may overwrite your group's submission (submitting again on Gradescope simply overwrites the old submission) as many times as you want before the deadline. However, make sure you link your group members in each submission.
5. Do not submit Microsoft Word or text files. Prepare your report in PDF using the style file we have provided (instructions on formatting given later).

1.2 How to submit the ZIP file

1. Password protect your ZIP file using a password with 8-10 characters. Use only alphanumeric characters (a-z A-Z 0-9) in your password. Do not use special characters, punctuation marks, whitespaces etc in your password. Name your ZIP file `submit.zip`. Specify the file name in the URL properly in the Google form.
2. Remember, your file is not under attack from hackers with access to supercomputers. This is just an added security measure so that even if someone guesses your submission URL, they cannot see your code immediately. A length 10 alphanumeric password (that does

not use dictionary phrases and is generated randomly e.g. 2x4kPh02V9) provides you with more than 55 bits of security. It would take more than 1 million years to go through all $> 2^{55}$ combinations at 1K combinations per second.

3. Make sure that the ZIP file does indeed unzip when used with that password (try `unzip -P your-password file.zip` on Linux platforms). If we are unable to unzip your file, you may lose marks.

4. Upload the password protected ZIP file to your IITK (CC or CSE) website (for CC, log on to `webhome.cc.iitk.ac.in`, for CSE, log on to `turing.cse.iitk.ac.in`).

5. Fill in the following Google form to tell us the exact path to the file as well as the password `https://forms.gle/idnd4QvHwUBPq3596`

6. Do not host your ZIP submission file on file-sharing services like Dropbox or Google drive. Host it on IITK servers only. We will be using a script to autodownload your submissions and if not careful, Dropbox or Google Drive URLs may send us an HTML page (instead of your submission) when we try to autodownload your file. Thus, it is best to host your code submission file locally within IITK servers.

7. While filling in the form, you have to provide us with the password to your ZIP file in a designated area. Write just the password in that area. For example, do not write "Password: helloworld" in that area if your password is "helloworld". Instead, simply write "helloworld" (without the quotes) in that area. Remember that your password should contain only alphabets and numerals, no spaces, special or punctuation characters.

8. While filling the form, give the complete URL to the file, not just to the directory that contains that file. The URL should contain the filename as well.

   (a) Example of a proper URL:
   `https://web.cse.iitk.ac.in/users/purushot/mlassn2/submit.zip`

   (b) Example of an improper URL (file name missing):
   `https://web.cse.iitk.ac.in/users/purushot/mlassn2/`

   (c) Example of an improper URL (incomplete path):
   `https://web.cse.iitk.ac.in/users/purushot/`

9. We will use an automated script to download all your files. If your URL is malformed or incomplete, or if you have hosted the file in a way that cannot be downloaded, then your group will either lose a lot of marks or else get a straight zero in these cases.

10. Make sure you fill in the Google form with your file link before the deadline. We will close the form at the deadline.

11. Make sure that your ZIP file is actually available at that link at the time of the deadline. We will run a script to automatically download these files after the deadline is over. If your file is missing, we will assign your group zero marks.

12. We will entertain no submissions over email, Piazza etc. All submissions must take place before the stipulated deadline over the Gradescope and the Google form. The PDF file must be submitted on Gradescope at or before the deadline and the ZIP file must be available at the link specified on the Google form at or before the deadline.

Problem 2.1 (The Code Corrector). The problem of automated program repair is quite interesting since it has pedagogical value for students who are beginning to learn programming. Given a program in (say the C language) that fails to compile, we want to suggest repairs to the student who wrote that code. Such a tool can lessen the load of teaching assistants and make the learning experience more enjoyable for students. However, before we can perform repair, we need to find out what sort of error is present in the program. Although the compiler often does give some hints as to what sort of error might be present, these are often insufficient. Thus, in this assignment, we will try to solve this problem using machine learning techniques.

Data Points. Each data point corresponds to a single line in the code (C programming language) that contained the error (for sake of simplicity, we are only considering simple cases where the error was localized in a single line of the code – correcting that one line would make the program compile-worthy). A line of code is represented as a sequence of tokens e.g. `for(j = 0, j < N, j++)` which consists of 13 tokens namely `for`, `(`, `j`, `=`, `0`, `,`, `j`, `<`, `N`, `,`, `j`, `++`, and `)`, in that order.

However, since variable names (identifiers) like `j`, `N`, and constant values (literals) like `0` do not matter when correcting single line errors (they may matter when correcting long-range errors like variable name mistakes, scoping issues etc), we abstract out lines by replacing identifiers and literals by stand-in tokens as follows

```
for(TypeKind.INT = TokenKind.LITERAL_INT,TypeKind.INT < TypeKind.INT, TypeKind.INT++)
```

This gives us a dictionary of tokens consisting of reserved keywords e.g. `for`, stand-in tokens e.g. `TypeKind.INT` and punctuation marks and operators e.g. `,`, `++`, `(` etc. The dictionary for this assignment contains 225 different tokens and is given to you as a part of the assignment package in a file called `dict` (it is not compulsory to use this file in your solution – we just gave it in case you are curious or if it helps in some way).

We would like to caution you that the dictionary contains some junk tokens as well. Despite our best efforts, some of these programs were so buggy that our tokenizer broke! Thankfully, these junk tokens are rare and should not affect your training. The dictionary also contains a special `UNK` token (short for "unknown") that is popularly used as a catch-all replacement for extremely rare and uninformative tokens.

We then create a bag-of-words (BoW) representations for the line by counting the number of occurrences of each dictionary token in that line. For example, for the above line, the BoW representation would look something like the following

`for`: 1
`(`: 1
`TypeKind.INT`: 4
`=`: 1
`TokenKind.LITERAL_INT`: 1
`,`: 2
`<`: 1
`++`: 1
`)`: 1

since the token `TypeKind.INT` appears four times in the line whereas the token `<` appears only once in the line etc. Since we have 225 dictionary tokens, each BoW vector is represented as a $d = 225$ dimensional vector. However, this vector is usually very sparse since most lines contain only a few unique tokens. You are given the BoW representations for 10000 erroneous lines in the file `train` in the assignment package. Note that you are not given the original token sequences, just the BoW representations.

Classes. There are a total of $C = 50$ error classes which we are considering in this assignment. The identity of these error classes is given to you as a part of the assignment package in a file called `error_codes` (again, it is not compulsory to use this file in your solution – we just gave it in case you are curious or if it helps in some way). Each line corresponds to exactly one error code. We used the C language and the Clang compiler to generate this data. The error class associated with each line is also given in the file `train`. Please see the file `eval.py` to see how to load data to obtain the error class and the BoW representations for the 10000 training examples.

Caution: in the data, the error classes are numbered 1 to 50 and not 0 to 49. Also, please note that error class numbers 33, 36, 38 never appear in the training set and will never appear in our secret test data either. These error classes were too rare and were eliminated from this assignment.

Your Task. Your task is to learn a multi-class classification model which, when given the BoW representation of a test line, can tell us the error class associated with that line. Please note that you do not have access to the exact code line – you only have access to the BoW encoding of that line. However, your model is allowed to output 5 error classes which it thinks may be the correct error class. Given a test line, your method should return a vector `pred` of length 5 with each entry between 1 and 50. The first entry of the vector i.e. `pred`[0] will be interpreted by us as the error class your method thinks is most likely to be the correct error class for that line, the second entry of the vector i.e. `pred`[1] is the next most likely error class, and so on. Thus, if you think the most suitable error class for the line is the ninth error class, you must set `pred`[0] = 9 (error classes are numbered 1 to 50 and not 0 to 49). Thus, we expect a *ranking* of the 5 most suitable error classes for that line. Please do not repeat error classes in your prediction vector e.g. by setting `pred`[0] = `pred`[1] = ... Our evaluation code will automatically remove duplicates from your predictions (see `eval.py` and `utils.py`).

Evaluation Measures. We will evaluate your method on our secret test data (which will have exactly the same value of $d = 225$ tokens and $C = 50$ classes as the training data) using two kinds of performance measures described below

1. Precision at k (prec@k): For this, we will first choose some $k \in [5]$. Then we will ask, for every test line, if the correct error class is somewhere in the top k predictions given by your method. If yes, then we award a score of 1 else a score of 0. Taking the average of this score across all test lines will give us prec@k of your method.

2. Macro Precision at k (mprec@k): For this, first choose some $k \in [5]$. Then, we will go over each of the error classes $j \in [C]$ and for each error class, look at all the test lines that belonged to that error class. Then we will calculate the fraction of these lines for whom your method did predict the error class j in its top k predictions. This will be a fractional number $\in [0, 1]$. Taking the average of this number across all error classes will give us mprec@k of your method.

The difference between prec@k and mprec@k largely arise due to the presence of *rare error classes* in the dataset i.e. errors that made more rarely by programmers. You will see in your data itself that an average error class is associated with just $\hat{n} = 200$ lines whereas there are a total of $n = 10000$ lines! Whereas a method can get very high prec@k by just predicting popular error classes in every test scenario, such a method may do poorly on mprec@k which gives a high score only if that method pays good attention to all error classes, not just the popular ones.

Your Data. You have been provided in the assignment package, training data for $n = 10000$ lines, each line i is represented as a $d = 225$ dimensional BoW feature vector $\mathbf{x}^i$. The feature vectors are sparse and the average number of non-zero features in a training feature vector is only $\hat{d} \approx 8$. There are a total of $C = 50$ error classes and each line is associated with a single error class $y^i \in [C]$. Routines have also been provided to you (see `utils.py`) that read the data as well as show you how we will perform evaluation for your submitted code using prec@k and mprec@k.

Caution: all matrices in this problem (feature and label) are sparse and should be stored in compressed representation (`csr_matrix` from `scipy`). The data loading routine given to you as a part of the assignment package will return compressed matrices. Be careful about densifying these matrices (e.g. using `toarray()` from `scipy`) as you may run into memory issues.

1. Suggest a method that you would use to solve the problem. Describe your method in great detail including any processing you do on the features, what classifier(s) you use to obtain the final predictions, what hyperparameters you had to tune to get the best performance, how you tuned those hyperparameters (among what set did you search for the best hyperparameter and how) e.g. you may say that we tuned the depth of a decision tree and I tried 5 depths $\{2, 3, 4, 5\}$ and found 3 to be the best using held out validation.

2. Discuss at least two advantages of your method over some other approaches you may have considered. Discuss at least two disadvantages of your method compared to some other method (which either you tried or wanted to try but could not). Advantages and disadvantages may be presented in terms of prediction, macro precision, training time, prediction time, model size, ease of coding and deployment etc.

3. Train your chosen method on the train data we have provided (using any validation technique you feel is good and also splitting the train data into validation split(s) in any way you wish). Store the model you obtained in the form of any number of binary/text/pickled/compressed files as is convenient and write a prediction method in Python in the file `predict.py` that can take a new data point and use the model files you have stored to make predictions on that data point. Include all the model files, the `predict.py` file, as well as any library files that may be required to run the prediction code, in your ZIP submission. You are allowed to have subdirectories within the archive.

   Caution: The two files `eval.py` and `utils.py` are a part of the evaluation script. Do not submit these with your submission. Also, do not use these filenames as a part of your own code. If you want to create a utility file, name it something else e.g. `my_utils.py` etc. If we find files named `eval.py` or `utils.py` in your submission, **we will overwrite them with our version of these files**.

(40+10+50 marks)

Suggested Methods. The following techniques are known to succeed in various forms on problems with a large number of classes. Since your features are rather raw, you may want to normalize your features suitably first.

1. LwP: in literature this is often used as a *reranking* step (reference [31] in the repository) but can be used as an algorithm in its own right as well. This involves learning one or more prototypes per error class (e.g. mean of feature vectors of all data points of a class can serve as a prototype for that class, or else multiple prototypes can be obtained by clustering data points of that class).

2. OvA: in literature this is also known as *binary relevance* (reference [32, 33, 34] in the repository) and this involves learning a binary classifier to predict for every error class, whether a line has that error or not. One may alternatively, use logistic regression to predict a probability score per error class and rank error classes using that score.

3. DT: two main types of decision trees are popular in literature for multi-label/multi-class problems

   (a) The first kind of approaches (examples are references [02, 31, 41] in the repository) use the decision tree to take a data point to a leaf where the labels of training data points who also reached that leaf, are used to perform prediction

   (b) The second kind of approaches (examples are references [04, 36] in the repository) use the decision tree to merely eliminate which error classes are definitely not relevant to that data point. An OvA-style classifier or ranker is then used to decide among the remaining error classes, which should be predicted.

Apart from this there are a few other methods such as output codes, dimensionality reduction (aka embedding) methods, and deep learning methods, are also popular. Several references are available on the repository for all such techniques (although code may not be readily available for all of them). For example, the following repository offers the state of the art in prediction with a large number of classes with papers that describe various methods that do well on such problems, as well as readily available code for some of these methods. We understand that the problem in this assignment is that of multi-class learning and not multi-label learning and the repository seems to be focused on multi-label learning. However, multi-label learning methods readily work (and are indeed popular) for multi-class problems as well.
`manikvarma.org/downloads/XC/XMLRepository.html`

Some Observations. You may use these to guide your exploration. DT methods usually offer fast prediction as well as good precision. However, their model size is usually large too. OvA methods usually offer very high precision values. However, they are usually slow at prediction. LwP methods usually offer high mprec@k since they do well on rare error classes. However, they are usually slow at prediction as well. It is notable however, that these are broad observations and need not apply to every single method in these families. For instance, DT techniques of the second kind (that prepare a shortlist of error classes) can use LwP methods in every leaf within the shortlisted labels to obtain reasonable prediction speed as well as good accuracies on rare labels.

Code Usage from the Internet. You have full freedom to download any code available online to perform training. This is allowed (without any penalty). Thus, you are being given full freedom to use code written by others and there will be absolutely no penalty even if you (are able to) directly use someone's code without making any changes to it. However **you must cite all code that you take from others** by giving names of authors and name of the URLs from where you obtained said code. There is no penalty for using code for which you have cited the original author. However, there will be heavy penalty for using someone's work and not giving them credit for it.

Marking Scheme. Parts 1-2 need to be answered in the PDF file itself and part 3 needs to be answered in the ZIP file submission. The 50 marks for part 3 will be awarded as follows:
Total size of the submission, once unzipped (smaller is better): 10 marks
Total time taken to return top-5 recommendations for all test users (smaller is better): 10 marks

prec@1,3,5 (larger is better): 3 x 5 = 15 marks
mprec@1,3,5 (larger is better): 3 x 5 = 15 marks

2 How to Prepare the PDF File

Use the following style file to prepare your report.
`https://media.neurips.cc/Conferences/NeurIPS2022/Styles/neurips_2022.sty`

For an example file and instructions, please refer to the following files
`https://media.neurips.cc/Conferences/NeurIPS2022/Styles/neurips_2022.tex`
`https://media.neurips.cc/Conferences/NeurIPS2022/Styles/neurips_2022.pdf`

You must use the following command in the preamble

```
\usepackage[preprint]{neurips_2022}
```

instead of `\usepackage{neurips_2022}` as the example file currently uses. Use proper LaTeX commands to neatly typeset your responses to the various parts of the problem. Use neat math expressions to typeset your derivations. Remember that all parts of the question need to be answered in the PDF file. All plots must be generated electronically - no hand-drawn plots would be accepted. All plots must have axes titles and a legend indicating what the plotted quantities are. Insert the plot into the PDF file using proper LaTeX `\includegraphics` commands.

3 How to Prepare the ZIP File

Your submission ZIP archive must contain a file called `predict.py` which we will call to get recommendations for test data points. The assignment package contains a sample submission file `sample_submit.zip` which shows you how you must structure your submission as well as showing you the way in which your prediction code `predict.py` must accept input and return recommendations. Do not change this input output behavior otherwise our autograder will not be able to process your recommendations and award you zero marks.

1. Your ZIP archive itself must contain a python file named `predict.py` which should not be contained inside any other directory within the archive. Apart from this file your ZIP archive may contain any number of files or even directories and subdirectories.

2. Do not change the name `predict.py` to anything else. The file must contain a method called `findErrorClass` which must take in the test features as a `csr_matrix` and a value $k > 0$ and return k suggestions for each test data point.

3. There are no "non-editable regions" in any of the files for this assignment. So long as you preserve the input output behavior, feel free to change the structure of the file as you wish. However, do not change the name of the file `predict.py` to anything else. Also, do not change the name of the *main method* `findErrorClass` to anything else.

4. Code you download from the internet may be in C/C++. You are free to perform training in C/C++ etc. However, your prediction code must be a python file called `predict.py`. This file may internally call C/C++ code but it is your job to manage that using extension modules etc. Do not expect us to call anything other than the single file `predict.py`

5. The assignment package also contains files called `eval.py` and `utils.py` (inside the ZIP archive `sample_submit.zip`) to show you how we will evaluate the code and model that you submit. Before submitting your code, make sure to run `eval.py` and confirm that there are no errors etc.

6. Do not submit `eval.py` or `utils.py` to us – we already have these files with us. We have given you access to these two files just to show you how we would be evaluating your code. If we find files named `eval.py` or `utils.py` in your submission, **we will overwrite them with our version of these files**. If you want to create a utility file, name it something else e.g. `my_utils.py` etc.

7. Make sure that running `predict.py` does not require us to install any Python libraries which are not available using `pip`. Libraries that are installable using `pip` are fine. However, if your prediction routine requires using external code not available using `pip`, you must supply all that external code to us within your ZIP archive – we should not have to install it. To be sure, we will not install or download any libraries from places like GitHub, personal homepages, or other repositories (even the repository mentioned in the question itself) even if you want us to. Your archive itself should contain all such files.

8. Other than the above restrictions, you are free to import libraries as you wish,e.g. sklearn, scipy, pandas. Files for libraries not available on `pip` must be provided by you.

9. Once we have installed `pip` libraries required by your code, we should just be able to call `predict.py` and get predictions i.e. your prediction code must be self contained. We should not have to create any new directories, move files around or rename them. All of that must be done by you while creating the archive (remember, you are allowed to have as many files and (sub)directories apart from the file `predict.py` as you want).

10. We do not need your training code for part 3. We also do not need code for any other methods that you tried and which did not work. We simply need your prediction code for part 3. Do not increase your submission file size (remember there are marks for submission size too) by including unnecessary files.

11. The system on which we will run your prediction code will **not have a GPU**. Make sure your code can run simply on a CPU. Thus, use the CPU version of deep learning libraries if you are using them and not the GPU version of those libraries.

12. You may use a GPU (if you can find one) to train your method but your prediction method must still work on a CPU. To be sure, we will not offer your prediction code a GPU even if you ask us to. Do not use any libraries that require GPU access during prediction. There are no restrictions on using GPU for training but we will not provide your code GPU access during prediction.

13. We will test your submitted code on a secret dataset which would have the same features per data point as well as same number of error classes.