Handed out: 24 Jan 2024 **Due:** 14 Feb 2024 (11:59 PM)

You are to work in groups of three for the guided project. Compress your code and answers into a single zipped file and submit it on Canvas. Only ONE submission is required from each group. In your submission, please clearly write the names and IDs of every member. Please ensure that your code actually runs; the TA cannot give you any points if your code does not run. If a piece of your code is not working, clearly state so in your submission.

In this guided project, you will develop a logistic regression model, through a series of steps, to classify SMS messages as *spam* or *ham* (i.e., not spam).

Please back up your code as you work through the project (preferably on another disk drive). Please do **NOT** post your code for this and all other course assignments *publicly* online (e.g., on GitHub or Google Drive) to avoid unwittingly (or wilfully) facilitating plagiarism (see Week 1's handout "*wk1 admin.pdf*" for the consequences of cheating).

Download `MachineLearningCourse.zip` from Piazza and unzip it into a folder. For the ease of exposition, I shall assume that the zipped file is uncompressed into the folder `mydir` and `MachineLearningCourse/` is the only item in it. I suggest you spend some time exploring the the contents of `mydir/MachineLearningCourse/` to familiarize yourself with the file structure of the project. You can peruse the SMS data (originally from a Kaggle competition) in the folder `mydir/MachineLearningCourse/MLProjectSupport/SMSSpam/dataset/`. (Here, I assume you are using a Mac or Linux machine that uses the forward slash `/` as the file separator. If you have a Windows machine, please use the backslash \ as the file separator instead. The same applies to any file directory specified in the project's Python code.)

You may use any Python integrated development environment (IDE). (You can even write your Python code in a good old text editor if you are gung-ho enough. Who needs syntax highligting and convenient debugging anyway?) I use the PyCharm IDE (`https://www.jetbrains.com/pycharm/`; you can use the professional version for free when you register on that website with your NUS student email account.) Again, for the ease of exposition, I assume that you are using PyCharm. After installing and launching PyCharm, open the directory `mydir/` (and *not* `MachineLearningCourse/`) in it. (The code should still run even if you open `MachineLearningCourse/` instead because I have added some preamble code in some of the project's Python files to guard against this. But if you do so, PyCharm may warn you of some non-existing errors.) If you are using another IDE or (heaven forbid) developing straight in a terminal or shell, you may wish to set your `PYTHONPATH` environment variable to include the full path to `mydir/`. (If you are a Python aficionado, please forgive me for belaboring the point on the seemingly simple task of merely opening the project and pointing the Python interpreter on where to look for the Python files. I have witnessed supposedly tech-savvy individuals spending eons figuring out how to accomplish that task because they do not know how to open a project correctly. No joke!)

You need to install the Python packages `Pillow` (for reading and creating of images) and `matplotlib` (for plotting graphs). To install the packages in PyCharm, you may refer to `https://www.jetbrains.com/help/pycharm/installing-uninstalling-and-upgrading-packages.html`. If you are working from a terminal, then you can install the packages with `pip` by using the following command: `pip install Pillow matplotlib`.

Finally, please note that you are ultimately responsible for setting up your own Python environment.

Question 1. (0 point)

This question is worth a whopping zero point because it is merely meant to help you verify that you have set up your Python environment correctly. Look in the `mydir/MachineLearningCourse/Assignments/Module01` directory (I will simply call this `Module01` henceforth). Open the Python file `Framework-1-GettingStarted.py`. This file contains a naive model that always predicts the majority class of a dataset (i.e., it always predicts an SMS as being not spam) and evaluates its accuracy. Run the code. If you get the output *"Predicting the most common class gives: 0.87 accuracy on the training set"*, all is well and you can move on to the next question. Before doing so, you should step through the code using a debugger (and even step into its methods) to explore the code structure. This will be time well spent because the structure is similar to those in subsequent questions.

Question 2. (1 point)

In `Module01`, open `Framework-2-Evaluations.py`. This file uses a (still naive) linear classifier that is slightly better than the majority-class classifier in the previous question. You do not have to modify this file. Instead, you have to amend the methods it uses in `EvaluateBinaryClassification.py` found in the `MachineLearningCourse/MLUtilities/Evaluations/` directory. You have to implement the functions that return the following five evaluation metrics.

- A confusion matrix of the types of mistakes the model makes
- Precision
- Recall
- False positive rate
- False negative rate

I suggest you get the confusion matrix right first because the other metrics can be computed with the confusion matrix. You have to get ALL five metrics correct to get 1 point for this question; getting any metric wrong would earn you zero point (this question is easy and is really meant as a warm up).

Please hand in your completed `EvaluateBinaryClassification.py`. Also submit a table tabulating the five metrics after training the linear classifier in `Framework-2-Evaluations.py` and testing it on the spam domain. Similarly, submit a table with the five metrics after evaluating the majority-class classifer `MostCommonClassModel` on the spam test set.

Question 3. (7 points)

Now, the real work begins and the kiddy gloves are off. In `Module01`, open the framework code `Framework-3-LogisticRegression.py`. Complete the implementation of `LogisticRegression` (found in `MachineLearningCourse/MLUtilities/Learner`) using the stub referenced from the framework. To get your logistic regression model working, you are expected to trace through the code and add in all the necessary code, e.g., you may have to amend `EvaluateBinaryProbabilityEstimate.py` too. (It is not that bad; you just have to patiently walk through the code with a debugger to know what the code is doing and where you need to add code.)

Note that the framework provides a small test case `runUnitTest` with some visualizations to help you debug. Use a threshold of 0.5 for classification (if the sigmoid function outputs a score ≥ 0.5 for an example, it is classified as 1 (spam), otherwise it is classified as 0). Use gradient descent for optimization as outlined in the framework code.

At the top of `Framework-3-LogisticRegression.py`, set `kOutputDirectory` to the directory where you wish to store your visualization files. (You will have to set the directory for subsequent questions too.)

(a) *(1 point)* After your logistic regression model works on the small test case `runUnitTest`, submit the visualization of the model learned on the unit test using 10 iterations of gradient descent, stepSize=1.0, convergence=0.005. (Please do not hand in the model weights!)

(b) *(2 points)* Submit your completed `LogisticRegression.py` and `EvaluateBinaryProbabilityEstimate.py`. Make sure your code works on the spam dataset and is clearly written. If the TAs have to do special mental gymnastics to find and check the key components (e.g., the gradient calculation in `__gradientDescentStep`, the logic in `incrementalFit`, and loss calculation), they will be miffed and deduct points. If your implementation is wrong here, the mistakes will cascade and result in point deductions in the subsequent questions. Long story short: get the code right in this step!

(c) *(1 point)* This and subsequent parts involve the spam dataset. Tune the hyperparameter `convergence` by trying [0.01, 0.001, 0.0001, 0.00001] (with `stepSize` of 1.0). Produce a table showing:
<convergence parameter>, <steps to convergence>, <validation set accuracy>
for each setting of the `convergence` hyperparameter. For this and subsequent parts, you may (of course) add code to the provided framework.

(d) *(1 point)* Describe in 3-4 sentences (not an essay, please!) your interpretation of the output of the hyperparameter sweep in (c). Include a justification for trying more values of the `convergence` hyperparameter or stopping there. If you think you should try more hyperparameters, what is the next value you would try? Why?

(e) *(2 point)* Produce a *single* plot showing the loss on the training set and the loss on the validation set every 100 steps during the run with the best convergence value you found in the parameter sweep and stepSize of 1.0. Describe the difference between the training and validation loss. Turn in the plot and your answer to following: On which set does the model have better loss? Why? (No more than 100 words, please.) If you need help plotting graphs, check out the MatPlotLib tutorials at `https://matplotlib.org/3.3.1/tutorials/index.html` (this is a good skill to learn).

Question 4. (6 points)

In this question, you will be processing and selecting features for the spam dataset. Open `Framework-4-FeatureSelection.py` and understand its flow. Then modify the `CreateVocabulary` method in `SMSSpamFeaturize.py` to do feature selection with:

- `numFrequentWords` $< N >$: the number of most frequent words to add to the vocabulary, and
- `numMutualInformationWords` $< N >$: the number of highest mutual information words to add to the vocabulary.

This will require you to change the method definition
`def CreateVocabulary(self, xTrainRaw, yTrainRaw, supplementalVocabularyWords=[]):`
to
`def CreateVocabulary(self, xTrainRaw, yTrainRaw, numFrequentWords=0, numMutualInformationWords=0, supplementalVocabularyWords=[]):`.

Next, you have to amend `CreateVocabulary` to call the provided stub functions `FindMostFrequentWords` and `FindTopWordsByMutualInformation` and implement the stub functions.

In the code that you add, please tokenize by splitting on whitespace and use smoothing when calculating the probabilities, i.e., $Prob = (\#observed + 1)/(totalsamples + 2)$.

You may amend `Framework-4-FeatureSelection.py` to answer the following questions.

(a) *(2 points)* Submit code for your implementation of `FindMostFrequentWords` and `FindTopWordsByMutualInformation` (and their supporting code). Make sure your code is clear and easy to grade! Also submit a document containing a list of the top 10 bag-of-word features selected by filtering by frequency and another list of the top 10 bag-of-word features selected by filtering by mutual information.

(b) *(1 point)* (For this and subsequent parts in Question 4, set the hyperparameters as follows: `stepSize = 1.0`, `convergence = 0.001`. Then train on the training set and evaluate on validation set of the spam dataset.)
Run your logistic regression model with the top 10 words by frequency. In a document, answer the following questions (1-3 sentences). Compare the accuracy of the model learned with the 10 most frequent words to the model that predicts the most common class. Increase the number of features selected by 5 until you outperform the most common class model. What number do you need?

(c) *(1 point)* Perform a parameter sweep on the number of features that your logistic regression model uses as selected by frequency, using `n = [1, 10, 20, 30, 40, 50]`. Produce and submit a *clearly labeled* plot with the number of features used on the x-axis, and the train and validation losses plotted on the y-axis.

(d) *(1 point)* Perform a parameter sweep on the number of features that your logistic regression model uses as selected by mutual information, using `n = [1, 10, 20, 30, 40, 50]`. Produce and submit a *clearly labeled* plot with the number of features used on the x-axis, and the train and validation losses plotted on the y-axis.

(e) *(1 point)* Provide a short answer (1-3 sentences) to the following: Which feature selection seems better based on the information you have? Why?

Question 5. (0 point)

You do not need to submit anything for this question (hence it is worth 0 point). The purpose of this question is to get you to look at some ROC curves and the code that generates them. Open `Framework-5-ROCAndOperatingPoints.py`, run the framework and look at the ROC curve that it outputs. Make sure to check out the `TabulateModelPerformanceForROC` helper function. The framework code uses your code from the previous questions (e.g., your logistic regression model and featurizer).

You can quantize the thresholds that are used in the framework to the nearest percent, e.g., you can use the thresholds in [0.0, 0.01, 0.02 ... 0.99, 1.0]. Consider the models produced by the framework code, and ask yourself:

- Which model is better at a 50% False positive rate?
- Which model is better at a 10% False positive rate?
- Which model is better at a 40% False negative rate?

Question 6. (0 point)

Again, this question is worth zero point because you need not submit anything. The purpose of this question is to familiarize you with error bounds and the code that generates them. Open `Framework-6-ComparingModels.py` and trace through the code. (The framework code depends on your code from the previous questions.) Also look at `MachineLearningCourse.MLUtilities.Evaluations.ErrorBounds` (especially its `GetAccuracyBounds` method). Please also take note of how cross validation is done with the help of `MachineLearningCourse.MLUtilities.Data.CrossValidation`.

Question 7. (6 points)

WARNING! This questions can take a lot of CPU runtime. Plan accordingly. If you try to do this question on the day before it is due, you will probably run out of time. (The tuning you will be doing mimics what really happens in real-world machine-learning engineering environments.)

In this question, you will produce the most accurate SMS spam filtering model you can using the code you have developed over the previous questions. Open `Framework-7-ModelTuning_SMSSpam.py` and understand its flow. You may modify this Python file to derive your best model.

In order to keep runtimes down, please use 2-fold (instead of 5-fold) cross validation on the training set (not validation set or test set) to carry out your hyperparameter search.

Start with the following hyperparameters values:

- `bestParameters['stepSize'] = 1.0`
- `bestParameters['convergence'] = 0.005`
- `bestParameters['numFrequentWords'] = 0`
- `bestParameters['numMutualInformationWords'] = 20`

Optimize one hyperparameter at a time by doing a parameter sweep.

1. For each sweep try at least 6 settings, ranging somewhat uniformly between 'probably too high' to 'probably too low', e.g.,
   - for the feature selection parameters, try varying by 20 (20, 40, 60, ...), and
   - for the gradient descent optimization parameters, try several small integers and several much smaller ones, like 0.01, 0.001, ... 0.00001.
2. For each sweep, produce the following.
   - A plot showing the values of the hyperparameter being optimized on the x-axis and (cross-validation) accuracy with 50% (2-sided) error bars on the y-axis.
   - A plot showing the values of the hyperparameter being optimized on the x-axis and the total runtime for the entire cross-validation run on y-axis (in seconds).
   - 2-3 sentences of interpretation answering: Were the hyperparameters you tried too high or low? Would you tune further? Which way?
3. After each sweep, update the hyperparameter value (for future sweeps) in a way that trades off runtime vs accuracy, so that:

- the value you pick is tied with the highest accuracy value according to a 75% 1-sided bound (or beats all others according to this bound), and
- the value you pick has the lowest runtime among these 'tied' values.

4. After optimizing each hyperparameter, evaluate the accuracy on the validation set by doing the following.
   - Produce a model with all the best hyperparameters you have found so far (training on the full training set).
   - Produce a plot with validation accuracy and 50% (2-sided) error bars on the y-axis and the number of optimization sweeps done to that point on the x-axis (one point per sweep you do; increasing the x value indicates further optimization).
5. Continute iterating to optimize hyperparameters until convergence by doing the following.
   - Start off by doing an initial round of sweeps so that each hyperparameter is optimized at least one time.
   - Continue doing sweeps based on your interpretation of the charts – choose the best hyperparameter and direction.
   - Stop when you can no longer significantly improve accuracy according to a 75% 1-sided bound

Evaluate on the test data by doing the following.

1. Evaluate the model learned with the best hyperparameters you found on the test data.
2. Evaluate the model learned with the initial hyperparameters on the test data.
3. Produce an ROC curve comparing these two models (the initial vs the best).

Submit a short report showing the following.

- *(3.0 points)* The plots you produce in Step 2 above for the first 4 steps of optimization (one sweep on each of the hyperparameters). Label clearly so the TA can follow the steps (and potentially reproduce them)!
- *(1.0 points)* The final plot you produce in step 4, showing how validation accuracy improves throughout the entire optimization run.
- *(1.0 points)* The final hyperparameters you learned.
- *(1.0 points)* The ROC plot you produced on the test set (comparing your best model to the initial model).
- *(0 point)* Your code for this entire question (so that the TA can verify your results).

Here are some tips for approaching this assignment.

- You may want to have your logistic regression model check for convergence less frequently (e.g. every 10 iterations could give an approximate 2x speedup).
- Make sure to limit the number of folds you use with cross-validation (2 is fine for this question).

- Test your hyperparameter settings before launching a giant loop because some settings can take a long time!
  (a) If you pick the wrong hyperparameters, a single run can take hours.
  (b) It is okay to limit your parameter sweeps, trying values that take 'reasonable time' on your machine (less than 30 minutes).
  (c) *If* all your code is working correctly, you will be able to find highly accurate models, even with that constraint. (I hope it is not a big *if* by the time you come to this point of the project.)
- Consider outputting all the results you need as you go and making it possible to restart from where you crash (manually at least). If you do not plan for disaster and something crashes after you have run your code a long time (and with no output!)...you will be very, very, very sad...
- If you are still terrified about the code crashing, consider writing the output and results of your runs into a file so that you can pick up from where it left off.
- You probably want to use the Python package `joblib` to run your evaluations in parallel. You can install `joblib` in the same way you install `Pillow` and `matplotlib`. Below are some code snippets on how `joblib` can be used. More information about the package can be found at `https://joblib.readthedocs.io/en/latest/installing.html`.

```
# Try a bunch of different parameter settings in serial
evaluations = [ Execute(parameters) for parameters in listOfAllParametersToTry ]

# Using joblib to parallelize the same task across 8 CPU cores
from  import Parallel, delayed
evaluations = Parallel(n_jobs=8)(delayed(Execute) (parameters) for parameters \
  in listOfAllParametersToTry)
```

Good luck!