CS 475 Machine Learning: Project 1
Supervised Classifiers 1

Due: Thursday February 22, 2018, 11:59pm

50 Points Total Version 1.0

Make sure to read from start to finish before beginning the assignment.

1 Introduction

The goal of the programming homeworks in this course is to build a machine learning library. In each homework assignment you will expand your learning library by implementing and evaluating new algorithms. Most, but not all, programming assignments will build upon previous assignments by comparing algorithms on common data. The purpose of the first assignment is to familiarize you with the data, the framework, and some very simple classifiers.

1.1 Components of Assignments

Assignments consist of two parts.

1. **Programming:** This will vary by assignment. In this first assignment you will implement two simple classification algorithms: the sum of features classifier and the perceptron. Both of these algorithms will be tested on the provided data.
2. **Analytical questions:** These questions will ask you to consider questions related to the topics covered by the assignment. You will be able to answer these questions without relying on your programming.

Assignments are worth a variable number of points. The current plan is for the first and last homework to be worth 50 points, with the other 3 being worth 100 each. The point totals will be indicated in the assignment.

Each assignment will contain a version number at the top. While we try to ensure every homework is perfect when we release it, small errors do happen. When we correct these, we'll update the version number, post a new PDF, and announce the change. Each homework starts at version 1.0 (no beta).

2 Data

The first part of the semester will focus on supervised classification. We consider several real world binary classification datasets taken from a range of applications. Each dataset is in the same format (described below) and contains a train, development, and test file. You will train your algorithm on the train file and use the development set to test that your algorithm works. The test file contains unlabeled examples that we will use to test your algorithm. It is **a very good idea** to run on the test data just to make sure your code doesn't crash. You'd be surprised how often this happens.

2.1 Biology

Biological research produces large amounts of data to analyze. Applications of machine learning to biology include finding regions of DNA that encode for proteins, classification of gene expression data, and inferring regulatory networks from mRNA and proteomic data.

Our biology task of characterizing gene splice junction sequences comes from molecular biology, a field interested in the relationships of DNA, RNA, and proteins. Splice junctions are points on a sequence at which "superfluous" RNA is removed before the process of protein creation in higher organisms. Exons are nucleotide sequences that are retained after splicing while introns are spliced out. The goal of this prediction task is to recognize DNA sequences that contain boundaries between exons and introns. Sequences contain exon/intron (EI) boundaries, intron/exon (IE) boundaries, or do not contain splice examples.

For a binary task, you will classify sequences as either EI boundaries (label 1) or non-splice sequences (label 0). Each learning instance contains a 60 base pair sequence (ex. ACGT), with some ambiguous slots. Features encode which base pair occurs at each position of the sequence.

2.2 Finance

Finance is a data rich field that employs numerous statistical methods for modeling and prediction, including the modeling of financial systems and portfolios.[1]

Our financial task is to predict which Australian credit card applications should be accepted (label 1) or rejected (label 0). Each example represents a credit card application, where all values and attributes have been anonymized for confidentiality. Features are a mix of continuous and discrete attributes and discrete attributes have been binarized.

2.3 NLP

Natural language processing studies the processing and understanding of human languages. Machine learning is widely used in NLP tasks, including document understanding, information extraction, machine translation and document classification.

Our NLP task is sentiment classification. Each example is a product review taken from Amazon kitchen appliance reviews. The review is either positive (label 1) or negative (label 0) towards the product. Reviews are represented as uni-gram and bi-grams; each one and two word phrase is extracted as a feature.

2.4 Speech

Statistical speech processing has its roots in the 1980s and has been the focus of machine learning research for decades. The area deals with all aspects of processing speech signals, including speech transcription, speaker identification and speech information retrieval.

Our speech task is spoken letter identification. Each example comes from a speaker saying one of the twenty-six letters of English alphabet. Our goal is to predict which letter was spoken. The data was collected by asking 150 subjects to speak each letter of the alphabet twice.

[1] For an overview of such applications, see the proceedings of the 2005 NIPS workshop on machine learning in finance. `http://www.icsi.berkeley.edu/~moody/MLFinance2005.htm`

Each spoken utterance is represented as a collection of 617 real valued attributes scaled to be between -1.0 and 1.0. Features include spectral coefficients; contour features, sonorant features, pre-sonorant features, and post-sonorant features. The binary task is to distinguish between the letter M (label 0) and N (label 1).

2.5 Vision

Computer vision processes and analyzes images and videos and it is one of the fundamental areas of robotics. Machine learning applications include identifying objects in images, segmenting video and understanding scenes in film.

Our vision task is image segmentation. In image segmentation, an image is divided into segments are labeled according to content. The images in our data have been divided into 3x3 regions. Each example is a region and features include the centroids of parts of the image, pixels in a region, contrast, intensity, color, saturation and hue. The goal is to identify the primary element in the image as either a brickface, sky, foliage, cement, window, path or grass. In the binary task, you will distinguish segments of foliage (label 0) from grass (label 1).

2.6 Synthetic Data

When developing algorithms it is often helpful to consider data with known properties. We typically create synthetic data for this purpose. To help test your algorithms, we are providing two synthetic datasets. These data are to help development.

2.6.1 Easy

The easy data is labeled using a trivial classification function. Any reasonable learning algorithm should achieve near flawless accuracy. Each example is a 10 dimensional instance drawn from a multi-variate Gaussian distribution with 0 mean and a diagonal identity covariance matrix. Each example is labeled according to the presence one of 6 features; the remaining features are noise.

2.6.2 Hard

Examples in this data are randomly labeled. Since there is no pattern, no learning algorithm should achieve accuracy significantly different from random guessing (50%). Data is generated in an identical manner as *Easy* except there are 94 noisy features.

3 Programming (30 points)

In this assignment you will implement two simple classifiers: the sum of features classifier and the perceptron. We have provided Python code to serve as a testbed for your algorithms. You will fill in the details. Search for comments that begin with `TODO`; these sections need to be written. You may change the internal code as you see fit but **do not change the names of any of the files or command-line arguments that have already been provided.** Other than the given filenames and command-line arguments, you are free to change what you wish: you can modify internal code, add internal code, add files, and/or add new command-line arguments (in which case you should include appropriate defaults).

3.1 Python Libraries

We will be using Python 3.5.x. We are *not* using Python 2, and will not accept assignments written for this version. We recommend using a recent release of Python 3.5.x, but anything in this line (e.g. 3.6.x) should be fine.

For each assignment, we will tell you which Python libraries you may use. We will do this by providing a `requirements.txt` file. We *strongly* recommend using a *virtual environment* to ensure compliance with the permitted libraries. By strongly, we mean that unless you have a very good reason not to, and you really know what you are doing, you should use a virtual environment.

3.2 Virtual Environments

Virtual environments are easy to set up and let you work within an isolated Python environment. In short, you can create a directory that corresponds to a specific Python version with specific packages, and once you activate that environment, you are shielded from the various Python / package versions that may already reside elsewhere on your system. Here is an example:

```
# Create a new virtual environment.
python3 -m venv python3-hw1
# Activate the virtual environment.
source python3-hw1/bin/activate
# Install packages as specified in requirements.txt.
pip3 install -r requirements.txt
# Optional: Deactivate the virtual environment, returning to your system's setup.
deactivate
```

When we run your code, we will use a virtual environment with *only* the libraries in `requirements.txt`. If you use a library not included in this file, your code will fail when we run it, leading to a large loss of points. By setting up a virtual environment, you can ensure that you do not mistakenly include other libraries, which will in turn help ensure that your code runs on our systems.

Make sure you are using the correct `requirements.txt` file from the current assignment. We may add new libraries to the file in subsequent assignments, or even remove a library (less likely). If you are using the wrong assignments `requirements.txt` file, it may not run when we grade it. For this reason, we suggest creating a new virtualy environment for each assignment.

It may happen that you find yourself in need of a library not included in the `requirements.txt` for the assignment. You may request that a library be added by posting to Piazza. This may be useful when there is some helpful functionality in another library that we omitted. However, we are unlikely to include a library if it either solves a major part of the assignment, or includes functionality that isn't really necessary.

In this and future assignments we will allow you to use `numpy` and `scipy`.

3.3 How to Run the Provided Framework

The framework operates in two modes: train and test. Both stages are in the main method of `classify.py`.

3.3.1 Train Mode

The usage for train mode is

```
python3 classify.py --mode train --algorithm algorithm_name --model-file model_file --data train_file
```

The `mode` option indicates which mode to run (train or test). The `algorithm` option indicates which training algorithm to use. Each assignment will specify the string argument for an algorithm. The `data` option indicates the data file to load. Finally, the `model-file` option specifies where to save the trained model.

3.3.2 Test Mode

The test mode is run in a similar manner:

```
python3 classify.py --mode test --model-file model_file --data test_file --predictions-file predictions_file
```

The `model_file` is loaded and run on the `data`. Results are saved to the `predictions_file`.

3.3.3 Examples

As an example, the following trains a perceptron classifier on the speech training data:

```
python3 classify.py --mode train --algorithm perceptron --model-file speech.perceptron.model \
--data speech.train
```

To run the trained model on development data:

```
python3 classify.py --mode test --model-file speech.perceptron.model --data speech.dev \
--predictions-file speech.dev.predictions
```

As we add new algorithms we will also add command line flags using the `argparse` library to specify algorithmic parameters. These will be specified in each assignment.

3.4 Data Formats

The data are provided in what is known as SVM-light format. Each line contains a single example:

```
0 1:-0.2970 2:0.2092 5:0.3348 9:0.3892 25:0.7532 78:0.7280
```

The first entry on the line is the label. The label can be an integer (0/1 for binary classification) or a real valued number (for regression.) The classification label of -1 indicates unlabeled. Subsequent entries on the line are features. The entry `25:0.7532` means that feature 25 has value 0.7532. Features are 1-indexed.

Model predictions are saved as one predicted label per line in the same order as the input data. The code that generates these predictions is provided in the library. The script `compute_accuracy.py` can be used to evaluate the accuracy of your predictions for classification:

```
python3 compute_accuracy.py data_file predictions_file
```

We provide this script since it is exactly how we will evaluate your output. Make sure that your algorithm is outputting labels as they appear in the input files. If you use a different internal representation of your labels, make sure the output matches what's in the data files. The above script will do this for you, as you'll get low accuracy if you write the wrong labels.

3.5 Existing Components

The foundations of the learning framework have been provided for you. You will need to complete this library by filling in code where you see a `TODO` comment. You are free to make changes to the code as needed provided you do not change the behavior of the command lines described above. We emphasize this point: **do not change the existing command line flags, existing filenames, or algorithm names.** We use these command lines to test your code. If you change their behavior, we cannot test your code.

The code we have provided is fairly compact, and you should spend some time to familiarize yourself with it. Here is a short summary to get you started:

- `data.py` – This contains the `load_data` function, which parses a given data file and returns features and labels. The features are stored as a sparse matrix of floats (and in particular as a `scipy.sparse.csr_matrix` of floats), which has `num_examples` rows and `num_features` columns. The labels are stored as a dense 1-D array of integers with `num_examples` elements.

- `classify.py` – This is the main testbed to be run from the command line. It takes care of parsing command line arguments, entering train/test mode, saving models/predictions, etc. Once again, **do not change the names of existing command-line arguments**.

- `models.py` – This contains a `Model` class which you should extend. Models have (in the very least) a `fit` method, for fitting the model to data, and a `predict` method, which computes predictions from features. You are free to add other methods as necessary. **Note that all predictions from your model must be 0 or 1; if you use other intermediate values for internal computations, then they must be converted before they are returned.**

- `compute_accuracy.py` – This is a script which simply compares the true labels from a data file (e.g., `bio.dev`) to the predictions that were saved by running `classify.py` (e.g., `bio.dev.SumOfFeatures.predictions`).

- `run_on_all_datasets.py` – This is not necessarily needed, but is included simply to make it easier for you to test your algorithms on all datasets, and to make sure that your algorithms even run on the `test` sets. Inside this script you can specify the main data directory, which should contain all of the `*.train, *.dev, *.test` files, along with an output directory (for models and predictions). The script loops through the datasets to train and test on each. Feel free to modify this script as needed; by default, it assumes the data directory is `./datasets` and that the desired output directory is `./output`.

We have also included a toy model, which we call `Useless`, for your reference. This is a binary classifier which forms predictions in a very simple and naive way. At training time, it simply memorizes the first example it sees (both its features and label). At prediction time, it forms the dot product between all of the examples and the memorized training example, and forms a prediction based on this dot product. (If the dot product is greater than 0, then it returns the same label as the memorized example; otherwise, the opposite label is returned.) Do not think too much about this toy model – it's simply included as a reference for how the `fit` and `predict` methods can be defined. In fact you can go

through the whole training/testing process with this toy model on the provided datasets, and we recommend that you do so.

3.6 Sum of Features Classifier (10 points)

Here you will implement a classifier which computes two sums for each data point: The sum of the first half of the features and the sum of the second half of the features. Predict 1 if the sum of the first half is greater than or equal to the sum of second half. Otherwise, predict 0.

As an example, for a data point $x = [0.2, -1.3, 0.4, 0.9, 3.4, -0.1]$ the two corresponding sums would be $0.2 - 1.3 + 0.4 = -0.7$ and $0.9 + 3.4 - 0.1 = 4.2$. Thus, your classifier would predict 0. If the number of features is odd, then omit the feature in the middle for your calculations; e.g. $x = [0.2, -1.3, 0.4, 0.5, 0.9, 3.4, -0.1]$ should return the same sums as $x = [0.2, -1.3, 0.4, 0.9, 3.4, -0.1]$.

We note that this algorithm doesn't necessarily make sense in a practical setting and will not return good classification error. However, implementing it successfully demonstrates that you can manipulate and make use of the features.

3.7 Perceptron (20 points)

Here you will implement the Perceptron algorithm for binary classification. The perceptron is a mistake-driven online learning algorithm. It takes as input a vector of real-valued inputs $\mathbf{x}$ and makes a prediction $\hat{y} \in \{-1, +1\}$ (for this assignment we consider only binary labels). **This simplifies computations internally, but we again emphasize that your model's `predict` method must return labels that are 0 or 1.** Predictions are made using a linear classifier: $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$. The term $\mathbf{w} \cdot \mathbf{x}$ is the dot product of $\mathbf{w}$ and $\mathbf{x}$ computed as $\sum_i x_i w_i$. Updates to $\mathbf{w}$ are made only when a prediction is incorrect: $\hat{y} \neq y$. The new weight vector $\mathbf{w}'$ is a function of the current weight vector $\mathbf{w}$ and example $\mathbf{x}$, y. The weight vector is updated so as to improve the prediction on the current example. Note that Perceptron naturally handles continuous and binary features, so no special processing is needed.

The basic structure of the algorithm is:

1. Initialize $\mathbf{w}$ to $\mathbf{0}$, set learning rate η and number of iterations I
2. For each training iteration $k = 1 \dots I$:
   (a) For each example $i = 1 \dots N$:
      i. Receive an example $\mathbf{x}_i$
      ii. Predict the label $\hat{y}_i = \text{sign}(\mathbf{w} \cdot \mathbf{x}_i) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x}_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$
      iii. If $\hat{y}_i \neq y_i$, make an update to $\mathbf{w}$: $\mathbf{w}' = \mathbf{w} + \eta y_i \mathbf{x}_i$

Note that there is no bias term in this version and you should *not* include one in your solution. Also observe the definition of "sign" to account for 0 values. Once again, while sign returns -1 and 1, the outputs from your `predict` method must be the actual labels, which are 0 or 1.

3.7.1 Deliverables

You need to implement the Sum of Features Classifier and Perceptron and Perceptron algorithms. To do this, see the `TODO` comments in the code, and feel free to modify / add code as necessary. However, once again, **do not modify the filenames or command-line arguments that have already been provided.** Your predictors will be selected by passing the string `sumoffeatures` or `perceptron` as the argument for the `algorithm` parameter.

3.7.2 Perceptron: the learning rate

In the case of the perceptron, you will need to specify a learning rate η, where $0 < \eta \leq 1$. Your default value for η should be 1. You *must* add a command line argument to allow this value to be adjusted via the command line.

Add this command line option by adding the following code to the `get_args` function in `classify.py`.

```
parser.add_argument("--online-learning-rate", type=float, help="The learning rate for perceptron",
default=1.0)
```

Be sure to add the option name exactly as it appears above. You can then use the value read from the command line in your main method by referencing it as `args.online_learning_rate`. Note that the dashes have been replaced by underscores.

3.8 Perceptron: the number of training iterations

Usually we iterate multiple times over the data. This can improve performance by increasing the number of updates made. We will define the number of times each algorithm iterates over all of the data using the parameter `online_training_iterations`. You *must* define a new command line option for this parameter. Use a default value of 5 for this parameter.

You can add this option by adding the following code to the `get_args` function of `classify.py`.

```
parser.add_argument("--online-training-iterations", type=int,
help="The number of training iterations for online methods.", default=5)
```

You can then use the value read from the command line in your main method by referencing it as `args.online_training_iterations`.

During training, you should not change the order of examples. You must iterate over examples exactly as they appear in the data file, i.e. as provided by the data loader.

3.9 Grading Programming

The programming section of your assignment will be graded using an automated grading program. Your code will be run using the provided command line options, as well as other variations on these options (different parameters, data sets, etc.) The grader will consider the following aspects of your code.

1. **Exceptions:** Does your code run without crashing?
2. **Output:** Some assignments will ask you to write some data to the console. Make sure you follow the provided output instructions exactly.

3. **Accuracy:** If your code works correctly, then it should achieve a certain accuracy on each data set. While there are small difference that can arise, a correctly working implementation will get the right answer.

4. **Speed/Memory:** As far as grading is concerned, efficiency largely doesn't matter, except where lack of efficiency severely slows your code (so slow that we assume it is broken) or the lack of efficiency demonstrates a lack of understanding of the algorithm. For example, if your code runs in two minutes and everyone else runs in 2 seconds, you'll lose points. Alternatively, if you require 2 gigs of memory, and everyone else needs 10 MB, you'll lose points. In general, this happens not because you did not optimize your code, but when you've implemented something incorrectly.

3.10 Code Readability and Style

In general, you will not be graded for code style. However, your code should be readable, which means minimal comments and clear naming / organization. If your code works perfectly then you will get full credit. However, if it does not we will look at your code to determine how to allocate partial credit. If we cannot read your code or understand it, then it is very difficult to assign partial credit. Therefore, it is in your own interests to make sure that your code is reasonably readable and clear.

3.11 Code Structure

Your code must support the command line options and the example commands listed in the assignment. Aside from this, you are free to change the internal structure of the code, write new classes, change methods, add exception handling, etc. However, once again, do not change the names of the files or command-line arguments that have been provided. We suggest you remember the need for clarity in your code organization.

3.12 Knowing Your Code Works

How do you know your code really works? That is a very difficult problem to solve. Here are a few tips:

1. Check results on `easy` and `hard`. For the sum of features classifier, you can make sure it is computing the sums correctly with a small subset of one of the datasets with a small number of features. In the case of the perceptron, you should get close to 100% on `easy` and close to 50% on `hard`.

2. Use Piazza. While **you cannot share code**, you can share results. It is acceptable to post your results on dev data for your different algorithms. A common result will quickly emerge that you can measure against.

3. Output intermediate steps. Looking at final predictions that are wrong tells you little. Instead, print output as you go and check it to make sure it looks right. This can also be helpful when sharing information on the bulletin board.

4. Debug. Find a Python debugger that you like and use it. This can be very helpful.

3.13 Debugging

The most common question we receive is "how do I debug my code?" The truth is that machine learning algorithms are very hard to debug because the behavior of the algorithm is unknown. In these assignments, you won't know ahead of time what accuracy is expected for your algorithm on a dataset. This is the reality of machine learning development, though in this class you have the advantage of your classmates, who may post the output of their code to the bulletin board. While debugging machine learning code is therefore harder, the same principles of debugging apply. Write tests for different parts of your code to make sure it works as expected. Test it out on the easy datasets to verify it works and, when it doesn't, debug those datasets carefully. Work out on paper the correct answer and make sure your code matches. Don't be afraid of writing your own data for specific algorithms as needed to test out different methods. This process is part of learning machine learning algorithms and a reality of developing machine learning software.

4 Analytical (20 Points)

In addition to completing the analytical questions, your assignment for this homework is to learn Latex. All homework writeups must be PDFs compiled from Latex. Why learn latex?

1. It is incredibly useful for writing mathematical expressions.
2. It makes references simple.
3. Many academic papers are written in latex.

The list goes on. Additionally, it makes your assignments much easier to read than if you try to scan them in or complete them in Word.

We realize learning latex can be daunting. Fear not. There are many tutorials on the Web to help you learn. We recommend using pdflatex. It's available for nearly every operating system. Additionally, we have provided you with the tex source for this PDF, which means you can start your writeup by erasing much of the content of this writeup and filling in your answers. You can even copy and paste the few mathematical expressions in this assignment for your convenience. As the semester progresses, you'll no doubt become more familiar with latex, and even begin to appreciate using it.

Be sure to check out this cool latex tool for finding symbols. It uses machine learning! `http://detexify.kirelabs.org/classify.html`

1) Supervised vs. Unsupervised Learning (3 points) .

1. Give an example of a problem that could be solved with both supervised learning and unsupervised learning. Is data readily available for this problem? How would you measure your 'success' in solving this problem for each approach?
2. What are the pros and cons of each approach? Which approach do you think the problem better lends itself to?

2) Model Complexity (3 points) Explain when you would want to use a simple model over a complex model and vice versa. Are there any approaches you could use to mitigate the disadvantages of using a complex model?

3) Training and Generalization (3 points) Suppose you're building a system to classify images of food into two categories: either the image contains a hot dog or it does not. You're given a dataset of 25,000 (image, label) pairs for training and a separate dataset of 5,000 (image, label) pairs.

1. Suppose you train an algorithm and obtain 96% accuracy on the larger training set. Do you expect the trained model to obtain similar performance if used on newly acquired data? Why or why not?

2. Suppose that, after training, someone gives you a new test set of 1,000 (image, label) pairs. Which do you expect to give greater accuracy on the test set: The model after trained on the dataset of 25,000 pairs or the model after trained on the dataset of 5,000 pairs? Explain your reasoning.

3. Suppose your models obtained greater than 90% accuracy on the test set. How might you proceed in hope of improving accuracy further?

4) Loss Function (3 points) State whether each of the following is a valid loss function for binary classification. Wherever a loss function is not valid, state why. Here y is the correct label and $\hat{y}$ is a decision confidence value, meaning that the predicted label is given by $\text{sign}(\hat{y})$ and the confidence on the classification increases with $|\hat{y}|$.

1. $\ell(y, \hat{y}) = \frac{3}{4}(y - \hat{y})^2$

2. $\ell(y, \hat{y}) = |(y - \hat{y})|/\hat{y}$

3. $\ell(y, \hat{y}) = \max(0, 1 - y \cdot \hat{y})$

5) Linear Regression (4 points) Suppose you observe n data points $(x_1, y_1), \ldots, (x_n, y_n)$, where all x_i and all y_i are scalars.

1. Suppose you choose the model $\hat{y} = wx$ and aim to minimize the sum of squares error $\sum_i (y_i - \hat{y}_i)^2$. Derive the closed-form solution for w from scratch, where 'from scratch' means without using the least-squares solution presented in class.

2. Suppose you instead choose the model $\hat{y} = w \sin(x)$ and aim to minimize the sum of squares error $\sum_i (y_i - \hat{y}_i)^2$. Is there a closed-form solution for w? If so, what is it?

6) Logistic Regression (4 points) Explain whether each statement is true or false. If false, explain why.

1. In the case of binary classification, optimizing the logistic loss is equivalent to minimizing the sum-of-squares error between our predicted probabilities for class 1, $\hat{\mathbf{y}}$, and the observed probabilities for class 1, $\mathbf{y}$.

2. One possible advantage of stochastic gradient descent is that it can sometimes escape local minima. However, in the case of logistic regression, the global minimum is the only minimum, and stochastic gradient descent is therefore never useful.

5 What to Submit

You will need to create an account on gradescope.com and signup for this class. The course is `https://gradescope.com/courses/10244`. Use entry code `974Z3W`. See this video for instructions on how to upload a homework assignment: `https://www.youtube.com/watch?v=KMPoby5g_nE`. In each assignment you will submit two things to gradescope.

1. **Submit your code (.py files) to gradescope.com as a zip file**. **Your code must be uploaded as code.zip with your code in the root directory**. By 'in the root directory,' we mean that the zip should contain `*.py` at the root (`./*.py`) and not in any sort of substructure (for example `hw1/*.py`). One simple way to achieve this is to zip using the command line, where you include files directly (e.g., `*.py`) rather than specifying a folder (e.g., `hw1`):

   ```
   zip code.zip *.py
   ```

   We will run your code using the exact command lines described earlier, so make sure it works ahead of time, and make sure that it doesn't crash when you run it on the test data. Remember to submit all of the source code, including what we have provided to you. We will include `requirements.txt` but nothing else.

2. **Submit your writeup to gradescope.com**. **Your writeup must be compiled from latex and uploaded as a PDF**. It should contain all of the answers to the analytical questions asked in the assignment. Make sure to include your name in the writeup PDF and to use the provided latex template for your answers.

6 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: `https://piazza.com/class/j70ixzhajea1rt`.