CS 475 Machine Learning: Homework 1
Supervised Classifiers 1

Due: Thursday September 26, 2019, 11:59pm

100 Points Total Version 1.0

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

1 Introduction

Assignment in this course can consist of two parts.

1. **Programming:** The goal of the programming homeworks in this course is to learn about how machine learning algorithms work through hands on experience. In each homework assignment you will both implement an algorithm and run experiments with it. You may also be asked to experiment with these algorithms in a Python Notebook to learn more about how they work.
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 various points. Typical assignments are worth 100 each, with point totals 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, 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.

2 Data

The first part of the semester will focus on supervised classification. We consider several real-world 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 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 26 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.

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. In this homework, we'll focus on the multiclass version of this dataset, where our job is to classify each example as 1 of 26 letters.

3 Programming (50 points)

In this assignment you will implement two multiclass classifiers: perceptron and logistic regression. Additionally, you will consider two different approaches to implementing a multiclass classifier: reduction-to-binary and true multiclass models. We have provided Python code to serve as a testbed for your algorithms. We will reuse this testbed in future assignments as well. 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 as we won't add these arguments when we run your code).

3.1 Python Libraries

We will be using Python 3.7.3. We are *not* using Python 2, and will not accept assignments written for this version. We recommend using a recent release of Python 3.7.x, but any recent Python3 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. You can also use anaconda environments, which achieve the same goal. The point is that you should ensure you are only using libraries available in the basic Python library or the `requirements.txt` file.

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.
pip 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 virtual 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`.

"Can I use the command `import XXX`?" For every value of `XXX` the answer is: if you are using a virtual environment setup with the distributed `requirements.txt` file, then you can import and use anything in that environment.

Please verify you use the exact same command line flags we dictate. There are always students with a typo in their command line flags, which means their code fails when we run it.

3.3 How to Run the Provided Framework

The framework operates in two modes: train and test. Both stages are defined in `main.py`, which has the following arguments:

- `--mode` defines which mode to run it, either `train` or `test`
- `--train-data` Required in train mode! A file with labeled training data.
- `--test-data` Required in test mode! A file with labeled (dev) or unlabeled (test) data, to perform inference over.
- `--model-file` Required in both modes! In train mode, this will determine where to save your model after training. In test mode, this will tell the program where to load a pretrained model from, to perform inference.
- `--predictions-file` Required in test mode! Tells the program where to store a model's predictions over the test data.
- `--algorithm` Required in train mode! Which model to train. You need to implement these! This is only required during train-mode, since test-mode loads a pretrained model from a pickled file. The algorithms we'll use are `mc-perceptron, mc-logistic, 1vA-perceptron, 1vA-logistic`

In addition to these meta arguments, `main` also takes some hyperparameter arguments, namely:

- `--learning-rate` The η to use for weight updates. The default is $\eta = 0.001$

- `--train-epochs` The number of iterations the model should train for on the entire dataset. By default, this value is 15.

Feel free to explore values for the last two hyperparameters to try to find hyperparameters that do the best for your algorithms. Since we're not implementing any regularization, these are the only hyperparameters we have to search over.

Note that when we run your code, we will use predetermined values for these, to ensure consistency amongst everyone's implementation.

3.3.1 Train Mode

The usage for train mode is

```
python3 main.py --mode train --algorithm algorithm_name --model-file model_file --train-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 `train-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 main.py --mode test --model-file model_file --test-data test_file --predictions-file predictions_file
```

The `model-file` is loaded and run on the `test-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 main.py --mode train --algorithm mc-logistic --model-file speech.perceptron.model \
                --train-data speech.train
```

To run the trained model on development data:

```
python3 main.py --mode test --model-file speech.perceptron.model --test-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, 1-k for k multiclass 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 test_file test_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.

- `main.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.

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

3.6 Multiclass Classification

In multiclass classification, we are tasked with classifying our data into a set of K classes, rather than just 2, as in binary classification setting.

There are many ways to implement multiclass classifiers. In this assignment we will consider two.

- **One-against-All (OA)**: For each class k of K we train a single binary classifier to differentiate between examples labeled with class k versus all other examples (from all other classes.) We train K such models. At prediction time, we apply each model to a new example and select the label that corresponds to the classifier with the highest score.

- **Multiclass Classifier (MC)**: We train a true multiclass classifier, in which a single classifier can output all K different labels. All model parameters are optimized jointly during training.

We will begin the assignment with the true **Multiclass Classifier** model, and then explain how to implement **One-against-All**.

3.6.1 Multiclass Perceptron (20 points)

You will implement a multiclass Perceptron algorithm. This is an extension of the binary version of the Perceptron you saw in class. The binary Perceptron predicts, for $\mathbf{x}_i$:

$$\widehat{y}_i = \text{sign}(\mathbf{w} \cdot \mathbf{x}_i) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x}_i \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

And updates only if $\widehat{y}_i \neq y_i$ with learning rate η:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i \tag{2}$$

We can extend this to a multiclass setting, where the model instead uses K weight vectors, $\mathbf{w}_k$. To make a prediction, we predict the class with the highest score

$$\widehat{y}_i = \underset{k \in 1,\ldots,K}{\text{argmax}}\, \mathbf{w}_k \cdot \mathbf{x}_i \tag{3}$$

There are several strategies for updating this model. We ask you to implement the following:

$$\begin{aligned} &\texttt{if } \widehat{y}_i \neq y_i \texttt{ then}: \\ &\quad \mathbf{w}'_{\widehat{y}_i} \leftarrow \mathbf{w}_{\widehat{y}_i} - \eta \mathbf{x}_i \\ &\quad \mathbf{w}'_{y_i} \leftarrow \mathbf{w}_{y_i} + \eta \mathbf{x}_i \end{aligned} \tag{4}$$

This update rule only modifies 2 of the K weight vectors for each incorrect prediction. It modified the parameters for the incorrectly predicted class so that they produce a lower score for $\mathbf{x}_i$ and the parameters for the correct class so that they produce a higher score for $\mathbf{x}_i$.

Implement this model in the `MCPerceptron` class in `models.py`.

3.6.2 Multiclass Logistic Regression (20 points)

The second model you will implement is a Multiclass Logistic Regression algorithm. Similar to the Multiclass Perceptron algorithm, this model will have a separate weight vector $\mathbf{w}_k$ for each of the K classes.

In binary logistic regression, our likelihood function is given by the sigmoid function over the dot product of our weight vector $\mathbf{w}$ with our feature vector $\mathbf{x}_i$:

$$P_{\text{Binary}}(y = 1 \mid \mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}_i}} \tag{5}$$

In the multiclass setting, we instead have a weight vector for each class $\mathbf{w}_k$ which defines a score for each class k given $\mathbf{x}_i$, namely

$$g_k = \mathbf{w}_k \cdot \mathbf{x}_i \tag{6}$$

The values of $\mathbf{g} = (g_1, ..., g_K)$ are often called *logits* or *goodness scores*, which represent the unnormalized distribution of $P(y \mid \mathbf{x}_i)$. To normalize our logits into a valid probability distribution over y, we need a slightly different function than the sigmoid, one which takes our K scores, and converts them into a probability distribution over our K classes. This function is called the Softmax function, and we'll define it as follows:

$$[\text{softmax}(\mathbf{g})]_k \stackrel{\text{def}}{=} \frac{e^{g_k}}{\sum_{k'=1}^{K} e^{g_{k'}}} = \frac{e^{\mathbf{w}_k \cdot \mathbf{x}_i}}{\sum_{k'=1}^{K} e^{\mathbf{w}_{k'} \cdot \mathbf{x}_i}} = P(y = k \mid \mathbf{x}_i) \tag{7}$$

In practice, the implementation of the above formula faces computational challenges. Since the score function is not constrained to output logits in a specified range, g_k can be arbitrarily large. Computing exp of large numbers causes numerical overflow.

The standard implementation uses the exp-normalize trick [1] to ensure numerical stability and protection from a denominator too close to 0. The trick works by subtracting the max value of $\mathbf{g}$ from all of $\mathbf{g}$, before we compute the Softmax. This ensures that the max logit in $\mathbf{g}$ is 0, so we can't overflow anymore. Additionally, since $e^0 = 1$, it ensures that our denominator is ≥ 1. Because the Softmax distribution is invariant to shifts, this trick doesn't change the output of the function.

The numerically stable version of the Softmax function looks like this:

$$g^* = \max(\mathbf{g}) \tag{8}$$

$$\text{softmax}(\mathbf{g})_k = \frac{e^{(g_k - g^*)}}{\sum_{k'=1}^{K} e^{(g_{k'} - g^*)}} \tag{9}$$

You will need to implement the numerically stable version of the `softmax` function in your `MCLogistic` class, in order to safely convert your class "logits" into a probability distribution over y.

Similar to binary logistic regression, we will train our model by maximizing the log-likelihood of our data

$$\mathcal{L} = \log \prod_{i=1}^{N} P(y_i \mid \mathbf{x}_i) = \sum_{i=1}^{N} \left(\mathbf{w}_{y_i} \cdot \mathbf{x}_i - \log \sum_{k=1}^{K} e^{\mathbf{w}_k \cdot \mathbf{x}_i} \right) \tag{10}$$

You will train your model using Stochastic Gradient Ascent, which uses the gradient of the likelihood of one example at a time, yielding this likelihood function:

$$\ell(y_i, \mathbf{x}_i) = \log P(y_i \mid \mathbf{x}_i) = \mathbf{w}_{y_i} \cdot \mathbf{x}_i - \log \sum_{k=1}^{K} e^{\mathbf{w}_k \cdot \mathbf{x}_i} \tag{11}$$

The gradient of this function with respect to one of our class weight vectors is:

$$\nabla_{\mathbf{w}_k} \ell(y_i, \mathbf{x}_i) = \begin{cases} \mathbf{x}_i - P(k \mid \mathbf{x}_i) \cdot \mathbf{x}_i & \text{if } \; k = y_i \\ -P(k \mid \mathbf{x}_i) \cdot \mathbf{x}_i & \text{otherwise} \end{cases} \tag{12}$$

You should use this gradient when updating the weights during training.

Conveniently, the inference rule for multiclass logistic regression is the same as that of our multiclass perceptron as well, that is to say that for some $\mathbf{x}_i$:

$$\widehat{y}_i = \underset{k \in 1,..,K}{\text{argmax}} \, \mathbf{w}_k \cdot \mathbf{x}_i \tag{13}$$

Implement this model in the `MCLogistic` class in `models.py`

[1] See `https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/` for more

3.6.3 One-against-All Binary Classification (10 points)

In addition to the multiclass models above, you will implement a One-against-All classifier. Recall that a OA classifier uses K binary classifiers, each trained to distinguish one of the k classes from all the other classes. During inference, the class with the highest confidence from it's classifier on a given example will be the predicted class.

You will re-use your multiclass models as binary classifiers (2 classes). These models aren't exactly the same as the binary classifiers you learned in class, as they will have 2 weight vectors for each binary classification problem, and in class we learned about binary classifiers with only 1 weight vector to distinguish both classes. Nevertheless, for both logistic regression and the perceptron, it can be shown that these 2 implementations are essentially equivalent, although one is a bit more intuitive than the other. Rather than ask you to implement more algorithms, we'll just reuse the ones you've already written!

You will probably want to implement some sort of `score` function in each of your MC classifiers, to make it easy for your OA algorithm to make predictions. Unfortunately, because of our decision to reuse the MC models, each binary classifier has 2 weight vectors and therefore 2 scores - you only want to consider one when you're performing inference! You should use the score for the positive label, i.e. "this example belongs to this class."

Implement this model in the `OneVsAll` class in `models.py`. Your implementation should be general enough that it works with both the `MCPerceptron` and `MCLogistic` models, which you've already written, when we reduce these models to 2 classes.

3.7 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.

The Python notebook will be graded manually based on including the requested outputs. See the notebook for details.

3.8 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.

3.9 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.

3.10 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`. The majority classifier is simple; you can count labels and then check its outputs to make sure it did the right thing. 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. We encourage you 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 Piazza.

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

3.11 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 (50 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`

For each homework assignment we will provide you with a LaTeX template. You **must use the template**. The template contains detailed directions about how to use it.

At this point, please open the file `homework1_template.pdf` to read and `homework1_template.tex` respond to the analytical questions.

5 What to Submit

In each assignment you may submit three different things.

1. **Submit your code (.py files) to cs475.org 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
   ```

   A common mistake is to use a program that automatically places your code in a subfolder. It is your job to make sure you have zipped your code correctly.

   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. A common mistake is to change the command line flags. If you do this, your code will not run.

   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**. The writeup 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 following the distributed template. You will submit this to the assignment called "Homework 1: Supervised Classifiers 1: Written".

You will need to create an account on gradescope.com and signup for this class. The course is `https://www.gradescope.com/courses/57285`. Use entry code `9ZJW22`. **You must either use the email account associated with your JHED, or specify your JHED as your student ID.** See this video for instructions on how to upload a homework assignment: `https://www.youtube.com/watch?v=KMPoby5g_nE`.

6 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: `https://piazza.com/class/jzk051b8fa4xg`. Use signup code `7RZC`.