Final Programming Project: Issued Nov. 22, Due Dec. 6 at 5 p.m.

This final programming project will consider classification. The primary goal is to analyze a satellite image dataset to determine what fraction of the imaged area is barren land, trees, etc. The two secondary goals are, first, to become familiar image data, and, second, to write a persuasive report that is consistent with the principles of data visualization discussed in class.

The Dataset

We will use the "SAT-4 Airborne Dataset," that accompanied the paper

S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki, R. Nemani, "DeepSat: a learning framework for satellite imagery," *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Nov. 2015,

the published version of which is available here:

`https://dl.acm.org/citation.cfm?id=2820816`

and a preprint version of which is available here:

`https://arxiv.org/abs/1509.03602`

The dataset consists of 500,000 satellite images of the Earth's surface. Each images is 28x28 pixels large. The images are in color and also include near-infrared (NIR) information. Thus each pixel has red, green, blue, and NIR components. Each image has been labeled to indicate whether it represents "barren land," "trees," "grassland," or "none." The dataset is divided into a training set consisting of 400,000 images and a test set consisting of 100,000 images, both of which are labeled. The data set is available here:

`http://csc.lsu.edu/~saikat/deepsat/`

It is also available on the `ecelinux` cluster at:

`/classes/ece2720/fpp/sat-4-full.mat`

It is in the MATLAB v5 mat-file format. It contains five variables:

```
annotations
test_x
test_y
train_x
train_y
```

The variable `text_x` contains the training images. It is a 28x28x4x400000 array of `uint8` variables (that is, integers between 0 and 255). The last dimension encodes the index of the image. The first and second dimensions encode the row and column, respectively, of the pixel. The third dimension describes the *channel:* index 0 is red, 1 is green, 2 is blue, and 3 is NIR. Thus

```
train_x[1,2,3,4]
```

is the NIR value for the pixel in the first row and second column of the fourth image (with the indexing starting at zero) in the training data set.

The variable `train_y` provides the labels for the training set. It is a 4x400000 array of bits. Each column i equals one of the following four possibilities:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \tag{1}$$

which indicate which class image i is in, according to the mapping described in the variable `annotations`. This is called *One Hot* encoding. The files `test_x` and `test_y` are structured similarly.

Recall that a large amount of a data scientist's time is spent simply importing the data and working it into a usable form. Accordingly, your first task is to read this data into Python and manipulate it into a form in which you can train a classifier.

Classification

Having read the data, the next step is to design a good classifier. There are many classification techniques that one can employ: logistic regression, support vector machines, k-nearest neighbors, perceptrons, neural networks, etc. We discussed support vector machines in detail in class, and it is possible to design a very satisfactory classifier for this dataset using support vector machines. You are not restricted to support vector machines, however. You are welcome and encouraged to explore other options (but see the submission requirements at the end before investing effort in anything too exotic). Your job is to design the best classifier you can. Nominally, the performance will be measured by the fraction of images in the test dataset that the classifier correctly classifies. But in optimizing this metric you should take care not to overfit to the test data; we reserve the right to test your classifier on a randomly-selected set of 100,000 training images if it appears that you are overfitting the test data.

The design of the classification algorithm is intentionally open-ended. You should experiment with different classification techniques, different parameters, different features, etc., to design the best classifier you can.

The Report

Your report should justify, with evidence, the choices you made in designing your classifier. It should describe different classifiers that you considered, how you optimized them, interesting properties of them, and what performance they achieved. If you optimized certain parameters (such as the C parameter in linear SVMs, or γ in the radial basis function kernel), you should justify your choice using plots that show the performance of the classifier for different parameter choices.

The report should consist of a coherent narrative with ample tables and figures. The figures should adhere to the best practices that we discuss in class for visualizing data: close attention to axis labels, choice of scales, legends, use of color, and, ultimately, creativity in deciding what to plot and how to plot it. Bear in mind that a small number of carefully-crafted plots are more valuable than a large number of simple plots, each of which conveys little information. Some suggestions include heat maps showing the performance of a classifer under different choices for pairs of parameters, "confusion" matrices, examples of specific images that are misclassified by your classifier, etc. The goal of the report is to convince the reader that you explored the space of classifiers with appropriate thoroughness before deciding on your final version.

Submission

You should submit four files. The first, `satellite.py`, should be a Python program that runs on the `ecelinux` cluster, reads the file

`/classes/ece2720/fpp/test_x_only.mat`

which is similar to `sat-4-full.mat` except that it only contains the variable `test_x`, and then creates an ASCII file `landuse.csv` that consists of a single line with the 100,000 classification decisions (represented as the ASCII strings 'barren land', 'trees', 'grassland', and 'none') separated by commas. The second file, `model.dat` is a binary file that `satellite.py` may read (in addition to the actual test image data file) in order to classify. The intent is that you will train your classifier on your own, save the classifier to `model.dat` using, e.g., the `pickle` module, and then have `satellite.py` read the classifier from `model.dat`, without retraining. Since we will be running your classifier, and since with some classification algorithms training can be quite time-consuming, it is important that `satellite.py` not retrain the classifer from scratch. It is worth noting that `satellite.py` is not permitted to read from the training datasets on `ecelinux`, only the test image file listed above. We reserve the right to delete the training datasets on the ecelinux cluster before running your code.

There is a sample `satellite.py` file in `/classes/ece2720/fpp` that illustrates the correct output format. This program does not do any real classification; it just guesses randomly 100,000 times.

It is important that your `satellite.py` script actually read the above file and compute its classifications decisions, as we reserve the right to populate the variable `text_x` with randomly-selected images. You should not hardcode the classification decisions in `satellite.py`.

The third file, `train.py`, should be a Python program that reads from the training files in `/classes/ece2720/fpp`, trains the classifier that is used in `satellite.py`, and creates a file `model.dat` that matches the one you submitted. It should not be necessary to run `train.py` in order to run `satellite.py` (assuming that `model.dat` has been created). Indeed, in most cases we will not actually run your `train.py` program, only `satellite.py`.

Both `satellite.py` and `train.py` should run under `ecelinux` cluster's Python 2.7.15 installation, but they should be programs that can be called directly from the command-line. As described in the `ecelinux` primer, this can be accomplished by making

```
#!/usr/bin/env python
```

the first line of the file, and then having the rest of the file be your Python code (and assuming that whomever calls your script first runs `module load python 2.7.15`, which we will do when grading). You are welcome to use any module that is available in the Python 2.7.15 installation (in particular, `sklearn`).

The final file should be a report in PDF format called `report.pdf`.

Grading

The grading breakout is as follows. One quarter of the credit will be awarded based on the performance of your submitted classifier, measured by its correct classification rate on the test dataset (although, as noted above, we may run your classifier on modified tetrainingst datasets obtained by randomly sampling from the training dataset if it appears that your classifier is overfitted to the test dataset). Another quarter of the credit will be awarded based on the sophistication of your submitted classifier. Another quarter will be awarded based on how convincingly your report establishes that you thoroughly optimized your design. The last quarter will be awarded based on the clarity of your report, the extent to which it follows good data visualization principles, the general aesthetic of the report (and especially of the figures), and the extent to which you follow the submission rules outlined in this document.

Remarks:

1. The `sklearn` module contains many helpful routines for classification, including ones for logistic regression and support vector machines. It is available on the Python 2.7.15 installation on the `ecelinux` cluster.

2. Even if one restricts attention to support vector machines, there are many design decisions to be made, such as what pre-processing to perform, what features to use, whether to do linear or kernel-based SVM, what value of C to select, etc.

3. You are not required to use the entire training data set to train your classifier. When starting out, you may find it helpful to debug your code by training on a data set containing only a few points. Also, a support vector machine can take a long time to train on the full data set, especially if you use the kernel trick.

4. The images are 3136-dimensional (28x28 pixels with 4 channels). Most of these features are not useful for prediction, however. The Basu *et al.* paper describes some derived features that are very useful for classification. In particular, by converting from RGB to HSV encoding of the color (say, using the `colorsys` module) and classifying based on the mean and standard deviation of the HSV components, one can construct an excellent classifier using only a handful of features. For other useful features, see the Basu *et al.* paper.

5. One of the unfortunate properties of support vector machines is that they are not *scale invariant:* scaling a feature by a constant (while leaving the other features the same) across all of the training and test data and retraining the classifier can change the classification decision for some of the training points. Thus if a feature measures length, expressing this feature in meters vs. centimeters can make a different in the classifier's performance. One

would hope that the conversion factor would get "absorbed" into other factors and not ultimately change anything, but for support vector machines this is not the case. Given this, it is common practice to normalize each feature to be between 0 and 1 before training the classifier.

6. The `scipy.io.loadmat` Python routine is useful for reading `.mat` files into Python.

7. The routine `matplotlib.pyplot.imshow()` is useful for displaying images. When dealing with image-based data sets, it is a good practice to view several images get a feel for the data set.