ISTA 331 HW: LINEAR CLASSIFIERS

Due Wednesday, May 6, 11:59 PM

1.1. **Introduction.** In this homework, we explore three classification methods based on linear models. The first is *logistic regression*, which is a modification of the standard least-squares linear model. The others are two types of *support vector machine*.

All of these can be trained by stochastic gradient descent. (We didn't need iterative methods for ordinary least squares, but the logit transformation in logistic regression changes the optimal solution, and the result is no longer something we can calculate directly with linear algebra.)

Our application here is the famous MNIST handwritten digit data set, one of the classic benchmark problems in machine learning. The data is a collection of 70,000 images of handwritten numerical digits, 0-9, collected from US Postal Service scans. The objective is to train an algorithm that is able to correctly identify which digit an image represents. Here are a few examples of images from the data set:

1.2. **Instructions.** Create a module named `hw7.py`. Start your module with the following imports, then code up the functions as specified below, and upload your module to the D2L Hw7 assignments folder.

```
from sklearn.linear_model import SGDClassifier, LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as sio
```

1.3. **Testing.** Download `hw7_test.py`, mnist-original.mat, and auxiliary testing files and put them in the same folder as your hw7.py module. Each of the 6 functions not including `main` and `plot_probability_matrices` is worth 16% of your correctness score. `main` and `plot_probability_matrices` is worth 20% of your grade. You can examine the test module in a text editor to understand better what your code should do. The test module should be considered part of the spec.

1.4. **Documentation.** Your module must contain a header docstring containing your name, your section leader's name, the date, ISTA 331 Hw7, and a brief summary of the module. Each function must contain a docstring. Each docstring should include a description of the function's purpose, the name, type, and purpose of each parameter, and the type and meaning of the function's return value.

1.5. **Grading.** Your module will be graded on correctness, documentation, and coding style. Code should be clear and concise. You will only lose style points if your code is a real mess. Include inline comments to explain tricky lines and summarize sections of code.

1.6. **Collaboration.** Collaboration is allowed. You are responsible for your learning. Depending too much on others will hurt you on the tests. "Helping" others too much harms them in reality. Cite any sources/collaborators in your header docstring. Leaving this out is dishonest.

1.7. **Resources.**

- `https://www.openml.org/search?type=data`
- `http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html`
- `http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`
- `https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`

2. Function specifications

- `get_data`: this function takes no arguments and returns two arrays: `X`, a $70,000 \times 784$ 2D array, and `y`, a 1D array with 70,000 elements.

  To see how to get the data, open a shell and type in these commands:

  ```
  import scipy.io as sio
  mnist = sio.loadmat('mnist-original.mat')
  ```

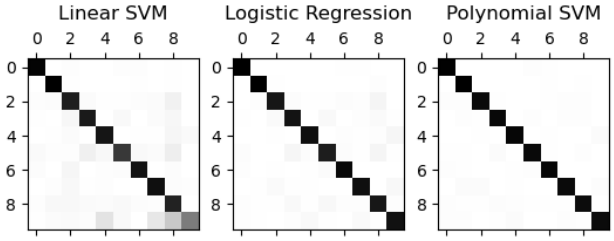
  Of course your current working directory will have to contain the file. There are other ways that you might figure out to load the data, but bear in mind that if your code won't run because a server is down, you get 0 points. Explore the `mnist` variable to find what you need to return as your `X` and `y`. To get the shapes of the two arrays right you might have to do a little work on them before returning them.
- `get_train_and_test_sets`: this function takes `X` and `y` as created by the previous function. The first 60,000 instances of each are for training, the rest for testing. The function breaks both into separate training and testing `X`'s and `y`'s. Use `np.random.permutation` to get a shuffled sequence of indices and then uses these indices to shuffle the training `X`'s and `y`'s. It then returns the training X, the testing X, the training y, and the testing y, in that order.
- `train_to_data`: this function takes a training X, a training y, and a string containing the name of the model. If the model name is `'SGD'`, make a `SGDClassifier` with `max_iter = 200` and a tolerance of 0.001 (Remember the default `SGDClassifier` is a linear SVM). If the model name is `'SVM'`, make a `SVC` with `kernel = 'poly'`. Otherwise, make a LogisticRegression with `multi_class = 'multinomial'` and `solver = 'lbfgs'`.

  Then, fit the model to training data. All of these classifiers expose a `fit` method that behaves like that for `DecisionTreeClassifier`, etc.
  - For the `SGDClassifier`, use the first 10,000 elements of the training set.
  - For the `LogisticRegression`, use the entire training set.
  - For the `SVC`, use the first 10,000 elements of the training set.

  This is just to save time. In real life you'd use all 60,000 elements of the training set, but `SVC` especially can be a bit slow fitting to larger data sets, and performs pretty well with only 10,000 examples. This makes the shuffling process really important, though, because if the training set is ordered then this sample of 10,000 will only contain 0's and 1's!

  Fit the model to the data and return it. (Note: the `LogisticRegression` will give a warning about the solver not converging. Don't worry about it.)
- `get_confusion_matrix`: this function takes a model, an `X`, and a `y`. Use the model's `predict` method to obtain predictions for this `X` and make a confusion matrix out of the `y` vector and your predictions. Return the matrix.

- `probability_matrix`: this function takes a confusion matrix and returns a probability matrix. Do not modify the original confusion matrix. Remember, row `i` of the confusion matrix contains the predictions for all of the digits that were actually `i`. Make and return a new matrix of floats where each number in position `[i, j]` is the estimated conditional probability that `j` was predicted given that `i` was the label (correct value). Round your probabilities to 3 decimal places (for prettier printing later).
- `plot_probability_matrices`: this function takes three probability matrices and produces a plot that looks like this when displayed (do not call `plt.show` in this function in your submitted version; you'll call `plt.show` in your `main` function instead.):



  You will need to use `plt.subplots` to create the layout and `axes.matshow` to display the matrices. This plot is worth 20 pts. It will show up when you run the test, but the test doesn't grade it. The test calls your `main`, which must work right for the plot to show up.
- `main`: get your data, split it into training and testing sets, and train a `SGDClassifier`, a `LogisticRegressionmodel`, and an `SVC` using your `train_to_data` function. Get confusion matrices for each of them, then probability matrices. Make your matrices plot. Put these lines in your code, which will print your matrices in a nice format:

```
for mod in (('Linear SVM:', probability_matrix(sgd_cmat)),
            ('Logistic Regression:', probability_matrix(soft_cmat)),
            ('Polynomial SVM:', probability_matrix(svm_cmat))):
    print(*mod, sep = '\n')
```

  Here, of course, `sgd_cmat`, etc., are the confusion matrices from each of the three models. Rename variables as needed.

  Finally, call `plt.show`.

Once you have written these functions, you have completed the graded form of the assignment. However, to get a feel for the creatures you have created, I encourage you to try the following function, `plot_example`. This function takes three or four arguments: an `X`, a `y`, a list of three models (intended to be one of each class), and an `index` with a default value of `None`, and displays an image from the MNIST data set, along with its correct label and the three predictions. If you run this a few dozen times (try it in a loop in a Jupyter notebook) you can see what the real life data looks like, and some of the patterns on which the models fail. Sometimes they (esp. the linear SVM) make weird errors, sometimes they are bad at recognizing rare forms (e.g. crossed 7's), and sometimes the digits are just kind of messed up and hard to ID even for a human!

```
def plot_example(X, y, model_list, index = None):
    if index is None:
        index = np.random.choice(np.arange(X.shape[0]), 1)
```

```
example = X[index, :]
prediction_list = [model.predict(example) for model in model_list]
plt.matshow(example.reshape(28, 28), cmap=plt.cm.gray)
plt.title('Correct label: ' + str(y[index]) +
          '\nLinear SVM Prediction: ' + str(prediction_list[0]) +
          '\nLogistic Regression Prediction: ' + str(prediction_list[1]) +
          '\nPolynomial SVM Prediction: ' + str(prediction_list[2]))
plt.show()
```