CIS 419/519 Introduction to Machine Learning
Assignment 3

Due: November 5, 2014 11:59pm

Instructions

Read all instructions in this section thoroughly.

Collaboration: Make certain that you understand the course collaboration policy, described on the course website. You must complete this assignment **individually**; you are **not** allowed to collaborate with anyone else. You may *discuss* the homework to understand the problems and the mathematics behind the various learning algorithms, but **you are not allowed to share problem solutions or your code with any other students.** You must also not consult code on the internet that is directly related to the programming exercise. We will be using automatic checking software to detect academic dishonesty, so please don't do it.

You are also prohibited from posting any part of your solution to the internet, even after the course is complete. Similarly, please don't post this PDF file or the homework skeleton code to the internet.

Formatting: This assignment consists of two parts: a problem set and program exercises.

For the problem set, you must write up your solutions electronically and submit it as a single PDF document. We will not accept handwritten or paper copies of the homework. Your problem set solutions must use proper mathematical formatting. For this reason, we **strongly** encourage you to write up your responses using LaTeX. (Alternative word processors, such as MS Word, produce very poorly formatted mathematics.)

Your solutions to the programming exercises must be implemented in python, following the precise instructions included in Part 2. Portions of the programing exercise will be graded automatically, so it is imperative that your code follows the specified API. A few parts of the programming exercise asks you to create plots or describe results; these should be included in the same PDF document that you create for the problem set.

Homework Template and Files to Get You Started: The homework zip file contains the skeleton code and data sets that you will require for this assignment. **Please read through the documentation provided in ALL files before starting the assignment.**

Citing Your Sources: Any sources of help that you consult while completing this assignment (other students, textbooks, websites, etc.) ***MUST*** be noted in the your `README` file. This includes anyone you briefly discussed the homework with. If you received help from the following sources, you do not need to cite it: course instructor, course teaching assistants, course lecture notes, course textbooks or other readings.

Submitting Your Solution: We will post instructions for submitting your solution one week before the assignment is due. Be sure to check Piazza then for details.

CIS 519 ONLY Problems: Several problems are marked as "[CIS 519 ONLY]" in this assignment. Only students enrolled in CIS 519 are required to complete these problems. However, we do encourage students in CIS 419 to read through these problems, although you are not required to complete them.

All homeworks will receive a percentage grade, but CIS 519 homeworks will be graded out of a different total number of points than CIS 419 homeworks. Students in CIS 419 choosing to complete CIS 519 ONLY exercises will not receive any credit for answers to these questions (i.e., they will not count as extra credit nor will they compensate for points lost on other problems).

Acknowledgements: The neural net exercise has been adapted from course materials by Andrew Ng.

Assignment Version 20141022a

PART I: PROBLEM SET

Your solutions to the problems will be submitted as a single PDF document. Be certain that your problems are well-numbered and that it is clear what your answers are. Additionally, you will be required to duplicate your answers to particular problems in the `README` file that you will submit.

1 Probability decision boundary (10pts)

Consider a case where we have learned a conditional probability distribution $P(y \mid \mathbf{x})$. Suppose there are only two classes, and let $p_0 = P(y = 0 \mid \mathbf{x})$ and let $p_1 = P(y = 1 \mid \mathbf{x})$. A loss matrix gives the cost that is incurred for each element of the confusion matrix. (E.g., true positives might cost nothing, but a false positive might cost us \$10.) Consider the following loss matrix:

	$y = 0$ (true)	$y = 1$ (true)
$\hat{y} = 0$ (predicted)	0	10
$\hat{y} = 1$ (predicted)	5	0

(a) Show that the decision $\hat{y}$ that minimizes the expected loss is equivalent to setting a probability threshold θ and predicting $\hat{y} = 0$ if $p_1 < \theta$ and $\hat{y} = 1$ if $p_1 \geq \theta$.

(b) What is the threshold for this loss matrix?

2 Double counting the evidence (15pts)

Consider a problem in which the binary class label $Y \in \{T, F\}$ and each training example $\mathbf{x}$ has 2 binary attributes $X_1, X_2 \in \{T, F\}$.

Let the class prior be $p(Y = T) = 0.5$ and $p(X_1 = T \mid Y = T) = 0.8$ and $p(X_2 = T \mid Y = T) = 0.5$. Likewise, $p(X_1 = F \mid Y = F) = 0.7$ and $p(X_2 = F \mid Y = F) = 0.9$. Attribute X_1 provides slightly stronger evidence about the class label than X_2.

Assume X_1 and X_2 are truly independent given Y. Write down the naive Bayes decision rule.

(a) What is the expected error rate of naive Bayes if it uses only attribute X_1? What if it uses only X_2?

The expected error rate is the probability that each class generates an observation where the decision rule is incorrect. If Y is the true class label, let $\hat{Y}(X_1, X_2)$ be the predicted class label. Then the expected error rate is $p(X_1, X_2, Y \mid Y \neq \hat{Y}(X_1, X_2))$.

(b) Show that if naive Bayes uses both attributes, X_1 and X_2, the error rate is 0.235, which is better than if using only a single attribute (X_1 or X_2).

(c) Now suppose that we create new attribute X_3 that is an exact copy of X_2. So for every training example, attributes X_2 and X_3 have the same value. What is the expected error of naive Bayes now?

(d) Briefly explain what is happening with naive Bayes (2 sentences max).

(e) Does logistic regression suffer from the same problem? Briefly explain why (2 sentences max).

3 Reject option (CIS 519 ONLY – 10pts)

In many applications, the classifier is allowed to "reject" a test example rather than classifying it into one of the classes. Consider, for example, a case in which the cost of misclassification is \$10 but the cost of having a human manually make the decision is only \$3. We can formulate this as the following loss matrix:

	$y = 0$ (true)	$y = 1$ (true)
$\hat{y} = 0$ (predicted)	0	10
$\hat{y} = 1$ (predicted)	10	0
reject	3	3

(a) Suppose $p(y = 1|\mathbf{x}) = 0.2$. Which decision minimizes the expected loss?

(b) Now suppose $p(y = 1|\mathbf{x}) = 0.4$. Now which decision minimizes the expected loss?

(c) Show that in cases such as this there will be two thresholds, θ_0 and θ_1, such that the optimal decision is to predict 0 if $p_1 < \theta_0$, reject if $\theta_0 \leq p_1 \leq \theta_1$, and predict 1 if $p_1 > \theta_1$.

(d) What are the values of these thresholds for the following loss matrix?

	$y = 0$ (true)	$y = 1$ (true)
$\hat{y} = 0$ (predicted)	0	10
$\hat{y} = 1$ (predicted)	5	0
reject	3	3

PART II: PROGRAMMING EXERCISES

1 Naive Bayes (25 pts)

In the first implementation exercise, you will implement two versions of naive Bayes, one for batch learning and the other for online learning. We will then use your naive Bayes implementation in the second exercise in an application to text processing.

Relevant Files in Homework 3 Skeleton[1]

- **naiveBayes.py**
- `test_naiveBayes.py`
- `test_onlineNaiveBayes.py`

1.1 Implementing Batch Naive Bayes

Implement a multinomial naive Bayes classifier in the `NaiveBayes` class in `naiveBayes.py`. Your implementation should support Laplace smoothing. Whether or not to use Laplace smoothing is controlled via an argument to the constructor; Laplace smoothing is enabled by default. Your implementation must follow the API below. (Note that all matrices are actually 2D numpy arrays in the implementation.)

- `__init(useLaplaceSmoothing=True)__` : constructor
- `fit(X,Y)`: method to train the naive Bayes model
- `predict(X)`: method to use the trained naive Bayes model for prediction
- `predictProbs(X)`: outputs a matrix of predicted posterior class probabilities

The training data for multinomial naive Bayes is specified as feature counts: `X`[i,j] is the number of times feature j occurs in instance i (or you can think of it as that instance i is characterized by a particular real-valued amount of feature j). For details of the multinomial naive Bayes classifier, see the start of the lecture on text processing.

Although you we aren't actually dealing with such data sets in this problem, for simple data sets with multi-valued discrete features (e.g., the Tennis play/don't play dataset), in order to use them in this classifier, we must first convert them to a set of binary features. (e.g, output = {sunny, overcast, rainy} to three features: isSunny?, isOvercast?, isRainy?). The i^{th} instance `X`[i,:] is then a binary vector for the presence or absence of each feature value.

The `predictProbs(X)` function takes in a matrix X of n instances and outputs an $n \times K$ matrix of posterior probabilities. Each row i of the returned matrix represents the posterior probability distribution over the K classes for the i^{th} training instance. (Note that each row of the returned matrix will sum to 1.)

Run `test_naiveBayes.py` to test your implementation. Your naive Bayes should achieve ~89% accuracy.

[1] **Bold text** indicates files or functions that you will need to complete; you should not need to modify any of the other files.

1.2 Online Learning with Naive Bayes

Once your naive Bayes model is working correctly, extend it to learn *online* by completing the `OnlineNaiveBayes` class in `naiveBayes.py`. So far, we have only examined batch learning methods, which process the entire training set in one batch to produce the model. In contrast, online learning methods receive data instances consecutively, updating the model each time. In most cases, after having seen the same set of training instances, the online learning algorithm will produce the same model as the batch learning algorithm, but it will have only seen the instances one at a time.

Essentially, the idea is that the `.fit()` method will be called multiple times, once for each training instance. At any time, we can call the `.predict()` method to use the classifier to predict labels for new instances. For example,

```
...
model = OnlineNaiveBayes()
model.fit(Xtrain[1,:], ytrain[1])
model.fit(Xtrain[2,:], ytrain[2])
print model.predict(Xtest)
model.fit(Xtrain[3,:], ytrain[3])
print model.predict(Xtest)
...
```

Unlike in the batch learning setting, calling `.fit()` does not overwrite the original model, but instead it updates it with the new training instance.

For convenience, we will actually write a hybrid approach, in which one or more training instances can be fed into `.fit()`. This form will support different types of training paradigms: (a) batch learning, where `.fit()` is called once with all training instances, (b) online learning, where `.fit()` is called numerous times, once for each training instance, and (c) a hybrid approach where `.fit()` is called several times, each time with a few training instances.

Naive Bayes is particularly easy to convert for online learning, since all we're doing is updating counts in each CPT. The only tricks are figuring out how to handle Laplace smoothing as you go and how to add instances with never-before-seen labels online! (You must figure those out yourself).

Your implementation must follow the same API as `NaiveBayes`. Although you are permitted to implement it entirely separately (e.g., duplicate `NaiveBayes` and edit it to learn online), it might be a better idea to implement it as a subclass of your Naive Bayes classifier. That way, if you discover any issues in your naive Bayes classifier, you only need to edit `NaiveBayes` to fix it in both the batch and online versions of the classifier. Also, note that certain aspects, like the `.predict()` method, do not necessarily need to change between the two versions.

Run `test_onlineNaiveBayes.py` to test your implementation and compare it against your standard naive Bayes classifier. You should see that the two algorithms are able to achieve the same performance.

2 Text Classification and ROC (20 points)

In this section, we'll apply naive Bayes and support vector machines to the problem of document classification. Sklearn provides a number of utilities that make text processing easy! For a tutorial, see http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html. Read through the tutorial first, and then come back to this document.

Welcome back! In class, we discussed using two algorithms for text classification: (a) naive Bayes and (b) SVMs with a cosine similarity kernel. Write a python program that reads evaluates both of these classifiers on the 20 newsgroups data set, outputting a number of performance metrics and an ROC plot of both classifiers. Here are the requirements for the program:

- Name your program `textShowdown20News.py`, and make sure the entire program is contained in this file.
- Your program should load both the training and testing portions of the 20 newsgroups data set (see http://scikit-learn.org/stable/datasets/twenty_newsgroups.html)
- Your program will process the text to create TF-IDF feature vectors, train models on the training data (optimizing parameters as needed), and evaluate performance on the test data. Note that we're only using one training/testing split.
- You may use your own naive Bayes implementation, or you may use sklearn's built in `MultinomialNB` classifier. You should use the SVM and cosine similarity kernel implementations provided with sklearn.
- Your program should output a table of the following metrics for both classifiers: (a) train and test accuracy (b) train and test precision (c) train and test recall (d) training time. Make sure the table is neat and clear.
- Your program should also produce an ROC plot that contains curves for both classifiers, and output that plot to the file `graphTextClassifierROC.pdf`
- Ensure that your program does not produce any other output when it is run. It is fine to add debugging or status output while you're developing the program, but remove this output before submission.

Recall from class that for document classification, we often do better if we use TF-IDF features instead of the bag of words representation (i.e., a feature vector of raw word counts). For the text processing aspects of this problem, be sure to follow these guidelines:

- Remove stop words (using the default english stop word list in `CountVectorizer`),
- Lowercase all terms,
- Compute the dictionary and inverse document frequency over the training data only, then use the same preprocessing values for the test data,
- Use log-scaled term counts for the term frequency, and
- Normalize the final TF-IDF feature vectors to have unit length.

Run your program on the 20 newsgroups data set. Include your table of performance statistics and your ROC plot in your PDF writeup. Which classifier is better? Write 2-3 sentences justifying your answer, discussing the results you obtained.

While writing this program, you may find it useful to reference the following websites:

- http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html
- http://scikit-learn.org/stable/auto_examples/plot_roc.html
- http://scikit-learn.org/stable/modules/metrics.html#cosine-similarity

3 Neural Networks (30 pts)

Relevant Files in Homework 3 Skeleton

- **nn.py**
- `digitsVisualization.tiff`
- `data/digitsX.dat`
- `data/digitsY.dat`

In this problem, you will implement an artificial neural network (NN) classifier that learns via back-propagation. Instead of choosing the architecture of the network at implementation time, you should implement your neural net in a more general manner; the specific architecture of the network will be given to the constructor. Recall from class that we can specify the architecture of a basic neural net via a vector $\mathbf{s}$, where each entry in the vector gives the number of nodes in that layer and the length of the vector is the total number of layers in the network. We will assume that within each layer, each node receives input from every other node in the previous layer.

Implement the neural network in a class called `NeuralNet` in `nn.py`. Your `NeuralNet` implementation should employ the API given below:

- `__init(layers, epsilon = 0.12, learningRate, numEpochs)__` : constructor

  `layers` – a vector of $L-2$ positive integers, where the number of layers in the network is L. The value contained in `layers[i]` specifies the number of nodes in the i^{th} hidden layer. Note that the specification of `layers` is a bit different from the $\mathbf{s}$ vector; in particular, it does not include s_0 and s_{L-1}, which are initialized from the training data in `fit()`.

  `epsilon` – one half the interval around zero for the initial weights (defaults to 0.12)

  `learningRate` – the learning rate for back-propagation

  `numEpochs` – the number of epochs for backpropagation

- `fit(X,Y)`: train the neural network model via backpropagation
- `predict(X)`: use the trained neural network model for prediction via forward propagation
- `visualizeHiddenNodes(filename)`: (CIS 519 ONLY) outputs an image representing the hidden layers

While implementing the neural net, I recommend you follow the steps outlined below.

3.1 Network Structure and Initialization

Complete the constructor, which simply saves the arguments for use later. Since the number of units in the first and last layer are specified by the training data, we cannot initialize the network architecture and weight matrices until `fit()`. After finishing `__init()__`, start writing `fit()`.

The first thing you should do in `fit()` is to create all of the weight matrices $\Theta^{(1)}, \ldots, \Theta^{(L-1)}$. Initialize all of the weights to be uniformly chosen from $[-\epsilon, \epsilon]$, where ϵ is specified by the `epsilon` argument to the constructor.[2] Then, unroll the weight matrices $\Theta^{(1)}, \ldots, \Theta^{(L-1)}$ into a single long vector $\boldsymbol{\theta}$ that contains all parameters for the neural net.

3.2 Forward-propagation

Implement a private function to perform forward-propagation. This method will be useful for both back-propagation and the `predict()` method. This private function should take in a vector of parameters (e.g., $\boldsymbol{\theta}$) for the neural network and an instance (or instances) and return the neural network's outputs.

3.3 Backpropagation

Finish the `fit()` method to use backpropagation to minimize $J(\boldsymbol{\theta})$. Details for implementing backpropagation are in the lecture slides. At a minimum, you'll need to compute $J(\boldsymbol{\theta})$ and $\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta})$.

[2] According to Andrew Ng, an alternative (and effective) strategy for choosing ϵ is to choose a different value for each layer's weights, with the ϵ for $\Theta^{(l)}$ as $\frac{\sqrt{6}}{\sqrt{s_l + s_{l+1}}}$, where $\mathbf{s}$ is the vector containing the number of nodes in each layer.

To confirm that your gradient computations are correct, I recommend that you implement the following gradient checking procedure to numerically estimate the gradient. Form two vectors:

$$\boldsymbol{\theta}_{i+c} \leftarrow \boldsymbol{\theta};\ \text{then}\ \ \theta_i \leftarrow \theta_i + c \qquad\qquad \boldsymbol{\theta}_{i-c} \leftarrow \boldsymbol{\theta};\ \text{then}\ \ \theta_i \leftarrow \theta_i - c\ . \tag{1}$$

In other words, $\boldsymbol{\theta}_{i+c}$ (or $\boldsymbol{\theta}_{i-c}$) is the same as $\boldsymbol{\theta}$, but the i^{th} element has been incremented (or decremented) by c. Then, we can numerically estimate the gradient and verify that our gradient computations are correct by confirming that

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{J(\boldsymbol{\theta}_{i+c}) - J(\boldsymbol{\theta}_{i-c})}{2c}\ . \tag{2}$$

Setting $c = 10^{-4}$, you should find that the computed gradient and the estimated gradient should agree to at least four significant digits.

This gradient checking procedure is very expensive, and so should only be used for small networks (small numbers of parameters). **Make absolutely certain to disable the gradient checking procedure once you're certain that your gradient implementation is correct.** Use the gradient checking procedure only for debugging purposes; be sure to disable it before running your learning algorithm.

Once your gradient computations are confirmed to be correct, finish `fit()` to run backpropagation for the number of epochs specified in the constructor to train the neural net.

3.4 Complete the Prediction Function

Your prediction function should call the forward-propagation method with the neural net parameters $\boldsymbol{\theta}$ trained via backpropagation in `fit()`.

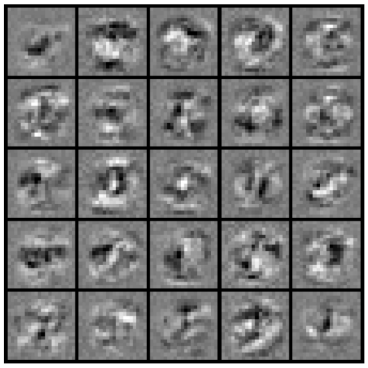
3.5 Apply Your Neural Network to Digit Recognition

Write a test script named `testNeuralNetDigits.py` to apply your neural network classifier to the problem of digit recognition. The homework skeleton contains a data set of 5,000 20×20 digit images (see `digitsVisualization.tiff` for a visualization). We can represent each image as a 400-dimensional vector of pixel intensities. The features for the digits are provided in the `data/digitsX.dat` file and their corresponding labels are in `data/digitsY.dat`.

Train your neural network on the digits data with one hidden layer of 25 nodes over 100 epochs. Tune the learning rate for the neural network. You should be able to get a training accuracy of approximately 95.3% or higher if your implementation is correct ($\pm 1\%$ due to random initialization). Report your optimal learning rate and the maximum training performance you obtained in your PDF writeup and README.

3.6 Visualizing the Hidden Layers (CIS 519 ONLY – 10 pts)

Complete the `visualizeHiddenNodes(filename)` function to visualize the hidden units in the network. For the neural network you trained above, note that the i^{th} row of $\Theta^{(1)}$ is a 401-dimensional vector that specifies the parameters for the i^{th} hidden unit. Discarding the bias term yields a 400-dimensional vector that we can reshape into an 20×20 matrix via the `numpy.reshape` command. If we remap[3] the values of this matrix to lie in $0 \ldots 255$, we can visualize the weights as a greyscale image.



Use the Python Image Library to create an image that visualizes all of the hidden layers. Separate the layers into blocks (e.g., you can visualize a 25 unit layer as an 5×5 grid, where each grid entry is the 8×8 greyscale image). You might find it useful to consult http://en.wikibooks.org/wiki/Python_Imaging_Library/Editing_Pixels. Save this image to the filename given as an argument to `visualizeHiddenNodes()`.

You should find that the hidden units correspond to different stroke detectors and other patterns in the input. For an example of the hidden unit visualization, see the image to the right. Yours will likely look slightly different from this one due to randomization. Include your output image visualizing the hidden layers of your network in your PDF writeup.

[3] I suggest you either map -1 to 0 and +1 to 255, or the min value over all units to 0 and the max value over all units to 255 – whichever gives you the nicer picture.