Final Project

Due May 9

For the final project, you will take a data set and use two classification approaches to distinguish classes in your data. For one of the classifications, you will be required to implement the classifier **and** introduce adjustments to the classifier in an attempt to optimize its performance. This is a group project with 2-3 person.

For the project you must:

- Program functions to
  - **extend** at least one classification and learning method (the second method can be taken from publicly available software but MUST be credited as such)
- Experiment with
  - two learning/classification methods
  - optimization of one learning/classification method
- Report on your methods, results (accuracy and run-time), and conclusions in an 8-15-page paper

Your grade will be calculated as follows:

- 40% for code to implement a new classifier and optimizations of the classifier
- 30% for your report's discussion of your implementation (how the code work) and justification of any design/optimization choices you made
- 30% for your report's presentation of results (accuracies and run-times) and conclusions

The data set:

You may choose your own data set. However, you may be interested in using any of the three data sets listed below.

Online News Popularity:

You can use the "Online News Popularity" data set provided by University of California, Irvine. It uses 58 features to predict the number of times a web page is shared — ignoring the first two features and using the last feature as class indicator. Read over the documentation for the data set on the Irvine web site. All feature values in our .mat file are the same as the feature values listed in the documentation.

Note there is no pre-determined "class" label; instead the final "feature" counts the number of visits for each page. **You will have to pick a threshold for high versus low page shares — you are free to experiment with this threshold as well.** To start, I suggest trying 1400 as share threshold.

SPECTF Heart Data Set:

You can use the "SPECTF Heart Data Set" data set provided by University of California, Irvine. It uses 44 features to predict the presence of a heart condition — using the first feature as class indicator. Read over the documentation for the data set on the Irvine web site. All feature values in our .mat file are the same as the feature values listed in the documentation.

You may download the data directly from the UC Irvine site, using the CSV files SPECTF.train and SPECTF.test.

Adult data set:

You can use the "Adult" data set provided by University of California, Irvine. It uses 14 features to predict whether a person makes over $50K per year. Read over the documentation for the data set on the Irvine web site. All feature values in our .mat file are the same as the feature values listed in the documentation, **except** the class labels have been converted to 0 and 1 in our .mat file, as described below.

The learners/classifiers:

You must **implement and expand** one of the following. For your second learner/classifier, you may use a previously-implemented version of another of these methods; the **second** method may be taken from a Matlab toolbox or Python package, or downloaded from the internet):

- Support vector machine
- Neural Network
- Bayes Network
- PCA, ICA, or NMF in combination with a previously implemented learner/classifiers
- Feature selection/removal in combination with Bayes classifier

Key details on implementation for the traditional and expanded version of each method are provided at the bottom of this document.

Experiments with settings/hyper-parameters for each method:

As we have discussed in class, each classification/learning method potentially has a variety of settings and hyper-parameters to manipulate. Possible settings and hyper-parameters include:

- Initialization values for parameters
- Regularization strength
- Update step size
- Maximum number of permitted repeated iterations on training data
- Slack variable strength
- Number of neuron units

- Size of training data set (you will have to divide the data into testing and training sets; note, testing data set should stay the same for all learning conditions to ensure consistent evaluation)

For each of the two classifier/learner methods, you are to experiment with at least two settings or hyper-parameters and report their effects on accuracy on test data and run-time for learning.

For each classifier/learner method, you should try at least five different values per setting/hyper-parameter. For example, using gradient ascent to learn the SVM separator, you may vary ε step size and tolerance of error ("slack") C. You can evaluate learning accuracy based on values:

ε	0.001	0.01	0.1	1	10	0.01	0.01	0.01	0.01
C	1	1	1	1	1	0.1	0.001	10	100

This would constitute 5 different values for each of two hyper-parameters.

<u>The optimized/"improved" version for one of the methods:</u>

In this step, you need to choose one of the models you selected to make it a sophisticated version. The expanded version of each method is provided at the bottom of this document.

You can choose the methods from the bottom list and you are also encouraged to make new optimization methods yourselves.

In your report, you need to:

- Illustrate how you experiment your optimization method on the original model.
- Explain how the optimized model improves the performance of the original model, e.g, the accuracy of the model, the speed of modeling
- Illustrate the advantage/disadvantage of your optimized model compared to your original model.

<u>Graded materials:</u>
You must submit: (1) Your complete Python code, (2) Your 8–15 page report

Your code must include:

- Commands you used to load/clean your data set, train your classifiers and test your classifiers. You may have a simple function to perform all these steps, or you may record these commands without a wrapping function
- All relevant Methods/functions to implement/expand/optimize upon a learning and classification method.
- A readme text file that explains how to run the functions you have written.

Your report must include:

- Introduction: Summary of the data and the methods you tried, and a brief preview of your final conclusions
- Methods: Which methods you tried, how you adjusted the standard method to try to make it "better", what settings/hyper-parameters you used.
- Results: Discuss the effects of using different learning methods, the effects of the changes you made to the traditional methods, and the effects of the learning settings/hyper-parameters. You must include at least one table/graph. More tables/graphs are welcome!
- Conclusion: Comment on the take-away messages you have gotten from your experiments.

Time commitment: This project should take you at least 50 hours over a month span.

Due date: The project will be due May 9.

Continue to the next page to read the details about the traditional implementation and improvements you are expected to make for your selected method

Learning/classification methods in detail:

You need to choose one of the five learning/classification models to implement and to further improve.

1) SVM:

- **Traditional "dual" kernel implementation**: Write function to find $\boldsymbol{\alpha}$ for support vectors. Run gradient ascent-or-descent to maximize:

$$\underset{\alpha,\gamma,\lambda}{\operatorname{argmax}} \sum_j \alpha^j - \frac{1}{2}\sum_{i,j} y^i y^j \alpha^i \alpha^j K(\boldsymbol{x}^i, \boldsymbol{x}^j) - \gamma_1 \max\left(0, \sum_j \alpha^j y^j\right) - \sum_j \lambda_j \max(0, \alpha^j)$$

Note this optimization is mathematically derived from the version in homework 2, with some modifications, more commonly called the "dual" form of SVM optimization. The $\max(0, \sum_j \alpha^j y^j)$ term pushes for $\sum_j \alpha^j y^j = 0$ and the $\max(0, \alpha^j)$ terms push for $\alpha^j \geq 0$.

Use at least the following two kernel definitions:

$K(\mathbf{x}, \mathbf{v}) = \mathbf{x}^T\mathbf{v}$ and $K(\mathbf{x},\mathbf{v}) = \exp\left(-\frac{1}{2\sigma^2}(\boldsymbol{x} - \boldsymbol{v})^2\right)$

You also may choose to use the "logistic loss" function instead of the max function: Instead of max(0,z) , use logloss(z)=log$(1 + e^z)$

- **Improvements:** add and test new constraints to the minimization, such as slack variables ($+C\sum_j \xi^j$) or alternative slack constraints (e.g., $+C\sum_i \xi^{i^2}$) or L1 constraint ($+\lambda\sum_j \text{sign}\, w_j$), implement new kernel functions. You also could implement "gradient ascent with momentum" (see bottom of document).

2) Multi-layer neural networks

- **Traditional implementation:** Write function to classify based on neural network parameters. Run gradient ascent to minimize neuron output error, following ascent rule given in class.

- **Improvements:** Incorporate "neural pruning" – after K learning iterations, remove neurons with low weights from each layer. Incorporate neural "competition" – the output of each neuron incorporates both its inputs **and** a constant suppression from the output of neighboring neurons. Replace sigmoid output rule with step function $g(h) = \begin{cases} 0 & h < 0 \\ 1 & h \geq 0 \end{cases}$, rectify-linear $g(h) = \begin{cases} 0 & h < 0 \\ h & h \geq 0 \end{cases}$, or hyperbolic tangent $g(h) = \frac{e^{2h}-1}{e^{2h}+1}$ and adjust the update rule. You also can implement "gradient ascent with momentum" (see bottom of document).

3) Implement Bayesian network classification and learning: You will have to investigate the structure and implementation of Bayes nets yourself and implement it from scratch using Python

4) Component analysis (Pick at least one from PCA, ICA, or NMF):

For method 3, you must:

- **feed the resulting reduced feature set to another classifier (e.g., logistic classifier)**
- **show the average reconstruction error using 1 through 10 principle components**

- "**Traditional" implementation (simplified):**

PCA:

Convert all features to have zero mean ($\mu = 0$) and unit variance ($\sigma = 1$)

For component q=1:Q:

Find u^q to minimize:

$$\underset{\boldsymbol{u}}{\operatorname{argmin}} \sum_i \sum_j \left(\boldsymbol{x}_j^i - \left(\boldsymbol{u}^{q^T} \boldsymbol{x}^i\right)\boldsymbol{u}_j^q\right)^2$$

Remove u^q from x^i: $\boldsymbol{x}^i \leftarrow \boldsymbol{x}_j^i - \left(\boldsymbol{u}^{q^T} \boldsymbol{x}^i\right)\boldsymbol{u}_j^q$

ICA:

Convert all features to have zero mean ($\mu = 0$) and unit variance ($\sigma = 1$)

Repeatedly loop:

Find u^q , z_q^i to minimize:

$$\underset{\boldsymbol{u},\boldsymbol{z},\gamma}{\operatorname{argmin}} \sum_i \sum_j \left(\boldsymbol{x}_j^i - \sum_q z_q^i \, \boldsymbol{u}_j^q\right)^2 + \gamma \sum_{i,q} \left|z_q^i\right|$$

Note: this is simpler than the traditional ICA objective function, but it is still a worthwhile exercise in dimensionality reduction through optimization.

NMF:

Convert all features to be non-negative

Repeatedly loop:

Find u^q , z_q^i to minimize:

$$\underset{\boldsymbol{u},z,\gamma}{\operatorname{argmin}} \sum_i \left(\sum_j \left(x^i - \sum_q z_q^i \boldsymbol{u}_j^q\right)^2\right) - \gamma \sum_{j,q} \boldsymbol{u}_j^q - \gamma \sum_{i,q} z_q^i$$

Set all negative u and z values to 0.

- **Improvements:** Merge some of the rules from two methods together to create a new method; add L1 or L2 regularization; add mapping functions/kernels For ICA or NMF, you also can implement "gradient ascent with momentum" (see bottom of document).

5) Bayes learning AND feature selection:

Bayes learning: Extend Laplacian learner/classifier from HW1 to fit models for at least three other probability distributions. Example distributions can be found here: https://en.wikipedia.org/wiki/List_of_probability_distributions#Continuous_distributions

. I would suggest Gaussian, Uniform, and Rayleigh . You may be able to find closed-form ways to calculate your probability parameters, or you may need to use gradient ascent.

Feature selection/removal

- **Traditional:** Greedily add/remove best feature, then add/remove best remaining feature, etc. If a feature has been added/removed, assume it is the optimal strategy to keep it always/have it permanently removed.
- **Improvements:**
    1) Greedily select/remove features in pairs or in triples; once a pair or triple is added/removed, never retract this decision.
    2) Greedily select/remove one at a time, but after every *k* steps randomly remove a previously-added feature/add back a previously-removed feature. You can place higher probability on reversing features selected/removed that had smallest impact on accuracy.

Extra note:
Here is guidance if you wish to implement "gradient descent with momentum". If the normal update rule for parameter θ at iteration t is $\theta^{t+1} \leftarrow \theta^t + \varepsilon_1 \frac{dL^t}{d\theta}$, the "momentum" learning rule is: $\theta^{t+1} \leftarrow \theta^t + \varepsilon_1 \frac{dL^t}{d\theta} + \varepsilon_2 \frac{dL^{t-1}}{d\theta}$, with $\varepsilon_2 > \varepsilon_1$ (e.g., $\varepsilon_2 = 0.1, \varepsilon_1 = 0.01$. Each update is computed based on the derivative at the present iteration **plus** the derivative at the previous iteration.