CS 4641: Machine Learning
Homework 2 — Classification

Instructions

Submission

Please submit **one** zipfile containing a folder with **all** your code, data files, and a **single** pdf with your answers to the written questions. This zipfile should have the following format: `hw2-GTusername.zip`, replacing "GTusername" with your username (for example "hw2-bhrolenok3.zip").

For this assignment, your submission should include:

- `hw2-answers.pdf` A PDF of your answers to the written questions.
- `hw2.py` The template file, modified to include your code.
- `README` Any special instructions for running your code, (library versions, network access, etc) **plus** any external sources you used to help you with this assignment. This includes other students and websites. When in doubt, cite it!

Collaboration

All of the assignments in this class are **individual work only**. You're welcome to *discuss* the math or high level concepts behind the algorithms you're being asked to implement, but you are **not allowed to share code or answers to the written questions**. We will be checking, manually and with automated tools, for any violations of this policy, so please **do not share solutions or code**.

Debugging tips

In order to verify that your learners are working correctly, you can compare your results with those from a library such as `scikit-learn`[1]. For a quick debugging check, running the `main` method in the skeleton file should give you performance results for a simple debugging dataset that should have relatively high accuracy for both k-NN (0.05-0.1 loss) and LogisticRegression (0.85 score). LogisticRegression should converge after 44389 iterations for the given parameters.

k-Nearest Neighbors

One of the simplest classifiers we've discussed also happens to be one of the most powerful, and surprisingly straightforward to implement. The short description in words is: "return the class label of the most common class amongst the k nearest training points," but for the sake of completeness, here's a formal description:

[1] `https://scikit-learn.org/`

Data: query point $\mathbf{x}$, distance function $d(\mathbf{x}, \mathbf{x}')$, number of neighbors k
Result: estimate of the label for $\mathbf{x}$, $\hat{y}$
Compute the distances between $\mathbf{x}$ and all of the training points $\mathbf{x}^{(i)}$
Find the k closest points, $\{\mathbf{x}^{(n_1)}, \mathbf{x}^{(n_2)}, \ldots, \mathbf{x}^{(n_k)}\}$ $(l < m \implies d(\mathbf{x}, \mathbf{x}^{(n_l)}) \leq d(\mathbf{x}, \mathbf{x}^{(n_m)}))$
Return the most frequent class in $\{y^{(n_1)}, y^{(n_1)}, \ldots, y^{(n_k)}\}$

In practice, efficient methods exist ($k-d$ trees, locality-sensitive hashing, etc) for doing quick ($O(\log N)$ vs $O(N)$) lookup that work by building data structures that take advantage of the properties of distance functions, but for this assignment we'll stick with the brute-force approach. Implement the algorithm above in the method `KNN.query_single_pt`.

Distance functions

There are three distance functions you should implement in `hw2.py`: `KNN.euclid`, `KNN.manhattan`, and `KNN.mahalanobis`, and their definitions are

Euclidean distance

$$d(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{j=1}^{D} (x_j - x'_j)^2} \tag{1}$$

Manhattan distance

$$d(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^{D} |x_j - x'_j| \tag{2}$$

Mahalanobis distance

$$d(\mathbf{x}, \mathbf{x}') = \sqrt{(\tilde{\mathbf{x}} - \tilde{\mathbf{x}}')^\top \Sigma^{-1} (\tilde{\mathbf{x}} - \tilde{\mathbf{x}}')} \tag{3}$$
$$\tilde{\mathbf{x}} = (\mathbf{x} - \bar{\mathbf{x}}), \quad \Sigma = \text{diag}(\sigma_1^2, \sigma_2^2, \ldots, \sigma_D^2)$$

Where $\bar{\mathbf{x}}$ is the average of all the training points, and σ_j is the standard deviation of the jth component. Note that what we're calling the Mahalanobis distance is actually a form of centering and re-scaling[2], which is still useful for making sure that all of our features are comparable at the same scale. You can implement this either with the linear algebra representation shown above, or by directly subtracting out the mean and rescaling each component iteratively. Implement each of these distance functions using the equations above. Since this computation is the bottleneck for k-NN, you may wish to spend some time trying to optimize these functions, particularly by making use of NumPy's ability to do quick vector operations.

Choosing k

While the choice of distance function definitely has an impact on the complexity of the decision boundary, the relationship is problem-dependent and not always entirely clear. The choice of k, however, has a direct connection with complexity: lower k means increased complexity. We already know that increasing complexity reduces bias at the expense of increasing variance, so we should think carefully about how to pick the best value of k for a given problem.

One way to do this is to look at how error on the training data and error on some (withheld) testing data changes as a function of k. The provided code already has functions which will

1. Return a set of testing and training data from a given file
2. Compute the misclassification rate for a given value of k on a set of testing data

[2]The true Mahalanobis distance requires the covariance matrix of the underlying distribution of the data.

3. Compute the misclassification rate for a given value of k on the training data

You will need to use these methods to explore a reasonable range of values for k for the provided datasets. The format of their return values should make it easy to plot misclassification error as a function of k, but you will have to write the code to do this yourself.

Important note: these types of experiments can take a long time to run, so don't leave this till the last minute! If even after you've optimized your implementation you're still having trouble running these experiments in a reasonable amount of time, you can modify the provided code to reduce the size of the datasets, but you **must** make a note of this both in your code and in your writeup, and provide an explanation of how you reduced the data and why what you did makes sense.

Logistic Regression

The Logistic Regression classifier directly models the probability that a given point has a given label. Though versions exist which model more than two labels, we will focus on a restriction of the problem to only binary labels, and use what's known as a "one versus rest" approach to handle problems with multiple classes. The form that the model takes is

$$p(y = 1 \mid \mathbf{x}) = h_\theta(\mathbf{x}) = \sigma(\theta^\top \mathbf{x}) = \left(1 + \exp(-\theta^\top \mathbf{x})\right)^{-1} \tag{4}$$

Fill in the method `LogisticRegression.prob_single_pt` which should implement the above equation. Note that the provided `sigmoid` function has built in checks to make sure the output never reaches 0 or 1, which can cause issues with some of the computations performed later. If you see `NaN`'s or `inf`'s show up in your answers, make sure you're not encountering numerical overflow or underflow.

We discussed in class some of the nice properties that the sigmoid function ($\sigma(z)$) has, chief among them being the fact that the gradient is especially easy to compute. The gradient of the negative log likelihood is given as

$$\nabla_\theta NLL(h_\theta; S) = \sum_{i=1}^{N} \left[h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\right] \mathbf{x}^{(i)} \tag{5}$$

$$= X^\top (\mathbf{h}_\theta - Y) \tag{6}$$

where X is the $N \times (D + 1)$ matrix of (1's augmented) training data points, $\mathbf{h}_\theta$ is the column vector of predicted outputs for each of the training points, and Y is the column vector of 1's and 0's such that each entry is 1 if the label matches the positive class and 0 otherwise.

Given this gradient, we can compute the optimal parameters by using gradient descent. The function for gradient descent has been provided to you in the template code, the only thing that you need to do is fill in `LogisticRegression.NLL_gradient` so that it returns the actual gradient for the current value of the parameters in `self.Theta`. You may want to look at how the function `LogisiticRegression.negative_log_likelihood` is implemented for ideas on how to use fast NumPy vector operations to compute the gradient.

Once you have these two functions implemented, you should be able to compare the performance of your logistic regression model with k-NN. For single-class problems, you can threshold the probability at 0.5 and use the resulting binary labels to compute the misclassification rate. For multi-class problems, you should fit one learner per class: you can specify the positive class label in the constructor to `LogisticRegression`, and when training this should treat any class label that's not the positive class label as a negative class label. Then, when testing, pick the positive class label of the logistic regression model with the highest probability. This is implemented for you in the `multiclass_logistic_score` function.

Support Vector Machines

Support Vector Machines are another linear classifier, but unlike logistic regression, they are built on the "maximum margin" principle. The classifier itself has a similar form to logistic regression, but replacing the

sigmoid function with a hard threshold

$$h_{\mathbf{w},w_0}(\mathbf{x}) = \text{sgn}(\mathbf{w}^\top \mathbf{x} + w_0) \tag{7}$$

Replacing the sigmoid function with a hard threshold makes it so we can't use the same nice gradient descent we've been using all along, but it turns out there is another (quadratic, constrained) optimization problem we can solve to find the best weights:

$$\alpha = \arg\max_\alpha \sum_{i=1}^N \alpha^{(i)} - \frac{1}{2}\sum_{i=1}^N \sum_{j=1}^N \alpha^{(i)}\alpha^{(j)} y^{(i)} y^{(j)} \mathbf{x}^{(i)\top}\mathbf{x}^{(j)} \tag{8}$$

Subject to

$$\sum_{i=1}^N \alpha^{(i)} y^{(i)} = 0, \quad \alpha^{(i)} \geq 0, \quad \forall \alpha^{(i)}$$

where the original parameters can be recovered as

$$\mathbf{w} = \sum_{i=1}^N \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)} \tag{9}$$

While the equations look complex, they're describing a very simple geometric picture: the best line for separating the data is the line which *maximizes* the distance between itself and the data. That way, any noise in the training data is less likely to change the predicted class label. In addition to this nice geometric interpretation, SVMs also allow us to easily perform something akin to basis function expansion but for classification instead of regression. We achieve this by replacing the inner product $\mathbf{x}^\top \mathbf{x}'$ with a function k such that

$$k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle = \phi(\mathbf{x})^\top \phi(\mathbf{x}')$$

For example, if we wanted to compute the linear SVM for 1D data that had been transformed so that $\phi(\mathbf{x}) = [x_1^2, \sqrt{2}x_1, 1]^\top$ we could use the following kernel

$$\begin{aligned} k(\mathbf{x}, \mathbf{x}') &= \phi(\mathbf{x})^\top \phi(\mathbf{x}') \\ &= (x_1 x_1')^2 + 2(x_1 x_1') + 1 \\ &= (x_1 x_1' + 1)^2 \end{aligned}$$

For $\mathbf{x}$ in two dimensions, including all the cross terms gives us $\phi(\mathbf{x}) = [x_1^2, x_2^2, \sqrt{2}x_1 x_2, \sqrt{2}x_1, \sqrt{2}x_2, 1]^\top$, which is in 6 dimensions, but the kernel function still operates in 2:

$$\begin{aligned} k(\mathbf{x}, \mathbf{x}') &= \phi(\mathbf{x})^\top \phi(\mathbf{x}') \\ &= (x_1)^2 (x_1')^2 + (x_2)^2 (x_2')^2 + 2x_1 x_2 x_1' x_2' + 2x_1 x_1' + 2x_2 x_2' + 1 \\ &= (x_1 x_1' + x_2 x_2' + 1)(x_1 x_1' + x_2 x_2' + 1) \\ &= (\mathbf{x}^\top \mathbf{x}' + 1)^2 \end{aligned}$$

This shows that instead of computing a potentially high-dimensional transform, we can use an easy to compute kernel function while still gaining the complexity of operating in higher dimensions. This process of replacing inner products with a function is known as the "kernel trick" throughout ML. You'll need to generalize the above result to answer the questions on SVMs at the end of this document.

Questions

1. k-Nearest Neighbors

   (a) Find a good setting for k for the datasets `HTRU`, `iris`, and `optdigits` using the `euclid` distance metric. Include plots (one for each dataset, three plots total) which show both testing and training error as a function of k, and explain your reasoning.

   (b) Repeat the previous question for the other two distance metrics, `manhattan`, and `mahalanobis` (so six more plots). Which had the best performance, and at which value of k?

2. Logistic Regression

   (a) Fit a Logistic Regression model to the dataset `HTRU`. Compare the performance with k-NN.

   (b) The other two datasets have more than one class. Fit a Logistic Regression model for each class (3 for `iris` and 10 for `optdigits`) and again compare the performance with k-NN.

   (c) In general, under what circumstances does Logistic Regression perform well? Looking at your results from the previous questions, do you think these conditions hold for any of the given datasets? Explain your reasoning.

3. Support Vector Machines

   (a) Show how you can re-write the model

   $$h_{\mathbf{w},w_0}(\mathbf{x}) = \text{sgn}\left\{w_0 + \phi(\mathbf{x})^\top \mathbf{w}\right\}$$

   so that the feature transform of the inputs, $\phi(\mathbf{x})$, never has to be explicitly computed. That is, show how you can go from the expression above to the one given in the slides. Explain why you *need* to do this to use the "kernelized" version of SVMs, and what role $\alpha^{(i)}$ plays in the efficiency of this computation.

   (b) How many dimensions would the feature transform $\phi(\mathbf{x})$ have if computed explicitly for the following kernels?

   $$\begin{aligned} k_{\text{lin}}(\mathbf{x}, \mathbf{x}') &= \mathbf{x}^\top \mathbf{x}' \\ k_{\text{poly}}(\mathbf{x}, \mathbf{x}') &= (1 + \mathbf{x}^\top \mathbf{x}')^M \\ k_{\text{exp}}(\mathbf{x}, \mathbf{x}') &= \exp\left\{-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right\} \end{aligned}$$

   Assume that the $\mathbf{x}$ have D components. (Hint for k_{exp}: consider the Taylor series expansion of e^z.)

   (c) (Murphy exercise 14.1) Consider a dataset with 2 points in 1d: $(x_1 = 0, y_1 = -1)$ and $(x_2 = \sqrt{2}, y_2 = 1)$. Consider mapping each point to 3d using the feature vector $\phi(x) = [1, \sqrt{2}x, x^2]^\top$. (This is equivalent to using a second order polynomial kernel.) The max margin classifier has the form

   $$\begin{aligned} \min &\|\mathbf{w}\|^2 \quad \text{s.t.} \\ y_1(\mathbf{w}^\top \phi(x_1) + w_0) &\geq 1 \\ y_2(\mathbf{w}^\top \phi(x_2) + w_0) &\geq 1 \end{aligned}$$

   Write down a vector that is parallel to the optimal vector $\mathbf{w}$. Hint: recall that $\mathbf{w}$ is perpendicular to the decision boundary between the two points in the 3d feature space.

   (d) What is the value of the margin that is achieved by this $\mathbf{w}$? Hint: recall that the margin is the distance from each support vecctor to the decision boundary. Hint 2: think about the geometry of 2 points in space, with a line separating one from the other.

(e) Solve for $\mathbf{w}$, using the fact that the margin is equal to $1/\|\mathbf{w}\|$.

(f) Solve for w_0 using your value for $\mathbf{w}$ and the two constraints listed above. Hint: the points will be on the decision boundary, so the inequalities will be tight.

(g) Write down the form of the discriminant function $f(x) = w_0 + \mathbf{w}^\top \phi(x)$ as an explicit function of x.