Optimization for Learning - FRTN50
Assignment 2

Written by: Martin Morin

Latest update: November 4, 2019

Introduction

This hand-in will cover the topic of overfitting and how it relates to regularization, model selection, and the problem data itself. For simplicity we restrict ourselves to models that results in convex problems but for non-convex models the principles remain the same.

ProximalOperators.jl In Hand-In 1 you implemented functions that calculate gradients and proximal operators of some objective functions. However, this be avoided whenever possible due to the ease of making mistakes. In this hand-in we will therefore use a software package that contain implementations of gradients and proximal operators for a wide range of functions.[1] The package is called `ProximalOperators.jl` and includes also basic operations such as scaling and conjugation. Following is a short example on how to use `ProximalOperators.jl` to compute the gradient and prox for the squared Euclidean norm. For more information we refer to the package documentation `http://kul-forbes.github.io/ProximalOperators.jl/stable/`. It contains simple examples and a list of supported functions and operations that can be performed on them.

```
f = SqrNormL2() # Create the function ½‖ · ‖²₂
val = f([1.0, 1.0]) # val = 1.0
df, _ = gradient(f, [1.0, 1.0]) # df = [1.0, 1.0]
pf, _ = prox(f, [1.0, 1.0], 0.5) # pf = [0.6666..., 0.6666...]
```

Assignment

The assignment consists of two parts: the first covers the effects of regularization on a simple least squares problem, and the second covers hyper-parameter tuning and validation for SVM.

[1]Another way to compute gradients is to automate it using *backpropagation/automatic differentiation*. We will cover the basics of this later in the course.

Regularized Least Squares Regression

A regression problem consists of some data, y_i, x_i, and a model, m_ω, parameterized by w. The goal is to find w such that $y_i \approx m_w(x_i), \forall i$. For simplicity of plotting, the data in this part will be scalar, $x_i, y_i \in \mathbf{R}$.

This problem depends on both what we mean by $y_i \approx m_w(x_i)$ and what model, m_w, we choose. Here we take $y_i \approx m_w(x)$ to mean they are close in the square L2-norm sense, $\|m_w(x_i) - y_i\|_2^2$, and restrict ourselves to models linear in w. This means models on the form $m_w(x) = w^T \phi(x)$ for some, possibly non-linear, *feature map* ϕ. This results in the classic least squares problem

$$\min_\omega \tfrac{1}{2} \sum_{i=1}^N \|m_w(x_i) - y_i\|_2^2 = \min_\omega \tfrac{1}{2} \|X^T w - y\|_2^2$$

where $X = [\phi(x_1), \phi(x_2), ..., \phi(x_N)]$, $y = (y_1, y_2, ..., y_n)$. We will further restrict ourselves to polynomial models, i.e. $y_i \approx m_w(x_i) = \sum_{k=0}^p w_k x_i^k$. Formulated in terms of a feature map this can be written as, $m_w(x) = w^T \phi_p(x)$ where $\phi_p(x) = (x^0, x^1, ..., x^p)$.[2]

It is good practice to include some pre-scaling, r, of the data x in the feature map, $\phi(r(x))$. Having elements of X with wildly varying size can result in slow convergence or other numerical problems. A simple translation and scaling, $r(x) = (x - \beta) \cdot \sigma$, that makes sure all the transformed data lie within $(-1, 1)$ can help with this.

It can often be beneficial to add a regularizing term that only act on the model parameters.

$$\min_\omega \tfrac{1}{2} \|X^T w - y\|_2^2 + \lambda \|w\|_q^q$$

where q will either be $q = 1$ or $q = 2$ in this task. Other regularization choices can of course be made, it also does not need to be a norm, but common to all choices is that they promote some property of the model which is deemed desirable.

The data you will use can be found in the file `problem.jl`. Use the proximal gradient method and `ProximalOperators.jl` to implement a solver, a suitable step-size is $\|XX^T\|^{-1}$. Make sure to use a pre-scaling so $r(x_i) \in (-1, 1)$ for all i.

Task 1 For $p = 1$ and $\lambda = 0$, plot and compare the resulting model $m_w(x)$ and the data (x_i, y_i). Increase p in steps up to $p = 10$ and study what happens. What happens with the model? In which ways is it better and/or worse? How many iterations does it take to find a solution?

Task 2 Set $p = 10$ and introduce regularization with $q = 2$. Try different λ in the range $[0.001, 10]$ and study the resulting model $m_w(x)$, the solution $w^\star$ and convergence rate. Repeat for $q = 1$. The regularization promotes $w^\star$ to be small in $\|\cdot\|_q^q$ sense, how does that manifest in $w^\star$ and $m_{w^\star}(x)$? Do you find structure in any of the solutions?

[2] Instead of the simple monomial basis, $x^0, x^1, ..., x^p$, some other polynomial basis could of course be used, for instance Legendre polynomials. This can be beneficial for numerical reasons but we choose the monomials here for simplicity.

Task 3 Chose a configuration, q and λ, from Task 2 and remove the pre-scaling. What affect did it have? Was it necessary?

Comments The effect of the regularization will of course depend on the model and/or how it is parameterized. For instance, given the linear model $m_w(x) = \phi(x)^T w$ an equivalent model can be formed simply by adding a shift of the parameters $m_{\hat{w}}(x) = \phi(x)^T(\hat{w} - a)$. A $\|w\|_q^q$-regularizer would then promote $m_{\hat{w}}(x) = -\phi(x)^T a$ instead of $m_{\hat{w}}(x) = 0$.

In deep learning is this type of explicit regularization not as common. Instead concepts like early termination, dropout or stochastic gradient methods are used to achieve regularization. The common property is that they prevent the algorithm from finding the true minimum and thereby avoid overfitting to the fine structure. An added benefit is that they also reduce the computational cost compared to standard gradient descent. Their main drawbacks are weaker theory and arguably less control.

Support Vector Machines

Given training data $x_i \in \mathbf{R}^n$ with corresponding class labels $y_i \in \{-1, 1\}$ we want to find a classifier such that $y_i \approx m_w(x_i) = \text{sign}(w^T \phi(x_i))$. Support vector machines do this by solving the following problem

$$\min_w h(YX^T w) + \tfrac{\lambda}{2}\|w\|_2^2$$

where $X = [\phi(x_1), \phi(x_2), ..., \phi(x_N)]$, $Y = \text{diag}(y_1, y_2, ..., y_N)$, and h is the hinge loss,

$$h(z) = \tfrac{1}{N}\sum_{i=1}^{N} \max(0, 1 - z_i).$$

This problem is easiest to solve via the dual

$$\min_\nu h^*(\nu) + \tfrac{1}{2\lambda}\|-XY\nu\|_2^2.$$

The last term is a quadratic, $\frac{1}{2\lambda}\|-XY\nu\|_2^2 = \frac{1}{2\lambda}\nu^T YX^T XY\nu = \frac{1}{2}\nu^T Q\nu$ where the matrix Q is given by $Q_{ij} = \lambda^{-1} y_i \phi(x_i)^T \phi(x_j) y_j$, i.e. Q can be evaluated by only evaluating the *kernel*, $K(x, y) = \phi(x)^T \phi(y)$, associated with the feature map ϕ. This allows for simpler/cheaper computation since there is no need to explicitly evaluate the (potentially) very high dimensional feature map. For example, a polynomial model of order p is simply given by $K(x, y) = (x^T y + a)^p$.

Using only kernel evaluations also removes the need to find the feature map explicitly. It is possible to define a kernel and as long as it is proper, i.e. there exists some feature map such that $K(x, y) = \phi(x)^T \phi(y)$, it will result in a well-posed convex problem. Existence of a feature map is typically be established via *Mercer's condition.* When choosing/designing a kernel it can be viewed as a correlation measure between data points. Choosing the right model/kernel/feature map is then choosing a measure such that data points of different classes are far away form each other.

For the rest of this assignment the Gaussian kernel will be used $K(x_i, x_j) = e^{-\frac{1}{2\sigma^2}\|x_i - x_j\|_2^2}$. This kernel is interesting since the corresponding feature map is infinite dimensional and can therefore not be used directly. The hyper-parameter σ in the kernel determines the length-scale of which points are considered near each other. In practice this controls the smoothness of the resulting model.

If a small length-scale is used, even points close to each other will be considered far apart in the kernels point of view. The SVM can then relatively easy find a separating 'plane' between the points and the resulting decision boundary will be very curvy as it snakes its way between points. For longer length-scales will the kernel not be able to distinguish points close to each other. The SVM will only be able to separate point further apart, resulting in a smoother model with less overfitting. However, too large σ will result in the SVM not being able to separate any points at all.

The data you will use can be found in the file `problem.jl`. Use the proximal gradient method and `ProximalOperators.jl` to implement a solver for the SVM dual problem. A suitable step-size is $1/\|Q\|$. Note that `ProximalOperators.jl` can perform conjugation for you and that the data is already appropriately scaled. Implement the predictive model, $m_w(x)$, using the results from Task 4.

Task 4 Derive an expression for recovering the primal solution from the dual. Using this expression, show that it is possible to evaluate the model $m_w(\hat{x}) = \text{sign}(w^T\phi(\hat{x}))$ with only the kernel, data points (x_i, y_i), and dual solution.

Testing

In the least squares problem, the model and data could easily be inspected visually but it is hard to get any quantitative performance from it. When the data lies in higher dimensional spaces it can also be hard to visualize the results. In these cases must different models and/or regularization selections be evaluated in some other way.

Arguably the simplest form of validation is *holdout cross-validation.* It splits the data set into two disjoint sets, one training data set and one validation data set. You train the model using the training data and perform some form of performance test on the validation data. For a classifier, a natural performance measure is the error rate on the validation set, i.e. ratio between the number of miss-classified validation data points and the total number of validation data points.

Task 5 Use the data from `svm_train()` for training. Try different hyper-parameters λ and σ over coarse grids, for instance $\lambda \in \{0.1, 0.01, 0.001, 0.0001\}$ and $\sigma \in \{1, 0.5, 0.25\}$. For each configuration, calculate the error rate on the *validation data* from `svm_test_1()` but also calculate the error rate on the *training data.* Are the error rates on the two sets the same? What is the lowest error rate you can achieve on each set? Can you identify overfitting? Based on your findings, select the λ and σ you feel is best.

Splitting data into a training and validation data is not without drawbacks. The most obvious drawback is that you do not use all available data for training. For this reason, once the hyper-parameters have been selected, it is a good idea to retrain the model using both the training and validation data before the model is used.

Another potential issue is that any time you do some sort of selection of data, there is a risk of inadvertently introducing bias. The goal is to train a model that fits well to all data, even future data that are yet to be sampled. By tuning the hyper-parameters for specific validation data, there is actually a risk of overfitting to the bias of the validation data. For this reason, in for instance classification competitions, the final performance benchmark is performed on a *test data* set that is withheld completely during all hyper-parameter tuning to avoid overfitting to the benchmark data.

Task 6 Four different validation sets are contained in `problem.jl`: `svm_test_1()`, `svm_test_2()`, `svm_test_3()`, and `svm_test_4()`. Does your selected model from Task 5 have the same error rate on all four validation sets? Try different hyper-parameters, would you have made different hyper-parameter choices if you had been given a different validation set?

Bias have intentionally been introduced to two of the validation sets, making them unsuitable to validate the model on. Can you identify the biased sets? (Without looking at the code.) Can you explain what has been done to them and explain why it is bad? Hint: Look at the error rates on both the training and validation sets for different hyper-parameter choices. For instance, $(\lambda, \sigma) = (0.1, 2)$, $(\lambda, \sigma) = (0.001, 0.5)$, and $(\lambda, \sigma) = (0.00001, 0.25)$. It can also be a good idea to compare with a constant classifier ($m(x) = 1$ or $m(x) = -1$).

If a relatively small amount of data is available it can be hard to make a training/validation split that keeps enough diversity in both data sets for holdout cross-validation to work well. The resulting performance estimate would then have a high dependence on the particular split, resulting in a large variance. An alternative to holdout cross-validation that tries avoid this problem by using all the data both for training and validation is *k-fold cross-validation*.

k-fold cross-validation works by randomly splitting the data into k equal size data sets and then performing k rounds of holdout cross-validation. For each round a different sub-set is used for validation while the remaining sub-sets are used for training. After all rounds of training/validation, the k performance scores are averaged to get a final score.

The drawback of this approach is that for each hyper-parameter configuration we wish to test, k models needs to be trained. For this reason, k is usually quite small, $k = 10$ is a common choice.

Task 7 Using the λ and σ you chose from Task 5, compare the error rate estimates given by 10-fold cross-validation and holdout cross-validation. Use only the data from `svm_training()` and randomly set aside 100 data points for the holdout cross-validation.

Do the two methods give similar error rate estimates? If you re-run the cross-validations with different random permutations of the training/validation splits, are the error rates the same? How much do they vary? Examine the variance and/or create a histogram of the estimated error rates. Which cross-validation method seem more reliable? Hint: To randomly permute the data, look at the function `randperm` in the standard library `Random`.

Submission

See the latest version of the course program for instruction on how to submit the assignment. Your submission should contain the following.

- Your code that solves the regularized least squares and SVM problems.
- Your code for k-fold and holdout cross-validation from Task 7.
- A single pdf containing the following:
  - A couple of paragraphs describing your findings form Task 1-3.
  - Your derivation from Task 4.
  - A couple of paragraphs describing your findings form Task 5-7.

Use plots, figures and tables to motivate your answers when possible.