TME 5 - Perceptron et SVMs

You must have completed TME 2 before starting this one. You will need the cost functions and their gradients that you implemented in the previous session.

Perceptron and `Linear` Class

Implement the function `perceptron_loss(w,x,y)` that returns the perceptron cost ($max(0, -y < \mathbf{x}.\mathbf{w} >)$) and its gradient `perceptron_grad(w,x,y)`.
To be more generic and flexible, instead of implementing a script version of gradient descent, we will code a `Linear` class that will group different possible variants. A class skeleton is provided in the source file : it mainly contains the constructor that initializes the model parameters (the cost used `loss`, its gradient `loss_g`, the number of gradient descent iterations `max_iter`, the step size `eps`, the weight vector `w`). Complete the skeleton by implementing :

- The method `predict(self,datax)` which infers the label of the data `datax` (i.e., calculates $sign(< \mathbf{w}.\mathbf{x} >)$).
- The method `score(self,datax,datay)` which calculates the percentage of correct classifications on the given dataset.
- The method `fit(self,datax,datay)` which performs gradient descent for `max_iter` iterations with a step size `eps`, using the cost `loss` and its gradient `loss_g`. To implement a perceptron, you just need to pass the perceptron cost and gradient when constructing the object. Your method should also store the history of costs over iterations.

As in the previous TME, your methods should take matrices of data as input (not single data points). Test your class on last week's example.

USPS Data

The USPS dataset consists of handwritten digit images represented by grayscale pixel matrices of size 16×16. Code is provided to load and visualize these data (`load_usps` and `show_usps`).
Load the data and visualize some examples. Select two classes (e.g., 6 vs. 9) and train a perceptron on them. Visualize the obtained weight matrix using `show_usps`. What do you notice ? How do you interpret the result ?
Observe the weight matrix when training the perceptron with a single class (e.g., 6) against all other classes.
Using test data, plot learning and test error curves as a function of the number of iterations (you can modify the `fit` function of the `Linear` class to take test data as input). Do you observe overfitting ?

Mini-batch and Stochastic Descent

Modify your `fit` method to support stochastic descent (examples are randomly ordered and only one example is used to compute the gradient and update weights) and mini-batch descent (the dataset is divided into small random batches of m examples, the gradient is averaged over each batch before updating).
To fairly compare the different variants, `max_iter` will denote the number of epochs rather than the number of gradient updates (an epoch means all examples have been seen once). Note that stochastic descent is a mini-batch descent of size 1, while batch descent is a mini-batch descent of size equal to the number of examples.

Compare convergence speed, particularly in relation to noise in the dataset.

Projections and Regularization

To increase the expressivity of the linear model, we will use projections.
Implement the function `proj_poly(datax)` which returns the degree-2 polynomial projection of the data : $(1, x_1, x_2, \ldots, x_d, x_1^2, x_1x_2, \ldots x_d^2)$.
Also implement `proj_biais(datax)` which adds a column of 1s in the first column of the data to introduce a bias.
Modify your constructor to potentially take a projection as a parameter. Modify `fit` to project data before training and `predict` accordingly.
Test your polynomial projection on artificial data from `gen_arti` of type 1 and 2. Plot the decision boundaries.
Code the function `proj_gauss(datax, base, sigma)` that performs a Gaussian projection of `datax` onto the points $(\mathbf{b}_1, \ldots, \mathbf{b}_b)$ of `base` with a parameter `sigma` : for a given data point $\mathbf{x}$ in `datax`, its representation is given by $(e^{-\|\mathbf{x}-\mathbf{b}_1\|^2/2\sigma}, e^{-\|\mathbf{x}-\mathbf{b}_2\|^2/2\sigma}, \ldots, e^{-\|\mathbf{x}-\mathbf{b}_b\|^2/2\sigma})$. Experiment with the three artificial datasets.
Is it better to have many or few points in the projection base ? Should σ be large or small ? Which points have the most weight ? Represent them in the figures and plot the decision boundaries.
You can also study the effect of a margin and penalization on the weights of the Gaussian projection : introduce a new cost function `hinge_loss(w, x, y, alpha, lambda)` that returns $\max(0, \alpha - y < \mathbf{w}, \mathbf{x} >) + \lambda\|w\|^2$ and its gradient `hinge_loss_grad`. Observe the effect of `alpha` and `lambda`, particularly on the checkerboard problem.
What are the connections with SVM ?

SVM and *Grid Search*

The scikit-learn module (`sklearn` : `http://scikit-learn.org/stable/documentation.html`) is the main statistical learning module in Python. The submodule `sklearn.linear_model` provides an implementation of the perceptron, `sklearn.neighbors` an implementation of k-NN, and `sklearn.tree` decision trees. Quickly explore these implementations and compare, for example, the results of your previous practicals with those from `sklearn`.
Ensure you have the latest version installed ! Otherwise, run `pip install -U sklearn --user`
The `sklearn.svm` module provides an implementation of SVMs. Using datasets from previous practicals (2D artificial datasets and digit recognition), explore different kernels (linear, Gaussian, polynomial) and various kernel parameterizations.
In particular, study the decision boundaries and the support vectors—the points whose coefficients are non-zero. How does their number vary depending on the kernel and its parameters ? Is this behavior expected ? What do you observe in the linear case ?
To find the best parameters, we perform cross-validation over a grid of parameters (grid search). For different kernels and varying numbers of training examples, perform a grid search to find the optimal parameters. Plot the error curves for training and testing. Are the results consistent ?
To visualize decision boundaries in 2D, you can use the following code snippet :

```
svm = sklearn.svm.SVC(probability = True, ...)
...
def plot_frontiere_proba(data, f, step=20):
    grid, x, y = make_grid(data=data, step=step)
    plt.contourf(x, y, f(grid).reshape(x.shape), 255)

plot_frontiere_proba(data, lambda x: svm.predict_proba(x)[:,0], step=50)
```

String Kernel

The string kernel is a kernel defined on words Σ^* from an alphabet Σ. For a word s, we denote $|s|$ as its length : $s = s_1, s_2, \ldots, s_{|s|}$, and $s[i:j]$ as the substring $s_i, \ldots, s_j$ of s. We say that u is a subsequence of s if there exists a sequence of indices $\mathbf{i} = (i_1, \ldots, i_{|u|})$ with $1 \leq i_1 < i_2 < \ldots < i_{|u|} \leq |s|$ such that $u_j = s_{i_j}$ for $j = 1, \ldots, |u|$, denoted as $u = s[\mathbf{i}]$. The length $l(\mathbf{i})$ of the subsequence in s is $i_{|u|} - i_1 + 1$.
The projection used is the set of coordinates $\{\phi_u(s) = \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{l(i)}\}\ u \in \Sigma^*$ with $\lambda \leq 1$.
Thus, we can define the kernel $K_n(s,t) = \sum_{u\in\Sigma^n} < \phi_u(s), \phi_u(t) >= \sum_{u\in\Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \sum_{\mathbf{j}:u=t[\mathbf{j}]} \lambda^{l(\mathbf{i})+l(\mathbf{j})}$.
Implement a string kernel, visualize the similarity matrix on text examples from different authors, and test learning performance.