

COMP90051 Statistical Machine Learning

Project 1 Description

Updated: 5nd Sept 2022; 12 Sept 2022, updates shown in red.

Due date: 5:00pm Friday, 16 Sept 2022

Weight: 25%; forming combined hurdle with Proj2

Copyright statement: All the materials of this project—including this specification and code skeleton—are copyright of the University of Melbourne. These documents are licensed for the sole purpose of your assessment in COMP90051. You are not permitted to share or post these documents.

Academic misconduct: You are reminded that all submitted work is to be your own individual work. Automated similarity checking software will be used to compare all submissions. It is University policy that academic integrity be enforced. For more details, please see the policy at <http://academichonesty.unimelb.edu.au/policy.html>. Academic misconduct hearings can determine that students receive zero for an assessment, a whole subject, or are terminated from their studies. You **may not** use software libraries or code snippets found online, from friends/private tutors, or anywhere else. You can only submit your own work.

The Support Vector Machine (SVM) are a powerful framework for classification, formulated around the idea of maximum margin of separation which ensures strong generalisation performance. We have seen the equivalent primal and dual formulations of the SVM in lectures, and how these lead to different time complexities of inference, as well as the ability to incorporate kernels when using the dual. In this project, you will work *individually* to implement several SVM algorithms. These go beyond what was covered in class, and includes real research papers that you will have to read and understand yourself.

By the end of the project you should have developed:

ILO1. A deeper understanding of the SVM and its primal and dual formulation;

ILO2. An appreciation of how SVMs are applied;

ILO3. Demonstrable ability to implement ML approaches in code; and

ILO4. Ability to understanding research papers, understand their focus, contributions, and algorithms enough to be able to implement and apply them. (While ignoring details not needed for your task.)

Overview

The project consists of four parts.


- | | |
|---------------------------------------------------------------------|-----------|
| 1. Solving the primal problem with stochastic online updates | [5 marks] |
| 2. Solving the dual problem with Kernel-Adtron | [7 marks] |
| 3. Incorporating kernels | [5 marks] |
| 4. Implementing the sequential minimal optimization (SMO) algorithm | [8 marks] |

All parts are to be completed in the provided Python Jupyter notebook `proj1.ipynb`.¹ Detailed instructions for each task are included in this document.


¹We appreciate that while some have entered COMP90051 feeling less confident with Python, many workshops so far have exercised and built up basic Python and Jupyter knowledge. Both are industry standard tools for the data sciences.

SVM algorithms. The project's tasks require you to implement SVM training algorithms by completing provided skeleton code in `proj1.ipynb`. All of the SVM training algorithms must be implemented as sub-classes of the provided base SVM class. This ensures all SVM training algorithms inherit the same interface, and your implementations *must* conform to this interface. You may implement functionality in the base SVM class if you desire—*e.g.*, to avoid duplicating common functionality in each sub-class. Your classes may also use additional private methods to better structure your code. And you may decide how to use class inheritance.

Python environment. You must use the Python environment used in workshops to ensure markers can reproduce your results if required. We assume you are using Python ≥ 3.8 , numpy $\geq 1.19.0$, scikit-learn $\geq 0.23.0$ and matplotlib $\geq 3.2.0$.

Other constraints. You may not use functionality from external libraries/packages, beyond what is imported in the provided Jupyter notebook highlighted here with margin marking . You must preserve the structure of the skeleton code—please only insert your own code where specified. You should not add new cells to the notebook. You may discuss the SVM learning slide deck or Python at a high-level with others, including via Piazza, but do not collaborate with anyone on direct solutions. You may consult resources to understand SVM conceptually, but do not make any use of online code *whatsoever*. (We will run code comparisons against online partial implementations to enforce these rules. See ‘academic misconduct’ statement above.)

Submission Checklist

 **You must complete all your work in the provided `proj1.ipynb` Jupyter notebook.** When you are ready to submit, follow these steps. You may submit multiple times. We will mark your last attempt. *Hint: it is a good idea to submit early as a backup. Try to complete Part 1 in the first week and submit it; it will help you understand other tasks and be a fantastic start!*

1. Restart your Jupyter kernel and run all cells consecutively.
2. Ensure outputs are saved in the `ipynb` file, as we may choose not to run your notebook when grading.
3. Rename your completed notebook from `proj1.ipynb` to `username.ipynb` where `username` is your university central username.²
4. Upload your submission to the Project 1 Canvas page.

Marking

Projects will be marked out of 25. Overall approximately 60%, 20%, 20% of available marks will come from correctness, code structure & style, and experimentation. Markers will perform code reviews of your submissions with **indicative** focus on the following. We will endeavour to provide (indicative not exhaustive) feedback.

1. *Correctness*: Faithful implementation of the algorithm as specified in the reference or clarified in the specification with possible updates in the Canvas changelog. It is important that your code performs other basic functions such as: raising errors if the input is incorrect, working for any dataset that meets the requirements (*i.e.*, not hard-coded).
2. *Code structure and style*: Efficient code (*e.g.*, making use of vectorised functions, avoiding recalculation of expensive results); self-documenting variable names and/or comments; avoiding inappropriate data structures, duplicated code and illegal package imports.

²LMS/UniMelb usernames look like `tcohn`, not to be confused with email such as `trevor.cohn`.

3. *Experimentation*: Each task you choose to complete directs you to perform some experimentation with your implementation, such as evaluation, tuning, or comparison. You will need to choose a reasonable approach to your experimentation, based on your acquired understanding of the SVM learners.

Late submission policy. Late submissions will be accepted to 4 days at -2.5 penalty per day or part day. Weekends and holidays will also be counted towards the late penalty.

Part Descriptions

Part 1: Primal problem with stochastic gradient update [5 marks total]

Your first task is to implement a soft-margin SVM in its primal formulation, using stochastic gradient descent (SGD) for training. This is an online algorithm which is similar to the perceptron training algorithm (see week 6 tutorial), where training involves an outer loop run several times, and an inner loop over each instance in the training set, where the parameters are updated using the gradient of the loss for a single example. The stopping criterion here for the outer is to finish `iterations` iterations.

At each step, the algorithm will update the weights \mathbf{w} and bias b , with update rules:

$$\begin{aligned}\mathbf{w}_t &= \mathbf{w}_{t-1} - \eta \nabla_{\mathbf{w}} L_i(\mathbf{w}, b) \\ b_t &= b_{t-1} - \eta \nabla_b L_i(\mathbf{w}, b)\end{aligned}$$

where η is the learning rate, and $L_i(\mathbf{w}, b)$ is the loss function for the i^{th} example. You will first have to figure out the per-instance loss, $L_i(\mathbf{w}, b)$, based on the total loss of the training set,

$$L(\mathbf{w}, b) = \sum_{i=1}^n \max(0, 1 - y_i(\mathbf{w}'\mathbf{x}_i + b)) + \frac{1}{2} \lambda \|\mathbf{w}\|^2$$

and then derive the relevant gradients needed for the SGD updates. You will then need to implement a `PrimalSVM` class with the following functions:

1. `fit`, which trains using SGD as described above
2. `predict`, which classifies a new instance \mathbf{x} based on sign of $\mathbf{w}'\mathbf{x} + b$
3. `__init__`, to retain state or precompute values to support the above methods

Experiments. Once your `PrimalSVM` class is implemented, it is time to perform some basic experimentation.

(a) Include and run an evaluation on the given dataset:

```
psvm = PrimalSVM(eta = 0.1, lambda0 = 0.1)
psvm.fit(X,y,iterations = 100)
print(f"Accuracy is {round(psvm.evaluate(X,y),4)}")
psvm.visualize(X,y)
```

Print the training accuracy and plot the dataset and decision surface.

- (b) Now you will need to tune your `PrimalSVM`'s hyperparameters. Based on your understanding of the λ value, test a range of values (consider equally spaced points in logarithmic space) and run experiments to find the best value.³ You must set `eta` to 0.1 and `iterations` to 100. Output the result of this strategy—which could be a graph, number, etc. of your choice.

³Best here means the setting with the best training accuracy. This is used here as a sanity check to test whether the model can fit the data, but of course it is not a good measure of the model's generalisation accuracy.

Part 2: Dual SVM formulation with stochastic online update [7 marks total]

In this part, you are to implement the dual formulation of the soft-margin SVM. Your implementation should be done in the provided DualSVM skeleton code, and be based on SGD training algorithm in Table 7.1 of Chapter 7 of the following book:

Cristianini, N., & Shawe-Taylor, J. (2000). An Introduction to Support Vector Machines and Other Kernel-based Learning Methods (pp. 125-148). Cambridge: Cambridge University Press.⁴

You will need to implement the following functions:

1. `__init__`
2. `fit` with the training algorithm described above. For the training loop the same stopping criterion as Part 1, that is stopping after a given number of iterations. The algorithm listed in the paper assumes a fixed bias, however for in this project, you should implement `get_bias` function to compute b . The standard method for doing so uses the equation:

$$b = y_i - \sum_{j=1}^n \alpha_j y_j k(x_i, x_j)$$

for an arbitrary instances i with $0 < \alpha_i < C$. However, this assumes training has converged and the KKT conditions have been satisfied, which may not be true with SGD training. For this reason, you should compute the bias estimate for all candidate instances i , and use the average estimate as the bias.⁵

3. `predict` which should classify a new instance \mathbf{x} based on sign of $b + \sum_{i=1}^n \alpha_i y_i k(\mathbf{x}_i, \mathbf{x})$, where k is the kernel function (linear for now, but your implementation should be general to support part 3, below).
4. `primal_weights` which computes the equivalent primal weights using a linear kernel, i.e., $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$.

Experiments. Evaluate your model with given dataset, use $\eta = 0.1$ $C = 100$, $iterations = 100$ and a linear model. Perform the following experiments, with `dsvm = DualSVM(eta = 0.1)` and keep the other hyperparameter values unchanged except C value:

- (a) Tune the value of C , to optimise for training accuracy.
- (b) Report the equivalent weights for the dual solution, and compare to the primal trained with equivalent loss. *Hint:* consider how C and λ are related, such that you compare an equivalent model. Test to check the weights are similar, using the norm of the difference.
- (c) Identify the support vectors, and points where $\alpha = C$. Display your results either on a plot, or by printing the index of the relevant instances.

Part 3: Kernel [5 marks total]

In this part, you are going to use the kernel trick. Specifically, you need to use the kernelised version of SVM in combination with a few different kernels. *In this section, you cannot use any libraries other than 'numpy'.* First, implement polynomial and RBF kernels. The linear kernel is simply a dot product of its inputs and has been provided. Polynomial and RBF kernels should be implemented as defined in the lecture slides. For the polynomial kernel, you should include a scalar offset and degree, i.e., $(\mathbf{u}^T \mathbf{v} + c)^d$, and for the RBF include a width parameter, σ . Your kernels should support evaluation with vector (instance) or matrix (dataset) inputs in both the first and second position, as documented in the `__call__` function, and return a scalar, vector, or matrix as appropriate.

⁴Access via e-book in the library catalogue, e.g., [using this link and the 'Get access' option](#).

⁵This isn't always the most reliable method of setting the bias. It could be improved by optimising the training objective for the best bias scalar given α . But for the purpose of the project, we will keep with the simple method as described.

Experiments. Here run `dsvm` with the three different kernels, you should use $\eta = 0.1$, $C = 100$ and `iterations = 100`. You don't need to show how you tune the hyperparameters in this part, but you should run experiments with reasonable values. Use the `visualize` function to display the decision boundary for each of the three kernels. In a few sentences, explain which kernel you think is best suited to this problem, and why.

Part 4: Sequential minimal optimization [8 marks total]

In this part, you will implement a final training algorithm for the soft-margin SVM, sequential minimal optimization (SMO), based on the following paper:

Platt, J. (1998). Sequential minimal optimization: A fast algorithm for training support vector machines.

See in particular the pseudocode in §2.5 of. You can also find useful information from the book cited in Part 2. The algorithm is somewhat complex, so we adapted it lightly and provide our framework to help you get started with your implementation. Specifically, we have removed the `examineExample` function, and instead provide some simpler heuristics for selecting which pairs of instances will be processed in each iteration. The provided heuristics are based on random pairs or exhaustively considering all pairs, and are in general less efficient than the SMO algorithm. We did not add detailed comments to the code, you will need to read the literature to fully understand the code.

Your job is to implement the missing and incomplete functions: `lower` and `higher` function based on equations (13) and (14), and then complete the implementation of the `__take_step` function (please ignore the corner case where η is less or equal to 0, in which instance you should make no update), and implement `update_bias` function based on §2.3 of the above paper. The `predict` function should be same as for Part 2, but differs in the sign of bias term.

Once you've completed your version of SMO algorithm and performed preliminary experiments (part a, below), you should apply it to a real-world dataset, fashion MNIST.⁶ We extracted 1200 instances of shirt and coat classes to create a binary classification dataset, and then split dataset in half, for training and testing. As part of applying your SMO implementation to this dataset, you will also need to a better heuristic selection function `my_heuristic` to improve over the two default strategies. You should read §2.2 of the paper to get some insights about heuristics. Note that your heuristic function should not involve expensive calculations which would invalidate the point of using it. Overall it may take a few minutes to train models to convergence on this dataset.

Experiments. Do the following:

- Train `svm = SMO(C = 1, eps = 0.01, tol = 0.001)` with Linear, Polynomial and RBF kernels with the synthetic dataset from above. Report the training accuracy for each model.
- Train models with `PrimalSVM`, `DualSVM` and `SMO` on fashion dataset. You are welcome to tune the hyperparameters, but there's no need for extensive (expensive!) experimentation. Print the training and evaluation accuracy, but please do not attempt to plot the decision surface.
- Run `SMO` with the heuristic functions you implement (with the same hyperparameters as part 4.a, and the linear kernel). Compare the number of steps tested and taken for the `SMO` with the two default strategies (random pairs and all pairs) versus your own method, and report the results.

⁶<https://github.com/zalandoresearch/fashion-mnist>