Performance Evaluation of Linear SVM with Nyström Approximation

Andrew Ruder

December 9, 2024

Abstract

This project explores the implementation and evaluation of a custom One-vs-Rest (OvR) for a linear Support Vector Machine (SVM) classifier combined with the Nystroem method for kernel approximation. The goal is to compare the accuracy and runtime of the custom implementation against baseline models, including scikit-learn's SVMs and Nystroem transformations. The analysis focuses on scalability and performance improvements when applying kernel methods to large datasets. The results provide insights into the trade-offs between computational efficiency and model accuracy.

1 Introduction

1.1 Motivation

I was interested in kernel approximation methods as they, in theory, reduce the O(n) complexity of the model training. Kernel-based methods, while powerful, are computationally expensive for large datasets. The Nystroem approximation provides a strategy to reduce computational costs by approximating the kernel matrix.

1.2 Objectives

My objectives for this project started with creating a Linear SVM and then implementing the Nystroem method for kernel approximation. I would then compare my implementations against baseline models to evaluate trade-offs between runtime and predictive performance. Based on the dataset that I decided to use, outlined the next section, I would also have to implement a One-vs-Rest classifier as it would allow for mutliclass datasets instead of just binary classification.

2 Methods

2.1 Dataset

The MNIST dataset is a widely recognized benchmark for evaluating machine learning algorithms, particularly in image recognition and classification tasks. It contains grayscale images of handwritten digits (0-9) and is divided into a training set of 60,000 examples and a test set of 10,000 examples. To prepare the dataset for classification tasks, the pixel values were normalized to the range [0, 1] by dividing all values by 255. The MNIST dataset is inherently a multiclass problem, requiring classification across 10 unique labels (digits). To handle this, an One-vs-Rest (OvR) Support Vector Machine (SVM) approach was employed. Due to computational constraints, only a subset of the MNIST dataset was used for this project.
Specifically:
Training Set: The first 5,000 samples from the randomized training dataset were used for model training.
Test Set: The next 2,000 samples from the randomized dataset was used for evaluation.

2.2 Linear SVM

A standard linear SVM aims to find a hyperplane defined by weights w and bias b that maximizes the margin between two classes. The optimization problem for a linear SVM with soft margins is given by:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$
subject to:
$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

where ξ_i are slack variables to handle misclassified points, C is a regularization parameter, and $\mathbf{x}_i, y_i$ represent the data points and their labels. For this project, I use a hinge loss function that is represneted by:

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

The standard SVM maximizes the margin, where I am minimizing the squared hinge loss. This is because hinge loss works better with gradient descent methods. I am using a gradient descent method as it tends to be more efficient for larger datasets. The gradients are computed based on the squared hinge loss.

For a misclassified point ($1 - y_i(\mathbf{w}^T\mathbf{x}_i + b) > 0$), the gradients are:

$$\nabla_{\mathbf{w}} = \mathbf{w} - 2C \cdot \text{weight} \cdot (1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))y_i\mathbf{x}_i$$

$$\nabla_b = -2C \cdot \text{weight} \cdot (1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))y_i$$

I am also limiting the set range of the gradients to try to limit divergence in training. Additionally, I am using a dynamic learning rate to help with convergence. Lastly, I am including class weights to address imbalanced samples, which is not inherently handled by standard SVMs. This ensures that the minority classes are not underrepresented during optimization, which will happen a lot in the One-vs-Rest SVM.

2.3 One-vs-Rest SVM

The MNIST dataset involves classification across 10 distinct digits, making OvR a natural choice. OvR provides a straightforward approach to extending binary SVMs to multiclass problems. Put simply, you make a binary classifier for each class and test on one class versus the rest of the classes. For a multiclass dataset with K classes, the OvR SVM trains K binary classifiers, one for each class. For the kth class, the optimization problem is:

$$\min_{\mathbf{w}_k, b_k} \frac{1}{2}\|\mathbf{w}_k\|^2 + C\sum_{i=1}^{n}\xi_i$$

subject to:

$$y_i^{(k)}(\mathbf{w}_k^T\mathbf{x}_i + b_k) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

where $y_i^{(k)} = 1$ if the sample belongs to class k and -1 otherwise.

For prediction, the class is assigned based on the classifier with the highest decision score:

$$\hat{y} = \arg\max_k(\mathbf{w}_k^T\mathbf{x} + b_k).$$

I made a few changes to this regarding normalization. The decision scores from each binary classifier are normalized using L2 regularization for the weight.

$$\text{normalized score}_k = \frac{\mathbf{w}_k^T\mathbf{x} + b_k}{\|\mathbf{w}_k\| + 1e^{-6}},$$

The denominator has an extremely small constant being added to it to avoid division by 0. Another addition that I made to align with my Linear SVM

changes was class weighting. This is done during training when weights are adjusted for the positive class in each binary classifier. The magnitude of the adjustment is dependent on the frequency of the positive class. This modification mitigates the risk of underfitting for less frequent classes and is necessary to align with the Linear SVM changes.

2.4 Nystroem Transformation

The Nystroem method is a powerful technique for approximating kernel matrices in machine learning. The method makes it feasible to apply kernel-based methods to large datasets. Kernel methods map input data into a higher-dimensional feature space where linear algorithms can perform non-linear separations. This mapping involves the computation of a kernel matrix, which has a time complexity of $\mathcal{O}(n^2)$ for n samples. This makes it computationally expensive for large datasets. The Nystroem method alleviates this computational bottleneck by approximating the full kernel matrix using a subset of representative data points called landmarks. This reduces the computational requirements to $\mathcal{O}(nm)$, where m is the number of selected landmarks. For a given dataset $n \times d$ and a kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$, the kernel matrix is defined as:

$$\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j).$$

The Nystroem method approximates the feature mapping by selecting m landmark points $\mathbf{Z} \in \mathbf{R}_{m \times d}$. How you go about selecting the landmarks is your choice. You can do it randomly or have knowledge of your dataset and define them. How you select them does affect if you model underfits or overfits. The method that I used was by using K-Means. Once your landmarks are selected, we can compute the two matrices from the kernel. The landmark matrix and cross kernel matrix. The landmark kernel matrix, $\mathbf{K}_{mm} = k(\mathbf{Z}, \mathbf{Z})$, computes kernel values between the landmarks. The cross kernel matrix, $\mathbf{K}_{nm} = k(\mathbf{X}, \mathbf{Z})$, computes kernel values between all data points and the landmarks. The feature mapping $\Phi(\mathbf{X})$ is approximated as

$$\Phi(\mathbf{X}) = \mathbf{K}_{nm}\mathbf{W},$$

where $\mathbf{W} = \mathbf{U}\mathbf{\Lambda}^{-1/2}$. $\mathbf{U}$ and $\mathbf{\Lambda}$ are the eigenvectors and eigenvalues of $\mathbf{K}_{mm}$, respectively. $\mathbf{\Lambda}^{-1/2}$ is the diagonal matrix of inverse square roots of eigenvalues. This transforms the input data into feature space:

$$\mathbf{X}' = \Phi(\mathbf{X}) = \mathbf{K}_{nm}\mathbf{U}\mathbf{\Lambda}^{-1/2}.$$

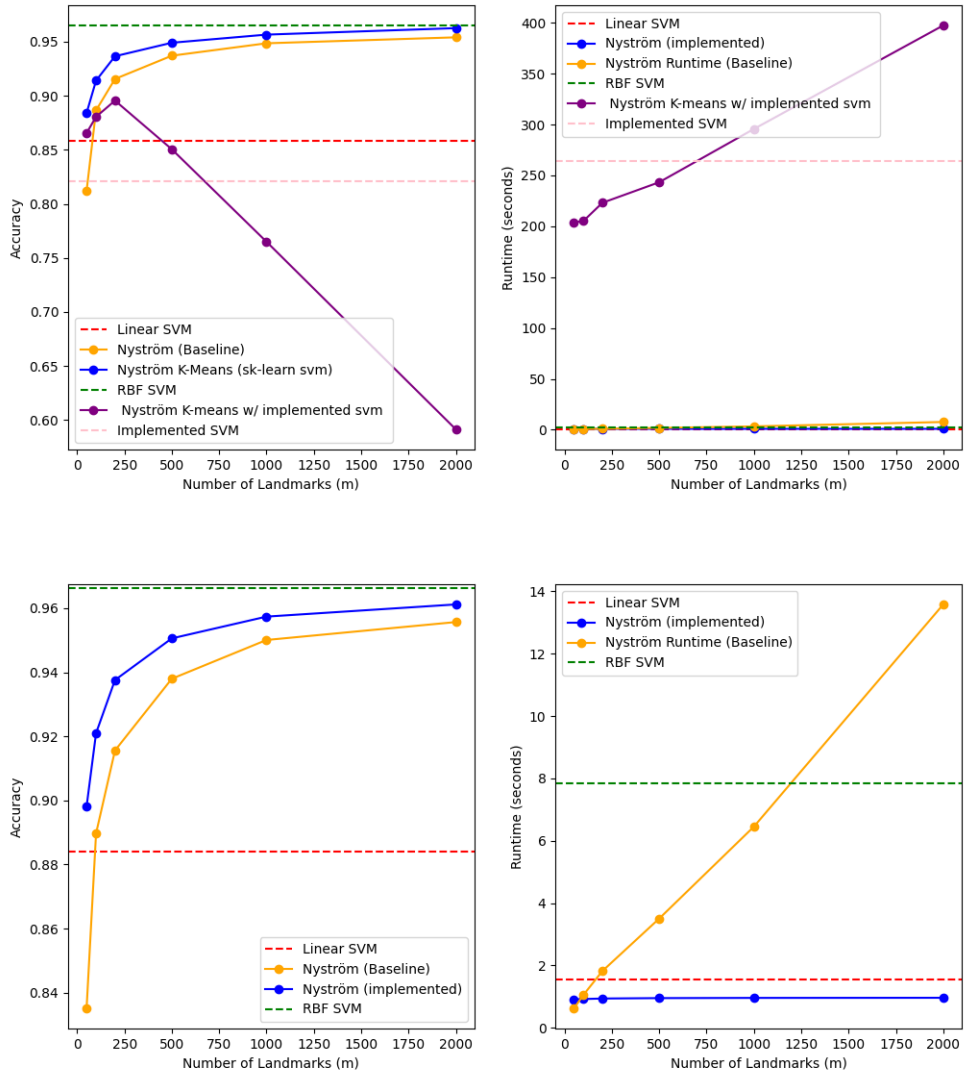
The values for $\mathbf{U}$ and $\mathbf{\Lambda}$ can be calculated from eigendecomposition or singular value decomposition. I implemented the singular value decomposition as it allows for use with non-square matrices. Additionally, I implemented ridge regularization prior to the singular value decomposition step to make sure the approximation does not diverge after the inversion step.

3 Model Evaluation

3.1 Model Evaluation Setup

I wanted to test my model/s against a baseline for accuracy and runtime. I choose to compare my models against sk-learn's implementations for a radial basis function SVM, a linear SVM, a linear SVM using the sk-learn Nystroem transformation and a linear SVM using my Nystroem transformation. I originally did not intent to compare the models against their linear SVM using my Nystroem transformation but during testing I wanted to validate that my method worked and I decided to keep it. To evaluate performance, I would be comparing my SVM implementation, using and not using my Nystroem implementation, to the the listed models above. I varied the number of landmarks for the Nystroem approximations and plotted them against accuracy and runtime. The different points for the approximations were 50, 100, 200, 500, 1000 and 2000. Lastly, I parallelized the evaluation of my models as the process was time consuming across the variations of landmark selections. The results of this model evaluation can be shown on the next page.

3.2 Results



3.3 Interpretation of Results

I want to start with the negatives that immediately pop out when I see these graphs. The first thing that was painfully obvious from these visualizations is that my SVM implementation is incredibly slow. I had to re-graph the

models I was benchmarking against because they were that much faster than my SVM implementation. The second major takeaway from these graphs is that something goes wrong with either my Nystroem transformation or how my SVM interprets the approximations as the number of landmarks increase. We can see this as the accuracy drops steadily in the purple line. I'm not sure where this issue stems from but I imagine it has to do with a bias towards classes that is derived from the Nystroem approximations. That or there is a divergence for the gradient calculations. That being said, there are some successful takeaways from this. The first being that the approximation method was useful for the first couple hundred landmarks as it reduced computational time and increased accuracy for the linear SVM. Additionally, it showed to be incredibly effective when I was using my K-Means Nystroem transformation on the baseline SVM. It again increased accuracy and reduced the runtime. Another interesting impression that I got from this was that my Nystroem transformation outperformed the sk-learn implementation in accuracy at every landmark step and blew it out of the way in terms of runtime. I do not think that using KMeans would be that much faster than randomly selecting points for landmarks. I looked into how I was calculating time and everything checked out. I surprised myself with how well the implemented transformation worked. I imagine there is some expensive function within the sk-learn function that is causing their transformation to be so slow.

4 Challenges

4.1 Computational Bottlenecks

This project has been a wild ride of computational woes and dataset pivots. The first and most glaring issue? Memory. Attempting to compute kernel matrices with high landmark counts sent my poor Macbook into overdrive. Crashes became a regular occurrence, and I found myself frantically scaling back ambitions just to keep things running. Parallelization was a noble attempt at keeping my laptop afloat but even then the memory demands from these models were hefty. At one point my activity monitor showed that my laptop had multiple cores that were using 20+ Gb of ram. This was shortly followed by my laptop crashing. I ultimately decided to reduce the dataset size to 5000 training samples and 2000 test samples. This was probably the best decision I made for this project as it allowed to routinely test and make modifications to my implementations.

4.2 Accuracy Degradation

The custom Nystroem method combined with the OvR SVM showed a noticeable degradation in accuracy as the number of landmarks increased. This may stem from a divergence issue similar to a runtime error I encountered earlier with unbounded weights, which I addressed by implementing gradient clipping. While clipping resolved the runtime error, the accuracy degradation persisted, suggesting that the issue could be tied to how the kernel approximations influence decision boundaries or the stability of the gradient updates during training. Further investigation into this phenomenon is needed to fully address this issue.

4.3 Dataset Challenges

I originally decided to use the online news popularity dataset. Both sklearn's and my implementations for the Nystroem method had the tendency to overfit, which was frustrating. Because of this I decided to switch to MNIST dataset. This added more complexity because instead of creating one classifier, I was now creating 10. For the longest time, I was struggling with getting my OvR SVM to predict more than a few classes. That is when I started added normalizations for everything beyond just data input. This feature weights, kernel approximations, and decision scores. This definitely helped with OvR as every binary classifier was a minority class since it was compare one digit to the rest.

5 Conclusion

This project was all about trying to make SVMs work better and faster by using the Nystroem method to approximate the kernel matrix. That being said, there were definitely challenges along the way. Memory issues were a huge problem. One of the biggest lessons I learned was how important normalization is—when I normalized everything, like the features and kernel approximations, the OvR SVM became way more accurate, especially for harder-to-classify digits. Overall, even though there were challenges, this project showed how useful the Nystroem method can be when combined with SVMs, as long as you take the time to handle the data and computations carefully. I am definitely more interested in seeing how different landmark selection and further optimizations compare.

References

1. "Support Vector Machines," Scikit-learn. (https://scikit-learn.org/1.5/modules/svm.html)
2. "MNIST in CSV format," Oddrationale. (https://www.kaggle.com/datasets/oddrationale/mnist-in-csv/data?select=mnist_train.csv)
3. "The MNIST Database of Handwritten Digits," Yann LeCun. (https://yann.lecun.com/exdb/mnist/)
4. "Cyclical Feature Engineering Example," Scikit-learn. (https://scikit-learn.org/1.5/auto_examples/applications/plot_cyclical_feature_engineering.html#sphx-glr-auto-examples-applications-plot-cyclical-feature-engineering-py)
5. "Nyström Method for Kernel Approximation," Scikit-learn. (https://scikit-learn.org/dev/modules/generated/sklearn.kernel_approximation.Nystroem.html)
6. "Singular Value Decomposition," Wikipedia. (https://en.wikipedia.org/wiki/Singular_value_decomposition)
7. "Kernel Approximation," Scikit-learn. (https://scikit-learn.org/1.5/modules/kernel_approximation.html)
8. "Kernel Approximations for Efficient Machine Learning," Peekaboo Vision. (https://peekaboo-vision.blogspot.com/2012/12/kernel-approximations-for-ef
html)
9. "Nyström Approximation Overview," Andrew Charles Jones. (https://andrewcharlesjones.github.io/journal/nystrom-approximation.html)
10. "Low-Rank Matrix Approximations," Wikipedia. (https://en.wikipedia.org/wiki/Low-rank_matrix_approximations)