Kernelized Linear Classification

Statistical Methods for Machine Learning

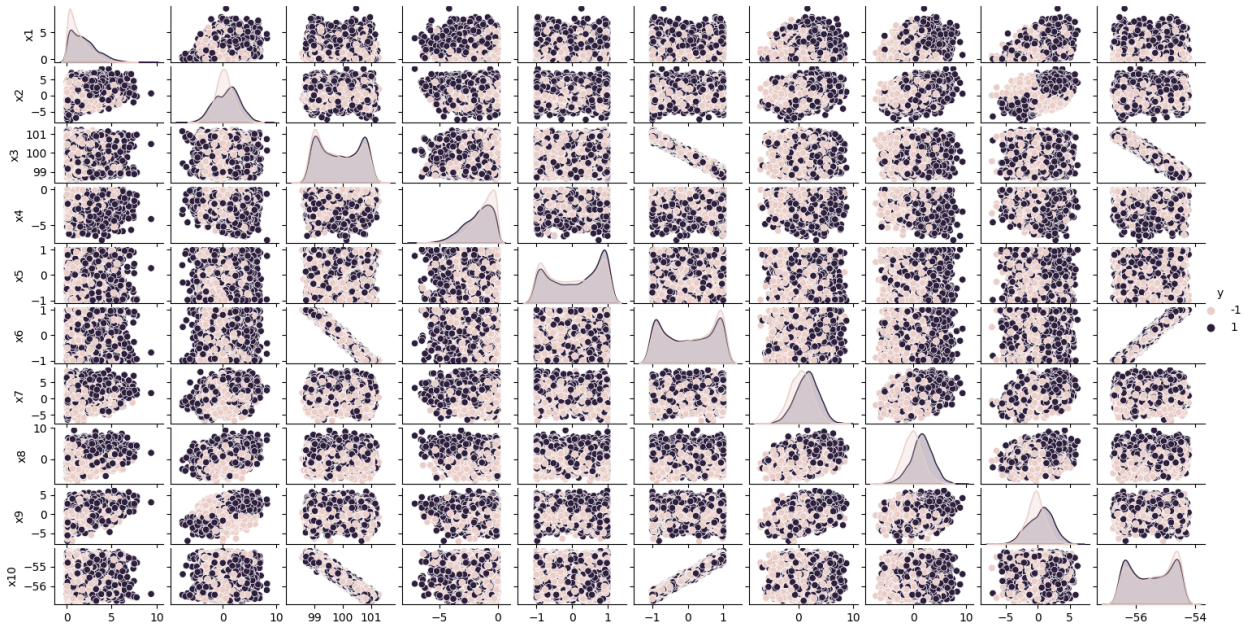
University of Milan,
Department of Computer Science
Vadim Sokolov

2024

Table of Contents:

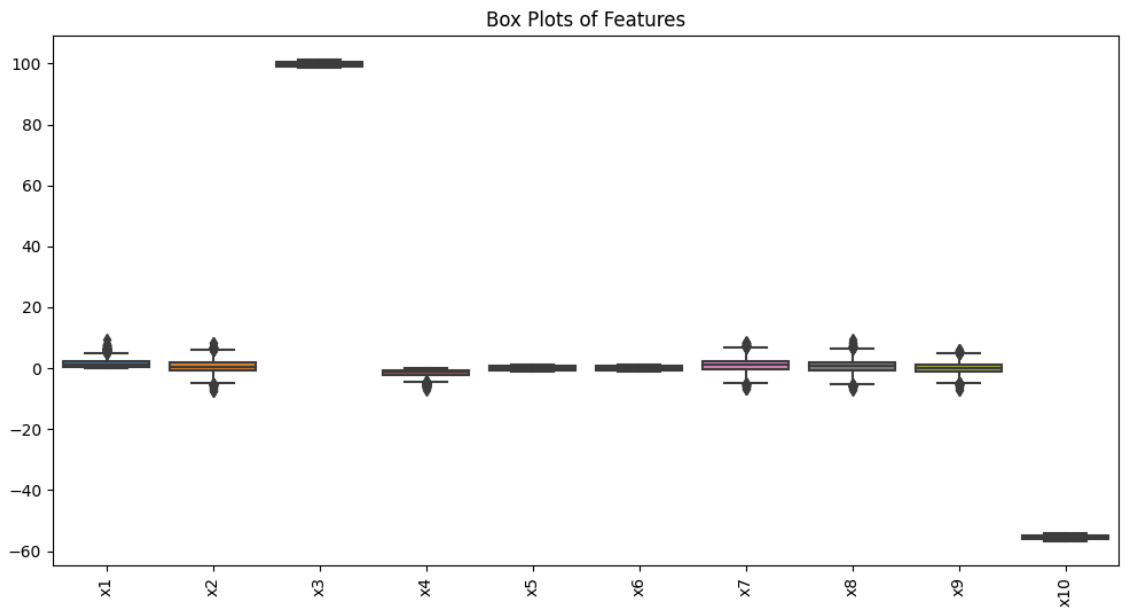
1. Dataset Overview

1.1 Pair Plot (Relationships between Features)



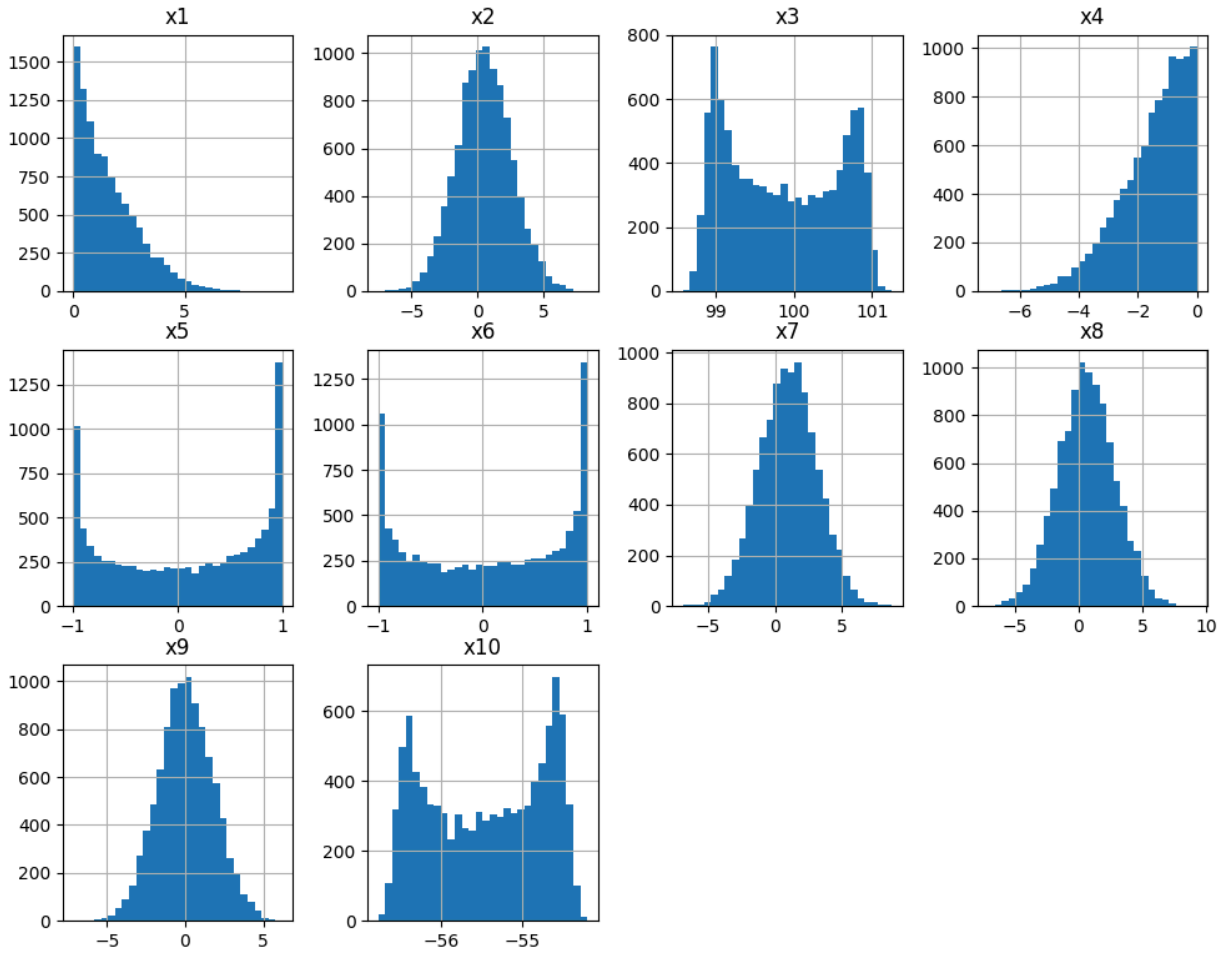
A pair plot helps visualize relationships between each pair of features and the distribution of individual features. This is particularly useful for smaller datasets with fewer features that fit in our case.

1.2 Box Plots (Outliers and Spread)



Box plots are useful for spotting outliers and understanding the spread of each feature.

1.3 Distribution of Individual Features



We can visualize the distribution of individual features using histograms. This helps us understand the spread and skewness of each feature.

1.4 Labels distribution

Number of labels with value 1 is 4992
Number of labels with value -1 is 5008

It means that this dataset is balanced and there is no need for undersample or other methods reducing imbalance.
There are no labels other than 1 or -1, so there are no label outliers / missing values.

1.5 Split data into train and test

The data was split into train and test with test size of 40%~4000 values by train_test_split method from sklearn.

2. Perceptron

The Perceptron algorithm is a linear classifier, a type of algorithm used for supervised learning of binary classifiers. It attempts to find a separating hyperplane between two classes in a dataset.

1000 epochs Test Accuracy for Perceptron: 71.10%

3. SVM Pegasos without regularization

The algorithm iteratively updates the weight vector w based on misclassifications. There's no regularization (penalizing large weights), so the focus is purely on minimizing hinge loss. The loop continues until it meets a convergence criterion, checking for the stabilization of the margin and the presence of support vectors.

Performance:
Without regularization, the model might overfit the data, especially if the dataset contains noise or outliers. This algorithm is faster since no regularization factor is computed, but it may result in a model that performs worse on unseen data due to overfitting.

epoch 2000 time 78.50s
Test Accuracy for Pegasos: 64.10%
The accuracy is low due to underfitting and lack of regularization (see Performance Discussion).

4. SVM Pegasos log loss with regularization

The algorithm is based on **stochastic gradient descent** with **regularization** to find the optimal weight vector w that separates the data points with the largest margin. It uses **support vector detection** to identify important data points that influence the decision boundary. **Convergence** is determined based on the stability of the margin and the presence of support vectors from both classes.

Performance:
epoch 2000 time 227.50s
Test Accuracy: 81.10%

The accuracy is higher for the regularized algorithm.

5. Polynomial feature expansion of degree 2

5.1 Perceptron

To improve the performance of the Perceptron by using polynomial feature expansion of degree 2, we can transform the input features into a higher-dimensional space by adding polynomial combinations of the features. This transformation allows the Perceptron to learn more complex decision boundaries, potentially improving performance, especially for non-linearly separable data.

Steps to Apply Polynomial Expansion:

1. Expand the Features: Use polynomial feature transformation to expand the feature space.
2. Train the Algo: Train the algorithm on the expanded feature set.
3. Evaluate the Model: After training, evaluate the model on a test set.

Perceptron with Polynomial Features (Degree 2) Test Accuracy: 78.88%
The perceptron now gives better results.

5.2 Pegasos

Similar to perceptron, we can apply the steps for Pegasos algorithm
Pegasos with Polynomial Features (Degree 2) Test Accuracy: 79.31%

5.3 Interpret and Compare Weights

The magnitude and sign of each weight provide insights into how each feature influences the prediction. Larger weights (positive or negative) suggest that the feature has a stronger influence.

Interpret the Weights:

1. The weights indicate how much each feature influences the prediction.
- Positive weight: The feature increases the likelihood of a positive class.
- Negative weight: The feature decreases the likelihood of a positive class.
- Magnitude: A higher magnitude means a stronger influence (either positive or negative).
2. For polynomial features, weights corresponding to squared or interaction terms give insight into how combinations of features influence the model.

How to Compare:

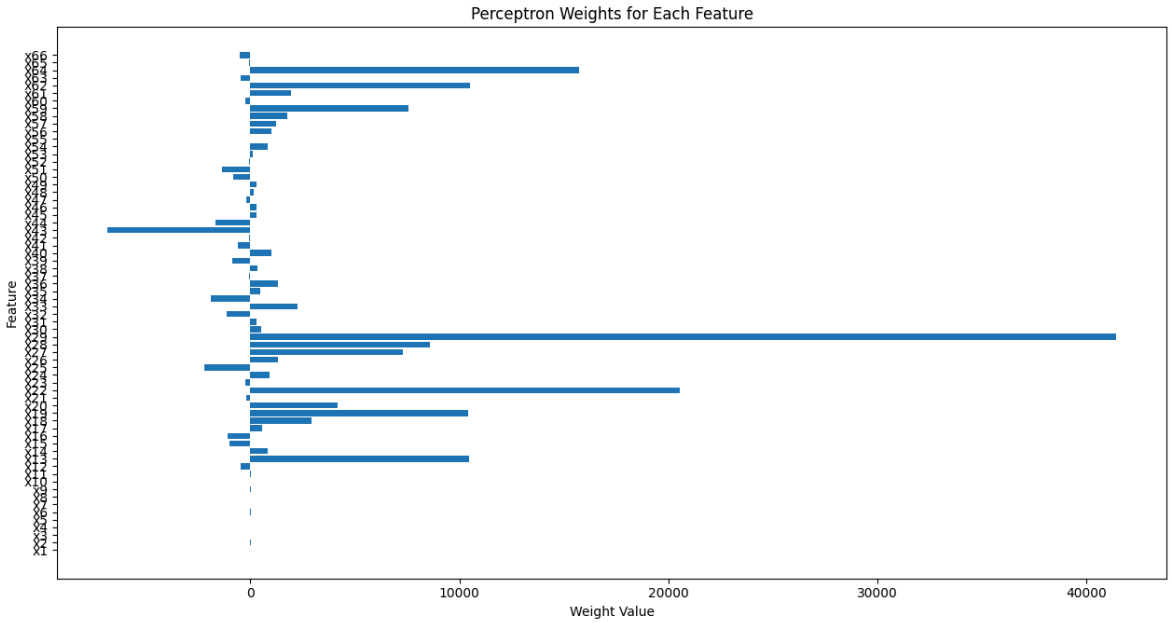
1. Positive vs Negative Weights: Features with positive weights push predictions towards one class, while negative weights push towards the other class.

2. Magnitude of Weights: Features with larger absolute values for weights are more influential in the model's predictions.
3. Bias Term: The bias term adjusts the overall prediction threshold.

5.4 Polynomial features discussion:

When we have applied polynomial feature expansion, the weights will also correspond to the new polynomial features (like squared terms or interaction terms). We can compare the relative importance of the original features and their polynomial transformations by analyzing the weights in the expanded feature set.

5.4.1 Perceptron:



Perceptron with Polynomial Features (Degree 2) Test Accuracy: 79.25%

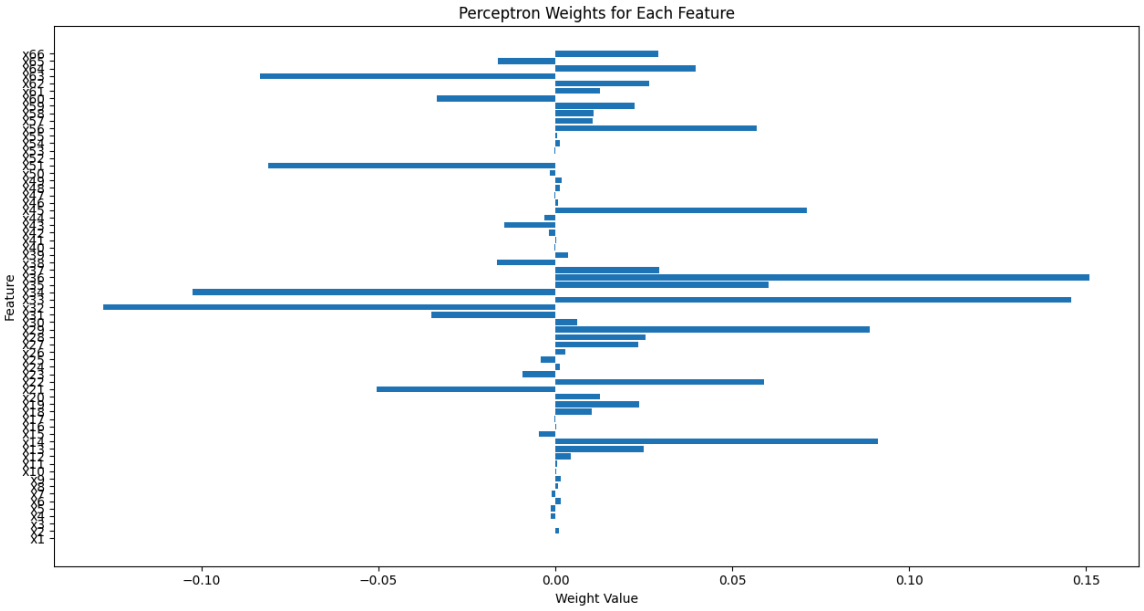
Bias term: -0.51

We can see that most of the features with high weights magnitudes also have positive value. It will push predictions towards one class and might affect the quality of the algorithm. Bias term though is closer to zero.

5.4.2 Pegasos parametrized

The choice of regularization parameter (λ), learning rate (η), and the number of epochs can greatly affect the convergence of Pegasos:

Lambda (λ): A value of 0.0001 means we're applying moderate regularization, which helps avoid overfitting by preventing the model from assigning overly large weights to features.
Learning rate (η): Our learning rate is quite small, which may slow down the learning process but provides stable and gradual updates to the weight vector. This often leads to better convergence.
Epochs: With 2000 epochs, the model had enough time to explore and adjust its weights based on the training data, given the small learning rate.



Pegasos with Polynomial Features (Degree 2) Test Accuracy: 70.85%
Bias term: -3.97e-05

Here we can see that the weights are distributed without bias towards large positive values. Bias term is also low.

6. Kernelized algorithms

6.1 Kernelized Gaussian perceptron

Kernelized Perceptron Algorithm

The Perceptron works as follows:

Initialize an empty list of support vectors and their corresponding labels. For each sample in the training set:

- Calculate the weighted sum of the kernel values between the current sample and the stored support vectors.
- Make a prediction based on the sign of this sum.
- If the prediction is wrong, add the current sample and its label to the list of support vectors.

For a new test sample, compute the prediction based on the kernel values between the test sample and all stored support vectors.

Key Steps:
gaussian_kernel(x1, x2, sigma): This computes the RBF kernel value between two points x1 and x2.

fit(X, y): During training, we loop through the dataset multiple times (defined by epochs). For each sample, we compute the prediction by summing the kernel values between the current sample and the stored support vectors. If the prediction is incorrect, the sample is added as a new support vector.

predict_single(x): This computes the weighted sum of the kernel evaluations for a single input x, using the support vectors and their labels.

predict(X): This is the vectorized prediction for the whole dataset, calling predict_single(x) for each sample.

After training, we can use the predict method to make predictions on test data.

Tuning:

sigma (bandwidth): Controls how much influence a support vector has over the prediction. A smaller σ results in more localized influence.
epochs: Increasing the number of epochs can help the Perceptron converge if the data is not linearly separable in the original space.

Gaussian Kernel Accuracy: 93.75%

Polynomial Kernel Accuracy: 84.40%

6.2 kernelized Pegasos

The kernelized Pegasos with the Gaussian and the polynomial kernels for SVM.

Key Concepts:

Pegasos Algorithm: The primal optimization method for SVM.
Kernelized Version: In kernelized SVM, we replace the dot product ⟨xi,xj⟩ with a kernel function K(xi,xj).

Kernelized Pegasos SVM:

In the kernelized version, we don't directly compute a weight vector w. Instead, we maintain a list of support vectors and their corresponding alphas. The prediction is done by evaluating the kernel function between the support vectors and new data points.
Kernelized Pegasos Algorithm:

Objective: Minimize the SVM hinge loss with the kernel function.
Support Vectors: Store only the data points that are active in the optimization (non-zero alphas).

Key Steps in the Code:

Kernel Functions: gaussian_kernel and polynomial_kernel compute similarity between points based on the chosen kernel.

Kernelized Pegasos SVM:
fit() trains the model by updating alphas, which are weights for the support vectors.During training, the kernel function replaces the dot product.

For prediction, the decision function is computed using the support vectors and their associated alphas and labels. After training, only the support vectors (non-zero alphas) are retained.

Hyperparameters to Tune:
sigma (for Gaussian kernel): Determines the width of the Gaussian kernel.
degree (for Polynomial kernel): Degree of the polynomial.
lam: Regularization parameter.
epochs: Number of iterations over the data.

Train kernelized pegasos with gaussian model:
epoch 25
--- 8585.58 seconds ---
Train kernelized pegasos with polynomial model

epoch 10
--- 3834.11 seconds ---
Gaussian Kernel Accuracy: 96.03%
Polynomial Kernel Accuracy: 87.95%

7. Models performance and theoretical interpretation

Model	Parameters	Test Accuracy (%)
Perceptron (Linear)	epochs = 1000, η = 0.01	71.10
Pegasos (Hinge Loss, No Regularization)	epochs = 1000	64.10
Pegasos (Hinge Loss, Regularized)	λ = 0.001, epochs = 2000, η = 0.0001	81.10
Perceptron (Polynomial Features, Degree 2)	Polynomial Degree = 2	78.88
Pegasos (Polynomial Features, Degree 2)	λ = 0.1, epochs = 100, η = 0.000001, Polynomial Degree = 2	79.31
Perceptron (Gaussian Kernel)	σ = 2.7, epochs = 25	93.75
Perceptron (Polynomial Kernel)	Degree = 3, coef0 = 0.7, epochs = 25	84.40
Pegasos (Gaussian Kernel)	σ = 1.3, λ = 0.001, epochs = 30	96.03
Pegasos (Polynomial Kernel)	Degree = 3, coef0 = 1.7, λ = 0.001, epochs = 10	87.95

7.1 Perceptron (epochs=1000, eta=0.01): Test Accuracy = 71.10%

Performance: The perceptron without any kernel or polynomial feature expansion achieves 71.10% accuracy, which is decent for a linear classifier on a potentially complex dataset.

Interpretation: The perceptron is a simple linear classifier that works well when the data is linearly separable. However, if the data has non-linear patterns, the linear boundary created by the perceptron may not be able to capture them, leading to lower accuracy.

7.2 Pegasos hinge loss (1000 epochs without regularization): Test Accuracy = 64.10%

Performance: The Pegasos algorithm, without regularization, performs worse than the perceptron.

Interpretation: Without regularization, Pegasos can overfit or underfit, depending on the margin between classes. The absence of regularization means it lacks control over the model's

complexity, leading to underperformance (64.10%) compared to the perceptron. Regularization often helps balance model complexity.

7.3 Pegasos with regularization (lam=0.001, epochs=2000, eta=0.0001): Test Accuracy = 81.10%

Performance: Introducing regularization significantly improves the accuracy to 81.10%.

Interpretation: Regularization (controlled by lam) reduces the risk of overfitting by penalizing large weight vectors. This encourages a simpler decision boundary that generalizes better to unseen data. The improvement in performance suggests that the original Pegasos model was overfitting or failing to generalize well without regularization.

7.4 Perceptron with Polynomial Features (Degree 2): Test Accuracy = 78.88%

Performance: Using polynomial features helps the perceptron capture more complex relationships, boosting the accuracy to 78.88%.

Interpretation: Polynomial feature expansion allows the perceptron to model non-linear relationships by effectively creating higher-order interactions between features. This explains the improved performance over the linear perceptron. However, it's still not as effective as kernelized approaches that might adapt better to the data's complexity.

7.5 Pegasos regularized (lam=0.1, epochs=100, eta=0.000001 with Polynomial Features Degree 2): Test Accuracy = 79.31%

Performance: Pegasos with polynomial features and regularization achieves 79.31%.

Interpretation: This model incorporates both regularization and non-linear feature interactions. Although regularization prevents overfitting, the fact that it does not outperform the regularized Pegasos on the original features suggests that the data may not benefit as much from polynomial expansion of degree 2, or the parameters (lam and eta) may need further tuning.

7.6 Perceptron with Gaussian Kernel (sigma=0.1, epochs=25): Accuracy = 93.75%

Performance: This kernelized perceptron with a Gaussian kernel performs significantly better, with 93.75% accuracy.

Interpretation: The Gaussian (RBF) kernel allows the perceptron to find non-linear decision boundaries, mapping the data into a higher-dimensional space where it becomes linearly

separable. The high accuracy shows that the dataset likely has non-linear structures that the perceptron alone could not capture.

7.7 Perceptron with Polynomial Kernel (degree=3, coef0=0.7, epochs=25): Accuracy = 84.40%

Performance: The polynomial kernel version of the perceptron achieves 84.40%.

Interpretation: The polynomial kernel allows the perceptron to model complex non-linear decision boundaries. While it performs better than the linear perceptron, it is not as effective as the Gaussian kernel, likely because the non-linear relationships in the data are better captured by the Gaussian kernel's smoother boundaries rather than the polynomial kernel's higher-order interactions.

7.8 Pegasos with Gaussian Kernel (sigma=0.1, lam=0.1, epochs=30): Accuracy = 96.03%

Performance: The Pegasos algorithm with a Gaussian kernel achieves the highest accuracy of 96.03%.

Interpretation: Pegasos with a Gaussian kernel not only captures complex non-linear relationships but also benefits from the margin-based SVM framework, which aims to maximize the margin between classes. The regularization parameter (lam) also helps control overfitting. This model's superior performance suggests that the dataset contains intricate, non-linear patterns best captured by a Gaussian kernel.

7.9 Pegasos with Polynomial Kernel (degree=3, coef0=1.7, lam=0.001, epochs=10): Accuracy = 87.95%

Performance: Pegasos with the polynomial kernel achieves 87.95%.

Interpretation: Similar to the perceptron with a polynomial kernel, this model can capture non-linear relationships in the data, although it is not as effective as the Gaussian kernel version. The relatively high accuracy compared to the linear models indicates that the polynomial kernel is useful, but the Gaussian kernel is better suited to this dataset's structure.

7.10 Overall Discussion:

7.10.1 Linear vs Non-linear Models: The perceptron and Pegasos models without kernels perform reasonably well on the dataset but struggle with more complex, non-linear patterns. The introduction of polynomial features improves performance but not as dramatically as kernelized approaches.

7.10.2 Kernelized Models: The models using Gaussian and polynomial kernels outperform the linear models, with the Gaussian kernel versions of both perceptron and Pegasos showing superior performance. This indicates that the dataset likely has non-linear structures that are better captured by kernelized methods.

7.10.3 Regularization: The Pegasos models show substantial improvement when regularization is applied. Regularization helps prevent overfitting by controlling the complexity of the decision boundary, and this effect is more pronounced in the Pegasos algorithm than in the perceptron.

7.10.4 Performance Trade-offs: While Pegasos with the Gaussian kernel performs best overall, the choice of kernel, regularization, and the tuning of parameters like sigma, lam, and eta plays a significant role in the balance between model complexity and generalization.

Overall, kernelized models (especially with the Gaussian kernel) significantly enhance performance, highlighting the importance of non-linear methods for capturing complex patterns in the dataset. Regularization further stabilizes these models, improving their generalization ability.

8. Overfitting and underfitting

8.1 Underfitting:

Definition: Underfitting occurs when a model is too simple to capture the underlying structure of the data, leading to poor performance on both the training and test sets. This typically happens when the model is unable to form sufficiently complex decision boundaries or when important features are missing.

Examples in Our Models:

Pegasos without regularization (64.10% accuracy): This model likely suffers from underfitting. Without regularization, the Pegasos algorithm isn't able to fully adapt to the dataset's complexity, leading to suboptimal performance. It likely lacks the flexibility to create a sufficiently expressive decision boundary.

Perceptron (71.10% accuracy): The linear perceptron, while performing better than unregularized Pegasos, still underfits the data. The linear model cannot capture complex, non-linear relationships, which is why it struggles to achieve high accuracy on this dataset.

Reason for Underfitting:

Simple Decision Boundaries: Both models use linear decision boundaries, which can only separate data that is linearly separable. If the dataset has complex relationships between features, these models won’t be able to capture them effectively.

No Regularization: In the case of Pegasos without regularization, there's no mechanism to balance the model's complexity, and the updates may not lead to a meaningful decision boundary.

8.2 Overfitting:

Definition: Overfitting occurs when a model becomes too complex and starts to fit not just the underlying patterns in the training data but also noise or random fluctuations. This results in excellent performance on the training set but poor generalization to unseen data (test set).

Examples in Our Models:

Perceptron with Polynomial Features (78.88% accuracy): While adding polynomial features increases complexity and improves performance, there is a risk of overfitting when high-degree polynomial expansions are used, especially if the model is not regularized. However, the test accuracy here suggests that the model balances performance, so the overfitting might be minimal.

Pegasos regularized with Polynomial Features (79.31% accuracy): This model is less likely to overfit compared to the perceptron because of the regularization term. However, the accuracy

isn't drastically higher than the linear Pegasos with regularization (81.10%), indicating that the degree-2 polynomial features may not add significant value and might even risk mild overfitting if used excessively.

Potential for Overfitting:

Perceptron with Gaussian Kernel (93.75% accuracy) and **Pegasos with Gaussian Kernel (96.03% accuracy)**: These models perform very well on the test set, but kernelized models, particularly with small regularization (lam) and high flexibility (sigma), are prone to overfitting. The fact that both models achieve high test accuracies suggests good generalization.

Pegasos with Polynomial Kernel (87.95% accuracy): While polynomial kernels are less flexible than Gaussian kernels, high-degree polynomial kernels can also lead to overfitting if the model starts capturing specific details in the training data rather than general patterns.

8.3 Managing Underfitting and Overfitting:

8.3.1 For Underfitting:

Increase Model Complexity: Introduce non-linear features (e.g., polynomial expansion or kernel functions). As seen with the perceptron and Pegasos models using polynomial and Gaussian kernels, increasing model complexity helps the model capture more intricate patterns.

Use More Epochs: Training the model for longer (more epochs) allows it to explore more complex solutions, especially for simpler models like the linear perceptron.

8.3.2 For Overfitting:

Regularization: Both L2 regularization (as in Pegasos with regularization) and early stopping (terminating training before the model overfits) are useful techniques. Regularization helps by penalizing large weight values, encouraging simpler models that generalize better. The improvement in the Pegasos model with regularization shows this effect clearly.

Control Kernel Parameters:

- **Gaussian Kernel**: Tuning the sigma value in the Gaussian kernel is crucial. A very small sigma makes the kernel too flexible, leading to overfitting, while a large sigma can cause underfitting.

- **Polynomial Kernel**: Similarly, controlling the degree of the polynomial kernel (degree=3 in our case) balances complexity. Higher-degree polynomials capture more detail but risk fitting noise in the training data.

8.3.3 Cross-Validation: Implementing cross-validation during training helps identify whether the model is overfitting or underfitting. By evaluating performance across different folds of the data, we can better estimate how well the model generalizes.

9. Computational costs

The computational cost of machine learning algorithms, especially kernelized models, is a critical factor when evaluating their practicality. In our project, we observed that **kernelized versions took a long time to converge**, which is common given the complexity of these methods. Let's break down the computational costs and why kernelized models, in particular, can be so resource-intensive.

9.1 Computational Costs of Different Models:

9.1.1 Linear Perceptron and Pegasos (Without Kernel):

Time Complexity: For both the **linear perceptron** and **linear Pegasos**, the time complexity per epoch is roughly **O(n * d)**, where **n** = number of samples, **d** = number of features

Both algorithms involve a simple dot product between the feature vector x and the weight vector www for each sample. This scales relatively well with larger datasets since the operations are linear in both the number of samples and the number of features.

Memory Complexity: The memory requirement for these models is **O(d)**, where **d** is the number of features. This is efficient, and in our case (with standard features), the computational load is manageable.

9.1.2 Polynomial Feature Expansion:

Time Complexity: After expanding the features with a **polynomial transformation of degree 2**, the number of features grows substantially. Specifically, the number of features increases from d to O(d^2). Therefore, the time complexity for algorithms that use polynomial features becomes **O(n * d^2)**.
This quadratic growth in the number of features makes training slower, especially for models like Pegasos or perceptron, which must now handle a much larger feature space.

Memory Complexity: The memory cost also scales to **O(d^2)**, which increases significantly with larger feature sets. Storing and processing these expanded features adds to the overall resource consumption.

9.1.3 Kernelized Perceptron and Pegasos:

The kernelized versions (with Gaussian or polynomial kernels) have substantially higher computational costs due to their non-linear nature.

9.1.3.1 Gaussian Kernel (RBF):

Time Complexity: Each epoch of the **Gaussian kernel perceptron or Pegasos** requires calculating the kernel function K(x_i, x_j)between pairs of samples. This involves a distance computation for each pair of samples, making the time complexity per epoch **O(n^2 * d)**, where **n** = number of samples, **d** = number of features

The quadratic dependency on n means that the cost grows very quickly as the dataset size increases. This is the primary reason why kernelized models are much slower to converge, especially for large datasets.

Memory Complexity: The memory cost is **O(n^2)** because the kernel matrix stores the similarity between all pairs of samples, which must be held in memory for efficient training. This makes kernelized methods impractical for very large datasets due to the quadratic memory growth.

9.1.3.2 Polynomial Kernel:

Time Complexity: Similar to the Gaussian kernel, the polynomial kernel also requires computing pairwise similarities between samples, resulting in a time complexity of **O(n^2 * d)**. The actual computation depends on the polynomial degree, as raising the dot product to higher powers increases computational demand.

Memory Complexity: Also **O(n^2)**, because we need to store the kernel matrix for all sample pairs. This becomes resource-intensive for large datasets.

9.2 Why Kernelized Models Take Longer to Converge:

Pairwise Kernel Computations: Both Gaussian and polynomial kernels involve computing the similarity between every pair of training samples. This quadratic growth in computation makes kernelized models significantly slower than linear models.

Difficulty of Separating Non-Linear Data: While kernelized models have more expressive power, they may require more epochs to converge to an optimal solution due to the complexity of the non-linear decision boundaries. In our case, we found that **Pegasos with the Gaussian kernel** took significantly longer to converge, even though it achieved a higher accuracy.

Complex Decision Boundaries: As the models aim to separate non-linearly separable data, the decision boundaries become more intricate, requiring more iterations and finer weight adjustments to converge.

9.3 Key Parameters Impacting Computational Costs:

Kernel Parameters (σ for Gaussian, degree for Polynomial):

A smaller **σ** in the Gaussian kernel increases the complexity of the decision boundary, making the model more sensitive to small differences between points. This results in more support vectors and a slower convergence.

A higher **degree** in the polynomial kernel increases the dimensionality of the feature space, making the computation more intensive and convergence slower.

Regularization Parameter (λ):

The regularization parameter **λ** controls the trade-off between model complexity and fitting the training data. A very small λ may cause the algorithm to focus on fitting every training point, leading to more iterations before convergence and more time spent on optimizing the complex decision boundary.

Number of Support Vectors: In kernelized models, the number of **support vectors** plays a crucial role in determining computational costs. As more support vectors are needed to describe the decision boundary, the training time increases due to the frequent updates and kernel evaluations for a larger subset of points.

9.4 Improving Computational Efficiency:

To reduce the high computational cost, several strategies can be applied:

Use of Approximations: Different approximations can be used to approximate kernel functions, reducing the time complexity from **O(n^2)** to **O(n)** in some cases.

Parallelization: In this project, I used CPU computation. Many kernelized methods can benefit from parallel computation. By distributing kernel evaluations across multiple processors, training time can be reduced.

Tune Kernel Parameters: Adjusting **σ** for the Gaussian kernel or the **degree** for the polynomial kernel can help reduce the complexity of the decision boundary, speeding up convergence without sacrificing much accuracy.

9.5 Comparison of the computational costs between different models

Model	Time Complexity per Epoch	Memory Complexity	Convergence Speed	Computational Load
Linear Perceptron	O(n * d)	O(d)	Fast	Low
Pegasos without Regularization	O(n * d)	O(d)	Fast	Low
Pegasos with Regularization (Hinge Loss)	O(n * d)	O(d)	Fast	Low
Perceptron with Polynomial Features (deg 2)	O(n * d^2)	O(d^2)	Medium	Medium
Pegasos with Polynomial Kernel	O(n^2 * d)	O(n^2)	Slow	High
Pegasos with Gaussian Kernel	O(n^2 * d)	O(n^2)	Slow	Very High

10. Appendix

10.1 Perceptron with Polynomial Features (Degree 2) Test Accuracy: 78.88%

Feature 1: Weight = -0.5100000000000002
Feature 2: Weight = 13.893606617998444
Feature 3: Weight = 12.34944521603801
Feature 4: Weight = -23.620586318666938
Feature 5: Weight = -16.723665304206868
Feature 6: Weight = 20.439116407026603
Feature 7: Weight = -19.55637538601126
Feature 8: Weight = 5.859894250597445
Feature 9: Weight = 22.073887553861294
Feature 10: Weight = -0.4207287224345229
Feature 11: Weight = 18.288911048778687
Feature 12: Weight = -440.1228692903385
Feature 13: Weight = 10449.00023563632
Feature 14: Weight = 838.1582584491596
Feature 15: Weight = -992.5962240148892
Feature 16: Weight = -1092.9287181757934
Feature 17: Weight = 585.4232864810298
Feature 18: Weight = 2921.6880503414905
Feature 19: Weight = 10405.437061547382
Feature 20: Weight = 4180.693645010309
Feature 21: Weight = -199.08571063099467
Feature 22: Weight = 20529.57744854524
Feature 23: Weight = -226.0690591124983
Feature 24: Weight = 905.7776466889638
Feature 25: Weight = -2214.9103668510725
Feature 26: Weight = 1342.6795502602092
Feature 27: Weight = 7289.6768963393615
Feature 28: Weight = 8574.329348154371
Feature 29: Weight = 41420.20029880763
Feature 30: Weight = 545.1776367351339
Feature 31: Weight = 292.9082795815996
Feature 32: Weight = -1111.92127989384
Feature 33: Weight = 2257.343594228754
Feature 34: Weight = -1886.9173870681873
Feature 35: Weight = 476.81386364732
Feature 36: Weight = 1322.7772104272426
Feature 37: Weight = -41.65687365168206
Feature 38: Weight = 367.1786962335269

Feature 39: Weight = -842.7611561309117
Feature 40: Weight = 1003.308902136254
Feature 41: Weight = -583.1488940527182
Feature 42: Weight = -39.11542028126787
Feature 43: Weight = -6825.020692757325
Feature 44: Weight = -1652.0067190420716
Feature 45: Weight = 313.30223296471854
Feature 46: Weight = 308.52600541368537
Feature 47: Weight = -208.09168018106948
Feature 48: Weight = 148.19648316854085
Feature 49: Weight = 290.46668529973124
Feature 50: Weight = -798.4759240548976
Feature 51: Weight = -1333.899934730419
Feature 52: Weight = -54.92653869477473
Feature 53: Weight = 130.84757432021223
Feature 54: Weight = 846.9319907559511
Feature 55: Weight = -0.7710416371249302
Feature 56: Weight = 1036.256310407171
Feature 57: Weight = 1217.2809243570548
Feature 58: Weight = 1791.0710245011815
Feature 59: Weight = 7559.30562758123
Feature 60: Weight = -229.0708776452436
Feature 61: Weight = 1931.798055182003
Feature 62: Weight = 10511.451647626614
Feature 63: Weight = -468.3370606962206
Feature 64: Weight = 15740.331686079888
Feature 65: Weight = -43.1892024137251
Feature 66: Weight = -500.2955449853478

10.2 Pegasos with Polynomial Features (Degree 2) Test Accuracy: 70.85%

Feature 1: Weight = -3.970061386426988e-05
Feature 2: Weight = 0.001095317271366397
Feature 3: Weight = -7.940566218474924e-05
Feature 4: Weight = -0.0020457662367589593
Feature 5: Weight = -0.0015489438315632832
Feature 6: Weight = 0.002160636661891941
Feature 7: Weight = -0.0016451006678886086
Feature 8: Weight = 0.0006580976481293098
Feature 9: Weight = 0.0015935009234747772
Feature 10: Weight = 0.00012053334426899961

Feature 11: Weight = 0.0009957174575671004
Feature 12: Weight = 0.007391329778476694
Feature 13: Weight = 0.13932715117964795
Feature 14: Weight = 0.10563266393214325
Feature 15: Weight = -0.01176088082471361
Feature 16: Weight = -0.007988941118168743
Feature 17: Weight = 0.004508085600470771
Feature 18: Weight = 0.05101798829303752
Feature 19: Weight = 0.12358775923783985
Feature 20: Weight = 0.06789912760083457
Feature 21: Weight = -0.05580294073794776
Feature 22: Weight = 0.3310973087757115
Feature 23: Weight = -0.026203383485741232
Feature 24: Weight = 0.011525841532660765
Feature 25: Weight = -0.02715659295713639
Feature 26: Weight = 0.016952768654389473
Feature 27: Weight = 0.12772260108153627
Feature 28: Weight = 0.13968633783908843
Feature 29: Weight = 0.5081007406377509
Feature 30: Weight = 0.020146942293744457
Feature 31: Weight = -0.013555186130282604
Feature 32: Weight = -0.15210430201125302
Feature 33: Weight = 0.2181755029038255
Feature 34: Weight = -0.16345935599484712
Feature 35: Weight = 0.06438294664665176
Feature 36: Weight = 0.15090864912406443
Feature 37: Weight = 0.010168256853445516
Feature 38: Weight = -0.006339117106483183
Feature 39: Weight = -0.002021356737361367
Feature 40: Weight = 0.007707890230111892
Feature 41: Weight = -0.003186411677208136
Feature 42: Weight = -0.00030769615608042917
Feature 43: Weight = -0.06802049058459103
Feature 44: Weight = -0.017078393463345973
Feature 45: Weight = 0.08191092945068301
Feature 46: Weight = 0.003529186199737533
Feature 47: Weight = -0.0022671434325822323
Feature 48: Weight = 0.0024214474790437515
Feature 49: Weight = 0.005492032226205501
Feature 50: Weight = -0.01048155194773688
Feature 51: Weight = -0.1220032616941202
Feature 52: Weight = -0.0007386784823134753
Feature 53: Weight = 0.0014839345948941011
Feature 54: Weight = 0.007861693358466615

Feature 55: Weight = 0.001820523860847676
Feature 56: Weight = 0.09065341868277557
Feature 57: Weight = 0.04709080614693652
Feature 58: Weight = 0.050312817153126785
Feature 59: Weight = 0.12352486381687477
Feature 60: Weight = -0.03524564182614587
Feature 61: Weight = 0.05931608477982823
Feature 62: Weight = 0.15009899908352062
Feature 63: Weight = -0.08151798429771688
Feature 64: Weight = 0.22705971980055115
Feature 65: Weight = -0.005648597701579488
Feature 66: Weight = 0.011242383528421064