CS 4/591: Neural Network
Assignment 2: Basic Models for Classification

Jyrus Cadman Robert McCourt Bethany Peña Gabriel Urbaitis

7 October 2024

Contents

1 Introduction

Classification is a key task of neural network models. This report focuses on the implementation and evaluation of three binary classification models, Widrow-Hoff Learning, Linear Support Vector Machine (SVM), and Logistic Regression. We also explore expanding beyond the binary classification task into multi-class classification by implementing and evaluating the Weston-Watkins Support Vector Machine.

In this assignment, we train and test each of our binary classification models on two, toy datasets from the Python scikit-learn library. We compare the loss convergance and runtime for training the three different models, and then compare the results from testing on a validation data set. Additionally, the Weston-Watkins SVM model is evaluated and tested using a multiclass dataset from scikit-learn.

The goal of this work is to deepen our understanding of how these algorithms work through implementing them in Python code and testing and comparing their performance across different datasets.

2 Classification Methods

2.1 Widrow-Hoff Learning

Widrow-Hoff learning, or Least Mean Squares (LMS) algorithm is a direct application of the least-squares regression method used for binary classification, i.e, least-squares classification. Widrow-Hoff learning is used in shallow, supervised learning where parameters are adjusted in an iterative fashion to approximate a desired target function.

This implementation of Widrow-Hoff learning focuses on a **single layer network with n input nodes and a single output node**. Given a set of training data, the least squares regression method aims to solve the following optimization problem

$$\min \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

When the sample output can only be -1 or 1, the loss function, L_i can be defined as

$$L_i = (1 - y_i \hat{y}_i)^2$$

This loss function measures the squared error between the target value (label) y_i and the predicted output value $\hat{y}_i$ The gradient update rule is given by

$$\overline{W} \leftarrow \overline{W} + \alpha y_i (1 - y_i \hat{y}_i) X_i^T$$

Which can be derived by computing the gradient of L_i with respect to $\overline{W}$

$$\frac{\partial L_i}{\partial \overline{W}} = \frac{\partial}{\partial \overline{W}} ((1 - y_i \overline{W}^T X_i)^2)$$

2.1.1 Implementation details)

The Widrow-Hoff learning algorithm is based on a perceptron model implemented in Python. This implementation (Listing 1) assumes that the bias term is absorbed into the sample matrix, X, and focuses on adjusting the weights based on the Widrow-Hoff update rule.

Attributes:

weights: An array containing the weights to be learned by the model. The weights are initialized randomly before training.

learning rate: The learning rate (`learning_rate`) is a hyperparameter that must be specified by the user. It controls the step size during weight updates.

errors: A list that tracks the error at each epoch during training.

Methods:

forward: Computes the forward pass of the model. It calculates the dot product of the weights and the input samples ($\mathbf{w}^T\mathbf{X}$). The output is clipped to avoid numerical overflow.

loss: Computes the squared error loss between the predicted output and the true label, given by $(1 - y_i\hat{y}_i)^2$. The result is clipped to avoid overflow.

fit: Trains the model using the training data and corresponding labels. The weights are updated at each step based on the Widrow-Hoff rule: $\overline{W} \leftarrow \overline{W} + \alpha y_i(1 - y_i\hat{y}_i)X_i^T$. The method also tracks and stores the error for each epoch.

predict: Applies the sign activation function on the forward pass results to predict the class labels ($\{-1, 1\}$) for the input samples.

2.2 Linear Support Vector Machine (SVM)

A linear support vector machine uses the same neural architecture as the Widrow-Hoff method, but uses a different loss function. The Widrow-Hoff method calculates the loss using a least-squares method. As a result, the loss solely depends on whether or not the classification is correct. The SVM improves upon this using a "hinge loss" function, given below, which penalizes predictions that do not lie within a margin of confidence.

$$\begin{aligned} L_i &= \max(0, 1 - y_i\hat{y}_i) \\ &= \max(0, 1 - y_i(\mathbf{w} * \mathbf{x_i}^T)) \end{aligned}$$

We can search for the optimal parameters of the using the gradient descent method. The gradient of the loss for the *ith* sample with respect to the weights is given by:

$$\frac{\delta L_i}{\delta \mathbf{w}} = -y_i\mathbf{x_i}^T$$

We can use the gradient to iteratively update our weight predictions by using the following expression. The following expression involves the regularization parameter, λ, and prevents overly large weights.

$$W = W(1 - \alpha\lambda) + \alpha(y_i\mathbf{x_i^T}) * I(y_i\hat{y}_i < 1)$$

2.2.1 Implementation details

The linear SVM is implemented in a Python class (Listing 2). This model assumes that the bias is absorbed into the sample matrix, X, as a row of ones and an additional weight parameter.

Linear_SVM

Attributes:

weights: an array containing the weights to be learned by the SVM; they are randomly initialized before training.

learning rate: the learning rate must be set by the user.

C: the regularization parameter must be set by the user.

Methods:

forward: computes the forward pass of the neural network; this is done by computing $\mathbf{w}^T\mathbf{X}$.

loss: Computes the hinge loss for a prediction.

update: Updates the weight predictions according the gradient descent algorithm, using the SVM regularization parameter.

fit: Learn weights using training data and labels.

predict: applies the sign activation function to predict the class labels of the input samples based on the forward pass.

2.3 Logistic Regression

Logistic Regression is a well-known method for binary classification that uses a logistic function to model the probability of an instance belonging to a particular class. Unlike Widrow-Hoff Learning and Linear SVM, which directly predict class labels, logistic regression outputs probabilities between 0 and 1.

The logistic function (also known as the sigmoid function) is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

where $z = \mathbf{w}^T\mathbf{x} + b$, $\mathbf{w}$ is the weight vector, $\mathbf{x}$ is the input feature vector, and b is the bias term.

The probability of an instance belonging to the positive class ($y = 1$) is modeled as:

$$P(y = 1|\mathbf{x}) = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right)$$

The loss function used in this implementation is the binary cross-entropy loss:

$$L(\mathbf{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

where m is the number of training examples, y_i is the true label, and $\hat{y}_i$ is the predicted probability.

The gradients of the loss function with respect to the weights and bias are used in the optimization process to update these parameters.

2.3.1 Implementation details

The Logistic Regression algorithm is implemented in a Python class named `LogisticRegression` (Listing 5). This implementation separates the bias term from the weight vector for clarity.

LogisticRegression

Attributes:

`weights`: A NumPy array containing the weights to be learned by the model. The weights are initialized as zeroes before training.

`bias`: A scalar value representing the bias term, initialized as 0 before training.

`learning_rate`: A hyperparameter specified by the user that controls the step size during weight updates.

`num_iterations`: The number of iterations for the optimization process.

Methods:

`sigmoid(z)`: Computes the sigmoid function given input `z`.

`forward(x)`: Computes the forward pass of the model. It calculates the sigmoid of the dot product of the weights and the input samples plus the bias ($\sigma\left(\mathbf{w}^T\mathbf{X}+b\right)$).

`loss(y_pred, y`: Computes the binary cross-entropy loss between the predicted probabilities and the true labels. Note that the implementation adjusts the true labels from $\{-1, 1\}$ to $\{0, 1\}$ for the loss calculation.

`fit(X, y)`: Trains the model using the training data and corresponding labels. The weights and bias are updated at each iteration using gradient descent. The method also tracks and stores the loss for each iteration, returning the loss history.

`predict(X)`: Applies a threshold of 0.5 on the predicted probabilities to determine the class labels (-1 or 1) for the input samples.

2.4 Weston-Watkins SVM

The Weston Watkins SVM is one of the neural architectures for multiclass models. Much the same way as the Multiclass Perceptron, there is a different update rule for the weights of incorrect classes than the weights of correct classes. Correct class weights are increased to improve their outputs and incorrect class weights are decreased.

The loss function is calculated as:

$$L_i = \sum_{r:r\neq c(i)} \max\left(\bar{W}_r \cdot \bar{X}i^T - \bar{W}c(i) \cdot \bar{X}_i^T + 1, 0\right)$$

This is designed to make the class index r with the largest value for $W_r \cdot X_i^T$ the correct class prediction and to penalize deviation from this ideal condition.

The gradient is calculated as:

$$\frac{\partial L_i}{\partial W_r} = \begin{cases} -\bar{X}i^T \left[\sum_{j \neq r} \delta(j, \bar{X}_i)\right] & \text{if } r = c(i) \\ \bar{X}_i^T \left[\delta(r, \bar{X}_i)\right] & \text{if } r \neq c(i) \end{cases}$$

where $\delta(j, \bar{X}_i)$ is an indicator function which is 1 when the j-th class separator has a positive contribution to the loss function for sample $\bar{X}_i$.

$$\delta(j, \bar{X}_i) = \begin{cases} 1 & \text{if } W_j \cdot \bar{X}_i^T - W_{c(i)} \cdot \bar{X}_i^T + 1 > 0, \\ 0 & \text{otherwise.} \end{cases}$$

As the gradient tells us the direction of steepest ascent, to perform gradient descent, we move in the opposite direction of the gradient to minimize the loss function. This means we flip the sign to achieve the following stochastic gradient-descent step:

$$\bar{W}_r \leftarrow \bar{W}_r + \alpha \begin{cases} \bar{X}i^T \left[\sum j \neq r \delta(j, \bar{X}_i)\right] & \text{if } r = c(i) \\ -\bar{X}_i^T \left[\delta(r, \bar{X}_i)\right] & \text{if } r \neq c(i) \end{cases}$$

2.4.1 Implementation details)

The Weston Watkins SVM is implemented in a Python class.

weston_watkins

Attributes:

W: an array containing the weights to be learned by the SVM; they are randomly initialized before training.

lr: the learning rate must be set by the user.

epochs: the number of epochs must be set by the user.

Methods:

loss_function: Computes the Weston Watkins loss for a sample. (Used purely for tracking)

gradient: Computes how the weights need to be adjusted for a given sample. This is done by computing $\frac{\partial L_i}{\partial W_r}$.

fit: Learn weights using training data and labels.

predict: Predicts the class labels of the input samples based on the forward pass.

3 Data Sets

3.1 Binary Classification Data Set

In this experiment, we utilized two well-known toy datasets from the scikit-learn library: the wine and breast cancer datasets. Both datasets are commonly used for classification experiments due to their manageable

size and variety of features.

- **Breast Cancer Dataset:** This dataset contains 569 samples with 30 features that describe characteristics of cell nuclei from breast cancer biopsies. It is a binary classification problem, where the goal is to predict whether a tumor is malignant or benign. The data is already structured for binary classification
- **Wine Dataset:** The wine dataset consists of 178 samples, each with 13 numerical features derived from chemical analysis of wines grown in the same region of Italy. The dataset originally contains three classes, corresponding to three types of wine. Since our models were designed for binary classification, we filtered out of class of wine and used only two for training and testing purposes

3.2 Multi-Class Classification Data Set

In this experiment, we utilized the wine dataset from sci-kit learn. The wine dataset is a toy dataset commonly used for classification experiments due to its manageable size and variety of features.

- **Wine Dataset:** The wine dataset consists of 178 samples, each with 13 numerical features derived from chemical analysis of wines grown in the same region of Italy. The dataset contains three classes, corresponding to three types of wine. We used all 3 classes for the multi-class classification portion of our research.

4 Evaluation

In this experiment, we divided each dataset into training and testing subsets to evaluate the performance of our classification models. 75% of the data was reserved for training, allowing the models to learn the underlying patterns, while the remaining 25% was allocated for testing to assess the generalization ability of the models. The training set was used to optimize the model parameters, and the testing set provided an unbiased evaluation of the model's performance. This split ensured that we could measure the accuracy, execution time, and accumulated loss on data that the model had not seen during training.

4.1 Training

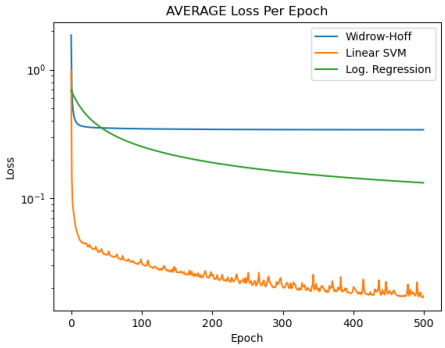
4.1.1 Binary Classification with Breast Cancer Dataset

We used scikit-learn's `train_test_split` method to split the breast cancer dataset into training and testing samples. We used the default splitting proportion which uses 25% of the data for testing, and 75% of the data for training.

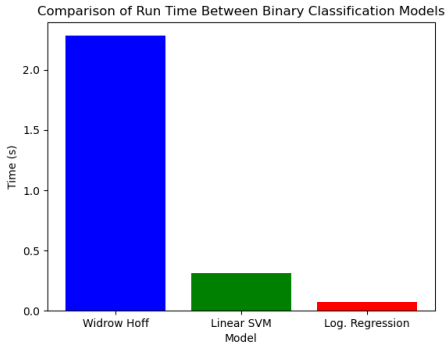
For all of our models the bias is absorbed into the sample matrix, X, as a row of ones and an additional weight parameter. The initial weights in our models are uniformly chosen between values -1 and 1. We use the same initial weights for each model so we can directly compare the results.

We also transform the data using scikit-learn's `StandardScaler` tool. This tool transforms the data so that it has its mean at 0 and a variance of 1. Since our binary classifications methods are optimized using gradient descent, this greatly improved the ability of our algorithms to converge to a final solution.

We trained the binary classification models using the learning rates 0.01 and 0.001 and trained for 500 epochs. For both learning rates (see plot a. in figures 1 and 2), the loss converged within 500 epochs. We



(a) Loss Comparison



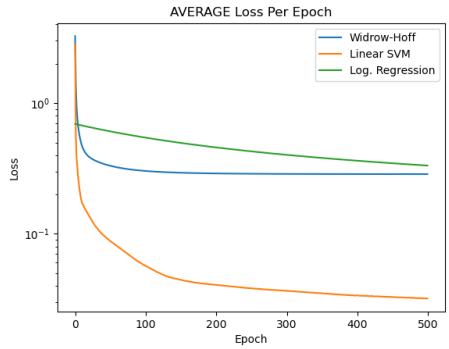
(b) Time to fit model (run 500 epochs)

Figure 1: Comparison between Widrow-Hoff, Linear SVM, and Logistic Regression classification methods using a 0.01 learning rate on the breast cancer dataset using a semilog plot.

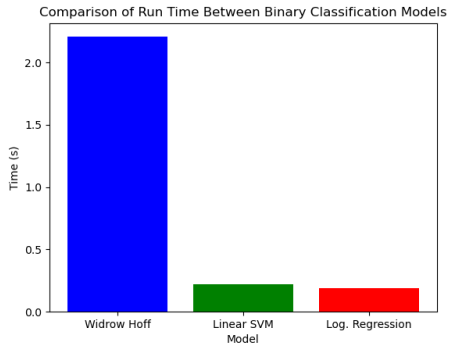
noticed some slight variation in the loss convergence depending on how the weights were randomized, which is to be expected.

For both learning rates, the Widrow-Hoff model appeared to converge fastest. The log regression and SVM model appeared to converge at similar rates. The SVM converged to a lower loss than the other two models. Modifying the learning rate to 0.001 resulted in a slightly slower convergence for all models. This is expected since the learning rate determines how much the weights are updated each iteration.

Additionally we measured the time for each algorithm to train. Plot (b.) in figures 1 and 2 compares the run times. The Widrow-Hoff model took the longest to run of the three models.



(a) Loss Comparison



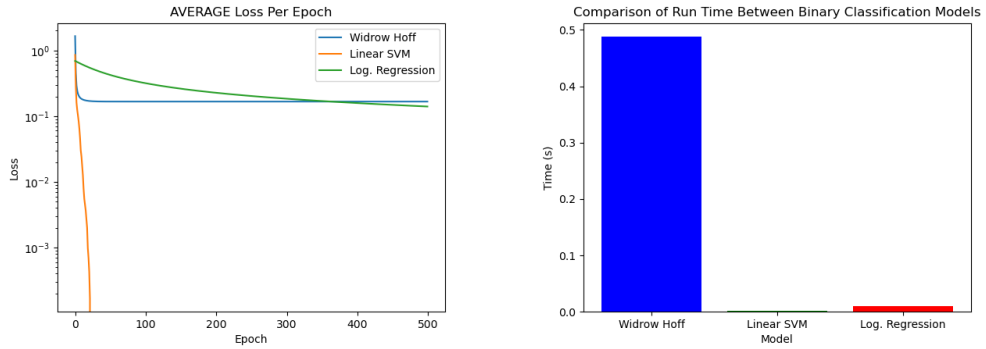
(b) Time to fit model (run 500 epochs)

Figure 2: Comparison between Widrow-Hoff, Linear SVM, and Logistic Regression classification methods using a 0.001 learning rate on the breast cancer dataset on a semilog plot.

4.1.2 Binary Classification with Wine Dataset

The wine dataset was originally meant for multi-class classification. In order to adapt it to our binary methods, we filtered out the data and labels from the third class of wine and used only the first two classes in our training and testing.

We used scikit-learn's `train_test_split` method to split the dataset into training and testing samples. We used the default splitting proportion which uses 25% of the data for testing, and 75% of the data for training.



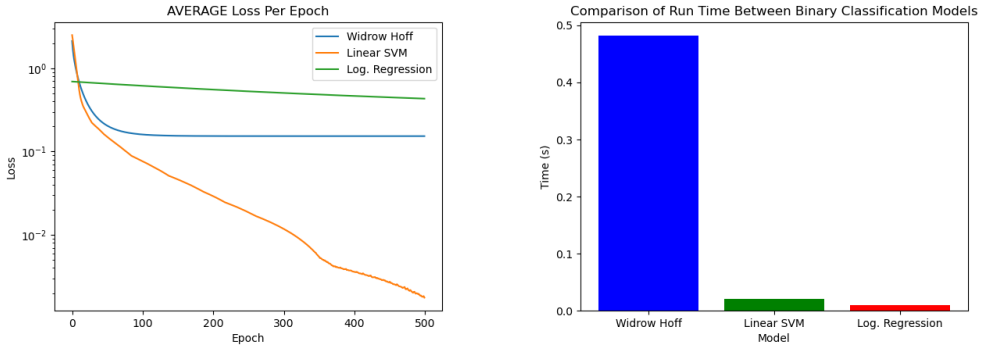
(a) Loss Comparison

(b) Time to fit model (run 500 epochs)

Figure 3: Comparison between Widrow-Hoff, Linear SVM, and Logistic Regression classification methods using a 0.01 learning rate on the wine dataset on a semilog plot.

For all of our models the bias is absorbed into the sample matrix, X, as a row of ones and an additional weight parameter. We also transform the data using scikit-learn's `StandardScaler` tool. This tool transforms the data so that it has its mean at 0 and a variance of 1. Since our binary classifications methods are optimized using gradient descent, this greatly improved the ability of our algorithms to converge to a final solution. There was some slight variation in the loss convergence from the randomized weights, but this is to be expected.

We trained the binary classification models using the learning rates 0.01 and 0.001 and trained for 500 epochs. For this dataset, with a learning rate of 0.01, the SVM converged to a signficiantly lower learning rate than the other two models. The log regression model converged at a much more gradual pace compared to the Widrow-Hoff model. Modifying the learning rate to 0.001 resulted in a much more gradual convergence for all models. This is expected since the learning rate determines how much the weights are updated each iteration.



(a) Loss Comparison

(b) Time to fit model (run 500 epochs)

Figure 4: Comparison between Widrow-Hoff, Linear SVM, and Logistic Regression classification methods using a 0.001 learning rate on the wine dataset on a semilog plot.

4.1.3 Multiclass Training with the Wine DataSet

We used scikit-learn's train test split method to split the wine dataset into training and testing samples. We used half the samples for training and half for testing, and a random seed of 25 to select which samples went in each category for repeatability. The initial weights in our models are uniformly chosen between values -1 and 1, also with a random seed of 25. We use the same initial weights for each model so we can directly compare the results. We also transform the data using scikit-learn's StandardScaler tool. This tool transforms the data so that it has its mean at 0 and a variance of 1. Since the Weston Watkins method is optimized using stochastic gradient descent, this greatly improved the ability of our algorithm to converge to a final solution.

4.2 Testing

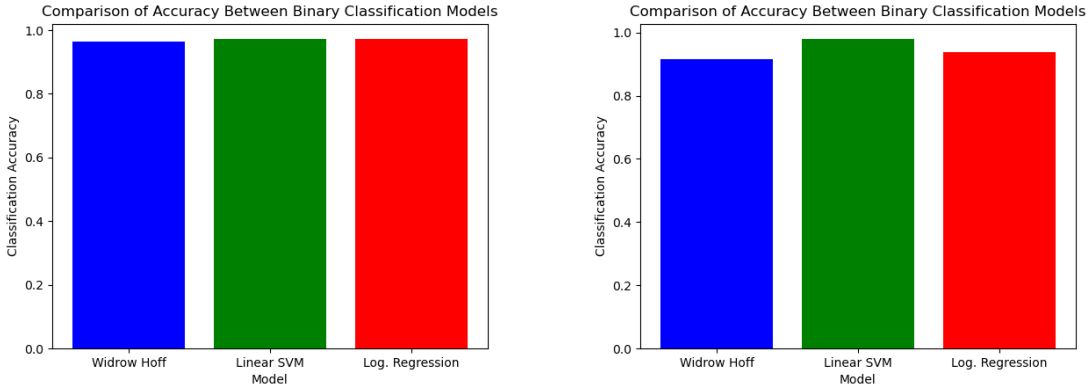
4.2.1 Binary Classification

To test the binary classification models we used the remaining 25% of the data that was not used in training.

4.2.2 Results from Breast Cancer Dataset

The proportion of accurate classifications for models trained using the 0.01 and 0.001 leanings rates is shown in figure 5. All models achieved high levels of accuracy for both learning rates. However, we achieved slightly higher average accuracy in using the 0.01 for the learning rate. This is due to less variation between the models.

However, as mentioned in training section, the slight variation could be a result of the random initialization of the weights rather than a reflection on the models.



(a) 0.01 Learning Rate

(b) 0.001 Learning Rate

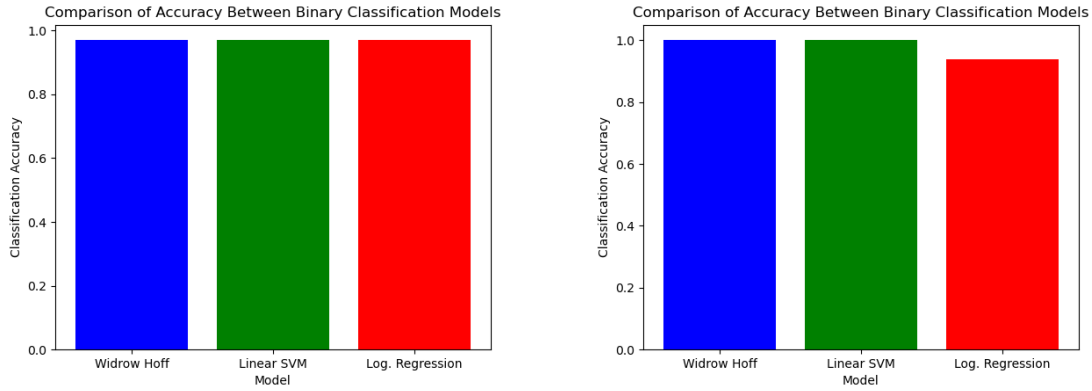
Figure 5: Comparison between Widrow-Hoff, Linear SVM, and Logistic Regression classification methods using different learning rates on the wine dataset.

4.2.3 Results from Modified Wine Dataset

The proportion of accurate classifications for models trained using the 0.01 and 0.001 leanings rates is shown in figure 5. All models achieved high levels of accuracy for both learning rates. The model trained with the

learning rate of 0.01 achieved 97% accuracy. The model trained the learning rate of 0.001 had high levels of accuracy, but the logistic regression model had a slightly lower (94%) accuracy than the other two models (97%).

However, as mentioned in training section, the slight variation could be a result of the random initialization of the weights rather than a reflection on the models.



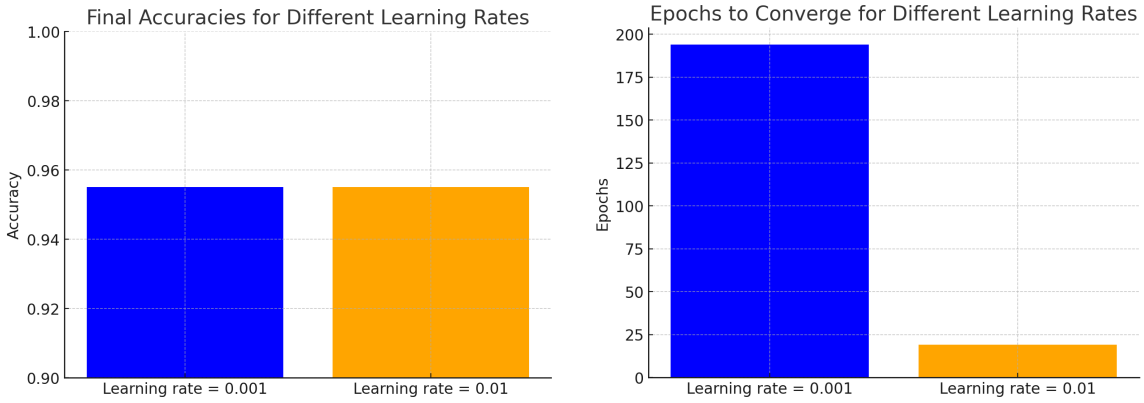
(a) 0.01 Learning Rate

(b) 0.001 Learning Rate

Figure 6: Comparison between Widrow-Hoff, Linear SVM, and Logistic Regression classification methods using different learning rates on the wine dataset.

4.2.4 Multiclass Classification with Wine Dataset

As seen in the confusion matrix for the 3 class Wine DataSet, Class 0 and 1 were totally predicted correctly, and Class 2 had 4 misclassifications, 3 as Class 1 and 1 as Class 0. The confusion matrix was identical for both the .001 and .01 learning rates. This doesn't mean that the same samples were misclassified, though it is likely that they were the same in both cases. The accuracy was thus also the same for both, though the learning rate of .01 converged in 19 epochs, compared to 194 for .001. This makes sense as the smaller learning rate would only be able to adjust the weights by about 1/10th smaller increments each time, so it would take it 10 times as long to find the minimum.



(a) Accuracy

(b) Time to fit model (run 500 epochs)

Figure 7: Comparison between accuracy and time to converge using different learning rates on the 3 class wine dataset.

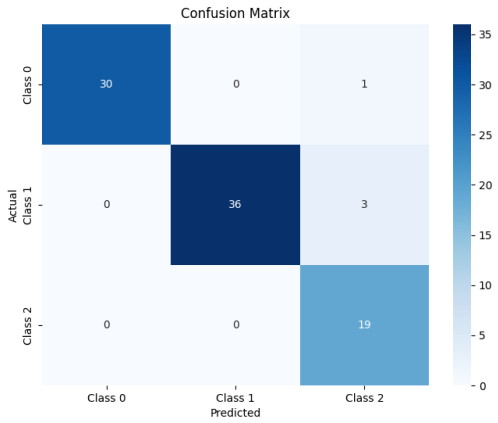


Figure 8: Confusion Matrix for both learning rates (they were identical)

5 Discussion

The objective of this experiment was to evaluate the performance of different classification models, Widrow-Hoff, Linear Support Vector Machine, Linear Regression, and Weston-Watkins Support Vector Machine. The experiment was conducted by implementing each classification algorithm as a Python class, and evaluated their effectiveness of each method across different hyper parameters to measure their accuracy, execution time, and accumulated loss. We tested our implementations on different datasets from the scikit-learn library, specifically the wine dataset was used for multi-class classification, and a modified version of the wine dataset and breast cancer datasets were used for binary classification.

Our results align with what we expected, as the Linear SVM is the more advanced binary classification algorithm out of the three and as a result had the highest average accuracy, as well as the second best execution time. However, the execution time of the Widrow-Hoff model stood out as an outlier, taking nearly 10 times longer to complete compared to Linear SVM, and about 20 times longer than Linear Regression. Despite Widrow-Hoff's reasonably good accuracy, its significantly longer training time highlights its inefficiency in comparison to the other methods.

We had an effective implementation of the Weston Watkins SVM method for multiclass classification, as it classified two of the 3 classes perfectly and only misclassified 4 samples in the third class. We only had one hyperparameter for comparison amongst runs of the code, learning rate. There was no difference in accuracy, provided we ran for enough epochs, but the smaller .001 learning rate converged in 194 epochs compared to the 19 epochs for the .01 learning rate.

6 Conclusion

Our analysis demonstrates that the Linear SVM model consistently outperforms the Widrow-Hoff algorithm in terms of both accuracy and execution time. While Widrow-Hoff achieves reasonably good accuracy, its training times are significantly loner compared to Linear SVM. Linear Regression on the other hand, delivers solid accuracy cand exhibits the fastest execution times of all three mdoels, making it a competitive option in terms of performance and efficieny. Overall, Linear SVM offers the best balance of speed and accuracy for these datasets, though Linear Regression is a strong candidate for cases where computational efficiency is a priority.

We were also able to demonstrate the effectiveness of the Weston Watkins SVM, achieving 95 percent accuracy for both .01 and .001 learning rates. An analysis of testing accuracy revealed 4 samples were misclassified for Class 2, 3 as Class 1 and 1 as Class 0, for both learning rates. As expected a 10 times smaller learning rate took roughly 10 times as many epochs to converge to 0 loss.

Appendix

```python
import numpy as np

class WidrowHoff:
    def __init__(self, num_inputs, learning_rate=0.01):
        self.learning_rate = learning_rate
        self.weights = np.random.uniform(-1, 1, num_inputs)
        self.errors = []

    def forward(self, X):
        return np.clip(X @ self.weights, -1e10, 1e10)  # Clipping to prevent overflow

    def loss(self, y_pred, y):
        return np.clip((1 - y_pred * y) ** 2, -1e10, 1e10)  # Clipping to prevent overflow

    def fit(self, X_train, y_train, max_epochs):
        for epoch in range(max_epochs):
            total_error = 0

            for X, y in zip(X_train, y_train):
                y_pred = self.forward(X)

                #calculate the error
                error = self.loss(np.array([y_pred]), np.array([y]))
                total_error += error

                self.weights += self.learning_rate * y * (1 - y * y_pred) * X

                if np.all(error == 0):
                    print(f"Converged at epoch {epoch}")
                    break

            avg_error = total_error / len(y_train)
            self.errors.append(avg_error)
            #print(f"Epoch {epoch} - Total Error: {total_error}")

        return np.array(self.errors)


    def predict(self, X):
        return np.where(self.forward(X) >= 0, 1, -1)
```

Listing 1: `widrowhoff.py`

```python
import numpy as np
import numpy.typing as npt

"""
TODO:
Do we need to use regularization in hinge loss function?
"""

class Linear_SVM:

    def __init__(self, weights: np.ndarray, learning_rate: float, C: float) -> None:
        self.weights = weights
        self.learning_rate = learning_rate
        self.C = C

    def forward(self, X: np.ndarray) -> np.ndarray:
        """Compute raw scores for input data X."""
        return X @ self.weights

```

```
    def loss(self, y: np.ndarray, yhat: np.ndarray) -> np.ndarray:
        """Hinge loss calculation."""
        return np.maximum(0, 1 - y * yhat)

    def update(self, yhat: np.ndarray, y: np.ndarray, X: np.ndarray) -> None:
        """Update weights using gradient descent."""
        for i in range(len(y)):
            if y[i] * yhat[i] < 1:
                regularization_term = self.weights*(1 - self.learning_rate*self.C)
                self.weights = regularization_term + self.learning_rate * (y[i] * X[i, :])

    def fit(self, X: np.ndarray, y: np.ndarray, max_epochs) -> np.ndarray:
        """Fit the model to the data."""
        loss_per_epoch = []
        for epoch in range(max_epochs):
            #print(f"Epoch: {epoch}")
            yhat = self.forward(X)
            loss = self.loss(y, yhat)
            self.update(yhat, y, X)

            epoch_loss = np.sum(loss) / len(y)
            loss_per_epoch.append(epoch_loss)

            if np.all(loss == 0):
                break

        return np.array(loss_per_epoch)

    def predict(self, X: np.ndarray) -> np.ndarray:
        """Predict class labels for samples in X."""
        return np.where(self.forward(X) >= 0, 1, -1)
```

Listing 2: `linear_svm.py`

```
import numpy as np

class LogisticRegression:
    """
    Logistic Regression classifier.
    """

    def __init__(self, learning_rate=0.01, num_iterations=100):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.weights = None
        self.bias = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def forward(self, X):
        return self.sigmoid(np.dot(X, self.weights) + self.bias)

    def loss(self, y_pred, y):
        return -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))

    def fit(self, X, y):
        num_samples, num_features = X.shape
        self.weights = np.zeros(num_features)
        self.bias = 0

        loss_per_epoch = []

        for _ in range(self.num_iterations):
```

```
            y_pred = self.forward(X)

            # Compute the loss and store it for the current epoch
            epoch_loss = self.loss(y_pred, (y + 1) / 2)
            loss_per_epoch.append(epoch_loss)

            # Gradient descent
            dw = (1 / num_samples) * np.dot(X.T, (y_pred - (y + 1) / 2))
            db = (1 / num_samples) * np.sum(y_pred - (y + 1) / 2)

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

        return np.array(loss_per_epoch)

    def predict(self, X):
        y_pred = self.forward(X)
        return np.where(y_pred > 0.5, 1, -1)
```

Listing 3: `logistic_regression.py`

```
from linear_svm import Linear_SVM
from logistic_regression import LogisticRegression
from widrowhoff import WidrowHoff
import time

import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler

# Breast Cancer Dataset
# Load Data
data = load_breast_cancer()
print(f"Class distribution: {sum(data.target == 1)} positive, {sum(data.target == 0)} negative")

# Normalize and Scale Sample Data
X = data.data
X = np.hstack([X, np.ones((X.shape[0], 1))]) # add bias to data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Grab labels and fix between -1 and 1
y = data.target
y = np.where(y == 1, 1, -1)

# split data into training and testing
X_train, X_Test, y_train, y_test = train_test_split(X_scaled, y)

#
# SVM Model
#
# add one to the weights matrix because we are absorbing the bias
svm = Linear_SVM(weights = np.random.uniform(-1, 1, X_train.shape[1]),
                 learning_rate=0.001,
                 C = 0)

# Fit model
start_time = time.time()
svm_loss_per_epoch = svm.fit(X_train, y_train, max_epochs=500)
end_time = time.time()
svm_time = end_time - start_time
```

```
44  print(f"SVM time: {end_time-start_time}")
45
46  # Predict Using Model
47  svm_preds = svm.predict(X_Test)
48
49  # Evaluate
50  svm_accuracy = np.sum(np.where(y_test == svm_preds, 1, 0)) / y_test.shape[0]
51  print(f"SVM Accuracy: {svm_accuracy}")
52
53  #
54  # Widrow Hoff
55  #
56  wh = WidrowHoff(num_inputs=X_train.shape[1], learning_rate=0.001) #Don't modify learning rate
57
58  # Fit model
59  start_time = time.time()
60  wh_loss_per_epoch = wh.fit(X_train, y_train, max_epochs=500)
61  end_time = time.time()
62  print(f"WH time: {end_time-start_time}")
63  wh_time = end_time - start_time
64
65
66  # Predict Using Model
67  wh_preds = wh.predict(X_Test)
68
69  # Evaluate
70  wh_accuracy = np.sum(np.where(y_test == wh_preds, 1, 0)) / y_test.shape[0]
71  print(f"WH Accuracy: {wh_accuracy}")
72
73  #
74  # Logistic Regression Model
75  #
76  logr = LogisticRegression(learning_rate=0.001, num_iterations=500)
77
78  # Fit model
79  start_time = time.time()
80  logr_loss_per_epoch = logr.fit(X_train, y_train)
81  end_time = time.time()
82  logr_time = end_time - start_time
83  print(f"LR time: {end_time-start_time}")
84
85  # Predict Using Model
86  logr_preds = logr.predict(X_Test)
87
88  # Evaluate
89  logr_accuracy = np.sum(np.where(y_test == logr_preds, 1, 0)) / y_test.shape[0]
90  print(f"LR Accuracy: {logr_accuracy}")
91
92
93  #
94  # Compare Models
95  #
96
97  #Plot Loss
98
99  # print(f"wh shape: {wh_loss_per_epoch.shape[0]}")
100 # print(f"svm shape: {svm_loss_per_epoch.shape[0]}")
101 # print(f"logr shape: {logr_loss_per_epoch.shape[0]}")
102
103 plt.figure()
104 plt.plot(wh_loss_per_epoch, label="Widrow-Hoff")
105 plt.plot(svm_loss_per_epoch, label="Linear SVM")
106 plt.plot(logr_loss_per_epoch, label="Log. Regression")
107 plt.yscale('log')
108 #plt.xscale('log')
```

```
plt.title('AVERAGE Loss Per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Compare Accuracy
plt.figure()
plt.bar(["Widrow Hoff", "Linear SVM", "Log. Regression"],
        [wh_accuracy, svm_accuracy, logr_accuracy],
        color=['blue', 'green', 'red'])
plt.title('Comparison of Accuracy Between Binary Classification Models')
plt.xlabel('Model')
plt.ylabel('Classification Accuracy')
plt.show()

# Compare Times
plt.figure()
plt.bar(["Widrow Hoff", "Linear SVM", "Log. Regression"],
        [wh_time, svm_time, logr_time],
        color=['blue', 'green', 'red'])
plt.title('Comparison of Run Time Between Binary Classification Models')
plt.xlabel('Model')
plt.ylabel('Time (s)')
plt.show()
## brute force test params
# new_param_test = False
# if new_param_test:
#     weights = np.random.uniform(-1, 1, X_train.shape[1])
#     format_str = "{:<20} {:<20} {:<20} {:<20}\n"
#     for c in np.linspace(0,10,20):
#         for a in np.linspace(0,1,20):
#             svm = Linear_SVM(weights = weights,
#                     learning_rate=a,
#                     C = c)
#
#             # Fit model
#             loss_per_epoch = svm.fit(X_train, y_train, max_epochs=1000)
#             preds = svm.predict(X_Test)
#             accuracy = np.sum(np.where(y_test == preds, 1, 0)) / y_test.shape[0]
#             loss = svm.loss(y_test, preds)
#             avg_loss = np.sum(loss) / y_test.shape[0]
#
#
#             with open('param_search.txt', 'a+') as f:
#                 f.write(format_str.format(c, a, accuracy, avg_loss))
```

Listing 4: `test_model_breast_cancer.py`

```
import numpy as np
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

class weston_watkins:
    def __init__(self, n_classes, n_features, learning_rate=.001, epochs=500):
        # weight matrix
        np.random.seed(25)
        self.W = np.random.uniform(-1, 1, size=(n_classes, n_features))
        # learning rate
```

```
        self.lr = learning_rate
        # number of epochs
        self.epochs = epochs

    # get loss for debugging
    def loss_function(self, X_i, y_i):
        correct_class_score = np.dot(self.W[y_i], X_i)
        loss = 0
        # loop over classes
        for r in range(self.W.shape[0]):
            if r != y_i:
                margin = np.dot(self.W[r], X_i) - correct_class_score + 1
                loss += max(0, margin)
        return loss

    def gradient(self, X_i, y_i):
        # get gradient wrt weights
        grad_W = np.zeros_like(self.W)
        correct_class_val = np.dot(self.W[y_i], X_i)

        # update for margin violation, to ensure correct class is only updated once
        margin_violation = False
        # loop over classes
        for r in range(self.W.shape[0]):
            if r != y_i:
                margin = np.dot(self.W[r], X_i) - correct_class_val + 1
                if margin > 0:
                    margin_violation = True
                    # incorrect class gradient
                    grad_W[r] += X_i
        if margin_violation:
            # correct class gradient
            grad_W[y_i] -= X_i
        return grad_W




    def fit(self, X, y):
        for epoch in range(self.epochs):
            total_loss = 0
            # loop over samples
            for i in range(len(y)):
                X_i = X[i]
                y_i = y[i]

                # get loss for debugging
                loss = self.loss_function(X_i, y_i)
                # get gradient
                grad_W = self.gradient(X_i, y_i)

                # update weights
                self.W -= self.lr * grad_W

                total_loss += loss

            print(f"Epoch {epoch + 1}, Loss: {total_loss / len(y)}")

    def predict(self, X):
        scores = np.dot(X, self.W.T)  # Compute scores for each class
        # use class with highest score for prediction
        return np.argmax(scores, axis=1)

wine = load_wine()
```

```
X, y = wine.data, wine.target

scaler = StandardScaler()
X = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=25)

num_classes = len(set(y_train))
num_features = X_train.shape[1]

model = weston_watkins(n_classes=num_classes, n_features=num_features, learning_rate=.001, epochs
    =500)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

wine_acc = accuracy_score(y_test, y_pred)
print(f"Wine dataset accuracy: {wine_acc:.6f}")

# Generate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix using seaborn
plt.figure(figsize=(8,6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=[f"Class {i}" for i in range(
    num_classes)], yticklabels=[f"Class {i}" for i in range(num_classes)])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
```

Listing 5: `weston_watkins.py`