Question 1

a

b

In these regression schemes, w_0 acts as the y-intercept of the model and w_1 acts as the gradient. Therefore we would expect regression scheme (i) to have a larger gradient than (ii) since there is a smaller penalty on w_1 ($\lambda = 1$ vs $\lambda = 10$ respectively). Regression schemes (iii) and (iv) would have a smaller y-intercept than (i) and (ii) since w_0 is included in their penalties. Scheme (iv) would have a smaller gradient and y-intercept than (iii) since its penalty hyperparameter is larger ($\lambda = 10$).

From the above reasoning:

- **(i) matches (a)**: (i) has the largest gradient
- **(ii) matches (b)**: a smaller gradient than (i) but the highest w_0 since this is not in the regression's penalty term
- **(iii) matches (c)**: a relatively small gradient y-intercept
- **(iv) matches (d)**: (iv) has a large penalty hyperparameter so the model would be encouraged to have a small gradient and y-intercept. (d) has the smallest y-intercept and a relatively small gradient

c

i

The expected error when testing on the training data and $k = 1$ is 0. Assuming no contradictions in the data, then because the testing data set is the same as the training data set and $k = 1$, there is a classification region for each data point which will correctly classify the original training set.

ii

There are 10,000 data points in total. 50% of these are positive classes and 50% are negative classes. In leave-one-out cross validation on 1NN, the data point being considered will be correctly classified if its nearest neighbour (but not the data point itself, since we are leaving this out) matches its classification. Even though the distribution classifications varies depending on the region of the dataset (e.g. the left rectangle has 25% +ve and 75% -ve), each rectangle has the same number of data points so if you are choosing any data point randomly in either rectangular region, then the probability of that the data point's nearest neighbour matches its classification is 50%. Therefore the expected error is 5,000.

iii

Training a 1NN model on the left rectangle of the training data will result in 25% of the rectangle being a region that classifies a point as +ve class and 75% of the rectangle being a region that classifies a point as a -ve class. The left rectangle of the testing data is entirely -ve classes and since the data points in the test data are uniformly randomly, each data point has a 75% chance of landing in a region that classifies the point as negative and therefore correctly. The same is true for the right rectangle region. So overall, a 1NN model trained on the entire training dataset is expected to correctly classify 75% of the test data giving it an expected error of 2,500.

iv

Looking at the left rectangle of the training data and using 21-NN, you would expect the model to classify the entire region as -ve classes. This is because for each point in the left rectangle, of the 21 nearest neighbours, you would expect 25% to be +ve and 75% to be -ve so overall the region is classified as negative. When the training data is used as test data, we know that the 1,250 points that are +ve will be incorrectly classified as -ve and therefore the expected error is 1,250. This is the same for the right rectangle however the classifications are inverted. Overall, you would expect a training set error of 2,500 using a 21-NN classifier model.

d

i

The space of X is 2^p. The space of Y is 2. Therefore the size of the hypothesis class is 2^{2^p}.

ii

The probability that the version space is not $\epsilon-exhausted$ after n training examples is at most $|H|e^{-\epsilon n}$. The number of samples needed to ensure that the sample space has a 0.9 probability (P) of being $\epsilon-exhausted$ is:

$$n = \frac{1}{\epsilon}(ln|H| + ln(\frac{1}{P}))$$

Substitute in the values to yield:

$$n = \frac{1}{0.1}(ln(2^{2^{10}}) + ln(\frac{1}{0.9}))$$

$$n = 10 * (1024 * ln(2) + ln(\frac{10}{9}))$$

$$n = 7,098.88$$

Therefore at least 7,099 data points are needed to ensure the version space is $\epsilon - exhausted$ with 0.9 probability.

Question 2

a

d

The statement is falsifiable since SVMs do have a way to handle non-linearly separable data. The first way is by creating a "soft margin" to accommodate for some errors. The second is by projecting the dataset into higher dimensions such that it is linearly separable. Although the projection to a higher dimension comes at a computational cost, it does not affect the performance of the SVM. This is how SVMs can handle non-linearly separable data and why the statement is incorrect.

e

As mentioned in (d), non-linearly separable data can be projected into higher dimensions such that it is linearly separable. Working in these higher dimensional spaces can come at a computational cost that is sometimes impractical. The **kernel trick** is a method of dealing with this problem by representing the data as a mapping from an input vector to the dot product of the vectors in the higher dimensional space. The "trick" is that this method doesn't explicitly apply the expensive transformation into a higher dimensional space. This is how impractical projections into higher dimensional space can become practical, which is important for SVM and other ML methods that require linearly separable data and is why the kernel trick is such an important technique in machine learning.

f

After the forgotten data point is added back in, the dataset it still linearly separable. Support vectors are instances that are closest to the maximum margin hyperplane. Let's assume that this is a binary classification. Every single data point could be a support vector if they all lie exactly on the maximum margin hyperplane. This means the maximum number of support vectors in this dataset is 51. Conversely, the minimum number of support vectors needed is 2 since, in a hard-margined SVM, there will always be a hyperplane with 2 decision boundaries and each of these boundaries will have an associated closest instance (support vector).

Question 3

a

Overview

Online machine learning methods are ones that are exposed to new data over time and adjust the model as this happens. The paper looks at online learning methods and how they compare to more traditional batch learners. Typically, online learning methods are faster and have smaller memory footprints than batch learners. The trade-off is that online learners can be less accurate than batch learners on one pass over the data, and so often require multiple passes to achieve a similar accuracy.

Online Learning Methods

The fact that online learning methods are faster and more memory efficient means that they are more scalable. They can also easily be converted into a batch algorithm.

The general format of a mistake-driven online learning algorithm is to receive a new binary observation $X_i \in \mathbb{R}^j$ (i.e with j features) with some corresponding real value $y_i \in \{-1, 1\}$. A prediction is then made using the current model and X_i and compared with its actual value. If a mistake is made on the observation, then the model is updated.

The paper looks at existing mistake-driven online learning algorithms such as the Perceptron, Relaxed Online Maximum Margin Algorithm (ROMMA) and Passive-Aggressive and introduces their algorithm which is the basis for the Winnow algorithms. The paper then looks at 3 algorithms based off of the Winnow concept which are Positive Winnow, Balanced Winnow and Modified Balanced Winnow.

For each of the Winnow algorithms, there is an assumption that each observation x_t is a vector of positive weights which is usually satisfied in NLP tasks where the x_t^j is related to the frequency of a term. Incoming data is augmented such that the $(m+1)^{th}$ feature is set to 1. The data is also normalised across all features if that feature occurs in the current model (w_i).

Each Winnow algorithm has a promotion parameter $\alpha > 1$, a demotion parameter $0 < \beta < 1$ and a threshold parameter $\theta_{th} > 0$. The Positive Winnow algorithm follows very similarly to the Perceptron however its decision function (classification) is given by $f = sign(\langle x_t, w_i \rangle - \theta_{th})$. Additionally, the update rule on vector w_t is different: for all j where $x_t^j > 0$

$$w_{i+1}^j = \begin{cases} w_i^j * \alpha & y_t > 0 \\ w_i^j * \beta & y_t < 0 \end{cases}$$

The Balanced Winnow is a further modification on the Positive Winnow. The w_t vector is now a combination of a positive model u_t and a negative model v_t where each vector is initialised with all positive values and negative values respectively. The decision function then becomes $f = sign(\langle x_t, u_i \rangle - \langle x_t, v_i \rangle - \theta_{th})$ and the update rule for both vectors is:

For all j in x_t: $u_{i+1}^j = \begin{cases} u_i^j * \alpha & y_t > 0 \\ u_i^j * \beta & y_t < 0 \end{cases}$ and $v_{i+1}^j = \begin{cases} u_i^j * \beta & y_t > 0 \\ u_i^j * \alpha & y_t < 0 \end{cases}$

The paper then proposes a new *Modified Balanced Winnow* algorithm which builds on the Balanced Winnow algorithm bu including a *margin* $M \geq 0$. A prediction is considered a mistake not online when y_t is different from $\hat{y}_t$, but also when the score function multiplied by y_t is smaller than the margin. As an equation, this is $y_t * (\langle x_t, u_i \rangle - \langle x_t, v_i \rangle - \theta_{th}) \leq M$. There is also a small change to the update rule which can be referred to in the paper.

Voting

This section of the paper talks about averaging or *voting* to decide which hypothesis to use so far. In mistake-driven learning, each time a new piece of data arrives that the model predicts incorrectly, the model is updated and the latest hypothesis is used. With voting, rather than using the latest hypothesis, a hypothesis is yielded by averaging intermediary hypotheses which are weighted by the number of correct predictions each hypothesis made in its learning process. The motivation for this is that a voted hypothesis is less likely to be over-fit to the data.

Mathematically, the average model w_a is denoted as $w_a = \frac{1}{Z}\sum_i (w_i * c_i)$. A voted algorithm is prefixed with a *v-* for example v-ROMMA or v-Winnow means the voted versions of ROMMA and Winnow respectively.

Experiments

This section is an attempt to evaluate the algorithms discussed in the *Online Learning Methods* section. The algorithms selected were Passive-Aggressive, Positive Winnow, Balanced Winnow and Modified Balanced Winnow. Batch algorithms such as the linear SVM and Naive Bayes were also selected to be compared to. The hyper-parameters used in each of the models can be found in Table 3 of the paper.

Various datasets were chosen. Some of the datasets were NLP based (came from a text-based input and tokenised) and others were non-NLP datasets. The feature count and length of each dataset can be found in section 4.1 of the paper.

For evaluation, the F1 score or harmonic precision-recall mean was used as a metric for each model. This is a ratio of the *precision* of the model and the *recall* and is mathematically denoted as $\frac{2*Precision*Recall}{Recall+Precision}$. Each of the above models were trained on each the datasets with 5-fold cross-validation and the F1 mean was recorded. A voted version for each of the models was also trained on the datasets. The results are included in Tables 4 and 5 of the paper.

The key findings from the results was that across all 8 NLP datasets, MBW had the highest median F1 mean even when compared with the batch learners SVM and Naive Bayes. On non-NLP datasets however, MBW had the lowest median F1 mean. The effect of voting on the online learners only seemed to improve their performance with non-NLP datasets and had little effect on NLP datasets.

Online Feature Selection

Feature selection is when there is a dataset with a large number of features and rather then training on all of the features, only a subset of "important" features are selected. This improves training and testing times, reduces memory requirements and in some cases can improve prediction accuracy.

Feature selection in batch methods is a well-known task. Each feature is ranked in terms of its "strength" of being a predictor (Gain or Chi-Square) and the most important features are selected. With online learning methods, this selection can be expensive as the strength of each feature needs to be re-calculated with each new example.

The paper proposes a method of feature selection for its Modified Balanced Winnow algorithm that it calls *Extremal Feature Selection* (EFS). This uses the two weight vectors u_t^j and v_t^j for feature j and calculates its importance score Iat iteration t with:

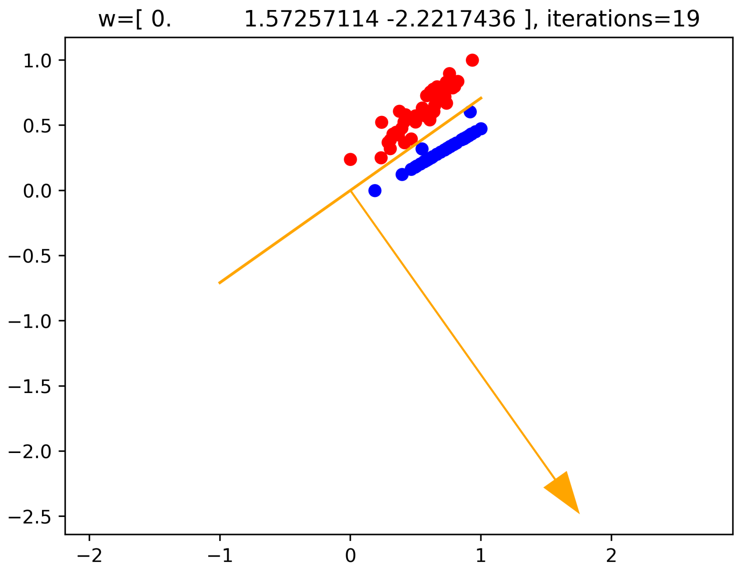
$$I_t^j = |u_t^j - v_t^j|$$

Higher values of I are associated with important features and low values of I with unimportant features. In fact, in the *20newsgroup* dataset, the features with low importance scores were stop words such as "as, you, what, they, are, the".

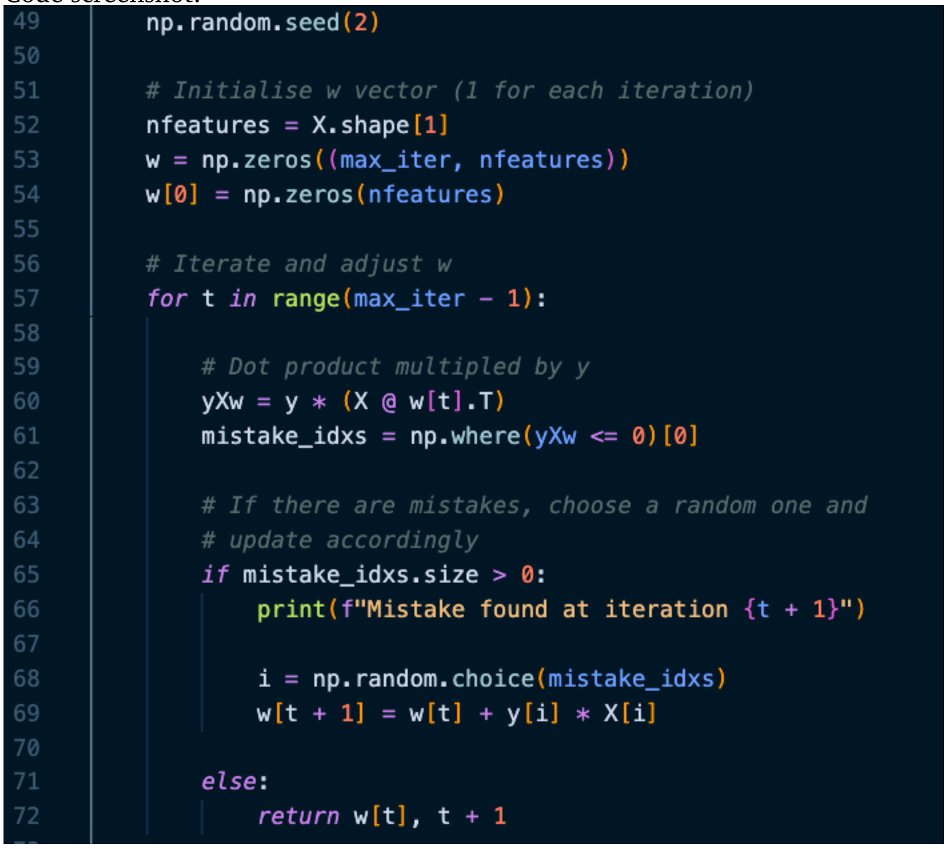
Preliminary experiments showed however that not only do the most important features play a role when improving accuracy, but so does the least important features. For example, when choosing 100 features from a dataset, EFS would select 90% of the features from the extreme top T and 10% of the features from the extreme bottom B. Such a model would out-perform other models that selected 100% of its feature from T specifically when the number of selected features was relatively large. This result is speculated to be explained by the smoothing-like effect in the MBW learner.

b

Plot of final converged w vector amongst the dataset



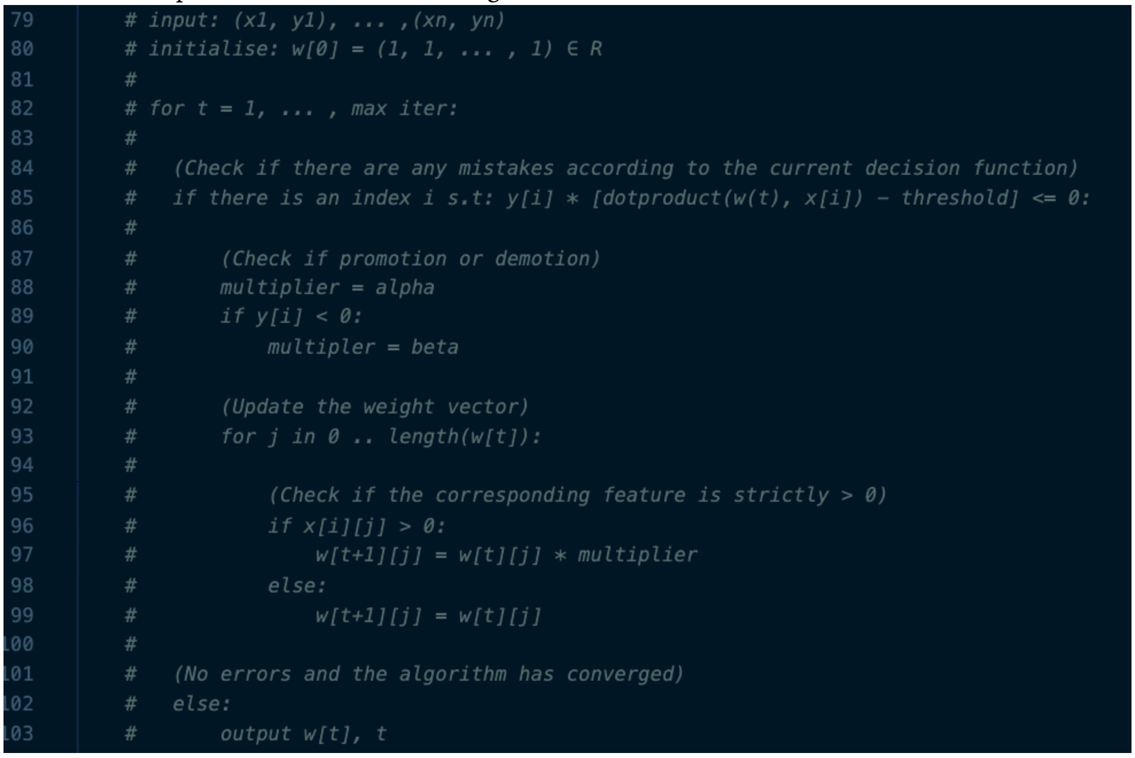
Code screenshot:



```
    np.random.seed(2)

    # Initialise w vector (1 for each iteration)
    nfeatures = X.shape[1]
    w = np.zeros((max_iter, nfeatures))
    w[0] = np.zeros(nfeatures)

    # Iterate and adjust w
    for t in range(max_iter - 1):

        # Dot product multipled by y
        yXw = y * (X @ w[t].T)
        mistake_idxs = np.where(yXw <= 0)[0]

        # If there are mistakes, choose a random one and
        # update accordingly
        if mistake_idxs.size > 0:
            print(f"Mistake found at iteration {t + 1}")

            i = np.random.choice(mistake_idxs)
            w[t + 1] = w[t] + y[i] * X[i]

        else:
            return w[t], t + 1
```

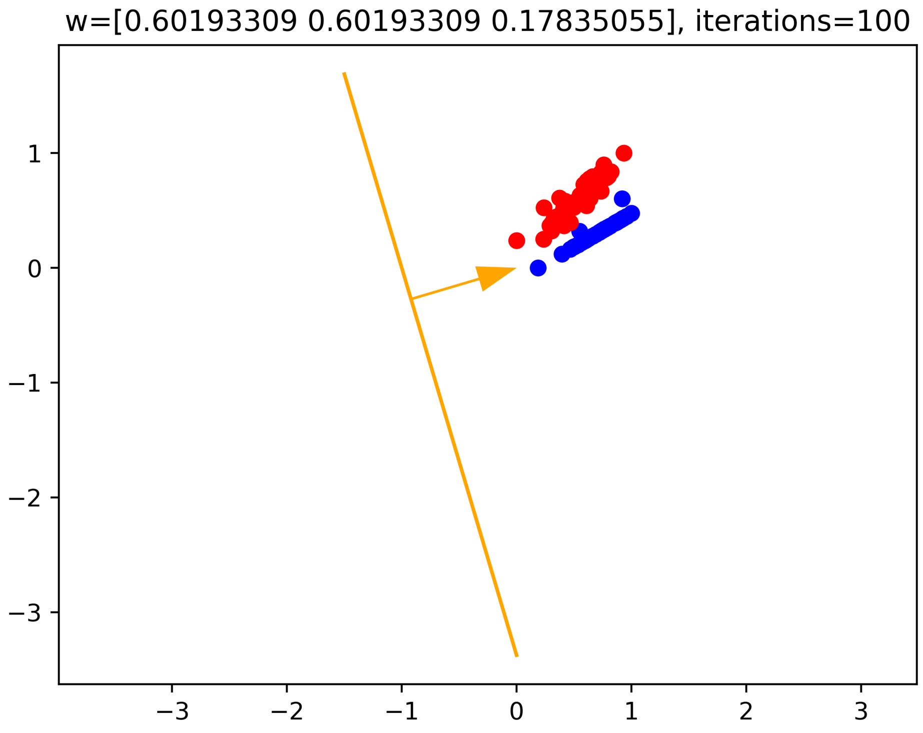
c

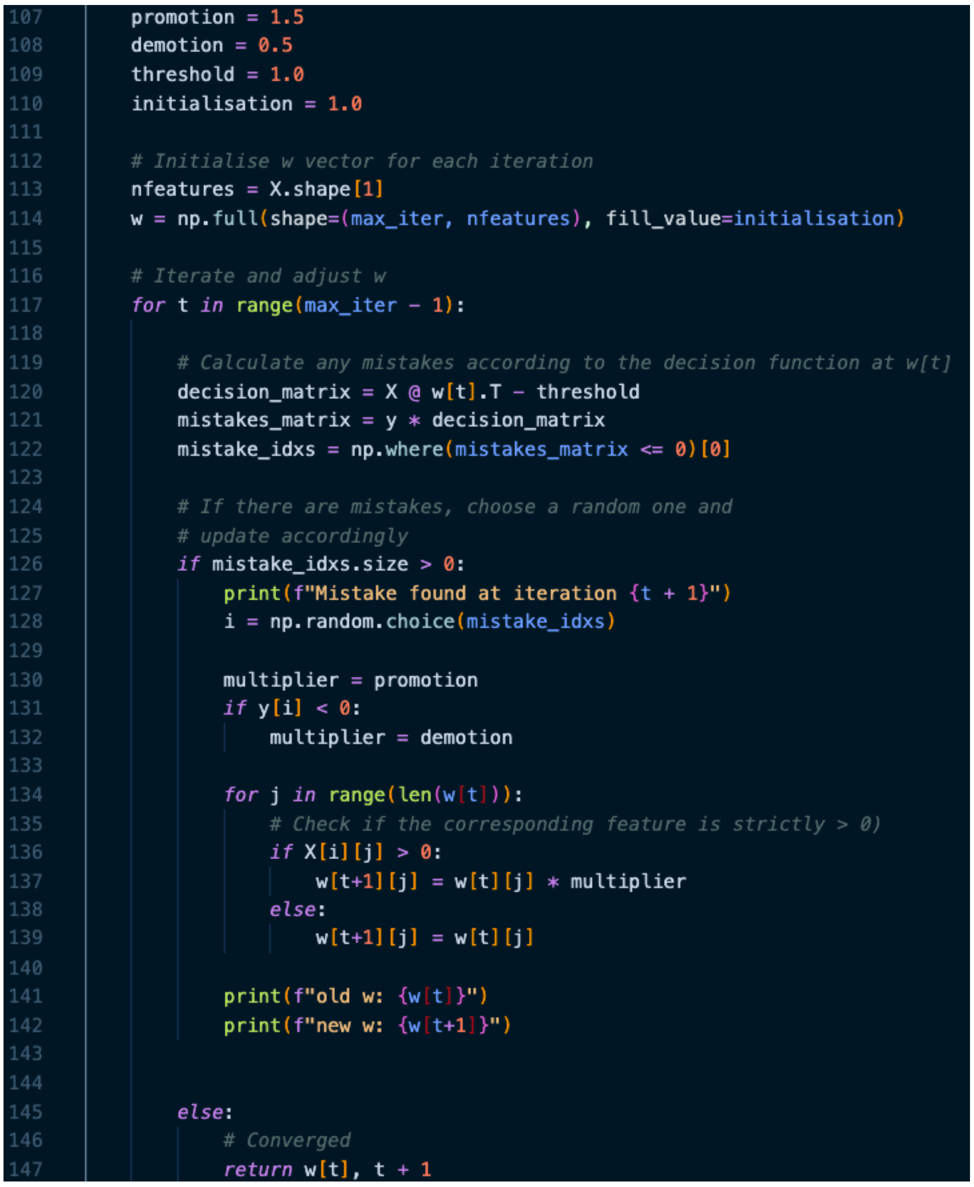
Pseudo code to implement *Positive Winnow* algorithm:



```
# input: (x1, y1), ... ,(xn, yn)
# initialise: w[0] = (1, 1, ... , 1) ∈ R
#
# for t = 1, ... , max iter:
#
#   (Check if there are any mistakes according to the current decision function)
#   if there is an index i s.t: y[i] * [dotproduct(w(t), x[i]) - threshold] <= 0:
#
#       (Check if promotion or demotion)
#       multiplier = alpha
#       if y[i] < 0:
#           multipler = beta
#
#       (Update the weight vector)
#       for j in 0 .. length(w[t]):
#
#           (Check if the corresponding feature is strictly > 0)
#           if x[i][j] > 0:
#               w[t+1][j] = w[t][j] * multiplier
#           else:
#               w[t+1][j] = w[t][j]
#
#   (No errors and the algorithm has converged)
#   else:
#       output w[t], t
```

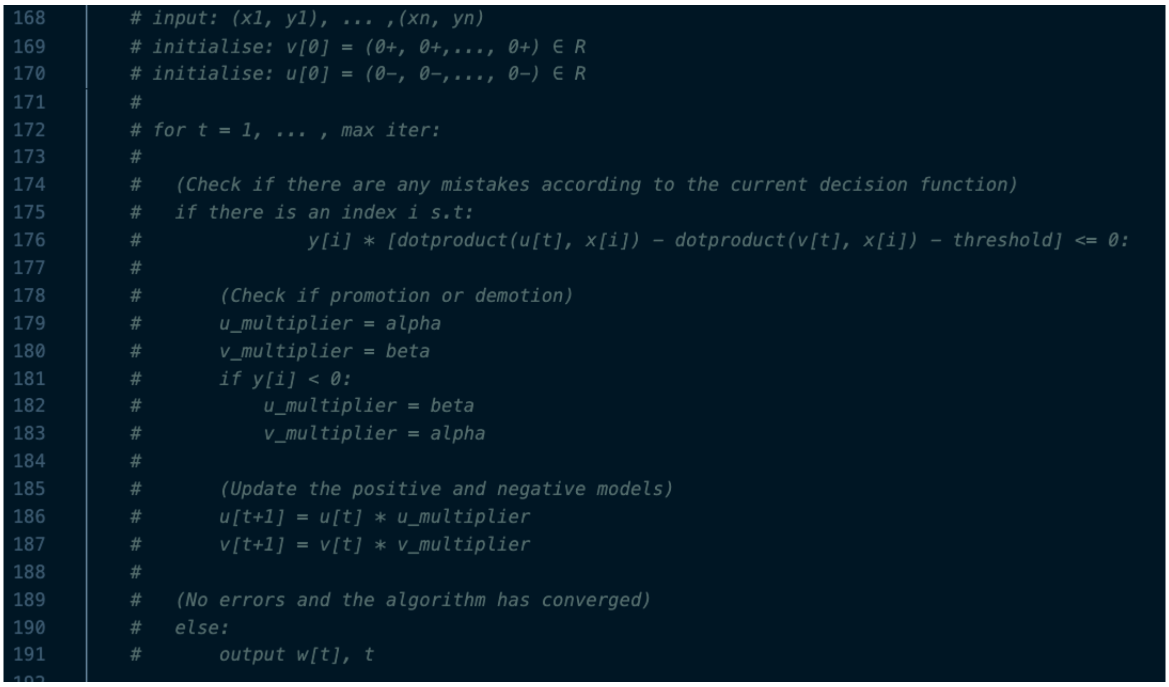
Note that I am using t as the w vector iteration number and i as the observation that the mistake occurred at. This is the other way around in the provided paper.



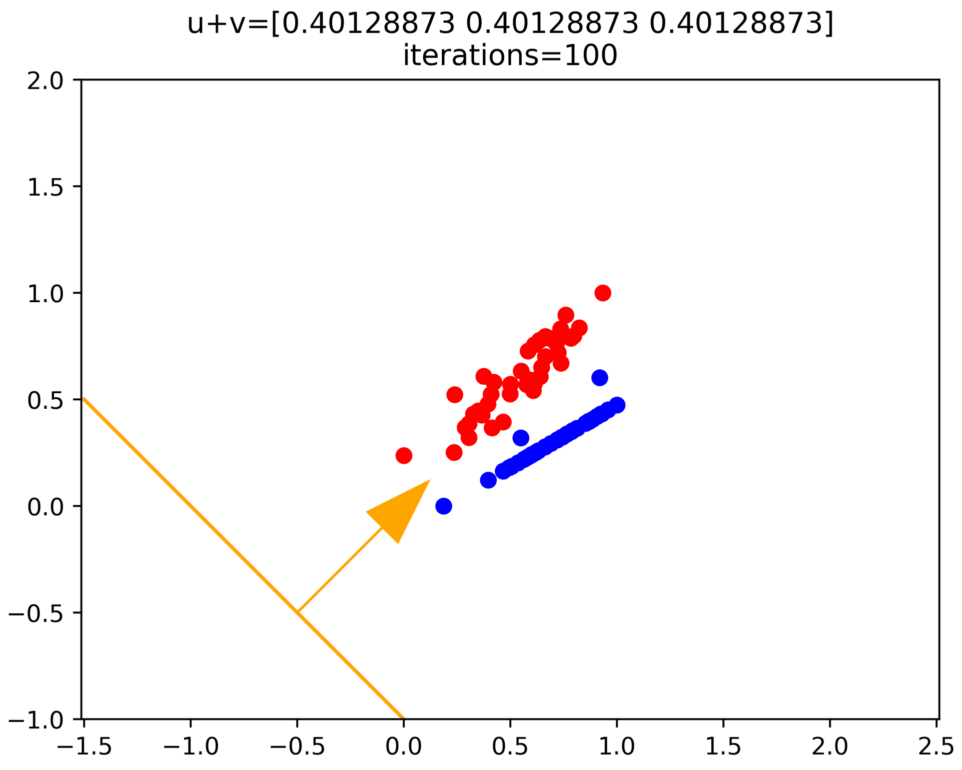


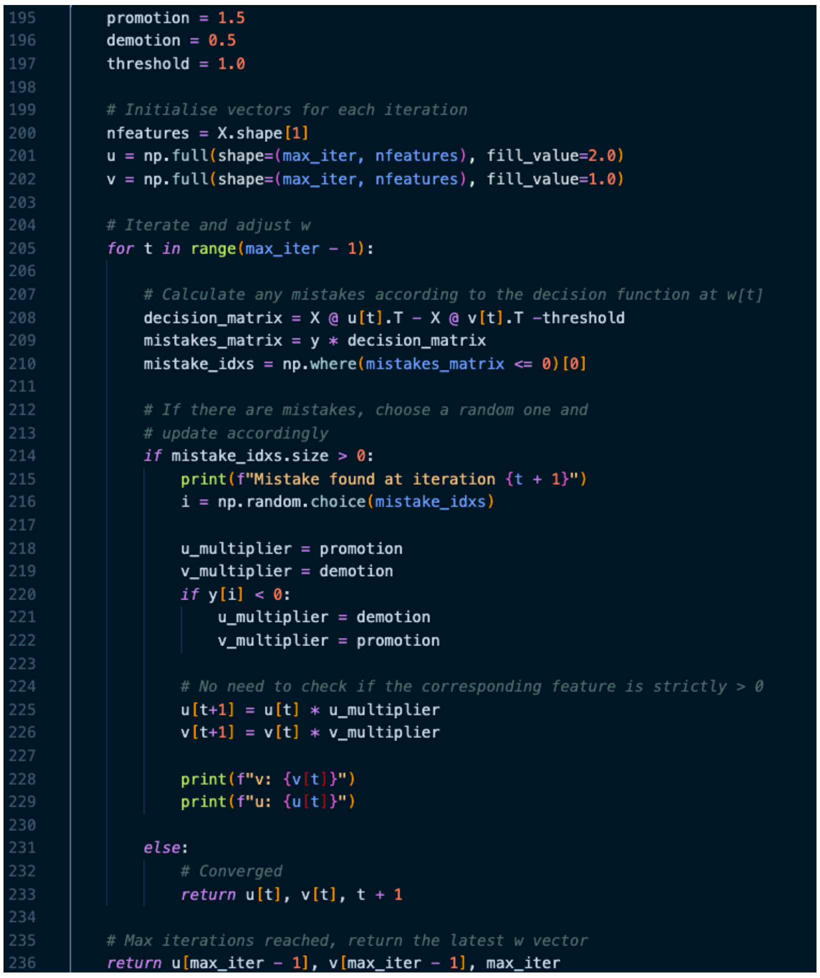
```python
    promotion = 1.5
    demotion = 0.5
    threshold = 1.0
    initialisation = 1.0

    # Initialise w vector for each iteration
    nfeatures = X.shape[1]
    w = np.full(shape=(max_iter, nfeatures), fill_value=initialisation)

    # Iterate and adjust w
    for t in range(max_iter - 1):

        # Calculate any mistakes according to the decision function at w[t]
        decision_matrix = X @ w[t].T - threshold
        mistakes_matrix = y * decision_matrix
        mistake_idxs = np.where(mistakes_matrix <= 0)[0]

        # If there are mistakes, choose a random one and
        # update accordingly
        if mistake_idxs.size > 0:
            print(f"Mistake found at iteration {t + 1}")
            i = np.random.choice(mistake_idxs)

            multiplier = promotion
            if y[i] < 0:
                multiplier = demotion

            for j in range(len(w[t])):
                # Check if the corresponding feature is strictly > 0)
                if X[i][j] > 0:
                    w[t+1][j] = w[t][j] * multiplier
                else:
                    w[t+1][j] = w[t][j]

            print(f"old w: {w[t]}")
            print(f"new w: {w[t+1]}")


        else:
            # Converged
            return w[t], t + 1
```

d



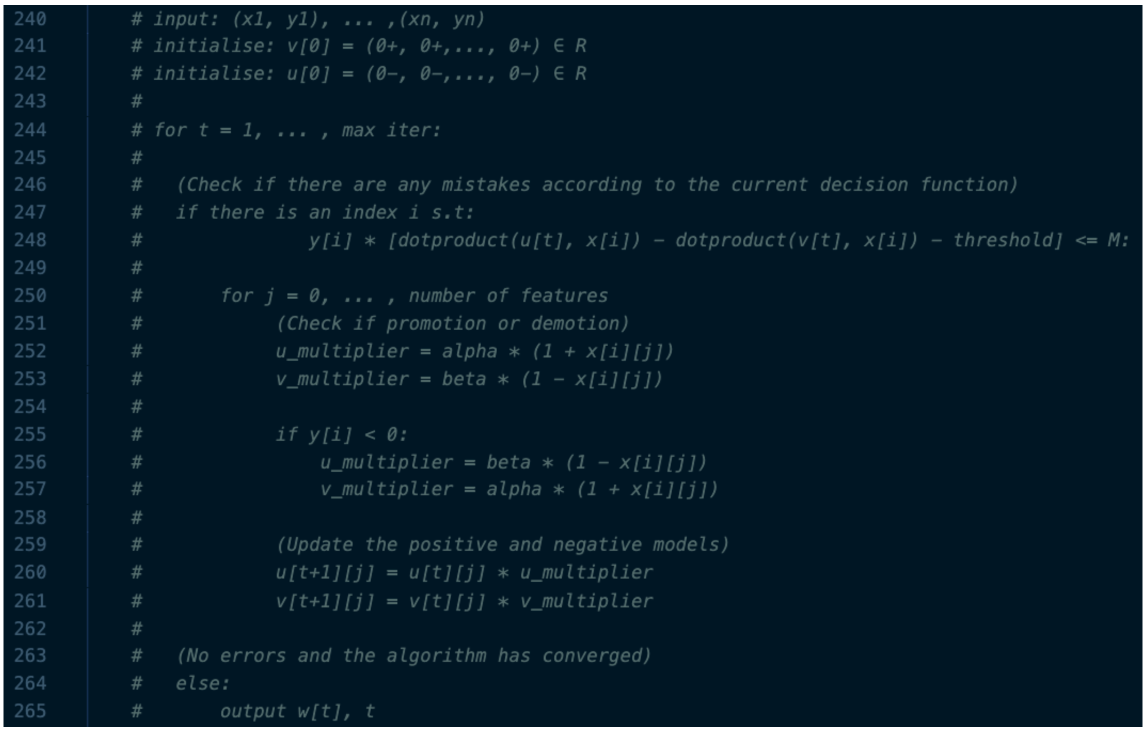
```
# input: (x1, y1), ... ,(xn, yn)
# initialise: v[0] = (0+, 0+,..., 0+) ∈ R
# initialise: u[0] = (0−, 0−,..., 0−) ∈ R
#
# for t = 1, ... , max iter:
#
#   (Check if there are any mistakes according to the current decision function)
#   if there is an index i s.t:
#               y[i] ∗ [dotproduct(u[t], x[i]) − dotproduct(v[t], x[i]) − threshold] <= 0:
#
#       (Check if promotion or demotion)
#       u_multiplier = alpha
#       v_multiplier = beta
#       if y[i] < 0:
#           u_multiplier = beta
#           v_multiplier = alpha
#
#       (Update the positive and negative models)
#       u[t+1] = u[t] ∗ u_multiplier
#       v[t+1] = v[t] ∗ v_multiplier
#
#   (No errors and the algorithm has converged)
#   else:
#       output w[t], t
```



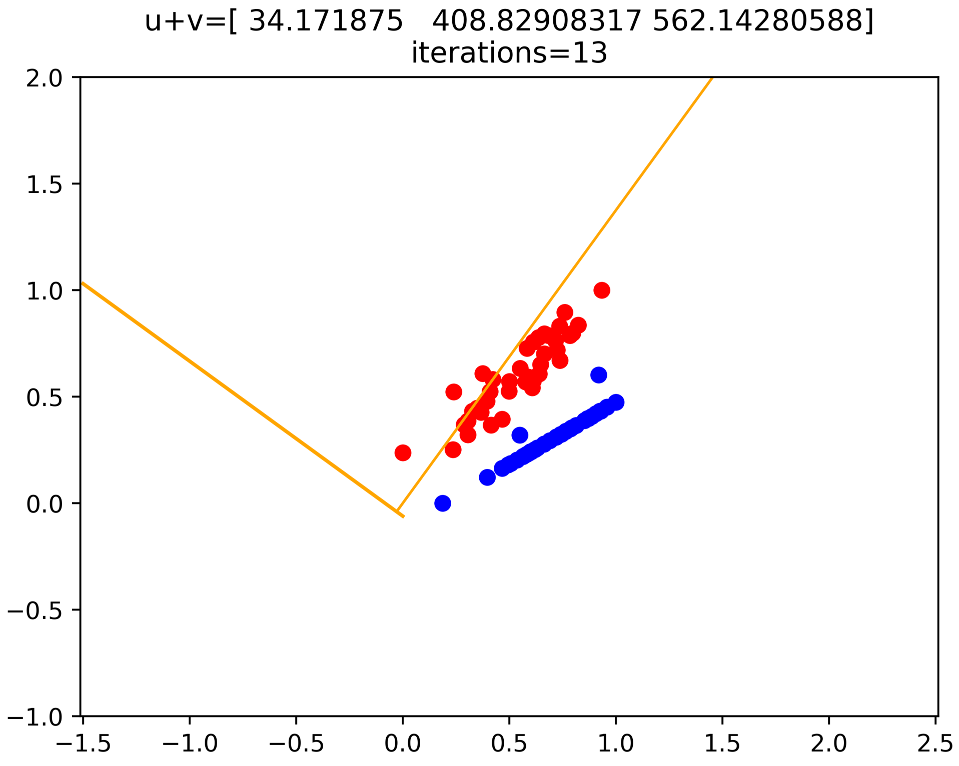


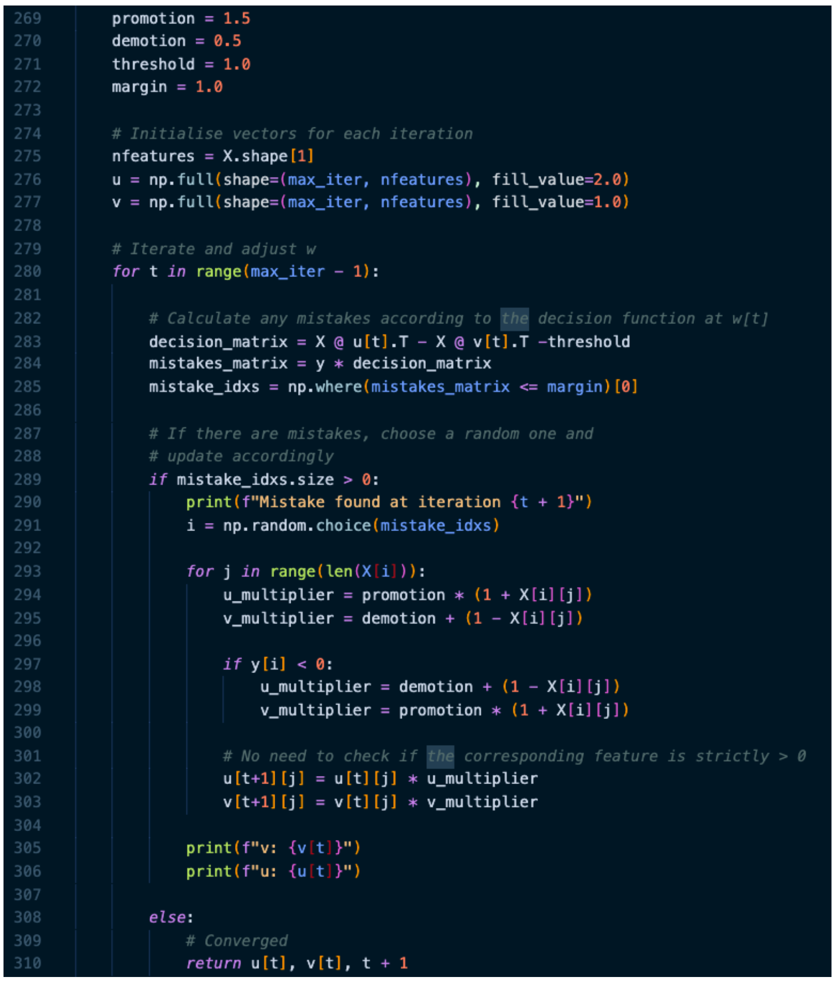
```python
    promotion = 1.5
    demotion = 0.5
    threshold = 1.0

    # Initialise vectors for each iteration
    nfeatures = X.shape[1]
    u = np.full(shape=(max_iter, nfeatures), fill_value=2.0)
    v = np.full(shape=(max_iter, nfeatures), fill_value=1.0)

    # Iterate and adjust w
    for t in range(max_iter - 1):

        # Calculate any mistakes according to the decision function at w[t]
        decision_matrix = X @ u[t].T - X @ v[t].T -threshold
        mistakes_matrix = y * decision_matrix
        mistake_idxs = np.where(mistakes_matrix <= 0)[0]

        # If there are mistakes, choose a random one and
        # update accordingly
        if mistake_idxs.size > 0:
            print(f"Mistake found at iteration {t + 1}")
            i = np.random.choice(mistake_idxs)

            u_multiplier = promotion
            v_multiplier = demotion
            if y[i] < 0:
                u_multiplier = demotion
                v_multiplier = promotion

            # No need to check if the corresponding feature is strictly > 0
            u[t+1] = u[t] * u_multiplier
            v[t+1] = v[t] * v_multiplier

            print(f"v: {v[t]}")
            print(f"u: {u[t]}")

        else:
            # Converged
            return u[t], v[t], t + 1

    # Max iterations reached, return the latest w vector
    return u[max_iter - 1], v[max_iter - 1], max_iter
```

e



```python
    # input: (x1, y1), ... ,(xn, yn)
    # initialise: v[0] = (0+, 0+,..., 0+) ∈ R
    # initialise: u[0] = (0−, 0−,..., 0−) ∈ R
    #
    # for t = 1, ... , max iter:
    #
    #   (Check if there are any mistakes according to the current decision function)
    #   if there is an index i s.t:
    #               y[i] * [dotproduct(u[t], x[i]) - dotproduct(v[t], x[i]) - threshold] <= M:
    #
    #       for j = 0, ... , number of features
    #           (Check if promotion or demotion)
    #           u_multiplier = alpha * (1 + x[i][j])
    #           v_multiplier = beta * (1 - x[i][j])
    #
    #           if y[i] < 0:
    #               u_multiplier = beta * (1 - x[i][j])
    #               v_multiplier = alpha * (1 + x[i][j])
    #
    #           (Update the positive and negative models)
    #           u[t+1][j] = u[t][j] * u_multiplier
    #           v[t+1][j] = v[t][j] * v_multiplier
    #
    #   (No errors and the algorithm has converged)
    #   else:
    #       output w[t], t
```





```python
    promotion = 1.5
    demotion = 0.5
    threshold = 1.0
    margin = 1.0

    # Initialise vectors for each iteration
    nfeatures = X.shape[1]
    u = np.full(shape=(max_iter, nfeatures), fill_value=2.0)
    v = np.full(shape=(max_iter, nfeatures), fill_value=1.0)

    # Iterate and adjust w
    for t in range(max_iter - 1):

        # Calculate any mistakes according to the decision function at w[t]
        decision_matrix = X @ u[t].T - X @ v[t].T -threshold
        mistakes_matrix = y * decision_matrix
        mistake_idxs = np.where(mistakes_matrix <= margin)[0]

        # If there are mistakes, choose a random one and
        # update accordingly
        if mistake_idxs.size > 0:
            print(f"Mistake found at iteration {t + 1}")
            i = np.random.choice(mistake_idxs)

            for j in range(len(X[i])):
                u_multiplier = promotion * (1 + X[i][j])
                v_multiplier = demotion + (1 - X[i][j])

                if y[i] < 0:
                    u_multiplier = demotion + (1 - X[i][j])
                    v_multiplier = promotion * (1 + X[i][j])

                # No need to check if the corresponding feature is strictly > 0
                u[t+1][j] = u[t][j] * u_multiplier
                v[t+1][j] = v[t][j] * v_multiplier

            print(f"v: {v[t]}")
            print(f"u: {u[t]}")

        else:
            # Converged
            return u[t], v[t], t + 1
```

The balanced and modified Winnow algorithms have 2 main differences. The first is how the algorithms consider a mistake. The Balanced Winnow will consider an incoming data point mistake if the decision function predicts it incorrectly ($y[t] * sign(\langle x_t, u_i \rangle - \langle x_t, v_i \rangle - \theta_{th}) \leq 0$). The Modified Winnow considers a data point a mistake if the decision function predicts the incoming data point *within a margin*. ($y[t] * sign(\langle x_t, u_i \rangle - \langle x_t, v_i \rangle - \theta_{th}) \leq M$) The second is how the vectors update when a mistake occurs. In the Modified Winnow, the multiplicative correction depends on the particular feature weight of the incoming example (x_t^j), rather than correcting by a constant (α or β). This makes the corrections in the Modified Winnow more aggressive.