STATS M231A Final Homework

Laxman Dahal

Abstract

The final homework consisted of two parts: written and coding. The written part had five proof-based questions on different topics covered in the class while the coding part was focused on two of the most popular machine learning algorithms, XGBoost and SVM. The two algorithms are implemented on the data provided by the TA to detect spam email. It was found that the the XGBoost performed well as compared to SVM (linear or gaussian). The testing accuracy for the XGB algorithm with default setting was 95.87% as compared to 95.11% when the hyper-parameters (n_estimators and max_depth) are tuned. When SVM was implemented with default setting, the testing accuracy for the default setting was 86.43% and 66.23% respectively for linear and gaussian kernel. When the hyperparameters (c, and max_iter) are tuned, the testing accuracy improve to 91.75% and 83.06%, respectively. It indicates that, hyperparameter tuning has much greater impact for SVM than XGB. Additionally, SVM was also implemented on a synthetic data from scratch.

1 Written Part

1.1 Problem 1

For VAE, prove

$$\log p_\theta(x) - D_{\mathrm{KL}}(q_\phi(z|x)|p_\theta(z|x)) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x)|p(z)).$$

With reparametrization $z = \mu_\phi(x) + \sigma_\phi(x) \odot e$, with $e \sim \mathrm{N}(0,I)$, explain the right hand side is tractable.

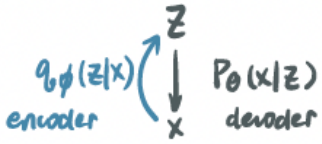

Figure 1: Illustration of encoder and decoder model

As shown in Figure 1, the decoder model is $z \sim p(z) \sim \mathrm{N}(0, I_d)$, and $[x|z] \sim p_\theta(x|z) \sim \mathrm{N}(g_\theta(z), \sigma^2 I)$. It defines a joint distribution $p_\theta(z,x) = p(z)p_\theta(x|z)$. The encoder model is $[z|x] \sim q_\phi(z|x) \sim \mathrm{N}(\mu_\phi(x), V_\phi(x))$, where V is a diagonal matrix. Together with $p_{\mathrm{data}}(x)$, we have a joint distribution $q_\phi(z,x) = p_{\mathrm{data}}(x)q_\phi(z|x)$. Lets looks at the probabilistic formulations first:

1. **Joint:** $p_\theta(x,z) = p(z)p_\theta(x|z)$. It is explicit or in closed-form
2. **Marginal:** $p_\theta(x) = \int p_\theta(x|z)p(z)\mathrm{dz}$. It is implicit or intractable integral
3. **Posterior:** $p_\theta(z|x) = \frac{p_\theta(x,z)}{p_\theta(x)} \propto p(z)p(x|z)$

4. **MLE:** $L(\theta) = \frac{1}{n}\sum_{i=1}^{n} \log p_\theta(x_i) = E_{p_{\text{data}}}[\log p_\theta(x)]$. Here, $\log p_\theta(x)$ is intractable.
5. **KL Divergence:** $D_{\text{KL}}(p_{\text{data}}|p_\theta) = \mathbb{E}_{p_{\text{data}}}[\log p_{\text{data}}(x)] - \mathbb{E}_{p_{\text{data}}}[\log p_\theta(x)]$

For Variational AutoEncoder (VAE), the evidence lower bound (ELBO) is given by:

$$L(\theta,\phi) = \log p_\theta(x) - D_{\text{KL}}(q_\phi(z|x)|p_\theta(z|x))$$

Lets introduce re-parametrization in the form of encoder as follows: $z \sim \text{N}(\mu_p hi(x)V_\phi(x))$ which then becomes $z = \mu_\phi(x) + \sigma_\phi(x) \odot e$, with $e \sim \text{N}(0,I)$. With this, the ELBO becomes:

$$L(\theta,\phi) = \log p_\theta(x) - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)] + \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(z|x)]$$

Using re-parameterization trick,

$$L(\theta,\phi) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x,z)] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)]$$

where $\log p_\theta(x,z)$ is a complete-data model which can be equivalently also written as

$$L(\theta,\phi) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x) + \log p_\theta(z|x)] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)]$$

re-organizing the terms, we get,

$$L(\theta,\phi) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(z|x)] - \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x) - \log p_\theta(x)]$$

which is the same as

$$L(\theta,\phi) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\text{KL}}(q_\phi(z|x)|p(z))$$

The right-hand side is tractable as long as the encoder or the inference model $q_\phi(z|x)$ is explicit or in closed form.

1.2 Problem 2

For GAN, explain that the minimax value function is

$$V(G,D) = \mathbb{E}_{p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{\tilde{z}\sim\text{N}(0,I)}[\log(1 - D(G(\tilde{z}))].$$

Let $p_\theta(x)$ be the density of the generator model. Assuming optimal D

$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_\theta(x)}.$$

Explain that

$$V = D_{\text{KL}}(p_{\text{data}}|p_{\text{mix}}) + D_{\text{KL}}(p_\theta|p_{\text{mix}}) = \text{JSD}(p_{\text{data}}, p_\theta),$$

the Jensen-Shannon distance, where $p_{\text{mix}}(x) = (p_{\text{data}}(x) + p_\theta(x))/2$.

Lets assume z as a latent vector with a known prior distribution $z \sim p(z)$. The idea of treating z as a latent vector is to learn the decoder model in an unsupervised manner so that we can use it to define a generative model (G). The model is then learned by GAN where the generator model, G is paired with a discriminator model, D, where for an image x, $D(x)$ is the probability that x is a true image instead of a generated image.

We can train the pair of (G,D) by an adversarial, zero-sum game. Specifically, let $G(z) = g_\theta(z)$ be a generator. Let

$$V(D,G) = \mathbb{E}_{p_{\text{data}}}[\log D(X)] + \mathbb{E}_{h\sim p(z)}[\log(1 - D(G(z))],$$

where $\mathbb{E}_{p_{\text{data}}}$ can be approximated by averaging over the observed examples, and $\mathbb{E}_z$ is usually approximated by Monte Carlo simulation method. The Monte Carlo sample are typically averaged over the fake examples generated by the generator part of the model.

Thus, we learn D and G by $\min_G \max_D V(D,G)$. $V(D,G)$ is the log-likelihood for D, i.e., the log-probability of the real and fake examples. However, $V(D,G)$ is not a very convincing objective for G. In practice, the training of G is usually modified into maximizing $\mathbb{E}_{h\sim p(z)}[\log D(G(z))]$ to avoid the vanishing gradient problem.

For a given θ, let p_θ be the distribution of $g_\theta(z)$ with $z \sim p(z)$. Assuming a perfect discriminator according to the Bayes rule $D(x) = p_{\text{data}}(x)/(p_{\text{data}}(x) + p_\theta(x))$ (assuming equal numbers of real and fake examples). Then θ minimizes Jensen-Shannon

$$\text{JSD}(p_{\text{data}}|p_\theta) = \text{KL}(p_\theta|p_{\text{mix}}) + \text{KL}(p_{\text{data}}|p_{\text{mix}}),$$

where $p_{\text{mix}} = (p_{\text{data}} + p_\theta)/2$.

This result implies that GAN has mode collapsing behavior.

1.3 Problem 3

Let $\pi_\theta(a|s)$ be the policy. Let $z(a)$ be the expected total reward. Suppose we want to maximize

$$J(\theta) = \mathbb{E}_{\pi_\theta(a|s)}[z(a)]$$

Prove

$$J'(\theta) = \mathbb{E}_{\pi_\theta(a|s)}\left[z(a)\frac{\partial}{\partial\theta}\log\pi_\theta(a|s)\right]$$

Explain the difference between policy gradient reinforcement learning and gradient ascent for maximum likelihood supervised learning.

For a policy network $\pi_\theta(a|s)$, τ is the trajectory obtained by playing out the policy.The distribution of τ is given by

$$p_\theta(\tau) = \prod_t \pi(a_t|s_t)p(s_{t+1}|s_t,a_t)$$

We want to find θ to maximize $J(\theta) = \mathbb{E}_{\pi_\theta(a|s)}[z(a)]$.

The gradient is

$$\begin{aligned} J'(\theta) = \frac{\partial}{\partial\theta}\mathbb{E}_{\pi_\theta(a|s)}[z(a)] &= \int z(a)\frac{\partial}{\partial\theta}\pi_\theta(a|s)da \\ &= \int\left[z(a)\frac{\partial}{\partial\theta}\log\pi_\theta(a|s)\right]\pi_\theta(a|s)da \\ &= \mathbb{E}_{p_\theta(a|s)}\left[z(a)\frac{\partial}{\partial\theta}\log\pi_\theta(a|s)\right] \end{aligned}$$

The stochastic gradient (or empirical gradient) is to sample τ by playing out the policy π_θ, and update θ by

$$\Delta\theta \propto z(a)\frac{\partial}{\partial\theta}\log\pi_\theta(a|s).$$

The difference between policy gradient reinforcement learning and gradient ascent for MLE supervised learning are as follows:

1. In reinforcement learning, the trajectory obtained by playing out the policy, τ is generated by the self policy. In MLE, τ is generated by the expert.
2. In reinforcement learning, the gradient consists of there is $z(a)$ in the gradient. In MLE, there is no $z(a)$ term.

1.4 Problem 4

For extreme gradient boosting (XGboost) for logistic regression, explain that we can grow a new tree $h_m(x)$ by minimizing a weighted least squares loss

$$L = \sum_{i=1}^{n} w_i\left(\frac{r_i}{w_i} - h_m(x_i)\right)^2,$$

where $r_i = y_i - \hat{p}_i$, and $w_i = \hat{p}_i(1-\hat{p}_i)$, with $\hat{p}_i$ being the predicted probability $p(y_i = 1|x_i)$ based on the existing trees. Explain the regularizations to be added to L.

We want to learn a function

$$f(x) = \sum_{k=1}^{d} h_k(x)$$

for the purpose of regression or classification, where $h_k(x)$ are base functions that are not necessarily classifiers, i.e., $h_k(x)$ may return continuous values.

Let $F_{m-1}(x) = \sum_{k=1}^{m-1} h_k(x)$ be the current function. We want to add a base function $h_m(x)$ so that

$$F_m(x) = F_{m-1}(x) + h_m(x).$$

For regression, the least squares loss is

$$\mathscr{L} = \sum_{i=1}^{n} [y_i - (F_{m-1}(x_i) + h_m(x_i))]^2 = \sum_{i=1}^{n} (r_i - h_m(x_i))^2,$$

where

$$r_i = y_i - F_{m-1}(x_i)$$

is the residual error for observation i. We can then treat $\{(x_i, r_i), i = 1, ..., n\}$ as our new dataset, and learn a regression tree $h_m(x)$ from this new dataset.

For a general loss function, e.g., the mean absolution deviation,

$$\mathscr{L} = \sum_{i=1}^{n} |y_i - (F_{m-1}(x_i) + h_m(x_i))|,$$

which penalizes large deviations less than the least squares loss and is thus more robust to outliers, we may learn a new tree $h_m(x)$ by minimizing $|r_i - h_m(x_i)|$. However we need to rewrite the code for learning the tree. It is better to continue to use the code for least squares regression tree.

Let

$$s_i = F_{m-1}(x_i) + h_m(x_i),$$

for $i = 1, ..., n$, and let $\hat{s}_i = F_{m-1}(x_i)$ be the prediction by F_{m-1}. Let the loss function be $\mathscr{L} = \sum_{i=1}^{n} L(y_i, s_i)$. In the case of mean absolute deviation, $\mathscr{L}(y_i, s_i) = |y_i - s_i|$. Let

$$r_i = -\frac{\partial L(y_i, s_i)}{\partial s_i}\Big|_{\hat{s}_i},$$

Then $r = (r_i, i = 1, ..., n)^\top$ is the direction to change $f = (f_i, i = 1, ...,)$ for the steepest descent of $\mathscr{L}$. r_i becomes the residual error in the least squares regression. In general, r_i means the lack of fitness for example i because if r_i is close to 0, there is no need to change s_i to fit the data. Otherwise we need to make big change in f_i to fit the data.

We can fit a tree to approximate r by minimizing

$$\sum_{i=1}^{n} (r_i - h_m(x_i))^2$$

using the code of least squares regression tree.

A more principled implementation of gradient boosting is to expand $L(y_i, s_i)$ by the second order Taylor expansion,

$$L(y_i, s_i) \approx L(y_i, \hat{s}_i) + L'(y_i, \hat{s}_i) h_m(x_i) + \frac{1}{2} L''(y_i, \hat{s}_i) h_m(x_i)^2,$$

where $L'(y, s) = \frac{\partial}{\partial s} L(y, s)$, and $L''(y, s) = \frac{\partial^2}{\partial s^2} L(y, s)$. Let $r_i = L'(y_i, \hat{s}_i)$ and $w_i = L''(y_i, \hat{s}_i)$, then we can write

$$\begin{aligned} L(y_i, s_i) &= \frac{w_i}{2} \left[h_m(x_i)^2 - 2 \frac{r_i}{w_i} h_m(x_i) \right] + \text{const} \\ &= \frac{w_i}{2} \left[\frac{r_i}{w_i} - h_m(x_i) \right]^2 + \text{const} \\ &= \frac{w_i}{2} \left[\tilde{y}_i - h_m(x_i) \right]^2 + \text{const}. \end{aligned}$$

Thus, weighted least squares can be used to learn h_m with $\tilde{y}_i = r_i / w_i$ as the response and w_i as the weights. Compared to the original gradient boosting, we take into account the curvature w_i. The bigger the curvature it, the more important the observation is. It implies that $\frac{r_i}{w_i}$ will see a bigger decrease for a small step is curvature is big.

1.5 Problem 5

For separable linear support vector machine (SVM), $\hat{y} = \text{sign}(\langle w, x \rangle)$. *Consider the primal max margin problem*

$$\min_w \frac{1}{2}\|w\|^2,$$
$$\textit{subject to } y_i w^\top x_i \geq 1, \forall i = 1, ..., n.$$

Translate it to the dual maximization problem and derive the dual coordinate ascent algorithm. Explain how to kernelize the above linear SVM.

For ease of notation, lets redefine $\hat{y}$ as $\hat{y} = \text{sign}(x^\top \beta)$.

The optimization problem in terms of primal min-max is given by:

$$\hat{\beta} \leftarrow \min_\beta \sum_{i=1}^n \max_{\alpha_i \in [0,1]} \alpha_i (1 - y_i x_i^\top \beta) + \frac{\lambda}{2}|\beta|^2.$$

We can change min max to max min, so that we change the primal problem into the dual problem. For convenience, let $C = 1/\lambda$ and $\alpha_i \leftarrow C\alpha_i$:

$$\begin{aligned}\hat{\beta} &\leftarrow \max_{\alpha_i \in [0,C] \forall i} \min_\beta \sum_{i=1}^n \alpha_i (1 - y_i x_i^\top \beta) + \frac{1}{2}|\beta|^2 \\ &= \max_{\alpha_i \in [0,C] \forall i} \min_\beta \left[\frac{1}{2} \left| \beta - \sum_{i=1}^n \alpha_i y_i x_i \right|^2 - \frac{1}{2} \left| \sum_{i=1}^n \alpha_i y_i x_i \right|^2 + \sum_{i=1}^n \alpha_i \right].\end{aligned}$$

The minimization gives us the representer

$$\hat{\beta} = \sum_{i=1}^n \alpha_i y_i x_i.$$

The dual problem is $\max_{\alpha_i \in [0,C], \forall i} q(\alpha)$,

$$q(\alpha) = \left[\sum_{i=1}^n \alpha_i - \frac{1}{2} \left| \sum_{i=1}^n \alpha_i y_i x_i \right|^2 \right] = \frac{1}{2}\alpha^\top Q \alpha - \alpha^\top \mathbf{1},$$

where $\mathbf{1}$ is a vector of 1's.

The dual problem can be solved by stochastic coordinate descent. At each time step t, we randomly pick an index i and compute the optimal one-variable update:

$$\delta^* = \underset{\delta: 0 \leq \alpha_i + \delta \leq C}{\arg\max}\ q(\alpha + \delta e_i),$$

where e_i is the one-hot vector where the i-th element is 1.

$$\begin{aligned}q(\alpha + \delta e_i) &= \frac{1}{2}(\alpha + \delta e_i)^\top Q (\alpha + \delta e_i) - (\alpha + \delta e_i)^\top \mathbf{1} \\ &= \frac{Q_{ii}}{2}\delta^2 + \alpha^\top Q_i \delta - \delta,\end{aligned}$$

where Q_i is the i-th column of Q, and Q_{ii} is the i-th diagonal element of Q. By differentiating the above with respect to δ and setting it to be zero, we have

$$\delta = \frac{1 - \alpha^\top Q_i}{Q_{ii}}.$$

However, we require $0 \leq \alpha_i + \delta \leq C$, so the optimal solution is

$$\delta^* = \max\left(-\alpha_i, \min\left(C - \alpha_i, \frac{1 - \alpha^\top Q_i}{Q_{ii}} \right) \right).$$

To kernelize the linear SVM, we assume feature vector $h(z)$ ad formulate the response as $\hat{y} = \text{sign}(h(x)^\top \beta)$. For detailed expression, please refer to subsection 2.3.

2 Coding Part

2.1 Gradient Boosting Algorithm

Gradient boosting algorithm is one of the most popular algorithms used in regression and classification tasks. At its core, the gradient boosting implements boosting mechanism. Boosting mechanism is an idea of using weak learners to make strong prediction. A learner is said to be weak if it can perform slightly better than a random guess. Similarly, a learner is said to be strong if it is able to perform much better than random guessing. The idea of boosting is rooted in iterative process of adding data, weak learners, and adjusting weights such that with each increasing iterations the error decreases as shown in Figure 2. In the simplest scenario, given the data points, a weak learning is first initialized and training and testing errors are computed. In the first iteration, the weight would simply be 1 because there is only one weak learner. Once more weak learners are added, the weights are adjusted accordingly such that the loss is minimized. If equal weighting scheme is used $w_i = 1/N$ is used as the weights where N is the total number of the weak learners. However, if the weights are computed based on the gradients, then it is called gradient boosting.

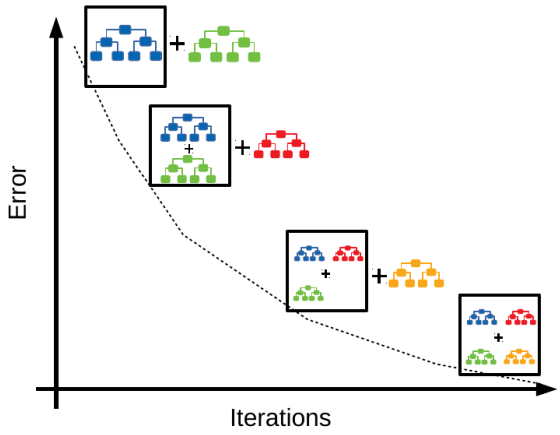

Figure 2: Graphical illustration of gradient boosting iteration and decrease in loss/error as a function of number of iterations.

2.1.1 Annotating the Code

For implementation of the gradient boosting algorithm, an example code was provided. The steps implemented in the code is listed as follows

1. Fit a simple linear regressor or decision treen on the dataset. In the provided code, decision tree is used.
2. Calculate the residual by taking the difference between the true value and predicted value.
3. Fit a new model on the residual as the target variable while utilizing the same feature variables.
4. Append the predicted residual to the previous predictions.
5. Fit another model on the residuals that's left and repeat the steps of refitting new model on residuals until the sum of residuals minimize or converge to a certain constant number.

2.2 Xtreme Gradient Boosting (XGB)

2.2.1 Algorithm

Figure 3 highlights the algorithm of the XGBoost. The first step is to initialize the model with a constant value because it is the first step. In the second, the base learner are fitted such that the residuals are minimized for a total of M iterations. The output of the algorithm is the updated model that minimizes the loss.

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Figure 3: Pseudo-code of the XGBoost algorithm.

2.2.2 XBGClassifier Default Setting

Figure 4 shows the setting used in the default setting in XGBClassifier. As we can see that the default setting uses 100 estimators while the maximum depth not specified. When the default setting was used, the model achieved a 99.78% training accuracy and 95.87% testing accuracy.

```
XGBClassifier(base_score=None, booster=None, colsample_bylevel=None,
              colsample_bynode=None, colsample_bytree=None,
              eval_metric='mlogloss', gamma=None, gpu_id=None,
              importance_type='gain', interaction_constraints=None,
              learning_rate=None, max_delta_step=None, max_depth=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=None, num_parallel_tree=None,
              random_state=None, reg_alpha=None, reg_lambda=None,
              scale_pos_weight=None, subsample=None, tree_method=None,
              use_label_encoder=False, validate_parameters=None,
              verbosity=None)
Training accuracy is 99.78 and testing accuracy is 95.87 %
```

Figure 4: Structure of the XGBoost classifier with default setting.

2.2.3 XBGClassifier Hyperparameter Tuning

Figure 5 shows the setting used in the default setting in XGBClassifier. As we can see that after tuning the hyperparameters (n_estimators and max_depth) the model achieved a 99.86% training accuracy and 95.11% testing accuracy. The optimal parameters found were 30 estimators and 400 maximum depth.

2.3 Support Vector Machine (SVM)

SVM is another popular algorithm used in supervised learning for classification and regression tasks. As shown in Figure 6, for a given labeled dataset with the response variable marked as belonging to one of two categories, an SVM algorithm builds a model that assigns new/unseen data point to one of the two categories, making it a non-probabilistic binary linear classifier. SVM maps training examples to points in space such that the width between the two categories which is also called margin, is maximized.

2.3.1 Linear SVM

Lets assume that the training data in in the form of $(X_1, y_1), ..., (X_n, y_n)$ where y_i are either 1 or -1 indicating the class to which the point X_i belongs to and the X_i refers to p-dimensional vectors. The objective function is to find the hyperplane such the margin is maximized as show in Figure 6. Here, a hyperplane can be written as:

$$w^T x - b = 0$$

where w is a normal vector to the hyperplane.

```
RandomizedSearchCV(cv=10,
                   estimator=XGBClassifier(base_score=None, booster=None,
                                           colsample_bylevel=None,
                                           colsample_bynode=None,
                                           colsample_bytree=None,
                                           eval_metric='mlogloss', gamma=None,
                                           gpu_id=None, importance_type='gain',
                                           interaction_constraints=None,
                                           learning_rate=None,
                                           max_delta_step=None, max_depth=None,
                                           min_child_weight=None, missing=nan,
                                           monotone_constraint...
                                           num_parallel_tree=None,
                                           random_state=None, reg_alpha=None,
                                           reg_lambda=None,
                                           scale_pos_weight=None,
                                           subsample=None, tree_method=None,
                                           use_label_encoder=False,
                                           validate_parameters=None,
                                           verbosity=None),
                   n_iter=20, n_jobs=10,
                   param_distributions={'max_depth': [100, 200, 300, 400, 500,
                                                      600, 700, 800, 1000,
                                                      2000],
                                        'n_estimators': [10, 30, 50, 70, 90,
                                                         150, 200]},
                   random_state=0, scoring='accuracy')
```

```
{'n_estimators': 30, 'max_depth': 400}
Training Accuracy 99.86 %
Testing Accuracy 95.11 %
```

Figure 5: Randomized search implemented using Scikit-Learn library. It can be seen that the all other settings were set to default values except for maximum depth and number of estimators. The second figure shows the best hyperparameters that were found to have maximize the evaluation metric which was chosen to be accuracy.

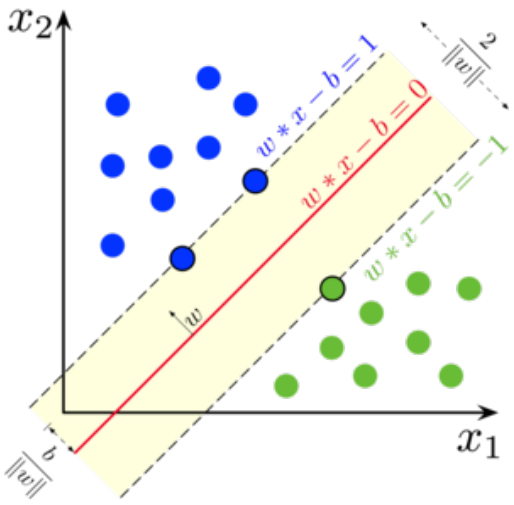

Figure 6: Graphical illustration of the SVM algorithm. The primary idea of SVM is to maximize the margin between the clusters which is simply $\frac{2}{||w||}$ in case of two linear classifier.

2.3.2 Hard Margin

If the training data is linearly separable such that two parallel hyperplanes can be used to distinguish the two classes such that the margin is maximized, then the hyperplanes can be described by:

$$w^T x - b = 1$$

and

$$w^T x - b = -1$$

where the distance between two hyperplanes is given by $\frac{2}{||w||}$ Now, the optimization problem can be written as:

$$\begin{array}{ll} \text{minimize} & ||w||, \\ \text{subject to} & y_i(w^T x_i - b) \text{ for } i = 1, ..., n. \end{array}$$

2.3.3 Soft Margin: Hinge Loss

To extend SVM to cases in which the data are not linearly separable, the hinge loss function is helpful. For classification where $y_i \in \{+1, -1\}$, we can classify y_i by the perceptron model:

$$\hat{y}_i = \text{sign}(x_i^\top \beta).$$

To estimate β, we can modify the loss function of ridge regression $\mathscr{L}(\beta) = |Y - X\beta|^2 + \lambda|\beta|^2$ to

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \max(0, 1 - y_i x_i^\top \beta) + \frac{\lambda}{2}|\beta|^2.$$

This is the objective function of linear SVM.

2.3.4 Kernel SVM

For kernel SVM, we a assume feature vector $h(x)$, and classify y_i by

$$\hat{y}_i = \text{sign}(h(x_i)^\top \beta).$$

The loss function is

$$\mathscr{L}(\beta) = \sum_{i=1}^{n} \max(0, 1 - y_i h(x_i)^\top \beta) + \frac{\lambda}{2}|\beta|^2.$$

The representer is

$$\hat{\beta} = \sum_{i=1}^{n} \alpha_i y_i h(x_i).$$

The dual problem is $\max_{\alpha_i \in [0,C], \forall i} q(\alpha)$,

$$\begin{aligned} q(\alpha) &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle h(x_i), h(x_j) \rangle \\ &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j). \end{aligned}$$

Again the problem can be solved by dual coordinate ascent.

After solving α_i, we get $\hat{\beta}$ from the representer, and the estimated function is

$$\begin{aligned} \hat{f}(x) &= h(x)^\top \hat{\beta} \\ &= \sum_{i=1}^{n} \alpha_i y_i \langle h(x_i), h(x) \rangle \\ &= \sum_{i=1}^{n} \alpha_i y_i K(x_i, x). \end{aligned}$$

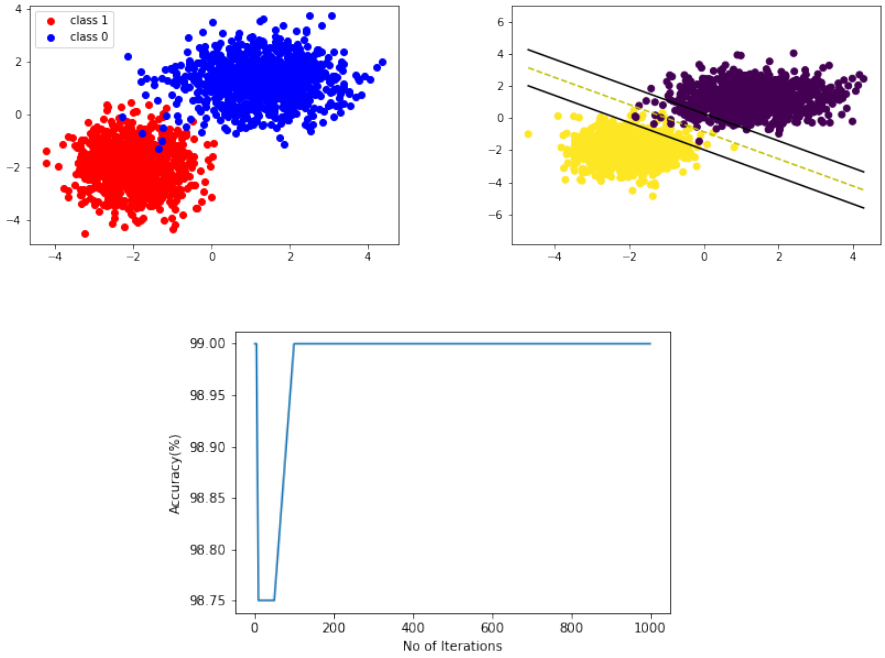

Figure 7: Scatter plot of the synthetic data provided by the TA (top left); fitted linear SVM from scratch for the 20% of the testing dataset (top right) and accuracy plot as a function of the number of iterations (bottom middle).

2.3.5 SVM from Scratch

Figure 7 shows a visual illustration of implementation of SVM algorithm with linear classifier. It also shows the optimal hyperplane that separates the two classes of synthetic data provided by the TA. The third figure shows the performance of the algorithm with respect to the number of iterations. As we can see, the performance of the algorithm is pretty high as it achieves 99% accuracy for a baseline algorithm. When the number of iterations are increased, the accuracy converges to 99%.

2.3.6 SVM Default Setting

Table 1 lists the training and testing accuracy for the default setting. We can see that for the linear SVM, the training and testing accuracy was 88.59 and 86.43%, respectively. The relatively similar accuracy for training and testing indicates that the algorithm generalized well. For Gaussian kernel SVM, the training and testing accuracy decreases to 71.36 and 66.23 %. The difference between Linear and Gaussian SVM indicates that Linear SVM might be more relevant to the given synthetic dataset.

Table 1: Training and testing accuracy of Linear and Gaussian SVM with default setting.

Type of Model	Training Accuracy (%)	Testing Accuracy (%)
Linear SVM	88.59	86.43
Gaussian SVM	71.36	66.23

2.3.7 SVM Hyperparameter Tuning

Figure 8 shows the output of the grid search to find the optimal parameters for SVM with linear and gaussian kernel. For the linear SVM $c = 0.1$ and $max\_iter = 100,000$ were found to give the optimal results whereas $c = 100$ and $max\_iter = 10,000$ were found to be optimal for the gaussian SVM.

Table 2 outlines the training and testing accuracy for the Linear and Gaussian SVM algorithm along with the optimal hyperparameters for the respective algorithms. We can see that the both the training

```
GridSearchCV(estimator=LinearSVC(),
             param_grid={'C': [0.1, 1, 10, 100],
                         'max_iter': [10, 1000, 10000, 100000]})
Best parameters are: LinearSVC(C=0.1, max_iter=100000)

GridSearchCV(estimator=SVC(),
             param_grid={'C': [0.1, 1, 10, 100],
                         'max_iter': [10, 1000, 10000, 100000]})
Best parameters are: SVC(C=100, max_iter=10000)
```

Figure 8: Best hyperparameters for linear SVM and Gaussian SVM.

and testing accuracy increased for SVM with linear and gaussian kernel as opposed to using the default setting.

Table 2: Training and testing accuracy of the Linear and Gaussian SVM with the best hyperparameters.

Type of Model	Training Accuracy (%)	Testing Accuracy (%)	Best Hyperparameters
Linear SVM	92.45	91.75	c=0.1 and max_iter = 100,000
Gaussian SVM	84.13	83.06	c=100 and max_iter = 10,000

How to run There are two Jupyter Notebook attached along with this document. All two notebooks were executed in local computer utilizing CPU capability. The combined run time for the two notebooks is about 20 minutes. However, Google Colab usage is encouraged to minimize the runtime. There are not data files required to run the notebook, meaning both the notebooks are standalone files.

1. “hw6_UsingScikit_Learn.ipynb" implements XGBoost and SVM algorithm from the scikit-Learn package. Hyperparameter tuning is also implemented within this notebook.
2. “hw6_SVMfromScratch.ipynb" implements Linear SVM from scratch and plots accuracy curve as a function of number of iterations.

3 Conclusion

In this homework, two of the most popular classification algorithms, XGBoost and SVM, are implemented to on the data provided by the TA to detect spam email. It was found that the the XGBoost performed well as compared to SVM (linear or gaussian). The testing accuracy for the XGB algorithm with default setting was 95.87% as compared to 95.11% when the hyperparameters (n_estimators and max_depth) are tuned. When SVM was implemented with default setting, the testing accuracy for the default setting was 86.43% and 66.23% respectively for linear and gaussian kernel. When the hyperparameters (c, and max_iter) are tuned, the testing accuracy improve to 91.75% and 83.06%, respectively. It indicates that, hyperparameter tuning has much greater impact for SVM than XGB.