2021 Fall STATSM231A/CSM276A: Pattern Recognition and Machine Learning

Deniz Orkun Eren
Department of Electrical and Computer Engineering
University of California, Los Angeles
Engineering IV Building, Los Angeles, California 90095
`deren@g.ucla.edu`

Abstract

In this course, you will learn multiple algorithms in machine learning. In general, for each homework, you are required to write the corresponding codes. We will provide a structure of code and a detailed tutorial online. Your job is (1) to deploy and understand the code. (2) Run the code and output reasonable results. (3) Make ablation study and parameter comparison. (4) Write the report with algorithms, experiment results and conclusions. For detailed instructions, please check the specification of each homework.

Homework submission forms

You have to submit both code and reports. For the code, zip everything into a single zip file. For the report, use this latex template. When submitting, no need to submit the tex file. Submit one single pdf file only.

1 Written Part

1.1 Problem 1

We are given the following expression:

$$\log p_\theta(x) - D_{\mathrm{KL}}(q_\phi(z|x)|p_\theta(z|x))$$

We know that $D_{KL}(P||Q) = E_P[log(P/Q)]$. So, the above expression becomes:

$$\log p_\theta(x) - E_{q_\phi(z|x)}(log(\frac{q_\phi(z|x)}{p_\theta(z|x)}))$$

Now, we can move $logp_\theta(x)$ inside the expectation expression. Doing so gives:

$$E_{q_\phi(z|x)}[logp_\theta(x) - log(q_\phi(z|x) + logp_\theta(z|x))]$$

Now, notice that $logp_\theta(x) + logp_\theta(z|x)$ can be combined as $log(p_\theta(x)p_\theta(z|x))$ Furthermore, we know that $p_\theta(z|x) = p_\theta(x,z)/p_\theta(x)$. Using this, $log(p_\theta(x)p_\theta(z|x))$ becomes $logp_\theta(z,x)$. So, our expression turns to the following:

$$E_{q_\phi(z|x)}[logp_\theta(x,z) - log(q_\phi(z|x)]$$

Now, notice that $p_\theta(x,z) = p(z) * p_\theta(x,z)/p(z) = p(z) * p_\theta(x|z)$. So, our expression can now be expressed as:

$$E_{q_\phi(z|x)}[logp(z) + logp_\theta(x|z)] - E_{q_\phi(z|x)}[log(q_\phi(z|x)]$$
$$E_{q_\phi(z|x)}[logp_\theta(x|z)] - E_{q_\phi(z|x)}[log(q_\phi(z|x)) - logp(z)]$$
$$E_{q_\phi(z|x)}[logp_\theta(x|z)] - E_{q_\phi(z|x)}[log(\frac{q_\phi(z|x)}{p(z)})]$$

Now, from the KL divergence formula, we know that $D_{KL}(P||Q) = E(log(P/Q))$. So, the expression finally becomes:

$$E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x)|p(z))$$

This concludes the proof.

Now, to calculate D_{KL}, we need to calculate an integral over all the values of z, which is intractable. However, if z is reparametrized as $z = \mu_\phi(x) + \sigma_\phi(x) \odot e$, then D_{KL} simply becomes the KL divergence between two Gaussian distributions, which is explicitly defined. Thus, we can use Monte Carlo sampling to approximate the expectation, which makes the total expression tractable.

1.2 Problem 2

Let us approach this problem by considering the discriminator. The goal of the discriminator is to determine if a given input originated from the actual dataset or the generator. Let us call the label of a discriminator input y_i where y_i is 1 if the input originated from the data and 0 if the input came from the generator. Now, we can define the output of the discriminator as:

$$D(x) = P(y_i = 1|x_i)$$

where x_i is the input to the discriminator. Then, the goal of the discriminator is to maximise the likelihood

$$\prod_{i=1}^{n} P(y_i = 1|x_i)^{y_i} P(y_i = 0|x_i)^{1-y_i}$$

We can take the logarithm of this expression to get the log-likelihood:

$$\sum_{i=1}^{n} y_i log(P(y_i = 1|x_i)) + (1 - y_i) log(P(y_i = 0|x_i))$$

We can divide this expression by the number of samples.

$$\sum_{i=1}^{n} \frac{y_i}{n} log(P(y_i = 1|x_i)) + \frac{(1 - y_i)}{n} log(P(y_i = 0|x_i))$$

Now, we can see that $\sum_{i=1}^{n} y_i/n$ is the probability of getting a sample from the dataset. In contrast,$\sum_{i=1}^{n}(1 - y_i)/n$ is the probability of getting a sample generated by the generator. Furthermore, if $D(x) = P(y_i = 1|x_i)$, then $1 - D(x) = P(y_i = 0|x_i)$. Based on this information, our expression becomes:

$$\sum_{i=1}^{n} P(x_i fromdata) log(D(x_i)) + \sum_{i=1}^{n} P(\hat{x_i} fromgenerator) log(1 - D(\hat{x_i}))$$

where x_i is a sample from the real dataset and $\hat{x_i}$ is a sample from the generator. Let us refer to this expression as $\star$. Now, we know that $\sum p(x_i)x_i$ is the expectation. Furthermore, we also know that $\hat{x_i}$ is the output of the generator when a value is sampled from a normal distribution with 0 mean and

1 variance. So, $\hat{x_i} = G(\tilde{z})$ where $\tilde{z}$ is the sampled value and G is the generator function. Combining this, we get:

$$E_{p_{\text{data}}(x)}[\log D(x)] + E_{\tilde{z}\sim \text{N}(0,I)}[\log(1 - D(G(\tilde{z}))]$$

This is the function that the discriminator is trying to maximise. Since the discriminator and the generator oppose each other, this is also the function that the generator is trying to minimise. So, the minimax value function is:

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

Let $P(x_i from data)$ in $\star$ be referred to as $p_{\text{data}}(x)$. Furthermore, let $P(\hat{x_i} from generator)$ be referred to as $p_\theta(x)$. Now, to find the optimal D in $\star$, we can take the derivative of the expression and set it to 0. The derivative of the expression becomes:

$$\frac{\partial V(G, D))}{\partial D(x)} = \frac{p_{data}(x)}{D(x)} + \frac{-p_\theta(x)}{1 - D(x)} = 0$$

$$(1 - D(x))p_{data}(x) - D(x)p_\theta(x) = 0$$

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

So, optimal D(x) is:

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

If we plug this in to the V(G,D) expression, we get:

$$V(G, D) = E_{p_{\text{data}}(x)}[\log(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_\theta(x)})] + E_{\tilde{z}\sim \text{N}(0,I)}[\log(\frac{p_\theta(x)}{p_{\text{data}}(x) + p_\theta(x)})]$$

Let us now call $(p_{\text{data}}(x) + p_\theta(x))/2$ as p_{mix}. Furthermore, notice that the denominator of the expression inside the logarithm can be divided by 2, since finding the maximum of both expressions gives the same result. So, we have:

$$V(G, D) = E_{p_{\text{data}}(x)}[\log(\frac{p_{\text{data}}(x)}{p_{mix}})] + E_{p_\theta(x)}[\log(\frac{p_\theta(x)}{p_{mix}})]$$

Now, we know that $D_{KL}(P||Q) = E(log(P/Q))$. So, our expression becomes:

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

The rigth hand side of this equation is equivalent to Jensen-Shannon distance. So, $V(G, D)$ becomes:

$$V(G, D) = \text{JSD}(p_{\text{data}}, p_\theta)$$

This concludes the question.

1.3 Problem 3

We are given that:

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

Now, here a refers to all the actions that are taken to get the expected reward. So,it represents the trajectory. Now, the probability of a trajectory is

$$P(a|\theta) = \rho(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

Now, we can write $J(\theta)$ can be written as

$$J(\theta) = \int_a P(a|\theta)z(a)$$

Taking the derivative of this expression gives

$$J'(\theta) = \int_a \nabla_\theta P(a|\theta)z(a)$$

Now, we can make use of the log derivative trick. This trick makes use of the fact that $\frac{\partial f(x)}{\partial x} = f(x)\frac{\partial log(f(x))}{\partial x}$. Using this in the above expression gives:

$$J'(\theta) = \int_a P(a|\theta)\nabla_\theta log(P(a|\theta))z(a)$$
$$J'(\theta) = E_{\pi_\theta(a|s)}[\nabla_\theta log(P(a|\theta))z(a)]$$

Now, we need to calculate $\nabla_\theta log(P(a|\theta))$. Note that,

$$log(P(a|\theta)) = log(\rho_0(s_0)) + \sum_{t=0}^{T}(log(P(s_{t+1})) + log(\pi_\theta(a_t|s_t)))$$

Notice that in this expression, the only part that depends on θ is the part with π_θ. So, the derivative becomes:

$$\nabla_\theta log(P(a|\theta)) = \sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t) = \nabla_\theta log\pi_\theta(a|s)$$

Plugging this value to $J'(\theta)$, we get:

$$J'(\theta) = E_{\pi_\theta(a|s)}[\nabla_\theta log\pi_\theta(a|s)z(a)] = E_{\pi_\theta(a|s)}[z(a)\frac{\partial}{\partial\theta}log\pi_\theta(a|s)]$$

This concludes the derivation part of the question.

The difference between policy gradient reinforcement learning and gradient ascent for maximum likelihood supervised learning is that in the supervised version, the action to take at a specific state is given by an outside "expert". In the AlphaGo example, this can mean an expert human. However, in policy gradient reinforcement learning, there is no outside source. To determine the action to take at a certain state, we make use of the current policy instead of using outside data.

1.4 Problem 4

The new tree is defined by $h_m(x)$. We can define the prediction function that we have before adding the new tree as $s_i(x)$. Now, the prediction function after adding the tree is then $f(x) = s_i(x)+h_m(x)$. For logistic regression, the probability of predicting label 1 is:

$$\begin{aligned} P(y_i = 1|x_i) &= sigmoid(f(x_i)) \\ &= \frac{e^{f(x_i))}}{1+e^{f(x_i)}} \end{aligned}$$

Then, the goal is to maximise the likelihood which is defined as:

$$Likelihood = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i}$$

Plugging in the value for p_i, we get:

$$\begin{aligned} Likelihood &= \prod_{i=1}^{n}(\tfrac{e^{f(x_i)}}{1+e^{f(x_i)}})^{y_i}(1-\tfrac{e^{f(x_i)}}{1+e^{f(x_i)}})^{1-y_i} \\ &= \tfrac{e^{f(x_i)y_i}}{1+e^{f(x_i)}} \end{aligned}$$

Taking the logarithm of the likelihood gives the following:

$$LogLikelihood = \sum_{i=1}^{n} y_i f(x_i) - log(1+e^{f(x_i)})$$

We can replace $f(x_i)$ with $s_i(x_i) + h_m(x_i)$. Then, the log likelihood becomes:

$$\sum_{i=1}^{n} y_i(s(x_i) + h_m(x_i)) - log(1+e^{s(x_i)+h_m(x_i)})$$

Now, we can approximate the log likelihood by using the second order Taylor expansion. This process is defined as:

$$f(x+a) \sim f(x) + \frac{df(x)}{dx}a + \frac{1}{2}\frac{d^2f(x)}{dx}a^2$$

Now, we can see that the log likelihood function depends on $s(x_i)$ and $h_m(x_i)$. So, we can use second order Taylor expansion to approximate the log likelihood. Let us call the log likelihood L. Let us refer to $s(x_i)$ as s_i. Now, to form the approximation, we need to calculate $L(s_i)$ and its derivatives.

$$L(s_i) = y_i s_i - log(1+e^{s_i})$$

$$L'(s_i) = y_i - \frac{e^{s_i}}{1+e^{s_i}} = y_i - \hat{p_i} = r_i$$

$$L''(s_i) = -\hat{p_i}(1-\hat{p_i}) = -w_i$$

Now, let us approximate the log likelihood:

$$\begin{aligned} LogLikelihood &\sim \sum_{i=1}^{n} L(s_i) + L'(s_i)h_m(x_i) + \tfrac{1}{2}L''(s_i)h_m(x_i)^2 \\ &\sim \sum_{i=1}^{n} L(s_i) + r_i h_m(x_i) - \tfrac{1}{2}w_i h_m(x_i)^2 \end{aligned}$$

We need an $h_m(x_i)$ that maximises this goal. Note that $L(s_i)$ does not depend on $h_m(x_i)$, So, our objective becomes:

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

Maximising a function is equal to minimising the negative of that function. So, our new goal becomes:

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

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

We can add r_i^2/w_i to this expression since that does not depend on $h_m(x_i)$. Our new goal is then:

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

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

We can get rid of the $1/2$ term since it does not affect the result. So, our loss function to minimise becomes:

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

This concludes the derivatiion of the loss.

However, in XGBoost, loss is just one component that needs to be taken into consideration during optimization. The other component is regularization. From the documentation on XGBoost, we can understand that the complexity for a single tree is defined as:

$$\Omega(tree) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} k_j^2$$

where T is the number of leaves in the tree and k_j is the vector of scores on leaf j. The goal of the new $h_m(x_i)$ is then to minimise both the loss function obtained previously and the complexity of the tree. Minimising the complexity serves as regularization for XGBoost. Therefore, the total function to minimise can be written as:

$$L = \sum_{i=1}^{n} w_i(\frac{r_i^2}{w_i} - h_m(x_i))^2 + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} k_j^2$$

1.5 Problem 5

The goal in optimizing SVM is to minimise $\frac{1}{2}|w|^2$ subject to $1 - y_i w^T xi \leq 0$ for label y_i and data x_i. This is a constrained optimization problem, which can be expressed using the Lagrangian formulation as follows:

$$L(w, a) = \frac{1}{2}|w|^2 + \sum_{i=1}^{n} a_i(1 - y_i w^T x_i)$$

$$L(w, a) = \frac{1}{2}|w|^2 - \sum_{i=1}^{n} a_i y_i w^T x_i + \sum_{i=1}^{n} a_i$$

Our goal is to find the w that minimises this Lagrangian. If we take the derivative with respect to w and set it to 0, we get the following expression:

$$w - \sum_{i=1}^{n} a_i y_i x_i = 0$$

$$w = \sum_{i=1}^{n} a_i y_i x_i$$

Minimizing the Lagrangian is equal to maximising the dual formulation. To get the dual, we can replace w in the Lagrangian with $\sum_{i=1}^{n} a_i y_i x_i$. So, the dual problem $D(a)$ becomes:

$$D(a) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j y_i y_j x_i \cdot x_j$$

Let us refer to the $a_i a_j x_i \cdot x_j$ part of the expression as Q_{ij}. Then, the dual becomes:

$$D(a) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j Q_{ij}$$

with the condition each $a_i \geq 0$. Our goal is to find a_i that maximises the dual expression. After that, we can find w from $w = \sum_{i=1}^{n} a_i y_i x_i$. This completes the formulation of the dual maximisation problem.

To find the a_i that maximises the dual, we can use dual coordinate ascent. For an index k, the dual can be expressed as:

$$D(a_k) = a_k - \sum_{i \neq k} Q_{ik} a_i a_k - \frac{1}{2} Q_{kk} a_k^2 + \sum_{i \neq k} a_i$$

Here, we can see that $\sum_{i \neq k} a_i$ does not depend on a_k so it can be neglected when trying to find the a_k that maximises $D(a_k)$. To find the optimum a_k, we need to take the derivative of the remaining expression with respect to a_k and set it to 0. Doing so gives the following:

$$-Q_{kk} a_k + 1 - \sum_{i \neq k} Q_{ik} a_i = 0$$

$$a_k = \frac{1 - \sum_{i \neq k} Q_{ik} a_i}{Q_{kk}}$$

So, the optimal $\hat{a_k} = (1 - \sum_{i \neq k} Q_{ik} a_i)/Q_{kk}$. However, remember that all a_i must be greater than or equal to 0. So, our correct expression for the optimal a_k is:

$$\hat{a_k} = max(\frac{1 - \sum_{i \neq k} Q_{ik} a_i}{Q_{kk}}, 0)$$

We just derived the dual coordinate ascent algorithm! For each iteration in the algorithm, we iterate through all the samples in the training set. For each sample, we update the corresponding a_k using the above derivation. This process goes on until the limit for the number of iterations has been reached.

Now, the above derivations were for the linear SVM. However, for non-linearly separable data, we make use of kernelized SVM. The only difference here is in the formulation of the dual. Instead of the previously derived expression, we use:

$$D(a) = \sum_{i=1}^{n} a_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j y_i y_j K(x_i, x_j)$$

where K is the kernel function. We do this to obtain what the dot product would have been if we first projected the samples to a different space. Instead of actually projecting the samples, we make use of the kernel trick to calculate the projected dot product.

2 Coding part: Boosting and SVM

2.1 Problems

In this homework, we are asked to use two classification algorithms to perform classification on various datasets. The first algorithm that we use is a type of boosting algorithm called eXtreme Gradient Boosting (XGBoost) and the other algorithm is called the Support Vector Machine (SVM). We first implement SVM from scratch and use it to classify a toy dataset. Then, we use the sklearn implementation of the two algorithms to perform email spam detection. From the problem definitions, it is apparent that we are dealing with classification problems in this homework.

Let us now learn more about the algorithms that we will utilise.

2.2 Boosting

Boosting is an ensemble machine learning algorithm to perform supervised learning. The first polynomial-time boosting algorithm was introduced by Robert E. Schapire in his paper "The strength of weak learnability".

The goal of boosting algorithms is to iteratively grow a set of weak learners and use this set to perform classification. Here, a weak learner is defined as a classifier that can only perform slightly well on the entire dataset. Each weak learner is added to the set with a specific weight coefficient that reflects the weak learners performance. Moreover, when creating a new weak learner, not all data entries are given equal weight. More weight is given to the entries which are incorrectly classified by the current set and less weight is given to the ones classified correctly by the current set. The goal is to focus the weak learner on the data that the current set fails to classify.

There are a number of boosting algorithms that have been developed through the years (AdaBoost, LogitBoost, etc.) However, the one that we will focus on and the one that is used in the homework is titled eXtreme Gradient Boosting (XGBoost).

2.3 XGBoost

XGBoost is a type of gradient boosting algorithm that was introduced by Tianqi Chen and Carlos Guestrin in their paper: XGBoost: A Scalable Tree Boosting System.

Gradient boosting views boosting as an optimization of a specific cost function. The goal for any learning algorithm is to find a function $\hat{F}$ that minimizes a expected loss L.

$$\hat{F} = \arg\min_{F} E_{x,y}[L(y, F(x))]$$

This loss function can be mean squared error for regression tasks or logistic loss for binary classification tasks. In gradient boosting, we try to reach the optimal function by utilising a weighted combination of weak learners.

$$\hat{F} = \sum_{i=1}^{M} \gamma_i h_i(x) + constant$$

Here, $h_i(x)$ is the ith weak learner and γ_i is the weight of the weak learner.

The algorithm to grow the weak learners is as follows. First, we initialise a learner as a constant value that minimises the loss function. Of course, this constant value can not give optimal performance. We need to update our set of learners in such a way that the loss is as minimised as possible. To do so, we use the gradient descent algorithm. First, we calculate the derivative of the loss with respect to our existing prediction function:

$$r_{i,m} = -[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}]$$

This is referred to as the pseudo residual. We need a new learner in our set that captures this residual as much as possible. Therefore, we train a new weak learner on this residual. Let us call this weak residual h_m. Now, all that remains is to determine the weight of this new weak learner. To do so, we solve the following optimisation problem:

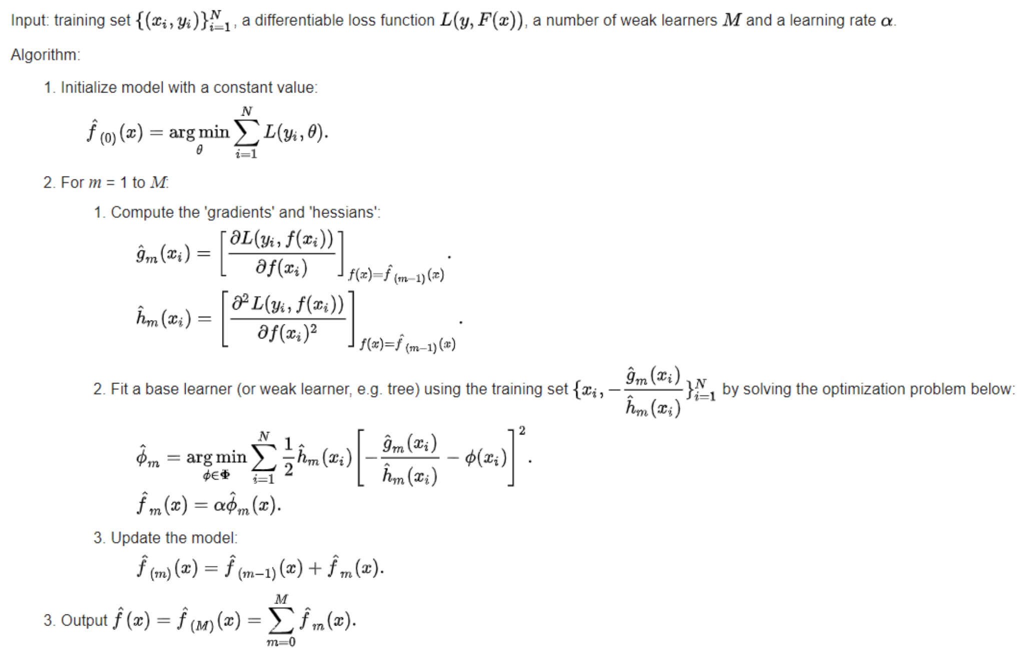
$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$$

Here, the goal is to find the weight of the new tree so that the loss after adding the new tree with the weight coefficient to our existing set of predictors minimises the loss. Now all that remains is to update our set of learners:

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

This process repeats itself until the amount of learners in our system reaches a set amount.

XGBoost then is simply a variation of the explained gradient boosting algorithm. The main difference is that it uses the Newton-Rhapson method by performing a second order Taylor expansion on the loss function instead of using gradient descent. The algorithm for XGBoost without regularization can be summarised as follows:


```
Input: training set $\{(x_i, y_i)\}_{i=1}^{N}$, a differentiable loss function $L(y, F(x))$, a number of weak learners $M$ and a learning rate $\alpha$.
Algorithm:
  1. Initialize model with a constant value:
       $\hat{f}_{(0)}(x) = \arg\min_{\theta} \sum_{i=1}^{N} L(y_i, \theta).$
  2. For $m$ = 1 to $M$:
       1. Compute the 'gradients' and 'hessians':
            $\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}_{(m-1)}(x)}.$
            $\hat{h}_m(x_i) = \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2}\right]_{f(x)=\hat{f}_{(m-1)}(x)}.$
       2. Fit a base learner (or weak learner, e.g. tree) using the training set $\{x_i, -\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)}\}_{i=1}^{N}$ by solving the optimization problem below:
            $\hat{\phi}_m = \arg\min_{\phi \in \Phi} \sum_{i=1}^{N} \frac{1}{2}\hat{h}_m(x_i)\left[-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)} - \phi(x_i)\right]^2.$
            $\hat{f}_m(x) = \alpha\hat{\phi}_m(x).$
       3. Update the model:
            $\hat{f}_{(m)}(x) = \hat{f}_{(m-1)}(x) + \hat{f}_m(x).$
  3. Output $\hat{f}(x) = \hat{f}_{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x).$
```

Compared to the standard gradient boosting algorithm, XGBoost gives more accurate and faster to train models since it uses the strengths of second order derivative of the loss function, utilises L1 and L2 regularization and is very suitable for parallel computing.

To better understand gradient boosting, we can examine the Python implementation of gradient boosting provided in the homework description.

Here, a decision tree with a single split is chosen as our weak learner. The implementation of the weak learner is defined in the DecisionTree class in the second section. As this part of the code is not vital for understanding the gradient boosting algorithm, we will not elaborate it further.

The main part of the gradient boosting algorithm is at section 8. Here, we first define our constant value as 0 with the variable predf. Then, we start building weak learners in the for loop.

Initially, we fit a decision tree to the values and labels given in the training set on line 9 and 10 of this section. This tree splits the values in such a way that the standard deviation of the values in the resulting branches is minimum. We then find the indexes of the values that should be on the left of the split and on the right of the split using the lines 14 and 15. Since this is a regression problem, it makes sense to have our weak learner output the mean of the values at the left of the split whenever a given sample is at the left of the split and the mean of the values at the right of the split whenever a given sample is at the right of the split. Based on this, we use our weak learner to generate predictions for the values in the training data in lines 17 to 19.

Now, we are ready to add the weak learner to our set. To do so, we sum the previous predictions (which is just 0 in the initial case) and the predictions of the new learner (line 22). We then find the residual between the desired targets and the prediction of our system (line 24). This pretty much completes one iteration. To prepare for the next iteration, we set the targets of the new weak learner to the residual (line 25). Furthermore, our current set is now the combination of the constant value and the recently generated weak learner.

In the next loop iteration, we fit a weak learner to the previous residual. We then add the new learner to our current set, resulting in a set with two weak learners. We use the new set to get the residual in order to obtain the target for the next weak learner that will be grown. This process repeats itself until the amount of weak learners reaches a given limit, which is set as 30 for the given code.

2.4 Support Vector Machine

Support Vector Machine (SVM) is a supervised machine learning that can be used for classification and regression. The method was developed at AT&T Bell labs by Vladimir Vapnik and his colleagues.

The goal of SVM is to calculate a separating hyperplane that maximises the distance between data points of different classes. This distance is referred to as the margin. The hyperplane is calculated using the data entries that are on the edge of the boundary for their specific class.In other words, these are the data points that are closest to the points from the other class. These data points are referred to as support vectors.

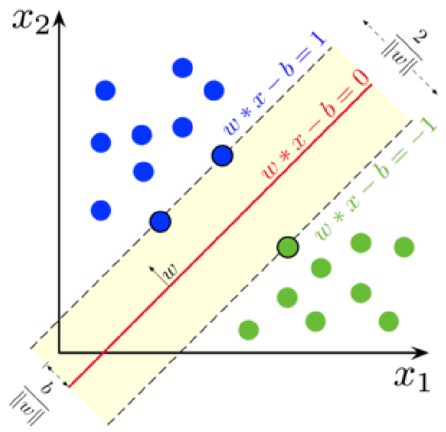


Figure 1: The visualisation of the SVM

Here, the blue dots that are on the dotted line are the support vectors for the blue class and the green dot on the dotted line is the support vector for the green class. Our goal in this picture is to find the w and b values that maximise the margin between the dotted lines. Since the margin can be calculated by $\frac{2}{\|w\|}$, we need to minimise w. However, we also need to ensure the following holds in order to perform accurate classifications:

$$x_i \cdot w + b \geq +1 \text{ when } y_i = +1$$
$$x_i \cdot w + b \leq -1 \text{ when } y_i = -1$$

Here, x_i is the ith data entry and y_i is the ith class label. The conditions above can be combined as:

$$y_i(x_i \cdot w) \geq 1$$

Since we are minimising w with respect to a condition, this is a constrained optimization problem. We can express this problem using the Lagrangian formulation as follows:

$$minL_p = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{l} a_i y_i (x_i \cdot w + b) + \sum_{i=1}^{l} a_i$$

If we take the derivative of the Lagrangian with respect to w and b, we get the following conditions:

$$w = \sum_{i=1}^{l} a_i y_i x_i \text{ and } \sum_{i=1}^{l} a_i y_i = 0$$

By plugging in the values for w and b, we can get a new expression. Here, instead of minimising over w and b, our goal is to maximise over a subject to the following:

$$\sum_{i=1}^{l} a_i y_i = 0 \text{ and } a_i \geq 0$$

So our problem now becomes:

$$maxL_D(a_i) = \sum_{i=1}^{l} a_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} a_i a_j y_i y_j (x_i \cdot x_j)$$

This is called the dual problem. By solving this problem, we can get optimum a_i values. We then can use these a_i values on the conditions derived from the Lagrangian formulation to get the values for w and b.

This approach, dubbed the linear SVM, works when the data in question is linearly separable. However, in real life this is rarely the case. To use the SVM method on non-linearly separable data, we make use of kernel SVMs.

The goal of the kernel SVM is to project the data to a higher dimension where the data is linearly separable. We then perform the same operations as the linear SVM.

A clever trick that makes the kernel SVM computationally feasible is to use the kernel trick. For the dual problem, instead of projecting the data first and then calculating the dot product, we instead project the dot product to the higher dimension. This makes the calculations much faster.

In summation, the dual function we need to optimize for kernel SVM is:

$$maxL_D(a_i) = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} a_i a_j y_i y_j K(x_i \cdot x_j)$$

There are a couple of popular kernel functions that we can use in the kernel SVM. These are:

$$\text{Polynomial kernel} = (x \cdot y + 1)^p$$
$$\text{Radial basis function kernel} = exp(\frac{-\|x-y\|^2}{2\sigma^2})$$
$$\text{Sigmoid kernel} = tanh(kx \cdot y - \delta)$$

Now that we have a better understanding of the relevant classification algorithms, we can move on to their implementation in the homework.

2.5 Experiments

2.5.1 XGBoost for spam email classification

How to run To run the code for this section, use the real_data.py python file. Here, getAccuracy returns the accuracy of a model on a particular data, tuneParamsXGB trains a number of XGBoost models with specified number of estimators and maximum depth. Lines 78 to 84 are for training the default XGBoost model and getting the accuracy of the trained model on the training and testing sets. Lines 86 to 91 are for creating a number of XGBoost models with different parameters and getting the training/testing accuracy of these models.

In this part of the homework, we are asked to use the XGBoost model to detect f an email is spam or not based on 57 features. In total, there are 4601 data instances.

The first step is to split the data for training and testing. To do this, the train_test_split function of the sklearn package was utilised. After this step, 3680 random data instances were reserved for training and the rest 921 data instances were reserved for testing. Then, using the XGBClassifier class of the xgboost package, we fit the default XGBoost model on the training dataset. After training, we tested the model on the training and testing data to get the desired accuracy values. The training accuracy for the default XGBoost model was found to be **99.78%** while the testing accuracy was found to be **95.87%**.

After that, we changed the n_estimator and max_depth parameters of the XGBoost classifier to see the change in performance. Here, n_estimator determines the amount of trees that will be used in the final classifier and max_depth determines the maximum depth of a single tree. The tried values for these two parameters were as follows:

$$n\_estimator = 300, 500, 1000$$
$$max\_depth = 5, 10, 20$$

We can summarise the results for each parameter combination in the following table:

Table 1: Comparison of training and testing performance for XGBoost with different parameters

n_estimators	max_depth	Training Accuracy (%)	Testing Accuracy (%)
300	5	99.4	96.0
300	10	99.9	95.2
300	20	99.9	95.5
500	5	99.4	96.0
500	10	99.9	95.2
500	20	99.9	95.5
1000	5	99.4	96.0
1000	10	99.9	95.2
1000	20	99.9	95.5

As can be seen, the best training accuracy with the least number of estimators is obtained when the number of estimators is 300 and the max depth is 10. On the other hand, the best testing accuracy is obtained when the number of estimators is 300 and the maximum depth is 5.

An interesting point to note here is that the models that perform slightly worse on the training set performs better on the test set. This can be attributed to overfitting. The model that learns the training data too well has worse generalization capabilities than the model that does not learn the training data as well.

2.5.2 SVM from scratch

How to run To run this code, use the toy_data.py file. Here, the first section loads the data and then splits it for training and testing. The second section trains an SVM on the training data and tests the SVM on the testing data. The third section draws the progress of the training and testing accuracies through the training iterations. The actual implementation of the SVM is on the svmImplementation.py file within the SVM class. Here, predictSVM is used to predict the label of a data entry and trainSVM is used to train an SVM model. As the code was written using the Spyder IDE, this IDE needs to be used to run the code section by section.

In this part of the homework, we are asked to implement the SVM classifier from scratch without using any advanced libraries. To do so, we first initialise the w and b values. Then, for each data entry in the training set, we generate a prediction for the data instance. If this prediction is true, we update the w value by using the following formula:

$$w_{new} = w_{old} - \alpha(2 * \lambda * w_{old})$$

where α is the learning rate for the gradient descent and λ is the regularization parameter.

On the other hand, if the classification is wrong, w update w and b using the following formula:

$$w_{new} = w_{old} - \alpha(2 * \lambda * w_{old} - x \cdot y)$$
$$b_{new} = b_{old} - \alpha * y$$

where x is the features of a data entry and y is the label of that data entry.

Performing the steps described above for all the data entries of the training data completes an iteration. At the end of each iteration, we test the current SVM on the training and testing data and record the accuracy values. Training continues until the limit for the number of iterations is reached.

To make sure our algorithm works as intended, let us use the code provided in the homework description to generate two Gaussian distributions of two different classes. The resulting data can be visualised as below:

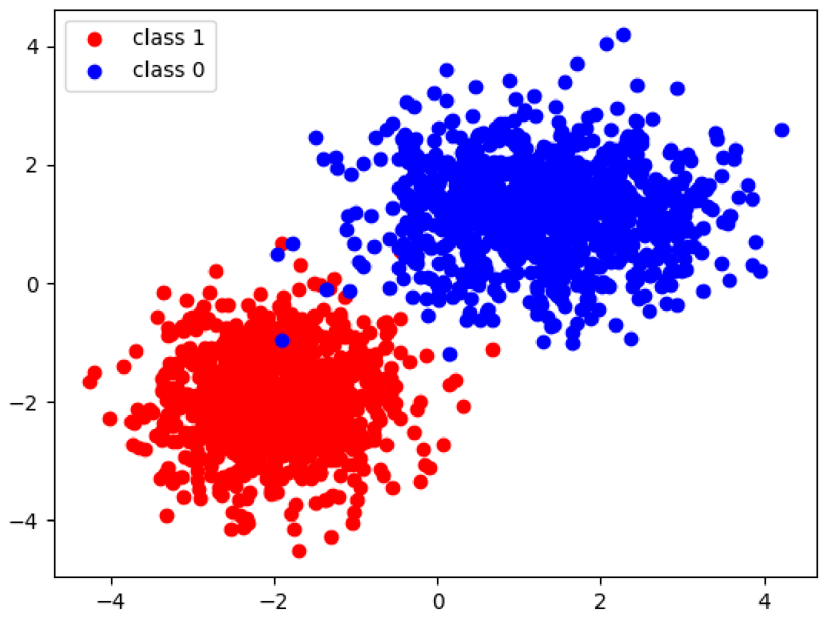


Figure 2: The generated toy data

As can be seen, the data is almost linearly separable, meaning that we should expect to get high accuracy scores for both training and testing data.

Now, we can train our implemented SVM . First, we split the generated data for training and testing using a 80-20 split. We then train an SVM on the training data with α equal to 0.001 and λ equal to 0.001. We let the training algorithm run for 10 iterations and record the training and testing accuracy at the end of each iteration. In the end, we get an SVM model with a training accuracy of **99.68%** and testing accuracy of **99.25%**. The progress of the training and testing accuracies through training iterations can be visualised as below:

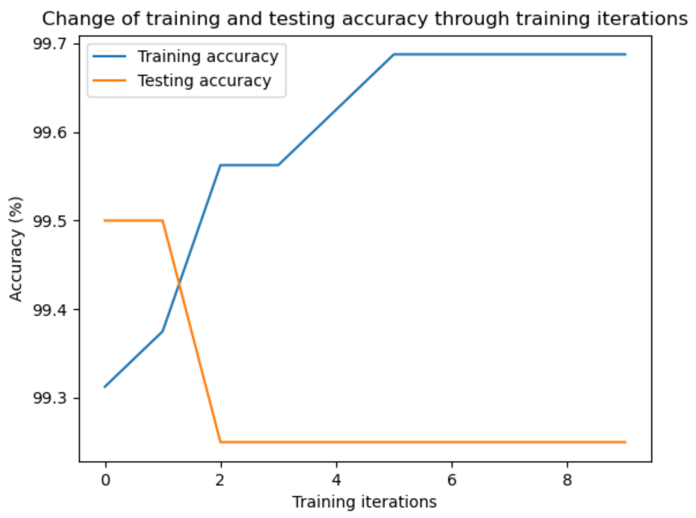


Figure 3: Change of training and testing accuracy through training iterations

As can be seen, the implemented SVM is capable of getting high accuracy scores from the first iteration. For maximum testing accuracy, training could have been stopped at the end of the first iteration. However, even after performing training for more than necessary, the SVM is capable of achieving high scores for both training and testing. The success of the model implies that our implementation of the SVM is correct.

2.5.3 SVM for spam email classification

How to run To run the code for this section, use the real_data.py python file. Here, getAccuracy returns the accuracy of a model on a particular data, tuneParamsLinearSVC trains a number of linear SVM models with specified values for the C and max_iter parameters and tuneParamsGaussianSVC trains a number of SVM models that use the Gaussian kernel with specified values for the C and max_iter parameters. Lines 96 to 102 are for training the default linear SVM model and getting the accuracy of the trained model on the training and testing sets. Lines 104 to 109 are for creating a number of linear SVMs with different parameters. Lines 114 to 120 are for training and testing the default Gaussian SVM. Lines 122 to 127 are for creating different Gaussian SVMs with varying parameters.

In this part of the homework, we revisit the email spam classification problem described in the XGBoost for spam email classification section of the report. However, this time instead of the XGBoost algorithm, we use SVMs with linear and Gaussian (radial basis function) kernels.

To train the models, the training and testing sets generated during the run for XGBoost were utilised. Then, we fit the default linear SVM to the training data using the SVC class in the sklearn.svm package. After training, we tested the model on the training and test sets to get the desired accuracy values. The training accuracy for the default linear SVM was observed to be **93.5%** and the testing accuracy was found to be **92.2%**.

After that, we changed the C and max_iter parameters of the linear SVM model. Here, the C parameter controls the regularization of the model. As C increases, the amount of regularization decreases, meaning that misclassified points are penalized heavily. The reverse happens when C decreases. Max_iter determines the number of optimization steps that the SVM solver will go through. The tried values for these two parameters were as follows:

$$C = 0.25, 0.5, 2$$
$$max\_iter = 10000, 100000, 1000000$$

We can summarise the results for each parameter combination for the linear SVM as follows:

Table 2: Comparison of training and testing performance for the linear SVM with different parameters

C	max_iter	Training Accuracy (%)	Testing Accuracy (%)
0.25	10000	58.4	59.1
0.25	100000	70.1	66.9
0.25	1000000	91.2	90.3
0.5	10000	58.6	59.7
0.5	100000	73.3	70.0
0.5	1000000	92.8	93.2
2	10000	34.5	38.2
2	100000	70.1	71.3
2	1000000	75.4	71.8

As can be seen from the table, the highest training accuracy is obtained when the linear SVM is set to have a C value of 0.5 and maximum iteration of 1000000. The highest test accuracy is also obtained with this combination. By observing the results, we can also reach the conclusion that increasing the number of iterations has a positive effect on the performance of the linear SVM.

Now, instead of using the linear SVM, let us use the Gaussian SVM which uses the radial basis function kernel. This SVM projects the data to a higher dimension in order to find a better separating hyperplane.

First, we fit the default Gaussian SVM to the same training data using the SVC class of the sklearn.svm package. After training, the model was tested on the training and testing datasets to get the accuracy values. The training accuracy of the default Gaussian SVM was found to be **71.3%** while the testing accuracy was found to be **66.2%**.

It is interesting to note that the performance of the default Gaussian SVM is worse than the performance of the default linear SVM. From here,it could be implied that the data in its current form is linearly separable, which would explain the high performance of the linear SVM.

As before, let us change the C and max_iter parameters of the Gaussian SVM to see the change in performance. The tried parameter values for this experiment were:

$$C = 0.25, 0.5, 2$$

$$max\_iter = 500, 1000, 10000$$

The obtained results can be seen on the table below:

Table 3: Comparison of training and testing performance for the Gaussian SVM with different parameters

C	max_iter	Training Accuracy (%)	Testing Accuracy (%)
0.25	500	33.2	37.3
0.25	1000	70.5	68.0
0.25	10000	70.9	66.1
0.5	500	33.2	37.3
0.5	1000	69.8	65.9
0.5	10000	71.3	65.9
2	500	68.7	64.9
2	1000	37.7	42.2
2	10000	73.4	69.4

From the table, we can see that the best training accuracy and testing accuracy is obtained when the C value is 2 and the maximum number of iterations is 10000.

Overall, it appears that in this dataset, using the linear kernel seems to provide better training and testing results. However, it is important to note that one of the primary parameters of the Gaussian SVM is the gamma parameter, which dictates how far the influence of a single sample will reach. Tuning this gamma parameter can lead to a better performance for the Gaussian SVM. In fact, it has been observed that changing the gamma to 0.125 in the default Gaussian SVM leads to a training accuracy of **99.1%** and a testing accuracy of **74.4%**.

2.6 Conclusion

In conclusion, we learned about the eXtreme Gradient Boosting method and the Support Vector Machine and we saw how to use these algorithms using various libraries in Python. We also gained a better understanding of the SVM algorithm by implementing it from scratch.

The coding part of this homework was pretty straightforward and did not have many challenges. The hardest part that took the longest was figuring out how to implement the SVM from scratch.

Reference

"Introduction to Boosted Trees."Xgboost 1.5.1 Documentation, https://xgboost.readthedocs.io/en/stable/tutorials/model.html.

Wu, Ying Nian . "A Note on Machine Learning Methods." Updated March 2020.

Variational Autoencoders. https://www.cs.cmu.edu/∼ bhiksha/courses/deeplearning/Spring.2017/slides/lec12.vae.pdf.

"Part 3: Intro to Policy Optimization." Spinning Up Documentation, https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html.

Gan Foundations. https://www.cs.toronto.edu/∼duvenaud/courses/csc2541/slides/gan-foundations.pdf.

Schapire, Robert E. "The strength of weak learnability." Machine learning 5.2 (1990): 197-227.

Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016.

Mahto, Krishna Kumar. "Demystifying Maths of Gradient Boosting." Medium, Towards Data Science, 25 Feb. 2019, https://towardsdatascience.com/demystifying-maths-of-gradient-boosting-bd5715e82b7c.

Grover, Prince. "Gradient Boosting from Scratch." Medium, ML Review, 1 Aug. 2019, https://blog.mlreview.com/gradient-boosting-from-scratch-1e317ae4587d.

Berwick, Robert. "An Idiot's guide to Support vector machines (SVMs)." Retrieved on October 21 (2003): 2011.