國立清華大學電機資訊學院資訊工程研究所

碩士論文

Department of Computer Science
College of Electrical Engineering and Computer Science
National Tsing Hua University

Master Thesis

使用感興趣片段優化之遊戲實況服務

Optimizing Live Game Streaming Platforms Using Segment-of-Interests

范姜陶亞
Tao-Ya Fan Chiang

學號：103062586
Student ID:103062586

指導教授：徐正炘 博士
Advisor: Cheng-Hsin Hsu, Ph.D.

中華民國 105 年 07 月
July, 2016

國立清華大學
資訊工程研究所

碩士論文

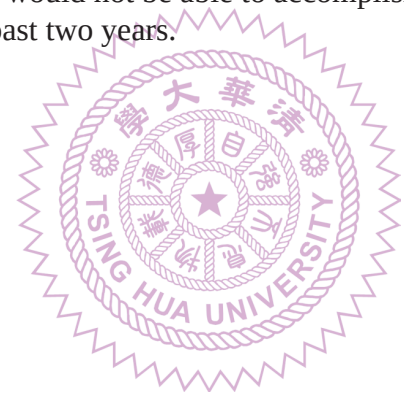
使用感興趣片段優化之遊戲實況服務

撰 范姜陶亞

105
07

Acknowledgments

I would like to express my gratitube toward all the people who helped me in the past two years. I wouldn't be able to finish my thesis were it not for your help along the way. I want to thank my parents specifically, for it is they who provided me with the firm support and stand behind my decisions. I would also like to thank my labmates in Networking and Multimedia Systems Laboratory, especially Hua-Jun Hong, who helped me a great deal in the course of my research. Lastly, I would like to express my gratitude toward my adviser: Prof Cheng-Hsin Hsu. Without the guidance and the suggestion I received from him, I would not be able to accomplish what I have done and learn so much in the past two years.



致謝

在此我要感謝在過去兩年中所有幫助過我的人， 如果沒有你們的幫助我一定沒有辦法順利完成我的論文。 在此我要特別感謝我的父母，他們提供我堅定不移的支持， 同時也支持我所作的每一個決定。我也要感謝網路與多媒體系統實驗室的同學們，特別是洪華駿在過去兩年的研究中幫助我非常的多。 最後，我要感謝我的指導教授：徐正炘教授。如果沒有他的給予我的指導以及建議， 在過去的兩年內我一定沒辦法完成如此多的事情以及學到如此多的東西。

中文摘要

近年來遊戲實況串流平台大為流行，然而從近來的研究中顯示這一些平台需要使用大量的 網路流量，間接的導致不易提高使用者體驗。在這一篇論文中我們提出了使用『感興趣片段』之概念的遊戲實況串流平台， 並以此概念對平台進行優化。 我們的平台使用從實況主以及觀眾電腦中收集來的特徵， 結合當前最優秀之機器學習模型來自動化的偵測 目前實況影片的片段是不是會吸引觀眾的注意。 在決定了當前實況影片片段的重要性之後， 系統中可以使用的頻寬會以一個利用『資料率失真理論』進行最佳化的方式來分配給感興趣的觀眾， 其中會使用影片的片段重要性來作為要傳輸的影片品質依據。 在這個系統運作背後的理念為：唯有在觀眾正在注意實況影片時，才會發生降低使用者體驗的情況。 在我們的使用從現實世界中收集來的數據進行的模擬顯示，在使用10次交叉驗證的條件下， 我們的SoI偵測演算法在使用分類器偵測『感興趣片段』的F度量高達0.96，在使用迴歸器下的平方度量高達0.87。 而在針對資源分配的模擬實驗中我們的資源分配演算法可以達到：(i)增進影片品質高達 5 dB， (ii) 最高可以節省 50 Gbps的頻寬使用率，以及(iii)該演算法可以有效率的完成資源分配決定， 並且可以承受巨量的使用者。 我們在本篇論文中介紹的平台是一個開源平台， 可以被眾多研究者以及工程師使用來改進現有的遊戲實況串流平台。

Abstract

Live game streaming is tremendously popular, and recent reports indicate that such platforms impose high traffic volume, leading to degraded user experience. In this thesis, we propose a Segment-of-Interest (SoI) driven platform, so as to optimize live game streaming. Our platform uses various features collected from streamers and viewers combined with sophisticate algorithms empowered by stat-of-the-art machine learning models to determine if the current segments of gameplays attract viewers. Upon determining the importance of individual segments, the limited bandwidth is allocated to the interested viewers in a Rate-Distortion (R-D) optimized manner, where the levels of segment importance are used as weights of game streaming quality. The underlaying intuition is: viewer experience is degraded only when the game streaming degradation is noticed by viewers. Evaluation results using real world traces shows that our SoI detecting algorithms can correctly detect SoI with up to 0.96 F-measure in classifier variant, and up to 0.87 R-squared score in regressor variant using 10-fold evaluation. Simulation results show the benefits of our proposed resource allocation solution: (i) it improves viewing quality by up to 5 dB, (ii) it saves bandwidth by up to 50 Gbps, and (iii) it efficiently performs resource allocation and scales to many viewers. Our presented testbed is opensource and can be leveraged by researchers and engineers to further improve live game streaming platforms.

Contents

List of Figures

List of Tables


TSING HUA UNIVERSITY

Chapter 1

Introduction

In the recent years, there is a rapid growth in live game streaming business. Corporations such as Twitch.tv, UStreaming thrives in this trend. Among them Twitch is reported to be the most successful one. In 2014, Twitch is the 4^{th} largest traffic on the Internet during peak hour in United States [31], trailing behind Netflix, Google and Apple, 43% of the live video streaming traffic volume is produced by Twitch service [2]. According to Twitch official blog, in 2013 and 2014 the monthly unique viewer number doubled in each year, which reach over 100 million in 2014 [33, 34]. In 2015, Twitch have over 550,000 concurrent viewer on average, with over 2 million peak concurrent viewers and over 30000 concurrent broadcasters running live on the system [35]. Twitch is also the de-facto standard broadcast platform for numerous E-sport tournament and charity event. Seeing the potential and rapid growth of live game streaming, Amazon acquired Twitch for 970 million in 2014 [2], Youtube also launched it's own live game streaming service in August, 2015 [38].

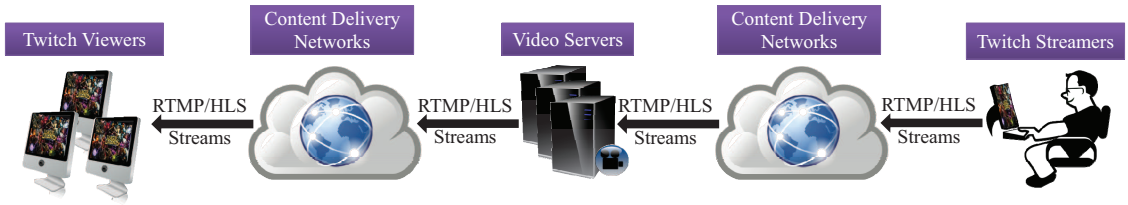


Figure 1.1: A typical live game streaming platform, using Twitch for illustrations.

This section gives a general idea on how live game streaming works. Fig. 1.1 illustrates the overall architecture of typical live game streaming platforms, such as Twitch. Twitch streamers use broadcasting software, such as Open Broadcast Software (OBS) [22], to capture the game screens, and send the videos to servers. The videos are sent using off-the-shelf protocols, like Real Time Messaging Protocol (RTMP) [25] and HTTP Live

Streaming (HLS) [11] protocol. Videos from webcams of streamers can also be sent using picture-in-picture effects, so that viewers can better interact with streamers. Video servers may transcode the videos before streaming them to the viewers. Viewers use streaming clients, native or web-based, to render live game streams. Given the time sensitive nature of live game streaming, Twitch employ several Content Delivery Networks (CDNs) for lower latency and higher bandwidth.

While live game streaming services have seen a steady growth, there are still problems lying in the current state-of-the-art approach that needs to be addressed. Live game streaming itself consumes a considerable amount of resource, Twitch uses over 1.5 Tbps at peak hours with over 1 Tbps average bandwidth usage, reported by Pires et al [24]. According a work by Zhang et al [39], the viewers may perceived latency up to 12 seconds, which hampers the sense of interactivity between streamer and viewers. In order to maintain a reasonable user experience, Twitch doubled it's edge capacity in CDN in Europe region [34], Twitch also opened up new transcoding clusters to provide video with different quality to viewers [32]. These expansion would hurt the profit margin for the company, which is bad for the business. These sign clearly shows that the current implementation of live game streaming system have rooms for improvements.

Based on a critical observation: different *segments* of each live game stream have different importance to viewers, we proposed the concept called *Segment of Interest (SoI)*. By a *segment*, we refer to a continuous time period in a live game streaming video. Fig. 1.2 depict the life cycle of a live game streaming *session*, which refer to the continuous time between the start and end of a stream from a streamer. Each stream is consist of one or multiple *gameplays*, which refers to the continuous time of a match or a round of game. At the beginning of each gameplay, the streamer sets up gears, waits to join a server, and stuck at the loading screen when the level is still loading. Viewers probably would not be upset if the game stream quality is degraded at this time. In other words, *viewer experience is degraded only when the game streaming quality degradation is noticed by viewers*. If we lower the video quality when viewers are not paying attention, we save resources and may support more viewers at better video quality. Therefore, we can *save bandwidth without degrading viewer experience*.

In this work we aim to solve the following problems:

- How to efficiently detect SoI in a live game streaming environment?
- How to dynamically allocate resources between ongoing streams?

using the concept of SoI-driven streaming,

In this work we developed efficient SoI detecting algorithms empowered by stat-of-the-art machine learning models Random Forest and Gradient Boosting Tree. The algo-

Figure 1.2: Life cycle of a live game streaming session.

rithms leverage features that can be collected, process and used in prediction in a real-time fashion, which is crucial for a live game streaming platform. We also developed an SoI driven resource allocator, which allocate resources among different streams currently on the system in a Rate-Distortion optimized manner, with the SoI as the weights of streaming quality. We then speed up the resource allocator by developing a real-time version of the resource allocating algorithm with a controllable error, which would be useful in a practical scenario.

In the evaluation of our system, we achieved up to 0.96 in terms of F-measure using classifier variant of our SoI detector and up to 0.87 in terms of R-squared score in the regressor variant. In a simulation using real world SoI traces collected on our platform, the result shows that our resource allocating algorithm outperforms the state-of-the-art by up to 5 dB in PSNR metric in terms of view quality, and up to 50 Gbps reduction in terms of bandwidth usage. While in the same time complete the resource allocating under 2 ms when there are 256 streams with over 100+ thousands viewers in the system.

The rest of this thesis is organized as follows. Sec. 2 presents the system architecture we propose in this work. Sec. 3 describe the research problems we solved in our work. Sec. 4 and Sec. 5 describe the design of the core component of our system in detail. Sec. 6 presents the implementation of our live game streaming testbed and the methodology used in collecting the data. This is followed by the experiment setup and discussion of results in Sec. 7. We survey the related work in the literature in Sec. 8. Sec. 9 concludes this paper and discussed the future work.

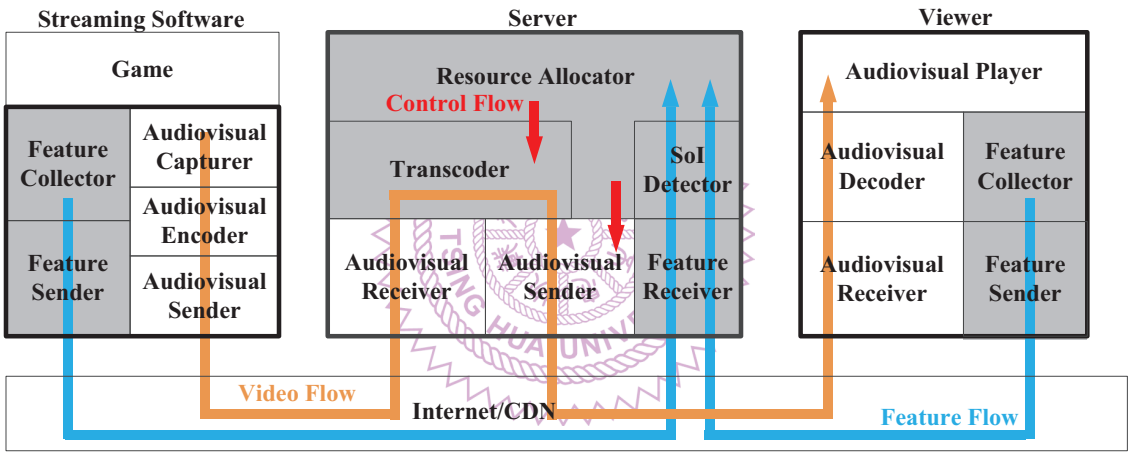


Figure 1.3: The architecture of our proposed live game streaming platform.

Chapter 2

Proposed Architecture

In this chapter we give an overview of the architecture of our proposed system and introduce the component in the system. Live game streaming platforms contain three major software components: streaming software, server, and viewer, which are summarized in Fig. 1.3 and detailed below.

- Streaming software runs on the streamer's computer along with the game itself. The streaming software captures game screens, and sends them to the server. The streaming software consists of audiovisual capturer, encoder, and sender.
- Server relays the game streams to viewers, and performs transcoding if necessary. The server consists of audiovisual sender, receiver, and transcoder.
- Viewer receives the live game stream from the server, and playbacks audio/video. The viewer consists of audiovisual receiver, decoder, and player.

As emphasized in Fig. 1.3, there are five unique components for SoI driven live game streaming: feature collector, feature sender/receiver, SoI detector, , resource allocator and Adaptive transcoder. We present them below.

- Feature collector collects features, such as CPU utilization, GPU utilization, keyboard/mouse inputs, and webcam images from both streamer and viewers' computers.
- Feature sender/receiver transfers captured features from streamers and viewers to the server.
- SoI detector leverages the collected features to determine whether the stream is in the middle of an SoI or not.
- Resource allocator uses the output of SoI detector to allocate resources, so as to optimize the overall performance under resource constraints.

- Adaptive transcoder perform transcoding on the stream video according to the decisions given by resource allocator.

We next use several usage scenarios to illustrate how the proposed live game streaming platform works. When a streamer starts streaming, the rendered screens are captured by audiovisual capturer, sent to audiovisual encoder, and then streamed to the server by audiovisual sender. The feature collectors collect several features, and send the collected feature to the server using the feature sender.

When the audiovisual receiver in the server receives game streams, the server passes the video to transcoder and waits for requests from viewers. The feature received by feature receiver is sent to SoI detector to determine whether the stream is in the middle of an SoI. The result is sent to the resource allocator for optimally allocating resources among all viewers. In particular, the resource allocator decides the coding parameter of each viewer depending on whether the stream is in an SoI or not and the current status of the whole system. Last, the resource allocator instructs the audiovisual sender to send the transcoded stream.

When the audiovisual receiver in the viewer gets the game stream, it passes the game stream to audiovisual decoder and then audiovisual player for display. The feature collector collects the features, and sends them to the server using feature sender. In the server, the features are passed to the resource allocator.

Chapter 3

Research Problems

In the chapter we describe and formulate the research problems we solve in this thesis, namely SoI detecting problem and resource allocating problem.

Figure 3.1: Interactions among the core components of the server.

Table 3.1: Symbols Used Throughout This Paper

Symbol	Description
S	Number of stream
s	Index of stream
l_s	Length video recorded from stream s
t	Current timestamp
V_s	Number of viewer in stream s
v	Index of viewer
R	Outbound bandwidth of the server
$\mathbf{F}_s$	Set of feature collected from streamer s
b_s	Streaming bitrate feature for streamer s
k_s	Mouse/key event feature for streamer s
p_s	Face presence feature for streamer s
u_s^C	CPU usage feature for streamer s
u_s^G	GPU usage feature for streamer s

n_s	Foreground window name feature for streamer
$n\prime_s$	Boolean indicator of whether the game is in foreground window for streamer s
t_s	Tag suggesting that the current segment is in SoI given by streamer s
$\mathbf{F}_{v,s}$	Set of features collected from viewer v of streamer s
$n_{v,s}$	Foreground window name feature for viewer v of streamer s
$p_{v,s}$	Face presence feature for viewer v of streamer s
$t_{v,s}$	Tag suggesting that the current segment is in SoI given by viewer v of streamer s
$\hat{w_{s,R}}$	SoI weight result from SoIR algorithms for stream s
$\hat{w_{s,C}}$	SoI weight result from SoIC algorithms for stream s
w_s	SoI weight used in resource allocator for stream s
α	Index of stream videos used in evaluation
β	Index of timestamp in stream video
$\hat{t_{v,\alpha,\beta}}$	Tag suggesting β second in α video is in SoI
$\hat{V_\alpha}$	Number of viewers that have marked video α
G_α	Aggregated array of all $\hat{t_{v,\alpha,\beta}}$ for video α ground truth
$G_{\alpha,R}$	Ground truth for video α in SoIR problem
$g_{\alpha,\beta,R}$	Ground truth for video α at β second in SoIR problem
$G_{\alpha,C}$	Ground truth for video α in SoIC problem
$g_{\alpha,\beta,C}$	Ground truth for video α at β second in SOIC problem
D_R	Dataset for SoIR problem
D_C	Dataset for SoIC problem
$D_{e,R}$	Evaluation dataset for SoIR problem
$D_{e,C}$	Evaluation dataset for SOIC problem
$D_{t,R}$	Training dataset for SoIR problem
$D_{t,C}$	Training dataset for SoIC problem
$\hat{D_{t,R}}$	Actual training dataset for SoIR problem used in 10-fold cross validation
$\hat{D_{t,C}}$	Actual training dataset for SoIC problem used in 10-fold cross validation
$D_{v,R}$	Validation dataset for SoIR problem used in 10-fold cross validation
$D_{v,C}$	Validation dataset for SoIC problem used in 10-fold cross validation
N	Hyperparameter: number of trees
H	Hyperparameter: maximum depth for trees
X	Hyperparameter: maximum number of features for splitting
E	Hyperparameter: shrinkage

M	Hyperparameter: subsample rate
i_s	Video segment from stream s
o_s	Output video segment for stream s
q_s	Video quality for stream s
r_s	Available bandwidth for each viewer of stream s
$D_{0,s}, R_{0,s}, \theta_{0,s}$	R-D model parameters for stream s
d_s	Distortion for video segment of stream s
λ	Lagrangian multiplier
λ^*	The optimal Lagrangian multiplier
r_s^*	The optimal bandwidth for each viewer of stream s
ϵ	The error between λ and λ^*

3.1 Notations

Fig. 3.1 shows the interactions among the core components of our server. We consider S streamers with V viewers in our system. The total outbound bandwidth of our server is R. From each streamers s at time t, we collect a feature set $\mathbf{F}_s$ with the following features: streaming bitrate b_s, keystroke per second k_s, CPU utilization u_s^C, GPU utilization u_s^G, face presence in webcam $p_s \in \{0, 1\}$, in-game sound amplitude z_s^G, microphone sound amplitude z_s^M, foreground window name n_s, and a tag $t_s \in \{0, 1\}$. The tag t_s is an indicator coming from streamer s suggesting of whether the current segment is an SoI. In the implementation it can be manually marked by streamer s using predefined hotkey, automatically detected by an extra component in streaming software, or even generated by game engine. From each viewer v of streamer s, we collect feature set $\mathbf{F}_{v,s}$, with the following features: foreground window name $n_{v,s}$, face presence in webcam $p_{v,s}$, and a tag $t_{v,s} \in \{0, 1\}$. $t_{v,s}$ is given by viewer v of stream s using either the manual way or automatically detected using extra component in the video player, indicating if he/she is interested in.

In the streaming server there are multiple SoI detectors and transcoders, each is in charge of a streamer s. At t second, the SoI detector periodically receives $\mathbf{F}_s$ and $\mathbf{F}_{v,s}$ from the streamer and viewer. It uses the features to derive the SoI weight $\hat{w_s} \in R^+ \cup \{0\}$ for the current segment from streamer s, where 0 indicates completely uninteresting. The derived $\hat{w_s}$ for each stream s on the system is then sent to the resource allocator, which will be detailed in Sec. 3.3 after necessary post-processing. After resource allocator make the allocation decision for the system, each transcoder receives the encoding bitrate r_s for stream s from the resource allocator. The transcoder then uses r_s to encode the video segments i_s for streamer s, and generates the transcoded video segment o_s, which is sent

by the audiovisual sender. Notice that, for brevity, the considered resource allocator only produces encoding bitrate r_s, while more comprehensive encoding parameters, including frame rate, resolution, and quantization parameter, can also be intelligently chosen by the resource allocator for better user experience.

3.2 SoI Detection

Leveraging features F_s and $F_{v,s}$ periodically collected from streamers s and his/her audience at time t, the SoI detection problem is to determine $\hat{w_s} \in R^+ \cup \{0\}$ with 0 indicate the current segment holds not importance to viewers at all and 1 indicate current video segment is crucial to the viewing experience. $\hat{w_s}$ serves as the decision variable in this problem. This problem is solved independently for each stream s on currently residing in the system. Then $\hat{w_s}$ for all streams are sent to the resource allocator after post-processing into w_s to help it make decisions on how to allocate resource among there streams.

3.3 Resource Allocation

At time t with w_s for all $s = 1, 2, ..., S$, and available bandwidth R, the resource allocation problem is to distribute R among all live game streams s in order to maximize the viewing quality. For concreteness, we use Peak Signal-to-Noise Ratio (PSNR) to quantify the viewing quality q_s of stream s. Nonetheless, our proposed algorithm is general, and can utilize other quality metrics with monotonically increasing (or decreasing) property. We take bitrate r_s of stream s as the decision variables.

Chapter 4

SoI Detector

In this chapter we describe the design of our SoI detector, which detect Segment-of-Interest from the collected features. We also lay out our design philosophy and the way we determine the hyperparameter in the machine learning algorithms we use in our solution.

4.1 Solution Approach

The main goal of SoI detector is to answer the following question: is the segment at time t from streamer s interesting to his/her viewer? The most intuitive approach for SoI detector is to gather the information from viewers. However, either through manually or automatically means, this approach is hard in a practical scenario. Few viewer would be willing to shift their attention from the content of the stream to constantly mark SoI information with high precision. As for automatically detection, it would require massive deployment on heterogeneous devices, and privacy issue may rise if we gather information on viewer devices such as webcam.

In this work we focus detecting SoI from information gathered from streamers. There are already existing research that perform highlight detection using content-based feature [6]. While content-based features have the potential of providing further insight into the context of the game, extracting content-based features are often computationally expensive, which is not suitable for systems that require interactivity such as live game streaming platforms. Therefore we aim to leverage features that can be extracted and process in real time in our solution to detect SoI.

We formulate the SoI detecting problem as a regression problem, with historical data set of SoI information on past video segments, we also developed a classification variant of the module to adapt to different kinds of application. In the rest of this thesis, we will refer the regression variant of the SoI detecting problem as SoIR problem, and refer to the classification variant as SoIC problem. Then we solve the problem using state of

the art machine learning models to design our SoI detecting algorithm. The two learning models we choose are Gradient Boosting Tree (GBT) and Random Forest (RF). These two learning models are both popular and have been applied to various fields with good performance [10]. The two selected learning models have different approach on learning from the given data. GBT uses multiple decision tree as a weak learning model, each aims to perform predict from the residuals of the preceding tree. These trees are then combined using sophisticated weighting scheme to form a single consensus. Random forest also leverage decision trees, but these trees were grew independent of each other using randomly sampled data, the results of these tree are then combined together through means such as majority vote.

We develop our algorithms with the two selected learning models using open source packages [26, 37], these packages provide classifier and regressor version of both the selected learning model. In the rest of this thesis, we refer to the algorithm based on Gradient Boosting Tree regressor as GBTR-based algorithm, and the one based on classifier version as GBTC-based algorithm. Similarly we refer to the algorithm based on Random Forest regressor as RFR-based algorithm, and the one based on classifier version as RFC-based algorithm.

4.2 Dataset

We collected traces from a tournament of famous MOBA game League of Legends in a local event. The reason we choose to capture League of Legends first is that, it has been the most popular games on Twitch since 2013 [7, 33, 35], and by itself alone, take the 29% of the viewer in Twitch platform [7].

In the tournament the game is run on PC equipped with Intel I5-4570 CPU, Intel HD4600 Graphic and 8 GB RAM. The game is captured with OBS 0.653b using X264 encoder. Both the game resolution and capture resolution are set to 1920X1080, with 30 frame per second capture and no resolution downscale. The encoder is set to use variable bitrate (VBR) with 8 as quality balance factor. The maximum bitrate is set to 10 Mbps, note that this limit is much higher than the actual bitrate consumption.

The tournament is consist of 10 match, with each match there are 10 players. We collected 100 traces from all 10 players in each match. After we filtered out the corrupted files, there are 81 valid files with 162841 samples.

We selected one video from each match in the tournament, and recruited 16 viewers to watch the collected videos, and manually mark Segment of Interest tag $\hat{t_{v,s}}$ for us using a modified video player, we then collect these SoI trace files from them. In each SoI trace file, each second of the video is marked with $\hat{t_{v,s}}$ equals to 1 or 0, with 1 means the re-

cruited viewer thinks the current second is in an SoI, 0 means otherwise. We collected 64 of such trace files, which covers 27010 samples in the overall data traces, these traces are used as ground truth in the hyperparameter tuning and the evaluation of our SoI detector module.

4.3 Features

For each samples in the collected data trace, we record the CPU usage, context switch numbers, GPU usage, mouse/keyboard input event, microphone volume, system volume, streaming bitrate, number of faces detected in webcam image and foreground window name. We also modeled the device capability using Novabench [20], which perform benchmarking on the system and give different score in CPU, GPU, RAM and harddrive category. In our collected data GPU usage was not available due to API issue, and face number detected by webcam suffer from low high noise due to ambient lighting in the tournament ground. Therefore GPU usage and number of faces detected were not used in our hyperparameter tuning in Sec. 4.4 and evaluation in Sec. 7.1.

Besides the raw feature numbers we collected except n_s, for each sample we also calculated the minimum, maximum, mean, variance, dynamic range (maximum minus minimum) based on the historic data of this ongoing stream alongside the moving average with window size of 5 seconds. We also compare the foreground window name n_s with the name of the current game being played, then we generate a $n\prime_s \in 0, 1$, with $n\prime_s = 1$ if the game window is the foreground window, and $n\prime_s = 0$ if otherwise. Then all the features including the numbers reported from the Novabench benchmark are concatenated into a single sample with 60 in length.

For the SoI trace we collected from the recruited viewers, we processed them in two different ways according to the variant of the SoI detector. In each stream video α with a duration of l_α seconds, assume there are $\hat{V_\alpha}$ viewers who marked the video. Each viewer $v \in \hat{V_\alpha}$ will mark the video with l_α tag $\hat{t_{v,\alpha,\beta}} = 1$, if viewer v thinks the video segment of β second is in an SoI, otherwise $\hat{t_{v,\alpha,\beta}} = 0$. For each video, all the $\hat{t_{v,\alpha,\beta}}$ are then aggregated together by summing up all the $\hat{t_{v,\alpha,\beta}}, \forall v \in \hat{V_\alpha}$ for each second β to form an array G_α consists of l_α integers.

In the SoIR problem, we take the mean of $\hat{t_{v,\alpha,\beta}}$ collected from $\hat{V_\alpha}$ viewer to generate the ground truth $G_{\alpha,R}$ for video α, i.e., second β in video α have $g_{\alpha,\beta,R} = \frac{\sum_{v \in \hat{V_\alpha}} \hat{t_{v,\alpha,\beta}}}{\hat{V_\alpha}}$ as the SoI ground truth, with $g_{\alpha,\beta,R} \in R^+ \cup \{0\}$. As for the SoIC problem, we perform majority vote among the $\hat{V_\alpha}$ viewers to form consensus on whether each second β in the video is in SoI or not, i.e., for each $g_{\alpha,\beta,C} \in G_{\alpha,C}$, $g_{\alpha,\beta,C} = 1$, if $\sum_{v \in \hat{V_\alpha}} \hat{t_{v,\alpha,\beta}} \geq \frac{\hat{V_\alpha}}{2}$.

The result $G_{\alpha,R}$ and $G_{\alpha,C}$ are then concatenated with the 60-column samples to form

the data D_α from video α. We then concatenate all the D_α together to form the data set D_R and D_C, with $G_{\alpha,R}$ and $G_{\alpha,C}$ serve as the ground truth.

4.4 Optimal Hyperparameter

Hyperparameters refer to the parameters that can not be learned during the training phase by the machine learning model. Therefore they need to be chosen manually beforehand. To find the optimal hyperparameters, we conduct 10-fold cross-validation to find out the optimal hyperparameter using grid search technique for GBTR-based, GBTC-based, RFR-based and RFC-based algorithm.

We first partition 10% of the data in dataset D_R, D_C into evaluation dataset $D_{e,R}$, $D_{e,C}$ and the rest 90% of the data into training dataset $D_{t,R}$ and $D_{t,C}$. The data in $D_{e,R}$ and $D_{e,C}$ will be used in the evaluation of the performance of the algorithm in Sec. 7.1. The rest of the data in training dataset are then used in 10-fold cross validation to derive the optimal hyperparameter set. In 10-fold cross validation, the data in the training dataset $D_{t,R}$ are further divided into 10 equal share, and each share take turns to be the validation set $D_{v,R}$ with the rest 9 share being the actual training set $\hat{D_{t,R}}$. The $\hat{D_{t,R}}$ are then used to train model, with the data in $D_{v,R}$ to validation the result. After the 10 rounds of validation, the performance metrics are then averaged then reported as output. The same procedure is also conducted on the hyperparameter training of SoIC problem using $D_{t,C}$, with the validation set being $D_{v,C}$ and actual training set be $\hat{D_{t,C}}$. In our dataset, the size of $D_{v,R} = D_{v,C} = 2076$ and the size of $\hat{D_{t,R}} = \hat{D_{t,C}} = 18677$.

For RFR- and RFC-based algorithm there are three hyperparameters to be chosen: (i) N, the number of trees, (ii) X, maximum number of considered feature in splitting, and (iii) H, maximum depth of the tree. We perform grid search on space where $N = \{30, 60, 120, 240, 480\}$, $X = \{5, 10, 20, 40\}$, $H = \{10, 20, 40, 80, 160\}$ to find out the optimal parameter for RFR- and RFC-based algorithm.

For GBTR- and GBTC-based algorithm there are four hyperparameters to be chosen: (i) N, the number of tree, (ii) E,the shrinkage, which is the learning rate, (iii) H, the maximum depth of each tree, and (iiii) M, the subsample rate. We perform grid search on space where $N = \{5, 10, 20, 40, 80\}$, $E = \{0.01, 0.05, 0.1, 0.2, 0.4\}$, $H = \{5, 10, 20, 40, 80\}$, $M = \{0.5, 0.6, 0.7, 0.8, 0.9\}$ to find out the optimal hyperparameter for GBTR- and GBTC-based algorithm.

To quantify the performance of SoI detection algorithms, we take R-square score and F-measure as the performance metrics in SoIR and SoIC problem respectively. We choose our hyperparameter with these metrics as the optimization objective. In SoIC variant we also calculate precision and recall rate alongside accuracy. We also record the training

time γ in both problems.

We plot the result from 10-fold cross-validation in Figs. 4.1 to 4.14. We pick $N = 120$, $X = 10$, $H = 40$ and $N = 20$, $E = 0.1$, $H = 20$, $M = 0.7$ as the default hyperparameter setting for Random Forest based and Gradient Boosting Tree based algorithms, and plot figures altering one hyperparameter at a time.

Here we make a few observation on the more notable results from the figures, and give basic guidelines on how to choose some of the hyperparameters.

Pick the highest N with acceptable γ for both RF-based and GBT-based algorithms. From Fig. 4.1, Fig. 4.5, Fig. 4.8 and Fig. 4.12 shows that higher N generally achieve better performance at the cost of significantly longer γ, From Fig. 4.8 and Fig. 4.5 we can see the marginal gain of higher N, may drop rapidly once N exceeds a threshold. Therefore we should pick the highest N with acceptable γ.

High E may leads to overfitting in the model for GBT-based algorithms. From Fig. 4.2 we can see that higher E help the algorithm to adapt to the trait of the dataset more rapidly, which would yield better result. However, Fig. 4.9 shows that if the E is too high, it may lead to overfitting in the model and hurt the performance in the end.

Pick a reasonable H using cross validation to avoid overfitting for both RF-based and GBT-based algorithm. The maximum depth of the trees dictate how complex each tree can be, if the H is too low, the trees may be too simple to capture the behavior of the dataset. However, if the H is too high, the model may overfit the $\hat{D}_t$, as shown in Fig. 4.10, when $H = 80$ the performance decreases comparing to $H = 40$. Fig. 4.14 also shows overfitting when $H = 160$.

High X in RF-based algorithms may not help the performance. Fig. 4.13 and Fig. 4.6 shows that high X may hurt the overall performance, while significantly increase the training time for the algorithms.

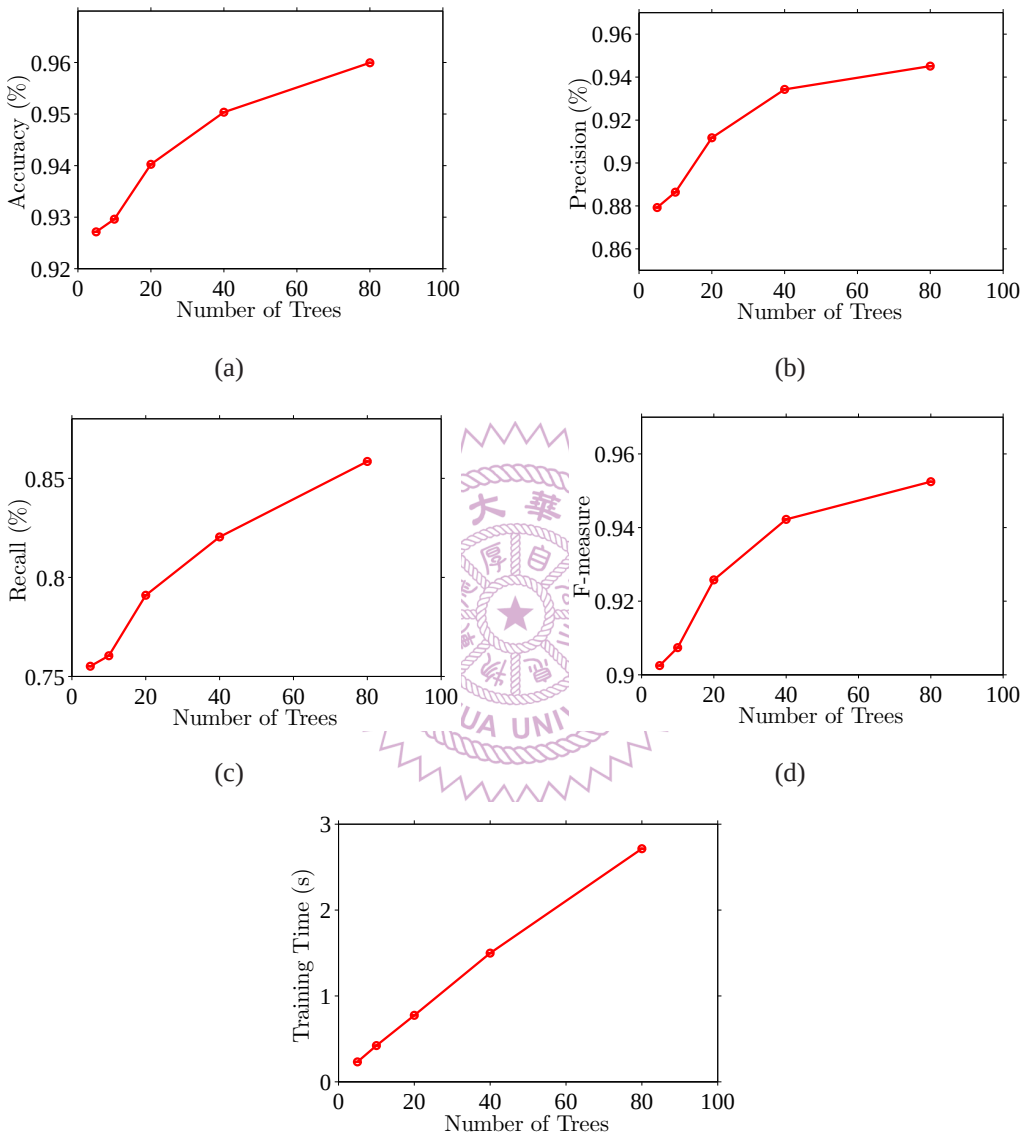


Figure 4.1: Results with different number of tree in GBTC: (a) accuracy, (b) precision rate, (c) recall rate, (d) F-measure, and (e) training time.

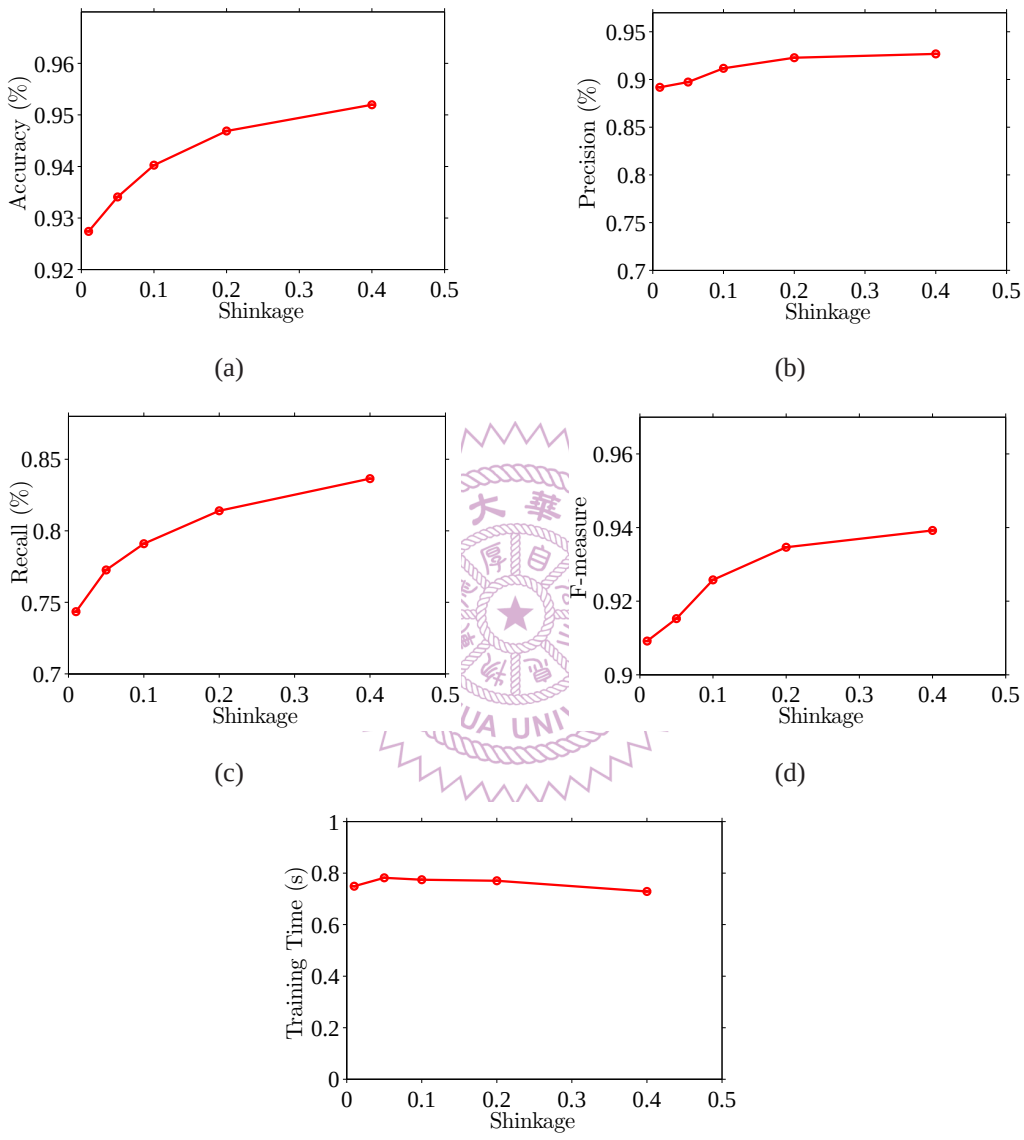


Figure 4.2: Results with different shrinkage in GBTC: (a) accuracy, (b) precision rate, (c) recall rate, (d) F-measure, and (e) training time.

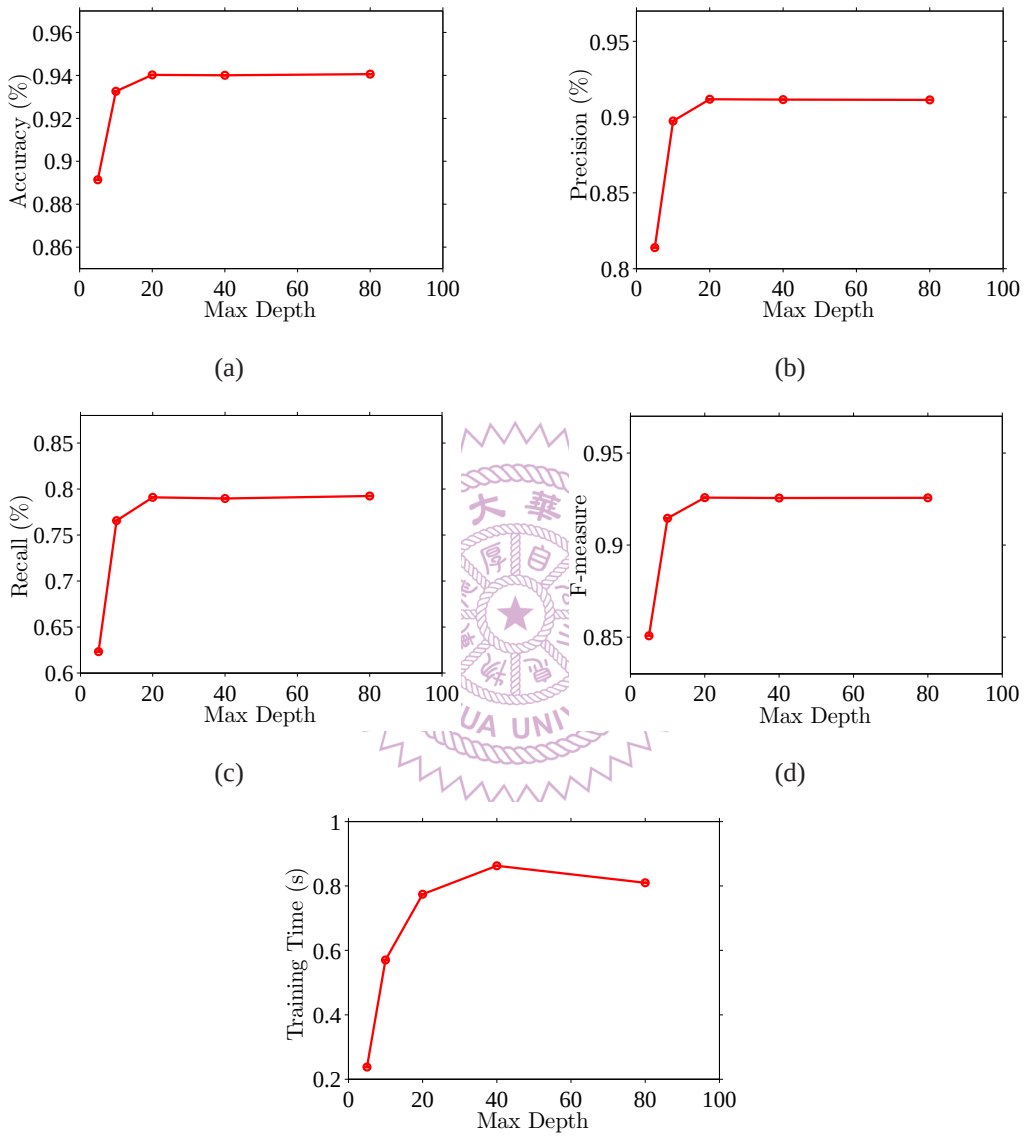


Figure 4.3: Results with different maximum tree depth in GBTC: (a) accuracy, (b) precision rate, (c) recall rate, (d) F-measure, and (e) training time.

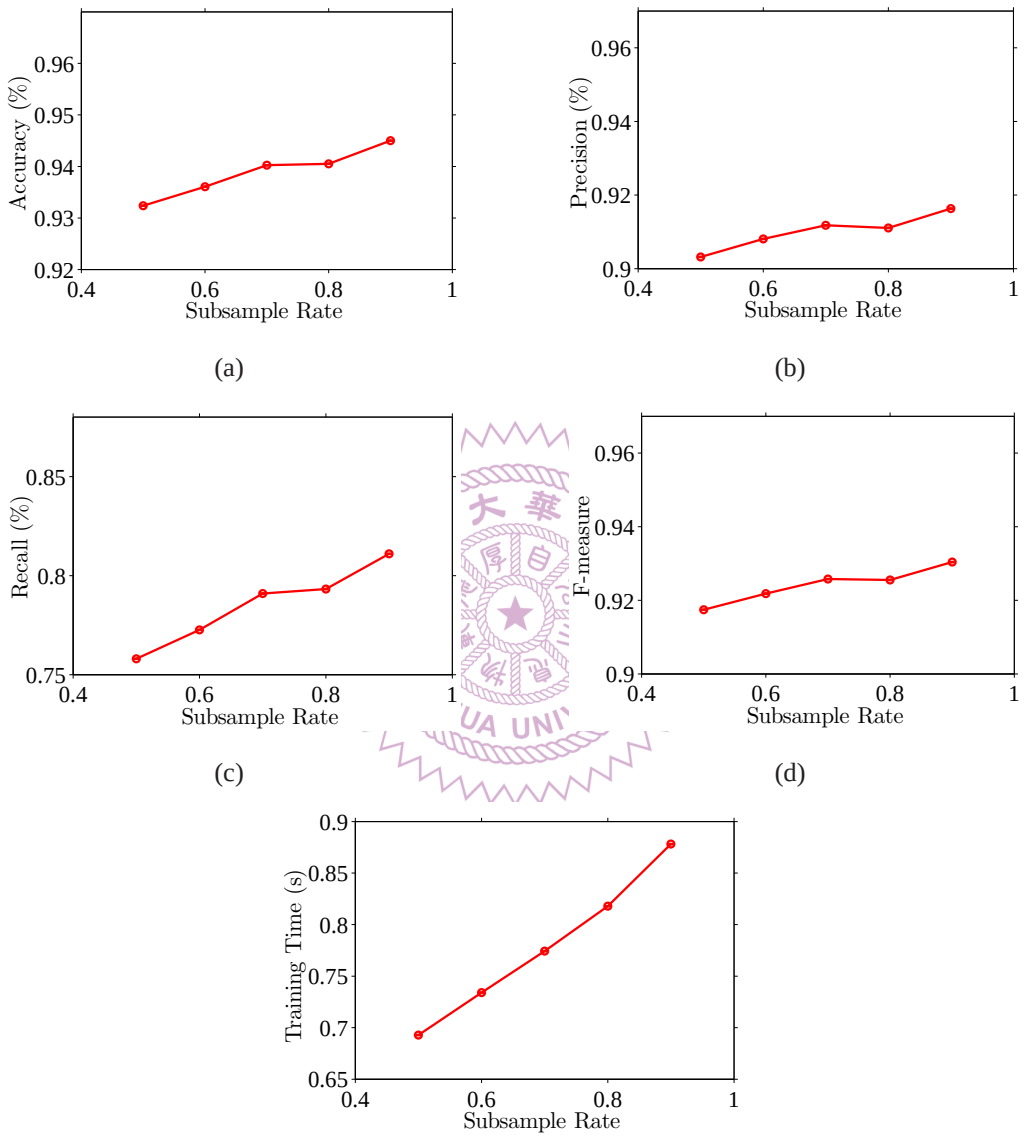


Figure 4.4: Results with different subsample in GBTC: (a) accuracy, (b) precision rate, (c) recall rate, (d) F-measure, and (e) training time.

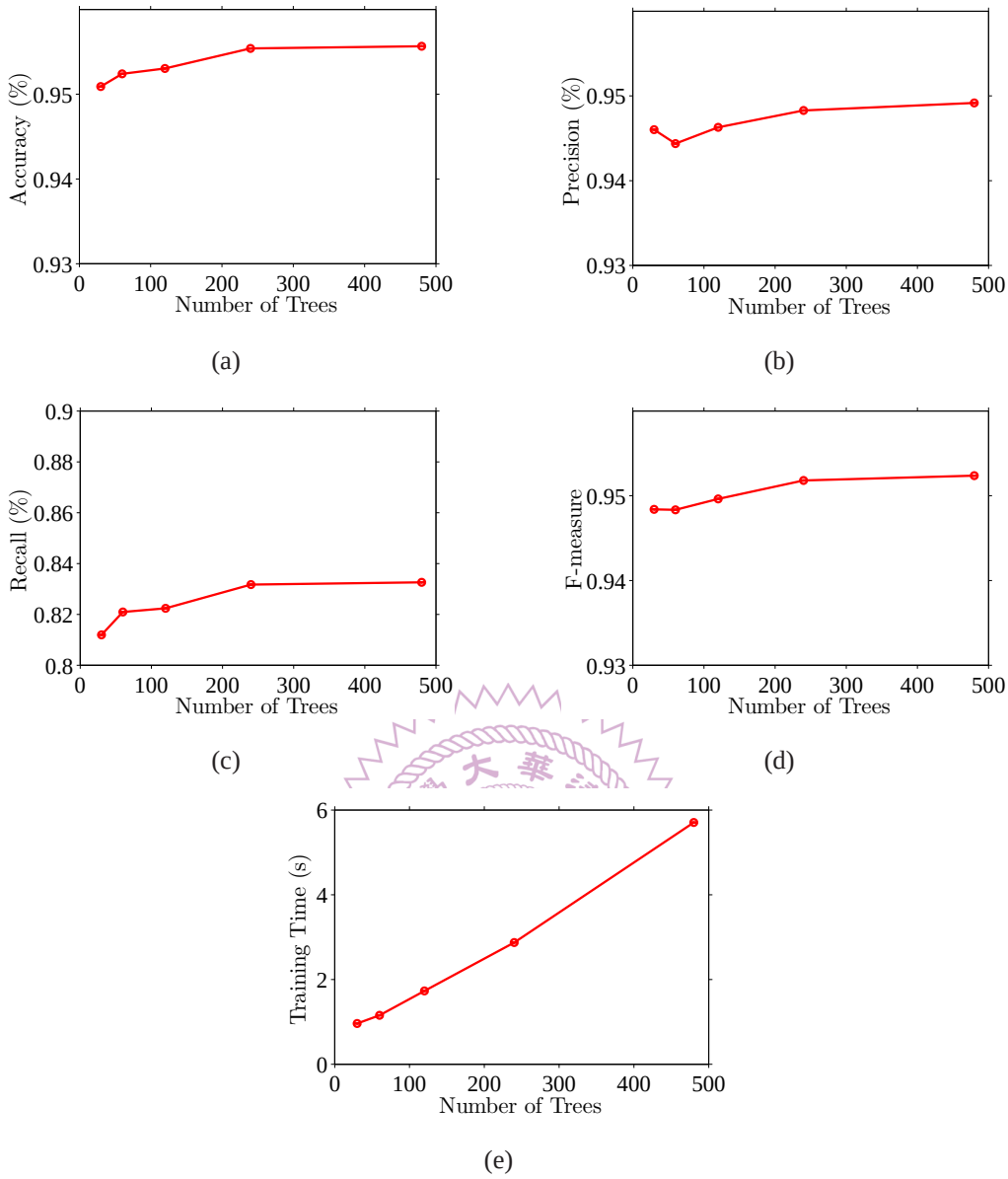


Figure 4.5: Results with different number of tree in RFC: (a) accuracy, (b) precision rate, (c) recall rate, (d) F-measure, and (e) training time.

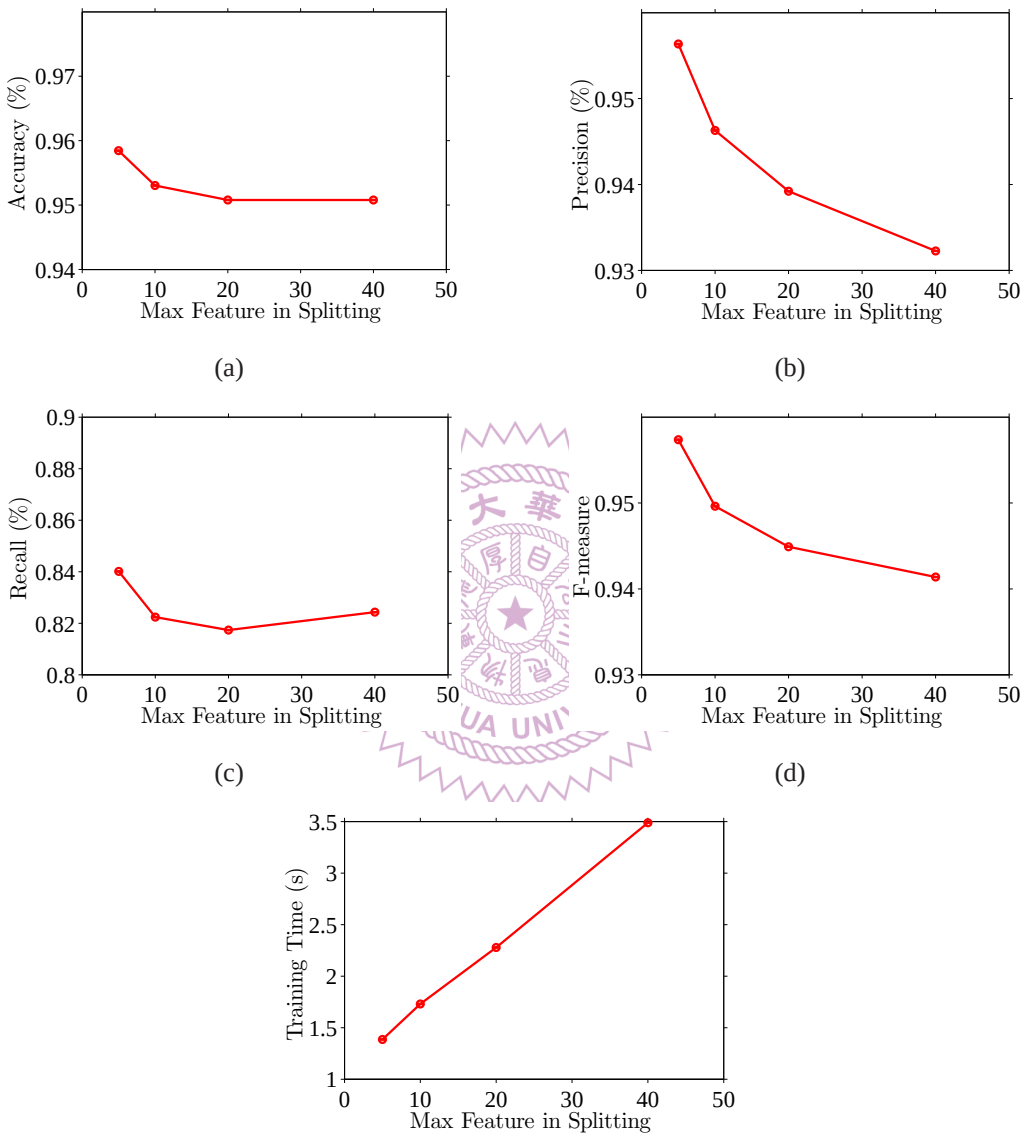


Figure 4.6: Results with different maximum number of feature in splitting in RFC: (a) accuracy, (b) precision rate, (c) recall rate, (d) F-measure, and (e) training time.

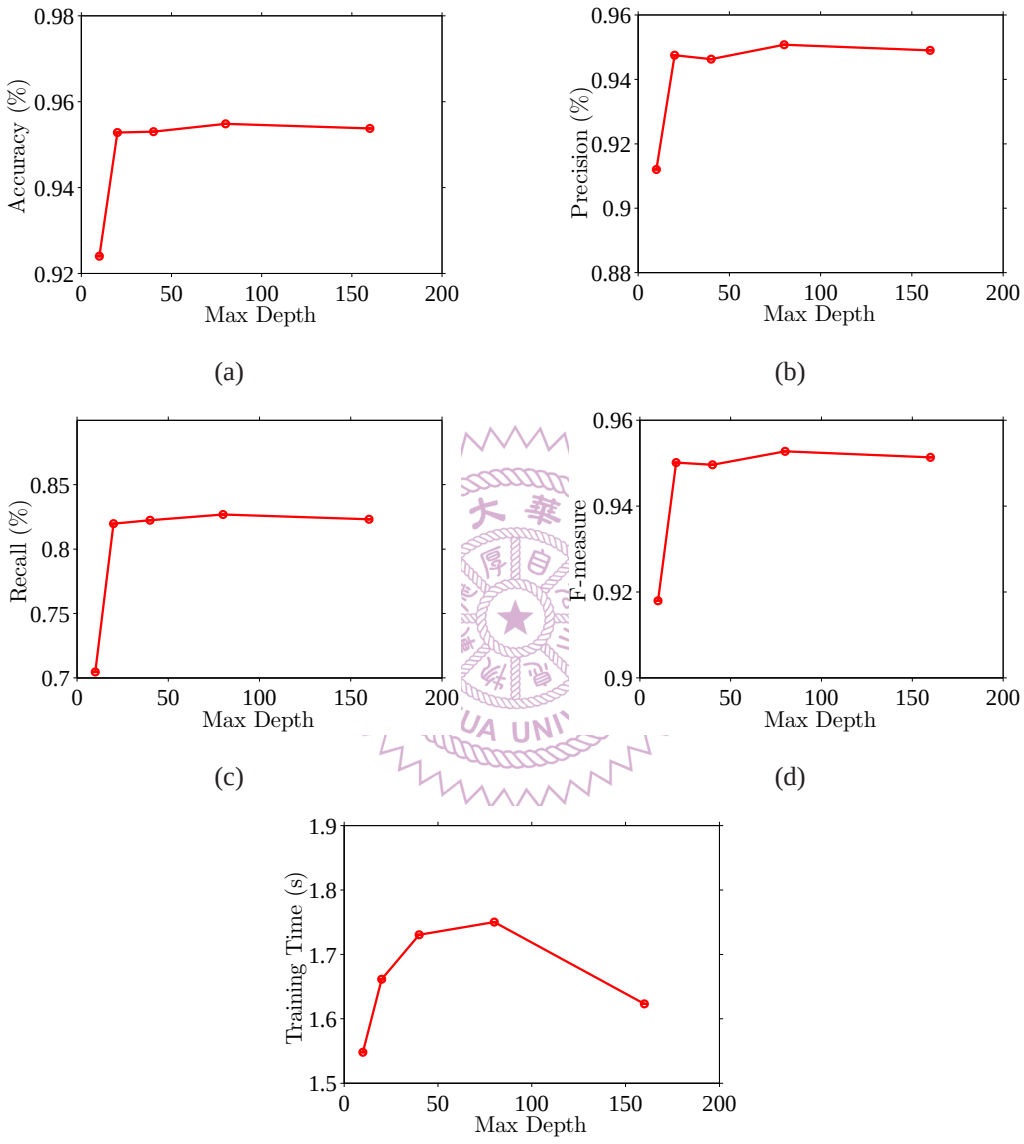
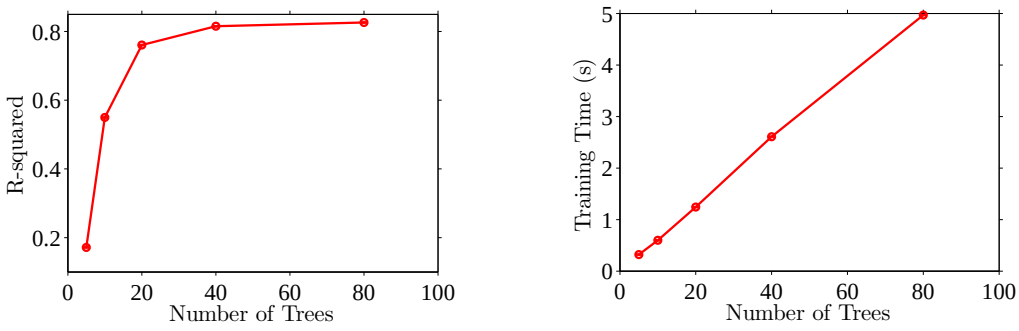
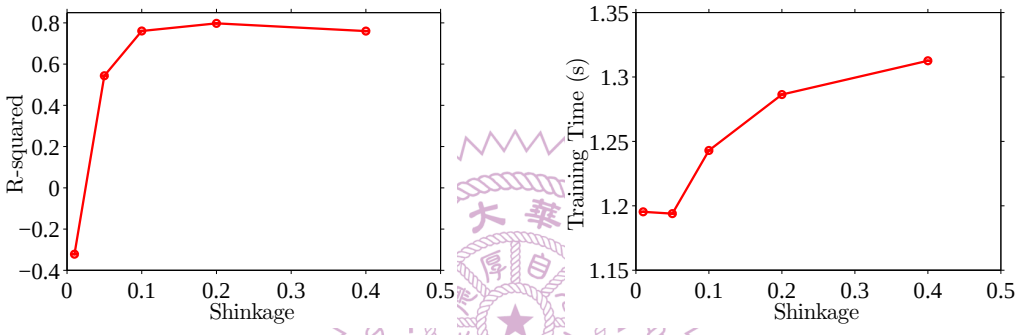


Figure 4.7: Results with different maximum depth in RFC: (a) accuracy, (b) precision rate, (c) recall rate, (d) F-measure, and (e) training time.



(a) (b)

Figure 4.8: Results with different number of tree in GBTR: (a) R-squared, (b) training time.

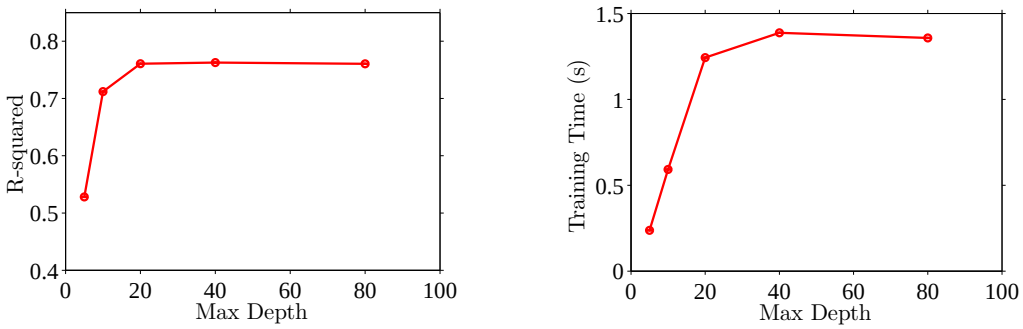


(a) (b)

Figure 4.9: Results with different shrinkage in GBTR: (a) R-squared, (b) training time.

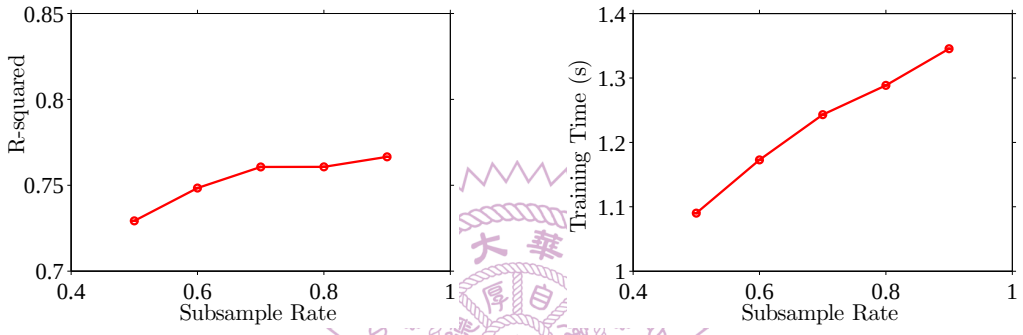
4.5 SoI Detecting Algorithm

From the result of the 10-fold cross validation, we obtain the optimal hyperparameter set. For RFR-based algorithm in SoIR problem, the reported optimal hyperparameter set is when $N = 240$, $X = 5$, $H = 40$. The average training time using this setting is 18.09 seconds, with 0.822 average R-squared score. For GBTR-based algorithm in SoIR problem, the reported hyperparameter set is when $N = 80$, $E = 0.2$, $H = 10$, $M = 0.9$. The average training time is 1.86 seconds, with 0.833 average R-squared score. For RFC-based algorithm in SoIC problem, the reported hyperparameter set is when $N = 480$, $X = 5$, $H = 80$. The average training time for this hyperparameter set is 40.32 seconds, with 0.959 in F-measure score, 0.960 in accuracy rate, 0.958 in precision rate, and 0.844 in recall rate. For GBTC-based algorithm, the reported optimal hyperparameter set is when $N = 80$, $E = 0.2$, $H = 80$, $M = 0.7$. The average training time is 2.45 seconds, with 0.957 in F-measure score, 0.964 in accuracy rate, 0.95 in precision rate, 0.874 in recall rate.



(a) (b)

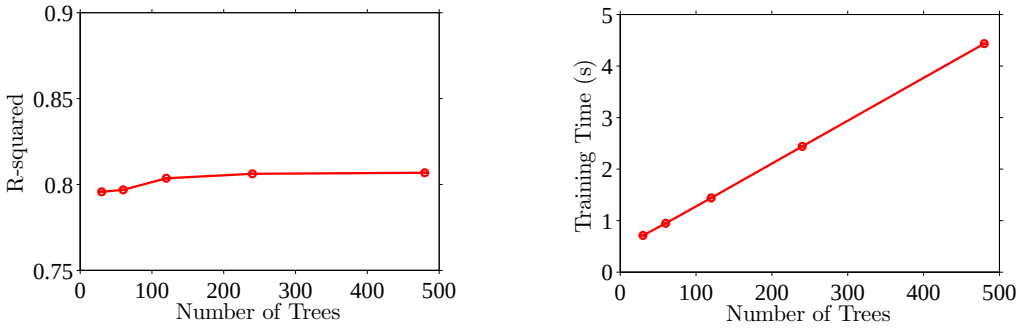
Figure 4.10: Results with different maximum depth in GBTR: (a) R-squared, (b) training time.



(a) (b)

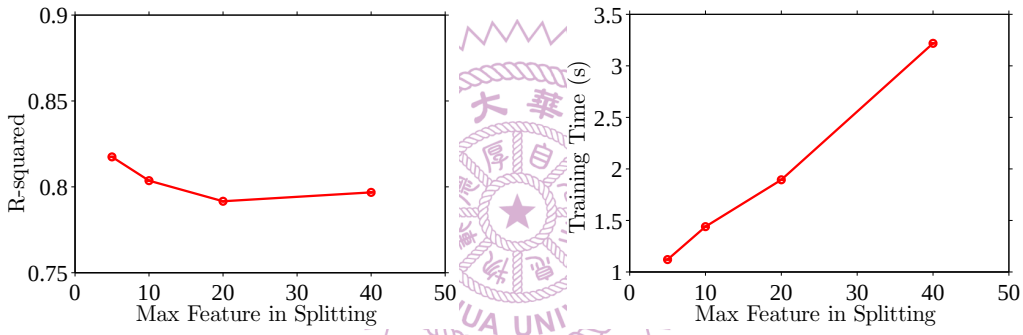
Figure 4.11: Results with different subsample in GBTR: (a) R-squared, (b) training time.

From the result above we make the observation that, for SoIR problem the GBTR-based algorithm have slightly better performance, while in the same time finish the training more efficiently then RFR-based algorithm. This makes GBTR-based algorithm a more viable pick in the two of the algorithms. In the SoIC problem, the GBTC-based algorithm still finished earlier than RFC-based algorithm, however, the comes with the cost of slightly decrease in performance. The evaluation of these algorithms using dataset which have not be exposed to the trained model in the process of cross-validation is conducted in Sec. 7.1.



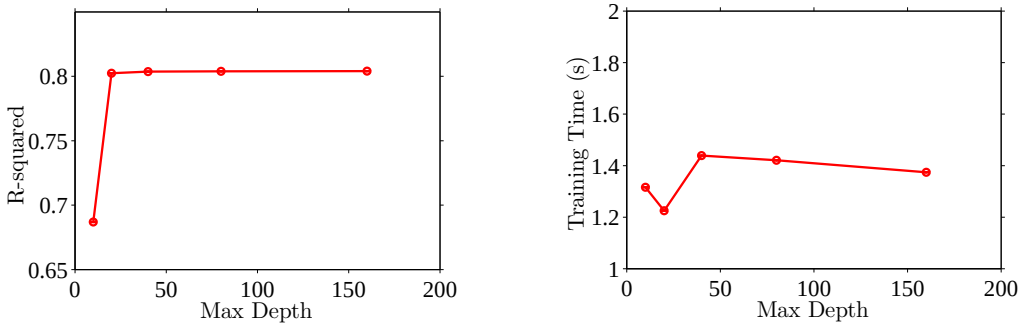
(a) (b)

Figure 4.12: Results with different number of tree in RFR: (a) R-squared, (b) training time.



(a) (b)

Figure 4.13: Results with different maximum number of feature in splitting in RFR: (a) R-squared, (b) training time.



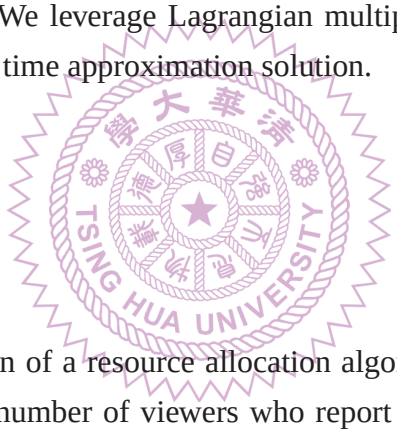
(a) (b)

Figure 4.14: Results with different maximum depth in RFR: (a) R-squared, (b) training time.

Chapter 5

Resource Allocator

In this chapter we present the design of our resource allocator in detail. Resource allocator is responsible for allocating the limited resources among different streams in the system given the SoI information. We leverage Lagrangian multiplier to solve the formulated problem and proposed a real time approximation solution.



5.1 Formulation

We concentrate on the design of a resource allocation algorithm, and adopt a basic SoI detector that sets w_s as the number of viewers who report $t_{v,s} = 1$. That is, w_s keeps track of the number of viewers who think the current segment of stream s is an SoI. We acknowledge that $t_{v,s}$ may not always be provided, and a more comprehensive SoI detector is proposed in Sec. 4. With w_s for all $s = 1, 2, ..., S$, and available bandwidth R, the resource allocation problem is to distribute R among all live game streams s in order to maximize the viewing quality. For concreteness, we use Peak Signal-to-Noise Ratio (PSNR) to quantify the viewing quality q_s of stream s. Nonetheless, our proposed algorithm is general, and can utilize other quality metrics with monotonically increasing (or decreasing) property. We take bitrate r_s of stream s as the decision variables.

To relate r_s with q_s, we adopt a rate-distortion model [42]:

$$d_s = D_{0,s} + \frac{\theta_s}{r_s - R_{0,s}}, \tag{5.1}$$

where d_s is the distortion in Mean Squared Error (MSE), $D_{0,s}$, θ_s, and $R_{0,s}$ are model parameters derived by non-linear regression. We write the problem formulation as:

$$\text{maximize} \sum_{s=1}^{S} q_s w_s = \sum_{s=1}^{S} 10 \log_{10} \frac{255^2}{D_{0,s} + \frac{\theta_s}{r_s - R_{0,s}}} w_s \tag{5.2a}$$

$$\text{s.t.} \sum_{s=1}^{S} r_s w_s \leq R; \tag{5.2b}$$

$$r_s \in R^+, \forall s = 1, 2, ..., S. \tag{5.2c}$$

The objective function in Eq. (5.2a) maximizes the weighted viewing quality, where the weights are number of interested viewers of individual live gaming streams. The constraint in Eq. (5.2b) ensures that we do not incur excessive traffic ($> R$) which may lead to playout glitches.

5.2 Proposed Algorithm

We leverage Lagrangian Multiplier method [30] to derive closed-form formulas as follows. We first introduce a Lagrangian Multiplier λ into the formulation in Eq. (5.2), which leads to the Lagrangian function $\mathscr{L}$:

$$\sum_{s=1}^{S} (10 \log_{10} \frac{255^2}{D_{0,s} + \frac{\theta_s}{r_s - R_{0,s}}} w_s) + \lambda((\sum_{s=1}^{S} r_s w_s) - R). \tag{5.3}$$

To get the extreme value, we take the partial derivatives of $\mathscr{L}$ with respect to λ and r_s, where $s = 1, 2, ..., S$. This leads to:

$$\frac{\partial \mathscr{L}}{\partial \lambda} = (\sum_{s=1}^{S} r_s w_s) - R; \tag{5.4a}$$

$$\frac{\partial \mathscr{L}}{\partial r_s} = \lambda + \frac{10\theta s}{\log 10 (r_s - R_{0,s})^2 (D_{0,s} + \frac{\theta_s}{r_s - R_{0,s}})}. \tag{5.4b}$$

Letting $\frac{\partial \mathscr{L}}{\partial r_s} = 0$, and after some simplifications, we get:

$$r_s = R_{0,s} + \frac{-(\lambda \log 10 \theta_s) - \sqrt{(\lambda \log 10 \theta_s)^2 - 40 \lambda \log 10 \theta_s^2}}{2(\lambda \log 10 D_{0,s})}. \tag{5.5}$$

Last we let $\frac{\partial \mathscr{L}}{\partial \lambda} = 0$ and solve the system of Eqs. (5.4a) and (5.5) for the optimal r_s^*. The resulting bitrate r_s^* for all s are sent to the transcoder to encode the upcoming video frames. One practical concern is the overhead of bitrate adaptation and negative impacts of fast viewing quality fluctuations. We leave the frequency of invoking our proposed algorithm, say T, as a system parameter.

Solving the equation system for r_s^* may take a long time, and we next develop a real-time algorithm that offers a controllable error of ϵ. Algorithm 1 summarizes our efficient algorithm that is based on binary search on the optimal λ^* value, where U and L are the upper and lower bounds of the current λ. For practical reasons, we set the upper

bound $R_U = 8$ Mbps and the lower bound $R_L = 128$ kbps. In line 7, we make sure that r_s remains in $[R_L, R_U]$ and derive valid q_s. Line 9 ensures that the shared link is not overloaded. Line 14 returns the answer after the target error ϵ is met in line 3.

5.3 Analysis

Lemma 1 (Optimality)**.** *Algorithm 1 always finds λ within a gap ϵ to the optimal λ^* under the given bandwidth constraint R.*

Proof. From Eq. (5.1), we see that as the bitrate r_s increases, d_s of the video segment decreases monotonically. Combine this with the definition of PSNR, we see that q_s also increases monotonically. From Eq. (5.4b) we observe that, as the $|\lambda|$ decreases, r_s also increases monotonically under the constraint $r_s > R_{0,s}$. From the above two observation we prove that, as $|\lambda|$ decreases, q_s of every stream s increases monotonically, and the objective function Eq. (5.2a) also increases monotonically. Therefore, the λ returned value approximates the optimal lambda λ^* of the equation system with a worst-case error of ϵ under the constraint that the searched domain does not cross 0. □

Algorithm 1 Efficient SoI RDO Algorithm

1: $U \leftarrow 0$, $L \leftarrow -1$ // Upper/lower bounds
2: $\epsilon \leftarrow 10^{-7}$ // Default error
3: **while** $L + \epsilon < U$ **do**
4: $\quad \lambda = (U + L)/2$
5: $\quad$ **for** $\forall s \in S$ **do**
6: $\quad\quad$ Derive all the r_s using λ and Eq. (5.5)
7: $\quad$ **if** $\exists r_s$, such that $q_s \in \mathbb{C}$ or $r_s < R_L$ or $r_s > R_U$ **then**
8: $\quad\quad$ Adjust U or L accordingly
9: $\quad$ **else if** $\sum_{s \in S} r_s w_s > R$ **then**
10: $\quad\quad U = \lambda$
11: $\quad$ **else**
12: $\quad\quad$ *obj* $= \sum_{s \in S} q_s w_s$
13: $\quad\quad L = \lambda$
14: if *obj* is undefined, return no answer, o.w. return λ and r_s.

Lemma 2 (Complexity)**.** *Algorithm 1 runs in $O(S \log \epsilon^{-1})$.*

Proof. In the worst-case scenario, it is executed $\log(1/\epsilon)$ rounds. In each round, there are S streams, each needs to derive its own r_s, q_s and check if r_s is valid. Each of these operations takes $O(1)$ time. Therefore in each round the calculation and the examination

of all r_s and q_s take $O(S)$ time. We also calculate the consumed bandwidth and the objective value, both operations take $O(S)$ time. So the total time complexity is $(O(S) + O(S) + O(S))O(\log \epsilon^{-1}) = O(S \log \epsilon^{-1})$. □

5.4 Leveraging Features From Viewer

While not in the current solution, here we discuss the potential of leveraging $F_{v,s}$ collected in the system. The $p_{v,s}$ and $n_{v,s}$ can be leveraged to determine whether viewer v is not paying attention to the stream or even not present at the scene. After we obtain the information, resource allocator can then decrease the quality video image sent to viewer v. The allocator could even decide to send only the audio signal to v if the video is not being played in the foreground. This way we can further optimize the resource allocation in the system with a finer granularity

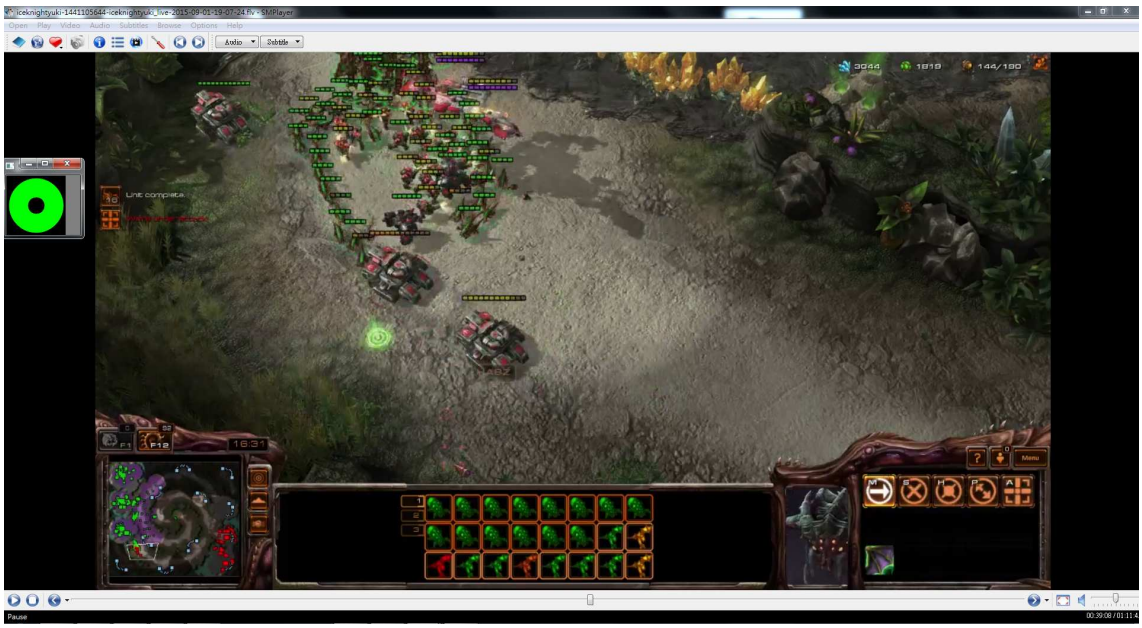


Figure 5.1: Modified SMPlayer is used to mark SoI as ground truth.

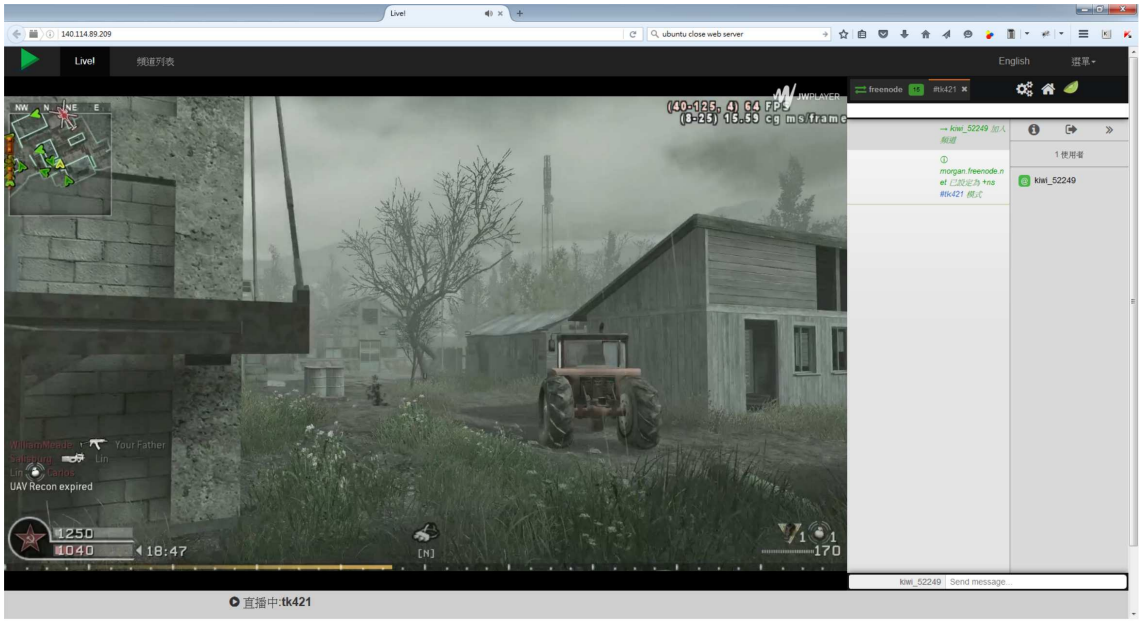


Figure 5.2: The Web interface of the server in our testbed.

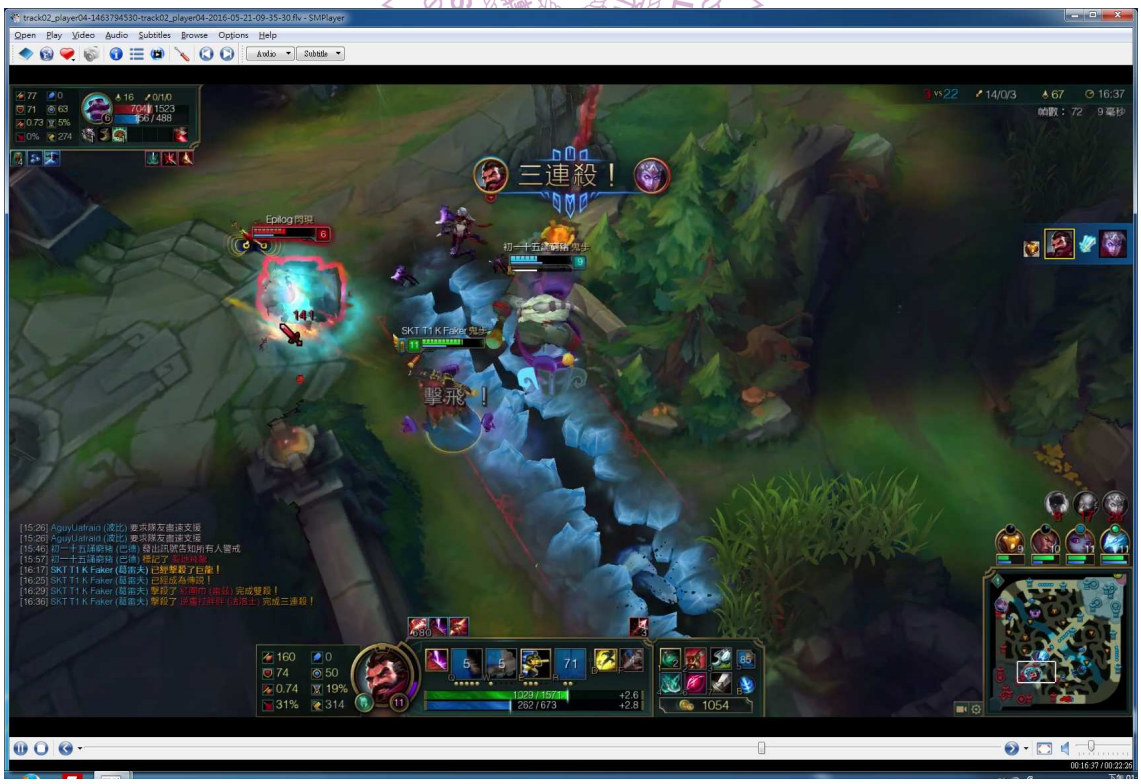


Figure 5.3: Screenshot of the modified SMPlayer.

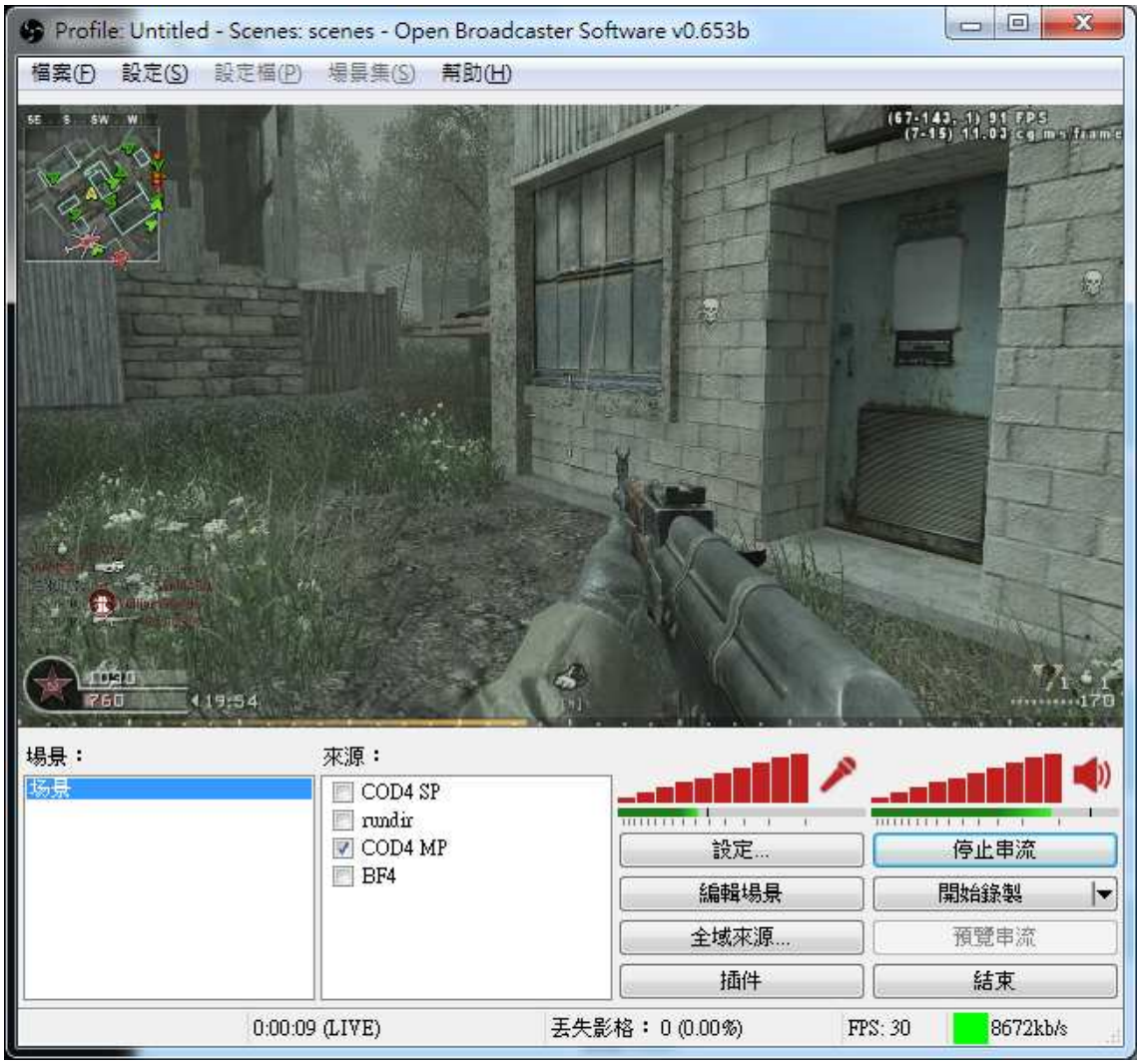


Figure 5.4: Screenshot of the modified OBS.

Chapter 6

An Opensource Testbed

We have implemented a complete live game streaming platform consisting of the streamer, server, and viewers. We enhance OBS [22] to collect features from streamers as in Fig. 5.4. In particular, the modified OBS collects the CPU usage, GPU usage, keystrokes, mouse events, and sound amplitudes (microphone and in-game), streaming bitrate, and webcam face detection results (to determine if the streamer has left). In particular, we save the streaming bitrate reported by OBS. We use Windows Performance Counter [36] API to measure CPU utilization every second. We adopt NVAPI [21] provided by NVidia to measure the GPU utilization on its GeForce graphic cards, we also adopted AMD Display Library (ADL) [3] provided by AMD to measure the GPU utilization on its Radeon graphic cards. We use `GetAsyncKeyState` provided by Windows to capture the keystrokes and mouse events. Note that `GetAsyncKeyState` does not correctly capture mouse clicks in some games, such as League of Legends. Addressing this limitation is one of our future tasks. We use Direct Show (DS) framework to capture the sound amplitudes. We adopt the `VideoIO` module in OpenCV to replace the webcam module of OBS, and perform face detection using Haar object detection algorithms on webcam images. All the collected features are sent to the server, and saved as log files.

We enhance SMPlayer [28] as in Fig. 5.3 to collect features from viewers. The modified SMPlayer captures current foreground window name (to determine if the viewer is watching), and webcam face detection results (to determine if the viewer is gone). For foreground window detection, we use `GetActiveWindowTitle` function provided by Windows. For face detection, we also adopt the `VideoIO` module in OpenCV as in OBS. Fig. 5.1 is a screenshot of the modified SMPlayer, in which the circle button on the left allow viewers to mark if the stream is in SoI; the viewer toggles his/her choice by hitting the TAB key. The collected features, including the SoI (used as the ground truth in Sec. 7) are sent to the server, and saved as log files.

We integrate NGINX [19], JWPlayer, and IRC into our server. NGINX is a web

service that provides RTMP plug-in to relay the streams from streamers to viewers. Fortunately, NGINX provides interface for transcoders, such as FFmpeg [8], which are leveraged in our testbed. To encourage more streamers and viewers to use our platform, we create a unified web interface with flash player (JWPlayer) and chat room (KiwiIRC). Viewers may preview game streaming before installing SMPlayer. Fig. 5.2 is a screenshot of the web interface.

The presented testbed can be used by researchers and engineers, and we solve the SoI detection problem and resource allocation problem using this testbed in Sec. 4 and Sec. 5 as case studies. Given that our platform is composed of opensource projects, we will release our patches and documents to the communities.

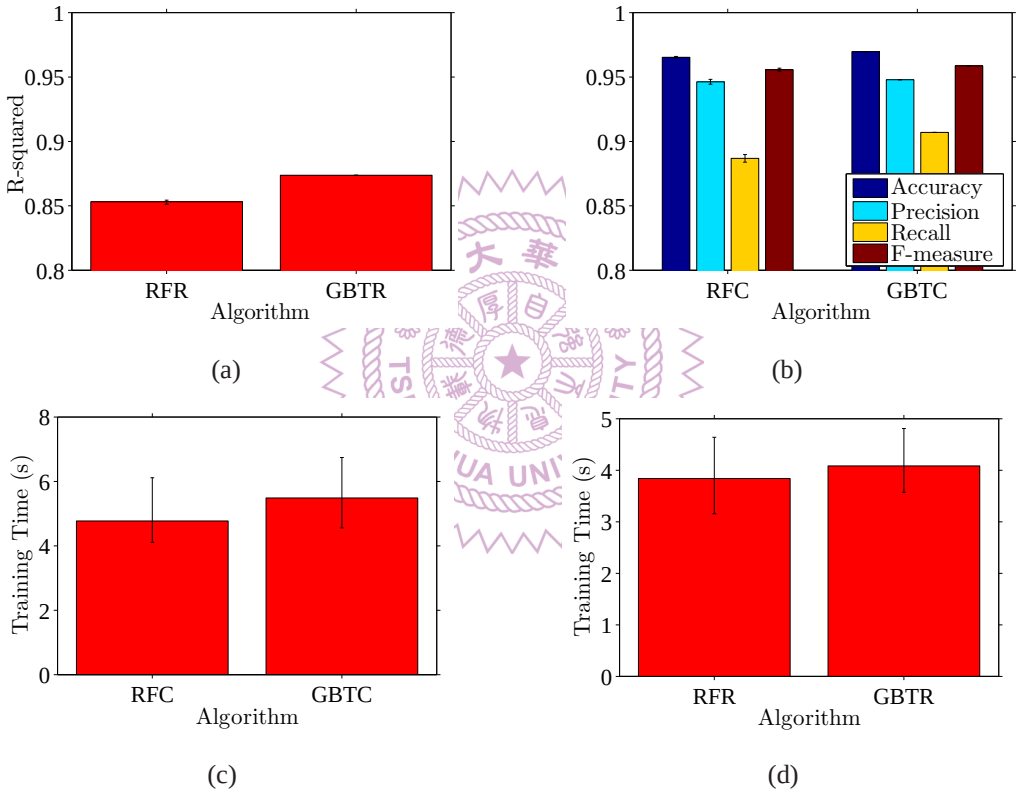


Figure 6.1: Results evaluation on, (a) SoIR, (b) SoIC, and (c) training time of all algorithms.

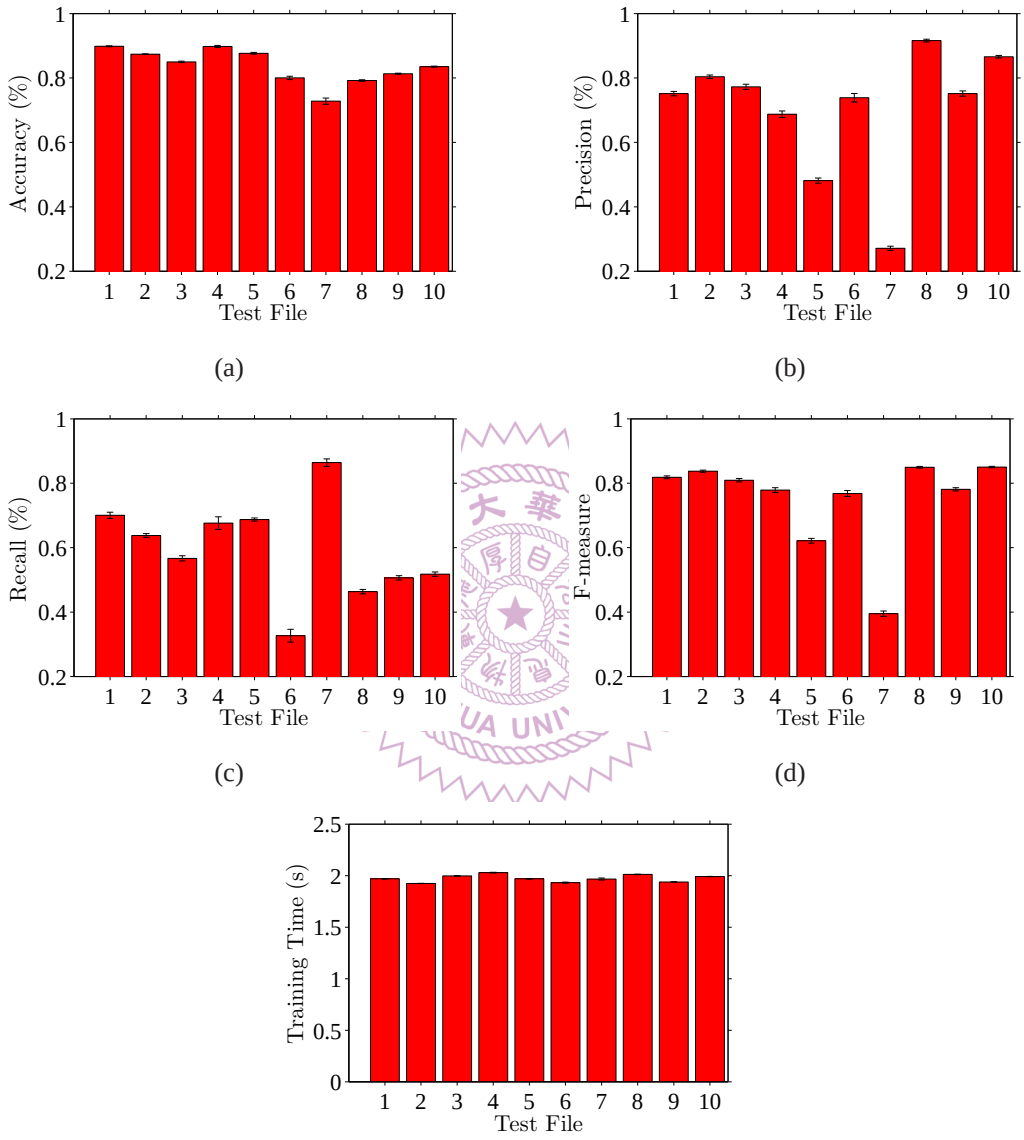


Figure 6.2: Evaluation result of SoIC problem using RFC-based algorithm, (a) accuracy rate, (b) precison rate, (c) recall rate, (d) F-measure and (e) training time.

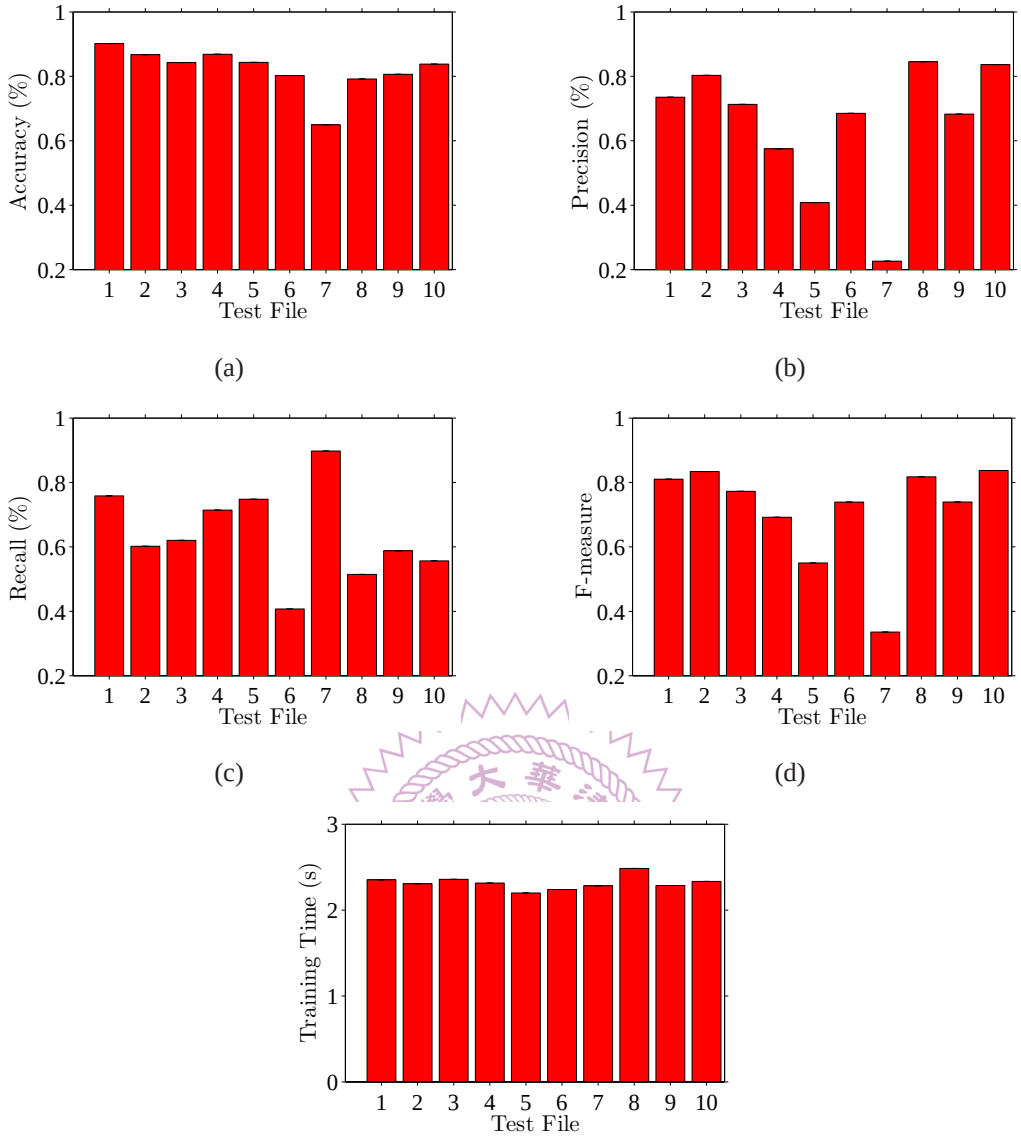


Figure 6.3: Evaluation result of SoIC problem using GBTC-based algorithm, (a) accuracy rate, (b) precison rate, (c) recall rate, (d) F-measure and (e) training time.

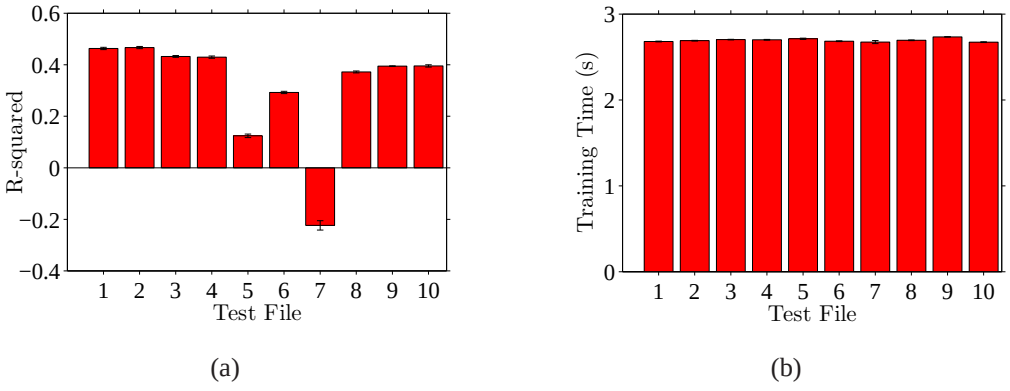
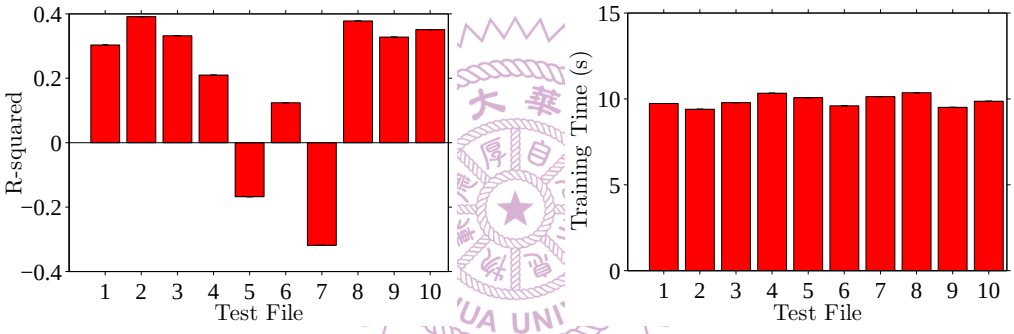
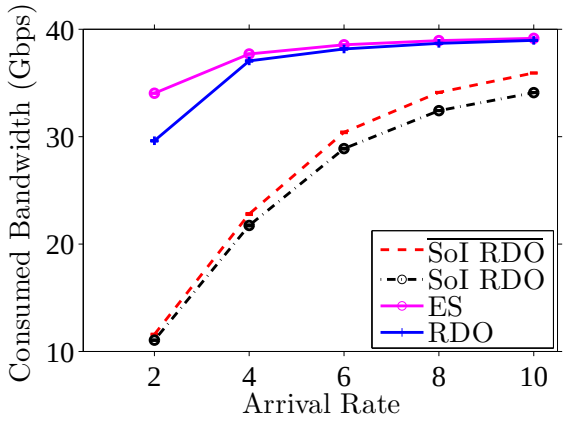


Figure 6.4: Evaluation result of SoIR problem using RFR-based algorithm, (a) R-squared score, (b) training time.

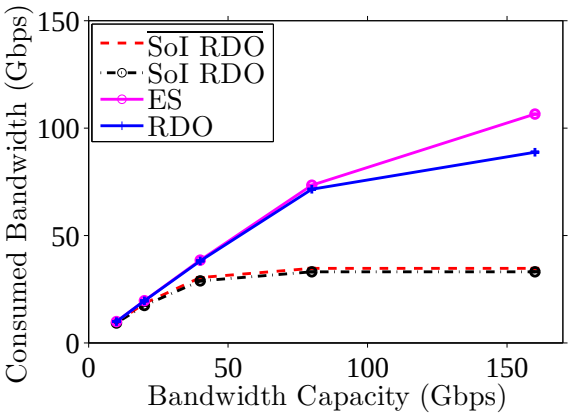


(a) (b)

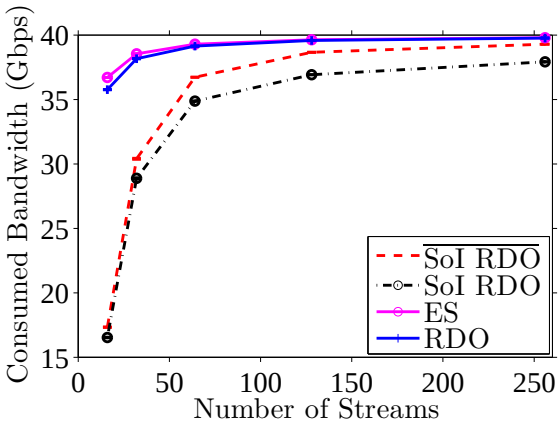
Figure 6.5: Training time in SoIR problem using GBTR-based algorithm, (a) R-squared score, (b) training time.



(a)

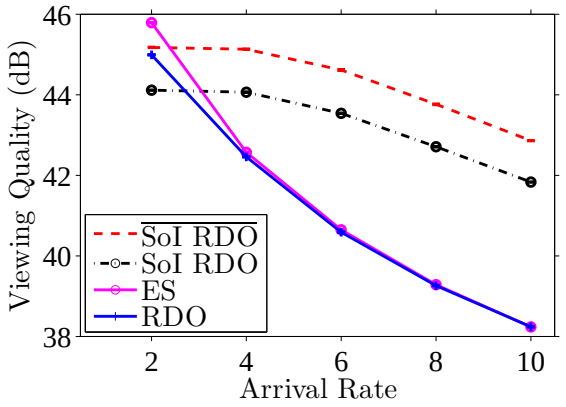


(b)

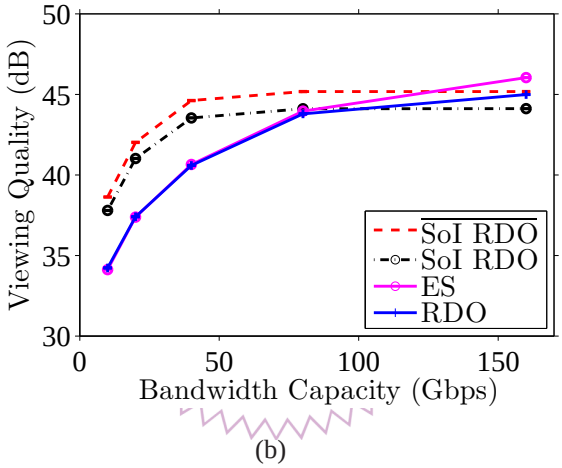


(c)

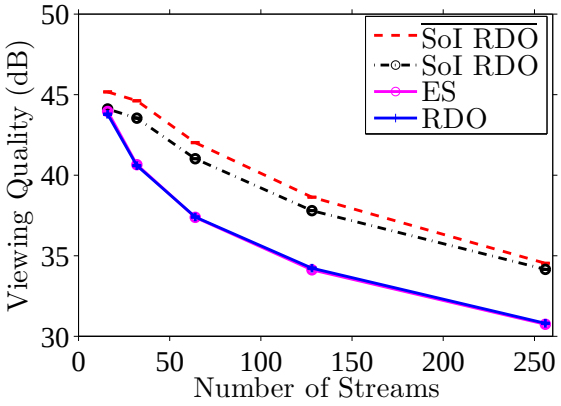
Figure 6.6: Total consumed bandwidth with diverse: (a) arrival rate, (b) total bandwidth, and (c) number of streams.



(a)



(b)



(c)

Figure 6.7: Average viewing quality with diverse: (a) arrival rate, (b) total bandwidth, and (c) number of streams.

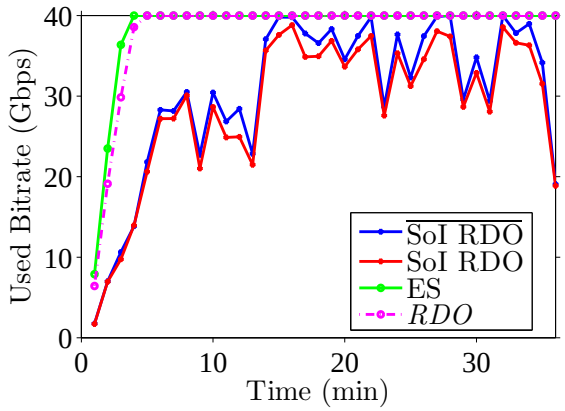
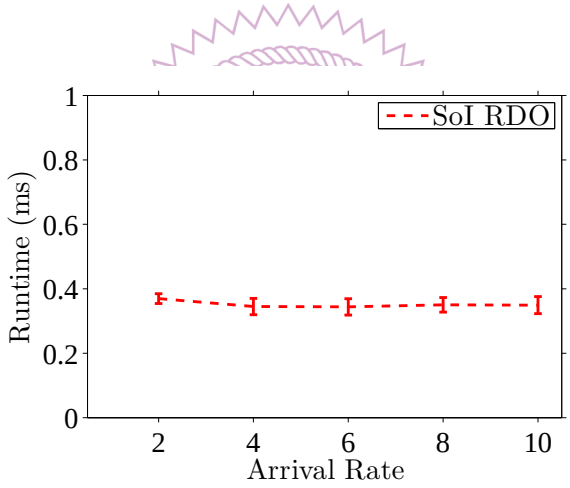
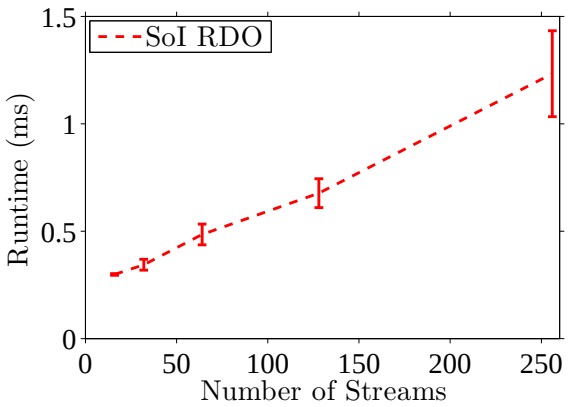


Figure 6.8: Sample consumed bandwidth over time from round 1.



(a)



(b)

Figure 6.9: Algorithm runtime with diverse: (a) arrival rate and (b) number of streams.

Chapter 7

Evaluations

In this chapter we present the rult of evaluation on SoI detector and resource allocator proposed in this thesis. We conduct two kinds of evaluation on the proposed SoI detecting algorithms, and we implemented a simulator to evaluate our resource allocating algorithm.

7.1 SoI Detector Evaluation

7.1.1 Evaluation Setup

We conduct out evaluation with $D_{e,R}$ and $D_{e,C}$ on SoIR problem and SoIC problem using a AMD 64-core server with 377 GB of RAM We repeat each of the evaluation 10 times and report 95% confident interval whenever possible in our result. There are two set of evaluations we conducted, (i) results from $D_{e,R}$ and $D_{e,C}$ using the optimal hyperparameter set from Sec. 4.4, (ii) results from our SoI detecting simulator.

7.1.2 Results From $D_{e,R}$ And $D_{e,C}$

In Fig. 6.1 we report the evaluation result using the evaluation dataset from Sec. 4.4. From the figures we make the following observation.

GBTR-based algorithm outperforms RFR-based algorithm in SoIR problem. From Fig. 6.1(a) we can observe that GBTR-based algorithm clearly outperform RFR-based algorithm.

GBTC-based algorithm provide slightly better performance at the cost of training time in SoIC problem. From Fig. 6.1(b) we can observe that GBTC-based algorithm slightly outperform RFC-based algorithm. However, from the numbers reported in Fig. 6.1(c) this comes at the cost of slightly higher training time.

Overall RF-based algorithms provide better training time comparing to GBT-based algorithms. In Fig. 6.1(c) and Fig. 6.1(d) we report the training of all the proposed

algorithms, the figures shows that overall RF-based algorithm provide slightly better training time comparing to GBT-based algorithms. We notice that both the Random Forest training module provided in Scikit-learn package [26] and the Gradient Boosting Tree module provided in XGBoost package [37] are capable of leveraging multiple CPUs in the server. Both of these modules use up to 64 cores in our 64-core CPU server when training with certain parameter set. In concept the training of Random Forest is an embarrassingly parallel problem, which may be the decision factor between the training time of the algorithms.

7.1.3 Results From SoI Simulator

We implemented a simulator to evaluate the SoI detecting algorithms we proposed for SoIC and SoIR problem. The simulator uses the historical data as training data and use the trained model in our proposed algorithms to detect SoI. We take the dataset D_R and D_C from Sec. 4 and take nine videos as training set, with the remaining one video as the evaluation set. We conduct grid search with 10-fold cross validation on the hyperparameter space mentioned in Sec. 4.4, and select optimal hyperparameterset, then we evaluate result using the remaining video. The procedure is repeated for 10 times with each video in the 10 videos take turn to be the evaluating one.

We report the result in Fig. 6.2, Fig. 6.3, Fig. 6.4 and Fig. 6.5. From the figure we make the following observations:

We achieve good results in SoIC problem. Fig. 6.2 and Fig. 6.3 show that in SoIC problem both of the algorithms can achieve close to 0.8 in terms of F-measure except when file number 5 or 7 is used as the evaluating file.

There is still room for improvements in SoIR problem. Figure. 6.4 and Fig. 6.5 show that both RFR-based and GBTR-based algorithm can only achieve around 0.4 in terms of R-squared score, and when the evaluating file is 5 or 7, the performance drop even more. These results indicate that there is still room for us to improve our SoIR algorithms.

RF-based algorithms outperform GBT-based algorithms. Consistent with the evaluation results from Sec. 7.1.2, RF-based algorithms outperforms GBT-based algorithms in terms of both training time and prediction performance. One obvious difference can be observe from Fig. 6.4(a) and Fig. 6.5(a), when the evaluating file is file 5. Consider that Random Forest is known its' noise resistance, this is a reasonable outcome.

7.2 Resource Allocator Evaluation

7.2.1 Simulation Setup

We have captured 8 live gaming sessions on our testbed. Two are from Age of Empires II, three are from Spellweaver, one is from Hearthstone: Heroes of Warcraft, one is from Minecraft, and one is from Starcraft II. The sessions are captured in resolutions varying between 720p and 1080p. The minimal, mean, and maximal lengths of these live game sessions are 17 minutes, 82.65 minutes, and 3.69 hours. We transcode each captured video into $\{128, 256, 512, 1024, 2048, 4096, 8192\}$ kbps using x264 video codec with ultrafast preset. We use the bitrate and PSNR reported by x264 to perform non-linear regression for the model parameters in Eq. (5.1). The resulting models are fairly accurate with an average (maximal) MSE of 0.43 (1.14). We recruit several viewers to watch the live game session, and mark SoI segments as the ground truth. We get 14 viewer logs in total, with an average (maximal) SoI number of 5.64 (19) in each game session. The model parameters and viewer logs are used to drive our simulator. In our simulations, live game sessions and viewer logs are randomly chosen from the dataset, i.e., in the scenarios with many game sessions (or viewers), some model parameters (or viewer logs) may be used several times to test the scalability of our solution.

Our simulator is implemented using Python. The proposed SoI RDO algorithm is implemented in C. We compare our solution, called SoI-based R-D Optimized (SoI RDO), with two baseline solutions: (i) Equal Share (ES), which equally divides the outbound bandwidth among all viewers, (ii) R-D Optimized (RDO), which is optimized in the R-D sense, but does not consider SoI. The ES algorithm mimics the state-of-the-art live game streaming platforms, while RDO allows us to quantify the benefit of the SoI driven resource allocation. In SoI RDO, we transcode streams to viewers who are not in SoI at R_L for basic quality. In extreme cases, where RDO or SoI RDO fail to find feasible solutions, they fall back to ES. Last, we implemented an optimal algorithm in Matlab using `solve` function to solve the equation system as the benchmark.

We consider the following performance metrics, , and use overline to indicate the expected values in the figures.

- **Expected quality**: the objective function value reported by the resource allocation algorithm.
- **Actual quality**: the achieved objective function value reported by the simulator.
- **Running time**: the run-time of solving a resource allocation problem.
- **Expected consumed bandwidth**: reported by the resource allocation algorithm.

- **Actual consumed bandwidth**: reported by the simulator.
- **Network overhead**: the network traffic volume generated by feature senders.

Each round of simulations is 6 hours long, with several parameters: (i) Poisson arrival rate of viewers, which is in $\{2, 4, 6, 8, 10\}$ per minute for each stream, (ii) number of streams in the system, which is in $\{16, 32, 64, 128, 256\}$, (iii) total outbound bandwidth of the server, which is in $\{10, 20, 40, 80, 160\}$ Gbps, and (iv) interval of calling the proposed algorithm, which is in $\{1, 2, 5, 10, 60\}$ seconds. If not otherwise specified, we set the arrival rate as 6, the number of streams as 32, the bandwidth as 40 Gbps, and the interval of calling our algorithm as 60 seconds. We assume the viewers always finish the videos they watch. We repeat each simulation 5 times. We report the average performance results, and give 95% confidence intervals whenever applicable.

7.2.2 Results

Our algorithm outperforms RDO and ES in terms of viewing quality and consumed bandwidth. Figs. 6.6 and 6.7 show total consumed bandwidth and average viewing quality of viewers. We observe that as the load of the system increases, e.g., when the arrival rate reaches 10 in Fig. 6.7(a) or the number of streams reaches 256 in Fig. 6.7(c), SoI RDO significantly outperforms RDO and ES, achieving better viewing quality for the viewers by up to 5 dB, while consuming less total bandwidth in the system (see Figs. 6.6(a) and 6.6(c)).

From Figs. 6.6(b) and 6.7(b), we observe that when the consumed bandwidth is close, SoI RDO gives better viewing quality, e.g., when the bandwidth is set to 10 Gbps. We also note that when the load of the system is light, e.g., when the arrival rate is set to 2 or when the system has 160 Gbps bandwidth, RDO and ES slightly outperform the proposed SoI RDO. However, this is achieved with significantly more bandwidth consumption: 20 Gbps and 70 Gbps more, respectively.

Fig. 6.6 also shows that the expected bandwidth consumption reported by simulator is lower than that of actual consumed bandwidth, which is counter intuitive. We plot the bandwidth usage over time with default parameters in Fig. 6.8. We see that the expected consumed bandwidth is slightly higher than actual consumed bandwidth. We note that the instantaneous expected consumed bandwidth may occasionally become smaller than actual consumed bandwidth, due to the dynamic nature of the system.

Our algorithm runs efficiently and scales well. Figs. 6.9(a) and 6.9(b) report the runtime of finding the λ using SoI RDO algorithm. In all of the experiments the average runtime stays < 1 ms, except when the number of streams is 256, which leads to 100+ thousand viewers. In contrast, the optimal algorithm implemented in Matlab may take

> 6 minutes to terminate, with merely 58% chance for feasible (real number) solutions. The above results show that our algorithm scales well with larger systems.

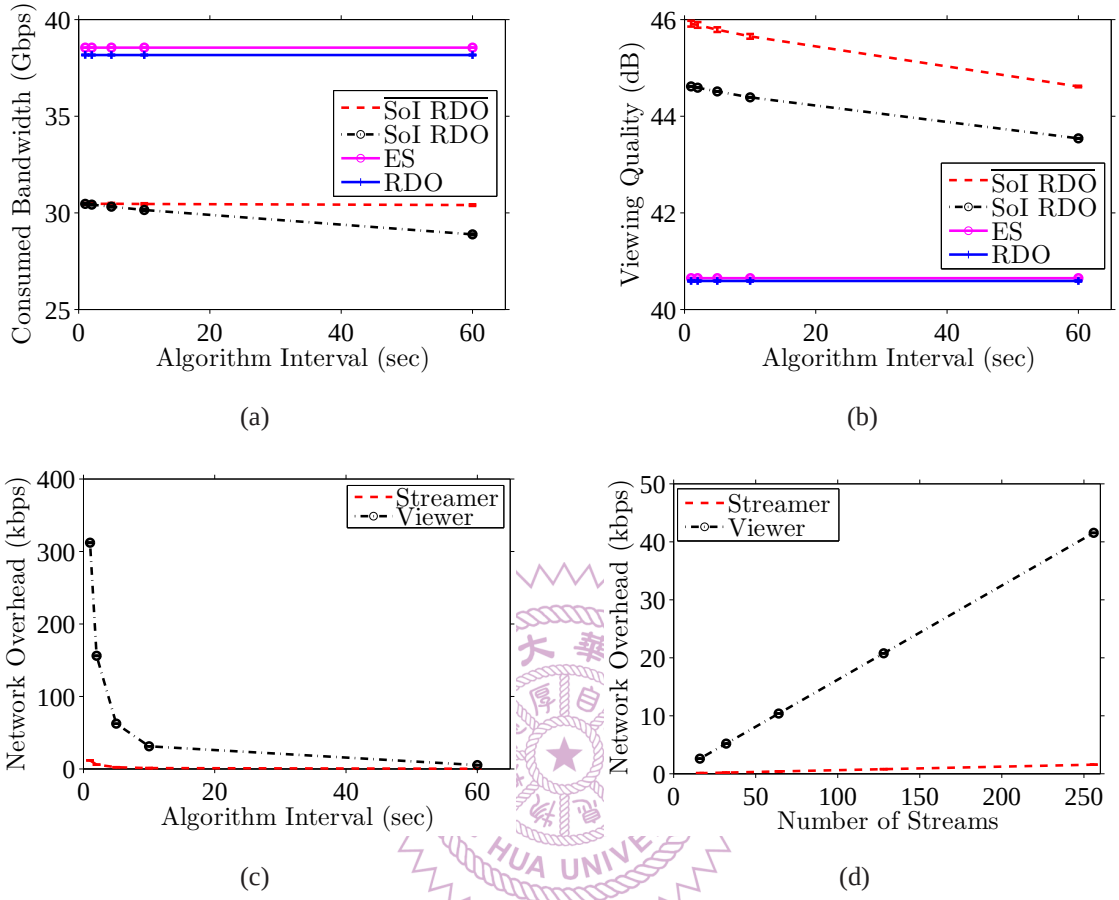


Figure 7.1: Implications of algorithm interval on: (a) consumed bandwidth, (b) average viewing quality, and (c) network overhead; (d) network overhead under different number of streams.

Implication of calling frequency Fig. 7.1 reports the implication of T, the interval of calling our proposed algorithm. In Figs. 7.1(a) and 7.1(b) show that, as T decreases, viewing quality slightly increases by 1 dB. This is because running our algorithm more frequently, the system has a better chance at adjusting to the new viewers that just arrive in the system.

Feature senders generate negligible network overhead. Fig. 7.1(c) reports the overhead generated by different interval parameters. Fig. 7.1(d) reports the overhead generated by different number of streams. We observe that, the network overhead generated by feature senders is negligible, even when the feature is returned on a per second basis, at just around 300 kbps with 12000+ viewers in the system.

Chapter 8

Related Work

8.1 General Live Gaming Streaming Related Research

Live game streaming service have not appeared in the literature until recent years. Kaytoue et al. [13] and Nascimeto et al. [18] have published work related to the modeling of the characteristics of live game streaming communities. The research by Kaytoue is the first one with topic highly related to live game streaming in the literature.

Zhang and Liu also published several works focusing on the viewer perceived latency [27] and region-based job placement algorithms [40] aimed to shorten the perceived latency by viewers. Pires and Simon conducted experiments to measure the behaviors of both Youtube and Twitch servers [24], and provide information such as the bandwidth consumption and number of channels of both service. They also made the dataset available to the research community. Pires and Simon also discussed integrating DASH into live game streaming system [29], and propose strategies on how to pick channels to transcode considering the tradeoff of transcoding overhead and the burden of delivering high quality raw video segments. In 2015 Deng et al. published a work which explore the popularity of games and channels on Twitch service in detail [7]. They discussed the trend during the period of their collected data, and analyzed the distribution of both popular games and channels in Twitch service in detail.

Hamilton et al. published a work related to Twitch which focus on the human-computer interaction (HCI) aspect [9]. They discussed how the sense of the community is formed between the audience and the streamer, and among audience themselves. They also discussed how different kinds of media, e.g., the video, the chat message and facecam interact with each other to form the experience of live game streaming. Margel published a work focusing on social dynamics behind the phenomenon called *TwitchPlaysPokemon* [16], which swept the Twitch community in 2014. At one time it holds up to 23% of the viewers in Twitch service.

In contrast to the work mentioned above, we proposed the concept of SoI to reduce cost without sacrificing viewer experience. Some of the solutions proposed in these works, e.g., region-based job placement, are complement to our system, and can be adopted to further enhance our proposed system.

8.2 Large Scale Transcoding

Our resource allocator is closely related to some of the large scale transcoding that aim to improve viewing quality for viewer in the video streaming service. Aparicio-Pardo et al. proposed a solution to automatically decide transcoding parameters and machine designation in cloud servers [4]. Zheng et al. also proposed a similar system with adaptive transcoding parameter decision and machine designation [41].

Our system are different from the above systems in the way that we take the SoI information into consideration, which offers another layer of possibility to further optimize the viewing quality perceived by viewers in our systems. We have also published our resource allocator in 2015 [5]. Combining our SoI concept with the some of the more sophisticated solutions is an interesting future work for our system.

8.3 Video Summarization, Highlight Detection and ROI

The concept of Segment-of-Interest is remotely related to Region of Interest (ROI), which have been extensively explored in video communications [14] and distance educations [17]. However, SoI operates on the temporal domain, while ROI applies to the spatial domain.

The other fields that are related to SoI are video summarization and highlight detection. There are existing work that aim to reduce bandwidth usage in cloud gaming systems using technique such as attention model derived from saliency map [1]. Hossain et al. also proposed a system aimed to alter the video frame delivered to players in a cloud gaming scenario according to the emotions detected from players' image frame and audio. [12] In 2015 Chu et al. proposed automatically highlight detection solution for live game streaming videos [6] that is highly related to the SoI detection problem in this thesis. The SoI detector proposed in this thesis is different from the existing works, we explore the possibility of detecting SoI information using other lightweight features in the system instead of content based analysis [6, 12], user attention model [1, 15], or model using eyeball analysis [23].

Chapter 9

Conclusion and Future Work

From the observation that *user experience only degrades when the users are paying attention*, we proposed to differentiate the importance of segment in a live game streaming, which opens up a wide spectrum of possibilities for further optimizing lie game streaming platforms. We refer to those important segments as *Segment-of-Interest (SoI)*. In this thesis we proposed a system called SoI-driven live game streaming platform.

We enhanced and integrated several open source projects such as OBS and SMPlayer to build a testbed. We use the testbed to collect real world data trace and use these traces to develop our proposed algorithms.

At the core of the proposed system are SoI detector and resource allocator. The SoI detector leverage machine learning algorithm, namely Random Forest and Gradient Boosting Tree to detect SoI from the lightweight features we collect from streamers and viewers. The resource allocator uses the SoI information given by the SoI detector to allocate resource among streams in the system. We leverage an R-D model and Lagrangian Multiplier to formulate the viewing quality optimization problem. We also developed an efficient approximation algorithm with a controllable error to cope with real world usage scenario, which is a highly dynamic one for a live game streaming system.

We recruited viewers to mark SoI ground truth for us and use these information in our supervised training for SoI detector and simulation for resource allocator. We conducted grid search with 10-fold cross validation to find the optimal hyperparameter set for Random Forest and Gradient Boosting Trees in SoI detector under supervised learning. We then use the optimal hyperparameter set to train and evaluate our SoI detector in two different way, (i) result from randomized selected data that haven't been used in the training shows that we can obtain up to 0.96 in terms of F-measure and up to 0.87 in terms of R-squared score, (ii) result using a simulator that takes historical data as training set shows that it can achieve close 0.8 in terms F-measure in SoIC problem.

We developed a simulator which uses real world data trace and rate distortion infor-

mation from non-linear regression to simulate a large scal live game streaming system in action. The results from our six hour long simulation shows that our proposed approximation resource allocation algorithm SoI RDO algorithm outperforms the state-of-the-art implementation by up to 5 dB in viewing quality and up to 50 Gbps in bandwidth consumption.

This work can be extended in several directions.

Larger scale dataset We are currently collecting features from more real gameplays in order to further expand the size of our dataset and develop new optimization algorithms.

Transcoder integration. Multi-viewer transcoders can be developed and evaluated using our testbed to alleviate the heavy workload incurred by the transcoding process in current live game streaming platforms.

Resource allocator with finer transcoder control. More sophisticated resource allocator which provide more quality options, e.g. frame per seconds and resolution, can be developed to provide more fine-grained user experience optimization.

Deep integration with game engines. To further optimize the performance of the platform, we can provide API for game engines to provide the streaming platform with SoI informations. With the deep insight of game logic and context of the game from game engine, Segment of Interest detection could be even more accurate, and can be easily applied to a wide variety of games.

By differentiating the segments' importance to viewers, we open up new opportunities for researchers and engineers to further optimizing live game streaming platforms in terms of user experience.

Bibliography

[1] H. Ahmadi, S. Khoshnood, M. Hashemi, and S. Shirmohammadi. Efficient bitrate reduction using a game attention model in cloud gaming. In *Proc. of IEEE International Symposium on Haptic Audio Visual Environments and Games (HAVE'13)*, Istanbul, Turkey, 2013.

[2] Amazon buys twitch for 970 million in cash. `http://www.businessinsider.com/amazon-buys-twitch-2014-8/`.

[3] Amd API homepage. `http://tinyurl.com/AMD-ADL`.

[4] R. Aparico-Pardo, K. Pires, A. Blanc, and G. Simon. Transcoding live adaptive video streams at a massive scale in the cloud. In *Proc. of ACM Multimedia Systems Conference (MMSys'15)*, Portland, Oregon, 2015.

[5] T. F. Chiang, H. Hong, and C. Hsu. Segment-of-Interest driven live game streaming: Saving bandwidth without degrading experience. In *Proc. of IEEE International Workshop on Network and Systems Support for Games (NetGames'15)*, Zagreb, Croatia, 2015.

[6] W. Chu and Y. Chou. Event detection and highlight detection of broadcasted game videos. In *Proc. of ACM Workshop on Computational Models of Social Interactions: Human-Computer-Media Communication (HCMC'15)*, Brisbane, Australia, 2015.

[7] J. Deng, F. Cuadrado, G. Tyson, and S. Uhlig. Behind the game: Exploring the twitch streaming platform. In *Proc. of IEEE International Workshop on Network and Systems Support for Games (NetGames'15)*, Zagreb, Croatia, 2015.

[8] FFmpeg homepage. `https://ffmpeg.org`.

[9] W. Hamilton, O. Garretson, and A. Kerne. Streaming on twitch: fostering participatory communities of play within live mixed media. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*, Toronto, Canada, 2014.

[10] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2th edition, 2001.

[11] HTTP live streaming. `http://tinyurl.com/HLS-draft`.

[12] M. Hossain, M. Muhammad, B. Song, M. Hassan, A. Alelaiwi, and A. Alamri. Audio–visual emotion-aware cloud gaming framework. *IEEE Transaction on Circuits and Systems for Video Technology*, 25(12):2105–2118, 2015.

[13] M. Kaytoue, A. Silva, L. Cerf, W. Meira, and C. Raïssi. Watch me playing, I am a professional: a first study on video game live streaming. In *Proc. of ACM Workshop on Mining Social Network Dynamics (MSND'12)*, Lyon, France, 2012.

[14] Y. Liu, Z. Li, and Y. Soh. Region-of-Interest based resource allocation for conversational video communication of H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(1):134–139, 2008.

[15] Y. Ma, L. Lu, H. Zhang, and M. Li. A user attention model for video summarization. In *Proc. of ACM International Conference on Multimedia (MM'02)*, Juan Les Pins, France, 2002.

[16] M. Margel. Twitch plays pokemon: An analysis of social dynamics in crowdsourced games. Technical report, University of Toronto, 2014.

[17] A. Mavlankar, P. Agrawal, D. Pang, S. Halawa, N. Cheung, and B. Girod. An interactive region-of-interest video streaming system for online lecture viewing. In *Proc. of Packet Video Workshop (PV'10)*, Hong Kong, China, 2010.

[18] G. Nascimento, M. Riberio, L. Cerf, N. Cesário, M. Kaytoue, C. Raïssi, T. Vasconcelos, and W. Meira. Modeling and analyzing the video game live-streaming community. In *Proc. of Latin American Web Congress (LA-WEB'14)*, Ouro Preto, Brazil, 2014.

[19] NGINX homepage. `http://nginx.org`.

[20] Novabench. `https://novabench.com/`.

[21] Nvidia API homepage. `https://developer.nvidia.com/nvapi`.

[22] Open Broadcast Software. `https://obsproject.com/`.

[23] W. Peng, W. Chu, C. Chang, C. Chou, W. Huang, W. Chang, and Y. Hung. Editing by viewing: Automatic home video summarization by viewing behavior analysis. *IEEE Transactions on Multimedia*, 13(3):539–550, 2011.

[24] K. Pires and G. Simon. Youtube Live and Twitch: A tour of user-generated live streaming systems. In *Proc. of ACM Multimedia Systems Conference (MMSys'15)*, Portland, Oregon, 2015.

[25] Adobe's Real Time Messaging Protocol. `http://tinyurl.com/rtmp-spec`.

[26] Scikit-learn. `http://scikit-learn.org`.

[27] R. Shea, D. Fu, and J. Liu. Towards bridging online game playing and live broadcasting: design and optimization. In *Proc. of ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'15)*, Portland, Oregon, 2015.

[28] SMPlayer homepage. `http://smplayer.sourceforge.net/`.

[29] I. Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *IEEE Multimedia*, 18(4):62–67, 2011.

[30] G. Sullivan and T. Wiegand. Rate-distortion optimization for video compression. *IEEE Signal Processing*, 20(6):74–90, 1998.

[31] Science: Surfing the 4th largest stream of data. `http://blog.twitch.tv/2015/05/science-surfing-the-4th-largest-stream-of-data/`.

[32] BOOM. More transcode servers. `http://blog.twitch.tv/2015/07/boom-more-transcode-servers/`.

[33] Twitch retrospective 2013. `http://www.twitch.tv/year/2013`.

[34] Twitch retrospective 2014. `http://www.twitch.tv/year/2014`.

[35] Twitch retrospective 2015. `https://www.twitch.tv/year/2015`.

[36] Windows performance counter website. `http://tinyurl.com/Window-PDH`.

[37] Xgboost. `http://github.com/dmlc/xgboost`.

[38] YouTube gaming website. `https://gaming.youtube.com/`.

[39] C. Zhang and J. Liu. On crowdsourced interactive live streaming: a twitch.tv-based measurement study. In *Proc. of ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'15)*, Portland, Oregon, 2015.

[40] C. Zhang, J. Liu, and H. Wang. Towards hybrid cloud-assisted crowdsourced live streaming: measurement and analysis. In *Proc. of ACM Workshop on Networks and Operating Systems Support for Digital Audio and Video (NOSSDAV'16)*, Klagenfurt am Wörthersee, Austria, 2016.

[41] Y. Zheng, D. Wu, Y. Ke, C. Yang, M. Chen, and G. Zhang. Online cloud transcoding and distribution for crowdsourced live game video streaming. *IEEE Transaction on Circuits and Systems for Video Technology*, PP(99):1–1, 2016.

[42] X. Zhu, E. Setton, and B. Girod. Congestion-distortion optimized video transmission over ad hoc networks. *Signal Processing: Image Communication*, 20(8):773–783, 2005.