OffDQ: An Offline Deep Learning Framework for QoS Prediction

Soumi Chattopadhyay*, Richik Chanda*, Suraj Kumar*, Chandranath Adak†
*Indian Institute of Information Technology Guwahati, India
†Indian Institute of Information Technology, Lucknow, India
India
{soumi,richik.chanda,suraj.kumar}@iiitg.ac.in,chandra@iiitl.ac.in

ABSTRACT

With the increasing trend of web services over the Internet, developing a robust Quality of Service (QoS) prediction algorithm for recommending services in real-time is becoming a challenge today. Designing an efficient QoS prediction algorithm achieving high accuracy, while supporting faster prediction to enable the algorithm to be integrated into a real-time system, is one of the primary focuses in the domain of Services Computing. The major state-of-the-art QoS prediction methods are yet to efficiently meet both criteria simultaneously, possibly due to the lack of analysis of challenges involved in designing the prediction algorithm. In this paper, we systematically analyze the various challenges associated with the QoS prediction algorithm and propose solution strategies to overcome the challenges, and thereby propose a novel offline framework using deep neural architectures for QoS prediction to achieve our goals. Our framework, on the one hand, handles the sparsity of the dataset, captures the non-linear relationship among data, figures out the correlation between users and services to achieve desirable prediction accuracy. On the other hand, our framework being an offline prediction strategy enables faster responsiveness. We performed extensive experiments on the publicly available WS-DREAM dataset to show the trade-off between prediction performance and prediction time. Furthermore, we observed our framework significantly improved one of the parameters (prediction accuracy or responsiveness) without considerably compromising the other as compared to the state-of-the-art methods.

CCS CONCEPTS

• **Information systems → World Wide Web**.

KEYWORDS

QoS Prediction, Web Service Recommendation, Collaborative Filtering

ACM Reference Format:
Soumi Chattopadhyay*, Richik Chanda*, Suraj Kumar*, Chandranath Adak†. 2022. OffDQ: An Offline Deep Learning Framework for QoS Prediction. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3485447.3512107

1 INTRODUCTION

With the enhancement of network technology and access to sufficient computing resources (e.g., cloud/edge servers), the native software is transforming into web-based software [6]. Consequently, the number of public web services is escalating day by day. At www.programmableweb.com, 24211 APIs have been reported as of October 19, 2021. For a consumer, selecting a web service among multiple similar services is getting difficult with the growing options [3, 5]. As a result, the automated service recommendation is gaining significant attention, where the task is to put forward a service based on the user demand [33]. For service recommendation, a set of non-functional attributes of service, called Quality of Service (QoS) [15] (e.g., throughput, response time, availability, reliability, accessibility, interoperability, confidentiality), often play a crucial role. Therefore, predicting the QoS parameters of service is one of the prime criteria for service recommendation [30] due to their nature of being fluctuated in real-time. In this paper, we aim to design a QoS prediction framework, which is an established problem in the domain of Services Computing [6, 33].

The principal objective of designing a QoS prediction framework is to obtain high prediction accuracy [6] since the less accurate model defeats the purpose of its existence in the first place. Most of the state-of-the-art methods [2, 20, 23, 26, 27, 29] on QoS prediction, therefore, focus on achieving high prediction accuracy. However, considering the need-of-the-hour, prediction time [33] is another primary concern to develop an automated QoS prediction framework. Yet, the goal of attaining lesser prediction time is often compromised while designing a highly accurate QoS prediction method [14]. A slower QoS prediction method impedes the objective of being deployed into a real-time system. Furthermore, in recent times, with the exponential increase in the number of online users, the number of services is rising proportionately, which entails the prediction framework being scalable enough [6] to handle a large amount of data.

A robust QoS prediction framework necessitates the above goals to be fulfilled simultaneously. However, to the best of our knowledge, there is hardly any prediction method in the literature [6, 33], which successfully achieved all these goals concurrently owing to various associated challenges. For example, data sparsity in the QoS invocation log matrix [6] is one of the major obstacles to succeeding in high prediction accuracy. The memory-based collaborative filtering (CF) methods [1, 17, 30] mostly suffer from this data sparsity, and therefore, attain low accuracy. The cold-start problem

[11], where a set of new users/services is added, is another hurdle that often occurs in CF, which needs to be addressed to achieve high prediction accuracy. Scalability [6] is another issue associated with such memory-based CF due to the high number of similarity computations, which can be partially resolved by a clustering stage in offline mode [19]. The matrix factorization (MF)-based methods [4, 13, 31] have been proposed in the literature to resolve data sparsity and cold-start problems. However, MF-based methods sometimes fail to capture the complex relationship among QoS data [18], and thereby, are unable to yield satisfactory prediction accuracy. Hence, advanced machine learning architectures [23, 27] have been proposed in the literature, which can successfully capture the relationship among QoS data, and thereby, attain high accuracy. However, these methods end up taking high prediction time [2, 14]. This, in turn, makes these methods less preferable for the real-time recommendation system.

In this paper, we aim to address various challenges to achieve the goals discussed above. Here, we propose an offline QoS prediction framework (OffDQ), which includes two stages: (i) preprocessing and (ii) generating an offline prediction model. The preprocessing stage aims to remove the sparsity, while the second stage focuses on predicting the QoS value for a target user-service pair. The second stage again comprises a feature extraction phase to train the model, followed by constructing the model.

The major **contributions** of this paper are summarized below:

i) *Systematic analysis of challenges in QoS prediction*: We aim to analyze different challenges for QoS prediction problem systematically and propose possible solutions to address those challenges to achieve three major goals of QoS prediction concerning prediction accuracy, prediction time, and scalability.

ii) *Filling sparsity*: Sparse QoS matrix is one of the main reasons for obtaining less prediction accuracy. To fill up the sparse QoS matrix, we propose a new strategy that is a part of the preprocessing stage. The memory-based CF conjoined with neural network-based regression [7] is employed here to handle sparsity.

iii) *Building offline QoS prediction framework*: We develop a novel offline framework (OffDQ) for QoS prediction. Being offline, OffDQ is compatible with the real-time system due to taking less time for the prediction.

iv) *Offline feature extraction*: In this paper, we propose a novel strategy to extract features using a neural network and a denoising autoencoder [7, 22].

v) *Extensive experimentation*: We further performed an extensive experiment on the WS-DREAM-1 dataset [32] to validate the performance of OffDQ.

2 PROBLEM FORMULATION & CHALLENGES

The objective of a classical QoS prediction problem is to predict the value of a QoS parameter of a service for a given user, from the following inputs:

- A set of m services $S = \{s_1, s_2, \ldots, s_m\}$
- A set of n users $U = \{u_1, u_2, \ldots, u_n\}$
- A QoS matrix Q of dimension $n \times m$ with each entry, say q_{ij}, defined as:

$$q_{ij} = Q(i,j) = \begin{cases} x \in \mathbb{R}_{>0} & \text{; denotes QoS value of } s_j \text{ invoked by } u_i \\ 0 & \text{; invalid entry, i.e., } u_i \text{ never invoked } s_j \end{cases}$$

- A query $\mathcal{R} = (u_{r_1}, s_{r_2})$; $r_1 \in \{1, 2, \ldots, n\}, r_2 \in \{1, 2, \ldots, m\}$

The objective of this paper is to predict the value of $q_{r_1 r_2}$ in a reasonable prediction time with high prediction accuracy. Here, we aim to build a QoS prediction framework by systematically addressing different challenges of the prediction algorithm to achieve high prediction accuracy and faster responsiveness.

Before discussing our prediction framework, we first describe our goals and associated challenges to achieve these goals.

2.1 Goals and Challenges

We now describe the goals of designing a QoS prediction algorithm and various challenges associated with them.

- *Goal*-1 *(High prediction accuracy)*: The primary objective of an efficient QoS prediction algorithm is to attain high prediction accuracy. However, several issues impede achieving high accuracy, as discussed below:
  - C_0 *(Presence of outliers)*: The QoS matrix usually contains a few outliers, which has a significant impact on training a network. Handling outliers from the training dataset is important to achieve high accuracy.
  - C_1 *(Data Sparsity)*: The QoS matrix is usually sparse. In most cases, the missing QoS values cause severe degradation in prediction accuracy [6]. For example, prediction based on the similarity between users/services on the basis of their QoS profile suffers considerably due to the computation of the similarity from two partially valid vectors.
  - C_2 *(Complex relationship among QoS data)*: The prediction accuracy is often compromised if the complex relationship among QoS data of the user/service is not considered [18].
  - C_3 *(Influence of uncorrelated users/services)*: Sometimes, the correlations between users/services are required to be captured before predicting the QoS value of a given user-service pair. The prediction from uncorrelated data often causes accuracy degradation [33].
  - C_4 *(Cold-start)*: Addition of new users/services may affect prediction accuracy owing to lack of data to take any informed decision [6, 11].
- *Goal*-2 *(Faster prediction time)*: Faster responsiveness of QoS prediction algorithm is desirable if the algorithm requires to be incorporated in a real-time system. However, the following challenges are required to be addressed to achieve this goal.
  - C_5 *(Significant amount of online computation)*: A considerable amount of computation is carried out after knowing the target user-service pair including the training of a few online prediction frameworks [2]. This, in turn, indicates the requirement of preprocessing and designing a pre-trained model.
  - C_6 *(Designing offline algorithm)*: To reduce the prediction time, an offline learning algorithm can be beneficial. However, the challenge of designing an offline learning algorithm is to make it query-agnostic.
- *Goal*-3 *(High scalability)*: Scalability refers to handling a large dataset efficiently [6]. Therefore, to design a scalable QoS prediction framework following issue needs to tackle.
  - C_7 *(High dimensional data)*: The number of users and services can be significantly large [14]. Therefore, the algorithm should tackle a large-scale dataset.

In the next section, we describe the foundation of our framework that addresses the above challenges.

3 OffDQ: PROPOSED FRAMEWORK

In this paper, we develop an offline learning-based QoS prediction framework by exploiting the QoS matrix $\mathcal{Q}$. As discussed in Section 2.1, being an offline framework, the model should be designed in such a way so that it is independent of the query. In other words, it is not necessary to know the target user/service during the model training.

Two different types of feature vectors have been proposed in the literature: the first one for the online training model and the second one for the offline training model. In the online training model [2], given a target user-service pair, feature vectors in the training dataset is constructed from the QoS invocation profile of each user except the target user. Each feature vector comprises QoS values of all services except the target service for a user, where the QoS value of the target service of that user is considered as the target value. The feature vector corresponding to the target user is used to predict the target QoS value. The problem with this approach is that the feature vector can be constructed after having the information of the target user-service pair, which is only possible during prediction. Therefore, the prediction time of the model is very high.

Another type of feature vector takes user-id and service-id under consideration and creates one-hot vector [27]. Sometimes, the contextual (e.g., location) information of user/service is embedded with the feature vector to predict the target QoS. The problem occurs when a new user/service is added to the system. Moreover, the contextual information may not always be available to generate the feature vector. This indicates the necessity of introducing a new feature vector by exploiting $\mathcal{Q}$ to train an offline model.

In this paper, we build an offline neural network that takes the concatenation of the QoS profiles of the target user and service as the feature vector and aims to produce the target QoS value. The training details of the neural network are presented in Table 1.

Table 1: Training-testing details of offline neural network

Feature vectors: $\mathcal{F} = \{(\mathcal{Q}(i), \mathcal{Q}^T(j)) \mid \forall i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\}, \mathcal{Q}(i, j) \neq 0\}$ where, $\mathcal{Q}(i)$ denotes the i^{th} row of $\mathcal{Q}$, $\mathcal{Q}^T$ denotes the transpose of $\mathcal{Q}$
Training dataset: $\mathcal{T} = \{((\mathcal{Q}(i), \mathcal{Q}^T(j)), \mathcal{Q}(i, j)) \mid \forall(\mathcal{Q}(i), \mathcal{Q}^T(j)) \in \mathcal{F}\}$ where, $\mathcal{Q}(i, j)$ is the target value for the corresponding feature vector $(\mathcal{Q}(i), \mathcal{Q}^T(j))$

Being an offline model, the prediction time of the framework is reasonably good, which is an advantage of the proposed model. However, the model has the following limitations:

- $\mathcal{L}_0$: The presence of outliers in $\mathcal{Q}$ may have severe effects on the prediction accuracy.
- $\mathcal{L}_1$: Since $\mathcal{Q}$ is a sparse matrix, in most cases, many entries of the feature vector are invalid. This leads to severe degradation of the prediction accuracy.
- $\mathcal{L}_2$: The correlations between uses/services are mostly ignored.
- $\mathcal{L}_3$: The QoS profile of different services may have different trends (refer to the Appendix-A).
- $\mathcal{L}_4$: Large number of users/services may reduce scalability.

The above limitations are required to be addressed to make our framework robust. Therefore, in the following subsections, we discuss our strategies to achieve all three goals while dealing with their associated challenges. We begin by introducing a preprocessing module before discussing the design of our offline prediction framework. The preprocessing module aims to handle outliers and sparsity, as presented below.

3.1 Detection and Removal of Outliers

The objective of this module is to identify the outliers and remove them from $\mathcal{Q}$ for further processing. We propose an outlier detection algorithm, given an input vector $\mathcal{V}$, and then discuss its utilization for our case.

3.1.1 Outlier Detection Algorithm. Given an input vector $\mathcal{V}$, the objective of this algorithm is to detect the outliers from it, if any. Here, our algorithm exploits the fact that the mean of a vector is outlier sensitive. If a vector contains an outlier, the difference between mean and median can be observed.

The outlier detection algorithm considered in this paper is iterative. In each iteration, we compute the mean of $\mathcal{V}$ (say, μ), followed by the percentages of the data lesser than μ (say, l_μ) and greater than μ (say, r_μ). If the absolute difference between l_μ and r_μ is greater than a given threshold T_μ, this implies the existence of outliers at the side of μ (i.e., greater or lesser side) containing less amount of data. If the outlier is on the greater (lesser) side of μ, the highest (lowest) element is identified as an outlier. However, we remove the outlier, if it is beyond the range of $\mu \pm T_r\sigma$, where σ is the standard deviation of $\mathcal{V}$, T_r is an empirically fixed threshold. The algorithm terminates when an outlier cannot be removed further. In other words, the algorithm terminates if either $|l_\mu - r_\mu| \leq T_\mu$ or the detected outlier is within the range of $\mu \pm T_r\sigma$.

3.1.2 Removal of Outliers from QoS Matrix. We first identify the set of outliers from the QoS vector of each service (i.e., from each column of $\mathcal{Q}$) using the above algorithm. We then investigate the QoS vector of each user (i.e., from each row of $\mathcal{Q}$). However, we do not apply the outlier detection algorithm directly on the QoS vector of each user. Here, the main intuition is a few services can be more computationally intensive than others. In this case, the higher computationally intensive service can be identified as an outlier, which may be inappropriate in our case. Therefore, instead of considering the row vector of $\mathcal{Q}$, we use a transformed vector, where each element of the vector is computed by calculating the deviation of the QoS of a service for a user with respect to the mean QoS value of the service. In other words, the outlier detection algorithm is applied on the transformed row vector of $\mathcal{Q}$, where every element of each row in $\mathcal{Q}$ is subtracted from the corresponding column mean to obtain the transformed row. The final outlier is considered as the one that is an outlier with respect to the QoS vector of the corresponding service, as well as the transformed QoS vector of the corresponding user. After removing the outliers, we obtain Q as shown in Equation 1.

$$Q(i, j) = \begin{cases} \mathcal{Q}(i, j) & ; \mathcal{Q}(i, j) \text{ is not an outlier} \\ 0 & ; \text{ otherwise} \end{cases} \tag{1}$$

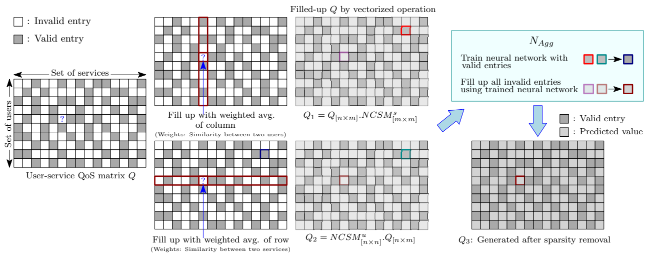

□ : Invalid entry
▨ : Valid entry
Set of services
Set of users
?
User-service QoS matrix Q
Fill up with weighted avg. of column
(Weights: Similarity between two users)
Fill up with weighted avg. of row
(Weights: Similarity between two services)
Filled-up Q by vectorized operation
$Q_1 = Q_{[n\times m]}.NCSM^s_{[m\times m]}$
$Q_2 = NCSM^u_{[n\times n]}.Q_{[n\times m]}$
N_{Agg}
Train neural network with valid entries
Fill up all invalid entries using trained neural network
▨ : Valid entry
□ : Predicted value
Q_3: Generated after sparsity removal

Figure 1: Sparsity removal

3.2 Sparsity Removal

To remove sparsity, we combine memory-based and model-based CF to fill up missing values in Q. The pictorial overview of this module is shown in Figure 1. The main idea here is to replace each invalid entry of Q with the aggregated weighted averages of column and row of the corresponding entry. We use the cosine similarity measure [2] as the weight in this computation.

We now elaborate on each step of this stage in detail.

i) We first compute the cosine similarity measure for each pair of users/services, and construct two matrices, say $CSM^u_{[n\times n]}$ and $CSM^s_{[m\times m]}$, where each entry of CSM^u and CSM^s are defined as:

$$CSM^u(i,j) = \begin{cases} CSM(Q(i), Q(j)) & ; Q(i) \text{ and } Q(j) \text{ are non-zero vectors} \\ 1 & ; \text{either } Q(i) \text{ or } Q(j) \text{ is zero vector} \end{cases} \tag{2}$$

$$CSM^s(i,j) = \begin{cases} CSM(Q^T(i), Q^T(j)) & ; Q^T(i) \text{ and } Q^T(j) \text{ are non-zero vectors} \\ 1 & ; \text{either } Q^T(i) \text{ or } Q^T(j) \text{ is zero vector} \end{cases} \tag{3}$$

where, $CSM(Q(i), Q(j))$ denotes the cosine similarity value between $Q(i)$ and $Q(j)$. It may be noted that the zero vector represents the corresponding user (service) that is newly added to the system. Furthermore, CSM^u and CSM^s are symmetric matrices.

ii) In the second step, we normalize CSM^u and CSM^s to obtain $NCSM^u$ and $NCSM^s$, where each entry of $NCSM^u$ and $NCSM^s$ are defined as:

$$NCSM^u(i,j) = \frac{CSM^u(i,j)}{\sum_{j=1}^{n} CSM^u(i,j)} \quad (4) \; ; \quad NCSM^s(i,j) = \frac{CSM^s(i,j)}{\sum_{j=1}^{m} CSM^s(i,j)} \quad (5)$$

iii) We then compute Q_1 and Q_2 as:

$$Q_1 = Q.NCSM^s \quad (6) \; ; \qquad Q_2 = NCSM^u.Q \quad (7)$$

It may be noted that each entry (i,j) of Q_1 and Q_2 represents the weighted average of i^{th} row and j^{th} column of Q, respectively, as shown below:

$$\underbrace{\begin{pmatrix} q^1_{11} & q^1_{12} & \cdots & q^1_{1j} & \cdots & q^1_{1m} \\ q^1_{21} & q^1_{22} & \cdots & q^1_{2j} & \cdots & q^1_{2m} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ q^1_{i1} & q^1_{i2} & \cdots & q^1_{ij} & \cdots & q^1_{im} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ q^1_{n1} & q^1_{n2} & \cdots & q^1_{nj} & \cdots & q^1_{nm} \end{pmatrix}}_{Q_1} =$$

$$\underbrace{\begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1m} \\ q_{21} & q_{22} & \cdots & q_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ q_{i1} & q_{i2} & \cdots & q_{im} \\ \cdots & \cdots & \cdots & \cdots \\ q_{n1} & q_{n2} & \cdots & q_{nm} \end{pmatrix}}_{Q} . \underbrace{\begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1j} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2j} & \cdots & w_{2m} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ w_{m1} & w_{m2} & \cdots & w_{mj} & \cdots & w_{mm} \end{pmatrix}}_{NCSM^s}$$

$$Q_1(i,j) = q^1_{ij} = \sum_{k=1}^{m} q_{ik} w_{kj} = \frac{\sum_{k=1}^{m} Q(i,k) CSM^s(k,j)}{\sum_{l=1}^{m} CSM^s(l,j)} \tag{8}$$

iv) Once we have Q_1 and Q_2, we aggregate them to obtain Q_3. The main intuition of this step is to reduce error by learning the difference between the actual QoS value and the value obtained by the similarity-based computations discussed above. We employ a neural network (say, N_{Agg}) for this aggregation. The purpose of N_{Agg} is to learn the difference between the actual and the predicted QoS values on the valid entries of Q and finally, predict the values of invalid entries of Q. The architecture of N_{Agg} is discussed in Section 4.2. The training-testing details of the network are presented in Table 2.

Table 2: Training-testing details of N_{Agg}

Feature vectors: $\mathcal{F}_{Agg} = \{(Q_1(i,j), Q_2(i,j)) \mid \forall i \in \{1,2,\ldots,n\}, j \in \{1,2,\ldots,m\}, Q(i,j) \neq 0\}$
Training set: $T_{Agg} = \{((Q_1(i,j), Q_2(i,j)), Q(i,j)) \mid \forall (Q_1(i,j), Q_2(i,j)) \in \mathcal{F}_{Agg}\}$ where, $Q(i,j)$ is the target value for the feature vector $(Q_1(i,j), Q_2(i,j)) \in \mathcal{F}_{Agg}\}$
Test vectors: $Te_{Agg} = \{(Q_1(i,j), Q_2(i,j)) \mid \forall i \in \{1,2,\ldots,n\}, j \in \{1,2,\ldots,m\}, Q(i,j) = 0\}$
To be predicted: All invalid entries of Q

Finally, Q_3 is constructed as follows:

$$Q_3(i,j) = \begin{cases} N_{Agg}(Q_1(i,j), Q_2(i,j)) & ; \text{if } Q(i,j) = 0 \\ Q(i,j) & ; \text{otherwise} \end{cases} \tag{9}$$

where, $N_{Agg}(Q_1(i,j), Q_2(i,j))$ is the predicted value obtained by N_{Agg} using feature vector $(Q_1(i,j), Q_2(i,j))$.

This stage handles the following challenges:

- The sparsity problem (C_1) is addressed here since all the invalid entries of Q are replaced by the predicted values generated in this stage.
- This stage takes into account the correlation between users and services. Therefore, challenge C_3 is addressed here.
- The cold-start problem (C_4) is also addressed in this stage, as Equations 2 and 3 consider the new user/service.
- This stage helps to achieve *Goal*-1 in two ways. Firstly, the data sparsity is handled here, which, in turn, helps to improve the performance of the offline model. Secondly, one level of prediction is already performed in this stage, which improves further in the subsequent phases of prediction.
- This is a preprocessing stage. Therefore, it does not have any impact on the prediction time. In other words, this stage does not impede in achieving *Goal*-2.

Once the missing values in Q are replaced by the predicted value, the feature vector considered in Table 1 is no longer contains any invalid entries since the feature vector is now generated from Q_3. However, the other challenges discussed earlier (i.e., $\mathcal{L}_3$, $\mathcal{L}_4$) remain unhandled. From the next subsection onwards, we discuss the detailed construction of our offline model. We begin with describing the feature extraction for our model.

3.3 Generation of Feature Vector

As discussed earlier, to predict the QoS value (say q_{ij}) of a user-service pair (u_i, s_j), we use the QoS profile of both the user (i.e., $Q_3(i)$) and the service (i.e., $Q^T_3(j)$) to generate a feature vector. However, as shown in Appendix-A, the QoS trends for different services vary. Therefore, if we directly use $Q_3(i)$ and $Q^T_3(j)$ as the feature vector, the accuracy of the algorithm may drop significantly. Hence,

Table 3: Feature vectors of $N_{\mathcal{F}}$ and $N_{\mathcal{G}}$

μ^s_j : Mean($Q^T_3(j)$)	μ^s_k : Mean($Q^T_3(k)$)	μ^u_i : Mean($Q_3(i)$)	μ^u_k : Mean($Q_3(k)$)
		μ^{u*}_i : Mean($Q^*_3(i)$)	μ^{u*}_k : Mean($Q^*_3(k)$)
σ^s_j : Stdev($Q^T_3(j)$)	σ^s_k : Stdev($Q^T_3(k)$)	σ^u_i : Stdev($Q_3(i)$)	σ^u_k : Stdev($Q_3(k)$)
		σ^{u*}_i : Stdev($Q^*_3(i)$)	σ^{u*}_k : Stdev($Q^*_3(k)$)
min^s_j : Min($Q^T_3(j)$)	min^s_k : Min($Q^T_3(k)$)	min^u_i : Min($Q_3(i)$)	min^u_k : Min($Q_3(k)$)
		min^{u*}_i : Min($Q^*_3(i)$)	$minu*_k$: Min($Q^*_3(k)$)
max^s_j : Max($Q^T_3(j)$)	max^s_k : Max($Q^T_3(k)$)	max^u_i : Max($Q_3(i)$)	max^u_k : Max($Q_3(k)$)
		max^{u*}_i : Max($Q^*_3(i)$)	max^{u*}_k : Max($Q^*_3(k)$)
ρ^s_{jk} : $CSM^s(j,k)$	σ^s_{jk} : Covariance ($Q^T_3(j), Q^T_3(k)$)	ρ^u_{ik} : $CSM^u(i,k)$	σ^u_{ik} : Covariance ($Q_3(i), Q_3(k)$)

$Q^*_3(i) = (q^*_{i1}, q^*_{i2}, \ldots, q^*_{im})$; $q^*_{ij} = Q_3(i,j) - \mu^s_j$; $j \in \{1, 2, \ldots, m\}$

Table 4: Training details of $N_{\mathcal{F}}$ and $N_{\mathcal{G}}$

Training details of $N_{\mathcal{F}}$
Feature vectors: $F_{\mathcal{F}} = \{f^{\mathcal{F}}_{ijk} \mid i \in \{1, \ldots, n\}; j, k \in \{1, \ldots, m\}; Q(i,j) \neq 0; Q(i,k) \neq 0\}$ where, $f^{\mathcal{F}}_{ijk} = (Q_3(i,k), \mu^s_j, \sigma^s_j, min^s_j, max^s_j, \mu^s_k, \sigma^s_k, min^s_k, max^s_k, \rho^s_{jk}, \sigma^s_{jk})$
Training dataset: $T_{\mathcal{F}} = \{(f^{\mathcal{F}}_{ijk}, Q(i,j)) \mid \forall f^{\mathcal{F}}_{ijk} \in F_{\mathcal{F}}\}$ where, $Q(i,j)$ is the target value for the corresponding feature vector $f^{\mathcal{F}}_{ijk}$
Training details of $N_{\mathcal{G}}$
Feature vectors: $F_{\mathcal{G}} = \{f^{\mathcal{G}}_{ijk} \mid i, k \in \{1, \ldots, n\}; j \in \{1, \ldots, m\}; Q(i,j) \neq 0; Q(k,j) \neq 0\}$ where, $f^{\mathcal{G}}_{ijk} = (Q_3(k,j), \mu^u_i, \sigma^u_i, min^u_i, max^u_i, \mu^u_k, \sigma^u_k, min^u_k, max^u_k, \mu^{u*}_i, \sigma^{u*}_i, min^{u*}_i,$ $max^{u*}_i, \mu^{u*}_k, \sigma^{u*}_k, min^{u*}_k, max^{u*}_k, \mu^s_j, \sigma^s_j, min^s_j, max^s_j, \rho^u_{ik}, \sigma^u_{ik})$
Training dataset: $T_{\mathcal{G}} = \{(f^{\mathcal{G}}_{ijk}, Q(i,j)) \mid \forall f^{\mathcal{G}}_{ijk} \in F_{\mathcal{G}}\}$ where, $Q(i,j)$ is the target value for the corresponding feature vector $f^{\mathcal{G}}_{ijk}$

The features of $F_{\mathcal{F}}$ and $F_{\mathcal{G}}$ are defined in Table 3.

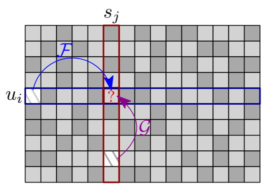


Figure 2: Transformation of feature vector

we design two transformation functions, $\mathcal{F}$ and $\mathcal{G}$, that take a value from $Q_3(i)$ and $Q^T_3(j)$, respectively, along with a few other statistical parameters to predict q_{ij}, as shown in Figure 2.

We first define $\mathcal{F}$, as $\mathcal{F} : \mathbb{R}^{11} \to \mathbb{R}$. The function $\mathcal{F}$ takes $Q_3(i,k)$, for $k \in \{1, 2, \ldots, m\}$, and 10 other statistical parameters (as shown in Table 4) to obtain the value of q_{ij}. To approximate $\mathcal{F}$, we employ a neural network $N_{\mathcal{F}}$. The architecture of $N_{\mathcal{F}}$ is discussed in Section 4.2. To train $N_{\mathcal{F}}$, we generate the pairs $(Q_3(i,j), Q_3(i,k))$ such that $Q(i,j) \neq 0, Q(i,k) \neq 0$. The formal description of the training of $N_{\mathcal{F}}$ is presented in Table 4.

$\mathcal{G} : \mathbb{R}^{23} \to \mathbb{R}$ is defined in the similar way as does $\mathcal{F}$. The objective of $\mathcal{G}$ is to obtain the value of q_{ij} from $Q_3(k,j)$, for $k \in \{1, 2, \ldots, n\}$ and 22 other statistical parameters, as shown in Table 4. Similar to $N_{\mathcal{F}}$, we employ another neural network $N_{\mathcal{G}}$, to approximate $\mathcal{G}$, for which the architectural details and the training details are presented in Section 4.2 and Table 4, respectively.

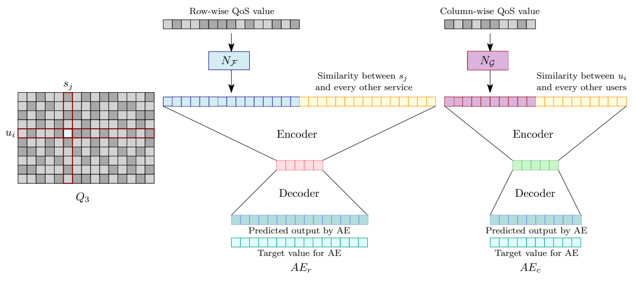


Figure 3: Noise removal using autoencoder

It may be noted that since all the statistical parameters are precomputed in the preprocessing stage, and $N_{\mathcal{F}}$ and $N_{\mathcal{G}}$ are pretrained, the transformation of the feature vector does not affect the prediction time.

The transformation functions $\mathcal{F}$ and $\mathcal{G}$ aim to predict the value of $Q(i,j)$ from $Q(i,k)$ and $Q(l,j)$, respectively, for $k = 1, 2, \ldots, m$, and $l = 1, 2, \ldots, n$. Therefore, the vectors obtained after transformations include the values that object to predict $Q(i,j)$. With these transformations, this phase aims to resolve $\mathcal{L}_3$. However, the feature vectors, generated after transformation, are noisy. This is possibly due to the fact that the prediction from the uncorrelated data using $N_{\mathcal{F}}$ and $N_{\mathcal{G}}$ introduces noise. Therefore, to remove noise from feature vectors, we employ denoising autoencoders [22] that are discussed in the next subsection.

3.4 Noise Removal from Feature Vector

The objective of this phase is to reduce the dimension of the feature vector while removing the noise introduced in the previous phase. The reduction of the feature dimension, in turn, helps to improve the scalability of OffDQ. For this, we employ two denoising autoencoders AE_r and AE_c.

A denoising autoencoder [22] comprises an encoder (E) followed by a decoder (D), where E approximates a function, which takes an input vector and generates a rich feature representation with a smaller dimension. The D, on the other hand, approximates a function, which takes the output of E and aims to generate the target vector. The input vector of the autoencoder may contain noise, however, the target vector is noise-free mostly. Therefore, such autoencoder is called *denoising*.

The autoencoder AE_r (AE_c) consists of an encoder E_r (E_c) followed by a decoder D_r (D_c). While AE_r operates on the transformed vector obtained from $Q_3(i)$ by employing $N_{\mathcal{F}}$, AE_c works on the transformed vector yielded from $Q^T_3(j)$ by $N_{\mathcal{G}}$. The training details of AE_r and AE_c are shown in Table 5. The overview of this module is presented in Figure 3.

The combined outputs of E_r and E_c serve as the input feature vector of the final prediction network N_{pred}. The smaller dimension of the input feature vector of N_{pred} resolves $\mathcal{L}_4$. Furthermore, the autoencoders reduce the noise from the feature vector of N_{pred}.

3.5 Offline QoS Prediction Model Construction

This is the final phase of our framework, where we intend to build the model N_{pred} for the QoS prediction. Given a target user-service

Table 5: Training details of AE_r and AE_c

Training details of AE_r
Feature vectors: $F_{AER} = \{f_{ij}^{AER} \mid i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\}, Q(i,j) \neq 0\}$ where, $f_{ij}^{AER} = (N_{\mathcal{F}}(Q_3(i,1), .), \ldots, N_{\mathcal{F}}(Q_3(i,m), .), CSM^s(j,1), \ldots, CSM^s(j,m))$ $N_{\mathcal{F}}(Q_3(i,k), .)$ is predicted by $N_{\mathcal{F}}$, given following feature vector: $(Q_3(i,k), \mu_j^s, \sigma_j^s, min_j^s, max_j^s, \mu_k^s, \sigma_k^s, min_k^s, max_k^s, \rho_{jk}^s, \sigma_{jk}^s)$
Training dataset: $T_{AER} = \{(f_{ij}^{AER}, \underbrace{(Q(i,j), Q(i,j), \ldots, Q(i,j))}_{\text{target vector } (t_{ij}^{AER}) \text{ of length } m}) \mid \forall f_{ij}^{AER} \in F_{AER}\}$
Custom loss function: $\mathscr{L}(f_{ij}^{AER}, t_{ij}^{AER}) = \text{MSE}(f_{ij}^{AER*}, t_{ij}^{AER})$, where, $f_{ij}^{AER*} = (N_{\mathcal{F}}(Q_3(i,1), .), \ldots, N_{\mathcal{F}}(Q_3(i,m), .))$
Training details of AE_c
Feature vectors: $F_{AEC} = \{f_{ij}^{AEC} \mid i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\}, Q(i,j) \neq 0\}$ where, $f_{ij}^{AEC} = (N_{\mathcal{G}}(Q_3(1,j), .), \ldots, N_{\mathcal{G}}(Q_3(n,j), .), CSM^u(i,1), \ldots, CSM^u(i,n))$ $N_{\mathcal{G}}(Q_3(k,j), .)$ is predicted by $N_{\mathcal{G}}$, given following feature vector: $(Q_3(k,j), \mu_i^u, \sigma_i^u, min_i^u, max_i^u, \mu_k^u, \sigma_k^u, min_k^u, max_k^u, \rho_{ik}^u, \sigma_{ik}^u)$
Training dataset: $T_{AEC} = \{(f_{ij}^{AEC}, \underbrace{(Q(i,j), Q(i,j), \ldots, Q(i,j))}_{\text{target vector } (t_{ij}^{AEC}) \text{ of length } n}) \mid \forall f_{ij}^{AEC} \in F_{AEC}\}$
Custom loss function: $\mathscr{L}(f_{ij}^{AEC}, t_{ij}^{AEC}) = \text{MSE}(f_{ij}^{AEC*}, t_{ij}^{AEC})$, where, $f_{ij}^{AEC*} = (N_{\mathcal{F}}(Q_3(i,1), .), \ldots, N_{\mathcal{F}}(Q_3(i,m), .))$

pair (u_i, s_j), we first generate the feature vector for N_{pred} by employing the feature generation technique discussed above. The training details of N_{pred} are shown in Table 6.

Table 6: Training details of N_{pred}

Feature vectors: $F_{pred} = \{(E_r(f_{ij}^{AER}), E_c(f_{ij}^{AEC})) \mid i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\}, Q(i,j) \neq 0\}$ where, $E_r(f_{ij}^{AER}), E_c(f_{ij}^{AEC})$ are outputs of E_r, E_c, respectively
Training dataset: $T_{pred} = \{((E_r(f_{ij}^{AER}), E_c(f_{ij}^{AEC})), Q(i,j)) \mid \forall (E_r(f_{ij}^{AER}), E_c(f_{ij}^{AEC})) \in F_{pred}\}$ where, $Q(i,j)$ is the target vector for $(E_r(f_{ij}^{AER}), E_c(f_{ij}^{AEC}))$

In the next section, we discuss the performance of OffDQ through experimental analysis.

4 EXPERIMENTAL RESULTS

In this section, we demonstrate our experimental results. OffDQ was implemented in Python 3.9.5. All experiments were performed on a system with the following configuration: Apple M1 @ 3.2 GHz with 16 GB LPDDR4X RAM.

4.1 Dataset Employed

OffDQ was validated on the publicly available WS-DREAM-1 [32] benchmark dataset. The details of this dataset are presented in Appendix-B.

We used mean absolute error (MAE) [14] as the metric to measure the prediction performance of OffDQ. The lower the MAE, the better is the performance.

In the experiment, each dataset was divided into training and testing sets. The experiment was performed on different densities (TrD) of the training set. Here, TrD = x% implies that x% of the QoS matrix (Q) is used to train the model, while the remaining (100 - x)% of Q is used for testing. We performed each experiment five times and recorded the average results.

Table 7: Performance (MAE) of OffDQ and comparison with major SoA methods over WS-DREAM-1 [32]

QoS: RT

Type		Method	TrD	
			10%	20%
memory -based	CF (similarity -based)	IPCC [17]	0.7000	0.5351
		UPCC [1]	0.6063	0.5379
		WSRec [30]	0.6394	0.5024
		NRCF [21]	0.5312	0.4607
		RACF [24]	0.4937	0. 4208
		RECF [34]	0.4332	0.3946
model-based	MF	MF [13]	0.5103	0.4981
		NIMF [31]	0.4854	0.4357
		WRAMF [4]	0.4657	-
	FM	AFM [12]	0.4849	0.3847
		EFM [25]	0.4446	-
	DA	CADFM [18]	0.3896	0.3600
		LDCF [29]	0.3670	0.3310
		DNM [23]	0.3628	-
		DAFR [27]	0.3461	0.3404
memory +model	Hybrid	CNR [2]	0.2597	0.1711
		CAHPHF [14]	0.0590	0.0419
		OffDQ	0.2550	0.2452

QoS: TP

Type	Method	TrD	
		10%	20%
CF (similarity -based)	IPCC [17]	27.3993	25.0273
	UPCC [1]	21.2695	17.5546
	WSRec [30]	19.9754	16.0762
MF	NMF [10]	17.8411	15.2516
	PMF [16]	16.1755	14.6694
	NIMF [31]	16.0542	13.7099
FM	EFM [25]	15.0268	-
DA	DAFR [27]	16.9020	15.5670
	DNM [23]	12.9200	-
	LDCF [29]	12.3820	10.8410
Hybrid	CAHPHF [14]	5.9800	4.1890
	OffDQ	10.4598	10.2437

TrD: *Training Density*; CF: *Collaborative Filtering*; MF: *Matrix Factorization*; FM: *Factorization Machine*; DA: *Deep Architecture*

4.2 Model Configurations

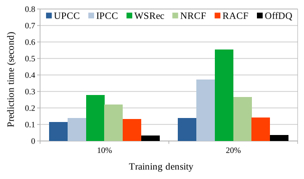
In our work, we employed multiple neural network-based architectures, where the hyper-parameters were decided empirically. We used 15% of the training data as a validation set to tune the hyper-parameters. For outlier detection, we empirically set T_μ = 30% and T_r = 2.5. The architectural details (e.g., loss functions, optimization functions, activation functions, number of hidden layers, number of neurons per layer [7]) are provided in Appendix-C.

4.3 Experimental Analysis

We now present the experimental analysis to show the performance of OffDQ.

Table 7 and Figure 4 show the performance of OffDQ with respect to the state-of-the-art (SoA) methods in terms of prediction accuracy and prediction time, respectively. We have the following **observations** from experimental results:

i) *Comparison with memory-based CF*: As observed from Table 7, OffDQ outperformed the memory-based CF methods in terms of prediction accuracy. For WS-DREAM-1: RT dataset [32], OffDQ achieved 41.14% and 37.86% improvements for 10% and 20% of training density (TrD), respectively, over the best memory-based CF method enlisted in the table. Similarly, for WS-DREAM-1: TP



(a)

Prediction time (second)		
TrD	10%	20%
CAHPHF [14]	312.6	319.8
CNR [2]	107.4	148.2
OffDQ	0.0317	0.0351

(b)

Figure 4: Performance (prediction time) of OffDQ and comparison with various SoA methods over WS-DREAM-1: RT dataset [32]

dataset [32], OffDQ attained 47.64% and 36.28% improvements for 10% and 20% of TrD, respectively, over the same.

ii) *Comparison with model-based CF*: Similar to memory-based CF methods, OffDQ performed better than the model-based CF methods in terms of prediction accuracy. As evident from Table 7, for RT dataset, OffDQ attained 26.32% and 25.92% improvements for 10% and 20% of TrD, respectively, over the best-predicted values obtained by the model-based methods. For TP dataset also, we observe a similar trend, where OffDQ achieved 15.52% and 5.51% improvements for the above cases.

iii) *Comparison in terms of prediction time*: As evident from Figure 4 (a), OffDQ outperformed the computation-efficient SoA methods concerning the prediction time. It may be noted that in the figure, we consider only a few methods having less prediction time for our comparison since the sophisticated methods take more time to predict. On an average, OffDQ achieved 4.12x speed-up as compared to the fastest method recorded in Figure 4.

Furthermore, we present the prediction time for a few online methods that are either comparable or better than OffDQ in terms of prediction accuracy. As evident from Figure 4 (b), the prediction time of the online methods (e.g., CNR, CAHPHF) is in the order of hundreds of seconds, while the prediction time of OffDQ is in the order of tens of milliseconds. Hence, OffDQ is better suited for the real-time system over the above methods.

iv) *Comparison with offline methods*: Most of the offline prediction models (e.g., DAFR [27], LDCF [29]) employ the contextual information of users/services as the feature vector to train the model. As observed from Table 7, OffDQ outperformed these methods in terms of prediction accuracy. On an average, OffDQ achieved 28.42% and 26.94% improvements for 10% and 20% of TrD on the RT dataset, respectively. Similarly, for the TP dataset, OffDQ obtained 26.82% and 19.85% improvements for the same. Moreover, the prediction time for OffDQ are comparable with these methods.

v) *Comparison with online methods*: Among hybrid methods, on average, OffDQ achieved 1.81% improvement over CNR [2] for 10% of TrD. However, for 20% of TrD, on average, CNR achieved 30.22% improvement over OffDQ. In some cases, CNR performed better than OffDQ in terms of prediction accuracy; however, OffDQ, being an offline framework, achieved a significant amount of improvement over CNR in terms of prediction time. As discussed above, while the prediction time of CNR is in the order of hundreds of seconds, the prediction time of OffDQ is in the order of tens of milliseconds.

Although the prediction accuracy of CAHPHF [14] was the best (on average, CAHPHF achieved 65.42% improvement over OffDQ), the prediction time for CAHPHF is considerably larger than OffDQ (on average, OffDQ achieved 9486.15x speed-up over CAHPHF), as evident from Figure 4 (b).

vi) *Observation on outliers*: During the testing of OffDQ, 5%-7% of testing data were detected as outliers, where the MAE was considerably high.

Tables 8 and 9 show the **performance and requirement of intermediate stages**, which are discussed below:

i) *Impact of sparsity removal phase*: This phase comprises similarity-based user and service collaborative filtering followed by their aggregation using the neural network N_{Agg}. The MAE generated by N_{Agg} is shown in Table 8, which captures the performance of the sparsity removal module. As presented in Table 9, the performance of OffDQ is better than OffDQwoOPM, which consists of only the preprocessing module of OffDQ. More specifically, on average, OffDQ is 56.91% better than OffDQwoOPM. This justifies the requirement of the latter stages of OffDQ. On the other hand, if the sparsity removal module is ablated from OffDQ (refers to OffDQwoSR), the performance degrades by 44.99%, as shown in Table 9.

ii) *Impact of outlier removal*: As evident from Table 9, OffDQ achieved 26.57% improvement over the model without removing outliers (refers to OffDQwoOL), which justifies the utility of eliminating outliers.

iii) *Impact of feature extraction*: Two neural networks, $N_{\mathcal{F}}$ and $N_{\mathcal{G}}$, are responsible for transforming the feature vectors required by the online prediction module of OffDQ. The validation MAEs obtained by these networks are presented in Table 8. From Table 9, we observe that the performance of OffDQ without feature transformation (refers to OffDQwoFT) decays, on average, 47.72% from OffDQ, which comprehends the utility of the feature transformation.

iv) *Impact of noise removal*: AE_r and AE_c autoencoders are employed to remove the noises from the feature vectors obtained from $N_{\mathcal{F}}$ and $N_{\mathcal{G}}$. The performances of AE_r and AE_c are mentioned in Table 8 in terms of their validation MAE. The performance of

Table 8: Performance (MAE) of intermediate stages

	QoS: RT		QoS: TP	
TrD / Network	10%	20%	10%	20%
N_{Agg}	0.5674	0.4502	33.8949	24.3351
$N_{\mathcal{F}}$	0.4629	0.5239	30.5727	22.9580
$N_{\mathcal{G}}$	0.4380	0.4506	34.6815	23.2642
AE_r	0.3451	0.4661	11.4721	14.0550
AE_c	0.1423	0.1583	8.7349	10.3547
N_{pred}	0.2550	0.2452	10.4598	10.2437

Table 9: Performance (MAE) by ablating stages

	QoS: RT		QoS: TP	
TrD / Network	10%	20%	10%	20%
OffDQwoOPM	0.5674	0.4502	33.8949	24.3351
OffDQwoSR	0.3107	0.3397	30.1606	32.9149
OffDQwoOL	0.2704	0.2528	30.9632	14.9206
OffDQwoFT	0.3790	0.3215	32.4481	30.7257
OffDQwoNR	0.4063	0.2923	31.1481	30.1691
OffDQ	0.2550	0.2452	10.4598	10.2437

OffDQwoOPM: OffDQ without Offline Prediction Module; OffDQwoSR: OffDQ without Sparsity Removal; OffDQwoOL: OffDQ without handling outliers; OffDQwoFT: OffDQ without Feature Transformation; OffDQwoNR: OffDQ without Noise Removal

OffDQ without the noise removal module (refers to OffDQwoNR) deteriorates by 46.45%, on average, from OffDQ, as shown in Table 9, which depicts the significance of the noise removal phase.

As evident from our experimental results, OffDQ outperformed the SoA in terms of prediction accuracy and prediction time.

5 RELATED WORKS

In this section, we discuss various state-of-the-art methods and their strength and limitations from the point of view of addressing different challenges and achieving the goals discussed in this paper. As mentioned in Section 2.1, in this paper, we mainly focus on three major goals: high prediction accuracy, faster prediction time, and high scalability. We further analyze a few associated challenges to attain these goals.

The first challenge discussed in this paper is related to data sparsity [6]. The QoS matrix, containing the values of a QoS parameter of services invoked by different users, is often sparse. Due to this sparsity, various functions (e.g., distance/similarity calculation) produce inappropriate results when operated on two partially valid vectors. The data sparsity problem becomes severe when new users/services are added for recommendation due to the lack of their QoS profile. This situation is called the cold-start problem [6, 11]. Memory-based CF [1, 17, 21, 24, 30, 34], where the prediction largely relies on the similarity values between users/services, usually suffers from data sparsity and cold-start problems. Therefore, these methods hardly attain desirable prediction accuracy. A set of model-based CF methods (e.g., leveraging matrix factorization: MF [4, 10, 13, 16, 31], or factorization machines: FM [12, 25], or various deep architectures: DA [2, 14, 20, 23, 26, 27]) has been proposed in the literature to resolve these issues. However, the model-based CF has its own limitations, as discussed below.

Capturing the relationship among data is one of the driving factors that leads to high prediction accuracy [18]. For example, when a QoS value of a target user-service pair is predicted from the QoS matrix, the correlation between the target user/service with other users/services needs to be captured [33]. The prediction from uncorrelated data may yield to less accurate results. Although the memory-based CF methods can capture the correlation among users/services, their suffering from the other problems impedes them to meet satisfactory prediction accuracy. The model-based CF methods [4, 10, 12, 25], on the other hand, often fail to apprehend the correlation among users/services, and thereby, they are incapable of delivering desired results. The other important challenge to address is capturing the complex relationship among QoS data. Some of the model-based CF, such as MF-based methods [4, 10, 13, 16, 31] fail to resolve this issue. The DA methods [20, 27] have been proposed to comprehend the relationship among QoS data. Among these DA methods, a few models [20, 23, 26] turn out to be inadequate to meet expected prediction accuracy due to ignoring a few challenges discussed already. The others [2, 14], however, have further disadvantages, as discussed next.

A few hybrid methods [2, 14] combining the idea of memory-based and model-based CF have been proposed in the literature, which can address the above challenges to achieve satisfactory prediction accuracy. However, these methods experience a significantly high prediction time. The main reason for having a high prediction time is that a lot of computations are done after having the knowledge of the target user-service pair, including the training of the deep learning models proposed in [2, 14]. This necessitates designing an offline training model. A few offline training models exist in the literature [18, 27, 29], where contextual features (e.g., location of users/services) have been used. However, the prediction accuracies obtained by these methods have yet to reach a satisfactory level due to their limitation of not taking into account the relationship among QoS data.

Scalability is another concern needs to deal with to handle large dataset [6]. This issue arises when a lot of online computations are required to be performed on large data. Clearly, memory-based CF suffers from scalability since a considerable amount of similarity computations are required to be computed during real-time recommendations. The advanced model-based CF also suffers from low scalability due to their trend of online training [2]. The memory-based CF has been put together with offline clustering [19, 28] to resolve the issue partially. A summary of different related works from the perspective of handling various challenges is discussed in Appendix-D.

In contrast to the above QoS prediction methods, we propose an offline training model (OffDQ), which can also comprehend the complex relationship of QoS data. OffDQ attempts to achieve all three goals while addressing the above challenges. From Table 12, the positioning of our work in the literature can be observed with respect to attaining the goals.

6 CONCLUSION

In this paper, we propose a scalable and reasonably faster offline QoS prediction framework by analyzing various challenges to solve the prediction problem to attain high prediction accuracy. Our framework comprises a preprocessing step followed by designing a prediction module by leveraging different deep-learning architectures. Our preprocessing phase is a hybrid method consisting of memory-based and model-based collaborative filtering to solve the sparsity problem, aiming to improve the prediction accuracy. The prediction module includes the feature extraction phase followed by the design of the final prediction module. The feature extraction phase aims to obtain appropriate feature vectors to achieve desirable prediction accuracy that consists of neural networks and autoencoder.

OffDQ is different from the contemporary QoS prediction model in understanding the complex relationship of data without contextual information. Furthermore, OffDQ is also scalable and can handle a sufficiently large database within a reasonable time limit. We performed extensive experiments on the WS-DREAM-1 dataset, which confirms that OffDQ outperformed the major state-of-the-art methods in terms of prediction accuracy and prediction time simultaneously.

Although OffDQ achieved all three goals simultaneously by addressing the challenges discussed in Section 2.1, there is still scope for improving the prediction accuracy. The QoS matrix contains a few outliers, which are difficult to predict. As our future endeavor, we aim to propose an outlier resilient QoS prediction system. Furthermore, we will attempt to build a more robust system that can not only compete with the online prediction algorithms but also can capture the temporal aspect of the prediction.

ACKNOWLEDGMENTS

This work is supported by the Science and Engineering Research Board, Department of Science and Technology, Government of India, under Grant SRG/2020/001454.

REFERENCES

[1] J. S. Breese et al. 1998. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Uncertainty in Artificial Intelligence.* 43–52.
[2] S. Chattopadhyay and A. Banerjee. 2019. QoS Value Prediction Using a Combination of Filtering Method and Neural Network Regression. In *ICSOC.* 135–150.
[3] S. Chattopadhyay, A. Banerjee, and T. Mukherjee. 2016. A Framework for Top Service Subscription Recommendations for Service Assemblers. In *IEEE SCC.* 332–339.
[4] Z. Chen et al. 2020. An Accurate and Efficient Web Service QoS Prediction Model with Wide-range Awareness. *Future Gener. Comput. Syst.* 109 (2020), 275–292.
[5] X. Fan et al. 2021. CASR-TSE: Context-Aware Web Services Recommendation for Modeling Weighted Temporal-Spatial Effectiveness. *IEEE TSC* 14, 1 (2021), 58–70.
[6] S. H. Ghafouri, S. M. Hashemi, and P. C. K. Hung. 2020. A Survey on Web Service QoS Prediction Methods. *IEEE TSC* (2020), 1–1. https://doi.org/10.1109/TSC.2020.2980793
[7] I. J. Goodfellow, Y. Bengio, and A. C. Courville. 2016. *Deep Learning.* MIT Press.
[8] G. Hinton et al. 2012. RMSprop: Divide the Gradient by a Running Average of its Recent Magnitude. *Lecture 6e: Neural Networks for Machine Learning* (2012).
[9] D. P. Kingma and J. Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR, arXiv:1412.6980.*
[10] D. D. Lee and H. S. Seung. 1999. Learning the Parts of Objects by Non-Negative Matrix Factorization. *Nature* 401, 6755 (1999), 788.
[11] K. Lee, J. Park, and J. Baik. 2015. Location-based Web Service QoS Prediction via Preference Propagation for Improving Cold Start Problem. In *IEEE ICWS.* 177–184.
[12] M. Li et al. 2020. A Two-Tier Service Filtering Model for Web Service QoS Prediction. *IEEE Access* 8 (2020), 221278–221287.
[13] W. Lo et al. 2012. An Extended Matrix Factorization Approach for QoS Prediction in Service Selection. In *IEEE SCC.* 162–169.
[14] R. R. Chowdhury, S. Chattopadhyay, and C. Adak. 2020. CAHPHF: Context-aware Hierarchical QoS Prediction with Hybrid Filtering. *IEEE TSC* (2020), 1–1. https://doi.org/10.1109/TSC.2020.3041626
[15] S. Ran. 2003. A Model for Web Services Discovery with QoS. *SIGecom Exch., ACM* 4, 1 (2003), 1–10.
[16] R. Salakhutdinov and A. Mnih. 2007. Probabilistic Matrix Factorization. In *NIPS.* 1257–1264.
[17] B. M. Sarwar et al. 2001. Item-based Collaborative Filtering Recommendation Algorithms. *WWW* 1 (2001), 285–295.
[18] L. Shen et al. 2020. Contexts Enhance Accuracy: On Modeling Context Aware Deep Factorization Machine for Web API QoS Prediction. *IEEE Access* 8 (2020), 165551–165569.
[19] Y. Shi et al. 2011. A New QoS Prediction Approach based on User Clustering and Regression Algorithms. In *IEEE ICWS.* 726–727.
[20] M. I. Smahi et al. 2018. An Encoder-Decoder Architecture for the Prediction of Web Service QoS. In *ESOCC.* 74–89.
[21] H. Sun et al. 2013. Personalized Web Service Recommendation via Normal Recovery Collaborative Filtering. *IEEE TSC* 6, 4 (2013), 573–579.
[22] P. Vincent and et al. 2010. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *JMLR* 11, 110 (2010), 3371–3408.
[23] H. Wu et al. 2021. Multiple Attributes QoS Prediction via Deep Neural Model with Contexts. *IEEE TSC* 14, 4 (2021), 1084–1096.
[24] X. Wu et al. 2017. Collaborative Filtering Service Recommendation based on a Novel Similarity Computation Method. *IEEE TSC* 10, 3 (2017), 352–365.
[25] Y. Wu et al. 2017. An Embedding Based Factorization Machine Approach for Web Service QoS Prediction. In *ICSOC.* 272–286.
[26] Y. Yin et al. 2019. QoS Prediction for Mobile Edge Service Recommendation with Auto-Encoder. *IEEE Access* 7 (2019), 62312–62324.
[27] Y. Yin et al. 2020. QoS Prediction for Service Recommendation with Features Learning in Mobile Edge Computing Environment. *IEEE TCCN* 6, 4 (2020), 1136–1145.
[28] C. Yu and L. Huang. 2017. CluCF: A Clustering CF Algorithm to Address Data Sparsity Problem. *SOCA* 11, 1 (2017), 33–45.
[29] Y. Zhang et al. 2021. Location-Aware Deep Collaborative Filtering for Service Recommendation. *IEEE TSMC: Systems* 51, 6 (2021), 3796–3807.
[30] Z. Zheng et al. 2011. QoS-Aware Web Service Recommendation by Collaborative Filtering. *IEEE TSC* 4, 2 (2011), 140–152.
[31] Z. Zheng et al. 2013. Collaborative Web Service QoS Prediction via Neighborhood Integrated Matrix Factorization. *IEEE TSC* 6, 3 (2013), 289–299.
[32] Z. Zheng et al. 2014. Investigating QoS of Real-World Web Services. *IEEE TSC* 7, 1 (2014), 32–39.
[33] Z. Zheng et al. 2020. Web Service QoS Prediction via Collaborative Filtering: A Survey. *IEEE TSC* (2020), 1–1. https://doi.org/10.1109/TSC.2020.2995571
[34] G. Zou et al. 2018. QoS-aware Web Service Recommendation with Reinforced Collaborative Filtering. In *ICSOC.* 430–445.

APPENDIX-A

As mentioned in Section 3 of the main paper, trends in the QoS profile of various services are shown in Figure 5. We have chosen five random services from WS-DREAM-1 dataset [32] to investigate the variation of the QoS profile of various services.

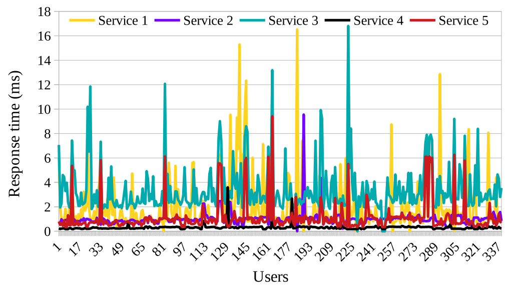


Figure 5: Response time of 5 different services from WS-DREAM-1 dataset [32]

APPENDIX-B

The brief details of the dataset employed for our experiments are presented in Table 10, as mentioned in Section 4.1 of the main paper.

Table 10: WS-DREAM-1 [32] dataset description

Parameters	Values
No. of users	339
No. of services	5825
QoS parameters	Response Time (RT), Throughput (TP)
Total entries in Q	1974675
No. of valid entries in Q: (RT, TP)	(1873838, 1831253)
RT: (min, max, mean, stdev)	(0.001, 19.999, 0.9086, 1.9727) seconds
TP: (min, max, mean, stdev)	(0.004, 1000, 46.5617, 110.7971) kbps

APPENDIX-C

The configuration details of different networks used in OffDQ are provided in Table 11, as mentioned in Section 4.2 of the main paper.

APPENDIX-D

In Table 12, we summarize major related works from the perspective of handling various challenges discussed in the main paper. Although a generic type of QoS prediction (shown in the second column of Table 12) tackles the marked challenges (shown by ✓) while ignoring the rest of the challenges, a specific method of that type may emphasize a particular unmarked challenge separately.

Table 11: Configuration of OffDQ

Configuration of N_{Agg}			
Optimizer	RMSprop [8]	No. of epochs	500
No. of layers	3	No. of neurons	20; 10; 1
Activation functions	$\mathcal{S}$; $\mathcal{S}$; ℓ	Loss function	MSE [7]

Parameters	Configuration of $N_{\mathcal{F}}$	Configuration of $N_{\mathcal{G}}$
Loss function	MSE	MSE
Optimizer	Adam [9]	Adam
No. of layers	5	4
No. of neurons	64; 32; 16; 8; 1	64; 32; 16; 1
No. of epochs	500	500
Activation functions	$\mathscr{T}$; $\mathscr{T}$; $\mathscr{T}$; $\mathscr{T}$; ℓ	$\mathscr{T}$; $\mathscr{T}$; $\mathscr{T}$; ℓ

Parameters	Configuration of AE_r		Configuration of AE_c	
	E_r	D_r	E_c	D_c
Regularization	L_2 [7]	L_2	L_2	L_2
No. of layers	4	4	3	3
No. of neurons	4096; 1024; 256; 100	100; 256; 1024; 4096	512; 256; 100	100; 256; 512
Activation functons	$\mathscr{R}$; $\mathscr{R}$; $\mathscr{R}$; $\mathscr{R}$	$\mathscr{R}$; $\mathscr{R}$; $\mathscr{R}$; ℓ	$\mathscr{R}$; $\mathscr{R}$; $\mathscr{R}$	$\mathscr{R}$; $\mathscr{R}$; ℓ
Epoch	500		500	
Loss function	Custom loss function [Table 5]		Custom loss function [Table 5]	
Optimizer	Adam		Adam	

Configuration of OffDQ			
Optimizer	RMSprop	No. of epochs	1000
No. of layers	4	No. of neurons	100; 50; 10; 1
Activation functions	$\mathcal{S}$; $\mathcal{S}$; $\mathcal{S}$; ℓ	Loss Function	MSE

Activation functions [7]: $\mathcal{S}$: sigmoid ; $\mathscr{T}$: tanh ; ℓ: linear ; $\mathscr{R}$: ReLU

Table 12: Related works and their standing on achieving three major goals

Model	Type	*Goal*-1					*Goal*-2		*Goal*-3
		C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
Memory -based CF	Similarity [17, 30], [1, 21, 24, 34]				✓				
Model -based CF	Clustering [19, 28]				✓				✓
	MF [4, 13, 31]		✓			✓			✓
	FM [12, 25]		✓	✓		✓			✓
	DA [20, 23, 26] [27, 29]		✓			✓		✓	✓
Hybrid	online [2, 14]		✓	✓	✓	✓			✓
	offline [OffDQ]	✓	✓	✓	✓	✓	✓	✓	✓