Final Report on Recommendation Systems

Final project for Machine Learning 2023-2024

1st Hermes P. Katsaounis, 2532
Electrical & Computer Engineering
University of Thessaly
ekatsaounis@uth.gr

2nd Ioannis Kallergis, 3472
Electrical & Computer Engineering
University of Thessaly
ikallergis@uth.gr

3rd Nikolaos Zarifis, 3514
Electrical & Computer Engineering
University of Thessaly
nzarifis@uth.gr

***Abstract*—In this hustling world, entertainment becomes the soul release for a lot of people. That, especially after the Covid-19 lockdowns, has led to a rise in the movie streaming industry. With this study, we hope to learn more about the algorithms that enable people to navigate an exponentially rising volume of content. Even though recommender algorithms are fairly widespread now, there is still room for improvement and optimization.**

For the purposes of this study, we experimented with some collaborative filtering algorithms like Singular Value Decomposition (SVD), some variations of the Nearest Neighbor algorithm and others. In addition, we dipped our toes in the field of Neural Networks with an Autoencoder system and some others. As we will analyze further below, the best of the collaborative filtering algorithms was proven to be a Singular Value Decomposition algorithm, specifically SVD++, something that we expected, as it is the most popular in the industry for sparse datasets, like movie critics, but also the use of an Autoencoder Neural Network Model is examined which can be used as part of a collaborative filtering algorithm.

In this paper, we propose a hybrid model, which combines the predictions from an Autoencoder Neural Network Model with the results of data analysis to find objectively good movies, in order to provide recommendations that are liked both by the general public and the user.

***Index Terms*—Recommender System, Singular Value Decomposition, Autoencoders, Movie recommendations**

I. Introduction

The amount of information generated by the internet is growing at an overwhelming rate [1]. The fact that more products and services are available to consumers, now more than ever, has not only compounded the problem, but has also increased its scope and diversity [2]. Recent psychological research implies that at some point, the abundant choice becomes overwhelming and leads to greater consumer delay and lower consumer satisfaction. Especially in the audiovisual domain like movies, there is an actual problem where good movies struggle to find an audience due to the overwhelming uncertainty that comes with the volume of content available. Unlike shorter forms of audiovisual content, movies are considered, by most people, to be a waste of precious time, if they are not entertaining.

By introducing a recommender system, we filter content based on user preferences and minimize wasted time. More specifically, movie recommendation algorithms can save people time by narrowing their choices and making it easier to find a film that they would enjoy. Additionally, these systems can help people discover titles that a less sophisticated system wouldn't recommend otherwise. They can also serve as virtual film critics, steering users away from low-quality films and towards those that have better reviews. Thus, the extreme monopolization that we can see with brands like Paramount Pictures, 20^{th} Century Studios and Disney can be lessened through the use of recommender systems in, the increasingly popular, streaming services.

In practice, movie recommendation systems are computer programs that employ data science and artificial intelligence to suggest films that a user might be interested in watching. These systems analyze user data such as past film-watching preferences, genre preferences, and related film metadata to make movie recommendations. Additionally, they often incorporate ratings from various sources such as professional critics, user-generated reviews, and viewing patterns on streaming platforms, to ensure that users receive accurate, reliable recommendations. By leveraging machine learning and natural language processing algorithms, movie recommendation systems can provide users with thoughtful, personalized film recommendations in seconds.

Contrary to popular belief, recommender systems have a long history, dating back to the early days of information retrieval and information filtering research. The "Grundy" system, created in 1979 by J. Patrick Murphy at Bellcore (now Telcordia Technologies), is one of the earliest instances of a recommender system. Using a combination of keywords, topics, and user preferences, this system was designed to automatically generate a personalized news service that would summarize news articles that were relevant to the user. Another early example of a recommender system is the "Tapestry" system, which was developed in 1992 by researchers at Xerox PARC [3], and used a combination of content-based filtering and collaborative filtering to present recommendations to users. The term "Collaborative filtering" was first introduced in a paper by Paul Resnick and others in 1994 [4], which details a system that uses collaborative filtering to filter Usenet news articles.

In the late 1990s and early 2000s, the growth of the internet and the increasing availability of vast amounts of data led to a renewed interest in recommender systems. The Netflix Prize, a competition to improve the accuracy of the Netflix movie recommendation system, was launched in 2006 and was won in 2009 by a team that used an ensemble of matrix factorization techniques, including Singular Value Decomposition (SVD). This increased the popularity and success of the recommendation systems. With the advent of big data and machine learning, recommender systems have become more sophisticated and have been able to handle large and complex datasets. Today, recommender systems are widely used and are considered to be a critical component of many online platforms and services. They are widely utilized in e-commerce, social networking, news recommendation, and several other fields.

In our case, on the user-level, after conducting many tests using the MovieLens dataset [5], we too came to the conclusion that the best performance among the currently available collaborative filtering algorithms for recommender systems was achieved with the SVD++ algorithm. As input, we use some movie-specific information and the output is a predicted rating by a user for a movie. In the objective part, we use the ratings to compute the average rating and the total number of ratings for each movie. Thus, we are able to assign a *weight* for each movie, in order to compare them.

It's no mystery why Neural Networks have become such a popular algorithm. Their accuracy and speed are usually competitive with the best ML algorithms in most applications, and additionally their flexibility gives them a remarkable range of applications from Classification to Computer Vision and Generative models. The actual best MSE results in our experiments were by far achieved by the Neural Network models and especially the Autoencoder we designed. In addition to impressive accuracy and speed that Neural Networks provide, the Autoencoder model gives us the ability to compress the encoding part and leave the decoding part to be retrained on new data. Thus improving a lot real world retaining needs.

II. Literature Review

A. Hybrid Collaborative Filtering

[6] presents a hybrid model that combines improved K-means clustering and genetic algorithms (GAs) to partition transformed user space. They started by utilizing the PCA technique to compress feature data into a relatively small and dense space, and then using the transformed user space to develop an effective GA-KM (genetic algorithm k-means) clustering algorithm.

Another approach is [7], which focuses on movie recommendation through data clustering and computational intelligence. More specifically, it's a hybrid cluster and optimization-based technique that makes use of k-means clustering by adopting a Cuckoo search optimization algorithm. cuckoo search is an optimization algorithm developed by Xin-She Yang and Suash Deb in 2009 [**?**]. It is based on the behavior of the cuckoo bird, which lays its eggs in the nests of other birds. In the algorithm, a population of "cuckoos" is used to explore the search space, and the best solutions are "stolen" from other birds' nests. The algorithm is inspired by the Lévy flight behavior of cuckoos, which is characterized by a random walk with long steps in a random direction. CS has been used in a variety of optimization problems, such as function optimization, feature selection, and machine learning. It is also a global optimization algorithm, which means that it is able to find the global optimal solution in the search space rather than getting stuck in local optima.

[8] provides a summary of various types of recommendation methods based on user preferences, ratings, domain knowledge, user demographic information, and user context. It also, suggests a method for recommending movies that uses collaborative filtering and singular value decomposition plus-plus (SVD++).

Collaborative filtering's key benefit is its ability to accurately predict product ratings or preferences without relying on an objective measure of similarity. It is independent of how the user's information is analyzed [9]. It can generalize across comparable users and offer more precise recommendations in cases where ratings data is sparse. Additionally, collaborative filtering is adaptive and efficient, because it can quickly update recommendations in real time as new ratings come in.

On the other hand, one of the main drawbacks of collaborative filtering is that it can be vulnerable to the "cold start" problem, which occurs when there is insufficient data about a user or item for the system to provide accurate recommendations i.e., it can't handle new items or new users. Additionally, it can suffer from data sparsity, meaning that when there are few ratings for many items, the accuracy of the system can be adversely affected [10]. Furthermore, collaborative filtering can generate inaccurate recommendations if users give ratings that are not consistent with their real preferences.

B. Deep Neural Networks

An interesting approach that uses a Hierarchical Bayesian Neural Network, in an attempt to include complex data in the model, was introduced in [11]. They established the term collaborative deep learning (CDL), which combines deep representation learning for the content information and collaborative filtering for the ratings' matrix. CDL is innovative since it's the first hierarchical Bayesian model to bridge the gap between state-of-the-art deep learning models and Recommender Systems.

[12] proposes a switching hybrid recommender system using two different recommender system techniques. They employ OpenAI's Proximal Policy Optimization algorithm and transition to an autoencoder-based content filtering system to address the cold start problem.

Deep learning can be beneficial for movie recommendation systems in various ways. Notably, deep learning algorithms can be used to better understand the users' preferences and interests. This allows the algorithms to generate more personalized and accurate recommendations. Secondly, deep learning can help to identify key features in the movies that can be used to improve the accuracy of recommendations. As for Deep Neural Networks' structure, it allows them to include various types of data like, in our subject, stills and clips of the movies, as well as, scripts and critics. Finally, deep learning algorithms are usually good at handling sparse datasets [13].

In terms of disadvantages, there are a few drawbacks to using deep learning for movie recommendation systems. Mainly, it can be computationally expensive, as the algorithms require large amounts of data to train and can take a long time to execute. Additionally, the complexity of deep learning algorithms may make them prone to errors, leading to incorrect recommendations or even worse false positives. Additionally, it should be noted that deep learning recommendations can often be complex to explain and vulnerable to "black-box" type problems.

Lastly, as we focused our Deep learning endeavors on an Autoencoder, it is important to present the cutting edge research in this field. In their paper **Collaborative Variational Autoencoder for Recommender Systems** [14], Xiaopeng Li and James She are blurring the limits between Collaborative and Content-Based filtering with their collaborative variational autoencoder (CVAE) model that considers both rating and content. Kapil Saini and Ajmer Singh propose an interesting approach to Recommender Systems [15] In advanced Recommendation systems it is important to consider temporal dependencies and patterns,

Name	Date Range	Rating Scale	Users	Movies	Ratings	Tag Apps
ML 100K	9/1997–4/1998	1–5, stars	943	1,682	100,000	0
ML 1M	4/2000–2/2003	1–5, stars	6,040	3,706	1,000,209	0
ML 10M	1/1995–1/2009	0.5–5, half-stars	69,878	10,681	10,000,054	95,580
ML 20M	1/1995–3/2015	0.5–5, half-stars	138,493	27,278	20,000,263	465,564
ML 25M	1/1995–11/2019	0.5–5, half-stars	162,541	62,423	25,000,095	1,093,360

TABLE I: Summary of the MovieLens Datasets.

Fig. 1: Number of movies for each number of stars

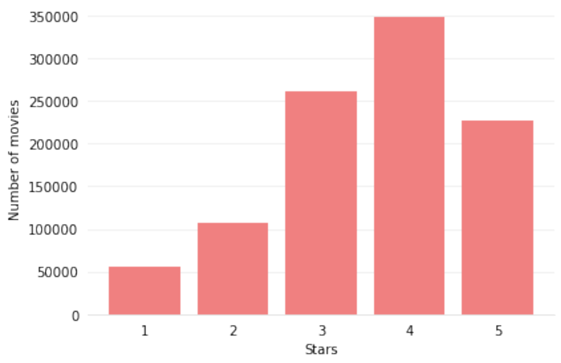
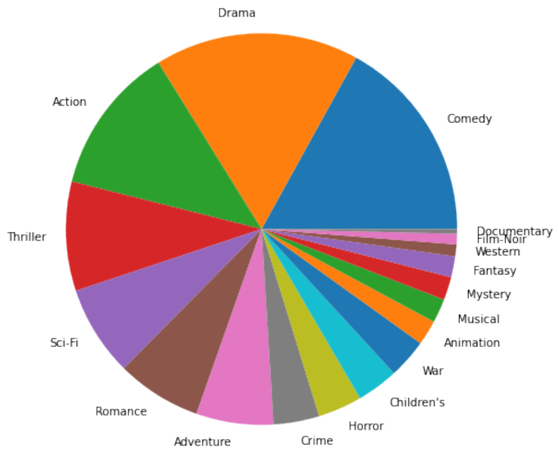


Fig. 2: Popularity of genres



which introduces us to the vanishing/exploding gradient problem, by passing stable sequential data to the system leads to extreme weights. By designing an Autoencoder based on a Stacked Long Short-Term Memory (LSTM) architecture. Long Short-Term Memory (LSTM) networks are a type of recurrent neural network architecture designed to capture and learn long-range dependencies in sequential data, addressing the vanishing exploding gradient problem through memory cells and gates (building blocks of the LSTM algorithm).

III. Dataset and Features

In our approach, we used the popular *MovieLens* dataset [5]. There is a variety of options for the size of it [16], as it is explained in the summary statistics for the different MovieLens datasets in *Table 1*. For our experiments, we used the 100k and 1M datasets, because, on the bigger ones, some methods needed so much space that was leading to memory error, and others were taking a forbiddingly long time to execute. For the final model, we chose the 1M dataset, in order to obtain as much info as we could.

The data we used was conducted by the following csv files:

1) Ratings

Each line of this file after the header row represents one rating of one movie by one user. It contains the features userId (ID of the user), movieId (ID of the movie), rating (5-star scale), and timestamp (seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970). A distribution of movies between the different ratings, derived from the 1M dataset, is shown in *Fig.1*.

2) Movies

Each line of this file after the header row represents one movie and includes information about the title and the genres. The title includes the year of release in parentheses. Genres are selected from the following:

- Action
- Adventure
- Animation
- Children's
- Comedy
- Crime
- Documentary
- Drama
- Fantasy
- Film-Noir
- Horror
- Musical
- Mystery
- Romance
- Sci-Fi
- Thriller
- War
- Western
- (no genres listed)

and a comparison between them, on the 1M dataset, based on how many movies are included in each category, is shown in *Fig.2*.

On the Collaborative Filtering part, we used the features userId, movieId, and rating as input to the SVD++ algorithm. The matrix that is created by the algorithm is formed by movies as columns and users as rows. As the dataset increases in size, its density, i.e. the percentage of cells in the full user-item matrix that contain rating values, decreases [16]. Hence, as it is expected, our matrix is sparsely populated.

For the general recommendations, based on the popularity of the movies, we used the *rating.csv*, in order to calculate the average rating and the number of ratings for each movie. Hence, after some calculations that will be explained in the next chapter, we were able to generate a *weight* for each movie, that expresses how objectively good a movie is, in ascending order. Thus, this is the key feature to extracting the best movies.

Overall, for the final recommendations, we used the *genres*, in order for the user to choose between the different categories. Hence, it's possible to receive recommendations depending on the genre of the movie or not. Additionally, we added the *title* in the final dataframe, so it can be displayed on the recommendation list.

The data were really well organized and didn't need any feature engineering to be useful in training our model. For evaluation, we used the cross-validation method with 10 folds on every dataset, i.e. 10% of the dataset is used for testing.

IV. Methods

We explored many algorithms to acquire a comprehensive overview of the subject. Most of them were part of the surprise library, thus making it easier to compare them.

A. Nearest Neighbor algorithm

Neighborhood methods calculate how similar the users or the items are in the known ratings' matrix. The similarity scores or the measure of proximity are used in neighborhood-based methods to identify similar users and items [17]. The general approach for neighborhood-based methods consists of two steps to find the predicted rating for a user-item:

1) Find the cohort of other similar users (or items) who have rated the item (or the user) in question.
2) Deduce the rating from the ratings of similar users (or items).

After some comparisons with variations of k-NN (Nearest Neighbor algorithm) included in the surprise library, the minimum RMSE (Root Mean Squared Error) we managed to achieve was with the k-NN Baseline function. The main difference between the k-NN Baseline and other variants of k-NN is that it uses the baseline ratings as an additional feature when finding the nearest neighbors. This helps to improve the accuracy of the recommendations by accounting for the bias of users and items.

The baseline ratings for each user, b_u, and each item, b_i, are computed as the mean of all ratings provided by the user and received by the item, respectively. The baseline rating prediction for a user-item pair (u, i) is computed as

$$\hat{r}_{ui} = b_u + b_i - \mu$$

where μ is the overall mean rating. Once the baseline ratings have been computed, the k-NN prediction is made by finding the k nearest neighbors to a user-item pair (u, i), using a similarity measure. The most common similarity measure used in the Surprise library is the cosine similarity, which is defined as:

$$sim(u, v) = \frac{\sum_{i \in I_{uv}} r_{ui} * r_{vi}}{\sqrt{\sum_{i \in I_{uv}} r_{ui}^2} * \sqrt{\sum_{i \in I_{uv}} r_{vi}^2}}$$

where I_{uv} is the set of items that are rated by both users u and v.
Once the k nearest neighbors have been found, the rating prediction for the user-item pair (u, i) is computed as:

$$\hat{r}ui = \hat{r}ui + \frac{\sum_{v \in N} sim(u, v)(r_{vi} - \hat{r}vi)}{\sum v \in N sim(u, v)}$$

where N is the set of k nearest neighbors and $\hat{r}_{vi}$ is the baseline rating prediction for the user-item pair (v, i). As is shown in more detail in [18]

B. Slope One

Slope One is a simple recommendation algorithm that was first introduced in 2005 by Daniel Lemire and Anna Maclachlan. [19] The algorithm is based on the idea of predicting the difference between the ratings of two items, rather than predicting the ratings themselves. The prediction for a given item is made by averaging the differences between its ratings and the ratings of other items. The algorithm stores a matrix of these differences, known as the "deviation matrix," which is used to make predictions. The basic idea behind Slope One is that the deviation between two items is more stable than the actual ratings, and can be used to make recommendations even when there is limited data available. This simplicity of the algorithm makes it computationally efficient and easy to implement, making it a popular choice for many recommendation systems.

C. Baseline Recommendations

BaselineOnly() is a simple algorithm in the Surprise library for recommendation systems that only considers baseline estimates. It computes the average of the ratings for each user and item, and then uses these averages to estimate the ratings for all unknown entries. This algorithm makes use of the fact that most users have a tendency to rate items with a certain baseline score, and it tries to capture this by estimating the average ratings. The algorithm then predicts the rating for a user-item pair by taking the sum of the user and item baseline estimates, resulting in a simple yet effective recommendation algorithm for small datasets.

D. Singular Value Decomposition

We also tested Singular Value Decomposition (SVD), a matrix factorization algorithm based in the paper [20]. The SVD is a generalization of the eigen-decomposition which can be used to analyze rectangular matrices. By analogy with the eigen-decomposition, which decomposes a matrix into two simple matrices, the main idea of the SVD is to decompose a rectangular matrix into three simple matrices: Two orthogonal matrices and one diagonal matrix

Given a user-item matrix $R(m \times n)$ of ratings, SVD factorizes it into the product of three matrices:

$$R = U \times S \times V^T$$

where $U(m \times k)$ is a user-to-feature matrix, $S(k \times k)$ is a diagonal matrix of singular values, and $V^T(k \times n)$ is a feature-to-item matrix.

The columns of U and V are called latent features, which are the underlying factors that determine the ratings. The diagonal elements of S are called singular values, which represent the importance of each latent feature. The latent features are computed by solving the following optimization problem:

$$\min_{U,V} \|R - USV^T\|_F^2$$

where $\|.\|_F^2$ is the Frobenius norm of a matrix, which is the sum of the squares of its elements.

By using SVD, we can approximate the original matrix R by keeping only the most important latent features, thus reducing the dimensionality of the data. Once the matrices U, S, and V^T

have been computed, the rating prediction for a user-item pair (u, i) can be computed as the dot product of the corresponding row in the matrix U, the corresponding entry in the matrix S, and the corresponding column in the matrix V^T. The prediction is represented by the following equation:

$$\hat{r_{ui}} = U_u \times S \times V_i^T$$

where $\hat{r}_{ui}$ is the predicted rating for the user-item pair (u, i), U_u is the row of the matrix U corresponding to user u, S is the diagonal matrix of singular values, and V_i^T is the column of the matrix V^T corresponding to item i.

We also tested the Biased SVD (SVD++), which is an extension of SVD specifically designed to improve the accuracy of recommendation systems. It includes a bias term for both users and items. This bias term allows the algorithm to account for the different rating tendencies of users and items. The bias term is represented by b_u for users and b_i for items, so the above equation becomes:

$$\hat{r_{ui}} = U_u \times S \times V_i^T + b_u + b_i$$

E. Popularity

To use in the biased algorithms like SVD++ and to enhance their effectiveness, we opted to create a basic popularity metric for every movie in the dataset. To achieve that, we assigned *weights* to all the movies [21]. The bigger the weight, the better the movie. At first, we calculated the average rating and the number of votes for each movie. Making use of those values, we applied the following formula:

$$\left(\frac{v}{v+m} \cdot R\right) + \left(\frac{m}{v+m} \cdot C\right) \tag{1}$$

where

- v is the number of votes for the movie
- m is the minimum votes required to be listed in the chart
- R is the average rating of the movie
- C is the mean vote across the whole report

We decided to set m at 0.8, meaning that, for a movie to feature in the charts, it must have more votes than at least 80% of the movies in the list.

F. Neural Networks

Artificial neural networks (ANN, also shortened to neural networks (NN)) are a branch of machine learning based around interconnected nodes known as artificial neurons, which loosely mimic biological brain neurons. Artificial neurons process incoming signals using a non-linear activation function, and their outputs are generally determined by the sum or some other function of their inputs. Connections, referred to as edges, have adjustable weights that change during learning, impacting signal strength (real number magnitude). Neurons in some architectures trying to minimize complexity feature a threshold, ensuring signals are sent only when the aggregate signal surpasses it.

Deep learning is an extension of ANNs that utilizes multiple layers to progressively extract higher-level features. Image processing is a good example to explain this, as there is a visual component. In lower layers we try to detect edges and basic shapes, while higher layers recognize more complex concepts like digits, letters, or faces. Crucially, in a deep learning process, the system autonomously discovers the optimal placement of features within different levels. However, this doesn't eliminate the necessity for manual adjustments. For instance, adjusting the number of layers and their sizes can yield varying degrees of abstraction.

1) The Keras library: Keras is a versatile high-level neural networks API that stands out for its ease of use and flexibility. Developed by François Chollet [22], Keras can operate on top of various low-level libraries, including TensorFlow [23], or Theano [24]. It provides a modular and user-friendly interface, making it accessible for both beginners and experienced deep learning practitioners. The library simplifies the process of constructing neural networks through a collection of pre-built layers, activation functions, and optimization algorithms. Its abstraction allows developers to focus on the model's architecture rather than the underlying implementation details. Keras supports a wide range of neural network types, including feedforward networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and more. Additionally, its integration with TensorFlow ensures compatibility with advanced functionalities, such as distributed computing and GPU acceleration.

2) Activation Functions in Keras: Keras provides a variety of activation functions that can be used to dictate the behavior of each neuron in neural networks. These functions introduce non-linearities to the network, enabling it to learn complex patterns. Here are some commonly used activation functions:

- **ReLU (Rectified Linear Unit):** The ReLU activation function is widely used for hidden layers. It outputs the input if it is positive, otherwise, it outputs zero.

$$f(x) = (x)^+ = max(0, x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{2}$$

- **Sigmoid:** Sigmoid is often used in the output layer for binary classification problems, where the goal is to produce a probability between 0 and 1.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

- **Tanh:** Tanh squashes the output to be between -1 and 1, making it suitable for situations where the data has negative as well as positive values.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4}$$

- **Softmax:** Softmax is often used in the output layer for multi-class classification problems. It converts the network's raw output into probabilities for each class.

$$f(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{5}$$

- **Leaky ReLU:** Leaky ReLU is similar to ReLU but allows a small, non-zero gradient when the input is negative, preventing dead neurons.

$$f(x) = \max(\alpha x, x) = \begin{cases} ax & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{6}$$

  , where α is a small positive constant

- **Exponential Linear Unit (ELU):** ELU is an alternative to ReLU, introducing a smooth curve for negative inputs.

$$f(x) = \begin{cases} a(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{7}$$

  , where α is a small positive constant

3) Optimizers in Keras: Below, we list some of the most popular optimizer algorithms that are used to minimize the loss function during the training of neural networks. These optimizers adjust the weights of the network in order to improve its performance.

- **Stochastic Gradient Descent (SGD):** SGD is a classic optimization algorithm that improves the speed of Gradient Descent by using a small random subset of the samples to recalibrate the fitting line. This way, especially in big datasets, we reduce the computations for each step by a lot.

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t) \tag{8}$$

- **Adam:** Adam (short for Adaptive Moment Estimation) is an adaptive learning rate optimization algorithm. It combines ideas from both momentum and RMSprop. Adam maintains two moving averages, m_t for the gradient and v_t for the squared gradient. The algorithm adapts the learning rates for each parameter individually.

$$m_{t+1} = \beta_1 \cdot m_t + (1 - \beta_1) \cdot \nabla J(\theta_t) \tag{9}$$

$$v_{t+1} = \beta_2 \cdot v_t + (1 - \beta_2) \cdot (\nabla J(\theta_t))^2 \tag{10}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_{t+1}} + \epsilon} \cdot m_{t+1} \tag{11}$$

- **Nadam:** On part of the ADAM optimizer, NADAM (short for Nesterov-accelerated Adaptive Moment Estimation) is an extension of the ADAM optimizer mixed with Nesterov Momentum. Nesterov Momentum is a technique that can improve the convergence speed of stochastic gradient descent, a popular optimization algorithm used to train machine learning models. To implement, we have to know about the value of v which indicates the momentum term and momentum is a hyperparameter that controls the strength of the momentum. A value of momentum = 0 corresponds to regular SGD, while larger values of momentum correspond to more aggressive momentum.

$$v = momentum \cdot v - lr \cdot grad(w) \tag{12}$$

$$w = w + v \tag{13}$$

  In practice, Nesterov Momentum is often used in combination with other techniques such as learning rate scheduling and mini-batch training. It is an essential tool in the machine learning practitioner's toolkit and is widely used in deep learning and other areas of machine learning.

- **NAG:** To stay into the Nesterov region there also exists NAG (short for Nesterov Accelerated Gradient). NAG is an optimization technique that is developed to solve the slower convergence of momentum optimizers as weight update occur with 2 terms history velocity and gradient at a point in a single step.

$$update_i = \gamma \cdot update_{t-1} + \eta \nabla w_t \tag{14}$$

$$w_{t+1} = w_t - update_t \tag{15}$$

- **RMSprop:** Root Mean Square Propagation (RMSprop) is an optimization algorithm that adjusts the learning rates of each parameter individually. It uses a moving average of squared gradients to scale the learning rates. The parameter β controls the decay rate of the moving average.

$$v_{t+1} = \beta \cdot v_t + (1 - \beta) \cdot (\nabla J(\theta_t))^2 \tag{16}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_{t+1}} + \epsilon} \cdot \nabla J(w_t) \tag{17}$$

- **Adagrad:** Adagrad is an adaptive learning rate optimization algorithm that adapts the learning rates based on the historical gradient information. It maintains a sum of the squares of past gradients for each parameter. Adagrad is effective for sparse data but may lead to overly aggressive learning rates in later stages of training.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}} + \epsilon} \cdot g_{t,i} \tag{18}$$

- **Adadelta:** Adadelta is an extension of Adagrad that addresses its diminishing learning rates over time. Instead of maintaining a sum of squared gradients, Adadelta uses a running average of both squared gradients and squared parameter updates. This allows it to adapt the learning rates without explicitly storing historical gradients.

$$\text{RMS}[\Delta\theta]_t = \beta \cdot \text{RMS}[\Delta\theta]_{t-1} + (1 - \beta) \cdot (\Delta\theta_t)^2 \tag{19}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\text{RMS}[\nabla J(\theta)]_t + \epsilon}} \cdot \nabla J(\theta_t) \tag{20}$$

- **Eve:** Epoch-wise Variance Estimation (Eve) is an optimization algorithm designed to intesify the training of neural networks. On the contraty of mostly used optimizers that maintain a learning rate that is constant, Eve algorithm dynamically adapts learning rates for every parameter while training is happening based on the estimated variance of the gradients.

$$\theta_{t+1} = \theta_t - at \frac{\widehat{m}_t}{\sqrt{\widehat{u}_t} + \epsilon} \tag{21}$$

V. Experiments/Results/Discussion

A. Experiments

To choose the suitable algorithm for our recommender system, we opted to test some of the most popular ones included in the Surprise library [25]. We used two varieties of the MovieLens dataset, in order to analyze how each algorithm scales with the dataset's size and how sensitive they are to their density. To compare the algorithms, we used the Surprise function *cross_validate()*, where the dataset is split into train and test sets through a k-fold method, thus giving us a really accurate measure of the expected performance of each algorithm. For all methods, we chose to use 10 folds (k = 10), because both datasets have a very big number of instances, so it should be sufficient to reliably test our models. Additionally, cross-validation is a good choice, as its results are unfazed by the frequent problem of overfitting [26]. For the Neural network a k-folds approach is not fitted as in the training posses something similar happens by splitting the data in batches. Despite that, we implemented a similar function to have the same metrics and same accuracy to the rest of the experiments.

As for the evaluation metrics, we used the *Root Mean Square Error (RMSE)* and *Mean Absolute Error (MAE)*. RMSE is the standard deviation of the residuals (prediction errors). Residuals are a measure of how far from the regression line, data points are; RMSE is a measure of how spread out these residuals are. In other words, it tells us how concentrated the data is around the line of the best fit [27]. The formula is:

$$RMSE = \sqrt{\frac{\Sigma_{i=1}^{N}||y(i) - \widehat{y}(i)||^2}{N}} \tag{22}$$

where N is the number of data points, y(i) is the i^{th} measurement, and $\widehat{y}(i)$ is its corresponding prediction.
RMSE is an effective metric for evaluating recommendation systems, as it gives an indication of how accurately the system is able to predict ratings. Additionally, it is easy to interpret since it is directly related to the errors made by the system.
MAE is the average of all absolute errors, where absolute error is the difference between the measured value and true value [28]. The formula is:

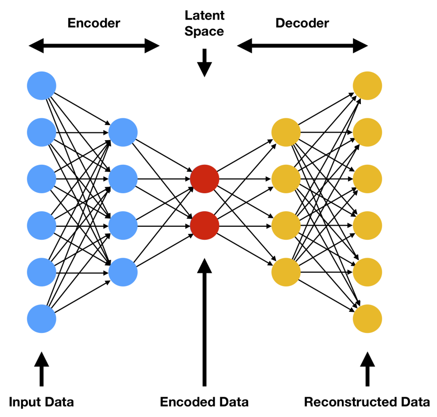
$$MAE = \frac{1}{n}\Sigma_{i=1}^{n}|x_i - x| \tag{23}$$

where n is the number of errors, x_i is the true value, and x is the prediction.
MAE is easy to understand, and it provides an accurate assessment of the performance of a system. It is also easily comparable across different systems, as it gives a clear indication of how close the predictions made by a system are to the actual ratings provided by users.
We also report the cross-validation time in seconds for all methods.

Firstly, we tested k-NN (Nearest Neighbor) algorithms, as they seem to be surprisingly accurate for their low complexity and simplicity. The Surprise library in python provides us with some variations of the algorithm, namely KNNBaseline(), KNNWithZScore(), and KNNWithMeans(). Each one of these choices differs in the way they handle the baseline ratings and normalize the ratings. KNNBaseline uses the baseline ratings as an additional feature when finding the nearest neighbors, it computes the baseline rating prediction for a user-item pair (u, i) as $\hat{r}_{ui} = b_u + b_i - \mu$, where b_u and b_i are the means of all ratings provided by the user and received by the item, respectively, and μ is the overall mean rating. On the other hand, KNNWithZScore normalizes the ratings using Z-scores, and KNNWithMeans normalizes the ratings using the mean ratings.

We also tested the BaselineOnly(), a really simple prediction algorithm included in Surprise, to give us a baseline against

Fig. 3: Autoencoder symbolic visualization



whom we can compare the rest of the algorithms and the improvements in accuracy they make.

The SlopeOne() algorithm is another simple algorithm that is included in the Surprise library. We included this one for an extra perspective for our measurements, as we did not expect it to perform so well in recommender systems.

Most importantly, we tested the SVD (Singular Value Decomposition) variants available in the Surprise library. SVD and SVD++, or SVDpp as it is noted in Surprise, differ in the way they handle implicit feedback. SVD is a standard matrix factorization method that factorizes the user-item matrix into latent features, represented by the equation $R = USV^T$. On the other hand, SVD++ incorporates both explicit and implicit feedback. It considers the additional information about the item, such as the number of times an item has interacted with a user, to improve the prediction of latent features. More specifically, we tested SVD++ with and without cache ratings, meaning whether to cache ratings during fit(). Caching ratings refers to the process of temporarily storing ratings so that they can be used more quickly when needed. This is especially useful when a system needs to process a large quantity of data quickly and efficiently. Caching ratings helps speed up the training, reduce latency and improve overall performance.

Finally, we conducted an evaluation of an **Autoencoder** neural network, an architecture commonly employed for applications akin to recommendation systems. Autoencoders consist of two symmetrical models (**Encoder**,**Decoder**). The encoder part consists of a funnel like deep neural network, meaning that the number of outputs is significantly smaller than its input, thus the name Encoder. This type of neural network forces the model to make generalizations about the data. Now that the network has extracted important patterns from the dataset, we need a way to explode those patterns to fill the ratings for each person about all the movies. To do that, we use a symmetric to the encoder model. This algorithm can also be designed to accommodate supplementary data from diverse sources, such as implicit feedback, sentiment analysis on critics and audiovisual inputs (trailers, stills etc.). Despite that we were confined to identical input parameters as the other algorithms, namely a matrix of user/item ratings, to have a more level playfield between different algorithms and as the implementation of such a system was beyond our capabilities and computational power.

Fig. 4: Average RMSE and MAE for each method on the 100K dataset.

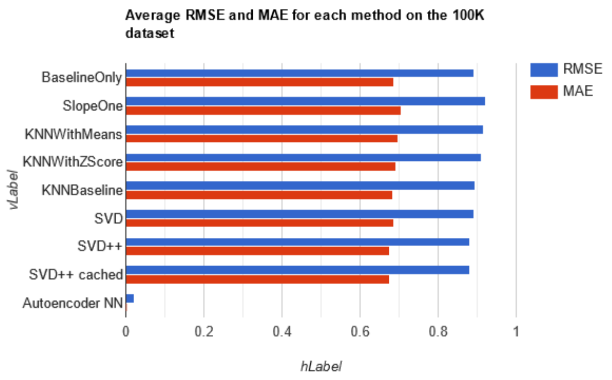
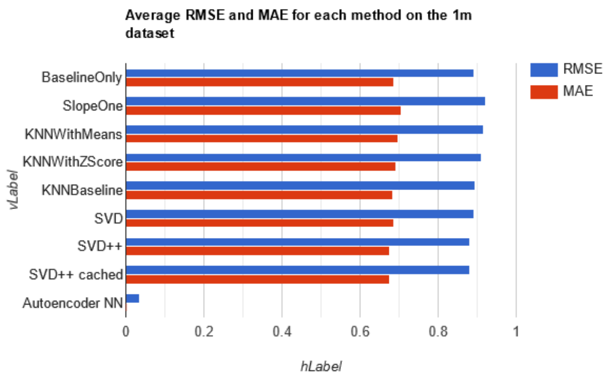


Fig. 5: Average RMSE and MAE for each method on the 1M dataset.



A visual representation of the Neural Network model lies in Fig. 3.

As we were using a cross-validation algorithm based on k folds, we implemented a similar test for our autoencoder too. An important observation, worth mentioning, is that a neuron stopper would be a good idea to help not overfit the data. On top of that, a second encoder provides an elegant way to accelerate the training process of the neural network model. Additionally, as a sparse matrix with mostly null inputs would probably break a system like the one we designed, we opted to fill them with a median value. This introduced a secondary problem where our system would train itself on the fake values. We addressed that by creating a custom loss function where it would only calculate the MSE of the existing values.

B. Results

The numerical results for all methods on both datasets are shown in Tables 2 and 3. The same results are also visualized in Figures 3 and 4.

Method	RMSE	MAE	Total C.V. Time
BaselineOnly	0.8906 ± 0.0055	0.6878 ± 0.0038	**9.86**
SlopeOne	0.9223 ± 0.0057	0.7060 ± 0.0033	99.42
KNNWithMeans	0.9158 ± 0.0060	0.6980 ± 0.0048	18.36
KNNWithZScore	0.9117 ± 0.0080	0.6909 ± 0.0046	19.79
KNNBaseline	0.8958 ± 0.0092	0.6846 ± 0.0065	26.03
SVD	0.8917 ± 0.0083	0.6858 ± 0.0059	25.08
SVD++	0.8811 ± 0.0052	0.6759 ± 0.0036	1066.68
SVD++ cached	$\mathbf{0.8810 \pm 0.0063}$	$\mathbf{0.6749 \pm 0.0057}$	384.92
Autoencoder NN	**0.0355 ± 0.0026**	**0.0013 ± 0.0002**	12.42*

TABLE II: Average RMSE, average MAE and Testing Time (in seconds) for each method on the 100K dataset.

Method	RMSE	MAE	C.V. Time
BaselineOnly	0.9079 ± 0.0013	0.7187 ± 0.0013	137.47
SlopeOne	0.9057 ± 0.0018	0.7038 ± 0.0016	705.72
KNNWithMeans	0.9119 ± 0.0029	0.7175 ± 0.0023	1541.54
KNNWithZScore	0.9089 ± 0.0022	0.7134 ± 0.0016	1597.98
KNNBaseline	0.8899 ± 0.0024	0.7014 ± 0.0018	1670.38
SVD	0.8660 ± 0.0018	0.6792 ± 0.0015	157.88
SVD++	0.8568 ± 0.0019	0.6678 ± 0.0015	5468.10
SVD++ cached	$\mathbf{0.8559 \pm 0.0018}$	$\mathbf{0.6671 \pm 0.0017}$	2834.45
Autoencoder NN	**0.0209 ± 0.0007**	**0.0019 ± 0.0001**	**32.55***

TABLE III: Average RMSE, average MAE and Testing Time (in seconds) for each method on the 1M dataset.

In figure 6. in an attempt to visualize the patterns that the model extracted from the dataset we created the heatmap for a part of the dataset. From that we can see that the model puts much more weight on the general "quality" of each movie compared to the preferences of each user. This is something that we expected as the performance of the rest of the algorithms that focused on the movies (Item-based Filtering) had a noticeable advantage compared to those that focused on the user (Collaborative Filtering). Additionally due to the shape of the dataset, where the users are much fewer than the items, an item-based filtering algorithm would always have an advantage.

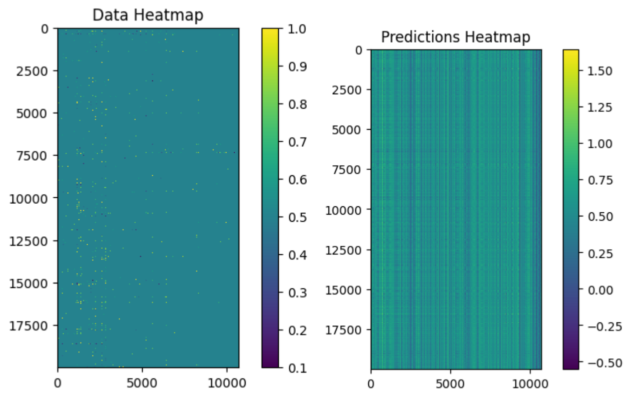


Fig. 6: Heatmap Visualization of the results

C. Discussion

Considering the above results, we used the Autoencoder Neural Network Model that we built to create a recommender system. Considering the results of the experiments with different methods, also shown above, prove the fact that not every model is suitable to handle every task and that development and execution time are parameters that must be taken into consideration. About the previously implemented methods into this dataset, in terms of accuracy, SVD++ had a good performance, both with and without cached ratings, as the difference in error between them was statistically insignificant. SVD++ cached had the lowest RMSE and MAE of all, and it was also much faster than SVD++ without cache, but after the introduction of the Autoencoder into the game, everything seems to change with an extreme drop in losses values. The flexibility of the Autoencoder Neural Network with its ability to adapt not only to the data given to it as input, but also to the changes in itself the programmer does, is a really strong point in its favor. Obviously, not all data are the same, so it sounds logical that there cannot be a single model to handle every task. Consequently, Neural Networks and especially Autoencoder Neural Network Models could be considered a much better choice for handling various tasks not only due to the reasons mentioned above, but also due to the actual experiments and the numbers produced. The comparisons of the results of the experiments from different methods speak for themselves. But not to get too excited, it is vital to mention that the accuracy and the advantages of this model can not make us consider it as optimal, as there are many iterations and optimizations around autoencoders and by extension Neural networks in general, that we haven't tested. An absolute optimal model will, likely, never be created, but new models will always tend to reach optimality.

Among the three variants of k-NN, the difference in time between them was minimum, with KNNBaseline() being the most accurate. The tests concluded with KNNBaseline() giving the lowest mean RMSE (Root Mean Squared Error), as well as mean MAE (Mean Absolute error) for all tests. That is logical, considering its ability to incorporate user bias, item bias, and various metrics.

As for BaselineOnly, it was the fastest on both datasets, but not the most accurate. This is because it is a simple algorithm, as it does not take into account the relationships between users and items, which can limit its performance in recommendation systems.

SlopeOne was also quite fast and had moderate accuracy. It is a relatively simple algorithm, its performance may be limited in scenarios where the relationships between users and items are complex and cannot be captured by simple differences in ratings.

It's also interesting to compare the performances between the two datasets. As we can see, kNN algorithms do not scale well and perform worse and slower as the datasets get larger. They compare each test sample to all training samples in order to find the nearest neighbors, which can become computationally expensive and time-consuming. BaselineOnly is not a good choice for large datasets too, because of its simplicity. On the other hand, all SVD variants, as well as SlopeOne, perform better with large datasets. Lastly, the Autoencoder Neural Network Model according to the number of layers may take more time to train on the same dataset as the other algorithms. The deepest the neural network, the more time it needs to train, but this time is worth the increased accuracy. Thankfully, there exist many open-source libraries and modules that can help store an instance of the trained model and a retraining every time we want to use it is not needed. The models are exposed to a larger training set, leading to a more effective learning. They can take advantage of the increased information to make more accurate predictions.

VI. Conclusion/Future Work

To sum up, Autoencoder Neural Network Model was by far the best-performing algorithm, but also the slowest one. However, with sufficient computing power, the time penalty of the Autoencoder can be insignificant. Thus, simple and fast algorithms like k-NN are rarely used nowadays. With the help of clustered supercomputers and distributed computing, we can use heavy algorithms like Biased SVD and very deep multilayered Neural Networks and focus mostly on the accuracy, as the vast computing power of such systems makes the difference in time performance insignificant, especially in the subject of Recommender Systems, where we don't need to make predictions in real time. Another thing that we realized is that the Autoencoder Model is really economical in terms of memory, as in our commercial machines every other algorithm could not run on datasets larger than 1M, due to lack of memory. We attempted to solve the problem by introducing a lot of virtual memory to our systems, but the penalty made the calculation times range in days.

In conclusion, after the addition of an autoencoder, this type of self-supervised machine learning model could be a better choice than SVD++ (as SVD++ was the leader algorithm of the set of the rest algorithms we experimented with) for a real movie recommendation system with a normal amount of data. Autoencoders, as neural networks, they understand complex patterns and user preferences in a more effective way than traditional SVD++ methods. They adapt well to diverse user behaviors without the need of explicit feature engineering and can handle missing data gracefully.

AI and machine learning belong to a field of computer science that will never stop evolving. It always gets better year after year, day after day, with this evolution increasing exponentially. Most likely, a more optimal algorithm will be found in the future. It is important to keep that in mind and be sure to read articles about new methods and techniques every day and also freshen up the old knowledge, as everything new depends on old knowledge.

VII. Contributions

All our members contributed to the execution of the experiments, as well as the building of the final model (the Collaborative Filtering and the Weighting part) and research for the writing of this paper.

References

[1] Marko Balabanović and Yoav Shoham. Fab: content-based, collaborative recommendation. *Communications of the ACM*, 40(3):66–72, 1997.

[2] Mehmet H Göker and Cynthia A Thompson. Personalized conversational case-based recommendation. In *European Workshop on Advances in Case-Based Reasoning*, pages 99–111. Springer, 2000.

[3] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[4] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, 1994.

[5] Movielens dataset. https://grouplens.org/datasets/movielens/.

[6] Zan Wang, Xue Yu, Nan Feng, and Zhenhua Wang. An improved collaborative movie recommendation system using computational intelligence. *Journal of Visual Languages & Computing*, 25(6):667–675, 2014.

[7] Rahul Katarya and Om Prakash Verma. An effective collaborative movie recommender system with cuckoo search. *Egyptian Informatics Journal*, 18(2):105–112, 2017.

[8] Taushif Anwar and V Uma. Comparative study of recommender system approaches and movie recommendation using collaborative filtering. *International Journal of System Assurance Engineering and Management*, 12(3):426–436, 2021.

[9] Collaborative filtering advantages & disadvantages. https://developers.google.com/machine-learning/recommendation/collaborative/summary.

[10] Jyoti Gupta and Jayant Gadge. Performance analysis of recommendation system based on collaborative filtering and demographics. In *2015 International Conference on Communication, Information & Computing Technology (ICCICT)*, pages 1–6. IEEE, 2015.

[11] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1235–1244, 2015.

[12] Vaibhav Padhye, Kailasam Lakshmanan, and Amrita Chaturvedi. Proximal policy optimization based hybrid recommender systems for large scale recommendations. *Multimedia Tools and Applications*, pages 1–22, 2022.

[13] SM Taheri and Iman Irajian. Deepmovrs: a unified framework for deep learning-based movie recommender systems. In *2018 6th Iranian Joint Congress on Fuzzy and Intelligent Systems (CFIS)*, pages 200–204. IEEE, 2018.

[14] Xiaopeng Li and James She. Collaborative variational autoencoder for recommender systems. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 305–314, 2017.

[15] Kapil Saini and Ajmer Singh. A content-based recommender system using stacked lstm and an attention-based autoencoder. *Measurement: Sensors*, page 100975, 2023.

[16] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

[17] Vijay Kotu and Bala Deshpande. *Data science: concepts and practice*. Morgan Kaufmann, 2018.

[18] Yehuda Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(1):1–24, 2010.

[19] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 471–475. SIAM, 2005.

[20] Ruslan Salakhutdinov and Andriy Mnih. Bayesian probabilistic matrix factorization using markov chain monte carlo. In *Proceedings of the 25th international conference on Machine learning*, pages 880–887, 2008.

[21] Movie recommender systems. https://www.kaggle.com/code/rounakbanik/movie-recommender-systems.

[22] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[23] Google Brain Team. Tensorflow. https://www.tensorflow.org/, 2019.

[24] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th python in science conf*, volume 1, pages 3–10, 2010.

[25] Surprise library. https://surpriselib.com/.

[26] Daniel Berrar. Cross-validation., 2019.

[27] Rmse: Root mean square error. https://www.statisticshowto.com/probability-and-statistics/regression-analysis/rmse-root-mean-square-error/.

[28] Mean absolute error (mae). https://www.statisticshowto.com/absolute-error/.