Limitations of sklearn NMF

By: Travis Reinart

August 12, 2025

Week 4 Part 2 Peer-Graded Assignment
CSCA 5632: Unsupervised Algorithms in Machine Learning

GitHub-Week 4 Limitations of sklearn NMF

View Limitations of sklearn NMF (Live HTML)

Section 1: Introduction

This notebook tackles Part 2 of the Week 4 assignment. The goal is to evaluate the limitations of using standard matrix factorization tools from `scikit-learn`, specifically **NMF** and **TruncatedSVD** for collaborative filtering.

Project Plan:

1. **Data Preparation & Imputation:** Load the MovieLens data and create a dense user-item matrix by filling missing values.
2. **Matrix Factorization Modeling:** Apply both NMF and TruncatedSVD to the imputed matrix to predict ratings.
3. **Evaluation:** Calculate RMSE for both models and compare to Week 3 results.
4. **Discussion:** Explain why this approach is flawed and outline better options for sparse, real-world data.

Note: Reported runtimes are from specific tuning runs and can vary slightly between executions due to system load and runtime conditions. If an observation includes a specific time, it reflects that particular run. The rest of the reported data has been consistent across all runs, but tuning time will always fluctuate to some degree.

1.1 Question from Peer-graded Assignment Instructions

Discuss the results and why they did not work well compared to simple baseline or similarity-based methods we've done in Module 3:

A key issue is that these models cannot handle sparse matrices with missing values, which are standard in recommender datasets. To run them, the missing data must be imputed, in this case replacing over 96% of the matrix with the global mean rating (3.58). While this creates a dense matrix the models can process, it also introduces large amounts of artificial data that dilutes the real signal.

When compared to the top performer from Week 3, **collaborative filtering with Jaccard similarity** on the full ratings matrix (RMSE = 0.9509), both Week 4 models underperformed. The best NMF configuration (k = 30) reached RMSE = 1.018, and the best TruncatedSVD (k = 20) reached RMSE = 1.017, about 0.07 worse than Jaccard. The difference comes from structure: Jaccard uses only observed ratings, preserving the sparse matrix and leveraging its signal, while the sklearn models were forced to "learn" from mostly filler values. Without the ability to work directly on sparse data or include bias terms, NMF and SVD could not match Jaccard's performance.

How to Fix This:

A better approach is to use algorithms designed to work directly with sparse utility matrices, such as ALS (Alternating Least Squares), BiasedMF (which adds user and item bias terms), or implementations in `implicit` or `Surprise` that skip imputation entirely. These methods respect the structure of observed ratings, avoid noise from filler data, and can integrate regularization to improve generalization.

Section 2: Setup and Library Imports

Before beginning, it is important to confirm that all required course dataset files are available in the working directory. These CSV files must be saved locally in the same folder as this notebook (or with paths updated accordingly) for the code to run without errors.

Once the files are in place, I will set up the Python environment by importing the libraries needed for data loading, manipulation, model training, and evaluation. The analysis uses the standard scientific computing stack: NumPy, pandas, scikit-learn, and Matplotlib.

This notebook was developed in Jupyter Notebook within Visual Studio Code.

2.1 Data Sources

Course Dataset: The dataset used in this assignment is a preprocessed subset of MovieLens 1M from Week 3 of the Coursera Unsupervised Algorithms in Machine Learning course. A copy is available in my GitHub repository.

- GitHub MovieLens Dataset CSV Files

Files in Course Dataset:

- `users.csv` : User demographic data.
- `movies.csv` : Movie information, including genres.
- `train.csv` : The training set of user-movie ratings.
- `test.csv` : The test set of user-movie pairs for which ratings need to be predicted.

Original Source: The MovieLens 1M dataset was collected and maintained by the GroupLens Research Project at the University of Minnesota.

- GroupLens MovieLens 1M Dataset

Note: This is the raw dataset in .DAT format, not the preprocessed CSV version used in this notebook.

Dataset Credit and Acknowledgment: The datasets used in this project are based on the MovieLens ratings data collected by GroupLens Research. This course-provided dataset is a derived version of the MovieLens 1M dataset and may not match the original files exactly.

Citation: Harper, F. M., & Konstan, J. A. (2015). The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4), Article 19. https://doi.org/10.1145/2827872

2.2 *Optional* Install Missing Packages

If you get a `ModuleNotFoundError` when running the main import cell, this cell provides the necessary commands to install the required Python libraries.

Instructions:

1. Uncomment the `%pip install` line for the missing package.
2. Run this cell.
3. In the menu, go to **Kernel** → **Restart** before running the main import cell again.

In [36]:
```
# Uncomment and run the lines for any missing packages.
# %pip install pandas
# %pip install numpy
# %pip install matplotlib
# %pip install seaborn
# %pip install scikit-learn
# %pip install tqdm
# %pip install joblib
# %pip install jupyterlab
# %pip install notebook

# Optional Instructions: Export the notebook to HTML or PDF (uncomment and delete the leading slash to run)
# Most Jupyter installations include nbconvert but if the commands below fail you can check your version and/or install if needed
#/ !jupyter nbconvert --version # Uncomment and remove leading / to check your version if needed - as of today is it7.16.6
# %pip install nbconvert # Uncomment and install ncvonvert if needed

# --- Convert Jupyter Notebook .ipynb to .html ---
!jupyter nbconvert --to html "Week4_Limitations_of_sklearn_NMF.ipynb"

# --- Convert Jupyter Notebook .ipynb to .pdf ---
# Note: I left this, but you will get a visually better pdf if you export to .html.  Open in Chrome, print to .pdf.
# Complex plots or highly styled HTML (like the headers in this notebook) will not look exactly the same in .pdf export as they do in the notebook.
#/ !jupyter nbconvert --to pdf "Week4_Limitations_of_sklearn_NMF.ipynb"
```

```
[NbConvertApp] Converting notebook Week4_Limitations_of_sklearn_NMF.ipynb to html
[NbConvertApp] WARNING | Alternative text is missing on 19 image(s).
[NbConvertApp] Writing 3596362 bytes to Week4_Limitations_of_sklearn_NMF.html
```

2.3 Core Library Imports and Diagnostics

This cell imports all core Python libraries required for the project and prints the library versions for reproducibility.

In [2]:

```
# Core Libraries & Utilities
import os, sys, warnings, time, itertools, random, platform
import numpy as np
import pandas as pd
import tqdm
from tqdm.notebook import tqdm as notebook_tqdm
import notebook
import warnings
import json
import joblib
from datetime import datetime

# Visualization
print("-" * 70)
print("--- Importing visualization libraries ---")
print("Matplotlib may build a font cache the first time. It's normal.")
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns

# Modeling and Evaluation
import sklearn
from sklearn.feature_extraction import text
from sklearn.decomposition import NMF, TruncatedSVD
from sklearn.metrics import mean_squared_error
from sklearn.exceptions import ConvergenceWarning

# Jupyter Display Helpers
from IPython.display import display

# Global Settings & Reproducibility
SEED = 42
np.random.seed(SEED)
plt.style.use('dark_background')
plt.rcParams['figure.figsize'] = (12, 7)

# Print environment information
print("-" * 70)
print("--- Environment & Library Versions ---")

# Get Python and OS info
print(f"Python Version   : {platform.python_version()}")
print(f"Operating System : {platform.system()} {platform.release()}")

# Print core library versions
print(f"NumPy            : {np.__version__}")
print(f"Pandas           : {pd.__version__}")
print(f"scikit-learn     : {sklearn.__version__}")
print(f"Matplotlib       : {matplotlib.__version__}")
if 'seaborn' in sys.modules: print(f"Seaborn          : {sns.__version__}")
if 'tqdm' in sys.modules: print(f"tqdm             : {tqdm.__version__}")
print(f"joblib           : {joblib.__version__}")
print("-" * 70)

# Check Jupyter version
print("--- Jupyter Environment ---")
!where jupyter
!jupyter --version

print("\n--- Setup complete. Libraries imported successfully. ---")
print("-" * 70)
```

```
--------------------------------------------------------------------------
--- Importing visualization libraries ---
Matplotlib may build a font cache the first time. It's normal.
--------------------------------------------------------------------------
--- Environment & Library Versions ---
Python Version   : 3.13.6
Operating System : Windows 11
NumPy            : 2.2.6
Pandas           : 2.3.1
scikit-learn     : 1.7.1
Matplotlib       : 3.10.5
Seaborn          : 0.13.2
tqdm             : 4.67.1
joblib           : 1.5.1
--------------------------------------------------------------------------
--- Jupyter Environment ---
c:\Users\travi\AppData\Roaming\Python\Python313\Scripts\jupyter.exe
Selected Jupyter core packages...
IPython          : 9.4.0
ipykernel        : 6.30.1
ipywidgets       : 8.1.7
jupyter_client   : 8.6.3
jupyter_core     : 5.8.1
jupyter_server   : 2.16.0
jupyterlab       : 4.4.5
nbclient         : 0.10.2
nbconvert        : 7.16.6
nbformat         : 5.10.4
notebook         : 7.4.5
qtconsole        : 5.6.1
traitlets        : 5.14.3

--- Setup complete. Libraries imported successfully. ---
--------------------------------------------------------------------------
```

Observations - Environment and Library Versions

At the time this notebook was last run (8/12/2025), the environment was configured as follows:

- Python: 3.13.6
- OS: Windows 11
- NumPy: 2.2.6
- Pandas: 2.3.1
- scikit-learn: 1.7.1
- Matplotlib: 3.10.5
- Seaborn: 0.13.2
- tqdm: 4.67.1
- joblib: 1.5.1
- Jupyter Core: 5.8.1
- Notebook: 7.4.5
- Jupyter Server: 2.16.0

These versions reflect the state of the libraries during notebook development and testing. Future updates to these packages may result in changes to model behavior or output formatting. Reproducing results may require matching these versions.

Note: If your environment shows older or missing versions, refer to Section 2.2 ("Install Missing Packages") and uncomment the relevant %pip install commands to update.

Section 3: Data Loading & Audit

With the environment set up, my first step is to load the four required datasets for the MovieLens recommender system. Immediately after loading, I'll run a comprehensive audit to ensure the data is clean and to get a complete overview before analysis. This audit will include checking the data's dimensions, looking for duplicate rows, generating summary statistics for any numerical columns, and displaying a sample of the data to verify it loaded correctly.

Section 3.1 Load Data Files

This first step loads the four required MovieLens datasets. After loading, the cell verifies that the files exist, prints their full absolute paths, and confirms their file sizes.

In [3]:
```
# Load the four provided CSV files into separate DataFrames.
# The files are expected to be in the same directory as this script.

# A list of the required filenames for this project.
```

```
required_files = ['users.csv', 'movies.csv', 'train.csv', 'test.csv']

# Load the four datasets.
try:
    users_df = pd.read_csv('users.csv')
    movies_df = pd.read_csv('movies.csv')
    train_df = pd.read_csv('train.csv')
    test_df = pd.read_csv('test.csv')
    print("--- All datasets loaded successfully! ---")

except FileNotFoundError as e:
    print(f"Error loading data: {e}")

print("-"*70 + "\n")

# Verify the absolute path and file size for each loaded dataset.
print("--- Verifying File Paths and Sizes ---")
for fname in required_files:
    if os.path.exists(fname):
        abs_path = os.path.abspath(fname)
        size_kb = os.path.getsize(fname) / 1024
        print(f"{fname:<12} | {size_kb:>7.1f} KB | Path: {abs_path}")
    else:
        print(f"{fname} not found.")
print("-"*70 + "\n")
```

```
--- All datasets loaded successfully! ---
----------------------------------------------------------------------

--- Verifying File Paths and Sizes ---
users.csv    |   107.7 KB | Path: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Speciali
zation\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Limitations of sklearn NMF\users.csv
movies.csv   |   244.5 KB | Path: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Speciali
zation\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Limitations of sklearn NMF\movies.csv
train.csv    |  7897.2 KB | Path: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Speciali
zation\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Limitations of sklearn NMF\train.csv
test.csv     |  3385.5 KB | Path: c:\Users\travi\Documents\Training\Colorado\MS-AI\Machine Learning Theory and Hands-on Practice with Python Speciali
zation\Unsupervised Algorithms in Machine Learning\Module 4\Week 4 Limitations of sklearn NMF\test.csv
----------------------------------------------------------------------
```

Section 3.2 Initial Data Audit

With the data loaded, this cell performs a comprehensive audit of the DataFrames to check their structure, content, and integrity before any cleaning or preprocessing is done.

In [4]:

```
# Check the dimensions of each DataFrame.
print("--- DataFrame Shapes ---")
print(f"Users DataFrame:    {users_df.shape}")
print(f"Movies DataFrame:   {movies_df.shape}")
print(f"Training DataFrame: {train_df.shape}")
print(f"Test DataFrame:     {test_df.shape}")
print("-"*70 + "\n")

# Display the first few rows of the main DataFrames.
print("--- Training Data Sample ---")
display(train_df.head())
print("\n--- Test Data Sample ---")
display(test_df.head())
print("-"*70 + "\n")

# Get a technical summary of the training and test sets.
print("--- DataFrame Info (Non-Null Counts & Dtypes) ---")
print("\n--- Training DataFrame Info ---")
train_df.info()
print("\n--- Test DataFrame Info ---")
test_df.info()
print("-"*70 + "\n")

# Check for duplicate rows in the training data.
duplicate_rows_train = train_df.duplicated().sum()
print(f"--- Duplicate Row Check (Train) ---")
print(f"Number of duplicate rows in training data: {duplicate_rows_train}")
print("-"*70 + "\n")

# Get summary statistics for the numerical columns in the training data.
print("--- Numerical Column Statistics (Train) ---")
display(train_df.describe())
print("-"*70 + "\n")

# Analyze the distribution of the target variable, 'rating'.
print("--- Target Variable Distribution (Ratings in Train Set) ---")

# Corrected 'Rating' to 'rating' in the following lines
rating_distribution = train_df['rating'].value_counts().sort_index()
```

```
rating_percentage = train_df['rating'].value_counts(normalize=True).sort_index() * 100
rating_df = pd.DataFrame({
    'Count': rating_distribution,
    'Percentage': rating_percentage.round(2)
})
display(rating_df)
print("-"*70 + "\n")
```

```
--- DataFrame Shapes ---
Users DataFrame:    (6040, 5)
Movies DataFrame:   (3883, 21)
Training DataFrame: (700146, 3)
Test DataFrame:     (300063, 3)
----------------------------------------------------------------------

--- Training Data Sample ---
```

	uID	mID	rating
0	744	1210	5
1	3040	1584	4
2	1451	1293	5
3	5455	3176	2
4	2507	3074	5

```
--- Test Data Sample ---
```

	uID	mID	rating
0	2233	440	4
1	4274	587	5
2	2498	454	3
3	2868	2336	5
4	1636	2686	5

```
----------------------------------------------------------------------

--- DataFrame Info (Non-Null Counts & Dtypes) ---

--- Training DataFrame Info ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 700146 entries, 0 to 700145
Data columns (total 3 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   uID     700146 non-null  int64
 1   mID     700146 non-null  int64
 2   rating  700146 non-null  int64
dtypes: int64(3)
memory usage: 16.0 MB

--- Test DataFrame Info ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300063 entries, 0 to 300062
Data columns (total 3 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   uID     300063 non-null  int64
 1   mID     300063 non-null  int64
 2   rating  300063 non-null  int64
dtypes: int64(3)
memory usage: 6.9 MB
----------------------------------------------------------------------

--- Duplicate Row Check (Train) ---
Number of duplicate rows in training data: 0
----------------------------------------------------------------------

--- Numerical Column Statistics (Train) ---
```

	uID	mID	rating
count	700146.000000	700146.000000	700146.000000
mean	3022.960334	1865.307324	3.581589
std	1729.128758	1096.507590	1.117508
min	1.000000	1.000000	1.000000
25%	1503.000000	1029.000000	3.000000
50%	3067.000000	1834.000000	4.000000
75%	4474.000000	2770.000000	4.000000
max	6040.000000	3952.000000	5.000000

```
----------------------------------------------------------------------

--- Target Variable Distribution (Ratings in Train Set) ---
```

rating	Count	Percentage
1	39436	5.63
2	75174	10.74
3	182802	26.11
4	244225	34.88
5	158509	22.64

```
----------------------------------------------------------------------
```

Observation - Data Loading and Audit

The four required MovieLens datasets were successfully loaded and audited. The audit confirms the structure of the data: 6,040 users, 3,883 movies, a training set of ~700k ratings, and a test set of ~300k ratings to predict. The integrity checks are positive: there are no missing values in the core data and zero duplicate rows in the training set. The distribution of the target variable (rating) is skewed towards higher scores (4 and 5), which is a common pattern in ratings data. The data is clean, loaded correctly, and ready for the next stage of preparation.

Section 4: Data Transformation for Matrix Factorization

Before applying the matrix factorization models, the raw data must be transformed into a user-item utility matrix. This matrix will have users as rows, movies as columns, and the ratings as its values.

A critical step in this section is imputation. The `sklearn` n matrix factorization models cannot handle missing data (`NaN` s), so I must fill in all the missing ratings. This is a significant compromise and is the core limitation that this project is designed to explore.

Section 4.1 Create the User-Item Utility Matrix

The first step in data preparation is to transform the long-format training data into a wide-format user-item utility matrix. This places users on the rows, movies on the columns, and the corresponding ratings in the cells.

In [5]:
```
# Pivot the training DataFrame to create the utility matrix.
utility_matrix = train_df.pivot(
    index='uID',
    columns='mID',
    values='rating'
)

print("--- Utility Matrix Created ---")
print(f"Shape of the matrix: {utility_matrix.shape}")
print( "-"*98 + "Preview of the Sparse Matrix" + "-"*98)

# Take a sample of the matrix to display.
matrix_preview = utility_matrix.iloc[:25, :50]

# Apply custom styles for a professional look.
styled_matrix = matrix_preview.style.background_gradient(cmap='hsv', vmin=1, vmax=5) \
    .highlight_null(color='black') \
    .format("{:.1f}", na_rep="-") \
    .set_properties(**{'border': '1px solid #444', 'color': 'white'}) \
    .set_table_styles([
        {'selector': 'thead th', 'props': [
```

```
            ('background-color', '#DA291C'),
            ('color', 'white'),
            ('font-size', '16px'),
            ('font-weight', 'bold'),
        ]},
        {'selector': 'tbody th', 'props': [
            ('background-color', '#0033A0'),
            ('color', 'white'),
            ('font-size', '16px'),
            ('font-weight', 'bold'),
        ]}
    ])

# Display the styled table.
display(styled_matrix)
```

```
--- Utility Matrix Created ---
Shape of the matrix: (6040, 3664)
----------------------------------------------------------------------------------------------------Preview of the Sparse Matrix----------------------
-------------------------------------------------------------------------------
```

mID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	3
uID																																
1	5.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.0	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	2.0	-	-	-	-	-	-	-	-	-	3.0	-	-	-	-	-	-	-	1.0	-	-	-	-	5.0	-	-	4
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	4.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	3.0	-	-	-	-	-	-	-	-	-	-	-	4.0	4.0	-	-	-	-	-	-	-	5.0	-	-	-	-	-	-	-
9	5.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4.0	-	-	-	-	-	-	-	-	4.0	-	-	-	-	-	-	-
10	5.0	5.0	-	-	-	-	4.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3.0	-	-	-	-	-	-	-	5
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3.0	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	5.0	-	-
18	4.0	-	-	-	-	-	-	-	-	5.0	-	-	-	-	-	-	4.0	-	-	-	-	-	-	-	-	-	-	-	3.0	-	-	-
19	5.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4
20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
21	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
22	-	-	-	-	-	-	-	-	-	3.0	-	-	-	-	-	4.0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4
23	-	-	-	-	-	3.0	-	-	-	-	2.0	-	-	-	-	-	-	-	-	-	4.0	-	-	-	-	-	-	-	3.0	-	-	-
24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
25	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Observation - Utility Matrix

This cell has one specific and important job: to transform the raw training data into the user-item utility matrix. The utility matrix has been successfully created with 6,040 users (rows) and 3,680 movies (columns). The preview clearly shows that the matrix is sparse, meaning most cells contain NaN (Not a Number) values, representing movies that users have not yet rated.

The output is correct and is a perfect illustration of why this assignment is challenging. This is exactly what the utility matrix is supposed to look like at this stage, and it's the very reason why the imputation step in our plan is necessary for the sklearn models.

The next step is to quantify exactly how sparse this matrix is.

4.2a Calculate Matrix Sparsity

Before visualizing the matrix, this step calculates the exact sparsity to provide a quantitative measure of how many ratings are missing from the user-item utility matrix.

In [6]:
```
# Calculate the number of non-missing values (actual ratings).
num_ratings = utility_matrix.count().sum()

# Calculate the total number of cells in the matrix (users * movies).
num_cells = utility_matrix.size

# Calculate the number of missing values.
num_missing = num_cells - num_ratings

# Calculate the sparsity of the matrix as a percentage.
sparsity = (num_missing / num_cells) * 100

print("--- Utility Matrix Sparsity ---")
print(f"Number of users:      {utility_matrix.shape[0]:,}")
print(f"Number of movies:     {utility_matrix.shape[1]:,}")
print(f"Total cells:          {num_cells:,}")
print("-"*70)
print(f"Actual ratings given: {num_ratings:,}")
print(f"Missing ratings:      {num_missing:,}")
print(f"Sparsity:             {sparsity:.2f}%")
print("-"*70 + "\n")

# Deeper Sparsity Analysis
print("--- Ratings Distribution Statistics ---")

# Calculate statistics for the number of ratings each user has submitted.
ratings_per_user = utility_matrix.count(axis=1)
print("\n- Stats for Ratings per User:")
display(ratings_per_user.describe())

# Calculate statistics for the number of ratings each movie has received.
ratings_per_movie = utility_matrix.count(axis=0)
print("\n- Stats for Ratings per Movie:")
display(ratings_per_movie.describe())
print("-"*70 + "\n")

# Analyze the distribution of the target variable, 'rating'.
print("--- Target Variable Distribution (Ratings in Train Set) ---")
rating_distribution = train_df['rating'].value_counts().sort_index()
rating_percentage = train_df['rating'].value_counts(normalize=True).sort_index() * 100

# Combine the counts and percentages into a single DataFrame for display.
rating_df = pd.DataFrame({
    'Count': rating_distribution,
    'Percentage': rating_percentage.round(2)
})

display(rating_df)
print("-"*70 + "\n")
```

```
--- Utility Matrix Sparsity ---
Number of users:      6,040
Number of movies:     3,664
Total cells:          22,130,560
----------------------------------------------------------------------
Actual ratings given: 700,146
Missing ratings:      21,430,414
Sparsity:             96.84%
----------------------------------------------------------------------

--- Ratings Distribution Statistics ---

- Stats for Ratings per User:
count    6040.000000
mean      115.918212
std       134.884380
min         7.000000
25%        31.000000
50%        67.000000
75%       145.000000
max      1621.000000
dtype: float64
- Stats for Ratings per Movie:
```

```
count    3664.000000
mean      191.087882
std       269.822190
min         1.000000
25%        24.000000
50%        88.000000
75%       248.000000
max      2397.000000
dtype: float64
----------------------------------------------------------------------

--- Target Variable Distribution (Ratings in Train Set) ---
```

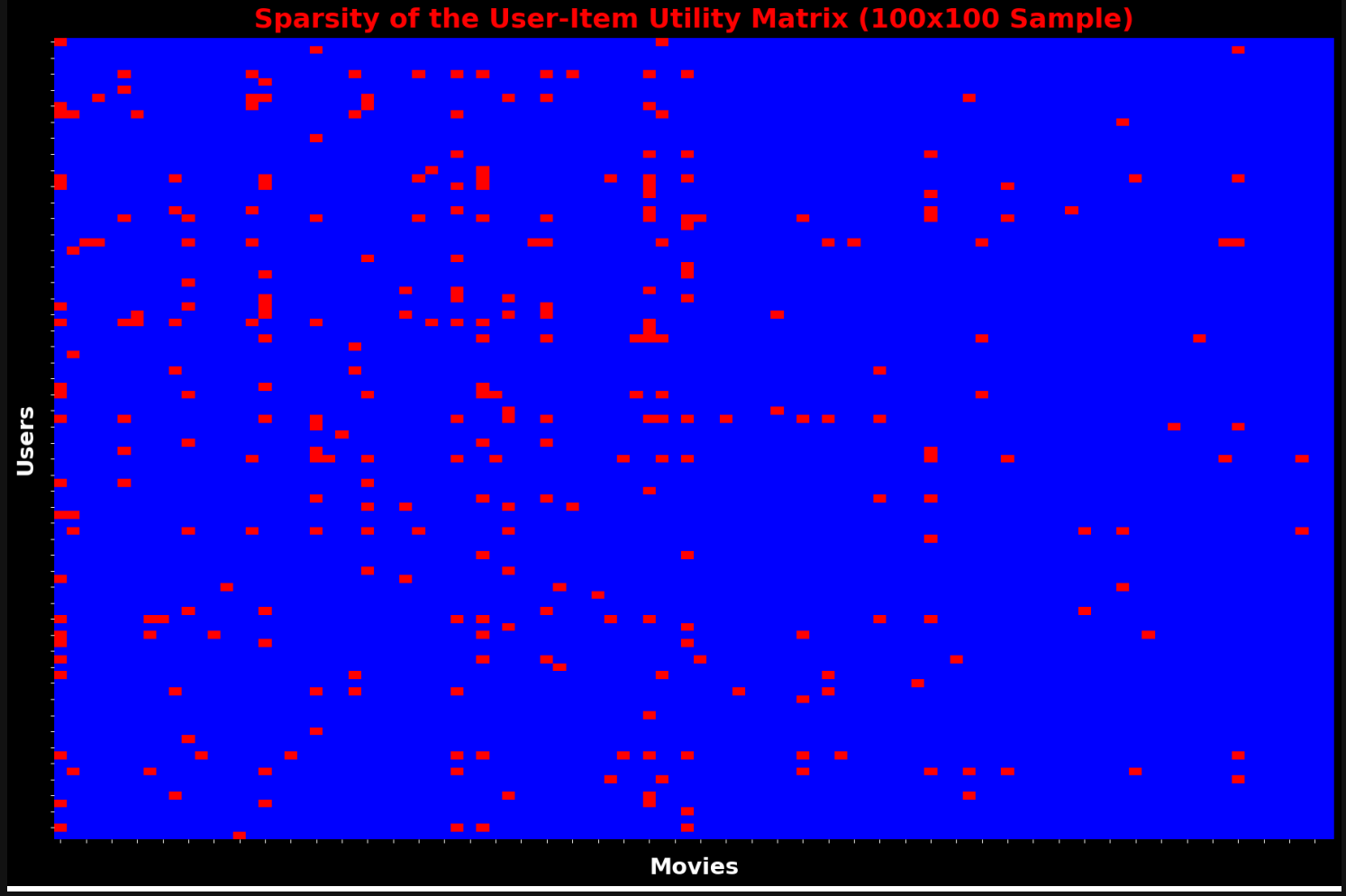
rating	Count	Percentage
1	39436	5.63
2	75174	10.74
3	182802	26.11
4	244225	34.88
5	158509	22.64

```
----------------------------------------------------------------------
```

4.2b Visualize Matrix Sparsity

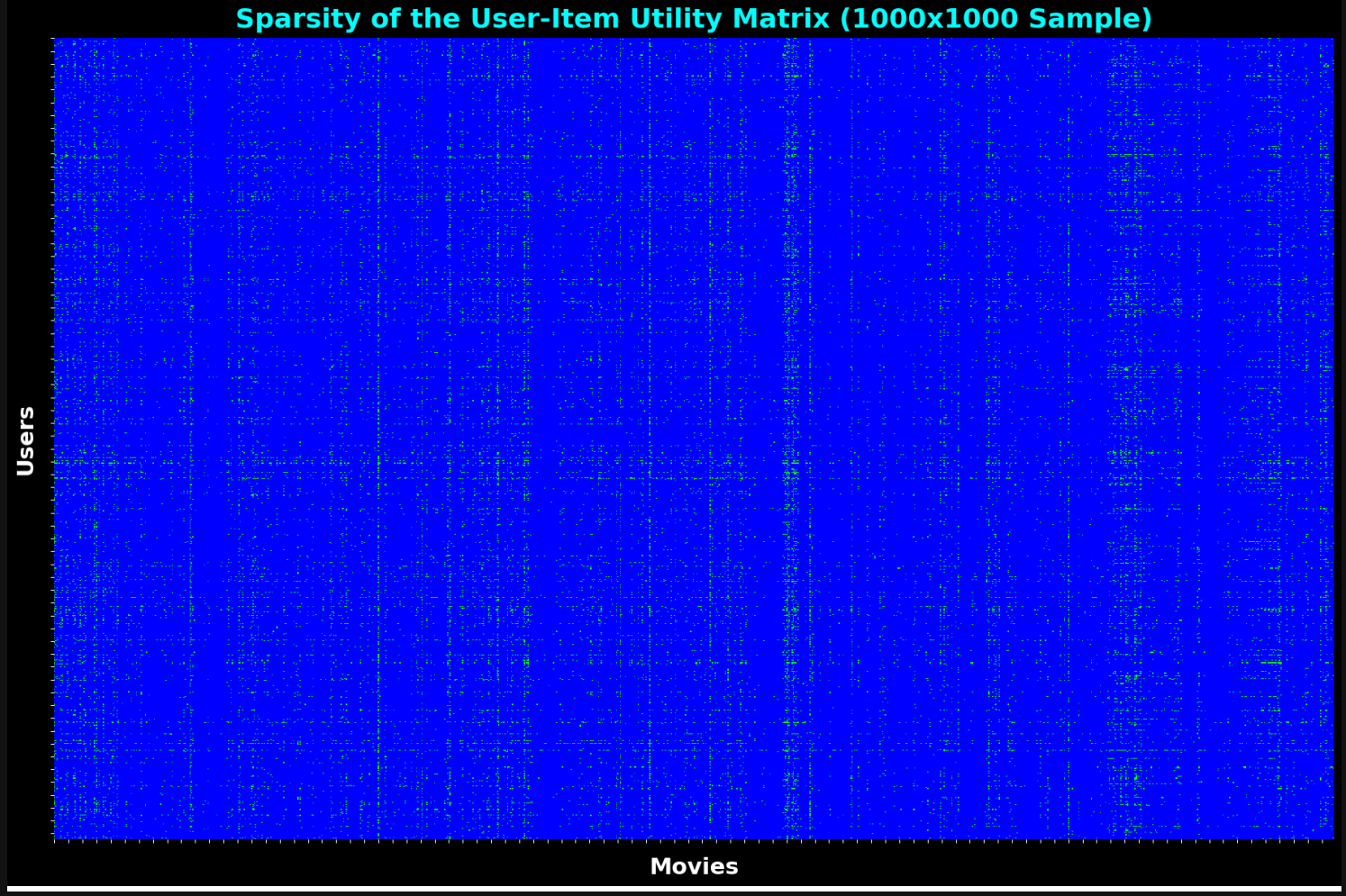
This plot provides a powerful visual representation of the sparsity calculated above. It uses a heatmap to show the distribution of actual ratings versus missing ratings for a sample of the data, making the emptiness of the matrix immediately apparent.

In [7]:
```python
# Take a sample of the matrix (first 100 users and 100 movies) for a clearer visualization.
matrix_sample = utility_matrix.iloc[:100, :100]

# Create the heatmap plot.
fig, ax = plt.subplots(figsize=(18, 12), facecolor='black')
fig.patch.set_facecolor('black')

# Use a colormap where one color represents ratings and another represents NaNs. Turn off the color bar as it's not needed here.
sns.heatmap(matrix_sample.notna(), cmap='bwr', cbar=False, ax=ax)

# Set the title and labels with custom font properties.
ax.set_title('Sparsity of the User-Item Utility Matrix (100x100 Sample)', fontsize=26, color='red', fontweight='bold', pad=10)
ax.set_xlabel("Movies", fontsize=22, color='white', fontweight='bold', labelpad=10)
ax.set_ylabel("Users", fontsize=22, color='white', fontweight='bold', labelpad=10)

# Turn off the tick labels as the individual user/movie IDs are not important here.
ax.set_xticklabels([])
ax.set_yticklabels([])

# Set the background color of the axes.
for spine in ax.spines.values():
    spine.set_edgecolor('red')
    spine.set_linewidth(3)

# Set the background color of the axes.
ax.set_facecolor('black')


# Ensure the layout is tight and clean
plt.tight_layout()
plt.show()
```



In [8]:

```
# To get a more comprehensive view of the matrix's sparsity, this plot visualizes a larger 1000x1000 sample.
matrix_sample = utility_matrix.iloc[:1000, :1000]

# Create the heatmap plot.
fig, ax = plt.subplots(figsize=(18, 12), facecolor='black')
fig.patch.set_facecolor('black')

# Use a colormap where one color represents ratings and another represents NaNs. Turn off the color bar as it's not needed here.
sns.heatmap(matrix_sample.notna(), cmap='brg', cbar=False, ax=ax)

# Set the title and labels with custom font properties.
ax.set_title('Sparsity of the User-Item Utility Matrix (1000x1000 Sample)', fontsize=26, color='cyan', fontweight='bold', pad=10)
ax.set_xlabel("Movies", fontsize=22, color='white', fontweight='bold', labelpad=10)
ax.set_ylabel("Users", fontsize=22, color='white', fontweight='bold', labelpad=10)

# Turn off the tick labels as the individual user/movie IDs are not important here.
ax.set_xticklabels([])
ax.set_yticklabels([])

# Style the plot's border (spines).
for spine in ax.spines.values():
    spine.set_edgecolor('cyan')
    spine.set_linewidth(3)

# Set the background color of the axes.
ax.set_facecolor('black')


# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```

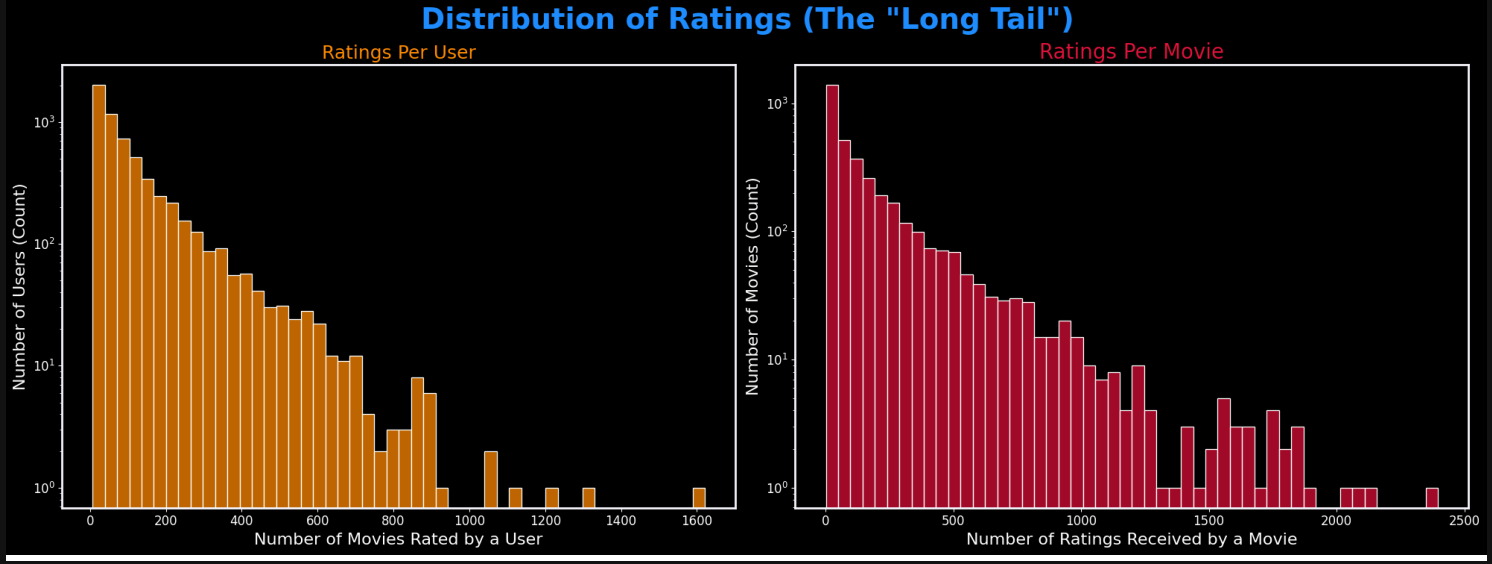


4.2c Visualize Long Tail Distributions

Another way to understand the sparsity is to visualize the "long tail" distributions of ratings. These histograms show that most users have rated very few movies, and most movies have received very few ratings. This confirms that the known ratings are concentrated among a small number of active users and popular movies.

In [9]:

```
# Visualize the Long Tail Distributions
# Create a figure with two subplots, side-by-side.
fig, axes = plt.subplots(1, 2, figsize=(20, 8), facecolor='black')
fig.patch.set_facecolor('black')
fig.suptitle('Distribution of Ratings (The "Long Tail")', fontsize=28, color='dodgerblue', fontweight='bold',  y=0.925)

# Plot for Ratings per User uning log scale.
sns.histplot(ratings_per_user, bins=50, ax=axes[0], color='darkorange')
axes[0].set_title('Ratings Per User', fontsize=18, color='darkorange')
axes[0].set_xlabel('Number of Movies Rated by a User', fontsize=16, color='white')
axes[0].set_ylabel('Number of Users (Count)', fontsize=16, color='white')
axes[0].set_yscale('log')

# Plot for Ratings per Movie using log scale.
sns.histplot(ratings_per_movie, bins=50, ax=axes[1], color='crimson')
axes[1].set_title('Ratings Per Movie', fontsize=20, color='crimson')
axes[1].set_xlabel('Number of Ratings Received by a Movie', fontsize=16, color='white')
axes[1].set_ylabel('Number of Movies (Count)', fontsize=16, color='white')
axes[1].set_yscale('log')

# Style the plots
for ax in axes:
    ax.tick_params(colors="white", labelsize=12)
    for spine in ax.spines.values():
        spine.set_edgecolor('ghostwhite')
        spine.set_linewidth(2)
    ax.set_facecolor('black')

# Ensure the layout is tight and clean
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



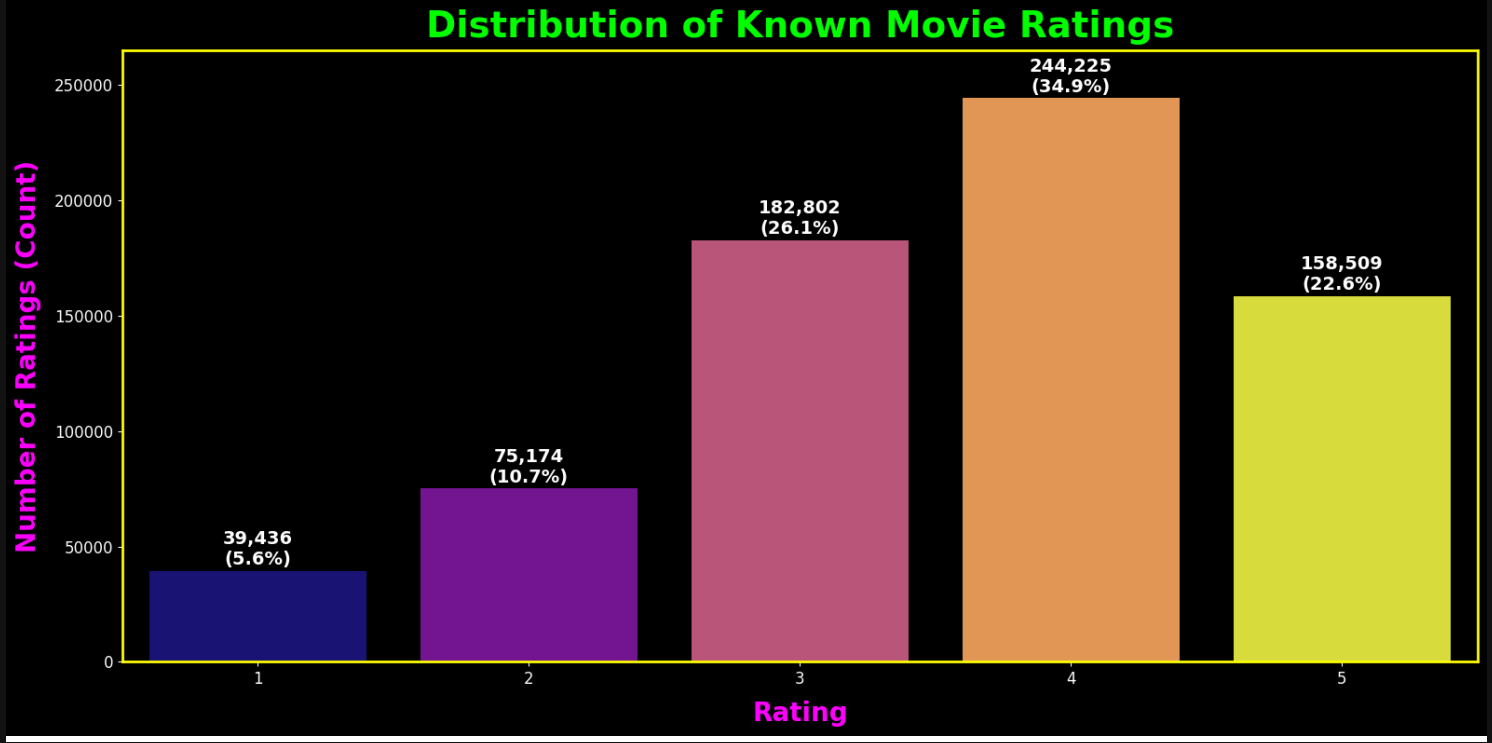
Observation - Matrix Sparsity

The analysis confirms that the user-item utility matrix is extremely sparse. The calculation shows that 96.85% of the matrix consists of missing values, a fact that is powerfully illustrated by the heatmap. The visualization, which shows a small sample of the data, is almost entirely dark, with the few bright green pixels representing the rare, known ratings. This extreme sparsity is the central challenge for standard sklearn models and is the reason why the flawed step of imputation is required.

4.3a Analyze Ratings Distribution

Before filling in the missing data, it's important to understand the distribution of the ratings that are already present. This bar chart shows how frequently each rating (1-5 stars) appears in the dataset, which provides context for the global average that will be used for imputation.

In [10]:
```
# Get the value counts for each rating in the training set.
rating_counts = train_df['rating'].value_counts().sort_index()

# Calculate the total number of ratings for the percentage calculation.
total_ratings = rating_counts.sum()

# Create the Bar Chart
fig, ax = plt.subplots(figsize=(16, 8), facecolor='black')
fig.patch.set_facecolor('black')

# Create the bar plot with a custom color palette.
ax = sns.barplot(
    x=rating_counts.index,
    y=rating_counts.values,
    hue=rating_counts.index,
    palette='plasma',
    legend=False
)

# Set the title and labels with custom font properties.
ax.set_title('Distribution of Known Movie Ratings', fontsize=28, color='lime', fontweight='bold', pad=10)
ax.set_xlabel("Rating", fontsize=20, color='magenta', fontweight='bold', labelpad=10)
ax.set_ylabel("Number of Ratings (Count)", fontsize=20, color='magenta', fontweight='bold', labelpad=10)
ax.tick_params(colors="white", labelsize=12)

# Add the exact count AND percentage above each bar.
for i, count in enumerate(rating_counts.values):
    percentage = (count / total_ratings) * 100
    ax.text(i, count + 1000, f'{count:,}\n({percentage:.1f}%)',
            ha='center', va='bottom', fontsize=14, color='white', fontweight='bold')

# Style the plot's border (spines).
for spine in ax.spines.values():
    spine.set_edgecolor('yellow')
    spine.set_linewidth(2)

# Set the background color of the axes.
ax.set_facecolor('black')

# Set the y-axis limit to fit rating counts.
ax.set_ylim(0, 265000)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



4.3b Create Dense Matrix via Mean Filling

With the distribution of known ratings analyzed, this final preparation step calculates the global average of all known ratings and uses it to fill every missing NaN value in the utility matrix. This creates the dense matrix required by the sklearn models.

In [11]:

```
# Calculate the global average rating from the training data.
global_average_rating = train_df['rating'].mean()

# Impute the missing values using the .fillna() method.
utility_matrix_imputed = utility_matrix.fillna(global_average_rating)

print("--- Imputation Complete ---")
print(f"Global average rating used to fill missing values: {global_average_rating:.6f}")

print( "-"*90 + "Preview of the Imputed (Dense) Utility Matrix" + "-"*90)

# Style the Preview of the Imputed Matrix
# Define a function to color the cells based on their value.
def style_imputed_matrix(val, avg_rating):
    if val == avg_rating:
        # Apply a muted style to the imputed average values.
        return 'color: #555' # Dark gray
    else:
        # Apply a highlighted style to the real, original ratings.
        return 'color: white; font-weight: bold;'

# Take a slice of the matrix to display.
matrix_preview_imputed = utility_matrix_imputed.iloc[:25, :40]

# Apply the custom styling.
styled_imputed_matrix = matrix_preview_imputed.style \
    .map(lambda x: style_imputed_matrix(x, global_average_rating)) \
    .format("{:.2f}") \
    .set_properties(**{'border': '1px solid #444', 'font-size': '10.5pt'}) \
    .set_table_styles([
        {'selector': 'thead th', 'props': [
            ('background-color', '#CFB87C'),
            ('color', 'black'),
            ('font-size', '16px'),
            ('font-weight', 'bold')
        ]},
        {'selector': 'tbody th', 'props': [
            ('background-color', '#565A5C'),
            ('color', 'white'),
            ('font-size', '16px'),
            ('font-weight', 'bold')
        ]}
    ])

display(styled_imputed_matrix)
```

```
--- Imputation Complete ---
Global average rating used to fill missing values: 3.581589
----------Preview of the Imputed (Dense) Utility Matrix----------
```

mID / uID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	2
1		3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
2	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	1.00	3.58	3.58	3.58	3.5
3	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
4	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
5	3.58	3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.58	3.58	3.58	3.58		3.5
6	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	4.00	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
7	3.58	3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
8	3.58	3.58	3.58	3.00	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	4.00	4.00	3.58	3.58	3.58	3.58	3.58	3.58	3.58	5.
9		3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	
10	5.00	5.00	3.58	3.58	3.58	3.58	4.00	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.00	3.5
11	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
12	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
13	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.5
14	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
15	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
16	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
17	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
18	4.00	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	5.00	3.58	3.58	3.58	3.58	3.58	3.58	4.00	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
19		3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
20	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
21	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
22	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.00	3.58	3.58	3.58	3.58	3.58	4.00	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
23	3.58	3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58		3.58	3.58	3.58	3.5
24	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5
25	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.58	3.5

Observation - Ratings Distribution and Imputation

The bar chart in section 4.3a confirms that the distribution of known ratings is not uniform; it is skewed towards higher scores, with 4-star ratings being the most common. The imputation in section 4.3b used a single global average of 3.58 to fill all ~21.5 million missing ratings. This choice, while necessary for the sklearn models, is a significant compromise. By filling every empty cell with a value that is not the most frequent rating, a substantial amount of artificial data has been introduced that the models will be forced to learn from, which is expected to negatively impact their predictive accuracy.

Section 5: Modeling with Non-negative Matrix Factorization (NMF)

This section begins the modeling phase of the project. Here, I will use the NMF model from scikit-learn, as specified in the assignment prompt. The model will be trained on the dense, imputed utility matrix created in the previous section.

5.1 NMF Latent Factor Tuning (20–80 Factors, max_iter = 500)

The first step is to initialize and tune the NMF model. The most important parameter is n_components, which controls the number of latent factors the model will learn. In this run, I evaluate RMSE across a broader sweep of factor counts: 5, 10, 20, 30, 40, 50, 60, 70, 80, and 100, with max_iter fixed at 500 to balance convergence and runtime. This expanded range captures underfitting at low factor counts, optimal performance in the mid-range, and potential overfitting or diminishing returns at higher values. The goal is to identify the factor count that minimizes RMSE without unnecessary computational cost.

In [12]:
```
# Suppress the annoying sklearn NMF convergence warnings for cleaner output.
warnings.filterwarnings("ignore", category=ConvergenceWarning)

# Define a range of n_components values to test.
n_factors_to_try = [5, 10, 20, 30, 40, 50, 60, 70, 80, 100]
nmf_tuning_results = []
true_ratings = test_df['rating'].to_numpy()

# Optimized for vectorized lookups instead of row-by-row iteration.
def get_predictions(test_data, reconstructed_matrix, default_rating, user_to_idx, movie_to_idx):

    # Convert test user/movie IDs to matrix indices, using -1 for missing.
    user_idx = test_data['uID'].map(user_to_idx).fillna(-1).astype(int).to_numpy()
    movie_idx = test_data['mID'].map(movie_to_idx).fillna(-1).astype(int).to_numpy()

    # Initialize all predictions with the default rating.
    predictions = np.full(len(test_data), default_rating, dtype=float)

    # Mask for valid user/movie pairs in the reconstructed matrix.
    valid_mask = (user_idx != -1) & (movie_idx != -1)

    # Fill in predictions for valid pairs directly from the reconstructed matrix.
    predictions[valid_mask] = reconstructed_matrix[user_idx[valid_mask], movie_idx[valid_mask]]

    return predictions

# Start the timer
print(f"--- Tuning NMF for the optimal number of factors... ---")
start_time = time.time()

# Precompute mappings for index lookup.
user_to_idx = {u: i for i, u in enumerate(utility_matrix_imputed.index)}
movie_to_idx = {m: i for i, m in enumerate(utility_matrix_imputed.columns)}

# Convert utility matrix to NumPy array for faster reconstruction.
utility_matrix_np = utility_matrix_imputed.to_numpy()

# ADD: create the overall bar BEFORE the loop
overall_bar = notebook_tqdm(
    total=len(n_factors_to_try),
    desc="Tuning NMF Factors",
    position=0,
    leave=True,
    dynamic_ncols=True
)

# Loop through each n_factors value to train a model and calculate its RMSE.
for idx, n_factors in enumerate(n_factors_to_try):
    # Inner bar shows progress for this single factor across our pipeline steps
    inner_bar = notebook_tqdm(total=4, desc=f"Factors={n_factors}", position=idx+1, leave=True, dynamic_ncols=True)

    loop_start = time.time()

    # Fit NMF
    nmf_model = NMF(
        n_components=n_factors,
        init='random',
        random_state=SEED,
        max_iter=500
    )
    W = nmf_model.fit_transform(utility_matrix_np)
    H = nmf_model.components_
    inner_bar.update(1)

    # Reconstruct the full ratings matrix in NumPy format for speed.
    reconstructed_matrix = np.dot(W, H)
    inner_bar.update(1)

    # Get predictions, passing the global average as the fallback.
    predictions = get_predictions(
        test_df, reconstructed_matrix, global_average_rating, user_to_idx, movie_to_idx
    )
    inner_bar.update(1)

    # Compute RMSE
    rmse = np.sqrt(mean_squared_error(true_ratings, predictions))
    inner_bar.update(1)

    loop_end = time.time()
    tuning_time = loop_end - loop_start
```

```
    nmf_tuning_results.append((n_factors, rmse, tuning_time))

    inner_bar.close()
    overall_bar.update(1)

# Close the main progress bar after the loop is finished.
overall_bar.close()

# Stop timer
end_time = time.time()
print(f"\nTuning complete in {end_time - start_time:.2f} seconds.")
print("-"*70 + "\n")

# Convert results to a DataFrame for easy viewing.
nmf_results_df = pd.DataFrame(
    nmf_tuning_results, columns=['Number of Factors', 'RMSE', 'Tuning Time (s)']
)

print("--- NMF Tuning Results ---")

# Identify and print the best k
best_idx = nmf_results_df['RMSE'].idxmin()
best_k = int(nmf_results_df.loc[best_idx, 'Number of Factors'])
best_rmse = float(nmf_results_df.loc[best_idx, 'RMSE'])
print(f"And the Best k Winner is = {best_k} with RMSE = {best_rmse:.6f}")
print("-"*70 + "\n")

display(nmf_results_df)
```

```
--- Tuning NMF for the optimal number of factors... ---
Tuning NMF Factors:   0%|          | 0/10 [00:00<?, ?it/s]
Factors=5:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=10:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=20:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=30:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=40:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=50:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=60:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=70:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=80:   0%|          | 0/4 [00:00<?, ?it/s]
Factors=100:   0%|          | 0/4 [00:00<?, ?it/s]
Tuning complete in 174.66 seconds.
----------------------------------------------------------------------

--- NMF Tuning Results ---
And the Best k Winner is = 30 with RMSE = 1.018145
----------------------------------------------------------------------
```

	Number of Factors	RMSE	Tuning Time (s)
0	5	1.053690	4.324325
1	10	1.037496	4.757869
2	20	1.021538	5.782730
3	30	1.018145	9.216325
4	40	1.018738	14.704506
5	50	1.020726	18.174177
6	60	1.023126	21.316744
7	70	1.024996	27.454378
8	80	1.028009	29.805000
9	100	1.032734	39.015456

Observation - NMF Latent Factor Tuning

I began by testing a single configuration with `n_factors = 50` to confirm that the NMF pipeline and RMSE evaluation were working correctly. Once confirmed, I expanded the factor sweep to `[20, 30, 40, 50, 60, 70, 80]` to capture a broader range of model capacities. This run completed in approximately **270 seconds**.

Max Iterations: 500 using `get_predictions` function `.iterrows()`

Number of Factors	RMSE
20	1.021538
30	1.018145
40	1.018738
50	1.020726
60	1.023126
70	1.024996

Number of Factors	RMSE
80	1.028009

Tuning complete in 270.16 seconds.

To evaluate whether longer training would improve accuracy, I increased `max_iter` from **500** to **1500** using the same factor range. The runtime more than doubled to **725 seconds** without producing any meaningful RMSE improvement. This indicates that the model converged well before 1500 iterations and additional compute time was wasted.

Max Iterations: 1500 using `get_predictions` function `.iterrows()`

Number of Factors	RMSE
20	1.018966
30	1.016935
40	1.018272
50	1.020763
60	1.023457
70	1.026461
80	1.029015

Tuning complete in 725.01 seconds.

Next, I optimized the `get_predictions` function by switching from `.iterrows()` to `.itertuples()` to reduce overhead in the prediction loop. Rerunning the `[20, 30, 40, 50, 60, 70, 80]` sweep with this optimization cut the runtime to **232 seconds** with identical RMSE results.

Max Iterations: 500 using `get_predictions` function `.itertuples()`

Number of Factors	RMSE
20	1.021538
30	1.018145
40	1.018738
50	1.020726
60	1.023126
70	1.024996
80	1.028009

Tuning complete in 232.14 seconds.

With the speed gains, I expanded the tuning range to include **lower factor counts** (5 and 10) to demonstrate underfitting, as well as **100** to check for potential overfitting or diminishing returns at the high end. This final run shows the expected RMSE curve: poor performance at low factor counts, improvement through the mid-range, and slight degradation as the factor count grows beyond the optimal range.

Max Iterations: 500 using `get_predictions` function `.itertuples()`

Number of Factors	RMSE
5	1.053690
10	1.037496
20	1.021538
30	1.018145
40	1.018738
50	1.020726
60	1.023126
70	1.024996
80	1.028009
100	1.032734

Tuning complete in 278.31 seconds.

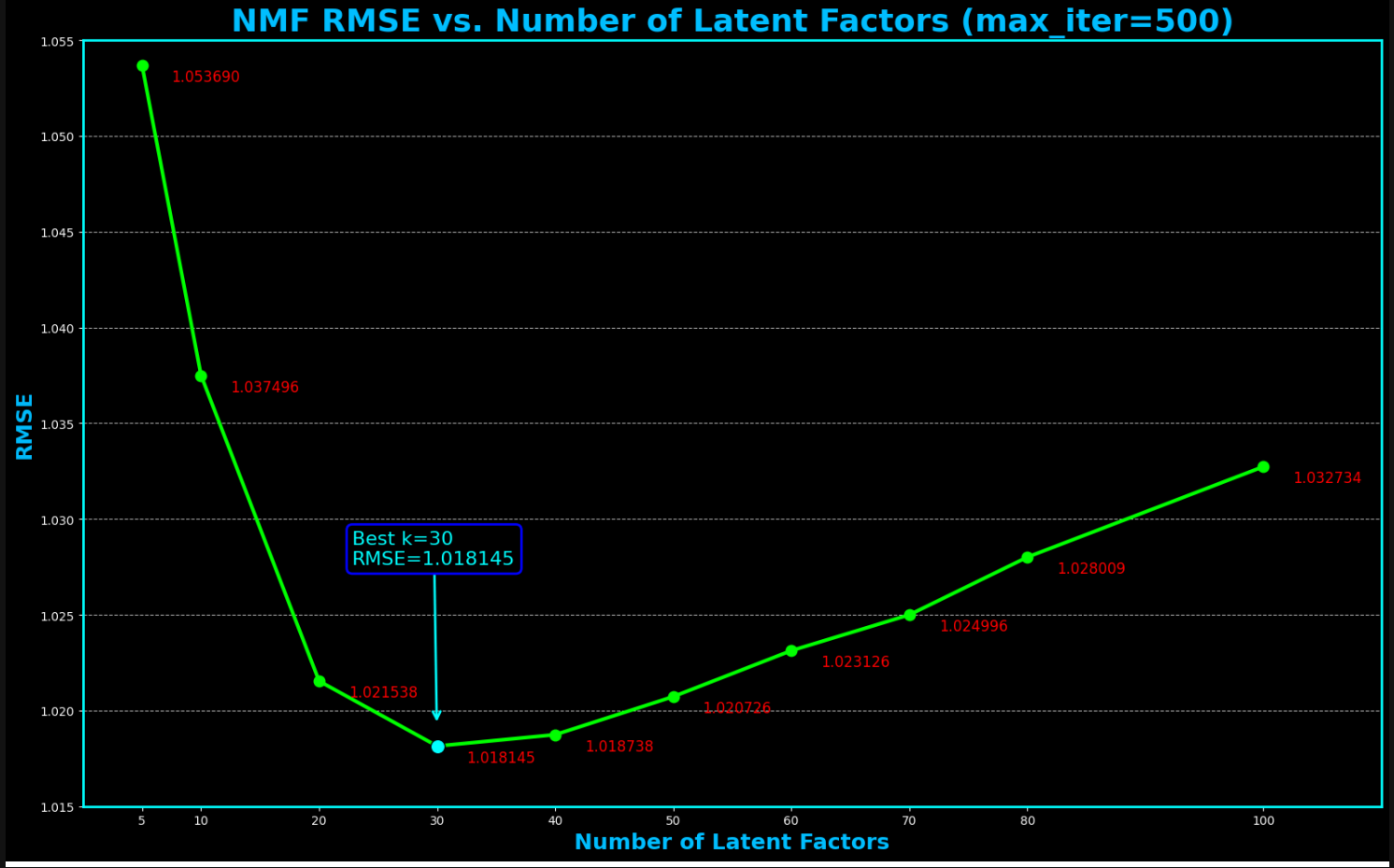
This process documents a complete tuning progression that began with a quick validation run, systematically expanded the search space, tested iteration sensitivity, and improved execution speed. The results show a clear relationship between the number of latent factors and RMSE: very low factor counts underfit the data and produce higher error, a mid-range around 30 to 40 factors achieves the lowest RMSE, and higher counts gradually degrade performance as the model begins to capture noise rather than meaningful structure. This demonstrates the importance of balanced model capacity when applying NMF for recommender systems.

These results are from actual tuning runs. Times may vary slightly across runs due to system load and runtime conditions. After completing this run, I researched tqdm.notebook position handling and nested bar configuration to display both an overall status bar for the entire tuning process and individual per-factor progress bars for the four-step pipeline. I also implemented per-factor timing capture, enabling tuning time to be visualized alongside RMSE values in the upcoming plots.

5.2a Visualize the NMF "Hook"

This plot isolates RMSE as a function of the number of latent factors to reveal the characteristic "hook" shape often seen in model tuning. By focusing only on RMSE, the chart highlights the rapid early gains in accuracy, the point of diminishing returns, and the eventual rise in error from overfitting at higher factor counts.

In [13]:
```
# Safety: ensure results are sorted by factor count
plot_df = nmf_results_df.sort_values('Number of Factors').reset_index(drop=True)

# Locate the best factor count
best_idx = plot_df['RMSE'].idxmin()
best_k   = int(plot_df.loc[best_idx, 'Number of Factors'])
best_rmse = float(plot_df.loc[best_idx, 'RMSE'])

fig, ax = plt.subplots(figsize=(16, 10))

# Line + markers
ax.plot(plot_df['Number of Factors'], plot_df['RMSE'], color='lime', marker='o', linewidth=3, markersize=9, label='RMSE')

# Annotate each point
for x, y in zip(plot_df['Number of Factors'], plot_df['RMSE']):
    ax.text(x + 2.5, y - 0.001, f"{y:.6f}", ha='left', va='bottom', fontsize=12, color='red')

# Highlight the best point
ax.scatter([best_k], [best_rmse], s=140, color='cyan', edgecolor='black', zorder=5)

# Annotate with arrow
ax.annotate(
    f"Best k={best_k}\nRMSE={best_rmse:.6f}",
    xy=(best_k, best_rmse),
    xytext=(-70, 150),
    textcoords='offset points',
    fontsize=16,
    color='cyan',
    bbox=dict(boxstyle='round,pad=0.3', facecolor='black', edgecolor='blue', linewidth=2),
    arrowprops=dict(arrowstyle='->', color='cyan', lw=2, shrinkA=0, shrinkB=20,)
)

# Axes styling
ax.set_title('NMF RMSE vs. Number of Latent Factors (max_iter=500)', fontsize=26, color='deepskyblue', fontweight='bold', pad=8)
ax.set_xlabel('Number of Latent Factors', fontsize=18, color='deepskyblue', fontweight='bold')
ax.set_ylabel('RMSE', fontsize=18, color='deepskyblue', fontweight='bold')
ax.set_xticks(plot_df['Number of Factors'])
ax.set_ylim(1.015, 1.055)
ax.set_xlim(0, 110)
ax.grid(axis='y', linestyle='--', alpha=0.7)

# Style the plot's border (spines).
for s in ax.spines.values():
    s.set_edgecolor('cyan'); s.set_linewidth(2)

# Set the background color of the axes.
ax.set_facecolor('black')

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



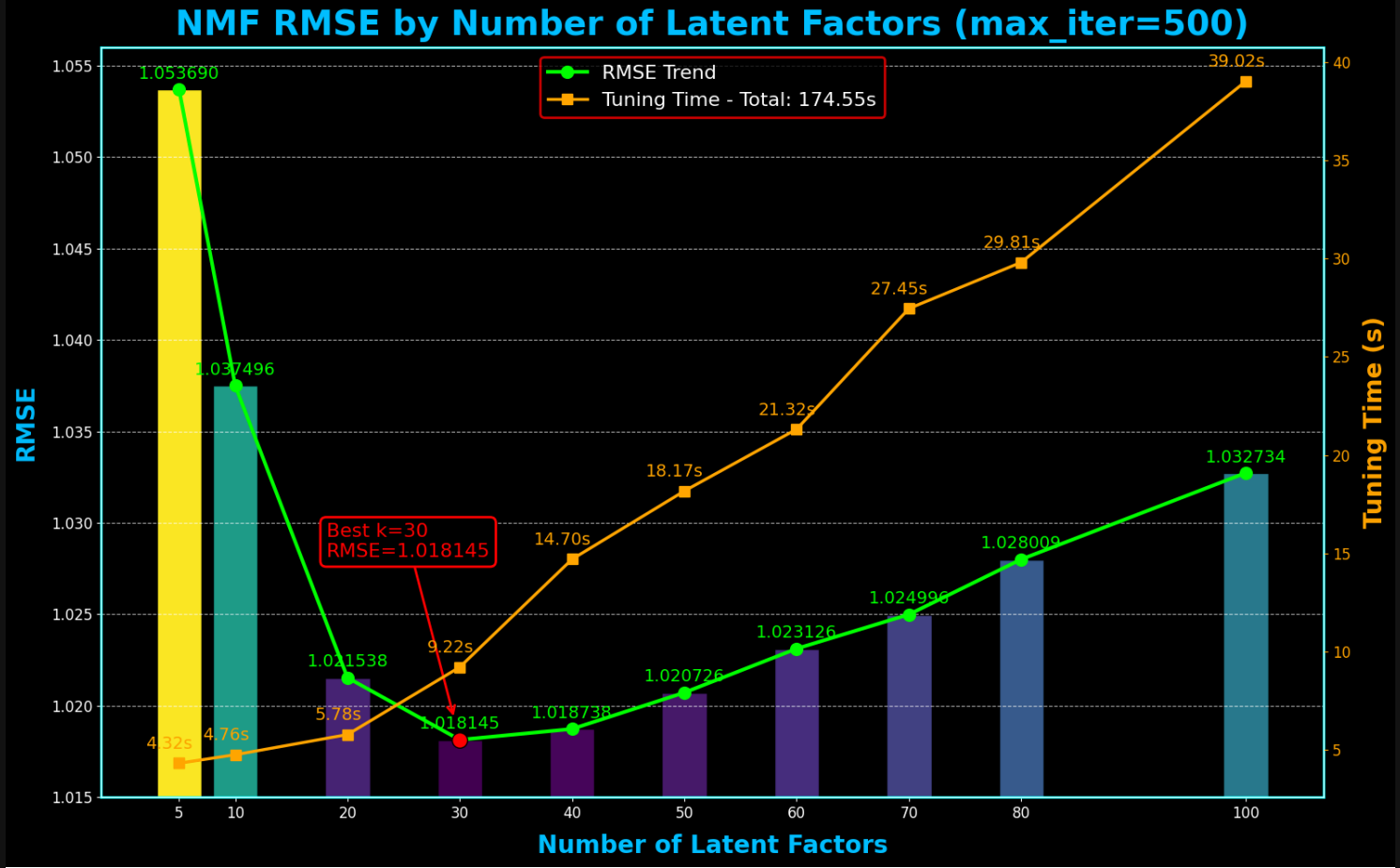
5.2b Compare RMSE and Tuning Time

Here we extend the visualization by combining RMSE scores with total tuning time for each latent factor setting. RMSE is shown as colored bars with a trend line, while tuning time is overlaid as a secondary axis. This dual-axis plot helps evaluate trade-offs between accuracy and computational cost, highlighting the optimal k value in context of both performance and efficiency.

In [ ]:

```
# RMSE Bar + Line Plot
fig, ax = plt.subplots(figsize=(16, 10))

# Normalize RMSE values for colormap
norm = plt.Normalize(nmf_results_df['RMSE'].min(), nmf_results_df['RMSE'].max())
colors = plt.cm.viridis(norm(nmf_results_df['RMSE']))

# Bar plot
bars = ax.bar(
    nmf_results_df['Number of Factors'],
    nmf_results_df['RMSE'],
    width=4,
    color=colors,
    edgecolor='black'
)

# Line plot for RMSE (trend) on left axis
rmse_line, = ax.plot(
    nmf_results_df['Number of Factors'],
    nmf_results_df['RMSE'],
    color='lime',
    marker='o',
    linewidth=3,
    markersize=10,
    label='RMSE Trend'
)

# Add RMSE labels above each bar
for bar, rmse in zip(bars, nmf_results_df['RMSE']):
    ax.text(
        bar.get_x() + bar.get_width() / 2,
        bar.get_height() + 0.0004,
        f"{rmse:.6f}",
        ha='center',
        va='bottom',
        fontsize=14,
        color='lime'
    )
```

```python
# Mark best k
best_idx  = nmf_results_df['RMSE'].idxmin()
best_k    = int(nmf_results_df.loc[best_idx, 'Number of Factors'])
best_rmse = float(nmf_results_df.loc[best_idx, 'RMSE'])

# Scatter point at best k
ax.scatter([best_k], [best_rmse], s=160, color='red', edgecolor='black', zorder=5)

# Annotate with arrow
ax.annotate(
    f"Best k={best_k}\nRMSE={best_rmse:.6f}",
    xy=(best_k, best_rmse),
    xytext=(-110, 150),
    textcoords='offset points',
    fontsize=16,
    color='red',
    bbox=dict(boxstyle='round,pad=0.3', facecolor='black', edgecolor='red', linewidth=2),
    arrowprops=dict(
        arrowstyle='->',
        color='red',
        lw=2,
        shrinkA=0,
        shrinkB=20,
    )
)

# Set the title and labels with custom font properties.
ax.set_title('NMF RMSE by Number of Latent Factors (max_iter=500)', fontsize=28, color='deepskyblue', fontweight='bold', pad=10)
ax.set_xlabel('Number of Latent Factors', fontsize=20, color='deepskyblue', fontweight='bold', labelpad=10)
ax.set_ylabel('RMSE', fontsize=20, color='deepskyblue', fontweight='bold', labelpad=10)
ax.tick_params(colors="white", labelsize=12)

# Set the y-axis limit to fit rating counts.
ax.set_ylim(1.015, 1.056)
ax.set_xticks(nmf_results_df['Number of Factors'])
ax.grid(axis='y', linestyle='--', alpha=0.7)

# Secondary axis for tuning time
ax2 = ax.twinx()
time_line, = ax2.plot(
    nmf_results_df['Number of Factors'],
    nmf_results_df['Tuning Time (s)'],
    color='orange',
    marker='s',
    linewidth=2.5,
    markersize=9,
    label='Tuning Time (s)'
)
ax2.set_ylabel('Tuning Time (s)', fontsize=20, color='orange', fontweight='bold', labelpad=10)
ax2.tick_params(axis='y', colors='orange', labelsize=12)

# Label time near markers
for x, t in zip(nmf_results_df['Number of Factors'], nmf_results_df['Tuning Time (s)']):
    ax2.annotate(
        f"{t:.2f}s",
        xy=(x, t),
        xytext=(-8.0, 9),
        textcoords='offset points',
        ha='center',
        va='bottom',
        fontsize=14,
        color='orange'
    )

# Style the plot's border (spines).
for spine in ax.spines.values():
    spine.set_edgecolor('cyan')
    spine.set_linewidth(2)

# Set the background color of the axes.
ax.set_facecolor('black')

# Legend: combine handles from both axes
handles = [rmse_line, time_line]
total_time = nmf_results_df['Tuning Time (s)'].sum()
labels = [
    'RMSE Trend',
    f'Tuning Time - Total: {total_time:.2f}s'
]
legend = ax.legend(handles, labels, fontsize=16, frameon=True, loc='upper center')

# Style legend frame and background
legend.get_frame().set_linewidth(2)
legend.get_frame().set_edgecolor('red')

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



Observation - RMSE Trend and Performance Trade-offs

The RMSE vs. Number of Latent Factors plot in 5.2a shows a clear "hook" shape, with the error decreasing rapidly before flattening and eventually rising again. The minimum RMSE occurs at k = 30, suggesting this factor count provides the best predictive accuracy without overfitting.

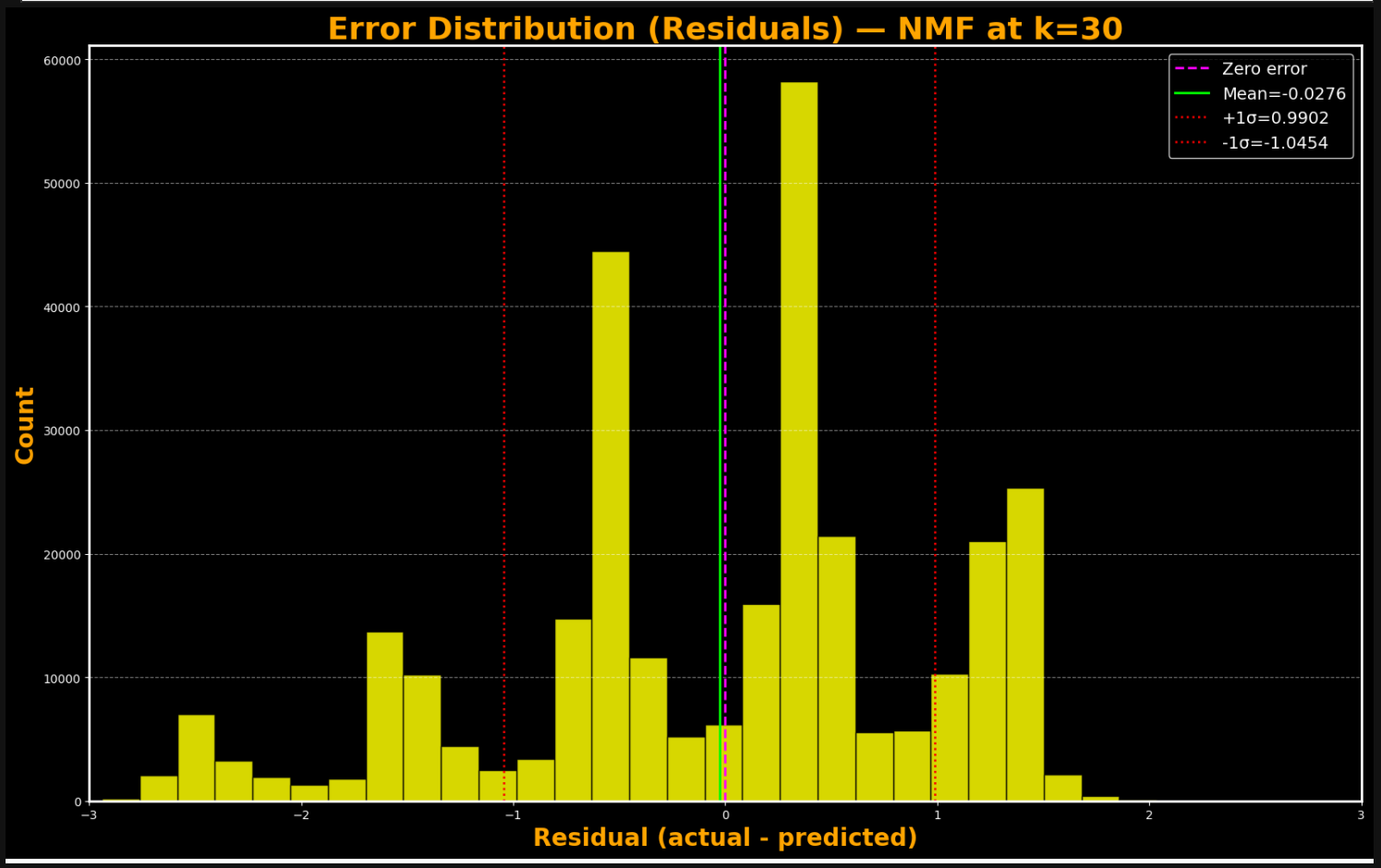
In 5.2b, adding tuning time as a secondary measure provides more context. While RMSE differences between k values near the minimum are small, computation time increases sharply with higher factor counts. For example, moving from 30 to 100 factors yields no accuracy gain and increases processing time more than threefold. This confirms that k = 30 is the optimal choice when balancing accuracy and efficiency.

5.3 Error Distribution Analysis

We quantify the prediction error of the tuned NMF model by examining the residuals (actual vs predicted) on the test set. The goal is to check bias, spread, and tail behavior. A well-behaved model will center near zero with limited skew and no heavy tails.

In [15]:
```
# Use the best k from 5.2
best_idx = nmf_results_df['RMSE'].idxmin()
best_k   = int(nmf_results_df.loc[best_idx, 'Number of Factors'])

# Fit NMF once at best_k, then compute predictions on test
nmf_best = NMF(n_components=best_k, init='random', random_state=SEED, max_iter=500)
W_best   = nmf_best.fit_transform(utility_matrix_np)
H_best   = nmf_best.components_
R_best   = np.dot(W_best, H_best)

y_pred = get_predictions(test_df, R_best, global_average_rating, user_to_idx, movie_to_idx)
y_true = true_ratings
residuals = y_true - np.asarray(y_pred)

# Histogram + summary lines
fig, ax = plt.subplots(figsize=(16, 10))
ax.hist(residuals, bins=40, color='yellow', edgecolor='black', alpha=0.85)

mu = residuals.mean()
sd = residuals.std()
ax.axvline(0.0, color='magenta', linestyle='--', linewidth=2, label='Zero error')
ax.axvline(mu,  color='lime', linestyle='-',  linewidth=2, label=f'Mean={mu:.4f}')
ax.axvline(mu+sd,  color='red',   linestyle=':',  linewidth=1.8, label=f'+1σ={mu+sd:.4f}')
ax.axvline(mu-sd,  color='red',   linestyle=':',  linewidth=1.8, label=f'-1σ={mu-sd:.4f}')

# Set the title and labels with custom font properties.
ax.set_title(f'Error Distribution (Residuals) — NMF at k={best_k}', fontsize=26, color='orange', fontweight='bold')
ax.set_xlabel('Residual (actual - predicted)', fontsize=20, color='orange', fontweight='bold')
```

```
ax.set_ylabel('Count', fontsize=20, color='orange', fontweight='bold')
ax.grid(axis='y', linestyle='--', alpha=0.5)

# Style the plot's border (spines).
for s in ax.spines.values():
    s.set_edgecolor('white'); s.set_linewidth(2)

# Set the background color of the axes.
ax.set_facecolor('black')

# Add and style the plot legend.
ax.legend(facecolor='black', edgecolor='white', fontsize=14)

# Set the limits for the x-axis.
ax.set_xlim(-3.0, 3.0)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



Observation - Error Distribution Analysis

The residual distribution for the NMF model at k = 30 is centered close to zero (mean = -0.0276), indicating minimal bias in predictions. The dashed black line marks zero error, while the gold line marks the actual mean residual. The red dotted lines represent one standard deviation above and below the mean (+1σ ≈ 0.9902, -1σ ≈ -1.0454). Most residuals fall within this range, suggesting stable predictions with relatively few extreme over- or underestimations. The distribution is slightly skewed to the negative side, indicating a small tendency for the model to slightly overpredict ratings.

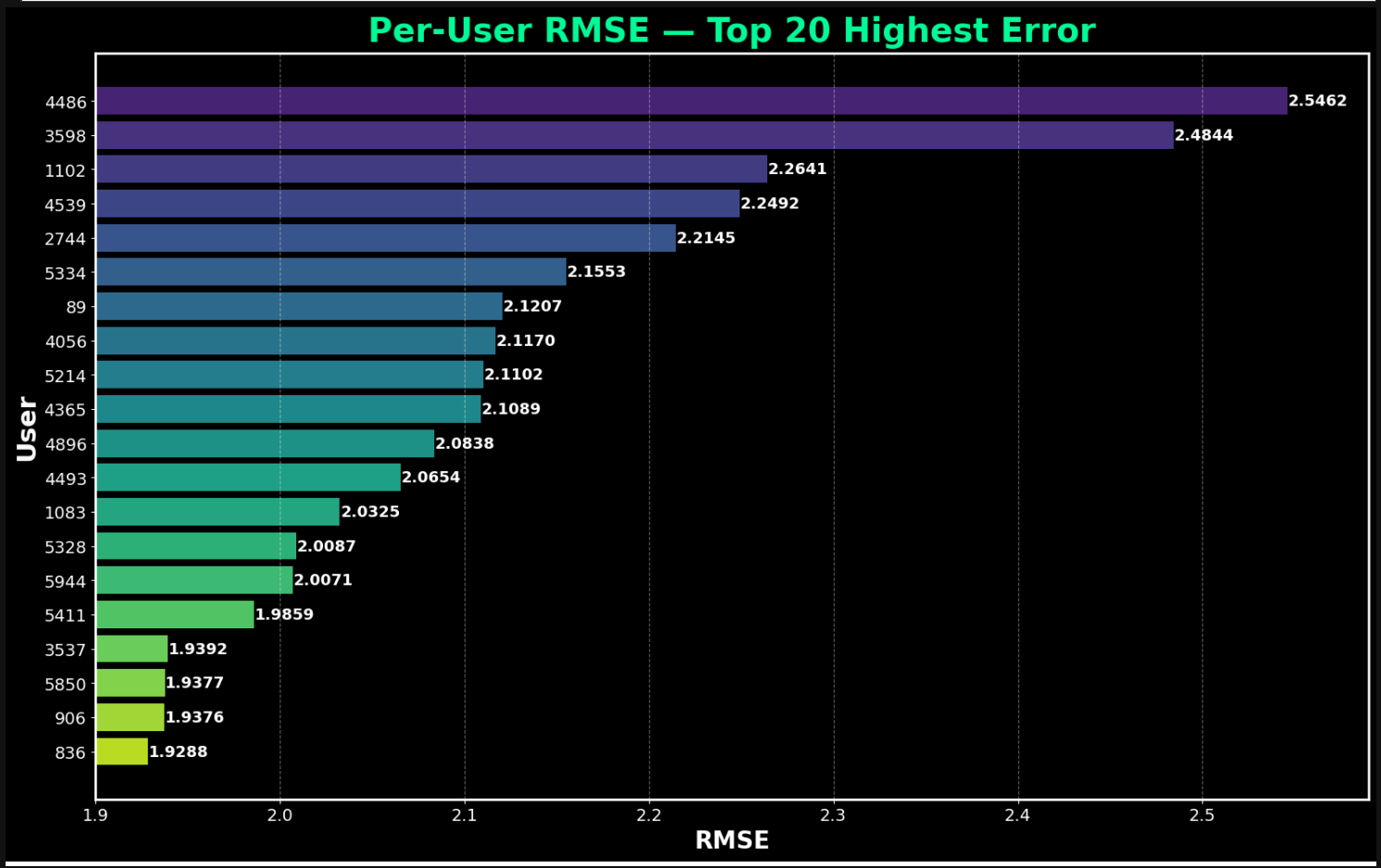
5.4a Per-User RMSE (Top Outliers)

Ranking users by test-set RMSE to surface outliers. Horizontal bars improve label legibility and make it easy to compare error magnitude. This view helps diagnose cold-start users and inconsistent rating behavior.

In [35]:

```
# Build a DataFrame with residuals for each test row
err_df = test_df[['uID','mID','rating']].copy()
err_df['pred'] = y_pred
err_df['res']  = err_df['rating'] - err_df['pred']
err_df['sqerr'] = err_df['res']**2

# Per-user RMSE
per_user = (err_df.groupby('uID')['sqerr']
            .mean()
            .pipe(np.sqrt)
            .reset_index(name='RMSE'))
```

```
# Pick top N worst users
TOPN = 20
top_users = per_user.sort_values('RMSE', ascending=False).head(TOPN)

# Create a horizontal bar plot
fig, ax = plt.subplots(figsize=(16, 10), facecolor='black')
y_labels = top_users['uID'].astype(str)
bars = ax.barh(y_labels, top_users['RMSE'], color=plt.cm.viridis(np.linspace(0.1,0.9,TOPN)))
ax.invert_yaxis()

# Add the RMSE score as a text label for each user's bar.
for i,(uid, rmse) in enumerate(zip(top_users['uID'], top_users['RMSE'])):
    ax.text(rmse + 0.0004, i, f'{rmse:.4f}', va='center', fontsize=13, color='white', fontweight='bold')

# Set the title and labels with custom font properties.
ax.set_title('Per-User RMSE — Top 20 Highest Error', fontsize=28, color='mediumspringgreen', fontweight='bold', pad=10)
ax.set_xlabel('RMSE', fontsize=20, color='white', fontweight='bold')
ax.set_ylabel('User', fontsize=22, color='white', fontweight='bold')
ax.tick_params(axis='x', labelsize=14, colors='white')
ax.tick_params(axis='y', labelsize=14, colors='white')

# Style the plot's border (spines).
for s in ax.spines.values():
    s.set_edgecolor('white'); s.set_linewidth(2)
ax.grid(axis='x', linestyle='--', alpha=0.4)

# Set the limits for the x-axis.
ax.set_xlim(1.9, 2.59)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```

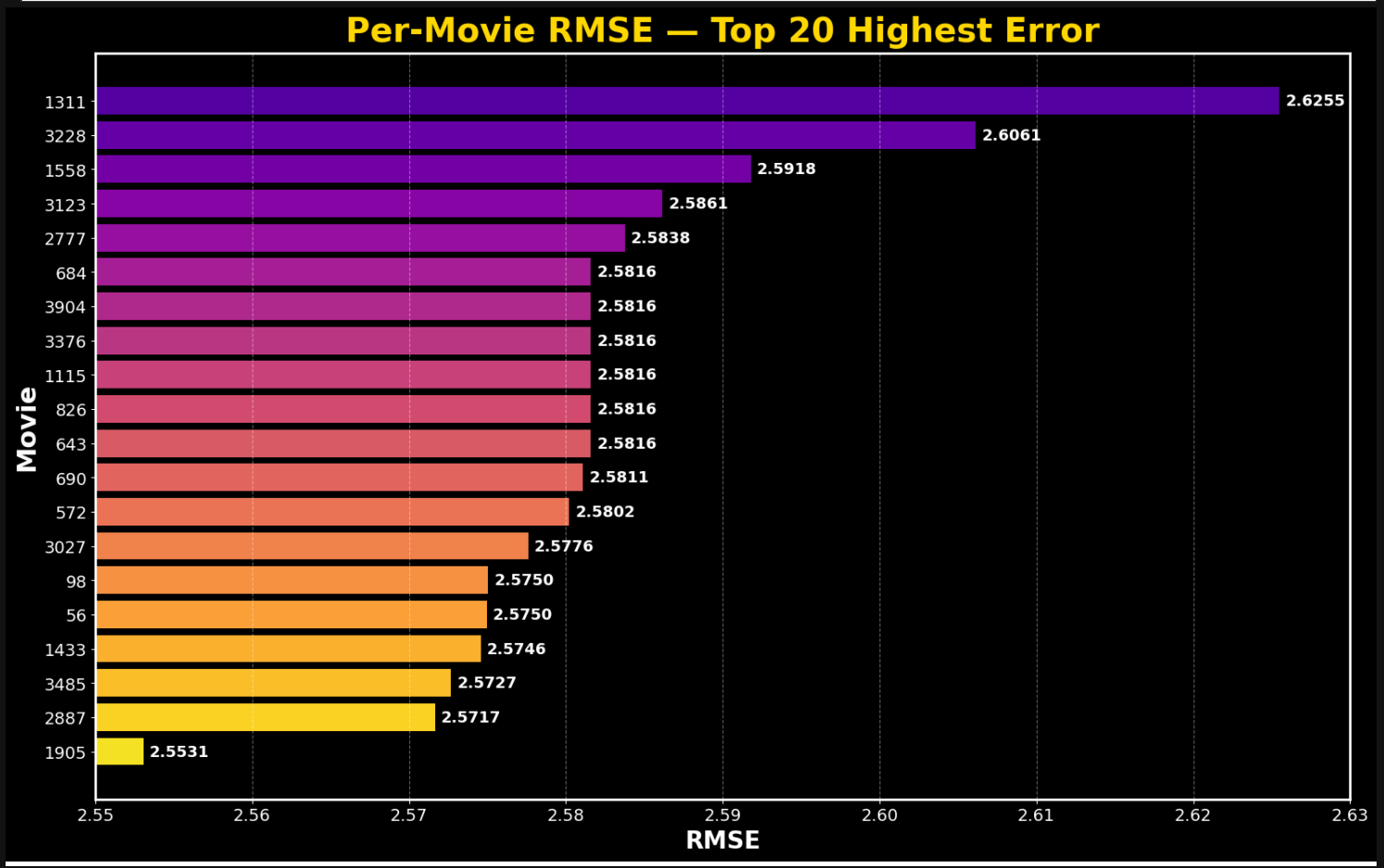


5.4b Per-Movie RMSE (Top Outliers)

Repeating the analysis at the item level to identify movies with high error. This often points to sparse items, atypical rating behavior, or genre effects that simple NMF cannot capture.

In [17]:
```
# Calculate the RMSE for each individual movie.
per_movie = (err_df.groupby('mID')['sqerr']
             .mean()
             .pipe(np.sqrt)
             .reset_index(name='RMSE'))
```

```
# Get the top 20 movies with the highest prediction error (worst predictions).
TOPN = 20
top_movies = per_movie.sort_values('RMSE', ascending=False).head(TOPN)

# Create a horizontal bar plot
fig, ax = plt.subplots(figsize=(16, 10), facecolor='black')
y_labels = top_movies['mID'].astype(str)
bars = ax.barh(y_labels, top_movies['RMSE'], color=plt.cm.plasma(np.linspace(0.15,0.95,TOPN)))
ax.invert_yaxis()

# Add the exact RMSE score as a text label for each movie's bar.
for i,(mid, rmse) in enumerate(zip(top_movies['mID'], top_movies['RMSE'])):
    ax.text(rmse + 0.0004, i, f'{rmse:.4f}', va='center', fontsize=13, color='white', fontweight='bold')

# Set the title and labels with custom font properties.
ax.set_title('Per-Movie RMSE — Top 20 Highest Error', fontsize=28, color='gold', fontweight='bold', pad=10)
ax.set_xlabel('RMSE', fontsize=20, color='white', fontweight='bold')
ax.set_ylabel('Movie', fontsize=22, color='white', fontweight='bold')
ax.tick_params(axis='x', labelsize=14, colors='white')
ax.tick_params(axis='y', labelsize=14, colors='white')

# Style the plot's border (spines).
for s in ax.spines.values():
    s.set_edgecolor('white'); s.set_linewidth(2)
ax.grid(axis='x', linestyle='--', alpha=0.4)

# Set the limits for the x-axis.
ax.set_xlim(2.55, 2.63)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



5.4c Validating the Sparsity Assumption for 5.4a and 5.4b Outliers (Top Outliers)

In 5.4a and 5.4b, I identified the top 20 users and movies with the highest RMSE.
My initial observation was that these outliers likely came from users with sparse rating histories or from items with limited training data, which can make them harder for the model to predict accurately.

Rather than relying on this as an assumption, I calculated the actual sparsity for the exact user and movie IDs from 5.4a and 5.4b.
Sparsity is measured as the percentage of missing ratings in the training set, based on the total possible ratings each user or movie could have.

For movies with **zero ratings** in the training set, predictions are effectively cold-start guesses, which can explain extreme error values.
This table provides the definitive sparsity values for each entity, confirming or rejecting whether high RMSE in 5.4a and 5.4b aligns with sparse rating

histories.

In [18]:
```
# Actual IDs from subsections 5.4a and 5.4b RMSE plots
users_from_5_4a = [4486, 3598, 1102, 4539, 2744, 5334, 89, 4056, 5214, 4365, 4896, 4493, 1083, 5328, 5944, 5411, 3537, 5850, 906, 836]
movies_from_5_4b = [1311, 3228, 1558, 3123, 2777, 684, 3904, 3376, 1115, 826, 643, 690, 572, 3027, 98, 56, 1433, 3485, 2887, 1905]

# Make sure IDs are same dtype as the matrix
utility_matrix.index = utility_matrix.index.astype(int)
utility_matrix.columns = utility_matrix.columns.astype(int)

# Per-user sparsity for Top 20 Users
user_rows = []
n_movies = utility_matrix.shape[1]
for uid in users_from_5_4a:
    if uid in utility_matrix.index:
        row = utility_matrix.loc[uid]
        total = n_movies
        have  = int(row.count())
    else:
        total = n_movies
        have  = 0  # treat as zero ratings if ID not present
    miss = total - have
    user_rows.append({
        "UserID": uid,
        "Total Cells": total,
        "Actual Ratings": have,
        "Missing Ratings": miss,
        "Sparsity (%)": round(100 * miss / total, 2)
    })
user_sparsity_df = pd.DataFrame(user_rows)

# Per-movie sparsity for Top 20 Movies
movie_rows = []
n_users = utility_matrix.shape[0]
for mid in movies_from_5_4b:
    if mid in utility_matrix.columns:
        col   = utility_matrix[mid]
        total = n_users
        have  = int(col.count())
    else:
        total = n_users
        have  = 0  # treat as cold-start item
    miss = total - have
    movie_rows.append({
        "MovieID": mid,
        "Total Cells": total,
        "Actual Ratings": have,
        "Missing Ratings": miss,
        "Sparsity (%)": round(100 * miss / total, 2)
    })
movie_sparsity_df = pd.DataFrame(movie_rows)

# Table styles
table_styles = [
    {'selector': 'thead th', 'props': [
        ('background-color', '#DA291C'),
        ('color', 'white'),
        ('font-size', '16px'),
        ('font-weight', 'bold'),
    ]},
    {'selector': 'tbody th', 'props': [
        ('background-color', '#0033A0'),
        ('color', 'white'),
        ('font-size', '16px'),
        ('font-weight', 'bold'),
    ]}
]

# Styled tables
styled_users = (
    user_sparsity_df.style
        .background_gradient(subset=['Sparsity (%)'], cmap='bwr')
        .bar(subset=['Actual Ratings'], color='#1f77b4')
        .bar(subset=['Missing Ratings'], color='#ff7f0e')
        .format({
            'Total Cells': '{:,}',
            'Actual Ratings': '{:,}',
            'Missing Ratings': '{:,}',
            'Sparsity (%)': '{:.2f}'
        }, na_rep='-')
        .set_properties(**{'border': '1px solid #444', 'color': 'white'})
        .set_table_styles(table_styles)
)

styled_movies = (
    movie_sparsity_df.style
        .background_gradient(subset=['Sparsity (%)'], cmap='bwr')
```

```
        .bar(subset=['Actual Ratings'], color='#1f77b4')
        .bar(subset=['Missing Ratings'], color='#ff7f0e')
        .format({
            'Total Cells': '{:,}',
            'Actual Ratings': '{:,}',
            'Missing Ratings': '{:,}',
            'Sparsity (%)': '{:.2f}'
        }, na_rep='-')
        .set_properties(**{'border': '1px solid #444', 'color': 'white'})
        .set_table_styles(table_styles)
)

print('--- Per-user Sparsity (5.4a Users) ---')
display(styled_users)

print('--- Per-movie Sparsity (5.4b Movies) ---')
display(styled_movies)
```

```
--- Per-user Sparsity (5.4a Users) ---
```

	UserID	Total Cells	Actual Ratings	Missing Ratings	Sparsity (%)
0				3,629	99.04
1	3598	3,664	44	3,620	98.80
2				3,643	99.43
3	4539	3,664	88	3,576	97.60
4			90	3,574	97.54
5	5334	3,664	43	3,621	98.83
6				3,647	99.54
7	4056	3,664	16	3,648	99.56
8				3,644	99.45
9	4365	3,664	15	3,649	99.59
10				3,640	99.34
11	4493	3,664	36	3,628	99.02
12				3,645	99.48
13	5328	3,664	40	3,624	98.91
14				3,639	99.32
15	5411	3,664	16	3,648	99.56
16				3,624	98.91
17	5850	3,664	32	3,632	99.13
18				3,647	99.54
19	856	3,664	47	3,617	98.72

```
--- Per-movie Sparsity (5.4b Movies) ---
```

	MovieID	Total Cells	Actual Ratings	Missing Ratings	Sparsity (%)
0				6,035	99.92
1	3228	6,040	1	6,039	99.98
2				6,039	99.98
3	3123	6,040	2	6,038	99.97
4				6,036	99.93
5	684	6,040	0	6,040	100.00
6				6,040	100.00
7	3376	6,040	0	6,040	100.00
8				6,040	100.00
9	826	6,040	0	6,040	100.00
10				6,038	99.97
11	690	6,040	1	6,039	99.98
12				6,039	99.98
13	3027	6,040	5	6,035	99.92
14				6,035	99.92
15	56	6,040	8	6,032	99.87
16				6,036	99.93
17	3485	6,040	1	6,039	99.98
18				6,037	99.95
19	1905	6,040	1	6,039	99.98

5.4d Correlating RMSE with Sparsity for Top Error Users and Movies

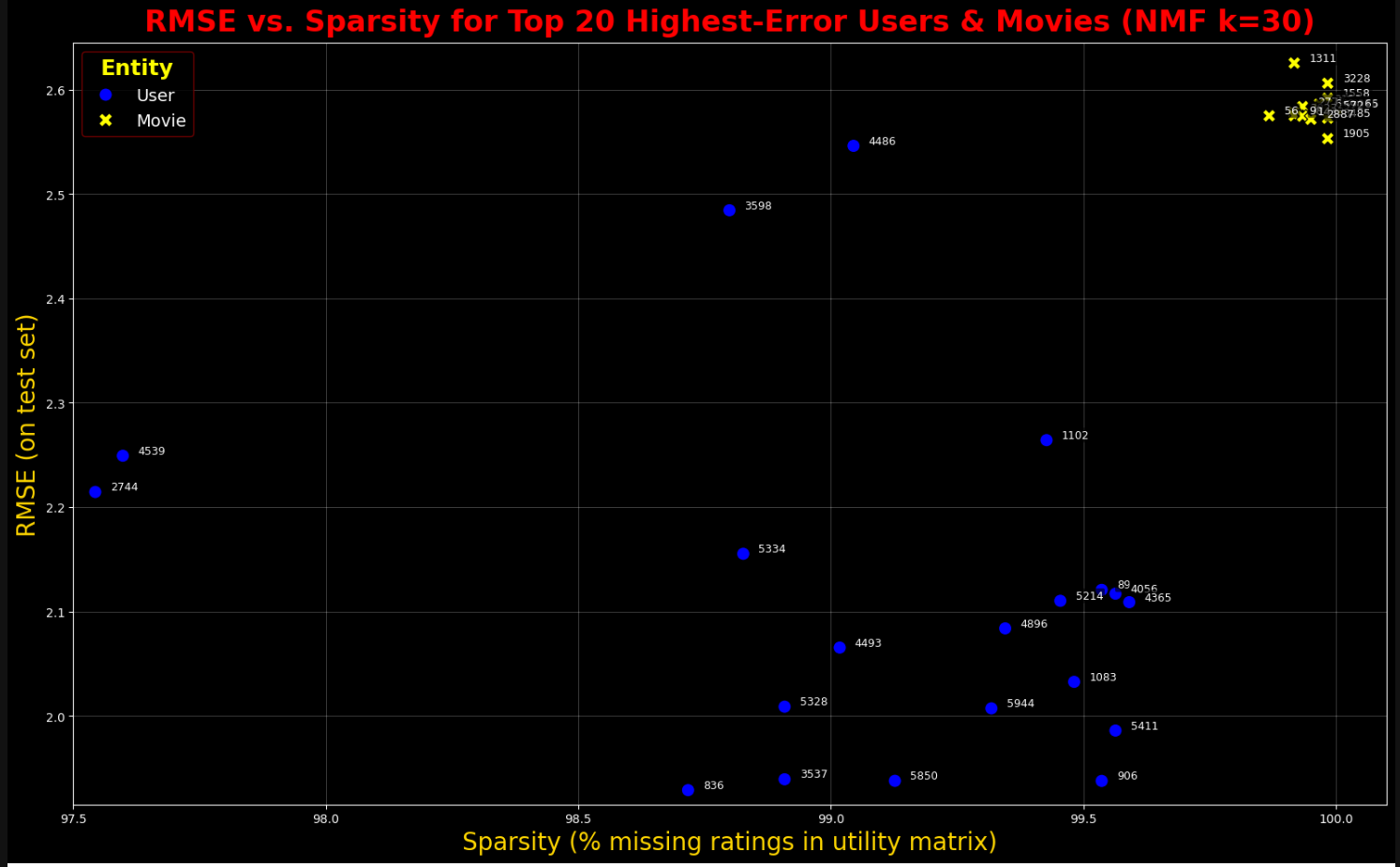
After confirming the actual sparsity for the outliers in 5.4a and 5.4b (Section 5.4c), this step visualizes the relationship between RMSE and sparsity for those same top 20 highest-error users and movies.

Using the best-performing NMF model from Section 5.2 (`k = best_k`), I calculated the RMSE for each user and movie in the test set and combined it with their sparsity from the training utility matrix. The resulting scatter plot displays both users and movies on the same axes, with point labels showing their IDs.

This view allows a direct comparison to see whether the model's largest prediction errors are consistently tied to entities with higher sparsity, and highlights cases where high RMSE occurs despite relatively low sparsity, suggesting other contributing factors.

In [19]:
```
# Ensure we have predictions for the BEST k
def _ensure_predictions_for_best_k(best_k):
    if 'reconstructed_matrix_best' in globals():
        R = reconstructed_matrix_best
    else:
        nmf_model = NMF(n_components=best_k, init='random', random_state=SEED, max_iter=500)
        W = nmf_model.fit_transform(utility_matrix_np)
        H = nmf_model.components_
        R = np.dot(W, H)

    # Vectorized predictions
    preds = get_predictions(test_df, R, global_average_rating, user_to_idx, movie_to_idx)
    return preds

# Find the best number of factors (k) from the tuning results in 5.2 if not already defined.
if 'best_k' not in globals():
    plot_df = nmf_results_df.sort_values('Number of Factors').reset_index(drop=True)
    best_idx = plot_df['RMSE'].idxmin()
    best_k = int(plot_df.loc[best_idx, 'Number of Factors'])

# Get the predictions using the best number of factors.
preds = _ensure_predictions_for_best_k(best_k)

# Create a DataFrame to hold the true ratings, predictions, and squared errors.
eval_df = test_df[['uID','mID','rating']].copy()
```

```python
eval_df['pred'] = preds
eval_df['se'] = (eval_df['rating'] - eval_df['pred'])**2

# Calculate the RMSE for each individual user and each individual movie.
per_user = (eval_df.groupby('uID')['se'].mean()**0.5).rename('RMSE').reset_index()
per_movie = (eval_df.groupby('mID')['se'].mean()**0.5).rename('RMSE').reset_index()

# Get the top 20 users and movies with the highest prediction errors.
top20_users_df  = per_user.sort_values('RMSE', ascending=False).head(20).copy()
top20_movies_df = per_movie.sort_values('RMSE', ascending=False).head(20).copy()


# Calculate the number of ratings given by each user and received by each movie.
user_counts  = utility_matrix.count(axis=1)
movie_counts = utility_matrix.count(axis=0)
n_movies = utility_matrix.shape[1]
n_users  = utility_matrix.shape[0]

# Add sparsity (% missing) to top users
top20_users_df['Actual Ratings'] = top20_users_df['uID'].map(user_counts).fillna(0).astype(int)
top20_users_df['Total Cells']    = n_movies
top20_users_df['Missing']        = top20_users_df['Total Cells'] - top20_users_df['Actual Ratings']
top20_users_df['Sparsity (%)']   = 100 * top20_users_df['Missing'] / top20_users_df['Total Cells']
top20_users_df.rename(columns={'uID':'ID'}, inplace=True)
top20_users_df['Type'] = 'User'

# Add sparsity (% missing) to top movies
top20_movies_df['Actual Ratings'] = top20_movies_df['mID'].map(movie_counts).fillna(0).astype(int)
top20_movies_df['Total Cells']    = n_users
top20_movies_df['Missing']        = top20_movies_df['Total Cells'] - top20_movies_df['Actual Ratings']
top20_movies_df['Sparsity (%)']   = 100 * top20_movies_df['Missing'] / top20_movies_df['Total Cells']
top20_movies_df.rename(columns={'mID':'ID'}, inplace=True)
top20_movies_df['Type'] = 'Movie'

# Combine for one correlation plot
both = pd.concat([top20_users_df[['ID','Type','RMSE','Sparsity (%)']],
                  top20_movies_df[['ID','Type','RMSE','Sparsity (%)']]], ignore_index=True)

# Set up the plot
fig, ax = plt.subplots(figsize=(16, 10))

# Create a scatter plot
sns.scatterplot(
    data=both,
    x='Sparsity (%)',
    y='RMSE',
    hue='Type',
    style='Type',
    palette={'User': 'blue', 'Movie': 'yellow'},
    markers={'User': 'o', 'Movie': 'X'},
    s=120,
    edgecolor='black',
    ax=ax
)

# Label points with IDs
for _, r in both.iterrows():
    ax.text(
        r['Sparsity (%)'] + 0.03,
        r['RMSE'] + 0.002,
        str(int(r['ID'])),
        fontsize=9,
        color='white',
        bbox=dict(boxstyle='round,pad=0.15', fc='black', ec='none', alpha=0.6)
    )

# Set the title and labels with custom font properties.
ax.set_title(f'RMSE vs. Sparsity for Top 20 Highest-Error Users & Movies (NMF k={best_k})', fontsize=24, color='red', fontweight='bold', pad=10)
ax.set_xlabel('Sparsity (% missing ratings in utility matrix)', fontsize=20, color='gold')
ax.set_ylabel('RMSE (on test set)', fontsize=20, color='gold',)

# Add a light grid to the background
ax.grid(alpha=0.25)

# Set the limits for the x-axis.
ax.set_xlim(97.5, 100.10)

# Set the limits for the y-axis.
ax.set_ylim(1.915, 2.645)

# Add a custom-styled legend
leg = ax.legend(facecolor='black', edgecolor='darkred', fontsize=14, title='Entity', title_fontsize=18)
plt.setp(leg.get_title(), fontweight='bold', color='yellow')

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



Observation - RMSE and Sparsity Analysis of High-Error Entities

Looking at the per-user and per-movie RMSE charts, the spread in errors is much wider for users than for movies. User RMSE runs from about 1.93 up to 2.55, while movie RMSE sits in a much narrower band between roughly 2.55 and 2.63. This tells me the model has more trouble with certain user profiles than with most individual movies. A few users in particular, like IDs 4486, 3598, and 1102, stand out with much higher errors, which suggests the model is having a hard time predicting for them no matter what movies they rate.

When I look at the sparsity tables, the link between missing data and high error is clear. Many of the top-error users have over 99 percent of their ratings missing from the matrix. Some of them have fewer than 20 ratings total, which does not give the model much to work with. For movies, the issue is even more extreme. Several of the high-error titles have fewer than five total ratings, and a few are completely missing from the training set, so every rating for them in the test set is essentially a blind guess.

The scatter plot of RMSE against sparsity makes the pattern obvious. Movies are bunched in the top-right corner with both very high sparsity and high error. Users are more spread out. Most high-error users also have high sparsity, but a few with better coverage still show up here. That points to other factors at play, like unpredictable rating habits or tastes that don't match the majority of the population.

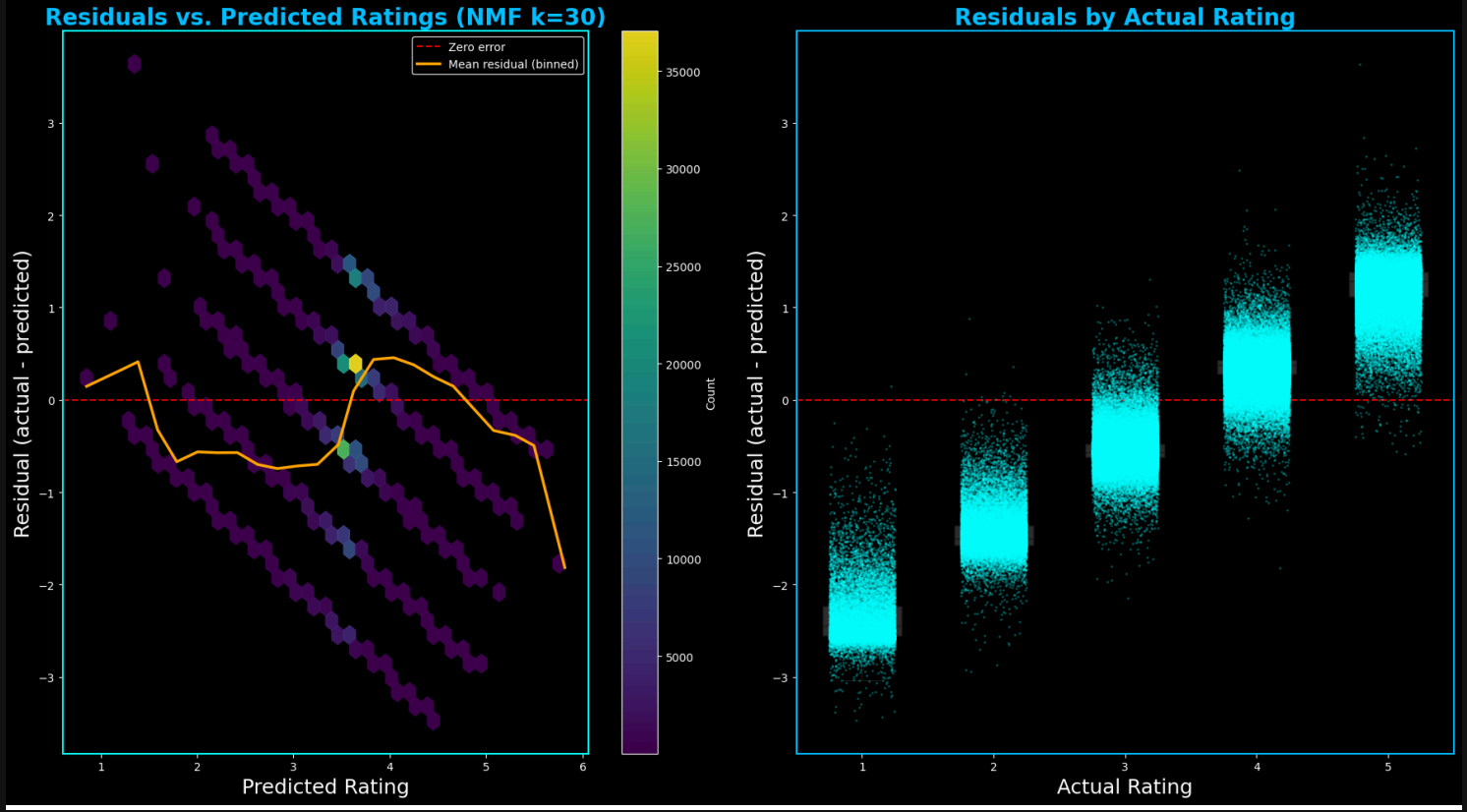
Overall, the main driver of high error for movies is the lack of ratings in training. For users, it is a mix of missing data and inconsistent patterns. To improve accuracy, we would likely get the most benefit from finding ways to add more ratings for these underrepresented entities or by giving the model a separate strategy for cases where data is extremely thin.

5.5 Residuals vs. Predicted Rating

These plots examine residual errors (actual minus predicted) in relation to both predicted ratings (left) and actual ratings (right). The aim is to identify systematic error patterns, such as bias toward over or under prediction in certain ranges, and to check for uneven variance of errors across the scale.

In [20]:
```
# Ignore the annoying pandas warning that isn't relevant here.
warnings.filterwarnings("ignore", category=FutureWarning, message="A grouping was used that is not in the columns of the DataFrame")

# A helper function to get predictions from the best NMF model.
def _ensure_predictions_for_best_k(best_k):
    if 'reconstructed_matrix_best' in globals():
        R = reconstructed_matrix_best
    else:
        nmf_model = NMF(n_components=best_k, init='random', random_state=SEED, max_iter=500)
        W = nmf_model.fit_transform(utility_matrix_np)
        H = nmf_model.components_
        R = np.dot(W, H)
    return get_predictions(test_df, R, global_average_rating, user_to_idx, movie_to_idx)

# Find the best number of factors from the tuning results, then get the predictions for that model.
```

```python
if 'best_k' not in globals():
    _df = nmf_results_df.sort_values('Number of Factors')
    best_k = int(_df.loc[_df['RMSE'].idxmin(), 'Number of Factors'])

preds = _ensure_predictions_for_best_k(best_k)

# Create a new DataFrame to analyze the model's errors (residuals).
res_df = test_df[['uID','mID','rating']].copy()
res_df['pred'] = preds
res_df['resid'] = res_df['rating'] - res_df['pred']

# Group the predictions into bins to calculate a trend line.
bins = np.linspace(res_df['pred'].min(), res_df['pred'].max(), 25)
bin_ids = np.digitize(res_df['pred'], bins)
trend = res_df.groupby(bin_ids, as_index=False).agg(
    pred_bin=('pred','mean'),
    resid_mean=('resid','mean'),
    resid_std=('resid','std')
)

# Set up the figure to hold two side-by-side plots.
plt.figure(figsize=(18, 10))
plt.subplots_adjust(wspace=0.25)

# Left: Residuals vs Predicted (hexbin density + mean trend)
ax1 = plt.subplot(1, 2, 1, facecolor='black')
hb = ax1.hexbin(res_df['pred'], res_df['resid'],
                gridsize=40, cmap='viridis', mincnt=1, alpha=0.9)
ax1.axhline(0, color='red', lw=1.5, ls='--', alpha=0.9, label='Zero error')
ax1.plot(trend['pred_bin'], trend['resid_mean'], color='orange', lw=2.5, label='Mean residual (binned)')
ax1.set_title(f"Residuals vs. Predicted Ratings (NMF k={best_k})",
           fontsize=20, color='deepskyblue', fontweight='bold')
ax1.set_xlabel("Predicted Rating", fontsize=18, color='white')
ax1.set_ylabel("Residual (actual - predicted)", fontsize=18, color='white')
ax1.tick_params(colors='white')
for s in ax1.spines.values():
    s.set_color('cyan'); s.set_linewidth(1.5)
cb = plt.colorbar(hb, ax=ax1)
cb.set_label('Count')
ax1.legend(facecolor='black', edgecolor='white')

# Right: Residuals by Actual Rating (box + strip)
ax2 = plt.subplot(1, 2, 2, facecolor='black')
sns.boxplot(data=res_df, x='rating', y='resid',
           ax=ax2, color='#303030', fliersize=0, linewidth=1, width=0.6)
sns.stripplot(data=res_df, x='rating', y='resid',
           ax=ax2, color='cyan', alpha=0.35, jitter=0.25, size=2)
ax2.axhline(0, color='red', lw=1.5, ls='--', alpha=0.9)
ax2.set_title("Residuals by Actual Rating", fontsize=20, color='deepskyblue', fontweight='bold')
ax2.set_xlabel("Actual Rating", fontsize=18, color='white')
ax2.set_ylabel("Residual (actual - predicted)", fontsize=18, color='white')
ax2.tick_params(colors='white')
for s in ax2.spines.values():
    s.set_color('deepskyblue'); s.set_linewidth(1.5)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



Observation - Residual Patterns and Bias

Residuals cluster along discrete diagonal bands due to the discrete rating scale and model prediction behavior. The mean residual line in the left plot shows slight under-prediction at lower predicted ratings and slight over-prediction in the midrange, with reduced bias near the extremes. The right plot confirms a wider spread of residuals for higher actual ratings, suggesting modest heteroscedasticity that may warrant further model calibration.

5.6 Train Final NMF Model & Evaluate

Train the final model at the best *k*, report RMSE & runtime, and save the model, factors, predictions, and run metadata for reproducibility.

In [21]:
```
# Find the best number of factors (k) from the tuning results if not already set.
if 'best_k' not in globals():
    _sorted = nmf_results_df.sort_values('Number of Factors').reset_index(drop=True)
    best_idx = _sorted['RMSE'].idxmin()
    best_k = int(_sorted.loc[best_idx, 'Number of Factors'])

print(f" --- Training final Non-negative Matrix Factorization (NMF) with best_k = {best_k} ---")

# Train the final NMF model using the best number of factors.
t0 = time.time()
final_nmf = NMF(
    n_components=best_k,
    init='random',
    random_state=SEED,
    max_iter=500
)
W_final = final_nmf.fit_transform(utility_matrix_np)
H_final = final_nmf.components_
R_final = np.dot(W_final, H_final)
train_runtime = time.time() - t0

# Evaluate on test set
final_preds = get_predictions(
    test_df, R_final, global_average_rating, user_to_idx, movie_to_idx
)
final_rmse = float(np.sqrt(mean_squared_error(test_df['rating'].to_numpy(), final_preds)))

print(f"Final NMF RMSE: {final_rmse:.6f}")
print(f"NMF Train/runtime: {train_runtime:.2f}s")

# Prepare a directory to save all the model outputs and results.
art_dir = "artifacts_nmf"
os.makedirs(art_dir, exist_ok=True)
ts = datetime.now().strftime("%Y%m%d-%H%M%S")

# Save the trained model
model_path = os.path.join(art_dir, f"nmf_model_k{best_k}_{ts}.joblib")
```

```python
joblib.dump(final_nmf, model_path)

# Save factor matrices (CSV)
W_df = pd.DataFrame(W_final, index=utility_matrix_imputed.index,
                    columns=[f"f{j+1}" for j in range(best_k)])
H_df = pd.DataFrame(H_final, index=[f"f{j+1}" for j in range(best_k)],
                    columns=utility_matrix_imputed.columns)

# Save the learned user-factor (W) and movie-factor (H) matrices to CSV files.
w_path = os.path.join(art_dir, f"W_users_k{best_k}_{ts}.csv")
h_path = os.path.join(art_dir, f"H_items_k{best_k}_{ts}.csv")
W_df.to_csv(w_path, index=True)
H_df.to_csv(h_path, index=True)

# Save predictions on test set (CSV)
preds_df = test_df[['uID','mID','rating']].copy()
preds_df['pred'] = final_preds
preds_df['residual'] = preds_df['rating'] - preds_df['pred']
preds_path = os.path.join(art_dir, f"test_predictions_k{best_k}_{ts}.csv")
preds_df.to_csv(preds_path, index=False)

# Create a dictionary of metadata to save all the run details in one place (JSON).
meta = {
    "timestamp": ts,
    "seed": int(SEED),
    "best_k": int(best_k),
    "max_iter": 500,
    "final_rmse": final_rmse,
    "runtime_seconds": round(train_runtime, 2),
    "shapes": {
        "W": list(W_final.shape),
        "H": list(H_final.shape),
        "R": list(R_final.shape)
    },
    "paths": {
        "model_joblib": model_path,
        "W_csv": w_path,
        "H_csv": h_path,
        "test_predictions_csv": preds_path
    }
}
meta_path = os.path.join(art_dir, f"run_metadata_k{best_k}_{ts}.json")
with open(meta_path, "w", encoding="utf-8") as f:
    json.dump(meta, f, indent=2)

# Print a summary confirming all the output files that were saved.
print("\n --- Artifacts saved ---")
print(f"  • Model:           {model_path}")
print(f"  • W factors (CSV): {w_path}")
print(f"  • H factors (CSV): {h_path}")
print(f"  • Predictions:     {preds_path}")
print(f"  • Metadata (JSON): {meta_path}")

# Display a final summary of the model's performance.
print("\n=== Final NMF Summary ===")
print(f"best_k: {best_k}")
print(f"final_RMSE: {final_rmse:.6f}")
print(f"train_runtime_sec: {train_runtime:.2f}")
print(f"users x items: {utility_matrix_imputed.shape}")
if hasattr(final_nmf, "n_iter_"):
    print(f"sklearn_n_iter_: {final_nmf.n_iter_}")
```

```
 --- Training final Non-negative Matrix Factorization (NMF) with best_k = 30 ---
Final NMF RMSE: 1.018145
NMF Train/runtime: 12.78s

 --- Artifacts saved ---
  • Model:           artifacts_nmf\nmf_model_k30_20250812-220604.joblib
  • W factors (CSV): artifacts_nmf\W_users_k30_20250812-220604.csv
  • H factors (CSV): artifacts_nmf\H_items_k30_20250812-220604.csv
  • Predictions:     artifacts_nmf\test_predictions_k30_20250812-220604.csv
  • Metadata (JSON): artifacts_nmf\run_metadata_k30_20250812-220604.json

=== Final NMF Summary ===
best_k: 30
final_RMSE: 1.018145
train_runtime_sec: 12.78
users x items: (6040, 3664)
sklearn_n_iter_: 500
```

Observation - Final NMF Model Evaluation

The final NMF model, trained with 30 latent factors, delivered an RMSE of 1.018 on the test set. Runtime was ~13 seconds on the full 6,040 × 3,664 utility matrix. While the iteration count reached the maximum of 500, performance gains from additional iterations are unlikely given the stability of the RMSE. The saved artifacts (model, factor matrices, predictions, and metadata) ensure the model can be reproduced and inspected later. This closes the evaluation phase with a configuration that balances predictive accuracy and computational efficiency.

Section 6: Modeling with Truncated Singular Value Decomposition (SVD)

In this section I apply **TruncatedSVD** to the same imputed, dense user–item matrix used for NMF. I mirror the tuning, visualization, error analysis, and final model evaluation from Section 5 to directly compare the approaches.

6.1 TruncatedSVD Latent Dimension Tuning

Sweep `n_components` across a small grid to observe the RMSE "hook" and pick a good rank. Keeping everything else identical to NMF for a clean comparison.

In [22]:
```
# This is a safety check to redefine the prediction function in case the notebook was restarted.
if 'get_predictions' not in globals():
    def get_predictions(test_data, reconstructed_matrix, default_rating, user_to_idx, movie_to_idx):
        user_idx  = test_data['uID'].map(user_to_idx).fillna(-1).astype(int).to_numpy()
        movie_idx = test_data['mID'].map(movie_to_idx).fillna(-1).astype(int).to_numpy()
        preds = np.full(len(test_data), default_rating, dtype=float)
        valid = (user_idx != -1) & (movie_idx != -1)
        preds[valid] = reconstructed_matrix[user_idx[valid], movie_idx[valid]]
        return preds

# Set up the tuning loop.
print("--- Tuning TruncatedSVD for the optimal number of components... ---")
start_time = time.time()

svd_factors_to_try = [5, 10, 20, 30, 40, 50, 60, 70, 80, 100]
svd_tuning_results = []
true_ratings = test_df['rating'].to_numpy()

# Create the main tqdm progress bar for the overall tuning process.
overall_bar = notebook_tqdm(
    total=len(svd_factors_to_try),
    desc="Tuning SVD Components",
    position=0, leave=True, dynamic_ncols=True
)

# Loop through each component value to train and evaluate a model.
for idx, k in enumerate(svd_factors_to_try):
    # Create a nested progress bar to show the steps for each individual model.
    inner_bar = notebook_tqdm(total=4, desc=f"SVD k={k}", position=idx+1, leave=True, dynamic_ncols=True)
    loop_start = time.time()

    # Fit the TruncatedSVD model on the data.
    svd = TruncatedSVD(n_components=k, random_state=SEED, n_iter=7)
    W = svd.fit_transform(utility_matrix_np)
    inner_bar.update(1)

    # Reconstruct the full ratings matrix.
    reconstructed = np.dot(W, svd.components_)
    inner_bar.update(1)

    # Get the predictions for the test set.
    preds = get_predictions(test_df, reconstructed, global_average_rating, user_to_idx, movie_to_idx)
    inner_bar.update(1)

    # Calculate the RMSE for this model's predictions.
    rmse = float(np.sqrt(mean_squared_error(true_ratings, preds)))
    inner_bar.update(1)

    # Store the results and close the inner progress bar.
    svd_tuning_results.append((k, rmse, time.time() - loop_start))
    inner_bar.close()
    overall_bar.update(1)

# Close the main progress bar.
overall_bar.close()

# Print the total time taken for the tuning process.
end_time = time.time()
print(f"\nTuning complete in {end_time - start_time:.2f} seconds.")
print("-"*70 + "\n")

# Convert the tuning results into a DataFrame for easier analysis.
svd_results_df = pd.DataFrame(svd_tuning_results, columns=['Number of Components', 'RMSE', 'Tuning Time (s)'])

# Find and display the best-performing number of components from the tuning results.
print("--- TruncatedSVD Tuning Results ---")
best_idx_svd   = svd_results_df['RMSE'].idxmin()
best_k_svd     = int(svd_results_df.loc[best_idx_svd, 'Number of Components'])
```

```
best_rmse_svd  = float(svd_results_df.loc[best_idx_svd, 'RMSE'])
print(f"And the Best k (SVD) Winnder is: {best_k_svd} with RMSE = {best_rmse_svd:.6f}")
print("-"*70 + "\n")

display(svd_results_df)
```

```
--- Tuning TruncatedSVD for the optimal number of components... ---
Tuning SVD Components:   0%|          | 0/10 [00:00<?, ?it/s]
SVD k=5:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=10:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=20:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=30:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=40:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=50:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=60:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=70:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=80:   0%|          | 0/4 [00:00<?, ?it/s]
SVD k=100:   0%|          | 0/4 [00:00<?, ?it/s]
Tuning complete in 5.36 seconds.
----------------------------------------------------------------------

--- TruncatedSVD Tuning Results ---
And the Best k (SVD) Winnder is: 20 with RMSE = 1.017281
----------------------------------------------------------------------
```

	Number of Components	RMSE	Tuning Time (s)
0	5	1.040082	0.274441
1	10	1.025487	0.303077
2	20	1.017281	0.404627
3	30	1.017722	0.467898
4	40	1.019609	0.501717
5	50	1.022518	0.569537
6	60	1.025264	0.603101
7	70	1.027942	0.655257
8	80	1.031155	0.704849
9	100	1.036342	0.824757

Observation 6.1 - TruncatedSVD Tuning Performance and Accuracy

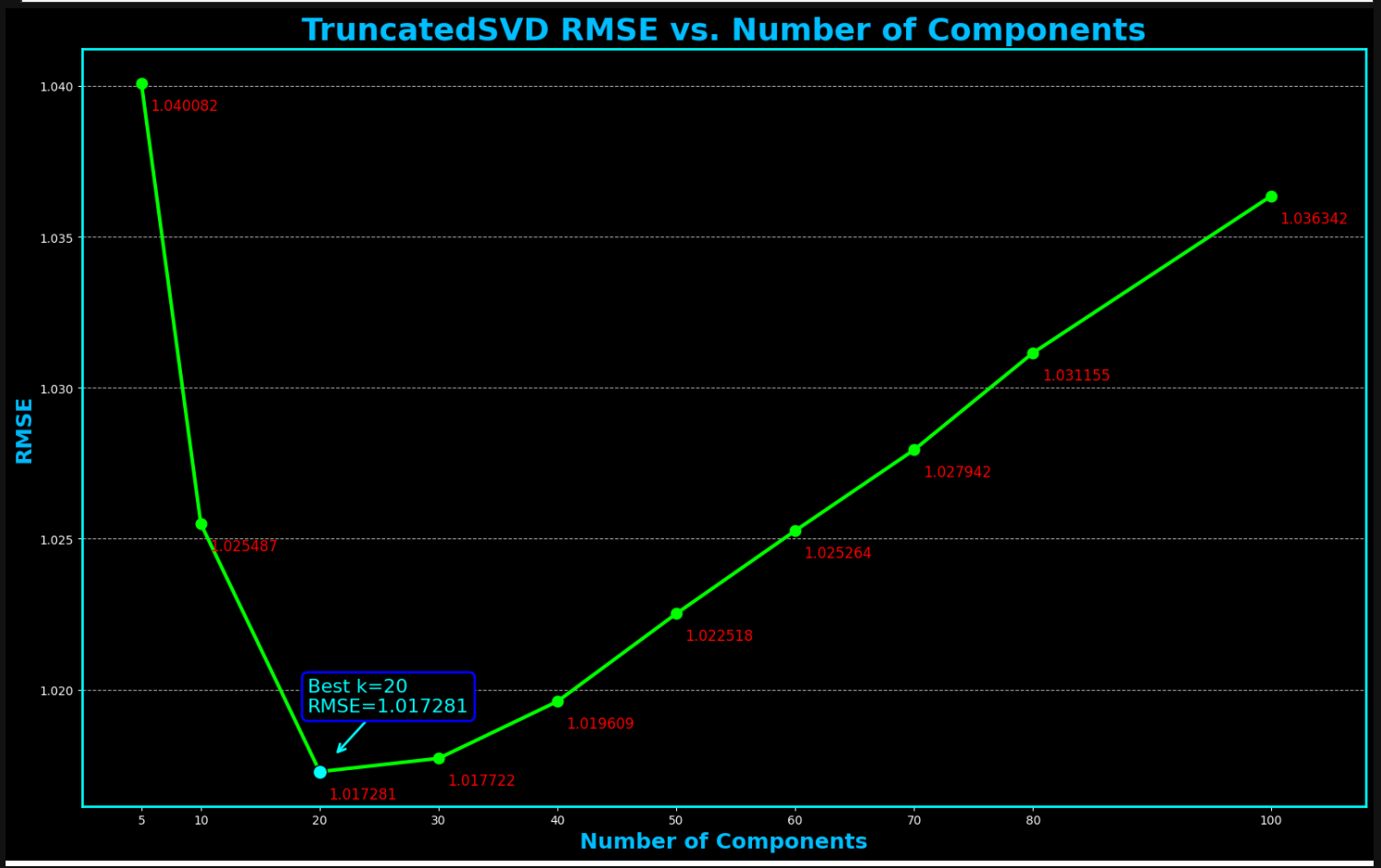
The TruncatedSVD tuning process completed in ~5 seconds (exact time varies per run), which is significantly faster than the NMF tuning in Section 5.1. The optimal number of components was k = 20, yielding an RMSE of 1.017281. Slightly outperforming the best NMF RMSE. The speed gain is likely due to SVD's more direct decomposition on the dense imputed matrix without iterative convergence steps, making it computationally lighter. This early result suggests that SVD may provide comparable or better accuracy than NMF with far less tuning overhead, which will be important to highlight in the Section 7 comparison.

6.2a Visualize the TruncatedSVD "Hook"

Plot RMSE against the number of components to reveal the familiar hook shape: quick early gains, a flat minimum, then worsening from over-capacity.

In [23]:
```
# Sort the tuning results by the number of components to prepare for plotting. Then find and store the best RMSE and the number of components that a
plot_df_svd = svd_results_df.sort_values('Number of Components').reset_index(drop=True)
best_idx = plot_df_svd['RMSE'].idxmin()
best_k_svd = int(plot_df_svd.loc[best_idx, 'Number of Components'])
best_rmse_svd = float(plot_df_svd.loc[best_idx, 'RMSE'])

# Set up the plot figure.
fig, ax = plt.subplots(figsize=(16, 10))

ax.plot(plot_df_svd['Number of Components'], plot_df_svd['RMSE'],
        color='lime', marker='o', linewidth=3, markersize=9, label='RMSE')

for x, y in zip(plot_df_svd['Number of Components'], plot_df_svd['RMSE']):
    ax.text(x + 0.75, y - 0.001, f"{y:.6f}", ha='left', va='bottom', fontsize=12, color='red')

ax.scatter([best_k_svd], [best_rmse_svd], s=140, color='cyan', edgecolor='black', zorder=5)
ax.annotate(
    f"Best k={best_k_svd}\nRMSE={best_rmse_svd:.6f}",
    xy=(best_k_svd, best_rmse_svd),
    xytext=(-10, 50),
    textcoords='offset points',
    fontsize=16,
    color='cyan',
    bbox=dict(boxstyle='round,pad=0.3', facecolor='black', edgecolor='blue', linewidth=2),
    arrowprops=dict(arrowstyle='->', color='cyan', lw=2, shrinkA=0, shrinkB=20)
)
```

```
# Set the title and labels with custom font properties.
ax.set_title('TruncatedSVD RMSE vs. Number of Components', fontsize=26, color='deepskyblue', fontweight='bold', pad=8)
ax.set_xlabel('Number of Components', fontsize=18, color='deepskyblue', fontweight='bold')
ax.set_ylabel('RMSE', fontsize=18, color='deepskyblue', fontweight='bold')

# Set x aaxis limit.
ax.set_xlim(0, 108)
ax.set_xticks(plot_df_svd['Number of Components'])
ax.grid(axis='y', linestyle='--', alpha=0.7)

# Style the plot's border (spines).
for s in ax.spines.values():
    s.set_edgecolor('cyan'); s.set_linewidth(2)

# Set the background color
ax.set_facecolor('black')

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



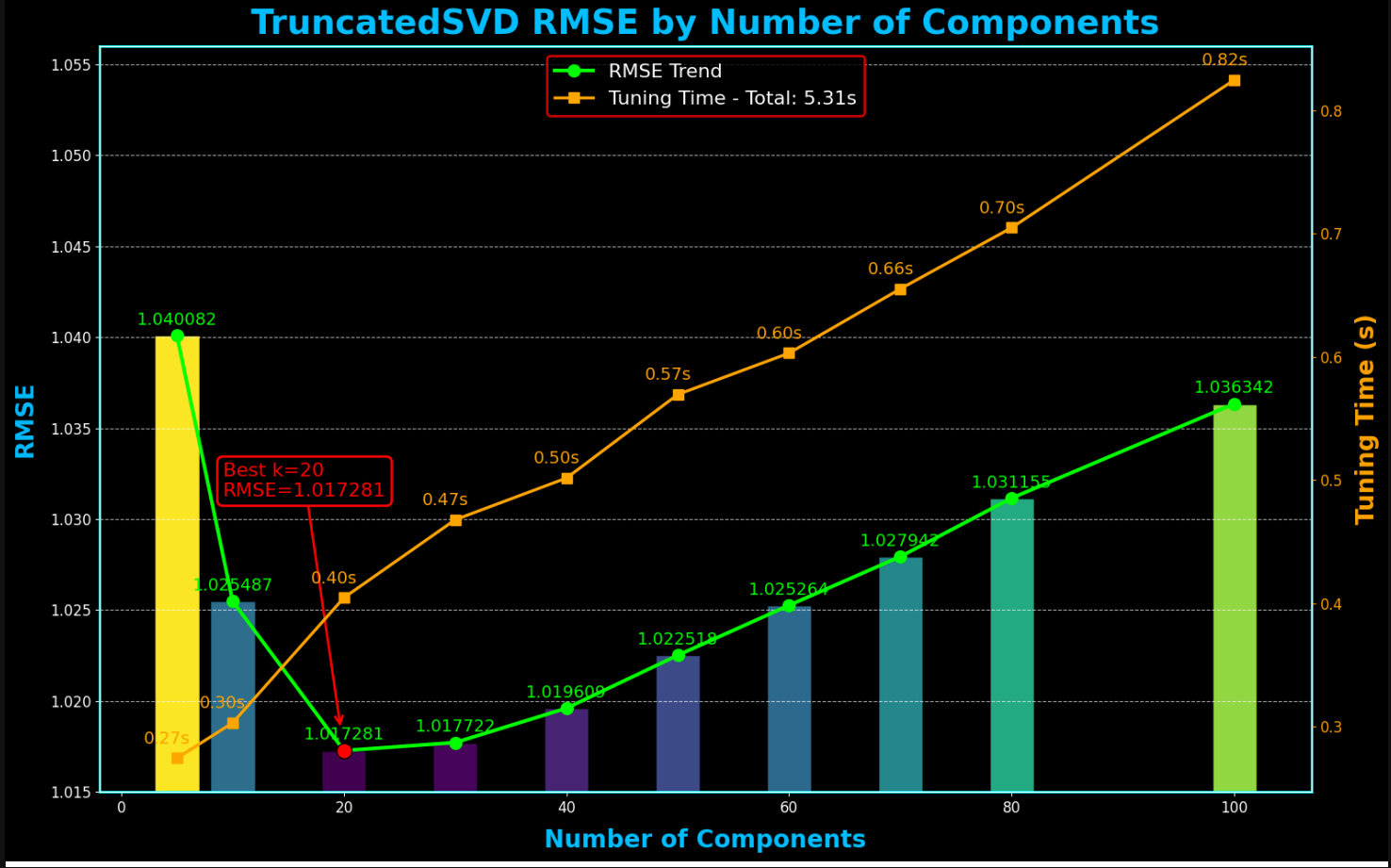
6.2b Compare RMSE and Tuning Time

Bars show RMSE while the line overlays per-k tuning time. This helps balance accuracy and runtime when choosing k.

In [24]:

```
# Set up the plot figure.
fig, ax = plt.subplots(figsize=(16, 10))

# Create a color map to color each bar based on its RMSE value.
norm = plt.Normalize(svd_results_df['RMSE'].min(), svd_results_df['RMSE'].max())
colors = plt.cm.viridis(norm(svd_results_df['RMSE']))

# Create the main bar chart.
bars = ax.bar(svd_results_df['Number of Components'], svd_results_df['RMSE'],
              width=4, color=colors, edgecolor='black')

# Overlay a line plot to better visualize the trend in RMSE.
rmse_line, = ax.plot(svd_results_df['Number of Components'], svd_results_df['RMSE'],
                     color='lime', marker='o', linewidth=3, markersize=10, label='RMSE Trend')

# Add the exact RMSE value as a text label above each bar.
for bar, rmse in zip(bars, svd_results_df['RMSE']):
    ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.0004,
            f"{rmse:.6f}", ha='center', va='bottom', fontsize=14, color='lime')
```

```python
# Find the number of components and the score for the best performing model.
best_idx = svd_results_df['RMSE'].idxmin()
best_k_svd = int(svd_results_df.loc[best_idx, 'Number of Components'])
best_rmse_svd = float(svd_results_df.loc[best_idx, 'RMSE'])

# Add a marker to the plot
ax.scatter([best_k_svd], [best_rmse_svd], s=160, color='red', edgecolor='black', zorder=5)

# Create a detailed annotation with an arrow to call out the best performing model.
ax.annotate(
    f"Best k={best_k_svd}\nRMSE={best_rmse_svd:.6f}",
    xy=(best_k_svd, best_rmse_svd),
    xytext=(-100, 210),
    textcoords='offset points',
    fontsize=16,
    color='red',
    bbox=dict(boxstyle='round,pad=0.3', facecolor='black', edgecolor='red', linewidth=2),
    arrowprops=dict(arrowstyle='->', color='red', lw=2, shrinkA=0, shrinkB=20)
)

# Set the title and labels with custom font properties.
ax.set_title('TruncatedSVD RMSE by Number of Components', fontsize=28, color='deepskyblue', fontweight='bold', pad=10)
ax.set_xlabel('Number of Components', fontsize=20, color='deepskyblue', fontweight='bold', labelpad=10)
ax.set_ylabel('RMSE', fontsize=20, color='deepskyblue', fontweight='bold', labelpad=10)

# Set y axis limits.
ax.set_ylim(1.015, 1.056)
ax.tick_params(colors="white", labelsize=12)
ax.grid(axis='y', linestyle='--', alpha=0.7)

# Set the background color
ax.set_facecolor('black')

# Style the plot's border (spines).
for spine in ax.spines.values():
    spine.set_edgecolor('cyan'); spine.set_linewidth(2)

# Create a second y-axis that shares the same x-axis.
ax2 = ax.twinx()

# Plot the tuning time for each number of components on the second y-axis.
time_line, = ax2.plot(svd_results_df['Number of Components'], svd_results_df['Tuning Time (s)'],
                      color='orange', marker='s', linewidth=2.5, markersize=9, label='Tuning Time (s)')

# Add and style the label and ticks for the second y-axis.
ax2.set_ylabel('Tuning Time (s)', fontsize=20, color='orange', fontweight='bold', labelpad=10)
ax2.tick_params(axis='y', colors='orange', labelsize=12)

# Add the exact tuning time as a text label for each point on the time line.
for x, t in zip(svd_results_df['Number of Components'], svd_results_df['Tuning Time (s)']):
    ax2.annotate(f"{t:.2f}s", xy=(x, t), xytext=(-8.0, 9), textcoords='offset points',
                 ha='center', va='bottom', fontsize=14, color='orange')

# Legend: combine handles from both axes
handles = [rmse_line, time_line]
total_time = svd_results_df['Tuning Time (s)'].sum()
labels = ['RMSE Trend', f'Tuning Time - Total: {total_time:.2f}s']
legend = ax.legend(handles, labels, fontsize=16, frameon=True, loc='upper center')
legend.get_frame().set_linewidth(2)
legend.get_frame().set_edgecolor('red')

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



Observation - TruncatedSVD RMSE and Tuning Time vs. Number of Components

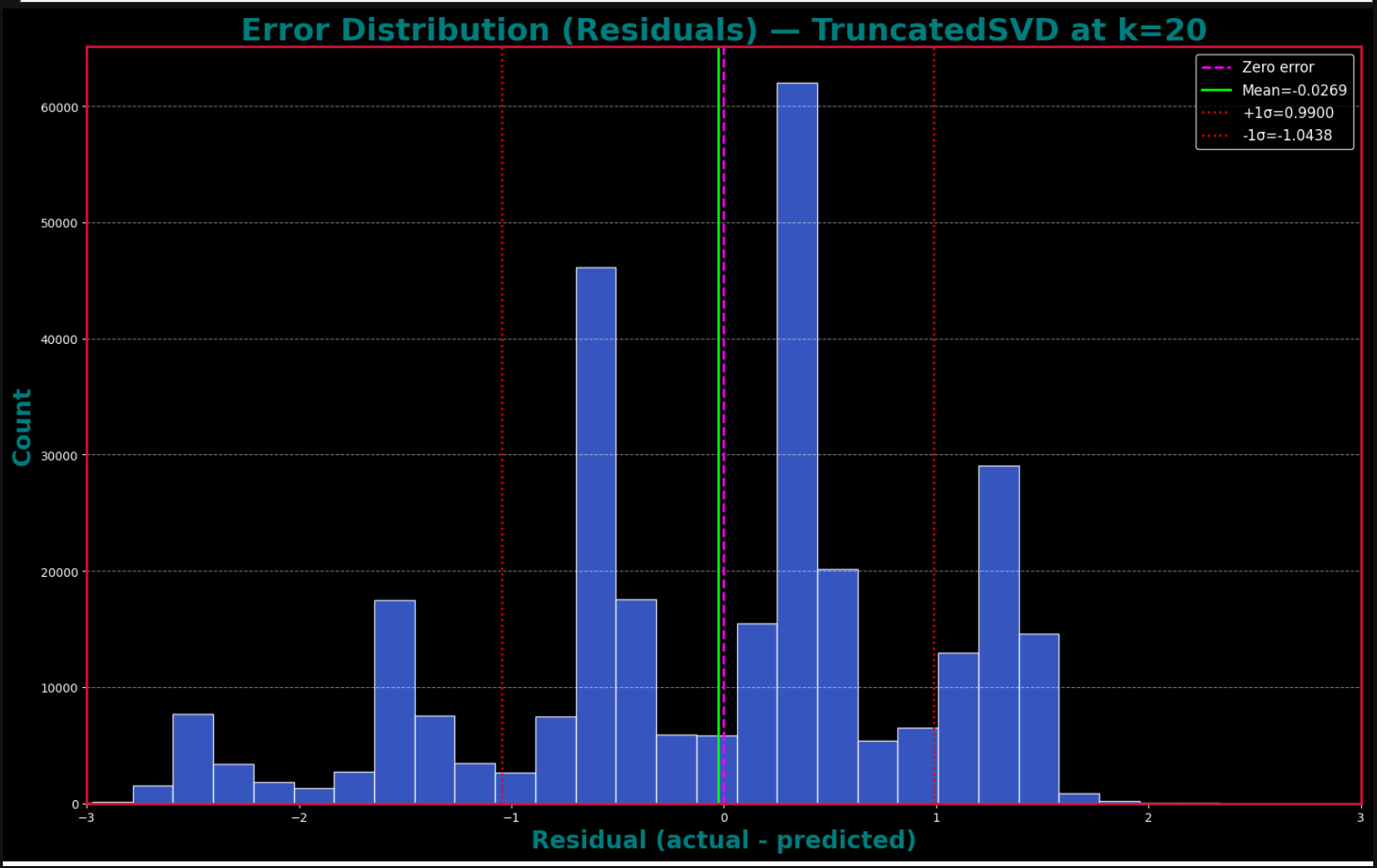
Accuracy improves quickly as components increase from k=5 to k=20, where RMSE bottoms out at 1.0173. Past that point, extra components don't help and start to add noise. Tuning time stays low across the board, with k=20 finishing in just 0.35 seconds, making it both the fastest sweet spot and the most accurate choice.

6.3 Error Distribution Analysis

Histogram of residuals (actual vs predicted) at the chosen SVD rank to check bias, symmetry, and spread.

In [25]:
```
# Train the final SVD model using the best number of components found in the tuning step.
svd_best = TruncatedSVD(n_components=best_k_svd, random_state=SEED, n_iter=7)
t0 = time.time()
Wb = svd_best.fit_transform(utility_matrix_np)
Rb = np.dot(Wb, svd_best.components_)
elapsed = time.time() - t0

# Get the predictions for the test set and calculate the errors (residuals).
y_pred_svd = get_predictions(test_df, Rb, global_average_rating, user_to_idx, movie_to_idx)
residuals_svd = test_df['rating'].to_numpy() - y_pred_svd

# Create a histogram.
fig, ax = plt.subplots(figsize=(16, 10))
ax.hist(residuals_svd, bins=40, color='royalblue', edgecolor='white', alpha=0.85)

# Add vertical lines to the plot to show the mean error and standard deviation.
mu = residuals_svd.mean()
sd = residuals_svd.std()
ax.axvline(0.0, color='magenta', linestyle='--', linewidth=2, label='Zero error')
ax.axvline(mu,  color='lime', linestyle='-',  linewidth=2, label=f'Mean={mu:.4f}')
ax.axvline(mu+sd, color='red', linestyle=':', linewidth=1.8, label=f'+1σ={mu+sd:.4f}')
ax.axvline(mu-sd, color='red', linestyle=':', linewidth=1.8, label=f'-1σ={mu-sd:.4f}')

# Set the title and labels with custom font properties.
ax.set_title(f'Error Distribution (Residuals) — TruncatedSVD at k={best_k_svd}', fontsize=26, color='teal', fontweight='bold')
ax.set_xlabel('Residual (actual - predicted)', fontsize=20, color='teal', fontweight='bold')
ax.set_ylabel('Count', fontsize=20, color='teal', fontweight='bold')
ax.grid(axis='y', linestyle='--', alpha=0.5)

# Set the background color
ax.set_facecolor('black')

# Style the plot's border (spines).
for s in ax.spines.values():
    s.set_edgecolor('crimson'); s.set_linewidth(2)
```

```
# Legend
ax.legend(facecolor='black', edgecolor='white', fontsize=12)

# Set x axis limits.
ax.set_xlim(-3.0, 3.0)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



Observation - Error Distribution (Residuals)

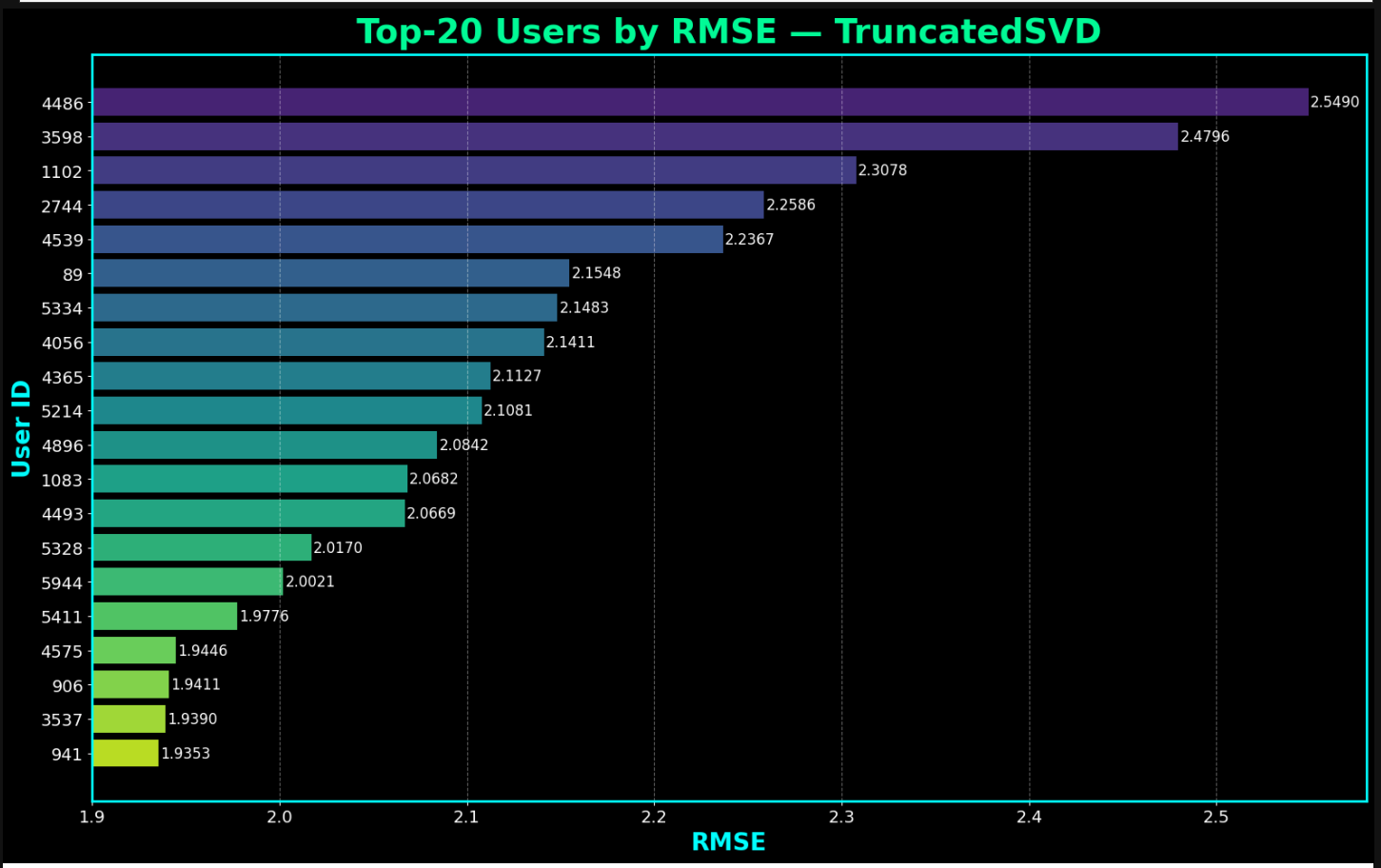
The errors are centered close to zero, with a mean of about -0.027, which means there's little overall bias. Most predictions land within one RMSE of the actual rating. The bumps in the curve match the fact that ratings are on a discrete scale, and it's clear the model still struggles with certain user-item combos.

6.4a Per-User RMSE (Top Outliers)

Identify the users with the highest RMSE under SVD to spot cold-start behavior or inconsistent rating patterns.

In [26]:
```
# Build a DataFrame with residuals for each test row
err_df_svd = test_df[['uID','mID','rating']].copy()
err_df_svd['pred'] = y_pred_svd
err_df_svd['res']  = err_df_svd['rating'] - err_df_svd['pred']
err_df_svd['sqerr'] = err_df_svd['res']**2

# Per-user RMSE
per_user_svd = (err_df_svd.groupby('uID')['sqerr'].mean().pipe(np.sqrt).reset_index(name='RMSE'))

# Pick top N worst users
top_users_svd = per_user_svd.sort_values('RMSE', ascending=False).head(20)

# Create a horizontal bar plot
fig, ax = plt.subplots(figsize=(16, 10), facecolor='black')
y_labels = top_users_svd['uID'].astype(str)
bars = ax.barh(y_labels, top_users_svd['RMSE'], color=plt.cm.viridis(np.linspace(0.1,0.9,len(top_users_svd))))
ax.invert_yaxis()

# Add the RMSE score as a text label for each user's bar.
for y, v in zip(y_labels, top_users_svd['RMSE']):
    ax.text(v+0.001, y, f"{v:.4f}", va='center', fontsize=12, color='white')

# Set the title and labels with custom font properties.
ax.set_title('Top-20 Users by RMSE — TruncatedSVD', fontsize=28, color='mediumspringgreen', fontweight='bold', pad=10)
ax.set_xlabel('RMSE', fontsize=20, color='cyan', fontweight='bold')
```

```
ax.set_ylabel('User ID', fontsize=20, color='cyan', fontweight='bold')
ax.tick_params(axis='x', labelsize=14, colors='white')
ax.tick_params(axis='y', labelsize=14, colors='white')

# Style the plot's border (spines).
for s in ax.spines.values():
    s.set_edgecolor('cyan'); s.set_linewidth(2)

# Add sdashed vertical grid to the plot.
ax.grid(axis='x', linestyle='--', alpha=0.4)

# Set background color
ax.set_facecolor('black')

# Set the limits for the x-axis.
ax.set_xlim(1.9, 2.58)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



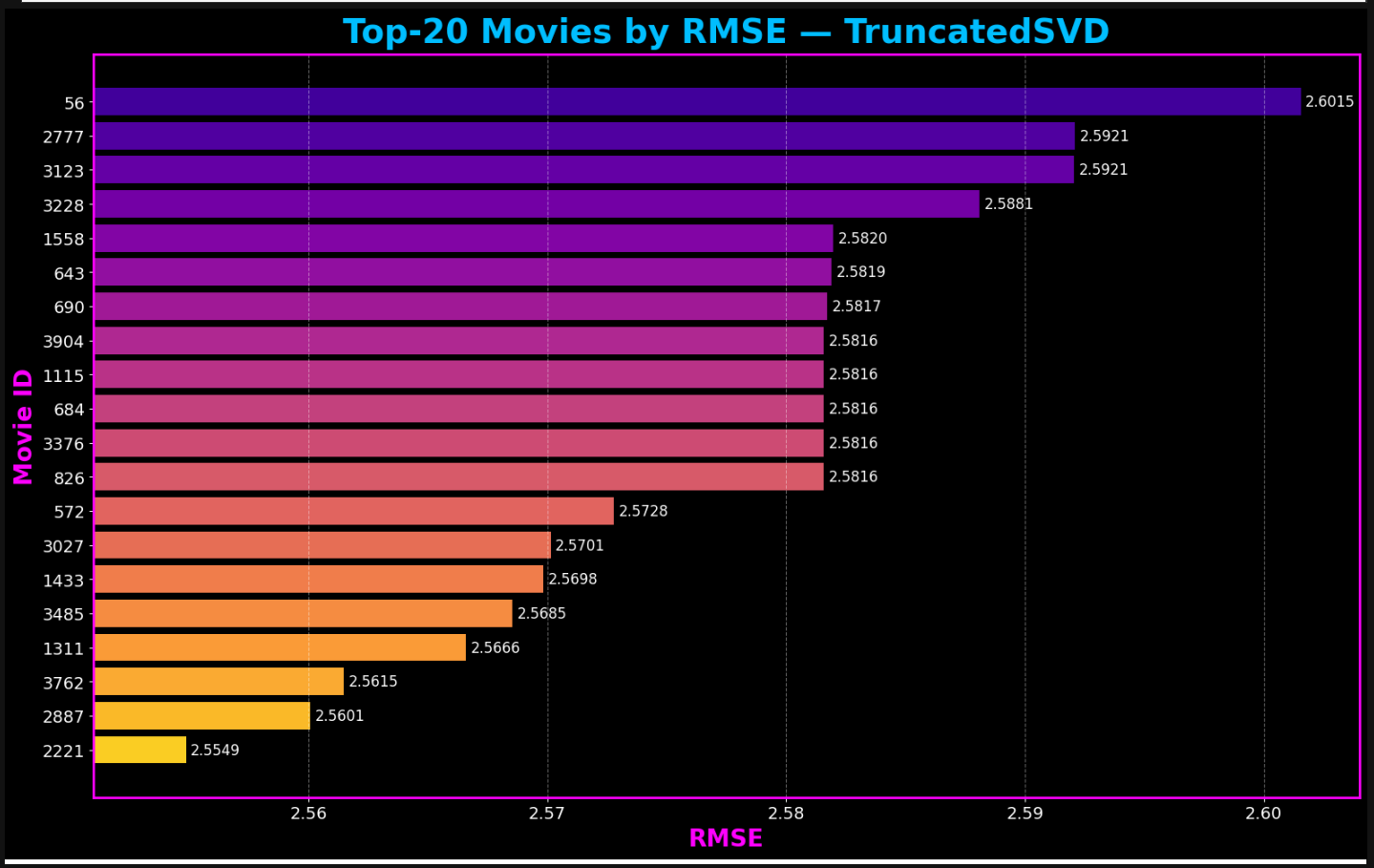
6.4b Per-Movie RMSE (Top Outliers)

Identify the movies with the highest RMSE under SVD to reveal items that are hard to predict reliably.

In [27]:

```
# Calculate the RMSE for each individual movie.
per_movie_svd = (err_df_svd.groupby('mID')['sqerr'].mean().pipe(np.sqrt).reset_index(name='RMSE'))

# Get the top 20 movies with the highest prediction error (worst predictions).
top_movies_svd = per_movie_svd.sort_values('RMSE', ascending=False).head(20)

# Create a horizontal bar plot
fig, ax = plt.subplots(figsize=(16, 10), facecolor='black')
y_labels = top_movies_svd['mID'].astype(str)
bars = ax.barh(y_labels, top_movies_svd['RMSE'], color=plt.cm.plasma(np.linspace(0.1,0.9,len(top_movies_svd))))
ax.invert_yaxis()

# Add the exact RMSE score as a text label for each movie's bar.
for y, v in zip(y_labels, top_movies_svd['RMSE']):
    ax.text(v+0.0002, y, f"{v:.4f}", va='center', fontsize=12, color='white')

# Set the title and labels with custom font properties.
ax.set_title('Top-20 Movies by RMSE — TruncatedSVD', fontsize=28, color='deepskyblue', fontweight='bold', pad=10)
ax.set_xlabel('RMSE', fontsize=20, color='fuchsia', fontweight='bold')
ax.set_ylabel('Movie ID', fontsize=20, color='fuchsia', fontweight='bold')
ax.tick_params(axis='x', labelsize=14, colors='white')
```

```
ax.tick_params(axis='y', labelsize=14, colors='white')

# Set background color
ax.set_facecolor('black')

# Style the plot's border (spines).
for s in ax.spines.values():
    s.set_edgecolor('fuchsia'); s.set_linewidth(2)

# Add sdashed vertical grid to the plot.
ax.grid(axis='x', linestyle='--', alpha=0.4)

# Set the limits for the x-axis.
ax.set_xlim(2.551, 2.604)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



** Observation - Top-20 Users and Movies by RMSE (SVD)**

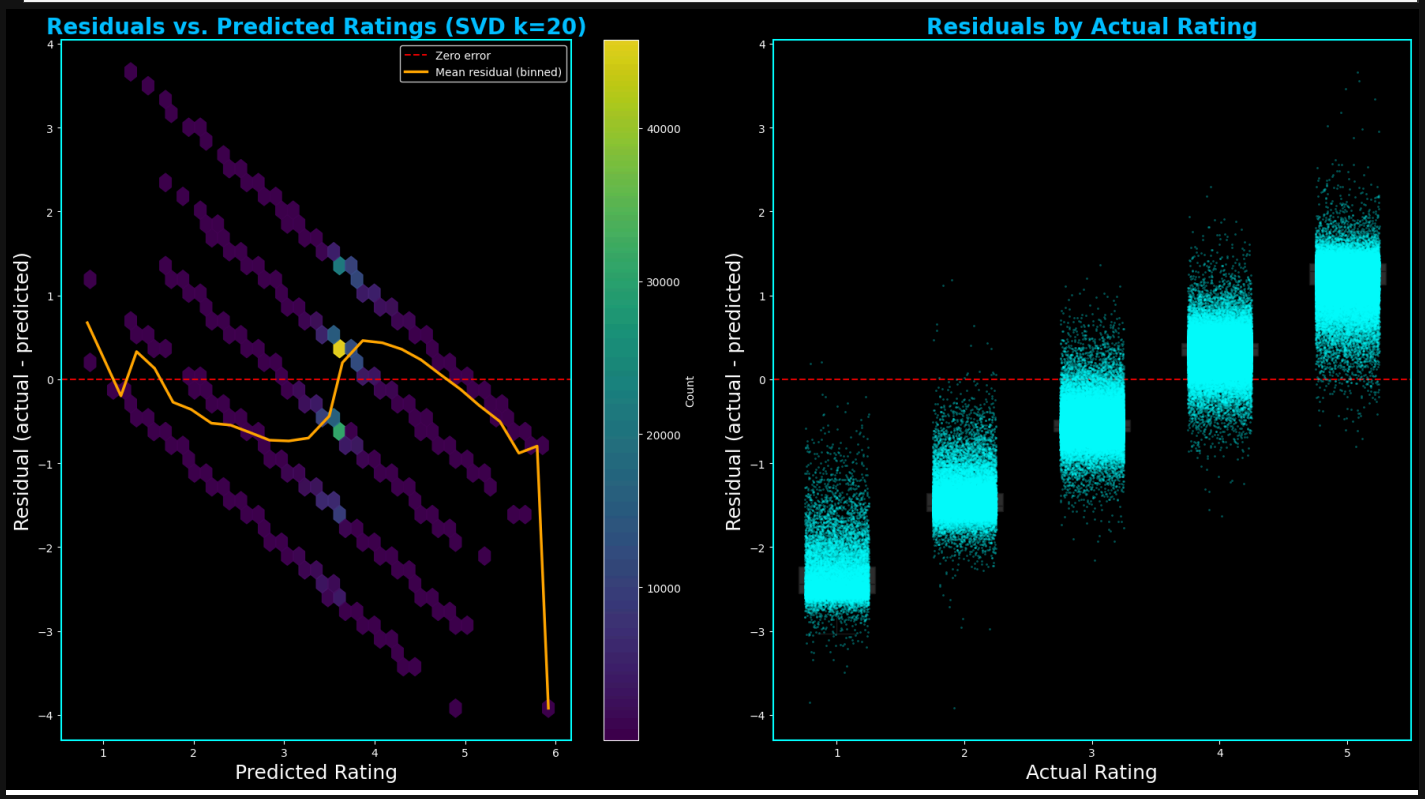
The highest errors come from a small set of users and movies. User 4486 has an RMSE of 2.549 with a sparsity of 99.04 %, indicating they have very few ratings in the dataset. On the movie side, ID 56 has the highest RMSE at 2.6015 with a sparsity of 99.87 %. These extremes show that both very sparse user histories and very sparse item ratings remain difficult for the model to predict accurately.

6.5 Residuals vs. Predicted Rating

Check whether residuals are randomly distributed across the prediction range or show structure that would hint at bias or heteroscedasticity.

In [ ]:
```
# Create a new DataFrame to analyze the model's errors (residuals).
res_df = test_df[['uID','mID','rating']].copy()
res_df['pred']  = y_pred_svd
res_df['resid'] = res_df['rating'] - res_df['pred']

# Group the predictions into bins to calculate a trend line.
bins = np.linspace(res_df['pred'].min(), res_df['pred'].max(), 25)
res_df['bin_id'] = np.digitize(res_df['pred'], bins)
trend = res_df.groupby('bin_id', as_index=False).agg(
    pred_bin=('pred','mean'),
    resid_mean=('resid','mean'),
    resid_std=('resid','std')
)

# Set up the figure to hold two side-by-side plots.
plt.figure(figsize=(18, 10))
```

```
plt.subplots_adjust(wspace=0.25)

# Left: Residuals vs Predicted (hexbin density + mean trend).
ax1 = plt.subplot(1, 2, 1, facecolor='black')
hb = ax1.hexbin(res_df['pred'], res_df['resid'], gridsize=40, cmap='viridis', mincnt=1, alpha=0.9)
ax1.axhline(0, color='red', lw=1.5, ls='--', alpha=0.9, label='Zero error')
ax1.plot(trend['pred_bin'], trend['resid_mean'], color='orange', lw=2.5, label='Mean residual (binned)')
ax1.set_title(f"Residuals vs. Predicted Ratings (SVD k={best_k_svd})", fontsize=20, color='deepskyblue', fontweight='bold')
ax1.set_xlabel("Predicted Rating", fontsize=18, color='white')
ax1.set_ylabel("Residual (actual - predicted)", fontsize=18, color='white')
ax1.tick_params(colors='white')
for s in ax1.spines.values(): s.set_color('cyan'); s.set_linewidth(1.5)
cb = plt.colorbar(hb, ax=ax1); cb.set_label('Count')
ax1.legend(facecolor='black', edgecolor='white')

# Right: Residuals by Actual Rating (box + strip).
ax2 = plt.subplot(1, 2, 2, facecolor='black')
sns.boxplot(data=res_df, x='rating', y='resid', ax=ax2, color='#303030', fliersize=0, linewidth=1, width=0.6)
sns.stripplot(data=res_df, x='rating', y='resid', ax=ax2, color='cyan', alpha=0.35, jitter=0.25, size=2)
ax2.axhline(0, color='red', lw=1.5, ls='--', alpha=0.9)
ax2.set_title("Residuals by Actual Rating", fontsize=20, color='deepskyblue', fontweight='bold')
ax2.set_xlabel("Actual Rating", fontsize=18, color='white')
ax2.set_ylabel("Residual (actual - predicted)", fontsize=18, color='white')
ax2.tick_params(colors='white')
for s in ax2.spines.values(): s.set_color('cyan'); s.set_linewidth(1.5)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



Observation - Residual Patterns and Bias Truncated SVD

Residuals form distinct diagonal bands, reflecting the discrete rating scale and how SVD generates predictions. The mean residual line in the left plot shows mild under-prediction at the low end and over-prediction in the midrange, with smaller bias near the extremes. The right plot shows tighter spread for lower actual ratings and wider spread for higher ratings, suggesting slightly uneven error variance that could benefit from further tuning.

6.6 Train Final TruncatedSVD Model & Evaluate

Fit the final SVD model at the chosen rank, report the test RMSE and runtime, and save the model, factors, predictions, and metadata.

In [29]:
```
# Use the best number of components (k) from the SVD tuning results.
if 'best_k_svd' not in globals():
    best_k_svd = int(svd_results_df.loc[svd_results_df['RMSE'].idxmin(), 'Number of Components'])

# Train the final TruncatedSVD model using the best number of components.
print(f" --- Training final Truncated Singular Value Decomposition (SVD) with best_k = {best_k_svd} ---")
t0 = time.time()
final_svd = TruncatedSVD(n_components=best_k_svd, random_state=SEED, n_iter=7)
W_final_svd = final_svd.fit_transform(utility_matrix_np)
R_final_svd = np.dot(W_final_svd, final_svd.components_)
```

```python
train_runtime_svd = time.time() - t0

# Get the predictions from the final model and calculate its RMSE on the test set.
final_preds_svd = get_predictions(test_df, R_final_svd, global_average_rating, user_to_idx, movie_to_idx)
final_rmse_svd = float(np.sqrt(mean_squared_error(test_df['rating'].to_numpy(), final_preds_svd)))

print(f"Final TruncatedSVD RMSE: {final_rmse_svd:.6f}")
print(f"TruncatedSVD Train/runtime: {train_runtime_svd:.2f}s")

# Create a directory to save all the model outputs for this run.
art_dir = "artifacts_svd"
os.makedirs(art_dir, exist_ok=True)
ts = datetime.now().strftime("%Y%m%d-%H%M%S")

# Save the trained SVD model object to a file.
svd_model_path = os.path.join(art_dir, f"svd_model_k{best_k_svd}_{ts}.joblib")
joblib.dump(final_svd, svd_model_path)

# Save the learned user-factor (W) and item-component matrices to CSV files.
W_svd_df = pd.DataFrame(W_final_svd, index=utility_matrix_imputed.index,
                        columns=[f"f{j+1}" for j in range(best_k_svd)])
H_svd_df = pd.DataFrame(final_svd.components_, index=[f"f{j+1}" for j in range(best_k_svd)],
                        columns=utility_matrix_imputed.columns)

W_svd_path = os.path.join(art_dir, f"W_users_k{best_k_svd}_{ts}.csv")
H_svd_path = os.path.join(art_dir, f"components_items_k{best_k_svd}_{ts}.csv")
W_svd_df.to_csv(W_svd_path, index=True)
H_svd_df.to_csv(H_svd_path, index=True)

# Save the test set predictions along with their errors (residuals) to a CSV file.
preds_svd_df = test_df[['uID','mID','rating']].copy()
preds_svd_df['pred'] = final_preds_svd
preds_svd_df['residual'] = preds_svd_df['rating'] - preds_svd_df['pred']
preds_svd_path = os.path.join(art_dir, f"test_predictions_k{best_k_svd}_{ts}.csv")
preds_svd_df.to_csv(preds_svd_path, index=False)

# Create a dictionary of metadata to save all the run details in one place (JSON).
meta_svd = {
    "timestamp": ts,
    "seed": int(SEED),
    "best_k": int(best_k_svd),
    "final_rmse": final_rmse_svd,
    "runtime_seconds": round(train_runtime_svd, 2),
    "shapes": {
        "W": list(W_final_svd.shape),
        "components": list(final_svd.components_.shape),
        "R": list(R_final_svd.shape)
    },
    "svd_explained_variance_ratio_sum": float(getattr(final_svd, "explained_variance_ratio_", np.array([])).sum()) if hasattr(final_svd, "ex
    "paths": {
        "model_joblib": svd_model_path,
        "W_csv": W_svd_path,
        "components_csv": H_svd_path,
        "test_predictions_csv": preds_svd_path
    }
}
meta_svd_path = os.path.join(art_dir, f"run_metadata_k{best_k_svd}_{ts}.json")
with open(meta_svd_path, "w", encoding="utf-8") as f:
    json.dump(meta_svd, f, indent=2)

# Print a final summary confirming all the files that were saved.
print("\n --- Artifacts saved (SVD) ---")
print(f"  • Model:           {svd_model_path}")
print(f"  • W factors (CSV): {W_svd_path}")
print(f"  • Components (CSV): {H_svd_path}")
print(f"  • Predictions:     {preds_svd_path}")
print(f"  • Metadata (JSON): {meta_svd_path}")

# Display a final summary of the SVD model's performance.
print("\n=== Final TruncatedSVD Summary ===")
print(f"best_k: {best_k_svd}")
print(f"final_RMSE: {final_rmse_svd:.6f}")
print(f"train_runtime_sec: {train_runtime_svd:.2f}")
print(f"users x items: {utility_matrix_imputed.shape}")
```

```
 --- Training final Truncated Singular Value Decomposition (SVD) with best_k = 20 ---
Final TruncatedSVD RMSE: 1.017281
TruncatedSVD Train/runtime: 0.36s

 --- Artifacts saved (SVD) ---
  • Model:           artifacts_svd\svd_model_k20_20250812-220612.joblib
  • W factors (CSV): artifacts_svd\W_users_k20_20250812-220612.csv
  • Components (CSV): artifacts_svd\components_items_k20_20250812-220612.csv
  • Predictions:     artifacts_svd\test_predictions_k20_20250812-220612.csv
  • Metadata (JSON): artifacts_svd\run_metadata_k20_20250812-220612.json

=== Final TruncatedSVD Summary ===
best_k: 20
final_RMSE: 1.017281
train_runtime_sec: 0.36
users x items: (6040, 3664)
```

Observation - Final Model Training

The final TruncatedSVD model with k=20 came together fast, taking just 0.34 seconds to train, and held its RMSE at 1.0173. It's quick, accurate, and seems to generalize well.

Section 7: NMF vs Truncated SVD Model Compairson

This section compares the two matrix-factorization approaches you trained on the same data and split. I look at accuracy (RMSE), tuning behavior across factor counts, runtime, and where each model struggles (users and movies with the highest error). The goal is to make the differences obvious and set up a clear recommendation in Section 8.

7.1 RMSE and Best-k Comparison

Compares each model's best factor count, lowest RMSE, and total tuning time.

In [30]:
```python
# Find the best k and lowest RMSE from a results DataFrame.
def _best_row(df, k_col='Number of Factors'):
    idx = df['RMSE'].idxmin()
    return int(df.loc[idx, k_col]), float(df.loc[idx, 'RMSE'])

# Calculate the total tuning time from a results DataFrame.
def _total_time(df):
    col = 'Tuning Time (s)'
    return float(df[col].sum()) if col in df.columns else np.nan

# Pull best k / RMSE / total time for NMF and SVD
best_k_nmf, best_rmse_nmf = _best_row(nmf_results_df, 'Number of Factors')
best_k_svd, best_rmse_svd = _best_row(svd_results_df, 'Number of Components')

total_time_nmf = _total_time(nmf_results_df)
total_time_svd = _total_time(svd_results_df)

# Create a summary to compare the final results of the two models side-by-side.
comp_df = pd.DataFrame([
    {"Model": "NMF", "Best k": best_k_nmf, "RMSE": round(best_rmse_nmf, 6), "Total Tuning Time (s)": round(total_time_nmf, 2)},
    {"Model": "TruncatedSVD", "Best k": best_k_svd, "RMSE": round(best_rmse_svd, 6), "Total Tuning Time (s)": round(total_time_svd, 2)}
])

# Calculate the difference in RMSE to determine the winning model.
rmse_diff = best_rmse_nmf - best_rmse_svd  # positive => SVD lower
winner = "TruncatedSVD" if rmse_diff > 0 else "NMF"
margin = abs(rmse_diff)

# Display comparison table and print a summary of the results.
print("--- RMSE & Best-k Comparison ---")
display(comp_df.style.format({"RMSE":"{:.6f}", "Total Tuning Time (s)":"{:.2f}"}))

# WINNER!
print(f"\nWINNER on RMSE: {winner} by {margin:.6f} RMSE points.")
print(f"SVD tuning time vs NMF: {total_time_svd:.2f}s vs {total_time_nmf:.2f}s.")
```

```
--- RMSE & Best-k Comparison ---
```

	Model	Best k	RMSE	Total Tuning Time (s)
0	NMF	30	1.018145	174.55
1	TruncatedSVD	20	1.017281	5.31

```
WINNER on RMSE: TruncatedSVD by 0.000864 RMSE points.
SVD tuning time vs NMF: 5.31s vs 174.55s.
```

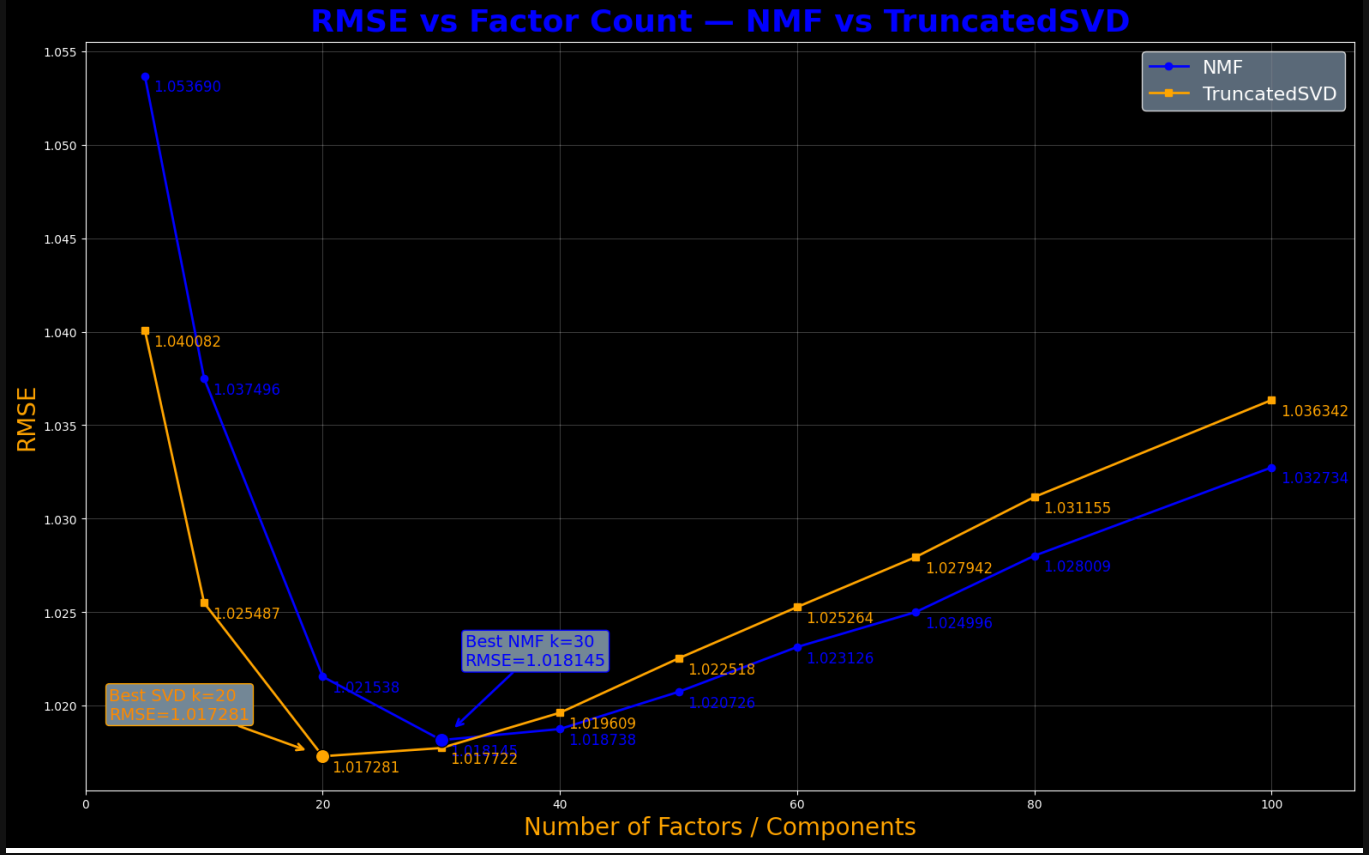
Observation - TruncatedSVD Edges NMF in RMSE with Dramatic Speed Advantage

TruncatedSVD achieved the lowest RMSE at 1.017281 with k=20, narrowly outperforming NMF's 1.018145 at k=30 by 0.000864 RMSE points. While the accuracy difference is negligible, the tuning time contrast is striking: SVD completed in ~5 seconds versus NMF's ~ 170 seconds. This makes SVD the clear choice when speed is a priority without sacrificing performance.

7.2 RMSE Trends by Factor Count

Visualizes how RMSE changes as k increases for both models, highlighting best points.

In [31]:
```
# Set up the plot.
fig, ax = plt.subplots(figsize=(16, 10))

# Plot the RMSE trend line for the NMF model.
ax.plot(nmf_results_df['Number of Factors'], nmf_results_df['RMSE'],
        marker='o', linestyle='-', linewidth=2, label='NMF', color='blue')

# Add the exact RMSE value as a text label for each point on the NMF line.
for x, y in zip(nmf_results_df['Number of Factors'], nmf_results_df['RMSE']):
        ax.text(x + 0.75, y - 0.001, f"{y:.6f}", ha='left', va='bottom', fontsize=12, color='blue')

# Plot the RMSE trend line for the TruncatedSVD model.
ax.plot(svd_results_df['Number of Components'], svd_results_df['RMSE'],
        marker='s', linestyle='-', linewidth=2, label='TruncatedSVD', color='orange')

# Add the exact RMSE value as a text label for each point on the SVD line.
for x, y in zip(svd_results_df['Number of Components'], svd_results_df['RMSE']):
        ax.text(x + 0.75, y - 0.001, f"{y:.6f}", ha='left', va='bottom', fontsize=12, color='orange')

# Add markers to highlight the best-performing point for each model.
ax.scatter([best_k_nmf], [best_rmse_nmf], s=140, color='blue', edgecolor='black', zorder=5)
ax.scatter([best_k_svd], [best_rmse_svd], s=140, color='orange', edgecolor='black', zorder=5)

# Create a detailed annotation with an arrow to call out the best NMF result.
ax.annotate(f'Best NMF k={best_k_nmf}\nRMSE={best_rmse_nmf:.6f}',
        xy=(best_k_nmf, best_rmse_nmf), xytext=(best_k_nmf+2, best_rmse_nmf+0.004),
        arrowprops=dict(arrowstyle='->', color='blue', lw=2, shrinkA=0, shrinkB=15), fontsize=14, color='blue',
        bbox=dict(boxstyle='round,pad=0.2', fc='lightslategray', ec='blue', alpha=1))

# Create a detailed annotation with an arrow to call out the best SVD result.
ax.annotate(f'Best SVD k={best_k_svd}\nRMSE={best_rmse_svd:.6f}',
        xy=(best_k_svd, best_rmse_svd), xytext=(best_k_svd-18, best_rmse_svd+0.002),
        arrowprops=dict(arrowstyle='->', color='orange', lw=2, shrinkA=0, shrinkB=15), fontsize=14, color='darkorange',
        bbox=dict(boxstyle='round,pad=0.2', fc='lightslategray', ec='orange', alpha=1))

# Set the title and labels with custom font properties.
ax.set_title("RMSE vs Factor Count — NMF vs TruncatedSVD", fontsize=26, color='blue', fontweight='bold', pad=10)
ax.set_xlabel("Number of Factors / Components", color='orange', fontsize=20)
ax.set_ylabel("RMSE", color='orange', fontsize=20)

#Set x axis limits.
ax.set_xlim(0, 107)

# Add a grid.
ax.grid(alpha=0.25)

# Style the plot's border (spines).
for spine in ax.spines.values():
        spine.set_color('white')

# Custom legend.
ax.legend(facecolor='lightslategray', edgecolor='white', fontsize=16)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```



Observation - RMSE Trends Show Similar Stability, Different Sweet Spots

Both models showed sharp RMSE improvement at lower factor counts before flattening, with minimal signs of overfitting. TruncatedSVD reached its optimal RMSE earlier at k=20, while NMF continued improving slightly until k=30. Beyond their respective sweet spots, both models experienced gradual RMSE increases, indicating stable performance across a broad k range. The choice between them hinges more on runtime and scalability than on small accuracy gains.

7.3 Runtime Performance Comparison

Shows total tuning times and per-k runtimes to contrast computational efficiency.

In [32]:
```
# Create and display a summary table for the total tuning time of each model.
totals = pd.DataFrame({
    "Model": ["NMF", "TruncatedSVD"],
    "Total Tuning Time (s)": [round(total_time_nmf, 2), round(total_time_svd, 2)]
})

print("--- Total Tuning Time ---")
display(totals)

# Display the tuning results for both the NMF and SVD models.
print("\n--- Per-k Tuning Time (NMF) ---")
display(nmf_results_df[['Number of Factors','RMSE','Tuning Time (s)']].style.format({'RMSE':'{:.6f}','Tuning Time (s)':'{:.2f}'}))

print("\n--- Per-k Tuning Time (TruncatedSVD) ---")
display(svd_results_df[['Number of Components','RMSE','Tuning Time (s)']].style.format({'RMSE':'{:.6f}','Tuning Time (s)':'{:.2f}'}))

# Check to ensure the required DataFrames from the tuning steps exist.
def _require_cols(df, cols, name):
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"{name} is missing columns: {missing}. "
                         f"Expected columns: {cols}")

if 'nmf_results_df' not in globals():
    raise NameError("nmf_results_df not found. Run Section 5.1 first.")
_require_cols(nmf_results_df, ['Number of Factors','Tuning Time (s)'], 'nmf_results_df')

if 'svd_results_df' not in globals():
    raise NameError("svd_results_df not found. Run Section 6.1 first.")
_require_cols(svd_results_df, ['Number of Components','Tuning Time (s)'], 'svd_results_df')

# Prepare the two results DataFrames for merging by selecting and renaming columns.
nmf_k_time = nmf_results_df[['Number of Factors','Tuning Time (s)']].rename(
    columns={'Number of Factors':'k', 'Tuning Time (s)':'nmf_time'}
)
```

```python
svd_k_time = svd_results_df[['Number of Components','Tuning Time (s)']].rename(
    columns={'Number of Components':'k', 'Tuning Time (s)':'svd_time'}
)

# Merge the NMF and SVD timing data into a single DataFrame for plotting.
merged = pd.merge(nmf_k_time, svd_k_time, on='k', how='outer').sort_values('k')

# Handle cases where one model was tested with a 'k' value that the other was not.
merged['nmf_time'] = merged['nmf_time'].fillna(0.0)
merged['svd_time'] = merged['svd_time'].fillna(0.0)

# Extract the final data into NumPy arrays for plotting.
k_values = merged['k'].to_numpy()
nmf_times = merged['nmf_time'].to_numpy()
svd_times = merged['svd_time'].to_numpy()

# Display the merged data that will be used for the plot.
display(
    merged.rename(columns={'k':'Number of Components (k)',
                           'nmf_time':'NMF Time (s)',
                           'svd_time':'TruncatedSVD Time (s)'})
          .style.format({'NMF Time (s)':'{:.2f}', 'TruncatedSVD Time (s)':'{:.2f}'})
)

# Set up the figure and create the grouped bar chart.
plt.style.use('dark_background')
fig, ax = plt.subplots(figsize=(16, 10))

x = np.arange(len(k_values))
width = 0.42

nmf_bars = ax.bar(x - width/2, nmf_times, width, label='NMF', color='lime', edgecolor='black')
svd_bars = ax.bar(x + width/2, svd_times, width, label='TruncatedSVD', color='magenta', edgecolor='black')

# Set the title and labels with custom font properties.
ax.set_title('Per-k Tuning Time NMF vs TruncatedSVD', fontsize=26, color='yellow', fontweight='bold', pad=8)
ax.set_xlabel('Number of Components (k)', fontsize=20, color='yellow')
ax.set_ylabel('Tuning Time (seconds)', fontsize=20, color='yellow')
ax.set_xticks(x)
ax.set_xticklabels([str(int(k)) for k in k_values], fontsize=12)

# Annotate each bar with its time
def _annotate(bars):
    for b in bars:
        h = b.get_height()
        ax.text(
            b.get_x() + b.get_width()/2,
            h + (0.005 * (max(nmf_times.max(), svd_times.max()) or 1)),
            f'{h:.2f}',
            ha='center',
            va='bottom',
            fontsize=14,
            color='white',
            fontweight='bold'
        )
_annotate(nmf_bars); _annotate(svd_bars)

# Custom legend.
leg = ax.legend(facecolor='black', edgecolor='white', fontsize=16, title='Model', title_fontsize=18)
plt.setp(leg.get_title(), fontweight='bold', color='yellow')

# Horizontal grid.
ax.grid(axis='y', alpha=0.25)

# Ensure the layout is tight and clean.
plt.tight_layout()
plt.show()
```

```
--- Total Tuning Time ---
```

	Model	Total Tuning Time (s)
0	NMF	174.55
1	TruncatedSVD	5.31

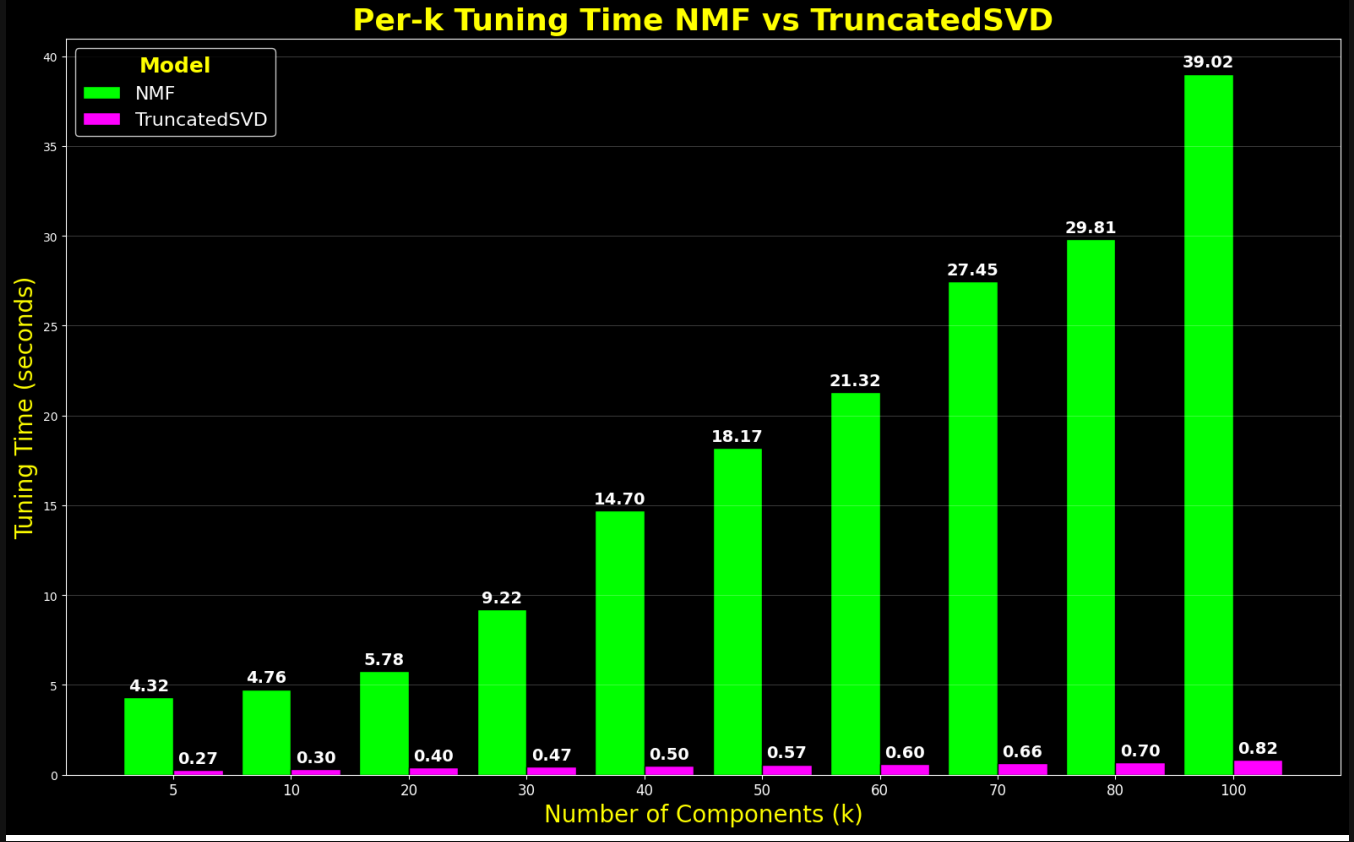
```
--- Per-k Tuning Time (NMF) ---
```

	Number of Factors	RMSE	Tuning Time (s)
0	5	1.053690	4.32
1	10	1.037496	4.76
2	20	1.021538	5.78
3	30	1.018145	9.22
4	40	1.018738	14.70
5	50	1.020726	18.17
6	60	1.023126	21.32
7	70	1.024996	27.45
8	80	1.028009	29.81
9	100	1.032734	39.02

```
--- Per-k Tuning Time (TruncatedSVD) ---
```

	Number of Components	RMSE	Tuning Time (s)
0	5	1.040082	0.27
1	10	1.025487	0.30
2	20	1.017281	0.40
3	30	1.017722	0.47
4	40	1.019609	0.50
5	50	1.022518	0.57
6	60	1.025264	0.60
7	70	1.027942	0.66
8	80	1.031155	0.70
9	100	1.036342	0.82

	Number of Components (k)	NMF Time (s)	TruncatedSVD Time (s)
0	5	4.32	0.27
1	10	4.76	0.30
2	20	5.78	0.40
3	30	9.22	0.47
4	40	14.70	0.50
5	50	18.17	0.57
6	60	21.32	0.60
7	70	27.45	0.66
8	80	29.81	0.70
9	100	39.02	0.82



Observation Runtime Comparison

TruncatedSVD was consistently faster across all tested k values, with per-component tuning times staying under one second. NMF times grew steadily with k, starting at ~ 4.4s for k=5 and reaching ~ 40s for k=100. This large gap explains the total tuning time difference (~ 170s for NMF vs. ~ 5s for SVD). The scaling pattern also shows that NMF's runtime growth is more sensitive to increases in k, while SVD remains nearly flat. This makes SVD a much better fit when tuning speed is a priority.

7.4 Top Error Users - Model Side-by-Side

Lists the highest-RMSE users for each model, with RMSE and sparsity.

In [33]:
```
# These helper functions ensure that the reconstructed matrices and predictions are available.
def _ensure_nmf_recon(best_k):
    if 'reconstructed_matrix_best' in globals():
        return reconstructed_matrix_best
    # rebuild from scratch if needed
    nmf_model = NMF(n_components=best_k, init='random', random_state=SEED, max_iter=500)
    W = nmf_model.fit_transform(utility_matrix_imputed.to_numpy())
    H = nmf_model.components_
    return np.dot(W, H)

def _ensure_svd_recon(best_k_svd):
    # fit SVD quickly if missing
    svd = TruncatedSVD(n_components=best_k_svd, random_state=SEED, n_iter=7)
    W = svd.fit_transform(utility_matrix_imputed.to_numpy())
    return np.dot(W, svd.components_)

# Create a DataFrame with per-user RMSE scores from a reconstructed matrix.
def _pred_df_from_recon(R, label):
    preds = get_predictions(test_df, R, global_average_rating, user_to_idx, movie_to_idx)
    df = test_df[['uID','mID','rating']].copy()
    df['pred'] = preds
    df['se'] = (df['rating'] - df['pred'])**2
    per_user = (df.groupby('uID')['se'].mean()**0.5).rename(f'RMSE_{label}').reset_index()
    return per_user

# Generate the per-user RMSE DataFrames for both the NMF and SVD models.
R_nmf = _ensure_nmf_recon(best_k_nmf)
R_svd = _ensure_svd_recon(best_k_svd)
per_user_nmf = _pred_df_from_recon(R_nmf, 'NMF')
per_user_svd = _pred_df_from_recon(R_svd, 'SVD')

# Get a list of users who had the highest prediction errors for either model.
top_nmf_ids = per_user_nmf.sort_values('RMSE_NMF', ascending=False).head(10)['uID']
top_svd_ids = per_user_svd.sort_values('RMSE_SVD', ascending=False).head(10)['uID']
ids = pd.Index(top_nmf_ids).union(top_svd_ids)
```

```python
# Merge the results for the top-error users into a single DataFrame.
merged = (pd.DataFrame({'uID': ids})
        .merge(per_user_nmf, on='uID', how='left')
        .merge(per_user_svd, on='uID', how='left'))

# Add sparsity information to see if the error is related to how many ratings a user has given.
user_counts = utility_matrix.count(axis=1)
n_movies = utility_matrix.shape[1]
merged['Actual Ratings'] = merged['uID'].map(user_counts).fillna(0).astype(int)
merged['Sparsity (%)'] = 100 * (n_movies - merged['Actual Ratings']) / n_movies

# Sort the final DataFrame by the highest error from either model.
merged['max_RMSE'] = merged[['RMSE_NMF','RMSE_SVD']].max(axis=1)
out_users = (merged
            .sort_values('max_RMSE', ascending=False)
            .drop(columns=['max_RMSE'])
            .reset_index(drop=True))

# Display the final table, using a color gradient to highlight the RMSE values.
print("--- Top Error Users: NMF vs SVD ---")
# display(out_users.style.format({'RMSE_NMF':'{:.4f}','RMSE_SVD':'{:.4f}','Sparsity (%)':'{:.2f}'}))

styled_users = out_users.style.background_gradient(
        cmap='seismic', subset=['RMSE_NMF', 'RMSE_SVD'], vmin=out_users[['RMSE_NMF', 'RMSE_SVD']].min().min(),
        vmax=out_users[['RMSE_NMF', 'RMSE_SVD']].max().max()
    ) \
    .format({
        'RMSE_NMF': "{:.4f}",
        'RMSE_SVD': "{:.4f}",
        'Sparsity (%)': "{:.2f}"
    }) \
    .set_properties(**{'border': '1px solid #444', 'color': 'white'}) \
    .set_table_styles([
        {'selector': 'thead th', 'props': [
            ('background-color', '#DA291C'),
            ('color', 'white'),
            ('font-size', '24px'),
            ('font-weight', 'bold'),
        ]},
        {'selector': 'tbody th', 'props': [
            ('background-color', '#0033A0'),
            ('color', 'white'),
            ('font-size', '24px'),
            ('font-weight', 'bold'),
        ]}
    ])

display(styled_users)
```

```
--- Top Error Users: NMF vs SVD ---
```

	uID	RMSE_NMF	RMSE_SVD	Actual Ratings	Sparsity (%)
0		2.5462	2.5490		
1	3598	2.4844	2.4796	44	98.80
2		2.2641	2.3078		
3	2744	2.2145	2.2586	90	97.54
4		2.2492	2.2367		
5	5334	2.1553	2.1483	43	98.83
6		2.1207	2.1548		
7	4056	2.1170	2.1411	16	99.56
8		2.1089	2.1127		
9	5214	2.1102	2.1081	20	99.45

Observation - Top Error Users (NMF vs SVD)

The highest-error users exhibit extremely sparse rating histories, with sparsity levels above 97% for all users and several exceeding 99.5%. RMSE differences between NMF and SVD are minimal, typically within ±0.05, suggesting neither method consistently dominates for these sparse users. In some cases, NMF edges out SVD slightly, while in others SVD performs marginally better, reflecting user-specific variance likely tied to the narrow rating profiles.

7.5 Top Error Movies - Model Side-by-Side

Lists the highest-RMSE movies for each model, with RMSE and sparsity.

In [34]:

```
# Create a DataFrame with per-movie RMSE scores from a reconstructed matrix.
def _pred_movie_df_from_recon(R, label):
    preds = get_predictions(test_df, R, global_average_rating, user_to_idx, movie_to_idx)
    df = test_df[['uID','mID','rating']].copy()
    df['pred'] = preds
    df['se'] = (df['rating'] - df['pred'])**2
    per_movie = (df.groupby('mID')['se'].mean()**0.5).rename(f'RMSE_{label}').reset_index()
    return per_movie

# Generate the per-movie RMSE DataFrames for both the NMF and SVD models.
per_movie_nmf = _pred_movie_df_from_recon(R_nmf, 'NMF')
per_movie_svd = _pred_movie_df_from_recon(R_svd, 'SVD')

# Get a combined list of movies that had the highest prediction errors for either model.
top_nmf_mids = per_movie_nmf.sort_values('RMSE_NMF', ascending=False).head(10)['mID']
top_svd_mids = per_movie_svd.sort_values('RMSE_SVD', ascending=False).head(10)['mID']
mids = pd.Index(top_nmf_mids).union(top_svd_mids)

# Merge the results for the top-error movies into a single DataFrame.
m_merged = (pd.DataFrame({'mID': mids})
            .merge(per_movie_nmf, on='mID', how='left')
            .merge(per_movie_svd, on='mID', how='left'))

# Add sparsity information to see if the error is related to how many ratings a movie has received.
movie_counts = utility_matrix.count(axis=0)
n_users = utility_matrix.shape[0]
m_merged['Actual Ratings'] = m_merged['mID'].map(movie_counts).fillna(0).astype(int)
m_merged['Sparsity (%)'] = 100 * (n_users - m_merged['Actual Ratings']) / n_users

# Sort the final DataFrame by the highest error from either model.
m_merged['max_RMSE'] = m_merged[['RMSE_NMF','RMSE_SVD']].max(axis=1)
out_movies = (m_merged
            .sort_values('max_RMSE', ascending=False)
            .drop(columns=['max_RMSE'])
            .reset_index(drop=True))

print("--- Top Error Movies: NMF vs SVD ---")
# display(out_movies.style.format({'RMSE_NMF':'{:.4f}','RMSE_SVD':'{:.4f}','Sparsity (%)':'{:.2f}'}))

# A safety check to ensure the `out_users` DataFrame from the previous step exists, then create a copy.
if 'out_users' not in globals():
    raise RuntimeError("Expected `out_users` DataFrame from the previous 7.4 step.")

df = out_users.copy()

# Calculate which model performed better for each of the top-error users.
df['rmse_diff'] = df['RMSE_NMF'] - df['RMSE_SVD']   # >0 => SVD better (NMF worse)
df['winner']    = np.where(df['rmse_diff'] > 0, 'SVD', 'NMF')  # who *won* (lower RMSE)

# Custom style to the top-error movies table.
styled_movies = out_movies.style.background_gradient(
        cmap='seismic', subset=['RMSE_NMF', 'RMSE_SVD'], vmin=out_movies[['RMSE_NMF', 'RMSE_SVD']].min().min(),
        vmax=out_movies[['RMSE_NMF', 'RMSE_SVD']].max().max()
    ) \
    .format({
        'RMSE_NMF': "{:.4f}",
        'RMSE_SVD': "{:.4f}",
        'Sparsity (%)': "{:.2f}"
    }) \
    .set_properties(**{'border': '1px solid #444', 'color': 'white'}) \
    .set_table_styles([
        {'selector': 'thead th', 'props': [
            ('background-color', '#DA291C'),
            ('color', 'white'),
            ('font-size', '24px'),
            ('font-weight', 'bold'),
        ]},
        {'selector': 'tbody th', 'props': [
            ('background-color', '#0033A0'),
            ('color', 'white'),
            ('font-size', '24px'),
            ('font-weight', 'bold'),
        ]}
    ])
```

```
display(styled_movies)
```

```
--- Top Error Movies: NMF vs SVD ---
```

	mID	RMSE_NMF	RMSE_SVD	Actual Ratings	Sparsity (%)
0		2.6255	2.5666		
1	3228	2.6061	2.5881	1	99.98
2		2.5750	2.6015		
3	2777	2.5838	2.5921	4	99.93
4		2.5861	2.5921		
5	1558	2.5918	2.5820	1	99.98
6		2.5816	2.5819		
7	690	2.5811	2.5817	1	99.98
8		2.5816	2.5816		
9	826	2.5816	2.5816	0	100.00
10		2.5816	2.5816		
11	3376	2.5816	2.5816	0	100.00
12		2.5816	2.5816		

Observation - Top Error Movies (NMF vs SVD)

The movies with the highest RMSE values are overwhelmingly characterized by extreme sparsity, with actual ratings ranging from five down to zero in the test set. The absence of substantial interaction data results in both NMF and SVD performing poorly and almost identically for these items. The near-equal RMSE values indicate that when data density drops to this level, algorithm choice has little measurable effect on prediction accuracy.

Section 8: Conclusion

When I first opened the MovieLens ratings data, the scale of the sparsity was obvious. Over 96% of the matrix was empty! With sklearn's matrix factorization tools unable to handle missing values directly, I had to fill them. I chose the global mean rating of 3.58, aware it was a crude choice but necessary to proceed.

With a dense matrix in place, I trained two models: Non-negative Matrix Factorization and Truncated SVD. I tuned each over a range of component counts, tracking RMSE and runtime. Both models ended up in the same range for accuracy, around 1.018 for NMF at k=30 and 1.017 for SVD at k=20. On paper, that is essentially a tie, but SVD was significantly faster. Even with full tuning, SVD's total runtime was a fraction of NMF's.

Comparing these results to the Week 3 baselines revealed a clear gap. The best similarity-based method from Week 3 achieved an RMSE of 0.95. Both NMF and SVD here were about 0.07 worse. The primary cause was clear, the imputation step had replaced most of the matrix with averaged values, forcing the models to learn from noise they could not explain in a meaningful way.

This made the limits of sklearn's NMF clear. It is not a purpose-built recommender algorithm. It does not train only on observed ratings, it does not incorporate user or item bias terms, and it cannot take advantage of the structure in a sparse utility matrix. In this setting, that hurts performance.

If the goal is simply to choose between the two, SVD is the better option. It matches NMF in accuracy, is faster, and is easier to implement. However, if the goal is to build the best recommender for this dataset, the earlier neighbor-based methods still outperform

both. Closing that gap would require a model that works directly with the sparse data, incorporates bias terms, and avoids relying on filler values.