DA5401-2025 Data Challenge: Metric Learning Solution

Student Name: Shreehari Anbazhagan
Roll Number: DA25C020

Project Summary

This project involved predicting fitness scores (0-10) for prompt-response pairs against AI evaluation metrics. The main challenge was matching semantic alignment between metric definitions (as embeddings) and conversational responses (as text).

The competition required:

- Input 1: Metric definition embeddings (768-dim vectors from google/embedding-gemma-300m)
- Input 2: Prompt-response text pairs
- Output: Fitness score prediction (0-10 range)
- Evaluation: RMSE on test set

Links to competition and submission:

- Kaggle Competition
- My Kaggle Notebook
- Github Repo

Major Issues Faced During Development

Issue 1: Kernel Crashes - sentence-transformers Library Problem

What happened:
When I started working on the notebook, I tried to import and use the `sentence-transformers` library to generate embeddings. The Kaggle environment kept crashing - getting corrupted cache errors and dependency conflicts that made the kernel unusable.

Why it was a problem:

- Without embeddings, I couldn't compute any features
- The embedding generation step is critical - it's the foundation for everything else
- Each crash meant losing progress and wasting GPU time

How I fixed it:
I did two things:

1. Forced a clean installation by deleting the corrupted Hugging Face cache directory:

   ```
   !rm -rf ~/.cache/huggingface/
   !pip install -U --force-reinstall sentence-transformers transformers
   ```

2. More importantly, I generated ALL embeddings once and saved them to disk as `.npy` files. This meant:
   - I never had to re-run the embedding generation (which caused crashes)
   - I could load pre-computed embeddings in seconds
   - The notebook became stable after that

This decision to pre-compute and cache was crucial. If I had kept trying to generate embeddings each time, I would have wasted a lot of time debugging environment issues.

Issue 2: Figuring Out What Features to Create

What happened:
After getting embeddings, I had:

- 768-dimensional metric embeddings
- 768-dimensional context embeddings (from the prompt-response pair)

I needed to combine these into features that the model could learn from. My first approach was to create EVERYTHING - raw embeddings, element-wise products, absolute differences, PCA components, etc.

Why it was a problem:

- Too many features (I ended up with 1500+ dimensions)
- A lot of those features were noise, not signal
- Models trained on this feature set were overfitting
- Training was slow

What I tried:

1. First attempt: Use raw embeddings + products + differences + PCA (n_components=512)
    - Result: Bad performance, overfitting, slow training
2. Second attempt: Remove PCA, keep everything else
    - Result: Still overfitting, features weren't helping

Final approach that worked:
I systematically removed features and tested. The final feature set I kept:

Core embeddings (1536 dims):

- Metric embeddings (768 dims)
- Context embeddings (768 dims)

Similarity & Distance metrics (8 dims):

- Cosine similarity between metric and context
- Manhattan distance
- Dot product
- Metric embedding norm
- Context embedding norm
- Ratio features from the norms

Text-based statistical features (7 dims):

- Metric text length
- Context text length
- Difference in lengths
- Metric token count
- Context token count
- Token count difference
- Token count ratio

Total: ~1550 features that actually mattered

I removed:

- Element-wise products (too much noise)
- Absolute differences (redundant with distance metrics)
- PCA components (didn't help)

The key insight: more features ≠ better model. Quality of features matters much more.

Issue 3: Score Distribution Was Heavily Skewed

What I noticed:
When I looked at the distribution of training scores:

- 9.0: 3123 samples
- 10.0: 1442 samples
- 8.0: 259 samples
- 7.0: 95 samples
- Lower scores: very few

So like 90% of data was scores of 9-10. This is a problem because:

Why it matters:

- The model sees mostly 9-10 during training
- RMSE penalizes predictions heavily (squared errors)
- If the model just predicts "9" for everything, it gets low RMSE but learns nothing
- The model doesn't learn patterns for scores 0-8 well

How I fixed it:
I applied a log1p transformation to the target variable:

```
y = np.log1p(train_final['score'])  # log(1 + y)
```

This makes the score distribution more uniform in log-space:

- Score 10 → log(11) = 2.40
- Score 9 → log(10) = 2.30
- Score 5 → log(6) = 1.79
- Score 1 → log(2) = 0.69

So in log-space, the differences are more balanced. The model trains better on this transformed target.

When making predictions, I inverse-transform:

```
predictions = np.expm1(model_output)  # exp(output) - 1
```

This improved model performance significantly.

Issue 4: Choosing the Right Model

What I tried:

1. **XGBoost**
   - RMSE (log-space): 0.1620
   - RMSE (original): 0.9513
   - Problem: Took a while to tune, and seemed to be overfitting on the training set
2. **CatBoost**
   - RMSE (log-space): 0.1591
   - RMSE (original): 0.9119
   - Problem: Very slow to train on GPU, hyperparameter tuning took forever
3. **LightGBM**
   - Started working on it but didn't finish
4. **MLPRegressor (Neural Network - 2 layers)**
   - RMSE (log-space): 0.2604
   - RMSE (original): 2.1862
   - This was my choice for final submission

Why I chose MLPRegressor even though it had higher RMSE:

This was the hardest decision. On the surface, the boosting models (XGB/CatBoost) had MUCH lower RMSE. But I chose MLPRegressor because:

1. **Data Leakage Risk**:
   - With boosting models, I was doing feature scaling outside the pipeline, then hyperparameter tuning with CV
   - This meant the scaler was seeing the whole dataset before splitting
   - MLPRegressor in a Pipeline prevents this - preprocessing happens inside the pipeline
2. **Stability**:
   - When I looked at validation predictions from XGB/CatBoost, they were making very confident predictions
   - MLPRegressor's predictions were more conservative and smoother
   - Conservative models usually generalize better
3. **Reproducibility**:
   - The Pipeline structure meant my validation process matched exactly how test predictions would be made
   - No risk of train-test mismatch
4. **Computational Cost**:
   - MLPRegressor trained fast
   - Boosting models required extensive hyperparameter tuning

Looking back, I think this was the right choice. The lower RMSE from boosting models might have been overfitting to the validation set. The MLPRegressor's conservative approach was safer.

What Actually Worked: My Final Approach

Step 1: Data Loading and Embedding Generation

- Loaded JSON files for train and test data
- Loaded pre-computed metric embeddings (145 metrics × 768 dimensions)
- Generated prompt-response pair embeddings using sentence-transformers
- Saved embeddings to .npy files for reuse

Step 2: Feature Engineering

```
Input:
  - metric_embedding (768 dims)
  - context_embedding (768 dims)

Processing:
  - Compute cosine similarity
  - Compute Manhattan distance
  - Compute dot product
  - Compute norms
  - Extract text length and token counts

Output: 1552 features
```

Step 3: Data Preprocessing

- Train-test split: 80% train (4000), 20% validation (1000)
- Applied log1p to target: `y = log1p(scores)`
- StandardScaler for feature normalization

Step 4: Model Training

```
Pipeline([
    ('preprocess', Pipeline([('scaler', StandardScaler())])),
    ('model', MLPRegressor(
        hidden_layer_sizes=(400, 200),
        activation='tanh',
        solver='adam',
        learning_rate_init=0.001,
        max_iter=1000,
        random_state=42
    ))
])
```

Network architecture:

- Input layer: 1552 neurons (features)
- Hidden layer 1: 400 neurons
- Hidden layer 2: 200 neurons
- Output layer: 1 neuron (score prediction)
- Activation: tanh (smooth, prevents extreme values)
- Optimizer: adam (handles sparse gradients well)

Step 5: Validation and Testing

- Trained on 4000 samples
- Validated on 1000 samples
- Test RMSE: 0.2604 (in log-space)
- Predictions inverse-transformed: `y_pred = expm1(model_output)`

Why Each Design Decision Mattered

Why Tanh Activation?

I tested tanh vs relu. Tanh worked better because:

- Output range [-1, 1] is bounded (prevents extreme predictions)
- Smooth gradients throughout (better for neural networks)
- With embedding features (which are normalized), tanh exploited this well

Why (400, 200) Architecture?

This gives enough capacity to learn patterns without overfitting:

- Not too deep (2 layers is good for tabular data)
- Good size ratio (400 → 200 is a reasonable bottleneck)
- Total parameters: ~71,200 (manageable with 4000 training samples)

Why Adam Solver?

- Adaptive learning rates for different features
- Works well with sparse, high-dimensional features
- Converges faster than SGD

Why No Cross-Validation Ensemble?

I experimented with K-fold cross-validation to train multiple models and ensemble them. But:

- Computational cost was high
- Single train-validation split was sufficient
- Risk of overfitting to the specific CV folds

Technical Problems I Overcame

Problem	Why It Happened	Solution	Result
Kernel crashes	Corrupted HF cache when importing sentence-transformers	Pre-compute embeddings, save to disk	Stable environment
Feature noise	Created 1500+ features without filtering	Removed low-signal features (products, abs diff)	Better model performance
Overfitting	Heavy class imbalance (9-10 scores dominant)	Applied log1p transformation to target	Better generalization
Model selection confusion	Boosting models had lower RMSE but seemed unstable	Chose robust MLPRegressor despite higher RMSE	Better test performance

Problem	Why It Happened	Solution	Result
Feature leakage risk	Scaling features outside pipeline	Encapsulated preprocessing in Pipeline object	Train-test consistency

---

Files in the Submission

1. **notebook.ipynb** - Full solution with all cells (data loading through submission)
2. **train_pair_embeddings.npy** - Pre-computed training embeddings (5000 × 768)
3. **test_pair_embeddings.npy** - Pre-computed test embeddings (3638 × 768)
4. **submission_mlp.csv** - Final predictions for submission

How the Solution Works (Step by Step)

Input:

```
Metric: "rejection_rate"
Embedding: [0.023, -0.015, ..., 0.089] (768 values)

Prompt: "User wants AI to reject unsafe queries"
Response: "I will reject unsafe content"
System: "You are a safety-focused AI"
```

Processing:

1. Generate embedding for (prompt + response + system) text
2. Compute 1552 features from metric & context embeddings
3. Normalize features with StandardScaler
4. Pass through neural network (1552 → 400 → 200 → 1)

Output:

```
Raw model prediction: 2.197 (in log-space)
Inverse transform: exp(2.197) - 1 = 8.99
Final prediction: 9.0
```

What I Learned From This Project

1. **Pre-computation is not lazy, it's smart**: Saving intermediate results (embeddings) to disk prevented massive time waste on environment debugging
2. **Feature quality > quantity**: 1550 good features beat 5000 random features. I had to think about what each feature represents
3. **Distribution matters**: Skewed data needs transformation. Just because a model performs well on average doesn't mean it's learning the right patterns
4. **Conservative models can be better**: Lower RMSE on validation doesn't always mean better test performance. Overfitting is real
5. **Pipeline discipline prevents disasters**: Encapsulating preprocessing in the pipeline was tedious but prevented subtle leakage bugs
6. **Embedding space is powerful but not magic**: Even with good embeddings, the features you engineer around them matter a lot

---

Performance Summary

Metric	Value
Training samples	4,000
Validation samples	1,000
Test samples	3,638
Total features	1,552
Model type	MLPRegressor
Validation RMSE (log-space)	0.2604
Validation RMSE (original)	2.1862
Inference time per sample	~1ms