Sarcasm Detection Baseline Pipeline

Technical Code Documentation

Overview

This document describes each script in the baseline experiment pipeline, explaining what it does, how it works, and what outputs it produces.

The pipeline consists of 7 scripts executed sequentially:

Order	Script	Purpose
1	step1_split_data.py	Data loading and splitting
2	step2_preprocess.py	Text preprocessing
3	step3_run_baselines.py	Model training and evaluation
4	step4_analysis_quantitative.py	Quantitative error analysis
5	step5_analysis_linguistic.py	Linguistic feature analysis
6	step6_analysis_clustering.py	TF-IDF clustering
6b	step6b_analysis_clustering_semantic.py	Semantic clustering

Step 1: Data Loading and Splitting

Script: `step1_split_data.py`

What It Does

1. Loads the main dataset (orange) and the gold dataset from CSV files
2. Removes gold examples from the orange dataset to prevent data leakage
3. Creates stratified train/validation/test splits from the remaining orange data
4. Saves all splits to CSV files

How It Works

Loading:

```
orange_df = pd.read_csv(ORANGE_DATASET_PATH)
gold_df = pd.read_csv(GOLD_DATASET_PATH)
```

Removing Gold from Orange:

- Creates a set of all text values in the gold dataset

- Filters orange to exclude any rows where the text matches a gold example
- This prevents the model from seeing gold examples during training

```
gold_texts = set(gold_df['text'].str.strip())
mask = ~orange_df['text'].str.strip().isin(gold_texts)
orange_clean_df = orange_df[mask].reset_index(drop=True)
```

Creating Splits:

- Uses scikit-learn's `train_test_split` with stratification
- First split: 80% train, 20% temporary
- Second split: temporary split 50/50 into validation and test
- Stratification ensures label distribution is preserved in each split

```
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y,
    test_size=0.20,
    stratify=y,
    random_state=42
)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp,
    test_size=0.5,
    stratify=y_temp,
    random_state=42
)
```

Outputs

File	Location	Contents
train.csv	outputs/data_splits/	529 training examples
val.csv	outputs/data_splits/	66 validation examples
test.csv	outputs/data_splits/	67 test examples
gold.csv	outputs/data_splits/	28 gold examples
split_info.txt	outputs/data_splits/	Summary statistics

Results

```
Original sizes:
  Orange: 690
  Gold: 28
  Orange-clean (after removing gold): 662
```

```
Split sizes:
  Train: 529 (79.9%)
  Val: 66 (10.0%)
  Test: 67 (10.1%)
  Gold: 28 (held-out)

Label distributions:
  Train - Label 0: 266, Label 1: 263
  Val   - Label 0: 33, Label 1: 33
  Test  - Label 0: 34, Label 1: 33
  Gold  - Label 0: 12, Label 1: 16
```

Step 2: Text Preprocessing

Script: `step2_preprocess.py`

What It Does

1. Loads the splits created in Step 1
2. Applies preprocessing transformations for different experimental conditions
3. Creates preprocessed versions of each split for each condition
4. Saves 48 preprocessed CSV files (4 splits × 12 conditions)

How It Works

Preprocessing Class:

The `TextPreprocessor` class applies the following transformations:

1. **Basic Cleaning** (applied to all text):
    - Removes HTML tags (e.g., `<i>text</i>`)
    - Normalizes smart quotes to ASCII equivalents
    - Removes carriage returns
    - Normalizes whitespace
2. **TF-IDF Preprocessing:**
    - Applies basic cleaning
    - Replaces pipe separators " | " with ". " (period + space)
    - Optionally normalizes character names to "SPEAKER:"
    - Converts to lowercase
3. **Embedding Preprocessing:**
    - Applies basic cleaning
    - Replaces pipe separators " | " with newline characters
    - Optionally normalizes character names to "SPEAKER:"
    - Preserves original case

Name Normalization:

```
def normalize_names(self, text: str) -> str:
    # Pattern matches "SHELDON:", "PERSON1:", etc.
    return self.name_pattern.sub('SPEAKER:', text)
```

Input Type Preparation:

```
if input_type == 'text_only':
    return txt
elif input_type == 'context_only':
    return ctx
else:  # context_text
    return f"Context: {ctx} Response: {txt}"
```

Experimental Conditions

Condition	Model Type	Input Type	Name Normalization
1	tfidf	text_only	original
2	tfidf	text_only	normalized
3	tfidf	context_text	original
4	tfidf	context_text	normalized
5	tfidf	context_only	original
6	tfidf	context_only	normalized
7	embeddings	text_only	original
8	embeddings	text_only	normalized
9	embeddings	context_text	original
10	embeddings	context_text	normalized
11	embeddings	context_only	original
12	embeddings	context_only	normalized

Outputs

Location	Contents
outputs/preprocessed/	48 CSV files

Each file contains the original columns plus a `preprocessed` column with the transformed text.

File naming convention: `{split}_{model_type}_{input_type}_{normalization}.csv`

Example: `train_tfidf_text_only_original.csv`

Results

48 preprocessed files created:

- 4 splits (train, val, test, gold)
- × 2 model types (tfidf, embeddings)
- × 3 input types (text_only, context_text, context_only)
- × 2 normalization options (original, normalized)

Step 3: Run Baseline Models

Script: `step3_run_baselines.py`

What It Does

1. Loads preprocessed data from Step 2
2. Trains and evaluates three tiers of baseline models
3. Saves predictions for each condition
4. Saves a summary of all results

How It Works

Tier 1: Sanity Check Baselines

Random Baseline:

- Calculates class probabilities from training data
- Predicts by sampling from these probabilities

```
class_probs = [(train_labels == 0).mean(), (train_labels == 1).mean()]
y_pred_random = np.random.choice([0, 1], size=n, p=class_probs)
```

Majority Baseline:

- Identifies the most common class in training data
- Predicts that class for all examples

```
majority_class = int(pd.Series(train_labels).mode()[0])
y_pred_majority = np.full(n, majority_class)
```

Tier 2: TF-IDF + Classical ML

1. Loads preprocessed TF-IDF data
2. Applies TF-IDF vectorization:

```
vectorizer = TfidfVectorizer(max_features=5000, ngram_range=(1, 2))
X_train_tfidf = vectorizer.fit_transform(X_train)
```

3. Trains Logistic Regression and SVM classifiers
4. Evaluates on test and gold sets

Tier 3: Frozen XLM-R Embeddings

1. Loads XLM-RoBERTa model and tokenizer:

```
tokenizer = XLMRobertaTokenizer.from_pretrained('xlm-roberta-base')
model = XLMRobertaModel.from_pretrained('xlm-roberta-base')
model.eval()  # Freeze weights
```

2. Extracts embeddings using two pooling methods:

CLS Pooling:

```
batch_embeddings = hidden_states[:, 0, :].cpu().numpy()
```

Mean Pooling:

```
attention_mask = inputs['attention_mask'].unsqueeze(-1)
masked_hidden = hidden_states * attention_mask
sum_hidden = masked_hidden.sum(dim=1)
count = attention_mask.sum(dim=1)
batch_embeddings = (sum_hidden / count).cpu().numpy()
```

3. Trains Logistic Regression on extracted embeddings
4. Evaluates on test and gold sets

Evaluation Metrics:

- Accuracy
- Precision (macro)
- Recall (macro)
- F1 Score (macro)
- Confusion matrix (TP, TN, FP, FN)

Outputs

File	Location	Contents
results_summary.csv	outputs/	All metrics for all conditions

File	Location	Contents
*.csv	outputs/predictions/	Per-example predictions (48 files)

Prediction files contain:

- example_id
- context
- text
- true_label
- predicted_label
- correct (0 or 1)
- confidence (probability of class 1)
- model_name
- condition

Results

Tier 1 Results:

Model	Test Accuracy	Gold Accuracy
Random	50.75%	53.57%
Majority	50.75%	42.86%

Tier 2 Results (selected):

Model	Condition	Test Acc	Gold Acc
tfidf_logreg	text_only_original	68.66%	75.00%
tfidf_svm	text_only_original	64.18%	71.43%
tfidf_logreg	text_only_normalized	53.73%	60.71%
tfidf_svm	context_text_original	68.66%	60.71%

Tier 3 Results (selected):

Condition	Pooling	Test Acc	Gold Acc
text_only_original	mean	68.66%	82.14%
text_only_original	cls	61.19%	71.43%
text_only_normalized	mean	59.70%	78.57%
context_text_original	cls	68.66%	75.00%

Step 4: Quantitative Error Analysis

Script: `step4_analysis_quantitative.py`

What It Does

1. Loads all prediction files from Step 3
2. Computes error breakdown by experimental condition
3. Analyzes the effect of name normalization
4. Analyzes the effect of including context
5. Identifies examples that "flip" between conditions
6. Generates a summary report

How It Works

Loading Predictions:

- Reads all CSV files from the predictions directory
- Parses filenames to extract metadata (tier, model, condition, dataset)
- Combines into a single DataFrame

Error Breakdown: For each experimental condition, computes:

- Total examples
- Correct/incorrect counts
- TP, TN, FP, FN
- Accuracy, precision, recall, F1
- FP rate, FN rate

Normalization Effect Analysis:

- Pairs conditions that differ only in normalization setting
- Computes accuracy difference between original and normalized
- Identifies conditions where normalization helps or hurts

Input Type Effect Analysis:

- Pairs conditions that differ only in input type
- Computes accuracy difference between text_only and context_text
- Identifies conditions where context helps or hurts

Flip Analysis:

Normalization flips:

```
for ex_id in common_ids:
    orig_correct = orig[orig['example_id'] == ex_id]['correct'].iloc[0]
    norm_correct = norm[norm['example_id'] == ex_id]['correct'].iloc[0]
    if orig_correct != norm_correct:
        # Record flip
```

Context flips: Same logic, comparing text_only vs context_text

Model flips: Identifies examples where different models disagree

Outputs

File	Location	Contents
error_breakdown.csv	outputs/analysis/	Metrics per condition
normalization_effect.csv	outputs/analysis/	Original vs normalized comparison
input_type_effect.csv	outputs/analysis/	text_only vs context_text comparison
flips_by_normalization.csv	outputs/analysis/	Examples that flip with normalization
flips_by_input_type.csv	outputs/analysis/	Examples that flip with context
flips_by_model.csv	outputs/analysis/	Examples where models disagree
quantitative_report.txt	outputs/analysis/	Human-readable summary

Results

Normalization Effect:

- Mean accuracy change: -4.10%
- Conditions where normalization helps: 3
- Conditions where normalization hurts: 12
- Conditions with no change: 1

Input Type Effect:

- Mean accuracy change: +0.32%
- Conditions where context helps: 8
- Conditions where context hurts: 5

Flip Counts:

- Normalization flips: 157 total (66 help, 91 hurt)
- Input type flips: 141 total (83 help, 58 hurt)
- Model disagreements: 92 unique examples

Step 5: Linguistic Feature Analysis

Script: `step5_analysis_linguistic.py`

What It Does

1. Loads all prediction files
2. Extracts linguistic features from each text
3. Compares features between correct and incorrect predictions
4. Compares features between False Positives and False Negatives
5. Performs statistical significance testing
6. Generates a summary report

How It Works

Feature Extraction:

The `LinguisticFeatureExtractor` class extracts 28 features:

Basic Features:

- char_length: Number of characters
- word_count: Number of words
- avg_word_length: Average word length
- sentence_count: Number of sentences

Punctuation Features:

- exclamation_count: Count of "!"
- question_count: Count of "?"
- ellipsis_count: Count of "..."
- quotes_count: Count of quotation marks
- comma_count: Count of commas
- has_exclamation, has_question, has_ellipsis: Binary flags
- punct_density: Punctuation per word

Lexical Features:

- has_negation, negation_count: Presence of negation words (not, never, don't, etc.)
- has_intensifier, intensifier_count: Presence of intensifiers (very, really, totally, etc.)
- has_sarcasm_marker, sarcasm_marker_count: Presence of marker words (oh, great, sure, etc.)
- type_token_ratio: Lexical diversity
- all_caps_count: Words in ALL CAPS

Sentiment Features:

- positive_word_count: Count of positive words
- negative_word_count: Count of negative words
- sentiment_polarity: (positive - negative) / total
- sentiment_strength: Total sentiment words / word count

Structural Features:

- starts_with_interjection: Starts with oh, wow, well, etc.
- has_rhetorical_pattern: Contains "who/what/why/how ... ?"
- has_contrast: Contains but, however, although, etc.

Statistical Comparison:

```
from scipy import stats
t_stat, p_value = stats.ttest_ind(group1_values, group2_values)
```

Features with p < 0.05 are marked as statistically significant.

Outputs

File	Location	Contents
linguistic_features.csv	outputs/analysis/	All features for all predictions
correct_vs_incorrect.csv	outputs/analysis/	Feature comparison with p-values
fp_vs_fn.csv	outputs/analysis/	FP vs FN feature comparison
linguistic_report.txt	outputs/analysis/	Human-readable summary

Results

Total Predictions Analyzed: 2,280

- Correct: 1,463 (64.2%)
- Incorrect: 817 (35.8%)

Statistically Significant Features: 20 out of 28

Top Features Higher in Incorrect Predictions:

Feature	Difference
all_caps_count	+1440%
ellipsis_count	+165.5%
has_ellipsis	+165.5%
sentiment_polarity	+113.7%
sarcasm_marker_count	+64.1%
starts_with_interjection	+60.7%
positive_word_count	+56.9%

Top Features Lower in Incorrect Predictions:

Feature	Difference
negative_word_count	-38.1%
has_contrast	-37.9%
has_rhetorical_pattern	-26.3%
negation_count	-24.8%

FP vs FN Comparison:

False Positives: 365 False Negatives: 452

Features higher in FN than FP:

Feature	Difference
intensifier_count	+582.7%
has_intensifier	+465.3%
exclamation_count	+121.1%
sarcasm_marker_count	+116.9%

Step 6: TF-IDF Clustering Analysis

Script: `step6_analysis_clustering.py`

What It Does

1. Loads all prediction files and filters to errors only
2. Deduplicates errors by text (same text appears in multiple conditions)
3. Computes TF-IDF vectors for error texts
4. Applies K-Means clustering with automatic k selection
5. Analyzes each cluster for patterns
6. Generates a summary report

How It Works

Loading Errors:

```
errors = combined[combined['correct'] == 0].copy()
errors_dedup = errors.drop_duplicates(subset=
['text']).reset_index(drop=True)
```

TF-IDF Embedding:

```
vectorizer = TfidfVectorizer(max_features=1000, ngram_range=(1, 2),
stop_words='english')
X = vectorizer.fit_transform(texts).toarray()
```

Automatic K Selection:

- Tests k values from 4 to 12
- Computes silhouette score for each k
- Selects k with highest silhouette score

```
for k in range(min_k, max_k + 1):
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(X)
    score = silhouette_score(X, labels)
```

Cluster Analysis: For each cluster, computes:

- Size (number of examples)
- Label distribution (sarcastic vs literal)
- FP count and FN count
- Primary error type (FP or FN dominant)
- Top words (excluding stopwords)
- Character mentions
- Average word count
- Punctuation percentages (exclamation, question, ellipsis)
- Example texts

Outputs

File	Location	Contents
clustered_errors.csv	outputs/clustering_analysis/	Errors with cluster labels
cluster_analysis.json	outputs/clustering_analysis/	Per-cluster statistics
cluster_report.txt	outputs/clustering_analysis/	Human-readable summary

Results

Clustering Overview:

- Number of clusters: 12
- Embedding type: TF-IDF
- Silhouette score: 0.0153
- Total unique errors: 78

Cluster Summary:

Cluster	Size	Type	Top Character
0	3	FN	bernadette
1	12	FN	chandler
2	10	FP	monica
3	9	FN	chandler
4	13	FP	sheldon
5	5	FN	leonard
6	3	FP	raj
7	7	FN	howard
8	5	FP	person

Cluster	Size	Type	Top Character
9	4	FN	dorothy
10	3	FP	phoebe
11	4	FP	amy

Character Mentions in Errors:

- chandler: 21
- howard: 10
- leonard: 9
- person: 7
- sheldon: 6

Step 6b: Semantic Embedding Clustering

Script: `step6b_analysis_clustering_semantic.py`

What It Does

1. Loads all prediction files and filters to errors only
2. Deduplicates errors by text
3. Computes semantic embeddings using sentence-transformers
4. Applies K-Means clustering with automatic k selection
5. Analyzes each cluster for patterns
6. Generates a summary report

How It Works

Semantic Embedding:

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
embeddings = model.encode(texts, show_progress_bar=True)
```

The model converts each text into a 384-dimensional vector that captures semantic meaning.

Clustering and Analysis: Same as Step 6, but operating on semantic embeddings instead of TF-IDF vectors.

Difference from TF-IDF Clustering

Aspect	TF-IDF	Semantic
Similarity basis	Shared words	Shared meaning
Vector dimension	1000 (sparse)	384 (dense)

Aspect	TF-IDF	Semantic
Model	Bag of words	Neural network

Outputs

File	Location	Contents
clustered_errors_semantic.csv	outputs/clustering_semantic/	Errors with cluster labels
cluster_analysis_semantic.json	outputs/clustering_semantic/	Per-cluster statistics
cluster_report_semantic.txt	outputs/clustering_semantic/	Human-readable summary

Results

Clustering Overview:

- Number of clusters: 8
- Embedding type: semantic
- Embedding model: all-MiniLM-L6-v2
- Embedding dimensions: 384
- Silhouette score: 0.0860
- Total unique errors: 78

Cluster Summary:

Cluster	Size	Type	Top Character
0	5	FN	bernadette
1	34	FN	chandler
2	6	FN	amy
3	11	FP	leonard
4	3	FN	dorothy
5	13	FN	howard
6	3	FN	person
7	3	FP	sheldon

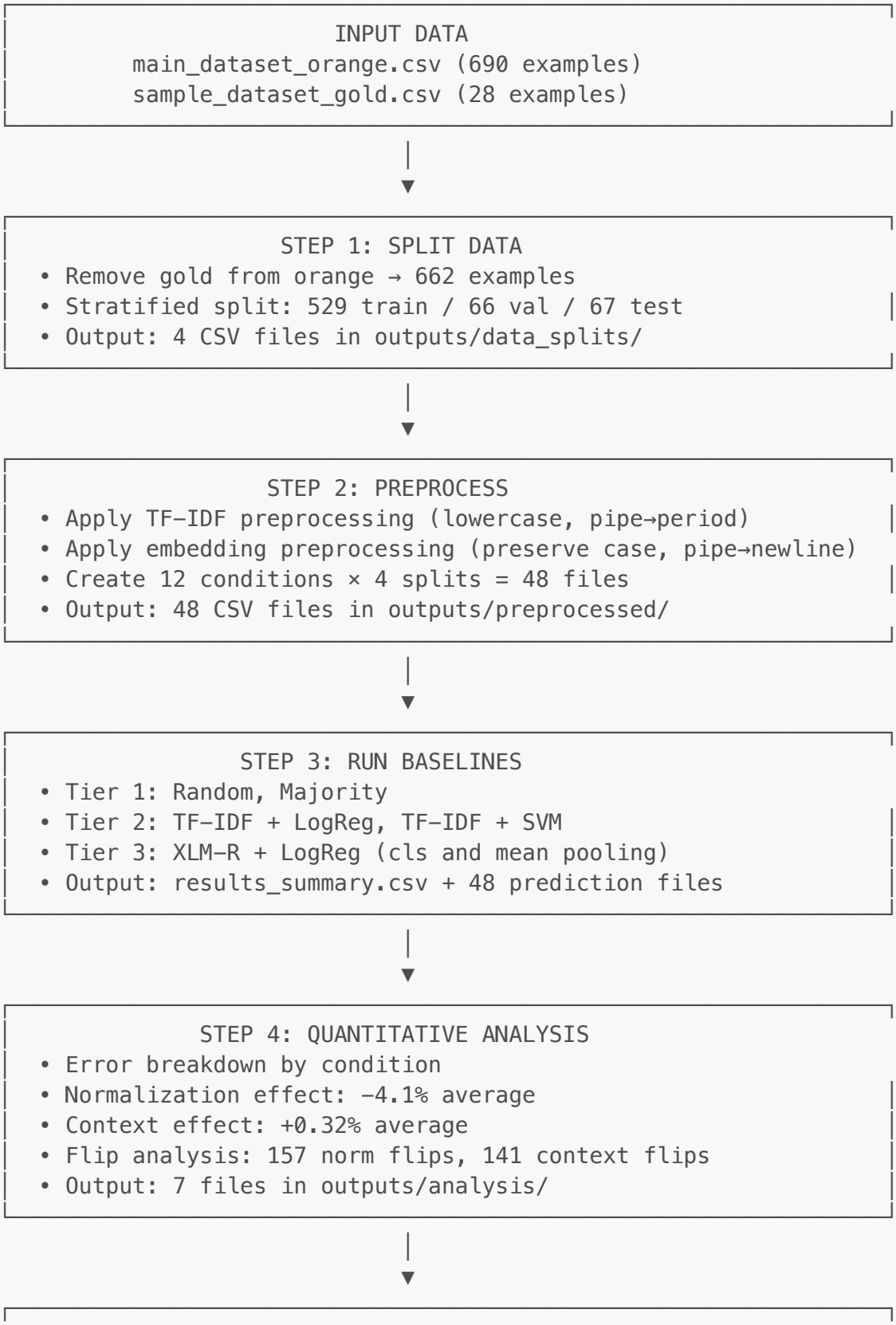
Character Mentions in Errors:

- chandler: 21
- howard: 10
- leonard: 9
- person: 7
- sheldon: 6

Comparison with TF-IDF:

- Semantic clustering produces fewer, larger clusters (8 vs 12)
- Higher silhouette score (0.0860 vs 0.0153) indicates better cluster coherence
- Cluster 1 contains 34 examples (44% of all errors), mostly Chandler
- Character mention counts are identical across both methods

Summary: Pipeline Flow



```
┌─────────────────────────────────────────────────────────────┐
│              STEP 5: LINGUISTIC ANALYSIS                    │
│  • Extract 28 linguistic features                           │
│  • Compare correct vs incorrect predictions                 │
│  • Compare FP vs FN                                         │
│  • Statistical significance testing                         │
│  • Output: 4 files in outputs/analysis/                     │
└─────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────┐
│               STEP 6: TF-IDF CLUSTERING                     │
│  • 78 unique errors                                         │
│  • 12 clusters, silhouette = 0.0153                         │
│  • Output: 3 files in outputs/clustering_analysis/          │
└─────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────┐
│             STEP 6b: SEMANTIC CLUSTERING                    │
│  • 78 unique errors                                         │
│  • 8 clusters, silhouette = 0.0860                          │
│  • Output: 3 files in outputs/clustering_semantic/          │
└─────────────────────────────────────────────────────────────┘
```

Configuration Reference

Global Settings

Parameter	Value
Random seed	42
Train ratio	80%
Validation ratio	10%
Test ratio	10%

TF-IDF Settings

Parameter	Value
max_features	5000
ngram_range	(1, 2)
stop_words	english

XLM-R Settings

Parameter	Value

Parameter	Value
Model	xlm-roberta-base
Max sequence length	512
Batch size	16
Weights	Frozen (no fine-tuning)

Clustering Settings

Parameter	Value
Algorithm	K-Means
K selection	Auto (silhouette score)
K range	4-12
Semantic model	all-MiniLM-L6-v2

End of Technical Documentation