Contents

1 Using Class Weights in RNN Model

1.1 Overview

Class weights allow you to address class imbalance by penalizing errors on minority classes more heavily during training. This is particularly useful when you have severe imbalance (e.g., Nomenclature:Description:Misc = 1:10:100).

1.2 Quick Start

1.2.1 1. Define Class Weights

Class weights are specified as a dictionary mapping label strings to weight values:

```
# Recommended for SKOL text classification
class_weights = {
    "Nomenclature": 100.0,  # Rarest class - penalize errors heavily
    "Description": 10.0,    # Moderate weight
    "Misc": 0.1             # Most common - barely penalize
}
```

1.2.2 2. Configure Model with Class Weights

Add the `class_weights` parameter to your model configuration:

```
model_config = {
    'model_type': 'rnn',
    'window_size': 15,
    'prediction_stride': 5,
    'hidden_size': 128,
    'num_layers': 2,
    'dropout': 0.3,
    'epochs': 6,
    'batch_size': 32,
    'verbosity': 1,

    # NEW: Add class weights
    'class_weights': {
        "Nomenclature": 100.0,
        "Description": 10.0,
        "Misc": 0.1
    }
}
```

1.2.3 3. Train Classifier

The classifier will automatically use weighted loss during training:

```
from skol_classifier.classifier_v2 import SkolClassifierV2

classifier = SkolClassifierV2(
    spark=spark,
    input_source='files',
    file_paths=annotated_files,
    auto_load_model=False,
    **model_config
)

# Train with class weights
```

```
# The labels must be provided for class weights to work
results = classifier.fit()
```

1.3 Complete Example

```
import redis
from pyspark.sql import SparkSession
from skol_classifier.classifier_v2 import SkolClassifierV2

# Initialize Spark
spark = SparkSession.builder \
    .appName("RNN with Class Weights") \
    .getOrCreate()

# Initialize Redis (for model storage)
redis_client = redis.Redis(host='localhost', port=6379, decode_responses=False)

# Define class weights (adjust based on your class distribution)
class_weights = {
    "Nomenclature": 100.0,  # 1x - rarest, most important
    "Description": 10.0,    # 10x - important
    "Misc": 0.1             # 100x+ - least important
}

# Configure model with class weights
model_config = {
    'model_type': 'rnn',
    'window_size': 15,
    'prediction_stride': 5,
    'hidden_size': 128,
    'num_layers': 2,
    'dropout': 0.3,
    'epochs': 6,
    'batch_size': 32,
    'verbosity': 1,
    'class_weights': class_weights,  # Apply class weights
}

# Annotated training files
annotated_files = [
    'data/annotated/file1.txt.ann',
    'data/annotated/file2.txt.ann',
    # ... more files
]
```

```python
# Create classifier
classifier = SkolClassifierV2(
    spark=spark,
    input_source='files',
    file_paths=annotated_files,
    auto_load_model=False,
    model_storage='redis',
    redis_client=redis_client,
    redis_key='rnn_model_with_weights',
    **model_config
)

# Train classifier
print("Training classifier with class weights...")
results = classifier.fit()

# The model will print:
# [BiLSTM] Using weighted loss with class weights:
#   Description: 10.0
#   Misc: 0.1
#   Nomenclature: 100.0

# Evaluate (per-class metrics now include loss)
stats = results['test_stats']
print(f"Nomenclature F1: {stats['Nomenclature_f1']:.4f}")
print(f"Nomenclature Loss: {stats['Nomenclature_loss']:.4f}")
print(f"Description F1: {stats['Description_f1']:.4f}")
print(f"Description Loss: {stats['Description_loss']:.4f}")

# Save model
classifier.save_model()
print(f"Model saved to Redis with class weights applied")
```

1.4 Choosing Class Weights

1.4.1 Strategy 1: Inverse Frequency (Recommended Starting Point)

Calculate weights based on class frequencies:

```python
# If you have:
# - Nomenclature: 100 instances
# - Description: 1000 instances
# - Misc: 10000 instances

# Inverse frequency weights:
```

```
class_weights = {
    "Nomenclature": 10000 / 100,   # = 100.0
    "Description": 10000 / 1000,   # = 10.0
    "Misc": 10000 / 10000          # = 1.0
}

# Or normalize to make Misc very low (since you don't care about it):
class_weights = {
    "Nomenclature": 100.0,
    "Description": 10.0,
    "Misc": 0.1  # Nearly ignore Misc errors
}
```

1.4.2 Strategy 2: Task-Specific Weighting

Adjust based on what matters for your application:

```
# If Nomenclature is CRITICAL and Misc doesn't matter at all:
class_weights = {
    "Nomenclature": 100.0,  # Critical
    "Description": 20.0,    # Important
    "Misc": 0.01            # Essentially ignore
}

# If you want balanced performance on Nomenclature and Description:
class_weights = {
    "Nomenclature": 50.0,
    "Description": 50.0,
    "Misc": 0.1
}
```

1.4.3 Strategy 3: Iterative Tuning

Start conservative and adjust based on results:

```
# Iteration 1: Moderate weights
class_weights = {"Nomenclature": 10.0, "Description": 5.0, "Misc": 1.0}
# Train, evaluate Nomenclature F1

# If Nomenclature F1 still too low, increase its weight:
# Iteration 2: More aggressive
class_weights = {"Nomenclature": 50.0, "Description": 10.0, "Misc": 0.5}
# Train, evaluate again

# If overfitting on Nomenclature (perfect recall, low precision):
```

```
# Iteration 3: Balance
class_weights = {"Nomenclature": 30.0, "Description": 10.0, "Misc": 0.5}
```

1.5 Monitoring Training

When class weights are applied, you'll see output like:

```
[BiLSTM] Using weighted loss with class weights:
  Description: 10.0
  Misc: 0.1
  Nomenclature: 100.0
```

During training, watch for: - **Overall loss**: May be higher than without weights (this is OK) - **Per-class loss**: Check if minority class loss is decreasing - **Per-class F1**: Your primary metric for success

1.6 Expected Improvements

With proper class weights, you should see:

- ✅ **10-30% improvement** in Nomenclature F1 score
- ✅ **Better precision/recall balance** on minority classes
- ✅ **More predictions** of minority classes (less bias toward majority)
- ⚠️ **Slightly lower overall accuracy** (due to focusing on hard classes)
  - This is expected and desirable!
  - Overall accuracy is dominated by Misc, which you don't care about

1.7 Evaluation Focus

Don't use overall accuracy - use per-class metrics:

```
# After training
stats = classifier.model.calculate_stats(predictions)

# Focus on these metrics:
nom_f1 = stats['Nomenclature_f1']
desc_f1 = stats['Description_f1']

# Your success metric (ignore Misc):
important_f1 = (nom_f1 + desc_f1) / 2

print(f"Nomenclature F1: {nom_f1:.4f} □")
print(f"Description F1: {desc_f1:.4f} □")
```

```
print(f"Important Classes Avg F1: {important_f1:.4f} ← PRIMARY METRIC")
print(f"Misc F1: {stats['Misc_f1']:.4f} (don't care)")
```

1.8 Combining with Other Strategies

Class weights work well with:

1. **Increased window size** (capture more context):

   ```
   model_config = {
       'window_size': 20,  # Increased from 10
       'class_weights': {"Nomenclature": 100.0, "Description": 10.0, "Misc": 0.
   }
   ```

2. **Document-level oversampling** (preserve sequences):

   ```
   # Oversample documents containing Nomenclature
   train_data = oversample_documents_with_rare_labels(train_data, rare_label=0,

   # Then train with class weights
   classifier.fit()
   ```

3. **Sequence-aware post-processing** (apply rules after prediction):

   ```
   predictions = classifier.predict(test_data)
   predictions = apply_sequence_rules(predictions)  # Leverage Nom→Desc patterr
   stats = classifier.model.calculate_stats(predictions)
   ```

1.9 Troubleshooting

1.9.1 Problem: Loss is very high

Solution: This is normal with high class weights. Focus on per-class F1 scores, not loss magnitude.

1.9.2 Problem: Model predicts everything as minority class

Solution: Weights too aggressive. Reduce them:

```
# Too aggressive:
class_weights = {"Nomenclature": 1000.0, "Description": 100.0, "Misc": 0.01}

# Better:
class_weights = {"Nomenclature": 100.0, "Description": 10.0, "Misc": 0.1}
```

1.9.3 Problem: No improvement in minority class F1

Solution: Try increasing the minority class weight or combining with sampling:

```
# Increase weight
class_weights = {"Nomenclature": 200.0, "Description": 20.0, "Misc": 0.1}

# And/or oversample documents with Nomenclature
train_data = oversample_documents_with_rare_labels(train_data, rare_label=0, rep
```

1.9.4 Problem: Class weights not being applied

Cause: Labels not provided to fit()

Solution: Ensure labels are set:

```
# In SkolClassifierV2, labels are automatically set from the data
# If creating RNNSkolModel directly:
model = RNNSkolModel(
    input_size=300,
    class_weights={"Nomenclature": 100.0, "Description": 10.0, "Misc": 0.1}
)

# Must provide labels in fit():
model.fit(train_data, labels=["Nomenclature", "Description", "Misc"])
```

1.10 Technical Details

1.10.1 How It Works

Class weights modify the loss function:

Standard cross-entropy:

```
Loss = -log(p(true_class))
```

Weighted cross-entropy:

```
Loss = -weight[true_class] * log(p(true_class))
```

During backpropagation, errors on high-weight classes have larger gradients, causing the model to: - Update weights more aggressively for minority class errors - Learn to distinguish minority classes better - Pay less attention to majority class (if weight is low)

1.10.2 Implementation

The weighted loss is implemented in `build_bilstm_model()`:

```python
def weighted_categorical_crossentropy(y_true, y_pred):
    """Weighted cross-entropy loss."""
    epsilon = 1e-7
    y_pred = tf.clip_by_value(y_pred, epsilon, 1.0 - epsilon)

    # Standard cross-entropy
    loss = -tf.reduce_sum(y_true * tf.math.log(y_pred), axis=-1)

    # Apply class-specific weights
    class_indices = tf.argmax(y_true, axis=-1)
    weights = tf.gather(weight_tensor, class_indices)
    weighted_loss = loss * weights

    return tf.reduce_mean(weighted_loss)
```

1.11 References

- See `docs/class_imbalance_strategies.md` for comprehensive strategies
- He & Garcia “Learning from Imbalanced Data” (IEEE TKDE 2009)
- Class weights are a standard technique in imbalanced learning