Contents

1 Strategies for Handling Class Imbalance in RNN Text Classification

1.1 Problem Overview

The SKOL classifier faces severe class imbalance:

- **Nomenclature**: ~1x (rarest, most important)
- **Description**: ~10x (important, clusters after Nomenclature)
- **Misc**: ~100x+ (not important, misclassifications acceptable)

Key Characteristics: - Sequential patterns: Description lines cluster after/near Nomenclature lines - Nomenclature lines occur singly - Asymmetric importance: Nomenclature/Description matter, Misc doesn't - Sequential model (BiLSTM) can exploit temporal patterns

1.2 Strategy 1: Class Weights in Loss Function □□□

Best immediate fix - Weight the loss function to heavily penalize minority class errors.

1.2.1 Implementation

Modify `build_bilstm_model()` to accept class weights:

```
def build_bilstm_model_with_weights(
    input_shape: Tuple[int, int],
    num_classes: int,
    hidden_size: int = 128,
    num_layers: int = 2,
    dropout: float = 0.3,
    class_weights: Optional[Dict[int, float]] = None
) -> 'keras.Model':
    """
    Build a Bidirectional LSTM model with class weighting.

    Args:
        input_shape: Shape of input (sequence_length, feature_dim)
        num_classes: Number of output classes
        hidden_size: Size of LSTM hidden state
        num_layers: Number of LSTM layers
        dropout: Dropout rate
        class_weights: Dict mapping class index to weight (e.g., {0: 100, 1: 10,

    Returns:
        Compiled Keras model with weighted loss
```

```
"""
model = models.Sequential()
model.add(layers.Input(shape=input_shape))

# BiLSTM layers
for i in range(num_layers):
    model.add(layers.Bidirectional(
        layers.LSTM(hidden_size, return_sequences=True, dropout=dropout)
    ))

# Time-distributed dense layer for per-timestep classification
model.add(layers.TimeDistributed(layers.Dense(num_classes, activation='softm

# Use weighted categorical crossentropy if class weights provided
if class_weights is not None:
    import tensorflow as tf
    weight_tensor = tf.constant([class_weights.get(i, 1.0) for i in range(nu

    def weighted_categorical_crossentropy(y_true, y_pred):
        """
        Weighted cross-entropy loss for class imbalance.

        Args:
            y_true: One-hot encoded true labels, shape (batch, timesteps, nu
            y_pred: Predicted probabilities, shape (batch, timesteps, num_cl

        Returns:
            Weighted loss scalar
        """
        # Clip predictions to avoid log(0)
        epsilon = 1e-7
        y_pred = tf.clip_by_value(y_pred, epsilon, 1 - epsilon)

        # Calculate per-sample cross-entropy loss
        loss = -tf.reduce_sum(y_true * tf.math.log(y_pred), axis=-1)  # (bat

        # Apply class weights based on true class
        class_indices = tf.argmax(y_true, axis=-1)  # (batch, timesteps)
        weights = tf.gather(weight_tensor, class_indices)  # (batch, timeste
        weighted_loss = loss * weights

        return tf.reduce_mean(weighted_loss)

    loss_fn = weighted_categorical_crossentropy
else:
    loss_fn = 'categorical_crossentropy'
model.compile(
    optimizer=optimizers.Adam(learning_rate=0.001),
    loss=loss_fn,
    metrics=['accuracy']
)

return model
```

1.2.2 Recommended Class Weights

Based on your class distribution (Nom:Desc:Misc = 1:10:100+):

```
# Strategy 1: Inverse frequency weighting
class_weights = {
    0: 100.0,  # Nomenclature (rarest) - heavily penalize errors
    1: 10.0,   # Description - moderately penalize errors
    2: 0.1     # Misc - barely penalize (you don't care)
}

# Strategy 2: More aggressive (if Strategy 1 insufficient)
class_weights = {
    0: 100.0,  # Nomenclature
    1: 20.0,   # Description
    2: 0.01    # Misc - essentially ignore misclassifications
}

# Strategy 3: Only care about Nomenclature
class_weights = {
    0: 100.0,  # Nomenclature - critical
    1: 5.0,    # Description - less important
    2: 0.01    # Misc - ignore
}
```

1.2.3 Usage in 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,
    # NEW: Add class weights
    'class_weights': {0: 100.0, 1: 10.0, 2: 0.1},
}
```

1.3 Strategy 2: Focal Loss (For Extreme Imbalance) □□

Focal loss automatically down-weights easy examples (like abundant Misc labels) and focuses learning on hard examples.

1.3.1 Implementation

```
def focal_loss(gamma: float = 2.0, alpha: Optional[Dict[int, float]] = None):
    """
    Focal loss for addressing class imbalance.

    Focal Loss = -α * (1 - p_t)^γ * log(p_t)

    Where:
    - p_t is the probability of the true class
    - γ (gamma) controls how much to down-weight easy examples (typical: 2.0)
    - α (alpha) provides per-class weighting

    Args:
        gamma: Focusing parameter. Higher = more focus on hard examples
               γ=0 reduces to standard cross-entropy
               γ=2 is standard for focal loss
        alpha: Class weights dict {0: weight0, 1: weight1, 2: weight2}
               If None, uses equal weights

    Returns:
        Loss function compatible with Keras model.compile()

    References:
        Lin et al. "Focal Loss for Dense Object Detection" (2017)
    """
    import tensorflow as tf

    if alpha is None:
        alpha = {0: 100.0, 1: 10.0, 2: 0.1}

    alpha_tensor = tf.constant([alpha.get(i, 1.0) for i in range(len(alpha))])

    def loss_fn(y_true, y_pred):
        """
        Args:
            y_true: One-hot encoded labels, shape (batch, timesteps, num_classes
            y_pred: Predicted probabilities, shape (batch, timesteps, num_classe
        """
        # Clip to avoid numerical instability
        epsilon = 1e-7
        y_pred = tf.clip_by_value(y_pred, epsilon, 1 - epsilon)

        # Standard cross entropy
        ce = -y_true * tf.math.log(y_pred)

        # Focal term: (1 - p_t)^gamma
        # This down-weights easy examples (high confidence correct predictions)
        # For hard examples (low p_t), this is close to 1 (full loss)
        # For easy examples (high p_t), this is close to 0 (reduced loss)
        p_t = tf.reduce_sum(y_true * y_pred, axis=-1, keepdims=True)  # Probabil
        focal_weight = tf.pow(1 - p_t, gamma)

        # Apply focal weight and class weights
        class_indices = tf.argmax(y_true, axis=-1)
        alpha_weights = tf.gather(alpha_tensor, class_indices)
        alpha_weights = tf.expand_dims(alpha_weights, -1)

        focal_ce = focal_weight * ce * alpha_weights

        return tf.reduce_mean(tf.reduce_sum(focal_ce, axis=-1))

    return loss_fn
```

1.3.2 Usage

```
# In build_bilstm_model():
model.compile(
    optimizer=optimizers.Adam(learning_rate=0.001),
    loss=focal_loss(gamma=2.0, alpha={0: 100, 1: 10, 2: 0.1}),
    metrics=['accuracy']
)
```

1.3.3 When to Use Focal Loss

- Class imbalance is extreme (100:1 or worse)
- Weighted cross-entropy isn’t sufficient
- Many “easy” examples drowning out “hard” examples
- You want automatic hard example mining

1.4 Strategy 3: Sampling Strategies □

1.4.1 Option A: Document-Level Oversampling (Preserves Sequential Structure)

Important: Oversample entire documents, not individual lines, to preserve the sequential patterns.

```
def oversample_documents_with_rare_labels(
    train_data: DataFrame,
    rare_label: int = 0,  # Nomenclature
    replication_factor: int = 5
) -> DataFrame:
    """
    Oversample entire documents that contain rare labels.
    This preserves sequential structure and Nomenclature→Description patterns.

    Args:
        train_data: Training DataFrame with doc_id, line_number, label columns
        rare_label: Class ID to boost (0 = Nomenclature)
        replication_factor: How many times to duplicate documents with rare labe

    Returns:
        DataFrame with documents containing rare labels replicated
    """
    from pyspark.sql.functions import lit
    from functools import reduce

    # Find documents containing the rare label
    docs_with_rare = train_data.filter(col('label') == rare_label) \
        .select('doc_id').distinct()

    print(f"Found {docs_with_rare.count()} documents containing label {rare_labe

    # Mark documents
    marked = train_data.join(
        docs_with_rare.withColumn('has_rare_label', lit(True)),
        on='doc_id',
        how='left'
    ).fillna(False, subset=['has_rare_label'])

    # Split into rare and normal
    rare_docs = marked.filter(col('has_rare_label'))
    normal_docs = marked.filter(~col('has_rare_label'))

    # Replicate rare documents
    replicated = [rare_docs] * replication_factor
    balanced = reduce(lambda df1, df2: df1.union(df2), replicated + [normal_docs

    # CRITICAL: Maintain document/line ordering
    result = balanced.drop('has_rare_label').orderBy('doc_id', 'line_number')

    print(f"Original size: {train_data.count()}")
    print(f"Balanced size: {result.count()}")

    return result
```

1.4.2 Option B: Stratified Document Sampling

```
def stratified_document_sample(
    train_data: DataFrame,
    target_label_ratio: Dict[int, float] = None
) -> DataFrame:
    """
    Sample documents to achieve target label distribution.

    Args:
        train_data: Training DataFrame
        target_label_ratio: Desired ratio, e.g., {0: 0.2, 1: 0.3, 2: 0.5}
                           If None, uses equal distribution

    Returns:
        Sampled DataFrame maintaining document integrity
    """
    if target_label_ratio is None:
        num_classes = 3
        target_label_ratio = {i: 1.0 / num_classes for i in range(num_classes)}

    # For each document, determine its "primary label" (most common label in doc
    from pyspark.sql.window import Window
    from pyspark.sql.functions import count, row_number

    # Count labels per document
    doc_label_counts = train_data.groupBy('doc_id', 'label').count()

    # Find most common label per document
    window = Window.partitionBy('doc_id').orderBy(col('count').desc())
    primary_labels = doc_label_counts.withColumn('rank', row_number().over(windo
        .filter(col('rank') == 1) \
        .select('doc_id', col('label').alias('primary_label'))

    # Sample documents by primary label
    sampled_docs = []
    total_docs = primary_labels.count()

    for label, target_ratio in target_label_ratio.items():
        label_docs = primary_labels.filter(col('primary_label') == label)
        label_count = label_docs.count()
        target_count = int(total_docs * target_ratio)

        if label_count < target_count:
            # Oversample
            fraction = target_count / label_count
            sampled = label_docs.sample(withReplacement=True, fraction=fraction,
        else:
            # Undersample
            fraction = target_count / label_count
            sampled = label_docs.sample(withReplacement=False, fraction=fraction

        sampled_docs.append(sampled)

    # Combine and join back to get full documents
    from functools import reduce
    all_sampled_docs = reduce(lambda df1, df2: df1.union(df2), sampled_docs)

    result = train_data.join(
        all_sampled_docs.select('doc_id'),
        on='doc_id',
        how='inner'
    ).orderBy('doc_id', 'line_number')

    return result
```

1.5 Strategy 4: Sequence-Aware Post-Processing □□

Leverage domain knowledge: Description lines cluster after Nomenclature.

```
def apply_sequence_rules(predictions_df: DataFrame) -> DataFrame:
    """
    Apply domain-specific rules based on sequential patterns.

    Rules:
    1. If previous line was Nomenclature, next 3-20 lines likely Description
    2. If high uncertainty between Description/Misc and context suggests Descrip
    3. Nomenclature lines are typically isolated (not in clusters)
```

```
Args:
    predictions_df: DataFrame with columns: doc_id, line_number, prediction,

Returns:
    DataFrame with adjusted predictions
"""
from pyspark.sql.window import Window
from pyspark.sql.functions import lag, lead, when, col

# Window by document, ordered by line number
window = Window.partitionBy('doc_id').orderBy('line_number')

# Look at surrounding context
with_context = predictions_df \
    .withColumn('prev_pred_1', lag('prediction', 1).over(window)) \
    .withColumn('prev_pred_2', lag('prediction', 2).over(window)) \
    .withColumn('next_pred_1', lead('prediction', 1).over(window)) \
    .withColumn('next_pred_2', lead('prediction', 2).over(window)) \
    .withColumn('prob_nom', col('probabilities')[0]) \
    .withColumn('prob_desc', col('probabilities')[1]) \
    .withColumn('prob_misc', col('probabilities')[2])

# Rule 1: If previous line was Nomenclature and current is Misc,
# but Description probability is reasonable, switch to Description
rule1 = when(
    (col('prev_pred_1') == 0) &           # Previous was Nomenclature
    (col('prediction') == 2) &            # Current prediction is Misc
    (col('prob_desc') > 0.25),            # But Description is plausible (>2
    1  # Change to Description
)

# Rule 2: If surrounded by Nomenclature predictions and current is Descripti
# it's probably actually Misc (Nomenclature doesn't cluster)
rule2 = when(
    (col('prev_pred_1') == 0) &
    (col('next_pred_1') == 0) &
    (col('prediction') == 0) &
    (col('prob_misc') > 0.3),
    2  # Change to Misc
)

# Rule 3: Description confidence boost if near Nomenclature
rule3 = when(
    (
        (col('prev_pred_1') == 0) | (col('prev_pred_2') == 0) |
        (col('next_pred_1') == 0) | (col('next_pred_2') == 0)
    ) &
    (col('prediction') == 2) &            # Currently Misc
    (col('prob_desc') > 0.2),             # Description probability > 20%
    1  # Change to Description
)

# Apply rules in sequence (priority order)
adjusted = with_context.withColumn(
    'adjusted_prediction',
    rule1.otherwise(
        rule2.otherwise(
            rule3.otherwise(col('prediction'))
        )
    )
)

# Clean up and return
return adjusted \
    .withColumn('prediction', col('adjusted_prediction')) \
    .drop('prev_pred_1', 'prev_pred_2', 'next_pred_1', 'next_pred_2',
          'prob_nom', 'prob_desc', 'prob_misc', 'adjusted_prediction')
```

1.5.1 Usage

```
# After prediction
predictions = classifier.predict(test_data)

# Apply sequence-aware rules
predictions = apply_sequence_rules(predictions)

# Evaluate
stats = classifier.model.calculate_stats(predictions)
```

1.6 Strategy 5: Proper Evaluation Metrics □□□

Critical: Don’t use overall accuracy - it’s dominated by Misc class.

1.6.1 Focus on Minority Class Metrics

```
def evaluate_imbalanced_classifier(stats: Dict[str, float]) -> None:
    """
    Evaluate classifier with focus on minority classes.

    Args:
        stats: Dictionary from calculate_stats() with per-class metrics
    """
    print("\n" + "="*70)
    print("MINORITY CLASS FOCUSED EVALUATION")
    print("="*70)

    # 1. Per-class F1 scores (already in stats from enhanced calculate_stats)
    print("\nPer-Class F1 Scores:")
    print(f"  Nomenclature F1: {stats.get('Nomenclature_f1', 0):.4f}  □ MOST IMP
    print(f"  Description F1:  {stats.get('Description_f1', 0):.4f}  □ IMPORTANT
    print(f"  Misc F1:         {stats.get('Misc_f1', 0):.4f}      (ignore)")

    # 2. Macro-averaged F1 (treats all classes equally)
    # Only average the classes you care about
    important_f1 = (stats.get('Nomenclature_f1', 0) + stats.get('Description_f1'
    print(f"\nImportant Classes Avg F1: {important_f1:.4f}  ← PRIMARY METRIC")

    # 3. Per-class precision and recall
    print("\nNomenclature Performance:")
    print(f"  Precision: {stats.get('Nomenclature_precision', 0):.4f} (of predic
    print(f"  Recall:    {stats.get('Nomenclature_recall', 0):.4f} (of actual No
    print(f"  Loss:      {stats.get('Nomenclature_loss', 0):.4f}")

    print("\nDescription Performance:")
    print(f"  Precision: {stats.get('Description_precision', 0):.4f}")
    print(f"  Recall:    {stats.get('Description_recall', 0):.4f}")
    print(f"  Loss:      {stats.get('Description_loss', 0):.4f}")

    # 4. Overall accuracy (for reference only)
    print(f"\nOverall Accuracy: {stats.get('accuracy', 0):.4f}  (dominated by Mi

    # 5. Class support
    print("\nClass Distribution:")
    print(f"  Nomenclature: {stats.get('Nomenclature_support', 0):.0f} instances
    print(f"  Description:  {stats.get('Description_support', 0):.0f} instances"
    print(f"  Misc:         {stats.get('Misc_support', 0):.0f} instances")

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

    return important_f1
```

1.6.2 Custom Metrics

```
# Define your success criteria
def custom_success_metric(stats: Dict[str, float]) -> float:
    """
    Custom metric prioritizing Nomenclature detection.

    Weight Nomenclature more heavily than Description.
    """
    nom_f1 = stats.get('Nomenclature_f1', 0)
    desc_f1 = stats.get('Description_f1', 0)

    # Weighted average: Nomenclature 70%, Description 30%
    weighted_f1 = 0.7 * nom_f1 + 0.3 * desc_f1

    return weighted_f1

# Usage
important_metric = custom_success_metric(stats)
print(f"Custom Success Metric: {important_metric:.4f}")
```

1.7 Strategy 6: BiLSTM-CRF for Explicit Transition Modeling □□

For learning transition probabilities (e.g., P(Description | prev=Nomenclature)).

1.7.1 Architecture

```
def build_bilstm_crf_model(
    input_shape: Tuple[int, int],
    num_classes: int,
    hidden_size: int = 128,
    num_layers: int = 2,
    dropout: float = 0.3
) -> 'keras.Model':
    """
    BiLSTM-CRF: Classic sequence labeling architecture.

    The CRF layer learns:
    - Transition probabilities between labels
    - Invalid transitions (can be constrained)
    - Sequential dependencies

    Excellent for tasks where:
    - Label sequences have patterns (Description follows Nomenclature)
    - Some transitions are more likely than others
    - You want to enforce constraints

    Requires: tensorflow-addons
    Install: pip install tensorflow-addons

    Args:
        input_shape: (sequence_length, feature_dim)
        num_classes: Number of output classes
        hidden_size: LSTM hidden size
        num_layers: Number of BiLSTM layers
        dropout: Dropout rate

    Returns:
        Compiled Keras model with CRF output layer
    """
    from tensorflow_addons.layers import CRF
    from tensorflow_addons.text import crf_log_likelihood

    inputs = layers.Input(shape=input_shape)

    # BiLSTM layers for context encoding
    x = inputs
    for i in range(num_layers):
        x = layers.Bidirectional(
            layers.LSTM(hidden_size, return_sequences=True, dropout=dropout)
        )(x)

    # Dense layer for emission scores (not probabilities)
    # These are unnormalized scores for each class at each timestep
    emissions = layers.TimeDistributed(layers.Dense(num_classes))(x)

    # CRF layer learns transition matrix
    # The transition matrix T[i,j] represents the cost of transitioning from cla
    crf = CRF(num_classes)
    outputs = crf(emissions)

    model = keras.Model(inputs=inputs, outputs=outputs)

    # CRF has its own loss function (negative log-likelihood)
    model.compile(
        optimizer='adam',
        loss=crf.loss,
        metrics=[crf.accuracy]
    )

    return model
```

1.7.2 Benefits of CRF

The CRF layer will automatically learn: - **High probability**: Description following Nomenclature - **Low probability**: Misc immediately after Nomenclature (if rare in training data) - **Low probability**: Long sequences of Nomenclature (since they're typically isolated)

1.7.3 When to Use BiLSTM-CRF

- ✅ Sequential patterns are strong and consistent
- ✅ You want explicit transition modeling
- ✅ Simple class weighting isn't capturing sequence dependencies
- ✅ Your labels have grammatical structure (like NER tasks)

1.7.4 When NOT to Use

- ❌ Sequence patterns are weak or inconsistent
- ❌ Simple approaches (class weights) work well enough
- ❌ Additional complexity isn't justified by performance gains

1.8 Strategy 7: Increase Context Window □

Since Description clusters after Nomenclature, the model needs to see enough context.

```
# Current config
model_config = {
    'window_size': 10,  # May be too small to capture patterns
    'prediction_stride': 5,
}

# Recommended for sequence patterns
model_config = {
    'window_size': 20,  # Larger window to see Nom → Desc transitions
    'prediction_stride': 5,  # Keep stride smaller for fine-grained predictions
}

# Aggressive (if memory allows)
model_config = {
    'window_size': 30,  # Even more context
    'prediction_stride': 10,
}
```

Trade-offs: - ✅ Larger window = more context = better pattern detection - ❌ Larger window = more memory = slower training - ❌

Too large window = may include irrelevant context

1.9 Recommended Implementation Plan

1.9.1 Phase 1: Quick Wins (Start Here) □□□

Implement these first - highest impact, lowest effort:

```python
# 1. Add class weights to your model configuration
model_config = {
    'model_type': 'rnn',
    'window_size': 15,  # Increased from 10
    'prediction_stride': 5,
    'hidden_size': 128,
    'num_layers': 2,
    'dropout': 0.3,
    'epochs': 6,
    'batch_size': 32,
    'class_weights': {0: 100.0, 1: 10.0, 2: 0.1},  # NEW
}

# 2. Use proper evaluation metrics
stats = classifier.model.calculate_stats(predictions)
important_f1 = evaluate_imbalanced_classifier(stats)

# 3. Apply sequence-aware post-processing
predictions = apply_sequence_rules(predictions)
stats_adjusted = classifier.model.calculate_stats(predictions)

# Compare
print(f"Before rules: {important_f1:.4f}")
print(f"After rules:  {evaluate_imbalanced_classifier(stats_adjusted):.4f}")
```

Expected improvement: 10-30% boost in Nomenclature F1 score.

1.9.2 Phase 2: If Phase 1 Insufficient □□

```python
# 4. Try focal loss instead of weighted cross-entropy
model.compile(
    optimizer='adam',
    loss=focal_loss(gamma=2.0, alpha={0: 100, 1: 10, 2: 0.1}),
    metrics=['accuracy']
)

# 5. Oversample documents containing Nomenclature
train_data = oversample_documents_with_rare_labels(
    train_data,
    rare_label=0,
    replication_factor=5
)

# 6. Increase window size
model_config['window_size'] = 20
```

Expected improvement: Additional 5-15% boost.

1.9.3 Phase 3: Architecture Changes (If Needed)

```
# 7. Add CRF layer for explicit transition modeling
model = build_bilstm_crf_model(
    input_shape=(window_size, input_size),
    num_classes=3,
    hidden_size=128,
    num_layers=2
)

# 8. Two-stage classification
# Stage 1: Binary classifier (Important vs Misc)
# Stage 2: Ternary classifier on Important subset (Nom vs Desc vs Misc)
```

Expected improvement: Additional 5-10% boost, but added complexity.

1.10 Complete Example

```
# Step 1: 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,
    'class_weights': {0: 100.0, 1: 10.0, 2: 0.1},
    'verbosity': 1,
}

# Step 2: Train classifier
classifier = SkolClassifierV2(
    spark=spark,
    input_source='files',
    file_paths=annotated_files,
    auto_load_model=False,
    **model_config
)

results = classifier.fit()

# Step 3: Predict with probabilities (new predict() returns both)
predictions = classifier.predict(test_data)

# Step 4: Apply sequence rules
predictions = apply_sequence_rules(predictions)

# Step 5: Evaluate with focus on minority classes
stats = classifier.model.calculate_stats(predictions)

print("\n" + "="*70)
print("RESULTS WITH CLASS IMBALANCE STRATEGIES")
print("="*70)
print(f"Nomenclature F1: {stats['Nomenclature_f1']:.4f}  □")
print(f"Description F1:  {stats['Description_f1']:.4f}  □")
print(f"Misc F1:         {stats['Misc_f1']:.4f}      (don't care)")

important_f1 = (stats['Nomenclature_f1'] + stats['Description_f1']) / 2
print(f"\nPrimary Success Metric (Avg F1 for Important Classes): {important_f1:.
print("="*70)
```

1.11 Summary

1.11.1 Keep Your BiLSTM Architecture ✅

The BiLSTM is well-suited for your problem because: - Sequential patterns (Description after Nomenclature) are valuable - Variable-length sequences are handled naturally - Temporal context improves classification

1.11.2 Don't Switch To ❌

- **Random Forest / XGBoost**: Lose sequential information critical for your task
- **Simple Feedforward NN**: No temporal context
- **Transformers**: Overkill for your sequence lengths, more complexity than needed

1.11.3 Priority Order

1. **Class weights** (easiest, high impact)
2. **Evaluation metrics** (critical for measuring real performance)
3. **Sequence rules** (leverage domain knowledge)
4. **Focal loss** (if class weights insufficient)
5. **Document oversampling** (preserves sequence structure)
6. **Larger context window** (capture more patterns)
7. **BiLSTM-CRF** (if you need explicit transition modeling)

Start with items 1-3, measure improvement, then proceed if needed.

1.12 References

- **Focal Loss**: Lin et al. “Focal Loss for Dense Object Detection” (ICCV 2017)
- **BiLSTM-CRF**: Huang et al. “Bidirectional LSTM-CRF Models for Sequence Tagging” (2015)
- **Class Imbalance**: He & Garcia “Learning from Imbalanced Data” (IEEE TKDE 2009)