Contents

1 Class Weights for Traditional ML Models

1.1 Overview

Class weights support has been extended to traditional Spark ML models (Logistic Regression, Random Forest, and Gradient Boosted Trees). This allows all model types to use the `weight_strategy` parameter for handling class imbalance.

1.2 How It Works

1.2.1 RNN Models vs. Traditional ML Models

The implementation differs between model types:

RNN Models: - Use weighted categorical cross-entropy loss function - Class weights directly modify the loss during backpropagation - Implemented in TensorFlow/Keras

Traditional ML Models (Logistic Regression, Random Forest, GBT): - Use PySpark's `weightCol` parameter - Class weights are converted to **instance weights** during training - Each training sample gets a weight based on its class label - Higher weight = more importance during optimization

1.2.2 Instance Weight Conversion

When you specify class weights like:

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

The system automatically: 1. Adds an `instance_weight` column to the training data 2. Maps each label index to its corresponding weight: - Label index 0 (Nomenclature) → weight 100.0 - Label index 1 (Description) → weight 10.0 - Label index 2 (Misc) → weight 0.1 3. Passes this column to the classifier via `weightCol` parameter

1.3 Usage

1.3.1 Automatic with weight_strategy

The simplest way is to use the automatic `weight_strategy` parameter:

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

spark = SparkSession.builder.appName("Logistic with Weights").getOrCreate()

# Works with any model type!
classifier = SkolClassifierV2(
    spark=spark,
    input_source='files',
    file_paths=['data/annotated/*.ann'],
    model_type='logistic',  # or 'random_forest' or 'gradient_boosted'
```

```
    weight_strategy='inverse',  # Automatic weight calculation
    verbosity=1
)

results = classifier.fit()

# Output shows:
# [Classifier] Label Frequencies:
#   Misc                    9523 ( 87.2%)
#   Description             1234 ( 11.3%)
#   Nomenclature             165 (  1.5%)
#
# [Classifier] Applied 'inverse' weight strategy:
#   Nomenclature         100.00
#   Description           14.19
#   Misc                   0.10
#
# [LogisticRegressionSkolModel] Using class weights:
#   Nomenclature         100.00
#   Description           14.19
#   Misc                   0.10
```

1.3.2 Manual class_weights

You can also manually specify weights:

```
classifier = SkolClassifierV2(
    spark=spark,
    input_source='files',
    file_paths=['data/annotated/*.ann'],
    model_type='random_forest',
    class_weights={
        "Nomenclature": 100.0,
        "Description": 10.0,
        "Misc": 0.1
    },
    n_estimators=100,
    max_depth=10,
    verbosity=1
)

results = classifier.fit()
```

1.4 Supported Models

1.4.1 ✅ Logistic Regression

```
classifier = SkolClassifierV2(
    spark=spark,
    model_type='logistic',
    weight_strategy='inverse',
    maxIter=20,
    regParam=0.01
)
```

Notes: - Uses `LogisticRegression.weightCol` internally - Weights affect the optimization objective - Works with multinomial family

1.4.2 ✅ Random Forest

```
classifier = SkolClassifierV2(
    spark=spark,
    model_type='random_forest',
    weight_strategy='balanced',
    n_estimators=100,
    max_depth=10
)
```

Notes: - Uses `RandomForestClassifier.weightCol` internally - Weights affect split quality calculations - Weighted samples count more in decision tree splits

1.4.3 ✅ Gradient Boosted Trees

```
classifier = SkolClassifierV2(
    spark=spark,
    model_type='gradient_boosted',
    weight_strategy='aggressive',
    max_iter=50,
    max_depth=5
)
```

Notes: - Uses `GBTClassifier.weightCol` internally - Weights affect gradient boosting updates - Higher weight = stronger influence on tree construction

1.4.4 ✅ RNN/BiLSTM

```
classifier = SkolClassifierV2(
    spark=spark,
```

```
    model_type='rnn',
    weight_strategy='inverse',
    window_size=15,
    hidden_size=128
)
```

Notes: - Uses weighted categorical cross-entropy loss - Different implementation than traditional ML - See `docs/class_weights_usage.md` for RNN-specific details

1.5 Complete Example: Comparing Model Types

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

spark = SparkSession.builder.appName("Class Weights Comparison").getOrCreate()
redis_client = redis.Redis(host='localhost', port=6379, decode_responses=False)

annotated_files = ['data/annotated/*.ann']
weight_strategy = 'inverse'

# Common configuration
base_config = {
    'spark': spark,
    'input_source': 'files',
    'file_paths': annotated_files,
    'auto_load_model': False,
    'weight_strategy': weight_strategy,
    'verbosity': 1
}

# 1. Logistic Regression with class weights
print("\n=== Logistic Regression ===")
logistic_classifier = SkolClassifierV2(
    model_type='logistic',
    maxIter=20,
    regParam=0.01,
    **base_config
)
logistic_results = logistic_classifier.fit()
print(f"Nomenclature F1: {logistic_results['test_stats']['Nomenclature_f1']:.4f}

# 2. Random Forest with class weights
print("\n=== Random Forest ===")
rf_classifier = SkolClassifierV2(
```

```
    model_type='random_forest',
    n_estimators=100,
    max_depth=10,
    **base_config
)
rf_results = rf_classifier.fit()
print(f"Nomenclature F1: {rf_results['test_stats']['Nomenclature_f1']:.4f}")

# 3. Gradient Boosted Trees with class weights
print("\n=== Gradient Boosted Trees ===")
gbt_classifier = SkolClassifierV2(
    model_type='gradient_boosted',
    max_iter=50,
    max_depth=5,
    **base_config
)
gbt_results = gbt_classifier.fit()
print(f"Nomenclature F1: {gbt_results['test_stats']['Nomenclature_f1']:.4f}")

# 4. RNN with class weights
print("\n=== RNN/BiLSTM ===")
rnn_classifier = SkolClassifierV2(
    model_type='rnn',
    window_size=15,
    hidden_size=128,
    num_layers=2,
    epochs=6,
    batch_size=32,
    model_storage='redis',
    redis_client=redis_client,
    redis_key='rnn_weighted',
    **base_config
)
rnn_results = rnn_classifier.fit()
print(f"Nomenclature F1: {rnn_results['test_stats']['Nomenclature_f1']:.4f}")
```

1.6 Implementation Details

1.6.1 Code Changes

The following files were modified to support class weights in traditional ML models:

skol_classifier/model.py:

1. `TraditionalMLSkolModel.__init__()`:
   - Extracts `class_weights` from model_params

- Sets `weight_col = "instance_weight"` if weights provided

2. `TraditionalMLSkolModel._add_instance_weights()`:
   - Creates PySpark when-otherwise expression
   - Maps label indices to weights
   - Adds `instance_weight` column to DataFrame
3. `TraditionalMLSkolModel.fit()`:
   - Calls `_add_instance_weights()` before training
   - Prints applied weights if verbosity >= 1
4. `LogisticRegressionSkolModel.build_classifier()`:
   - Adds `weightCol` parameter if weights specified
5. `RandomForestSkolModel.build_classifier()`:
   - Adds `weightCol` parameter if weights specified
6. `GradientBoostedSkolModel.build_classifier()`:
   - Adds `weightCol` parameter if weights specified
7. `create_model()`:
   - Sets `model.labels` for all model types
   - Ensures labels flow through for weight mapping

1.6.2 Weight Column Creation

The instance weight column is created using PySpark's when-otherwise chain:

```
# For class_weights = {"Nomenclature": 100.0, "Description": 10.0, "Misc": 0.1}
# And labels = ["Nomenclature", "Description", "Misc"]

weight_expr = when(col("label_indexed") == 0, 100.0) \
    .when(col("label_indexed") == 1, 10.0) \
    .when(col("label_indexed") == 2, 0.1) \
    .otherwise(1.0)

train_data = train_data.withColumn("instance_weight", weight_expr)
```

This column is then passed to the classifier:

```
LogisticRegression(
    featuresCol="combined_idf",
    labelCol="label_indexed",
    weightCol="instance_weight",  # Added
    ...
)
```

1.7 Choosing Strategies by Model Type

1.7.1 Logistic Regression

- **Best strategy**: `'inverse'` or `'balanced'`

- **Why**: Logistic regression is sensitive to class distribution; balanced weights help
- **Avoid**: Very aggressive weights (>100) can cause numerical instability

1.7.2 Random Forest

- **Best strategy**: `'inverse'` or `'aggressive'`
- **Why**: Tree-based models are robust to high weights
- **Note**: Weights affect split quality, so minority classes get more focus

1.7.3 Gradient Boosted Trees

- **Best strategy**: `'balanced'` or `'inverse'`
- **Why**: Boosting already focuses on hard examples; moderate weights work well
- **Avoid**: Extremely high weights can lead to overfitting on minority class

1.7.4 RNN/BiLSTM

- **Best strategy**: `'aggressive'` or `'inverse'`
- **Why**: Deep learning benefits from strong weight gradients
- **Note**: Can handle very high weights (100+) without issues

1.8 Expected Improvements

With proper class weights, you should see:

Model Type	Nomenclature F1 Improvement	Notes
Logistic Regression	+5-15%	Moderate improvement
Random Forest	+10-20%	Good improvement due to split weig
Gradient Boosted Trees	+10-25%	Strong improvement from boosting
RNN/BiLSTM	+15-30%	Best improvement with aggressive w

1.9 Monitoring

With `verbosity >= 1`, all models will print:

```
[Classifier] Label Frequencies:
  Misc                    9523 ( 87.2%)
  Description             1234 ( 11.3%)
  Nomenclature             165 (  1.5%)
```

```
  Total                   10922 (100.0%)

[Classifier] Applied 'inverse' weight strategy:
  Nomenclature         100.00
  Description           14.19
  Misc                   0.10

[LogisticRegressionSkolModel] Using class weights:
  Nomenclature         100.00
  Description           14.19
  Misc                   0.10
```

1.10 Troubleshooting

1.10.1 Problem: Weights have no effect (traditional ML)

Symptom: Similar performance with and without weights

Solution: Check that: 1. Labels were provided to `fit()` (happens automatically in SkolClassifierV2) 2. `verbosity >= 1` shows "Using class weights" 3. Weight values are significantly different (not all ~1.0)

1.10.2 Problem: Numerical instability (Logistic Regression)

Symptom: NaN predictions or convergence warnings

Solution: Reduce weight magnitude:

```
# Instead of:
class_weights = {"Nomenclature": 1000.0, "Misc": 0.01}

# Try:
class_weights = {"Nomenclature": 50.0, "Misc": 0.5}
```

1.10.3 Problem: Overfitting on minority class

Symptom: Perfect recall, very low precision on minority class

Solution: Reduce minority class weight:

```
# Instead of 'aggressive':
weight_strategy = 'balanced'  # More conservative

# Or manually reduce:
class_weights = {"Nomenclature": 20.0, "Description": 5.0, "Misc": 0.5}
```

1.11 References

- PySpark ML documentation: weightCol parameter
- Weight strategy usage: `docs/weight_strategy_usage.md`
- RNN class weights: `docs/class_weights_usage.md`
- All strategies: `docs/class_imbalance_strategies.md`