Contents

1 Missing Methods to Add to ist769_skol.ipynb

This document contains all the methods that are missing from the notebook classes and need to be added.

1.1 Summary of Missing Methods

1. **SkolClassifierV2**: 3 methods (load_raw, _save_model_to_redis, _load_model_from_redis)
2. **CouchDBFile**: 3 functions (read_couchdb_partition, read_couchdb_rows, read_couchdb_files_from_connection)

1.2 1. SkolClassifierV2 Class - Missing Methods

1.2.1 Add to class SkolClassifierV2(SC):

1.2.1.1 Method 1: load_raw()

```
def load_raw(self) -> DataFrame:
    """
    Load raw (unannotated) data from configured input source.

    Returns:
        DataFrame with raw text data

    Raises:
        ValueError: If input_source is not properly configured
    """
    if self.input_source == 'files':
        return self._load_raw_from_files()
    elif self.input_source == 'couchdb':
        return self._load_raw_from_couchdb()
    else:
        raise ValueError(f"load_raw() not supported for input_source='{self.inpu
```

1.2.1.2 Method 2: _save_model_to_redis()

```
def _save_model_to_redis(self) -> None:
    """Save model to Redis using tar archive."""
    import json
    import tempfile
    import shutil
    import tarfile
    import io

    if self._model is None or self._feature_pipeline is None:
        raise ValueError("No model to save. Train a model first.")

    temp_dir = None
    try:
        # Create temporary directory
        temp_dir = tempfile.mkdtemp(prefix="skol_model_v2_")
        temp_path = Path(temp_dir)

        # Save feature pipeline
        pipeline_path = temp_path / "feature_pipeline"
        self._feature_pipeline.save(str(pipeline_path))

        # Save classifier model
        classifier_model = self._model.get_model()
        if classifier_model is None:
            raise ValueError("Classifier model not trained")
        classifier_path = temp_path / "classifier_model.h5"
        classifier_model.save(str(classifier_path))

        # Save metadata
        # For RNN models, save the actual model parameters (not the original par
        if self.model_type == 'rnn':
            actual_model_params = {
                'input_size': self._model.input_size,
                'hidden_size': self._model.hidden_size,
                'num_layers': self._model.num_layers,
                'num_classes': self._model.num_classes,
                'dropout': self._model.dropout,
                'window_size': self._model.window_size,
                'batch_size': self._model.batch_size,
                'epochs': self._model.epochs,
                'num_workers': self._model.num_workers,
                'verbosity': self._model.verbosity,
            }
            if hasattr(self._model, 'prediction_stride'):
                actual_model_params['prediction_stride'] = self._model.predictio
            if hasattr(self._model, 'prediction_batch_size'):
                actual_model_params['prediction_batch_size'] = self._model.predi
            if hasattr(self._model, 'name'):
                actual_model_params['name'] = self._model.name
        else:
            actual_model_params = self.model_params

        metadata = {
            'label_mapping': self._label_mapping,
            'config': {
                'line_level': self.line_level,
                'use_suffixes': self.use_suffixes,
                'min_doc_freq': self.min_doc_freq,
                'model_type': self.model_type,
                'model_params': actual_model_params
            },
            'version': '2.0'
        }
        metadata_path = temp_path / "metadata.json"
        with open(metadata_path, 'w') as f:
            json.dump(metadata, f, indent=2)

        # Create tar archive
        archive_buffer = io.BytesIO()
        with tarfile.open(fileobj=archive_buffer, mode='w:gz') as tar:
            tar.add(temp_path, arcname='.')

        # Save to Redis
        archive_data = archive_buffer.getvalue()
        self.redis_client.set(self.redis_key, archive_data)
        if self.redis_expire is not None:
            self.redis_client.expire(self.redis_key, self.redis_expire)

    finally:
        # Clean up temporary directory
        if temp_dir and Path(temp_dir).exists():
            shutil.rmtree(temp_dir)
```

1.2.1.3 Method 3: _load_model_from_redis()

```
def _load_model_from_redis(self) -> None:
    """Load model from Redis tar archive."""
    import json
    import tempfile
    import shutil
    import tarfile
    import io
    from pyspark.ml import PipelineModel

    serialized = self.redis_client.get(self.redis_key)
    if not serialized:
        raise ValueError(f"No model found in Redis with key: {self.redis_key}")

    temp_dir = None
    try:
        # Create temporary directory
        temp_dir = tempfile.mkdtemp(prefix="skol_model_load_v2_")
        temp_path = Path(temp_dir)

        # Extract tar archive
        archive_buffer = io.BytesIO(serialized)
        with tarfile.open(fileobj=archive_buffer, mode='r:gz') as tar:
            tar.extractall(temp_path)

        # Load metadata first to know model type
        metadata_path = temp_path / "metadata.json"
        with open(metadata_path, 'r') as f:
            metadata = json.load(f)

        self._label_mapping = metadata['label_mapping']
        self._reverse_label_mapping = {v: k for k, v in self._label_mapping.item
        model_type = metadata['config']['model_type']

        # Load feature pipeline
        pipeline_path = temp_path / "feature_pipeline"
        self._feature_pipeline = PipelineModel.load(str(pipeline_path))

        # Load classifier model (different approach for RNN vs traditional ML)
        classifier_path = temp_path / "classifier_model.h5"

        if model_type == 'rnn':
            # For RNN models, load the Keras .h5 file directly
            from tensorflow import keras
            keras_model = keras.models.load_model(str(classifier_path))
            classifier_model = keras_model  # This is the Keras model itself
        else:
            # For traditional ML models, load as PipelineModel
            classifier_model = PipelineModel.load(str(classifier_path))

        # Recreate the SkolModel wrapper using factory
        features_col = self._feature_extractor.get_features_col() if self._featu

        # Merge saved model params with any new params provided in constructor
        # New params override saved params for runtime-tunable parameters
        saved_params = metadata['config'].get('model_params', {})
        merged_params = saved_params.copy()

        # Override runtime-tunable parameters if provided
        if self.model_params:
            # These parameters can be changed without retraining
            runtime_tunable = {
                'prediction_batch_size',
                'prediction_stride',
                'num_workers',
                'verbosity',
                'batch_size'  # Training batch size, can be changed for future f
            }
            for param, value in self.model_params.items():
                if param in runtime_tunable:
                    merged_params[param] = value
                    if self.verbosity >= 2:
                        print(f"[Load Model] Overriding {param}: {saved_params.g

        self._model = create_model(
            model_type=model_type,
            features_col=features_col,
            label_col="label_indexed",
            **merged_params
        )
        self._model.set_model(classifier_model)
        self._model.set_labels(list(self._label_mapping.keys()))

    finally:
        # Clean up temporary directory
        if temp_dir and Path(temp_dir).exists():
            shutil.rmtree(temp_dir)
```

1.3 2. CouchDBFile Module - Missing Functions

1.3.1 Add these as module-level functions (not class methods):

1.3.1.1 Function 1: read_couchdb_partition()

```
def read_couchdb_partition(
    partition: Iterator[Row],
    db_name: str
) -> Iterator[Line]:
    """
    Read annotated files from CouchDB rows in a PySpark partition.

    This is the UDF alternative to read_files() for CouchDB-backed data.
    It processes rows containing CouchDB attachment content and yields
    Line objects that preserve database metadata.

    Args:
        partition: Iterator of PySpark Rows with columns:
            - doc_id: CouchDB document ID
            - attachment_name: Attachment filename
            - value: Text content from attachment
        db_name: Database name to store in metadata (ingest_db_name)

    Yields:
        Line objects with content and CouchDB metadata (doc_id, attachment_name,

    Example:
        >>> # In a PySpark context
        >>> from pyspark.sql.functions import col
        >>> from couchdb_file import read_couchdb_partition
        >>>
        >>> # Assume df has columns: doc_id, attachment_name, value
        >>> def process_partition(partition):
        ...     lines = read_couchdb_partition(partition, "mycobank")
        ...     # Process lines with finder.parse_annotated()
        ...     return lines
        >>>
        >>> result = df.rdd.mapPartitions(process_partition)
    """
    for row in partition:
        # Extract url from row if available
        human_url = getattr(row, 'human_url', None)

        # Create CouchDBFile object from row data
        file_obj = CouchDBFile(
            content=row.value,
            doc_id=row.doc_id,
            attachment_name=row.attachment_name,
            db_name=db_name,
            human_url=human_url
        )

        # Yield all lines from this file
        yield from file_obj.read_line()
```

1.3.1.2 Function 2: read_couchdb_rows()

```
def read_couchdb_rows(
    rows: List[Row],
    db_name: str
) -> Iterator[Line]:
    """
    Read annotated files from a list of CouchDB rows.

    This is a convenience function for non-distributed processing or testing.
    For production use with PySpark, use read_couchdb_partition().

    Args:
        rows: List of Rows with columns:
            - doc_id: CouchDB document ID
            - attachment_name: Attachment filename
            - value: Text content from attachment
        db_name: Database name to store in metadata

    Yields:
        Line objects with content and CouchDB metadata

    Example:
        >>> from couchdb_file import read_couchdb_rows
        >>>
        >>> # Collect rows from DataFrame
        >>> rows = df.collect()
        >>>
        >>> # Process all lines
        >>> lines = read_couchdb_rows(rows, "mycobank")
        >>> paragraphs = parse_annotated(lines)
        >>> taxa = group_paragraphs(paragraphs)
    """
    return read_couchdb_partition(iter(rows), db_name)
```

1.3.1.3 Function 3: read_couchdb_files_from_connection()

```python
def read_couchdb_files_from_connection(
    conn,  # CouchDBConnection
    spark,  # SparkSession
    db_name: str,
    pattern: str = "*.txt.ann"
) -> Iterator[Line]:
    """
    Load and read annotated files from CouchDB using CouchDBConnection.

    This function integrates CouchDBConnection.load_distributed() with
    read_couchdb_rows() to provide a complete pipeline from database to lines.

    Args:
        conn: CouchDBConnection instance
        spark: SparkSession
        db_name: Database name for metadata (ingest_db_name)
        pattern: Pattern for attachment names (default: "*.txt.ann")

    Returns:
        Iterator of Line objects with CouchDB metadata

    Example:
        >>> from skol_classifier.couchdb_io import CouchDBConnection
        >>> from couchdb_file import read_couchdb_files_from_connection
        >>> from finder import parse_annotated
        >>> from taxon import group_paragraphs
        >>>
        >>> # Connect to CouchDB
        >>> conn = CouchDBConnection(
        ...     "http://localhost:5984",
        ...     "mycobank",
        ...     "user",
        ...     "pass"
        ... )
        >>>
        >>> # Load files
        >>> lines = read_couchdb_files_from_connection(
        ...     conn, spark, "mycobank", "*.txt.ann"
        ... )
        >>>
        >>> # Parse and extract taxa
        >>> paragraphs = parse_annotated(lines)
        >>> taxa = group_paragraphs(paragraphs)
    """
    # Load data from CouchDB
    df = conn.load_distributed(spark, pattern)

    # Collect rows (for small datasets) or use in distributed context
    rows = df.collect()

    # Read lines with metadata
    return read_couchdb_rows(rows, db_name)
```

1.4 Instructions for Adding to Notebook

1. **For SkolClassifierV2 class**:
   - Find the cell containing `class SkolClassifierV2(SC):`
   - Add the three methods anywhere within the class definition
   - Recommended: Add them after the existing `_load_raw_from_couchdb()` and `_save_to_couchdb()` methods
2. **For CouchDBFile functions**:
   - These are **module-level functions**, not class methods
   - Find the cell containing `class CouchDBFile(CDBF):`
   - Add these functions **after** the class definition, at the module level
   - Make sure they are not indented (they should be at the same level as the class definition)
3. **Required imports**:
   - Ensure `from typing import Iterator, List` is imported
   - Ensure `from pyspark.sql import Row` is imported
   - For the Redis methods, ensure `from pathlib import Path` is imported

1.5 Note on init Methods

The following classes inherit their `__init__` from their parent classes and do not need explicit definition: - **TaxaJSONTranslator** inherits from TJT - **TaxonClusterer** inherits from TC - **SKOL_TAXA** inherits from STX - **EmbeddingsComputer** inherits from EC

These are working as designed through inheritance and do not need to be added.