Contents

1 Model Refactoring: Inheritance-Based Architecture 1
1.1 Summary 1
1.2 Changes 1
1.2.1 1. New Base Module: base_model.py 1
1.2.2 2. Updated model.py: Inheritance Hierarchy . . . 2
1.2.3 3. Updated rnn_model.py: RNNSkolModel Inheritance 2
1.2.4 4. Updated classifier_v2.py: Use Factory Function 2
1.3 Benefits 3
1.4 Files Modified 3
1.5 Backward Compatibility 3
1.6 Related Refactoring 3

1 Model Refactoring: Inheritance-Based Architecture

1.1 Summary

Refactored the model architecture to use proper inheritance instead of conditional logic based on `model_type` strings.

1.2 Changes

1.2.1 1. New Base Module: `base_model.py`

Created an abstract base class `SkolModel` that defines the interface all models must implement:

```
class SkolModel(ABC):
    @abstractmethod
    def fit(self, train_data: DataFrame, labels: Optional[List[str]] = None) ->
        pass

    @abstractmethod
    def predict(self, data: DataFrame) -> DataFrame:
        pass

    def predict_with_labels(self, data: DataFrame) -> DataFrame:
        # Common implementation for label conversion
        pass
```

1.2.2 2. Updated `model.py`: Inheritance Hierarchy

Created a proper inheritance hierarchy for traditional ML models:

```
SkolModel (ABC)
├── TraditionalMLSkolModel (ABC)
│   ├── LogisticRegressionSkolModel
│   ├── RandomForestSkolModel
│   └── GradientBoostedSkolModel
└── RNNSkolModel
```

TraditionalMLSkolModel: Provides common implementation for Spark ML models - Implements `fit()` using Pipeline pattern - Implements `predict()` using pipeline transformation - Defines abstract `build_classifier()` for subclasses

Concrete Subclasses: Each model type has its own class - `LogisticRegressionSkolModel`: Implements Logistic Regression - `RandomForestSkolModel`: Implements Random Forest - `GradientBoostedSkolModel`: Implements Gradient Boosted Trees

Factory Function: `create_model(model_type, ...)` instantiates the appropriate subclass

1.2.3 3. Updated `rnn_model.py`: RNNSkolModel Inheritance

Made `RNNSkolModel` inherit from `SkolModel`:

```python
class RNNSkolModel(SkolModel):
    def __init__(self, input_size, hidden_size, ...):
        super().__init__(features_col=features_col, label_col=label_col)
        # RNN-specific initialization

    def fit(self, train_data: DataFrame, labels: Optional[List[str]] = None):
        # RNN-specific training logic

    def predict(self, data: DataFrame) -> DataFrame:
        # RNN-specific prediction logic
```

1.2.4 4. Updated `classifier_v2.py`: Use Factory Function

Replaced direct `SkolModel()` instantiation with `create_model()` factory:

```python
# Before:
self._model = SkolModel(
    model_type=self.model_type,
    features_col=features_col,
    **self.model_params
```

```
)

# After:
self._model = create_model(
    model_type=self.model_type,
    features_col=features_col,
    label_col="label_indexed",
    **self.model_params
)
```

1.3 Benefits

1. **Eliminated Conditional Logic**: Removed all `if self.model_type == ...` checks
2. **Better Separation of Concerns**: Each model type has its own class
3. **Easier to Extend**: Adding new model types requires creating a new subclass, not modifying existing code
4. **Type Safety**: Factory function ensures correct model instantiation
5. **Polymorphism**: All models follow the same interface through inheritance
6. **Maintainability**: Model-specific logic is isolated in dedicated classes

1.4 Files Modified

- `skol_classifier/base_model.py` (NEW): Abstract base class
- `skol_classifier/model.py`: Traditional ML model subclasses and factory
- `skol_classifier/rnn_model.py`: Made RNNSkolModel inherit from SkolModel
- `skol_classifier/classifier_v2.py`: Use create_model() factory

1.5 Backward Compatibility

The external API remains unchanged. Users continue to specify `model_type='logistic'` etc., but internally the code now uses proper OOP principles.

1.6 Related Refactoring

See EVALUATION_REFACTORING.md for details on moving evaluation statistics to model methods.