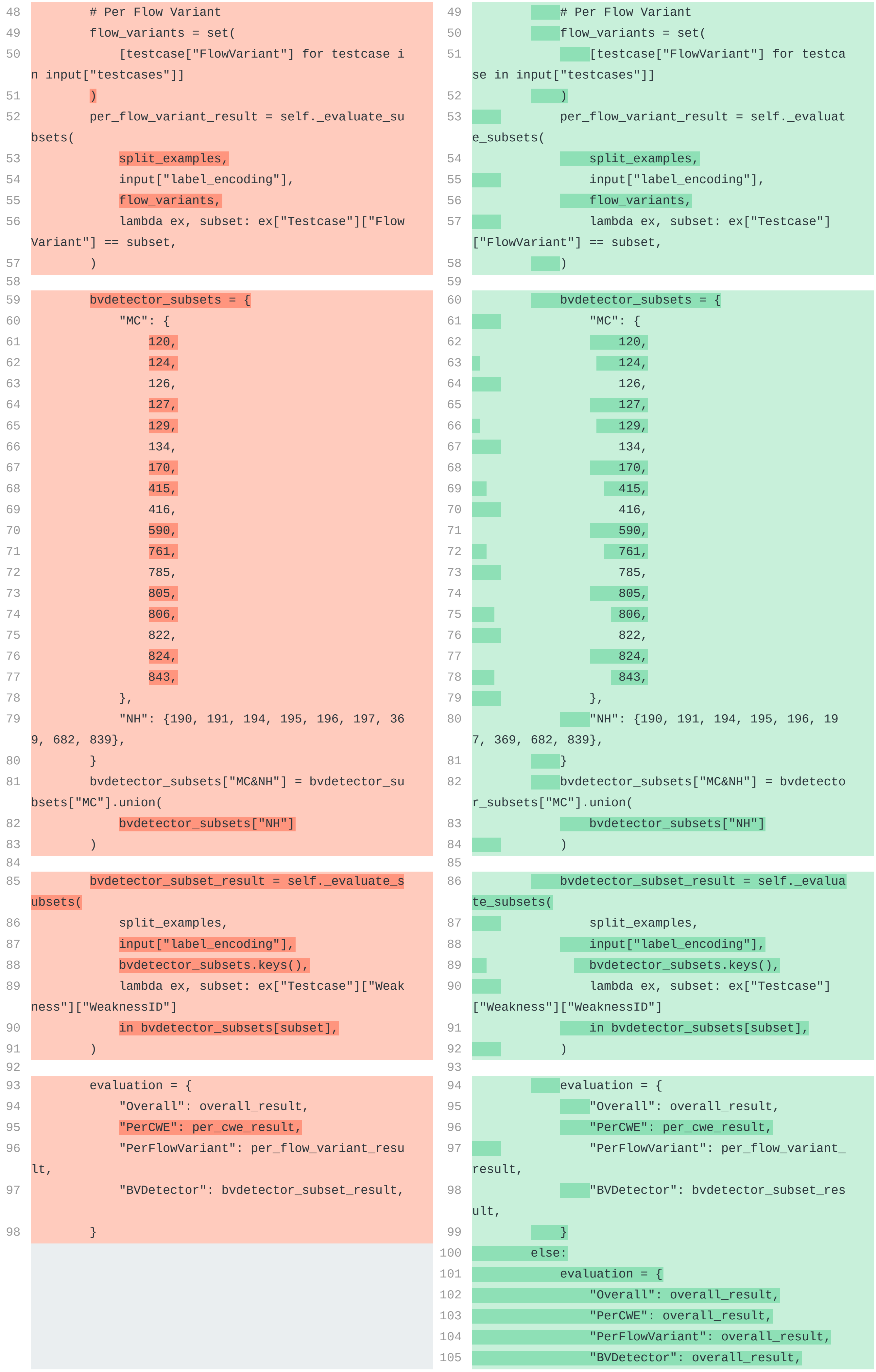
290 lines - 48 Removals

```
"""
    evaluation_pipeline_steps: Implementations of pipeline steps to calculate eval metrics and print tables.
"""

import logging

import numpy as np
import preprocessing
from pipeline_framework import PipelineState, PipelineStep


class EvaluatePredictionsPipelineStep(PipelineStep):
    step_key = "evaluate_predictions"

    def __init__(
        self, config: dict, state: PipelineState, logger: logging.Logger, split: str
    ):
        super().__init__(config, state, logger)

        self.split = split
        self.label_strategy = preprocessing.LabelStrategy[self.config["label_strategy"]]

        self.evaluation_yaml_path = (
            self.state.run_base_dir / f"evaluation_{self.split}.yaml"
        )

    def input_ready(self, input: dict) -> bool:
        return "examples" in input.keys() and any(
            ["Prediction" in example.keys() for example in input["examples"]]
        )

    def execute(self, input: dict, output: dict) -> None:
        # Collect relevant data
        split_examples = [ex for ex in input["examples"] if ex["Split"] == self.split]

        # Overall evaluation
        overall_result = self._evaluate_subset(split_examples, input["label_encoding"])

        # Per CWE
        weakness_ids = [cwe["WeaknessID"] for cwe in input["cwes"]]
        per_cwe_result = self._evaluate_subsets(
            split_examples,
            input["label_encoding"],
            set(weakness_ids),
            lambda ex, subset: ex["Testcase"]["Weakness"]["WeaknessID"] == subset,
        )

```

298 lines + 77 Additions

```
"""
    evaluation_pipeline_steps: Implementations of pipeline steps to calculate eval metrics and print tables.
"""

import logging

import numpy as np
import preprocessing
from pipeline_framework import PipelineState, PipelineStep


class EvaluatePredictionsPipelineStep(PipelineStep):
    step_key = "evaluate_predictions"

    def __init__(
        self, config: dict, state: PipelineState, logger: logging.Logger, split: str
    ):
        super().__init__(config, state, logger)

        self.split = split
        self.label_strategy = preprocessing.LabelStrategy[self.config["label_strategy"]]

        self.evaluation_yaml_path = (
            self.state.run_base_dir / f"evaluation_{self.split}.yaml"
        )

    def input_ready(self, input: dict) -> bool:
        return "examples" in input.keys() and any(
            ["Prediction" in example.keys() for example in input["examples"]]
        )

    def execute(self, input: dict, output: dict) -> None:
        # Collect relevant data
        split_examples = [ex for ex in input["examples"] if ex["Split"] == self.split]

        # Overall evaluation
        overall_result = self._evaluate_subset(split_examples, input["label_encoding"])

        if not self.state.malware:
            # Per CWE
            weakness_ids = [cwe["WeaknessID"] for cwe in input["cwes"]]
            per_cwe_result = self._evaluate_subsets(
                split_examples,
                input["label_encoding"],
                set(weakness_ids),
                lambda ex, subset: ex["Testcase"]["Weakness"]["WeaknessID"] == subset,
            )

```



```
        # Per Flow Variant
        flow_variants = set(
            [testcase["FlowVariant"] for testcase in input["testcases"]]
        )
        per_flow_variant_result = self._evaluate_subsets(
            split_examples,
            input["label_encoding"],
            flow_variants,
            lambda ex, subset: ex["Testcase"]["FlowVariant"] == subset,
        )

        bvdetector_subsets = {
            "MC": {
                120,
                124,
                126,
                127,
                129,
                134,
                170,
                415,
                416,
                590,
                761,
                785,
                805,
                806,
                822,
                824,
                843,
            },
            "NH": {190, 191, 194, 195, 196, 197, 369, 682, 839},
        }
        bvdetector_subsets["MC&NH"] = bvdetector_subsets["MC"].union(
            bvdetector_subsets["NH"]
        )

        bvdetector_subset_result = self._evaluate_subsets(
            split_examples,
            input["label_encoding"],
            bvdetector_subsets.keys(),
            lambda ex, subset: ex["Testcase"]["Weakness"]["WeaknessID"]
            in bvdetector_subsets[subset],
        )

        evaluation = {
            "Overall": overall_result,
            "PerCWE": per_cwe_result,
            "PerFlowVariant": per_flow_variant_result,
            "BVDetector": bvdetector_subset_result,
        }
```

```
            # Per Flow Variant
            flow_variants = set(
                [testcase["FlowVariant"] for testcase in input["testcases"]]
            )
            per_flow_variant_result = self._evaluate_subsets(
                split_examples,
                input["label_encoding"],
                flow_variants,
                lambda ex, subset: ex["Testcase"]["FlowVariant"] == subset,
            )

            bvdetector_subsets = {
                "MC": {
                    120,
                    124,
                    126,
                    127,
                    129,
                    134,
                    170,
                    415,
                    416,
                    590,
                    761,
                    785,
                    805,
                    806,
                    822,
                    824,
                    843,
                },
                "NH": {190, 191, 194, 195, 196, 197, 369, 682, 839},
            }
            bvdetector_subsets["MC&NH"] = bvdetector_subsets["MC"].union(
                bvdetector_subsets["NH"]
            )

            bvdetector_subset_result = self._evaluate_subsets(
                split_examples,
                input["label_encoding"],
                bvdetector_subsets.keys(),
                lambda ex, subset: ex["Testcase"]["Weakness"]["WeaknessID"]
                in bvdetector_subsets[subset],
            )

            evaluation = {
                "Overall": overall_result,
                "PerCWE": per_cwe_result,
                "PerFlowVariant": per_flow_variant_result,
                "BVDetector": bvdetector_subset_result,
            }
        else:
            evaluation = {
                "Overall": overall_result,
                "PerCWE": overall_result,
                "PerFlowVariant": overall_result,
                "BVDetector": overall_result,

        self.logger.info(f"Evaluation: {evaluation}")

        # Save to YAML
        with open(self.evaluation_yaml_path, "w") as evaluation_yaml:
            import yaml

            yaml.dump(evaluation, evaluation_yaml)

        if "evaluation" in output.keys():
            output["evaluation"][self.split] = evaluation
        else:
            output["evaluation"] = {self.split: evaluation}

    def _evaluate_subsets(
        self,
        examples: list[dict],
        label_encoding: dict,
        subsets: set[str],
        subset_filter,
    ):
        subsets_results = {}

        for subset in subsets:
            subset_examples = [ex for ex in examples if subset_filter(ex, subset)]
            subsets_results[subset] = self._evaluate_subset(
                subset_examples, label_encoding
            )

        return subsets_results

    def _evaluate_subset(self, examples: list[dict], label_encoding: dict):
        if self.label_strategy == preprocessing.LabelStrategy.BINARYCLASSIFICATION:
            return self._evaluate_subset_binary(examples, label_encoding)
        else:
            return self._evaluate_subset_multiclass(examples, label_encoding)

    def _evaluate_subset_binary(self, examples: list[dict], label_encoding: dict):
        confusion_matrix = self._calculate_confusion_matrix(examples, label_encoding)

        true_positives = float(confusion_matrix[1, 1])
        false_positives = float(confusion_matrix[0, 1])
        false_negatives = float(confusion_matrix[1, 0])
        true_negatives = float(confusion_matrix[0, 0])
        total_examples = (
            true_positives + false_positives + false_negatives + true_negatives
        )

        # Accuracy
        if total_examples > 0:
```

```
            }

        self.logger.info(f"Evaluation: {evaluation}")

        # Save to YAML
        with open(self.evaluation_yaml_path, "w") as evaluation_yaml:
            import yaml

            yaml.dump(evaluation, evaluation_yaml)

        if "evaluation" in output.keys():
            output["evaluation"][self.split] = evaluation
        else:
            output["evaluation"] = {self.split: evaluation}

    def _evaluate_subsets(
        self,
        examples: list[dict],
        label_encoding: dict,
        subsets: set[str],
        subset_filter,
    ):
        subsets_results = {}

        for subset in subsets:
            subset_examples = [ex for ex in examples if subset_filter(ex, subset)]
            subsets_results[subset] = self._evaluate_subset(
                subset_examples, label_encoding
            )

        return subsets_results

    def _evaluate_subset(self, examples: list[dict], label_encoding: dict):
        if self.label_strategy == preprocessing.LabelStrategy.BINARYCLASSIFICATION:
            return self._evaluate_subset_binary(examples, label_encoding)
        else:
            return self._evaluate_subset_multiclass(examples, label_encoding)

    def _evaluate_subset_binary(self, examples: list[dict], label_encoding: dict):
        confusion_matrix = self._calculate_confusion_matrix(examples, label_encoding)

        true_positives = float(confusion_matrix[1, 1])
        false_positives = float(confusion_matrix[0, 1])
        false_negatives = float(confusion_matrix[1, 0])
        true_negatives = float(confusion_matrix[0, 0])
        total_examples = (
            true_positives + false_positives + false_negatives + true_negatives
        )

        # Accuracy
        if total_examples > 0:
            accuracy = (true_positives + true_negatives) / (
                true_positives + false_positives + false_negatives + true_negatives
            )
        else:
            accuracy = None

        # Precision
        if (true_positives + false_positives) > 0:
            precision = true_positives / (true_positives + false_positives)
        else:
            precision = None

        # Recall, TPR, FNR
        if (true_positives + false_negatives) > 0:
            recall = true_positives / (true_positives + false_negatives)
            true_positive_rate = true_positives / (true_positives + false_negatives)
            false_negative_rate = false_negatives / (false_negatives + true_positives)
        else:
            recall = None
            true_positive_rate = None
            false_negative_rate = None

        # FPR, TNR
        if (true_negatives + false_positives) > 0:
            false_positive_rate = false_positives / (false_positives + true_negatives)
            true_negative_rate = true_negatives / (true_negatives + false_positives)
        else:
            false_positive_rate = None
            true_negative_rate = None

        # F1
        if precision is not None and recall is not None:
            if (precision + recall) > 0:
                f1 = 2 * precision * recall / (precision + recall)
            else:
                f1 = 0.0
        else:
            f1 = None

        return {
            "Accuracy": accuracy,
            "Precision": precision,
            "Recall": recall,
            "F1": f1,
            "TotalExamples": int(total_examples),
            "TotalPositives": int(true_positives + false_negatives),
            "TotalNegatives": int(true_negatives + false_positives),
            "TruePositiveRate": true_positive_rate,
            "FalsePositiveRate": false_positive_rate,
            "FalseNegativeRate": false_negative_rate,
            "TrueNegativeRate": true_negative_rate,
        }
```

```
            accuracy = (true_positives + true_negatives) / (
                true_positives + false_positives + false_negatives + true_negatives
            )
        else:
            accuracy = None

        # Precision
        if (true_positives + false_positives) > 0:
            precision = true_positives / (true_positives + false_positives)
        else:
            precision = None

        # Recall, TPR, FNR
        if (true_positives + false_negatives) > 0:
            recall = true_positives / (true_positives + false_negatives)
            true_positive_rate = true_positives / (true_positives + false_negatives)
            false_negative_rate = false_negatives / (false_negatives + true_positives)
        else:
            recall = None
            true_positive_rate = None
            false_negative_rate = None

        # FPR, TNR
        if (true_negatives + false_positives) > 0:
            false_positive_rate = false_positives / (false_positives + true_negatives)
            true_negative_rate = true_negatives / (true_negatives + false_positives)
        else:
            false_positive_rate = None
            true_negative_rate = None

        # F1
        if precision is not None and recall is not None:
            if (precision + recall) > 0:
                f1 = 2 * precision * recall / (precision + recall)
            else:
                f1 = 0.0
        else:
            f1 = None

        return {
            "Accuracy": accuracy,
            "Precision": precision,
            "Recall": recall,
            "F1": f1,
            "TotalExamples": int(total_examples),
            "TotalPositives": int(true_positives + false_negatives),
            "TotalNegatives": int(true_negatives + false_positives),
            "TruePositiveRate": true_positive_rate,
            "FalsePositiveRate": false_positive_rate,
            "FalseNegativeRate": false_negative_rate,
            "TrueNegativeRate": true_negative_rate,
        }
```

```python

    def _evaluate_subset_multiclass(self, examples: list[dict], label_encoding: dict):
        confusion_matrix = self._calculate_confusion_matrix(examples, label_encoding)

        total_examples = float(confusion_matrix.sum())
        if total_examples > 0:
            accuracy = float(confusion_matrix.diagonal().sum()) / total_examples
        else:
            return None

        return {"Accuracy": accuracy, "TotalExamples": total_examples}

    def _calculate_confusion_matrix(self, examples: list[dict], label_encoding: dict):
        label_count = len(label_encoding.keys())
        confusion_matrix = np.zeros((label_count, label_count), dtype=np.int64)

        for example in examples:
            prediction = example["Prediction"]
            label = example["Label"]

            label_idx = label_encoding[label]
            prediction_idx = label_encoding[prediction]

            confusion_matrix[label_idx, prediction_idx] += 1

        return confusion_matrix


class PrintEvalulationTablePipelineStep(PipelineStep):
    step_key = "print_evaluation_table"

    def __init__(
        self,
        config: dict,
        state: PipelineState,
        logger: logging.Logger,
        collection: str,
        split: str,
        keys: list[str],
    ):
        super().__init__(config, state, logger)

        self.collection = collection
        self.keys = keys
        self.split = split

    def input_ready(self, input: dict) -> bool:
        return (
            "evaluation" in input.keys()
            and self.split in input["evaluation"].keys()
            and self.collection in input["evaluation"][self.split].keys()
        )

    def execute(self, input: dict, output: dict) -> None:
```

```python

    def _evaluate_subset_multiclass(self, examples: list[dict], label_encoding: dict):
        confusion_matrix = self._calculate_confusion_matrix(examples, label_encoding)

        total_examples = float(confusion_matrix.sum())
        if total_examples > 0:
            accuracy = float(confusion_matrix.diagonal().sum()) / total_examples
        else:
            return None

        return {"Accuracy": accuracy, "TotalExamples": total_examples}

    def _calculate_confusion_matrix(self, examples: list[dict], label_encoding: dict):
        label_count = len(label_encoding.keys())
        confusion_matrix = np.zeros((label_count, label_count), dtype=np.int64)

        for example in examples:
            prediction = example["Prediction"]
            label = example["Label"]

            label_idx = label_encoding[label]
            prediction_idx = label_encoding[prediction]

            confusion_matrix[label_idx, prediction_idx] += 1

        return confusion_matrix


class PrintEvalulationTablePipelineStep(PipelineStep):
    step_key = "print_evaluation_table"

    def __init__(
        self,
        config: dict,
        state: PipelineState,
        logger: logging.Logger,
        collection: str,
        split: str,
        keys: list[str],
    ):
        super().__init__(config, state, logger)

        self.collection = collection
        self.keys = keys
        self.split = split

    def input_ready(self, input: dict) -> bool:
        return (
            "evaluation" in input.keys()
            and self.split in input["evaluation"].keys()
            and self.collection in input["evaluation"][self.split].keys()
        )

    def execute(self, input: dict, output: dict) -> None:
        evaluation_of_collection = input["evaluation"][self.split][self.collection]
        subsets = list(sorted(evaluation_of_collection.keys()))

        table_tuples = []

        # Compose table header
        table_tuples += [("Subset",) + tuple(self.keys)]
        table_tuples += [tuple(["-"] * (len(self.keys) + 1))]

        # Compose table rows
        for subset in subsets:
            row = [str(subset)]
            for key in self.keys:
                if isinstance(evaluation_of_collection[subset][key], float):
                    row += (f"{evaluation_of_collection[subset][key]:0.4f}",)
                else:
                    row += (str(evaluation_of_collection[subset][key]),)

            table_tuples.append(row)

        # Convert table to strings
        table_strings = []
        for table_tuple in table_tuples:
            table_strings.append(
                "| " + " | ".join([f"{str(el):15s}" for el in table_tuple]) + " |"
            )

        # Print table strings to logger
        self.logger.info(
            f"Evaluation table of split {self.split} (collection {self.collection}):\n"
            + "\n".join(table_strings)
        )


__all__ = ["EvaluatePredictionsPipelineStep", "PrintEvalulationTablePipelineStep"]
```

```
        evaluation_of_collection = input["evaluation"][self.split][self.collection]
        subsets = list(sorted(evaluation_of_collection.keys()))

        table_tuples = []

        # Compose table header
        table_tuples += [("Subset",) + tuple(self.keys)]
        table_tuples += [tuple(["-"] * (len(self.keys) + 1))]

        # Compose table rows
        for subset in subsets:
            row = [str(subset)]
            for key in self.keys:
                if isinstance(evaluation_of_collection[subset][key], float):
                    row += (f"{evaluation_of_collection[subset][key]:0.4f}",)
                else:
                    row += (str(evaluation_of_collection[subset][key]),)

            table_tuples.append(row)

        # Convert table to strings
        table_strings = []
        for table_tuple in table_tuples:
            table_strings.append(
                "| " + " | ".join([f"{str(el):15s}" for el in table_tuple]) + " |"
            )

        # Print table strings to logger
        self.logger.info(
            f"Evaluation table of split {self.split} (collection {self.collection}):\n"
            + "\n".join(table_strings)
        )


__all__ = ["EvaluatePredictionsPipelineStep", "PrintEvalulationTablePipelineStep"]
```