mlinspect: Lightweight Inspection of Native Machine Learning Pipelines

Stefan Grafberger
UvA

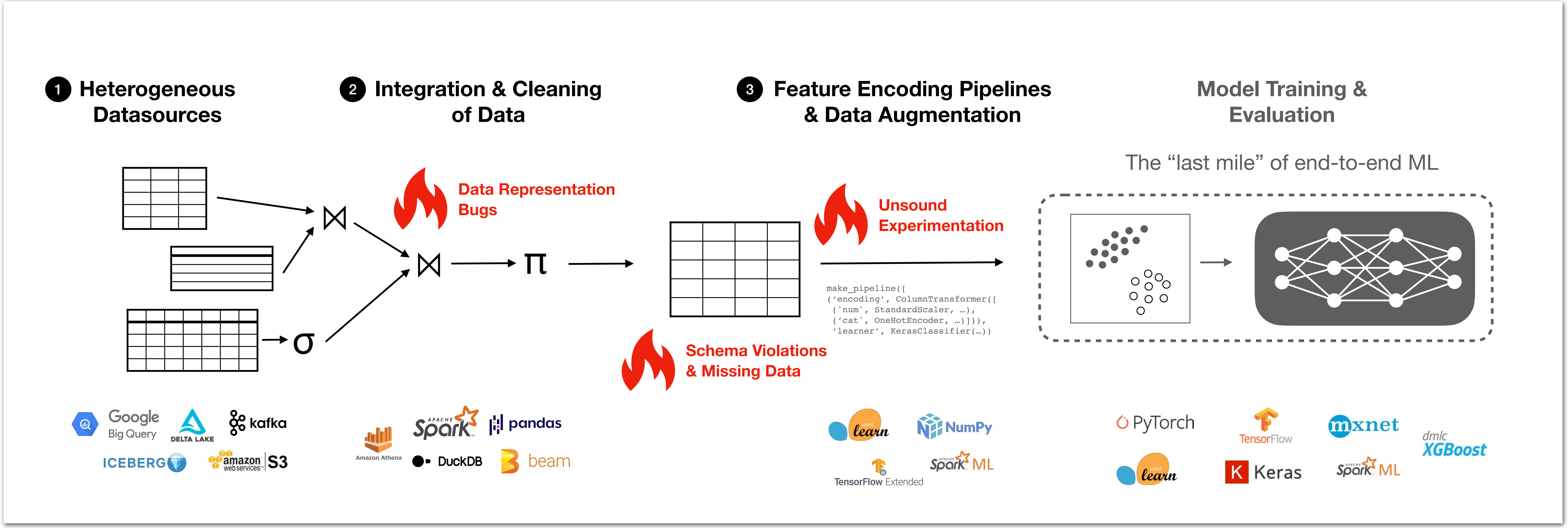
Joint work with

Paul Groth
UvA

Julia Stoyanovich
NYU

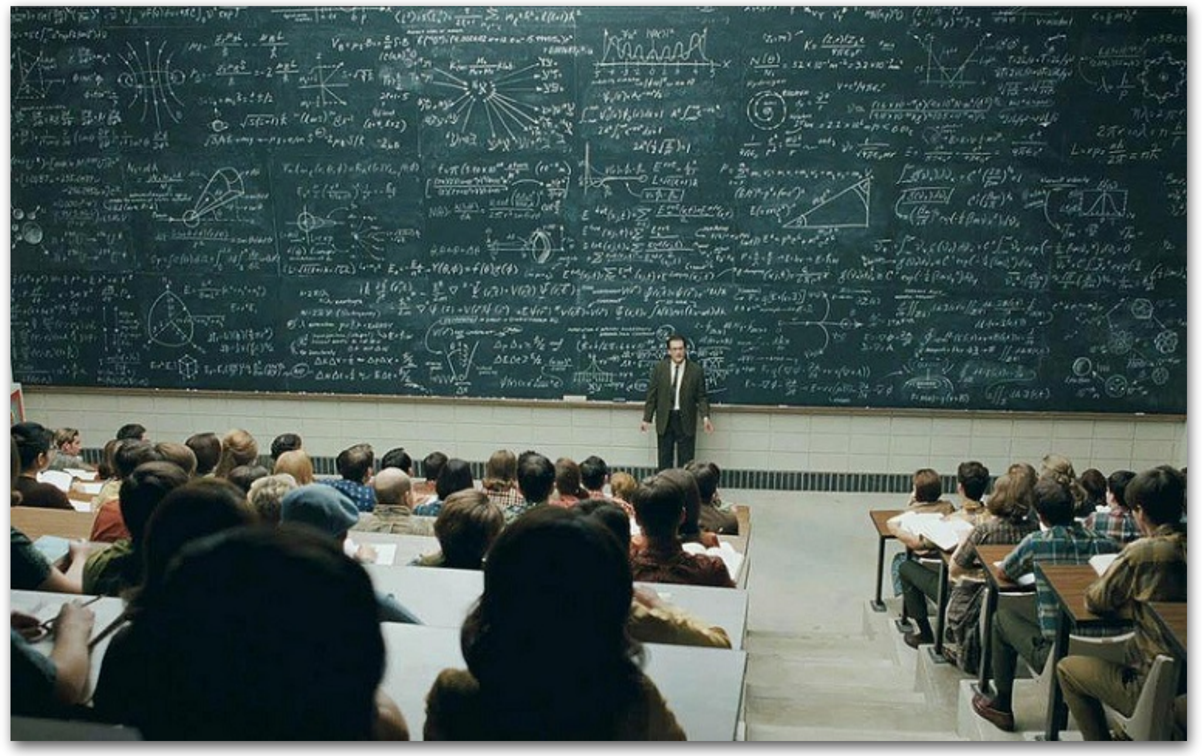
Sebastian Schelter
UvA

ML Pipelines in the Real World



ML in Research vs ML in Production

- **Lab conditions**
  - Mental model of working in a jupyter notebook
  - Dataset static, clean, well understood, often fits into memory
  - User has PhD in ML
- **Production conditions**
  - Data continuously produced, never clean
  - Data originates from many sources, not under control
  - Model training is only one piece of large, complex pipelines
  - Non-ML experts as end users / operators
- **Even experts make mistakes!**



https://chrisguillebeau.com/files/2016/11/Mathboard.jpg

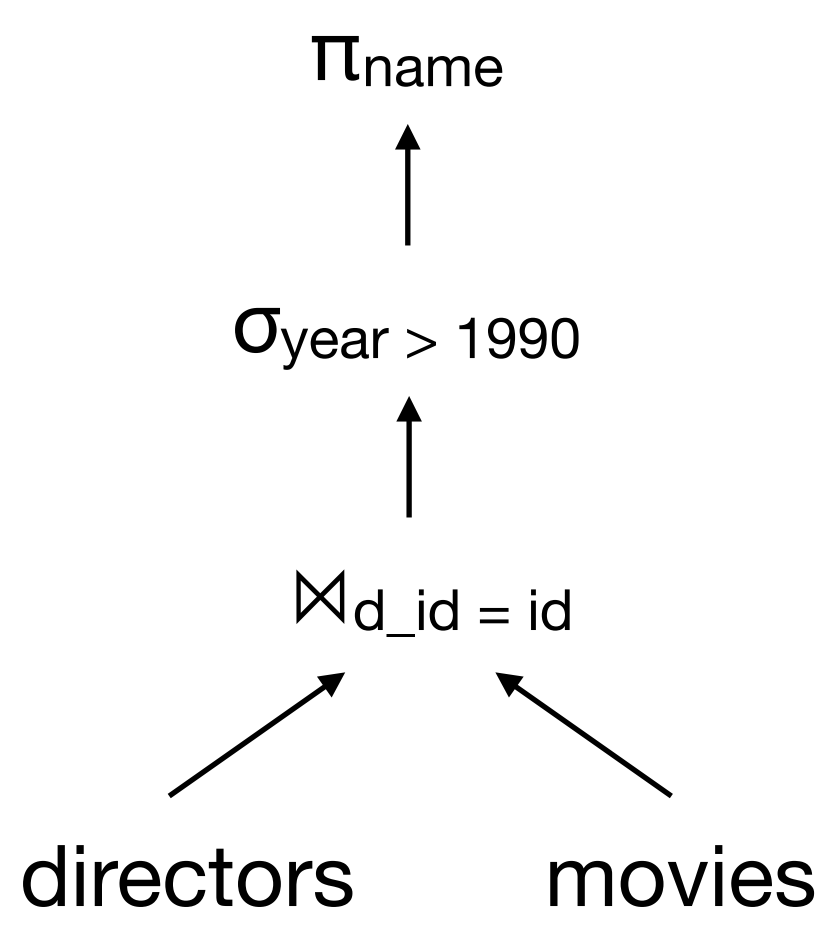
On Challenges in Machine Learning Model Management, IEEE Data Engineering Bulletin'19
FairPrep: Promoting Data to a First-Class Citizen in Studies on Fairness-Enhancing Interventions, EDBT'20

What Makes Inspection Difficult?

- **Relational DBMS:** Explicit data model (relations), computations (queries) expressed declaratively in relational algebra
- **Algebraic properties enable automatic inspection**: e.g. identifying all input records that contributed to a query result (why-provenance)
- **ML Pipelines: lack of unifying algebraic foundation** for data preprocessing, different technologies "glued together"

```
SELECT name
FROM directors
JOIN movies ON d_id = id
WHERE year > 1990
```

$\pi_{name}\ \sigma_{year\,>\,1990}\ (\text{directors} \bowtie_{d\_id\,=\,id} \text{movies})$



On Challenges in Machine Learning Model Management, Data Engineering Bulletin'19

The Way Forward

- First approach: **invent new holistic systems to regain control** -> **would require rewriting all existing code**
- Second approach: **manually annotating existing code** -> **does not happen in practice**
- **Our approach: retrofit inspection techniques into the existing DS landscape**
- Observation: **declarative specification of operations for preprocessing present in some popular ML libraries**:
    - Pandas mostly applies relational operations
    - Estimator / Transformer pipelines (scikit-learn / SparkML / Tensorflow Transform) offer nestable and composable way to declaratively specify feature transformations

Example

Potential issues in preprocessing pipeline:

1. Join might change proportions of groups in data
2. Column 'age_group' projected out, but required for fairness
3. Selection might change proportions of groups in data
4. Imputation might change proportions of groups in data
5. 'race' as a feature might be illegal!
6. Embedding vectors may not be available for rare names!

```
# load input data sources, join to single table
patients = pandas.read_csv(…)
histories = pandas.read_csv(…)
data = pandas.merge([patients, histories], on=['ssn'])

# compute mean complications per age group, append as column
complications = data.groupby('age_group')
 .agg(mean_complications=('complications','mean'))
data = data.merge(complications, on=['age_group'])

# Target variable: people with frequent complications
data['label'] = data['complications'] >
  1.2 * data['mean_complications']

# Project data to subset of attributes, filter by counties
data = data[['smoker', 'last_name', 'county',
             'num_children', 'race', 'income', 'label']]
data = data[data['county'].isin(counties_of_interest)]

# Define a nested feature encoding pipeline for the data
impute_and_encode = sklearn.Pipeline([
  (sklearn.SimpleImputer(strategy='most_frequent')),
  (sklearn.OneHotEncoder())])
featurisation = sklearn.ColumnTransformer(transformers=[
  (impute_and_encode, ['smoker', 'county', 'race']),
  (Word2VecTransformer(), 'last_name')
  (sklearn.StandardScaler(), ['num_children', 'income']])

# Define the training pipeline for the model
neural_net = sklearn.KerasClassifier(build_fn=create_model())
pipeline = sklearn.Pipeline([
  ('features', featurisation),
  ('learning_algorithm', neural_net)])

# Train-test split, model training and evaluation
train_data, test_data = train_test_split(data)
model = pipeline.fit(train_data, train_data.label)
print(model.score(test_data, test_data.label))
```

Can we find ways to **automatically hint data scientists at potentially problematic operations** in the preprocessing code of their ML pipelines?

Inspiration from software engineering, e.g. **code inspection in modern IDE's**

Example

Potential issues in preprocessing pipeline:

1. Join might change proportions of groups in data
2. Column 'age_group' projected out, but required for fairness
3. Selection might change proportions of groups in data
4. Imputation might change proportions of groups in data
5. 'race' as a feature might be illegal!
6. Embedding vectors may not be available for rare names!

Python script for preprocessing, written exclusively with native pandas and sklearn constructs

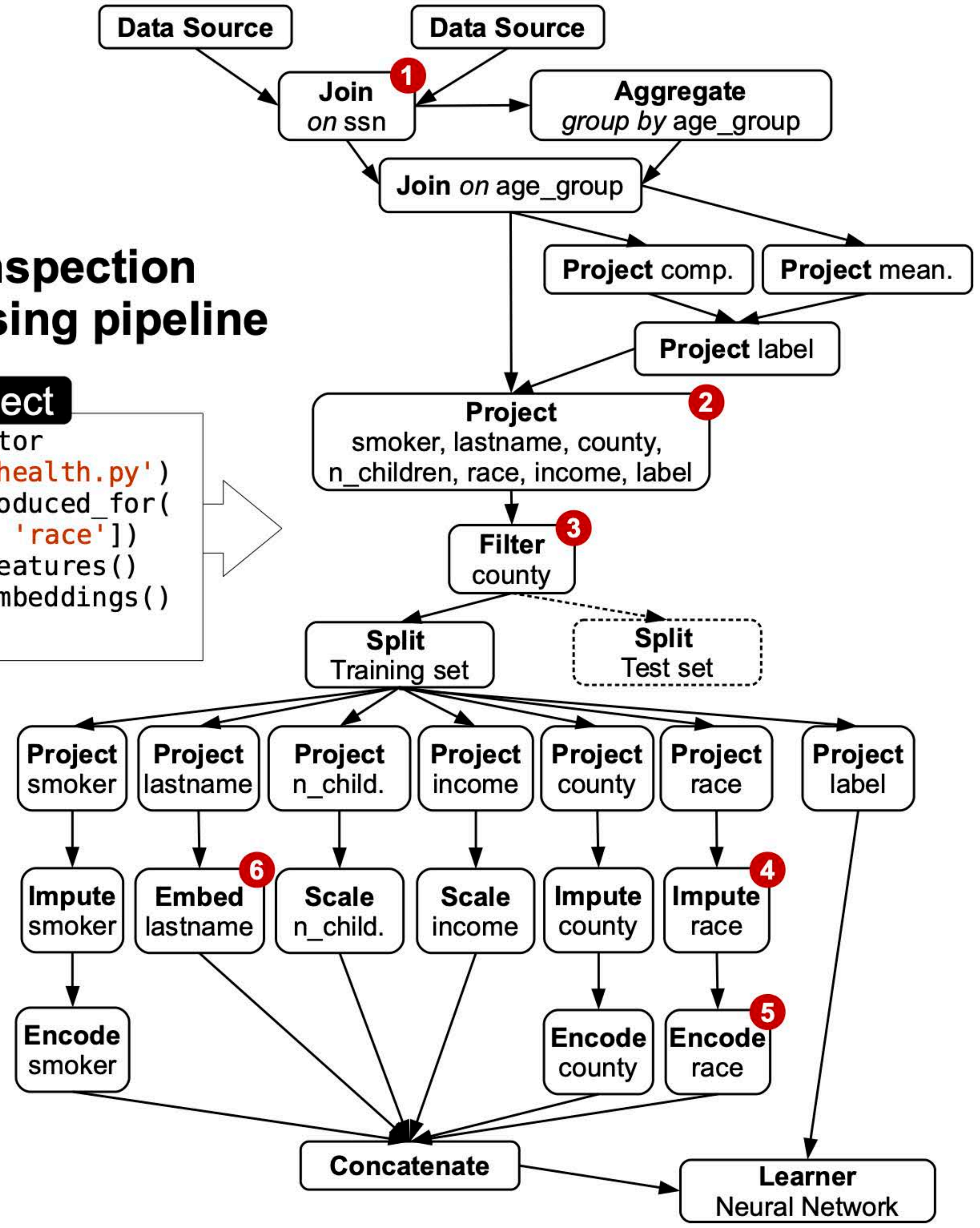
```
# load input data sources, join to single table
patients = pandas.read_csv(…)
histories = pandas.read_csv(…)
data = pandas.merge([patients, histories], on=['ssn'])

# compute mean complications per age group, append as column
complications = data.groupby('age_group')
 .agg(mean_complications=('complications','mean'))
data = data.merge(complications, on=['age_group'])

# Target variable: people with frequent complications
data['label'] = data['complications'] >
  1.2 * data['mean_complications']

# Project data to subset of attributes, filter by counties
data = data[['smoker', 'last_name', 'county',
             'num_children', 'race', 'income', 'label']]
data = data[data['county'].isin(counties_of_interest)]

# Define a nested feature encoding pipeline for the data
impute_and_encode = sklearn.Pipeline([
  (sklearn.SimpleImputer(strategy='most_frequent')),
  (sklearn.OneHotEncoder())])
featurisation = sklearn.ColumnTransformer(transformers=[
  (impute_and_encode, ['smoker', 'county', 'race']),
  (Word2VecTransformer(), 'last_name')
  (sklearn.StandardScaler(), ['num_children', 'income']])

# Define the training pipeline for the model
neural_net = sklearn.KerasClassifier(build_fn=create_model())
pipeline = sklearn.Pipeline([
  ('features', featurisation),
  ('learning_algorithm', neural_net)])

# Train-test split, model training and evaluation
train_data, test_data = train_test_split(data)
model = pipeline.fit(train_data, train_data.label)
print(model.score(test_data, test_data.label))
```

Declarative inspection of preprocessing pipeline

mlinspect

```
PipelineInspector
.on_pipeline('health.py')
 .no_bias_introduced_for(
 ['age_group', 'race'])
 .no_illegal_features()
 .no_missing_embeddings()
.verify()
```

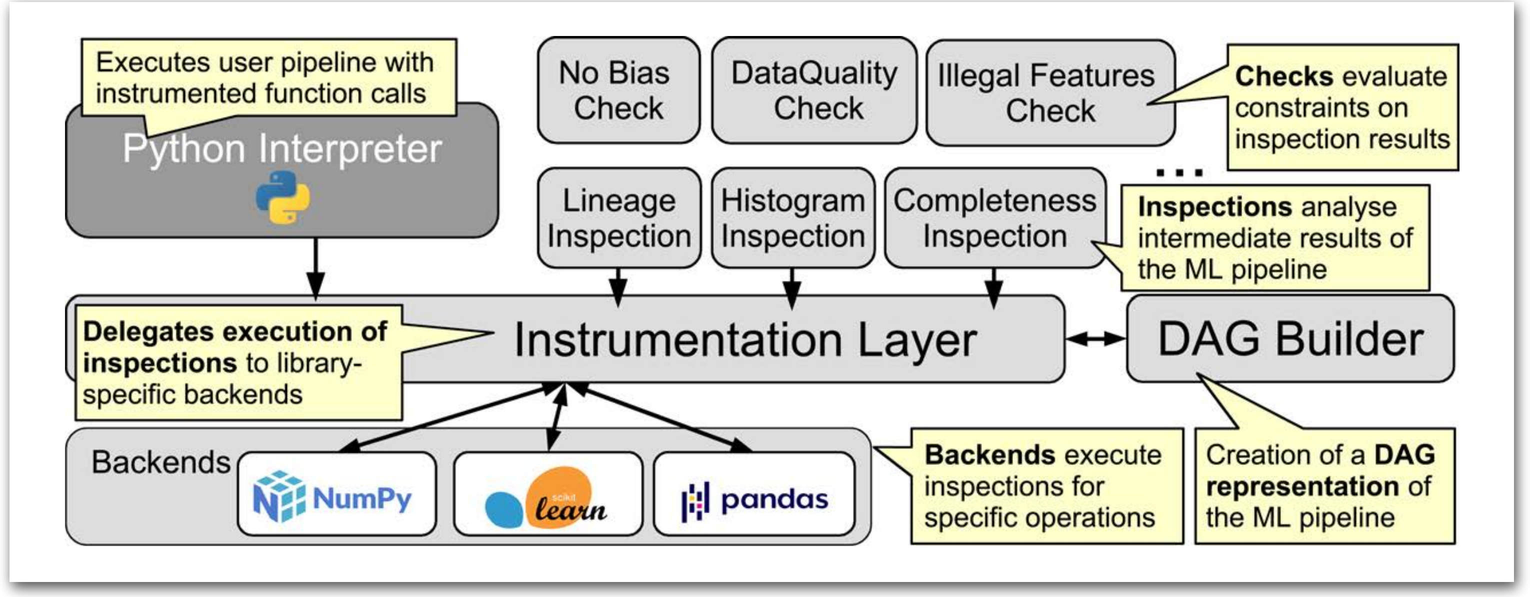
Corresponding dataflow DAG for instrumentation, extracted by *mlinspect*



mlinspect

- Library to instrument ML preprocessing code with custom inspections to **analyse a single pipeline execution and detect potential issues**
- Works with **“native” preprocessing pipelines** (no annotation / manual instrumentation required) in pandas / sklearn / keras
- **Representation of preprocessing operations based on dataflow graph**
- Allows users to **implement inspections as user-defined functions** which are **automatically applied to the inputs and outputs of certain operations**

```
PipelineInspector
 .on_pipeline_from_py_file('healthcare.py')
  .expect_no_bias_introduced_for(['age_group', 'race'])
  .expect_no_use_of_illegal_features()
  .expect_no_missing_embeddings()
 .verify()
```



Data Distribution Debugging in Machine Learning Pipelines, VLDBJ’22

Inspections & Checks

- The central entry point of mlinspect is the **PipelineInspector**
  - There, you can add **inspections** and **checks**
- **Inspections:**
  - Visit each operator in the extracted DAG to analyse the data flowing through it
  - Can also annotate individual tuples, to track how they flow through the pipeline
  - These annotations are only visible for mlinspect
- **Checks:**
  - Check constraints on the extracted DAG and, when needed, the results of inspections

```
from mlinspect import PipelineInspector
from mlinspect.inspections import MaterializeFirstOutputRows
from mlinspect.checks import NoBiasIntroducedFor

IPYNB_PATH = ...

inspector_result = PipelineInspector\
        .on_pipeline_from_ipynb_file(IPYNB_PATH)\
        .add_required_inspection(MaterializeFirstOutputRows(5))\
        .add_check(NoBiasIntroducedFor(['race']))\
        .execute()

extracted_dag = inspector_result.dag
dag_node_to_inspection_results = inspector_result.dag_node_to_inspection_results
check_to_check_results = inspector_result.check_to_check_results
```

Inspection: MaterializeFirstOutputRows

- The most simple **inspection** in mlinspect
- **Analyzes the data** flowing through each operator in your ML pipeline, and **materializes** the first *n* rows of each
- This is similar to how data scientists might try to debug their code with *print* statements, just to see what data looks like at different pipeline stages
- For pipelines using lots of data, inspections **should not materialize all intermediate data**!

```
data = pd.merge([patient, cost],
                on="ssn")
data = data[["smoke", "cost"]]
```

ssn	smoke
123	Y
456	N
789	Y
…	…

ssn	cost
123	100
789	200
…	…

ssn	smoke	cost
123	Y	100
789	Y	200
…	…	…

smoke	cost
Y	100
Y	200
…	…

Inspection: RowLineage

- This inspection uses **annotation propagation** to track individual tuples through the ML pipeline

- Operators like filters, joins, and sorting can make tracking tuples manually difficult

- Example: you encounter an unexpected *null*-value somewhere in your pipeline. Where does the *null*-value come from? What are the corresponding rows in the initial input tables?

```
data = pd.merge([patient, cost],
                 on=“ssn”)
data = data[[“smoke”, “cost”]]
```

ssn	smoke	
123	null	{(1,1)}
456	N	{(1,2)}
789	Y	{(1,3)}
…	…	

ssn	cost	
123	100	{(2,1)}
789	200	{(2,2)}
…	…	

ssn	smoke	cost	
123	null	100	{(1,1),(2,1)}
789	Y	200	{(1,3),(2,2)}
…	…	…	

smoke	cost	
null	100	{(1,1),(2,1)}
Y	200	{(1,3),(2,2)}
…	…	

HistogramForColumns & NoBiasIntroducedFor

- The *HistogramForColumns* inspection uses **annotation propagation** to track group memberships through the ML pipeline and materializes histograms of the groups
- The check *NoBiasIntroducedFor* checks for sudden distribution shifts by looking at the histograms before and after each operator
- Next to *HistogramForColumns,* there is also *IntersectionalHistogramForColumns*
- **Problem**: when should a distribution shift trigger a warning?

```
data = data[data.county = “CountyA”]
data = data[[“county”]]
```

age_group	county
< 60	CountyA
< 60	CountyA
< 20	CountyA
< 60	CountyB
< 20	CountyB
< 20	CountyB

age_group: 50% vs 50%

age_group	county
< 60	CountyA
< 60	CountyA
< 20	CountyA

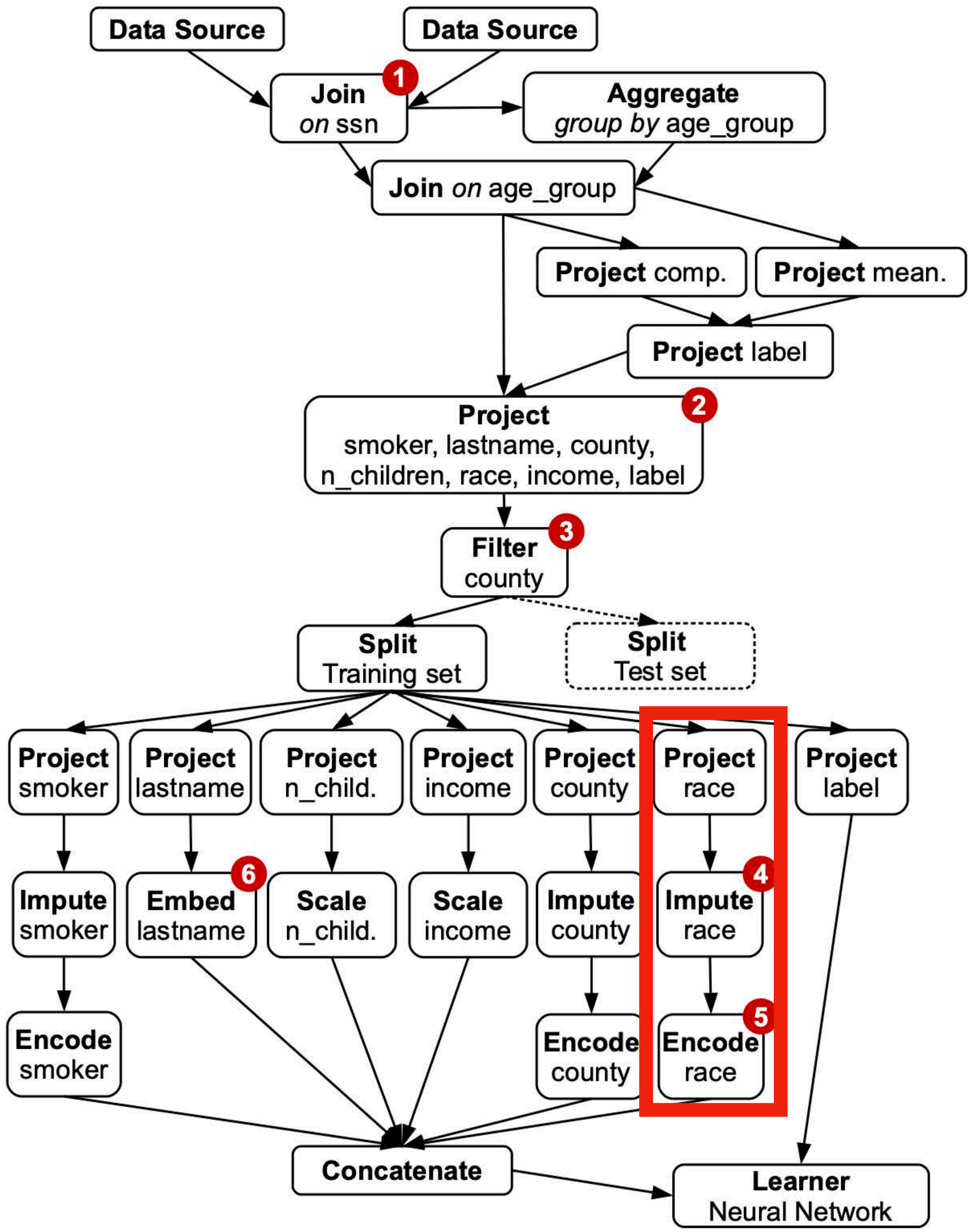
age_group: 66% vs 33%

county	
CountyA	< 60
CountyA	< 60
CountyA	< 20

age_group: 66% vs 33%

Check: NoIllegalFeatures

- This check only looks at the extracted DAG to see if columns with certain names are used as input for an ML model
- As with *NoBiasIntroducedFor*: this detection based only on comparing column names with pre-defined and user-defined lists is **no guarantee** that all features are okay to be used!
- However, this can help with spotting potential issues easier



Data Quality Inspections

- mlinspect also offers exemplary inspections for data quality checking: *CompletenessOfColumns* and *CountDistinctOfColumns*

- The completeness of a column is the fraction of non-null values in it

- On top of these inspections, it is again possible to build checks

```
data = pd.merge([patient, cost],
                on="ssn", how="right")
```

ssn	cost
123	100
456	200
789	150

smoke completeness: N/A

ssn	smoke
123	Y
789	N

smoke completeness: 100%

ssn	cost	smoke
123	100	Y
456	200	null
789	150	N

smoke completeness: 66%

Demo

https://surfdrive.surf.nl/files/index.php/s/ybriyzsdc6vcd2w 1:06-4:00

Inspection Implementation

- Experienced users can also implement their own inspections and checks
- **Implementation of inspections via for-comprehensions** on iterators
- **Efficient execution with loop fusion** (“banana-split law”)
- **Runtime overhead linear in the number of input and output records** as long as the row annotations have a fixed size limit

```
# Abstract base class for all inspections
class Inspection(metaclass=abc.ABCMeta):
  # Inspect intermediate data at a DAG operator, based on operator
  #  information (op_context), and an iterator over annotated
  #  input rows with the corresponding output rows (row_iterator);
  # Return computed annotations for output rows
  def visit_op(self, op_context, row_iterator) -> Iterable
  # Persist inspection result for the current DAG node
  def op_annotation_after_visit(self)
```

```
def visit_op(self, op_context,  row_iterator) -> Iterable
  for row in row_iterator:
    annotation = annotate_and_update_state(self, row)
    yield annotation
```

Inspection Execution

1. **Preparation**: Determination of a minimal required set of inspections based on the inspections and checks specified by the user.

2. **Instrumentation**: Instrumentation of function calls of the AST of the user program, monkey patching.

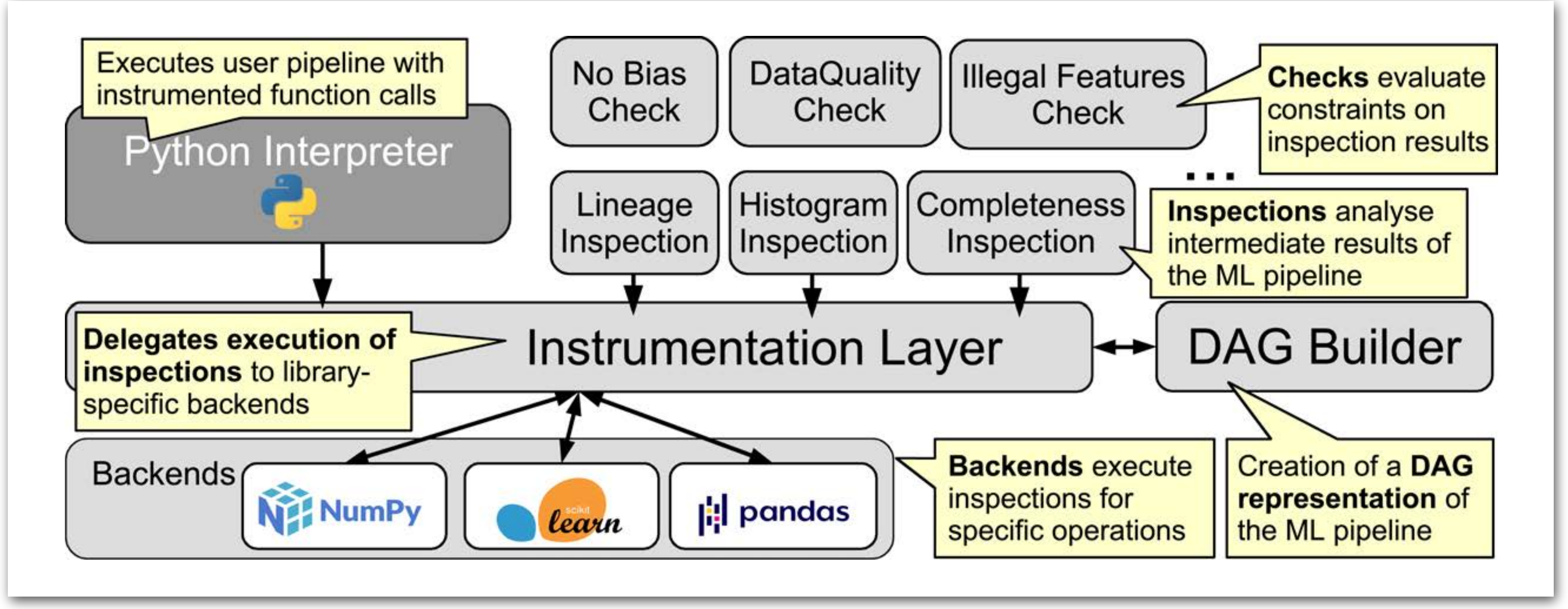
3. **Execution of the instrumented program**: Delegation of the execution of inspections to library-specific backends; joint execution with pipeline operations; creation of the dataflow DAG.

4. **Results**: Evaluation of checks using the DAG and the inspection results.

```
from mlinspect.instrumentation
 import monkey_patch, undo_monkey_patch
monkey_patch()
# ...original user code...
undo_monkey_patch()
```

```
obj.a_func("arg0", "arg1", my_arg="arg2")
```

```
obj.a_func("arg0", "arg1", my_arg="arg2",
            **set_code_reference(0,0,0,42))
```

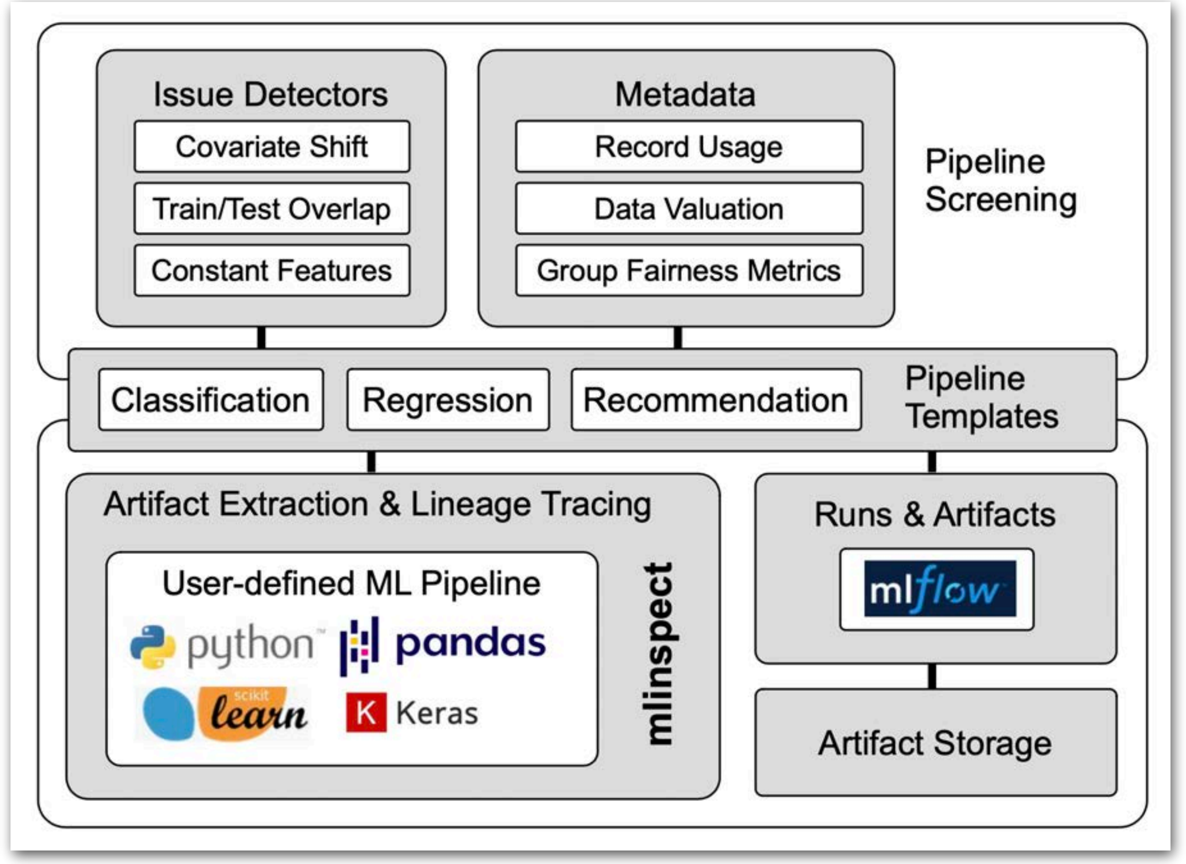


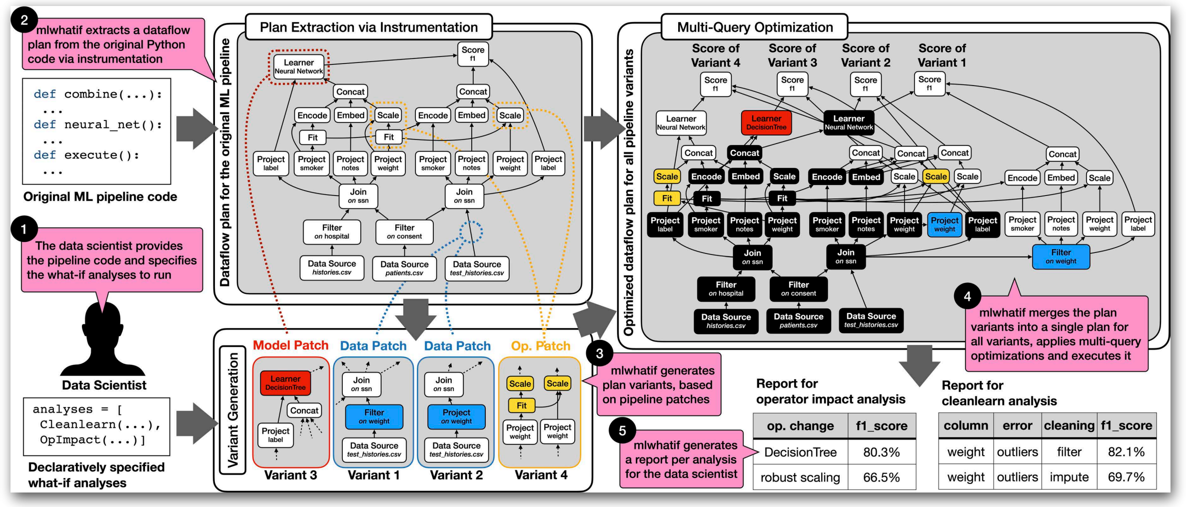
Monkey Patching

```
@gorilla.patches(sklearn.preprocessing)
class SklearnPreprocessingPatching:
  @gorilla.name('label_binarize')
  @gorilla.settings(allow_hit=True)
  def execute_label_binarize(*args, **kwargs):
    original = gorilla.get_original_attribute(sklearn.preprocessing, 'label_binarize')
    # Patched function
    def patched(...):
      function_info = FunctionInfo('sklearn.preprocessing._label', 'label_binarize')
      # Operator mapping for DAG
      op_ctx = OperatorContext(OperatorType.PROJECTION_MODIFY, function_info)
      parent_info = get_parent_node_info(args[0], ...)
      # Initiate inspection execution via backend
      input_df = SklearnBackend.before_call(op_ctx, [parent_info])
      # Execute original function
      result = original(input_df, *args[1:], **kwargs)
      # Finalize inspection execution via backend
      backend_result = SklearnBackend.after_call(op_ctx, input_df, result)
      # Append DAG node with inspection result
      add_new_operator_node_to_dag(DagNode(...), [parent_info], backend_result)
      # Return original result
      return backend_result.updated_result_df
    return execute(original, patched, *args, **kwargs)
```

Ongoing and Future Work

- **Moving the execution** of inspections **into more efficient runtime systems** like DuckDB
- Use mlinspect as runtime system to **enable different use cases**, e.g., automated screening of ML pipelines during CI pipelines (**ArgusEyes**)
- Assisting with **more advanced ML pipeline analysis that requires pipeline rewriting** and cannot be done by just observing a single execution of a given ML pipeline (**mlwhatif**)





Proactively Screening Machine Learning Pipelines with ArgusEyes, SIGMOD'23 (demo)

Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines, SIGMOD'23

Data-Centric What-If Analysis for ML Pipelines

- ML pipelines are **often brittle with respect to input data**

- Data scientists are interested in different **what-if analyses** for their pipelines, e.g.,

  - **What-if** there are data quality problems?

  - **What-if** I used different preprocessing?

- Currently, data scientists have to implement this **manually**, which is tedious and error-prone

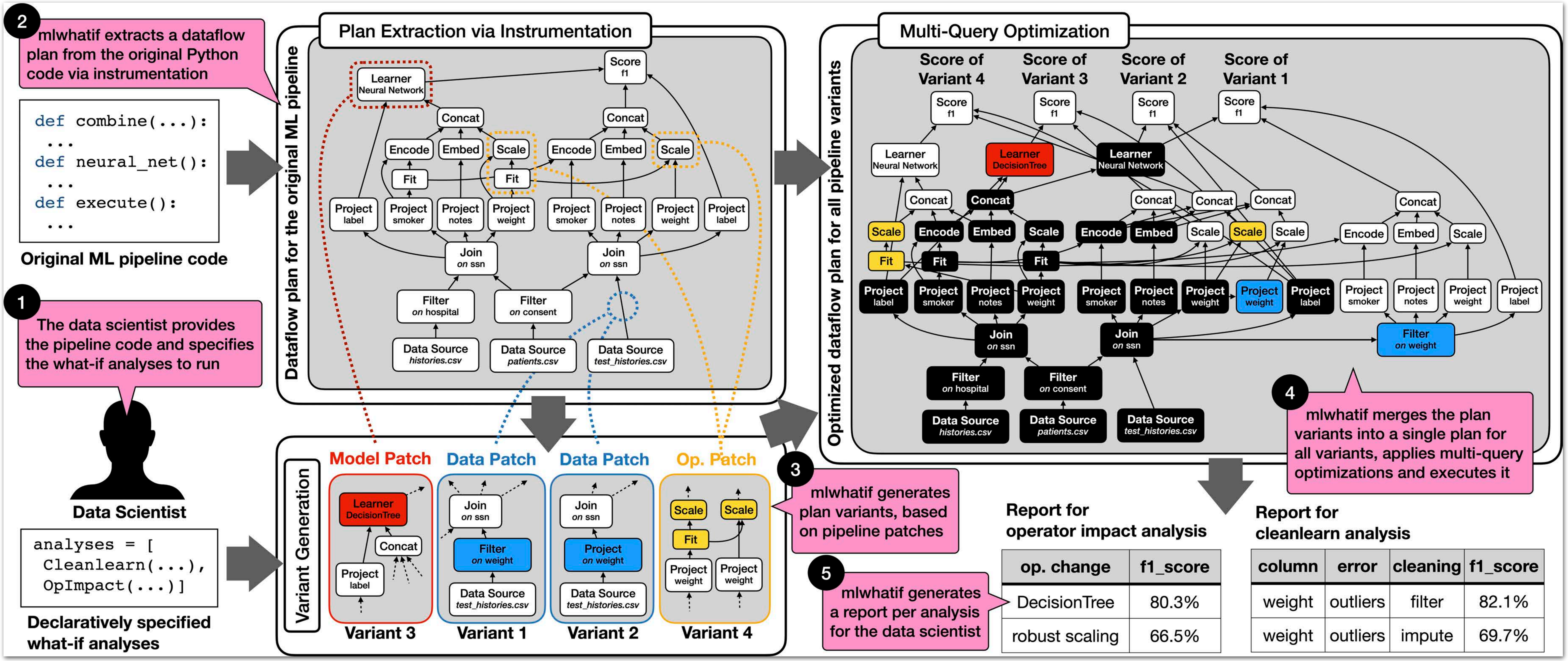
- **Goal: Allow declarative what-if analysis** by automatically rewriting extracted pipeline DAGs

```
from mlwhatif.analysis import Cleanlearn, OperatorImpact
analyses = [ # Declarative definition of what-if analyses to run
  Cleanlearn(column='weight', error=Error.OUTLIER,
             cleanings=[Clean.FILTER, Clean.IMPUTE]),
  OperatorImpact(robust_scaling=True, alternative_model=...)]
# Execution of what-if analyses on a given pipeline
report_with_scores = mlwhatif.execute_whatif('healthcare.py', analyses)
print(report_with_scores)
```

analysis	column	error_detector	cleaning	f1_score
cleanlearn	weight	outliers	filter	0.821
cleanlearn	weight	outliers	impute_const	0.697

Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines, SIGMOD'23

mlwhatif: Data-Centric What-If Analysis




```
def combine(...):
  ...
def neural_net():
  ...
def execute():
  ...
```

```
analyses = [
 Cleanlearn(...),
 OpImpact(...)]
```

Report for operator impact analysis

op. change	f1_score
DecisionTree	80.3%
robust scaling	66.5%

Report for cleanlearn analysis

column	error	cleaning	f1_score
weight	outliers	filter	82.1%
weight	outliers	impute	69.7%

Thanks!

- **Summary**
  - **mlinspect** allows inspecting a single execution of a given input ML pipeline: **https://github.com/stefan-grafberger/mlinspect**
  - **mlwhatif** allows declarative what-if analysis **https://github.com/stefan-grafberger/mlwhatif**
  - **Limitation:** Our approach relies on“declaratively” written ML pipelines, where we can identify the semantics of the operations
- For more about my research, visit **https://stefan-grafberger.com/**

```python
from mlinspect import PipelineInspector
from mlinspect.inspections import MaterializeFirstOutputRows
from mlinspect.checks import NoBiasIntroducedFor

IPYNB_PATH = ...

inspector_result = PipelineInspector\
        .on_pipeline_from_ipynb_file(IPYNB_PATH)\
        .add_required_inspection(MaterializeFirstOutputRows(5))\
        .add_check(NoBiasIntroducedFor(['race']))\
        .execute()

extracted_dag = inspector_result.dag
dag_node_to_inspection_results = inspector_result.dag_node_to_inspection_results
check_to_check_results = inspector_result.check_to_check_results
```

Banana split law

- Operations like calculating the sum and the length of numerical data can be done using folds. All folds can be combined into a single fold.
- sumlength :: [Int] → (Int,Int)
  sumlength xs = (sum xs, length xs)
- sumlength = fold (λn (x, y) → (n + x, 1 + y)) (0, 0)
- "The strange name of this property derives from the fact that the *fold* operator is sometimes written using brackets (| |) that resemble bananas, and the pairing operator is sometimes called split."

A tutorial on the universality and expressiveness of fold, Graham Hutton, 1999