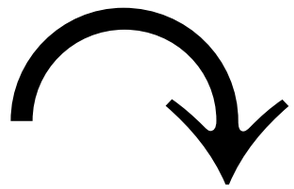
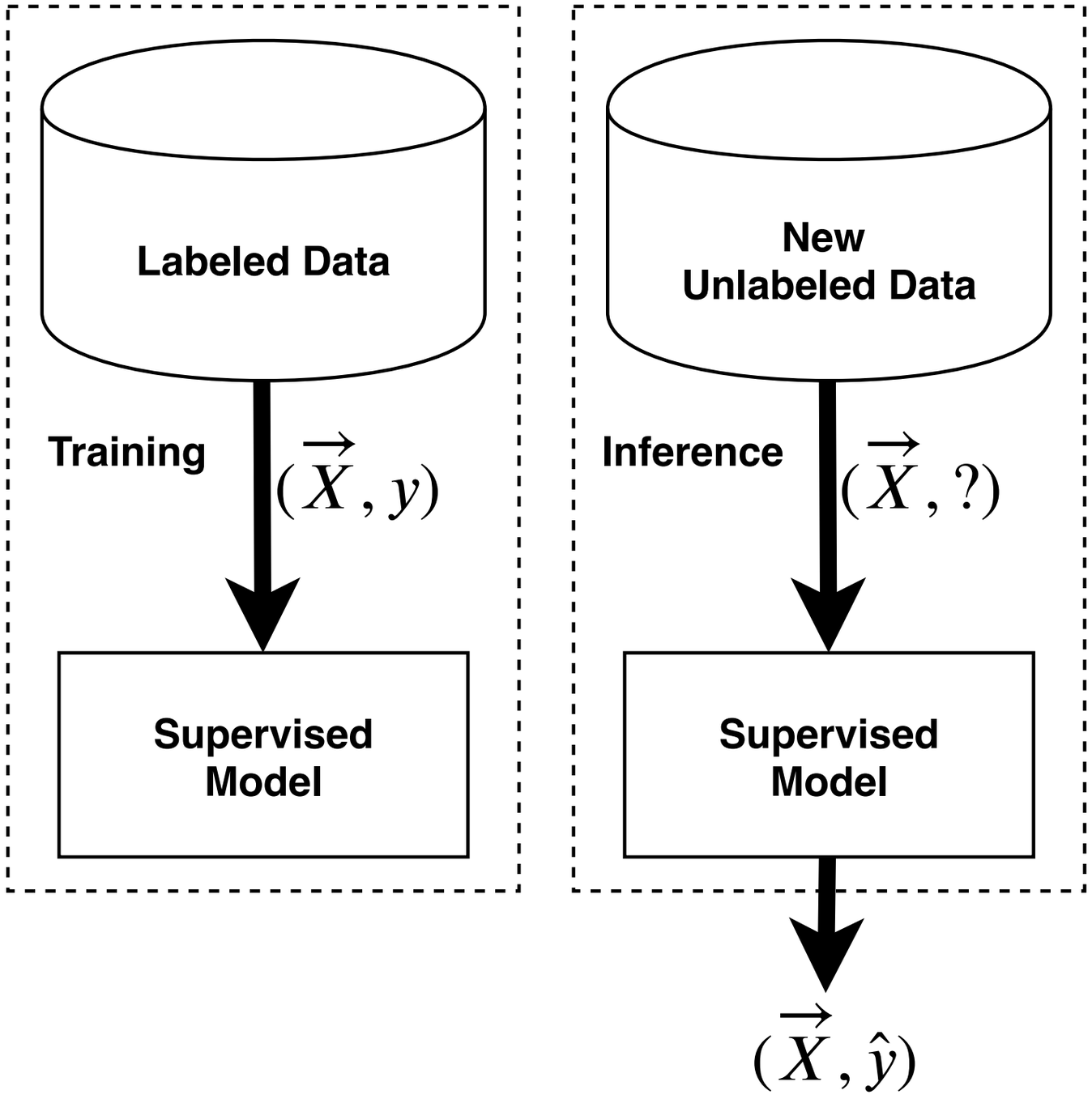


# {A / L}: Autogenerating Supervised Learning Programs

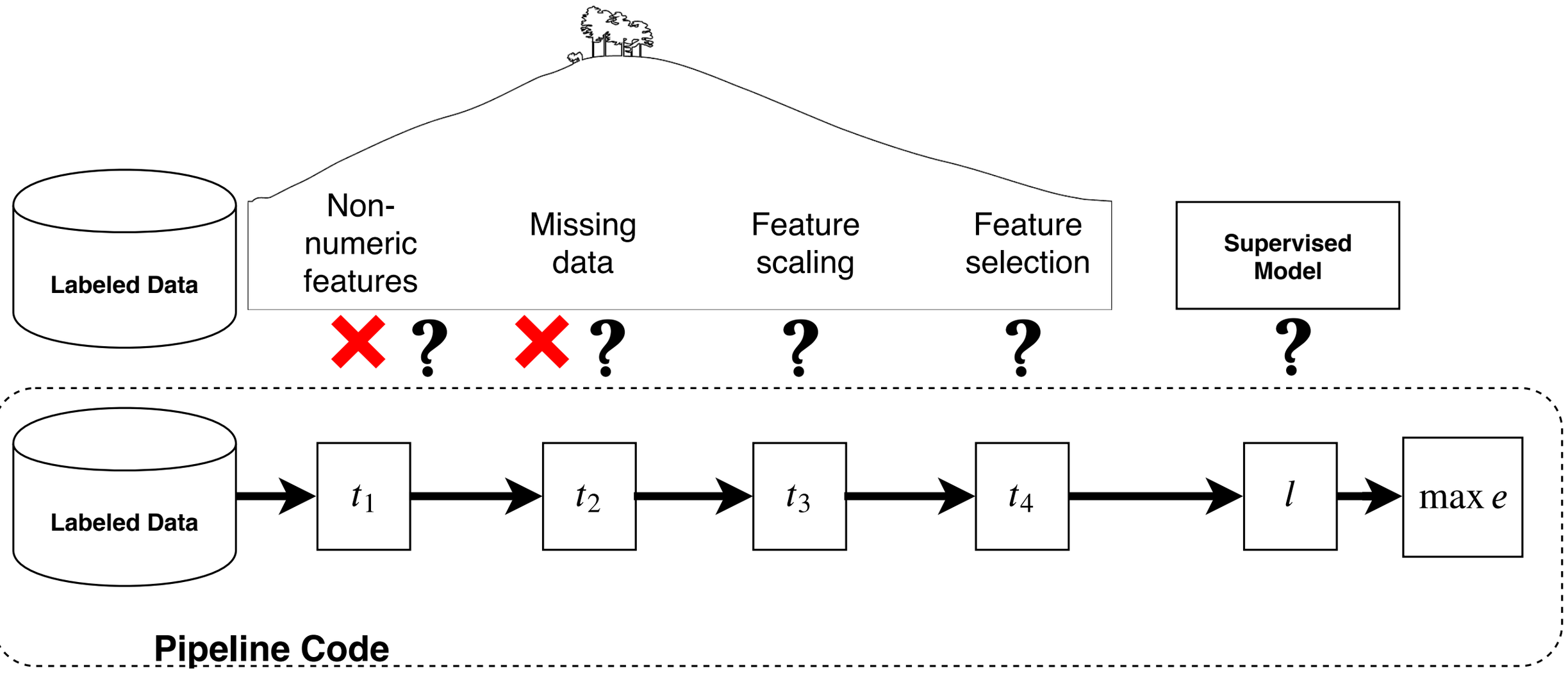
José P. Cambronero, Martin Rinard
MIT


CSG

# (Idealized) Supervised Learning



# (Reality) Pipelines



# (Reality) Choices

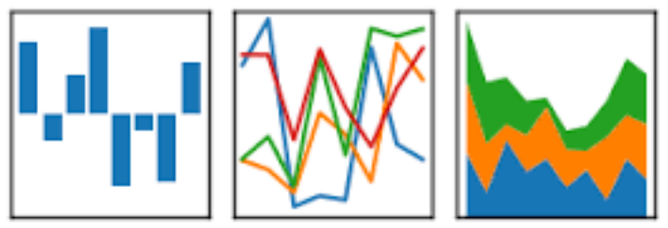

statsmodels

scikit learn

TensorFlow

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



PYTORCH

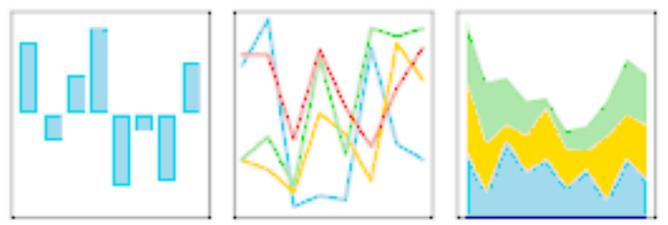
# (Reality) Choices


statsmodels

scikit learn

TensorFlow

pandas



PYTORCH

# (Reality) Choices

- 5.2. Feature extraction
  - 5.2.1. Loading features from dicts
  - 5.2.2. Feature hashing
    - 5.2.2.1. Implementation details
  - 5.2.3. Text feature extraction
    - 5.2.3.1. The Bag of Words representation
    - 5.2.3.2. Sparsity
    - 5.2.3.3. Common Vectorizer usage
      - 5.2.3.3.1. Using stop words
    - 5.2.3.4. Tf–idf term weighting
    - 5.2.3.5. Decoding text files
    - 5.2.3.6. Applications and examples
    - 5.2.3.7. Limitations of the Bag of Words representatic
    - 5.2.3.8. Vectorizing a large text corpus with the hashi
    - 5.2.3.9. Performing out-of-core scaling with HashingV
    - 5.2.3.10. Customizing the vectorizer classes
  - 5.2.4. Image feature extraction
    - 5.2.4.1. Patch extraction
    - 5.2.4.2. Connectivity graph of an image

**1.1. Generalized Linear Models**

- 1.1.1. Ordinary Least Squares
  - 1.1.1.1. Ordinary Least Squares Complexity
- 1.1.2. Ridge Regression
  - 1.1.2.1. Ridge Complexity
  - 1.1.2.2. Setting the regularization parameter: generalized Cross-Validation
- 1.1.3. Lasso
  - 1.1.3.1. Setting regularization parameter
    - 1.1.3.1.1. Using cross-validation
    - 1.1.3.1.2. Information-criteria based model selection
    - 1.1.3.1.3. Comparison with the regularization parameter of SVM
- 1.1.4. Multi-task Lasso
- 1.1.5. Elastic-Net
- 1.1.6. Multi-task Elastic-Net
- 1.1.7. Least Angle Regression
- 1.1.8. LARS Lasso
  - 1.1.8.1. Mathematical formulation
- 1.1.9. Orthogonal Matching Pursuit (OMP)
- 1.1.10. Bayesian Regression
  - 1.1.10.1. Bayesian Ridge Regression
  - 1.1.10.2. Automatic Relevance Determination - ARD
- 1.1.11. Logistic regression
- 1.1.12. Stochastic Gradient Descent - SGD
- 1.1.13. Perceptron
- 1.1.14. Passive Aggressive Algorithms
- 1.1.15. Robustness regression: outliers and modeling errors
  - 1.1.15.1. Different scenario and useful concepts
  - 1.1.15.2. RANSAC: RANdom SAmple Consensus
    - 1.1.15.2.1. Details of the algorithm
  - 1.1.15.3. Theil-Sen estimator: generalized-median-based estimator
    - 1.1.15.3.1. Theoretical considerations
  - 1.1.15.4. Huber Regression
  - 1.1.15.5. Notes
- 1.1.16. Polynomial regression: extending linear models with

**1.2. Linear and Quadratic Discriminant Analys**

- 1.2.1. Dimensionality reduction using Linear Discriminant An
- 1.2.2. Mathematical formulation of the LDA and QDA classifi
- 1.2.3. Mathematical formulation of LDA dimensionality reduc
- 1.2.4. Shrinkage
- 1.2.5. Estimation algorithms

**1.3. Kernel ridge regression**

**1.4. Support Vector Machines**

- 1.4.1. Classification
  - 1.4.1.1. Multi-class classification
  - 1.4.1.2. Scores and probabilities
  - 1.4.1.3. Unbalanced problems
- 1.4.2. Regression
- 1.4.3. Density estimation, novelty detection

- 5.3. Preprocessing data
  - 5.3.1. Standardization, or mean removal and variance scaling
    - 5.3.1.1. Scaling features to a range
    - 5.3.1.2. Scaling sparse data
    - 5.3.1.3. Scaling data with outliers
    - 5.3.1.4. Centering kernel matrices
  - 5.3.2. Non-linear transformation
    - 5.3.2.1. Mapping to a Uniform distribution
    - 5.3.2.2. Mapping to a Gaussian distribution
  - 5.3.3. Normalization
  - 5.3.4. Encoding categorical features
  - 5.3.5. Discretization
    - 5.3.5.1. K-bins discretization
    - 5.3.5.2. Feature binarization
  - 5.3.6. Imputation of missing values
  - 5.3.7. Generating polynomial features
  - 5.3.8. Custom transformers

- 5.4. Imputation of missing values
  - 5.4.1. Univariate vs. Multivariate Imputation
  - 5.4.2. Univariate feature imputation
  - 5.4.3. Multivariate feature imputation
    - 5.4.3.1. Flexibility of IterativeImputer
    - 5.4.3.2. Multiple vs. Single Imputation
  - 5.4.4. References
  - 5.4.5. Marking imputed values

# (Reality) Choices

- Just in Scikit-Learn and XGBoost:
  - 51 classes/functions for transformations
  - 96 classes/functions for learning

- Most pipelines have multiple steps

- Matching components requires expertise (and empirical evaluation)

```
import xgboost
import sklearn
import sklearn.feature_extraction.text
import sklearn.linear_model.logistic
import sklearn.preprocessing.imputation
import runtime_helpers

from sklearn.pipeline import Pipeline

# read inputs and split
X, y = runtime_helpers.read_input('titanic.csv', 'Survived')
X_train, y_train, X_val, y_val = train_test_split(X, y, test_size=0.25)

# build pipeline with transforms and model
pipeline = Pipeline([
  ('t0', runtime_helpers.ColumnLoop(sklearn.feature_extraction.text.CountVectorizer)),
  ('t1', sklearn.preprocessing.imputation.Imputer()),
  ('model', sklearn.linear_model.logistic.LogisticRegression()),
])

# fit pipeline
pipeline.fit(X_train, y_train)

# evaluate on held-out data
print(pipeline.score(X_val, y_val))

# train pipeline on train + val for new predictions
pipeline.fit(X, y)

def predict(X_new):
  return pipeline.predict(X_new)
```

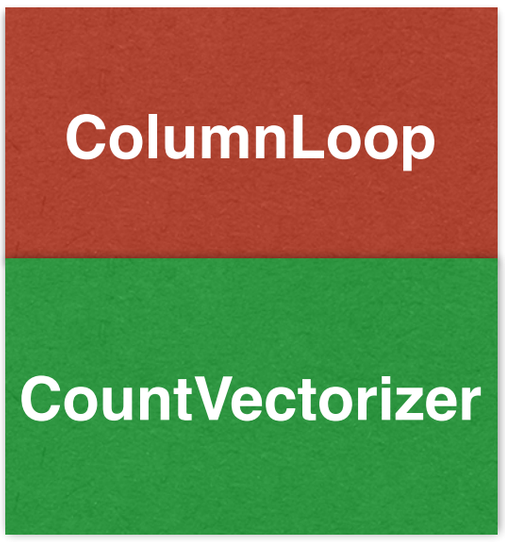
```
import xgboost
import sklearn
import sklearn.feature_extraction.text
import sklearn.linear_model.logistic
import sklearn.preprocessing.imputation
import runtime_helpers

from sklearn.pipeline import Pipeline

# read inputs and split
X, y = runtime_helpers.read_input('titanic.csv', 'Survived')
X_train, y_train, X_val, y_val = train_test_split(X, y, test_size=0.25)

# build pipeline with transforms and model
pipeline = Pipeline([
  ('t0', runtime_helpers.ColumnLoop(sklearn.feature_extraction.text.CountVectorizer)),
  ('t1', sklearn.preprocessing.imputation.Imputer()),
  ('model', sklearn.linear_model.logistic.LogisticRegression()),
])

# fit pipeline
pipeline.fit(X_train, y_train)

# evaluate on held-out data
print(pipeline.score(X_val, y_val))

# train pipeline on train + val for new predictions
pipeline.fit(X, y)

def predict(X_new):
  return pipeline.predict(X_new)
```

```
# build pipeline with transforms and model
pipeline = Pipeline([
   ('t0', runtime_helpers.ColumnLoop(
     sklearn.feature_extraction.text.CountVectorizer
   )),
   ('t1', sklearn.preprocessing.imputation.Imputer())    ,
   ('model', sklearn.linear_model.logistic.LogisticRegression())
])
```

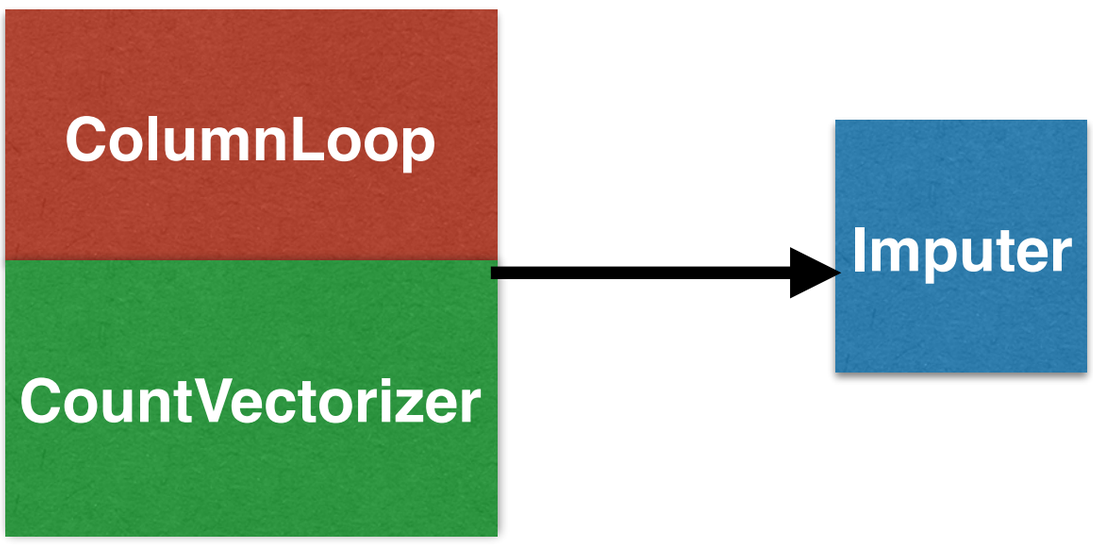
```
# build pipeline with transforms and model
pipeline = Pipeline([
   ('t0', runtime_helpers.ColumnLoop(
     sklearn.feature_extraction.text.CountVectorizer
   )),
   ('t1', sklearn.preprocessing.imputation.Imputer())    ,
   ('model', sklearn.linear_model.logistic.LogisticRegression())
])
```

**ColumnLoop**

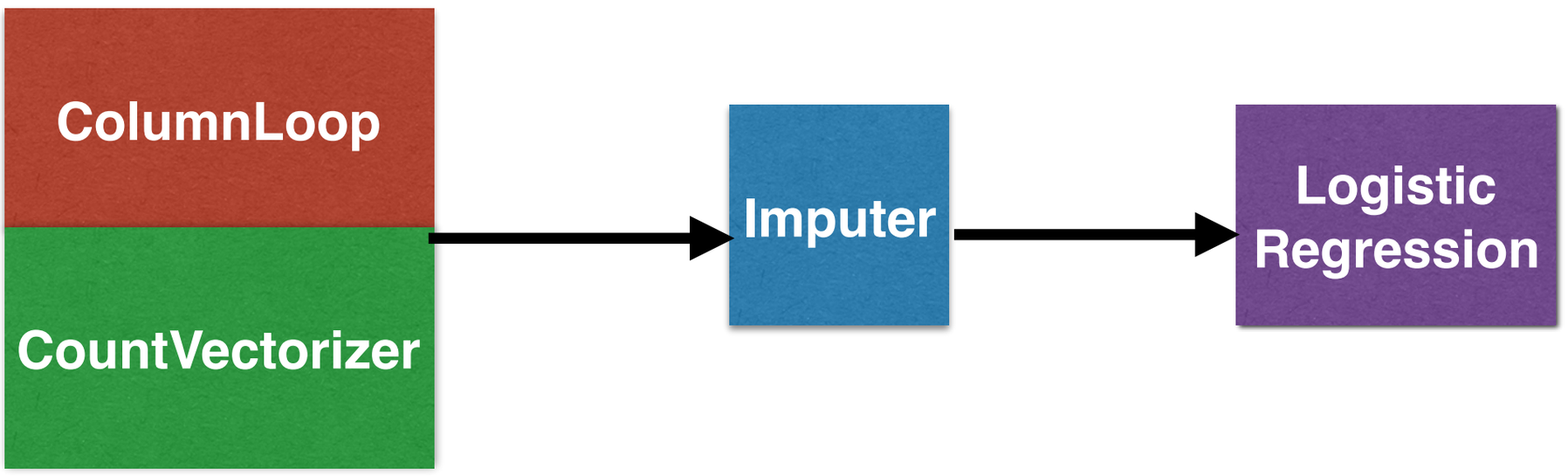
```
# build pipeline with transforms and model
pipeline = Pipeline([
   ('t0', runtime_helpers.ColumnLoop(
     sklearn.feature_extraction.text.CountVectorizer
   )),
   ('t1', sklearn.preprocessing.imputation.Imputer())    ,
   ('model', sklearn.linear_model.logistic.LogisticRegression())
])
```



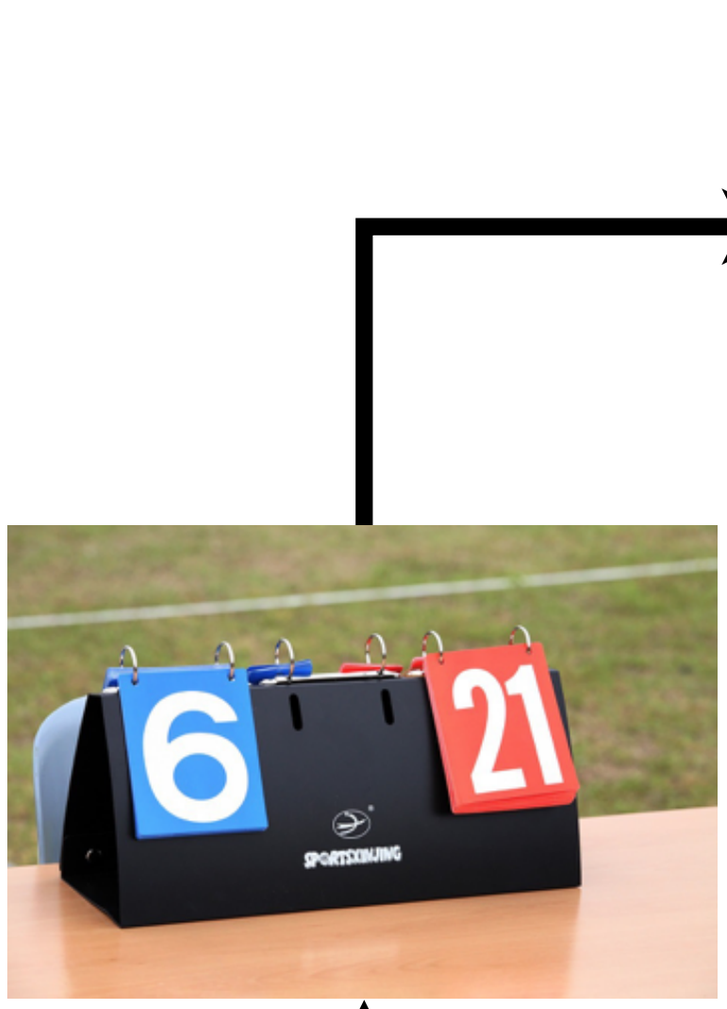
```
# build pipeline with transforms and model
pipeline = Pipeline([
   ('t0', runtime_helpers.ColumnLoop(
     sklearn.feature_extraction.text.CountVectorizer
   )),
   ('t1', sklearn.preprocessing.imputation.Imputer()) ,
   ('model', sklearn.linear_model.logistic.LogisticRegression())
])
```



```
# build pipeline with transforms and model
pipeline = Pipeline([
   ('t0', runtime_helpers.ColumnLoop(
     sklearn.feature_extraction.text.CountVectorizer
   )),
   ('t1', sklearn.preprocessing.imputation.Imputer()) ,
   ('model', sklearn.linear_model.logistic.LogisticRegression())
])
```

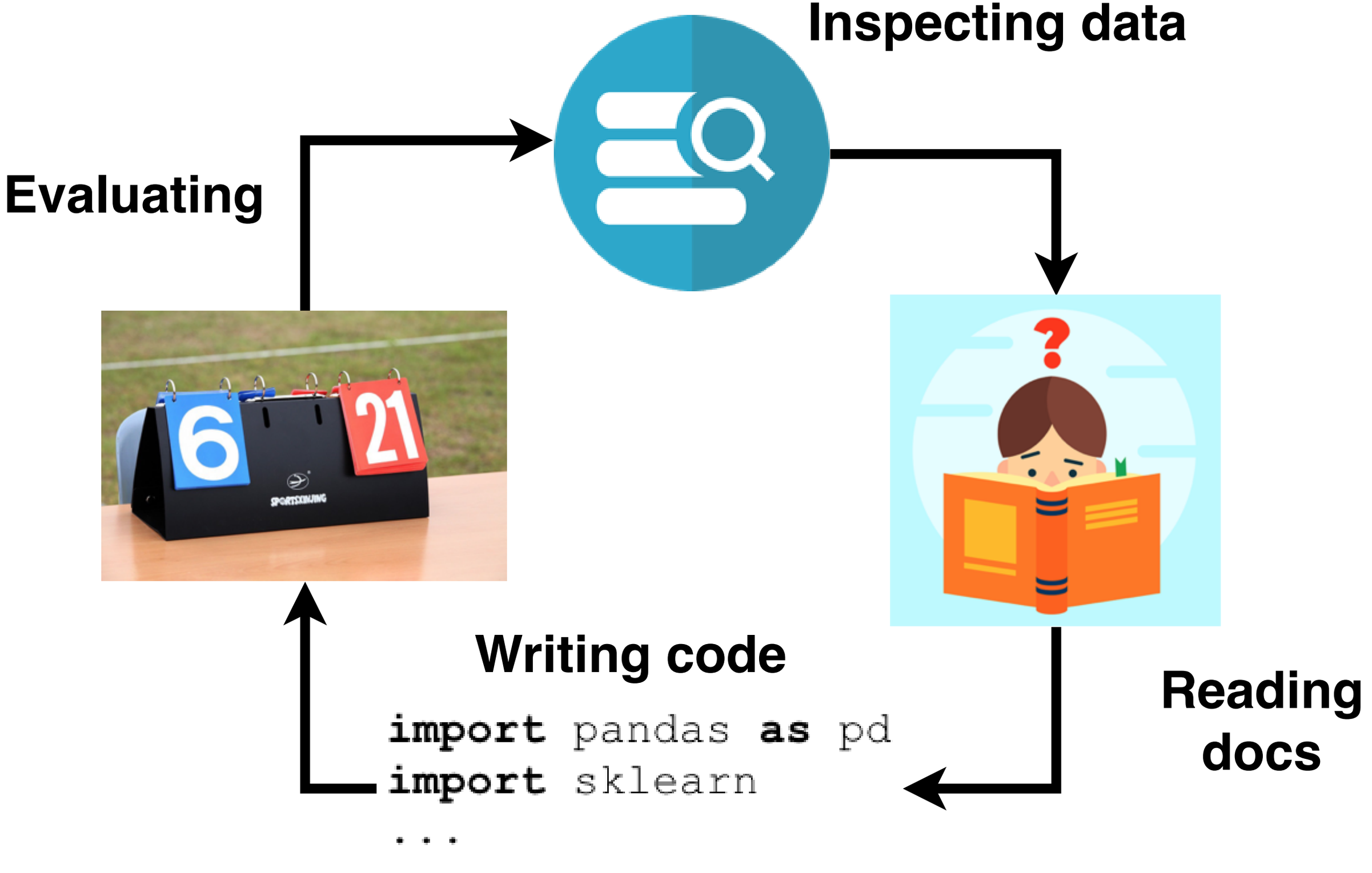


# Requires




```
import pandas as pd
import sklearn
...
```

# Requires


Inspecting data
Evaluating
6
21
Writing code
import pandas as pd
import sklearn
...
Reading docs

Why do I have to do this?

I’m not the first person to do machine learning.

Why can’t I just reuse what everyone else did?

# Conventional Answer

- Your data isn't like everyone else's data
- Their pipeline won't work for your data
- You are special and need to have your own pipeline!

# Conventional Answer is Both True and False

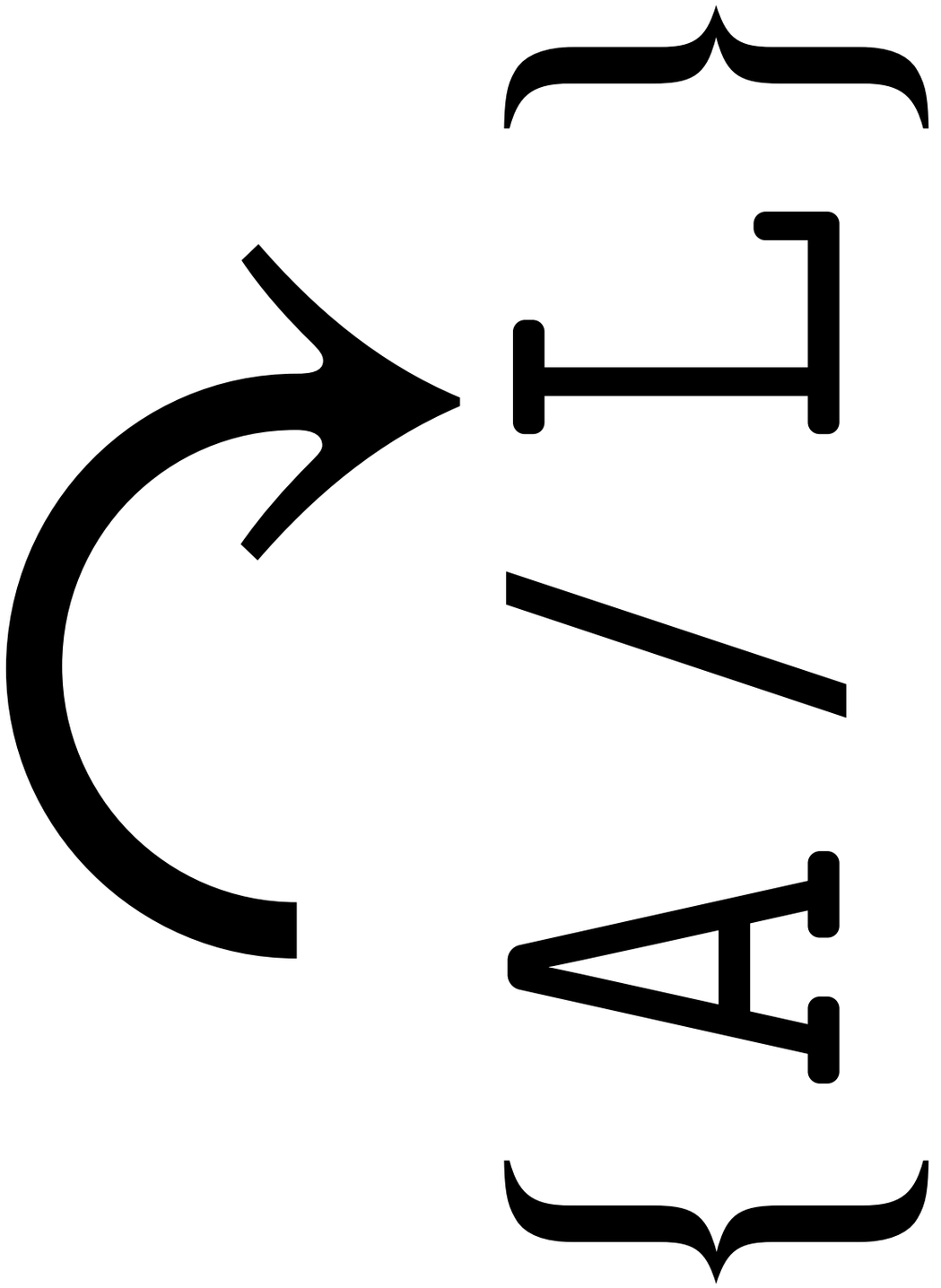
- You do need your own pipeline
- Pipeline does depend on your data
- So you can’t just reuse someone else’s pipeline
- But your pipeline will be a lot like pipelines that other people have developed for similar data

# Conventional Answer is Both True and False

- We collected 500 pipelines, which use 9 datasets, from a public forum (Kaggle)

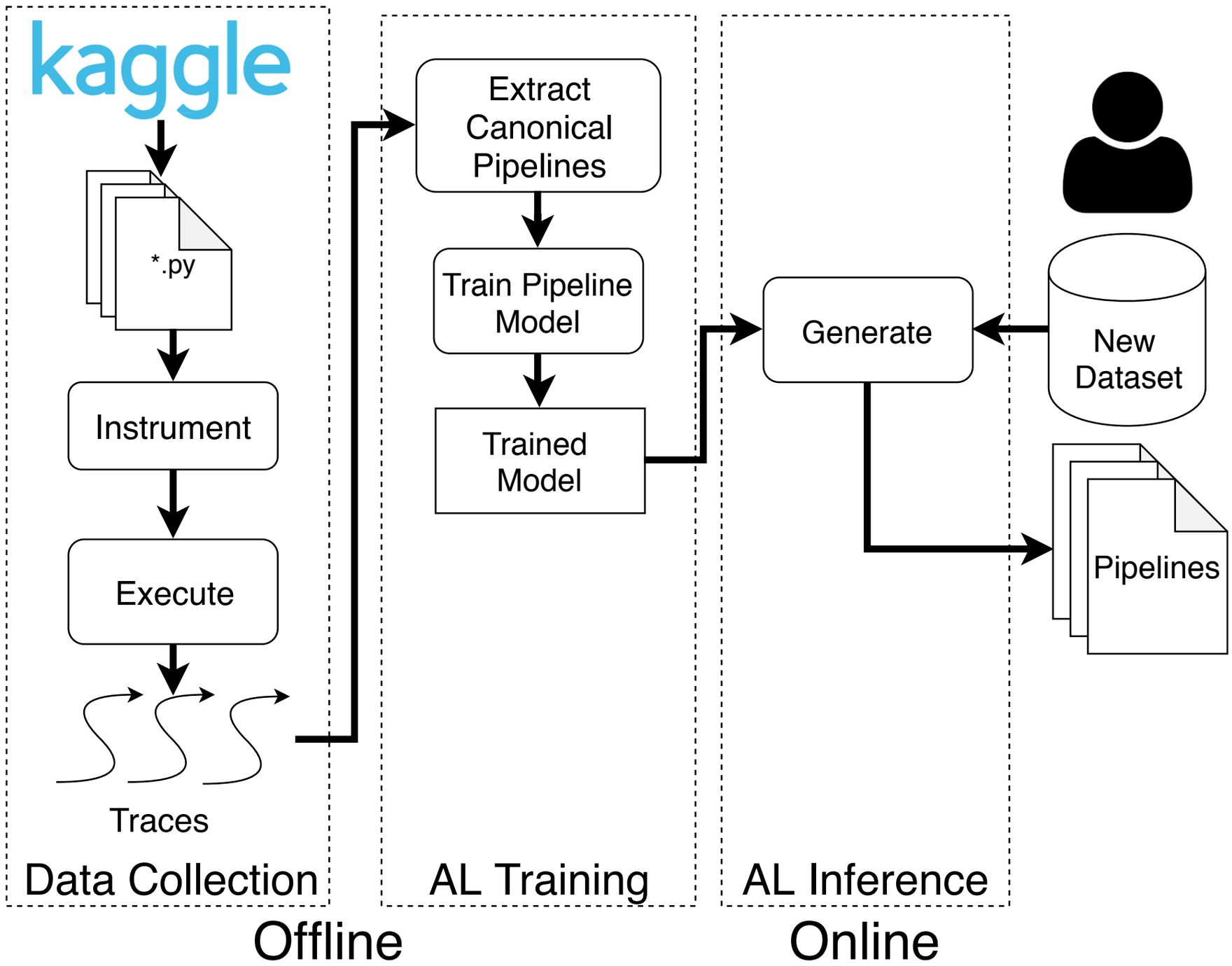
- On average 92% of the pipelines were different from pipelines targeting other datasets

- But within a dataset only 40% of pipelines are unique on average



# Learning from Programs

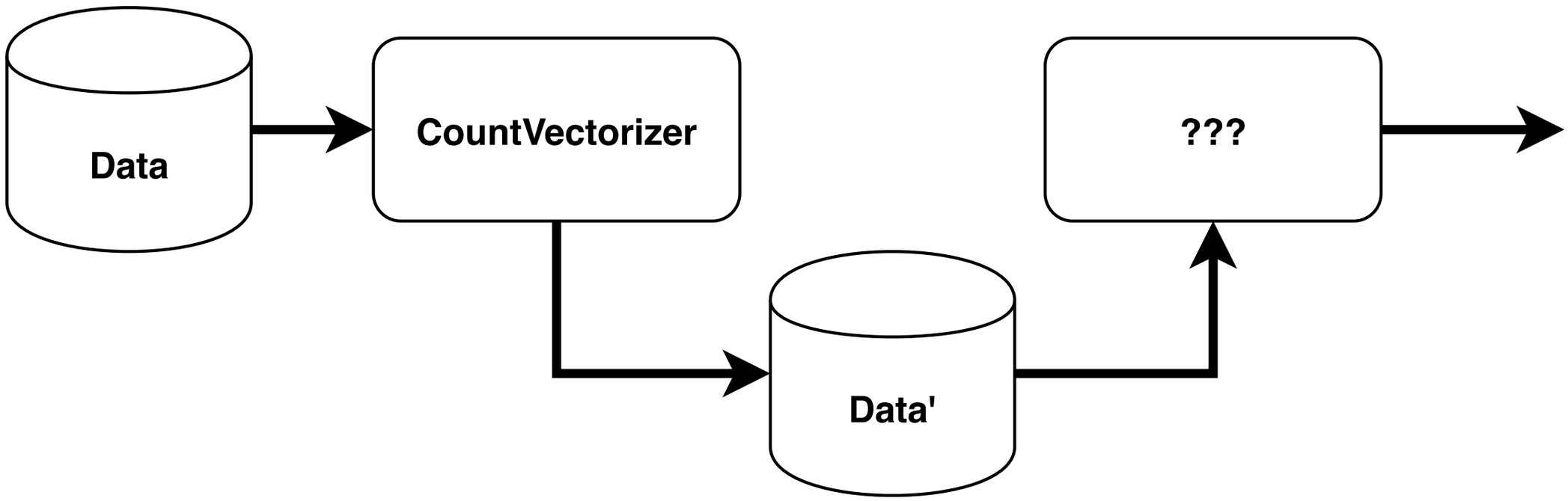

# {A/L} Approach

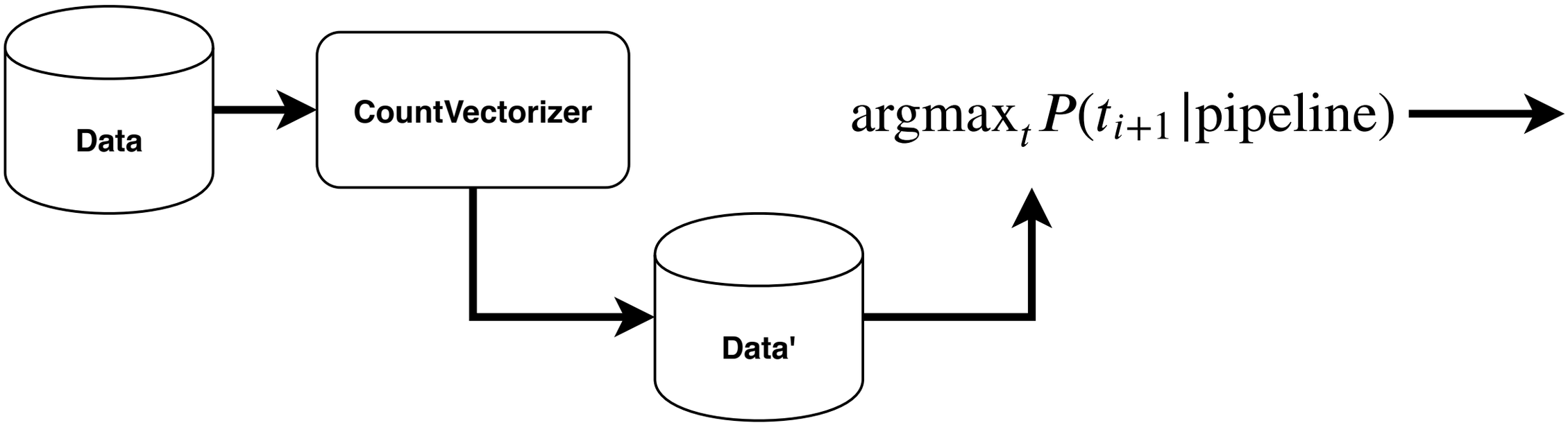

kaggle
*.py
Instrument
Execute
Traces
Data Collection
Extract Canonical Pipelines
Train Pipeline Model
Trained Model
AL Training
Generate
New Dataset
Pipelines
AL Inference
Offline
Online

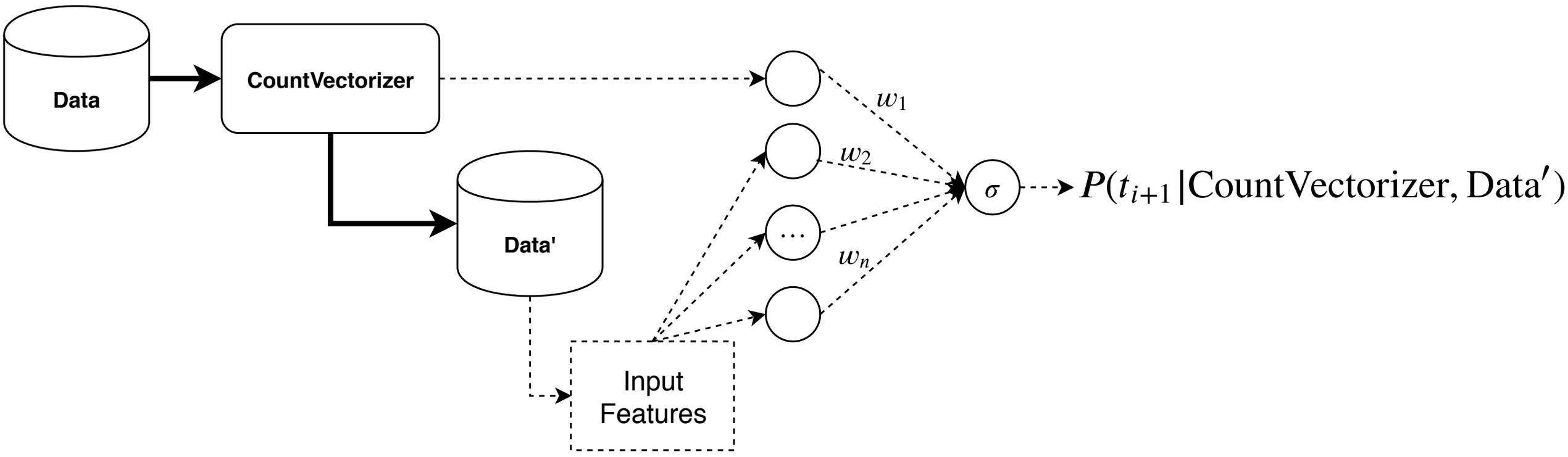
# Data Collection

- Approximately 500 programs
- 9 datasets
- Collected from Kaggle
- Dynamic instrumentation inserted to collect
  - Functions called in key libraries
  - Argument summaries
  - Call dependences

# How do you generate a pipeline?


Data
CountVectorizer
???
Data'


Data
CountVectorizer
Data'
argmax$_t$ $P(t_{i+1}$ |pipeline)


Data
CountVectorizer
Data'
Input
Features
$w_1$
$w_2$
$w_n$
$\sigma$
$P(t_{i+1} | \text{CountVectorizer, Data}')$

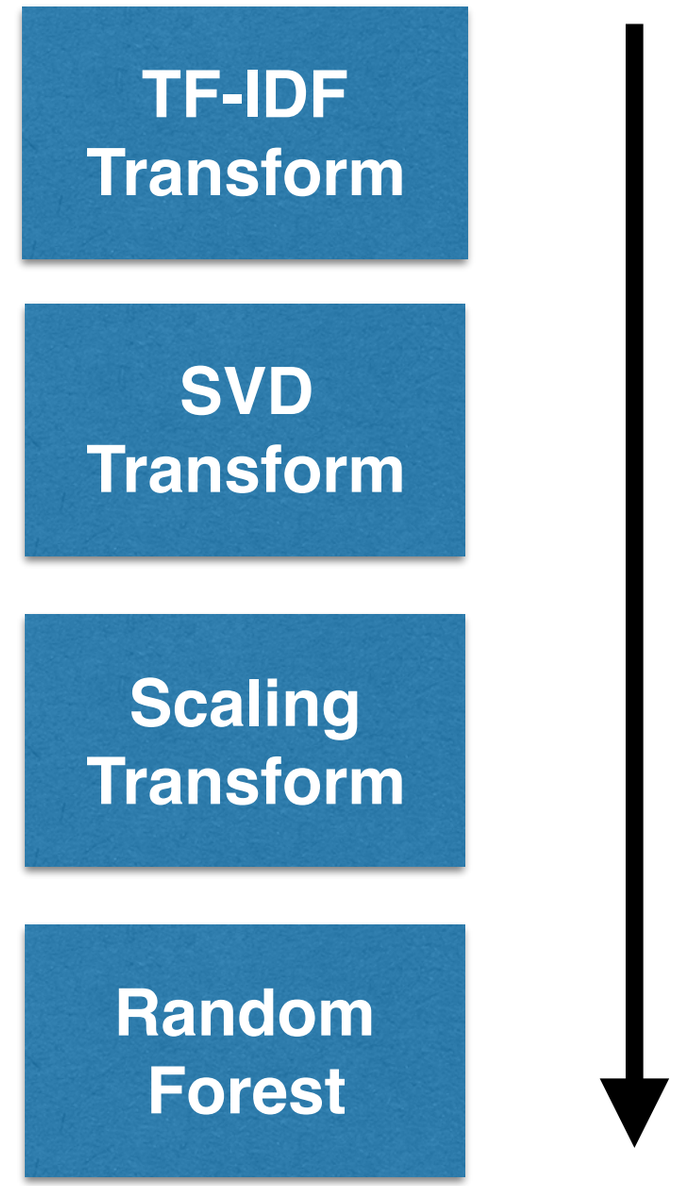
# Summarizing Inputs: Data Features

- Label probability (max, min mean)
- Mean kurtosis
- Count of columns and rows
- Count of columns by type
- Number of unique labels
- Mean skewness
- Label counts (max, min, mean)
- Percentage missing values
- Cross-column correlation (max, min, mean)
- Non-zero counts (mean)
- PDF/CDF with common distributions
- ...

# Dynamic Traces

- Slicing-based algorithm to extract canonical pipeline from dynamic trace

```
[('TfidfVectorizer', 'T'),
  ('TfidfVectorizer.fit', 'T'),
  ('TfidfVectorizer.transform', 'T'),
('TfidfVectorizer.transform', 'T'),
('TruncatedSVD', 'T'),
('TruncatedSVD.fit_transform', 'T'),
('TruncatedSVD.transform', 'T'),
('StandardScaler', 'T'),
('TransformerMixin.fit_transform', 'T'),
('StandardScaler.transform', 'T'),
('RandomForestClassifier', 'L'),
('BaseForest.fit', 'L'),
('ForestClassifier.predict', 'E')]
```



# Generating Multiple Pipelines: Beam Search with Probabilistic Model

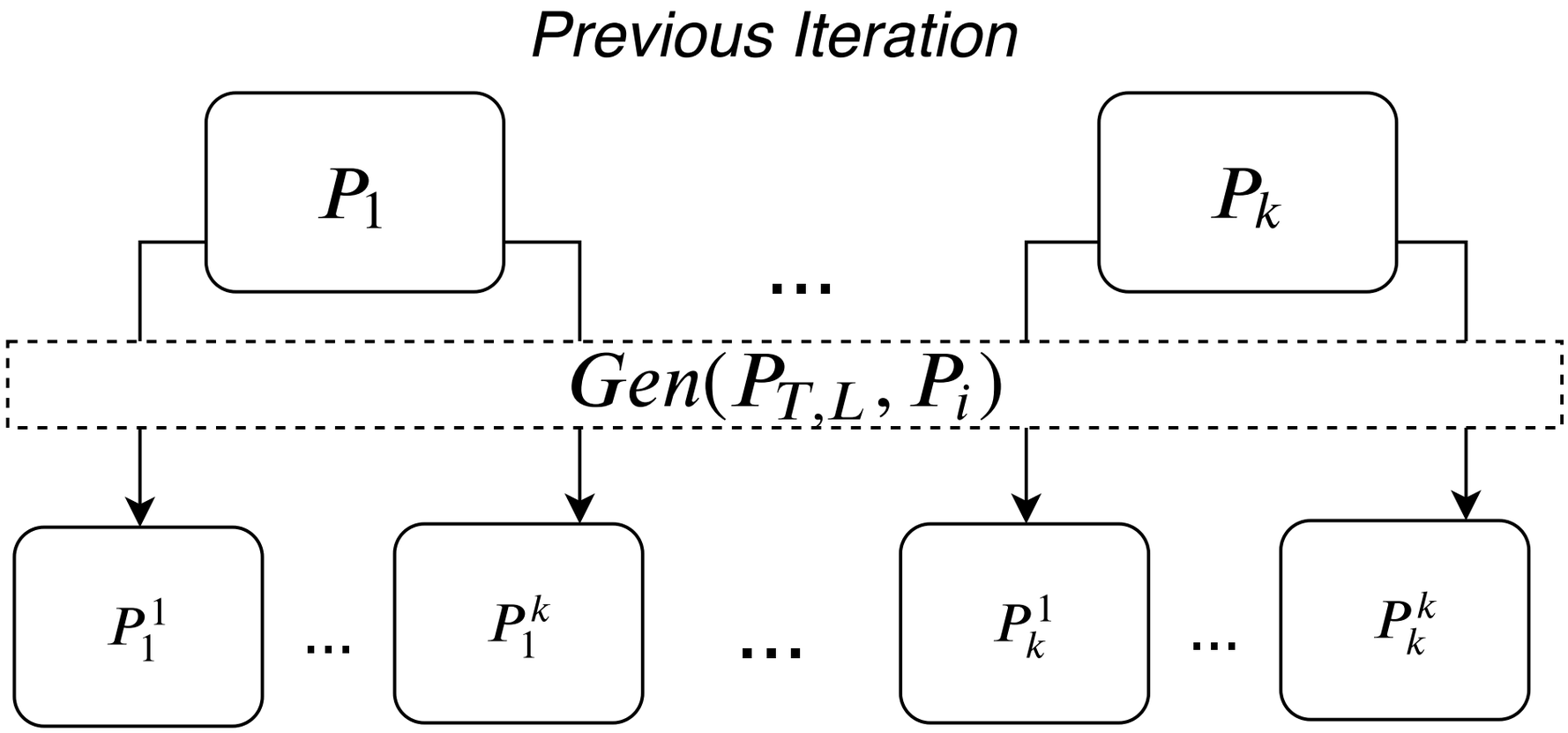
*Previous Iteration*

$P_1$

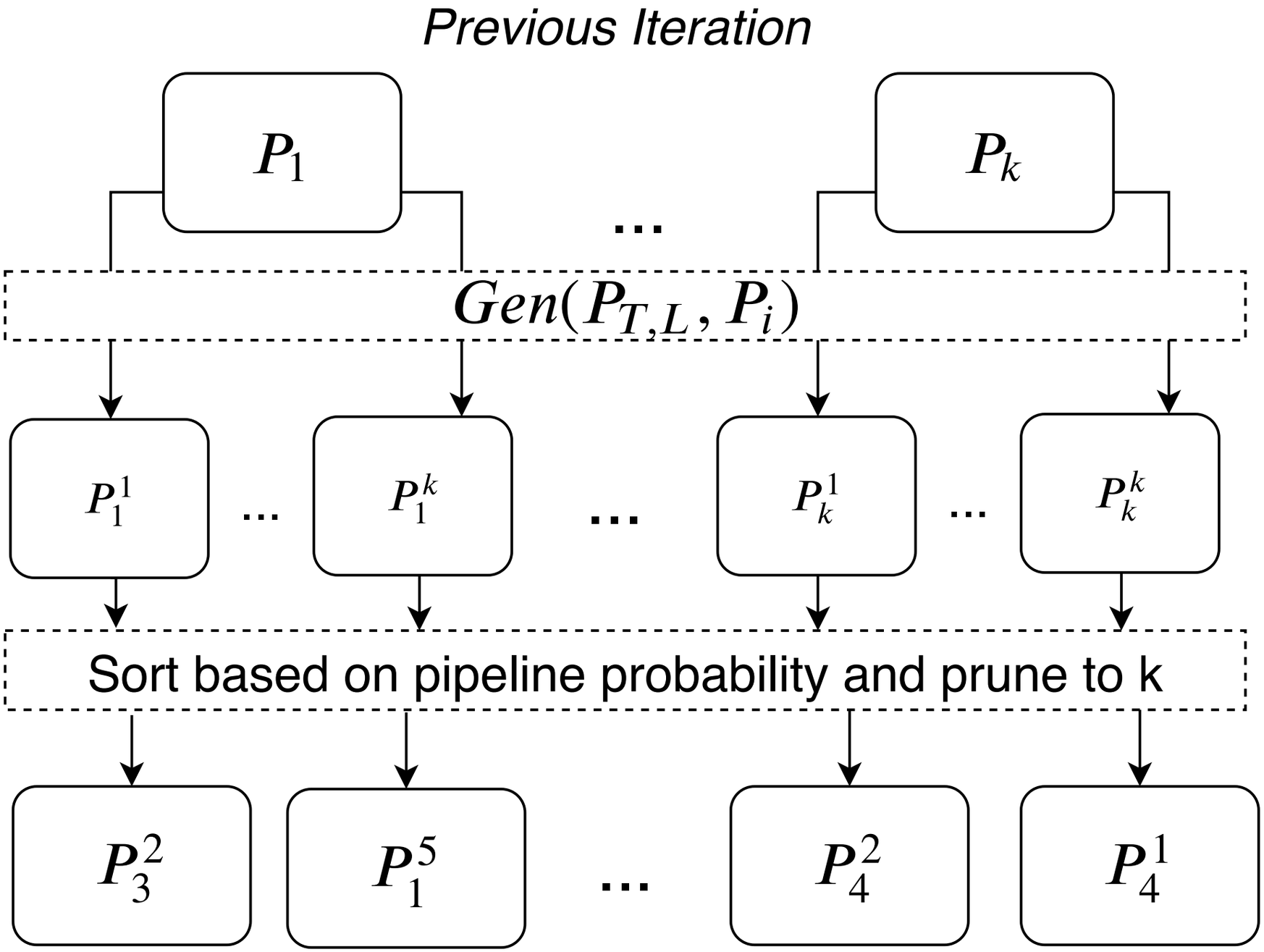
...

$P_k$

***k current pipelines***

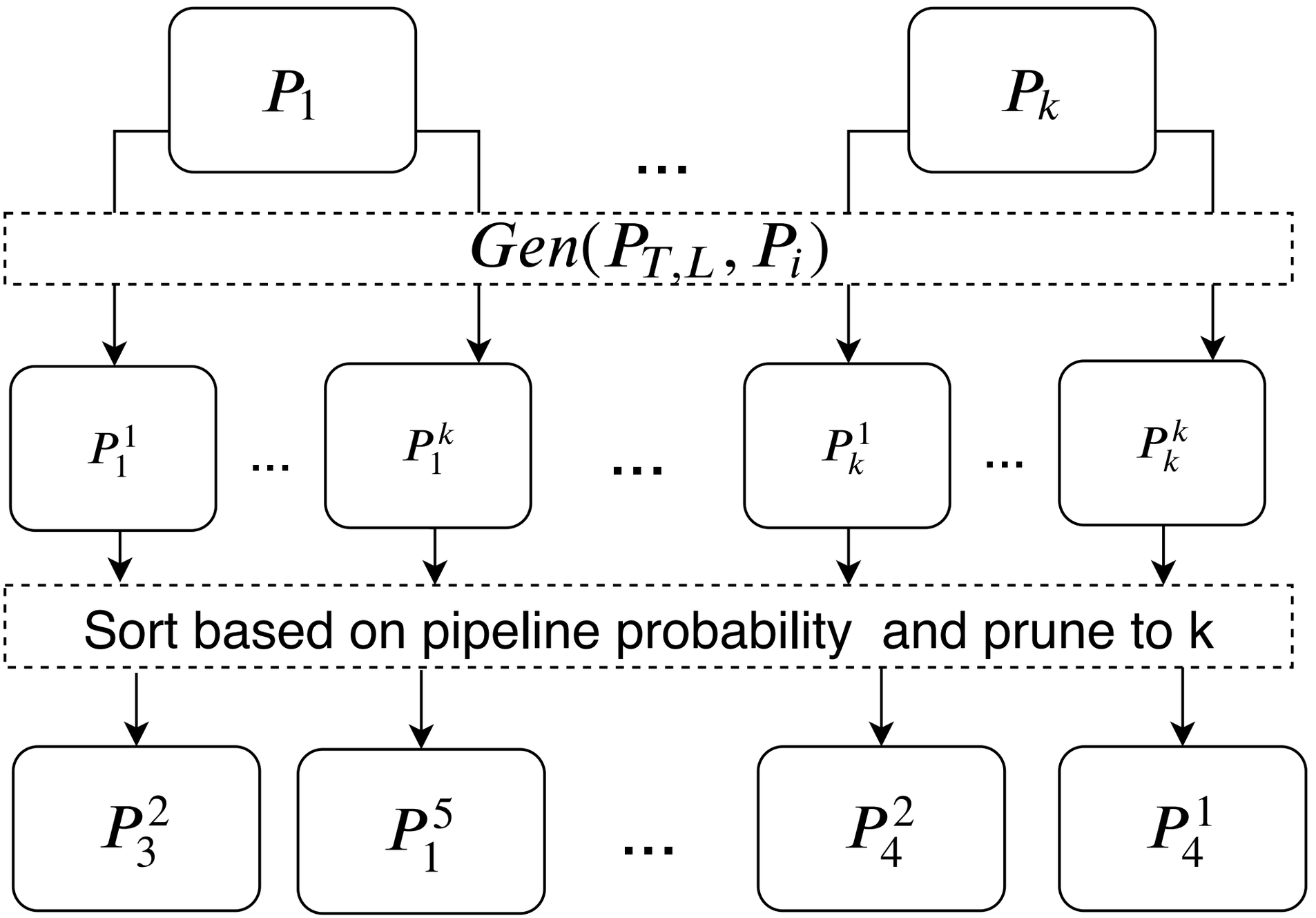



***Extend each with top k components based on predicted probability***




***Keep top k pipelines based on predicted probability***

*Previous Iteration*

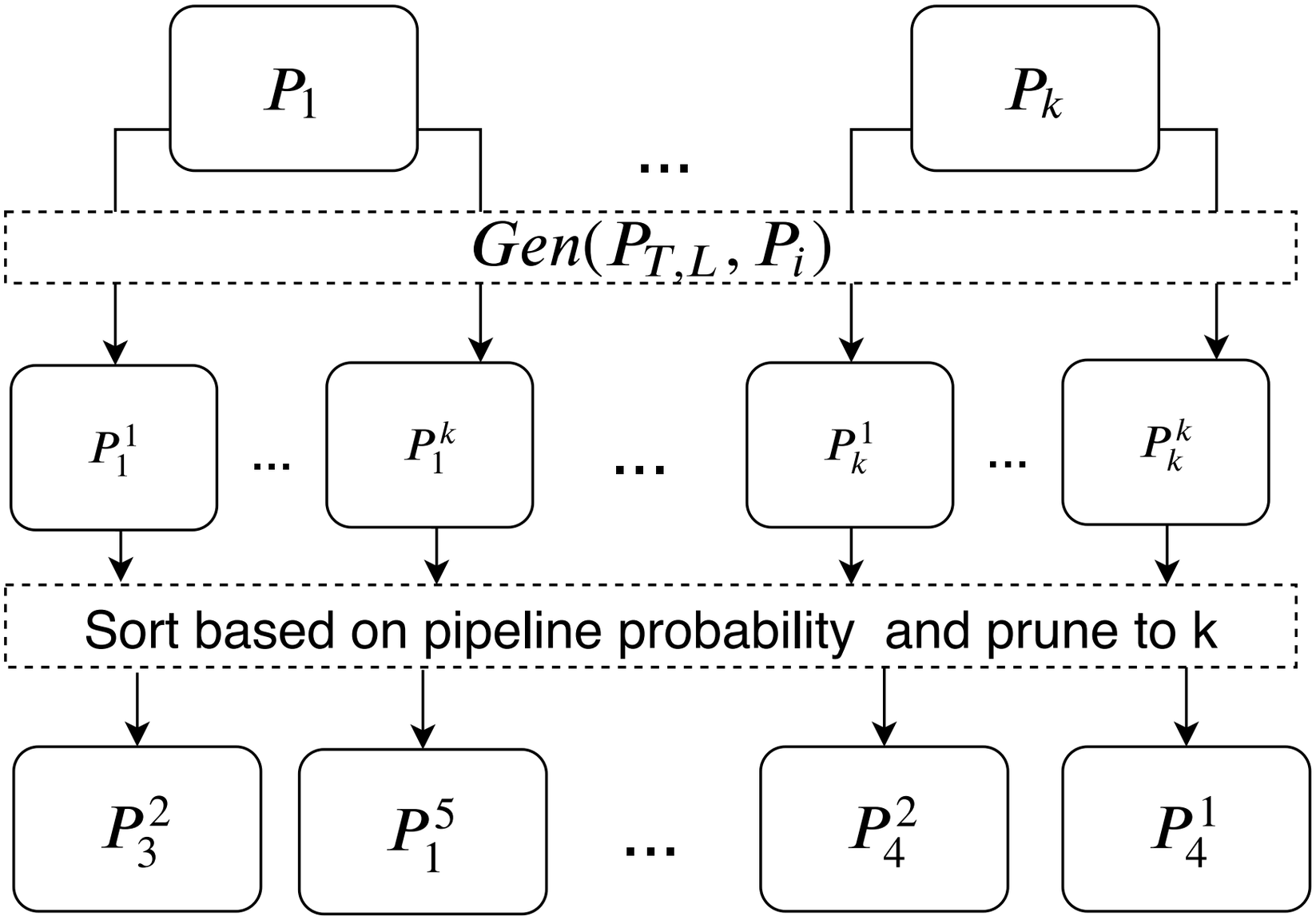



*Next Iteration*

...

***Iterate until provided depth-bound***

*Previous Iteration*




*Iterate to depth-bound*

...

*Final Iteration*

Sort on

$\texttt{Score}(P_i, X_{\text{val}}, y_{\text{val}})$

***Use validation data to sort final set of pipelines***

# Evaluation

# AutoML Systems

{A/L}

- (Ours) learning from programs

Auto-Sklearn

- Bayesian-optimization for pipelines, ensemble-based


- Genetic programming over tree-based pipelines

# Benchmark Datasets

- TPOT and Autosklearn paper datasets (21)
  - Pre-processed so all systems run
  - Sourced from PLMB and OpenML

- Kaggle datasets (6)
  - Varied datatypes
  - No initial pre-processing

- Mulan datasets (4)
  - Multi-target regression/classification

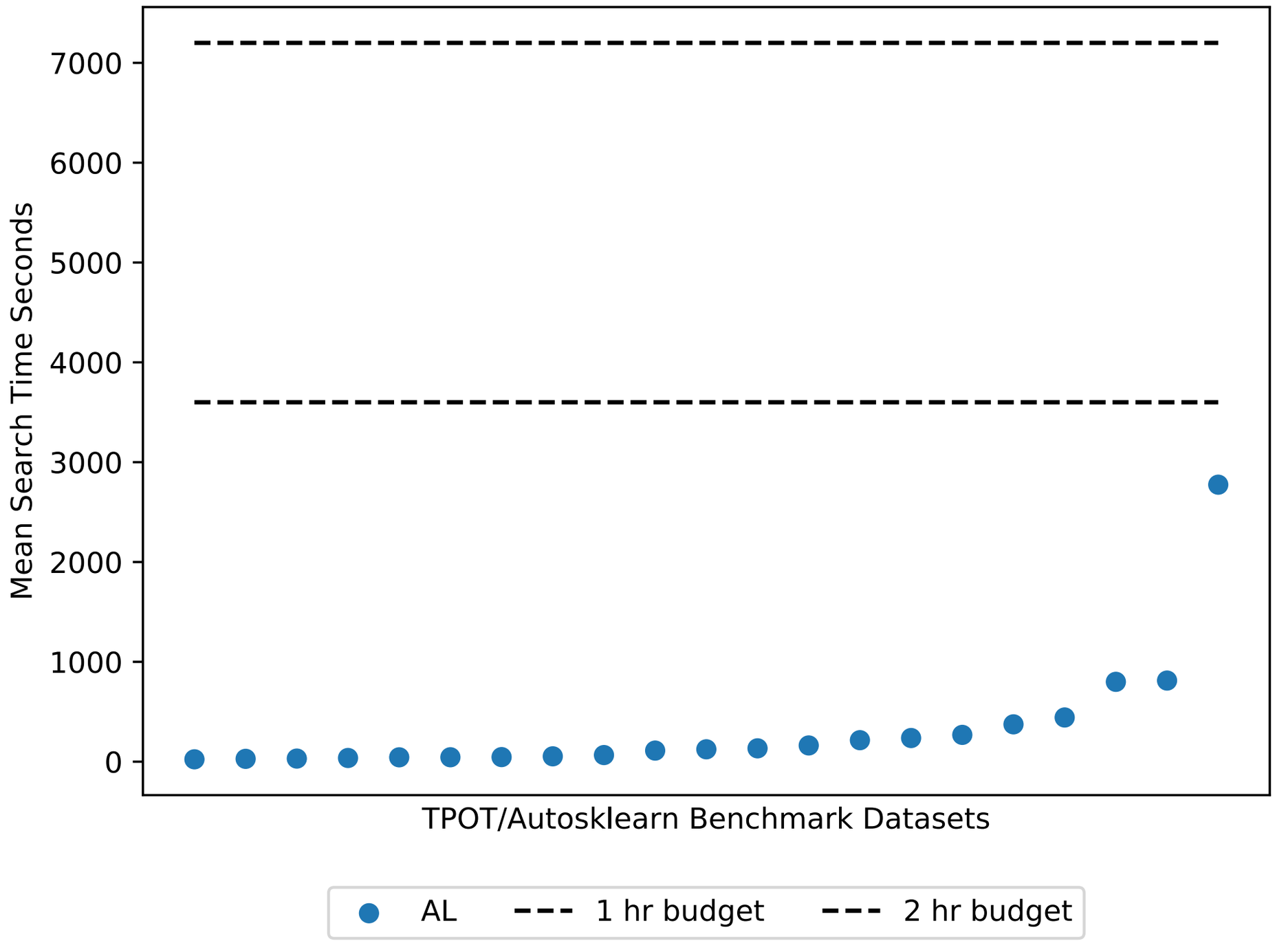
# Search Time Configuration

- Autosklearn/TPOT given 1 hour budget, increased to 2 hours if search does not complete
- AL search depth bounded, time limit per operation in pipeline

# Evaluation Dimensions

- Pipeline generation time
- Ability to handle new datasets
- Accuracy of generated pipelines

# Results:
# Pipeline generation time



# Results:
# Ability to handle new datasets

| Dataset Source | # Datasets | Outcomes |
|---|---|---|
| Kaggle | 6 | Only AL executed out-of-box |
| Mulan | 4 | Only AL executed out-of-box |

# Results:
# Accuracy of generated pipelines

| Dataset Source | # Datasets | Outcomes |
|---|---|---|
| TPOT Paper | 9 / 9 | 7 / 9<br>Comparable |
| Autosklearn Paper | 12 / 13 | 10 / 12<br>Comparable |

# Improvement over Random Strategies

Average performance of top 10 pipelines


Search Randomly

# Improvement over Random Strategies

Average performance of top 10 pipelines


Search Randomly
in AL subset of API

Improves
19 of 21
datasets

**+8.9 F1**
**on average**


Search Randomly

# Improvement over Random Strategies

Average performance of top 10 pipelines


Search with AL

Improves 19 of 21 datasets

**+6.5 F1 on average**

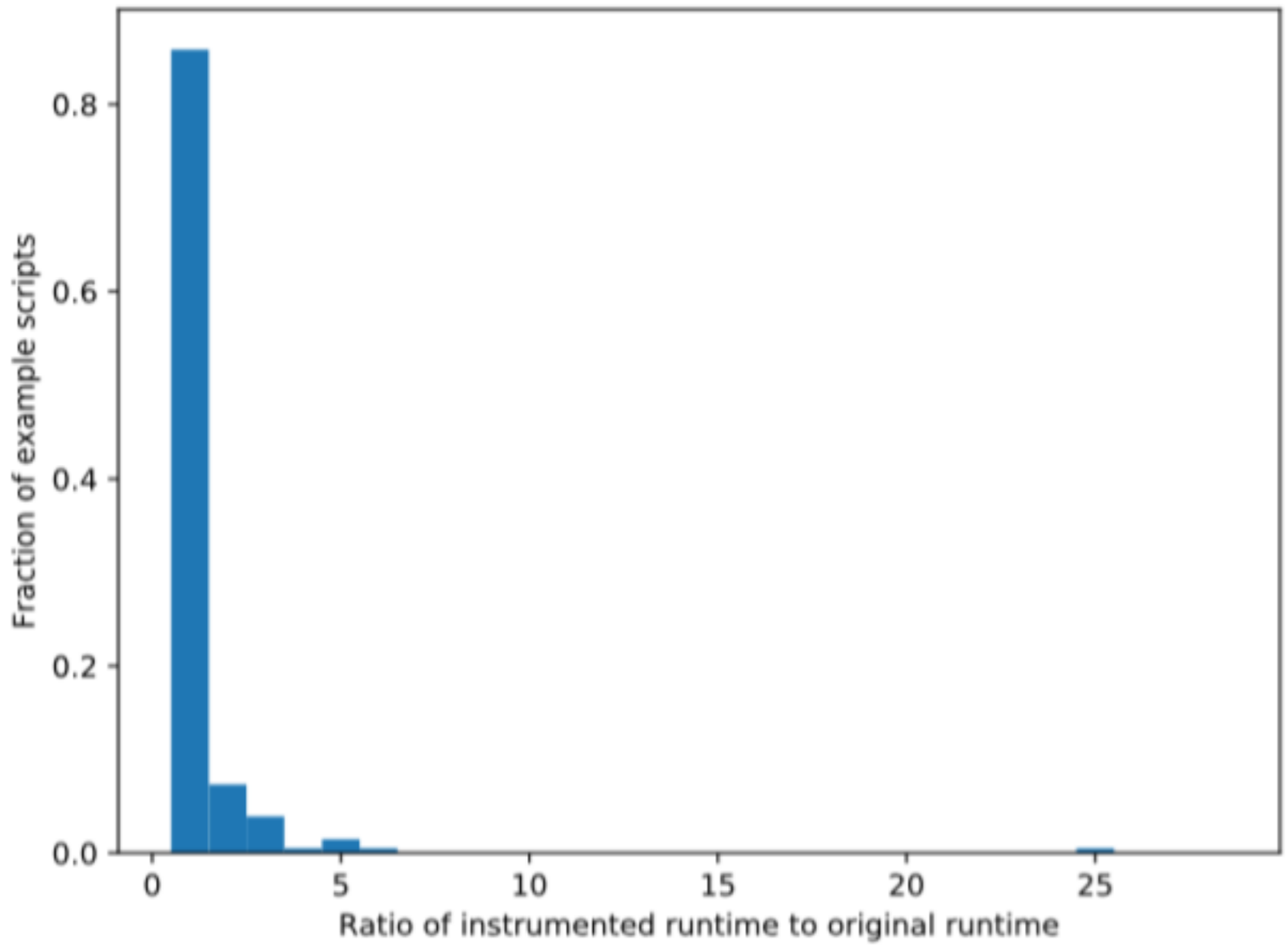

Search Randomly in AL subset of API

Improves 19 of 21 datasets


Search Randomly

# Instrumentation Impact

- 80% of example programs have runtime ratio between 1.0 and 2.0 when instrumented



# Future Work

- AL calls components with default hyperparameters
  - Modify instrumentation and probability model to account for hyperparameter “equivalence classes”
  - Or use AL pipelines as warm-start for hyper-parameter search in other tools

- Introduce new libraries
  - Adapt code generation/instrumentation as necessary

- Explore similar approach in other data analytics tasks (e.g. data visualization)

# Conclusion

- Learning supervised learning pipelines from programs can:
  - Speed up and simplify search
  - Produce comparable performance to existing AutoML tools
  - Extend to more datasets without additional effort