Ethics of Data Science – Part III

Week 1: Reproducibility and Robustness

Pipelines

Dr. Chris Anagnostopoulos, Hon. Senior Lecturer

A tale of three ways of working: the script

```
import pandas as pd
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.ensemble import RandomForestClassifier

# Load dataset
df = pd.read_csv('./data/arrh.csv')
X = df.drop('class', 1).copy()
y = df['class'].copy()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

Loading the data and splitting into train and test

A tale of three ways of working: the script

```
best_max_depth = GridSearchCV(
    RandomForestClassifier(),
    param_grid={'max_depth': [10, 20]}
).fit(X_train, y_train).best_params_['max_depth']

clf = RandomForestClassifier(
  max_depth=best_max_depth).fit(X_train, y_train)

vt = SelectKBest(f_classif, k=5).fit(X_train, y_train)

clf_vt = RandomForestClassifier(max_depth=best_max_depth).fit(
    vt.transform(X_train), y_train)
```

Find the best-performing maximum tree depth, and the 5 best variables for that depth of trees. Then refit.

A tale of three ways of working: the pipeline

```
pipe = Pipeline([
  ('ft', SelectKBest()), ('clf', RandomForestClassifier())])

grid = {'ft__k':[5, 10], 'clf__max_depth':[10, 20]}

grid_search = GridSearchCV(pipe, grid, scoring='roc_auc')
grid_search.fit(df.drop('class', 1), df['class'])

with open('pipe.pkl', 'wb') as file:
    pickle.dump(grid_search, file)
```

Create a pipeline object with a KBest variable selector, and a Random Forest classifier. Then pass it to a grid search routine to optimize across the values of K and max_depth. The entire pipeline including the trained model can then be saved into a pickled object and passed on to another system.

A tale of three ways of working: the config file

```
{
    "data": "./data/arrhythmias.csv",
    "pipeline": {
        "clf": "RandomForestClassifier()",
        "varsel": "SelectKBest()",
        "dim_reduction": ""
    },
    "params": {
        "clf": {
            "max_depth": "NA",
            "max_depth_grid": [10,20]
        },
        "varsel": {
            "k": 5,
            "k_grid": []
        }
    }
}
```

We push as many “decisions” as possible out of the script into a configuration file that is human-readable (e.g., JSON)

There are many ways to do this, and encode different decisions (e.g., tune max_depth but fix value of K).

Which solution works best depends on the use case

Code Version Control (e.g., Github)

Script

```
best_max_depth = GridSearchCV(
    RandomForestClassifier(),
    param_grid={'max_depth': [10, 20]}
).fit(X_train, y_train).best_params_['ma

clf = RandomForestClassifier(
  max_depth=bes                    h).fit(X_train,

vt = SelectKBest(f_classif, k=5).fit(X_t

clf_vt = RandomForestClassifier(max_dept
    vt.transform(X_train), y_train)
```

Sklearn pipeline

```
pipe = Pipeline([
  ('ft', SelectKBest()), ('clf', Randc

grid = {'ft__k':[5, 10], 'clf__max_dep

grid_s                                    grid,
grid_search.fit(df.drop('class', 1), c

with open('pipe.pkl', 'wb') as file:
    pickle.dump(grid_search, file)
```

Config file

```
{
    "data": "./data/arrhythmias.csv",
    "pipeline": {
        "clf": "RandomForestClassifier()",
        "varsel": "SelectKBest()",
        "dim_reduction": ""
    },
    "params": {
        "clf": {
                                  "max_depth"
            "max_depth_grid": [10,20]
        },
        "varsel": {
            "k": 5,
            "k_grid": []
        }
    }
}
```

Which solution works best depends on the use case

Shipping to production

Script

```
best_max_depth = GridSearchCV(
    RandomForestClassifier(),
    param_grid={'max_depth': [10, 20]}
).fit(X_train, y_train).best_params_['ma

clf = RandomForestClassifier(
  max_depth=bes                    h).fit(X_train,

vt = SelectKBest(f_classif, k=5).fit(X_t

clf_vt = RandomForestClassifier(max_dept
    vt.transform(X_train), y_train)
```

Sklearn pipeline

```
pipe = Pipeline([
  ('ft', SelectKBest()), ('clf', Randc

grid = {'ft__k':[5, 10], 'clf__max_dep

grid_s                                  grid,
grid_search.fit(df.drop('class', 1), c

with open('pipe.pkl', 'wb') as file:
    pickle.dump(grid_search, file)
```

Config file

```
{
    "data": "./data/arrhythmias.csv",
    "pipeline": {
        "clf": "RandomForestClassifier()",
        "varsel": "SelectKBest()",
        "dim_reduction": ""
    },
    "params": {
        "clf": {
                                "max_depth"
            "max_depth_grid": [10,20]
        },
        "varsel": {
            "k": 5,
            "k_grid": []
        }
    }
}
```

Which solution works best depends on the use case

Research and exploration of hyperparameters

Script

```
best_max_depth = GridSearchCV(
    RandomForestClassifier(),
    param_grid={'max_depth': [10, 20]}
).fit(X_train, y_train).best_params_['ma

clf = RandomForestClassifier(
  max_depth=bes                    h).fit(X_train,

vt = SelectKBest(f_classif, k=5).fit(X_t

clf_vt = RandomForestClassifier(max_dept
    vt.transform(X_train), y_train)
```

Sklearn pipeline

```
pipe = Pipeline([
  ('ft', SelectKBest()), ('clf', Randc

grid = {'ft__k':[5, 10], 'clf__max_dep

grid_s                                  grid,
grid_search.fit(df.drop('class', 1), c

with open('pipe.pkl', 'wb') as file:
    pickle.dump(grid_search, file)
```

Config file

```
{
    "data": "./data/arrhythmias.csv",
    "pipeline": {
        "clf": "RandomForestClassifier()",
        "varsel": "SelectKBest()",
        "dim_reduction": ""
    },
    "params": {
        "clf": {
                                "max_depth"
            "max_depth_grid": [10,20]
        },
        "varsel": {
            "k": 5,
            "k_grid": []
        }
    }
}
```

Pipelines are a sequence of input/output steps

```
pipe = Pipeline([
  ('ft', SelectKBest()), ('clf', RandomForestClassifier())])
```

```
from kedro.pipeline import node


# Prepare first node
def return_greeting():
    return "Hello"


return_greeting_node = node(func=return_greeting, inputs=None, outputs="my_salutation")
```

https://kedro.readthedocs.io/en/stable/get_started/hello_kedro.html

Pipelines are a sequence of input/output steps

```
pipe = Pipeline([
  ('ft', SelectKBest()), ('clf', RandomForestClassifier())])
```

```
# Prepare second node
def join_statements(greeting):
    return f"{greeting} Kedro!"


join_statements_node = node(
    join_statements, inputs="my_salutation", outputs="my_message"
)
```

https://kedro.readthedocs.io/en/stable/get_started/hello_kedro.html

Pipelines are a sequence of input/output steps

```
pipe = Pipeline([
  ('ft', SelectKBest()), ('clf', RandomForestClassifier())])
```

```
from kedro.pipeline import pipeline

# Assemble nodes into a pipeline
greeting_pipeline = pipeline([return_greeting_node, join_statements_node])

# Create a runner to run the pipeline
runner = SequentialRunner()

# Run the pipeline
print(runner.run(greeting_pipeline, data_catalog))
```

https://kedro.readthedocs.io/en/stable/get_started/hello_kedro.html

A data catalog concept allows tidy reference to datasets

```
from kedro.io import DataCatalog, MemoryDataSet

# Prepare a data catalog
data_catalog = DataCatalog({"my_salutation": MemoryDataSet()})
```

`kedro.extras.datasets.api.APIDataSet` (url[, ...])	`APIDataSet` loads the data from HTTP(S) APIs.
`kedro.extras.datasets.biosequence.BioSequenceDataSet` (...)	`BioSequenceDataSet` loads and saves data to a sequence file.
`kedro.extras.datasets.dask.ParquetDataSet` (...)	`ParquetDataSet` loads and saves data to parquet file(s).
`kedro.extras.datasets.email.EmailMessageDataSet` (...)	`EmailMessageDataSet` loads/saves an email message from/to a file using an underlying filesystem (e.g.: local, S3, GCS).
`kedro.extras.datasets.geopandas.GeoJSONDataSet` (...)	`GeoJSONDataSet` loads/saves data to a GeoJSON file using an underlying filesystem (eg: local, S3, GCS).
`kedro.extras.datasets.holoviews.HoloviewsWriter` (...)	`HoloviewsWriter` saves Holoviews objects to image file(s) in an underlying filesystem (e.g.

`kedro.extras.datasets.holoviews.HoloviewsWriter` (...)	`HoloviewsWriter` saves Holoviews objects to image file(s) in an underlying filesystem (e.g.
`kedro.extras.datasets.json.JSONDataSet` (filepath)	`JSONDataSet` loads/saves data from/to a JSON file using an underlying filesystem (e.g.: local, S3, GCS).
`kedro.extras.datasets.matplotlib.MatplotlibWriter` (...)	`MatplotlibWriter` saves one or more Matplotlib objects as image files to an underlying filesystem (e.g.
`kedro.extras.datasets.networkx.GMLDataSet` (...)	`GMLDataSet` loads and saves graphs to a GML file using an underlying filesystem (e.g.: local, S3, GCS).
`kedro.extras.datasets.networkx.GraphMLDataSet` (...)	`GraphMLDataSet` loads and saves graphs to a GraphML file using an underlying filesystem (e.g.: local, S3, GCS).

https://kedro.readthedocs.io/en/stable/get_started/hello_kedro.html

A standardized folder format also aids with transparency

```
get-started         # Parent directory of the template
├── conf            # Project configuration files
├── data            # Local project data (not committed to version control)
├── docs            # Project documentation
├── logs            # Project output logs (not committed to version control)
├── notebooks       # Project related Jupyter notebooks (can be used for experimental code before moving the code t
├── README.md       # Project README
├── setup.cfg       # Configuration options for `pytest` when doing `kedro test` and for the `isort` utility when d
└── src             # Project source code
```

https://kedro.readthedocs.io/en/stable/get_started/hello_kedro.html

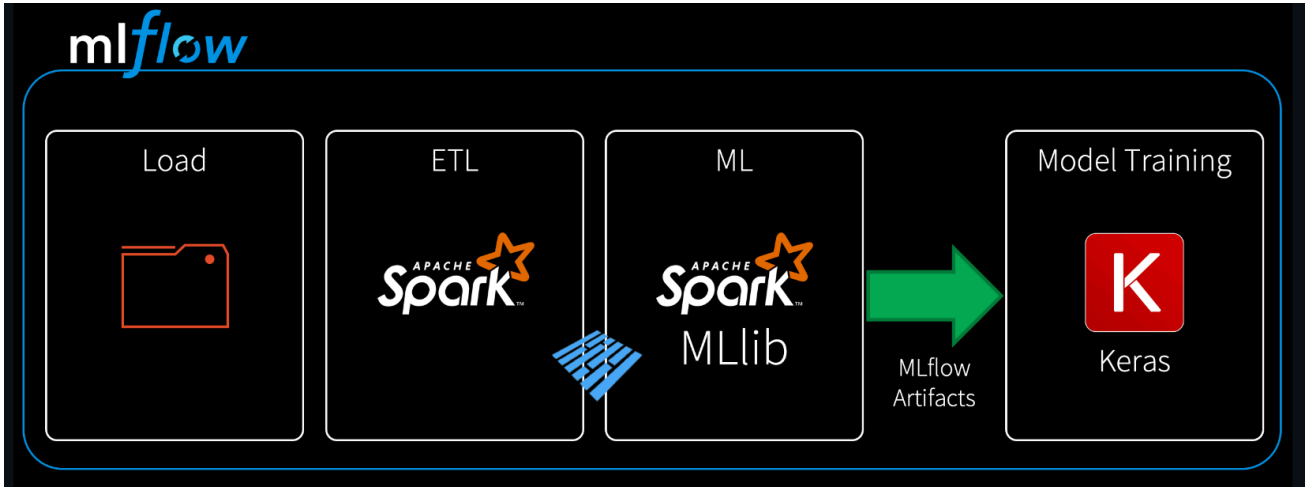
Experiment tracking allows transparency over past runs

					Parameters		Metrics		
	Date	User	Source	Version	alpha	l1_ratio	mae	r2	rmse
☐	2018-06-04 23:00:10	mlflow	train.py	05e956	1	1	0.649	0.04	0.862
☐	2018-06-04 23:00:10	mlflow	train.py	05e956	1	0.5	0.648	0.046	0.859
☐	2018-06-04 23:00:10	mlflow	train.py	05e956	1	0.2	0.628	0.125	0.823
☐	2018-06-04 23:00:09	mlflow	train.py	05e956	1	0	0.619	0.176	0.799
☐	2018-06-04 23:00:09	mlflow	train.py	05e956	0.5	1	0.648	0.046	0.859
☐	2018-06-04 23:00:09	mlflow	train.py	05e956	0.5	0.5	0.628	0.127	0.822
☐	2018-06-04 23:00:09	mlflow	train.py	05e956	0.5	0.2	0.621	0.171	0.801
☐	2018-06-04 23:00:09	mlflow	train.py	05e956	0.5	0	0.615	0.199	0.787
☐	2018-06-04 23:00:09	mlflow	train.py	05e956	0	1	0.578	0.288	0.742
☐	2018-06-04 23:00:09	mlflow	train.py	05e956	0	0.5	0.578	0.288	0.742
☐	2018-06-04 23:00:09	mlflow	train.py	05e956	0	0.2	0.578	0.288	0.742
☐	2018-06-04 23:00:08	mlflow	train.py	05e956	0	0	0.578	0.288	0.742

MLFlow implements an advanced logger together with an ability to execute experiments based on pipelines, and versioned code. The result is complete reproducibility and transparency of experimental runs (reproducible research).

https://mlflow.org/docs/latest/tutorials-and-examples/tutorial.html

Experiment tracking allows transparency over past runs



This level of transparency extends to pipelines that utilize multiple different systems, and could not possibly be tracked as a sequence of scripts.

https://github.com/mlflow/mlflow/tree/master/examples/multistep_workflow

Reminder: why is this Ethical AI?

- Transparency is a core ethical principle
- Imagine an inmate refused parole sues the state for discrimination, and the fact of the matter hinges on the precise data and training technique used to build the model which made the parole decision.
- The data scientist that built this project has left, and the model has been in production for years.
- Complete reproducibility, ability to scrutinize, and maintain/revisit, is needed for true accountability

Recall the Aristotelian “virtue’ ethics approach: “by honing virtuous habits, people will likely make the right choice when faced with ethical challenges”.

https://ethicsunwrapped.utexas.edu/glossary/virtue-ethics