

# Designing reproducible experiments

DEVELOPING MACHINE LEARNING MODELS FOR PRODUCTION

**Sinan Ozdemir**

Data Scientist and Author



# Reproducible experiments

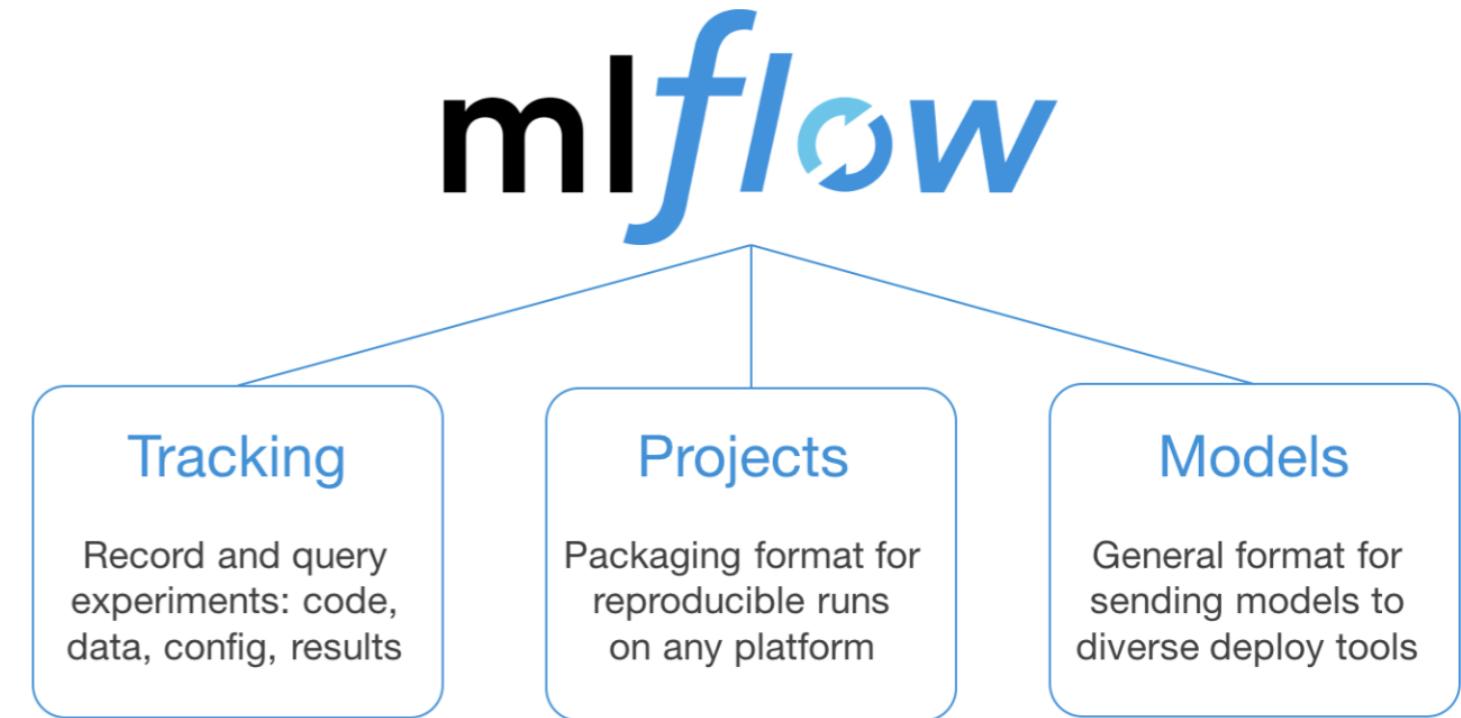
- Ensure accuracy and reliability.
- Easier to replicate results.
- Allow for better collaboration.
- Help reduce the risk of bias in ML models.
- Bolster the integrity of the research process.

# MLflow

An open-source platform for tracking and managing machine learning experiments.

MLflow can be used to:

- Create reproducible ML pipelines
- Track and manage:
  - package dependencies
  - code versions
  - experiment settings
- Allows multiple users to access experiments



<sup>1</sup> <https://www.databricks.com/blog/2018/06/05/introducing-mlflow-an-open-source-machine-learning-platform.html>

# Example of using Mlflow

```
# standard scikit-learn imports
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# new imports from MLflow
import mlflow
import mlflow.sklearn
```

# Example of using MLflow cont.

```
with mlflow.start_run(): # Start an MLflow run assuming we have data prepared

    # Build and train model
    rf = RandomForestClassifier()
    rf.fit(X_train, y_train)

    # Log parameters and model information
    mlflow.log_param("n_estimators", rf.n_estimators)
    mlflow.sklearn.log_model(rf, "model")

    y_pred = rf.predict(X_test) # Evaluate model
    accuracy = accuracy_score(y_test, y_pred)
    mlflow.log_metric("accuracy", accuracy) # log the test accuracy metric
```

# Tracking code

- Logging code versions and changes with MLflow
- Comparing different versions of code
- Identifying which version of the code was used to produce a given set of results
- Easily reproducing experiments
- Making it easy to debug and troubleshoot code

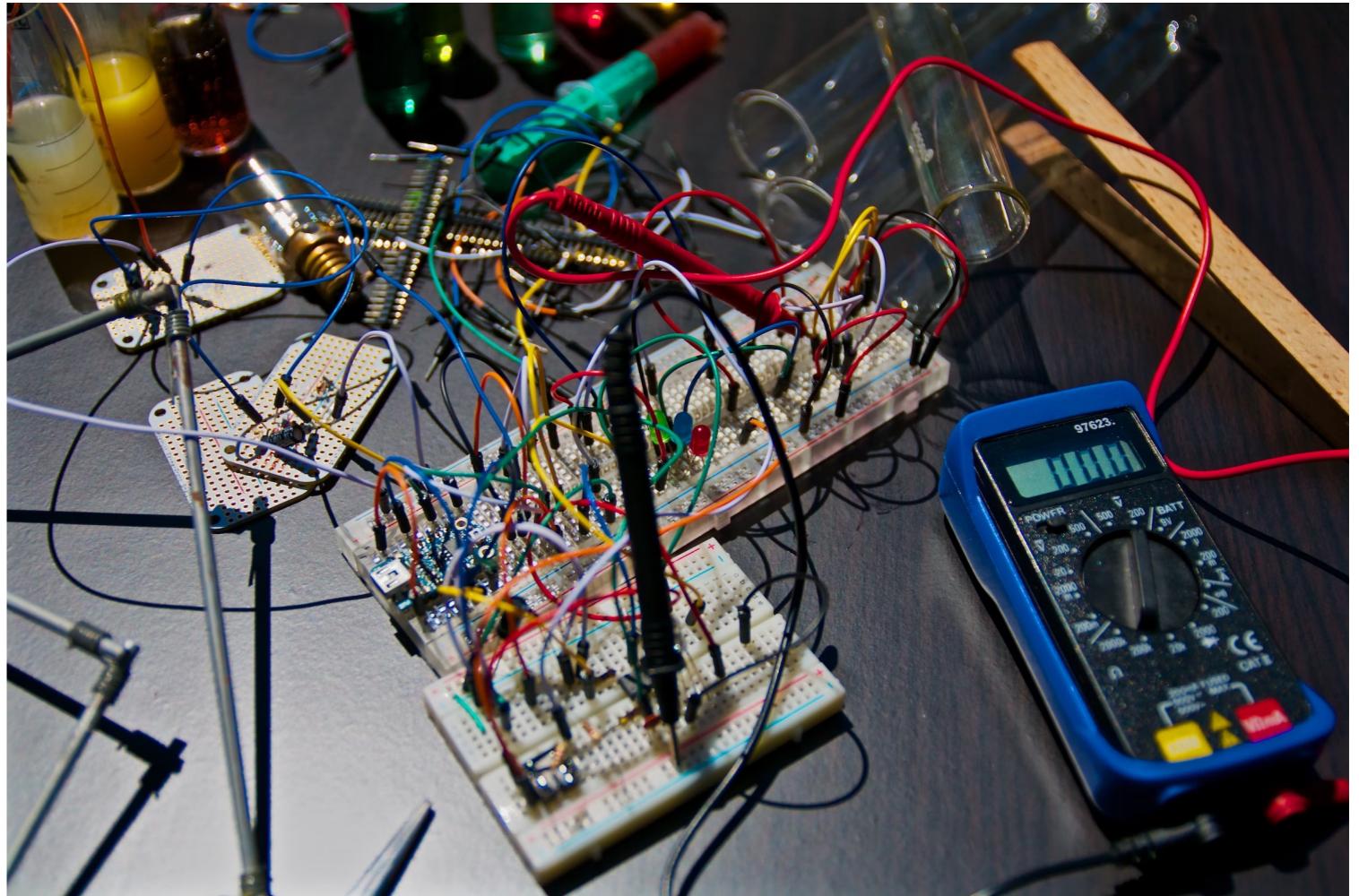
# Model registries

- Centralized repository of models and their metadata
- MLflow can be used to log, store and compare different versions of models
- Reproducing entire ML pipelines
- Used for comparing models
- Ensuring accuracy and reliability of models



# Experiment reproducibility

- Tracking and logging input data, code, and settings
- Reproducing entire ML pipelines
- Building trust in the results of machine learning models
- Allowing others to verify and build upon the work
- Ensuring results are consistent across different runs



# Revisiting documentation

- Documentation is essential for reproducible ML and should include information about:
  - Models' input data.
  - The code used to create models.
  - Any settings used in an experiment.
  - The results of the experiment (which model was chosen, etc)
- Documentation should be accessible
- Meant to be a full record of an experiment

# **Let's practice!**

**DEVELOPING MACHINE LEARNING MODELS FOR PRODUCTION**

# Feature engineering

DEVELOPING MACHINE LEARNING MODELS FOR PRODUCTION

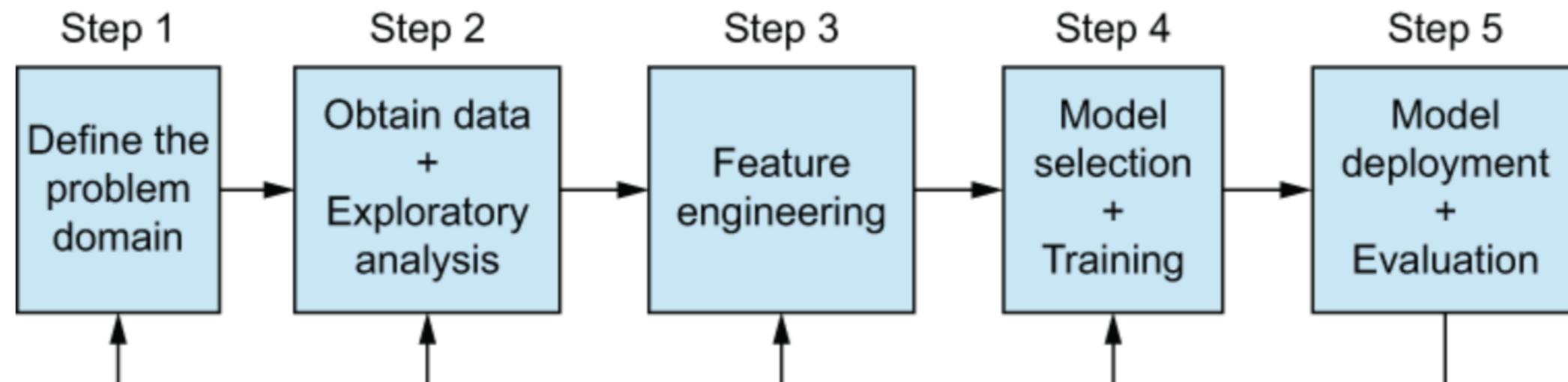


**Sinan Ozdemir**

Data Scientist and Author

# Introduction to feature engineering

- Transforming training data to maximize ML pipeline performance
- Reducing computation complexity
- Examples
  - Aggregating data from multiple sources
  - Constructing new features
  - Applying feature transformations



<sup>1</sup> <https://www.manning.com/books/feature-engineering-bookcamp>

# Aggregating data from multiple sources

- Combining data from different datasets
- Using multiple types of data (e.g. numerical and categorical)

This helps by:

- Improving the accuracy of models
- Enabling the use of more complex models



# Example of data aggregation

```
class DataAggregator:  
    def __init__(self):  
        pass  
  
    def fit(self, X, y=None):  
        return self # nothing to fit  
  
    def transform(self, X, y=None):  
        # Load data from multiple sources  
        data1 = pd.read_csv('data1.csv')  
        data2 = pd.read_csv('data2.csv')  
        data3 = pd.read_csv('data3.csv')  
  
        # Combine data from all sources (including X) into a single data frame  
        aggregated_data = pd.concat([X, data1, data2, data3], axis=0)  
  
        return aggregated_data # Return aggregated data
```

# Feature construction

- Creating new features from existing ones
- Generating new features from pre-existing data
- Improving model performance
- Enhancing model interpretability

# Example of feature construction

```
class FeatureConstructor:  
    def __init__(self):  
        pass  
  
    def fit(self, X, y=None):  
        return self  
  
    def transform(self, X, y=None):  
        # Calculate the mean of each column in the data  
        mean_values = X.mean()  
  
        # Create new features based on the mean values  
        X['mean_col1'] = X['col1'] - mean_values['col1']  
        X['mean_col2'] = X['col2'] - mean_values['col2']  
  
        return X # Return the augmented data set
```

# Feature transformations

Transforming existing features in place

- Normalizing data distributions
- Removing outliers
- Improving model accuracy and performance

```
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

data = load_breast_cancer() # Load dataset
X, y = data.data, data.target
```

```
model = LogisticRegression(random_state=42) # instantiate a model

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42)

model.fit(X_train, y_train) # Fit logistic regression model with
y_pred_no_scaling = model.predict(X_test)
acc_no_scaling = accuracy_score(y_test, y_pred_no_scaling)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

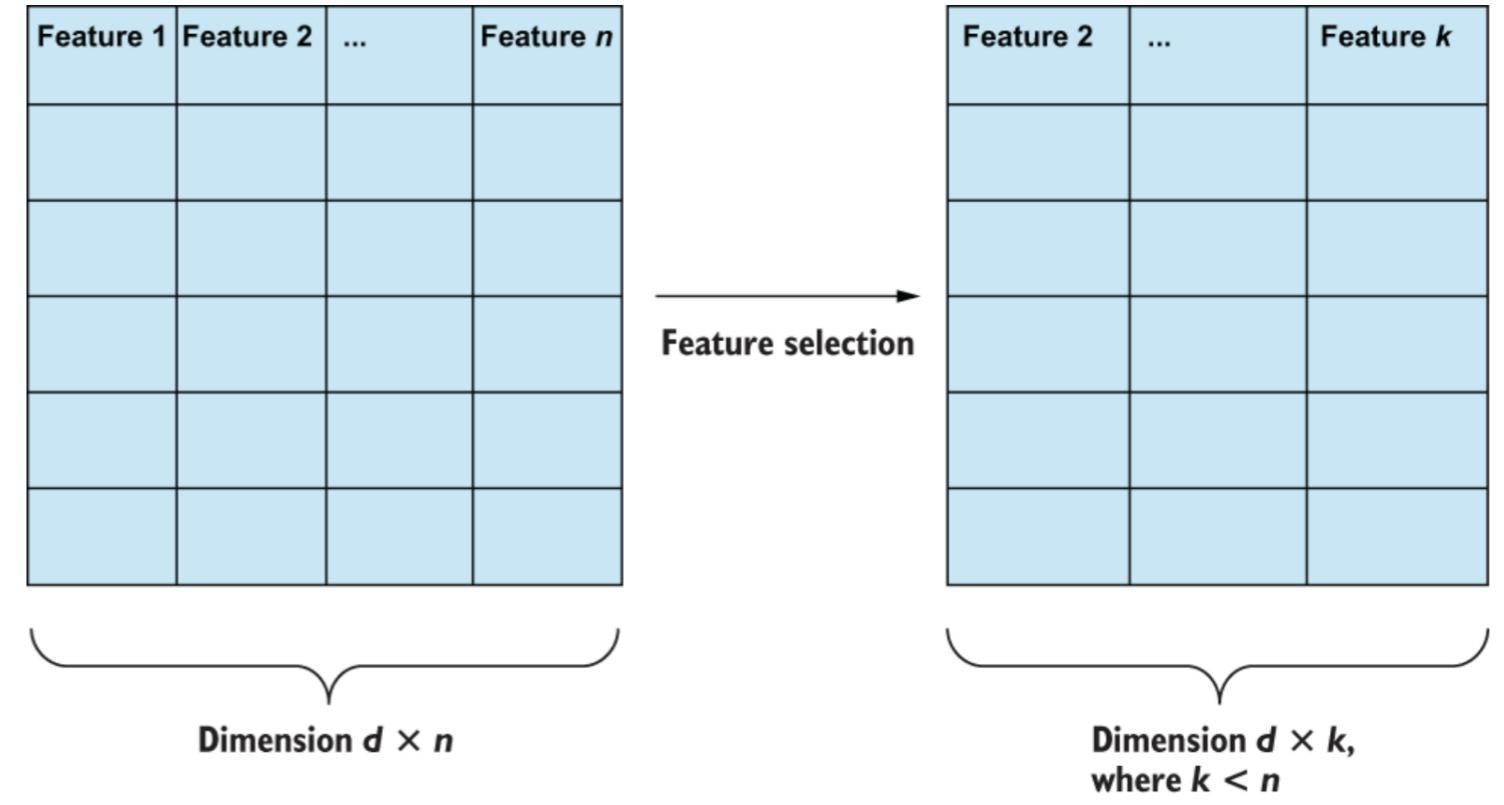
model.fit(X_train_scaled, y_train) # Fit logistic regression mod
y_pred_with_scaling = model.predict(X_test_scaled)
acc_with_scaling = accuracy_score(y_test, y_pred_with_scaling)

# Compare accuracies of models with and without scaling
print(f"Accuracy without scaling: {acc_no_scaling}") # 0.971
print(f"Accuracy with scaling: {acc_with_scaling}") # 0.982
```

# Feature selection

Selecting a subset of features from a larger set of features by removing redundant and irrelevant features

- Reduces overfitting
- Enhances model accuracy and performance
- Improves model interpretability



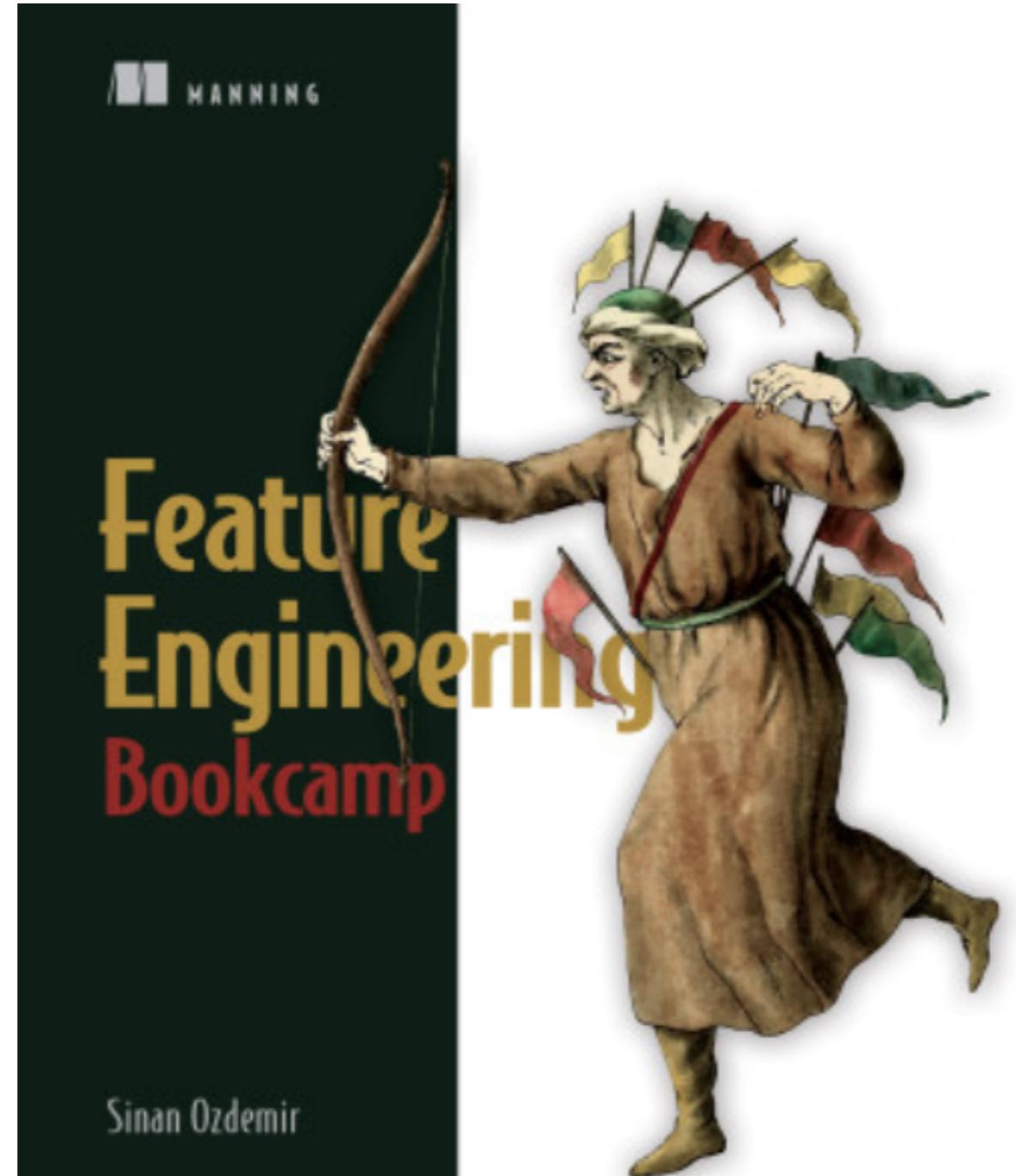
<sup>1</sup> <https://www.manning.com/books/feature-engineering-bookcamp>

# Example of feature engineering cont.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    # Define feature engineering pipeline
    ('aggregate', DataAggregator()), # Aggregate data from multiple sources
    ('construction', FeatureConstructor()), # Feature Construction
    ('scaler', StandardScaler()), # Feature Transformation
    ('select', SelectKBest(chi2, k=10)), # Feature Selection
])
X_transformed = pipeline.fit_transform(X) # Fit and transform data using pipeline
```

# Learn more about feature engineering



# **Let's practice!**

**DEVELOPING MACHINE LEARNING MODELS FOR PRODUCTION**

# Data and model versioning

DEVELOPING MACHINE LEARNING MODELS FOR PRODUCTION



**Sinan Ozdemir**

Data Scientist, Entrepreneur, and Author

# Major & minor versioning

Two types of versioning: major and minor

- Major versioning indicates a big change in data or model



- Minor versioning indicates a small change



With major/minor versions we can see how and what data or model changes were made

# Versioning training data

- Unique major/minor labels or timestamps
- Ensuring reproducibility and traceability of experiments
- Easily going back to a previous version of data
- Seeing how data has changed over time

Example:

1. Data V 1.0 was our initial dataset
2. Data V 1.1 included additional feature transformations
3. Data V 1.2 added a feature selection method
4. Data V 2.0 includes a brand new source of data

# Feature stores

- Central repository for storing and managing different feature versions
- Easily tracking different versions of features
- Using features for different experiments
- Improving collaboration and experimentation integrity
- Reducing duplication of work

# Versioning ML models

- Keeping track of different versions of ML models
- Ensuring reproducibility and traceability of experiments
- Easily rollback to a previous version of model
- Usually matches training data version but not always

Example:

1. Model V 1.0 is our initial model (Random Forest)
2. Model V 1.1 is the same model fine-tuned on Data V 1.1
3. **Model V 2.0** is now a XGBoost Model fine-tuned **Data V 1.2**
4. Model V 2.1 is the XGBoost model fine-tuned on Data V 2.0

# Model stores

- Central repository for storing and managing different model versions
- Easily tracking different versions of models
- Rollback to previous versions

# Example of model versioning with MLflow

```
import mlflow

# Start a new mlflow run
with mlflow.start_run() as run:
    # Log model version as a parameter
    mlflow.log_param("model_version", "1.0")

    # Train and save the model
    model = train_model()
    mlflow.sklearn.log_model(model, "model")
```

# **Let's practice!**

**DEVELOPING MACHINE LEARNING MODELS FOR PRODUCTION**