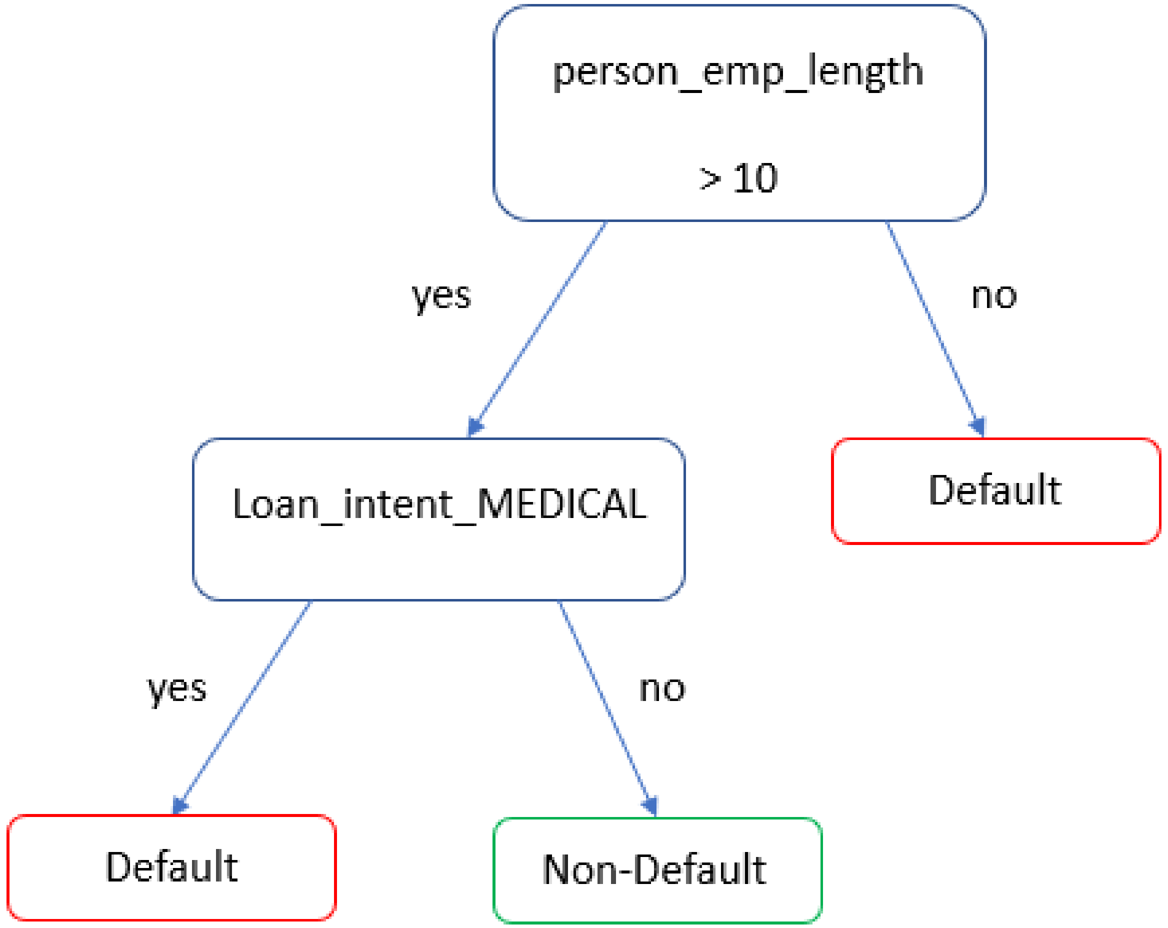
Gradient boosted trees with XGBoost

CREDIT RISK MODELING IN PYTHON

Michael Crabtree
Data Scientist, Ford Motor Company

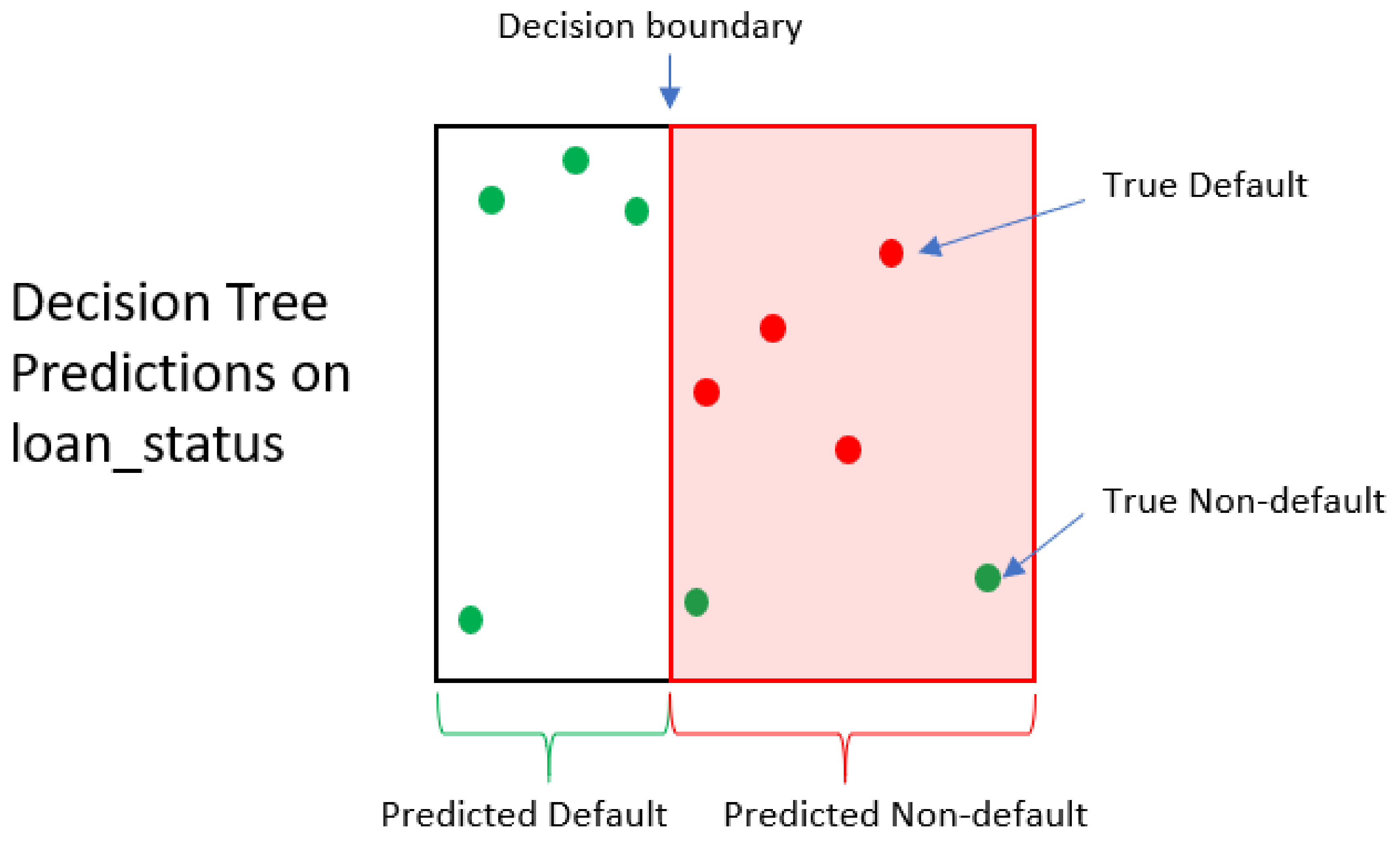
Decision trees

- Creates predictions similar to logistic regression
- Not structured like a regression

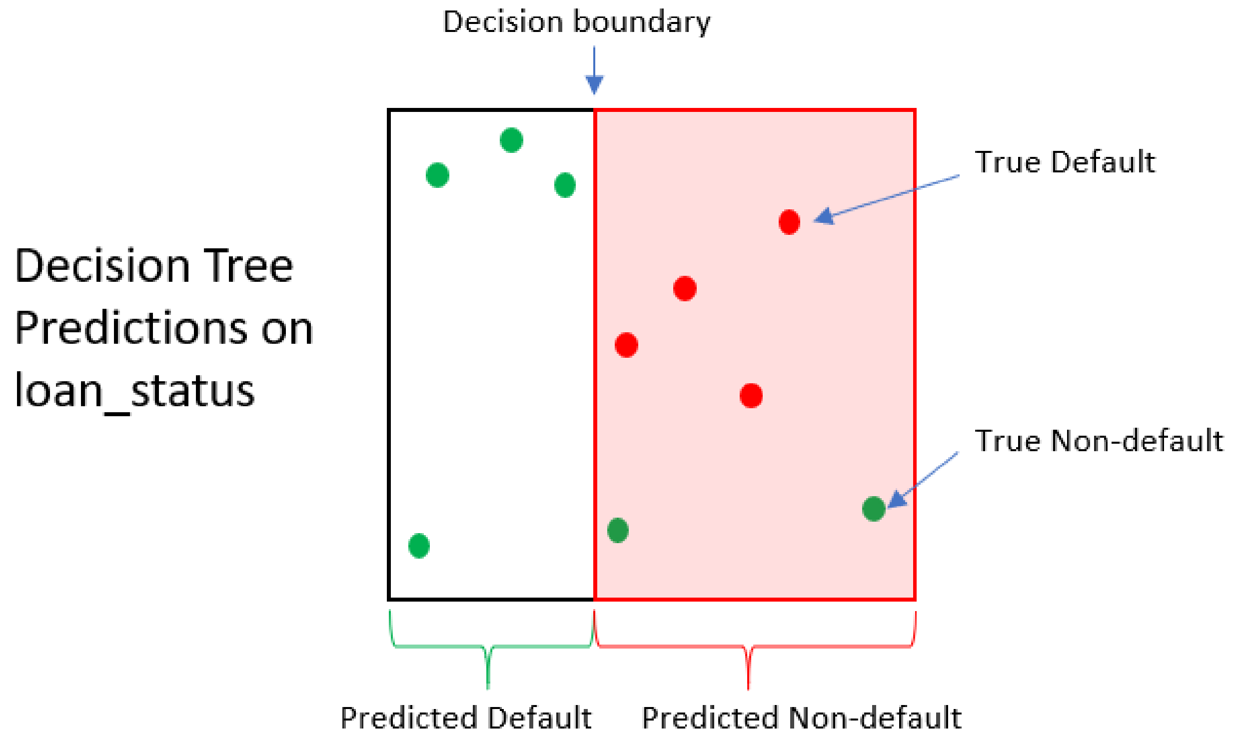


Decision trees for loan status

- Simple decision tree for predicting `loan_status` probability of default



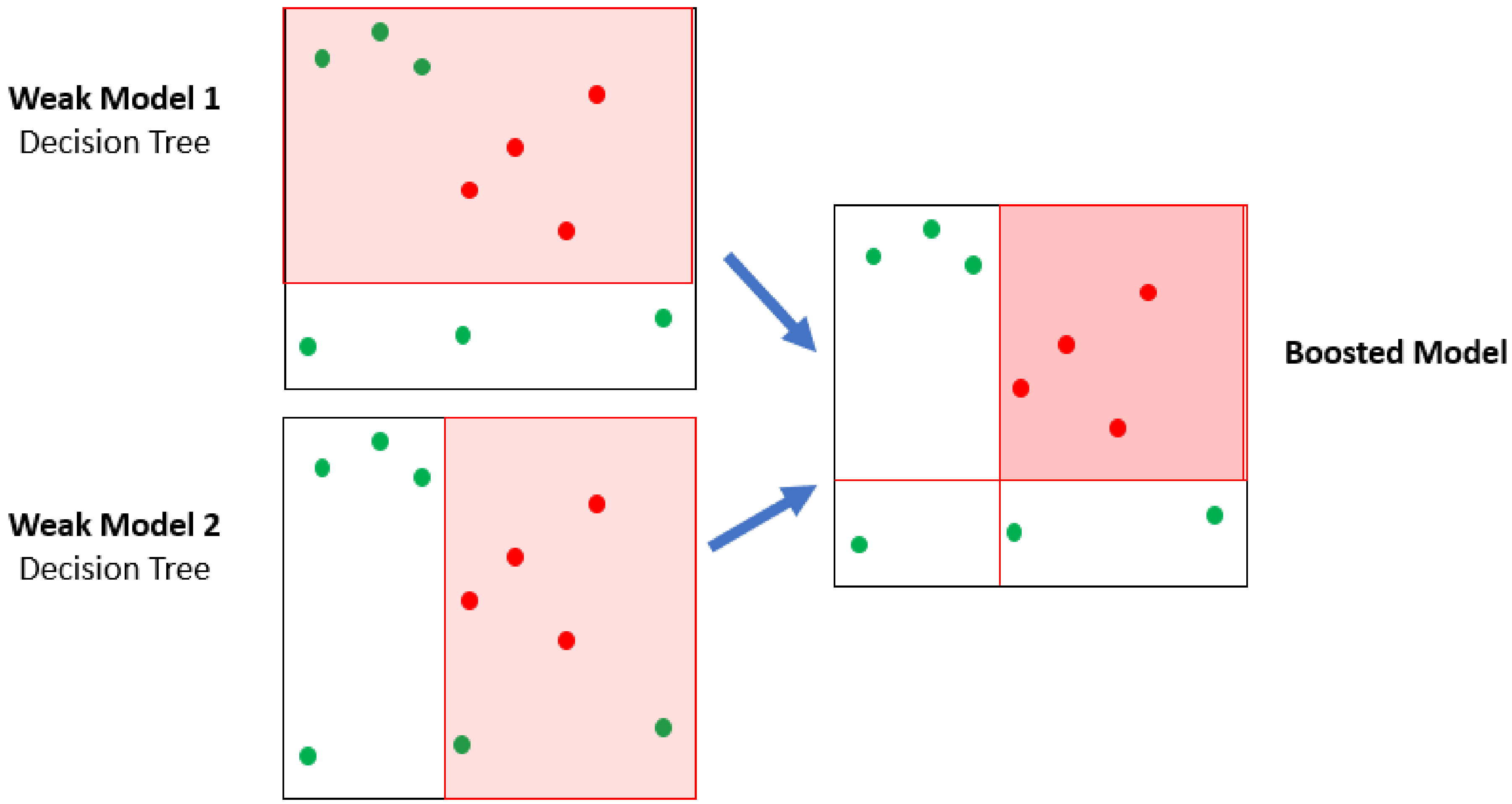
Decision tree impact



Loan	True loan status	Pred. Loan Status	Loan payoff value	Selling Value	Gain/Loss
1	0	1	$1,500	$250	-$1,250
2	0	1	$1,200	$250	-$950

A forest of trees

- XGBoost uses many simplistic trees (ensemble)
- Each tree will be slightly better than a coin toss



Creating and training trees

- Part of the `xgboost` Python package, called `xgb` here
- Trains with `.fit()` just like the logistic regression model

```
# Create a logistic regression model
clf_logistic = LogisticRegression()
# Train the logistic regression
clf_logistic.fit(X_train, np.ravel(y_train))
```

```
# Create a gradient boosted tree model
clf_gbt = xgb.XGBClassifier()
# Train the gradient boosted tree
clf_gbt.fit(X_train,np.ravel(y_train))
```

Default predictions with XGBoost

- Predicts with both `.predict()` and `.predict_proba()`
  - `.predict_proba()` produces a value between `0` and `1`
  - `.predict()` produces a `1` or `0` for `loan_status`

```
# Predict probabilities of default
gbt_preds_prob = clf_gbt.predict_proba(X_test)
# Predict loan_status as a 1 or 0
gbt_preds = clf_gbt.predict(X_test)
```

```
# gbt_preds_prob
array([[0.059, 0.940], [0.121, 0.989]])
# gbt_preds
array([1, 1, 0...])
```

Hyperparameters of gradient boosted trees

- Hyperparameters: model parameters (settings) that cannot be learned from data
- Some common hyperparameters for gradient boosted trees
  - `learning_rate` : smaller values make each step more conservative
  - `max_depth` : sets how deep each tree can go, larger means more complex

```
xgb.XGBClassifier(learning_rate = 0.2,
                  max_depth = 4)
```

Let's practice!

CREDIT RISK MODELING IN PYTHON

Column selection for credit risk

CREDIT RISK MODELING IN PYTHON

Michael Crabtree
Data Scientist, Ford Motor Company

Choosing specific columns

- We've been using all columns for predictions

```
# Selects a few specific columns
X_multi = cr_loan_prep[['loan_int_rate','person_emp_length']]
```

```
# Selects all data except loan_status
X = cr_loan_prep.drop('loan_status', axis = 1)
```

- How you can tell how important each column is
  - Logistic Regression: column coefficients
  - Gradient Boosted Trees: ?

Column importances

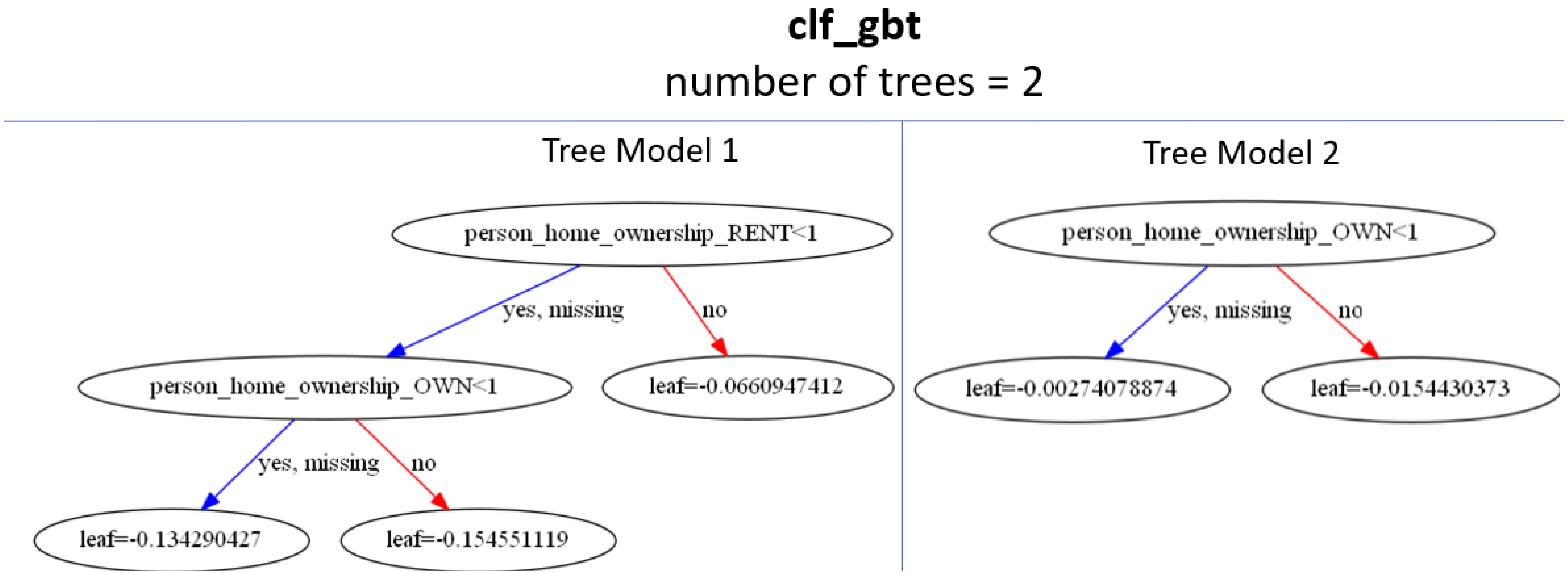
- Use the `.get_booster()` and `.get_score()` methods
  - Weight: the number of times the column appears in all trees

```
# Train the model
clf_gbt.fit(X_train,np.ravel(y_train))
# Print the feature importances
clf_gbt.get_booster().get_score(importance_type = 'weight')
```

```
{'person_home_ownership_RENT': 1, 'person_home_ownership_OWN': 2}
```

Column importance interpretation

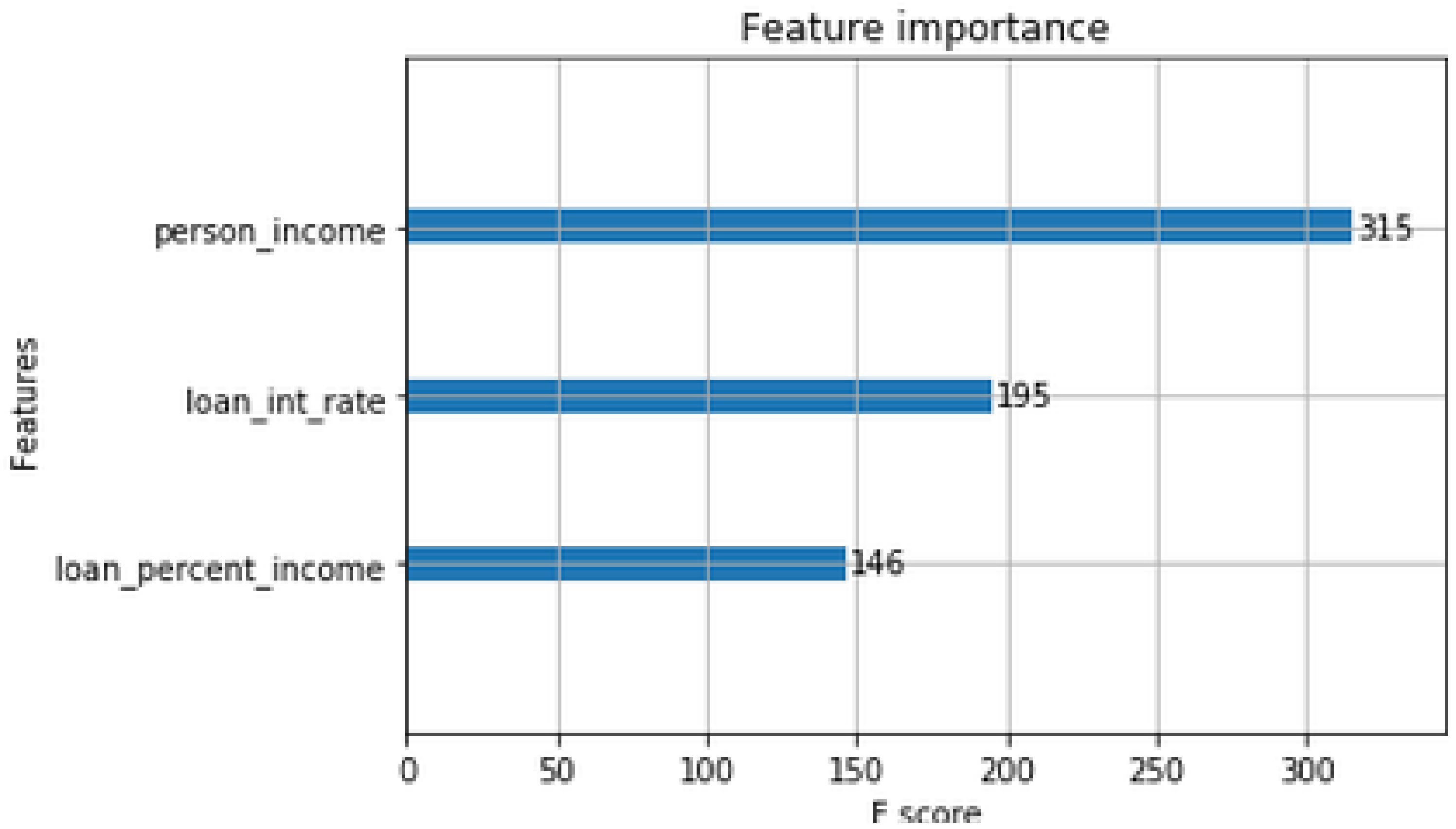
```
# Column importances from importance_type = 'weight'
{'person_home_ownership_RENT': 1, 'person_home_ownership_OWN': 2}
```



Plotting column importances

- Use the `plot_importance()` function

```
xgb.plot_importance(clf_gbt, importance_type = 'weight')
{'person_income': 315, 'loan_int_rate': 195, 'loan_percent_income': 146}
```



Choosing training columns

- Column importance is used to sometimes decide which columns to use for training
- Different sets affect the performance of the models

Columns	Importances	Model Accuracy	Model Default Recall
loan_int_rate, person_emp_length	(100, 100)	0.81	0.67
loan_int_rate, person_emp_length, loan_percent_income	(98, 70, 5)	0.84	0.52

F1 scoring for models

- Thinking about accuracy and recall for different column groups is time consuming
- F1 score is a single metric used to look at both accuracy and recall

$$F1\ Score = 2 * \left(\frac{precision * recall}{precision + recall}\right)$$

- Shows up as a part of the `classification_report()`

	precision	recall	f1-score	support
Non-Default	0.93	0.99	0.96	9198
Default	0.96	0.72	0.82	2586
micro avg	0.93	0.93	0.93	11784
macro avg	0.94	0.85	0.89	11784
weighted avg	0.93	0.93	0.93	11784

Let's practice!

Cross validation for credit models

CREDIT RISK MODELING IN PYTHON

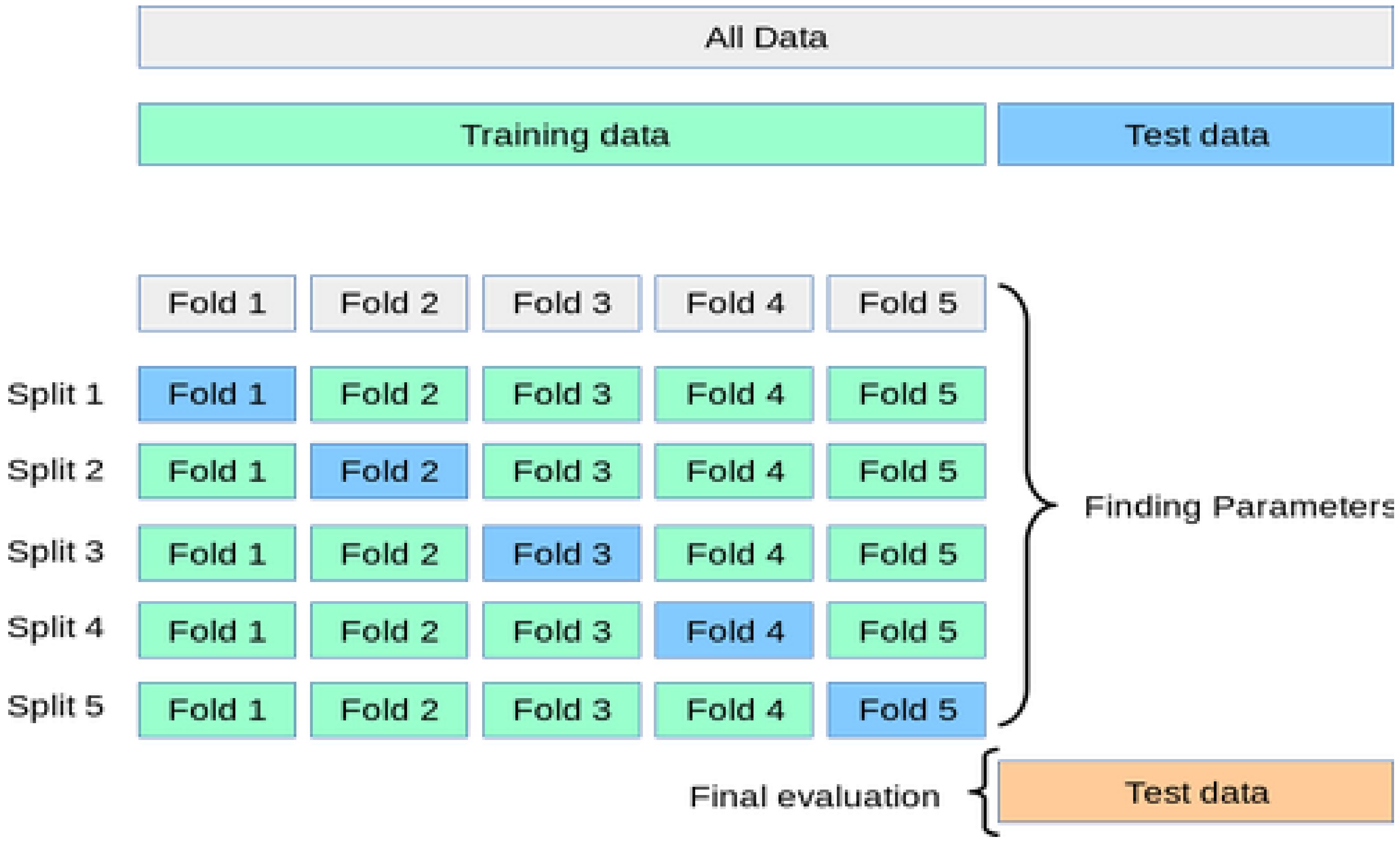
Michael Crabtree
Data Scientist, Ford Motor Company

Cross validation basics

- Used to train and test the model in a way that simulates using the model on new data
- Segments training data into different pieces to estimate future performance
- Uses `DMatrix` , an internal structure optimized for `XGBoost`
- Early stopping tells cross validation to stop after a scoring metric has not improved after a number of iterations

How cross validation works

- Processes parts of training data as (called folds) and tests against unused part
- Final testing against the actual test set



[1] https://scikit [2] learn.org/stable/modules/cross_validation.html

Setting up cross validation within XGBoost

```
# Set the number of folds
n_folds = 2
# Set early stopping number
early_stop = 5
# Set any specific parameters for cross validation
params = {'objective': 'binary:logistic',
          'seed': 99, 'eval_metric':'auc'}
```

- `'binary':'logistic'` is used to specify classification for `loan_status`
- `'eval_metric':'auc'` tells XGBoost to score the model's performance on AUC

Using cross validation within XGBoost

```
# Restructure the train data for xgboost
DTrain = xgb.DMatrix(X_train, label = y_train)
# Perform cross validation
xgb.cv(params, DTrain, num_boost_round = 5, nfold=n_folds,
       early_stopping_rounds=early_stop)
```

- `DMatrix()` creates a special object for `xgboost` optimized for training

The results of cross validation

- Creates a data frame of the values from the cross validation

	train-auc-mean	train-auc-std	test-auc-mean	test-auc-std
0	0.898444	0.002041	0.892701	0.006615
1	0.907534	0.001368	0.899609	0.008587
2	0.914467	0.002170	0.908039	0.007474
3	0.919102	0.000843	0.911437	0.007616
4	0.923488	0.001320	0.914825	0.006873

Cross validation scoring

- Uses cross validation and scoring metrics with `cross_val_score()` function in scikit-learn

```
# Import the module
from sklearn.model_selection import cross_val_score
# Create a gbt model
xg = xgb.XGBClassifier(learning_rate = 0.4, max_depth = 10)
# Use cross valudation and accuracy scores 5 consecutive times
cross_val_score(gbt, X_train, y_train, cv = 5)
```

```
array([0.92748092, 0.92575308, 0.93975392, 0.93378608, 0.93336163])
```

Let's practice!

CREDIT RISK MODELING IN PYTHON

Class imbalance in loan data

CREDIT RISK MODELING IN PYTHON

Michael Crabtree
Data Scientist, Ford Motor Company

Not enough defaults in the data

- The values of `loan_status` are the classes
  - Non-default: `0`
  - Default: `1`

```
y_train['loan_status'].value_counts()
```

loan_status	Training Data Count	Percentage of Total
0	13,798	78%
1	3,877	22%

Model loss function

- Gradient Boosted Trees in `xgboost` use a loss function of log-loss
  - The goal is to minimize this value

$$Log\ Loss = -\frac{1}{N}\sum_{i=1}^{N}[y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i)]$$

True loan status	Predicted probability	Log Loss
1	0.1	2.3
0	0.9	2.3

- An inaccurately predicted default has more negative financial impact

The cost of imbalance

- A false negative (default predicted as non-default) is much more costly

Person	Loan Amount	Potential Profit	Predicted Status	Actual Status	Losses
A	$1,000	$10	Default	Non-Default	-$10
B	$1,000	$10	Non-Default	Default	-$1,000

  - Log-loss for the model is the same for both, our actual losses is not

Causes of imbalance

- Data problems
  - Credit data was not sampled correctly
  - Data storage problems
- Business processes:
  - Measures already in place to not accept probable defaults
  - Probable defaults are quickly sold to other firms
- Behavioral factors:
  - Normally, people do not default on their loans
    - The less often they default, the higher their credit rating

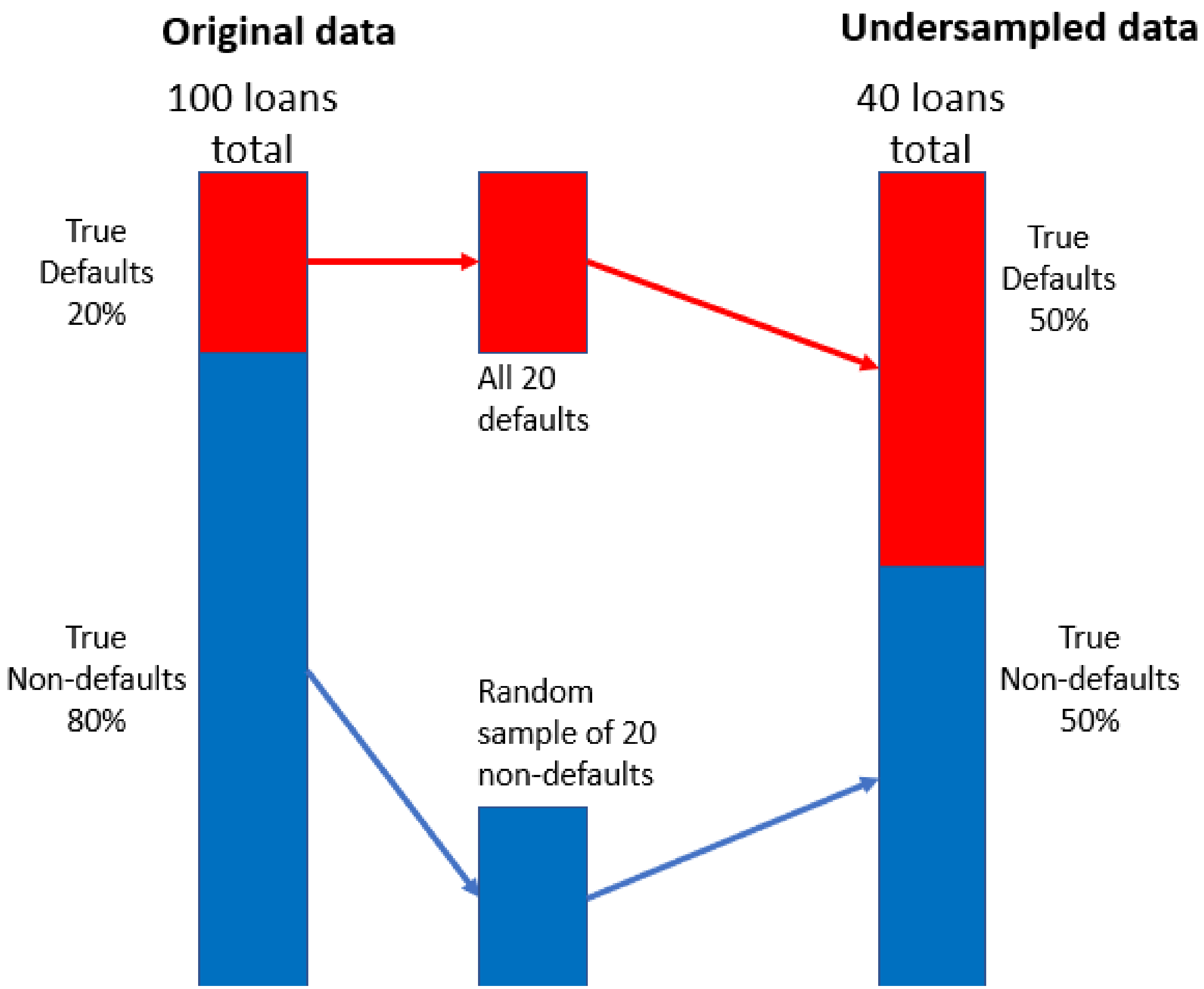
Dealing with class imbalance

- Several ways to deal with class imbalance in data

Method	Pros	Cons
Gather more data	Increases number of defaults	Percentage of defaults may not change
Penalize models	Increases recall for defaults	Model requires more tuning and maintenance
Sample data differently	Least technical adjustment	Fewer defaults in data

Undersampling strategy

- Combine smaller random sample of non-defaults with defaults



Combining the split data sets

- Test and training set must be put back together
- Create two new sets based on actual `loan_status`

```
# Concat the training sets
X_y_train = pd.concat([X_train.reset_index(drop = True),
                       y_train.reset_index(drop = True)], axis = 1)
# Get the counts of defaults and non-defaults
count_nondefault, count_default = X_y_train['loan_status'].value_counts()
# Separate nondefaults and defaults
nondefaults = X_y_train[X_y_train['loan_status'] == 0]
defaults = X_y_train[X_y_train['loan_status'] == 1]
```

Undersampling the non-defaults

- Randomly sample data set of non-defaults
- Concatenate with data set of defaults

```
# Undersample the non-defaults using sample() in pandas
nondefaults_under = nondefaults.sample(count_default)
# Concat the undersampled non-defaults with the defaults
X_y_train_under = pd.concat([nondefaults_under.reset_index(drop = True),
                             defaults.reset_index(drop = True)], axis=0)
```

Let's practice!

CREDIT RISK MODELING IN PYTHON