03b_pipeline

June 21, 2025

1 An initial training pipeline

[ ]:
```
%load_ext dotenv
%dotenv
import os
import sys
sys.path.append(os.getenv('SRC_DIR'))
import dask.dataframe as dd
import pandas as pd
import numpy as np

from glob import glob
ft_dir = os.getenv("FEATURES_DATA")
ft_glob = glob(ft_dir+'/*.parquet')
df = dd.read_parquet(ft_glob).compute().reset_index().dropna()
```

[ ]:
```
cat_file = os.path.join(
    os.getenv("PRICE_CSV_DATA"),
    'symbols_valid_meta.csv'
)
cat_df = (pd.read_csv(cat_file)
          .rename(columns = {'Symbol': 'ticker'})[['ticker', 'Listing␣
 ↪Exchange', 'Market Category']]
          )
df = df.merge(cat_df, on = 'ticker', how = 'left')
```

1.1 Preprocessing

- Previously, we produced a features data set.
- Most times, one or more preprocessing steps steps will be applied to data.
- The most practical way to apply them is by arranging them in `Pipeline` objects, wchich are sequential transformations applied to data.
- It is convenient for us to label these transformations and there is a standard way of doing so.

1.2 Transformations

- Transformations are classes that implement `fit` and `transform` methods.

1.2.1 StandardScaler

- For example, transform a numerical variable by standardizing it.
- Standardization is removing the mean value of the feature and scale it by dividing non-constant features by their standard deviation.

$$z = \frac{x - \mu}{\sigma}$$

- Using StandardScaler, one can do the following:

[ ]:
```
df.columns
```

[ ]:
```
df = (df.assign(
        returns = lambda x: x['Close']/x['Close_lag_1'] - 1,
        positive_return = lambda x: 1.0*(x['returns'] > 0),
        hi_lo = lambda x: x['High'] - x['Low'],
        op_cl = lambda x: x['Close'] - x['Open']
    ).groupby(['ticker'], group_keys=False).apply(
        lambda x: x.assign(target = x['positive_return'].shift(-1))
    )
    .reset_index(drop=True)
    .dropna(subset = ['target'])
    )
df
```

[ ]:
```
# Create a StandardScaler object

from sklearn.preprocessing import StandardScaler
std_scaler = StandardScaler()


# Fit the StandardScaler object with the returns data
std_scaler.fit(returns)
```

[ ]:
```
# Transform the returns data using the fitted scaler

scaled_returns_np = std_scaler.transform(returns)
scaled_returns = pd.DataFrame(scaled_returns_np, columns=returns.columns)
scaled_returns.describe()
```

1.2.2 OneHotEncoder

- Categorical features can be encoded as numerical values using `OneHotEncoder`.

[ ]:
```
df['Listing Exchange'].value_counts().plot(kind = 'bar')
```

- Use OneHotEncoder to encode a categorical variable as numerical.
- Important parameters:

  - `categories` allows you to specify the categories to work with.
  - `drop`: we can drop the `'first'` value (dummy encoding) or `'if_binary'`, a convenience setting for binary values.
  - `handle_unknown` allows three options, `'error'`, `'ignore'`, and `'infrequent_if_exist'`, depending on what we want to do with new values.

```
[ ]: from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
onehot.fit(df[['Listing Exchange']])
```

```
[ ]: listing_enc = onehot.transform(df[['Listing Exchange']])
listing_enc.toarray()
```

2 Pipelines

- It is impractical and costly to manipulate data "by hand".
- To manage data preprocessing steps within the cross-validation process use `Pipeline` objects.
- A `Pipeline` object allows us to sequentially apply transformation steps and, if required, a predictor.
- `Pipeline` objects compose transforms, i.e., classes that implement `transform` and `fit` methods.
- The purpose of `Pipeline` objects is to ensemble transforms and predictors to be used in cross-validation.
- A `Pipeline` is defined by a list of tuples.
- Each tuple is composed of `("name", <ColumnTransformer>)`, the name of the step and the `<ColumnTransformer>` function of our chosing.

```
[ ]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, log_loss, cohen_kappa_score,␣
 ↪f1_score
```

```
[ ]: pipe1 = Pipeline(
    [
        ('onehot', OneHotEncoder(handle_unknown='ignore')),
        ('knn', DecisionTreeClassifier(criterion = 'entropy', max_depth=3))

    ]
)
pipe1
```

```
[ ]: X0 = df[['Listing Exchange', 'Market Category']]
Y0 = df['target']
X0_train, X0_test, Y0_train, Y0_test = train_test_split(X0, Y0, test_size=0.2,␣
 ↪random_state=42)
```

```
pipe1.fit(X0_train, Y0_train)
```

[ ]:
```
Y_pred_train = pipe1.predict(X0_train)
Y_pred_test = pipe1.predict(X0_test)
```

[ ]:
```
Y_poba_train = pipe1.predict_proba(X0_train)
Y_proba_test = pipe1.predict_proba(X0_test)
```

[ ]:
```
res = {
    'accuracy_score_train': accuracy_score(Y0_train, Y_pred_train),
    'accuracy_score_test': accuracy_score(Y0_test, Y_pred_test),
    'cohen_kappa_train': cohen_kappa_score(Y0_train, Y_pred_train),
    'cohen_kappa_test': cohen_kappa_score(Y0_test, Y_pred_test),
    'log_loss_train': log_loss(Y0_train, Y_poba_train),
    'log_loss_test': log_loss(Y0_test, Y_proba_test),
    'f1_score_train': f1_score(Y0_train, Y_pred_train),
    'f1_score_test': f1_score(Y0_test, Y_pred_test)
}
res
```

- The model does not show great performance, but the pipeline shows results.
- Below, we expand the pipeline to include more variables, and further we will work with more robust model selection pipelines.

2.1 ColumnTransformer

- Use `ColumnTransformer` to apply transformers to specific columns of a DataFrame.
- In this case, we will scale numeric variables and apply one-hot encoding to categorical columns.

[ ]:
```
from sklearn.compose import ColumnTransformer
```

[ ]:
```
transformer = ColumnTransformer(
    transformers=[
        ('numeric_transfomer', StandardScaler(), ['returns', 'Volume', 'op_cl',␣
 ↪'hi_lo'] ),
        ('onehot', OneHotEncoder(handle_unknown='infrequent_if_exist'),␣
 ↪['Listing Exchange', 'Market Category']),
    ], remainder='drop'
)

pipe = Pipeline(
    [
        ('preproc', transformer),
        ('decisiontree', DecisionTreeClassifier(criterion = 'entropy',␣
 ↪max_depth=3))
    ]
```

```
)
```

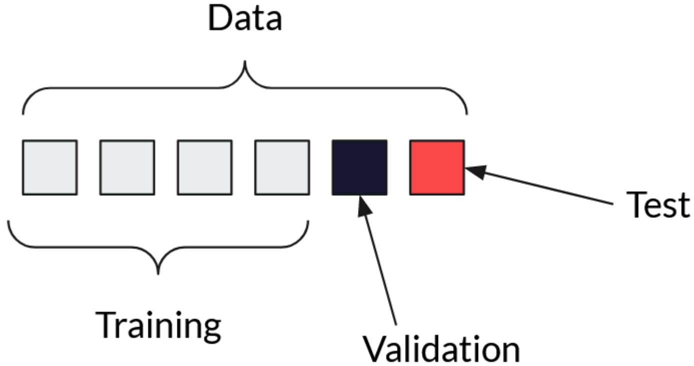
3 Model Selection

The model selection process is an iterative process in which :

- Select schema and load data.
- Define a pipeline and its (hyper) parameters.
  - Use ColumnTransformers to transform numeric and cateogrical variables.
  - Hyperparameters can be defined independently of code.
- Implement a splitting strategy.
  - Use cross_validate to select several metrics and operational details.
- Measure performance.
  - Select metrics
- Repeat

3.1 Training, Validation, Testing Split

- The first spliting strategy is to use a training, validation, and test set.
- Training set will be used to fit the model.
- Validation set is used to evaluate hyperparameter choice.
- Testing set is used to evaluate performance on data the model has not yet seen.
- In this case we want to compare two models:
  - Decision Tree with 3 minumum samples per leaf.
  - Decision Tree with 10 minimum samples per leaf.



3.2 Setting parameters in pipeline steps

- One can obtain the parameters of a pipeline with `pipe.get_params()`.
- We can set any parameter of a pipeline with `pipe.set_parames(**kwargs)`.
- The input `**kwargs` is a dictionary of the params to be modified. Params of the steps are labeled with the name of the step followed by `__` and the name of the parameter.

- There are a few steps that we will repeat:

  - Fit the candidate model on training data.
  - Predict on training and test data.
  - Compute training and test performance metrics.
  - Return.

- We encapsulate this procedure in a function.

[ ]:
```
def evaluate_model(clf, X_train, Y_train, X_test, Y_test):
    clf.fit(X_train, Y_train)
    Y_pred_train = clf.predict(X_train)
    Y_pred_test = clf.predict(X_test)
    Y_proba_train = clf.predict_proba(X_train)
    Y_proba_test = clf.predict_proba(X_test)
    performance_metrics = {
        'log_loss_train': log_loss(Y_train, Y_proba_train),
        'log_loss_test': log_loss(Y_test, Y_proba_test),
        'cohen_kappa_train': cohen_kappa_score(Y_train, Y_pred_train),
        'cohen_kappa_test': cohen_kappa_score(Y_test, Y_pred_test),
        'f1_score_train': f1_score(Y_train, Y_pred_train),
        'f1_score_test': f1_score(Y_test, Y_pred_test),
        'accuracy_score_train': accuracy_score(Y_train, Y_pred_train),
        'accuracy_score_test': accuracy_score(Y_test, Y_pred_test),
    }
    return performance_metrics
```

[ ]:
```
# Schema
X = df[['returns', 'op_cl', 'hi_lo', 'Volume', 'Listing Exchange', 'Market␣
 ↪Category']]
Y = df['target']

# Split the data
X_rest, X_test, Y_rest, Y_test = train_test_split(X, Y, test_size=0.2,␣
 ↪random_state=42)
X_train, X_validate, Y_train,  Y_validate = train_test_split(X_rest, Y_rest,␣
 ↪test_size=0.2, random_state=42)
```

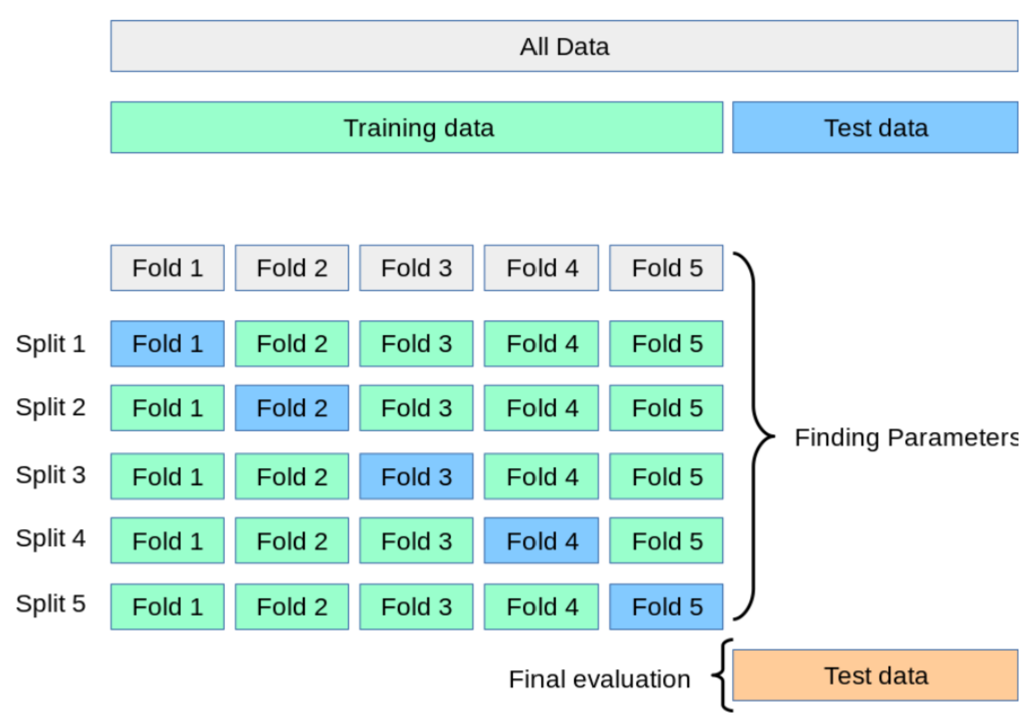
[ ]:
```
# Evaluate hyperparameter configuration 2
pipe_d3 = pipe.set_params(**{'decisiontree__max_depth': 3})
res_d3 = evaluate_model(pipe_d3, X_train, Y_train, X_validate, Y_validate)
res_d3
```

```python
# Evaluate hyperparameter configuration 2
pipe_d15 = pipe.set_params(**{'decisiontree__max_depth':15})
res_d15 = evaluate_model(pipe_d15, X_train, Y_train, X_validate, Y_validate)
res_d15
```

3.3 Cross-Validation

- Cross-validation is a resampling method.
- It is an iterative method applied to training data.
- Training data is divided into folds.
- Each fold is used once as a validation set and the rest of the folds are used for training.
- Test data is used for final evaluation.

From Scikit's Documentation, the diagram below shows the data divisions and folds during the cross-validation process.



There are two functions that can be used for calculating cross-validation performance scores: `cross_val_score()` and `cross_validate()`. The first function, `cross_val_score()`, is a convenience function to get quick perfromance calculations. We will discuss `cross_validate()` as it offers advantages over `cross_val_score()`.

3.4 Obtaining metrics

- Use `cross_validate` to measure one or more performance metrics and operational details.
- There are two advantages of using this function. From Scikit's documentation:
  - It allows specifying multiple metrics for evaluation.

- It returns a dict containing fit-times, score-times (and optionally training scores, fitted estimators, train-test split indices) in addition to the test score.

```
[ ]: from sklearn.model_selection import cross_validate
     scoring = ['accuracy', 'f1', 'precision', 'recall', 'roc_auc', 'neg_log_loss',␣
      ↪'neg_brier_score']
     d3_dict = cross_validate(pipe_d3, X, Y, cv=5, scoring = scoring,␣
      ↪return_train_score = True)
```

In DataFrame form:

```
[ ]: pd.DataFrame(d3_dict)
```

```
[ ]: d15_dict = cross_validate(pipe_d15, X, Y, cv=5, scoring = scoring,␣
      ↪return_train_score = True)
     pd.DataFrame(d15_dict)
```

4 About Performance

- Notice that in order to acquire information about our model and continue development, we are spending resources: time, electricity, equipment use, etc. As well, we are generating data and binary objects that implement our models (fitted `Pipeline` objects, for example).
- For certain applications, operating performance (latency or `'score_time'`) may be as important or more important than predictive performance metrics.
- Every experiment throws important information and we can log them, as well as run them systematically.

```
[ ]: pd.DataFrame(d15_dict).mean()
```