Data Science Fundamentals

Lecture 4. Feature Selection, Cross Validation & Grid Search.

Fernando Lovera
Mario Verstraeten

General Steps for ML

General Steps for ML

- Import the correct libraries
- Load in the data
- Do some exploratory data analysis
- Prepare the data for ML
- Split the data into testing & training sets
- Further prepare the data
- Chose a few models & compare the results to each other

Import the correct libraries

```
# Common imports
import numpy as np
import pandas as pd

# ML library
import sklearn

# To plot pretty figures
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

# Optional: If you want to visualize a sklearn Pipeline
from sklearn import set_config
set_config(display="diagram")
```

Load in the Data

If the data is stored somewhere online:

```
# Download the data and store it locally:
url = "https://github.com/ageron/data/raw/main/housing.tgz"
urllib.request.urlretrieve(url, local_path_to_save_file)
```

(Optional) Extract the data if it's archived:

```
with tarfile.open(local_path_to_save_file) as f:
            f.extractall(path="local_path_to_extract_file")
```

Read in the data using pandas:

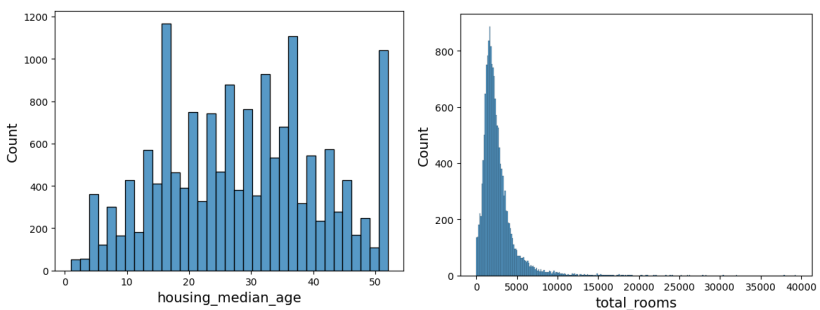
```
pd.read_csv(f"{local_path_to_extract_file}.csv")
```

Do Some EDA

```
df = pd.Dataframe()

# On a Pandas Dataframe you can do:
df.info()
df.head()
df.describe()

# Check the values for categorical columns:
df['a_categorical_column'].value_counts()
```

Plot a histogram for each numerical attribute

```
sns.histplot(data=df, x="a_specific_column" )
```



Look at the correlations between the attributes

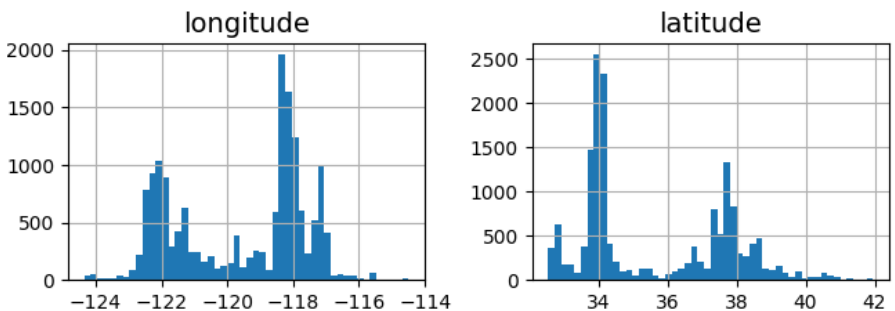
```
corr_matrix = df.corr()
corr_matrix["a_specific_column"].sort_values(ascending=False)
```

```
median_house_value    1.000000
median_income         0.688075
ratio                 0.643892
total_rooms           0.134153
housing_median_age    0.105623
households            0.065843
total_bedrooms        0.049686
population           -0.024650
longitude            -0.045967
latitude             -0.144160
```

Do Some EDA

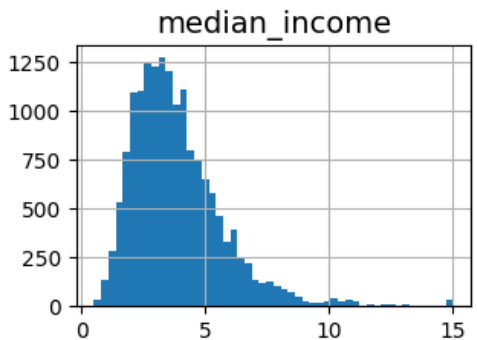
Things to look out for!

Check if the numerical columns have the same scaling



Left one: (-124.35, -114.49) ;
Right one: (32.55, 41.95)

Check if the histograms are 'tail heavy' (they extend much farther to the right of the median than to the left.)



Prepare the Data for ML

Operations on the whole dataset:

- Dealing with outliers
- Dealing with categorical data

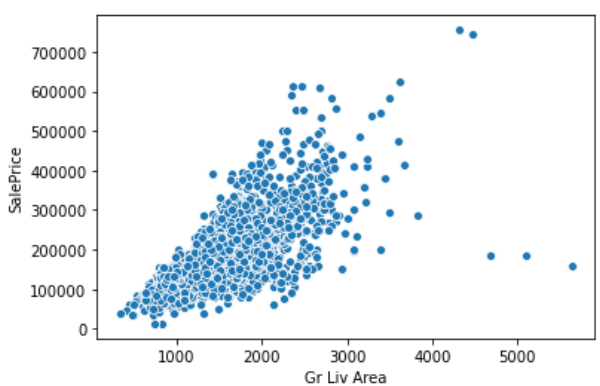
Splitting the data in Test-Train sets

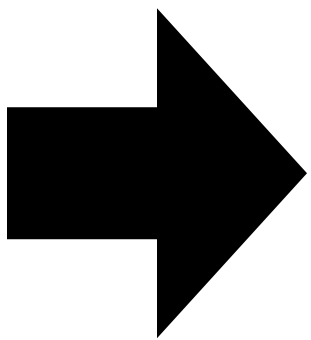
Operations ONLY on the training set:

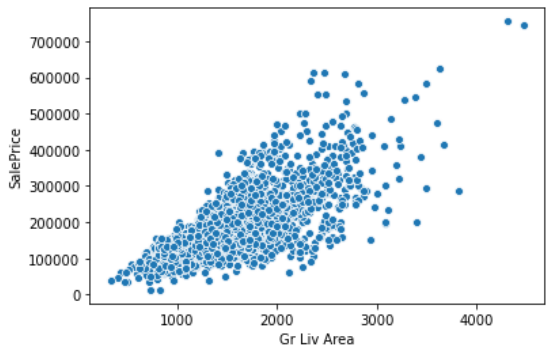
- Dealing with missing data
- Dealing with features scaling
- Dealing with tail-heavy (skewed) numerical data

Prepare the Data for ML

Dealing with outliers








```
df = df[(df['Gr Liv Area']>4000) & (df['SalePrice']<400000)]
```

OR

```
ind_drop = df[(df['Gr Liv Area']>4000) & (df['SalePrice']<400000)].index
df = df.drop(ind_drop,axis=0)
```

Prepare the Data for ML

Dealing with categorical data

```
1 from sklearn.preprocessing import OneHotEncoder
2
3 cat_encoder = OneHotEncoder()
4 housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
5 housing_cat_1hot
```

Notice that the output is a SciPy sparse matrix, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After one-hot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row. Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the location of the nonzero elements. You can use it mostly like a normal 2D array, but if you really want to convert it to a (dense) NumPy array, just call the toarray() method:

```
housing_cat_1hot.toarray()
✓ 0.0s
```

```
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       ...,
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.]])
```

Create Test-Train-(Validation) Set

Define the test-train split size (20% test, 80% train)

```
1 from sklearn.model_selection import train_test_split
2
3 train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

Is the splitting between test and training set fair?

Let's take a look:

```
1 housing['ratio'] = housing.apply(lambda x: x['income_cat'] / len(housing),axis=1)
2 test_set['ratio'] = test_set.apply(lambda x: x['income_cat'] / len(test_set),axis=1)
3 train_set['ratio'] = train_set.apply(lambda x: x['income_cat'] / len(train_set),axis=1)
4
5 print(f'PROPORTION OF TARGET IN THE ORIGINAL DATA\n{housing["income_cat"].value_counts() / len(
6       f'PROPORTION OF TARGET IN THE TRAINING SET\n{train_set["income_cat"].value_counts() / le
7       f'PROPORTION OF TARGET IN THE TEST SET\n{test_set["income_cat"].value_counts() / len(tes
```

Create Test-Train-(Validation) Set

```
PROPORTION OF TARGET IN THE ORIGINAL DATA
3    0.350581
2    0.318847
4    0.176308
5    0.114438
1    0.039826
Name: income_cat, dtype: float64

PROPORTION OF TARGET IN THE TRAINING SET
3    0.351926
2    0.321705
4    0.174358
5    0.112827
1    0.039184
Name: income_cat, dtype: float64

PROPORTION OF TARGET IN THE TEST SET
3    0.345203
2    0.307413
4    0.184109
5    0.120882
1    0.042393
Name: income_cat, dtype: float64
```

The ratio's in the test and validation are close compared to the original (full) dataset... but not the same. This is because the splitting did not consider the inherent distribution of the 'income_cat' column

Create Test-Train-(Validation) Set

We use 'stratified sampling' to make sure that the split is fair and resembles the distribution of the 'income_cat' column of the full dataset. We also set the 'shuffle' parameter to true, to make sure that the data is shuffled before the splitting occurs

```
1 from sklearn.model_selection import StratifiedShuffleSplit
2
3 strat_train_set, strat_test_set = train_test_split(
4     housing, test_size=0.2, stratify=housing["income_cat"], random_state=42, shuffle=True)
```

Create Test-Train-(Validation) Set

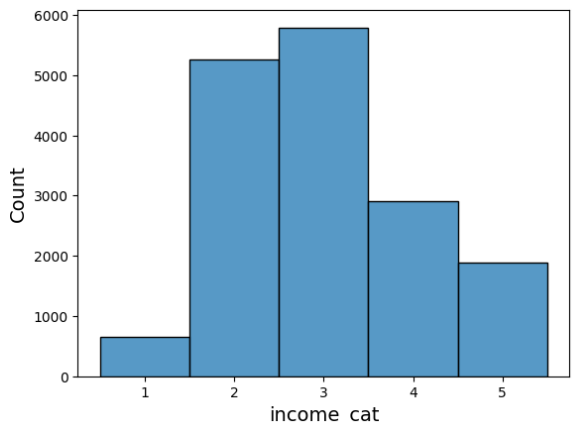
The result: An 'accurate' split

```
PROPORTION OF TARGET IN THE ORIGINAL DATA
3    0.350581
2    0.318847
4    0.176308
5    0.114438
1    0.039826
Name: income_cat, dtype: float64

PROPORTION OF TARGET IN THE TRAINING SET
3    0.350594
2    0.318859
4    0.176296
5    0.114462
1    0.039789
Name: income_cat, dtype: float64

PROPORTION OF TARGET IN THE TEST SET
3    0.350533
2    0.318798
4    0.176357
5    0.114341
1    0.039971
Name: income_cat, dtype: float64
```

So, remember:

Use 'stratified sampling' if you a have skew dataset

```
1 sns.histplot(data=housing, x="income_cat")
```



BUT... later in the slides we will see k-fold cross-validation to make even more 'accurate' splits

Further Prepare the Data for ML

Dealing with missing data

```
1 sample_incomplete_rows = df[df.isnull().any(axis=1)]
2 sample_incomplete_rows.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	NaN	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	NaN	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	NaN	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	NaN	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	NaN	375.0	183.0	9.8020	<1H OCEAN

Option 1: Drop all rows with NaN values for the column 'total bedrooms'

```
1 sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
```

Result:

longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity

Further Prepare the Data for ML

Dealing with missing data

Option 2: Drop the column 'total bedrooms'

```
1 sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
```

Result:

	longitude	latitude	housing_median_age	total_rooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	375.0	183.0	9.8020	<1H OCEAN
...	...	...	...	...	...	...	...	...
2348	-122.70	38.35	14.0	2313.0	954.0	397.0	3.7813	<1H OCEAN
366	-122.50	37.75	44.0	1819.0	1137.0	354.0	3.4919	NEAR OCEAN
18241	-121.44	38.54	39.0	2855.0	1217.0	562.0	3.2404	INLAND
18493	-116.21	33.75	22.0	894.0	830.0	202.0	3.0673	INLAND
16519	-117.86	34.01	16.0	4632.0	3038.0	727.0	5.1762	<1H OCEAN

168 rows × 8 columns

Further Prepare the Data for ML

Dealing with missing data

Option 3: **Impudence** (Fill in the empty values with something. f.e. mean/median/mode etc.)

```
1 median = housing["total_bedrooms"].median()
2 sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3
```

Result:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	434.0	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	434.0	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	434.0	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	434.0	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	434.0	375.0	183.0	9.8020	<1H OCEAN
...	...	...	...	...	...	...	...	...	...
2348	-122.70	38.35	14.0	2313.0	434.0	954.0	397.0	3.7813	<1H OCEAN
366	-122.50	37.75	44.0	1819.0	434.0	1137.0	354.0	3.4919	NEAR OCEAN
18241	-121.44	38.54	39.0	2855.0	434.0	1217.0	562.0	3.2404	INLAND
18493	-116.21	33.75	22.0	894.0	434.0	830.0	202.0	3.0673	INLAND
16519	-117.86	34.01	16.0	4632.0	434.0	3038.0	727.0	5.1762	<1H OCEAN

168 rows × 9 columns

Further Prepare the Data for ML

Feature Scaling

With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales.There are two common ways to get all attributes to have the same scale:

IMPORTANT: fit_transform scalers on the training data**,**

only use 'transform' on the test data

Min-max scaling(aka normalization)

- simplest scaler
- values are shifted and rescaled so that they end up ranging from 0 to 1.
- subtract the min value and divide by the max minus the min.

```
1 from sklearn.preprocessing import MinMaxScaler
2
3 min_max_scaler = MinMaxScaler()
4 housing_num_min_max_scaled = min_max_scaler.fit_t
```

Further Prepare the Data for ML

Feature Scaling

Standardization

- first it subtracts the mean value (so standardized values always have a zero mean), then it divides by the standard deviation so that the resulting distribution has unit variance.

- Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1).

- Standardization is **much less affected by outliers.**

- For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected.

```
1 from sklearn.preprocessing import StandardScaler
2
3 std_scaler = StandardScaler()
4 housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

Further Prepare the Data for ML

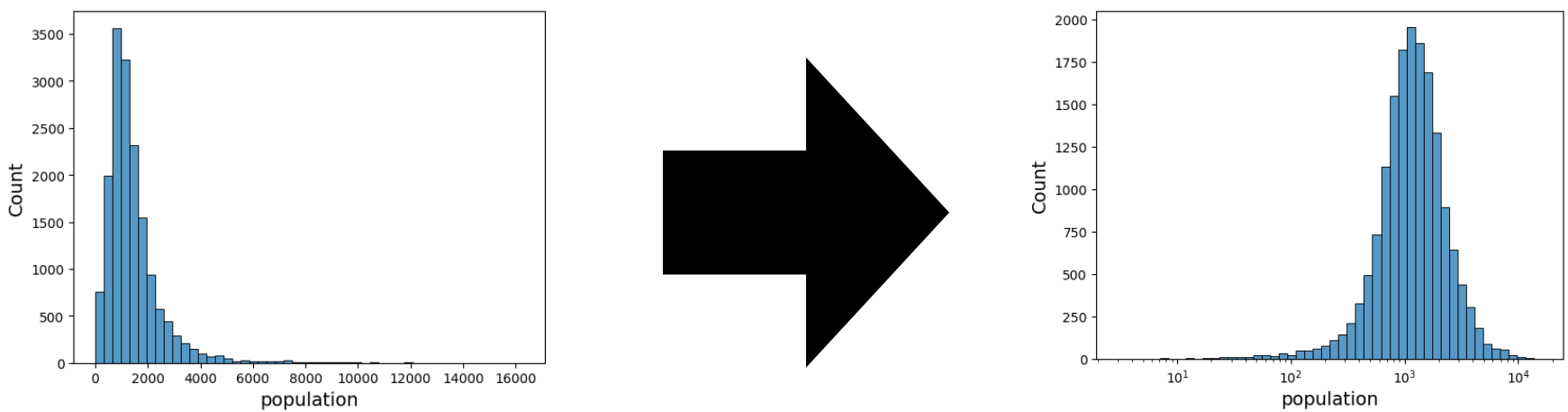
Dealing with tail-heavy (skewed) numerical data

When a feature's distribution has a heavy tail (i.e., when values far from the mean are not exponentially rare), both min-max scaling and standardization will squash most values into a small range. Machine learning models generally don't like this at all. So before you scale the feature, you should first transform it to shrink the heavy tail, and if possible to make the distribution roughly symmetrical. For example, a common way to do this for positive features with a heavy tail to the right is to replace the feature with its square root (or raise the feature to a power between 0 and 1).

Further Prepare the Data for ML

Dealing with tail-heavy (skewed) numerical data

```
# Or visulise using seaborn:
sns.histplot(data=housing, x="population", bins=50)
plt.show()
# apply log transformation
sns.histplot(data=housing, x="population", bins=50, log_scale=True)
plt.show()
```



The use of Pipelines

You can either do the data preparation steps manually for each column. **OR** you can make use of pipelines which you can apply to all (or a couple) at once.

```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),  # missing categorical values are fill
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```

The use of Pipelines

Since listing all the column names is not very convenient, Scikit-Learn provides a `make_column_selector()` function that returns a selector function you can use to automatically select all the features of a given type, such as numerical or categorical. For example, the following code creates the same `ColumnTransformer` as earlier, except the transformers are automatically named "pipeline-1" and "pipeline-2" instead of "num" and "cat":

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)
```

Now we're ready to apply this `ColumnTransformer` to the housing data:

```
housing_prepared = preprocessing.fit_transform(housing)
```

Select and Train a Model

```
from sklearn.linear_model import LinearRegression

lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)

housing_predictions = lin_reg.predict(housing)
housing_predictions[:5].round(-2)  # -2 = rounded to the nearest hundred
# RESULT:
# array([270900., 334800., 119900., 109200., 305600.])

# Compare against the actual results

housing_labels.iloc[:5].values
# array([458300., 483800., 101700.,  96100., 361800.])
```

Well, it works, but not always: the first prediction is way off, while the other predictions are better.

Remember that we chose to use the RMSE as our performance measure, so we want to measure this regression model's RMSE on the whole training set using Scikit-Learn's `mean_squared_error()` function, with the `squared` argument set to `False`:

Select and Train a Model

```
1 from sklearn.metrics import mean_squared_error
2
3 lin_rmse = mean_squared_error(housing_labels, housing_predictions,
4                               squared=False)
5 lin_rmse
6 # OUTPUT: 68232.83515124217
```

This is better than nothing, but clearly not a great score: the median_housing_values of most districts range between $120.000 and $265.000, so a typical prediction error of $68.232 is really not very satisfying. This is an example of a model underfitting the training data (The model has not learned the correct/necessary patterns).

When this happens it can mean 1/ that the features do not provide enough information to make good predictions, or 2/ that the model is not powerful enough.

Some ways to fix underfitting are:

- to select a more powerful model

- to feed the training algorithm with better features, or to reduce the constraints on the model.

Compare the results with other models

```
from sklearn.ensemble import RandomForestRegressor

tree_reg = make_pipeline(preprocessing, RandomForestRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)

housing_predictions = tree_reg.predict(housing)
tree_rmse = mean_squared_error(housing_labels, housing_predictions,
                               squared=False)
tree_rmse
# OUTPUT: 18057.976055305204
```

Is this the best we can do?

Is this way of evaluating models the best way to do so?

What would be the meaning of an RMSE = 0, is this good or bad?

Cross-validation

Cross-Validation

Cross validation is a more advanced set of methods for splitting data into training and testing sets.

We understand the intuition behind performing a train test split, we want to fairly evaluate our model's performance on unseen data.

Unfortunately this means we are not able to tune hyperparameters to the **entire** dataset.

Is there a way we can achieve the following:

- Train on all the data
- Evaluate on all the data

Cross-Validation

Imagine our dataset:

	X		y
Area m^2	Bedrooms	Bathrooms	Price
200	3	2	$500,000
190	2	1	$450,000
230	3	3	$650,000
180	1	1	$400,000
210	2	2	$550,000

Cross-Validation

Convert to generalized form

		X		y
x_1	x_2	x_3		y
x^1_1	x^1_1	x^1_1		y_1
x^2_1	x^2_1	x^2_1		y_2
x^3_1	x^3_1	x^3_1		y_3
x^4_1	x^4_1	x^4_1		y_4
x^5_1	x^5_1	x^5_1		y_5

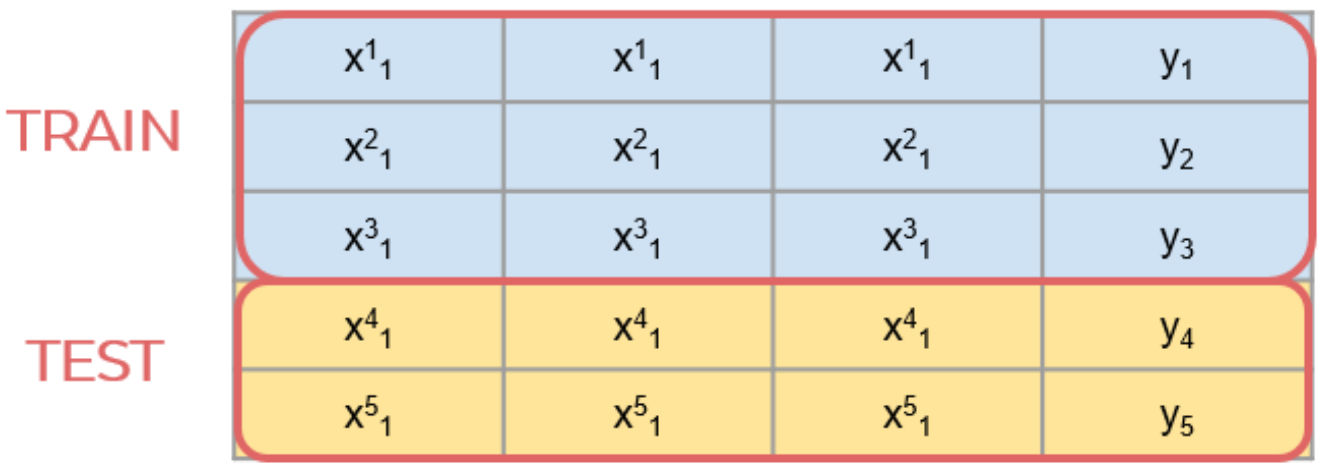
Cross-Validation

Color based off train vs. test set.

	X			y
	x_1	x_2	x_3	y
TRAIN	x^1_1	x^1_1	x^1_1	y_1
	x^2_1	x^2_1	x^2_1	y_2
	x^3_1	x^3_1	x^3_1	y_3
TEST	x^4_1	x^4_1	x^4_1	y_4
	x^5_1	x^5_1	x^5_1	y_5

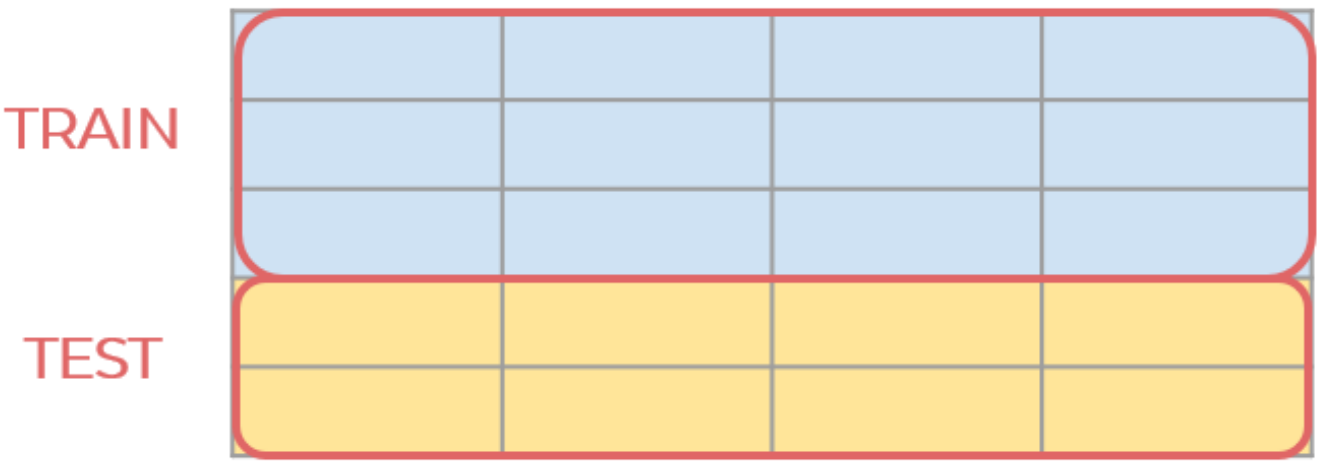
Cross-Validation

For now just consider training vs testing:



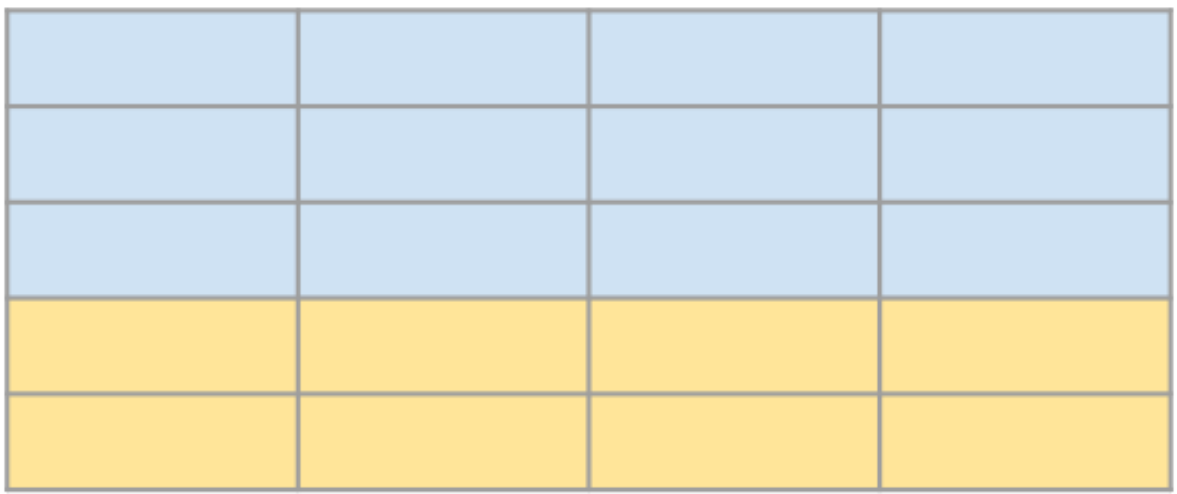
Cross-Validation

Rotate & Resize:



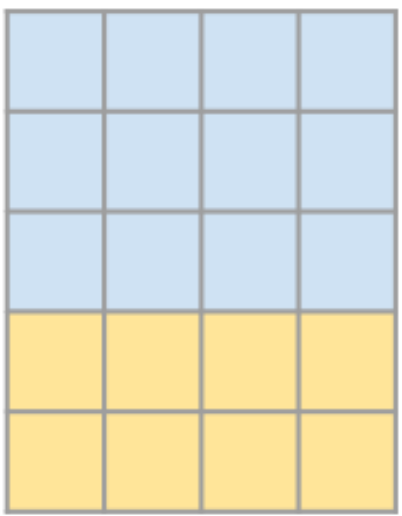
Cross-Validation

Rotate & Resize:



Cross-Validation

Rotate & Resize:

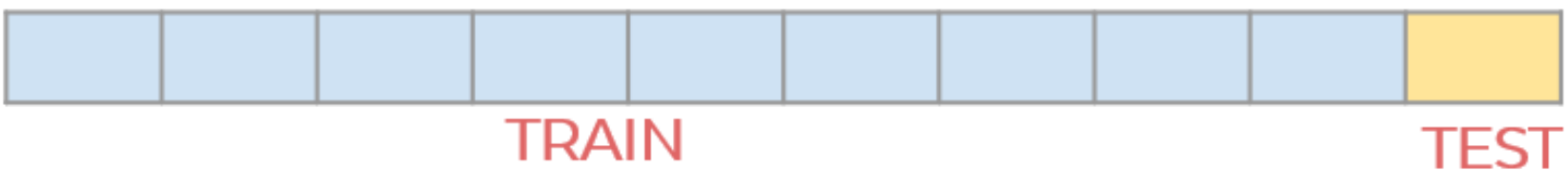


Cross-Validation

Rotate & Resize:

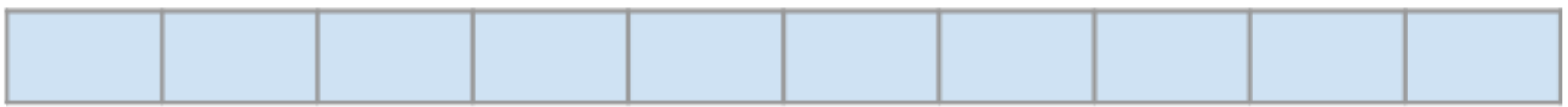
Cross-Validation

Now we can represent full data and splits:



Cross-Validation

Let's start with the entire original data:



Cross-Validation

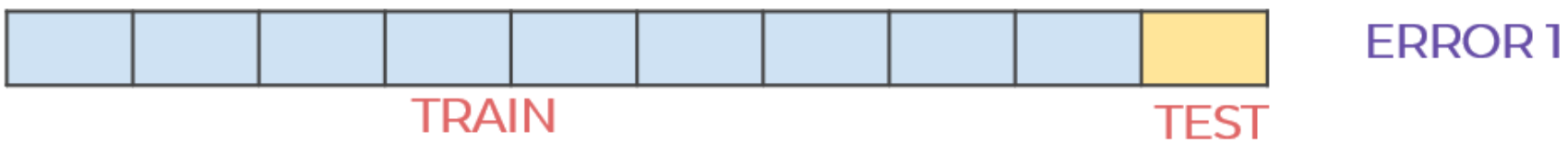
How does cross-validation work? Split data into K equal parts:

Cross-Validation

1/K left as test set

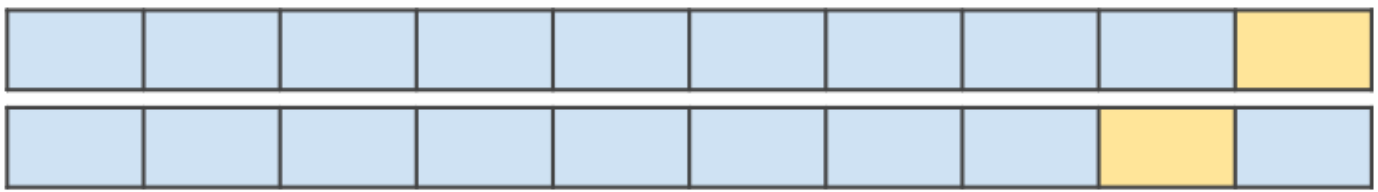
Cross-Validation

Train model and get error metric for split:



Cross-Validation

Repeat for another 1/K split

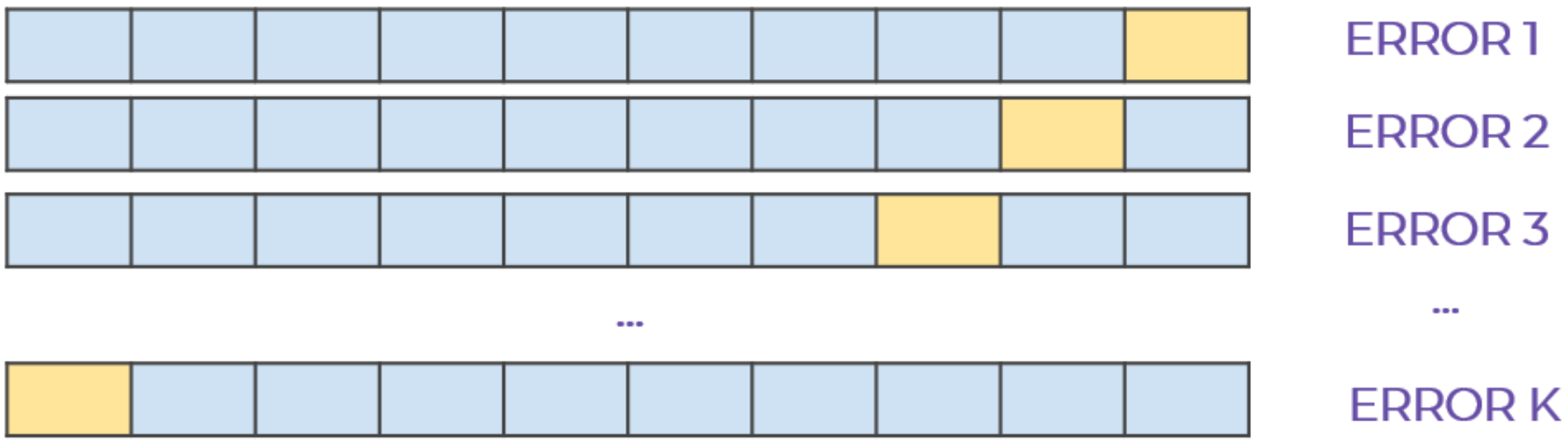


ERROR 1

ERROR 2

Cross-Validation

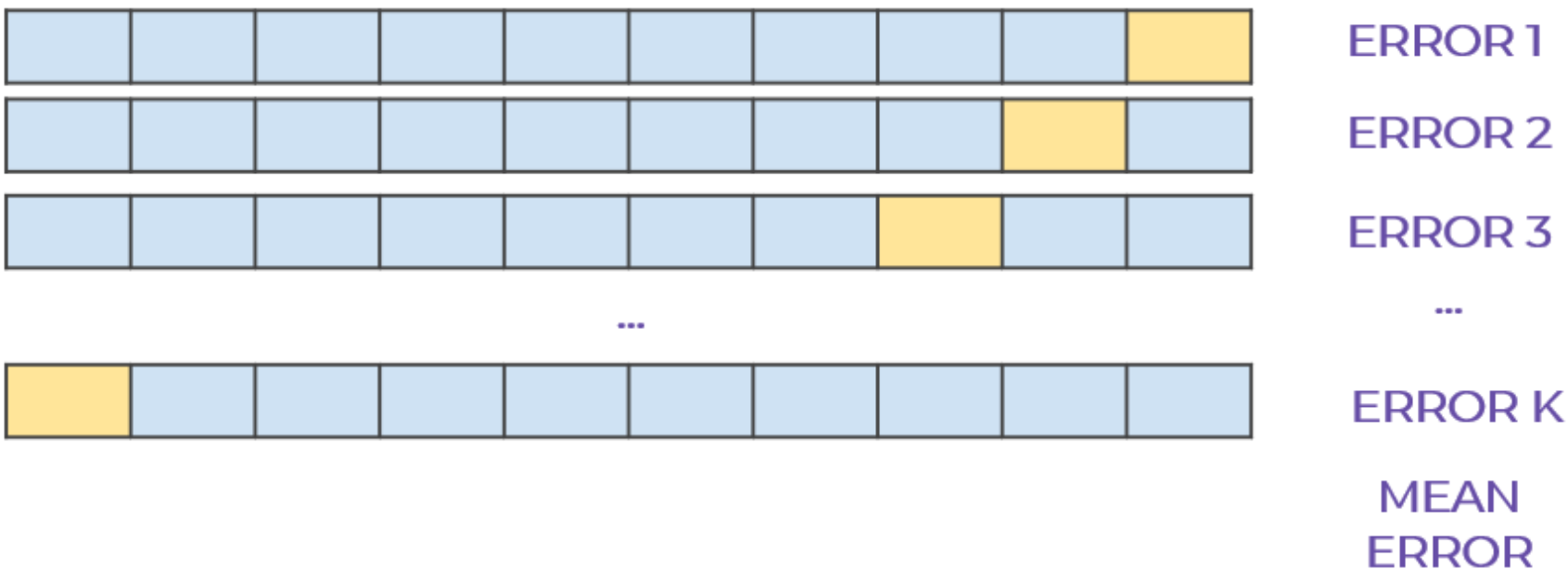
Keep repeating for all possible splits



Cross-Validation

Get average error

Average error is the expected performance



Cross-Validation

- We were able to train on all data **and** evaluate on all data!
- We get a better sense of true performance across multiple potential splits.
- What is the cost of this?
  - We have to repeat computations K number of times!

Cross-Validation

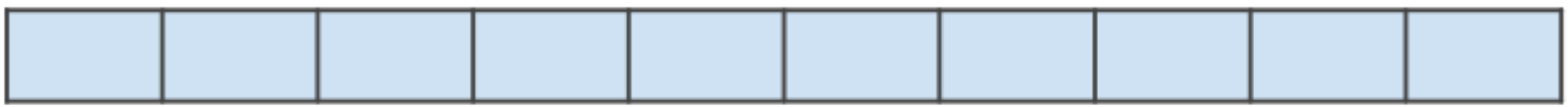
- This is known as K-fold cross-validation.
- Common choice for K is 10 so each test set is 10% of your total data.
- Largest K possible would be K equal to the number of number of rows.
  - This is known as **leave one out** cross validation.
  - Computationally expensive!

Cross-Validation

- One consideration to note with K-fold cross validation and a standard train test split is fairly tuning hyperparameters.
- If we tune hyperparameters to test data performance, are we ever fairly getting performance metrics?
- How can we understand how the model behaves for data that is has not seen **and** not been influenced by for hyperparameter tuning?
- For this we can use a **hold out** test set.

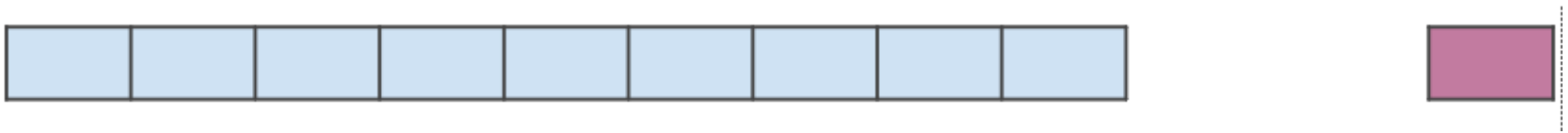
Cross-Validation

Start with entire data set:



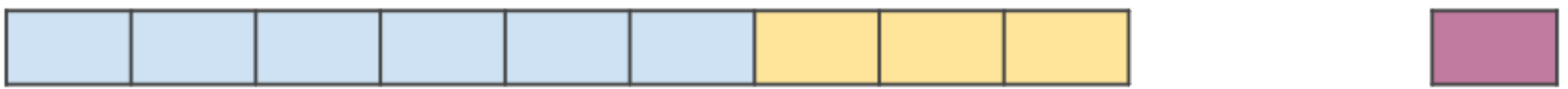
Cross-Validation

Remove a hold out test set



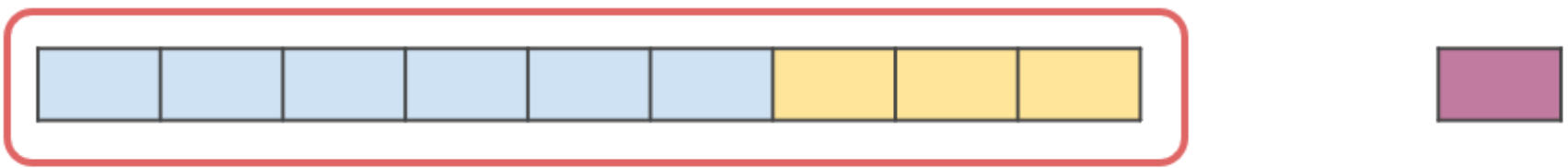
Cross-Validation

Perform “classic” train test split:



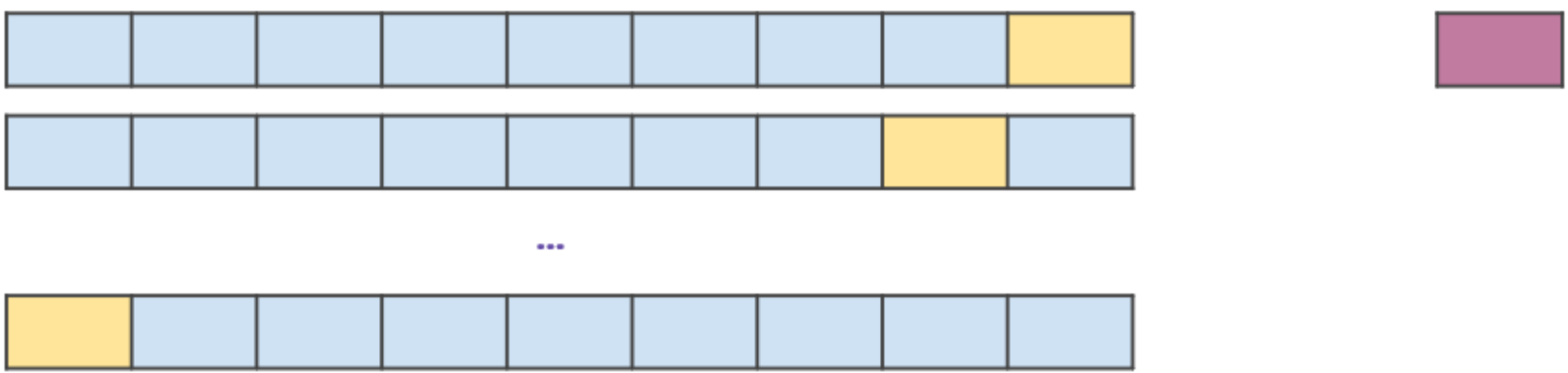
Cross-Validation

Train and tune on this data:



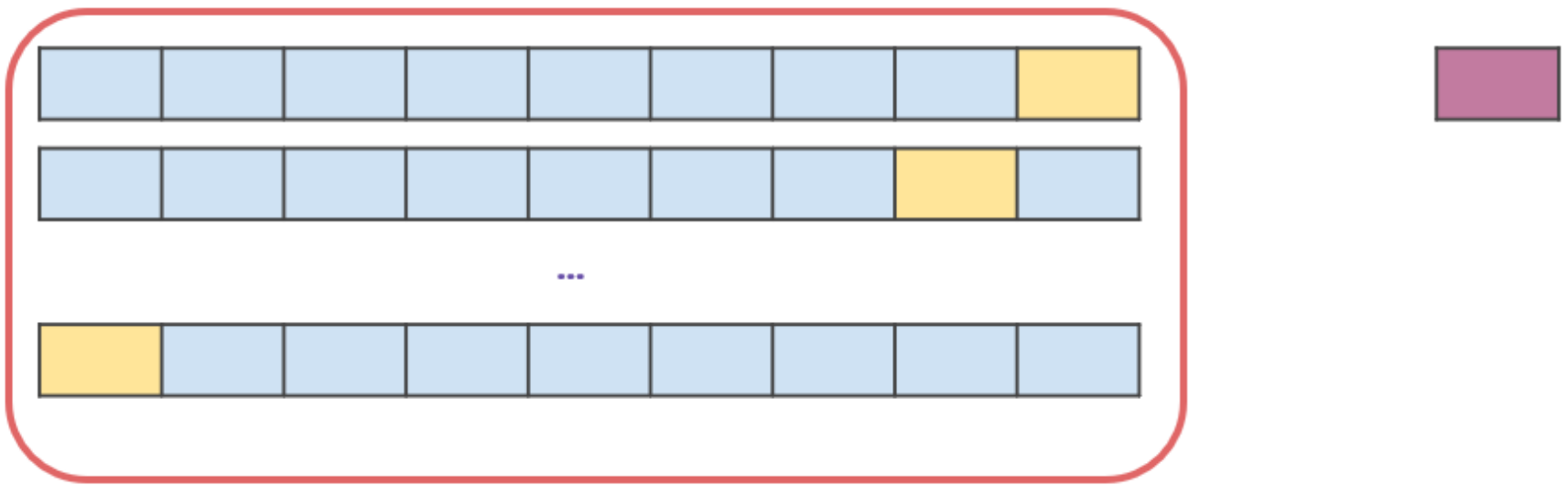
Cross-Validation

Or K-Fold cross validation



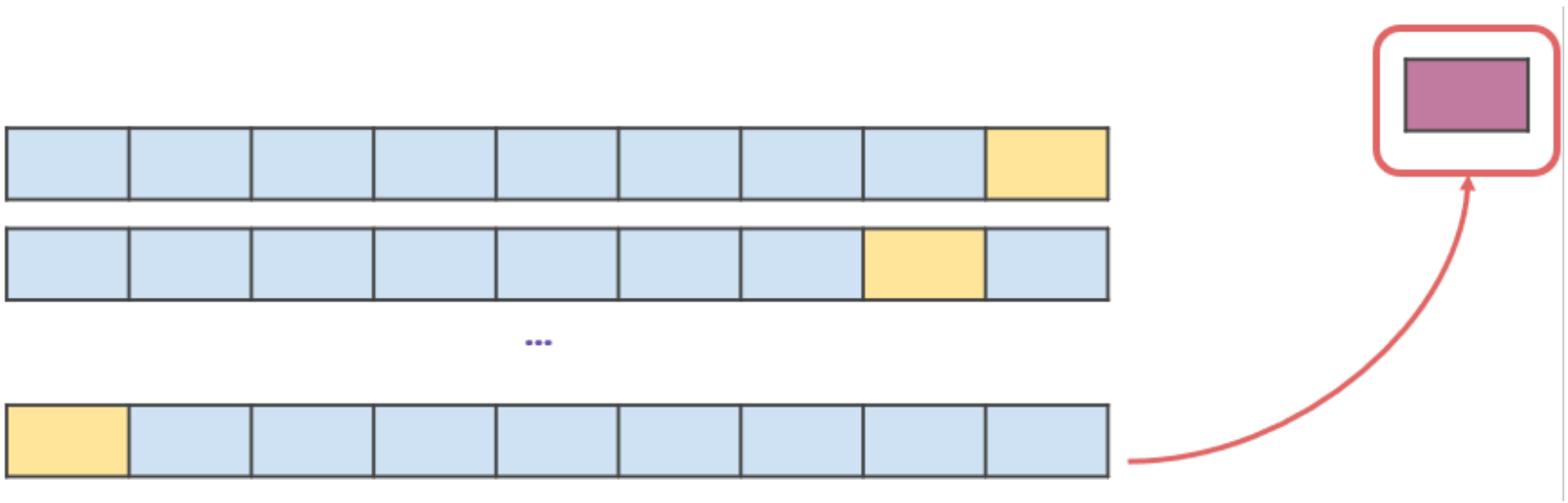
Cross-Validation

Train **and** tune on this data:



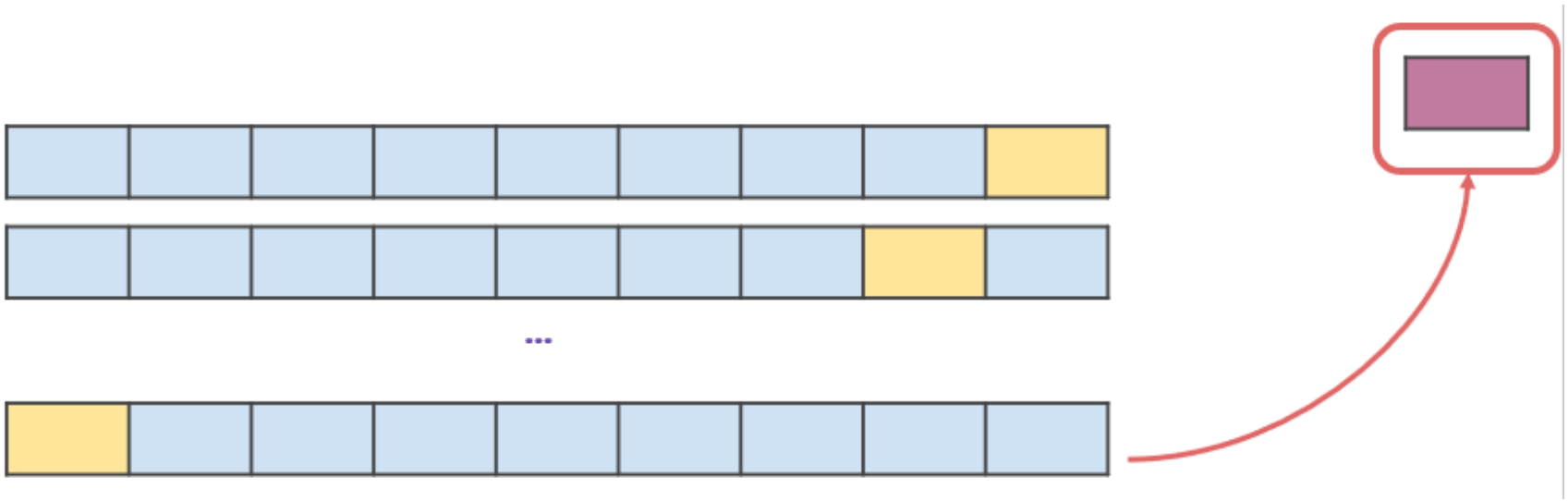
Cross-Validation

After training and tuning perform **final evaluation** hold out test set.



Cross-Validation

After training and tuning perform **final evaluation** hold out test set.



IMPORTANT: Can **not** tune after this **final** test evaluation!

Cross-Validation

- Train | Validation | Test Split

- Allows us to get a true final performance metric to report.
- No editing model after this!

Cross-Validation

Let's take a look at the notebook

Cross-Validation: Code

Code for Train | Validation | Test Split Procedure

```
##################################################################
#### SPLIT TWICE! Here we create TRAIN | VALIDATION | TEST  ########
##################################################################
from sklearn.model_selection import train_test_split

# 70% of data is training data, set aside other 30%
X_train, X_OTHER, y_train, y_OTHER = train_test_split(X, y, test_size=0.3, random_state=101)

# Remaining 30% is split into evaluation and test sets
# Each is 15% of the original data size
X_eval, X_test, y_eval, y_test = train_test_split(X_OTHER, y_OTHER, test_size=0.5, random_state=101)

# SCALE DATA
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_eval = scaler.transform(X_eval)
X_test = scaler.transform(X_test)
```

Cross-Validation: Code

Code for calculating mean error:

```
from sklearn.linear_model import Ridge
model = Ridge(alpha=100) #can be any model, Ridge Regression is just an example

model.fit(X_train,y_train)
y_eval_pred = model.predict(X_eval)
mean_squared_error(y_eval,y_eval_pred)

# Output: 7.320101458823871
```

Cross-Validation: Code

After doing this for one model, check for another setup of the same model
(Or an entirely different model)

```
1 from sklearn.linear_model import Ridge
2 model = Ridge(alpha=1) #can be any model, Ridge Regression is just an example
3
4 model.fit(X_train,y_train)
5 y_eval_pred = model.predict(X_eval)
6 mean_squared_error(y_eval,y_eval_pred)
7
8 # Output: 2.383783075056986
```

Choose the model you want to use,

```
1 y_final_test_pred = model.predict(X_test)
2 mean_squared_error(y_test,y_final_test_pred)
3
4 # Output: 2.254260083800517
```

Cross-Validation: Code

Using cross_val_score (or cross_validate -> See Notebook)

```
# TRAIN TEST SPLIT
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=101)

# SCALE DATA
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Chose Model
model = Ridge(alpha=100)

from sklearn.model_selection import cross_val_score
scores = cross_val_score(model,X_train,y_train,
                         scoring='neg_mean_squared_error',cv=5)
print(abs(scores.mean()))
```

Cross-Validation

- What about stratified sampling with K-fold Cross-validation?

```
kfold = KFold(n_splits=3,random_state=11,shuffle=True)
splits = kfold.split(data,data['target']) # each split has a train indexes and test indexes
print(f'PROPORTION OF TARGET IN THE ORIGINAL DATA\n{data["target"].value_counts() / len(data
for n,(train_index,test_index) in enumerate(splits):
    print(f'SPLIT NO {n+1}\nTRAINING SET SIZE: {np.round(len(train_index) / (len(train_index
          f'\tTEST SET SIZE: {np.round(len(test_index) / (len(train_index)+len(test_index)),
          f'{data.iloc[test_index,3].value_counts() / len(data.iloc[test_index,3])}\nPROPORT
          f'{data.iloc[train_index,3].value_counts() / len(data.iloc[train_index,3])}\n\n')
```

Cross-Validation

```
PROPORTION OF TARGET IN THE ORIGINAL DATA
1    0.342
2    0.330
0    0.328
Name: target, dtype: float64


SPLIT NO 1
TRAINING SET SIZE: 0.67     TEST SET SIZE: 0.33
PROPORTION OF TARGET IN THE TRAINING SET
0    0.395210
1    0.305389
2    0.299401
Name: target, dtype: float64
PROPORTION OF TARGET IN THE TEST SET
1    0.360360
2    0.345345
0    0.294294
Name: target, dtype: float64


SPLIT NO 2
TRAINING SET SIZE: 0.67     TEST SET SIZE: 0.33
PROPORTION OF TARGET IN THE TRAINING SET
1    0.353293
2    0.323353
0    0.323353
Name: target, dtype: float64
PROPORTION OF TARGET IN THE TEST SET
1    0.336336
2    0.333333
0    0.330330
Name: target, dtype: float64
```

The same distribution of out 'target' column in **NOT** present in all the folds

```
SPLIT NO 3
TRAINING SET SIZE: 0.67     TEST SET SIZE: 0.33
PROPORTION OF TARGET IN THE TRAINING SET
2    0.36747
1    0.36747
0    0.26506
Name: target, dtype: float64
PROPORTION OF TARGET IN THE TEST SET
0    0.359281
1    0.329341
2    0.311377
Name: target, dtype: float64
```

Cross-Validation

Better use: **StratifiedKFold** instead of **KFold**

```
kfold = StratifiedKFold(n_splits=3,shuffle=True,random_state=11)
#data['target'] IS THE VARIABLE USED FOR STRATIFIED SAMPLING.
splits = kfold.split(data,data['target'])
print(f'PROPORTION OF TARGET IN THE ORIGINAL DATA\n{data["target"].value_counts() / len(data)}
for n,(train_index,test_index) in enumerate(splits):
    print(f'SPLIT NO {n+1}\nTRAINING SET SIZE: {np.round(len(train_index) / (len(train_index)+
          f'\tTEST SET SIZE: {np.round(len(test_index) / (len(train_index)+len(test_index)),2)
          f'{data.iloc[test_index,3].value_counts() / len(data.iloc[test_index,3])}\nPROPORTIO
          f'{data.iloc[train_index,3].value_counts() / len(data.iloc[train_index,3])}\n\n')
```

Try it yourself to see that the distribution remains in all k folds

Cross-Validation

- All these approaches are valid, each situation is unique!
- Keep in mind:
  - Previous modeling work
  - Reporting requirements
  - Fairness of evaluation
  - Context of data and model
- Many regularization methods have tunable parameters we can adjust based on cross-validation techniques.
- For simplicity, there are times in the course we will opt for a simple two part train test split.

GridSearch

GridSearch

- Often more complex models have multiple adjustable hyperparameters.
- A grid search is a way of training and validating a model on every possible combination of multiple hyperparameter options.
- Scikit-Learn includes a **GridSearchCV** class capable of testing a dictionary of multiple hyperparameter options through cross-validation.
- This allows for both cross-validation and a grid search to be performed in a generalized way for any model.

GridSearch: Code

```
from sklearn.linear_model import ElasticNet
base_elastic_model = ElasticNet()

param_grid = {'alpha':[0.1,1,5,10,50,100],
              'l1_ratio':[.1, .5, .7, .9, .95, .99, 1]}

from sklearn.model_selection import GridSearchCV
grid_model = GridSearchCV(estimator=base_elastic_model,
                          param_grid=param_grid,
                          scoring='neg_mean_squared_error',
                          cv=5,
                          verbose=2)

grid_model.fit(X_train,y_train)

print(grid_model.best_estimator_)
# Output:ElasticNet(alpha=0.1, copy_X=True, fit_intercept=True, l1_ratio=1,
#            max_iter=1000, normalize=False, positive=False, precompute=False,
#            random_state=None, selection='cyclic', tol=0.0001, warm_start=False)

print(grid_model.best_params_)
# Output:
# {'alpha': 0.1, 'l1_ratio': 1}
```

GridSearch: Code

Don't forget to take a look at the final notebook around GridSearch