

CODING

Clarinda

November 13, 2024

Contents

1	Introduction	2
1.1	Machine Learning Model	4
2	Machine Learning Module 2	6
2.1	Missing values	6
2.2	Categorical variables	9
2.3	Pipelines	15
2.4	cross validation	17
2.5	XG Boost	18
2.6	Data leakage	19
3	PANDAS Module	20
3.1	Part 1	20
3.2	Part Indexing, Assigning	21
3.3	Part 3 Summary functions and maps	23
3.4	Group and Sort	26
4	Feature Engineering Module	28
4.1	Mutual information	28
4.2	Create Features	32
4.2.1	Group Transforms	34
4.3	Klustering	35
4.4	PCA	36
4.5	PCA part 2	37
4.6	Target Encoding	41
4.7	BOLUS	41
5	Time Series	42
5.0.1	Lag features	42
5.1	Trends	43
5.2	Seasonality	45
5.3	SEASONAL EXERCISE	49
5.4	Serial Dependence	52

5.5	LAG 2	54
5.6	Hybrid	57
5.6.1	Exercise Hybrid	64
5.7	Prediction	67
6	Optifer	74
6.1	Andere dingen	84
7	Stoch Vol estimation	85
8	Titanic	86
9	Out of Box	88
9.1	Part two	93
9.2	Part 3	95
9.3	P4	99

1 Introduction

```

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt

# save filepath to variable for easier access
melbourne_file_path = '../input/melbourne-housing-snapshot/melb_data.csv'
# read the data and store data in DataFrame titled melbourne_data
melbourne_data = pd.read_csv(melbourne_file_path)
# print a summary of the data in Melbourne data
melbourne_data.describe()

average_lot_size = df['lot_size'].mean()

print(df.columns.tolist())

df.shape gives row and columns

df.describe()
df.info()
df.isnull().sum()
missing_per_column = df.isna().sum()

so you dont have to

# Calculate the number of missing values per column

```

```

missing_per_column = df.isna().sum()

# Print the number of missing values for each column
print("Missing values per column:")
for column, missing_count in missing_per_column.items():
    print(f"{column}: {missing_count}")

NIET DOEN!!!
df_cleaned = df.dropna()

# Compute the median of the specific column (e.g., 'A')
median_A = df['A'].median()

# Impute missing values in column 'A' with its median
df['A'] = df['A'].fillna(median_A)

print(df['column_name'].unique())

Example DataFrame
df = pd.DataFrame({
    'feature': [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5]
})

# Plot histogram for the 'feature' column
df['feature'].hist(bins=10, edgecolor='black')

# Add title and labels
plt.title('Histogram of Feature')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Plot histogram for the 'feature' column
plt.hist(df['feature'], bins=10, edgecolor='black')

# Add title and labels
plt.title('Histogram of Feature')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Show the plot
plt.show()

df['feature'].describe()

```

```
# Verify the presence of NaN values
print("\nCount of NaN values in 'feature':")
print(df['feature'].isna().sum())

# Drop rows where 'feature' column has NaN values
df_cleaned = df.dropna(subset=['feature'])
```

Alternatively

```
df_c = df.dropna(axis=0)
```

axis=0: This specifies that you want to drop entire rows that contain any NaNs. (Alternatively, axis=1 would drop columns with any NaNs)

1.1 Machine Learning Model

```
# save filepath to variable for easier access
melbourne_file_path = '../input/melbourne-housing-snapshot/melb_data.csv'
# read the data and store data in DataFrame titled melbourne_data
melbourne_data = pd.read_csv(melbourne_file_path)
# print a summary of the data in Melbourne data
melbourne_data.describe()
home_data.info()

df_c = df.dropna(axis=0)
```

axis=0: This specifies that you want to drop entire rows that contain any NaNs. (Alternatively, axis=1 would drop columns with any NaNs)

Select target RV

```
y = melbourne_data.Price
```

choose feautures

```
features_names = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', 'Longtitude']
```

```
X = df_c_data[features_names]
```

```

Fit model

from sklearn.tree import DecisionTreeRegressor

# Define model. Specify a number for random_state to ensure same results
each run
melbourne_model = DecisionTreeRegressor(random_state=1)

# Fit model
melbourne_model.fit(X, y)

print(melbourne_model.predict(X.head()))

from sklearn.metrics import mean_absolute_error

predicted_home_prices = melbourne_model.predict(X)
MAE_training_error=mean_absolute_error(y, predicted_home_prices)

from sklearn.model_selection import train_test_split

# split data into training and validation data, for both features and
target
# The split is based on a random number generator. Supplying a numeric
value to
# the random_state argument guarantees we get the same split every
time we
# run this script.
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state
= 0)
# Define model
melbourne_model = DecisionTreeRegressor()
# Fit model
melbourne_model.fit(train_X, train_y)

# get predicted prices on validation data
val_predictions = melbourne_model.predict(val_X)
print(mean_absolute_error(val_y, val_predictions))

#Silly cross validaation

```

```

def get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=0)
    model.fit(train_X, train_y)
    preds_val = model.predict(val_X)
    mae = mean_absolute_error(val_y, preds_val)
    return(mae)

def get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=0)
    model.fit(train_X, train_y)
    preds_val = model.predict(val_X)
    mae = mean_absolute_error(val_y, preds_val)
    return(mae)

#choose model with lowers MAE on validation set

from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error

forest_model = RandomForestRegressor(random_state=1)
forest_model.fit(train_X, train_y)
melb_preds = forest_model.predict(val_X)
print(mean_absolute_error(val_y, melb_preds))

```

2 Machine Learning Module 2

2.1 Missing values

```
df_c = df.dropna(axis=0)
```

axis=0: This specifies that you want to drop entire rows that contain any NaNs. (Alternatively, axis=1 would drop columns with any NaNs)

```

# Get names of columns with missing values
cols_with_missing = [col for col in X_train.columns
                      if X_train[col].isnull().any()]

# Drop columns in training and validation data

```

```

reduced_X_train = X_train.drop(cols_with_missing, axis=1)
reduced_X_valid = X_valid.drop(cols_with_missing, axis=1)

from sklearn.impute import SimpleImputer

# Make copy to avoid changing original data (when imputing)
X_train_plus = X_train.copy()
X_valid_plus = X_valid.copy()

# Make new columns indicating what will be imputed
for col in cols_with_missing:
    X_train_plus[col + '_was_missing'] = X_train_plus[col].isnull()
    X_valid_plus[col + '_was_missing'] = X_valid_plus[col].isnull()

# Imputation
my_imputer = SimpleImputer()
imputed_X_train_plus = pd.DataFrame(my_imputer.fit_transform(X_train_plus))
imputed_X_valid_plus = pd.DataFrame(my_imputer.transform(X_valid_plus))

# Imputation removed column names; put them back
imputed_X_train_plus.columns = X_train_plus.columns
imputed_X_valid_plus.columns = X_valid_plus.columns

print("MAE from Approach 3 (An Extension to Imputation):")
print(score_dataset(imputed_X_train_plus, imputed_X_valid_plus, y_train,
y_valid))

```

`X_train_plus[col + '_was_missing']:`

Creates a new column in the DataFrame `X_train_plus`. The name of this new column is derived from the original column name (`col`) with `'_was_missing'` appended to it.

`X_train_plus[col].isnull():`

`X_train_plus[col]`: Selects the original column from the DataFrame.
`.isnull()`: Returns a boolean Series where each entry is True if the corresponding entry in the original column is NaN (missing), and False otherwise.

Summary:

For each column with missing values in `cols_with_missing`, this code creates a new boolean column indicating whether each entry in the original column was missing (True) or not (False).

Here's a breakdown of what each line does:

```
my_imputer = SimpleImputer():
```

This line creates an instance of the SimpleImputer class from the sklearn.impute module.

By default, this imputer fills missing values with the mean of each column, though this behavior can be changed by passing arguments like strategy='median', strategy='most_frequent', etc.

```
imputed_X_train_plus = pd.DataFrame(my_imputer.fit_transform(X_train_plus)):
```

The fit_transform() method does two things:

Fit: It computes the imputation values (e.g., mean, median) from the X_train_plus dataset.

Transform: It fills in the missing values in X_train_plus using the computed imputation values.

The result is a NumPy array, which is then converted into a pandas DataFrame (as imputation removes the original column labels).

imputed_X_train_plus is the imputed version of X_train_plus, with missing values replaced.

```
my_imputer = SimpleImputer(strategy='median')
```

```
# Columns to impute  
columns_to_impute = ['A', 'B']
```

```
# Create the imputer  
my_imputer = SimpleImputer(strategy='mean')
```

```
# Impute the selected columns and assign them back directly  
X_train_plus[columns_to_impute] = my_imputer.fit_transform(X_train_plus[columns_to_impute])
```


The double brackets (`[['Length']]`) are used to select a DataFrame with a single column rather than a Series. In pandas, there is a difference between selecting a column as a Series and selecting it as a DataFrame:

Single brackets (`['Length']`) select a column as a Series.

Double brackets (`[['Length']]`) select a column as a DataFrame.

Why the double brackets are required:

The `SimpleImputer.fit_transform()` method expects a 2D array-like structure (such as a DataFrame) as input, even if it only has one column.

`male_data['Length']` would give a 1D Series, which is not valid input for `fit_transform()`.

`male_data[['Length']]` gives a 2D DataFrame with one column, which matches the expected input format.

```
# Impute missing 'Length' based on 'Gender'
data['Length'] = data.groupby('Gender')['Length'].transform(lambda
x: x.fillna(x.median() if x.name == 'Male' else x.mean()))
```

2.2 Categorical variables

`pd.get_dummies(drop_first=True)`: Drops the first level to avoid multicollinearity (dummy variable trap).

`OneHotEncoder(drop='first')`: Similar functionality as `drop_first=True` in `pd.get_dummies()`, removing the first category in each feature to avoid multicollinearity.

```
# Print the data type of a specific column
print(X_train['column_name'].dtype)
```

Ordinal encoding assigns each unique value to a different integer.

Ordinal encoding assigns each unique value to a different integer. categorical variables that have a clear ordering in the values, but we refer to those that do as ordinal variables

One-hot encoding creates new columns indicating the presence (or absence) of each possible value in the original data. We refer to categorical variables without an intrinsic ranking as nominal variables.

This approach assumes an ordering of the categories: "Never" (0) < "Rarely" (1) < "Most days" (2) < "Every day" (3).

One-hot encoding generally does not perform well if the categorical variable takes on a large number of values (i.e., you generally won't use it for variables taking more than 15 different values).

```
# Get list of categorical variables
s = (X_train.dtypes == 'object')
object_cols = list(s[s].index)

print("Categorical variables:")
print(object_cols)

from sklearn.preprocessing import OrdinalEncoder

# Make copy to avoid changing original data
label_X_train = X_train.copy()
label_X_valid = X_valid.copy()

# Apply ordinal encoder to each column with categorical data
ordinal_encoder = OrdinalEncoder()
label_X_train[object_cols] = ordinal_encoder.fit_transform(X_train[object_cols])
label_X_valid[object_cols] = ordinal_encoder.transform(X_valid[object_cols])
```

We use the OneHotEncoder class from scikit-learn to get one-hot encodings.

There are a number of parameters that can be used to customize its behavior.

We set `handle_unknown='ignore'` to avoid errors when the validation data contains classes that aren't represented in the training data, and setting `sparse=False` ensures that the encoded columns are returned as a numpy array (instead of a sparse matrix). To use the encoder, we supply only the categorical columns that we want to be one-hot encoded. For instance, to encode the training data, we supply `X_train[object_cols]`. (`object_cols` in the code cell below is a list of the column names with categorical data, and so `X_train[object_cols]` contains all of the categorical data in the training set.)

```
from sklearn.preprocessing import OneHotEncoder

# Apply one-hot encoder to each column with categorical data
OH_encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
OH_cols_train = pd.DataFrame(OH_encoder.fit_transform(X_train[object_cols]))
OH_cols_valid = pd.DataFrame(OH_encoder.transform(X_valid[object_cols]))

# One-hot encoding removed index; put it back
OH_cols_train.index = X_train.index
OH_cols_valid.index = X_valid.index

# Remove categorical columns (will replace with one-hot encoding)
num_X_train = X_train.drop(object_cols, axis=1)
num_X_valid = X_valid.drop(object_cols, axis=1)

# Add one-hot encoded columns to numerical features
OH_X_train = pd.concat([num_X_train, OH_cols_train], axis=1)
OH_X_valid = pd.concat([num_X_valid, OH_cols_valid], axis=1)

# Ensure all columns have string type
OH_X_train.columns = OH_X_train.columns.astype(str)
OH_X_valid.columns = OH_X_valid.columns.astype(str)

label_X_train[object_cols] = ordinal_encoder.fit_transform(X_train[object_cols])
label_X_valid[object_cols] = ordinal_encoder.transform(X_valid[object_cols])
```

For instance, you can write a custom ordinal encoder to deal with new categories. The simplest approach, however, is to drop the problematic categorical columns.

Run the code cell below to save the problematic columns to a Python list `bad_label_cols`. Likewise, columns that can be safely ordinal encoded

are stored in good_label_cols

```
# Categorical columns in the training data
object_cols = [col for col in X_train.columns if X_train[col].dtype
== "object"]

# Columns that can be safely ordinal encoded
good_label_cols = [col for col in object_cols if
                    set(X_valid[col]).issubset(set(X_train[col]))]

# Problematic columns that will be dropped from the dataset
bad_label_cols = list(set(object_cols)-set(good_label_cols))

print('Categorical columns that will be ordinal encoded:', good_label_cols)
print('\nCategorical columns that will be dropped from the dataset:',
bad_label_cols)


# Get number of unique entries in each column with categorical data
object_nunique = list(map(lambda col: X_train[col].nunique(), object_cols))
d = dict(zip(object_cols, object_nunique))

# Print number of unique entries by column, in ascending order
sorted(d.items(), key=lambda x: x[1])


# Columns that will be one-hot encoded
low_cardinality_cols = [col for col in object_cols if X_train[col].nunique()
< 10]

# Columns that will be dropped from the dataset
high_cardinality_cols = list(set(object_cols)-set(low_cardinality_cols))

print('Categorical columns that will be one-hot encoded:', low_cardinality_cols)
print('\nCategorical columns that will be dropped from the dataset:',
high_cardinality_cols)

from sklearn.preprocessing import OneHotEncoder

# Use as many lines of code as you need!

OH_encoder = OneHotEncoder(handle_unknown='ignore', sparse=False)
OH_cols_train = pd.DataFrame(OH_encoder.fit_transform(X_train[low_cardinality_cols]))
```

```

OH_cols_valid = pd.DataFrame(OH_encoder.transform(X_valid[low_cardinality_cols]))

# One-hot encoding removed index; put it back
OH_cols_train.index = X_train.index
OH_cols_valid.index = X_valid.index

# Remove categorical columns (will replace with one-hot encoding)
num_X_train = X_train.drop(object_cols, axis=1)
num_X_valid = X_valid.drop(object_cols, axis=1)

# Add one-hot encoded columns to numerical features
OH_X_train = pd.concat([num_X_train, OH_cols_train], axis=1)
OH_X_valid = pd.concat([num_X_valid, OH_cols_valid], axis=1)

# Ensure all columns have string type
OH_X_train.columns = OH_X_train.columns.astype(str)
OH_X_valid.columns = OH_X_valid.columns.astype(str)

```

One-Hot Encoding with `pd.get_dummies`:

```
python
Copy code
X_2 = pd.get_dummies(df.BldgType, prefix="Bldg")
pd.get_dummies(df.BldgType, prefix="Bldg") creates one-hot encoded
columns for each unique category in the BldgType column of df.
Each category in BldgType (e.g., "1Fam", "2fmCon", "TwnhsE", etc.)
is turned into a new binary (0 or 1) column in X_2, with the prefix
"Bldg" added to the category name.
If BldgType has categories "1Fam", "2fmCon", and "TwnhsE", X_2 will
have columns: Bldg_1Fam, Bldg_2fmCon, and Bldg_TwnhsE.
For a given row, only the column representing the actual building type
will contain a 1; all other columns will contain 0.
Example of X_2 after this step:
```

```

Bldg_1Fam Bldg_2fmCon Bldg_TwnhsE
1 0 0
0 1 0
0 0 1

```

This process represents each building type as a distinct column, allowing for binary differentiation among building types.

Element-wise Multiplication with `mul`:

```
python
Copy code
X_2 = X_2.mul(df.GrLivArea, axis=0)

```

`X_2.mul(df.GrLivArea, axis=0)` performs element-wise multiplication between each column in `X_2` and the `GrLivArea` column from `df`. The `axis=0` argument specifies that the multiplication should occur row by row. Each one-hot encoded value (0 or 1) in `X_2` is multiplied by the corresponding `GrLivArea` value in that row. As a result, each column in `X_2` now represents the `GrLivArea` (ground living area) for a specific building type: only the column corresponding to the actual `BldgType` will contain the `GrLivArea` value, while the others will be 0. Example of `X_2` after this step: Suppose the `GrLivArea` column has values [1500, 800, 1200]. After the multiplication, `X_2` will look like this:

```
Bldg_1Fam Bldg_2fmCon Bldg_TwnhsE
1500 0 0
0 800 0
0 0 1200
```

Each row now shows the `GrLivArea` value under the specific building type that corresponds to that row, and all other columns have 0.

Interaction Effects

The resulting columns represent interaction effects between `BldgType` and `GrLivArea` because:

Each new column (e.g., `Bldg_1Fam`, `Bldg_2fmCon`, `Bldg_TwnhsE`) represents the combined effect of both `BldgType` and `GrLivArea` rather than either feature independently.

This approach essentially "splits" the `GrLivArea` value across building types, showing how the living area impacts `SalePrice` (or any other target variable) differently based on the type of building.

You can see that there is a more general categorization described (roughly) by the first word of each category. Create a feature containing only these first words by splitting `MSSubClass` at the first underscore `_`. (Hint: In the `split` method use an argument `n=1`.)

```
X_4["MSClass"] = df.MSSubClass.str.split("_", n=1, expand=True)[0]
```

2.3 Pipelines

First we do standard reading data, test split etc

```
# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality (convenient
# but arbitrary)
categorical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].nunique()
< 10 and
                    X_train_full[cname].dtype == "object"]

# Select numerical columns
numerical_cols = [cname for cname in X_train_full.columns if X_train_full[cname].dtype
in ['int64', 'float64']]

# Keep selected columns only
my_cols = categorical_cols + numerical_cols
X_train = X_train_full[my_cols].copy()
X_valid = X_valid_full[my_cols].copy()
```

Step 2: Define the Model

Next, we define a random forest model with the familiar RandomForestRegressor class.

```
from sklearn.ensemble import RandomForestRegressor
```

```
model = RandomForestRegressor(n_estimators=100, random_state=0)
```

Step 3: Create and Evaluate the Pipeline

Finally, we use the Pipeline class to define a pipeline that bundles the preprocessing and modeling steps. There are a few important things to notice:

With the pipeline, we preprocess the training data and fit the model in a single line of code. (In contrast, without a pipeline, we have to do imputation, one-hot encoding, and model training in separate steps. This becomes especially messy if we have to deal with both numerical and categorical variables!)

With the pipeline, we supply the unprocessed features in `X_valid` to the `predict()` command, and the pipeline automatically preprocesses the features before generating predictions. (However, without a pipeline, we have to remember to preprocess the validation data before making predictions.)

```
from sklearn.metrics import mean_absolute_error

# Bundle preprocessing and modeling code in a pipeline
my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('model', model)
                              ])

# Preprocessing of training data, fit model
my_pipeline.fit(X_train, y_train)

# Preprocessing of validation data, get predictions
preds = my_pipeline.predict(X_valid)

# Evaluate the model
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)
```

We construct the full pipeline in three steps.

Step 1: Define Preprocessing Steps

Similar to how a pipeline bundles together preprocessing and modeling steps, we use the `ColumnTransformer` class to bundle together different preprocessing steps. The code below:

imputes missing values in numerical data, and
imputes missing values and applies a one-hot encoding to categorical
data.

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Preprocessing for numerical data
numerical_transformer = SimpleImputer(strategy='constant')

# Preprocessing for categorical data
categorical_transformer = Pipeline(steps=[
```



```

        ('imputer', SimpleImputer(strategy='most_frequent')),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])

# Bundle preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

def missing_indicator(X):
    return pd.DataFrame(X.isna(), columns=[f"{col}_missing" for col
in X.columns])

# Preprocessing for numerical data with missing indicator
numerical_transformer = Pipeline(steps=[
    ('missing_indicator', FunctionTransformer(func=missing_indicator,
validate=False)),
    ('imputer', SimpleImputer(strategy='constant'))
])

# Preprocessing for categorical data with missing indicator
categorical_transformer = Pipeline(steps=[
    ('missing_indicator', FunctionTransformer(func=missing_indicator,
validate=False)),
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Combine preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

```

2.4 cross validation

```

from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

my_pipeline = Pipeline(steps=[('preprocessor', SimpleImputer()),
                              ('model', RandomForestRegressor(n_estimators=50,

```

```

random_state=0))

])

from sklearn.model_selection import cross_val_score

# Multiply by -1 since sklearn calculates *negative* MAE
scores = -1 * cross_val_score(my_pipeline, X, y,
                              cv=5,
                              scoring='neg_mean_absolute_error')

print("MAE scores:\n", scores)

```

2.5 XG Boost

```

from xgboost import XGBRegressor

my_model = XGBRegressor()
my_model.fit(X_train, y_train)

my_model = XGBRegressor(n_estimators=1000, learning_rate=0.05, n_jobs=4)
my_model.fit(X_train, y_train,
            early_stopping_rounds=5,
            eval_set=[(X_valid, y_valid)],
            verbose=False)

```

early_stopping_rounds offers a way to automatically find the ideal value for n_estimators. Early stopping causes the model to stop iterating when the validation score stops improving, even if we aren't at the hard stop for n_estimators. It's smart to set a high value for n_estimators and then use early_stopping_rounds to find the optimal time to stop iterating.

Since random chance sometimes causes a single round where validation scores don't improve, you need to specify a number for how many rounds of straight deterioration to allow before stopping. Setting early_stopping_rounds=5

is a reasonable choice. In this case, we stop after 5 straight rounds of deteriorating validation scores.

When using `early_stopping_rounds`, you also need to set aside some data for calculating the validation scores - this is done by setting the `eval_set` parameter.

`learning_rate`

Instead of getting predictions by simply adding up the predictions from each component model, we can multiply the predictions from each model by a small number (known as the learning rate) before adding them in.

This means each tree we add to the ensemble helps us less. So, we can set a higher value for `n_estimators` without overfitting. If we use early stopping, the appropriate number of trees will be determined automatically.

In general, a small learning rate and large number of estimators will yield more accurate XGBoost models, though it will also take the model longer to train since it does more iterations through the cycle. As default, XGBoost sets `learning_rate=0.1`.

`n_jobs`

On larger datasets where runtime is a consideration, you can use parallelism to build your models faster. It's common to set the parameter `n_jobs` equal to the number of cores on your machine. On smaller datasets, this won't help.

The resulting model won't be any better, so micro-optimizing for fitting time is typically nothing but a distraction. But, it's useful in large datasets where you would otherwise spend a long time waiting during the fit command.

2.6 Data leakage

3 PANDAS Module

3.1 Part 1

Creating data

There are two core objects in pandas: the DataFrame and the Series.

DataFrame

A DataFrame is a table. It contains an array of individual entries, each of which has a certain value. Each entry corresponds to a row (or record) and a column.

For example, consider the following simple DataFrame:

```
pd.DataFrame({'Yes': [50, 21], 'No': [131, 2]})  
Yes No
```

We are using the `pd.DataFrame()` constructor to generate these DataFrame objects. The syntax for declaring a new one is a dictionary whose keys are the column names (Bob and Sue in this example), and whose values are a list of entries. This is the standard way of constructing a new DataFrame, and the one you are most likely to encounter.

A Series is, in essence, a single column of a DataFrame. So you can assign row labels to the Series the same way as before, using an index parameter. However, a Series does not have a column name, it only has one overall name:

```
pd.Series([30, 35, 40], index=['2015 Sales', '2016 Sales', '2017 Sales'],  
name='Product A')
```

We'll use the `pd.read_csv()` function to read the data into a DataFrame. This goes thusly:

```
wine_reviews = pd.read_csv("../input/wine-reviews/winemag-data-130k-v2.csv")  
We can use the shape attribute to check how large the resulting DataFrame  
is:
```

```
wine_reviews.shape  
(129971, 14)
```

So our new DataFrame has 130,000 records split across 14 different columns. That's almost 2 million entries!

We can examine the contents of the resultant DataFrame using the `head()` command, which grabs the first five rows:

```
wine_reviews.head()

quantities = ['4 cups', '1 cup', '2 large', '1 can']
items = ['Flour', 'Milk', 'Eggs', 'Spam']
recipe = pd.Series(quantities, index=items, name='Dinner')

reviews = pd.read_csv("../input/wine-reviews/winemag-data_first150k.csv", index_col=0)

Dont do the index double!

Save a DataFrame as a csv file!
animals_df.to_csv("cows_and_goats.csv")
```

3.2 Part Indexing, Assigning

```
reviews['country']
reviews['country'][0]
```

Pandas indexing works in one of two paradigms. The first is index-based selection: selecting data based on its numerical position in the data. `iloc` follows this paradigm.

To select the first row of data in a DataFrame, we may use the following:

```
reviews.iloc[0]
```

This means that it's marginally easier to retrieve rows, and marginally harder to get retrieve columns. To get a column with `iloc`, we can do the following:

```
reviews.iloc[:, 0]
```

To select just the second and third entries, we would do:

```
reviews.iloc[1:3, 0]
```

It's also possible to pass a list:

```
reviews.iloc[[0, 1, 2], 0]
```

The second paradigm for attribute selection is the one followed by the `loc` operator: label-based selection. In this paradigm, it's the data index value, not its position, which matters.

For example, to get the first entry in `reviews`, we would now do the following:

```
reviews.loc[0, 'country']  
'Italy'
```

`iloc` is conceptually simpler than `loc` because it ignores the dataset's indices. When we use `iloc` we treat the dataset like a big matrix (a list of lists), one that we have to index into by position. `loc`, by contrast, uses the information in the indices to do its work. Since your dataset usually has meaningful indices, it's usually easier to do things using `loc` instead. For example, here's one operation that's much easier using `loc`:

```
reviews.loc[:, ['taster_name', 'taster_twitter_handle', 'points']]
```

`iloc` uses the Python `stdlib` indexing scheme, where the first element of the range is included and the last one excluded. So `0:10` will select entries `0,...,9`. `loc`, meanwhile, indexes inclusively. So `0:10` will select entries `0,...,10`.

This result can then be used inside of `loc` to select the relevant data:

```
reviews.loc[reviews.country == 'Italy']  
reviews.loc[(reviews.country == 'Italy') & (reviews.points >= 90)]
```

Suppose we'll buy any wine that's made in Italy or which is rated above average. For this we use a pipe (`|`):

```
reviews.loc[(reviews.country == 'Italy') | (reviews.points >= 90)]  
  
reviews.loc[reviews.country.isin(['Italy', 'France'])]
```

The second is `isnull` (and its companion `notnull`). These methods let you highlight values which are (or are not) empty (`NaN`). For example, to filter out wines lacking a price tag in the dataset, here's what we would do:

```
reviews.loc[reviews.price.notnull()]
```

```
These things select column description
reviews.loc[:, 'description']
reviews['description']
desc = reviews.description
```

: means everything!!!

Select the first value from the description column of reviews, assigning it to variable first_description

```
first_description = reviews.description[0]
first_description = reviews.description.iloc[0]
```

Select the first row using: reviews.iloc[0]

Select records with index labels using sample_reviews = reviews.iloc[[1,2,3,5,8]]

Select sub dataframe df = reviews.loc[[0,1,10,100], ['country', 'province', 'region_1', 'region_2']]

The first 100 entries, since loc uses INCLUSIVE indexing.

```
reviews.loc[0:99, ['country', 'variety']]
```

The full columns

```
reviews.loc[0:100, ['country', 'variety']]
```

3.3 Part 3 Summary functions and maps

```
median_points = reviews.points.median()
```

```
reviews['ratio'] = reviews.points / reviews.price
```

```
reviews['ratio'] = reviews.points / reviews.price
```

```
bargain_wine = reviews.iloc[reviews.ratio.argmax()].title
```

```
bargain_idx = (reviews.points / reviews.price).idxmax()
bargain_wine = reviews.loc[bargain_idx, 'title']
```

```
list of unique countries
reviews.country.unique()
```

```
reviews.taster_name.unique()
```

```
reviews.taster_name.value_counts()
```

map() is the first, and slightly simpler one. For example, suppose that we wanted to remean the scores the wines received to 0. We can do this as follows:

```
review_points_mean = reviews.points.mean()
reviews.points.map(lambda p: p - review_points_mean)
```

The function you pass to map() should expect a single value from the Series (a point value, in the above example), and return a transformed version of that value. map() returns a new Series where all the values have been transformed by your function.

apply() is the equivalent method if we want to transform a whole DataFrame by calling a custom method on each row.

```
def remean_points(row):
    row.points = row.points - review_points_mean
    return row
```

```
reviews.apply(remean_points, axis='columns')
```

If we had called reviews.apply() with axis='index', then instead of passing a function to transform each row, we would need to give a function to transform each column.

Note that map() and apply() return new, transformed Series and DataFrames, respectively. They don't modify the original data they're called on. If we look at the first row of reviews, we can see that it still has its original points value.


```
review_points_mean = reviews.points.mean()
reviews.points - review_points_mean
```

COUNT HOW MANY TIMES FRUITY IS IN DESCRIPTIONS OVER THE ENTIRE DATAFRAME

```
n_fruity = reviews.description.map(lambda desc: "fruity" in desc).sum()
```

To create a feature that is 1 if the description mentions the word "fruity" and 0 otherwise, you can use a method similar to the one in your example. Here's how you can do it:

```
reviews['is_fruity'] = reviews.description.map(lambda desc: 1 if "fruity"
in desc else 0)
```

We'd like to host these wine reviews on our website, but a rating system ranging from 80 to 100 points is too hard to understand - we'd like to translate them into simple star ratings. A score of 95 or higher counts as 3 stars, a score of at least 85 but less than 95 is 2 stars. Any other score is 1 star.

Also, the Canadian Vintners Association bought a lot of ads on the site, so any wines from Canada should automatically get 3 stars, regardless of points.

Create a series `star_ratings` with the number of stars corresponding to each review in the dataset.

```
def stars(row):
    if row.country == 'Canada':
        return 3
    elif row.points >= 95:
        return 3
    elif row.points >= 85:
        return 2
    else:
        return 1
```

```
star_ratings = reviews.apply(stars, axis='columns')
reviews['star_ratings'] = star_ratings
```

3.4 Group and Sort

```
reviews.groupby('points').points.count()
```

`groupby()` created a group of reviews which allotted the same point values to the given wines. Then, for each of these groups, we grabbed the `points()` column and counted how many times it appeared. `value_counts()` is just a shortcut to this `groupby()` operation.

For example, to get the cheapest wine in each point value category, we can do the following:

```
reviews.groupby('points').price.min()
```

For example, here's one way of selecting the name of the first wine reviewed from each winery in the dataset:

```
reviews.groupby('winery').apply(lambda df: df.title.iloc[0])
```

For an example, here's how we would pick out the best wine by country and province:

```
reviews.groupby(['country', 'province']).apply(lambda df: df.loc[df.points.idxmax()])
```

For example, we can generate a simple statistical summary of the dataset as follows:

```
reviews.groupby(['country']).price.agg([len, min, max])
```

converting back to a regular index, the `reset_index()` method:

```
countries_reviewed = reviews.groupby(['country', 'province']).description.agg([len])
countries_reviewed.reset_index()
```

To get data in the order want it in we can sort it ourselves. The `sort_values()` method is handy for this.

```
countries_reviewed = countries_reviewed.reset_index()
countries_reviewed.sort_values(by='len')
countries_reviewed.sort_values(by='len', ascending=False)
```

Finally, know that you can sort by more than one column at a time:

```
countries_reviewed.sort_values(by=['country', 'len'])
```

Data types

```
reviews.price.dtype
```

Alternatively, the `dtypes` property returns the dtype of every column in the DataFrame:

```
reviews.dtypes
```

For example, we may transform the `points` column from its existing `int64` data type into a `float64` data type:

```
reviews.points.astype('float64')

reviews[pd.isnull(reviews.country)]

reviews.region_2.fillna("Unknown")

reviews.taster_twitter_handle.replace("@kerinokeefe", "@kerino")

reviews.rename(columns={'points': 'score'})

canadian_youtube = pd.read_csv("../input/youtube-new/CAvideos.csv")
british_youtube = pd.read_csv("../input/youtube-new/GBvideos.csv")

pd.concat([canadian_youtube, british_youtube])
```

The middlemost combiner in terms of complexity is `join()`. `join()` lets you combine different DataFrame objects which have an index in common. For example, to pull down videos that happened to be trending on the same day in both Canada and the UK, we could do the following:

```
left = canadian_youtube.set_index(['title', 'trending_date'])
right = british_youtube.set_index(['title', 'trending_date'])

left.join(right, lsuffix='_CAN', rsuffix='_UK')
```

4 Feature Engineering Module

```
X = df.copy()
y = X.pop("CompressiveStrength")

# Create synthetic features
X["FCRatio"] = X["FineAggregate"] / X["CoarseAggregate"]
X["AggCntRatio"] = (X["CoarseAggregate"] + X["FineAggregate"]) / X["Cement"]
X["WtrCntRatio"] = X["Water"] / X["Cement"]

# Train and score model on dataset with additional ratio features
model = RandomForestRegressor(criterion="absolute_error", random_state=0)
score = cross_val_score(
    model, X, y, cv=5, scoring="neg_mean_absolute_error"
)
score = -1 * score.mean()

print(f"MAE Score with Ratio Features: {score:.4}")
```

4.1 Mutual information

Mutual information describes relationships in terms of uncertainty. The mutual information (MI) between two quantities is a measure of the extent to which knowledge of one quantity reduces uncertainty about the other. If you knew the value of a feature, how much more confident would you be about the target?

The entropy of a variable means roughly: "how many yes-or-no questions you would need to describe an occurrence of that variable, on average." The more questions you have to ask, the more uncertain you must be about the variable. Mutual information is how many questions you expect the feature to answer about the target.)

MI can help you to understand the relative potential of a feature as a predictor of the target, considered by itself. It's possible for a feature to be very informative when interacting with other features, but not so informative all alone. MI can't detect interactions between features. It is a univariate metric. The actual usefulness of a feature depends on the model you use it with. A feature is only useful to the extent that its relationship with the target is one your model can learn. Just because a feature has a high MI score doesn't mean your model will be able to do anything with that information. You may need to transform the feature first to expose the association.

```

X = df.copy()
y = X.pop("price")

# Label encoding for categoricals
for colname in X.select_dtypes("object"):
    X[colname], _ = X[colname].factorize()

# All discrete features should now have integer dtypes (double-check
this before using MI!)
discrete_features = X.dtypes == int
Scikit-learn has two mutual information metrics in its feature_selection
module: one for real-valued targets (mutual_info_regression) and one
for categorical targets (mutual_info_classif). Our target, price, is
real-valued. The next cell computes the MI scores for our features
and wraps them up in a nice dataframe.

from sklearn.feature_selection import mutual_info_regression

def make_mi_scores(X, y, discrete_features):
    mi_scores = mutual_info_regression(X, y, discrete_features=discrete_features)
    mi_scores = pd.Series(mi_scores, name="MI Scores", index=X.columns)
    mi_scores = mi_scores.sort_values(ascending=False)
    return mi_scores

mi_scores = make_mi_scores(X, y, discrete_features)

And now a bar plot to make comparisons easier:

def plot_mi_scores(scores):
    scores = scores.sort_values(ascending=True)
    width = np.arange(len(scores))
    ticks = list(scores.index)
    plt.barh(width, scores)
    plt.yticks(width, ticks)
    plt.title("Mutual Information Scores")

plt.figure(dpi=100, figsize=(8, 5))
plot_mi_scores(mi_scores)

import seaborn as sns

plt.style.use("seaborn-whitegrid")

```

```
sns.relplot(x="curb_weight", y="price", data=df);
```

The `fuel_type` feature has a fairly low MI score, but as we can see from the figure, it clearly separates two price populations with different trends within the horsepower feature. This indicates that `fuel_type` contributes an interaction effect and might not be unimportant after all. Before deciding a feature is unimportant from its MI score, it's good to investigate any possible interaction effects -- domain knowledge can offer a lot of guidance here.

```
sns.lmplot(x="horsepower", y="price", hue="fuel_type", data=df);
```

```
features = ["YearBuilt", "MoSold", "ScreenPorch"]
sns.relplot(
    x="value", y="SalePrice", col="variable", data=df.melt(id_vars="SalePrice",
value_vars=features), facet_kws=dict(sharex=False),
);
```

```
# YOUR CODE HERE:
feature = "MoSold"
```

```
sns.lmplot(
    x=feature, y="SalePrice", hue="BldgType", col="BldgType",
    data=df, scatter_kws={"edgecolor": 'w'}, col_wrap=3, height=4,
);
```

Correlation

```
for colname in df.select_dtypes(["object", "category"]):
    X[colname], _ = X[colname].factorize()
discrete_features = [pd.api.types.is_integer_dtype(t) for t in X.dtypes]
```

```
def make_corr_scores(X, y):
    # Compute correlation scores for numerical features only
    numerical_X = X.select_dtypes(include=[np.number]) # Select only
numerical columns
    corr_scores = numerical_X.corrwith(y)

    # Sort by absolute value and retain original sign
```

```

corr_scores = corr_scores.reindex(corr_scores.abs().sort_values(ascending=False).index)
corr_scores = pd.Series(corr_scores, name="Correlation Scores")
return corr_scores

# Example usage
pd.set_option("display.max_rows", None)
corr_scores = make_corr_scores(X, y)
print(corr_scores)

P-values

import pandas as pd
import statsmodels.api as sm

def make_pval_scores(X, y):
    X = X.copy() # Make a copy to avoid modifying the original DataFrame

    # Factorize categorical features to convert them into integer codes
    for colname in X.select_dtypes(["object", "category"]):
        X[colname], _ = X[colname].factorize()

    # Add a constant to the model (intercept)
    X = sm.add_constant(X.select_dtypes(include=[np.number]))

    # Fit the linear regression model
    model = sm.OLS(y, X).fit()

    # Get p-values for each feature and sort by significance
    pval_scores = model.pvalues.drop("const").sort_values()
    return pval_scores

# Example usage
X = df.drop(columns="SalePrice") # Assuming 'df' is your DataFrame
y = df["SalePrice"]

pval_scores = make_pval_scores(X, y)
print(pval_scores)

# Plotting the top 20 p-values (lowest p-values indicate highest significance)
plt.figure(dpi=100, figsize=(8, 5))
plot_mi_scores(pval_scores.head(20))

```

4.2 Create Features

```
autos["stroke_ratio"] = autos.stroke / autos.bore

autos[["stroke", "bore", "stroke_ratio"]].head()

# If the feature has 0.0 values, use np.log1p (log(1+x)) instead of
np.log
accidents["LogWindSpeed"] = accidents.WindSpeed.apply(np.log1p)

# Plot a comparison
fig, axs = plt.subplots(1, 2, figsize=(8, 4))
sns.kdeplot(accidents.WindSpeed, shade=True, ax=axs[0])
sns.kdeplot(accidents.LogWindSpeed, shade=True, ax=axs[1]);
```

In Traffic Accidents are several features indicating whether some roadway object was near the accident. This will create a count of the total number of roadway features nearby using the sum method: (GAAT OVER KOLOMMEN, DUS TELT PER RIJ)

```
roadway_features = ["Amenity", "Bump", "Crossing", "GiveWay",
                    "Junction", "NoExit", "Railway", "Roundabout", "Station", "Stop",
                    "TrafficCalming", "TrafficSignal"]
accidents["RoadwayFeatures"] = accidents[roadway_features].sum(axis=1)

accidents[roadway_features + ["RoadwayFeatures"]].head(10)
```

You could also use a dataframe's built-in methods to create boolean values. In the Concrete dataset are the amounts of components in a concrete formulation. Many formulations lack one or more components (that is, the component has a value of 0). This will count how many components are in a formulation with the dataframe's built-in greater-than gt method:

```
components = [ "Cement", "BlastFurnaceSlag", "FlyAsh", "Water",
               "Superplasticizer", "CoarseAggregate", "FineAggregate"]
concrete["Components"] = concrete[components].gt(0).sum(axis=1)
```



```
concrete[components + ["Components"]].head(10)
```

The `gt(0)` in this line stands for "greater than 0." It creates a boolean DataFrame where each element is True if the value is greater than 0, and False otherwise.

Building up and Breaking Down Features

The `str` accessor lets you apply string methods like `split` directly to columns. The Customer Lifetime Value dataset contains features describing customers of an insurance company. From the Policy feature, we could separate the Type from the Level of coverage:

```
customer[["Type", "Level"]] = ( # Create two new features
    customer["Policy"]          # from the Policy feature
    .str                        # through the string accessor
    .split(" ", expand=True)    # by splitting on " "
                                # and expanding the result into separate
    columns
)
```

```
customer[["Policy", "Type", "Level"]].head(10)
```

Neural nets especially need features scaled to values not too far from 0. Tree-based models (like random forests and XGBoost) can sometimes benefit from normalization, but usually much less so.

Tree models can learn to approximate almost any combination of features, but when a combination is especially important they can still benefit from having it explicitly created, especially when data is limited. Counts are especially helpful for tree models, since these models don't have a natural way of aggregating information across many features at once.

4.2.1 Group Transforms

For an "average income by state", you would choose State for the grouping feature, mean for the aggregation function, and Income for the aggregated feature. To compute this in Pandas, we use the `groupby` and `transform` methods:

```
customer["AverageIncome"] = (
    customer.groupby("State") # for each state
    ["Income"]               # select the income
    .transform("mean")       # and compute its mean
)

customer[["State", "Income", "AverageIncome"]].head(10)

customer["StateFreq"] = (
    customer.groupby("State")
    ["State"]
    .transform("count")
    / customer.State.count()
)

customer[["State", "StateFreq"]].head(10)
```

You could use a transform like this to create a "frequency encoding" for a categorical feature.

If you're using training and validation splits, to preserve their independence, it's best to create a grouped feature using only the training set and then join it to the validation set. We can use the validation set's `merge` method after creating a unique set of values with `drop_duplicates` on the training set:

```
# Create splits
df_train = customer.sample(frac=0.5)
```

```

df_valid = customer.drop(df_train.index)

# Create the average claim amount by coverage type, on the training
set
df_train["AverageClaim"] = df_train.groupby("Coverage")["ClaimAmount"].transform("mean")

# Merge the values into the validation set
df_valid = df_valid.merge(
    df_train[["Coverage", "AverageClaim"]].drop_duplicates(),
    on="Coverage",
    how="left",
)

df_valid[["Coverage", "AverageClaim"]].head(10)

```

ENCODING AGAIN

4.3 Klustering

When used for feature engineering, we could attempt to discover groups of customers representing a market segment, for instance, or geographic areas that share similar weather patterns. Adding a feature of cluster labels can help machine learning models untangle complicated relationships of space or proximity.

Cluster Labels as a Feature

Applied to a single real-valued feature, clustering acts like a traditional "binning" or "discretization" transform. On multiple features, it's like "multi-dimensional binning" (sometimes called vector quantization).

The motivating idea for adding cluster labels is that the clusters will break up complicated relationships across features into simpler chunks. Our model can then just learn the simpler chunks one-by-one instead having to learn the complicated whole all at once. It's a "divide and conquer" strategy.

Since k-means clustering is sensitive to scale, it can be a good idea rescale or normalize data with extreme values. Our features are already roughly on the same scale, so we'll leave them as-is.

As spatial features, California Housing's 'Latitude' and 'Longitude' make natural candidates for k-means clustering. In this example we'll cluster these with 'MedInc' (median income) to create economic segments in different regions of California.

```
X = df.loc[:, ["MedInc", "Latitude", "Longitude"]]
# Create cluster feature
kmeans = KMeans(n_clusters=6)
X["Cluster"] = kmeans.fit_predict(X)
X["Cluster"] = X["Cluster"].astype("category")

X.head()
```

Now let's look at a couple plots to see how effective this was. First, a scatter plot that shows the geographic distribution of the clusters. It seems like the algorithm has created separate segments for higher-income areas on the coasts.

```
sns.relplot(
    x="Longitude", y="Latitude", hue="Cluster", data=X, height=6,
);
```

```
X["MedHouseVal"] = df["MedHouseVal"]
sns.catplot(x="MedHouseVal", y="Cluster", data=X, kind="boxen", height=6);
```

4.4 PCA

PCA Best Practices

There are a few things to keep in mind when applying PCA:

PCA only works with numeric features, like continuous quantities or counts.

PCA is sensitive to scale. It's good practice to standardize your data before applying PCA, unless you know you have good reason not to.

Consider removing or constraining outliers, since they can have an undue influence on the results.

```
# Standardize
X_scaled = (X - X.mean(axis=0)) / X.std(axis=0)
Now we can fit scikit-learn's PCA estimator and create the principal
components. You can see here the first few rows of the transformed
dataset.
```

```
from sklearn.decomposition import PCA
```

```
# Create principal components
pca = PCA()
X_pca = pca.fit_transform(X_scaled)
```

```
# Convert to dataframe
component_names = [f"PC{i+1}" for i in range(X_pca.shape[1])]
X_pca = pd.DataFrame(X_pca, columns=component_names)
```

```
X_pca.head()
```

After fitting, the PCA instance contains the loadings in its `components_` attribute. (Terminology for PCA is inconsistent, unfortunately. We're following the convention that calls the transformed columns in `X_pca` the components, which otherwise don't have a name.) We'll wrap the loadings up in a dataframe.

```
loadings = pd.DataFrame(
    pca.components_.T, # transpose the matrix of loadings
    columns=component_names, # so the columns are the principal components
    index=X.columns, # and the rows are the original features
)
loadings
```

```
# Look at explained variance
plot_variance(pca);
```

```
mi_scores = make_mi_scores(X_pca, y, discrete_features=False)
mi_scores
```

4.5 PCA part 2

```

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.feature_selection import mutual_info_regression
from sklearn.model_selection import cross_val_score
from xgboost import XGBRegressor

# Set Matplotlib defaults
plt.style.use("seaborn-whitegrid")
plt.rc("figure", autolayout=True)
plt.rc(
    "axes",
    labelweight="bold",
    labelsiz="large",
    titleweight="bold",
    titlesiz=14,
    titlepad=10,
)

def apply_pca(X, standardize=True):
    # Standardize
    if standardize:
        X = (X - X.mean(axis=0)) / X.std(axis=0)
    # Create principal components
    pca = PCA()
    X_pca = pca.fit_transform(X)
    # Convert to dataframe
    component_names = [f"PC{i+1}" for i in range(X_pca.shape[1])]
    X_pca = pd.DataFrame(X_pca, columns=component_names)
    # Create loadings
    loadings = pd.DataFrame(
        pca.components_.T, # transpose the matrix of loadings
        columns=component_names, # so the columns are the principal
        components
        index=X.columns, # and the rows are the original features
    )

```

```

    )
    return pca, X_pca, loadings

def plot_variance(pca, width=8, dpi=100):
    # Create figure
    fig, axs = plt.subplots(1, 2)
    n = pca.n_components_
    grid = np.arange(1, n + 1)
    # Explained variance
    evr = pca.explained_variance_ratio_
    axs[0].bar(grid, evr)
    axs[0].set(
        xlabel="Component", title="% Explained Variance", ylim=(0.0,
1.0)
    )
    # Cumulative Variance
    cv = np.cumsum(evr)
    axs[1].plot(np.r_[0, grid], np.r_[0, cv], "o-")
    axs[1].set(
        xlabel="Component", title="% Cumulative Variance", ylim=(0.0,
1.0)
    )
    # Set up figure
    fig.set(figwidth=8, dpi=100)
    return axs

def make_mi_scores(X, y):
    X = X.copy()
    for colname in X.select_dtypes(["object", "category"]):
        X[colname], _ = X[colname].factorize()
    # All discrete features should now have integer dtypes
    discrete_features = [pd.api.types.is_integer_dtype(t) for t in
X.dtypes]
    mi_scores = mutual_info_regression(X, y, discrete_features=discrete_features,
random_state=0)
    mi_scores = pd.Series(mi_scores, name="MI Scores", index=X.columns)
    mi_scores = mi_scores.sort_values(ascending=False)
    return mi_scores

def score_dataset(X, y, model=XGBRegressor()):
    # Label encoding for categoricals
    for colname in X.select_dtypes(["category", "object"]):
        X[colname], _ = X[colname].factorize()

```

```

    # Metric for Housing competition is RMSLE (Root Mean Squared Log
    Error)
    score = cross_val_score(
        model, X, y, cv=5, scoring="neg_mean_squared_log_error",
    )
    score = -1 * score.mean()
    score = np.sqrt(score)
    return score

```

```
df = pd.read_csv("../input/fe-course-data/ames.csv")
```

The first component, PC1, seems to be a kind of "size" component, similar to what we saw in the tutorial: all of the features have the same sign (positive), indicating that this component is describing a contrast between houses having large values and houses having small values for these features.

The interpretation of the third component PC3 is a little trickier. The features GarageArea and YearRemodAdd both have near-zero loadings, so let's ignore those. This component is mostly about TotalBsmtSF and GrLivArea. It describes a contrast between houses with a lot of living area but small (or non-existent) basements, and the opposite: small houses with large basements.

PC3 is negatief in T0talBasementSF en positief in GRLivAREA

The first component, PC1, seems to be a kind of "size" component, similar to what we saw in the tutorial: all of the features have the same sign (positive), indicating that this component is describing a contrast between houses having large values and houses having small values for these features.

The interpretation of the third component PC3 is a little trickier. The features GarageArea and YearRemodAdd both have near-zero loadings, so let's ignore those. This component is mostly about TotalBsmtSF and GrLivArea. It describes a contrast between houses with a lot of living area but small (or non-existent) basements, and the opposite: small houses with large basements.

4.6 Target Encoding

4.7 BOLUS

5 Time Series

```
import numpy as np

df['Time'] = np.arange(len(df.index))

df.head()

import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use("seaborn-whitegrid")
plt.rc(
    "figure",
    autolayout=True,
    figsize=(11, 4),
    titlesize=18,
    titleweight='bold',
)
plt.rc(
    "axes",
    labelweight="bold",
    labelsiz="large",
    titleweight="bold",
    titlesize=16,
    titlepad=10,
)
%config InlineBackend.figure_format = 'retina'

fig, ax = plt.subplots()
ax.plot('Time', 'Hardcover', data=df, color='0.75')
ax = sns.regplot(x='Time', y='Hardcover', data=df, ci=None, scatter_kws=dict(color='0.25'))
ax.set_title('Time Plot of Hardcover Sales');
```

5.0.1 Lag features

To make a lag feature we shift the observations of the target series so that they appear to have occurred later in time. Here we've created a 1-step lag feature, though shifting by multiple steps is possible too.

```
df['Harcover_Lag_1'] = df['Hardcover'].shift(1)
```

```
df.head()
```

5.1 Trends

Notice how the Mauna Loa series above has a repeating up and down movement year after year – a short-term, seasonal change. For a change to be a part of the trend, it should occur over a longer period than any seasonal changes. To visualize a trend, therefore, we take an average over a period longer than any seasonal period in the series. For the Mauna Loa series, we chose a window of size 12 to smooth over the season within each year.

```
moving_average = tunnel.rolling(
    window=365,      # 365-day window
    center=True,     # puts the average at the center of the window
    min_periods=183, # choose about half the window size
).mean()            # compute the mean (could also do median, std,
min, max, ...)

ax = tunnel.plot(style=".", color="0.5")
moving_average.plot(
    ax=ax, linewidth=3, title="Tunnel Traffic - 365-Day Moving Average",
    legend=False,
);

trend = food_sales.rolling(
    window=12,
    center=True,
    min_periods=6,
).mean()

# Check your answer
q_1.check()

# Make a plot
ax = food_sales.plot(**plot_params, alpha=0.5)
ax = trend.plot(ax=ax, linewidth=3)
```

Engineering Trend

Once we've identified the shape of the trend, we can attempt to model it using a time-step feature. We've already seen how using the time dummy itself will model a linear trend:

$$\text{target} = a * \text{time} + b$$

We can fit many other kinds of trend through transformations of the time dummy. If the trend appears to be quadratic (a parabola), we just need to add the square of the time dummy to the feature set, giving us:

$$\text{target} = a * \text{time} ** 2 + b * \text{time} + c$$

Linear regression will learn the coefficients a , b , and c

```
from statsmodels.tsa.deterministic import DeterministicProcess

dp = DeterministicProcess(
    index=tunnel.index, # dates from the training data
    constant=True,      # dummy feature for the bias (y_intercept)
    order=1,            # the time dummy (trend)
    drop=True,          # drop terms if necessary to avoid collinearity
)
# 'in_sample' creates features for the dates given in the 'index' argument
X = dp.in_sample()

X.head()
```

```
from sklearn.linear_model import LinearRegression

y = tunnel["NumVehicles"] # the target

# The intercept is the same as the 'const' feature from
# DeterministicProcess. LinearRegression behaves badly with duplicated
# features, so we need to be sure to exclude it here.
```

```

model = LinearRegression(fit_intercept=False)
model.fit(X, y)

y_pred = pd.Series(model.predict(X), index=X.index)

ax = tunnel.plot(style=".", color="0.5", title="Tunnel Traffic - Linear
Trend")
_ = y_pred.plot(ax=ax, linewidth=3, label="Trend")

X = dp.out_of_sample(steps=30)

y_fore = pd.Series(model.predict(X), index=X.index)

y_fore.head()

ax = tunnel["2005-05:"].plot(title="Tunnel Traffic - Linear Trend Forecast",
**plot_params)
ax = y_pred["2005-05:"].plot(ax=ax, linewidth=3, label="Trend")
ax = y_fore.plot(ax=ax, linewidth=3, label="Trend Forecast", color="C3")
_ = ax.legend()

```

5.2 Seasonality

We say that a time series exhibits seasonality whenever there is a regular, periodic change in the mean of the series. Seasonal changes generally follow the clock and calendar – repetitions over a day, a week, or a year are common. Seasonality is often driven by the cycles of the natural world over days and years or by conventions of social behavior surrounding dates and times.

Seasonal indicators are binary features that represent seasonal differences in the level of a time series. Seasonal indicators are what you get if you treat a seasonal period as a categorical feature and apply one-hot encoding. The kind of feature we discuss now are better suited for long seasons over many observations where indicators would be impractical. Instead of creating a feature for each date, Fourier features try to capture the overall shape of the seasonal curve with just a few features.

```

from pathlib import Path
from warnings import simplefilter

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.linear_model import LinearRegression
from statsmodels.tsa.deterministic import CalendarFourier, DeterministicProcess

simplefilter("ignore")

# Set Matplotlib defaults
plt.style.use("seaborn-whitegrid")
plt.rc("figure", autolayout=True, figsize=(11, 5))
plt.rc(
    "axes",
    labelweight="bold",
    labelsiz="large",
    titleweight="bold",
    titlesiz=16,
    titlepad=10,
)
plot_params = dict(
    color="0.75",
    style=".-",
    markeredgecolor="0.25",
    markerfacecolor="0.25",
    legend=False,
)
%config InlineBackend.figure_format = 'retina'

# annotations: https://stackoverflow.com/a/49238256/5769929
def seasonal_plot(X, y, period, freq, ax=None):
    if ax is None:
        _, ax = plt.subplots()
    palette = sns.color_palette("husl", n_colors=X[period].nunique(),)
    ax = sns.lineplot(
        x=freq,
        y=y,
        hue=period,
        data=X,
        ci=False,
        ax=ax,
        palette=palette,
        legend=False,
    )

```

```

    )
    ax.set_title(f"Seasonal Plot ({period}/{freq})")
    for line, name in zip(ax.lines, X[period].unique()):
        y_ = line.get_ydata()[-1]
        ax.annotate(
            name,
            xy=(1, y_),
            xytext=(6, 0),
            color=line.get_color(),
            xycoords=ax.get_yaxis_transform(),
            textcoords="offset points",
            size=14,
            va="center",
        )
    return ax

def plot_periodogram(ts, detrend='linear', ax=None):
    from scipy.signal import periodogram
    fs = pd.Timedelta("365D") / pd.Timedelta("1D")
    frequencies, spectrum = periodogram(
        ts,
        fs=fs,
        detrend=detrend,
        window="boxcar",
        scaling='spectrum',
    )
    if ax is None:
        _, ax = plt.subplots()
    ax.step(frequencies, spectrum, color="purple")
    ax.set_xscale("log")
    ax.set_xticks([1, 2, 4, 6, 12, 26, 52, 104])
    ax.set_xticklabels(
        [
            "Annual (1)",
            "Semiannual (2)",
            "Quarterly (4)",
            "Bimonthly (6)",
            "Monthly (12)",
            "Biweekly (26)",
            "Weekly (52)",
            "Semiweekly (104)",
        ],
        rotation=30,
    )
    ax.ticklabel_format(axis="y", style="sci", scilimits=(0, 0))

```

```

        ax.set_ylabel("Variance")
        ax.set_title("Periodogram")
        return ax

data_dir = Path("../input/ts-course-data")
tunnel = pd.read_csv(data_dir / "tunnel.csv", parse_dates=["Day"])
tunnel = tunnel.set_index("Day").to_period("D")

X = tunnel.copy()

# days within a week
X["day"] = X.index.dayofweek # the x-axis (freq)
X["week"] = X.index.week # the seasonal period (period)

# days within a year
X["dayofyear"] = X.index.dayofyear
X["year"] = X.index.year
fig, (ax0, ax1) = plt.subplots(2, 1, figsize=(11, 6))
seasonal_plot(X, y="NumVehicles", period="week", freq="day", ax=ax0)
seasonal_plot(X, y="NumVehicles", period="year", freq="dayofyear",
ax=ax1);

plot_periodogram(tunnel.NumVehicles);

```

The periodogram agrees with the seasonal plots above: a strong weekly season and a weaker annual season. The weekly season we'll model with indicators and the annual season with Fourier features. From right to left, the periodogram falls off between Bimonthly (6) and Monthly (12), so let's use 10 Fourier pairs.

We'll create our seasonal features using `DeterministicProcess`, the same utility we used in Lesson 2 to create trend features. To use two seasonal periods (weekly and annual), we'll need to instantiate one of them as an "additional term":

```

from statsmodels.tsa.deterministic import CalendarFourier, DeterministicProcess

fourier = CalendarFourier(freq="A", order=10) # 10 sin/cos pairs for

```



```

"A"nnual seasonality

dp = DeterministicProcess(
    index=tunnel.index,
    constant=True,          # dummy feature for bias (y-intercept)
    order=1,                # trend (order 1 means linear)
    seasonal=True,          # weekly seasonality (indicators)
    additional_terms=[fourier], # annual seasonality (fourier)
    drop=True,              # drop terms to avoid collinearity
)

X = dp.in_sample() # create features for dates in tunnel.index

y = tunnel["NumVehicles"]

model = LinearRegression(fit_intercept=False)
_ = model.fit(X, y)

y_pred = pd.Series(model.predict(X), index=y.index)
X_fore = dp.out_of_sample(steps=90)
y_fore = pd.Series(model.predict(X_fore), index=X_fore.index)

ax = y.plot(color='0.25', style='.', title="Tunnel Traffic - Seasonal
Forecast")
ax = y_pred.plot(ax=ax, label="Seasonal")
ax = y_fore.plot(ax=ax, label="Seasonal Forecast", color='C3')
_ = ax.legend()

```

5.3 SEASONAL EXERCISE

```

# Setup feedback system
from learntools.core import binder
binder.bind(globals())
from learntools.time_series.ex3 import *

# Setup notebook
from pathlib import Path
from learntools.time_series.style import * # plot style settings
from learntools.time_series.utils import plot_periodogram, seasonal_plot

import pandas as pd

```

```

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.linear_model import LinearRegression
from statsmodels.tsa.deterministic import CalendarFourier, DeterministicProcess

comp_dir = Path('../input/store-sales-time-series-forecasting')

holidays_events = pd.read_csv(
    comp_dir / "holidays_events.csv",
    dtype={
        'type': 'category',
        'locale': 'category',
        'locale_name': 'category',
        'description': 'category',
        'transferred': 'bool',
    },
    parse_dates=['date'],
    infer_datetime_format=True,
)
holidays_events = holidays_events.set_index('date').to_period('D')

store_sales = pd.read_csv(
    comp_dir / 'train.csv',
    usecols=['store_nbr', 'family', 'date', 'sales'],
    dtype={
        'store_nbr': 'category',
        'family': 'category',
        'sales': 'float32',
    },
    parse_dates=['date'],
    infer_datetime_format=True,
)
store_sales['date'] = store_sales.date.dt.to_period('D')
store_sales = store_sales.set_index(['store_nbr', 'family', 'date']).sort_index()
average_sales = (
    store_sales
    .groupby('date').mean()
    .squeeze()
    .loc['2017']
)

X = average_sales.to_frame()
X["week"] = X.index.week

```

```
X["day"] = X.index.dayofweek
seasonal_plot(X, y='sales', period='week', freq='day');
```

```
plot_periodogram(average_sales);
```

Correct:

Both the seasonal plot and the periodogram suggest a strong weekly seasonality. From the periodogram, it appears there may be some monthly and biweekly components as well. In fact, the notes to the Store Sales dataset say wages in the public sector are paid out biweekly, on the 15th and last day of the month -- a possible origin for these seasons.

```
y = average_sales.copy()
```

```
fourier = CalendarFourier(freq='M', order=4)
dp = DeterministicProcess(
    index=y.index,
    constant=True,
    order=1,
    seasonal=True,
    additional_terms=[fourier],
    drop=True,
)
X = dp.in_sample()
```

```
model = LinearRegression().fit(X, y)
y_pred = pd.Series(
    model.predict(X),
    index=X.index,
    name='Fitted',
)
```

```
y_pred = pd.Series(model.predict(X), index=X.index)
ax = y.plot(**plot_params, alpha=0.5, title="Average Sales", ylabel="items sold")
ax = y_pred.plot(ax=ax, label="Seasonal")
ax.legend();
```

```

y_deseason = y - y_pred

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, sharey=True, figsize=(10,
7))
ax1 = plot_periodogram(y, ax=ax1)
ax1.set_title("Product Sales Frequency Components")
ax2 = plot_periodogram(y_deseason, ax=ax2);
ax2.set_title("Deseasonalized");

```

5.4 Serial Dependence

One especially common way for serial dependence to manifest is in cycles. Cycles are patterns of growth and decay in a time series associated with how the value in a series at one time depends on values at previous times, but not necessarily on the time step itself. Cyclic behavior is characteristic of systems that can affect themselves or whose reactions persist over time. Economies, epidemics, animal populations, volcano eruptions, and similar natural phenomena often display cyclic behavior.

What distinguishes cyclic behavior from seasonality is that cycles are not necessarily time dependent, as seasons are. What happens in a cycle is less about the particular date of occurrence, and more about what has happened in the recent past. The (at least relative) independence from time means that cyclic behavior can be much more irregular than seasonality.

```

import pandas as pd

# Federal Reserve dataset: https://www.kaggle.com/federalreserve/interest-rates
reserve = pd.read_csv(
    "../input/ts-course-data/reserve.csv",
    parse_dates={'Date': ['Year', 'Month', 'Day']},
    index_col='Date',
)

y = reserve.loc[:, 'Unemployment Rate'].dropna().to_period('M')
df = pd.DataFrame({
    'y': y,
    'y_lag_1': y.shift(1),
    'y_lag_2': y.shift(2),
})

df.head()

```

The partial autocorrelation tells you the correlation of a lag accounting for all of the previous lags -- the amount of "new" correlation the lag contributes, so to speak. Plotting the partial autocorrelation can help you choose which lag features to use. In the figure below, lag 1 through lag 6 fall outside the intervals of "no correlation" (in blue), so we might choose lags 1 through lag 6 as features for US Unemployment. (Lag 11 is likely a false positive.)

```
PLOT PARTIAL AUTOCORRELATION FUNCTION
plot_pacf(y_deseason, lags=8);
plot_lags(y_deseason, lags=8, nrows=2);
```

```
Adding plot title.
plt.title("Autocorrelation Plot")

# Providing x-axis name.
plt.xlabel("Lags")

# Plotting the Autocorrelation plot.
plt.acorr(y_deseason, maxlags = 10)

# Displaying the plot.
print("The Autocorrelation plot for the data is:")
plt.grid(True)

plt.show()

plot_pacf(y_deseason, lags=8);

plot_lags(y_deseason, lags=8, nrows=2);
```

5.5 LAG 2

```
# Setup feedback system
from learntools.core import binder
binder.bind(globals())
from learntools.time_series.ex4 import *

# Setup notebook
from pathlib import Path
from learntools.time_series.style import * # plot style settings
from learntools.time_series.utils import plot_lags, make_lags, make_leads

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_log_error
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.tsa.deterministic import CalendarFourier, DeterministicProcess

comp_dir = Path('../input/store-sales-time-series-forecasting')

store_sales = pd.read_csv(
    comp_dir / 'train.csv',
    usecols=['store_nbr', 'family', 'date', 'sales', 'onpromotion'],
    dtype={
        'store_nbr': 'category',
        'family': 'category',
        'sales': 'float32',
        'onpromotion': 'uint32',
    },
    parse_dates=['date'],
    infer_datetime_format=True,
)
store_sales['date'] = store_sales.date.dt.to_period('D')
store_sales = store_sales.set_index(['store_nbr', 'family', 'date']).sort_index()

family_sales = (
    store_sales
    .groupby(['family', 'date'])
    .mean()
    .unstack('family')
    .loc['2017', ['sales', 'onpromotion']]
)
```

```

supply_sales = family_sales.loc(axis=1)[:, 'SCHOOL AND OFFICE SUPPLIES']
y = supply_sales.loc[:, 'sales'].squeeze()

fourier = CalendarFourier(freq='M', order=4)
dp = DeterministicProcess(
    constant=True,
    index=y.index,
    order=1,
    seasonal=True,
    drop=True,
    additional_terms=[fourier],
)
X_time = dp.in_sample()
X_time['NewYearsDay'] = (X_time.index.dayofyear == 1)

model = LinearRegression(fit_intercept=False)
model.fit(X_time, y)
y_deseason = y - model.predict(X_time)
y_deseason.name = 'sales_deseasoned'

ax = y_deseason.plot()
ax.set_title("Sales of School and Office Supplies (deseasonalized)");

y_ma = y.rolling(7, center=True).mean()

ax = y_ma.plot()
ax.set_title("Seven-Day Moving Average");

y_ma = y.rolling(7, center=True).mean()

ax = y_ma.plot()
ax.set_title("Seven-Day Moving Average");

plot_pacf(y_deseason, lags=8);

plot_lags(y_deseason, lags=8, nrows=2);

```

```

# Adding plot title.
plt.title("Autocorrelation Plot")

# Providing x-axis name.
plt.xlabel("Lags")

# Plotting the Autocorrelation plot.
plt.acorr(y_deseason, maxlags = 10)

# Displaying the plot.
print("The Autocorrelation plot for the data is:")
plt.grid(True)

plt.show()

onpromotion = supply_sales.loc[:, 'onpromotion'].squeeze().rename('onpromotion')

# Drop days without promotions
plot_lags(x=onpromotion.loc[onpromotion > 1], y=y_deseason.loc[onpromotion
> 1], lags=3, leads=3, nrows=1);

X_lags = make_lags(y_deseason, lags=1)

X_promo = pd.concat([
    make_lags(onpromotion, lags=1),
    onpromotion,
    make_leads(onpromotion, leads=1),
], axis=1)

X = pd.concat([X_time, X_lags, X_promo], axis=1).dropna()
y, X = y.align(X, join='inner')

from sklearn.model_selection import train_test_split

X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=30,
shuffle=False)

model = LinearRegression(fit_intercept=False).fit(X_train, y_train)
y_fit = pd.Series(model.predict(X_train), index=X_train.index).clip(0.0)
y_pred = pd.Series(model.predict(X_valid), index=X_valid.index).clip(0.0)

rmsle_train = mean_squared_log_error(y_train, y_fit) ** 0.5

```



```

rmsle_valid = mean_squared_log_error(y_valid, y_pred) ** 0.5
print(f'Training RMSLE: {rmsle_train:.5f}')
print(f'Validation RMSLE: {rmsle_valid:.5f}')

ax = y.plot(**plot_params, alpha=0.5, title="Average Sales", ylabel="items
sold")
ax = y_fit.plot(ax=ax, label="Fitted", color='C0')
ax = y_pred.plot(ax=ax, label="Forecast", color='C3')
ax.legend();

y_lag = supply_sales.loc[:, 'sales'].shift(1)
onpromo = supply_sales.loc[:, 'onpromotion']

mean_7 = y_lag.rolling(7).mean()
median_14 = y_lag.rolling(14).median()
std_7 = y_lag.rolling(7).std()
promo_7 = onpromo.rolling(7, center=True).sum()

```

5.6 Hybrid

Introduction

Linear regression excels at extrapolating trends, but can't learn interactions. XGBoost excels at learning interactions, but can't extrapolate trends. In this lesson, we'll learn how to create "hybrid" forecasters that combine complementary learning algorithms and let the strengths of one make up for the weakness of the other.

Components and Residuals

So that we can design effective hybrids, we need a better understanding of how time series are constructed. We've studied up to now three patterns of dependence: trend, seasons, and cycles. Many time series can be closely described by an additive model of just these three components plus some essentially unpredictable, entirely random error:

series = trend + seasons + cycles + error

Each of the terms in this model we would then call a component of the time series.

The residuals of a model are the difference between the target the model was trained on and the predictions the model makes -- the difference between the actual curve and the fitted curve, in other words. Plot the residuals against a feature, and you get the "left over" part of the target, or what the model failed to learn about the target from that feature.

The difference between the target series and the predictions (blue) gives the series of residuals. On the left of the figure above is a portion of the Tunnel Traffic series and the trend-seasonal curve from Lesson 3. Subtracting out the fitted curve leaves the residuals, on the right. The residuals contain everything from Tunnel Traffic the trend-seasonal model didn't learn.

We could imagine learning the components of a time series as an iterative process: first learn the trend and subtract it out from the series, then learn the seasonality from the detrended residuals and subtract the seasons out, then learn the cycles and subtract the cycles out, and finally only the unpredictable error remains.

Learning the components of Mauna Loa CO2 step by step. Subtract the fitted curve (blue) from its series to get the series in the next step. Add together all the components we learned and we get the complete model. This is essentially what linear regression would do if you trained it on a complete set of features modeling trend, seasons, and cycles.

Add the learned components to get a complete model.

Hybrid Forecasting with Residuals

In previous lessons, we used a single algorithm (linear regression) to learn all the components at once. But it's also possible to use one algorithm for some of the components and another algorithm for the rest. This way we can always choose the best algorithm for each component. To do this, we use one algorithm to fit the original series and then the second algorithm to fit the residual series.

In detail, the process is this:

```
# 1. Train and predict with first model
model_1.fit(X_train_1, y_train)
y_pred_1 = model_1.predict(X_train)
```

```
# 2. Train and predict with second model on residuals
model_2.fit(X_train_2, y_train - y_pred_1)
y_pred_2 = model_2.predict(X_train_2)

# 3. Add to get overall predictions
y_pred = y_pred_1 + y_pred_2
```

We'll usually want to use different feature sets (`X_train_1` and `X_train_2` above) depending on what we want each model to learn. If we use the first model to learn the trend, we generally wouldn't need a trend feature for the second model, for example.

While it's possible to use more than two models, in practice it doesn't seem to be especially helpful. In fact, the most common strategy for constructing hybrids is the one we've just described: a simple (usually linear) learning algorithm followed by a complex, non-linear learner like GBDTs or a deep neural net, the simple model typically designed as a "helper" for the powerful algorithm that follows.

Designing Hybrids

There are many ways you could combine machine learning models besides the way we've outlined in this lesson. Successfully combining models, though, requires that we dig a bit deeper into how these algorithms operate.

There are generally two ways a regression algorithm can make predictions: either by transforming the features or by transforming the target. Feature-transforming algorithms learn some mathematical function that takes features as an input and then combines and transforms them to produce an output that matches the target values in the training set. Linear regression and neural nets are of this kind.

Target-transforming algorithms use the features to group the target values in the training set and make predictions by averaging values in a group; a set of feature just indicates which group to average. Decision trees and nearest neighbors are of this kind.

The important thing is this: feature transformers generally can extrapolate target values beyond the training set given appropriate features as inputs, but the predictions of target transformers will always be bound within the range of the training set. If the time dummy continues counting

time steps, linear regression continues drawing the trend line. Given the same time dummy, a decision tree will predict the trend indicated by the last step of the training data into the future forever. Decision trees cannot extrapolate trends. Random forests and gradient boosted decision trees (like XGBoost) are ensembles of decision trees, so they also cannot extrapolate trends.

A decision tree will fail to extrapolate a trend beyond the training set.

This difference is what motivates the hybrid design in this lesson: use linear regression to extrapolate the trend, transform the target to remove the trend, and apply XGBoost to the detrended residuals. To hybridize a neural net (a feature transformer), you could instead include the predictions of another model as a feature, which the neural net would then include as part of its own predictions. The method of fitting to residuals is actually the same method the gradient boosting algorithm uses, so we will call these boosted hybrids; the method of using predictions as features is known as "stacking", so we will call these stacked hybrids

Example - US Retail Sales

The US Retail Sales dataset contains monthly sales data for various retail industries from 1992 to 2019, as collected by the US Census Bureau. Our goal will be to forecast sales in the years 2016-2019 given sales in the earlier years. In addition to creating a linear regression + XGBoost hybrid, we'll also see how to set up a time series dataset for use with XGBoost.

```
from pathlib import Path
from warnings import simplefilter

import matplotlib.pyplot as plt
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from statsmodels.tsa.deterministic import CalendarFourier, DeterministicProcess
from xgboost import XGBRegressor

simplefilter("ignore")

# Set Matplotlib defaults
plt.style.use("seaborn-whitegrid")
plt.rc(
```

```

        "figure",
        autolayout=True,
        figsize=(11, 4),
        titlesize=18,
        titleweight='bold',
    )
plt.rc(
    "axes",
    labelweight="bold",
    labelsizes="large",
    titleweight="bold",
    titlesize=16,
    titlepad=10,
)
plot_params = dict(
    color="0.75",
    style=".-",
    markeredgecolor="0.25",
    markerfacecolor="0.25",
)

data_dir = Path("../input/ts-course-data/")
industries = ["BuildingMaterials", "FoodAndBeverage"]
retail = pd.read_csv(
    data_dir / "us-retail-sales.csv",
    usecols=['Month'] + industries,
    parse_dates=['Month'],
    index_col='Month',
).to_period('D').reindex(columns=industries)
retail = pd.concat({'Sales': retail}, names=[None, 'Industries'], axis=1)

retail.head()
Sales
Industries BuildingMaterials FoodAndBeverage
Month
1992-01-01 8964 29589
1992-02-01 9023 28570
1992-03-01 10608 29682
1992-04-01 11630 30228
1992-05-01 12327 31677
First let's use a linear regression model to learn the trend in each
series. For demonstration, we'll use a quadratic (order 2) trend. (The
code here is basically the same as that in previous lessons.) Though
the fit isn't perfect, it will be enough for our needs.

y = retail.copy()

```

```

# Create trend features
dp = DeterministicProcess(
    index=y.index, # dates from the training data
    constant=True, # the intercept
    order=2,       # quadratic trend
    drop=True,     # drop terms to avoid collinearity
)
X = dp.in_sample() # features for the training data

# Test on the years 2016-2019. It will be easier for us later if we
# split the date index instead of the dataframe directly.
idx_train, idx_test = train_test_split(
    y.index, test_size=12 * 4, shuffle=False,
)
X_train, X_test = X.loc[idx_train, :], X.loc[idx_test, :]
y_train, y_test = y.loc[idx_train], y.loc[idx_test]

# Fit trend model
model = LinearRegression(fit_intercept=False)
model.fit(X_train, y_train)

# Make predictions
y_fit = pd.DataFrame(
    model.predict(X_train),
    index=y_train.index,
    columns=y_train.columns,
)
y_pred = pd.DataFrame(
    model.predict(X_test),
    index=y_test.index,
    columns=y_test.columns,
)

# Plot
axs = y_train.plot(color='0.25', subplots=True, sharex=True)
axs = y_test.plot(color='0.25', subplots=True, sharex=True, ax=axs)
axs = y_fit.plot(color='C0', subplots=True, sharex=True, ax=axs)
axs = y_pred.plot(color='C3', subplots=True, sharex=True, ax=axs)
for ax in axs: ax.legend([])
_ = plt.suptitle("Trends")

```

While the linear regression algorithm is capable of multi-output regression, the XGBoost algorithm is not. To predict multiple series at once with XGBoost, we'll instead convert these series from wide format, with one time series per column, to long format, with series indexed by

categories along rows.

```
# The 'stack' method converts column labels to row labels, pivoting
from wide format to long
X = retail.stack() # pivot dataset wide to long
display(X.head())
y = X.pop('Sales') # grab target series
Sales
Month Industries
1992-01-01 BuildingMaterials 8964
FoodAndBeverage 29589
1992-02-01 BuildingMaterials 9023
FoodAndBeverage 28570
1992-03-01 BuildingMaterials 10608
So that XGBoost can learn to distinguish our two time series, we'll
turn the row labels for 'Industries' into a categorical feature with
a label encoding. We'll also create a feature for annual seasonality
by pulling the month numbers out of the time index.

# Turn row labels into categorical feature columns with a label encoding
X = X.reset_index('Industries')
# Label encoding for 'Industries' feature
for colname in X.select_dtypes(["object", "category"]):
    X[colname], _ = X[colname].factorize()

# Label encoding for annual seasonality
X["Month"] = X.index.month # values are 1, 2, ..., 12

# Create splits
X_train, X_test = X.loc[idx_train, :], X.loc[idx_test, :]
y_train, y_test = y.loc[idx_train], y.loc[idx_test]
Now we'll convert the trend predictions made earlier to long format
and then subtract them from the original series. That will give us
detrended (residual) series that XGBoost can learn.

# Pivot wide to long (stack) and convert DataFrame to Series (squeeze)
y_fit = y_fit.stack().squeeze() # trend from training set
y_pred = y_pred.stack().squeeze() # trend from test set

# Create residuals (the collection of detrended series) from the training
set
y_resid = y_train - y_fit

# Train XGBoost on the residuals
xgb = XGBRegressor()
xgb.fit(X_train, y_resid)
```

```

# Add the predicted residuals onto the predicted trends
y_fit_boosted = xgb.predict(X_train) + y_fit
y_pred_boosted = xgb.predict(X_test) + y_pred
The fit appears quite good, though we can see how the trend learned
by XGBoost is only as good as the trend learned by the linear regression
-- in particular, XGBoost wasn't able to compensate for the poorly
fit trend in the 'BuildingMaterials' series.

axs = y_train.unstack(['Industries']).plot(
    color='0.25', figsize=(11, 5), subplots=True, sharex=True,
    title=['BuildingMaterials', 'FoodAndBeverage'],
)
axs = y_test.unstack(['Industries']).plot(
    color='0.25', subplots=True, sharex=True, ax=axs,
)
axs = y_fit_boosted.unstack(['Industries']).plot(
    color='C0', subplots=True, sharex=True, ax=axs,
)
axs = y_pred_boosted.unstack(['Industries']).plot(
    color='C3', subplots=True, sharex=True, ax=axs,
)
for ax in axs: ax.legend([])

```

5.6.1 Exercise Hybrid

```

# You'll add fit and predict methods to this minimal class
class BoostedHybrid:
    def __init__(self, model_1, model_2):
        self.model_1 = model_1
        self.model_2 = model_2
        self.y_columns = None # store column names from fit method

    def fit(self, X_1, X_2, y):
        # Train model_1
        self.model_1.fit(X_1, y)

        # Make predictions
        y_fit = pd.DataFrame(
            self.model_1.predict(X_1),
            index=X_1.index, columns=y.columns,

```



```

    )

    # Compute residuals
    y_resid = y - y_fit
    y_resid = y_resid.stack().squeeze() # wide to long

    # Train model_2 on residuals
    self.model_2.fit(X_2, y_resid)

    # Save column names for predict method
    self.y_columns = y.columns
    # Save data for question checking
    self.y_fit = y_fit
    self.y_resid = y_resid

# Add method to class
BoostedHybrid.fit = fit

def predict(self, X_1, X_2):
    # Predict with model_1
    y_pred = pd.DataFrame(
        self.model_1.predict(X_1),
        index=X_1.index, columns=self.y_columns,
    )
    y_pred = y_pred.stack().squeeze() # wide to long

    # Add model_2 predictions to model_1 predictions
    y_pred += self.model_2.predict(X_2)

    return y_pred.unstack()

# Add method to class
BoostedHybrid.predict = predict

# Target series
y = family_sales.loc[:, 'sales']

# X_1: Features for Linear Regression
dp = DeterministicProcess(index=y.index, order=1)
X_1 = dp.in_sample()

```

```

# X_2: Features for XGBoost
X_2 = family_sales.drop('sales', axis=1).stack() # onpromotion feature

# Label encoding for 'family'
le = LabelEncoder() # from sklearn.preprocessing
X_2 = X_2.reset_index('family')
X_2['family'] = le.fit_transform(X_2['family'])

# Label encoding for seasonality
X_2["day"] = X_2.index.day # values are day of the month

model = BoostedHybrid(
    model_1=LinearRegression(),
    model_2=XGBRegressor(),
)
model.fit(X_1, X_2, y)

y_pred = model.predict(X_1, X_2)
y_pred = y_pred.clip(0.0)

model = BoostedHybrid(
    model_1=LinearRegression(),
    model_2=XGBRegressor(),
)
model.fit(X_1, X_2, y)

y_pred = model.predict(X_1, X_2)
y_pred = y_pred.clip(0.0)

y_train, y_valid = y["2017-07-01"], y["2017-07-02":]
X1_train, X1_valid = X_1[: "2017-07-01"], X_1["2017-07-02" :]
X2_train, X2_valid = X_2.loc[: "2017-07-01"], X_2.loc["2017-07-02":]

# Some of the algorithms above do best with certain kinds of
# preprocessing on the features (like standardization), but this is
# just a demo.
model.fit(X1_train, X2_train, y_train)
y_fit = model.predict(X1_train, X2_train).clip(0.0)
y_pred = model.predict(X1_valid, X2_valid).clip(0.0)

```

```

families = y.columns[0:6]
axs = y.loc(axis=1)[families].plot(
    subplots=True, sharex=True, figsize=(11, 9), **plot_params, alpha=0.5,
)
_ = y_fit.loc(axis=1)[families].plot(subplots=True, sharex=True, color='C0',
ax=axs)
_ = y_pred.loc(axis=1)[families].plot(subplots=True, sharex=True, color='C3',
ax=axs)
for ax, family in zip(axs, families):
    ax.legend([])
    ax.set_ylabel(family)

```

5.7 Prediction

In Lessons 2 and 3, we treated forecasting as a simple regression problem with all of our features derived from a single input, the time index. We could easily create forecasts for any time in the future by just generating our desired trend and seasonal features.

When we added lag features in Lesson 4, however, the nature of the problem changed. Lag features require that the lagged target value is known at the time being forecast. A lag 1 feature shifts the time series forward 1 step, which means you could forecast 1 step into the future but not 2 steps.

In Lesson 4, we just assumed that we could always generate lags up to the period we wanted to forecast (every prediction was for just one step forward, in other words). Real-world forecasting typically demands more than this, so in this lesson we'll learn how to make forecasts for a variety of situations.

Defining the Forecasting Task

There are two things to establish before designing a forecasting model:

what information is available at the time a forecast is made (features), and,
the time period during which you require forecasted values (target).
The forecast origin is time at which you are making a forecast. Practically, you might consider the forecast origin to be the last time for which you have training data for the time being predicted. Everything up

to the origin can be used to create features.

The forecast horizon is the time for which you are making a forecast. We often describe a forecast by the number of time steps in its horizon: a "1-step" forecast or "5-step" forecast, say. The forecast horizon describes the target.

A three-step forecast horizon with a two-step lead time, using four lag features. The figure represents what would be a single row of training data -- data for a single prediction, in other words. The time between the origin and the horizon is the lead time (or sometimes latency) of the forecast. A forecast's lead time is described by the number of steps from origin to horizon: a "1-step ahead" or "3-step ahead" forecast, say. In practice, it may be necessary for a forecast to begin multiple steps ahead of the origin because of delays in data acquisition or processing.

Preparing Data for Forecasting

In order to forecast time series with ML algorithms, we need to transform the series into a dataframe we can use with those algorithms. (Unless, of course, you are only using deterministic features like trend and seasonality.)

We saw the first half of this process in Lesson 4 when we created a feature set out of lags. The second half is preparing the target. How we do this depends on the forecasting task.

Each row in a dataframe represents a single forecast. The time index of the row is the first time in the forecast horizon, but we arrange values for the entire horizon in the same row. For multistep forecasts, this means we are requiring a model to produce multiple outputs, one for each step.

```
import numpy as np
import pandas as pd

N = 20
ts = pd.Series(
    np.arange(N),
    index=pd.period_range(start='2010', freq='A', periods=N, name='Year'),
    dtype=pd.Int8Dtype,
)

# Lag features
X = pd.DataFrame({
```

```

        'y_lag_2': ts.shift(2),
        'y_lag_3': ts.shift(3),
        'y_lag_4': ts.shift(4),
        'y_lag_5': ts.shift(5),
        'y_lag_6': ts.shift(6),
    })

    # Multistep targets
    y = pd.DataFrame({
        'y_step_3': ts.shift(-2),
        'y_step_2': ts.shift(-1),
        'y_step_1': ts,
    })

    data = pd.concat({'Targets': y, 'Features': X}, axis=1)

    data.head(10).style.set_properties(['Targets'], **{'background-color':
'LavenderBlush'}) \
        .set_properties(['Features'], **{'background-color':
'Lavender'})

```

	Targets	Features						
	y_step_3	y_step_2	y_step_1	y_lag_2	y_lag_3	y_lag_4	y_lag_5	y_lag_6
Year								
2010	2	1	0	nan	nan	nan	nan	nan
2011	3	2	1	nan	nan	nan	nan	nan
2012	4	3	2	0	nan	nan	nan	nan
2013	5	4	3	1	0	nan	nan	nan
2014	6	5	4	2	1	0	nan	nan
2015	7	6	5	3	2	1	0	nan
2016	8	7	6	4	3	2	1	0
2017	9	8	7	5	4	3	2	1
2018	10	9	8	6	5	4	3	2
2019	11	10	9	7	6	5	4	3

The above illustrates how a dataset would be prepared similar to the Defining a Forecast figure: a three-step forecasting task with a two-step lead time using five lag features. The original time series is y_step_1. The missing values we could either fill-in or drop.

Multistep Forecasting Strategies

There are a number of strategies for producing the multiple target steps required for a forecast. We'll outline four common strategies, each with strengths and weaknesses.

Multioutput model

Use a model that produces multiple outputs naturally. Linear regression and neural networks can both produce multiple outputs. This strategy

is simple and efficient, but not possible for every algorithm you might want to use. XGBoost can't do this, for instance.

Direct strategy

Train a separate model for each step in the horizon: one model forecasts 1-step ahead, another 2-steps ahead, and so on. Forecasting 1-step ahead is a different problem than 2-steps ahead (and so on), so it can help to have a different model make forecasts for each step. The downside is that training lots of models can be computationally expensive.

Recursive strategy

Train a single one-step model and use its forecasts to update the lag features for the next step. With the recursive method, we feed a model's 1-step forecast back in to that same model to use as a lag feature for the next forecasting step. We only need to train one model, but since errors will propagate from step to step, forecasts can be inaccurate for long horizons.

DirRec strategy

A combination of the direct and recursive strategies: train a model for each step and use forecasts from previous steps as new lag features. Step by step, each model gets an additional lag input. Since each model always has an up-to-date set of lag features, the DirRec strategy can capture serial dependence better than Direct, but it can also suffer from error propagation like Recursive.

Example - Flu Trends

In this example we'll apply the MultiOutput and Direct strategies to the Flu Trends data from Lesson 4, this time making true forecasts for multiple weeks beyond the training period.

We'll define our forecasting task to have an 8-week horizon with a 1-week lead time. In other words, we'll be forecasting eight weeks of flu cases starting with the following week.

The hidden cell sets up the example and defines a helper function `plot_multistep`.

```
from pathlib import Path
from warnings import simplefilter

import matplotlib.pyplot as plt
import numpy as np
```

```

import pandas as pd
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from xgboost import XGBRegressor

simplefilter("ignore")

# Set Matplotlib defaults
plt.style.use("seaborn-whitegrid")
plt.rc("figure", autolayout=True, figsize=(11, 4))
plt.rc(
    "axes",
    labelweight="bold",
    labelsiz="large",
    titleweight="bold",
    titlesiz=16,
    titlepad=10,
)
plot_params = dict(
    color="0.75",
    style=".-",
    markeredgecolor="0.25",
    markerfacecolor="0.25",
)
%config InlineBackend.figure_format = 'retina'

def plot_multistep(y, every=1, ax=None, palette_kwargs=None):
    palette_kwargs_ = dict(palette='husl', n_colors=16, desat=None)
    if palette_kwargs is not None:
        palette_kwargs_.update(palette_kwargs)
    palette = sns.color_palette(**palette_kwargs_)
    if ax is None:
        fig, ax = plt.subplots()
    ax.set_prop_cycle(plt.cycler('color', palette))
    for date, preds in y[::every].iterrows():
        preds.index = pd.period_range(start=date, periods=len(preds))
        preds.plot(ax=ax)
    return ax

data_dir = Path("../input/ts-course-data")
flu_trends = pd.read_csv(data_dir / "flu-trends.csv")
flu_trends.set_index(

```

```

        pd.PeriodIndex(flu_trends.Week, freq="W"),
        inplace=True,
    )
    flu_trends.drop("Week", axis=1, inplace=True)
    First we'll prepare our target series (weekly office visits for the
    flu) for multistep forecasting. Once this is done, training and prediction
    will be very straightfoward.

def make_lags(ts, lags, lead_time=1):
    return pd.concat(
        {
            f'y_lag_{i}': ts.shift(i)
            for i in range(lead_time, lags + lead_time)
        },
        axis=1)

# Four weeks of lag features
y = flu_trends.FluVisits.copy()
X = make_lags(y, lags=4).fillna(0.0)

def make_multistep_target(ts, steps):
    return pd.concat(
        {f'y_step_{i + 1}': ts.shift(-i)
         for i in range(steps)},
        axis=1)

# Eight-week forecast
y = make_multistep_target(y, steps=8).dropna()

# Shifting has created indexes that don't match. Only keep times for
# which we have both targets and features.
y, X = y.align(X, join='inner', axis=0)
Multioutput model
We'll use linear regression as a MultiOutput strategy. Once we have
our data prepared for multiple outputs, training and prediction is
the same as always.

# Create splits
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
shuffle=False)

model = LinearRegression()
model.fit(X_train, y_train)

```



```
y_fit = pd.DataFrame(model.predict(X_train), index=X_train.index, columns=y.columns)
y_pred = pd.DataFrame(model.predict(X_test), index=X_test.index, columns=y.columns)
Remember that a multistep model will produce a complete forecast for
each instance used as input. There are 269 weeks in the training set
and 90 weeks in the test set, and we now have an 8-step forecast for
each of these weeks.
```

```
train_rmse = mean_squared_error(y_train, y_fit, squared=False)
test_rmse = mean_squared_error(y_test, y_pred, squared=False)
print((f"Train RMSE: {train_rmse:.2f}\n" f"Test RMSE: {test_rmse:.2f}"))
```

```
palette = dict(palette='husl', n_colors=64)
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(11, 6))
ax1 = flu_trends.FluVisits[y_fit.index].plot(**plot_params, ax=ax1)
ax1 = plot_multistep(y_fit, ax=ax1, palette_kwargs=palette)
_ = ax1.legend(['FluVisits (train)', 'Forecast'])
ax2 = flu_trends.FluVisits[y_pred.index].plot(**plot_params, ax=ax2)
ax2 = plot_multistep(y_pred, ax=ax2, palette_kwargs=palette)
_ = ax2.legend(['FluVisits (test)', 'Forecast'])
Train RMSE: 389.12
Test RMSE: 582.33
```

Direct strategy

XGBoost can't produce multiple outputs for regression tasks. But by applying the Direct reduction strategy, we can still use it to produce multi-step forecasts. This is as easy as wrapping it with scikit-learn's MultiOutputRegressor.

```
from sklearn.multioutput import MultiOutputRegressor

model = MultiOutputRegressor(XGBRegressor())
model.fit(X_train, y_train)
```

```
y_fit = pd.DataFrame(model.predict(X_train), index=X_train.index, columns=y.columns)
y_pred = pd.DataFrame(model.predict(X_test), index=X_test.index, columns=y.columns)
XGBoost here is clearly overfitting on the training set. But on the
test set it seems it was able to capture some of the dynamics of the
flu season better than the linear regression model. It would likely
do even better with some hyperparameter tuning.
```

```
Train RMSE: 1.22
Test RMSE: 526.45
```

To use the DirRec strategy, you would only need to replace MultiOutputRegressor with another scikit-learn wrapper, RegressorChain. The Recursive strategy

we would need to code ourselves

6 Optifer

```
import pandas as pd
import numpy as np
import plotly.express as px

#goal = 'predict' # or 'test'
goal= 'test'

train = pd.read_csv('/kaggle/input/optiver-trading-at-the-close/train.csv')
train.head()

def add_custom_features(df):
    df['total_order_book_size'] = df['bid_size'] + df['ask_size']
    df['diff_order_book_imbalance'] = df['bid_size'] - df['ask_size']

    #bid/ask *sign(bid-ask)
    df['ratio_order_book_imbalnce'] = (df['bid_size'] /df['ask_size'])*np.sign(df['diff_order_book_imbalance'])

    df['ratio_imbalance_size_matched'] = np.where(df['matched_size']
    == 0, 2, df['imbalance_size'] / df['matched_size'])

    df['relative_bid_min_ask_size'] = (df['bid_size'] - df['ask_size'])/(df['bid_size']
    +df['ask_size'])

    df['ratio_bid_ask_size_imbalance_size'] = np.where(df['imbalance_size']
    == 0, 8, (df['bid_size'] +df['ask_size']) / df['imbalance_size'])

    df['imbalance_flag_times_size'] = df['imbalance_buy_sell_flag']*df['imbalance_size']

    df['auction_pressure1']=(df['near_price']-df['far_price'])*( df['imbalance_size'])

    df['auction_pressure2']=(df['near_price']-df['far_price'])*df['diff_order_book_imbalance']

    #df['auction_pressure2']=((df['near_price']-df['far_price'])*df['imbalance_size'])/
    (df['diff_order_book_imbalance']*df['total_order_book_size'])

    df['wap_far_mixture_price'] = (df['wap']*df['total_order_book_size']
    + df['imbalance_size']*df['far_price']) / (df['total_order_book_size']
    + df['imbalance_size'])
```

```

# Create an indicator feature for 'near_price' greater than 'reference_price'
indicates buying demand
    df['near_great_reference'] = (df['near_price'] > df['reference_price']).astype(int)

# Create an indicator feature for 'near_price' less to 'reference_price'
indicates sell demand
    df['near_less_reference'] = (df['near_price'] < df['reference_price']).astype(int)


    return df


train=add_custom_features(train)

    from sklearn.metrics import r2_score
    from tqdm import tqdm

data=train

num_stocks = data["stock_id"].nunique()
num_dates = data["date_id"].nunique()
num_updates = data["seconds_in_bucket"].nunique()

print(f"# stocks          : {num_stocks}")
print(f"# dates            : {num_dates}")
print(f"# updates per day: {num_updates}")

stock_returns = np.zeros((num_stocks, num_dates, num_updates))
index_returns = np.zeros((num_stocks, num_dates, num_updates))

for (stock_id, date_id), frame in tqdm(data.groupby(["stock_id", "date_id"])):
    frame["stock_return"] = ((frame["wap"] / frame["wap"].shift(6)).shift(-6)
- 1) * 10_000
    frame["index_return"] = frame["stock_return"] - frame["target"]

    stock_returns[stock_id, date_id] = frame["stock_return"].values
    index_returns[stock_id, date_id] = frame["index_return"].values

index_return = np.mean(index_returns, axis=0)

```

```

from sklearn.linear_model import LinearRegression
lr = LinearRegression()
y = index_return.reshape(-1)
X = stock_returns.reshape((num_stocks, -1)).T

mask = ~(np.isnan(y) | np.isnan(X).any(axis=1))
X, y = X[mask], y[mask]

lr.fit(X, y)

lr.coef_ = lr.coef_.round(3)
lr.intercept_ = 0.0

weights=lr.coef_

def add_feature_ratios_and_sums_to_index(df, weights, feature_list,
sum_feature_list):
    # Ensure the stock_id column is of type int for proper indexing
    with weights
        df['stock_id'] = df['stock_id'].astype(int)

    # Calculate the weighted average for each feature in feature_list
    index_values = {}
    for feature in feature_list:
        index_feature_name = f'index_{feature}'
        index_values[index_feature_name] = []

    # Iterate through each group and calculate the weighted average
    for (date_id, seconds), group in df.groupby(['date_id', 'seconds_in_bucket']):
        valid_rows = group.dropna(subset=[feature])
        stock_ids = valid_rows['stock_id'].astype(int)
        total_weight = weights[stock_ids].sum()
        weighted_feature = (valid_rows[feature] * weights[stock_ids]).sum()
        / total_weight if total_weight != 0 else np.nan
        index_values[index_feature_name].append(weighted_feature)

    # Add the index feature values to the DataFrame
    df[index_feature_name] = df.groupby(['date_id', 'seconds_in_bucket'])[feature].trans
_: index_values[index_feature_name].pop(0))

```

```

        # Calculate the ratio of the feature to its index value
        ratio_feature_name = f'{feature}_to_index_ratio'
        df[ratio_feature_name] = df[feature] / df[index_feature_name]

    # Calculate the sum for each feature in sum_feature_list
    for sum_feature in sum_feature_list:
        sum_feature_name = f'sum_{sum_feature}'
        df[sum_feature_name] = df.groupby(['date_id', 'seconds_in_bucket'])[sum_feature].transform('sum')

    return df

feature_list=['total_order_book_size','diff_order_book_imbalance','ratio_order_book_imbalance','ask_price']
sum_feature_list=['near_great_reference','near_less_reference']
train=add_feature_ratios_and_sums_to_index(train, weights, feature_list,
sum_feature_list)

def add_even_more_features(df):
    df['ratio_sum_great_less_ref_price'] = np.where( df['sum_near_less_reference']
== 0, np.where(df['sum_near_great_reference'] != 0, 2, 1),
    np.where(df['sum_near_less_reference'] == df['sum_near_great_reference'],
1, df['sum_near_great_reference'] / df['sum_near_less_reference']))

    return df

train=add_even_more_features(train)

def add_lagged_feature_ratios(df, features_to_lag, max_lag=6):
    """
    Adds lagged feature ratios and their differences with index ratios
    to the DataFrame.
    Also returns lists of names for these features.

    :param df: DataFrame to which the lagged feature ratios will be
    added.
    :param features_to_lag: List of feature names to create lagged
    ratios of.
    :param max_lag: Maximum number of lags to create.
    :return: DataFrame with added lagged feature ratios and their differences,
    and two lists of feature names.
    """
    new_columns = {} # Dictionary to hold new columns
    feature_to_lag_ratio_names = []

```

```

diff_feature_index_lag_ratio_names = []

for feature in features_to_lag:
    for lag in range(1, max_lag + 1):
        lagged_feature_name = f'{feature}_lag_{lag}'
        index_feature_name = f'index_{feature}'
        index_lagged_feature_name = f'index_{feature}_lag_{lag}'

        # Create lagged features
        new_columns[lagged_feature_name] = df[feature].shift(lag)
        new_columns[index_lagged_feature_name] = df[index_feature_name].shift(lag)

        # Create ratio features
        ratio_feature_name = f'{feature}_to_lag_{lag}_ratio'
        index_ratio_feature_name = f'{index_feature_name}_to_lag_{lag}_ratio'
        new_columns[ratio_feature_name] = df[feature] / new_columns[lagged_feature_name]
        new_columns[index_ratio_feature_name] = df[index_feature_name]
    / new_columns[index_lagged_feature_name]
    feature_to_lag_ratio_names.append(ratio_feature_name)

    # Calculate the difference between the feature ratio and
the index ratio
    difference_feature_name = f'diff_{feature}_index_lag_{lag}_ratio'
    new_columns[difference_feature_name] = new_columns[ratio_feature_name]
- new_columns[index_ratio_feature_name]
    diff_feature_index_lag_ratio_names.append(difference_feature_name)

    # Handle cases where seconds_in_bucket is less than lag
* 10
    condition = df['seconds_in_bucket'] < lag * 10
    new_columns[ratio_feature_name][condition] = 1
    new_columns[index_ratio_feature_name][condition] = 1
    new_columns[difference_feature_name][condition] = 0

    # Concatenate all new columns at once
    df = pd.concat([df, pd.DataFrame(new_columns)], axis=1)

    return df, feature_to_lag_ratio_names, diff_feature_index_lag_ratio_names

f2=['wap', 'reference_price']

train, feature_to_lag_ratio_names, diff_feature_index_lag_ratio_names
= add_lagged_feature_ratios(train, f2)

```

```

features_first_half = ['diff_wap_index_lag_6_ratio', 'diff_wap_index_lag_2_ratio', 'diff_wap_index_lag_1_ratio',
                        'reference_price',
                        'bid_price', 'ask_price', 'wap', 'imbalance_buy_sell_flag', 'diff_order_book_imbalance',
                        'ratio_order_book_imbalance_to_index_ratio',
                        'diff_reference_price_index_lag_1_ratio']

features_second_half = features_first_half + ['auction_pressure1',
                                              'ratio_sum_great_less_ref_price', 'wap_far_mixture_price',
                                              'near_great_reference', 'near_less_reference', 'reference_price_to_index_ratio']

def ProcessData(df):
    # Split data in two halves, with and without far/near price
    df_first_half = df[df['seconds_in_bucket'] < 300]
    df_second_half = df[df['seconds_in_bucket'] >= 300]

    # Drop rows with NaNs
    df_first_half = df_first_half[features_first_half + ['target']].dropna(axis=0)
    df_second_half = df_second_half[features_second_half + ['target']].dropna(axis=0)

    # Define training data
    y_first_half = df_first_half['target']
    X_first_half = df_first_half[features_first_half]

    y_second_half = df_second_half['target']
    X_second_half = df_second_half[features_second_half]

    return y_first_half, X_first_half, y_second_half, X_second_half

# Split in in and out of sample set
if goal == 'test':
    train_in_sample = train[train['date_id'] <= 400]
    train_out_sample = train[train['date_id'] > 400]

    y_in_sample_first_half, X_in_sample_first_half, y_in_sample_second_half,
    X_in_sample_second_half = ProcessData(train_in_sample)
    y_out_sample_first_half, X_out_sample_first_half, y_out_sample_second_half,
    X_out_sample_second_half = ProcessData(train_out_sample)
elif goal == 'predict':
    y_first_half, X_first_half, y_second_half, X_second_half = ProcessData(train)

```

```

import xgboost as xgb

# Function to split time series data into training and validation sets
def split_time_series_data(X, y, test_size=0.2):
    split_idx = int(len(X) * (1 - test_size))
    X_train = X.iloc[:split_idx]
    y_train = y.iloc[:split_idx]
    X_val = X.iloc[split_idx:]
    y_val = y.iloc[split_idx:]
    return X_train, X_val, y_train, y_val

# Predict using XGBoost
if goal == 'test':
    # Splitting the in-sample data into training and validation sets
    X_train_first_half, X_val_first_half, y_train_first_half, y_val_first_half
= split_time_series_data(X_in_sample_first_half, y_in_sample_first_half)
    X_train_second_half, X_val_second_half, y_train_second_half, y_val_second_half
= split_time_series_data(X_in_sample_second_half, y_in_sample_second_half)

    # Initialize XGBoost models with specified parameters
    xgb_model_first_half = xgb.XGBRegressor(max_depth=4, learning_rate=0.005,
n_estimators=350, gamma=1, subsample=0.5, colsample_bytree=0.5, early_stopping_rounds=6)
    xgb_model_second_half = xgb.XGBRegressor(max_depth=6, learning_rate=0.007,
n_estimators=350, gamma=1, subsample=0.5, colsample_bytree=0.5, early_stopping_rounds=7)

    # Fit the models with early stopping
    xgb_model_first_half.fit(X_train_first_half, y_train_first_half,
eval_set=[(X_val_first_half, y_val_first_half)], verbose=False)
    xgb_model_second_half.fit(X_train_second_half, y_train_second_half,
eval_set=[(X_val_second_half, y_val_second_half)], verbose=False)

    # Predict using the fitted models
    y_pred_first_half = xgb_model_first_half.predict(X_out_sample_first_half)
    y_pred_second_half = xgb_model_second_half.predict(X_out_sample_second_half)

elif goal == 'predict':

    X_train_first_half, X_val_first_half, y_train_first_half, y_val_first_half
= split_time_series_data(X_first_half, y_first_half)
    X_train_second_half, X_val_second_half, y_train_second_half, y_val_second_half
= split_time_series_data(X_second_half, y_second_half)
    # Use the entire dataset for prediction

    # Initialize XGBoost models with specified parameters

```



```

xgb_model_first_half = xgb.XGBRegressor(max_depth=2, learning_rate=0.003,
n_estimators=200, gamma=2.5, subsample=0.5, colsample_bytree=0.5,early_stopping_rounds=6)
xgb_model_second_half = xgb.XGBRegressor(max_depth=3, learning_rate=0.005,
n_estimators=200, gamma=2, subsample=0.5, colsample_bytree=0.4,early_stopping_rounds=7)

# Fit the models with early stopping
xgb_model_first_half.fit(X_train_first_half, y_train_first_half,
eval_set=[(X_val_first_half, y_val_first_half)], verbose=False)
xgb_model_second_half.fit(X_train_second_half, y_train_second_half,
eval_set=[(X_val_second_half, y_val_second_half)], verbose=False)

import matplotlib.pyplot as plt
# Assuming xgb_model_second_half is your trained XGBoost model
importances = xgb_model_second_half.feature_importances_

# Normalize the importances
importances_normalized = importances / importances.sum()

# Plot
xgb.plot_importance(xgb_model_second_half, importance_type='gain',
max_num_features=10)
plt.show()

xgb.plot_importance( xgb_model_first_half, importance_type='gain',
max_num_features=25)
plt.show()

import statsmodels.api as sm

if goal == 'test':

    # Concatenate predictions
    # Assuming y_in_sample_first_half and X_in_sample_first_half are
defined
# Add a constant to the independent variables matrix for the intercept
X_in_sample_first_half_with_const = sm.add_constant(X_in_sample_first_half)

# Fit the model
model_first_half = sm.OLS(y_in_sample_first_half, X_in_sample_first_half_with_const).fit()

```

```

# Print the summary
    print("First Half Model Summary:")
    print(model_first_half.summary())

# Identify and print statistically significant features for the first
half
    significant_features_first_half = model_first_half.pvalues[model_first_half.pvalues
< 0.02].index.tolist()
    print("\nStatistically Significant Features for First Half (p <
0.02):")
    print(significant_features_first_half)

# Repeat for the second half
# Assuming y_in_sample_second_half and X_in_sample_second_half are
defined
    X_in_sample_second_half_with_const = sm.add_constant(X_in_sample_second_half)

# Fit the model
    model_second_half = sm.OLS(y_in_sample_second_half, X_in_sample_second_half_with_const)

# Print the summary
    print("\nSecond Half Model Summary:")
    print(model_second_half.summary())

# Identify and print statistically significant features for the second
half
    significant_features_second_half = model_second_half.pvalues[model_second_half.pvalues
< 0.02].index.tolist()
    print("\nStatistically Significant Features for Second Half (p
< 0.02):")
    print(significant_features_second_half)

if goal == 'test':
    # Concatenate predictions
    X_out_sample_first_half['prediction'] = y_pred_first_half
    X_out_sample_second_half['prediction'] = y_pred_second_half
    predictions = pd.concat([X_out_sample_first_half['prediction'],
X_out_sample_second_half['prediction']])

    y_out_sample = pd.concat([y_out_sample_first_half, y_out_sample_second_half])
    MAE = np.mean(abs(predictions - y_out_sample))

```

```

# Compare with simple prediction
#simple_prediction = train['imbalance_buy_sell_flag'].map(simple_mapping)
simple_prediction = 0
MAE_simple = np.mean(abs(simple_prediction - train['target']))

print('MAE Linear Regression:', MAE)
print('MAE simple: ', MAE_simple)

from sklearn.model_selection import GridSearchCV, TimeSeriesSplit

# Define the parameter grid
param_grid = {
    'max_depth': [3, 4, 5], # Example values
    'gamma': [0.5, 0.7, 1, 2], # Example values
    'learning_rate': [0.005, 0.02, 0.1], # Example values
    'n_estimators': [200, 300, 400] # Example values
}

# Initialize the XGBoost regressor
xgb_model = xgb.XGBRegressor(subsample=0.5, colsample_bytree=0.5, early_stopping_rounds=4)

# Use TimeSeriesSplit for cross-validation
tscv = TimeSeriesSplit(n_splits=5)

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid,
cv=tscv, scoring='neg_mean_squared_error', verbose=False)

# Assuming X and y are your full dataset
grid_search.fit(X_train_second_half, y_train_second_half)

# Print the best parameters and the corresponding score
print("Best parameters found: ", grid_search.best_params_)
print("Best generalization error: ", -grid_search.best_score_)

# Submit predictions
if goal == 'predict':
    import optiver2023
    env = optiver2023.make_env()
    iter_test = env.iter_test()

    counter = 0

    for (test, revealed_targets, sample_prediction) in iter_test:
        test['prediction'] = 0

```

```

        #test['imbalance'] = test['imbalance_size'] * test['imbalance_buy_sell_flag']
        test=add_custom_features(test)
        test=add_feature_ratios_and_sums_to_index(test, weights, feature_list,
sum_feature_list)

        test=add_even_more_features(test)
        test, f_to_lag_ratio_names, diff_f_index_lag_ratio_names =
add_lagged_feature_ratios(test,f2)


first_half1 = test[test['seconds_in_bucket'] < 300]
second_half2 = test[test['seconds_in_bucket'] >= 300]

X_first_half1 = first_half1[features_first_half]
X_second_half2 = second_half2[features_second_half]


non_nan_rows_first_half1 = ~X_first_half1.isna().any(axis=1)
non_nan_rows_second_half2 = ~X_second_half2.isna().any(axis=1)

# Check if both halves have sufficient data for prediction
if len(X_first_half1[non_nan_rows_first_half1]) > 1 and len(X_second_half2[non_nan_r
> 1:
        # Make predictions for each half and update the 'prediction'
column
        test.loc[non_nan_rows_first_half1, 'prediction'] = xgb_model_first_half.predict
        test.loc[non_nan_rows_second_half2, 'prediction'] = xgb_model_second_half.predi
elif len(X_second_half2[non_nan_rows_second_half2]) > 1:
        # Make predictions for the second half if it has sufficient
data2
        test.loc[non_nan_rows_second_half2, 'prediction'] = xgb_model_second_half.predi
elif len(X_first_half1[non_nan_rows_first_half1]) > 1:
        # Make predictions for the first half if it has sufficient
data
        test.loc[non_nan_rows_first_half1, 'prediction'] = xgb_model_first_half.predict

sample_prediction['target'] = test['prediction']
env.predict(sample_prediction)
counter += 1

```

6.1 Andere dingen

```
# Sample data
```

```

data = {
    'time': [1, 1, 2, 2, 3, 3],
    'stock': ['A', 'B', 'A', 'B', 'A', 'B'],
    'imbalance': [10, 4, 5, -1, -2, 3]
}
df = pd.DataFrame(data)

# Calculate cumulative imbalance across all stocks at each time
df['cumulative_imbalance'] = df.groupby('time')['imbalance'].transform('sum').cumsum()

print(df)

```

```

ALs maar 1 stock
df['cumulative_imbalance'] = df['imbalance'].cumsum()

```

7 Stoch Vol estimation

To find the location of the stockdata2.csv file on your computer:

Locate the file: Find where stockdata2.csv is saved on your computer, such as in the Downloads or Documents folder.

Get the file path:

On Windows, right-click the file, select Properties, and copy the Location field, then append \stockdata2.csv to it.

On macOS, right-click the file, select Get Info, and copy the Where path, then append /stockdata2.csv to it.

Update your code with the full path: Replace ./stockdata2.csv with the full file path. For example:

```

python
Copy code
df = pd.read_csv('C:/Users/YourUsername/Downloads/stockdata2.csv')
# Windows
df = pd.read_csv('/Users/YourUsername/Downloads/stockdata2.csv') #
macOS

```

This will allow pandas to load the file directly from that location.

You said:

```

I ge this error Cell In[3], line 1
df=pd.read_csv('C:\Users\mapprt\stockdata2.csv')

```

^
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes
in position 2-3: truncated escape

ChatGPT said:

ChatGPT

The error occurs because backslashes van linksboven naar rechtsonder
slash%(\\)

in the file path are interpreted as escape characters in Python. To
fix this, you have two options:

Use raw strings by adding an r before the string:

python

Copy code

```
df = pd.read_csv(r'C:\Users\penge\stockdata2.csv')
```

Replace backslashes with forward slashes:

python

Copy code

```
df = pd.read_csv('C:/Users/penge/stockdata2.csv')
```

Both methods should work without triggering the Unicode escape error.

```
percentage_greater_than_3 = (df['Returns'] > 3).mean() * 100  
print(f"Percentage of Returns Greater Than 3: {percentage_greater_than_3}%")
```

8 Titanic

```
train_data = pd.read_csv("/kaggle/input/titanic/train.csv")  
train_data.head()
```

```
women = train_data.loc[train_data.Sex == 'female']["Survived"]  
rate_women = sum(women)/len(women)
```

```
print("% of women who survived:", rate_women)
```

```
surv_rate = train_data.groupby("Sex")["Survived"].mean()
```

```

surv_rate

df.info()
df.isna().sum()

coln = [col for col in df.columns if df[col].notna().all()]

from sklearn.impute import SimpleImputer

df['age' + '_was_missing'] = df['Age'].isnull()
df_plus = df.copy()

my_imputer = SimpleImputer(strategy='mean')
df_plus = pd.DataFrame(my_imputer.fit_transform(df_plus))

df_plus.columns=df.columns

from sklearn.experimental import enable_iterative_imputer # Required
to enable IterativeImputer
from sklearn.impute import IterativeImputer

# Initialize the iterative imputer
iter_imputer = IterativeImputer()

# Perform regression-based imputation
df_plus2 = pd.DataFrame(iter_imputer.fit_transform(train_data), columns=train_data.columns)

# Label encoding for categoricals
for colname in df.select_dtypes("object"):
    df[colname], _ = df[colname].factorize()
# All discrete features should now have integer dtypes (double-check

discrete_features = df.dtypes == int

```

```

X.info()
df['feat'] = df['feat'].astype('int64')

from sklearn.feature_selection import mutual_info_regression

def make_mi_scores(X, y):
    mi_scores = mutual_info_regression(X, y)
    mi_scores = pd.Series(mi_scores, name="MI Scores", index=X.columns)
    mi_scores = mi_scores.sort_values(ascending=False)
    return mi_scores

mi_scores = make_mi_scores(df3, y)

def plot_mi_scores(scores):
    scores = scores.sort_values(ascending=True)
    width = np.arange(len(scores))
    ticks = list(scores.index)
    plt.barh(width, scores)
    plt.yticks(width, ticks)
    plt.title("Mutual Information Scores")
    plt.figure(dpi=100, figsize=(8, 5))

    return

plot_mi_scores(mi_scores)

```

9 Out of Box

To predict temperature, you can generate several features that capture various aspects of the weather patterns that influence temperature. Below are common features you can create, along with Python code examples.

Features to Create: Time-based features:

Hour of the day: Temperature might vary by hour. Day of the week: Temperature patterns often vary between weekdays and weekends. Month: Seasonal variations can be captured by the month. Day of the year: This can capture the time of the year without breaking it into months. Weather-related features:

Humidity: High humidity often correlates with higher temperatures. Wind speed: Wind can affect perceived temperature and actual temperature. Pressure: Atmospheric pressure might indicate weather changes affecting temperature. Lagged features:

Previous temperature values: The temperature from a few hours or days ago. Rolling averages: Rolling mean or median of past temperatures, humidity, wind speed, etc. Interaction features:

Humidity * Wind Speed: The interaction of wind speed and humidity could influence temperature in unique ways. Cyclic features for time-based variables:

Sine and cosine transforms of hour, day, month to capture periodicity (e.g., time of day or seasonality).

```
import pandas as pd
import numpy as np

# Example dataframes (assuming they have a 'datetime' column and 'temperature'
# column)
# df1: weather data
# df2: additional features like humidity, wind_speed, and pressure
df1 = pd.DataFrame({
    'datetime': pd.date_range(start='2024-01-01', periods=100, freq='H'),
    'temperature': np.random.randn(100) * 10 + 20 # Simulated temperature
    data
})

df2 = pd.DataFrame({
    'datetime': pd.date_range(start='2024-01-01', periods=100, freq='H'),
    'humidity': np.random.rand(100) * 100, # Humidity in percentage
    'wind_speed': np.random.rand(100) * 10, # Wind speed in m/s
    'pressure': np.random.rand(100) * 1000 + 1000 # Pressure in hPa
})

# Merge dataframes on datetime
df = pd.merge(df1, df2, on='datetime')

# Feature creation
df['hour'] = df['datetime'].dt.hour # Hour of the day
df['day_of_week'] = df['datetime'].dt.dayofweek # Day of the week
df['month'] = df['datetime'].dt.month # Month of the year
df['day_of_year'] = df['datetime'].dt.dayofyear # Day of the year

# Cyclical features
df['hour_sin'] = np.sin(2 * np.pi * df['hour'] / 24) # Sine of the
hour
df['hour_cos'] = np.cos(2 * np.pi * df['hour'] / 24) # Cosine of the
hour
```

```

# Lag features (previous temperature, 1 hour lag)
df['temp_lag_1'] = df['temperature'].shift(1)

# Rolling features (rolling mean of temperature over 3 hours)
df['temp_rolling_mean_3'] = df['temperature'].rolling(window=3).mean()

# Interaction feature (humidity * wind speed)
df['humidity_wind_interaction'] = df['humidity'] * df['wind_speed']

# Display the final dataframe
import ace_tools as tools; tools.display_dataframe_to_user(name="Weather
Prediction Features", dataframe=df)

print(df.head())

# Long-term lag features (e.g., previous day's temperature and previous
week's average temperature)
df['temp_lag_24'] = df['temperature'].shift(24) # Temperature 24 hours
(1 day) ago
df['temp_lag_168'] = df['temperature'].shift(168) # Temperature 168
hours (1 week) ago

# Rolling statistics for temperature, humidity, and wind speed (7-day
and 30-day rolling means)
df['temp_rolling_mean_7d'] = df['temperature'].rolling(window=24*7).mean()
# 7-day rolling mean
df['humidity_rolling_mean_7d'] = df['humidity'].rolling(window=24*7).mean()
# 7-day rolling mean for humidity
df['wind_rolling_mean_7d'] = df['wind_speed'].rolling(window=24*7).mean()
# 7-day rolling mean for wind speed

# Temperature trend (indicator if the temperature has increased or
decreased over the last day)
df['temp_trend_1d'] = df['temperature'].diff(24).apply(lambda x: 1
if x > 0 else 0)

# Seasonal indicators (based on the month)
df['is_winter'] = df['month'].apply(lambda x: 1 if x in [12, 1, 2]
else 0)
df['is_spring'] = df['month'].apply(lambda x: 1 if x in [3, 4, 5] else
0)
df['is_summer'] = df['month'].apply(lambda x: 1 if x in [6, 7, 8] else
0)
df['is_autumn'] = df['month'].apply(lambda x: 1 if x in [9, 10, 11]

```

```

else 0)

# Additional interaction features
df['humidity_pressure_interaction'] = df['humidity'] * df['pressure']
df['wind_pressure_interaction'] = df['wind_speed'] * df['pressure']

# Cumulative sum features (e.g., cumulative temperature since the start
of the data)
df['cumulative_temp'] = df['temperature'].cumsum() # Cumulative sum
of temperature
df['cumulative_humidity'] = df['humidity'].cumsum() # Cumulative sum
of humidity

# Temperature anomalies (difference between temperature and rolling
mean)
df['temp_anomaly'] = df['temperature'] - df['temp_rolling_mean_7d']

# Display the enhanced dataframe with additional features
import ace_tools as tools; tools.display_dataframe_to_user(name="Enhanced
Weather Prediction Features", dataframe=df)

print(df.head())

Humidity and Wind Speed: High humidity and high wind speeds might have
unique effects on temperature perception and actual temperature variations.

df['humidity_wind_interaction'] = df['humidity'] * df['wind_speed']
Humidity and Pressure: Humidity and atmospheric pressure together may
indicate approaching weather changes, which can influence temperature.

df['humidity_pressure_interaction'] = df['humidity'] * df['pressure']
Wind Speed and Pressure: Wind speed and pressure interactions can capture
certain weather patterns like high-pressure systems.

df['wind_pressure_interaction'] = df['wind_speed'] * df['pressure']
Temperature and Humidity: The combination of current temperature and
humidity is often used in meteorology for indices like the Heat Index.

df['temp_humidity_interaction'] = df['temperature'] * df['humidity']
2. Time-based Interactions
Hour of Day and Weather Variables: Interactions between hour and temperature
or humidity may capture diurnal patterns.

df['hour_humidity_interaction'] = df['hour'] * df['humidity']
df['hour_temp_interaction'] = df['hour'] * df['temperature']
Seasonal Indicator and Weather Variables: Seasonal indicators (e.g.,

```

winter, summer) combined with humidity, wind speed, or pressure can capture seasonal variations.

```
df['is_winter_pressure_interaction'] = df['is_winter'] * df['pressure']  
df['is_summer_humidity_interaction'] = df['is_summer'] * df['humidity']
```

3. Temperature Trends with Weather Variables

Temperature Lag Interactions: Combine lagged temperatures with humidity or wind speed to account for temporal dependencies and recent trends in temperature.

```
df['temp_lag_1_humidity_interaction'] = df['temp_lag_1'] * df['humidity']  
df['temp_lag_24_wind_interaction'] = df['temp_lag_24'] * df['wind_speed']
```

Rolling Mean Interactions: Interaction between rolling mean of temperature and current weather variables to capture recent trends in temperature.

```
df['temp_rolling_mean_3_wind_interaction'] = df['temp_rolling_mean_3']  
* df['wind_speed']  
df['temp_rolling_mean_7d_pressure_interaction'] = df['temp_rolling_mean_7d']  
* df['pressure']
```

4. Non-linear Interactions

Squared and Cubic Terms: Squared terms of temperature and humidity may help capture non-linear effects in combination with other variables.

```
df['temp_sq_humidity_interaction'] = (df['temperature'] ** 2) * df['humidity']  
df['humidity_sq_pressure_interaction'] = (df['humidity'] ** 2) * df['pressure']
```

Logarithmic Interactions: Log transformations of weather variables can sometimes capture multiplicative effects more effectively.

```
df['log_temp_wind_interaction'] = np.log(df['temperature'] + 1) * df['wind_speed']  
df['log_humidity_pressure_interaction'] = np.log(df['humidity'] + 1)  
* df['pressure']
```

In a weather-related dataset aimed at predicting temperature, here are some commonly expected features, with quick explanations of each:

datetime: Timestamp of each record, used to derive time-based features (e.g., hour, day of the week).

temperature: The target variable, representing the recorded temperature at the weather station.

humidity: The amount of water vapor in the air, measured as a percentage. Higher humidity can make it feel warmer.

wind_speed: Speed of the wind at the station, usually in meters per second (m/s). Wind can impact temperature perception and cooling.

pressure: Atmospheric pressure in hPa (hectopascals). Changes in pressure often indicate weather changes, affecting temperature.

precipitation: Amount of rain or snowfall, typically in millimeters (mm). Precipitation often correlates with temperature drops.

solar_radiation: Amount of solar energy received, usually in watts per square meter (W/m²). Higher solar radiation increases temperature.

cloud_cover: A measure of cloudiness, often as a percentage. Higher cloud cover reduces solar radiation and may cool the temperature.

dew_point: The temperature at which air becomes fully saturated with moisture. It gives insight into humidity and likely temperature drops.

season: Derived feature indicating the season (e.g., winter, summer), helpful in capturing seasonal temperature patterns.

These features offer a comprehensive view of environmental conditions impacting temperature, enabling robust predictive modeling.

9.1 Part two

applicant_income: The income of the mortgage applicant, a primary factor in determining loan eligibility and affordability.

coapplicant_income: Income of a co-applicant, if any, which can increase the borrowing capacity.

loan_amount: The requested mortgage loan amount, typically in thousands of dollars or the local currency unit.

loan_term: Duration of the mortgage in years or months. Longer terms usually mean lower monthly payments but more interest over time.

interest_rate: The interest rate applied to the loan, directly affecting monthly payments and total interest paid.

credit_score: Applicant's credit score, indicating their creditworthiness. Higher scores generally lead to better loan terms.

property_value: Market value of the property being mortgaged, used to determine the loan-to-value ratio.

down_payment: Amount of upfront payment made by the applicant, often as a percentage of the property value.

employment_status: Employment type or stability of the applicant (e.g., employed, self-employed), which can affect income reliability.

age: Age of the applicant, often used in assessing risk, as older applicants might have shorter working lives.

marital_status: Marital status of the applicant, potentially impacting financial stability.

number_of_dependents: Number of dependents the applicant supports, which can influence disposable income.

debt_to_income_ratio: Ratio of applicant's monthly debt payments to their monthly income, indicating their debt burden.

loan_purpose: Purpose of the loan (e.g., purchase, refinance), as different purposes carry varying risk profiles.

Created Features

loan_to_value_ratio (LTV): Ratio of loan amount to property value. Higher LTVs are often riskier.

```
df['loan_to_value_ratio'] = df['loan_amount'] / df['property_value']
```

monthly_payment: The monthly mortgage payment based on loan amount, interest rate, and loan term.

```
import numpy as np
r = df['interest_rate'] / 1200 # Monthly interest rate
n = df['loan_term'] * 12 # Total number of monthly payments
df['monthly_payment'] = df['loan_amount'] * r / (1 - (1 + r) ** -n)
```

total_income: Sum of applicant and co-applicant income for a clearer view of combined income.

```
df['total_income'] = df['applicant_income'] + df['coapplicant_income']
```

income_per_dependent: Applicant's income divided by the number of dependents, indicating income burden.

```
df['income_per_dependent'] = df['applicant_income'] / (df['number_of_dependents'] + 1)
```

credit_utilisation_ratio: Ratio of total debt payments (including mortgage) to income, representing financial health.

```
df['credit_utilisation_ratio'] = df['debt_to_income_ratio'] + (df['monthly_payment'] / df['total_income'])
```

years_to_retirement: Derived from age, assuming retirement at 65, which can affect loan approval and terms.

```
df['years_to_retirement'] = 65 - df['age']
```

applicant_experience: Years in employment derived from employment start year, if available, as longer experience may indicate stability.

```
df['applicant_experience'] = df['current_year'] - df['employment_start_year']
```

interest_rate_category: Categorical feature for interest rate brackets (e.g., low, medium, high) for easier model interpretation.

```
def rate_category(rate):  
    if rate < 3: return 'low'  
    elif rate < 5: return 'medium'  
    else: return 'high'
```

```
df['interest_rate_category'] = df['interest_rate'].apply(rate_category)
```

property_age: Age of the property, calculated based on current year and property construction year, as older properties may have more risks.

```
df['property_age'] = df['current_year'] - df['property_construction_year']
```

employment_income_ratio: Ratio of applicant income to co-applicant income, showing income balance.

```
df['employment_income_ratio'] = df['applicant_income'] / (df['coapplicant_income'] + 1)
```

These created features aim to better capture financial risk, income stability, property value, and applicant creditworthiness, leading to a more robust mortgage prediction model.

9.2 Part 3

Expected Features

company_name: The name or ID of the company, useful for grouping and analysis.

loan_amount: The amount of the loan extended to the company, typically in the local currency.

loan_term: Duration of the loan, usually in months or years, which

affects the repayment schedule and default risk.

interest_rate: Interest rate applied to the loan, directly affecting monthly payments and total interest owed.

loan_purpose: Purpose of the loan (e.g., working capital, expansion, asset purchase), which can indicate risk associated with the loan type.

collateral_value: Value of any collateral provided to secure the loan, used to mitigate lender risk.

company_revenue: Annual revenue of the company, a core indicator of financial health and its ability to service debt.

company_profit: Net profit or operating profit of the company, showing profitability and ability to repay the loan.

debt_to_equity_ratio: Ratio of total debt to shareholder equity, measuring financial leverage. Higher values can indicate higher risk.

current_ratio: Ratio of current assets to current liabilities, showing liquidity. A ratio above 1 indicates the company can cover short-term liabilities with short-term assets.

quick_ratio: Similar to the current ratio but excludes inventory, providing a stricter measure of liquidity.

cash_flow: Cash flow from operating activities, giving insight into cash generated from core business activities, essential for repaying loans.

credit_score: Company credit score, if available. This metric is often used by lenders to gauge creditworthiness.

industry: The industry in which the company operates, as default risk varies significantly across industries.

years_in_business: The number of years the company has been operating, with longer histories often indicating stability.

loan_status: Indicates if the loan is active, paid off, or defaulted. In this case, the target variable would be "defaulted" or similar.

previous_defaults: Number of previous loan defaults by the company, a critical indicator of default risk.

number_of_employees: Size of the company in terms of employees, with larger companies often having more stable cash flows.

location: Geographic location of the company, as economic conditions vary by region and can affect default risk.

loan_installment: Monthly or quarterly loan payment amount, directly influencing the financial burden on the company.

growth_rate: Year-over-year growth in revenue or profit, which can indicate a stable or improving financial condition.

1. Loan-Specific Ratios

Loan-to-Value Ratio (LTV): Ratio of loan amount to collateral value, indicating the level of security for the loan.

```
df['loan_to_value_ratio'] = df['loan_amount'] / df['collateral_value']
```

Debt Service Coverage Ratio (DSCR): Ratio of company cash flow to loan installment, showing if cash flow covers debt obligations.

```
df['debt_service_coverage_ratio'] = df['cash_flow'] / df['loan_installment']
```

2. Leverage and Liquidity Ratios

Leverage Ratio: Combining debt and equity to understand overall financial risk exposure.

```
df['leverage_ratio'] = df['debt_to_equity_ratio'] + (df['loan_amount'] / df['company_revenue'])
```

Liquidity Ratio with Loan Impact: Adjusting the current ratio by including loan payments as liabilities, for a clearer picture of liquidity.

```
df['adjusted_current_ratio'] = df['current_ratio'] - (df['loan_installment'] / df['company_revenue'])
```

3. Profitability and Growth Indicators

Profit Margin: Ratio of profit to revenue, highlighting profitability and efficiency.

```
df['profit_margin'] = df['company_profit'] / df['company_revenue']
```

Revenue Growth Rate: Year-over-year revenue growth, which can indicate business health and risk of default.

```
df['revenue_growth_rate'] = df['company_revenue'].pct_change( periods=1)
```

Profit Growth Rate: Similar to revenue growth but focused on profit, showing improvement in profitability.

```
df['profit_growth_rate'] = df['company_profit'].pct_change( periods=1)
```

4. Company and Industry Risk Indicators

Industry-Average Comparison: Comparing debt-to-equity, revenue, and profit to industry averages. This can be done by grouping by industry.

```
df['industry_avg_debt_to_equity'] =
```

```
df.groupby('industry')['debt_to_equity_ratio'].transform('mean')
```

```
df['debt_to_equity_above_avg'] =
```

```
df['debt_to_equity_ratio'] > df['industry_avg_debt_to_equity']
```

Years in Business Category: Categorising the number of years in business into bins (e.g., "Startup," "Established," "Mature") to indicate stability.

```
df['business_age_category'] = pd.cut(df['years_in_business'], bins=[0, 5, 15, 30], labels=['Startup', 'Established', 'Mature'])
```

5. Historical Loan Behaviour

Default Rate: Company's historical default rate on loans (if there are multiple loans in the data).

```
df['default_rate'] = df['previous_defaults'] / (df['previous_defaults'] + 1)
```

Total Debt Exposure: Sum of current and previous loans to see if the company has taken on increasing debt over time.

```
df['total_debt_exposure'] = df['loan_amount'] +
```

```
df.groupby('company_name')['loan_amount'].transform('sum')
```

```
df['total_debt_exposure'] = df['loan_amount'] + df.groupby('company_name')['loan_amount'].transform('sum')
```

6. Temporal Features

Time Since Last Loan: For companies with multiple loans, the time since the last loan can indicate financial stability and risk-taking behaviour.

```
df['time_since_last_loan'] = df['loan_date'].diff().dt.days
```

Loan Purpose Categories: Encoding loan purpose as categories (e.g., "Working Capital," "Expansion") to identify default risks associated with certain loan purposes.

```
df['loan_purpose_encoded'] = df['loan_purpose'].map({'Working Capital': 0, 'Expansion': 1, 'Asset Purchase': 2})
```

7. Macro-Economic Indicators (if available)

Economic Conditions Adjustment: Adjust revenue, profit, or cash flow metrics by GDP growth rate, inflation, or regional economic conditions, if such data is available.

```
df['adjusted_revenue'] = df['company_revenue'] / df['gdp_growth_rate']
```

These created features offer insight into financial leverage, liquidity, profitability trends, industry comparison, and risk indicators that better capture the potential for corporate loan defaults, especially for smaller, non-public companies.

9.3 P4

Tutorial: Time Series Analysis and Forecasting

Scaling is crucial when computing Principal Component Analysis (PCA) because PCA identifies directions (principal components) that capture the maximum variance in the data. If features are not on a similar scale, those with larger ranges will dominate the variance, leading to biased principal components that reflect mainly the high-variance features rather than the true underlying structure.

Here's why scaling is beneficial for PCA:

Equal Contribution of Features: Standardizing features (giving them mean 0 and standard deviation 1) ensures that each feature contributes equally to the analysis, allowing PCA to capture variance from each feature fairly.

Avoiding Dominance by Large-Scale Features: Without scaling, features with large numerical ranges (e.g., age in years vs. income in thousands) would disproportionately influence the principal components, potentially obscuring insights from smaller-scale features.

Improved Component Interpretation: When all features are on a comparable scale, the principal components represent meaningful directions in the feature space, enabling more accurate dimensionality reduction.

Enhanced Performance in Distance-Based Algorithms: PCA relies on linear transformations that are sensitive to scale. Standardization aligns feature distributions, allowing PCA to perform optimally.

In summary, scaling ensures that PCA captures the most relevant patterns

in the data by emphasizing relationships between features rather than their raw numerical magnitudes.

You said:

why is it also good to standardise features for neural networks?

ChatGPT said:

Standardising features is important for neural networks because it can significantly improve training efficiency, stability, and performance. Here's why:

Improved Convergence: Neural networks use gradient-based optimisers (like stochastic gradient descent) that rely on efficient and stable updates to weights. Standardising features (mean 0, standard deviation 1) ensures that inputs are centred and scaled, preventing features with larger ranges from leading to unbalanced gradient updates. This often speeds up convergence, as the optimiser doesn't need to compensate for scale differences.

Balanced Weight Updates: Without standardisation, some features may dominate, leading to large or unstable gradients. This can cause weight updates to be skewed, potentially resulting in "vanishing" or "exploding" gradients, especially in deeper networks. Standardisation reduces the risk of these issues by balancing feature scales, ensuring smoother learning.

Easier Weight Initialisation: Standardising inputs makes it easier for the network to initialise weights effectively. Most initialisation schemes (e.g., Xavier or He initialization) assume inputs are standardised, ensuring activations are well-scaled through the layers and improving the model's ability to learn complex patterns.

Prevents Saturation of Activation Functions: For activation functions like sigmoid or tanh, large input values can cause saturation (where outputs are near 0 or 1), leading to very small gradients. Standardised inputs, centered around zero, keep activations in their sensitive range (near the origin), helping gradients remain effective during backpropagation.

Improved Generalisation: Standardisation helps by encouraging the network to learn relevant patterns across all features without being skewed toward those with larger scales, which can improve generalisation on unseen data.